第14章 C++工具

在C++发展的后期，有时C++编译系统根据实际工作的需要，增加了一些功能，作为工具来使用，其中主要有模板(包括函数模板和类模板)、异常处理、命名空间和运行时类型识别，以帮助程序设计人员更方便地进行程序的设计和调试工作。1997年ANSI C++委员会将它们纳入了ANSI C++标准，建议所有的C++编译系统都能实现这些功能。这些工具是非常有用的，C++的使用者应当尽量使用这些工具。

14.1 异常处理

14.1.1 异常处理的任务

程序编制者不仅要考虑程序没有错误的理想情况，更要考虑程序存在错误时的情况，应该能够尽快地发现错误，消除错误。

程序中常见的错误有两大类：语法错误和运行错误。在编译时，编译系统能发现程序中的语法错误。

有的程序虽然能通过编译，也能投入运行。但是在运行过程中会出现异常，得不到正确的运行结果，甚至导致程序不正常终止，或出现死机现象。这类错误比较隐蔽，不易被发现，往往耗费许多时间和精力。这成为程序调试中的一个难点。

在设计程序时，应当事先分析程序运行时可能出现的各种意外的情况，并且分别制订出相应的处理方法，这就是程序的异常处理的任务。

在运行没有异常处理的程序时，如果运行情况出现异常，由于程序本身不能处理，程序只能终止运行。如果在程序中设置了异常处理机制，则在运行情况出现异常时，由于程序本身已规定了处理的方法，于是程序的流程就转到异常处理代码段处理。用户可以指定进行任何的处理。

需要说明，只要出现与人们期望的情况不同，都可以认为是异常，并对它进行异常处理。因此，所谓异常处理指的是对运行时出现的差错以及其他例外情况的处理。

14.1.2 异常处理的方法

在一个小的程序中，可以用比较简单的方法处理异常。但是在一个大的系统中，如果在每一个函数中都设置处理异常的程序段，会使程序过于复杂和庞大。因此，C++采取的办法是：如果在执行一个函数过程中出现异常，可以不在本函数中立即处理，而是发出一个信息，传给它的上一级(即调用它的函数)，它的上级捕捉到这个信息后进行处理。如果上一级的函数也不能处理，就再传给其上一级，由其上一级处理。如此逐级上送，如果到最高一级还无法处理，最后只好异常终止程序的执行。

这样做使异常的发现与处理不由同一函数来完成。好处是使底层的函数专门用于解决实际任务，而不必再承担处理异常的任务，以减轻底层函数的负担，而把处理异常的任务上移到某一层去处理。这样可以提高效率。

C++处理异常的机制是由**3**个部分组成的，即检查**(try)**、抛出**(throw)**和捕捉**(catch)**。把需要检查的语句放在**try**块中，**throw**用来当出现异常时发出一个异常信息，而**catch**则用来捕捉异常信息，如果捕捉到了异常信息，就处理它。

例**14.1** 给出三角形的三边**a,b,c**，求三角形的面积。只有**a+b>c,b+c>a,c+a>b**时才能构成三角形。设置异常处理，对不符合三角形条件的输出警告信息，不予计算。

先写出没有异常处理时的程序**:**

```
#include <iostream>
#include <cmath>
using namespace std;
int main( )
{double triangle(double,double,double);
 double a,b,c;
 cin>>a>>b>>c;
 while(a>0 && b>0 && c>0)
  {cout<<triangle(a,b,c)<<endl;
   cin>>a>>b>>c;
  }
  return 0;
```

```
}

double triangle(double a,double b,double c)
{double area;
 double s=(a+b+c)/2;
 area=sqrt(s*(s-a)*(s-b)*(s-c));
 return area;
}
```

运行情况如下：

```
6 5 4↙(输入a,b,c的值)
9.92157          (输出三角形的面积)
1 1.5 2↙         (输入a,b,c的值)
0.726184         (输出三角形的面积)
1 2 1↙           (输入a,b,c的值)
0                (输出三角形的面积，此结果显然不对,因为不是三角形)
1 0 6↙           (输入a,b,c的值)
(结束)
```

修改程序，在函数**traingle**中对三角形条件进行检查，如果不符合三角形条件，就抛出一个异常信息，在主函数中的**try-catch**块中调用**traingle**函数，检测有无异常信息，并作相应处理。修改后的程序如下：

```
#include <iostream>
#include <cmath>
using namespace std;
void main( )
{double triangle(double,double,double);
 double a,b,c;
 cin>>a>>b>>c;
 try//在try块中包含要检查的函数
  {while(a>0 && b>0 && c>0)
    {cout<<triangle(a,b,c)<<endl;
   cin>>a>>b>>c;
    }
  }
```

```
 catch(double)                    //用catch捕捉异常信息并作相应处理
  {cout<<"a="<<a<<",b="<<b<<",c="<<c<<",that is not a triangle!"<<endl;}
 cout<<"end"<<endl;
}

double triangle(double a,double b,double c)  //计算三角形的面积的函数
{double s=(a+b+c)/2;
 if (a+b<=c||b+c<=a||c+a<=b) throw a;   //当不符合三角形条件抛出异常信息
 return sqrt(s*(s-a)*(s-b)*(s-c));
}
```

程序运行结果如下：

```
6 5 4↙(输入a,b,c的值)
9.92157          (计算出三角形的面积)
1 1.5 2↙         (输入a,b,c的值)
0.726184         (计算出三角形的面积)
1 2 1↙           (输入a,b,c的值)
a=1,b=2,c=1, that is not a triangle!  (异常处理)
end
```

现在结合程序分析怎样进行异常处理。

(1) 首先把可能出现异常的、需要检查的语句或程序段放在**try**后面的花括号中。

(2) 程序开始运行后，按正常的顺序执行到**try**块，开始执行**try**块中花括号内的语句。如果在执行**try**块内的语句过程中没有发生异常，则**catch**子句不起作用，流程转到**catch**子句后面的语句继续执行。

(3) 如果在执行**try**块内的语句(包括其所调用的函数)过程中发生异常，则**throw**运算符抛出一个异常信息。**throw**抛出异常信息后，流程立即离开本函数，转到其上一级的函数(**main** 函数)。

throw抛出什么样的数据由程序设计者自定，可以是任何类型的数据。

(4) 这个异常信息提供给**try-catch**结构，系统会寻找与之匹配的**catch**子句。

(5) 在进行异常处理后，程序并不会自动终止，继续执行**catch**子句后面的语句。

由于**catch**子句是用来处理异常信息的，往往被称为**catch**异常处理块或**catch**异常处理器。

下面讲述异常处理的语法。

throw语句一般是由**throw**运算符和一个数据组成的，其形式为

throw 表达式;

try-catch的结构为

```
try
    {被检查的语句}
catch(异常信息类型 [变量名])
    {进行异常处理的语句}
```

说明：

(1) 被检测的函数必须放在try块中，否则不起作用。

(2) try块和catch块作为一个整体出现，catch块是try-catch结构中的一部分，必须紧跟在try块之后，不能单独使用，在二者之间也不能插入其他语句。但是在一个try-catch结构中，可以只有try块而无catch块。即在本函数中只检查而不处理，把catch处理块放在其他函数中。

(3) try和catch块中必须有用花括号括起来的复合语句，即使花括号内只有一个语句，也不能省略花括号。

(4) 一个try-catch结构中只能有一个try块，但却可以有多个catch块，以便与不同的异常信息匹配。

(5) catch后面的圆括号中，一般只写异常信息的类型名， 如

catch(double)

catch只检查所捕获异常信息的类型，而不检查它们的值。因此如果需要检测多个不同的异常信息，应当由**throw**抛出不同类型的异常信息。

异常信息可以是**C++**系统预定义的标准类型，也可以是用户自定义的类型**(**如结构体或类**)**。如果由**throw**抛出的信息属于该类型或其子类型，则**catch**与**throw**二者匹配，**catch**捕获该异常信息。

catch还可以有另外一种写法，即除了指定类型名外，还指定变量名，如

catch(double d)

此时如果**throw**抛出的异常信息是**double**型的变量**a**，则**catch**在捕获异常信息**a**的同时，还使**d**获得**a**的值，或者说**d**得到**a**的一个拷贝。什么时候需要这样做呢？有时希望在捕获异常信息时，还能利用**throw**抛出的值，如

```
catch(double d)
  {cout<<"throw "<<d;}
```

这时会输出**d**的值(也就是**a**值)。当抛出的是类对象时，有时希望在**catch**块中显示该对象中的某些信息。这时就需要在**catch**的参数中写出变量名(类对象名)。

(6) 如果在**catch**子句中没有指定异常信息的类型，而用了删节号“**...**”，则表示它可以捕捉任何类型的异常信息，如

catch(…) {cout<<"OK"<<endl;}

它能捕捉所有类型的异常信息，并输出**"OK"**。

这种**catch**子句应放在**try　catch**结构中的最后，相当于**“**其他**”**。如果把它作为第一个**catch**子句，则后面的**catch**子句都不起作用。

(7) try　catch结构可以与**throw**出现在同一个函数中，也可以不在同一函数中。当**throw**抛出异常信息后，首先在本函数中寻找与之匹配的**catch**，如果在本函数中无**try　catch**结构或找不到与之匹配的**catch**，就转到离开出现异常最近的**try　catch**结构去处理。

(8) 在某些情况下，在**throw**语句中可以不包括表达式，如

```
throw;
```

表示“我不处理这个异常，请上级处理”。

(9) 如果**throw**抛出的异常信息找不到与之匹配的**catch**块，那么系统就会调用一个系统函数**terminate**，使程序终止运行。

例**14.2** 在函数嵌套的情况下检测异常处理。

这是一个简单的例子，用来说明在**try**块中有函数嵌套调用的情况下抛出异常和捕捉异常的情况。请自己先分析以下程序。

```
#include <iostream>
using namespace std;
int main( )
{void f1( );
 try
  {f1( );}//调用f1( )
 catch(double)
  {cout<<"OK0!"<<endl;}
 cout<<"end0"<<endl;
 return 0;
}
void f1( )
{void f2( );
 try
  {f2( );}                    //调用f2( )
 catch(char)
  {cout<<"OK1!";}
 cout<<"end1"<<endl;
}
```

```
void f2( )
{void f3( );
 try
 {f3( );}                    //调用f3( )
 catch(int)
 {cout<<"Ok2!"<<endl;}
 cout<<"end2"<<endl;
}
void f3( )
{double a=0;
 try
  {throw a;}                 //抛出double类型异常信息
 catch(float)
  {cout<<"OK3!"<<endl;}
 cout<<"end3"<<endl;
}
```

分3种情况分析运行情况：

（1） 执行上面的程序。图14.1为有函数嵌套时异常处理示意图。

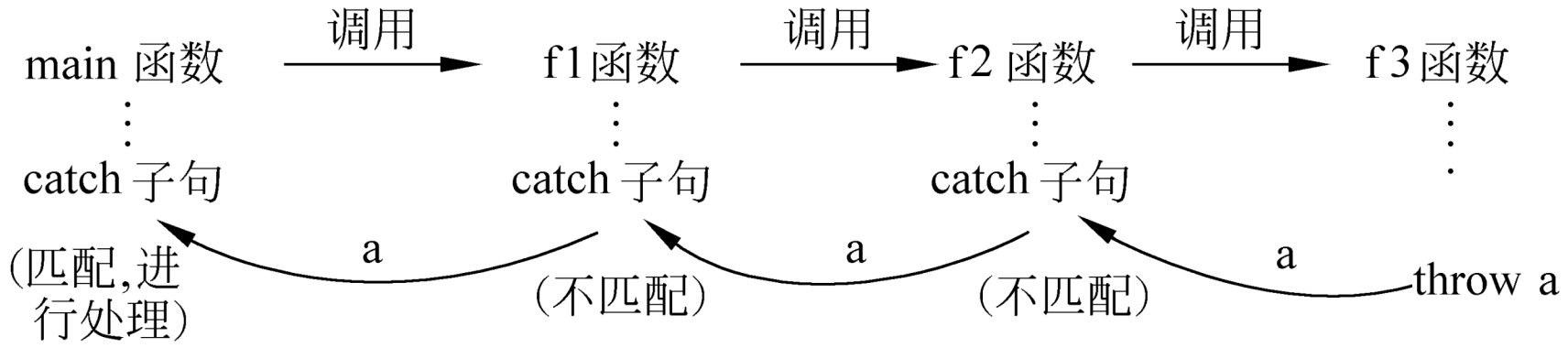

图14.1

程序运行结果如下：

OK0!(在主函数中捕获异常)

end0 (执行主函数中最后一个语句时的输出)

(2) 如果将**f3**函数中的**catch**子句改为**catch(double)**，而程序中其他部分不变，则程序运行结果如下：

```
OK3!(在f3函数中捕获异常)
end3          (执行f3函数中最后一个语句时的输出)
end2          (执行f2函数中最后一个语句时的输出)
end1          (执行f1函数中最后一个语句时的输出)
end0          (执行主函数中最后一个语句时的输出)
```

(3) 如果在此基础上再将**f3**函数中的**catch**块改为

```
catch(double)
  {cout<<"OK3!"<<endl;throw;}
```

程序运行结果如下：

```
OK3!(在f3函数中捕获异常)
OK0!          (在主函数中捕获异常)
end0          (执行主函数中最后一个语句时的输出)
```

14.1.3 在函数声明中进行异常情况指定

为便于阅读程序，使用户在看程序时能够知道所用的函数是否会抛出异常信息以及异常信息可能的类型，C++允许在声明函数时列出可能抛出的异常类型，如可以将例14.1中第二个程序的第3行改写为

```
double triangle(double,double,double) throw(double);
```

表示triangle函数只能抛出double类型的异常信息。如果写成

```
double triangle(double,double,double) throw(int,double,float,char);
```

则表示triangle函数可以抛出int,double,float或char类型的异常信息。异常指定是函数声明的一部分，必须同时出现在函数声明和函数定义的首行中，否则在进行函数的另一次声明时，编译系统会报告“类型不匹配”。

如果在声明函数时未列出可能抛出的异常类型，则该函数可以抛出任何类型的异常信息。如例**14.1**中第**2**个程序中所表示的那样。

如果想声明一个不能抛出异常的函数，可以写成以下形式**:**

```
double triangle(double,double,double) throw();//throw无参数
```

这时即使在函数执行过程中出现了**throw**语句，实际上也并不执行**throw**语句，并不抛出任何异常信息，程序将非正常终止。

14.1.4 在异常处理中处理析构函数

如果在try块(或try块中调用的函数)中定义了类对象，在建立该对象时要调用构造函数。在执行try块 (包括在try块中调用其他函数) 的过程中如果发生了异常，此时流程立即离开try块。这样流程就有可能离开该对象的作用域而转到其他函数，因而应当事先做好结束对象前的清理工作，C++的异常处理机制会在throw抛出异常信息被catch捕获时，对有关的局部对象进行析构(调用类对象的析构函数)， 析构对象的顺序与构造的顺序相反，然后执行与异常信息匹配的catch块中的语句。

例**14.3** 在异常处理中处理析构函数。

这是一个为说明在异常处理中调用析构函数的示例，为了清晰地表示流程，程序中加入了一些**cout**语句，输出有关的信息，以便对照结果分析程序。

```
#include <iostream>
#include <string>
using namespace std;
class Student
{public:
  Student(int n,string nam)//定义构造函数
{cout<<"constructor-"<<n<<endl;
num=n;name=nam;}
~Student( ){cout<<"destructor-"<<num<<endl;}//定义析构函数
  void get_data( );                    //成员函数声明
private:
int num;
string name;
 };
```

```
void Student::get_data( )                    //定义成员函数
{if(num==0) throw num;                       //如num=0,抛出int型变量num
 else cout<<num<<" "<<name<<endl;            //若num≠0，输出num,name
 cout<<"in get_data()"<<endl;                //输出信息，表示目前在get_data函数
中
}

void fun( )
{Student stud1(1101,"Tan");                  //建立对象stud1
stud1.get_data( );                           //调用stud1的get_data函数
Student stud2(0,"Li");                       //建立对象stud2
stud2.get_data( );                           //调用stud2的get_data函数
}

int main( )
{cout<<"main begin"<<endl;                   //表示主函数开始了
cout<<"call fun( )"<<endl;                   //表示调用fun函数
try
 {fun( );}                                   //调用fun函数
```

```
catch(int n)
  {cout<<"num="<<n<<",error!"<<endl;}      //表示num=0出错
cout<<"main end"<<endl;                    //表示主函数结束
return 0;
}
```

程序运行结果如下：

```
main begin
call fun( )
constructor-1101
1101 tan
in get_data()
constructor-0
destructor-0
destructor-1101
num=0,error!
main end
```

14.2 命名空间

在学习本书前面各章时，已经多次看到在程序中用了以下语句:

using namespace std;

这就是使用了命名空间**std**。在本节中将对它作较详细的介绍。

14.2.1 为什么需要命名空间

命名空间是**ANSI C++**引入的可以由用户命名的作用域，用来处理程序中常见的同名冲突。

在**C**语言中定义了**3**个层次的作用域，即文件**(**编译单元**)**、函数和复合语句。**C++**又引入了类作用域，类是出现在文件内的。在不同的作用域中可以定义相同名字的变量，互不干扰，系统能够区别它们。

下面先简单分析一下作用域的作用，然后讨论命名空间的作用。

如果在文件中定义了两个类，在这两个类中可以有同名的函数。在引用时，为了区别，应该加上类名作为限定，如

```
class A//声明A类
 {public:
void fun1( );        //声明A类中的fun1函数
  private:
int i;
 };
void A::fun1( )           //定义A类中的fun1函数
 {
//
 }
class B                   //声明B类
 {public:
void fun1( );        //B类中也有fun1函数
void fun2( );
 };
void B::fun1( )           //定义B类中的fun1函数
 {
   //
 }
```

这样不会发生混淆。

在文件中可以定义全局变量**(global variable)**，它的作用域是整个程序。如果在文件**A**中定义了一个变量**a**

```
int a=3;
```

在文件**B**中可以再定义一个变量**a**

```
int a=5;
```

在分别对文件**A**和文件**B**进行编译时不会有问题。但是，如果一个程序包括文件**A**和文件**B**，那么在进行连接时，会报告出错，因为在同一个程序中有两个同名的变量，认为是对变量的重复定义。问题在于全局变量的作用域是整个程序，在同一作用域中不应有两个或多个同名的实体**(entity)**，包括变量、函数和类等。

可以通过**extern**声明同一程序中的两个文件中的同名变量是同一个变量。如果在文件**B**中有以下声明**:**

```
extern int a;
```

表示文件**B**中的变量**a**是在其他文件中已定义的变量。由于有此声明，在程序编译和连接后，文件**A**的变量**a**的作用域扩展到了文件**B**。如果在文件**B**中不再对**a**赋值，则在文件**B**中用以下语句输出的是文件**A**中变量 **a**的值**:**

```
cout<<a;//得到a的值为3
```

在简单的程序设计中，只要人们小心注意，可以争取不发生错误。但是，一个大型的应用软件，往往不是由一个人独立完成的，假如不同的人分别定义了类，放在不同的头文件中，在主文件(包含主函数的文件)需要用这些类时，

就用**#include**命令行将这些头文件包含进来。由于各头文件是由不同的人设计的，有可能在不同的头文件中用了相同的名字来命名所定义的类或函数。这样在程序中就会出现名字冲突。

例**14.4** 名字冲突。

程序员甲在头文件**header1.h**中定义了类**Student**和函数**fun**。

```
//header1.h (头文件1，设其文件名为cc14-4-h1.h)
#include <string>
#include <cmath>
using namespace std;
class Student//声明Student类
{public:
  Student(int n,string nam,char s)
{num=n;name=nam;sex=s;}
  void get_data( );
private:
```

```
int num;
  string name;
  char sex;
 };
void Student::get_data( )            //成员函数定义
{cout<<num<<" "<<name<<" "<<sex<<endl;
}
double fun(double a,double b)        //定义全局函数(即外部函数)
{return sqrt(a+b);}
```

在**main**函数所在的文件中包含头文件**header1.h:**

```
#include <iostream>
#include "cc14-4-h1.h"   //注意要用双引号，因为文件一般是放在用户目录中的
using namespace std;
int main( )
{Student stud1(101,"Wang",18);  //定义类对象stud1
  stud1.get_data( );
  cout<<fun(5,3)<<endl;
  return 0;
  }
```

程序能正常运行，输出为

101 Wang 18

2.82843

如果程序员乙写了头文件**header2.h**，在其中除了定义其他类以外，还定义了类**Student**和函数**fun**，但其内容与头文件**header1.h**中的**Student**和函数**fun**有所不同。

```
//header2.h (头文件2，设其文件名为cc14-4-h2.h)
#include <string>
#include <cmath>
using namespace std;
class Student//声明Student类
{public:
  Student(int n,string nam,char s)      //参数与header1中的student不同
{num=n;name=nam;sex=s;}
  void get_data( );
```

```
private:
  int num;
  string name;
  char sex;                    //此项与header1不同
};

void Student::get_data( )          //成员函数定义
{cout<<num<<" "<<name<<" "<<sex<<endl;
}
double fun(double a,double b)      //定义全局函数
 {return sqrt(a-b);}               //返回值与header1中的fun函数不同
//头文件2中可能还有其他内容
```

假如主程序员在其程序中要用到**header1.h**中的**Student**和函数**fun**，因而在程序中包含了头文件**header1.h**，同时要用到头文件**header2.h**中的一些内容，因而在程序中又包含了头文件**header2.h**。如果主文件(包含主函数的文件)如下：

```
//main file
#include <iostream>
#include "cc14-4-h1.h"//包含头文件1
#include "cc14-4-h2.h"       //包含头文件2
using namespace std;
int main( )
{Student stud1(101,"Wang",18);
stud1.get_data();
cout<<fun(5,3)<<endl;
return 0;
 }
```

这时程序编译就会出错。 因为在预编译后，头文件中的内容取代了对应的**#include**命令行，这样就在同一个程序文件中出现了两个**Student**类和两个**fun**函数，显然是重复定义，这就是名字冲突，即在同一个作用域中有两个或多个同名的实体。

不仅如此，在程序中还往往需要引用一些库，为此需要包含有关的头文件。如果在这些库中包含有与程序的全局实体同名的实体，或者不同的库中有相同的实体名，则在编译时就会出现名字冲突。

为了避免这类问题的出现，人们提出了许多方法，例如：将实体的名字写得长一些；把名字起得特殊一些，包括一些特殊的字符；由编译系统提供的内部全局标识符都用下划线作为前缀，如**_complex()**，以避免与用户命名的实体同名；由软件开发商提供的实体的名字用特定的字符作为前缀。但是这样的效果并不理想，而且增加了阅读程序的难度，可读性降低了。

C语言和早期的C++语言没有提供有效的机制来解决这个问题，没有使库的提供者能够建立自己的命名空间的工具。人们希望ANSI C++标准能够解决这个问题，提供一种机制、一种工具，使由库的设计者命名的全局标识符能够和程序的全局实体名以及其他库的全局标识符区别开来。

14.2.2 什么是命名空间

为了解决上面这个问题，**ANSI C++**增加了命名空间**(namespace)**。所谓命名空间，实际上就是一个由程序设计者命名的内存区域。程序设计者可以根据需要指定一些有名字的空间域，把一些全局实体分别放在各个命名空间中，从而与其他全局实体分隔开来。如

```
namespace ns1//指定命名空间ns1
{int a;
double b;
}
```

现在命名空间成员包括变量**a**和**b**，注意**a**和**b**仍然是全局变量，仅仅是把它们隐藏在指定的命名空间中而已。

如果在程序中要使用变量**a**和**b**，必须加上命名空间名和作用域分辨符**“::”**，如**ns1::a**，**ns1::b**。这种用法称为命名空间限定**(qualified)**，这些名字**(**如**ns1::a)**称为被限定名**(qualified name)**。**C++**中命名空间的作用类似于操作系统中的目录和文件的关系。

命名空间的作用是建立一些互相分隔的作用域，把一些全局实体分隔开来，以免产生名字冲突。

可以根据需要设置许多个命名空间，每个命名空间名代表一个不同的命名空间域，不同的命名空间不能同名。这样，可以把不同的库中的实体放到不同的命名空间中。过去我们用的全局变量可以理解为全局命名空间，独立于所有有名的命名空间之外，它是不需要用**namespace**声明的，实际上是由系统隐式声明的，存在于每个程序之中。

在声明一个命名空间时，花括号内不仅可以包括变量，而且还可以包括以下类型：

- 变量(可以带有初始化)；
- 常量；
- 函数(可以是定义或声明)；
- 结构体；
- 类；
- 模板；
- 命名空间(在一个命名空间中又定义一个命名空间，即嵌套的命名空间)。

例如

```
namespace ns1
{const int RATE=0.08;//常量
double pay;                    //变量
double tax( )                  //函数
{return a*RATE;}
namespace ns2                  //嵌套的命名空间
   {int age;}
}
```

如果想输出命名空间**ns1**中成员的数据，可以采用下面的方法：

```
cout<<ns1::RATE<<endl;
cout<<ns1::pay<<endl;
cout<<ns1::tax()<<endl;
cout<<ns1::ns2::age<<endl;//需要指定外层的和内层的命名空间名
```

14.2.3 使用命名空间解决名字冲突

现在，对例**14.4**程序进行修改，使之能正确运行。

例**14.5** 利用命名空间来解决例**14.4**程序名字冲突问题。

修改两个头文件，把在头文件中声明的类分别放在两个不同的命名空间中。

```
//header1.h  (头文件1)
#include <string>
#include <cmath>
using namespace std;
namespace ns1//声明命名空间ns1
{class Student                    //在命名空间ns1内声明Student类
{public:
Student(int n,string nam,int a)
```

```
{num=n;name=nam;age=a;}
void get_data( );
    private:
int num;
string name;
int age;
   };
 void Student::get_data()    //定义成员函数
    {cout<<num<<" "<<name<<" "<<age<<endl;
    }

 double fun(double a,double b)  //在命名空间ns1内定义fun函数
   {return sqrt(a+b);}
}

//header2.h ((头文件2)
#include <string>
#include <cmath>
```

```
using namespace std;
namespace ns2                    //声明命名空间ns2
 {class Student
{public:
Student(int n,string nam,char s)
{num=n;name=nam;sex=s;}
void get_data( );
private:
int num;
char name[20];
char sex;
};
void Student::get_data( )
   {cout<<num<<" "<<name<<" "<<sex<<endl;
}
double fun(double a,double b)
{return sqrt(a-b);}
 }
```

```
//main file (主文件)
#include <iostream>
#include "cc14-5-h1.h"          //包含头文件1
#include "cc14-5-h2.h"          //包含头文件2
using namespace std;
int main( )
{ns1::Student stud1(101,"Wang",18);  //用命名空间ns1中声明的Student类定义stud1
stud1.get_data( );                  //不要写成ns1::stud1.get_data( );
cout<<ns1::fun(5,3)<<endl;          //调用命名空间ns1中的fun函数
ns2::Student stud2(102,"Li",'f');  //用命名空间ns2中声明的Student类定义stud2
stud2.get_data( );
cout<<ns2::fun(5,3)<<endl;          //调用命名空间ns1中的fun函数
return 0;
 }
```

程序能顺利通过编译，并得到以下运行结果：

101 Wang 18(对象**stud1**中的数据)
2.82843 (**5+3**的开方值)
102 Li f (对象**stud2**中的数据)
1.41421 (**5-3**的开方值)

14.2.4 使用命名空间成员的方法

在引用命名空间成员时，要用命名空间名和作用域分辨符对命名空间成员进行限定，以区别不同的命名空间中的同名标识符。即

命名空间名::命名空间成员名

这种方法是有效的，能保证所引用的实体有惟一的名字。但是如果命名空间名字比较长，尤其在有命名空间嵌套的情况下，为引用一个实体，需要写很长的名字。在一个程序中可能要多次引用命名空间成员，就会感到很不方便。

为此，C++提供了一些机制，能简化使用命名空间成员的手续。

（1） 使用命名空间别名

可以为命名空间起一个别名**(namespace alias)**，用来代替较长的命名空间名。如

```
namespace Television//声明命名空间，名为Television
{…}
```

可以用一个较短而易记的别名代替它。如

```
namespace TV = Television;//别名TV与原名Television等价
```

（2） 使用**using** 命名空间成员名

using后面的命名空间成员名必须是由命名空间限定的名字。例如

```
using ns1::Student;
```

using声明的有效范围是从**using**语句开始到**using**所在的作用域结束。如果在以上的**using**语句之后有以下语句**:**

Student stud1(101,"Wang",18);//此处的Student相当于ns1::Student

上面的语句相当于

ns1::Student stud1(101,"Wang",18);

又如

using ns1::fun;//声明其后出现的fun是属于命名空间ns1中的fun
cout<<fun(5,3)<<endl;　　//此处的fun函数相当于ns1::fun(5,3)

显然，这可以避免在每一次引用命名空间成员时都用命名空间限定，使得引用命名空间成员方便易用。

但是要注意：在同一作用域中用**using**声明的不同命名空间的成员中不能有同名的成员。例如

using ns1::Student;//声明其后出现的Student是命名空间ns1中的Student
using ns2::Student;　　//声明其后出现的Student是命名空间ns2中的Student
Student stud1;　　//请问此处的Student是哪个命名空间中的Student?

产生了二义性，编译出错。

（3） 使用**using namespace** 命名空间名

能否在程序中用一个语句就能一次声明一个命名空间中的全部成员呢？

C++提供了**using namespace**语句来实现这一目的。**using namespace**语句的一般格式为

using namespace命名空间名；

例如

```
using namespace ns1;
```

声明了在本作用域中要用到命名空间**ns1**中的成员，在使用该命名空间的任何成员时都不必用命名空间限定。如果在作了上面的声明后有以下语句：

```
Student stud1(101,"Wang",18);//Student隐含指命名空间ns1中的Student
cout<<fun(5,3)<<endl;          //这里的fun函数是命名空间ns1中的fun函数
```

在用**using namespace**声明的作用域中，命名空间**ns1**的成员就好像在全局域声明的一样。因此可以不必用命名空间限定。显然这样的处理对写程序比较方便。但是如果同时用**using namespace**声明多个命名空间时，往往容易出错。因此只有在使用命名空间数量很少，以及确保这些命名空间中没有同名成员时才用**using namespace**语句。

14.2.5 无名的命名空间

以上介绍的是有名字的命名空间，C++还允许使用没有名字的命名空间，如在文件A中声明了以下的无名命名空间：

```
namespace//命名空间没有名字
{void fun()                    //定义命名空间成员
   {cout<<"OK."<<endl;}
}
```

由于命名空间没有名字，在其他文件中显然无法引用，它只在本文件的作用域内有效。无名命名空间的成员fun函数的作用域为文件A。在文件A中使用无名命名空间的成员，不必(也无法)用命名空间名限定。如果在文件A中有以下语句：

```
fun();
```

则执行无名命名空间中的成员**fun**函数，输出**"OK."**。

在本程序中的其他文件中也无法使用该**fun**函数，
也就是把**fun**函数的作用域限制在本文件范围中。

14.2.6 标准命名空间std

为了解决C++标准库中的标识符与程序中的全局标识符之间以及不同库中的标识符之间的同名冲突，应该将不同库的标识符在不同的命名空间中定义(或声明)。标准C++库的所有的标识符都是在一个名为std的命名空间中定义的，或者说标准头文件(如iostream)中函数、类、对象和类模板是在命名空间std中定义的。

这样，在程序中用到C++标准库时，需要使用std作为限定。如

```
std::cout<<"OK."<<endl; //声明cout是在命名空间std中定义的流对象
```

在大多数的**C++**程序中常用**using namespace**语句对命名空间**std**进行声明，这样可以不必对每个命名空间成员一一进行处理，在文件的开头加入以下**using namespace**声明**:**

```
using namespace std;
```

这样，在**std**中定义和声明的所有标识符在本文件中都可以作为全局量来使用。但是应当绝对保证在程序中不出现与命名空间**std**的成员同名的标识符。由于在命名空间**std**中定义的实体实在太多，有时程序设计人员也弄不清哪些标识符已在命名空间**std**中定义过，为减少出错机会，有的专业人员喜欢用若干个**“using** 命名空间成员**”** 声明来代替**“using namespace** 命名空间**”**声明，如

```
using std::string;
using std::cout;
using std::cin;
```

等。为了减少在每一个程序中都要重复书写以上的**using**声明，程序开发者往往把编写应用程序时经常会用到的命名空间**std**成员的**using**声明组成一个头文件，然后在程序中包含此头文件即可。

14.3 使用早期的函数库

C语言程序中各种功能基本上都是由函数来实现的，在C语言的发展过程中建立了功能丰富的函数库，C++从C语言继承了这份宝贵的财富。在C++程序中可以使用C语言的函数库。

如果要用函数库中的函数，就必须在程序文件中包含有关的头文件，在不同的头文件中，包含了不同的函数的声明。

在C++中使用这些头文件有两种方法。

(1) 用**C**语言的传统方法。头文件名包括后缀**.h**，如**stdio.h**，**math.h**等。由于**C**语言没有命名空间，头文件并不存放在命名空间中，因此在**C++**程序文件中如果用到带后缀**.h**的头文件时，不必用命名空间。只需在文件中包含所用的头文件即可。如

```
#include <math.h>
```

(2) 用**C++**的新方法。**C++**标准要求系统提供的头文件不包括后缀**.h**，例如**iostream**、**string**。为了表示与**C**语言的头文件有联系又有区别，**C++**所用的头文件名是在**C**语言的相应的头文件名(但不包括后缀**.h)**之前加一字母**c**。

此外，由于这些函数都是在命名空间**std**中声明的，因此在程序中要对命名空间**std**作声明。如

```
#include <cstdio>
#include <cmath>
using namespace std;
```

目前所用的大多数C++编译系统既保留了C的用法，又提供了C++的新方法。下面两种用法等价，可以任选。

C传统方法

```
#include <stdio.h>
#include <math.h>
#include <string.h>
```

C++新方法

```
#include <cstdio>
#include <cmath>
#include <cstring>
using namespace std;
```

可以使用传统的C方法，但应当提倡使用C++的新方法。