

更多关于类

我们已经学习完 C++ 中大部分的重点。在过去的学习的过程中，我们为了讲述的系统性，把一些内容留到这一章来介绍。

这一章的内容比较零碎，包括：名字空间，类的静态成员，把类的声明和实现分开，使用预处理防止重定义错误，内联函数，友员函数和友员类。

1. 名字空间

当一个程序比较大或则多人同时开发一个程序的时候，我们经常会碰到命名冲突：多个变量或则函数的名字相同。C++ 中有解决这个问题的方法：把程序分割为多个互相不影响的名字空间。在不同的名字空间里可以有相同的标示符（变量、函数、对象的名字）。看下面的程序。

第 7-20 行的程序中包含一个全局变量、一个函数、和一个类。它们被一个名为 `space1` 的名字空间包围。我们可以看到，在同一个名字空间里面的标示符是相互可见的，如在函数 `print` 中可以访问全局变量 `data`。

那么，名字空间 `space1` 外的 `main` 函数能访问名字空间中的标示符，如类 `People`，吗？答案是否定的。整个程序都处

于一个名字空间里面：一个匿名的名字空间。而且嵌套的名字空间之间没有谁包含谁关系：匿名的名字空间中的标示符与 `space1` 中的标示符互不可见。也就是说，函数 `main` 与类 `People` 是处于不同的名字空间中，它们互不可见。

```
1 #include <iostream>
2 using namespace std;
3
4 namespace space1
5 {
6
7     int data = 10;
8
9     void print() {
10         cout << "data = " << data << endl;
11     }
12
13     class People
14     {
15     public:
16         void greet() {
17             cout << "Hello" << endl;
18         }
19
20     };
21
22 }
23
24 int main() {
25     using namespace space1;
26     People me;
27     me.greet();
28 }
```

要访问另一个名字空间中的标示符，我们可以使用语句 `using namespace`，如第 2 行与第 25 行所示。在第 2 行中有我们非常熟悉的 `std` 名字空间，我们学过的很多 C++ 的标准类都

是定义在这个名字空间里面。与 `#include` 不同，语句 `using namespace` 是可以出现在任何地方的，包括函数里面。语句 `using namespace` 是具有范围的。第 2 行的 `using namespace std` 的有效范围是第二行到整个文件的末位；而第 25 行的 `using namespace space1` 的有效范围仅为 `main` 函数的内部。再看下面的程序。

```
1 namespace space1
2 {
3     class People
4     {
5     public:
6         void greet() {
7             cout << "Hi" << endl;
8         }
9     };
10 }
11
12 class People
13 {
14 public:
15     void greet() {
16         cout << "What's up" << endl;
17     }
18 };
19
20 int main() {
21     space1::People me;
22     me.greet();
23 }
```

在 12-18 行中我们定义了另一个 `People` 类。由于与上面的 `People` 类不属于同一个名字空间，所以不会有重定义的错误。那么，如果我们要在 `main` 函数使用另一个名字空间 `space1` 中的 `People` 类，应该怎样访问呢？如果我们使用语句

`using namespace space1`, 那当我们使用标示符 `People` 时, 会看到编译器给出以下的错误: `reference to 'People' is ambiguous`。原因是 `using namespace space1` 后, `main` 函数既可以看到 `space1` 中 `People` 的, 也可以看到匿名的名字空间中的 `People`。解决方法是通过“全名” `space1::People` 来使用 `People` 类。再看下面的例子。

```
1 namespace space1
2 {
3     class People
4     {
5     public:
6         void greet() {
7             cout << "Hello" << endl;
8         }
9     };
10
11 namespace space2
12 {
13     class People
14     {
15     public:
16         void greet() {
17             cout << "Hey" << endl;
18         }
19     };
20 }
21 }
22
23 int main() {
24     using namespace space1::space2;
25     People().greet();
26     space1::space2::People().greet();
27 }
```

在一个名字空间里面可以定义另一个名字空间, 如 `space1` 里面的 `space2`。要访问中 `space2` 的 `People`, 我们需要使用它

的名字空间的路径：`space1::space2`，如 24 行所示。注意，只使用外层的名字空间（`space1`），是不能使用内层名字空间（`space1::space2`）中的标示符的。另外一种访问 `space1::space2` 中的 `People` 的方法是给出这个类在名字空间中的完整路径，如 26 行所示。再看下面的例子。

```
1 namespace space1
2 {
3     class People {};
4 }
5
6 namespace space2
7 {
8     class People {};
9 }
10
11 namespace space1
12 {
13     class Genius : public People {};
14 }
15
16 using namespace space1;
17
18 int main() {
19     Genius you;
20 }
```

同一个名字空间是可以在不同地方定义的。如我们非常熟悉的 `std` 就在不同的文件中都有定义，如 `iostream`、`vector`、`string` 等。

2. 静态成员

类有时也可以作为名字空间：对于类的静态成员，类就好

像是名字空间。看下面的例子。

```
1 class Student
2 {
3 public:
4     static int nextId;
5
6     static int getNextId() {
7         ++ nextId;
8         return nextId;
9     }
10 };
11
12 int Student::nextId = 0;
13
14 int main() {
15     cout << Student::nextId << endl;
16     cout << Student::getNextId() << endl;
17 }
```

在第 4 行中，我们声明了一个静态成员变量；在第 6 行中我们声明了一个静态成员函数。可以看到，静态成员的声明前面加上了保留字 **static**。静态成员变量除了要在类中声明外，还必须在类外面定义，如第 12 行所示。如果没有 12 行，那么编译器会报链接错误，因为这时静态成员变量只有声明没有定义。在 15 和 16 两行我们可以看到，静态成员可以看成是以类作为名字空间的变量，它们可以通过全名（完整路径）来访问。在 12 行我们给静态成员变量赋了一个初始值。静态成员变量实际上是全局变量，一个定义在某个名字空间里面的全局变量。所以如果不赋初始值，它会自动被初始为 0。

与名字空间不同，类里面的静态成员还可以受类的访问控

制的限制。看下面的程序。

```
1 class Student
2 {
3 private:
4     static int nextId;
5
6 public:
7     static int getNextId() {
8         ++ nextId;
9         return nextId;
10    }
11 };
12
13 int Student::nextId = 0;
14
15 int main() {
16     cout << Student::nextId << endl;
17     cout << Student::getNextId() << endl;
18 }
```

如果我们把静态成员变量定义为私有的，那么它只能在一个类里面的其它函数（包括静态成员函数和对象成员函数）中访问。如上例中的 `nextId` 能在第 7-10 行的静态成员函数 `getNextId` 中访问，而不能在类外面的 `main` 函数中访问。

实际上静态成员变量和普通成员变量是很不同的：在类中声明一个成员变量是声明该类的每个对象都拥有一个这样的成员变量，而静态成员变量只是把类作为一个名字空间和访问控制的工具。一个类中的静态成员变量是不属于该类的对象的。看下面的例子。

在 22 行中，我们发现 `Student` 类的一个对象的大小只有 4

各字节。这说明了为 `Student` 类的每对象分配的内存空间中只有成员变量 `id`，没有静态成员变量 `nextId`。

```
1 class Student
2 {
3 private:
4     static int nextId;
5     int id;
6
7 public:
8     Student() {
9         ++ nextId;
10        id = nextId;
11    }
12
13    void print() {
14        cout << "I am " << id << endl;
15    }
16 };
17
18 int Student::nextId = 0;
19
20 int main() {
21     Student s;
22     cout << sizeof s << endl; // 4
23 }
```

在上例中，我们看到了静态成员变量的一种用法。当一个类需要使用一个全局变量，但是又不想让这个全局变量被别的程序错误地修改，我们可以把这个全局变量封装起来，方法是把它定义为该类的私有静态成员变量。

类里面定义的成员函数只有对象函数和静态函数两种，它们的区别在于：

1. 每个对象成员函数都有一个隐形的参数 `this`，它是一

个地址，这个地址的存放的是所在类的一个对象。

2. 静态成员函数完全不同，它只是以类为名字空间，并且可以被类封装。静态成员函数是没有隐形的参数 `this` 的。

```
1 class Student
2 {
3 public:
4     static int nextId;
5     int id;
6
7     static int getNextId() {
8         this->id = 10;
9         id = 10;
10
11         this->print();
12         print();
13
14         ++ nextId;
15         return nextId;
16     }
17
18     Student() {
19         this->id = getNextId();
20         id = getNextId();
21
22         this->print();
23         print();
24     }
25
26     void print() {
27         cout << "I am " << id << endl;
28     }
29 };
30
31 int Student::nextId = 0;
```

因为每个对象函数都有隐形参数 `this`，它们可以通过 `this` 访问成员变量和对象成员函数，如第 19 行和第 22 行所示。

注意，必须给定一个对象，才可以访问成员变量和对象成员函数。在一个对象函数中，当访问成员变量和对象成员函数的时候，我们可以省略 `this->`，如第 20 行和第 23 行所示。

但是，在一个静态的成员函数中是不可以用第 8-12 行的方式访问成员变量和对象成员函数的，因为静态的成员函数中没有隐形参数 `this`。

最后，我们总结一下静态的成员变量和静态的成员函数：

1. 类中声明了的静态的成员变量必须在类外面定义。
2. 静态的成员变量是不属于对象的。
3. 静态的成员变量和静态的成员函数可被类封装。
4. 对象成员函数中可以访问静态的成员变量和静态的成员函数，因为它们本质上是全局变量和普通函数。
5. 静态的成员函数没有隐形参数 `this`，因此不能像对象成员函数一样访问所在类的成员变量和对象函数。

3. 类的声明和实现的分离

在很多 C++ 程序中，定义一个类的时候通常把声明和实现分开。看下面的程序。

在 2-13 行的类中，我们只给出各个函数的声明，但没有给出它们的定义。之后，这些函数被定义在类外部的 16-29

行中。注意这与声明纯虚对象函数是不同的，纯虚对象函数只会在子类中给出函数的定义，而本例中我们只是把函数的定义放到类的外部。在 16-29 行的函数定义中，函数的头部必须给出函数全名，如本例中高亮显示的函数名字。

```
1 // Student.h
2 class Student
3 {
4     private:
5         static int nextId;
6         int id;
7
8         static int getNextId();
9
10    public:
11        Student();
12        void print();
13 };
14
15 // Student.cpp
16 int Student::nextId = 0;
17
18 int Student::getNextId() {
19     ++ nextId;
20     return nextId;
21 }
22
23 Student::Student() {
24     id = getNextId();
25 }
26
27 void Student::print() {
28     cout << "I am " << id << endl;
29 }
```

通常一个类可以被写在两个文件，一个叫类声明的头文件，通常以后缀 .h 结尾，它用来存放如 2-13 行这种类的声明。另外一个文件通常以后缀 .cpp 结尾，它用来存放类中函数的。

把类的声明和实现分开，实现了 C 语言的一种“先定义，再实现”的程序风格。类声明的头文件提供一种只供阅读的程序接口。

4. 防止重定义错误的预处理

当我们使用预处理命令**#include**的时候，实际上把一个文件插入到另一个文件，插入的位置是预处理命令**#include**所在的行。当使用预处理命令**#include**包含同一个文件两次以上的时候，文件里面的所有类、函数、和全局变量就会被定义多次。为了避免这种错误，我们介绍几个预处理命令。

```
1 #ifndef STUDENT
2 #define STUDENT
3
4 class Student {};
```

如果我们要避免 **Student** 类被多次包含，那么我们可以使用上例中的 1、2、6 行的 3 个成组的预处理命令。其中，第 1 行和第 6 行的预处理命令命令是成对的，它们的意思是：如果 **STUDENT** 没有被定义过（**ifndef = if not defined**），那么编译器会扫描它们之间的各行，否则编译器把它们之间的所有行看成空白。这里的“定义”是一种预处理方式的定义，如第 2 行所示。

这 2 个预处理命令配合的效果是：当编译器第一次扫描这

段程序的时候，**STUDENT** 没有被定义过，然后编译器开始读这段程序的各行。在这个过程中，第 2 行定义了 **STUDENT**。这使得在编译器第 2 次扫描这段程序的时候，**STUDENT** 必定被定义过了。也就是说在第 2 次扫描这段程序的时候，编译器会跳过这段程序。因此，最后得效果是：**Student** 类只会被编译器编译一次，不会被认为多次重定义。

5. 内联函数

内联函数 **inline function** 是一种通过去除函数调用开销的程序优化方法。对于一般函数，我们可以通过在函数定义前加上保留字 **inline** 来建议编译器把函数作为内联函数。

对于对象成员函数和静态成员函数，建议编译器把函数作为内联函数的方法是把它们定义放在类的内部，无需在函数前面加 **inline**。注意编译器不一定接受建议，它只会把简单的（即行数少并没有循环的）程序作为内联函数。

6. 友员函数与友员类

友员关系用来破坏封装的：一个类的友员函数可以访问这个类的私有成员；一个类的友员类中的所有函数都是这个类的友员函数。看下面的例子。

在第 3 行中使用关键字 **friend** 和函数 **print** 的函数头部声

明了函数 `print` 是一个友员函数,所以在第 19 行中函数 `print` 可以访问 `Student` 的对象中的私有成员 `grade`。

```
1 class Student
2 {
3     friend void print(Student & s);
4     friend class Teacher;
5
6 private:
7     int grade;
8 };
9
10 class Teacher
11 {
12 public:
13     void assign(Student & s, int grade) {
14         s.grade = grade;
15     }
16 };
17
18 void print(Student & s) {
19     cout << s.grade << endl;
20 }
21
22 int main() {
23     Student s;
24     Teacher().assign(s, 100);
25     print(s);
26 }
```

在第 4 行中使用关键字 `friend` 和声明了类 `Teacher` 是一个友员类,所以在第 14 行类 `Teacher` 中的函数 `assign` 可以访问 `Student` 的对象中的私有成员 `grade`。请注意,类之间的友员关系是非对称和非传递的。也就是说,上例中类 `Teacher` 是类 `Student` 的友员,但反之不成立。如果要使得 `Student` 是类 `Teacher` 的友员,必须在类 `Teacher` 声明。