类

2021年11月6日 星期六 16:41

1、定义在类内部的函数是隐式的inline函数。

2、C++允许把cinst关键字放在成员函数的参数列表之后，此时，紧跟在参数列表后面的const表示this是一个指向常量的指针。像这样使用const的成员函数被称作常量成员函数。

常量对象，以及常量对象的引用或指针都只能调用常量成员函数。

3、一般来说，如果非成员函数是类接口的组成部分，则这些函数的声明应该与类在同一个头文件内。

4、类可以包含多个构造函数，和其他重载函数差不多，不同的构造函数之间必须在参数数量或参数类型上有所区别。

5、不同于其他成员函数，构造函数不能被声明成const的。当我们创建类的一个const对象时，直到构造函数完成初始化过程，对象才能真正取得其“常量”属性。因此，构造函数在const对象的构造过程中可以向其写值。

6、只有当类没有声明任何构造函数时，编译器才会自动地生成默认构造函数。

如果类包含有内置类型或者复合类型的成员，则只有当这些成员全都被赋予了类内初始值时，这个类才适合于使用合成的默认构造函数。

7、构造函数不应该轻易覆盖掉类内的初始值，除非新赋的值与原值不同。如果你不能使用类内初始值，则所有构造函数都应该显式地初始化每个内置类型的成员。

8、使用class和struct定义类唯一的区别就是默认的访问权限（class为private，struct为public）。

9、友元：允许其他类或者函数访问它的非公有成员。friend+函数声明

一般来说，最好在类定义开始或结束前的位置集中声明友元。

友元的声明仅仅指定了访问的权限，而非一个通常意义上的函数声明。如果我们希望类的用户能够调用某个友元函数，那么我们就必须在友元声明之外再专门对函数进行一次声明。

许多编译器并未强制限定友元函数必须在使用之前在类的外部声明，不过最好还是在之前提供一个独立的函数声明。

10、定义在类内部的成员函数是自动inline的。

我们可以在类的内部把inline作为声明的一部分显式地声明成员函数，同样的，也能在类的外部用inline关键字修饰函数的定义。

不过最好只在类外部定义的地方说明inline，这样更容易理解。inline成员函数也应该与相应的类定义在同一个头文件中。

11、mutable关键字可以使成员变为可变数据成员，一个可变数据成员永远不会是const，即使它是const对象的成员。

因此，一个const成员函数可以改变一个可变成员的值。

12、返回*this的成员函数。

接下来我们继续添加一些函数，它们负责设置光标所在位置的字符或者其他任一给定位置的字符：

```
class Screen {
public:
    Screen &set(char);
    Screen &set(pos, pos, char);
    // 其他成员和之前的版本一致
};
inline Screen &Screen::set(char c)
{
    contents[cursor] = c;              // 设置当前光标所在位置的新值
    return *this;                      // 将 this 对象作为左值返回
}
inline Screen &Screen::set(pos r, pos col, char ch)
{
    contents[r*width + col] = ch;      // 设置给定位置的新值
    return *this;                      // 将 this 对象作为左值返回
}
```

和 move 操作一样，我们的 set 成员的返回值是调用 set 的对象的引用（参见 7.1.2 节，第 232 页）。返回引用的函数是左值的（参见 6.3.2 节，第 202 页），意味着这些函数返回的是对象本身而非对象的副本。如果我们把一系列这样的操作连接在一条表达式中的话：

```
// 把光标移动到一个指定的位置，然后设置该位置的字符值
myScreen.move(4,0).set('#');
```

这些操作将在同一个对象上执行。在上面的表达式中，我们首先移动 myScreen 内的光标，然后设置 myScreen 的 contents 成员。也就是说，上述语句等价于

```
myScreen.move(4,0);
myScreen.set('#');
```

如果我们令 move 和 set 返回 Screen 而非 Screen&，则上述语句的行为将大不相同。在此例中等价于：

```
// 如果 move 返回 Screen 而非 Screen&
Screen temp = myScreen.move(4,0);     // 对返回值进行拷贝
temp.set('#');                        // 不会改变 myScreen 的 contents
```

13、一个类的成员类型不能是该类自己。但是类允许包含指向它自身类型的引用或指针。

```
class Person {
    string m_name;
    Person* next;
    Person* prev;
};
```

14、类之间的友元关系：

如果一个类指定了友元类，则友元类的成员函数可以访问此类包括非公有成员在内的所有成员。

```
class Screen {
    // Window_mgr 的成员可以访问 Screen 类的私有部分
    friend class Window_mgr;
    // Screen 类的剩余部分
};
```

必须要注意的一点是，友元关系不存在传递性。也就是说，如果Window_mgr有它自己的友元，则这些友元并不能理所当然地具有访问Screen的特权。

令成员函数作为友元：

当把一个成员函数声明成友元时，必须明确指出该成员函数属于那个类。

```
class Screen {
    // Window_mgr::clear 必须在 Screen 类之前被声明
    friend void Window_mgr::clear(ScreenIndex);
    // Screen 类的剩余部分
};
```

函数重载和友元：

尽管重载函数的名字相同，但它们仍然是不同的函数。因此，如果一个类想把一组重载函数声明成它的友元，它需要对这组函数中的每一个分别声明。

```
// 重载的 storeOn 函数
extern std::ostream& storeOn(std::ostream &, Screen &);
extern BitMap& storeOn(BitMap &, Screen &);
class Screen {
    // storeOn 的 ostream 版本能访问 Screen 对象的私有部分
    friend std::ostream& storeOn(std::ostream &, Screen &);
    // ...
};
```

友元函数和作用域：

> 类和非成员函数的声明不是必须在它们的友元声明之前。当一个名字第一次出现在一个友元声明中时，我们隐式地假定该名字在当前作用域中是可见的。然而，友元本身不一定真的声明在当前作用域中（参见 7.2.1 节，第 241 页）。
>
> 甚至就算在类的内部定义该函数，我们也必须在类的外部提供相应的声明从而使得函数可见。换句话说，即使我们仅仅是用声明友元的类的成员调用该友元函数，它也必须是被声明过的：

```
struct X {
    friend void f() { /* 友元函数可以定义在类的内部*/ }
    X() { f(); }                    // 错误：f 还没有被声明
    void g();
    void h();
};
void X::g() { return f(); }         // 错误：f 还没有被声明
void f();                           // 声明那个定义在 X 中的函数
void X::h() { return f(); }         // 正确：现在 f 的声明在作用域中了
```

> 关于这段代码最重要的是理解友元声明的作用是影响访问权限，它本身并非普通意义上的声明。

15、一个类就是一个作用域的事实能够很好地解释为什么当我们在类的外部定义成员函数时必须同时提供类名和函数名。

16、编译器处理完类中的全部声明后才会处理成员函数的定义。

17、如果成员是const、引用，或者属于某种未提供默认构造函数的类类型，我们必须通过构造函数初始值列表为这些成员赋值。

18、

> **建议：使用构造函数初始值**
>
> 在很多类中，初始化和赋值的区别事关底层效率问题：前者直接初始化数据成员，后者则先初始化再赋值。
>
> 除了效率问题外更重要的是，一些数据成员必须被初始化。建议读者养成使用构造函数初始值的习惯，这样能避免某些意想不到的编译错误，特别是遇到有的类含有需要构造函数初始值的成员时。

19、了解存在委托构造函数这一概念。

20、在实际中，如果定义了其他构造函数，那么最好也提供一个默认构造函数。

21、聚合类（略）、字面值常量类（略）

22、类的静态成员

> 如果某个类实现了多个这个类的对象，那么每一个对象都分别有自己的数据成员，不同对象的数据成员各自有值，互不相干。但是有时人们希望有某一个或几个数据成员为所有对象所共有。这样可以实现数据共享。
>
> 在前面介绍过全局变量能够实现数据共享。
>
> 如果在一个程序文件中有多个函数，在每一个函数中都可以改变全局变量的值，全局变量的值为各函数共享。但是用全局变量的安全性得不到保证。由于在各处都可以自由地修改全局变量的值，很有可能偶一失误，全局变量的值就被修改，导致程序的失败。因此在实际工作中很少使用全局变量。
>
> 如果想在同类的多个对象之间实现数据共享，也不要用全局对象，可以用静态的数据成员。

我们通过在成员的声明之前加上关键字static使得其与类关联在一起。类的静态成员存在于任何对象之外，对象中不包含任何与静态数据成员有关的数据。通常情况下我们最好在类的外部定义静态成员（只能定义一次），注意在定义时不能重复static关键字，该关键字只出现在类内部的声明语句中。

一般来说，不能在类的内部初始化静态成员。

23、构造函数初始化的两种写法：

写法一（我赌一包辣条很多人没见过这种写法）：

```
#include <iostream>
using namespace std;

template <typename Type>
class testClass {
public:
    // 构造函数
    testClass(Type key);

    // 参数
    int defaultValue;
    Type keyValue;
};

template <typename Type>
testClass<Type>::testClass(Type key):defaultValue(10),keyValue(key){}

int main() {
    testClass<int> test(20);
    cout<<"defaultValue: "<<test.defaultValue<<endl;
    cout<<"keyValue: "<<test.keyValue<<endl;
    return 0;
}
```

写法二（熟悉的味道！课本上教的就是这种写法）：

```
#include <iostream>
using namespace std;

template <typename Type>
class testClass {
public:
    // 构造函数
    testClass(Type key);

    // 参数
    int defaultValue;
    Type keyValue;
};

template <typename Type>
testClass<Type>::testClass(Type key) {
    this->defaultValue = 10;
    this->keyValue = key;
}

int main() {
    testClass<int> test(20);
    cout<<"defaultValue: "<<test.defaultValue<<endl;
    cout<<"keyValue: "<<test.keyValue<<endl;
    return 0;
}
```

方法一叫成员初始化列表，可以将方法一的构造函数格式归纳为： 类名::类名（形参表）：内嵌对象1（形参表），内嵌对象2（形参表）...{类的初始化}

方法二即为比较容易理解的赋值方式。

下面考量一下这两种写法在运行时的区别。

一般地这两种构造方法并没有太大的区别，在效率上也基本一致。在这里我推荐你使用第一种初始化列表方法，那么为什么会这样累？在《Effective C++》中条款04中有解释以上两种初始化的区别：

方法一应该叫作**初始化**，方法二应该叫作**赋值**。二者是有本质的区别的。

C++规定，对象的成员变量的初始化动作发生在进入构造函数本体之前。因此方法二的构造函数内属性不能算是被初始化，是被赋值了。初始化发生的时间更早，发生在这些成员的**default构造函数**被自动调用之时。使用方法二的构造函数需要执行两次copu构造：defaultValue以10为初值进行copy构造，keyValue以key为初值进行copy构造。因此理论上使用方法一的成员初始化列表（member initialization list）方式的效率更高，因为比起先调用default构造函数然后再调用copy assignment操作符，单只调用一次copy构造函数是比较高效的。

也就是说采用方法二的话，构造函数本体实际上不需要有任何操作，因此效率更高。借用《Effective C++》中的图片说明一下：

```
ABEntry::ABEntry(const std::string& name, const std::string& address,
                 const std::list<PhoneNumber>& phones)
  :theName(name),
   theAddress(address),          //现在，这些都是初始化（initializations）
   thePhones(phones),
   numTimesConsulted(0)
{ }                              //现在，构造函数本体不必有任何动作
```

26822029

24、

练习 7.46：下面哪些论断是不正确的？为什么？

（a）一个类必须至少提供一个构造函数。

（b）默认构造函数是参数列表为空的构造函数。

（c）如果对于类来说不存在有意义的默认值，则类不应该提供默认构造函数。

（d）如果类没有定义默认构造函数，则编译器将为其生成一个并把每个数据成员初始化成相应类型的默认值。

【出题思路】

本题旨在考查读者对默认构造函数原理的熟悉程度。

【解答】

(a)是错误的，类可以不提供任何构造函数，这时编译器自动实现一个合成的默认构造函数。

(b)是错误的，如果某个构造函数包含若干形参，但是同时为这些形参都提供了默认实参，则该构造函数也具备默认构造函数的功能。

(c)是错误的，因为如果一个类没有默认构造函数，也就是说我们定义了该类的某些构造函数但是没有为其设计默认构造函数，则当编译器确实需要隐式地使用默认构造函数时，该类无法使用。所以一般情况下，都应该为类构建一个默认构造函数。

(d)是错误的，对于编译器合成的默认构造函数来说，类类型的成员执行各自所属类的默认构造函数，内置类型和复合类型的成员只对定义在全局作用域中的对象执行初始化。

25、

练习 7.56：什么是类的静态成员？它有何优点？静态成员与普通成员有何区别？

【出题思路】

本题考查静态成员的含义及用法。

【解答】

静态成员是指声明语句之前带有关键字 `static` 的类成员，静态成员不是任意单独对象的组成部分，而是由该类的全体对象所共享。

静态成员的优点包括：作用域位于类的范围之内，避免与其他类的成员或者全局作用域的名字冲突；可以是私有成员，而全局对象不可以；通过阅读程序可以非常容易地看出静态成员与特定类关联，使得程序的含义清晰明了。

静态成员与普通成员的区别主要体现在普通成员与类的对象关联，是某个具体对象的组成部分；而静态成员不从属于任何具体的对象，它由该类的所有对象共享。另外，还有一个细微的区别，静态成员可以作为默认实参，而普通数据成员不能作为默认实参。