

第六章 类成员(二)

面向对象程序设计(C++)





6 类成员（二）

6.1 成员变量与成员函数

6.2 **this**指针

6.3 成员对象

6.4 **const**(常量)

6.5 **const**对象与**const**成员函数

6.6 静态成员变量与静态成员函数



6.4 const量(常量)

从高质量程序设计的角度考察C++中的const!



6.4.1 `const`的意义

👉 最低权限原则：软件工程的基本原则之一。

`const`的意义：在可更改与不可更改之间画一条明确的界线，提高程序的安全性和可控性。

eg:

```
const int i=100;  
i++; //编译错误
```

6.4.2 C中的const(常量)

☞ “一个不能被改变的普通变量”。

☞ 因此：

- 总是占用存储；
- 名字是全局的。也就是说，默认情况下，const是外部连接的(容易引起“名字冲突”)。

```
const int bufsize; // 无需初始化
```

```
const int bufsize = 100 ;  
char buf[bufsize] ;      // error!! Why??
```

在编译时，编译器并不知道const的值，它只是一个“运行时常量”。

6.4.3 C++的Const

通常，C++编译器不为const创建存储空间，而是把它保存在“符号表”里，即“**编译时常量**”。

```
const int bufsize;           // 非法，未赋初值
const int bufsize = 100;
char strbuf[bufsize];       // OK, Why?
```

默认情况下，C++中的const是内部连接的，也就是说，const仅在const被定义过的文件里才是可见的。（因此，不用担心名字冲突）

当定义一个const时，必须赋一个值给它，除非用extern做出了清楚的说明。当用extern说明了const时，编译器会强制为const分配空间，而不是保存在符号表中。

```
extern const int bufsize;    // 未赋初值，但extern声明
```

了bufsize在另一个文件中定义及赋初值。



6.4.3 C++的Const(续)

☞ **const**用于集合，必须为其分配内存，(因为编译器“不愿意”把集合保存到符号表中，太复杂)。

```
const int i[] = { 1, 2, 3, 4 };
```

```
float f[i[3]]; // 非法，编译期间无法知道存储空间的值。
```

```
struct S { int i, j; };
```

```
const S s[] = { { 1, 2 }, { 3, 4 } };
```

```
double d[s[1].j]; // 非法，理由同上
```

```
int main() {
```

```
} ///:~
```



6.4.3 C++的Const(续)

☞ **const**用于集合，必须为其分配内存，(因为编译器“不愿意”把集合保存到符号表中，太复杂)。

```
const int i[] = { 1, 2, 3, 4 };
```

```
int main() {  
    float f[i[3]]; // 这样呢?  
} ///:~
```


6.4.4 C++中const的作用



值替代: C++的const vs C中的宏替换

```
#define BUFSIZE 100; // 宏替换  
char str[BUFSIZE];
```

vs

```
const int bufsize = 100; // C++的const  
char str[bufsize];
```



在宏替换中，BUFSIZE没有类型信息，不能进行类型检查；



宏定义是全局的，容易名字冲突。

6.4.4 C++中const的作用(续)



安全性

如果想用运行期间产生的值初始化一个变量，并且知道在该变量的生命期内其值不变，则可用const限定该变量，达到最大限度地保证改变量安全性的目的。

```
int main() {  
    cout << "type a character & CR:";  
    const char c = cin.get(); //用运行期间产生的值初始化，之后不变  
    const char c2 = c + 'a';  
    cout << c2;  
} ///:~
```



6.4.5 const的应用: const指针

1. 指向const的指针(指针指向的内存地址的值不能改变)

```
const int *u;    // u是一个指针, 它指向const int;  
int const *v;    // v是一个指向恰好是const的int的  
                普通指针;
```

2. const指针(指针的值不能改变)

```
int d = 1;  
int * const w = &d; // w是一个指针, 这个指针是  
                  指向int的const指针
```

注意: C++中, const指针必须赋初值



6.4.5 const的应用: const指针(续)

3. const指针指向const对象

```
int d = 1;
```

```
const int * const x = &d;
```

```
int const * const x2 = &d;
```

```
// x,x2,*x,*x2都不能改变
```



注意

- ☞ 非const对象的地址可以赋给const指针;
- ☞ const对象的地址绝不可以赋给非const指针。
(因为这样做可能导致通过非const指针改变const对象的值的后果)

```
int d = 1;  
const int e = 2;  
int* u = &d;           // OK -- d not const  
//! int* v = &e;       // illegal -- e const  
int* w = (int*)&e;      // legal but bad practice  
int main() {} ///:~
```



6.4.6 const的应用: const参数

1. 传递const值

```
void f1(const int i) {  
    i++; // 编译时错误, i不能改变  
}
```

“形参” 不能被改变 or “实参” 不能被改变???



6.4.7 **const**的应用: **const**返回值

2. 按值返回内部**const** 常量

```
int f3() { return 1; }  
const int f4() { return 1; } // 返回const int ;  
  
int main() {  
    const int j = f3();    // Works fine  
    int k = f4();          // But this works fine  
    too!  
} ///:~
```

对内部类型来说, 按值返回const量并没有什么特别的意义。



6.4.7 const的应用:const返回值(续)

3. 按值返回自定义类型的**const**:

实际上阻止了返回值作为左值出现。

```
class X {  
    int i;  
public:  
    X(int ii = 0);  
    void modify();  
};
```

```
X::X(int ii) { i = ii; }  
void X::modify() { i++; }
```



```
X f5() {  
    X x(2)  
    return x;    // 返回变量  
}
```

```
const X f6() {  
    return X();    // 按值返回const  
}
```

```
void f7(X& x) { // 按值传递非const引用  
    x.modify();  
}
```

```
int main() {  
    f5() = X(1);    // 正确, f5()返回非const量;  
    f5().modify();  // 正确  
  
    f7(f5());        // 可能会有Warning,跟编译选项有关  
  
    f6() = X(1);      // Error: f6()是常量, 不能作左值  
    f6().modify();    // Error: f6()是常量, 不能被修改  
    f7(f6());        // Error: Why??  
} ///:~
```

error 2664: cannot
convert parameter 1
from 'const class X' to
'class X &'



6.4.7 const的应用: const指针

4. 传递和返回const指针

```
char * strcpy(char * dest, const char * src);
```

```
void t(int*) { }
```

```
void u(const int* cip) {
```

```
    *cip = 2;           // Error: 试图改变值
```

```
    int i = *cip;       // OK -- copies value
```

```
    int* ip2 = cip;     // Error: 试图让非const *  
                        指向 const *
```

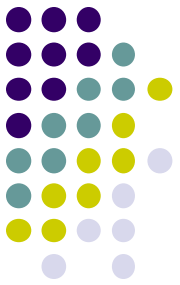
```
}
```

```
const int* const w() {
```

```
    static int i;
```

```
    return &i;          // 返回静态局部量的地址
```

```
}
```



```
int main() {  
    int x = 0;  
    int* ip = &x;  
    const int* cip = &x;  
    t(ip);    // OK  
    //! t(cip);    // Not OK  
    u(ip);    // OK  
    u(cip); // Also OK  
  
    //! int* ip2 = w();    // Not OK  
    const int* const ccip = w(); // OK  
    const int* cip2 = w(); // OK  
    //! *w() = 1; // Not OK  
}
```

当传递一个或返回一个地址时(指针或引用), 设置为 const 可以阻止客户程序员修改其值。

☞ 对参数传递而言，**C++**建议用**const**引用传递替代值传递。

☞ 问：有什么好处？

☞ 答：兼顾了效率和易用性

- 传递地址比传递整个对象更有效；
- 引用传递比指针传递形式上更简单。



6.5 **const**对象与**const**成员函数

本节讨论**const**在类中的应用

👉 **const**数据成员

👉 **const**成员函数

👉 **const**对象

6.5.1 const数据成员

```
class Fred {  
    const int size;  
public:  
    Fred(int sz);  
    void print();  
};
```

Const数据成员

构造函数初始化列表：
常量数据成员必须被初始化

```
Fred::Fred(int sz) : size(sz) {}  
void Fred::print() { cout << size << endl; }
```

```
int main() {  
    Fred a(1), b(2), c(3);  
    a.print(), b.print(), c.print();  
} ///:~
```

1. 每个对象的const成员经初始化后都不能改变。
2. 这些const成员相互独立，可以有不同的值。

6.5.1.1 static const: 静态常量类成员

👉 **const**数据成员实际上是一个运行期间常量。

👋 问：如何获得编译期间整个类的恒定常量？

👋 答：方法一：static const (见后续章节)


方法二：枚举常量

```
class A
{...
    enum { SIZE1 = 100, SIZE2 =
200}; // 枚举常量
    int array1[SIZE1];
    int array2[SIZE2];
};
```

🕒 注意：枚举常量并不是类成员，也不会占用对象的存储空间，它们在编译时被全部求值。



6.5.2 **const**对象

 **const**对象：对象被初始化后，它的数据成员在其生命期内不被改变。

```
const int i=1;    // const int  
const blob b(2); //const对象
```




6.5.2.1 如何保证const对象不被改变？

☞ 公有数据：只要用户不去改变，这些数据保持不变是很容易实现的。

☞ 问题是：用户在调用成员函数时，也必须保证不改变数据。

✓ **对象如何知道哪些成员函数将会改变数据？**

✓ **对象如何知道哪些成员函数对于const对象来说是安全的？**

☞ 解决方法：

强制声明和定义const成员函数，显式地告诉编译器这些函数对数据是安全的，可以被const对象调用。

6.5.3 const成员函数

👉 在成员函数的声明和定义后面加上const使之成为const成员函数。

```
class X {  
    int i;  
public:  
    X(int ii);  
    int const_f() const;  
    int f();  
};
```

```
X::X(int ii) : i(ii) {}  
int X::const_f() const { return i; }  
int X::f(){ ... ; }
```

```
|  
int main() {  
    X x1(10);  
    const X x2(20);  
    x1.f(); // OK  
    x2.const_f(); // OK  
}
```

🕶 **const_f()是const函数，保证不修改x2;**



6.5.4 const对象与const成员函数

- ❏ 声明为const的对象是不能被赋值的
- ❏ 声明为const的对象不能随便调用任意的成员函数
`x2.f(); // error , f()非const 成员函数`
- ❏ 声明为const的对象只能调用声明为const的成员函数
`x2.const_f() ; //OK`
- ❏ const的成员函数不能改变成员变量



例子

```
class C{  
  public:  
    void m1(int x) const{  
      m2(x);  
    }  
  
    void m2(int x) {dm=x;}  
  
  private:  
    int dm;  
};
```

程序是否正确？



小结

- 👉 **const**能将对象、函数参数、返回值和成员函数定义为常量，还可以进行值替代。
- 👉 **const**为程序设计提供了又一种非常好的类型检查形式及安全性。
- 👉 **const**几乎成了程序正确性的“救命稻草”。



6.6 静态成员变量与静态成员函数



6.6.1 static关键字

```
int generate()  
{  
    static int i = 0;  
    return ++i;  
}
```

```
int main()  
{  
    std::cout << generate() << '\n';  
    std::cout << generate() << '\n';  
    std::cout << generate() << '\n';  
  
    return 0;  
}
```



6.6.2 静态成员变量

- 1. 类的所有对象共用一个静态成员变量**
- 2. 静态成员变量不是对象的一部分**



类的所有对象共用一个静态成员变量

```
class C{  
public:  
    static int i;  
};  
  
int C::i= 1;  
  
int main()  
{  
    C first;  
    C second;  
  
    first.i= 2;  
  
    std::cout << first.i <<endl;  
    std::cout << second.i <<endl;  
    return 0;  
}
```



类的所有对象共用一个静态成员变量

```
class C{  
public:  
    static int i;  
};
```

```
int C::i= 1;
```

```
int main()  
{  
    C first;  
    C second;  
  
    first.i= 2;  
  
    std::cout << first.i <<endl;  
    std::cout << second.i <<endl;  
    return 0;  
}
```

```
2  
2
```



静态成员变量不是对象的一部分

```
class C{  
public:  
    static int i;  
};
```

```
int C::i = 1;
```

```
int main()
```

```
{
```

注意：这里没有初始化任何对象

```
    C::i = 2;
```

```
    std::cout << C::i << endl;
```

```
    return 0;
```

```
}
```



6.6.3 静态成员函数

```
class C
{
private:
    static int i;
};
```

```
int C::i = 1; // Okay, Why?
```

```
int main()
{
    // 如何调用私有数据成员i?
}
```

静态成员函数用于调用静态数据成员

注意：静态成员函数没有this指针！



小结

数据成员类型	const	static	普通变量
直接初始化（在声明时就赋值）	✓	×	✓
先声明再通过初始化列表赋初值	✓	×	✓
先声明再在构造函数体里赋初值	×	×	✓
先声明再在类外赋初值	×	✓	×