

第五章 类:隐藏实现

面向对象程序设计（C++）

5.2 隐藏实现：例

**内部电路（实现）有不同；
但操作界面(接口)总是相同的。**



Ver 1.0



Ver 1.1

KV-HR32M90



5.2 隐藏实现：必要性

- 让库的使用者远离一些他们不需要知道的内部实现；
 - 允许库的设计者改变内部实现，而不必担心对客户程序员造成什么影响。

5.2 隐藏实现：原则

- 将类的功能描述部分作为共有部分接口提供给用户；
- 将数据的内部表示、功能的内部实现作为私有部分隐藏起来。



Public:

```
turn_on();  
turn_off();  
volume_up();  
volume_down();  
change_channel();
```

private:

```
change_red();  
change_blue();  
change_green();  
.....
```

boundary



5.2.1 C++的访问控制

Public / private /protected

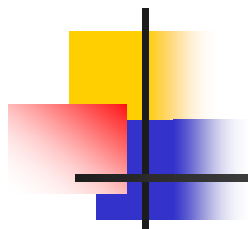
- **Public: 公有成员，其后声明的所有成员可以被所有的人访问。**
- **Private:私有成员，除了该类型的创建者和类的内部成员函数之外，任何人不能访问。**
- **Protected:保护成员，与Private基本相同，区别是继承的结构可以访问Protected成员，但不可以访问Private 成员。**

设置访问控制

```
struct B {  
    private:  
        char j;  
        float f;  
    public:  
        int i;  
        void func();  
};  
void B::func() {  
    B tempb;  
    tempb.j='a';// OK  
    i = 0;  
    j = '0';  
    f = 0.0;  
};
```

```
int main() {  
    B b;  
    b.i = 1;    // OK, public  
    //! b.j = '1'; // illegal,  
    //! b.f = 1.0; // illegal,  
    b.func(); // OK,public  
} ///:~
```

注：protected修饰符将在后续章节解释



OOP的观点

- 访问控制在库的设计者(实现者)和库的使用者之间划了一道明显的界线;

j和f 需要保护; i和func()可以公开

- 若试图访问一个私有成员, 就会产生一个编译时错误。



5.2.2 友元

- **友元：**显示地声明哪些人“可以访问我的私有实现部分”。表明的是一种“信任”的关系。

```
struct Cfather{  
    friend struct CMother; // 赋予CMother访问  
                           // Cfather的salary的权利  
private:  
    long salary;  
    ...;  
};  
  
struct CMother{  
private:  
    void inspect(){  
        Cfather obj_TaDie(8000);  
        cout<<“月收入:”<< obj_TaDie.salary;  
    };  
};
```




说明

- **友元不是类的成员，只是一个声明。**
- **友元可以是外部全局函数、类或类的成员函数；**
- **友元是一个“特权”函数，破坏了封装。当类的实现发生变化后，友元也要跟着变动。**

例：友元



```
// Friend allows special access
```

```
// Declaration
```

```
struct X; // 不完全的类型说明
```

```
struct Y {
```

```
    void f(X*);
```

```
};
```

```
struct X { // Definition
```

```
private:
```

```
    int i;
```

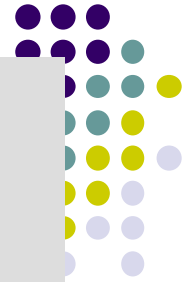
```
public:
```

```
    void initialize();
```

```
friend void g(X*, int); // 全局函数友元
```

```
friend struct Z; // 类友元
```

```
friend void Y::f(X*); // 成员函数友元
```



```
friend void h(); // 全局函数友元
};

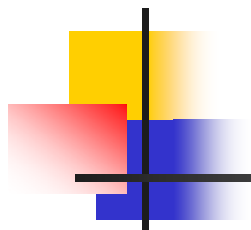
void X::initialize() {
    i = 0; // 类X的初始化函数
}

void g(X* x, int i) {
    x->i = i; // 访问X的私有成员i
}

void Y::f(X* x) {
    x->i = 47; // 访问X的私有成员i
}
```

```
struct Z {  
private:  
    int j;  
public:  
    void initialize();  
    void g(X* x);  
};  
  
void Z::initialize() {  
    j = 99;  
}  
  
void Z::g(X* x) {  
    x->i += j;  
}
```

```
void h() {  
    X x;  
    x.i = 100; //直接访问私有成员  
}  
  
int main() {  
    X x;  
    Z z;  
    z.g(&x);  
} ///:~
```

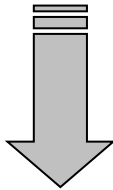


OOP的视点

- 友员不是OOP的特征，破坏了封装性，这也是C++不是“纯”面向对象语言的原因之一。

5.2.3 从struct到class

- 👉 封装：数据成员+ 成员函数 → 抽象数据类型
- 👉 访问控制：接口与实现的分离



C的struct + 封装 + 访问控制 = 类(Class)

思考：

**CStash和CStack中，哪些成员应该设置为私有的？
哪些成员应该设置为公有的？**

例1: 带访问控制的类Stash

```
class Stash {  
  
    private:  
        int size;    // Size of each space  
        int quantity; // Number of storage spaces  
        int next;    // Next empty space  
        unsigned char* storage;  
        void inflate(int increase);  
  
    public:  
        void initialize(int size);  
        void cleanup();  
        int add(void* element);  
        void* fetch(int index);  
        int count();  
};
```


例2:有访问控制的类 Stack

```
#ifndef STACK2_H
#define STACK2_H
class Stack {
    struct Link {      // 私有的
        void* data;
        Link* next;
        void initialize(void* dat, Link* nxt);
    }* head;
public:
    void initialize();
    void push(void* dat);
    void* peek();
    void* pop();
    void cleanup();
};
#endif // STACK2_H ///:~
```



5.2.3.1 Class和Struct的区别

- class的成员默认为Private;
- struct的成员默认为Public.

```
struct X{  
    int i;    // 公有成员  
    int y;    // 公有成员  
    void f(); // 公有成员  
}
```

```
class X{  
    int i;    // 私有成员  
    int y;    // 私有成员  
    void f(); // 私有成员  
}
```



5.2.4 访问控制的一个陷阱

- 返回private数据成员的非const引用：

当类的public成员函数f返回对该类private成员m的非const引用时，如果f的调用作为赋值语句的左值，则访问控制不起作用，对象的使用者可以直接存取m的值。

称这种情形为访问控制的“陷阱”。

例1

```
class X{
private :
    int i;
public:
    int & badseti(int j) {
        i=(j>0)?j:(-j);
        return i; // 返回私有成员i的引用
    }
};

void main(){
    X objx;
    objx.badseti(10);    // i的值为10;
    objx.badseti(-5) = -20 ; // 此时i的值为-20;
}
```

例2



```
class Time {  
public:  
    Time( int = 0, int = 0, int = 0 );  
    void setTime( int, int, int );  
    int getHour();  
    int &badSetHour( int ); // 危险的返回值  
private:  
    int hour;  
    int minute;  
    int second;  
};
```

// Fig. 6.11: time4.cpp

```
#include "time4.h"
#include <iostream.h>

Time::Time( int hr, int min, int sec ) {
    setTime( hr, min, sec );
}

void Time::setTime( int h, int m, int s ) {
    hour   = ( h >= 0 && h < 24 ) ? h : 0;
    minute = ( m >= 0 && m < 60 ) ? m : 0;
    second = ( s >= 0 && s < 60 ) ? s : 0;
}

// Get the hour value
int Time::getHour() { return hour; }
```

// 极其危险的做法:

// Returning a reference to a private data member.

```
int &Time::badSetHour( int hh )  
{  
    hour = ( hh >= 0 && hh < 24 ) ? hh : 0;  
    return hour; // DANGEROUS reference return  
}
```

// Fig. 6.11: fig06_11.cpp

```
#include <iostream.h>
```

```
#include "time4.h"
```

```
int main()
```

```
{
```

```
    Time t;
```

```
    int &hourRef = t.badSetHour( 20 );
```




```
cout << "Hour before modification: " << hourRef;  
hourRef = 30; // modification with invalid value  
cout << "\nHour after modification: " << t.getHour();
```

```
// Dangerous: Function call that returns  
// a reference can be used as an lvalue!
```

```
t.badSetHour(12) = 74;  
cout << "badSetHour as an lvalue, Hour: "  
      << t.getHour()  
      << "\n*****" << endl;
```

```
return 0;
```

```
}
```

```
cout << "Hour before modification: " << hourRef;
hourRef = 30; // modification with invalid value
cout << "\nHour after modification: " << t.getHour();

// Dangerous: Function call that returns
// a reference can be used as an lvalue!
t.badSetHour(12) = 74;
cout << "badSetHour as an lvalue, Hour: "
    << t.getHour()
    << "\n*****" << endl;

return 0;
}
```

20

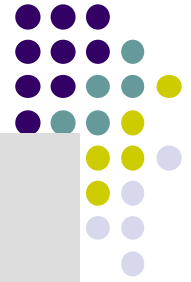
30

74

如何填补这个陷阱?

- 利用const!

例3:返回char *



```
#include <string.h>
#include <iostream.h>

class Ctest{
    char * str;
public:
    Ctest(int sz){
        str = new char[sz];
        strcpy(str,"ustc is a famouse uni.");
    }
    ~Ctest(){ delete [ ] str; }
    char const * getstr() const {
        return str;
    }
};
```



```
int main(int argc, char* argv[])
{
    Ctest  t1(100);
    strcpy(t1.getstr(),"TEST!!"); // ERROR
    cout << t1.getstr ();
    return 0;
}
```

编译时错误:

can not convert “const char *” to “char *”



5.2.6 句柄类

😬 类的实现者的烦恼:

- 尽管客户程序员(类的使用者)不能访问私有实现部分, 然而由于必须给他们提供.h文件, 他们可以轻易地“看到”类是如何实现的。

- 如何使得他们“看不见”实现?

- 如果改变了类的实现, 即使没有改变类的接口, 也可能需要重新编译所有使用了该类的客户程序。

- 如何尽可能地避免不必要的重复编译。

- 解决办法：

利用 “句柄类”。所谓句柄类，是一个特殊的类，该类中含有一个特殊的成员—— 一个指向被隐藏的类的指针；



接口：公开的部分(Handle.h)

```
// Handle classes
#ifndef HANDLE_H
#define HANDLE_H

class Handle {
    struct Cheshire;           // 类cheshire的声明 ;
    Cheshire* smile;          // smile 指针指向具体的实现
public:
    void initialize();
    void cleanup();
    int read();
    void change(int);
};
#endif // HANDLE_H ///
```

编译器从以上定义完全知道如何安排类Handle的存储，因此Handle无需知道Cheshire的具体实现，从而达到隐藏实现的目的。

实现：需要隐藏的部分(Handle.cpp)



```
// Handle implementation
#include "Handle.h"
#include "../require.h"

// 句柄类的实现: cheshire
struct Handle::Cheshire {
    int i;
};

void Handle::initialize() {
    smile = new Cheshire;
    smile->i = 0;
}
```

```
void Handle::cleanup() {
    delete smile;
}

int Handle::read() {
    return smile->i;
}

void Handle::change(int x) {
    smile->i = x;
} ///:~
```




优点

- 由于实现部分并没有暴露在.h中，保密性好；
- 因为实现的细节均隐藏在句柄类的背后，因此当实现发生变化时，句柄类并没有改变，也就不必重新编译客户程序，只需重新编译隐藏的实现这一小部分代码，然后连接(link)就行了。减少了大量的重复编译。



小结

- **访问控制: public / private / protected**
- **友元: friend**
- **访问控制的陷阱: 返回private数据成员的非const引用**
- **句柄类: 隐藏实现源代码, 两方面的优点**