chapter14 c++中的代码重用

1）包含成员对象的类

【1】引入：当某些类包含了其他类的成员对象时，应当如何调用?

【2】简介：模板特性声明对象时必须加上特定的具体数据类型

```
valarray<int> q_values;
valarray<double> weights;
```

【3】设计方式：用于建立 has-a 关系的技术是组合(包含)，即：创建一个包含其他类对象的类

```
class Student
{
private:
        string name;
        valarrat<double> scores;
        ...
};
```

上述类将数据成员声明为私有的，这意味着Student类的成员函数可以使用 string 和 valarray< double > 类的公有接口（这两个类中的public内容）来访问和修改name、scores对象，但在类的外面不可以!

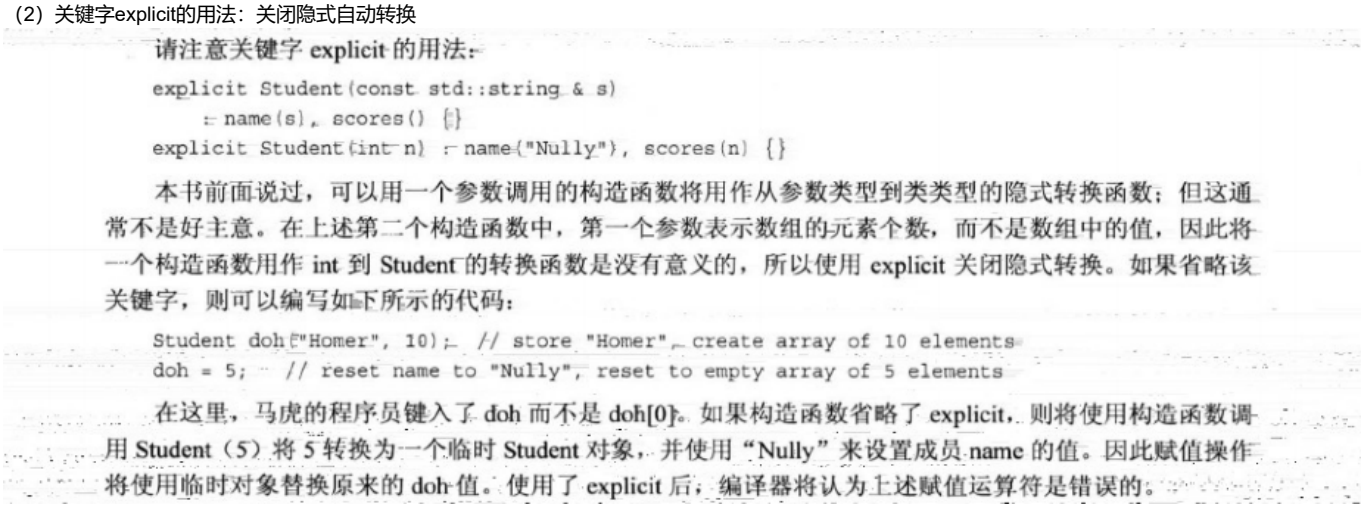
【4】示例解析：

```
// studentc.h -- defining a Student class using containment
#ifndef STUDENTC_H_
#define STUDENTC_H_

#include <iostream>
#include <string>
#include <valarray>
class Student
{
private:
    typedef std::valarray<double> ArrayDb;
    std::string name;       // contained object
    ArrayDb scores;         // contained object
    // private method for scores output
    std::ostream & arr_out(std::ostream & os) const;

public:
    Student() : name("Null Student"), scores() {}
    explicit Student(const std::string & s)
        : name(s), scores() {}
    explicit Student(int n) : name("Nully"), scores(n) {}
    Student(const std::string & s, int n)
        : name(s), scores(n) {}
    Student(const std::string & s, const ArrayDb & a)
        : name(s), scores(a) {}
    Student(const char * str, const double * pd, int n)
        : name(str), scores(pd, n) {}
    ~Student() {}
    double Average() const;
    const std::string & Name() const;
    double & operator[](int i);
    double operator[](int i) const;
// friends
    // input
    friend std::istream & operator>>(std::istream & is,
                                     Student & stu);  // 1 word
```

```
    friend std::istream & getline(std::istream & is,
                                  Student & stu);     // 1 line
    // output
    friend std::ostream & operator<<(std::ostream & os,
                                     const Student & stu);
};
#endif
```

（1）typedef std::valarray< double > ArrayDb; [1]typedef重命名；[2]放在类声明的私有部分：可以在Student类的实现中使用它，但在类外不能！

（2）关键字explicit的用法：关闭隐式自动转换



请注意关键字 explicit 的用法：

```
explicit Student(const std::string & s)
    : name(s), scores() {}
explicit Student(int n) : name("Nully"), scores(n) {}
```

本书前面说过，可以用一个参数调用的构造函数将用作从参数类型到类类型的隐式转换函数；但这通常不是好主意。在上述第二个构造函数中，第一个参数表示数组的元素个数，而不是数组中的值，因此将一个构造函数用作 int 到 Student 的转换函数是没有意义的，所以使用 explicit 关闭隐式转换。如果省略该关键字，则可以编写如下所示的代码：

```
Student doh("Homer", 10);  // store "Homer", create array of 10 elements
doh = 5;  // reset name to "Nully", reset to empty array of 5 elements
```

在这里，马虎的程序员键入了 doh 而不是 doh[0]。如果构造函数省略了 explicit，则将使用构造函数调用 Student（5）将 5 转换为一个临时 Student 对象，并使用“Nully”来设置成员 name 的值。因此赋值操作将使用临时对象替换原来的 doh 值。使用了 explicit 后，编译器将认为上述赋值运算符是错误的。

（3）c++及约束：
【1】初始化列表的功能（要多使用！）

> [1] 初始化内置类型的成员

```
Queue :: Queue(int qs) : qsize(qs) {...}
```

> [2] 初始化派生类对象的基类部分

```
hasDMA :: hasDMA(const hasDMA& hs) : baseDMA(hs) {...}
```

ps：对于派生类而言
对于**要继承的对象**，构造函数在成员初始化列表中使用**类名**来调用特定的基类构造函数；
对于**成员对象**，构造函数则使用**成员名**

```
Student(const char* str, const double* pd, int n) : name(str),scores(pd,n) {}
```

【2】初始化顺序

> 当初始化列表包含多个项目时，这些项目被初始化的顺序为它们被声明的顺序，而不是它们在初始化列表中的顺序

```
class Student
{
private:
    typedef std::valarray<double> ArrayDb;
    std::string name;       // contained object
    ArrayDb scores;         // contained object
    // private method for scores output
    std::ostream & arr_out(std::ostream & os) const;
public:
        Student(const char* str, const double* pd, int n) : scores(pd,n),name(str) {}
```

```
        ...
}

此时：name成员仍将首先被初始化，因为它在类定义中首先被声明！
```

【3】使用被包含对象的接口

> 被包含对象的接口不是共有的，但是可以在类方法中使用它
> eg1：

```
double Student::Average() const
{
        if( scores.size() > 0 ) return (scores.sum()/scores.size() );
        else return 0;
}
// scores是Student类的成员，scores还是一个valarray对象，所以它可以调用valarray类的成员函数scores.size() & scores.sum()
```

> eg2：

```
ostream& operator<<(ostream& os,const Student& stu)
{
        os << "Scores for " << stu.name <<": \n";
        ......
}

(1) stu.name是一个string对象，所以它将调用 operator<<(ostream&,const string&) ,而这已经在string类中了；
(2) operator<<(ostream& os,const Student& stu)    应当声明为Student类的友元函数，这样才能访问Student类中的name成员。
```

2）私有继承

子对象 (subobject)：通过继承or包含添加的对象
多重继承(muitiple inheritance , MI)：使用多个基类的继承
隐式向上转换(implicit upcasting)：无需进行显示类型转换，就可以将 基类指针 or 引用 指向派生类对象

3）多重继承

4）类模板

（1）引入：

c++的类模板为生成通用的类声明提供了一种更好的方法，模板提供参数化(parameterized)类型，即：能够将类型名作为参数传递给接收方来建立类或函数

（2）定义类模板：

【1】开头：

```
template <class Type>
```

【2】模板的具体实现被称为：实例化 (instantiation) 或 具体化 (specialization)
example:

```
// stacktp.h -- a stack template
#ifndef STACKTP_H_
#define STACKTP_H_
template <class Type>
class Stack
```

```
{
private:
    enum {MAX = 10};    // constant specific to class
    Type items[MAX];    // holds stack items
    int top;            // index for top stack item
public:
    Stack();
    bool isempty();
    bool isfull();
    bool push(const Type & item); // add item to stack
    bool pop(Type & item);        // pop top into item
};

template <class Type>
Stack<Type>::Stack()
{
    top = 0
}

template <class Type>
bool Stack<Type>::isempty()
{
    return top == 0;
}

template <class Type>
bool Stack<Type>::isfull()
{
    return top == MAX;
}

template <class Type>
bool Stack<Type>::push(const Type & item)
{
    if (top < MAX)
    {
        items[top++] = item;
        return true;
    }
    else
        return false;
}

template <class Type>
bool Stack<Type>::pop(Type & item)
{
    if (top > 0)
    {
        item = items[--top];
        return true;
    }
    else
        return false;
}
#endif
```

（3）使用模板类

> [1] 必须请求实例化，方法是使用所需要的**具体类型**替换**泛型名**

```
Stack<int> ker;    // a stack of ints
Stack<string> col; // a stack of string objects
```

> [2] 泛型标识符——称为类型参数(type parameter)。比如上面的Type。它们类似于变量，但赋给它们的只能是类型，必须显式提供类型！

```
template <class T>
void simple(T t) {cout << t <<endl;}
...
simple(2);      // generate void simple(int)
simple("two");  // generate void simple(const char*)
```

ps: 其余使用细则函数模板基本没区别！ 结合上面的实例自行品味！

（4）数组模板实例and非类型参数

示例：

```
//arraytp.h  -- Array Template
#ifndef ARRAYTP_H_
#define ARRAYTP_H_

#include <iostream>
#include <cstdlib>
template <class T, int n>
class ArrayTP
{
private:
    T ar[n];
public:
    ArrayTP() {};
    explicit ArrayTP(const T & v);
    virtual T & operator[](int i);
    virtual T operator[](int i) const;
};

template <class T, int n>
ArrayTP<T,n>::ArrayTP(const T & v)
{
    for (int i = 0; i < n; i++)
        ar[i] = v;
}

template <class T, int n>
T & ArrayTP<T,n>::operator[](int i)
{
    if (i < 0 || i >= n)
    {
        std::cerr << "Error in array limits: " << i
            << " is out of range\n";
        std::exit(EXIT_FAILURE);
    }
    return ar[i];
}

template <class T, int n>
T ArrayTP<T,n>::operator[](int i) const
{
    if (i < 0 || i >= n)
    {
        std::cerr << "Error in array limits: " << i
            << " is out of range\n";
        std::exit(EXIT_FAILURE);
    }
    return ar[i];
}
#endif
```

说明：
（1）模板头：

```
template < class T, int n >
```

模板声明关键字：template
关键字：class
类型参数：T
非类型(non-type) 或 表达式(expression)参数：int /...

（2）表达式参数的限制：

【1】必须是：整型、枚举、引用、指针
如：double m；不合法！
如：double* rm；合法！

【2】模板代码不能修改参数的值 or 地址
如：不能出现 n++ 或 &n 等表达式

【3】实例化模板时，用作表达式参数的值必须是常量表达式

（5）模板多功能性

【1】可以递归使用模板：

```
ArrayTP< ArrayTP<int,5> , 10 > twodee;

生成二维数组：包含10个元素的数组，每个元素都是一个包含5个int元素的数组
```

【2】使用多个类型参数：

```
希望可以保存两种值（键值对），创建并使用Pair模板：
// pairs.cpp -- defining and using a Pair template
#include <iostream>
#include <string>
template <class T1, class T2>
class Pair
{
private:
    T1 a;
    T2 b;
public:
    T1 & first();
    T2 & second();
    T1 first() const { return a; }
    T2 second() const { return b; }
    Pair(const T1 & aval, const T2 & bval) : a(aval), b(bval) { }
    Pair() {}
};

template<class T1, class T2>
T1 & Pair<T1,T2>::first()
{
    return a;
}

template<class T1, class T2>
T2 & Pair<T1,T2>::second()
{
    return b;
}

int main()
{
    using std::cout;
    using std::endl;
    using std::string;

    Pair<string, int> ratings[4] =
    {
        Pair<string, int>("The Purpled Duck", 5),
        Pair<string, int>("Jaquie's Frisco Al Fresco", 4),
        Pair<string, int>("Cafe Souffle", 5),
        Pair<string, int>("Bertie's Eats", 3)
    };

    int joints = sizeof(ratings) / sizeof (Pair<string, int>);
    cout << "Rating:\t Eatery\n";
    for (int i = 0; i < joints; i++)
```

```
        cout << ratings[i].second() << ":\t "
             << ratings[i].first() << endl;
    cout << "Oops! Revised rating:\n";

    ratings[3].first() = "Bertie's Fab Eats";
    ratings[3].second() = 6;
    cout << ratings[3].second() << ":\t "
         << ratings[3].first() << endl;
   // std::cin.get();
   return 0;
}
```

注意这里的声明：

```
template <class T1, class T2>
```

（6）模板的具体化

【1】隐式实例化 (implicit instantiation)：

```
ArrayTP<int,100> stuff;
```

【2】显式实例化 (explicit instantiation)：

```
template class ArrayTP<string,100>;
将 ArrayTP<string,100> 声明为一个类
```

【3】显式具体化 (explicit specialization)：

```
引入例子：
template <typename T>
class SortedArray
{
        ...
};
-----------------------------------------------------------------------------------------------------------------------------
可以提供一个显式模板具体化，这将采用为具体类型定义的模板，而不是为泛型定义的模板
当具体化模板和通用模板都与实例化请求匹配时，编译器将使用具体化版本

具体化类模板定义的格式：
template <> class Classname<specialized-type-name> {...};

例如：要提供一个专供 const char* 类型使用的SortedArray模板
template <> class SortedArray<const char*>
{
        ...
};
```

（7）模板类和友元

【1】引入分类：

> 非模板友元；
> 约束 (bound) 模板友元；
> 非约束 (unbound) 模板友元；

【2】模板类的非模板友元函数：

> （1）一旦声明，使得该函数成为模板所有实例化的友元：

```
template <class T>
class HasFriend
{
public:
        friend void counts(); // friend to all HasFriend instantiation
    ...
}

例如：它将是类HasFriend<int> 和 HasFriend<string>
```

> （2）要为友元函数提供模板类参数，则必须指明具体化

```
template <class T>
class HasFriend
{
public:
        friend void report(HasFriend<T>& );  // valid!    HasFriend<T> 是实例
        friend void report(HasFriend& );     // invalid!  HasFriend 是抽象，非具体!

假设这里T是int：带HasFriend<int>参数的report()将成为HasFriend<int> 类的友元
假设这里T是double：带HasFriend<double>参数的report()将成为HasFriend<double> 类的友元
}
```

> （3）要使用的友元全部都要定义显式具体化

```
比如上例的后续调用需要使用：HasFriend<int> 与 HasFriend<double>
则需要类外定义：

void report(HasFriend<int>& ) {...};
与
void report(HasFriend<double>& ) {...};
```

【3】模板类的约束（bound）模板友元函数：

aim：使得友元函数本身成为模板

> [1] 首先在类定义的前面声明每个模板函数

```
template <typename T> void counts();
template <typename T> void report(T& );
```

> [2] 其次在函数中将模板声明为友元（声明具体化）

```
template <typename TT>
class HasFriend2
{
        ...
        friend void counts<TT>();
        friend void report<>(HasFriend2<TT>& );
};

声明中的<>指出这是模板具体化：
对于 report() ，<>可以为空，因为可以从函数参数推断出模板类型参数：HasFriend2<TT>，自动明白是report< HasFriend2<TT> >(HasFriend2<TT>& );
对于 counts() ，由于该函数没有参数，所以必须使用TT来指明具体化
```

> [3] 为友元提供模板定义

程序说明：

```
// tmp2tmp.cpp -- template friends to a template class
#include <iostream>
using std::cout;
using std::endl;

// template prototypes
template <typename T> void counts();
template <typename T> void report(T &);

// template class
template <typename TT>
class HasFriendT
{
private:
    TT item;
    static int ct;
public:
    HasFriendT(const TT & i) : item(i) {ct++;}
    ~HasFriendT() { ct--; }
    friend void counts<TT>();
    friend void report<>(HasFriendT<TT> &);
};

template <typename T>
int HasFriendT<T>::ct = 0;

// template friend functions definitions
template <typename T>
void counts()
{
    cout << "template size: " << sizeof(HasFriendT<T>) << "; ";
    cout << "template counts(): " << HasFriendT<T>::ct << endl;
}

template <typename T>
void report(T & hf)
{
    cout << hf.item << endl;
}

int main()
{
    counts<int>();
    HasFriendT<int> hfi1(10);
    HasFriendT<int> hfi2(20);
    HasFriendT<double> hfdb(10.5);

    report(hfi1);  // generate report(HasFriendT<int> &)
    report(hfi2);  // generate report(HasFriendT<int> &)
    report(hfdb);  // generate report(HasFriendT<double> &)

    cout << "counts<int>() output:\n";
    counts<int>();
    cout << "counts<double>() output:\n";
    counts<double>();
    // std::cin.get();
    return 0;
}
```

【4】模板类的非约束（unbound）模板友元函数：

aim：通过在类内部声明模板，可以创建非约束友元函数，即：每个函数具体化都是每个类具体化的友元

1）格式:

```
template <typename T>
class ManyFriend
{
...
        template<typename A , typename B> friend void Func(A& , B&);
};
```

2）例子与如下具体化匹配：

```
void show2<ManyFriend<double>& , ManyFriend<int>&>(ManyFriend<double>& hbx1, ManyFriend<int>& hbx2);
```

3）示例：

```
// manyfrnd.cpp -- unbound template friend to a template class
#include <iostream>
using std::cout;
using std::endl;

template <typename T>
class ManyFriend
{
private:
    T item;
public:
    ManyFriend(const T & i) : item(i) {}
    template <typename C, typename D> friend void show2(C &, D &);
};

template <typename C, typename D> void show2(C & c, D & d)
{
    cout << c.item << ", " << d.item << endl;
}

int main()
{
    ManyFriend<int> hfi1(10);
    ManyFriend<int> hfi2(20);
    ManyFriend<double> hfdb(10.5);

    cout << "hfi1, hfi2: ";
    show2(hfi1, hfi2);

    cout << "hfdb, hfi2: ";
    show2(hfdb, hfi2);
    // std::cin.get();
    return 0;
}
```

（8）模板别名

【1】使用 typedef：

> 格式：typedef *To_be_trans_name* **Re_name** ;

例子：

```
typedef std::array<double,12> arrd;
typedef std::array<int,12> arri;
typedef std::array<std::string,12> arrst;

arrd gallons;  // gallons is type std::array<double,12>
arri days;     // ...
arrst name;    // ...
```

【2】使用 template：略

【3】使用 using：略