14 C++中的代码重用

代码重用

- 代码重用
  - 继承
  - 用已经存在的类来构建数据成员
    - 数据成员包含了另一个类的对象。即，包含(containment)、组合(composition) 或层次化(layering)
  - 类模板
    - 定义类模板，然后使用模板来创建针对特定类型定义的特殊类

包含对象成员的类

1 包含对象成员的类

- 新的类将包含另一个类的对象
  - 称为包含(containment)、组合(composition) 或层次化(layering)

- 将学生类简化成姓名和一组考试分数后，可使用一个包含两个成员的类来表示它：
  - 姓名
    - 使用一个开发好的类的对象来表示。例如，可以使用一个string 类
  - 分数
    - 可以设计一个使用动态内存分配的类来表示该数组；也可在标准C++库中找一个能够表示这种数据的类

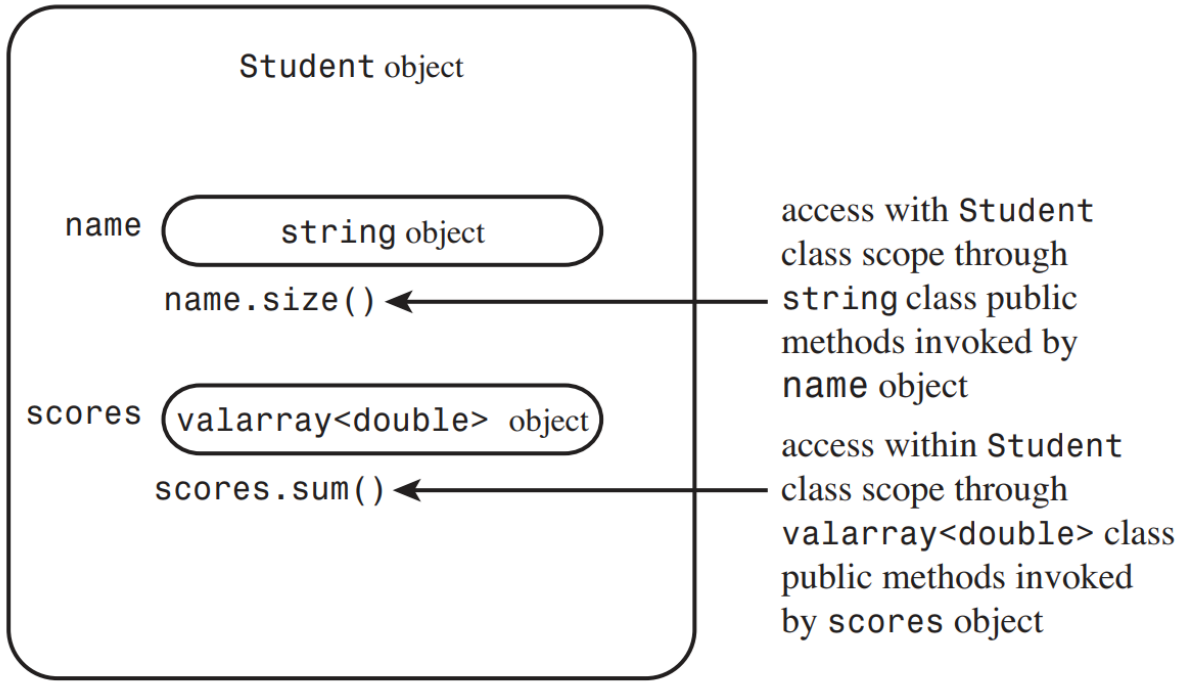
1.1 valarray类简介

- 一个模板类，类似vector
  - 用于处理数值（或具有类似特性的类）
    - 支持数组中数值操作。如，将数组中所有元素的值相加、在数组中找出最大和最小的值
- 定义了一组在两个相同长度和相同类型的valarray类对象之间的数字计算
  - 例如，xarr = cos(yarr) + sin(zarr);

```
double gpa[5] = {3.1, 3.5, 3.8, 2.9, 3.3};
valarray<double> v1;          // an array of double, size 0
valarray<int> v2(8);          // an array of 8 int elements
valarray<int> v3(10, 8);      // an array of 8 int elements, each set to 10
valarray<double> v4(gpa, 4); // an array of 4 elements, initialized to the first 4 elements
```

1.2 Student类的设计

- 使用一个string对象来表示姓名，使用一个valarray<double>来表示考试分数
- 从这两个类派生出Student类，多重公有继承
  - 不合适。因为学生与这些类之间不是is-a关系
  - 是has-a，学生有姓名，也有一组考试分数
- 对于has-a关系，类对象不能自动获得被包含对象的接口
  - string类将+运算符重载为将两个字符串连接起来；但从概念上说，将两个Student对象串接起来没有意义
- 另一方面，被包含的类，其部分接口对新类来说可能有意义
  - 如，string接口中的operator<()方法，可用于将Student对象按姓名进行排序




```
class Student
{
private
    string name;
    valarray<double> scores;
    …
};
```

1.3 Student类示例

P14.1 studentc.h P14.2 studentc.cpp P14.3 use_stuc.cpp

- typedef

```
typedef std::valarray<double> ArrayDb;
```

- explicit

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

- 初始化顺序
  - 声明的顺序，非初始化列表顺序
- 被包含对象的接口不是公有的，但可以在类方法中使用它

私有继承

2 私有继承(非必要，不使用)

多重继承

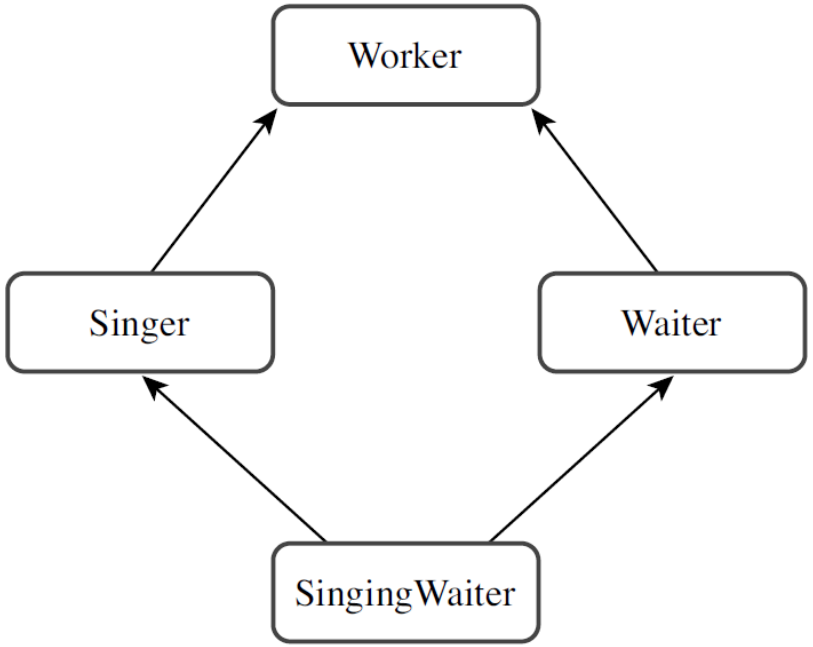
3 多重继承

P14.7 Worker0.h 14.8 worker0.cpp
14.9 worktest.cpp;

- MI 描述的是有多个直接基类的类

```
class SingleWaiter : public Waiter, public
Singer
```

- 关于MI的最大麻烦
  - 从两个不同的基类继承，对于同名方法的处理
  - 从多个相关基类继承的同一个类的多个实例



3.1 有多少worker

- 问题

虚基类

- 虚基类是的从多个具有共同基类的类派生出来的对象只有一个基类对象

  通过在类声明中使用关键字virtual，可以使Worker 被用作Singer 和Waiter 的虚基类(virtual 和public 的次序无关紧要）：

```
class Singer : virtual public Worker{//...};
class Waiter : virtual public Worker{//...};
```

- 然后，可以将SingingWaiter类定义为：

```
class SingingWaiter : public Singer, public Waiter {//...};
```

- 现在， SingingWaiter 对象将只包含Worker 对象的一个副本

3.2 哪个方法

- 二义性
  - newhire.Singer::Show();
  - void SingleWaiter::Show(){Singer::Show();}

3.3 MI小结

类模板

4 类模板

- 容器类(container class)，用于存储其他对象或数据的类
  - Stack类，Queue类
- typedef方式可用于支持不同类型的容器，但存在缺点
  - 要重新编译
  - 不能同时定义多种类型的容器
- 模板提供参数化(parameterized)类型，即，将类型名作为参数传递给接收方来建立不同类或函数
- C++库提供了多个模板类

4.1 定义类模板

- template <class Type>
  - Type是通用的类型说明符，在使用模板时，用实际的类型替换它
  - 较新的C++实现允许typename代替class

```
template <class Type>
Stack<Type>::Stack() {}
Stack<int> kernels;
// create a stack of ints
Stack<string> colonels; // create a stack
```

```
1.  template <class Type>
2.  class Stack{
3.  private:
4.      enum{MAX=10};     //  constant specific to class
5.      Type items[MAX]; //  holds stack items
6.      int top;          //  index for top stack item
7.  public:
8.      Stack();
9.      bool isempty();
10.     bool isfull();
11.     bool push(const Type &item); // add item to stack
12.     bool pop(Type & item); // pop top into item
13. };

14. template <class Type>
15. Stack<Type>::Stack(){
16.     top = 0;
17. }
```

4.2 使用模板类

- Stack<int> kernels;
- 编译器将按Stack<Type>模板来生成独立的类声明和独立的类方法
- 泛型标识符Type-类型参数(type parameter)，这意味着它们类似于变量，但赋给它们的不能是数字，而是类型
- 必须显式地提供所需的类型，与常规的函数模板不同

```
template <class T>
void simple(T t) { cout << t << '\n'; }

simple(2);     // generate void simple(int}
simple("two"); // generate void simple(const char*)
```

```
1.  int main(){
2.      Stack<std::string> st;//create an empty stack
3.      char ch;
4.      std::string po;
5.      while (cin >> ch && std::toupper(ch) != 'Q'){
6.          //...
7.          switch (ch){
8.          case 'A':
9.          case 'a':
10.             cout << "Enter a PO number to add: ";
11.             cin >> po;
12.             if (st.isfull())
13.                 cout << "stack already full\n";
14.             else
15.                 st.push(po);
16.             break;
17.         }
18.     }
19. }
```

4.3 深入探讨模板类

P14.15 stcktp1.h　　P14.16 stkoptr1.cpp

- 要非常小心使用模板类
  - 不正确地使用指针栈

```
Stack<char *> st;  // create a stack for pointers-to-char
```

  - 正确使用指针栈
    - 需要明确申请内存和释放内存的位置，应非常小心

4.4 数组模板示例和非类型参数

14.17 arraytp.h

- 非类型参数，表达式参数

  template <class T, int n>//n用来做数组长度

  - 每种数组大小都生成自己的模板

```
1.  template <class T, int n>
2.  class ArrayTP{
3.  private:
4.      T ar[n];
5.  public:
6.      ArrayTP(){};
7.      explicit ArrayTP(const T &v);
8.      virtual T &operator[](int i);
9.      virtual T operator[](int i) const;
10. };

11. template <class T, int n>
12. ArrayTP<T, n>::ArrayTP(const T &v){
13.     for (int i = 0; i < n; i++)
14.         ar[i] = v;
15. }
```

4.5 模板多功能性

- 模板类可以用作基类，也可以用作组件类
- 可以递归使用
  - 矩阵(P14.18 twod.cpp)
- 可以包含多个类型参数
  - (P14.19 paris.cpp)
- 默认类型模板参数
  - 为类型参数提供默认值

```
1.  int main(void){
2.      ArrayTP<int, 10> sums;
3.      ArrayTP<double, 10> aves;
4.      ArrayTP<ArrayTP<int,5>, 10> twodee;
5.  }
6.  template <class T1, class T2> class Pair{
7.  private:
8.      T1 a;
9.      T2 b;
10. public:
11.     T1 & first();
12.     T2 & second();
13.     T1 first() const { return a; }
14.     Pair(const T1 & av, const T2 & bv) : a(av), b(bv)
15.     { }
16.     Pair() {}
17. };
18. template<class T1, class T2>T1 & Pair<T1,T2>::first()
19. {return a;}
```

4.6 模板的具体化

- 模板以泛型的方式描述类，而具体化是使用具体的类型生成类声明
- 隐式实例化
  - 编译器使用的时候具体化

```
ArrayTP<int, 100> stuff;//implicit instantiation
```

  - 编译器在需要对象之前，不会生成类的隐式实例化
- 显式实例化

```
template class ArryTP<string, 100>;
```

- 显式具体化
  - PASS

```
1. ArrayTP<int, 100> stuff; // implicit instantiation
2. ArrayTP<double, 30> *pt; // a pointer,
3. pt = new ArrayTP<double, 30>;
```

4.7 成员模板

P14.20 tempmemb.cpp

- 模板可用作结构、类或模板类的成员
  - 模板类里套模板类

4.8 将模板用作参数

- PASS

4.9 模板类和友元

- 模板类声明也可以有友元
  - 模板类的非模板友元函数(P14.22 frnd2tmp.cpp)
    - 在模板类中将一个常规函数声明为友元：
  - 模板类的约束模板友元函数(P14.23 tmp2tmp.cpp)
    - 类的每一个具体化都获得与友元匹配的具体化
  - 模板类的非约束模板友元函数(P14.24 manyfrnd.cpp)
    - 每个函数具体化都是每个类具体化的友元。
    - 非约束友元，友元模板类型参数与模板类类型参数不同

```
1.  template <typename T> class ManyFriend {
2.  public:
3.      template <typename C, typename D>
4.      friend void show2(C &, D &);
5.  };
6.  template <typename C, typename D>
7.  void show2(C &c, D &d) {}
```

```
1.  template <typename T> void counts();
2.  template <typename T> void report(T &);
3.  template <typename TT> class HasFriendT{
4.  private:
5.      TT item;
6.      static int ct;
7.  public:
8.      HasFriendT(const TT &i) : item(i) { ct++; }
9.      ~HasFriendT() { ct--; }
10.     friend void counts<TT>();
11.     friend void report<>(HasFriendT<TT> &);
12. };
13. template <typename T> void counts(){
14.     cout << “T size: " << sizeof(HasFriendT<T>);
15.     cout << “T counts(): " << HasFriendT<T>::ct;
16. }
17. template <typename T> void report(T &hf) {
18.     cout << hf.item << endl; }
```

4.10 模板别名

- PASS
- 可使用typedef 为模板具体化指定别名

5 总结

- C++提供了几种重用代码的手段
- 公有继承能够建立is-a 关系，这样派生类可以重用基类的代码
- 还可以通过开发包含对象成员的类来重用类代码。这种方法被称为包含、层次化或组合，它建立的是has-a 关系
- 多重继承(MI)使得能够在类设计中重用多个类的代码
- 使用虚基类来避免继承多个基类对象的问题
- 类模板使得能够创建通用的类设计