

继承和多态性

继承的意思是产生子类。简单地说，子类会具有父类的成员变量和成员函数。

一个程序接口指定一个函数的集合；如果一个类实现一个程序接口，那么这个类中必须具有这个程序接口指定的所有函数。多态性是指具有同一个程序接口的多种数据类型可以通过这个接口做相同的处理。

在接下来的三小节里，我们先介绍 C++ 中支持继承和多态性的语法结构。然后我们进一步讨论它们的使用技巧和误区。

1. 继承

我们在定义一个类的时候可以指定它的父类，那么这个类就是一个子类，它会继承父类的成员变量和成员函数。请看下面的例子。

```
1 class A
2 {
3 public:
4     int value;
5
6     int getValue() {
7         return value;
8     }
9 };
10
11 int main() {
12     A object;
```

```
39 class A
40 {
41 private:
42     int value = 100;
43     int value;
44
45 public:
46     A() {
47         value = 100;
48     }
49
50 protected:
```

```

13     object.value = 100;
14     cout << object.
15         getValue();
16 }
17
18 class B : public A
19 {
20 public:
21     int value2;
22
23     int getSum() {
24         return
25             value + value2;
26     }
27 };
28
29 int main() {
30     B object;
31     object.value = 100;
32     object.value2 = 10;
33     cout <<
34         object.getValue();
35     cout <<
36         object.getSum();
37 }

51     int getValue() {
52         return value;
53     }
54 };
55
56 int main() {
57     A object;
58     object.value = 100;
59     cout << object.getValue();
60 }
61
62 class B : public A
63 {
64 public:
65     int value2;
66
67     int getSum() {
68         return value + value2;
69         return
70             getValue() + value2;
71     }
72 };
73
74
75 int main() {
76     B object;
77     object.value = 100;
78     object.value2 = 10;
79     cout << object.getValue();
80     cout << object.getSum();
81 }

```

第 1-9 行定义了一个普通的类 A，它有一个公有的成员变量和一个公有的成员函数。在 18-27 行定义了一个类 A 的子类 B。如第 18 行所示，定义子类的语法格式是：

```
class 子类名 : public 父类名 { ... };
```

父类前面的 `public` 表示继承的方式是公有继承，这是继承的一般用法。我们暂时不讨论什么是公有继承，以及另外

的两种继承方式（它们极少被使用，基本上是没用的）。

在第 18-27 行的类 B 中，我们定义了一个公有变量和一个公有函数。因为 B 继承了 A，所以 B 的每个对象一共具有两个成员变量和两个程序函数。在第 29-37 行中，我们可以看到类 B 的一个对象 `object` 具有成员变量 `value` 和 `value2`，也通过 `object` 调用成员函数 `getValue` 和 `getSum`。

下面我们通过第 39-81 行的程序讨论封装在继承的作用。在 39-54 行的类 A 中我们，成员变量 `value` 变成了私有的，而成员函数 `getValue` 变成了保护 `protected` 的。`protected` 是除了 `public` 和 `private` 之外的另一种成员访问层次 `access level`。如果一个成员的访问层次是 `protected` 的，那么它能在当前的类和当前类的子类的函数中被访问，而不能在其它函数中被访问。

注意到 42 行对成员变量的初始化方式是标准 C++ 不允许的。这是因为，这里并不是定义变量，只是一个声明：声明所有该类的对象都具有 `value` 这个成员。如果我们需要初始化成员变量，可以在构造函数里完成，如第 46-48 行所示。

现在，我们看看当类 A 作了上述的改动后，它的子类 B 应该如何改动。第 68 行不能通过编译，原因是 `value` 在类 A 中是 `private` 的：它只能在类 A 的函数中被访问。但是，我们

可以通过函数 `getValue` 间接地取得 `value` 的值。这是因为 `getValue` 在类 A 中是 `protected` 的，所以它可以在类 A 的子类 B 中被访问。这里我们总结一下 `private` 和 `protected`:

- 父类 A 中的 `private` 成员变量，实际上是存在于子类 B 的每一个对象的。但是，这个对象中的成员虽然存在，却不能在子类 B 里面所定义的函数中被访问，而只能通过类 A 所提供的 `public` 和 `protected` 函数间接地访问。
- 父类 A 中的 `private` 成员函数，只能被父类里定义的有关函数直接调用。
- 父类 A 中的 `protected` 成员函数和成员变量，被类 A 的子类 B 里定义的函数中视为 `public` 的，而在其它的函数中被视为 `private` 的。

最后，介绍一些同义词。子类 sub-class 又称为派生类 derived class，父类 super-class 又称为基类 base-class。

2. 虚对象函数和函数匹配的两种方式

```
3 class A
4 {
5 public:
6     void print1() {
7         cout << "A::print1()" << endl;
8     }
9
10    virtual void print2() {
11        cout << "A::print2()" << endl;
```

```

12     }
13
14     void print3() {
15         cout << "A::print3()" << endl;
16     }
17
18     virtual void print4() {
19         cout << "A::print4()" << endl;
20     }
21 };

```

这下面的两个小节我们将介绍较多的概念。首先我们通过下面的例子介绍虚对象函数 virtual object function的概念。

在类 A 里面，我们定义了两个函数，这两个函数分别向控制台 console 输出它们的函数全名，这让我们知道程序运行的时候有没有调用这个函数。其中，第 10-12 行和第 18-20 行的函数被声明为一个虚对象函数。如果类 A 没有子类，那么虚对象函数的作用不会显现。现在我们定义子类 B。

```

23 class B : public A
24 {
25 public:
26     // not related to A::print1()
27     void print1() {
28         cout << "B::print1()" << endl;
29     }
30
31     // overrides A::print2()
32     void print2() {
33         cout << "B::print2()" << endl;
34     }
35 };

```

在子类 B 中，我们“重新”定义了类 A 中的两个函数。首先看看 27-29 行的函数，它其实跟 A 中的 print1 函数没有

什么关系：它们的函数全名分别是 `A::print1()` 和 `B::print1()`。有的教科书中把这称做“类 B 重定义了 类 A 的函数 `print1`”，我们不采用这种讲法：因为这容易和 C 语言中的函数重定义 错误相互混淆。

另一方面，第 32-34 行的函数 `B::print2` 则与 A 中的函数 `A::print2` 非常有关。首先，函数 `B::print2` 也是一个虚对象函数：如果一个函数在父类里被声明为虚对象函数，那么在子类里具有同一个函数签名的函数也自动地是一个虚对象函数。

在介绍虚对象函数的作用之前我们需要先介绍另外一个概念：地址的向下类型转化。在 C 语言中不同类型的地址变量是不可以直接相互赋值的（必须通过强制的地址类型转化），如浮点数的地址不能直接赋值给整数的地址。

```
double * addressD = 0;
int * addressI = 0;
addressI = addressD;
```

唯一的例外是，在 C++ 中，子类的地址是可以赋值给父类的地址的，这种地址的赋值叫做地址的向下类型转化 down-casting。如下面的第 103 行和第 106 行所示。

```
101 B objectB;
102 B * addressB = &objectB;
103 A * addressA = addressB;
104 (*addressA).print1();
105
106 A & referenceA = objectB;
107 referenceA.print2();
```

```
108 referenceA.print3();  
109 referenceA.print4();
```

在第 104 行中 `addressA` 的类型是类 `A` 的对象的地址，但是它实际上存储的是一个类 `B` 的对象的地址。事实上，`down-casting` 允许 `addressA` 存储任何类 `A` 的子类的对象的地址。

与强制的地址类型转化不同，C++ 中允许的 `down-casting` 一般不会造成程序的运行错误。它的充分条件是：父类的成员必定也是子类的成员。如第 104 行所示，无论 `addressA` 中是 `A` 的对象还是 `B` 的对象，它们都具有 `print1`，`print2`，`print3`，和 `print4` 函数。

反过来，C++ 与许从父类地址到子类地址的直接转换 `up-casting` 吗？这是不允许的，因为可能产生错误的：子类可能定义新的成员，这使得子类的成员不一定存在于父类。

看到这里，我们可能会提出以下两个问题。`down-casting` 有什么用呢？第 104 行和第 107 行调用的是类 `A` 中的函数还是类 `B` 中的函数呢？简单的回答如下：`down-casting` 是为了体现虚对象函数的作用的；第 104 行会调用函数 `A::print1()`，因为 `print1` 在类 `A` 中不是虚对象函数；第 107 行会调用函数 `B::print2()`，因为 `print2` 是类 `A` 和类 `B` 中的虚对象函数；第 108 和 109 行会调用函数 `A::print3()` 和 `A::print4()`，因为这

两个函数在类 B 中没有被定义。再看下面的程序。

```
37 void task1(A & reference) {
38     reference.print1(); // static binding
39     reference.print2(); // late binding
40 }
41
42 int main() {
43     B objectB;
44     objectB.print1(); // static binding
45     objectB.print2(); // static binding
46     task1(objectB); // static binding
47 }
```

在第 43 行中我们定义了一个类 B 的对象，接着调用 print1 和 print2 函数，很明显 44-45 行将会调用类 B 中的函数。由于 task1 的参数是类 A 的引用，46 行中调用函数 task1 时，把 objectB 的引用传给 37 行 task1 的参数，是一个 down-casting。在这个程序的运行时候，我们可以确定，第 37-39 行的 reference 引用的是一个类 B 的对象。但事实上，第 38 行调用的是类 A 中的 print1 函数，而第 39 行调用的是类 B 中的 print2 函数。第 38 行和第 39 行的函数匹配的区别在于第 38 行中 print1 函数是一个普通函数，而第 39 行中的是一个虚对象函数。

为了更清楚地理解和更深刻地记住虚对象函数的调用规则，我们需要认识编译器是如何函数匹配过程。所谓函数匹配，就是把函数的名字替换成一个函数入口地址。每个函数被编译成机器代码后都会有一个入口地址，我们可以简单地

把它理解为函数中第一条机器代码的地址。

一般情况下，在编译的时候编译器会把函数名替换成函数入口地址。也就是说，这种匹配是发生在编译的时候的，是发生在程序执行之前的，函数名不会被保存在可执行文件中。这种函数匹配叫做静态匹配 static matching。另外一种匹配方式叫做推迟匹配 late matching 或者动态匹配 dynamic matching。在这第二种的匹配方式中，编译函数名被保留在可执行文件中，函数匹配被推迟到程序运行的时候进行。

简单地说，两种匹配的区别在于它们的时间不同，一个在编译的时候发生，一个推迟到运行的时候发生。在计算机科学的术语当中，静态用来形容编译时发生的事件，而动态用来形容运行时发生的事情。Static matching 和 dynamic matching 的另外一对同义词是 static binding 和 dynamic binding。

我们为什么分别需要区分静态匹配和动态匹配这两种匹配方式呢？因为它们各有各的必要性。首先，静态匹配函数的运行效率更高。我们更关心的是程序运行的速度而不是编译的速度，这是因为一个程序编译一次但可能运行无数次。而且，对于计算复杂度大的程序，编译的时间相对于运行的时间是可以忽略的。因此，静态匹配具有提高执行效率这个好处。在计算机的 CPU 中有一个指令寄存器，它存放的是下

一条指令的地址。运行时调用匹配好了的函数，只要把入口地址放入指令寄存器就可以开始执行了。如果在运行的时候才匹配函数，匹配所花的时间可能比执行程序实际所需要的时间更多。

其次，动态匹配也是必要的。经常地，在 `down-casting` 之后我们未必能知道对象的实际类型：如在上例 37-40 行的 `task1` 函数的参数的实际类型在编译的时候是不能确定的，因为它可以是任何类 `A` 的子类的对象。但在一般情况下由于 `task1` 可以被不同的函数用不同的参数调用，只有在运行的时候才能知道各次 `task1` 的调用时它的参数的实际类型。这里，我们可以看到：使用父类地址作为参数的好处是 `task1` 可以接受不同类型的参数，功能上相当于重载了多个函数。

我们知道，要匹配一个对象函数，除了知道函数的签名外，还需要知道在那个类里面找这个函数。那么，如果对象的类型只有在运行的时候才能确定，那么我们只能把函数匹配推迟到运行的时候才能进行。如在 39 行中，编译的时候只能确定 `reference` 的类型是父类 `A` 的引用，只有在运行的时候才能知道 `reference` 所引用的对象是类 `B` 的对象，这个时候它能够 在类 `B` 中查找和匹配函数 `B::print2()`。

了解了静态匹配的好处和动态匹配的必要性后，我们再了解 `C++` 的程序在分别在什么时候选择静态匹配和动态匹

配。首先，为了提高运行效率，编译器会尽可能使用静态匹配。在以下三种情况中静态匹配是足够的：

- 所调用的是非对象函数的匹配。这种情况下我们根本不需要在某个类里面查找。
- 使用对象而不是地址或引用来调用对象函数。这种情况下对象的类型是确定的，可以直接在该对象的类里面查找。如 34-35 行，由于 `objectB` 是对象，对象的类型 `B` 是可以确定的。
- 非虚对象函数。由于动态匹配的运行效率较低，C++ 仅当一个对象函数被声明为虚对象函数的时候才会执行动态匹配。如 38 行，由于 `print1` 不是虚对象函数，编译器会根据引用 `reference` 的类型（类 `A`），在类 `A` 中查找 `print1` 函数，所以匹配的是 `A::print()`。

另一方面，把以上的情况排除后，得到需要动态匹配函数的情况：调用的是虚对象函数，且虚对象函数所在的类型不能在编译的时候确定（即对象被保存在地址或引用中的情况）。如 39 行中，`print2` 是类 `A` 中的虚函数，而且 `reference` 所引用的对象可以是类 `A` 的对象，也可以是类 `A` 的任何子类的对象。

总结上例的函数匹配情况：由于通过对象调用，44-45 行的函数会在编译的时候分别被匹配为 `B::print1()` 和

B::print2(), 属于静态匹配; 由于是非虚对象函数, 38 行的函数会在编译的时候被匹配为 A::print1(), 属于静态匹配; 由于是虚对象函数而且通过被引用的对象调用, 39 行的函数会在运行的时候才被匹配为 B::print2(), 属于动态匹配。

3. 程序接口和多态性

即使在不同的 C++ 语言的教课书中, 程序接口和多态性这两个概念存在着多种的定义方式。我们介绍的是最接近于 Java 语言的一种, 有利于简化我们的进一步学习。

一个程序接口 (简称为接口 interface) 是一个函数的集合, 它是用来表示一个程序模块 (通常是一个类) 所提供的功能的。通常, 一个软件项目的各个接口在设计这个软件的时候就已经定义好了。也就是说, 在定义一个接口的时候通常不给出接口中各个函数的实现, 这些函数的实现可以等到软件设计好后再交给一个或多个的程序员实现。

在 C++ 的类中, 一个没有提供实现的虚对象函数叫做纯虚对象函数 pure virtual object function。包含一个或多个纯虚对象函数的类叫做抽象类 abstract class。注意, 这和抽象数据类型 (abstract data-type 就是 class) 中的抽象是不同意义的。接口可以使用抽象类定义: 一个接口就是一个只有纯虚对象函数的抽象类。看下面的例子。

```

1 class A // an abstract class
2 {
3 public:
4     // a pure virtual function
5     virtual void print() const = 0;
6 };
7
8 void task1(const A & reference) {
9     reference.print(); // late binding
10 }
11
12 class B : public A // B implements A
13 {
14 public:
15     // overrides A::print2()
16     virtual void print() const {
17         cout << "B::print()" << endl;
18     }
19 };
20
21 // C also implements A
22 class C : public A
23 {
24 public:
25     // overrides A::print2()
26     virtual void print() const {
27         cout << "C::print()" << endl;
28     }
29 };
30
31 int main() {
32     A a;
33     task1(B());
34     task1(C());
35 }

```

第 1-6 行定义了一个抽象类 A。A 是一个抽象类因为在第 5 行中声明了一个纯虚对象函数。一个纯虚对象函数定义的格式如下

`virtual 函数头 = 0;`

即把原来的函数体 `{ ... }` 使用占位标识 `= 0;` 代替。由于纯虚对象函数是只有声明没有定义的函数，*纯虚对象函数是不能被调用的*。由于这个原因，*在程序中不能定义抽象类的对象*：只要没有抽象类的对象，就不可能调用这个抽象类里面的纯虚对象函数了。不能产生对象，那么抽象类有什么用呢？抽象类是用来被继承的。

第 12-19 行定义了类 B，它继承了抽象类 A 并且实现了抽象类 A 中的纯虚对象函数。如果一个类继承了一个抽象类，并且实现了这个抽象中*所有的*纯虚对象函数，那么我们说这个类实现了这个抽象类所定义的接口。第 22-29 行定义类 C 也实现了接口 A。

虽然抽象类不能有对象，但是我们可以定义抽象类的地址和引用。如第 8 行所示，函数 `task1` 的参数是抽象类 A 的引用。其实抽象类的地址和引用是用来存储其它类型的对象的，这些类型必须是该抽象类的子类。也就是说，我们可以通过 `down-casting` 把子类对象的地址赋值给抽象类的地址，但是抽象类的地址中存储的一定不会是该抽象类的对象，因为抽象类不能有对象。

第 8-10 行的函数的参数 `reference` 是抽象类 A 的引用，在函数中用 `reference` 调用虚对象函数 `print`，因此这个函数的匹配方式会是动态匹配，即这个函数的匹配会推迟到运行

的时候进行。在 31-35 行的 main 函数中两次调用了函数 task1，参数分别为类 B 和类 C 的临时对象。这里，第 32 行中抽象类对象的定义是不允许的。

多态性是指具有同一个程序接口的多种数据类型可以通过这个接口做相同的处理。第 8-10 行的函数 task1 就是多态性的一个例子。抽象类 A 是类 B 和类 C 的接口，C++ 提供的 down-casting 可以把 B 和类 C 对象统一地标识为接口类 A 的地址然后在函数 task1 中做相同的处理。

然而，对不同类型的数据做完全相同的操作的情况并不多见。大多数情况下，在基本相同的操作中包含一些个别的处理。对于这些个别处理我们可以使用虚对象函数，让不同类型的对象通过执行各自的虚对象函数完成差异化的操作。

之所以叫多态性，就是要在统一中存在差异。统一的代码具有可读性高，可维护性高，代码量少等优点。*动态匹配则是多态性的基础*：没有动态匹配的函数带来的执行过程中的差异，功能单一的程序的用途将大大地受到限制。

下面我们通过两个例子来展示接口和多态性的实际应用。在第一个例子中，我们给出一个排序的函数，这个排序的函数具有多种排序方式，它的实现方法是使用一个具有多态性的“比较器”。看下面的程序。

```

1 class Student
2 {
3 public:
4     int id;
5     string name;
6     double GPA; // grade point average
7 };
8
9 // interface
10 class Comparator
11 {
12 public:
13     virtual bool isLarger(Student &, Student &)
14     const = 0;
15 };
16
17 // polymorphism function
18 void sort(vector<Student> & students,
19         const Comparator & comparator) {
20     int size = students.size();
21     for (int i = size - 1; i > 0; -- i) {
22         for (int j = 0; j < i; ++ j) {
23             Student & s1 = students[j];
24             Student & s2 = students[j + 1];
25             if (comparator.isLarger(s1, s2)) {
26                 Student temp = s1;
27                 s1 = s2;
28                 s2 = temp;
29             }
30         }
31     }
32 }
33
34 // implementation 1
35 class IdComparator : public Comparator
36 {
37     // overrides
38     bool isLarger(Student & s1, Student & s2) const {
39         return s1.id > s2.id;
40     }
41 };
42
43 // implementation 2
44 class NameComparator : public Comparator

```



```

45 {
46     bool isLarger(Student & s1, Student & s2) const {
47         return s1.name > s2.name;
48     }
49 };
50
51 // implementation 3
52 class GPAComparator : public Comparator
53 {
54     bool isLarger(Student & s1, Student & s2) const {
55         return s1.GPA > s2.GPA;
56     }
57 };
58
59 void print(vector<Student> & students) {
60     cout << "students = " << endl;
61     for (int i = 0; i < students.size(); ++ i) {
62         cout << students[i].id << " " <<
63         students[i].name << " " <<
64         students[i].GPA << endl;
65     }
66 }
67
68 int main() {
69     vector<Student> students;
70     for (int i = 0; i < 10; ++ i) {
71         Student s;
72         s.id = (5 + i) % 10 + 1;
73         s.name = "name0";
74         s.name[4] += 9 - i;
75         s.GPA = 3 + 0.05 * (i % 3);
76         students.push_back(s);
77     }
78     print(students);
79
80     sort(students, IdComparator());
81     print(students);
82
83     sort(students, NameComparator());
84     print(students);
85
86     sort(students, GPAComparator());
87     print(students);
88 }

```

第 1-7 行定义了一个代表学生的数据类型 `Student`，它有 3 个成员变量。第 10-15 行定义了一个“比较器”接口，它有一个纯虚对象函数。这个函数的两个参数都是 `Student` 的对象，它返回第一个对象是否“大于”第二个对象。注意在第 13-14 行中声明函数的时候并不一定要给出参数的名字，只给出参数的类型也可以。

第 17-31 行的是一个排序函数，它使用冒泡排序算法。冒泡排序算法的基本思想是每轮把当前最大的元素放到最后一个位置。这里排序的是元素为 `Student` 的 `vector`，当我们要根据 `Student` 的不同的属性，如学号、姓名、平均绩点，间的比较来对 `vector` 排序的时候，我们可以分别写三个排序算法，但这三个排序算法的代码基本相同。而第 17-31 行的函数是一个具有多态性的函数，它利用第 25 行中接口引用 `comparator` 的动态绑定的函数，调用不同的比较器中实现的不同的比较方法。

从 34-58 行定义了不同的比较器子类，它们都实现了第 10-15 行的比较器接口，它们分别使用学号、姓名、平均绩点来比较 `Student` 的对象。

在 59-66 行定义了输出一个 `vector` 中的所有 `Student` 对象的函数。在 69-77 行初始化了一个长度为 10 的 `vector`。最后，在 80、83 和 86 行分别使用不同的比较器对象调用 `sort`

函数对 `vector` 对象 `students` 进行不同的排序。

有同学会问：如果在实际运行的时候第 25 行的接口引用 `reference` 中实际存储的对象的类中没有定义函数 `isLarger` 的话，程序会怎么样呢？实际上这个对象的类型肯定已经定义了 `isLarger` 函数。这是因为，如果一个类型没有实现接口中的所有函数，那么这个类仍然是一个抽象类，那么这个类不能具有对象。所以，接口引用 `reference` 中实际存储的对象的类一定不是抽象类，它一定实现了接口中的函数 `isLarger`。

下面我们给出第二个更复杂的例子。这个例子中包含了两个接口，这里先介绍有关的概念。第一个接口叫做遍历器 `iterator`，它是用来遍历一个容器中的各个元素的。`Iterator` 也被从字面上翻译为迭代器。为了简化，我们架设容器中的元素为浮点数，在以后的章节中我们会再给出模版类型的 `iterator` 接口。`Iterator` 包含两个函数，第一个函数返回是否还没遍历到容器的最后一个元素，如果有的话第二个函数返回下一个元素的值。

第二个接口叫做可遍历的（容器） `iterable`。实现接口 `iterable` 的容器类必须具有函数 `getIterator`，它返回一个该容器的遍历器。也就是说，如果一个容器是可遍历的（或者说它实现了接口 `iterable`），那么它能给出一个遍历它自己的遍历器。看下面的程序。

```

1 // interface Iterator
2 class Iterator
3 {
4 public:
5     virtual bool hasNext() = 0;
6     virtual double next() = 0;
7 };
8
9 // interface Iterable
10 class Iterable
11 {
12 public:
13     virtual Iterator * getIterator() = 0;
14 };
15
16 // polymorphism function
17 void writeToFile(const char filename[],
18                 Iterable & iterable) {
19     Iterator * it = iterable.getIterator();
20     ofstream out(filename);
21     while ((*it).hasNext()) {
22         double value = (*it).next();
23         out << value << " ";
24     }
25     out.close();
26     delete it;
27 }
28
29 // implementation of Iterable
30 class FixedLengthVector : public Iterable
31 {
32 private:
33     int length;
34     double * array;
35
36 public:
37     FixedLengthVector(int length1) {
38         length = length1;
39         array = new double[length];
40     }
41
42     ~FixedLengthVector() {
43         delete [] array;
44     }

```

```

45
46     double & operator [] (int index) {
47         return array[index];
48     }
49
50 private:
51
52     // implementation of Iterator
53     class Iterator1 : public Iterator
54     {
55     private:
56         FixedLengthVector * vector;
57         int nextIndex;
58
59     public:
60         Iterator1(FixedLengthVector * vector1) {
61             vector = vector1;
62             nextIndex = 0;
63         }
64
65         // overrides
66         bool hasNext() {
67             return (nextIndex < (*vector).length);
68         }
69
70         // overrides
71         double next() {
72             ++ nextIndex;
73             return (*vector).array[nextIndex - 1];
74         }
75     }; // end of inner-class
76
77 public:
78     // overrides
79     Iterator * getIterator() {
80         return new Iterator1(this);
81     }
82 };
83
84 int main() {
85     FixedLengthVector vector(10);
86     for (int i = 0; i < 10; ++ i) {
87         vector[i] = 1 + i;
88     }

```

```
89     writeToFile("1.txt", vector);  
90 }
```

第 2-7 行定义了抽象类接口 `Iterator`，它包含上面解释过的两个纯虚对象函数。第 10-14 行定义了抽象类接口 `Iterable`，它包含一个返回一个 `Iterator` 对象地址的纯虚对象函数。

第 17-27 行定义了一个具有多态性的函数 `writeToFile`，这个函数把一个容器中的各个元素写到一个文件中。这个函数无须知道具体的容器是什么、或这个容器是如何存储数据的，它只要求这个容器实现接口 `Iterable`。也就是说，只要可以拿到容器的 `iterator`，这个函数就可以把容器的元素写到文件中。

这个多态性函数的实现如下。首先，第 19 行通过接口的引用 `iterable` 取得容器对象的 `iterator`，并存储于地址变量 `it`。第 21-24 行的循环中，通过遍历器取得容器中的各个元素，并写到文件中。最后，由于 `it` 中存储的对象是在 19 行中的 `getIterator` 函数中动态分配的对象，我在第 26 行中把它释放掉。

第在 30-82 行中我们定义了一个实现接口 `Iterable` 的简单容器类。在第 53-75 行我们定一个实现接口 `Iterator` 的遍历器类，注意到它是一个内部类。这个内部类中的函数可以访问外部类中的私有成员。我们会看到，在容器类外部我们

无须直接访问这个遍历器类，所以我们把这个内部类定义为私有类型。

最后在 `main` 函数中，第 85-88 行初始化了一个长度为 10 的容器，然后第 89 行调用多态性函数把容器中的元素写到文件。

在 13 行接口中的函数返回动态分配内存的地址并不是一个好的设计，因为它需要依赖用户使用后把它释放掉。我们可以思考如何把这个地址封装起来，以使得这个地址中的动态分配内存的释放可以自动的完成。

4. 正确使用继承和多态性（选读）

使用继承和多态性时是要有注意的地方的，否则可能产生错误。下面的程序试图通过继承来改进上一章介绍过的（非模版的）`Vector` 类。

首先说明一下改进的思路。在上一章的 `Vector` 类中，每插入一个元素，`Vector` 就要分配一个新的、大小足以存储原来的元素加上新的元素的内存空间，然后把原来的元素逐个地拷贝到新的内存空间中，最后拷贝新的数据。试想，如果 `Vector` 中已经有 10000 个元素，那么每插入一个元素就要拷贝 10000 以上之前的元素，这样开销将非常大。同样地，如果我们向一个 `Vector` 中顺序插入 10001 个元素，那么总共需

要拷贝的元素的次数将为 50005000, 即插入 1 个元素平均拷贝 5000 次。

$$1+2+\dots+10000=10000*(10000+1)/2=50005000$$

改进的方法是每次分配新的存储空间的时候, 预先分配更多的空间, 以减少拷贝的次数。一种常用的设置是: 每次分配两倍于所需的空間。例如, 当前的 **Vector** 中能存储 4 个元素, 当要插入第 5 个元素的时候分配能存储 8 个元素的空间, 然后拷贝已有的 4 个元素; 这样当后续插入第 6-8 个元素的时候, 就无须分配新的空间和拷贝已有的元素。

因为 **Vector** 中能存储元素的个数只能以两倍增长, 所以长度只能是 1, 2, 4, 8, ... 2^k 等的 2 的次方的数。它的好处是仅需在 **Vector** 的容量超越 2^k 的那些时候进行内存分配及拷贝; 缺点是要预先分配较多的内存空间而浪费存储空间。继续使用之前的例子, 如果我们向一个 **Vector** 中顺序插入 10001 个元素, 那么总共需要拷贝的元素的次数为 16384:

$$1+2+4+8+\dots+8192=(2^{10+1}-1)/(2-1)=16384$$

即插入每个元素仅需拷贝 1.6383 次。但是当插入完第 10001 个元素后, **Vector** 所占用的存储空间为 16384 个, 也就是浪费了 6383 个空间。这种元素的个数以两倍增长的空间分配方式是一种以浪费空间换取高运行效率的算法。

我们把这个新的 `Vector` 的子类命名为 `FastVector`。在 `FastVector` 中实际上存储的元素个数不再与已分配的存储空间相等。例如，在刚才的例子中在插入第 10001 个元素后，实际上存储的元素个数为 10001，而已分配的存储空间的大小为 16384。所以在 `FastVector` 中我们需要多一个成员变量 `count`，我们用 `count` 表示元素个数，而用原来 `Vector` 中的 `capacity` 表示存储空间的大小。

看下面的程序。与上一章的程序相比，`FastVector` 在第 4 行增加了一个成员变量 `count`，定义了私有成员函数 `ensureCapacity0` 和 `assign0`，定义了公有的成员函数 `operator =`，`push_back`，和 `size`，并继承了 `Vector` 中的所有成员和函数。

```
1 class FastVector : public Vector
2 {
3 private:
4     int count; // additional member variable
5
6     void ensureCapacity0(int capacity2) {
7         if (capacity >= capacity2) return;
8         while (capacity < capacity2) {
9             capacity *= 2;
10        }
11        double * temp = new double[capacity];
12        copy0(temp, elements, count);
13        delete [] elements;
14        elements = temp;
15    }
16
17    // additional member function
18    void assign0(const FastVector & vector2) {
```

```

19         if (elements != 0) {
20             delete [] elements;
21         }
22         capacity = vector2.capacity;
23         count = vector2.count;
24         elements = new double[capacity];
25         copy0(elements, vector2.elements, count);
26     }
27
28 public:
29     // calls the default constructor of Vector
30     FastVector() {
31         capacity = 1;
32         count = 0;
33     }
34
35     // calls the default constructor of Vector
36     FastVector(const FastVector & vector2) {
37         assign0(vector2); // static binding
38     }
39
40     // no need to define a destructor, the default
41     // destructor calls that of its super-class
42
43     // may cause error
44     FastVector & operator = (const FastVector &
45                             vector2) {
46         assign0(vector2);
47     }
48
49     // may cause error
50     void push_back(double elem) {
51         // static binding
52         ensureCapacity0(count + 1);
53         elements[count] = elem;
54         ++ count;
55     }
56
57     int size() {
58         return count;
59     }
60
61 };

```

第 6 行的函数 `ensureCapacity0` 保证 `FastVector` 对象的内存空间的大小最少有 `capacity2` 个。如果没有，那么第 8-10 行通过不断地成倍增大当前的内存空间值 `capacity`，使他不小于 `capacity2`。第 11-14 行分配一个新的（大小为新的 `capacity` 的）内存空间，并复制就内存空间中的元素的值到新内存空间中。注意第 12 行我们只复制 `count` 个元素，这是因为实际上存储的元素个数为 `count`，其余的位置上没有存放有意义的元素。

在 18-26 行定义了一新的 `assign0` 函数，因为原来的 `Vector::assign0` 中没有拷贝成员变量 `count`，因此需要加上第 23 行的赋值。其次，在 `Vector` 中，`capacity` 既表示存储的元素个数也表示内存空间的大小，而 `capacity` 在 `FastVector` 中只表示内存空间的大小，因此在 25 行需要把拷贝元素的个数改为 `count`。

在 30 行和 36 行的两个构造函数调用之前，C++ 都会自动地先调用父类 `Vector` 的默认构造函数。而在 `Vector` 的构造函数中已经给成员变量 `elements` 赋了值，这就是为什么我们无须在 `FastVector` 的上述两个构造函数中给 `elements` 赋值。我们在下一章中将会讨论关于子类自动调用父类构造函数的规则。注意 37 行中对 `assign0` 的调用是静态匹配的，因为 `assign0` 不是一个虚对象函数。

FastVector 的析构函数（包括自动生成的析构函数）也会自动地调用它的父类 **Vector** 的析构函数。所以，我们可以省略 **FastVector** 中的析构函数。

在 44-47 行中定义的 `operator =` 函数中，我们使用的是在 **FastVector** 中定义的 `assign0` 函数。在 50-55 行定义的函数 `push_back` 和在 57-59 行定义的函数 `size` 都不同于在类 **Vector** 中定义的对应该函数。

```
63 int main() {
64     FastVector vector1;
65     for (int i = 0; i < 10; ++ i) {
66         vector1.push_back(i + 0.5);
67     }
68     for (int i = 0; i < 10; ++ i) {
69         cout << vector1[i] << endl;
70     }
71 }
```

在上面的 `main` 函数中，我们调试了 **FastVector**，从这个 `main` 函数中，我们发现不了 **FastVector** 有什么问题。再看下面的程序。

```
73 void errorInside(Vector & vector4) {
74     vector4.push_back(100);
75     /*
76     Is 100 in the 6-th element in vector4?
77     - Yes, if FastVector::push_back was called.
78     - No, if Vector::push_back was called.
79     */
80     cout << vector4.size() << endl;
81     /*
82     We will see 9.
83     Which means that Vector::size() was called,
84     and Vector::push_back was called previously.
```

```

85     */
86 }
87
88 int main() {
89     FastVector vector1;
90     for (int i = 0; i < 5; ++ i) {
91         vector1.push_back(1 + i);
92     }
93     // now, count==5, capacity==8
94
95     // C++ allows downcasting
96     Vector * vector2 = &vector1;
97     Vector & vector3 = vector1;
98
99     // pass downcasted reference
100    errorInside(vector1);
101 }

```

首先看 88-101 行的 main 函数。从 89-92 行，我们创建了一个 **FastVector** 的对象并给它添加了 5 个元素。这时，容器中有 5 个元素，分配的内存空间大小为 8（**count==5**，**capacity==8**）。注意，C++ 允许地址和引用的 down-casting，如 96-97 行，和 100 行中的参数传递。

错误出现在第 73-88 行的函数中，在 74 行中容器添加了一个元素，之后在 80 行中输出容器的大小，但我们看到的是 9 而不是 6。这是因为函数 **push_back** 和 **size** 都不是虚对象函数，所以虽然引用 **vector4** 中存放的是 **FastVector** 的对象，但是调用的却是 **Vector** 中的函数，这是造成错误的原因。

现在，我们来思考一下这个错误。我们知道继承有两种用途：产生子类以共享代码的，和实现接口以使用多态性。

首先，我们看看第一种产生子类的用途。作这种用途的时候最好不要像上面的例子一样改变原来成员变量的意义，因为这样会使得原来的成员函数不能再被正确调用。

试图通过重写函数覆盖父类中不再能被正确调用的函数，如上例中 50-59 行的两个函数，也不是一个好的方法。因为，在默认情况下，为了提高程序的运行速度，对象函数被定义为非虚的。这个时候仍然可能通过父类的引用调用子类中试图覆盖的函数，如上例 74 和 80 行所示。

继承的第二种用途是使子类实现接口。地址的向下类型转换 **down-casting** 和继承是 C++ 中能编写多态性函数的必要的语法支持。在这第二种用途中，一般情况下，只要父类是接口就够了，如之前的一个小节中的例子所示。

谈一个关于我们这个课程不涉及的知识点：多重继承，指一个类同时继承多个父类。我们往往避免使用多重继承，因为它的使用复杂并容易出错。但是，如果多重继承中的多个父类中只有一个不是接口，那么使用起来就与单继承一样，这使得普通的多重继承时所可能引发的问题将不会出现。

下面归纳一下使用继承的一些建议：

- 继承父类的时候不要改变父类成员的意义，这样会使得父类中原有的一些函数变得一调用即出错。

- 不要在子类中定义与父类具有同样签名的非虚函数，这会使得这两个具有同样签名的函数的调用混乱。
- 当继承是为了支持多态性时，尽可能继承接口。

有时我们使用继承既有为了代码重用，又有为了多态性，应该怎样处理呢？例如，我们想上例中的两个容器中的相同代码可以重用，又想它们有同样的接口，应该怎样写程序呢？首先，我们因该正确找到两个容器中的相同的部分把它写在一个相同的类中。第二，我们应该为这两个类定义一个共有的接口。为了把这两个逻辑清楚地划分，以及使接口的中的函数清晰地呈现，在下面的例子中，我们先定义一个接口 **Vector**，再定义一个继承这个接口并实现两个容器中相同功能的类 **AbstractVector**。而最终要实现的两个容器类分别命名为 **SlowVector** 和 **FastVector**。看下面的程序。

```
1 // interface
2 class Vector
3 {
4 public:
5     virtual void push_back(double elem) = 0;
6
7     virtual void pop_back() = 0;
8
9     virtual double & operator [] (int index)
10    const = 0;
11
12    virtual int size() const = 0;
13 };
14
15 // common utility
16 class AbstractVector : public Vector
```

```

17 {
18     protected:
19
20         virtual int nextCapacity0(int capacity2) = 0;
21
22     protected:
23
24         double * elements;
25         int count;
26         int capacity;
27
28         void copy0(double to[], const double from[],
29                     int num) const {
30             for (int i = 0; i < num; ++ i) {
31                 to[i] = from[i];
32             }
33         }
34
35         void ensureCapacity0(int capacity2) {
36             int capacity3 = nextCapacity0(capacity2);
37             if (capacity == capacity3) return;
38             capacity = capacity3;
39             double * temp = new double[capacity];
40             copy0(temp, elements, count);
41             delete [] elements;
42             elements = temp;
43         }
44
45         void assign0(const AbstractVector & vector2) {
46             if (elements != 0) {
47                 delete [] elements;
48             }
49             capacity = vector2.capacity;
50             count = vector2.count;
51             elements = new double[capacity];
52             copy0(elements, vector2.elements, count);
53         }
54
55     public:
56
57         AbstractVector() {
58             elements = new double[1];
59             capacity = 1;
60             count = 0;

```



```

61     }
62
63     ~AbstractVector() {
64         delete [] elements;
65     }
66
67     AbstractVector(const AbstractVector & vector2) {
68         elements = 0;
69         assign0(vector2);
70     }
71
72     AbstractVector & operator =
73     (const AbstractVector & vector2) {
74         assign0(vector2);
75         return (*this);
76     }
77
78     void push_back(double elem) {
79         ensureCapacity0(count + 1);
80         elements[count] = elem;
81         ++ count;
82     }
83
84     void pop_back() {
85         -- count;
86         ensureCapacity0(count);
87     }
88
89     double & operator [] (int index) const {
90         return elements[index];
91     }
92
93     int size() const {
94         return count;
95     }
96
97 };
98
99 // implementation 1
100 class SlowVector : public AbstractVector
101 {
102 protected:
103     int nextCapacity0(int capacity2) {
104         if (capacity2 == 0) return 1;

```

```

105         return capacity2;
106     }
107
108 };
109
110 // implementation 2
111 class FastVector : public AbstractVector
112 {
113 protected:
114     int nextCapacity0(int capacity2) {
115         if (capacity2 == 0) return 1;
116         int capacity3 = capacity;
117         while (capacity3 < capacity2) {
118             capacity3 *= 2;
119         }
120         while (capacity3 >= capacity2 * 4) {
121             capacity3 /= 2;
122         }
123         return capacity3;
124     }
125
126 };

```

其实，两个容器的类 `SlowVector` 和 `FastVector` 的不同之处只在于每次分配存储空间的大小。所以在 20 行中，我们定义了一个纯虚对象函数 `nextCapacity0` 以让两个容器类能有差异地返回各自需要分配的存储空间的大小。

在第 100 行的类 `SlowVector` 中，我们给出了它的 `nextCapacity0`。我们可能觉得在 `SlowVector` 中必须给出默认构造函数、拷贝构造函数、析构函数、和赋值操作符（`operator =`）。其实，因为具有地址类型的成员变量，这些函数都是需要的，但是在上例中我们都可以忽略不写，其中的原因我们将会在下章中详细地讨论。简单地说，默认构

构造函数、拷贝构造函数、析构函数和赋值操作符函数都是编译器会自动生成（合成）的函数。在它们由编译器合成的版本中，会自动地调用父类中的对等的函数以构造、拷贝或释放属于父类部分的成员变量。

下面我们使用多态性程序来比较两个容器类的运行速度。我们发现，当容器大小为 100000 时，**SlowVector** 用了 30 秒，而对于大 100 倍的容器，**FastVector** 仅用了不到 1 秒的时间。

```
128 void pushValues(Vector & vector, int size) {
129     for (int i = 0; i < size; ++ i) {
130         vector.push_back(i + 0.5);
131     }
132 }
133
134 void popAll(Vector & vector) {
135     while (vector.size() > 0) {
136         vector.pop_back();
137     }
138 }
139
140 void print(Vector & vector) {
141     for (int i = 0; i < vector.size(); ++ i) {
142         cout << vector[i] << endl;
143     }
144 }
145
146 void test1() {
147     SlowVector vector1;
148     pushValues(vector1, 100000);
149     SlowVector vector2(vector1);
150     SlowVector vector3;
151     vector3 = vector2;
152     //print(vector3);
153     popAll(vector1);
154 }
155
```

```

156 void test2() {
157     FastVector vector1;
158     pushValues(vector1, 100000 * 100);
159     FastVector vector2(vector1);
160     FastVector vector3;
161     vector3 = vector2;
162     //print(vector3);
163     popAll(vector1);
164 }
165
166 int main() {
167     test1(); // about 30s
168     test2(); // < 1s
169 }

```

继承提供代码重用的功能。然而，由继承提供的代码重用功能完全可以由对象包含来取代。再下面的例子中，由 **FastVector** 继承 **AbstractVector** 将改为由 **FastVector** 包含 **VectorImpl** 的一个对象，即 **VectorImpl** 的一个对象是 **FastVector** 的一个成员对象。其中 **VectorImpl** 和 **AbstractVector** 的功能基本相同的功能。

```

15 // common utility
16 class VectorImpl
17 {
18 private:
19
20     double * elements;
21     int capacity;
22
23     void copy0(double to[], const double from[],
24               int num) const {
25         for (int i = 0; i < num; ++ i) {
26             to[i] = from[i];
27         }
28     }
29
30     void assign0(const VectorImpl & vector2) {

```

```

31         if (elements != 0) {
32             delete [] elements;
33         }
34         capacity = vector2.capacity;
35         elements = new double[capacity];
36         copy0(elements, vector2.elements, capacity);
37     }
38
39 public:
40
41     VectorImpl() {
42         elements = new double[1];
43         capacity = 1;
44     }
45
46     ~VectorImpl() {
47         delete [] elements;
48     }
49
50     VectorImpl(const VectorImpl & vector2) {
51         elements = 0;
52         assign0(vector2);
53     }
54
55     VectorImpl & operator =
56     (const VectorImpl & vector2) {
57         assign0(vector2);
58         return (*this);
59     }
60
61     double & operator [] (int index) const {
62         return elements[index];
63     }
64
65     void changeCapacity(int capacity2) {
66         if (capacity == capacity2) return;
67         int size = capacity;
68         capacity = capacity2;
69         if (size > capacity) size = capacity;
70         double * temp = new double[capacity];
71         copy0(temp, elements, size);
72         delete [] elements;
73         elements = temp;
74     }

```

```

75
76     int getCapacity() {
77         return capacity;
78     }
79
80 };
81
82 // implementation 1
83 class SlowVector : public Vector
84 {
85 private:
86     VectorImpl impl;
87     int count;
88
89     int nextCapacity0(int capacity2) {
90         if (capacity2 == 0) return 1;
91         return capacity2;
92     }
93
94 public:
95
96     SlowVector() {
97         count = 0;
98     }
99
100    // no need to define copy constructor
101    // no need to define operator =
102
103    double & operator [] (int index) const {
104        return impl[index];
105    }
106
107    int size() const {
108        return count;
109    }
110
111    void push_back(double elem) {
112        int capacity = nextCapacity0(count + 1);
113        impl.changeCapacity(capacity);
114        impl[count] = elem;
115        ++ count;
116    }
117
118    void pop_back() {

```

```

119         int capacity = nextCapacity0(count - 1);
120         impl.changeCapacity(capacity);
121         -- count;
122     }
123
124 };
125
126 // implementation 2
127 class FastVector : public Vector
128 {
129 private:
130     VectorImpl impl;
131     int count;
132
133     int nextCapacity0(int capacity2) {
134         if (capacity2 == 0) return 1;
135         int capacity3 = impl.getCapacity();
136         while (capacity3 < capacity2) {
137             capacity3 *= 2;
138         }
139         while (capacity3 >= capacity2 * 4) {
140             capacity3 /= 2;
141         }
142         return capacity3;
143     }
144
145 public:
146     // 以下省略，与 SlowVector 中的代码相似。

```

习题

1. 用自己的话解释下面的概念

继承

派生类

基类

虚对象函数

地址的向下类型转化

函数匹配

静态匹配

动态匹配

接口

纯虚对象函数

抽象类

实现一个接口

多态性

遍历器

2. 用自己的话回答下面的问题

父类中的私有成员与保护成员对于子类有什么区别？

如何初始化成员变量？

在子类中声明一个与父类中具有同样签名的普通对象函数

为什么不是一个函数重定义错误？

down-casting 的作用是什么？

静态匹配和动态匹配的区别是什么？

静态匹配和动态匹配各自的好处是什么？

编译器何时使用静态匹配和动态匹配？

在 C++ 中如何表示一个接口？

可以定义抽象类的地址吗？

3. 设计一个稀疏矩阵类 **Sparse**

在大多数的实际大规模矩阵的应用中，矩阵的元素只有少数为非零元素，这中矩阵叫做稀疏矩阵。因为稀疏矩阵中只有少部分非零元素，所以我们并不需要记录整个矩阵的元素，我们可以只记录这些非零元素来节省存储空间。

通常我们可以使用一个三元组(row, column, value)来表示稀疏矩阵中的一个元素。我们可以用一个容器来存放所有非零元素的三元组，那么这个容器中的所有三元组就可以用来表示这个稀疏矩阵。

本题要求设计一个稀疏矩阵类 `Sparse`，它实现下面给出接口的 `Matrix`，同时它具有以下的成员函数：

- 构造函数 `Sparse(int rows, int column)`，它把矩阵中所有的元素初始化为 0。
- 拷贝构造函数 `Sparse(Matrix & matrix2)`。注意，它实际上不是一个拷贝构造函数。
- 释构函数。
- 一个 `print` 函数，它打印把所有的非零元素的三元组 (Entry)。

`Matrix` 接口：

```
1 class Matrix
2 {
3 public:
4
5     class Entry
6     {
```

```

7     public:
8         int row;
9         int column;
10        double value;
11    };
12
13    class Iterator
14    {
15    public:
16        virtual bool hasNext() const = 0;
17        virtual Entry next() = 0;
18    };
19
20    public:
21
22        virtual Iterator * iterator() = 0;
23
24        virtual int size(int dimension) const = 0;
25
26        virtual void set(int row, int column,
27        double value) = 0;
28
29        virtual double get(int row, int column)
30        const = 0;
31
32        virtual void print() = 0;
33
34        virtual Matrix & operator =
35        (Matrix & matrix2) = 0;
36    };

```

主程序:

```

40    #include <iostream>
41    using namespace std;
42
43    void printMatrix(Matrix & matrix) {
44        matrix.print();
45    }
46
47    int main() {
48        int rows;
49        int columns;
50        cin >> rows >> columns;
51

```

```

52     int row1;
53     int column1;
54     double value1;
55     cin >> row1 >> column1 >> value1;
56
57     int row2;
58     int column2;
59     double value2;
60     cin >> row2 >> column2 >> value2;
61
62     int row3;
63     int column3;
64     double value3;
65     cin >> row3 >> column3 >> value3;
66
67     int row4;
68     int column4;
69     double value4;
70     cin >> row4 >> column4 >> value4;
71
72     Sparse sparse1(rows, columns);
73     Matrix & matrix1 = sparse1;
74     matrix1.set(row1, column1, value1);
75     matrix1.set(row2, column2, value2);
76     matrix1.print();
77     cout << endl;
78
79     Sparse sparse2(sparse1);
80     Matrix & matrix2 = sparse2;
81     matrix2.print();
82     cout << endl;
83
84     matrix2.set(row3, column3, value3);
85     matrix2.set(row4, column4, value4);
86     matrix1 = matrix2;
87     matrix1.print();
88 }

```

输入样本

```

1000000 1000000
1 1 10
1000000 1000000 20
1 1000000 30
1000000 1 40

```

输出样本

(1,1,10)

(1000000,1000000,20)

(1,1,10)

(1000000,1000000,20)

(1,1,10)

(1,1000000,30)

(1000000,1,40)

(1000000,1000000,20)