

# 封装

封装的例子在日常生活中可以经常看到。一台电视机里面包含很多的零件，但是我们只看到显示器的正面和一些控制用的按钮。厂家把电视机的零件封装起来，用户就不会因为不小心损坏零件，同时电视机的控制也更加简化和好用。

程序包含很多功能模块，每个功能模块包含数据和操作这些数据的函数。封装的目标就是使得每个功能模块的使用方法简单明了，并且消除由用户的错误操作而引起程序出错的情况。

## 1. 封装数据和程序

在 C++ 中，程序的功能模块以类为单位，类的对象中包含成员变量，并提供操作这些成员变量的成员函数。在 C++ 中，封装的功能由 `public`（共有的）和 `private`（私有的）两个保留字提供。看下面的例子。

```
1 class Student
2 {
3 private:
4     int id;
5     char name[20];
6
7 public:
```

```

8      double age;
9
10 public:
11     Student() {
12         id = 0;
13         strcpy(name, "NO NAME");
14         age = 0.0;
15     }
16
17     int getId() {
18         return id;
19     }
20
21     void setName(const char name2[]) {
22         strcpy(name, name2);
23     }
24 };
25
26 int main() {
27     Student s;
28     s.age = 18.5;
29     s.id = 12340001;
30     int id = s.getId();
31     s.setName("Cong Liu");
32 }

```

这两个保留字分别出现在第 3 行和第 7 行中。这样，第 7 行的 `age` 就是公有的，而 `id` 和 `name` 都是私有的。由于第 10 行有一个 `public`（其实它是多余的，由于第 7 行已有一个），所以类中的 3 个函数都是公有的。

如果一个变量被定义为私有的，那么它只能够在类里面定义的函数中访问，而不能在类外面定义的函数中被访问。例如，`id` 能在构造函数和 `getId` 函数中被访问，但不能在 `main` 函数中被访问。对于第 29 行，编译器会报错。

当我们定义一个类的时候，为了避免用户对数据的错误操作，我们可以把数据设为私有的，然后提供一些让用户简便而安全地操作这些数据的公有函数。除此以外，封装还可以提供数据读写控制的功能：在上面的例子中，我们只提供 `getId` 函数来读取 `id` 的值，只提供 `setName` 函数来设置 `name` 的值，即对于类外面的函数 `id` 实际上是一个只读的数据，而 `name` 实际上是一个只写的数据。

C++的库函数中提供了一个 `string` 类，这个类实际上封装了一个字符数组 `char array`。本章将以 `string` 类的实现作为实例讲解 C++中封装的一些程序编写方法。在此之前，我们先深入讨论两个重要的、有关的概念。

## 2. 常量参数和常量函数

当把一个对象传作为参数给一个函数的时候，为了节省拷贝整个对象的开销，我们可以传对象的引用或地址。我们知道 C++的引用其实跟地址是不同写法而已，编译后会被生成相同的代码。看下面的程序。

```
1 class Student
2 {
3 public:
4     int id;
5 };
6
7 void print(const Student & s2) {
8     cout << s2.id << endl;
9 }
```

```
10
11 void print(const Student * s2) {
12     cout << (*s2).id << endl;
13 }
14
15 int main() {
16     Student s;
17     s.id = 12340001;
18     print(s);
19     print(&s);
20 }
```

在 18 和 19 行，中分别调用了第 7 和 11 行的两个 print 函数。在第 7 和 11 行的两个 print 函数中的 s2 实际上是 main 函数中的 s 对象，为了避免在 print 函数中疏忽地对 s 对象的修改，我们在两个 print 函数中把参数设定为常量。

如果我们把 print 函数写成对象函数，那么原来的参数将变为隐形对象参数；而且隐形对象参数的类型为 Student \*。其实，隐形对象参数可以通过 this 访问，而 this 的类型就是 Student \*。那么，我们能不能与上例 11 行的 print 函数一样，把一个对象函数的隐形对象参数定义为常量呢？这是可以的，C++ 让我们在一个对象函数头后面加上 const，以表示这个函数的隐形对象参数是常量，并且称这样的函数为常量函数。看下面的例子。

```

1 #include <iostream>
2 using namespace std;
3
4 class Student
5 {
6 public:
7     int id;
8
9     void setId(int id2) {
10         id = id2;
11     }
12
13     void print() const {
14         cout << id << endl;
15         id = 12340002;
16         setId(12340002);
17     }
18 };
19
20 void print(const Student * s) {
21     cout << (*s).id << endl;
22 }
23
24 int main() {
25     Student s;
26     s.setId(12340001);
27     print(&s);
28     s.print();
29 }

```

在上面的例子中，13 行的函数是一个常量函数。在常量函数中，隐形对象参数是常量，因此 15 和 16 行是不正确的。首先，如果一个对象是常量，那么他的成员数据也是常量，所以 15 行在尝试给一个常量赋值。其次，*常量的对象函数不*

能调用不是常量的对象函数。

在一个对象函数中调用另外一个对象函数，会把调用者的隐形对象参数作为被调用者的隐形对象参数。以上例为例，第 13 行的 `print` 函数的隐形对象参数的类型是 `const Student *`，但是第 9 行的 `setId` 函数的隐形对象参数类型是 `Student *`，因此第 16 行尝试把一个常量地址（第 13 行函数的隐形对象参数）赋给一个变量地址（第 9 行函数的隐形对象参数）。这是 C++ 不允许的，因为这可能引入错误。

最后，请注意类 `Student` 中的函数 `print() const` 和函数 `print()` 可以同时存在，因为它们的函数签名是不同的。因为函数 `print() const` 的隐藏参数的类型是 `const Student *`，而函数 `print()` 的隐藏参数的类型是 `Student *`。看下面的例子。

```
1 class Student
2 {
3 public:
4     Student() {
5     }
6
7     void print() const {
8         cout << "print() const" << endl;
9     }
10
11    void print() {
12        cout << "print()" << endl;
13    }
14 };
15
16 int main() {
```

```
17     const Student s1;
18     Student s2;
19     s1.print();
20     s2.print();
21 }
```

在 19 和 20 行中，我们调用的是不同的函数，通过常量 s1 调用的函数 print 会匹配第 7 行的常量函数，而通过变量 s2 调用的函数 print 会匹配第 11 行的非常量函数。

### 3. 构造函数

C++中，构造函数是可以有参数的，每一个类可以有多个构造函数，它们的参数表必须不同，否则会出现一个以上函数签名相同，即函数重定义错误。看下面的例子。

```
1 class Student
2 {
3 private:
4     int id;
5     char name[20];
6
7 public:
8     Student() {
9         id = 0;
10        strcpy(name, "NO NAME");
11    }
12
13    Student(int id2, char name2[]) {
14        id = id2;
15        strcpy(name, name2);
16    }
17 };
```

这个类提供了两个构造函数，那么这个类的对象可以使用其中一个构造函数进行初始化。C++规定，每个对象只能（并且必须）调用所在类的其中一个构造函数。看下面的程序。

```
19 int main() {  
20     Student s1;  
21     Student s1();  
22     Student s2(12340001, "Cong Liu");  
23     Student s2(12340001);  
24 }
```

在第 20 行定义的对象 s1 会调用第 8 行中的没有参数的构造函数（即默认构造函数 default constructor），而第 22 行的对象 s2 会调用第 13 行的那个带两个参数的构造函数。在第 22 行，我们可以看到，当定义一个对象的时候，后面可以跟参数表。匹配这些参数的构造函数就会被调用。

有两个要注意的地方：如果要调用的是默认构造函数，则不需要写空的参数表，也就是不应写成 21 行的样子。如果我们如 21 行这样写，编译器会把它理解为一个函数的声明。另外，对于 23 行，编译器会报错，因为找不到对应的构造函数。

默认构造函数是一个特殊的构造函数。如果一个类中没有定义任何的构造函数，那么 C++ 将会自动地加入一个函数体为空的默认构造函数。也就是说，如果上例中没有第 8 和 13 行



的构造函数，那么第 20 行的定义方式仍然是正确的。但是，如果只是没有第 8 行的构造函数，但有 13 行的构造函数，那么 C++ 不会自动的加入一个默认构造函数，这时第 20 行的对象定义将是错误的。

另一个特殊的构造函数叫做拷贝构造函数 copy constructor，它的参数只有一个，而且是同一个类型的对象的引用。拷贝构造函数是最特殊的构造函数，它在对象作为函数参数传递的时候会自动地被调用。看下面的例子。

```
1 #include <iostream>
2 using namespace std;
3 #include <cstring>
4
5 class Student
6 {
7 private:
8     int id;
9     char name[20];
10
11 public:
12
13     Student() {
14         id = 10000000;
15         strcpy(name, "Cong Liu");
16     }
17
18     Student(const Student & s2) {
19         id = s2.id;
20         strcpy(name, s2.name);
21     }
22
```

```

23     void print() const {
24         cout << id << endl;
25     }
26
27 };
28
29 void print2(Student & s) {
30     s.print();
31 }
32
33 void print3(Student s) {
34     s.print();
35 }
36
37 int main() {
38     Student s1;
39     Student s2(s1);
40     Student s3 = s1;
41
42     print2(s1);
43     print3(s1);
44 }

```

在 18 行我们定义了一个拷贝构造函数。在 39 行中，我们用对象 s1 构造 s2，这时 s2 的拷贝构造函数被调用。这里，请注意在 C++ 中，40 行的写法，完全等同于 39 行的写法。

另外，在 43 行中，print3 函数被调用；这个时候，43 行中的 s1 对象被用于构造 33 行中的 s 对象。也就是说，在调用函数 print3 之前，C++ 会自动的调用拷贝构造函数，以初始化 print3 参数表中的对象。而在 42 行中的函数 print2 的调用之前，不会有拷贝构造函数的调用，这是因为 print2 的参数是引用，因此它的参数只需要拷贝 42 行中的对象 s1 的地

址。

如果一个类中没有给出拷贝构造函数，那么 C++ 会自动地给我们生成一个拷贝构造函数。这个自动生成的拷贝构造函数会做逐个成员变量的拷贝 *member-wise copy*。对于上例，如果我们不给出拷贝构造函数，那么自动生成的拷贝构造函数能做同样的事情。事实上，在大多数情况下我们不需要给每个类写拷贝构造函数。但是，某些情况下，如我们将会讲到的 C++ 的 `string` 类，我们必须定义更复杂的拷贝构造函数。

C++ 规定，*拷贝构造函数的参数必须是引用*。其实，如果参数不是引用，那么在调用一个拷贝构造函数之前，需要调用同一个拷贝构造函数来初始化他自己的参数。这造成了这个拷贝构造函数的无限循环的调用。

比较一下默认构造函数和拷贝构造函数。如果我们不在类中定义拷贝构造函数，C++ 总会自动生成一个拷贝构造函数。但是，只会在没有定义任何构造函数的情况下才会自动生成默认构造函数。另外，自动生成的默认构造函数是一个空函数，而自动生成的拷贝构造函数不是空函数。

除拷贝构造函数外，其它的只带一个参数的构造函数也是特殊的构造函数。首先在于定义对象的写法。其次是它们可用于类型转换。看下面的例子。

```
1 class Student
2 {
3 private:
4     int id;
5     char name[20];
6
7 public:
8
9     Student(int id2) {
10         id = id2;
11         strcpy(name, "Cong Liu");
12     }
13
14     void print() const {
15         cout << id << endl;
16     }
17
18 };
19
20 void print2(double value) {
21     cout << value << endl;
22 }
23
24 void print3(const Student & s) {
25     s.print();
26 }
27
28 void print4(Student s) {
29     s.print();
30 }
31
32 int main() {
33     Student s1(12340001);
34     Student s2 = 12340001;
35
36     s2 = 12340002;
37     print2(12340003);
38     print3(12340003);
39     print4(12340003);
40 }
```

在上例中的第 9 行，我们有一个用 `int` 构造 `Student` 类的构造函数。首先，在 C++ 中对于仅有一个参数的构造函数，第 33 和 34 行的写法是等同的。其次，有了第 9 行的构造函数，我们就可以把 `int` 类型的变量转化成 `Student` 类的对象，如第 36 行。其实，在 36 行，C++ 首先调用了第 9 行的构造函数，用 `int` 类型的常量 12340002 构造一个 `Student` 类的匿名对象 anonymous object，然后把这个匿名对象的值赋给 `s2`。

另外，C++ 中具有单个参数的构造函数是让我们定义类型转换的途径。对于基础数据类型 `int` 和 `double`，C++ 中已经定义了从 `int` 到 `double` 的转换。因此对于 37 行的函数调用，编译器在找不到 `print2(int)` 的完全匹配后，在 20 行找到了 `print2(double)` 这个近似的匹配。同样地，因为第 9 行的构造函数可用于从 `int` 到 `Student` 的类型转换，对于 38 行的函数调用，编译器在找不到 `print3(int)` 的完全匹配后，在 24 行找到了 `print2(Student)` 这个近似的匹配。

在 38 行中，在调用 24 行的 `print3` 之前，C++ 会调用构造函数 `Student(int)` 生成一个匿名对象，然后传给 `print3`。由于 C++ 规定匿名对象的类型是常量，所以如果去掉 24 行中的 `const`，编译器会报错：不能把常量赋给变量的引用。这是因为 C++ 的引用实际上是地址，所以 `Student *` 是可以赋给 `const`

`Student*` 的；但是反过来是不行的，因为它可能引入错误。

对于 28 行的函数，在 39 行调用前会有两个构造函数的调用：首先是 `Student(int)` 的调用，它用 39 行的整数参数构造一个匿名对象；然后是拷贝构造函数 `Student(Student &)` 的调用，这个函数用匿名对象来构造 `print4` 参数表中的对象 `s`。

最后，我们讲一下析构函数 `destructor` 和构造函数的区别。析构函数只能有一个，它的参数表必须为空。析构函数可以不写，这时 C++ 会自动加入一个函数体为空的析构函数。析构函数只能是共有的，而构造函数可以是私有的，今后我们会看到私有构造函数的用途。

## 4. 从 C 代码到面向对象

在 C 语言里，字符串用字符数组表示。C 字符串并不好用，体现在数组大小固定不能改变，有效字符后面必须以数组 0 结尾。用户必须时刻提醒自己这两点限制，否则程序会出错。

在 C++ 里的库里面定了一个字符串类，它封装了一个动态分配的字符数组，并且提供了一组比较容易使用的成员函数。这使得我们不再需要关心字符串的大小和字符串末尾的特殊数字 0，减少了潜在错误的可能性并且让程序员能更专注重要的问题。

这节课我们看看如何封装一个字符串的类。我们先给出一个由 C 语言实现的字符串处理程序。然后我们用面向对象的程序设计方法把这个程序修改一下。修改以后的程序仍然是 C 语言的。最后们通过把程序改为 C++，展示 C++的封装是如何给面向对象编程提供帮助的。

看下面的程序。

```
1  #include <iostream>
2  using namespace std;
3
4  int length0(const char text[]) {
5      for (int i = 0; ; ++ i) {
6          if (text[i] == '\0') return i;
7      }
8  }
9
10 void copy0(char to[], const char from[]) {
11     for (int i = 0; ; ++ i) {
12         to[i] = from[i];
13         if (from[i] == '\0') break;
14     }
15 }
16
17 int main() {
18     // initialize a 'string' variable
19     int length1 = length0("C L");
20     char * name1 = new char[length1 + 1];
21     copy0(name1, "C L");
22
23     int length2 = length0("L, C");
24     char * name2 = new char[length2 + 1];
25     copy0(name2, "L, C");
26
27     cout << name1 << endl; // C L
28     cout << name2 << endl; // L, C
```

```

29
30     // assign
31     delete [] name2;
32     int length3 = length0("C.");
33     name2 = new char[length3 + 1];
34     copy0(name2, "C.");
35
36     // append
37     int length4 = length0(name2);
38     int length5 = length0(" L");
39     char * temp = new char[length4 + length5 + 1];
40     copy0(temp, name2);
41     copy0(temp + length4, " L");
42     delete [] name2;
43     name2 = temp;
44     cout << name2 << endl; // C. L
45
46     // clean up
47     delete [] name2;
48     delete [] name1;
49 }

```

以上是使用动态内存分配的字符数组来处理字符串操作的一个程序。例子中地址变量 **name1** 和 **name2** 用来存储字符串的地址。第 18 到 25 行分别初始化两个字符串的值。第 31 到 34 行给 **name2** 赋一个新的字符串值。第 37 到 43 行在 **name2** 末尾追加字符串值。第 47 和 48 行释放动态分配的内存空间。

接下来我们把上面的程序修改一下，提高程序的可读性。修改的方法是，把程序中的逻辑层次找出来，然后按照这些层次把程序划分为不同的函数，这个过程也就是



decomposition。这个过程其实更面向对象编程其实是很一致的：我们根据常识定义好程序中使用的数据如何划分为基本的单位，也就是说哪些数据组属于同一个对象，然后把代码划分为对这些数据进行处理的基本单位，也就是要被 decompose 出来的函数。

下面是修改的结果。这里，划分出的对象比较简单，每个对象就是一个字符串地址，我们取名为 `this string`。划分出的函数包括除了 `main` 函数意外的所有函数。

```
17 char * assign0(const char text[]) {
18     int length = length0(text);
19     char * thisString = new char[length + 1];
20     copy0(thisString, text);
21     return thisString;
22 }
23
24 void construct(char * & thisString,
25               const char text[]) {
26     thisString = assign0(text);
27 }
28
29 void assign(char * & thisString,
30            const char text[]) {
31     delete [] thisString;
32     thisString = assign0(text);
33 }
34
35 void append(char * & thisString,
36            const char text[]) {
37     int length1 = length0(thisString);
38     int length2 = length0(text);
39     char * temp = new char[length1 + length2 + 1];
40     copy0(temp, thisString);
41     copy0(temp + length1, text);
42     delete [] thisString;
```

```

43     thisString = temp;
44 }
45
46 void destruct(char * thisString) {
47     delete [] thisString;
48 }
49
50 int main() {
51     // initialize a 'string' variable
52     char * name1 = 0;
53     construct(name1, "Cong Liu");
54
55     char * name2 = 0;
56     construct(name2, "Liu, Cong");
57
58     cout << name1 << endl; // Cong Liu
59     cout << name2 << endl; // Liu, Cong
60
61     assign(name2, "C.");
62
63     append(name2, " Liu");
64     cout << name2 << endl; // C. Liu
65
66     destruct(name2);
67     destruct(name1);
68 }

```

我们现在有一个更好的程序了。main 函数更短，逻辑更加清晰了。如果原来的 main 函数需要变得复杂 10 倍，那么增加的长度将会使得它不可阅读和修改，而修改后的程序由于逻辑清晰和简短，必定使得程序更可控。

但是，我们仍然发现程序有不理想的地方，主要包括两个。首先，第 22 行以前的操作无需被 main 函数直接调用，而且调用它们可能引起程序出错。第二，每一个新创建的字符串

都必须先调用 `construct` 函数, 程序结束时必须调用 `destruct` 函数, 如果忘记可能引起程序出错。

下面我们再用面向对象的 C++ 改写这个程序。在我们了解了 C++ 封装的提供的功能后, 我们或许已经知道应该如何修改了。第一, 我们把用户不需要直接使用的变量和函数定义为私有的。第二, 我们把必须调用的函数变成能自动被调用的构造函数和析构函数。

```
1 class String
2 {
3 private:
4     char * array;
5
6     int length0(const char text[]) const {
7         for (int i = 0; ; ++ i) {
8             if (text[i] == '\0') return i;
9         }
10    }
11
12    void copy0(char to[], const char from[]) {
13        for (int i = 0; ; ++ i) {
14            to[i] = from[i];
15            if (from[i] == '\0') break;
16        }
17    }
18
19    void assign0(const char text[]) {
20        if (array != 0) { // != NULL
21            delete [] array;
22        }
23        int length = length0(text);
24        array = new char[length + 1];
```

```
25         copy0(array, text);
26     }
27
28 public:
29     String() {
30         array = (char *)"";
31         array = new char[1];
32         array[0] = '\\0';
33     }
34
35     String(const char text[]) {
36         array = (char *)text;
37         array = 0; // NULL
38         assign0(text);
39     }
40
41     ~String() {
42         delete [] array;
43     }
44
45     const char * c_str() const {
46         return array;
47     }
48
49     void assign(const char text[]) {
50         assign0(text);
51     }
52
53     void append(const char text[]) {
54         int length1 = length0(array);
55         int length2 = length0(text);
56         int length = length1 + length2 + 1;
57         char * array2 = new char[length];
58         copy0(array2, array);
59         copy0(array2 + length1, text);
60         delete [] array;
61         array = array2;
62     }
63
64 };
65
```

```

66 #include <iostream>
67 using namespace std;
68
69 int main() {
70     String name1("Cong Liu");
71     String name2 = "Liu, Cong";
72     cout << name1.c_str() << endl; // Cong Liu
73     cout << name2.c_str() << endl; // Liu, Cong
74
75     name2.assign("C.");
76     name2.append(" Liu");
77     cout << name2.c_str() << endl; // C. Liu
78 }

```

在第 4 行，有一个私有的成员变量 `array`，它是一个地址变量，用来存储我们实际上使用的 `char array` 的地址。其实，这个类的成员变量只有一个地址，也就是说每个对象的大小只有 4 个字节。而且，所需要用来存储字符的内存空间，都是在程序运行的时候动态分配的，成员变量 `array` 只是用来存储分配所得的内存空间的地址。也就是说，如果对象是一个局部变量，那么成员变量 `array` 在程序的栈空间里，而实际的字符存储在程序的堆空间里。

在程序中，一个对象如果要存储一个更长的字符串，那么，一个新的更长的内存空间将被分配，这个对象的 `array` 将存储这个内存空间的地址，而原来较短的内存空间会被释放掉。因为我们想让不同的字符串对象相互独立，所以不同的对象

的 `array` 存储的是不同的内存空间的地址。这样，当一个对象修改或释放它的 `array` 对应的内存空间的时候，其它的对象才能不受影响。

第 6-26 行定义了 3 个私有的成员函数，前两个比较容易理解。私有成员是类的使用者不需要知道的，因为在类的外面不能使用这些函数，我们可以给它们特殊的名字，如本例使用 0 结尾的名字。函数 `assign0` 的功能是把私有变量 `array` 存储的 `char array` 赋值为给定的 `char array` 的值。

私有变量 `array` 存储的是一个动态分配所得的 `char array` 的地址，而这个 `char array` 的长度不一定跟所赋值的长度一样。所以在 20-22 行中，如果 `array` 已经存储了一个地址，函数 `assign0` 首先把这个地址对应的堆内存空间释放掉。在 23-24 行中，动态分配了一个与参数 `text` 给定的 `char array` 长度相同的内存空间，然后把这个内存空间的地址存入私有变量 `array`。最后，在 25 行把 `text` 的各个字符拷贝入刚才分配的内存空间。

类中有两个构造函数。第一个是第 29 行的默认构造函数，它把对象初始化为一个空的字符串。注意，第 30 行的写法是错误的：因为 `array` 是用来存储一个动态分配的内存空间的地址的，它存储的地址将在对象被释构的时候被释放，如果按照 30 行的写法，到时释放的将是一个常量的地址。正确的做

法，如第 31-32 行，是让地址 `array` 存储一个动态分配的空字符串的地址。

第二个构造函数把对象初始化为一个给定的字符串。第 36 行的写法是错误的：首先是因为这使得字符串的每一个字符都不能被改动；第二是因为这使得，当释放地址 `array` 所存储的内存空间的时候，程序出错。第 37 行把地址 0 赋值给 `array` 是为了告诉函数 `assign0`： `array` 没有存储任何地址，因此不需要释放 `array` 中的内存空间。

第 41-43 行的析构函数，在对象本身的内存空间（4 个字节）被释放前，释放动态分配的、用于存储不定长度的字符串的内存空间。

第 45-47 行的 `c_str` 函数返回对象所封装的字符数组 `char array`。这个函数的目的并不是为了破坏封装，让我们任意操作被封装的 `char array`。它的存在是为了让 `String` 对象可用于旧有的 `char array` 的处理程序。为了尽量保护封装，该函数的在返回的时候把所封装的 `char array` 标示为常量 `const char array`，这使得取得该函数返回值的程序不能直接地改变所封装的 `char array` 中的字符。我们在使用 `String` 对象的时候一般无须使用这个函数。在介绍 `String` 类的输出操作符之前，我们暂时使用 `c_str` 函数是为了获得，然后输出对象的内容。

类中还定义了两个对象函数。函数 `assign`，把对象的值赋为给定的 `char array` 的值。函数 `append` 给原来的对象添加字符串，即赋为原来的值后面加上给定的 `char array` 的值。注意在 `append` 函数中，`array` 中原来存储的 `char array`，不能一开始就释放。也不能一开始就让 `array` 存储别的地址，否则原来存储的 `char array` 的内存空间和它的内容将会丢失。所以，这个函数中使用一个临时的地址变量 `array2` 来存放一个新动态分配的、更大的内存空间。在给新的内存空间中存入添加后的字符串之后，再释放原来的内存空间和把新的内存空间的地址存入 `array`。

我们从 69-78 行的 `main` 函数中对 `String` 对象的使用可以看到封装的好处，它包括：可以任意改变字符串的长度而无须担心数组越界，无须知道动态内存分配的操作，无须担心忘记 `char array` 后面的数字 0 时所产生的错误。

## 5. C++字符串类

C++的字符串类 `string` 是用来封装一个字符数组 `char array` 的。需要设计一个类来封装 `char array` 的具体原因是：

- 因为需要包含字符串长度的信息，`char array` 使用数字 0 表示有意义字符的结束位置。否则，必须由一个额外的整数变量表示长度。这使得用户必须不能忘记在



每次构造或改变 `char array` 的时候给末尾字符赋 0。结果造成写程序的不便和产生错误的隐患。

- 标准 C++ 规定，`char array` 的长度必须是一个编译的时候可以确定的常量。但是，固定长度的字符串，不能满足大部分的程序需要在运行的时候进行的字符串处理，如输入输出，的实际要求：要么浪费空间，要么越界出错。
- 另外一个使用 `char array` 的方法是使用动态内存分配。它能够在需要更多内存空间的时候才分配一个更大的空间，达到节约内存空间的要求。但是，它要求用户必须不能忘记在使用前后分别进行分配与释放。

封装可以完全地解决以上的问题。C++ 的 `string` 类封装了一个 `char array`，并且提供很多操作于这个 `char array` 的公有函数。我们将会看到，`string` 类所提供的公有函数，展示了封装应有的特点：

- 健壮：函数的使用没有“潜规则”。也就是我们不需要记住要在那里赋 0，或要在那里分配与释放内存空间，才能避免落入错误的陷阱。
- 简易：对于同样的字符串处理，使用 `string` 比使用 `char array` 写的程序更简单。
- 完备：`string` 提供的函数必须能让用户完成任何的字符

串处理，如对任意字符的访问和修改，合并两个字符串等。否则，string 不能代替 char array。

在介绍 C++ 中 string 类的常用函数的实现之前，首先介绍它们的使用。在看下面的函数的时候请思考如何实现它们。

```
1 #include <string>
2 #include <iostream>
3 using namespace std;
4
5 int main() {
6     string name1("Cong Liu");
7     string name2 = "C. Liu";
8     cout << name1.c_str() << endl; // Cong Liu
9     cout << name2.c_str() << endl; // C. Liu
10
11     name2.assign("liucong3");
12     name2.append("@mail.sysu.edu.cn");
13     cout << name2.c_str() << endl;
14         // liucong3@mail.sysu.edu.cn
15
16     // MORE
17
18     // operator << (ostream, string)
19     cout << name2 << endl;
20         // liucong3@mail.sysu.edu.cn
21
22     // constructor: string(string)
23     string name3(name1);
24     cout << name3.size() << endl; // 8
25
26     // assign, operator =
27     name2.assign("Cong Liu");
28     name2.assign(name1);
29     name2 = "Cong Liu";
30     name2 = name1;
31     cout << name2 << endl; // Cong Liu
32
33     // operator +
34     name2 = (name1 + " Right");
```

```

35     name2 = ("Left " + name2);
36     cout << name2 << endl;
37     // Left Cong Liu Right
38
39     // operator +=
40     name2 = name1;
41     name2 += "*";
42     name2 += name1;
43     cout << name2 << endl; // Cong Liu*Cong Liu
44
45     // at, operator []
46     cout << name2.at(5) << endl; // L
47     cout << name2[5] << endl; // L
48     name2[4] = '+';
49     name2.at(13) = '-';
50     cout << name2 << endl; // Cong+Liu*Cong-Liu
51
52     // compare, operator >
53     int cmp = name2.compare(name1);
54     cout << cmp << endl; // 1
55     bool cmp2 = (name2 > name1);
56     cout << cmp2 << endl; // 1
57
58     // sub-string
59     name3 = name2.substr(5, 8);
60     cout << name3 << endl; // Liu*Cong
61     int length = name2.size() - 9;
62     cout << name2.substr(9, length) << endl;
63     cout << name2.substr(9) << endl;
64     // Cong-Liu
65
66     // find
67     int index = name2.find("Liu");
68     cout << index << endl; // 5
69     name3 = "Liu";
70     cout << name2.find(name3) << endl; // 5
71     // find all
72     int from = 0;
73     while (true) {
74         int index = name2.find(name3, from);
75         if (index == -1) break;
76         cout << index << endl; // 5 14
77         from = index + 1;
78     }

```

```

79
80     // erase, insert, replace
81     name2.erase(4, 10);
82     cout << name2 << endl; // CongLiu
83     name2.insert(4, "Liu");
84     cout << name2 << endl; // CongLiuLiu
85     name2.replace(4, 3, " ");
86     cout << name2 << endl; // Cong Liu
87
88 }

```

从 6-13 行的程序中我们可以看到，之前的例子中定义的 String 类中的函数在 C++ 的 string 类中都有定义。除此以外，在 19 行，我们可以看到，string 类的对象可以使用输出操作符直接输出到控制台。这时因为 C++ 的类库中定义了左操作数为输出流 ostream 类的对象而右操作数为 string 类的对象的输出操作符。

在 23 行中，我们看到了 string 类的拷贝构造函数。实际上由于 string 类包含一个用于存储地址的成员变量，后面我们会看到，自动生成的拷贝构造函数所执行的 member-wise copy 不能适当地拷贝对象的动态分配的存储空间中的内容，使得 string 类必须定义拷贝构造函数。

在 24 行中的 size 函数返回对象中字符的个数。27-28 行使用了函数 assign，它的隐形对象参数和唯一的一般参数都是 string 的对象。27 对函数 assign 的调用是一个近似匹配：在 assign 的调用前，类型为 char array 的参数会先被构造函数

`string(char[])`转换为一个 `string` 的对象，然后传给函数 `assign` 的参数。29-30 行使用了与函数 `assign` 完全相同的等于号操作符函数 `operator =`。C++会自动地为每个类生成一个等于号操作符函数，出于与拷贝构造函数相同的原因，`string` 类必须定义等于号操作符函数。

34-35 行使用了 `string` 类的加号操作符函数 `operator +`，这个操作符函数的返回值是一个 `string` 类的对象，返回值的内容是左右操作数字符串的连接。注意 `operator +`的其中一个操作符必须是 `string` 类的对象，因为两个 `char array` 的相加是基本数据类型（字符的地址）间的操作符，因此不能定义为操作符函数，而且地址间的相加是 C/C++不允许的。41-42 行使用了与 `append` 函数完全等同的 `operator +=`。

46-49 行使用了函数 `at` 和与它等同的操作符函数 `operator []`。这个操作符号实际上是一个二元操作符：例如，`name2[5]` 的左操作数是 `name2` 而右操作数是 5。函数 `at` 和操作符函数 `operator []`的返回值很特别，是对象的其中一个字符的引用。这使得通过函数 `at` 和操作符函数 `operator []`的返回值，我们既可以读取对象中字符的值，还可以改变对象中字符的值。

53 行的 `compare` 函数返回 -1，0，或 1。它们分别表示字典序中的 <，=，> 关系。除此以外，`string` 类还定义了返回 `bool` 值的操作符函数 >，>=，<，<=，!=，和 ==。

59-62 行的函数 `substr` 的返回值是一个 `string` 的对象，返回值的内容是一个子字符串。函数 `substr` 的两个参数分别表示所返回的子字符串在原字符串（函数 `substr` 的隐形对象参数）中的开始位置和子字符串的长度。如果子字符串一直延伸到原字符串的末尾，那么可以使用只带一个参数的函数 `substr`。所以，62 和 63 行的输出是完全相同的。

67 行的函数 `find` 返回参数在隐形对象参数中出现的第一个位置，如果不出现则返回-1。另一个函数 `find` 具有多一个参数，这个参数指定查找的开始位置：如果被查找的字串不在隐形对象参数中或者在查找的开始位置之前，那么函数 `find` 会返回-1。72-78 行的程序利用这个函数 `find` 找出了对象 `name3` 在对象 `name2` 中出现的所有位置。

81 行的函数 `erase`，可以删除隐形对象参数中的一些连续字符，`erase` 的参数表示这些字符的开始位置和个数。82 行的函数 `insert` 在隐形对象参数的指定位置插入给定的字串。85 行的函数 `replace` 把隐形对象参数的指定的子串替换为给定的字串。我们将会看到，函数 `replace` 可以代替函数 `erase` 和函数 `insert`。

## 6. C++字符串类的实现

下面我们开始介绍上述 `string` 类中的常用函数的实现。

C++的 `string` 类实现非常复杂，我们将实现一个比较简单的 `String` 类。注意，为了简化，`String` 类和 `string` 类的实现会有细微的差别。

首先，我们先定义一个私有的函数，它比较两个 `char array` 在字典序中的大小，在字典序中靠前的字符串较小。

```
33     int compare0(const char a1[],  
34                 const char a2[], int max) const {  
35  
36         for (int i = 0; i < max; ++ i) {  
37             if (a1[i] == 0 && a2[i] == 0)  
38                 return 0;  
39             if (a1[i] > a2[i]) return 1;  
40             if (a1[i] < a2[i]) return -1;  
41         }  
42         return 0;  
43     }
```

如果第一个 `char array` 小于第二个 `char array`，则 `compare0` 将返回-1，等于则返回 0，大于则返回 1。下面是返回字符个数的公有对象函数 `size`。

```
64     int size() const {  
65         return length0(array);  
66     }
```

以下我们定义 `String` 类的等于号操作符函数。这个函数包含了一些要注意的地方和新的知识点。首先，在每个类里，如果我们没有定义右操作数都是该类对象的等于号操作符函数，C++会自动生成一个，它会做 `member-wise copy`。因为 `String` 类的成员变量包含地址变量，所以我们必须为 `String` 类定义一个 `operator =`。

对于 `operator =`，C++并没有规定右操作数和返回值的类型，但是为了提高程序的可靠性，我们最好把右操作数和返回值的类型都定义为所在类的引用，即 `String &`，原因如下。因为对于一个变量或对象 `s`，C 语言允许这样的写法 `(s = 1) = 2`，这表明 `s = 1`，的返回值需要是 `s` 本身，即 `s` 自己的引用。不是每一个人都喜欢 C 语言的灵活性，但作为标准 C++库中的一员，`String` 类需要尽量地满足所有用户的使用偏好。

这里有一个问题：如果我们需要类型转化，如需要把一个 `int` 赋值给一个 `String`，那么我们需要实现 `String::operator = (int)` 吗？答案是实现构造函数 `String(int)` 而不是 `String::operator = (int)`。这是只要有了 `String(int)`，`operator =(const String&)` 函数的参数就能匹配 `int` 类型的变量。在调用 `operator =(const String &)` 时，如果参数是 `int`，那么 C++会先调用 `String(int)`，得到一个 `String` 的临时对象（类型为常量），再把它最为 `operator =(const String&)` 的参数。那么，我们应



避免不必要的函数了：String::operator = (int)。同样地，为了使用 String text("Hi"); text = "Hello",我们需要的是 String(const char []),而不是 operator =(const char [])。所以一般情况下，对于一个类，我们只需要一个赋值操作符。

```
84     String & operator = (const String & str) {  
85         assign0(str.array);  
86         return (*this);  
87     }
```

---

```
24     void assign0(const char text[]) {  
25         if (array != 0) { // != NULL  
26             delete [] array;  
27         }  
28         int length = length0(text);  
29         array = new char[length + 1];  
30         copy0(array, text, length + 1);  
31     }
```

注意，在 operator =里，我们不能把右操作数对象的成员 array 直接赋值给隐形对象参数的 array，因为这使得两个 String 的对象共有同一个动态分配的 char array：这将在改变一个的对象的某些字符时候同时改变了另一个对象的这些字符，而且这会在释构一个对象的时候释放这个共有的 char array。由于隐形对象参数已经构造，在 26 行中我们首先释放它当前所占有的、动态分配的 char array。在 29 行中，我们

给隐形对象参数分配新的、适当大小的 char array。然后，在 30 行，我们字串的值复制到这个新的 char array 里。

如果类的对象拥有资源（如文件，内存空间，运算资源，和网络资源），而简单拷贝（浅拷贝）不可以拷贝资源，那么就需要在类中提供拷贝构造函数，而不能依赖 C++ 给我们生成。提供资源拷贝的特殊方法，也恰恰是拷贝构造函数存在的必要性。

在 86 行中，我们返回隐形对象参数（即左操作数）的引用。返回的是引用，这是因为返回值的类型是引用 String &。

```
89     String & operator += (const String & str) {  
90         append(str.array);  
91         return (*this);  
92     }
```

第 89 行的 operator += 函数等同于 append 函数。同样地，由于 C 语言与许这样的写法(s += 1) += 2，operator += 函数因该返回左操作数（该对象函数的隐形对象参数）的引用。

```
99     String operator + (const String & str2)  
100     const {  
101         String str(*this);  
102         str += str2.array;  
103         return str;  
104     }
```

与符函数 `operator +=` 相同，操作符函数 `operator +` 的右操作数与返回值的类型也是可以任意指定的。把右操作数的类型设为 `const String &` 较为合适，这使得该参数能是常量类型，从而也能是可被转化为 `String` 的其他类型（反之不行，因为临时变量是常量）。

注意 `operator +` 的返回值不能是引用，因为它返回的是局部变量的值；而且 `operator +` 应该是一个常量对象函数，因为传统上的加法不应改变左操作数和右操作数。

```
106     char & at(int index) const {  
107         return array[index];  
108     }  
109  
110     char & operator [] (int index) const {  
111         return array[index];  
112     }
```

上面两个函数返回字符串中的一个字符的引用。注意，如果返回的是字符，调用该函数的程序只能读取这个字符的值；但如果返回的是字符的引用，调用该函数的程序既能读取这个字符的值，又能改变字符串（隐藏对象参数）中该字符的值。

```
114     int compare(const String & str) const {  
115         return compare0(array, str.array,  
116                         2147483647);  
117     }  
118  
119     bool operator > (const String & str)  
120     const {  
121         return compare(str) > 0;
```

上面两个是字符串比较函数。函数 `compare0` 的第三个参数是比较的最大字符数 `max`，如果两个字符串开始的 `max` 个字符都相同，则认为它们相同。函数 `compare` 中把 2147483647（`int` 的最大值）传给 `compare0` 实际上表示不设定比较的最大字符数。

```

124     String substr(int start) const {
125         String str(array + start);
126         return str;
127     }
128
129     String substr(int start, int length)
130     const {
131         String str(array + start);
132         str.array[length] = '\0';
133         return str;
134     }

```

函数 `substr` 返回（隐藏对象参数的）一个子串。

```

136     int find(const String & str, int start)
137     const {
138
139         int length = length0(array);
140         int length2 = length0(str.array);
141         int end = length - length2;
142         for (int i = start; i <= end; ++ i) {
143             if (compare0(array + i, str.array,
144                         length2) == 0) {
145                 return i;
146             }
147         }
148         return -1;

```

```

149     }
150
151     int find(const String & str) const {
152         return find(str.array, 0);
153     }

```

函数 `String::find(String, int)` 返回参数（在隐藏对象参数中）出现的位置。它在（隐藏对象参数）的各个位置上检验待查找的字串 `str` 是否存在。函数 `String::find(String)` 直接调用了 `String::find(String, int)`。

```

155     void replace(int start, int length,
156                 const String & str) {
157
158         String str1 = substr(0, start);
159         String str2 = substr(start + length);
160         assign(str1);
161         append(str);
162         append(str2);
163     }
164
165     void erase(int start, int length) {
166         replace(start, length, "");
167     }
168
169     void insert(int index, const String & str) {
170         replace(index, 0, str);
171     }

```

函数 `replace` 首先使用 `str1` 和 `str2` 记录了（隐藏对象参数中）被替换字符串前面的部分和后面部分，接着（把隐藏对象参数）重新复制为 `str1`，然后添加替换字符串 `str`，最后添加 `str2`。函数 `erase` 和 `insert` 是 `replace` 的特殊形式，它们直接调用了

replace。

在上面的函数实现中，我们多次重用了代码，如拷贝构造函数和等于号操作符函数都重用了函数 `assign0`，函数 `erase` 和 `insert` 重用了 `replace`。这看上去是一种懒惰的做法，代价是程序的执行速度。代码重用的好处包括：减少程序错误，增加程序的可读性，降低程序修改的难度，减少开发时间。一般的程序对于程序的执行速度要求不高。对程序执行速度斤斤计较的程序一般也不使用封装以减少封装引入的开销。

按照使用习惯，输出操作符的左操作数是输出流的对象，右操作数是待输出的对象。由于左操作数不是 `String` 类的对象，输出操作符函数不能是对象函数。输出操作符是左结合的操作符，注意只有赋值操作符（包括 `=`，`+=` 等等）是右结合的。输出操作符的连用，如 `cout << str << endl;` 是非常普遍的，它等效于 `cout << str; cout << endl;`。因此，要能使输出操作符的连用，输出操作符函数的返回值必须是左操作数本身，即 `cout`。这样 `cout << str` 的运算完后得到的又是 `cout`，然后得到的 `cout` 可以继续进行后面的 `cout << endl` 运算。

```
175 ostream & operator << (ostream & out,  
176                        const String & str) {  
177  
178     out << str.c_str();  
179     return out;  
180 }
```

但是，这个函数不能返回 `cout`，因为输出操作符的左操作数有时可能不是 `cout`。实际上 `cout` 的类型并不是 `stream`，而是 `ostream` 的子类(子类这个概念将在介绍继承的时候涉及)。当我们讲到多态性的时候，我们会知道上例的对象 `out` 可以存储多种对象，包括文件输出流对象和全局的控制台输出对象 `cout`。在写程序的时候我们不知道 `out` 是一个文件还是一个控制台，所以左操作数和右操作数都必须引用。由于输出操作符是改变左操作数的，如改变文件的输入计数器，所以左操作数和返回值不能是常量。上面提到的继承，多态性，和这章的主题封装，是面向对象语言中的 3 个重要的特性。

因为我们在第 35 行有定义了可用于类型转换的构造函数 `String(const char text[])`，使得 218 行的加法函数调用 `name1 + " Right"` 可以近似地匹配为 99 行的加法操作符的对象函数 `String::operator + (const String &)`。但是 C++ 规定，对象函数的隐形对象参数不能参与近似地匹配，也就是说 219 行的加法函数调用 `"Left " + name2` 的左操作数不能通过类型转化使得该加法函数调用不能近似地匹配为 99 行的加法操作符函数。为了 219 行的加法函数调用，我们要再定义一个加法操作符函数，看下面 182 行的例子。

```
218     name2 = (name1 + " Right");  
219     name2 = ("Left " + name2);
```

---

```
182 String operator + (const char text[],
183                     const String & str2) {
184
185     String str(text);
186     return str + str2;
187 }
```

注意，这里我们不把 182 行的函数定义为 `operator + (const String &, const String &)`。假设我们定义了这个操作符函数，第 218 行的操作符函数调用会是一个有歧义的函数调用，它可以及时匹配第 99 行的和我们假设的操作符函数。

因此，或许另外一种好的选择是把操作符函数定义在类之外。这样我们就无需定义多个相同功能的 `operator +` 了。

上面，我们通过 `String` 类的例子，介绍了封装这个重要的概念。我们为 `String` 类中定义了完备的函数，使得它可以代替被封装的基本数据类型 `char array`。有了 `String` 类，我们就可以避免使用 `char array` 时所需要注意的繁琐细节，还有由于它的不当使用可能引入的程序错误。

## 习题

1. 用自己的话解释下面的概念



封装

常量函数

默认构造函数

## 2. 用自己的回答下面的问题

公有成员跟私有成员的区别是什么？

封装的目标是什么？

C++中封装如何实现？

不能对常量对象执行那些操作？

不能在常量函数例执行那些操作？

C++的对象如何调用构造函数？

C++中有那集中特殊的构造函数，它们如何特殊？

## 3. 编写一个 Matrix 类

编写一个 Matrix 类，使以下的主程序具有给定的输出。

```
1 void printMatrix(Matrix matrix) {  
2     matrix.print();  
3 }  
4  
5 int main() {  
6     cout << "constructor 1" << endl;  
7     Matrix matrix1(3, 3);  
8     matrix1.print();  
9  
10    const double values2[] = {  
11        1, 2, 3,  
12        4, 5, 6,  
13        7, 8, 9,  
14    };
```

```
15     cout << "constructor 2" << endl;
16     Matrix matrix2(3, 3, values2);
17     matrix2.print();
18
19     cout << "copy constructor" << endl;
20     Matrix matrix3 = matrix2;
21     printMatrix(matrix3);
22
23     cout << "operator =" << endl;
24     matrix3.set(1, 1, 10.0);
25     matrix3 = matrix2;
26     matrix3.print();
27
28     cout << "getColumn" << endl;
29     matrix2.getColumn(2).print();
30     cout << "getRow" << endl;
31     matrix2.getRow(2).print();
32
33     cout << "concatenateRows" << endl;
34     matrix1.concatenateRows(matrix2).print();
35     cout << "concatenateColumns" << endl;
36     matrix1.concatenateColumns(matrix2).print();
37
38     cout << "reshape" << endl;
39     matrix1.concatenateColumns(matrix2).
40         reshape(6, 3).print();
41
42     cout << "transpose" << endl;
43     matrix2.transpose().print();
44
45     cout << "operator +" << endl;
46     (matrix2 + matrix2).print();
47     cout << "operator +" << endl;
48     (matrix2 + 10).print();
49     cout << "operator -" << endl;
50     (matrix2.transpose() - matrix2).print();
51     cout << "operator -" << endl;
52     (matrix2 - 10).print();
53
54     cout << "operator *" << endl;
55     (matrix2.transpose() * matrix2).print();
56     cout << "operator *" << endl;
57     (matrix2 * 2).print();
58
```

```

59     cout << "max" << endl;
60     cout << matrix2.max().max().get(1, 1) << endl;
61     cout << "min" << endl;
62     cout << matrix2.min().min().get(1, 1) << endl;
63     cout << "sum" << endl;
64     cout << matrix2.sum().sum().get(1, 1) << endl;
65
66     cout << "pow" << endl;
67     matrix2.pow(2).print();
68     cout << "exp" << endl;
69     matrix2.exp().print();
70     cout << "log" << endl;
71     matrix2.log().print();
72     cout << "abs" << endl;
73     (matrix2 - 5).abs().print();
74
75     const double value4[] = { 1, 2, 3, 4, 5 };
76     Matrix matrix4(1, 5, value4);
77     cout << "norm" << endl;
78     cout << matrix4.norm(1) << endl;
79     cout << matrix4.norm(2) << endl;
80
81 }

```

## 输入样本

constructor 1

```

0    0    0
0    0    0
0    0    0

```

constructor 2

```

1    2    3
4    5    6
7    8    9

```

copy constructor

```

1    2    3
4    5    6
7    8    9

```

operator =

```

1    2    3
4    5    6
7    8    9

```

getColumn

2

5

8

getRow

4

5

6

concatenateRows

0

0

0

0

0

0

0

0

0

1

2

3

4

5

6

7

8

9

concatenateColumns

0

0

0

1

2

3

0

0

0

4

5

6

0

0

0

7

8

9

reshape

0

0

2

0

0

5

0

0

8

0

1

3

0

4

6

0

7

9

transpose

1

4

7

2

5

8

3

6

9

operator +

2

4

6

8

10

12

14

16

18

operator +

11

12

13

14

15

16

17

18

19

operator -

0

2

4

-2

0

2

-4

-2

0

operator -

-9

-8

-7

-6

-5

-4

-3

-2

-1

operator \*

|    |     |     |
|----|-----|-----|
| 66 | 78  | 90  |
| 78 | 93  | 108 |
| 90 | 108 | 126 |

operator \*

|    |    |    |
|----|----|----|
| 2  | 4  | 6  |
| 8  | 10 | 12 |
| 14 | 16 | 18 |

max

9

min

1

sum

45

pow

|    |    |    |
|----|----|----|
| 1  | 4  | 9  |
| 16 | 25 | 36 |
| 49 | 64 | 81 |

exp

|         |         |         |
|---------|---------|---------|
| 2.71828 | 7.38906 | 20.0855 |
| 54.5982 | 148.413 | 403.429 |
| 1096.63 | 2980.96 | 8103.08 |

log

|         |          |         |
|---------|----------|---------|
| 0       | 0.693147 | 1.09861 |
| 1.38629 | 1.60944  | 1.79176 |
| 1.94591 | 2.07944  | 2.19722 |

abs

|   |   |   |
|---|---|---|
| 4 | 3 | 2 |
| 1 | 0 | 1 |
| 2 | 3 | 4 |

norm

15

7.4162