

Lesson08---string类

【本节目标】

- 1. 为什么要学习string类
- 2. 标准库中的string类
- 3. string类的模拟实现
- 4. 扩展阅读

1. 为什么学习string类?

1.1 C语言中的字符串

C语言中，字符串是以'\0'结尾的一些字符的集合，为了操作方便，C标准库中提供了一些str系列的库函数，但是这些库函数与字符串是分离开的，不太符合OOP的思想，而且底层空间需要用户自己管理，稍不留神可能还会越界访问。

1.2 两个面试题(暂不做讲解)

[字符串转整形数字](#)

[字符串相加](#)

在OJ中，有关字符串的题目基本以string类的形式出现，而且在常规工作中，为了简单、方便、快捷，基本都使用string类，很少有人去使用C库中的字符串操作函数。

2. 标准库中的string类

2.1 string类(了解)

[string类的文档介绍](#)

1. 字符串是表示字符序列的类
2. 标准的字符串类提供了对此类对象的支持，其接口类似于标准字符容器的接口，但添加了专门用于操作单字节字符串的设计特性。
3. string类是使用char(即作为它的字符类型，使用它的默认char_traits和分配器类型(关于模板的更多信息，请参阅basic_string)。
4. string类是basic_string模板类的一个实例，它使用char来实例化basic_string模板类，并用char_traits和allocator作为basic_string的默认参数(根于更多的模板信息请参考basic_string)。
5. 注意，这个类独立于所使用的编码来处理字节:如果用来处理多字节或变长字符(如UTF-8)的序列，这个类的所有成员(如长度或大小)以及它的迭代器，将仍然按照字节(而不是实际编码的字符)来操作。

总结:

1. string是表示字符串的字符串类
2. 该类的接口与常规容器的接口基本相同，再添加了一些专门用来操作string的常规操作。

- 3. string在底层实际是：basic_string模板类的别名，typedef basic_string<char, char_traits, allocator> string;
- 4. 不能操作多字节或者变长字符的序列。

在使用string类时，必须包含#include头文件以及using namespace std;

2.2 string类的常用接口说明（注意下面我只讲解最常用的接口）

1. string类对象的常见构造

(constructor)函数名称	功能说明
string()（重点）	构造空的string类对象，即空字符串
string(const char* s)（重点）	用C-string来构造string类对象
string(size_t n, char c)	string类对象中包含n个字符c
string(const string&s)（重点）	拷贝构造函数

```
1 void Teststring()
2 {
3     string s1;           // 构造空的string类对象s1
4     string s2("hello bit"); // 用C格式字符串构造string类对象s2
5     string s3(s2);       // 拷贝构造s3
6 }
```

2. string类对象的容量操作

函数名称	功能说明
size（重点）	返回字符串有效字符长度
length	返回字符串有效字符长度
capacity	返回空间总大小
empty（重点）	检测字符串释放为空串，是返回true，否则返回false
clear（重点）	清空有效字符
reserve（重点）	为字符串预留空间**
resize（重点）	将有效字符的个数该成n个，多出的空间用字符c填充

```
1 // size/clear/resize
2 void Teststring1()
3 {
4     // 注意：string类对象支持直接用cin和cout进行输入和输出
5     string s("hello, bit!!!");
```

```

6      cout << s.size() << endl;
7      cout << s.length() << endl;
8      cout << s.capacity() << endl;
9      cout << s << endl;
10
11     // 将s中的字符串清空, 注意清空时只是将size清0, 不改变底层空间的大小
12     s.clear();
13     cout << s.size() << endl;
14     cout << s.capacity() << endl;
15
16     // 将s中有效字符个数增加到10个, 多出位置用'a'进行填充
17     // "aaaaaaaaaa"
18     s.resize(10, 'a');
19     cout << s.size() << endl;
20     cout << s.capacity() << endl;
21
22     // 将s中有效字符个数增加到15个, 多出位置用缺省值'\0'进行填充
23     // "aaaaaaaaa\0\0\0\0\0"
24     // 注意此时s中有效字符个数已经增加到15个
25     s.resize(15);
26     cout << s.size() << endl;
27     cout << s.capacity() << endl;
28     cout << s << endl;
29
30     // 将s中有效字符个数缩小到5个
31     s.resize(5);
32     cout << s.size() << endl;
33     cout << s.capacity() << endl;
34     cout << s << endl;
35 }
36
37 //=====
38 void Teststring2()
39 {
40     string s;
41     // 测试reserve是否会改变string中有效元素个数
42     s.reserve(100);
43     cout << s.size() << endl;
44     cout << s.capacity() << endl;
45
46     // 测试reserve参数小于string的底层空间大小时, 是否会将空间缩小
47     s.reserve(50);
48     cout << s.size() << endl;
49     cout << s.capacity() << endl;
50 }
51
52 // 利用reserve提高插入数据的效率, 避免增容带来的开销
53 //=====
54 void TestPushBack()
55 {
56     string s;

```

```

57     size_t sz = s.capacity();
58     cout << "making s grow:\n";
59     for (int i = 0; i < 100; ++i)
60     {
61         s.push_back('c');
62         if (sz != s.capacity())
63         {
64             sz = s.capacity();
65             cout << "capacity changed: " << sz << '\n';
66         }
67     }
68 }
69
70 void TestPushBackReserve()
71 {
72     string s;
73     s.reserve(100);
74     size_t sz = s.capacity();
75
76     cout << "making s grow:\n";
77     for (int i = 0; i < 100; ++i)
78     {
79         s.push_back('c');
80         if (sz != s.capacity())
81         {
82             sz = s.capacity();
83             cout << "capacity changed: " << sz << '\n';
84         }
85     }
86 }

```

注意：

1. `size()`与`length()`方法底层实现原理完全相同，引入`size()`的原因是为了与其他容器的接口保持一致，一般情况下基本都是用`size()`。
2. `clear()`只是将string中有效字符清空，不改变底层空间大小。
3. `resize(size_t n)`与`resize(size_t n, char c)`都是将字符串中有效字符个数改变到n个，不同的是当字符个数增多时：`resize(n)`用0来填充多出的元素空间，`resize(size_t n, char c)`用字符c来填充多出的元素空间。注意：`resize`在改变元素个数时，如果是将元素个数增多，可能会改变底层容量的大小，如果是将元素个数减少，底层空间总大小不变。
4. `reserve(size_t res_arg=0)`：为string预留空间，不改变有效元素个数，当`reserve`的参数小于string的底层空间总大小时，`reserve`不会改变容量大小。

3. string类对象的访问及遍历操作

函数名称	功能说明
operator[] (重点)	返回pos位置的字符, const string类对象调用
begin + end	begin获取一个字符的迭代器 + end获取最后一个字符下一个位置的迭代器
rbegin + rend	begin获取一个字符的迭代器 + end获取最后一个字符下一个位置的迭代器
范围for	C++11支持更简洁的范围for的新遍历方式

```

1 void Teststring()
2 {
3     string s1("hello Bit");
4     const string s2("Hello Bit");
5     cout<<s1<<" "<<s2<<endl;
6     cout<<s1[0]<<" "<<s2[0]<<endl;
7
8     s1[0] = 'H';
9     cout<<s1<<endl;
10
11     // s2[0] = 'h'; 代码编译失败, 因为const类型对象不能修改
12 }
13
14 void Teststring()
15 {
16     string s("hello Bit");
17     // 3种遍历方式:
18     // 需要注意的以下三种方式除了遍历string对象, 还可以遍历是修改string中的字符,
19     // 另外以下三种方式对于string而言, 第一种使用最多
20     // 1. for+operator[]
21     for(size_t i = 0; i < s.size(); ++i)
22         cout<<s[i]<<endl;
23
24     // 2.迭代器
25     string::iterator it = s.begin();
26     while(it != s.end())
27     {
28         cout<<*it<<endl;
29         ++it;
30     }
31
32     string::reverse_iterator rit = s.rbegin();
33     while(rit != s.rend())
34         cout<<*rit<<endl;
35
36     // 3.范围for
37     for(auto ch : s)

```

```

38     cout<<ch<<endl;
39 }

```

4. string类对象的修改操作

函数名称	功能说明
push_back	在字符串后尾插字符c
append	在字符串后追加一个字符串
operator+= (重点)	在字符串后追加字符串str
c_str (重点)	返回C格式字符串
find + npos (重点)	从字符串pos位置开始往后找字符c，返回该字符在字符串中的位置
rfind	从字符串pos位置开始往前找字符c，返回该字符在字符串中的位置
substr	在str中从pos位置开始，截取n个字符，然后将其返回

```

1  void Teststring()
2  {
3      string str;
4      str.push_back(' '); // 在str后插入空格
5      str.append("hello"); // 在str后追加一个字符"hello"
6      str += 'b'; // 在str后追加一个字符'b'
7      str += "it"; // 在str后追加一个字符串"it"
8      cout<<str<<endl;
9      cout<<str.c_str()<<endl; // 以C语言的方式打印字符串
10
11     // 获取file的后缀
12     string file1("string.cpp");
13     size_t pos = file.rfind('.');
14     string suffix(file.substr(pos, file.size()-pos));
15     cout << suffix << endl;
16
17     // npos是string里面的一个静态成员变量
18     // static const size_t npos = -1;
19
20     // 取出url中的域名
21     string url("http://www.cplusplus.com/reference/string/string/find/");
22     cout << url << endl;
23     size_t start = url.find("://");
24     if (start == string::npos)
25     {
26         cout << "invalid url" << endl;
27         return;
28     }
29     start += 3;
30     size_t finish = url.find('/', start);
31     string address = url.substr(start, finish - start);

```

```

32     cout << address << endl;
33
34     // 删除url的协议前缀
35     pos = url.find("://");
36     url.erase(0, pos+3);
37     cout<<url<<endl;
38 }

```

注意:

1. 在string尾部追加字符时, s.push_back(c) / s.append(1, c) / s += 'c'三种的实现方式差不多, 一般情况下string类的+=操作作用的比较多, +=操作不仅可以连接单个字符, 还可以连接字符串。
2. 对string操作时, 如果能够大概预估到放多少字符, 可以先通过reserve把空间预留好。

5. string类非成员函数

函数	功能说明
operator+	尽量少用, 因为传值返回, 导致深拷贝效率低
operator>> (重点)	输入运算符重载
operator<< (重点)	输出运算符重载
getline (重点)	获取一行字符串
relational operators (重点)	大小比较

上面的几个接口大家了解一下, 下面的OJ题目中会有一些体现他们的使用。string类中还有一些其他的操作, 这里不一一列举, 大家在需要用到时不明白了查文档即可。

6. 牛刀小试

[仅仅反转字母](#)

```

1  class Solution {
2  public:
3      string reverseOnlyLetters(string S) {
4          char* pLeft = (char*)S.c_str();
5          char* pRight = pLeft + (S.size()-1);
6
7          while(pLeft < pRight)
8          {
9              // 从前往后找, 找到一个字母
10             while(pLeft < pRight)
11             {
12                 // 找到有效字母后停下来
13                 if(isalpha(*pLeft))
14                     break;
15
16                 ++pLeft;
17             }
18
19             // 从后往前找, 找一个字母

```

```

20         while(pLeft < pRight)
21         {
22             // 找到有效字母后停下来
23             if(isalpha(*pRight))
24                 break;
25
26             --pRight;
27         }
28
29         if(pLeft < pRight)
30         {
31             swap(*pLeft, *pRight);
32             ++pLeft;
33             --pRight;
34         }
35     }
36
37     return S;
38 }
39 };

```

找字符串中第一个只出现一次的字符

```

1  class Solution {
2  public:
3      int firstUniqChar(string s) {
4
5          // 统计每个字符出现的次数
6          int count[256] = {0};
7          int size = s.size();
8          for(int i = 0; i < size; ++i)
9              count[s[i]] += 1;
10
11         // 按照字符次序从前往后找只出现一次的字符
12         for(int i = 0; i < size; ++i)
13             if(1 == count[s[i]])
14                 return i;
15
16         return -1;
17     }
18 };

```

字符串里面最后一个单词的长度--课堂练习

```

1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  int main()
6  {
7      string line;

```



```

8      // 不要使用cin>>line,因为它遇到空格就结束了
9      // while(cin>>line)
10     while(getline(cin, line))
11     {
12         size_t pos = line.rfind(' ');
13         cout<<line.size()-pos-1<<endl;
14     }
15     return 0;
16 }

```

验证一个字符串是否是回文

```

1  class Solution {
2  public:
3      bool isLetterOrNumber(char ch)
4      {
5          return (ch >= '0' && ch <= '9')
6              || (ch >= 'a' && ch <= 'z')
7              || (ch >= 'A' && ch <= 'Z');
8      }
9
10     bool isPalindrome(string s) {
11         // 先小写字母转换成大写, 再进行判断
12         for(auto& ch : s)
13         {
14             if(ch >= 'a' && ch <= 'z')
15                 ch -= 32;
16         }
17
18         int begin = 0, end = s.size()-1;
19         while(begin < end)
20         {
21             while(begin < end && !isLetterOrNumber(s[begin]))
22                 ++begin;
23
24             while(begin < end && !isLetterOrNumber(s[end]))
25                 --end;
26
27             if(s[begin] != s[end])
28             {
29                 return false;
30             }
31             else
32             {
33
34                 ++begin;
35                 --end;
36             }
37         }
38
39         return true;
40     }

```

```
41     };
```

字符串相加

```
1  class Solution {
2  public:
3      string addstrings(string num1, string num2)
4      {
5          // 从后往前相加，相加的结果到字符串可以使用insert头插
6          // 或者+=尾插以后再reverse过来
7          int end1 = num1.size()-1;
8          int end2 = num2.size()-1;
9          int value1 = 0, value2 = 0, next = 0;
10         string addret;
11         while(end1 >= 0 || end2 >= 0)
12         {
13             if(end1 >= 0)
14                 value1 = num1[end1--]-'0';
15             else
16                 value1 = 0;
17
18             if(end2 >= 0)
19                 value2 = num2[end2--]-'0';
20             else
21                 value2 = 0;
22
23             int valueret = value1 + value2 + next;
24             if(valueret > 9)
25             {
26                 next = 1;
27                 valueret -= 10;
28             }
29             else
30             {
31                 next = 0;
32             }
33
34             //addret.insert(addret.begin(), valueret+'0');
35             addret += (valueret+'0');
36         }
37
38         if(next == 1)
39         {
40             //addret.insert(addret.begin(), '1');
41             addret += '1';
42         }
43
44         reverse(addret.begin(), addret.end());
45         return addret;
46     }
47 };
```

1. [课后作业练习——翻转字符串II：区间部分翻转--课后作业](#)
2. [课后作业练习——翻转字符串III：翻转字符串中的单词--课后作业](#)
3. [课后作业练习——字符串相乘](#)
4. [课后作业练习——找出字符串中第一个只出现一次的字符](#)

3. string类的模拟实现

3.1 经典的string类问题

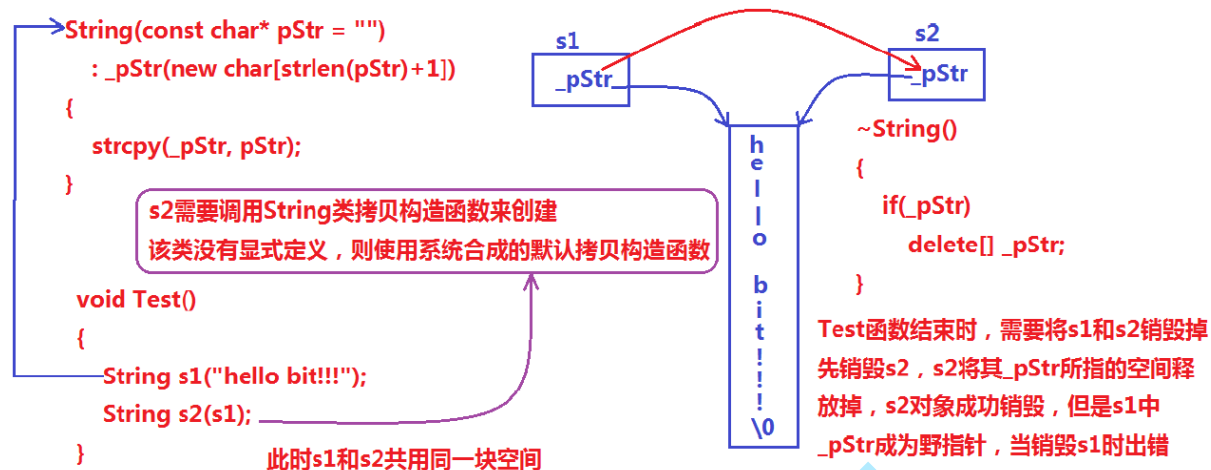
上面已经对string类进行了简单的介绍，大家只要能够正常使用即可。在面试中，面试官总喜欢让学生自己来模拟实现string类，最主要是实现string类的构造、拷贝构造、赋值运算符重载以及析构函数。大家看下以下string类的实现是否有问题？

```
1  class string
2  {
3  public:
4      /*string()
5          :_str(new char[1])
6          {_str = '\0';}
7      */
8      //string(const char* str = "\0") 错误示范
9      //string(const char* str = nullptr) 错误示范
10     string(const char* str = "")
11     {
12         // 构造string类对象时，如果传递nullptr指针，认为程序非法，此处断言下
13         if(nullptr == str)
14         {
15             assert(false);
16             return;
17         }
18
19         _str = new char[strlen(str) + 1];
20         strcpy(_str, str);
21     }
22
23     ~string()
24     {
25         if(_str)
26         {
27             delete[] _str;
28             _str = nullptr;
29         }
30     }
31
32 private:
33     char* _str;
34 };
35
36 // 测试
37 void Teststring()
38 {
```

```

39     string s1("hello bit!!!");
40     string s2(s1);
41 }

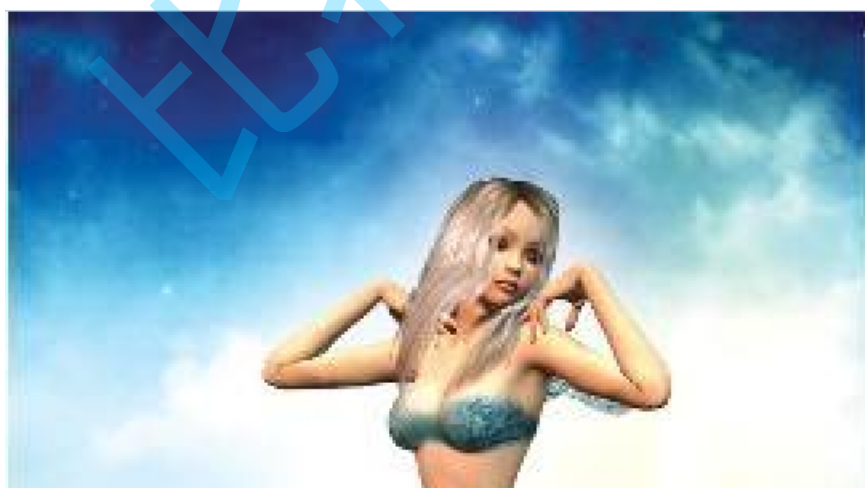
```



说明: 上述string类没有显式定义其拷贝构造函数与赋值运算符重载, 此时编译器会合成默认的, 当用s1构造s2时, 编译器会调用默认的拷贝构造。最终导致的问题是, s1、s2共用同一块内存空间, 在释放时同一块空间被释放多次而引起程序崩溃, 这种拷贝方式, 称为浅拷贝。

3.2 浅拷贝

浅拷贝: 也称位拷贝, 编译器只是将对象中的值拷贝过来。如果对象中管理资源, 最后就会导致多个对象共享同一份资源, 当一个对象销毁时就会将该资源释放掉, 而此时另一些对象不知道该资源已经被释放, 以为还有效, 所以 当继续对资源进项操作时, 就会发生发生了访问违规。要解决浅拷贝问题, C++中引入了深拷贝。

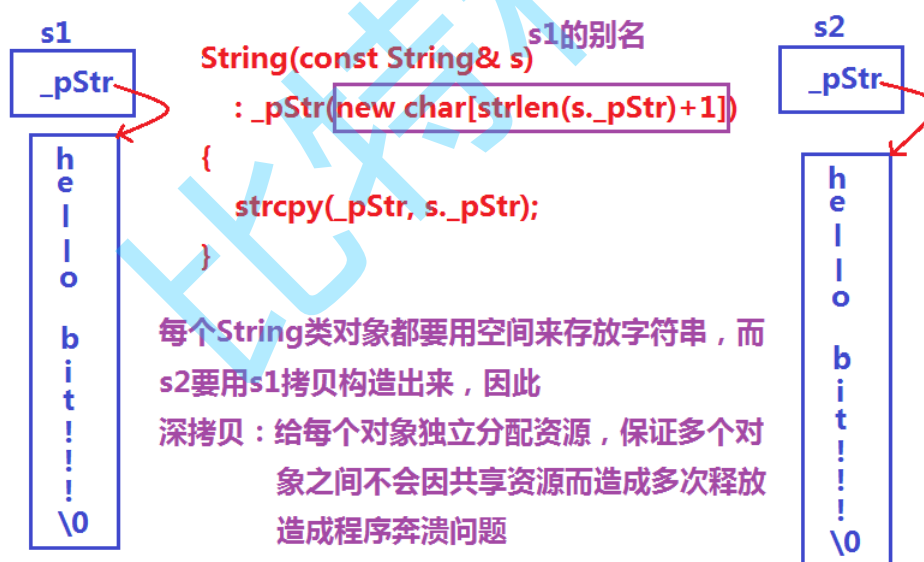


浅拷贝只关注了美人鱼美丽的上半身, 而深拷贝探索到了美人鱼不为人知的下半身。



3.3 深拷贝

如果一个类中涉及到资源的管理，其拷贝构造函数、赋值运算符重载以及析构函数必须要显式给出。一般情况都是按照深拷贝方式提供。



3.3.1 传统版写法的string类

```
1 class string
2 {
3 public:
4     string(const char* str = "")
5     {
6         // 构造string类对象时，如果传递nullptr指针，认为程序非法，此处断言下
```

```

7     if(nullptr == str)
8     {
9         assert(false);
10        return;
11    }
12
13    _str = new char[strlen(str) + 1];
14    strcpy(_str, str);
15 }
16
17 string(const string& s)
18     : _str(new char[strlen(s._str)+1])
19 {
20     strcpy(_str, s._str);
21 }
22
23 string& operator=(const string& s)
24 {
25     if(this != &s)
26     {
27         char* pStr = new char[strlen(s._str) + 1];
28         strcpy(pStr, s._str);
29         delete[] _str;
30         _str = pStr;
31     }
32
33     return *this;
34 }
35
36 ~string()
37 {
38     if(_str)
39     {
40         delete[] _str;
41         _str = nullptr;
42     }
43 }
44
45 private:
46     char* _str;
47 };

```

3.3.2 现代版写法的string类

```

1 class string
2 {
3 public:
4     string(const char* str = "")
5     {
6         if(nullptr == str)
7             str = "";

```

```

8
9     _str = new char[strlen(str) + 1];
10    strcpy(_str, str);
11    }
12
13    string(const string& s)
14        : _str(nullptr)
15    {
16        string strTmp(s._str);
17        swap(_str, strTmp);
18    }
19
20    // 对比下和上面的赋值那个实现比较好?
21    string& operator=(string s)
22    {
23        swap(_str, s._str);
24        return *this;
25    }
26
27    /*
28    string& operator=(const string& s)
29    {
30        if(this != &s)
31        {
32            string strTmp(s);
33            swap(_str, strTmp._str);
34        }
35
36        return *this;
37    }
38    */
39
40    ~string()
41    {
42        if(_str)
43        {
44            delete[] _str;
45            _str = nullptr;
46        }
47    }
48
49    private:
50        char* _str;
51    };

```

3.3 写时拷贝(了解)



写时拷贝就是一种拖延症，是在浅拷贝的基础之上增加了引用计数的方式来实现的。

引用计数：用来记录资源使用者的个数。在构造时，将资源的计数给成1，每增加一个对象使用该资源，就给计数增加1，当某个对象被销毁时，先给该计数减1，然后再检查是否需要释放资源，如果计数为1，说明该对象时资源的最后一个使用者，将该资源释放；否则就不能释放，因为还有其他对象在使用该资源。

写时拷贝

写时拷贝在读取是的缺陷

3.4 string类的模拟实现

```
1 namespace bit
2 {
3     class string
4     {
5     public:
6         typedef char* iterator;
7     public:
8         string(const char* str = "")
9         {
10             _size = strlen(str);
11             _capacity = _size;
12             _str = new char[_capacity+1];
13             strcpy(_str, str);
14         }
15
16         string(const string& s)
17             : _str(nullptr)
18             , _size(0)
19             , _capacity(0)
20         {
21             string tmp(s);
22             this->Swap(tmp);
23         }
24
25         string& operator=(string s)
26         {
27             this->Swap(s)
```



```

28         return *this;
29     }
30
31     ~string()
32     {
33         if (_str)
34         {
35             delete[] _str;
36             _str = nullptr;
37         }
38     }
39
40     //////////////////////////////////////
41     // iterator
42     iterator begin() {return _str;}
43     iterator end(){return _str + _size;}
44
45     //////////////////////////////////////
46     // modify
47     void push_back(char c)
48     {
49         if (_size == _capacity)
50             Reserve(_capacity*2);
51
52         _str[_size++] = c;
53         _str[_size] = '\0';
54     }
55
56     string& operator+=(char c)
57     {
58         PushBack(c);
59         return *this;
60     }
61
62     // 作业实现
63     void append(const char* str);
64     string& operator+=(const char* str);
65
66     void clear()
67     {
68         _size = 0;
69         _str[_size] = '\0';
70     }
71
72     void swap(string& s)
73     {
74         swap(_str, s._str);
75         swap(_size, s._size);
76         swap(_capacity, s._capacity);
77     }
78
79     const char* c_str()const
80     {

```

```

81         return _str;
82     }
83
84     //////////////////////////////////////
85     // capacity
86     size_t size()const
87     size_t capacity()const
88     bool empty()const
89
90     void resize(size_t newSize, char c = '\0')
91     {
92         if (newSize > _size)
93         {
94             // 如果newSize大于底层空间大小,则需要重新开辟空间
95             if (newSize > _capacity)
96             {
97                 Reserve(newSize);
98             }
99
100             memset(_str + _size, c, newSize - _size);
101         }
102
103         _size = newSize;
104         _str[newSize] = '\0';
105     }
106
107     void reserve(size_t newCapacity)
108     {
109         // 如果新容量大于旧容量,则开辟空间
110         if (newCapacity > _capacity)
111         {
112             char* str = new char[newCapacity + 1];
113             strcpy(str, _str);
114
115             // 释放原来旧空间,然后使用新空间
116             delete[] _str;
117             _str = str;
118             _capacity = newCapacity;
119         }
120     }
121
122     //////////////////////////////////////
123     // access
124     char& operator[](size_t index)
125     {
126         assert(index < _size);
127         return _str[index];
128     }
129
130     const char& operator[](size_t index)const
131     {
132         assert(index < _size);
133
134         return _str[index];

```

```

134     }
135
136     //////////////////////////////////////
137     // 作业
138     bool operator<(const string& s);
139     bool operator<=(const string& s);
140     bool operator>(const string& s);
141     bool operator>=(const string& s);
142     bool operator==(const string& s);
143     bool operator!=(const string& s);
144
145     // 返回c在string中第一次出现的位置
146     size_t find (char c, size_t pos = 0) const;
147     // 返回子串s在string中第一次出现的位置
148     size_t find (const char* s, size_t pos = 0) const;
149
150     // 在pos位置上插入字符c/字符串str, 并返回该字符的位置
151     string& insert(size_t pos, char c);
152     string& insert(size_t pos, const char* str);
153
154     // 删除pos位置上的元素, 并返回该元素的下一个位置
155     string& erase(size_t pos, size_t len);
156
157     private:
158         friend ostream& operator<<(ostream& _cout, const bit::string& s);
159         friend istream& operator>>(istream& _cin, bit::string& s);
160     private:
161         char* _str;
162         size_t _capacity;
163         size_t _size;
164     };
165 }
166
167 ostream& bit::operator<<(ostream& _cout, const bit::string& s)
168 {
169     // 不能使用这个
170     //cout << s._str;
171     for(size_t i = 0; i < s.size(); ++i)
172     {
173         _cout<<s[i];
174     }
175     return _cout;
176 }
177
178 //对自定义的string类进行测试
179 void TestBitstring()
180 {
181     bit::string s1("hello");
182     s1.push_back(' ');
183     s1.push_back('b');
184     s1.append(1, 'i');
185     s1 += 't';
186
187     cout << s1 << endl;

```

```
187     cout << s1.size() << endl;
188     cout << s1.capacity() << endl;
189
190     // 利用迭代器打印string中的元素
191     string::iterator it = s1.begin();
192     while (it != s1.end())
193     {
194         cout << *it<<" ";
195         ++it;
196     }
197     cout << endl;
198
199     // 这里可以看到一个类只要支持的基本的iterator, 就支持范围for
200     for(auto ch : s1)
201         cout<<ch<<" ";
202     cout<<endl;
203 }
```

4. 扩展阅读

[面试中string的一种正确写法](#)

[STL中的string类怎么了?](#)