# 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 |

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

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 |

```
// size/clear/resize
void Teststring1()
{
    // 注意：string类对象支持直接用cin和cout进行输入和输出

    string s("hello, bit!!!");
```

```
    cout << s.size() << endl;
    cout << s.length() << endl;
    cout << s.capacity() << endl;
    cout << s <<endl;

    // 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
    s.clear();
    cout << s.size() << endl;
    cout << s.capacity() << endl;

    // 将s中有效字符个数增加到10个，多出位置用'a'进行填充
    // “aaaaaaaaaa”
    s.resize(10, 'a');
    cout << s.size() << endl;
    cout << s.capacity() << endl;

    // 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
    // "aaaaaaaaaa\0\0\0\0\0"
    // 注意此时s中有效字符个数已经增加到15个
    s.resize(15);
    cout << s.size() << endl;
    cout << s.capacity() << endl;
    cout << s << endl;

    // 将s中有效字符个数缩小到5个
    s.resize(5);
    cout << s.size() << endl;
    cout << s.capacity() << endl;
    cout << s << endl;
}

//=============================================================================
====
void Teststring2()
{
    string s;
    // 测试reserve是否会改变string中有效元素个数
    s.reserve(100);
    cout << s.size() << endl;
    cout << s.capacity() << endl;

    // 测试reserve参数小于string的底层空间大小时，是否会将空间缩小
    s.reserve(50);
    cout << s.size() << endl;
    cout << s.capacity() << endl;
}

// 利用reserve提高插入数据的效率，避免增容带来的开销
//=============================================================================
====
void TestPushBack()
{

    string s;
```

```
    size_t sz = s.capacity();
    cout << "making s grow:\n";
    for (int i = 0; i < 100; ++i)
    {
        s.push_back('c');
        if (sz != s.capacity())
        {
            sz = s.capacity();
            cout << "capacity changed: " << sz << '\n';
        }
    }
}

void TestPushBackReserve()
{
    string s;
     s.reserve(100);
    size_t sz = s.capacity();

    cout << "making s grow:\n";
    for (int i = 0; i < 100; ++i)
    {
        s.push_back('c');
        if (sz != s.capacity())
        {
            sz = s.capacity();
            cout << "capacity changed: " << sz << '\n';
        }
    }
}
```

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，reserver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 |

```
void Teststring()
{
    string s1("hello Bit");
    const string s2("Hello Bit");
    cout<<s1<<" "<<s2<<endl;
    cout<<s1[0]<<" "<<s2[0]<<endl;

    s1[0] = 'H';
    cout<<s1<<endl;

    // s2[0] = 'h';   代码编译失败，因为const类型对象不能修改
}

void Teststring()
{
    string s("hello Bit");
    // 3种遍历方式:
    // 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，
    // 另外以下三种方式对于string而言，第一种使用最多
    // 1. for+operator[]
    for(size_t i = 0; i < s.size(); ++i)
        cout<<s[i]<<endl;

    // 2.迭代器
    string::iterator it = s.begin();
    while(it != s.end())
    {
        cout<<*it<<endl;
        ++it;
    }

    string::reverse_iterator rit = s.rbegin();
    while(rit != s.rend())
        cout<<*rit<<endl;

    // 3.范围for

    for(auto ch : s)
```

```
        cout<<ch<<endl;
}
```

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 |

```
void Teststring()
{
    string str;
    str.push_back(' ');   // 在str后插入空格
    str.append("hello");  // 在str后追加一个字符"hello"
    str += 'b';           // 在str后追加一个字符'b'
    str += "it";          // 在str后追加一个字符串"it"
    cout<<str<<endl;
    cout<<str.c_str()<<endl;   // 以C语言的方式打印字符串

    // 获取file的后缀
    string file1("string.cpp");
    size_t pos = file.rfind('.');
    string suffix(file.substr(pos, file.size()-pos));
    cout << suffix << endl;

    // npos是string里面的一个静态成员变量
    // static const size_t npos = -1;

    // 取出url中的域名
    string url("http://www.cplusplus.com/reference/string/string/find/");
    cout << url << endl;
    size_t start = url.find("://");
    if (start == string::npos)
    {
        cout << "invalid url" << endl;
        return;
    }
    start += 3;
    size_t finish = url.find('/', start);
    string address = url.substr(start, finish - start);
```

```
    cout << address << endl;

    // 删除url的协议前缀
    pos = url.find("://");
    url.erase(0, pos+3);
    cout<<url<<endl;
}
```

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

```
class Solution {
public:
    bool isLetter(char ch)
    {
        if(ch >= 'a' && ch <= 'z')
            return true;
        if(ch >= 'A' && ch <= 'Z')
            return true;

        return false;
    }
    string reverseOnlyLetters(string S) {
        if(S.empty())
            return S;

        size_t begin = 0, end = S.size()-1;
        while(begin < end)
        {
            while(begin < end && !isLetter(S[begin]))
```

```
                ++begin;

            while(begin < end && !isLetter(S[end]))
                --end;

            swap(S[begin], S[end]);
            ++begin;
            --end;
        }

        return S;
    }
};
```

[找字符串中第一个只出现一次的字符]

```
class Solution {
public:
    int firstUniqChar(string s) {

        // 统计每个字符出现的次数
    int count[256] = {0};
        int size = s.size();
        for(int i = 0; i < size; ++i)
            count[s[i]] += 1;

        // 按照字符次序从前往后找只出现一次的字符
        for(int i = 0; i < size; ++i)
            if(1 == count[s[i]])
                return i;

        return -1;
    }
};
```

[字符串里面最后一个单词的长度--课堂练习]

```
#include<iostream>
#include<string>
using namespace std;

int main()
{
    string line;
    // 不要使用cin>>line,因为会它遇到空格就结束了
    // while(cin>>line)
    while(getline(cin, line))
    {
        size_t pos = line.rfind(' ');
        cout<<line.size()-pos-1<<endl;

    }
```

```
    return 0;
}
```

[验证一个字符串是否是回文]

```
class Solution {
public:
    bool isLetterOrNumber(char ch)
{
        return (ch >= '0' && ch <= '9')
        || (ch >= 'a' && ch <= 'z')
            || (ch >= 'A' && ch <= 'Z');
    }

    bool isPalindrome(string s) {
        // 先小写字母转换成大写，再进行判断
        for(auto& ch : s)
        {
            if(ch >= 'a' && ch <= 'z')
                ch -= 32;
        }

        int begin = 0, end = s.size()-1;
        while(begin < end)
        {
            while(begin < end && !isLetterOrNumber(s[begin]))
                ++begin;

            while(begin < end && !isLetterOrNumber(s[end]))
                --end;

            if(s[begin] != s[end])
            {
                return false;
            }
            else
            {

                ++begin;
                --end;
            }
        }

        return true;
    }
};
```

[字符串相加]

```
class Solution {

public:
```

```
    string addstrings(string num1, string num2)
{
        // 从后往前相加，相加的结果到字符串可以使用insert头插
    // 或者+=尾插以后再reverse过来
        int end1 = num1.size()-1;
        int end2 = num2.size()-1;
        int value1 = 0, value2 = 0, next = 0;
        string addret;
        while(end1 >= 0 || end2 >= 0)
        {
            if(end1 >= 0)
                value1 = num1[end1--]-'0';
            else
                value1 = 0;

            if(end2 >= 0)
                value2 = num2[end2--]-'0';
            else
                value2 = 0;

            int valueret = value1 + value2 + next;
            if(valueret > 9)
            {
                next = 1;
                valueret -= 10;
            }
            else
            {
                next = 0;
            }

            //addret.insert(addret.begin(), valueret+'0');
            addret += (valueret+'0');
        }

        if(next == 1)
        {
            //addret.insert(addret.begin(), '1');
            addret += '1';
        }

        reverse(addret.begin(), addret.end());
        return addret;
    }
};
```

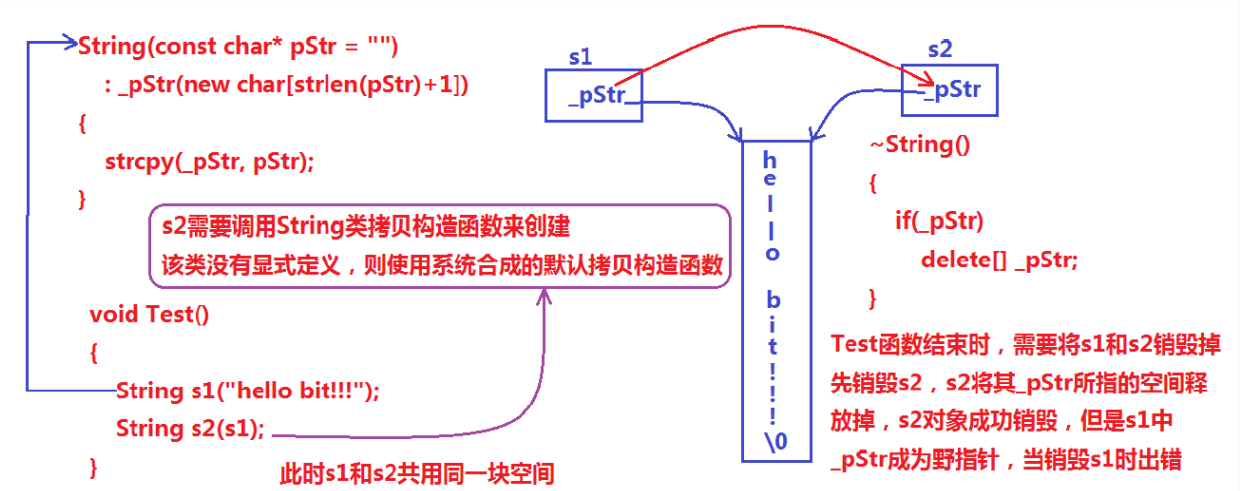
1. 课后作业练习——翻转字符串II：区间部分翻转--课后作业
2. 课后作业练习——翻转字符串III：翻转字符串中的单词--课后作业
3. 课后作业练习——字符串相乘
4. 课后作业练习——找出字符串中第一个只出现一次的字符

## 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？

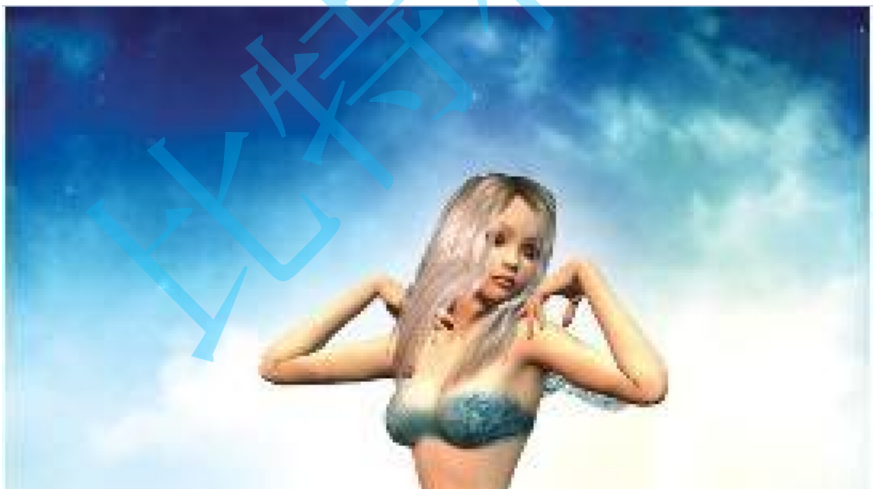
```
class string
{
public:
    /*string()
        :_str(new char[1])
        {*_str = '\0';}
    */
    //string(const char* str = "\0") 错误示范
    //string(const char* str = nullptr) 错误示范
    string(const char* str = "")
    {
        // 构造string类对象时，如果传递nullptr指针，认为程序非法，此处断言下
        if(nullptr == str)
        {
            assert(false);
            return;
        }

        _str = new char[strlen(str) + 1];
        strcpy(_str, str);
    }

    ~string()
    {
        if(_str)
        {
            delete[] _str;
            _str = nullptr;
        }
    }

private:
    char* _str;
};

// 测试
void Teststring()
{
    string s1("hello bit!!!");
    string s2(s1);
}
```




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

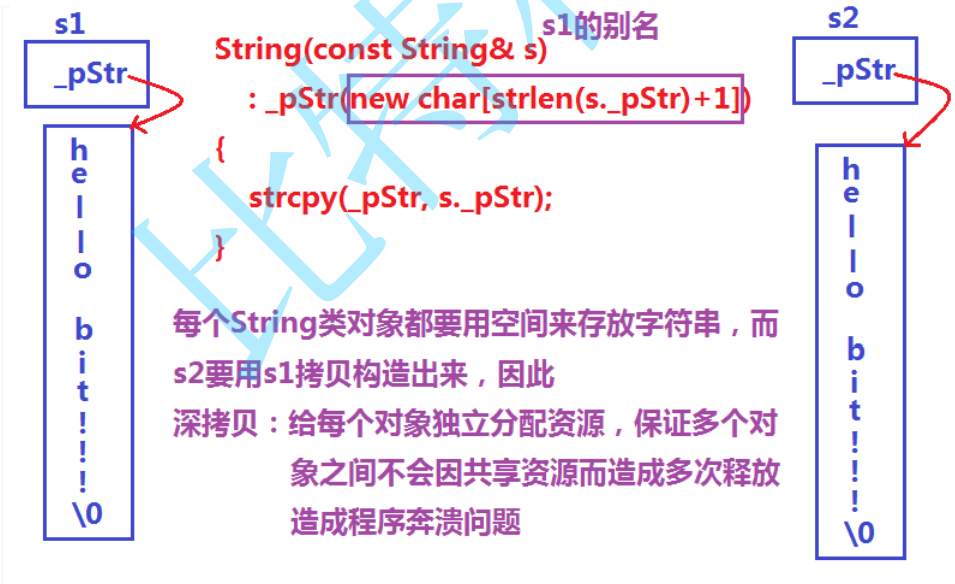
**浅拷贝：也称位拷贝，编译器只是将对象中的值拷贝过来。**如果**对象中管理资源**，最后就会**导致多个对象共享同一份资源，当一个对象销毁时就会将该资源释放掉，而此时另一些对象不知道该资源已经被释放，以为还有效，所以 当继续对资源进项操作时，就会发生发生了访问违规。**要解决浅拷贝问题，C++中引入了深拷贝。



**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

```
class string
{
public:
    string(const char* str = "")
    {
        // 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
```

```
        if(nullptr == str)
        {
            assert(false);
            return;
        }

        _str = new char[strlen(str) + 1];
        strcpy(_str, str);
    }

    string(const string& s)
        : _str(new char[strlen(s._str)+1])
    {
            strcpy(_str, s._str);
    }

    string& operator=(const string& s)
    {
        if(this != &s)
        {
            char* pStr = new char[strlen(s._str) + 1];
            strcpy(pStr, s._str);
            delete[] _str;
            _str = pStr;
        }

        return *this;
    }

    ~string()
    {
        if(_str)
        {
            delete[] _str;
            _str = nullptr;
        }
    }

private:
    char* _str;
};
```



### 3.3.2 现代版写法的string类

```
class string
{
public:
    string(const char* str = "")
    {
        if(nullptr == str)
            str = "";
```

```
        _str = new char[strlen(str) + 1];
        strcpy(_str, str);
    }

    string(const string& s)
        : _str(nullptr)
    {
        string strTmp(s._str);
        swap(_str, strTmp._str);
    }

    // 对比下和上面的赋值那个实现比较好?
    string& operator=(string s)
    {
        swap(_str, s._str);
        return *this;
    }

    /*
    string& operator=(const string& s)
    {
        if(this != &s)
        {
           string strTmp(s);
           swap(_str, strTmp._str);
        }

        return *this;
    }
    */

    ~string()
    {
        if(_str)
        {
            delete[] _str;
            _str = nullptr;
        }
    }

private:
    char* _str;
};
```



### 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

```
namespace bit
{
    class string
    {
    public:
        typedef char* iterator;
    public:
        string(const char* str = "")
        {
            _size = strlen(str);
            _capacity = _size;
            _str = new char[_capacity+1];
            strcpy(_str, str);
        }

        string(const string& s)
            : _str(nullptr)
            , _size(0)
            , _capacity(0)
        {
            string tmp(s._str);
             this->swap(tmp);
        }

        string& operator=(string s)
        {

             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             return _str[index];
```

```
        }

        ////////////////////////////////////////////////////////////////
        // 作业
        bool operator<(const string& s);
        bool operator<=(const string& s);
        bool operator>(const string& s);
        bool operator>=(const string& s);
        bool operator==(const string& s);
        bool operator!=(const string& s);

        // 返回c在string中第一次出现的位置
        size_t find (char c, size_t pos = 0) const;
         // 返回子串s在string中第一次出现的位置
         size_t find (const char* s, size_t pos = 0) const;

        // 在pos位置上插入字符c/字符串str，并返回该字符的位置
        string& insert(size_t pos, char c);
        string& insert(size_t pos, const char* str);

        // 删除pos位置上的元素，并返回该元素的下一个位置
        string& erase(size_t pos, size_t len);

    private:
        friend ostream& operator<<(ostream& _cout, const bit::string& s);
        friend istream& operator>>(istream& _cin, bit::string& s);
    private:
        char* _str;
        size_t _capacity;
        size_t _size;
    };
}

ostream& bit::operator<<(ostream& _cout, const bit::string& s)
{
    // 不能使用这个
    //cout << s._str;
    for(size_t i = 0; i < s.size(); ++i)
    {
        _cout<<s[i];
    }
    return _cout;
}

///////对自定义的string类进行测试
void TestBitstring()
{
    bit::string s1("hello");
    s1.push_back(' ');
    s1.push_back('b');
    s1.append(1, 'i');
    s1 += 't';

    cout << s1 << endl;
```

```
    cout << s1.size() << endl;
    cout << s1.capacity() << endl;

    // 利用迭代器打印string中的元素
    string::iterator it = s1.begin();
    while (it != s1.end())
    {
        cout << *it<<" ";
         ++it;
    }
    cout << endl;

    // 这里可以看到一个类只要支持的基本的iterator，就支持范围for
    for(auto ch : s1)
        cout<<ch<<" ";
    cout<<endl;
}
```

## 4. 扩展阅读

> 面试中string的一种正确写法
>
> STL中的string类怎么了？