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

string容量相关方法使用代码演示

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

string中元素访问及遍历代码演示

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

string中插入和查找等使用代码演示

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. vs和g++下string结构的说明

注意：下述结构是在32位平台下进行验证，32位平台下指针占4个字节。

- **vs下string的结构**

  string总共占28个字节，内部结构稍微复杂一点，先是**有一个联合体，联合体用来定义string中字符串的存储空间**：

  - **当字符串长度小于16时，使用内部固定的字符数组来存放**
  - **当字符串长度大于等于16时，从堆上开辟空间**

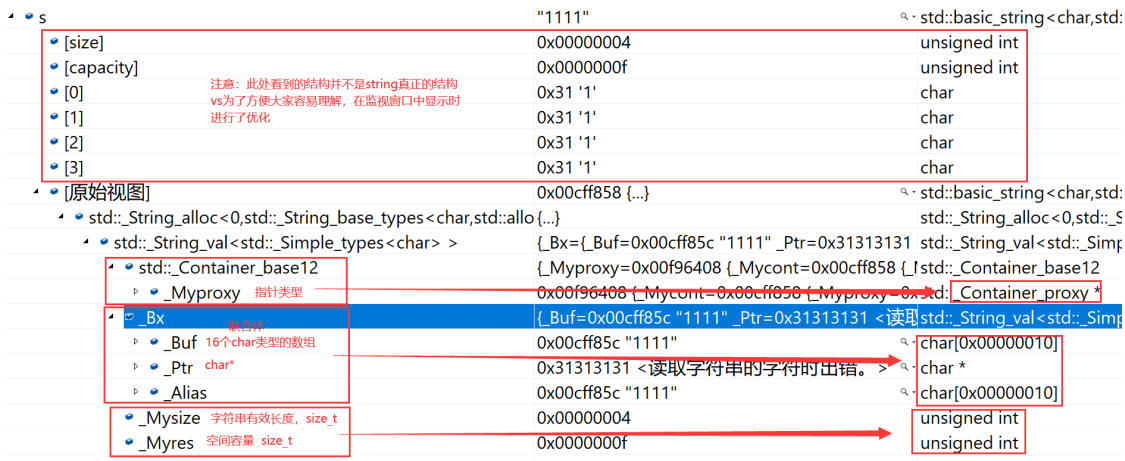
```
union _Bxty
{   // storage for small buffer or pointer to larger one
    value_type _Buf[_BUF_SIZE];
    pointer _Ptr;
    char _Alias[_BUF_SIZE]; // to permit aliasing
} _Bx;
```

这种设计也是有一定道理的，大多数情况下字符串的长度都小于16，那string对象创建好之后，内部已经有了16个字符数组的固定空间，不需要通过堆创建，效率高。

其次：还有**一个size_t字段保存字符串长度，一个size_t字段保存从堆上开辟空间总的容量**

最后：还**有一个指针**做一些其他事情。

故总共占16+4+4+4=28个字节。



- **g++下string的结构**

  G++下，string是通过写时拷贝实现的，string对象总共占4个字节，内部只包含了一个指针，该指针将来指向一块堆空间，内部包含了如下字段：

  - 空间总大小
  - 字符串有效长度
  - 引用计数

```
struct _Rep_base
{
    size_type               _M_length;
    size_type               _M_capacity;
    _Atomic_word            _M_refcount;
};
```

  - 指向堆空间的指针，用来存储字符串。

7. 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？

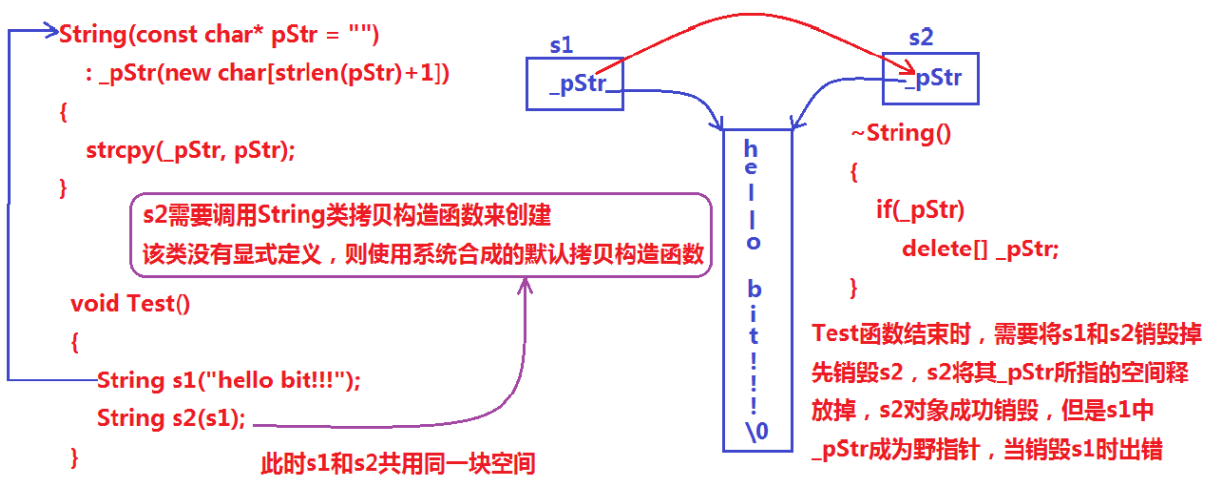
```
// 为了和标准库区分，此处使用String
class String
{
public:
```

```
    /*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，可以认为程序非
        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

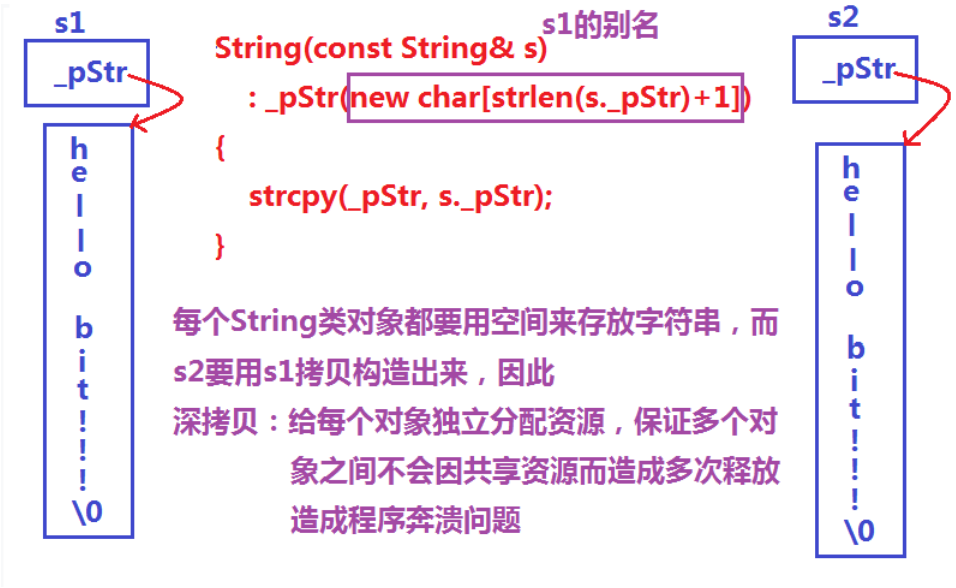
浅拷贝：也称位拷贝，编译器只是将对象中的值拷贝过来。如果**对象中管理资源**，最后就会**导致多个对象共享同一份资源，当一个对象销毁时就会将该资源释放掉，而此时另一些对象不知道该资源已经被释放，以为还有效，所以当继续对资源进项操作时，就会发生发生了访问违规。**

就像一个家庭中有两个孩子，但父母只买了一份玩具，两个孩子愿意一块玩，则万事大吉，万一不想分享就你争我夺，玩具损坏。

可以采用深拷贝解决浅拷贝问题，即：**每个对象都有一份独立的资源，不要和其他对象共享。**父母给每个孩子都买一份玩具，各自玩各自的就不会有问题了。

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，可以认为程序非
        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;
    }
```

```
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           {
8               assert(false);
9               return;
10          }
11
12          _str = new char[strlen(str) + 1];
13          strcpy(_str, str);
14      }
15
16      String(const String& s)
17          : _str(nullptr)
18      {
19          String strTmp(s._str);
20          swap(_str, strTmp._str);
21      }
22
23      // 对比下和上面的赋值那个实现比较好?
24      String& operator=(String s)
25      {
26          swap(_str, s._str);
27          return *this;
28      }
29
30      /*
31      String& operator=(const String& s)
32      {
33          if(this != &s)
34          {
35             String strTmp(s);
36             swap(_str, strTmp._str);
37          }
```

```
38
39          return *this;
40      }
41      */
42
43      ~String()
44      {
45          if (_str)
46          {
47              delete[] _str;
48              _str = nullptr;
49          }
50      }
51
52  private:
53      char* _str;
54  };
```

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

string模拟时间参考

4. 扩展阅读

面试中string的一种正确写法

STL中的string类怎么了？