字符串

一、C风格字符串

字符串处理在程序中应用广泛,C风格字符串是以'\0'(空字符)来结尾的字符数组。对字符串进行操作的C函数定义在头文件 < string. h>或 < cstring>中。常用的库函数如下:

```
1 //字符检查函数(非修改式操作)
   size_t strlen( const char *str );//返回str的长度,不包括null结束
   符
3
   //比较1hs和rhs是否相同。1hs等于rhs,返回0; 1hs大于rhs,返回正数; 1hs
   小于rhs,返回负数
   int strcmp( const char *lhs, const char *rhs );
   int strncmp( const char *lhs, const char *rhs, size_t count
   );
7
   //在str中查找首次出现ch字符的位置;查找不到,返回空指针
   char *strchr( const char *str, int ch );
   //在str中查找首次出现子串substr的位置:查找不到,返回空指针
10
11
   char *strstr( const char* str, const char* substr );
12
   //字符控制函数(修改式操作)
13
14 char *strcpy(char *dest, const char *src);//将src复制给dest, 返
   回dest
15 char *strncpy(char *dest, const char *src, size_t count);
16 char *strcat( char *dest, const char *src );//concatenates
   two strings
17 char *strncat( char *dest, const char *src, size_t count );
```

在使用时,程序员需要考虑字符数组大小的开辟,结尾空字符的处理,使用起来有诸多不便。

```
1 void test0()
2 {
3     char str[] = "hello";
4     char * pstr = "world";
5     //求取字符串长度
7     printf("%d\n", strlen(str));
```

```
8
 9
       //字符串拼接
       char * ptmp = (char*)malloc(strlen(str) + strlen(pstr) +
10
   1);
11
        strcpy(ptmp, str);
12
        strcat(ptmp, pstr);
13
        printf("%s\n", ptmp);
14
15
       //查找子串
16
        char * p1 = strstr(ptmp, "world");
17
18
       free(ptmp);
19 }
```

二、C++风格字符串

C++ 提供了 std::string(后面简写为 string)类用于字符串的处理。 string类定义在 C++ 头文件 <string> 中,注意和头文件 <cstring> 区分, <cstring> 其实是对 C 标准库中的 <string. h> 的封装,其定义的是一些对 C 风格字符串的处理函数。

尽管 C++ 支持 C 风格字符串,但在 C++ 程序中最好还是不要使用它们。这是因为 C 风格字符串不仅使用起来不太方便,而且极易引发程序漏洞,是诸多安全问题的根本原因。与 C 风格字符串相比, string 不必担心内存是否足够、字符串长度,结尾的空白符等等。 string 作为一个类出现,其集成的成员操作函数功能强大,几乎能满足所有的需求。从另一个角度上说,完全可以把 string 当成是 C++ 的内置数据类型,放在和 int 、 double 等内置类型同等位置上。 string 类本质上其实是 basic_string 类模板关于 char 型的实例化。

我们先来看一个简单的例子:

```
void test1()
2
   {
3
        //C风格字符串转换为C++风格字符串
        std::string s1 = "hello";
4
 5
        std::string s2("world");
6
7
       //求取字符串长度
8
        cout << s1.size() << endl;</pre>
9
        cout << s1.length() << endl;</pre>
10
       //字符串的遍历
11
```

```
for(size_t idx = 0; idx != s1.size(); ++idx) {
12
            cout << s1[idx] << " ";
13
14
        }
15
        cout << endl;</pre>
16
17
        //字符串拼接
18
        std::string s3 = s1 + s2;
19
        cout << "s3 = " << s3 << end1;
20
21
        //查找子串
        size_t pos = s1.find("world");
22
23
24
        //截取子串
25
        std::string substr = s1.substr(pos);
26
        cout << "substr = " << substr << endl;</pre>
27
   }
```

std::string提供了很多方便字符串操作的方法。

string 对象的构造

首先来看一下 string 类型常用的构造函数

```
1 string();//默认构造函数,生成一个空字符串
2 string(const char * rhs);//通过c风格字符串构造一个string对象
3 string(const char * rhs, size_type count);//通过rhs的前count个字符构造一个string对象
4 string(const string & rhs);//复制拷贝构造函数
5 string(size_type count, char ch);//生成一个string对象,该对象包含count个ch字符
6 string(InputIt first, InputIt last);//以区间[first, last)内的字符创建一个string对象
```

string与C风格字符串的转换

C 风格字符串转换为 string 字符串相对来说比较简单,通过构造函数即可实现。但由于 string 字符串实际上是类对象,其并不以空字符 '\0'结尾,因此, string 字符串向 C 风格字符串的转化是通过3个成员函数完成的,分别为:

```
1 const char * c_str() const;// 返回一个C风格字符串
2 const char * data() const;// c++11之后与c_str()效果一致
3 
4 //字符串的内容复制或写入既有的C风格字符串或字符数组内
5 size_type copy(char* dest, size_type count, size_type pos = 0) const;
```

元素遍历和存取

string 对象可以使用下标操作符[]和函数 at() 对字符串中包含的字符进行访问。需要注意的是**操作符[]并不检查索引是否有效**,如果索引超出范围,会引起未定义的行为。而 at() 会检查,如果使用 at() 的时候索引无效,会抛出 out_of_range 异常。

```
1 reference operator[]( size_type pos );// 返回下标为pos的元素
2 const_reference operator[]( size_type pos ) const;//
3
4 reference at( size_type pos );// 返回下标为pos的元素
5 const_reference at( size_type pos ) const;//
6
```

除此以外,还可以使用迭代器进行遍历访问

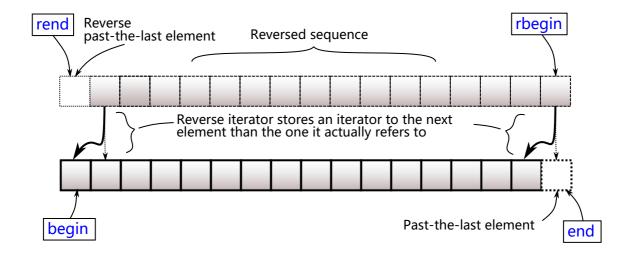
```
iterator begin();
const_iterator begin() const;

iterator end();
const_iterator end() const;

reverse_iterator rbegin();
const_reverse_iterator rbegin() const;

reverse_iterator rend();
const_reverse_iterator rend() const;
```

其示意图如下:



字符串的长度和容量相关

```
bool empty() const;
size_type size() const;
size_type length() const;
size_type capacity() const;
size_type max_size() const;
```

元素追加和相加

```
1 string & operator+=(const string & str);
   string & operator+=(CharT ch);
   string & operator+=(const CharT* s);
   string & append(size_type count, CharT ch);
   string & append(const basic_string & str);
6
   string & append(const CharT* s);
   string & append(InputIt first, InputIt last);
8
9
10
   //以下为非成员函数
11
   string operator+(const string & lhs, const string & rhs);
   string operator+(const string & lhs, const char* rhs);
12
   string operator+(const char* lhs, const string & rhs);
13
   string operator+(const string & lhs, char rhs);
14
15 | string operator+(char lhs, const string & rhs);
```

提取子串

```
string substr(size_type pos = 0, size_type count = npos)
const;
```

元素删除

```
1 iterator erase(iterator position);
2 iterator erase(const_iterator position);
3 iterator erase(iterator first, iterator last);
```

元素清空

```
1 | void clear();//
```

字符串比较

```
1 //非成员函数
2 bool operator==(const string & lhs, const string & rhs);
3 bool operator!=(const string & lhs, const string & rhs);
4 bool operator>(const string & lhs, const string & rhs);
5 bool operator<(const string & lhs, const string & rhs);
6 bool operator>=(const string & lhs, const string & rhs);
7 bool operator<=(const string & lhs, const string & rhs);</pre>
```

搜索与查找

```
1 //find系列:
   size_type find(const basic_string & str, size_type pos = 0)
   const;
   size_type find(const CharT* s, size_type pos = 0) const;
   size_type find(const CharT* s, size_type pos, size_type
   count) const;
   size_type find(char ch, size_type pos = npos ) const;
5
6
7
   //rfind系列:
   size_type rfind(const basic_string & str, size_type pos = 0)
   const;
9 size_type rfind(const CharT* s, size_type pos = 0) const;
10 size_type rfind(const CharT* s, size_type pos, size_type
   count) const;
11 size_type rfind(char ch, size_type pos = npos) const;
```