string类型

string类型变量的定义

在C++中提供了一个内建数据类型string，该数据类型可以替代C语言中char数组。需要使用string数据类型时则需要在程序中包含头文件string。string类型处理起来会比较方便，下面我们将逐一介绍该类型的功能。

例1：

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

int main(){
    string s2 = "string";
    string s1 = s2;
    string s4(10, 's');
    return 0;
}
```

在本例中介绍了几种定义string类型变量的方法，变量s1只是定义但是没有进行初始化，系统会将默认值赋给s1，默认值是""（空字符串）。变量s2在定义的时候就被初始化为了"string"，与C风格的char型数组不同，string类型的变量结尾是没有 '\0' 的，string类型的本质是一个string类，而我们定义的变量则是一个个的string类的对象。变量s3在定义的时候直接用s2进行初始化了，因此s3的内容也是"string"。变量s4初始化为10个's'字符组成的字符串，也即'ssssssssss'。

从例1中我们也可以看出string类型变量可以直接通过赋值操作符"="进行赋值。string类型变量可以用string类型变量或C风格字符串进行赋值。如s2则是用一个字符串常量进行初始化的，而s3变量则是通过s2变量进行初始化。

与C风格的字符数组不同，当我们需要知道字符串长度时，string类为我们提供了length函数。如下面例2所示，我们可以通过s变量来调用length函数，从而返回s变量的长度。因为string类型的变量末尾是没有 '\0' 字符的，因此其返回值就是变量的真实长度，而不是长度+1。

例2：

```
string s = "string";
int len = s.length();
```

转换为 char 数组字符串

虽然C++提供了string类型来替代C语言中的字符数组形式的字符串，但是我们在程序设计过程中还是不可避免地会碰到需要用字符数组形式字符串的地方。为此，系统为我们提供了一个转换函数c_str，该函数将string类型的变量转换为一个const的字符串数组的指针。

例1：

```
#include <fstream>

string filename = "input.txt";
ifstream in;
in.open(filename.c_str());
```

在本例中我们为了打开文件的函数open，因此必须将string类型变量转换为字符串指针。

string类型变量的输入与输出

既然新增string类型变量，则不可避免的需要涉及到string类型变量的输入与输出操作。在C++中，在输入输出方面，我们可以像对待普通变量那样对待string类型变量，其输入输出仍然可以用输入输出操作符进行处理。

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

int main(){
    string s;
    cin >> s;
    cout << s << endl;
    return  0;
}
```

如本例所示，输入一个字符串，然后再将输入的字符串输出。运行程序结果如下：

```
string string↙
string
```

这个例子的运行结果输出只是一个string，而我们输入的是两个string，并且中间用空格隔开了。其实问题就出现在空格上，输入操作符是默认忽略空格的，当遇到空格时就开始存储字符串。因此后面一个输入的string没有被存储进去。

对于string类型变量，我们可以直接用“+”或者“+=”进行字符串的连接，操作符非常方便。用“+”风格字符串进行字符串连接时，操作符左右两边既可以都是string字符串，也可以是一个string字符串和一个C风格的字符串，还可以是一个string字符串和一个char字符。而用“+=”风格字符串进行字符串连接时，操作符右边既可以是一个string字符串，也可以是一个C风格字符串或一个char字符。

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

int main(){
    string s1, s2, s3;
    s1 = "first";
    s2 = "second";
    s3 = s1 + s2;
    cout << s3 << endl;
    s2 += s1;
    cout << s2 << endl;
```

```
    s1 += "third";
    cout<< s1 <<endl;
    s1 += 'a';
    cout << s1 << endl;
    return 0;
}
```

在本例中利用“+”和“+=”操作符分别尝试进行字符串连接，上面的所有连接都是符合语法规定的。string字符串连接非常灵活，大家可以多进行尝试。

string类中提供的主要方法

函数名称	功能
构造函数	产生或复制字符串
析构函数	销毁字符串
=，assign	赋以新值
swap	交换两个字符串的内容
+=，append( )，push_back()	添加字符
insert ()	插入字符
erase()	删除字符
clear()	移除全部字符
resize ()	改变字符数量
replace()	替换字符
+	串联字符串
==，！=，<，<=，>，>=，compare()	比较字符串内容
size()，length()	返回字符数量
max_size ()	返回字符的最大可能个数
empty ()	判断字符串是否为空
capacity ()	返回重新分配之前的字符容量
reserve()	保留内存以存储一定数量的字符
[],at()	存取单一字符
>>，getline()	从 stream 中读取某值
<<	将值写入 stream
copy()	将内容复制为一个 C - string
c_str()	将内容以 C - string 形式返回
data()	将内容以字符数组形式返回
substr()	返回子字符串
find()	搜寻某子字符串或字符
begin()，end()	提供正向迭代器支持
rbegin()，rend()	提供逆向迭代器支持
get_allocator()	返回配置器

字符串处理函数具体使用

string字符串同样可以像字符串数组那样按照下标逐一访问字符串中的每一个字符，string字符串的起始下标仍是从0开始。

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

int main(){
   string s1 ;
   s1 = "1234567890";
   for(int i=0; i<s1.length(); i++)
       cout << s1[i] << " ";
   cout << endl;
   s1[5] = '5';
   cout << s1 << endl;
   return 0;
}
```

在本例中我们定义了一个string类型变量s1,并给该变量赋值"1234567890",之后用for循环逐一输出每一个字符串中的字符。除了能够访问每一个字符外,修改它们同样也是允许的，例如在程序后面s1[5] = '5'语句将第6个字符修改为'5',最后s1被修改为了"1234557890"。

除了能逐个的去访问字符串中每一个字符外，系统还提供了一些函数方便我们操作string类型变量。

erase函数可以删除string类型变量中的一个子字符串。erase函数有两个参数，第一个参数是要删除的子字符串的起始下标，第二参数是要删除子字符串的长度，如果第二个参数不指名的话则是直接从第一个参数获取起始下标，然后一直删除至字符串结束。

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

int main(){
   string s1, s2, s3;
   s1 = s2 = s3 = "1234567890";
   s2.erase(5);        // 擦除从下标5开始的所有字符
   s3.erase(5, 3);     // 擦除从下标5开始以后的三个字符
   cout << s1 << endl;
   cout << s2 << endl;
   cout << s3 << endl;
   return 0;
}
```

程序最终运行结果：

```
1234567890
12345
1234590
```

当然，在使用erase函数时，在第一个参数没有越界的条件下，第二个参数可能会导致需要删除的子字符串越界，但实际上是不会的，函数会从以下两个值中取出一个最小值作为待删除子字符串的长度：

- 第二个参数的实参值；
- 字符串长度减去第一个参数的实参值。

其实说的简单一些，待删除字符串最多删除至字符串结尾。当然如果第一个参数直接越界了，那么函数执行会抛出异常的。

函数insert可以在string字符串中指定的位置插入另一个字符串，该函数同样有两个参数，第一个参数表示插入位置，第二参数表示要插入的字符串，第二个参数既可以是string变量，又可以是C风格的字符串。

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

int main(){
   string s1, s2, s3;
   s1 = s2 = "1234567890";
   s3 = "aaa";
   s1.insert(5, s3);
   cout << s1 << endl;
   s2.insert(5, "aaa");
   cout << s2 << endl;
   return 0;
}
```

本例最终运行结果：

```
12345aaa567890
12345aaa567890
```

insert函数的第一个参数同样有越界的可能，如果第一个参数越界，则函数会运行异常。

replace函数可以用一个指定的字符串来替换string类型变量中的一个子字符串，该函数有三个参数，第一个参数表示待替换的子字符串的起始下标，第二个参数表示待替换子字符串的长度，第三个参数表示要替换子字符串的字符串。第三个参数同样可以是string类型变量或C风格字符串。

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

int main(){
   string s1, s2, s3;
   s1 = s2 = "1234567890";
   s3 = "aaa";
   s1.replace(5, 4, s3);
   cout << s1 << endl;
   s2.replace(5, 4, "aaa");
   cout << s1 << endl;
   return 0;
}
```

程序运行结果如下：

```
12345aaa0
12345aaa0
```

同样的，该函数会有溢出的问题，如果第一个参数越界则会抛出异常。在第一个参数没有越界的前提下，第二个参数如果导致越界，则会选择以下两个值中的最小值作为待替换子字符串的长度：

- 第二个参数的实参值；
- 字符串长度减去第一个参数的实参值。

其实说白了，这个也就是说最多到字符串结尾的意思，这个与erase函数相同。

swap函数可以用于将两个string 类型变量的值互换，其使用方式见例5。

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

int main(){
   string s1 = "string";
   string s2 = "aaaaaa";
   s1.swap(s2);
   cout<< s1 << endl;
   cout<< s2 << endl;
   return 0;
}
```

程序运行结果如下：

```
aaaaaa
string
```

提取子字符串

函数substr可以提取string字符串中的子字符串，该函数有两个参数，第一个参数为需要提取的子字符串的起始下标，第二个参数是需要提取的子字符串的长度。

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

int main(){
   string s1 = "first second third";
   string s2;
   s2 = s1.substr(6, 6);
   cout << s1 << endl;
   cout << s2 << endl;
   return 0;
}
```

程序运行结果：

```
first second third
second
```

该函数同样会出现参数越界的情况，如果第一个参数越界则函数会抛出异常。在第一个参数没有越界的情况下，第二个参数仍然会导致越界，该函数的处理方式与前面提到的erase函数、replace函数相同，子字符串最多从第一个参数所指明的下标开始一直延续到字符串结尾

查找字符串

find函数可以在字符串中查找子字符串中出现的位置。该函数有两个参数，第一个参数是待查找的子字符串，第二个参数是表示开始查找的位置，如果第二个参数不指明的话则默认从0开始查找，也即从字符串首开始查找。

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

int main(){
   string s1 = "first second third";
   string s2 = "second";
   int index = s1.find(s2, 5);
   if(index < s1.length())
       cout << "Found at index : " << index << endl;
   else
       cout << "Not found" << endl;
   return 0;
}
```

函数最终返回的是子字符串出现在字符串中的起始下标。例1程序最终实在下标6处找到了s2字符串。如果没有查找到子字符串，则会返回一个无穷大值4294967295。

rfind函数与find函数很类似，同样是在字符串中查找子字符串，不同的是find函数是从第二个参数开始往后查找，而rfind函数则是最多查找到第二个参数处，如果到了第二个参数所指定的下标还没有找到子字符串，则返回一个无穷大值4294967295。

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

int main(){
   string s1 = "first second third";
   string s2 = "second";
   int index = s1.rfind(s2,6);
   if(index < s1.length())
       cout << "Found at index : " << index << endl;
   else
       cout << "Not found" << endl;
   return 0;
}
```

例2中rfind函数第二个参数是6，也就是说起始查找从0到6，如果找到了则返回下标，否则返回一个无穷大。本例中刚好在下标6的时候找到了子字符串s2，故而返回下标6。

find_first_of函数是用于查找子字符串和字符串共同具有的字符在字符串中出现的位置。

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

int main(){
   string s1 = "first second second third";
   string s2 = "asecond";
   int index = s1.find_first_of(s2);
   if(index < s1.length())
       cout << "Found at index : " << index << endl;
   else
       cout << "Not found" << endl;
   return 0;
}
```

本例中s1和s2共同具有的字符是's'，该字符在s1中首次出现的下标是3，故查找结果返回3。

而find_first_not_of函数则相反，它查找的是在s1字符串但不在s2字符串中的首位字符的下标，如果查找不成功则返回无穷大。

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

int main(){
    string s1 = "secondasecondthird";
    string s2 = "asecond";
    int index = s1.find_first_not_of(s2);
    if(index < s1.length())
        cout << "Found at index : " << index << endl;
    else
        cout << "Not found" << endl;
    return 0;
}
```

在本例中在s1但是不在s2中的首字符是't'，其所在下标为13，故而返回下标13。

字符串的比较

"=="、"!="、"<="、">="、"<"和">"操作符都可以用于进行string类型字符串的比较，这些操作符两边都可以是string字符串，也可以一边是string字符串另一边是字符串数组。

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

int main(){
    string s1 = "secondasecondthird";
    string s2 = "asecond";
    int index = s1.find_first_not_of(s2);
    if(index < s1.length())
        cout << "Found at index : " << index << endl;
```

```
    else
        cout << "Not found" << endl;
    return 0;
}
```

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

int main(){
    string s1 = "secondsecondthird";
    string s2 = "secondthird";
    if( s1 == s2 )
        cout << " == " << endl;
    if( s1 != s2 )
        cout << " != " << endl;
    if( s1 < s2 )
        cout << " < " << endl;
    if( s1 > s2 )
        cout << " > " << endl;
    return 0;
}
```

程序最终运行结果：

```
!=
<
```

基本操作符重载

```
class Point{
public:
    Point(float, float){}
private:
    float m_fxPos;
    float m_fyPos;
};

int main(){
    string s1 = "abc";
    string s2 = "def";
    string s3 = s1 + s2;

    Point a(1.5, 2.2);
    Point b(3.4, 6.6);
    Point c = a + b;        // 这里"+"报错
    return 0;
}
```

重载操作符有一个关键字：**operator**

如果你要重载 " + ",只需要重写这个函数即可**operator+(...)**

如果你要重载 " - ",只需要重写这个函数即可**operator-(...)**

如果你要重载 " * ",只需要重写这个函数即可**operator*(...)**

.....

实现复数的加法(需求)：

```
a + bi
a: 实部
b: 虚部
a=0,b=0: 0
b=0: 实数
a=0: 虚数
```

实现：

用成员函数重载操作符

```
#include <iostream>
#include "complex.h"
using namespace std;

class Complex{
public:
    // 通过构造函数传参
    Complex();                  // 0
    explicit Complex(float);    // 实数
    Complex(float, float);      // 复数
    // "+"操作符重载
    Complex operator+(const Complex &) const;
    // Complex operator-(const Complex &) const;
    void display() const;       // 展示
private:
    float m_real;   // 复数的实部
    float m_imag;   // 复数的虚部
};

Complex::Complex(){
    m_real = 0.0;
    m_imag = 0.0;
}

Complex::Complex(float real){
    m_real = real;
    m_imag = 0.0;
}

Complex::Complex(float real, float imag){
    m_real = real;
    m_imag = imag;
```

```
}

Complex Complex::operator+(const Complex &A) const{
    Complex B;
    B.m_real = A.m_real + this->m_real;
    B.m_imag = A.m_imag + this->m_imag;
    return B;
}

void Complex::display() const{
    cout << m_real << " + " << m_imag << "i" << endl;
}

int main(){
    Complex c1(1.5, 2.2);
    Complex c2(2.7, 2.9);
    Complex c3 = c1 + c2;
    c3.display();

    return 0;
}
```

用顶层函数重载操作符

两种方法：

1. 友元(简单，大幅减少代码量)
2. 用公有方法获取类的私有成员(代码量相对较大)

```
#include <iostream>
#include "complex.h"
using namespace std;

class Complex{
public:
    // 通过构造函数传参
    Complex();                  // 0
    explicit Complex(float);    // 实数
    Complex(float, float);      // 复数
    // "+"操作符重载
    friend Complex operator+(const Complex &) const;
    void display() const;       // 展示

    // 常用但很繁杂的方法
    void setRealValue(float);       // 设置实部数据
    void setImagValue(float);       // 设置虚部数据

    float getRealValue() const;     // 获取实部数据
    float getImagValue() const;     // 获取虚部数据

private:
    float m_real;   // 复数的实部
    float m_imag;   // 复数的虚部
};
```

```
// "+"操作符重载
Complex operator+(const Complex &, const Complex &);

Complex::Complex(){
    m_real = 0.0;
    m_imag = 0.0;
}

Complex::Complex(float real){
    m_real = real;
    m_imag = 0.0;
}

Complex::Complex(float real, float imag){
    m_real = real;
    m_imag = imag;
}

// 使用友元
Complex operator+(const Complex &A, const Complex &B){
    Complex C;
    C.m_real = A.m_real + B.m_real;
    C.m_imag = A.m_imag + B.m_imag;
    return C;
}

// 使用公有方法获取类的私有数据(注意：这个方法和上面的方法不能同时存在，只能使用一个)
Complex operator+(const Complex &A, const Complex &B){
    Complex C;
    C.setRealValue(A.getRealValue() + B.getRealValue());
    C.setImagValue(A.getImagValue() + B.getImagValue());
    return C;
}

void Complex::display() const{
    cout << m_real << " + " << m_imag << "i" << endl;
}

inline void Complex::setRealValue(float real){
    m_real = real;
}

inline void Complex::setImagValue(float imag){
    m_imag = imag;
}

inline float Complex::getRealValue() const{
    return m_real;
}

inline float Complex::getImagValue() const{
    return m_imag;
}

int main(){
    Complex c1(1.5, 2.2);
    Complex c2(2.7, 2.9);
    Complex c3 = c1 + c2;
```

```
    c3.display();

    return 0;
}
```

作业：

实现string类的基本功能 ------- MyString

功能：

1. 重载 +
2. 重载 +=
3. 重载 [] （下标）
4. <<
5. **>>**
6. 计算字符串长度
7. 清空字符串

string s1 = "hello " ;

string s2 = "world";

string s3 = s1 + s2;

s1 += s2;