2024年春

程序设计实习：C++程序设计

第九讲 函数模板与类模板-下

贾川民
北京大学

课前提醒

- 教学网查阅作业解答 (参考郭神标程)
- 魔兽大作业的data文件教学组统一发布
- 输入输出与模板的作业release
- 清明假期：4.7上周五的课

课程回顾

- 泛型程序设计
- **Generic Programming**
- 算法实现时不指定具体要操作的数据的类型
- 泛型 — 算法实现一遍 → 适用于多种数据结构
- **优势:** 减少重复代码的编写
- 大量编写模板, 使用模板的程序设计
  - 函数模板
  - 类模板

课程回顾

- **函数模板**
  - 模板函数的重载
- **类模板**
  - 基本概念
  - 继承
  - 友元
  - static成员
- **String类**

课程回顾

- 用 函数模板 解决

```
template <class 类型参数1, class 类型参数2, … >
返回值类型 模板名 (形参表)
{
        函数体
}
```

课前测一测

- 以下说法不正确的是：

A. 函数模板中可以有不止一个类型参数
B. 函数模板可以重载
C. 函数模板中的类型参数也可以用来表示函数模板的返回值类型
D. 函数模板中的类型参数不能用于定义局部变量

课前测一测

- 以下说法不正确的是：

A. 函数模板中可以有不止一个类型参数
B. 函数模板可以重载
C. 函数模板中的类型参数也可以用来表示函数模板的返回值类型
D. 函数模板中的类型参数不能用于定义局部变量

课前测一测

```
1  template <class T1, class T2, class T3>
2  T1 func(T1 * a, T2 b, T3 c) {   }
3  int a[10], b[10];
4  void f(int n) {  }
```

func(a, b, f); 将模板类型参数实例化的结果是：

- A T1: int, T2: int *, T3: void (*) (int)
- B T1: int *, T2: int *, T3: void (*) (int)
- C T1: int, T2: int, T3: void (int)
- D T1: int *, T2 int, T3: void (int)

课前测一测

```
1  template <class T1, class T2, class T3>
2  T1 func(T1 * a, T2 b, T3 c) {   }
3  int a[10], b[10];
4  void f(int n) {  }
```

func(a, b, f); 将模板类型参数实例化的结果是：

- A T1: int, T2: int *, T3: void (*) (int)
- B T1: int *, T2: int *, T3: void (*) (int)
- C T1: int, T2: int, T3: void (int)
- D T1: int *, T2 int, T3: void (int)

模板

- 函数模板
  - 模板函数的重载
- 类模板
  - 基本概念
  - 继承
  - 友元
  - static成员
- String类

类模板—问题的提出

- 为了**多快好省**地定义出一批相似的类，可以定义**类模板**
- 由类模板 → 生成不同的类
- 数组是一种常见的数据类型，元素可以是：
  - 整数
  - 学生
  - 字符串 …
- 考虑一个**数组类**，需要提供的基本操作
  - **len():** 查看数组的长度
  - **getElement(int index):** 获取其中的一个元素
  - **setElement(int index):** 对其中的一个元素进行赋值

  …

类模板—问题的提出

- 这些数组类，除了**元素的类型**不同之外，其他的完全相同

- **类模板**
  - 在定义类的时候给它一个/多个参数，这个/些参数表示不同的数据类型
  - 在调用类模板时，指定参数，由编译系统根据参数提供的数据类型自动产生相应的**模板类**

类模板的定义

```
template <class T>   //类模板的首部, 声明类模板的参数
class CArray{
    T *ptrElement;
    int size;
public:
    CArray(int length);
    ~ CArray();
    int len();
    void setElement(T arg, int index);
    T getElement(int index);
 };
```

template:

- CArray是一个**类模板**
- 声明一个或多个**类型参数**, 用来定义CArray的属性类型, 成员函数的参数类型和返回值类型

类模板的定义

C++的类模板的写法如下：

```
template <类型参数表>
class 类模板名
{
    成员函数和成员变量
};
```

类型参数表的写法就是：

```
class 类型参数1, class 类型参数2, …
```

类模板的定义

- 类模板里的成员函数，在类模板外定义时：

```
template <类型参数表>
返回值类型 类模板名<类型参数名列表>::成员函数名(参数表){
        …
}
```

- 用类模板定义对象：

```
类模板名 <真实类型参数表> 对象名(构造函数实际参数表);
```

- 如果类模板有无参构造函数，可以只写：

```
类模板名 <真实类型参数表> 对象名;
```

类模板的定义:示例

Pair类模板:

```
template <class T1, class T2>
class Pair {
   public:
      T1 key;     //关键字
      T2 value;  //值
      Pair(T1 k, T2 v):key(k), value(v) { }
      bool operator < ( const Pair<T1, T2> & p) const;
};
template<class T1, class T2>
bool Pair<T1, T2>::operator < ( const Pair<T1, T2> & p) const
//Pair的成员函数 operator <
{   return key < p.key;    }
```

类模板的定义:示例

Pair类模板:

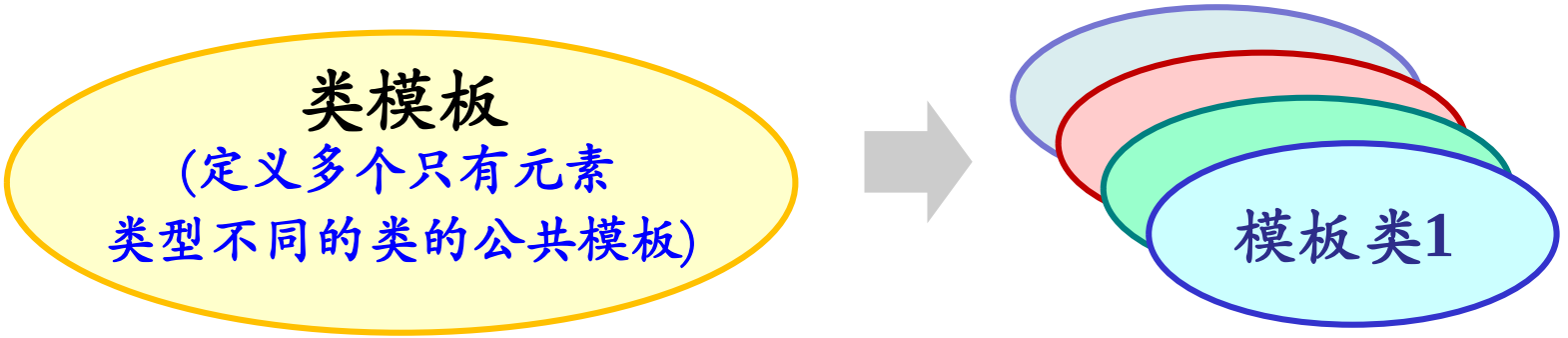
```
int main(){
    Pair<string, int> student("Tom", 19); //实例化出一个类
                                          // Pair<string, int>
    cout << student.key << " " << student.value;
    return 0;
}
```

输出结果:

Tom 19

模板类的概念

- **模板类** — 为**类模板**中各类型参数指定了具体的数据类型后，即得到一个**模板类**
  - 编译系统自动用具体的数据类型替换类模板中的类型参数，生成模板类的代码
  - 为类型参数指定的数据类型不同，得到的模板类不同
- **模板类 = 实例化的类模板**



使用类模板声明对象

- CArray<int> arrayInt(50), *ptrArrayInt;

  //创建一个元素类型为int的CArray模板类
  //声明该模板类的一个对象以及一个指针

- CArray<string> arrayStr(100);

  //创建一个元素类型为string的CArray模板类
  //声明该模板类的一个对象

  //其中string是C++标准类库中的字符串类

- CArray<CStudent> *ptrArrayStudent;

  //创建一个元素类型为CStudent的CArray模板类
  //声明该模板类的一个指针

  //其中CStudent是程序员自定义的一个类

使用类模板声明对象

- 同一个类模板的两个模板类是不兼容的

```
Pair<string, int> * p;
Pair<string, double> a;
p = & a;    //wrong
```

定义类模板的成员函数

- 定义类模板的成员函数

```
template <class T>   // T是模板类CArray<T>的类型参数
CArray<T>:: CArray(int length) { //模板类CArray<T>的构造函数
    ptrElement = new T[length];
    size = length;
}
template <class T>   // T是模板类CArray<T>的类型参数
CArray<T>:: ~ CArray(){  //模板类CArray<T>的析构函数
    Delete [] ptrElement;
}
```

定义类模板的成员函数

```
template <class T>          // T是模板类CArray<T>的类型参数
int CArray<T>:: len(){   //模板类CArray<T>的成员函数 len()
      return size;
}
template <class T>
void CArray<T>:: setElement(T arg, int index)
    *(ptr+index) = arg;
      return;
}
template <class T>
T CArray<T>:: getElement(int index) {
      return *(ptr+index);
}
```

函数模版作为类模板成员

```
#include <iostream>
using namespace std;
template <class T>
class A{
public:
        template<class T2>
        void Func(T2 t) {    cout << t;    }    //成员函数模板
};
int main()  {
        A<int> a;
        a.Func('K');    //成员函数模板 Func被实例化
        a.Func("Hello"); //成员函数模板 Func被实例化
        return 0;
}
```

类模板中的成员函数可以是一个函数模板，该成员函数只有在被调用时才会被实例化

函数模版作为类模板成员

```
#include <iostream>
using namespace std;
template <class T>
class A{
public:
        template<class T2>
        void Func(T2 t) {    cout << t;    }   //成员函数模板
};
int main()  {
        A<int> a;
        a.Func('K');    //成员函数模板 Func被实例化
        a.Func("Hello"); //成员函数模板 Func被实例化
        return 0;
}
```

若函数模板改为
template <class T>
void Func(Tt){ cout<<t; }
将报错 "declaration of 'class T' shadows template parm 'class T' "

类模板中的成员函数可以是一个函数模板，该成员函数只有在被调用时才会被实例化

类模板与非类型参数

- 类模板的 <**类型参数表**>中可以包括**非类型参数**

(non-type parameter)

  - **非类型参数:**

    用来说明类模板中的**属性**

  - **类型参数:**

    用来说明类模板中的**属性类型, 成员函数的参数类型和返回值类型**

类模板与非类型参数

- 类模板的 <**类型参数表**>中可以包括**非类型参数**

(non-type parameter)

  - **非类型参数:**

    用来说明类模板中的**属性**

  - **类型参数:**

    用来说明类模板中的**属性类型, 成员函数的参数类型和返回值类型**

类模板与非类型参数

- 类模板的 < **类型参数表**> 中可以出现**非类型参数**：

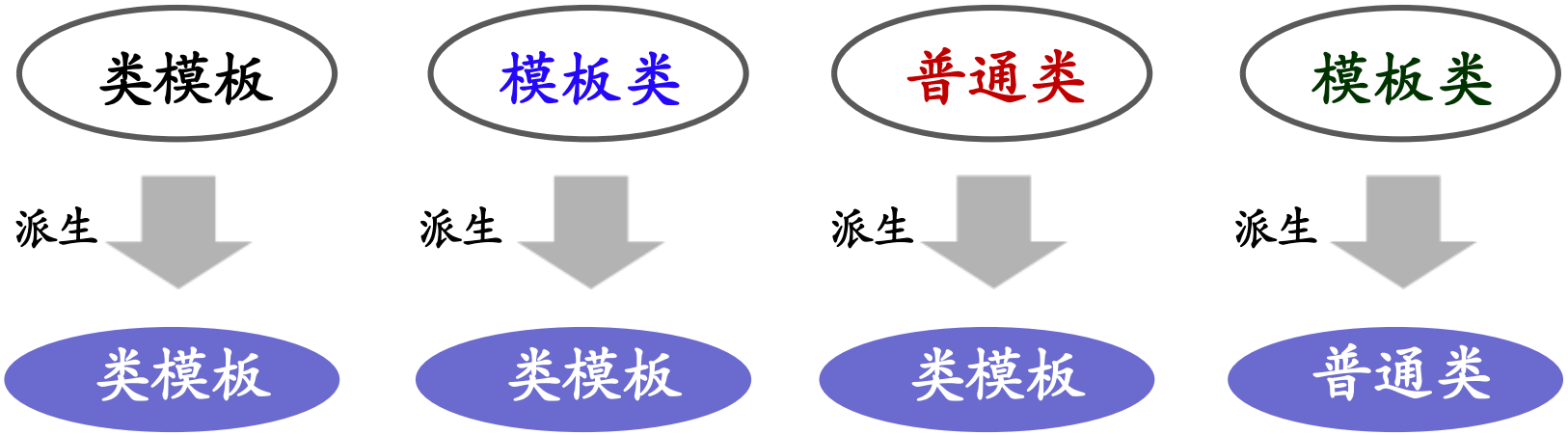
```
1  template <class T, int size>
2  class CArray{
3    T  array[size];
4  public:
5    void Print() {
6      for(int i = 0; i < size; ++i)
7        cout << array[i] << endl;
8    }
9  };
10 CArray<double,40> a2;
11 CArray<int,50> a3;
```

a2 和a3 属于不同的类

- 通常类模板参数声明中的非类型参数可以提高程序的执行效率
  - 在编译或链接期间即可确定参数的值

类模板与派生

- 类模板派生出类模板
- 模板类 (即类模板中类型/非类型参数实例化后的类)
  派生出类模板
- 普通类派生出类模板
- 模板类派生出普通类



(1) 类模板从 类模板 派生

```
template <class T1, class T2>
class A {
  T1 v1; T2 v2;
};
template <class T1, class T2>
class B: public A<T2, T1>  {
  T1 v3; T2 v4;
};
```

```
class B<int, double>:public A<double, int>{
    int v3; double v4;
};
class A<double, int> {
    double v1; intv2;
};
```

```
template <class T>
class C: public B<T,T>{
   T v5;
};
int main(){
   B<int, double> obj1;
   C<int> obj2;
   return 0;
}
```

(2) 类模板从 模板类 派生

```
template <class T1, class T2>
class A {
    T1 v1;
    T2 v2;
};
template <class T>
class B: public A <int, double> {
    T v;
};
int main()  {
    B<char> obj1; //自动生成两个模板类: A<int, double>和B<char>
    return 0;
}
```

(3) 类模板从 普通类 派生

```
class A {
    intv1;
};

template <class T>
class B: public A { //所有从B实例化得到的类, 都以A为基类
    T v;
};

int main() {
   B<char> obj1;
   return 0;
}
```

(4) 普通类从 模板类 派生

```
template <class T>
class A {
    T v1;
    int n;
};
class B : public A<int> {
    double v;
};
int main()  {
    B obj1;
    return 0;
}
```

类模板与友元函数

- 函数 / 类 / 类的成员函数作为类模板的友元
- 函数模板作为类模板的友元
- 函数模板作为类的友元
- 类模板作为类模板的友元

(1) 函数/类/类的成员函数作为类模板的友元

```
void Func1() {   }   //函数
class A {   };          //类
class B{
    public:
        void Func() {   } //类的成员函数
};
template <class T>
class Tmpl {
    friend void Func1();
    friend class A;
    friend void B::Func();
}; //任何从Tmpl实例化来的类, 都有以上三个友元
```

```
int main(){
    Tmpl<int> i;
    Tmpl<double> f;
    return 0;
}
```

(2) 函数模板作为类模板的友元

```
#include <iostream>
#include <string>
using namespace std;
template <class T1, class T2>
class Pair{
    T1 key;     //关键字
    T2 value;  //值
public:
    Pair(T1 k, T2 v):key(k), value(v) { };
    bool operator < ( const Pair<T1,T2> & p) const;
    template <class T3, class T4>
    friend ostream & operator<<(ostream & o, const
                                Pair<T3,T4> & p); //函数模版
};
```

(2) 函数模板作为类模板的友元

```
template<class T1, class T2>
bool Pair<T1,T2>::operator < ( const Pair<T1,T2> & p) const{
// " 小" → 关键字小
        return key < p.key;
}

template <class T3, class T4>
ostream & operator<< (ostream & o, const Pair<T3,T4> & p){
        o << "(" << p.key << "," << p.value << ")" ;
        return o;
}
```

(2) 函数模板作为类模板的友元

```
int main(){
        Pair<string, int> student("Tom", 29);
        Pair<int, double> obj(12, 3.14);
        cout << student << " " << obj;
        return 0;
}
```

输出结果:
(Tom, 29) (12, 3.14)

任意从

```
template <class T3, class T4>
ostream & operator<< (ostream & o, const Pair<T3,T4> & p)
```

生成的函数, 都是任意Pair 模板类的友元

(3) 函数模板作为类的友元

```
#include <iostream>
using namespace std;
class A {
    intv;
public:
    A(int n):v(n) { }
    template <class T>
    friend void Print(const T & p);
};
template <class T>
void Print(const T & p){  cout << p.v;  }
```

```
int main(){
    A a(4);
    Print(a);
    return 0;
}
```

输出结果:
4

所有从
template <class T>
void Print(const T & p)
生成的函数, 都成为A的友元

(4) 类模板作为类模板的友元

```
#include <iostream>
using namespace std;
template <class T>
class A {
public:
     void Func( const T & p )
     {    cout << p.v;    }
};
template <class T>
class B {
    T v;
public:
    B(T n):v(n) { }
    template <class T2>
    friend class A;  //把类模板A声明为友元
};
```

```
int main() {
     B<int> b(5);
     A< B<int> > a;
//用B<int>替换A模板中的T
     a.Func (b);
     return 0;
}
```

输出结果:
5
A< B<int> >类
→B<int>类的友元

类型参数的影响

```
template<class T>
class myClassA {
};

template<class T>
class myClassB {
};
```

```
template <class T1, class T2, int size>
class CTemp{
    friend myClassA<T1>;
    friend myClassB<T2>;
    private:
        T1 elements[size];
    public:
        CTemp(){};
};
```

myClassA<int>是CTemp<int, double, 50>的友元类

myClassB<double>是CTemp<int, double, 50>的友元类

myClassA<double>不是CTemp<int, double, 50>的友元类

myClassA<char>不是CTemp<int, double, 50>的友元类

类模板与static成员(1)

- 类模板中可以定义**静态成员**，那么从**该类模板实例化得到的模板类的所有对象**，都包含同样的静态成员

```
#include <iostream>
using namespace std;
template <class T>
class A {
    static int count;
public:
    A() { count ++;  }
    ~A() { count -- ; };
    A( A & ) { count ++ ; }
    static void PrintCount() { cout << count << endl; }
};
```

类模板与static成员(2)

```
template< > int A<int>::count = 0;
template< > int A<double>::count = 0;
int main() {
    A<int> ia;
    A<double> da;
    ia.PrintCount();
    da.PrintCount();
    return 0;
}
```

输出结果:

```
1
1
```

A<int>和A<double>是不同的模板类，不能共享静态变量count，因而需要分别初始化

类模板与static成员(2)

```
template< > int A<int>::count = 0;
template< > int A<double>::count = 0;
int main() {
    A<int> ia;
    A<double> da;
    ia.PrintCount();
    da.PrintCount();
    return 0;
}
```

输出结果:

```
1
1
```

A<int>和A<double>是不同的模板类，不能共享静态变量count，因而需要分别初始化

小结

- 函数模板
  - 函数重载 *vs.* 函数模板
  - 类型参数
  - 函数和模板的匹配顺序
- 类模板
  - 类模板 & 模板类
  - 使用类模板声明对象：同一个类模板的两个模板类是不兼容
  - 类模板的非类型参数
  - 类模板与继承：4种情况
  - 类模板与友元函数：4种情况
  - 类模板与静态变量：不同的模板类不能共享静态变量

模板

- 函数模板
  - 模板函数的重载
- 类模板
  - 基本概念
  - 继承
  - 友元
  - static成员
- **String类**

String类

- string类
- string的赋值与连接
- 比较string
- 子串
- 交换string
- string的特性
- 在string中寻找，替换和插入字符
- 转换成C语言式char *字符串
- 字符串流处理

String类

- **string类**是一个**模板类**，它的定义如下：

  typedef basic_string<char> string;

- 使用string类要包含头文件 **#include <string>**
- string对象的**初始化:**
  - **strings1("Hello");** // 一个参数的构造函数
  - **strings2(8, 'x');** // 两个参数的构造函数
  - **string month = "March";**

String类

- 类中不提供以字符和整数为参数的构造函数

  错误的初始化方法:

  - string error1 = 'c'; // 错
  - string error2('u'); // 错
  - string error3 = 22; // 错
  - string error4(8); // 错

- 可以将字符赋值给string对象
  - s = 'n';

String类 程序样例

```
#include <iostream>
#include <string>
using namespace std;
int main(int argc, char* argv[ ]){
    string s1("Hello");
    cout << s1 << endl;
    string s2(8, 'x');
    cout << s2 << endl;
    string month = "March";
    cout << month << endl;
    string s;
    s='n';
    cout << s << endl;
    return 0;
}
```

输出：

Hello

xxxxxxxx

March

n

String类

- 构造的string太长而无法表达时会抛出length_error异常
- string 对象的长度用**成员函数 length()**读取

```
string s("hello");
cout << s.length() << endl;
```

- string 支持流读取运算符

```
string stringObject;
cin >> stringObject;
```

- string 支持getline函数

```
string s;
getline(cin, s);
```

string 的赋值和连接

- 用 = 赋值

```
string s1("cat"), s2;
s2 = s1;
```

- 用 assign 成员函数复制

```
string s1("cat"), s3;
s3.assign(s1);
```

- 用 assign 成员函数部分复制

```
string s1("catpig"), s3;
s3.assign(s1, 1, 3);
//从s1中下标为1的字符开始复制3个字符给s3
```

string 的赋值和连接

- 单个字符复制

  s2[5] = s1[3] = ‘a’;

- 逐个访问string对象中的字符

```
string s1("Hello");
for(inti=0; i<s1.length(); i++)
    cout << s1.at(i) << endl;
```

- 成员函数at会做范围检查，如果超出范围，会抛出out_of_range异常

  下标运算符不做范围检查

  可以自己写个验证程序，观察两者的区别

string 的赋值和连接

- 用 + 运算符连接字符串

```
string s1("good "), s2("morning!");
s1 += s2;
cout << s1;
```

- 用 成员函数 append 连接字符串

```
string s1("good "), s2("morning!");
s1.append(s2);
cout << s1;
s2.append(s1, 3, s1.size());  //s1.size(), s1字符数
cout << s2;
```

// 下标为3开始, s1.size()个字符, 如果字符串内没有足够字符, 则复制到字符串最后一个字符

比较string

- 用关系运算符比较string的大小

  == ， >, >=, <, <=, !=

  返回值都是bool类型，成立返回true，否则返回false

  例如：

```
string s1("hello"), s2("hello"), s3("hell");
bool b = (s1 == s2);
cout << b << endl;
b = (s1 == s3);
cout << b << endl;
b = (s1 > s3);
cout << b << endl;
```

输出：

```
1
0
1
```

比较string

- 用**成员函数compare**比较string的大小

```
string s1("hello"), s2("hello"), s3("hell");

int f1 = s1.compare(s2);
int f2 = s1.compare(s3);
int f3 = s3.compare(s1);

int f4 = s1.compare(1, 2, s3, 0, 3); //s1 1-2; s3 0-3
int f5 = s1.compare(0, s1.size(), s3); //s1 0-end

cout << f1 << endl << f2 << endl << f3 << endl;
cout << f4 << endl << f5 << endl;
```

比较string

- 输出

```
0    // hello == hello
1    // hello > hell
-1   // hell < hello
-1   // el < hell
1    // hello > hell
```

子串

- 成员函数 substr

```
string s1("helloworld"), s2;
s2 = s1.substr(4, 5);  //下标4开始，共5个字符
cout << s2 << endl;
```

输出：

```
owor
```

交换string

- 成员函数 swap

```
string s1("helloworld"), s2("really");
s1.swap(s2);
cout << s1 << endl;
cout << s2 << endl;
```

输出:

```
really
helloworld
```

string的特性

- **成员函数 capacity()**

  返回无需增加内存即可存放的字符数

- **成员函数max_size()**

  返回string对象可存放的最大字符数

- **成员函数length()**和**size()**相同

  返回字符串的大小/长度

- **成员函数empty()**

  返回string对象是否为空

- **成员函数resize()**改变string对象的长度

string的特性

```
string s1("helloworld");
cout << s1.capacity() << endl;
cout << s1.max_size() << endl;
cout << s1.size() << endl;
cout << s1.length() << endl;
cout << s1.empty() << endl;
cout << s1 << "aaa" << endl;
```

```
22                          // capacity
4611686018427387897         // maximum size
11                          // length
11                          // size
0                           // empty
helloworldaaa               // string itself and "aaa"
```

string的特性

```
s1.resize(s1.length()+10);
cout << s1.capacity() << endl;
cout << s1.max_size() << endl;
cout << s1.size() << endl;
cout << s1.length() << endl;
cout << s1 << "aaa" << endl;
s1.resize(0);
cout << s1.empty() << endl;
```

```
22
4611686018427387897
21
21
helloworldaaa
1
```

寻找string中的字符

- **成员函数 find()**

  string s1("helloworld");
  s1.find("lo");

  - 在s1中从前**向后**查找 "lo" 第一次出现的地方，
    如果找到，返回 "lo"开始的位置，即 l 所在的位置下标；
    如果找不到，返回 string::npos (string中定义的静态常量)

- **成员函数 rfind()**

  string s1("helloworld");
  s1.rfind("lo");

  - 在s1中从后**向前**查找 "lo" 第一次出现的地方，
    如果找到，返回 "lo"开始的位置，即 l 所在的位置下标；
    如果找不到，返回 string::npos

寻找string中的字符

- **成员函数 find_first_of()**

  string s1("helloworld");

  s1.find_first_of("abcd");

  - 在s1中**从前向后**查找 "abcd" 中任何一个字符**第一次**出现的地方, 如果找到, 返回找到字母的位置;

    如果找不到, 返回 string::npos

- **成员函数 find_last_of()**

  string s1("helloworld");
  s1.find_last_of("abcd");

  - 在s1中查找 "abcd" 中任何一个字符**最后一次**出现的地方, 如果找到, 返回找到字母的位置;

    如果找不到, 返回 string::npos

寻找string中的字符

- **成员函数 find_first_not_of()**

  string s1("helloworld");

  s1.find_first_not_of("abcd");

  - 在s1中**从前向后**查找**不在** "abcd" 中的字母**第一次**出现的地方，如果找到，返回找到字母的位置；如果找不到，返回string::npos

- **成员函数 find_last_not_of()**

  string s1("helloworld");

  s1.find_last_not_of("abcd");

  - 在s1中**从后向前**查找**不在** "abcd" 中的字母**第一次**出现的地方，如果找到，返回找到字母的位置；如果找不到，返回string::npos

寻找string中的字符

```
string s1("hello worlld");
cout << s1.find("ll") << endl;
cout << s1.find("abc") << endl;
cout << s1.rfind("ll") << endl;
cout << s1.rfind("abc") << endl;
cout << s1.find_first_of("abcde") << endl;
cout << s1.find_first_of("abc") << endl;
cout << s1.find_last_of("abcde") << endl;
cout << s1.find_last_of("abc") << endl;
cout << s1.find_first_not_of("abcde") << endl;
cout << s1.find_first_not_of("helloworld") << endl;
cout << s1.find_last_not_of("abcde") << endl;
cout << s1.find_last_not_of("helloworld") << endl;
```

输出：

```
2
4294967295
9
4294967295
1
4294967295
11
4294967295
0
4294967295
10
4294967295
```

替换string中的字符

- 成员函数erase()

```
string s1("hello worlld");
s1.erase(5);
cout << s1;
cout << s1.length();
cout << s1.size();
```

// 去掉下标5及之后的字符

输出:

```
hello
5
5
```

替换string中的字符

- 成员函数find()

```
string s1("hello worlld");
cout << s1.find("ll", 1) << endl;
cout << s1.find("ll", 2) << endl;
cout << s1.find("ll", 3) << endl;
// 分别从下标1, 2, 3开始查找 "ll"
```

输出:

```
2
2
9
```

替换string中的字符

- 成员函数 replace()

```
string s1("helloworld");
s1.replace(2, 3, "haha");
cout << s1;
```

//将s1中下标2 开始的3个字符换成"haha"

输出:

hehaha world

替换string中的字符

- 成员函数 replace()

```
string s1("helloworld");
s1.replace(2, 3, "haha", 1, 2);
cout << s1;
```

```
// 将s1中下标2 开始的3个字符换成
// "haha" 中下标1开始的2个字符
```

输出:

```
heah world
```

在string中插入字符

- 成员函数insert()

```
string s1("helloworld");
string s2("show insert");
s1.insert(5, s2);   //将s2插入s1下标5的位置
cout << s1 << endl;
s1.insert(2, s2, 5, 3);
//将s2中下标5开始的3个字符插入s1下标2的位置
cout << s1 << endl;
```

输出：

```
helloshow insert world
heinslloshow insert world
```

转换成C语言式char *字符串

- 成员函数 c_str()

```
string s1("helloworld");
printf("%s\n", s1.c_str());
```

// s1.c_str() 返回传统的const char * 类型字符串

//并在末尾增加了一个‘\0’

输出:

helloworld

转换成C语言式char *字符串

- 成员函数copy()

```
string s1("helloworld");
int len = s1.length();
char * p2 = new char[len+1];
s1.copy(p2, 5, 0);
p2[5]=0;
cout << p2 << endl;
```

// s1.copy(p2, 5, 0) 从s1的下标0的字符开始制作一个最长5个
//字符长度的字符串副本并将其赋值给p2
//返回值表明实际复制字符串的长度

输出:

hello

字符串流处理

- 除了标准流和文件流输入输出外，还可以从string进行输入输出
- 类似 istream和osteram进行标准流输入输出，

  用**istringstream** 和 **ostringstream**进行字符串上的输入输出，也称为**内存输入输出**

```
#include <string>
#include <iostream>
#include <sstream>
```

字符串流处理

- 例: 字符串输入流

```
string input("Input test 123 4.7 A");
istringstream inputString(input);
string string1, string2;
int i;
double d;
char c;
inputString >> string1 >> string2 >> i >> d >> c;
cout << string1 << endl << string2 << endl;
cout << i << endl << d << endl << c <<endl;
long l;
if(inputString >> l)    cout << "long\n";
else    cout << "empty\n";
```

输出:

```
Input
test
123
4.7
A
empty
```

字符串流处理

- 例: 字符串输出流

```
string input("Output test 123 4.7 A");
istringstream inputString(input);
string string1, string2;
int i;
double d;
char c;
inputString >> string1 >> string2 >> i >> d >> c;
ostringstream outputString;
outputString << string1 << endl << string2 << endl;
outputString << i << endl << d << endl << c <<endl;
cout << outputString.str();
```

输出:

```
Output
test
123
4.7
A
```

Thanks!