

2018年春

程序设计实习(I): C++程序设计

第八讲 类模板和函数模板

刘家瑛

liujiaying@pku.edu.cn



模 板

□ 函数模板

- 模板函数的重载

□ 类模板

- 继承
- static成员
- 友元

□ String类



模板的基本概念

□ 排序问题

■ 实际应用中经常遇到的问题

- 对学生按成绩排序
- 对事件按发生的时间排序
- 对产品按销售量和顾客的投诉率排序

■ 采用的排序算法完全相同



模板的基本概念

□ 排序问题

■ 不同类型的对象, 排序时的不同: 比较两个对象的大小

- 函数sortA用于对类A的一组对象排序
- 函数sortB用于对类B的一组对象排序
- 类A和类B都重载了关系运算符 ">"
- sortA和sortB中, 除了被排序数组, 存储被排序数组元素的变量的类型声明不同之外, 其它部分可以完全相同



模板的基本概念

有两种解决办法:

- 函数重载
- 函数模板



□ 函数重载的办法

```
sort(...classA ...){...}
```

```
sort(...classB ...){...}
```

- 分别写两个同名的函数

- 由编译系统根据函数调用时实参的类型, 确定实际执行哪个函数

□ 函数模板

```
template<class T>
```

```
return-type sort(...T...)
```

- 由编译系统根据sort函数调用时实参的类型, 自动生成相应的模板函数



输出全部数组元素的函数模板

template<class T>

函数模板的类型参数

void print(const T array[], int size){

int i;

for (i =0; i<size; i++) cout<<array[i];

return;

}

CStudent undergraduates[number1], graduates[number2];

string telephoneNumbers[numbers3];

...

print(undergraduates, number1);

...

print(telephoneNumbers, number3);

...

print(graduates, number2);

...




```
void print( const CStudent array[], int size){  
    int i;  
    for ( i =0; i<size; i++) cout<<array[i];  
    return;  
} // 编译到print(undergraduates, number1)时自动产生
```

```
void print( const string array[], int size){  
    int i;  
    for ( i =0; i<size; i++) cout<<array[i];  
    return;  
} // 编译到print(telephoneNumbers, number3)时自动产生
```

要能编译通过，要求对 << 有适当的重载



一个函数模板可以有多个类型参数

```
template<class T1, class T2>  
void print(T1 arg1, T2 arg2)  
    { cout<<arg1<<" "<<arg2<<endl; return; }
```

- 函数模板的参数类型

- 可以用类型参数说明
- 也可以用基本数据类型, 其他的类说明

```
template<class T1, class T2>  
void print(T1 arg1, T2 arg2, string s, int k)  
{ cout<<arg1<<s<<arg2<<k<<endl; return; }
```



- 函数模板的类型参数可以用于函数模板的局部变量声明

```
template<class T1, class T2>  
void print(T1 arg1, T2 arg2)  
{  
    T1 locVar=arg1;  
    cout<<locVar<<" "<<arg2<<endl;  
    return;  
}
```



- 赋值兼容原则引起函数模板中类型参数的二义性

template<class T>

T myFunction(T arg1, T arg2)

{ cout<<arg1<<“ ”<<arg2<<“\n”; return arg1;} ...

myFunction(5, 7); //ok: replace T with int

myFunction(5.8, 8.4); //ok: replace T with double

myFunction(5, 8.4); //error: replace T with int or double? 二义性

- 可以在函数模板中使用多个类型参数, 可以避免二义性

template<class T1, class T2>

T1 myFunction(T1 arg1, T2 arg2)

{ cout<<arg1<<“ ”<<arg2<<“\n”; return arg1;} ...

myFunction(5, 7); //ok: replace T1 and T2 with int

myFunction(5.8, 8.4); //ok: replace T1 and T2 with double

myFunction(5, 8.4); //ok: replace T1 with int, T2 with double



重载与函数模板

- 函数模板与函数模板的重载: 同一函数名, 参数的数量不同

```
template<class T>
```

```
T myFunction( T arg )
```

```
{ cout<< “one argument\n”; return arg1;}
```

```
template<class T1, class T2>
```

```
T1 myFunction( T1 arg1, T2 arg2 )
```

```
{ cout<< “two arguments\n”; return arg1;}
```

...

```
myFunction(5); //ok: replace with int
```

```
myFunction(5.8, 8.4); //ok: replace T1 and T2 with double
```

```
myFunction(5, 8.4); //ok: replace T1 with int, T2 with double
```



函数和模板的匹配顺序

- 1) 先找一个参数完全匹配的函数
- 2) 再找一个参数完全匹配的模板
- 3) 在没有二义性的前提下, 再找一个参数经过自动转换后能够匹配的函数
- 4) 都找不到, 则报错



```

template <class T>
T Max( T a, T b ) {
    cout << "TemplateMax" << endl;
    return 0;
}
double Max(double a, double b){
    cout << "MyMax" << endl;
    return 0;
}
main(){
    int i=4, j=5;
    Max(1.2, 3.4); //输出MyMax
    Max(i, j);     //输出TemplateMax
    Max(1.2, 3);  //二义性, 强制类型转换3为double, 调用Max函数
}

```



例：函数模板调用顺序

```
template <class T>
```

```
T Max(T a, T b){
```

```
    cout << "Template Max 1" <<endl;
```

```
    return 0;
```

```
}
```

```
template <class T, class T2>
```

```
T Max(T a, T2 b){
```

```
    cout << "Template Max 2" <<endl;
```

```
    return 0;
```

```
}
```




```

double Max(double a, double b){
    cout << "MyMax" << endl;
    return 0;
}
int main()
{
    int i=4, j=5;
    Max(1.2, 3.4); //调用Max(double, double)函数
    Max(i, j);     //调用第一个T Max(T a, T b)模板生成的函数
    Max(1.2, 3);   //调用第二个T Max(T a, T2 b)模板生成的函数
    return 0;
}

```

运行结果:

MyMax

Template Max 1

Template Max 2



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**



函数模板

- 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

}



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, 拿到类模板定义外面写时的语法如下:

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){

...

}

用类模板定义对象的写法如下:

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, 那么也可以只写:

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;
}
```

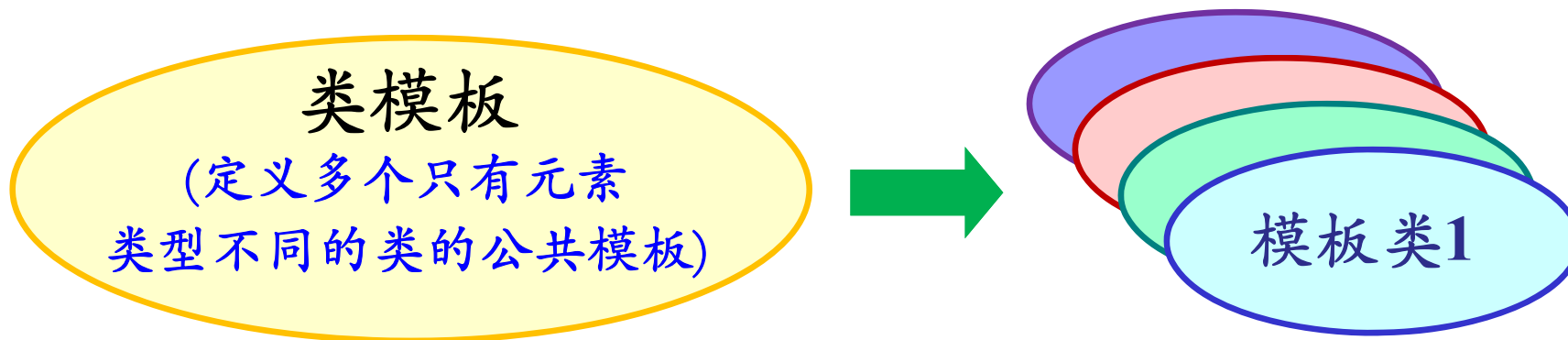
输出结果：

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
```

```
    return 0;
```

```
}
```

程序输出:

K

若函数模板改为

```
template <class T>
```

```
void Func(T t){cout<<t}
```

将报错 "declaration of 'class T'

shadows template parm 'class T' "



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

```
template <class T, int elementsNumber>
class CArray{
    //T *ptrElement;
    //int size;
    T elements[elementsNumber];
public:
    //Carray(int length);
    //~Carray();
    int len();
    void setElement(T arg, int index);
    T getElement(int index);
};
```

- 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



类模板与非类型参数

```
template <class T, int elementsNumber>
int CArray<T, elementsNumber>::len(){
    return elementsNumber;
}
```

```
template <class T, int elementsNumber>
void CArray <T, elementsNumber>::setElement(T arg, int index) {
    elements[index] = arg;
    return;
}
```

```
template <class T, int elementsNumber>
    T CArray<T, elementsNumber>::getElement(int index) {
    return elements[index] ;
}
```



CArray<int, 50 > arrayIntA, *ptrArrayInt;

//创建一个有50个int型元素的CArray模板类

// arrayIntA有50个int型元素

// ptrArrayInt所指向的对象需要有50个int型元素

CArray<int, 100 > arrayIntB;

//创建一个有100个int型元素的CArray模板类

// arrayIntB有100个int型元素

CArray<string, 50> arrayStr;

//创建一个有50个string型元素的CArray模板类

// arrayStr有50个string型元素

CArray<CStudent, 135> *ptrArrayStudent;

//创建一个有135个CStudent型元素的CArray模板类

// ptrStudent所指向的对象需要有135个CStudent型元素

ptrArrayInt = &arrayIntA; //ok

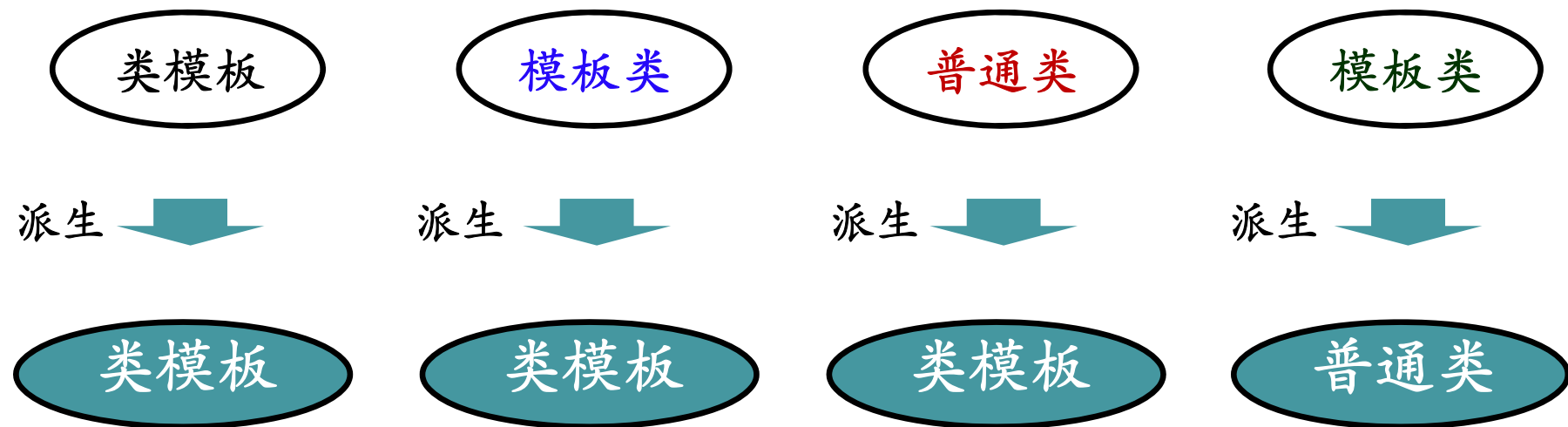
ptrArrayInt = &arrayIntB; //error

ptrArrayInt = &arrayStr; //error



类模板与继承

- 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; int v2;
};
```

```
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; return 0; }
```

自动生成两个模板类：A<int, double>和B<char>



(3) 类模板从普通类派生

```
class A { int v1; };
```

```
template <class T>
```

```
class B:public A { 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() { } //类的函数
};
template <class T>
class Tmpl {
    friend void Func1();
    friend class A;
    friend void B::Func();
};
```

```
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
};
```



```

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 T1, class T2>
ostream & operator<< (ostream & o, const Pair<T1,T2> & p){
    o << "(" << p.key << "," << p.value << ")" ;
    return o;
}
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)



(3) 函数模板作为类的友元

```
#include <iostream>
using namespace std;
class A {
    int v;
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



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

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



总结

□ 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：

■ `string s1("Hello");` // 一个参数的构造函数

■ `string s2(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(int i=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("hello world"), s2;
```

```
s2 = s1.substr(4, 5); // 下标4开始, 共5个字符
```

```
cout << s2 << endl;
```

输出:

o wor



交换 string

□ 成员函数 swap

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

输出:

really

hello world



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 **maximum_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("hello world");  
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  
hello worldaaa // 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

hello worldaaa

1



寻找string中的字符

□ 成员函数 find()

```
string s1("hello world");
```

```
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 1 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("hello world");
```

```
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 1 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("hello world");
```

```
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("hello world");
```

```
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("hello world");
```

```
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("hello world");
```

```
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 world");  
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("hello world") << endl;  
cout << s1.find_last_not_of("abcde") << endl;  
cout << s1.find_last_not_of("hello world") << 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 world");
```

```
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("hello world");
```

```
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("hello world");  
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("hello world");
```

```
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("hello world");
```

```
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:

hello world



转换成C语言式char *字符串

□ 成员函数copy()

```
string s1("hello world");
```

```
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ostream进行标准流输入输出,用 istreamstringstream 和 ostreamstringstream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

