文件操作

郭 炜　刘家瑛

北京大学

数据的层次

- 位　　bit
- 字节　byte
- 域/记录:

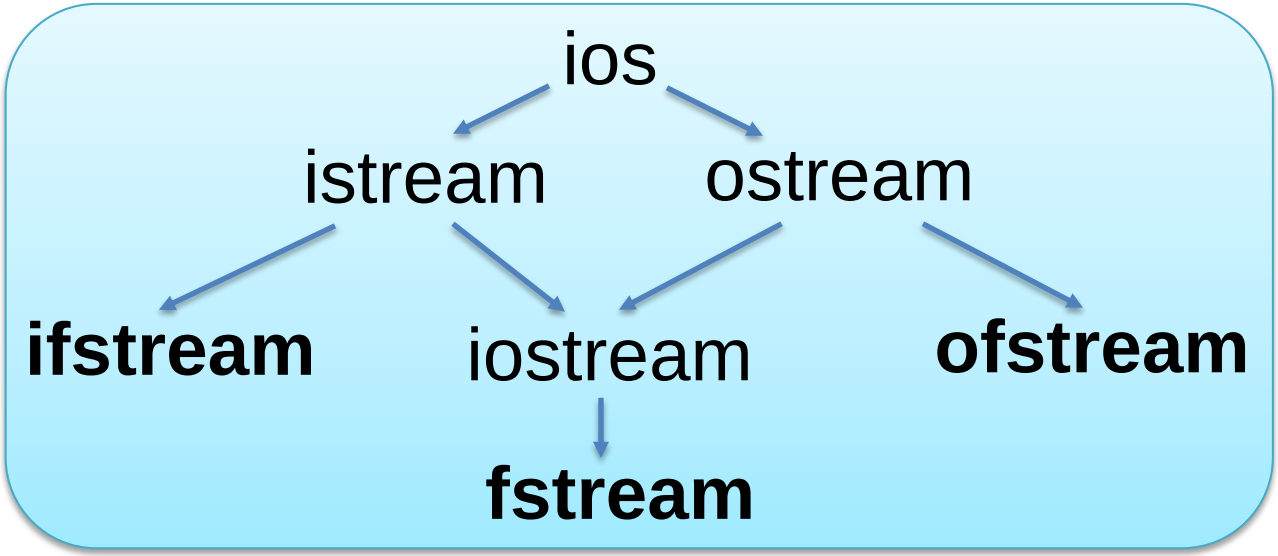
```
例如: 学生记录
    int ID;
    char name[10];
    int age;
    int rank[10];
```

- 将所有记录顺序地写入一个文件 → 顺序文件

文件和流

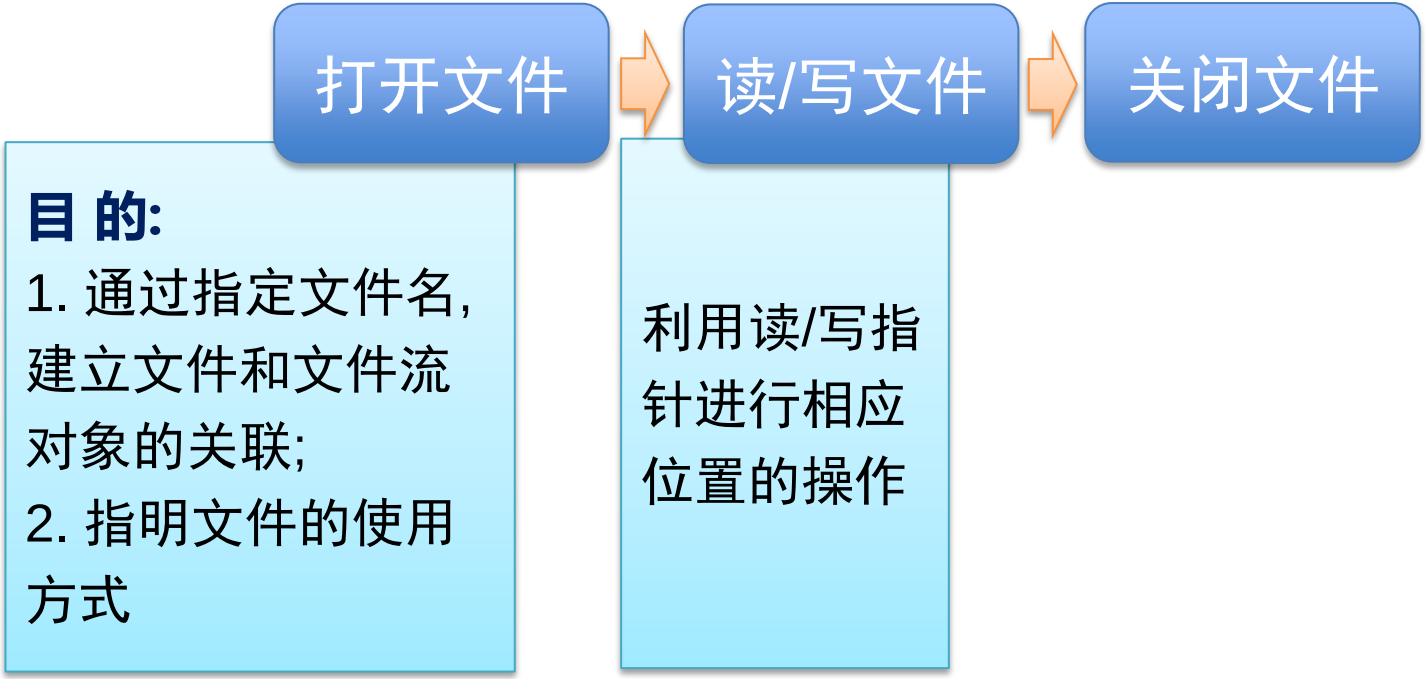
- 顺序文件 — 一个有限字符构成的顺序字符流
- C++标准库中: ifstream, ofstream和fstream共3个类

→用于文件操作 — 统称为文件流类



文件操作

- 使用/创建文件的基本流程



建立顺序文件

fstream中定义的类

将要建立的文件的文件名

```
#include <fstream>  // 包含头文件
ofstream outFile(“clients.dat”, ios::out|ios::binary); //打开文件
```

自定义的ofstream类的对象

打开并建立文件的选项

- ios::out 输出到文件, 删除原有内容
- ios::app 输出到文件, 保留原有内容, 总是在尾部添加
- ios::binary 以二进制文件格式打开文件

建立顺序文件

- 也可以先创建 ofstream对象, 再用 open函数 打开

```
ofstream fout;
fout.open( “test.out”, ios::out|ios::binary );
```

- 判断打开是否成功:

```
if(!fout) {  cerr << “File open error!”<<endl;  }
```

- 文件名可以给出绝对路径, 也可以给相对路径
- 没有交代路径信息, 就是在当前文件夹下找文件

文件的读写指针

- 对于输入文件,有一个读指针
- 对于输出文件, 有一个写指针
- 对于输入输出文件, 有一个读写指针
- 标识文件操作的当前位置,

  该指针在哪里 → 读写操作就在哪里进行

文件的读写指针

```
ofstream  fout(“a1.out”, ios::app);
long location = fout.tellp();       //取得写指针的位置
location = 10L;
fout.seekp(location);               // 将写指针移动到第10个字节处
fout.seekp(location, ios::beg);  //从头数location
fout.seekp(location, ios::cur);   //从当前位置数location
fout.seekp(location, ios::end);  //从尾部数location
```

- location 可以为负值

文件的读写指针

```
ifstream  fin(“a1.in”,ios::in);
long location = fin.tellg();          //取得读指针的位置
location = 10L;
fin.seekg(location);                  //将读指针移动到第10个字节处
fin.seekg(location, ios::beg);    //从头数location
fin.seekg(location, ios::cur);    //从当前位置数location
fin.seekg(location, ios::end);    //从尾部数location
```

- location 可以为负值

二进制文件读写

```
int x=10;
fout.seekp(20, ios::beg);
fout.write( (const char *)(&x), sizeof(int) );

fin.seekg(0, ios::beg);
fin.read( (char *)(&x), sizeof(int) );
```

- 二进制文件读写，直接写二进制数据，记事本看未必正确

二进制文件读写

```
//下面的程序从键盘输入几个学生的姓名的成绩,
//并以二进制, 文件形式存起来
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;
class CStudent {
    public:
        char szName[20];
        int nScore;
};
```

```
int main()
{
    CStudent s;
    ofstream OutFile( “c:\\tmp\\students.dat”, ios::out|ios::binary );
    while( cin >> s.szName >> s.nScore ) {
        if( stricmp(s.szName, “exit” ) == 0)    //名字为exit则结束
            break;
        OutFile.write( (char *) & s, sizeof(s) );
    }
    OutFile.close();
    return 0;
}
```

输入:

Tom 60

Jack 80

Jane 40

exit 0

Note -- 文本文件/二进制文件打开文件的区别:

- 在Unix/Linux下, 二者一致, 没有区别;
- 在Windows下, 文本文件是以 “\r\n”作为换行符

→读出时, 系统会将0x0d0a只读入0x0a

→写入时, 对于0x0a系统会自动写入0x0d

- 则形成的 students.dat 为 72字节
- 用 记事本打开, 呈现:

Tom 烫烫烫烫烫烫烫烫<Jack 烫烫烫烫烫烫烫烫蘼

Jane 烫烫烫烫烫烫烫?

二进制文件读写

```
//下面的程序将 students.dat 文件的内容读出并显示
#include <iostream>
#include <fstream>
using namespace std;
class CStudent
{
    public:
     char szName[20];
     int nScore;
};
```

```
int main(){
    CStudent s;
    ifstream inFile("students.dat", ios::in | ios::binary );
    if(!inFile) {
       cout << "error" <<endl;
       return 0;
    }
    while( inFile.read( (char* ) & s, sizeof(s) ) ) {
        int nReadedBytes = inFile.gcount(); //看刚才读了多少字节
       cout << s.szName << " " << s.score << endl;
    }
    inFile.close();
    return 0;
}
```

输出：
Tom 60
Jack 80
Jane 40

二进制文件读写

```
//下面的程序将 students.dat 文件的Jane的名字改成Mike
#include <iostream>
#include <fstream>
using namespace std;
class CStudent
{
    public:
        char szName[20];
        int nScore;
};
```

```
int main(){
    CStudent s;
    fstream iofile( “c:\\tmp\\students.dat”, ios::in|ios::out|ios::binary);
    if(!iofile) {
        cout << "error" ;
        return 0;
    }
    iofile.seekp( 2 * sizeof(s), ios::beg);    //定位写指针到第三个记录
    iofile.write( “Mike”, strlen("Mike")+1);
    iofile.seekg(0, ios::beg);      //定位读指针到开头
    while( iofile.read( (char* ) & s, sizeof(s)) )
        cout << s.szName << " " << s.nScore << endl;
    iofile.close();
    return 0;
}
```

输出:
Tom 60
Jack 80
Mike 40

显式关闭文件

- ifstream fin(“test.dat”, ios::in);

  fin.close();

- ofstream fout(“test.dat”, ios::out);

  fout.close();

例子: mycopy 程序, 文件拷贝

```
//用法示例：
//mycopy src.dat dest.dat
//即将 src.dat 拷贝到 dest.dat
//如果 dest.dat 原来就有, 则原来的文件会被覆盖
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char * argv[]){
    if(argc != 3) {
        cout << "File name missing!" << endl;
        return 0;
    }
    ifstream inFile(argv[1], ios::binary|ios::in);      //打开文件用于读
```

```
    if(! inFile) {
        cout << “Source file open error.” << endl;
        return 0;
    }
    ofstream outFile(argv[2], ios::binary|ios::out);    //打开文件用于写
    if(!outFile) {
        cout << “New file open error.” << endl;
        inFile.close();    //打开的文件一定要关闭
        return 0;
    }
    char c;
    while(inFile.get(c))    //每次读取一个字符
        outFile.put(c);     //每次写入一个字符
    outFile.close();
    inFile.close();
    return 0;
}
```

函数模板

郭 炜　刘家瑛

北京大学

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

- 为了交换两个int变量的值, 需要编写如下Swap函数:

```
void Swap(int & x, int & y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

函数模板

- 为了交换两个double型变量的值, 还需要编写如下Swap函数:

```
void Swap(double & x, double & y)
{
    double tmp = x;
    x = y;
    y = tmp;
}
```

能否只写一个Swap, 就能交换各种类型的变量?

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
}
```

函数模板

- 交换两个变量值的函数模板

```
template <class T>
void Swap(T & x, T & y)
{
    T tmp = x;
    x = y;
    y = tmp;
}
```

函数模板

```
int main(){
    int n = 1, m = 2;
    Swap(n, m);      //编译器自动生成 void Swap(int &, int &)函数
    double f = 1.2, g = 2.3;
    Swap(f, g);      //编译器自动生成 void Swap(double &, double &)函数
    return 0;
}
```

```
void Swap(int & x, int & y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

```
void Swap(double & x, double & y)
{
    double tmp = x;
    x = y;
    y = tmp;
}
```

函数模板

- 函数模板中可以有不止一个类型参数

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

函数模板

- 求数组最大元素的MaxElement函数模板

```
template <class T>
T MaxElement(T a[], int size) //size是数组元素个数
{
    T tmpMax = a[0];
    for( int i = 1; i < size; ++i )
        if( tmpMax < a[i] )
    tmpMax = a[i];
    return tmpMax;
}
```

函数模板

- 函数模板可以重载, 只要它们的形参表不同即可
- 例, 下面两个模板可以同时存在:

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

函数模板

- C++编译器遵循以下优先顺序:

  Step 1: 先找参数完全匹配的普通函数(非由模板实例化而得的函数)

  Step 2: 再找参数完全匹配的模板函数

  Step 3: 再找实参经过自动类型转换后能够匹配的普通函数

  Step 4: 上面的都找不到, 则报错

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
```

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
```

类模板

郭 炜 刘家瑛

北京大学

问题的提出

定义一批相似的类 → 定义类模板 → 生成不同的类

- 数组
  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
  - ……

问题的提出

- 对于这些数组类
  - 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

- 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，如在类模板外面定义时，

```
template <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)
{
        ……
}
```

类模板的定义

- 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

- 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

- Pair类模板的使用:

```
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

使用类模板声明对象

- 编译器由类模板生成类的过程叫类模板的实例化
  - 编译器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
- 由类模板实例化得到的类叫模板类
  - 为类型参数指定的数据类型不同，得到的模板类不同

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
```

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;
}
```

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’ ”

程序输出:
K

类模板与非类型参数

- 类模板的参数声明中可以包括非类型参数

  template <class T, int elementsNumber>

  - 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, 成员操作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:

```
template <class T, int size>
class CArray{
        T  array[size];
  public:
        void Print( )
        {
            for(int i = 0; i < size; ++i)
                cout << array[i] << endl;
        }
};
```

类模板与非类型参数

CArray<double, 40> a2;

CArray<int, 50> a3;

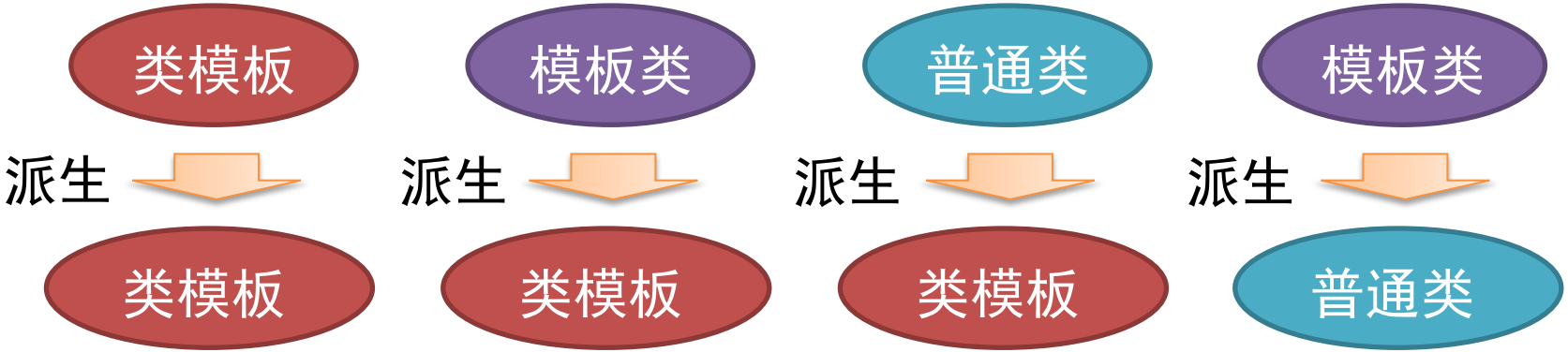
- 注意:

  CArray<int,40>和CArray<int,50>完全是两个类

  这两个类的对象之间不能互相赋值

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;
}
```

string类

郭 炜 刘家瑛

北京大学

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 <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

- 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
  - string s;

    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;
}
```

程序输出:

```
Hello
xxxxxxxx
March
n
```

string类

- 构造的string太长而无法表达 → 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, 而下标运算符不做范围检查

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;
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;
```

string的特性

31 // capacity
4294967293 // maximum_size
11 // length
11 // size
0 // empty
hello worldaaa // string itself and “aaa”
31
4294967293
21
21
hello worldaaa
1
不同编译器上可能会不一样

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 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("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 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("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 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("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 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:
hello55

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
//且该字符串以 '\0' 结尾
```

输出:
hello world

转换成C语言式char *字符串

- 成员函数data()

```
string s1("hello world");
const char * p1=s1.data();
for(int i=0; i<s1.length(); i++)
        printf("%c",*(p1+i));
//s1.data() 返回一个char * 类型的字符串
//对s1 的修改可能会使p1出错。
```

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
```

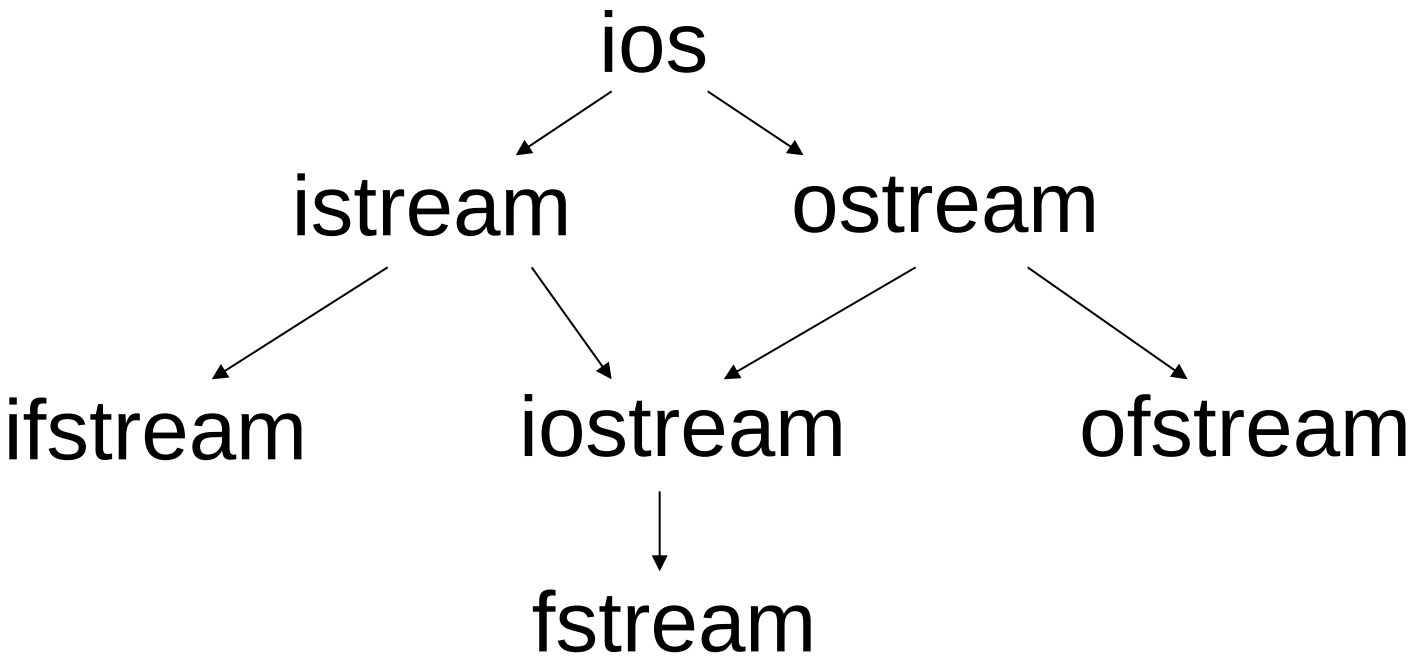
输出:
hello

程序设计实习

郭炜　微博 http://weibo.com/guoweiofpku
http://blog.sina.com.cn/u/3266490431
刘家瑛 微博 http://weibo.com/pkuliujiaying

输入和输出

与输入输出流操作相关的类



与输入输出流操作相关的类

istream是用于输入的流类，cin就是该类的对象。

ostream是用于输出的流类，cout就是该类的对象。

ifstream是用于从文件读取数据的类。

ofstream是用于向文件写入数据的类。

iostream是既能用于输入，又能用于输出的类。

fstream 是既能从文件读取数据，又能向文件写入数据的类。

标准流对象

- 输入流对象：cin　　与标准输入设备相连
- 输出流对象：cout　　与标准输出设备相连
  cerr　　与标准错误输出设备相连
  clog　　与标准错误输出设备相连

缺省情况下

```
cerr << "Hello,world" << endl;
clog << "Hello,world" << endl;
```

和

```
cout << “Hello,world” << endl;
```

一样

标准流对象

- cin对应于标准输入流，用于从键盘读取数据，也可以被重定向为从文件中读取数据。
- cout对应于标准输出流，用于向屏幕输出数据，也可以被重定向为向文件写入数据。
- cerr对应于标准错误输出流，用于向屏幕输出出错信息，
- clog对应于标准错误输出流，用于向屏幕输出出错信息，
- cerr和clog的区别在于cerr不使用缓冲区，直接向显示器输出信息；而输出到clog中的信息先会被存放在缓冲区，缓冲区满或者刷新时才输出到屏幕。

输出重定向

```
#include <iostream>
using namespace std;
int main()   {
    int x,y;
    cin >> x >> y;
    freopen("test.txt","w",stdout);  //将标准输出重定向到 test.txt文件
    if( y == 0 )  //除数为0则在屏幕上输出错误信息
            cerr << "error." << endl;
    else
             cout << x /y ;  //输出结果到test.txt
    return 0;
}
```

输入重定向

```
#include <iostream >
using namespace std;
int main()  {
        double f;          int n;
        freopen(“t.txt”,“r”,stdin);  //cin被改为从 t.txt中读取数据
        cin >> f >> n;
        cout << f << "," <<n << endl;
        return 0;
}
```

t.txt:
3.14 123

输出:
3.14,123

判断输入流结束

可以用如下方法判输入流结束：

```
int x;
while(cin>>x){
    ……
}
return 0;
```

```
istream &operator >>(int a)
    {
        .......
        return *this ;
    }
```

- 如果是从文件输入，比如前面有

  freopen(“some.txt”,”r”,stdin);

  那么，读到文件尾部，输入流就算结束

- 如果从键盘输入，则在单独一行输入Ctrl+Z代表输入流结束

istream类的成员函数

istream & getline(char * buf, int bufSize);
从输入流中读取bufSize-1个字符到缓冲区buf，或读到碰到‘\n’为止（哪个先到算哪个）。
istream & getline(char * buf, int bufSize,char delim);
从输入流中读取bufSize-1个字符到缓冲区buf，或读到碰到delim字符为止（哪个先到算哪个）。

两个函数都会自动在buf中读入数据的结尾添加\0’。，‘\n’或delim都不会被读入buf，但会被从输入流中取走。如果输入流中‘\n’或delim之前的字符个数达到或超过了bufSize个，就导致读入出错，其结果就是：虽然本次读入已经完成，但是之后的读入就都会失败了。

可以用 if(!cin.getline(…)) 判断输入是否结束

istream类的成员函数

bool eof(); 判断输入流是否结束

int peek(); 返回下一个字符,但不从流中去掉.

istream & putback(char c); 将字符ch放回输入流

istream & ignore(int nCount = 1, int delim = EOF);

从流中删掉最多nCount个字符，遇到EOF时结束。

istream类的成员函数

```
#include <iostream >
using namespace std;
int main()  {
        int x;
        char buf[100];
        cin >> x;
        cin.getline(buf,90);
        cout << buf << endl;
        return 0;
}
```

输入：
12 abcd↙
输出：
 abcd   (空格+abcd)

输入
12↙
程序立即结束，输出：
12

因为getline读到留在流中的'\n'就会返回