STL 和字符串处理

（OOP）

黄民烈

aihuang@tsinghua.edu.cn

http://coai.cs.tsinghua.edu.cn/

课程团队：刘知远 姚海龙 黄民烈

上期要点回顾

- 类模板与函数模板特化
- 命名空间
- STL初步——容器与迭代器

本讲内容提要

- string字符串类
- iostream输入输出流
- 字符串处理与正则表达式

string
字符串类

变长字符串

- 字符串是char的数组
- 但如果我们无法提前确认字符串长度?
  - vector<char>?

- STL为我们提供了更方便的string类型
  - 允许简洁的拼接操作

```
string fullname = firstname + " " + lastname;
```

  - 也能够使用惯用的输入输出方法

```
cout << fullname << endl;
```

string类常用函数

构造方式

- `string s0("Initial string"); //从c风格字符串构造`
- `string s1; //默认空字符串`
- `string s2(s0, 8, 3); //截取:“str”, index从8开始, 长度为3`
- `string s3(“Another character sequence”, 12); //截取:“Another char”`
- `string s4(10, 'x'); //复制字符: xxxxxxxxxx`
- `string s5(s0.begin(), s0.begin()+7); //复制截取: Initial`

转换为c风格字符串

- `str.c_str() //注意返回值为常量字符指针(const char*), 不能修改`

string类常用函数

- 和vector类似
  - 访问/修改元素：cout << str[1]; str[1]='a';
  - 查询长度：str.size()
  - 清空：　　str.clear()
  - 查询是否为空：str.empty()
  - 迭代访问：for(char c : str)
  - 向尾部增加：

```
str.push_back('a');
str.append(s2);
```

- 不同之处
  - 查询长度也可以使用str.length()，与size()返回值相同
  - 向尾部增加也可以使用 str += 'a' 或者 str += s2

string类常用函数

- 三种输入方式
  - 读取可见字符直到遇到空格

```
cin >> firstname;
        //Mike
```

输入文本

Mike William
Andy William
#

  - 读一行

```
getline(cin, fullname); //Mike William
```

  - 读到指定分隔符为止（可以读入换行符）

```
getline(cin, fullnames, '#');
        //"Mike William\nAndy William\n"
```

string类常用函数

拼接

- `string fullname = firstname + " " + lastname;`
- 注意：拼接的时间复杂度为生成的字符串长度
- 例如：

```
for(int i = 0; i < n; i++)
    allname = allname + name[i] + "\n"
//时间复杂度O(n^2*L)的时间，L表示每个子串的平均长度
```

- 拼接多个字符串最好使用 `operator+=` 或者 `stringstream`

比较

- 我们可以直接使用运算符比较字符串字典序
- `string a = "alice", b = "bob";`
- `a == b //False`
  `a < b  //True`

string类常用函数

数值类型字符串化

```
to_string(1)                    //"1"
to_string(3.14)                 //"3.14"
to_string(3.1415926)            //"3.141593" 注意精度损失
to_string(1+2+3)                //"6"
```

字符串转数值类型

```
int a = stoi("2001")              //a=2001
std::string::size_type sz;    // size_t alias
int b = stoi("50 cats", &sz)  //b=50 sz=2 读入长度
int c = stoi("40c3", nullptr, 16) //c=0x40c3 十六进制
int d = stoi("0x7f", nullptr, 0)  //d=0x7f 自动检查进制

double e = stod("34.5")         //e=34.5
```

iostream
输入输出流

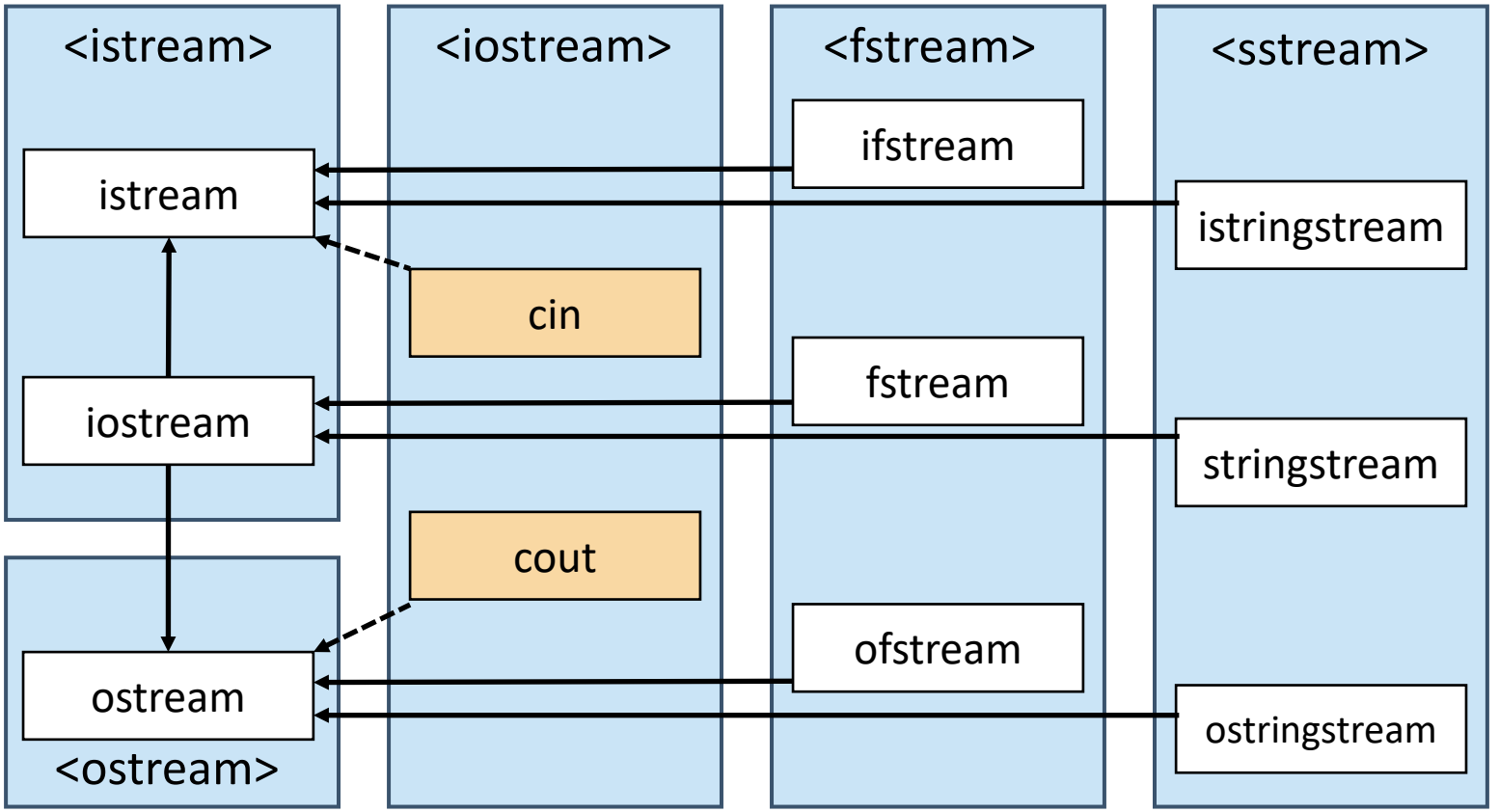
回忆：重载输出流运算符

```
cout << str << endl;
cin >> str;

ostream& operator<<(ostream& out, const Test& src)
{
   out << src.id << endl;
   return out;
}
```

- ostream到底是什么？

STL输入输出流


<istream>
<iostream>
<fstream>
<sstream>
ifstream
istream
istringstream
cin
fstream
iostream
stringstream
cout
ofstream
ostream
ostringstream
<ostream>

从ostream和cout开始

- ostream即output stream

  是STL库中所有输出流的基类

- 它重载了针对基础类型的输出流运算符（<<）

  接受不同类型的数据，再调用系统函数进行输出

- 统一了输出接口，改善了C中输出方式混乱的状况
  - printf("%d %f %s", 1, 2.3, "hello");

http://www.cplusplus.com/reference/ostream/ostream/operator%3C%3C/

- cout是STL中内建的一个ostream对象
- 它会将数据送到标准输出流（一般是屏幕）

实现自己的ostream

```
class ostream
{
public:
    ostream& operator<<(char c)
    {
        printf("%c", c);
        return *this;
    }
    ostream& operator<<(const char* str)
    {
        printf("%s", str);
        return *this;
    }
}cout;
```

```
int main()
{
    cout << "hello" << ' '
         << "world";
    return 0;
}
```

实现原理：

<<运算符为左结合

先执行cout << "hello" 调用第二个函数 返回c1（cout的引用）
再执行c1 << ' ' 调用第一个函数 返回c2 (cout的引用)
最后执行c2 << "world" 调用第二个函数

格式化输出

- 如何格式化输出 – #include <iomanip>

```
cout << fixed << 2018.0 << " " << 0.0001 << endl;
                //浮点数 -> 2018.000000 0.000100
cout << scientific << 2018.0 << " " << 0.0001 << endl;
                //科学计数法 -> 2.018000e+03 1.000000e-04
cout << defaultfloat;  //还原默认输出格式
cout << setprecision(2) << 3.1415926 << endl;
                //输出精度设置为2 -> 3.2
cout << oct << 12 << " " << hex << 12 << endl;
                //八进制输出 -> 14  十六进制输出 -> c
cout << dec; //还原十进制
cout << setw(3) << setfill('*') << 5 << endl;
                //设置对齐长度为3，对齐字符为* -> **5
```

格式化输出

- 以setprecision为例
  - cout << setprecision(2) << 1.05 << endl;
  - 保留2位精度，输出1.1
- 如何实现?
  - C++标准中未定义，不同编译器有自己的实现方式
  - 一种实现方式的示例

```
class setprecision
{
private:
    int precision;
public:
    setprecision(int p) : precision(p) {}
    friend class ostream;
};
// setprecision(2) 是一个类的对象
```

流操纵算子(stream manipulator)

```
class ostream
{
private:
    int precision; //记录流的状态
public:
    ostream& operator<<
            (const setprecision &m) {
        precision = m.precision;
        return *this;
    }
} cout;
```

- 借助辅助类，设置成员变量
- 这种类叫流操纵算子

```
cout << setprecision(2);
// setprecision(2) 是一个类的对象
```

流操纵算子：endl

- C++标准中endl的声明
  - ostream& endl(ostream& os);
- endl是一个函数
  - 等同于输出'\n'，再清空缓冲区 os.flush()

```
ostream& endl(ostream& os) {
    os.put('\n');
    os.flush();
    return os;
}
```

  - 可以调用 endl(cout);
- 缓冲区
  - 目的是减少外部读写次数
  - 写文件时，只有清空缓冲区或关闭文件才能保证内容正确写入

流操纵算子：endl

- endl同时也是流操纵算子，如何实现?
  - cout << endl;

- 一种实现方式的示例

```
ostream& operator<<
        (ostream& (*fn)(ostream&)) {
        //流运算符重载，函数指针作为参数
    return (*fn)(*this);
}
```

不能复制的cout

- 注意重载流运算符的方式

```
ostream& operator<<(const char &c)
friend ostream& operator<<(ostream& os, MyClass obj)
```

- 为什么重载流运算符要返回引用？
  - 避免复制

- 观察ostream的复制构造函数
  - ostream(const ostream&) = delete;
  - ostream(ostream&& x);
  - 禁止复制、只允许移动
  - 仅使用cout一个全局对象

不能复制的cout

- 为什么只能使用一个对象?
  - 减少复制开销
  - 一个对象对应一个标准输出，符合OOP思想
  - 多个对象之间无法同步输出状态

- 是否能做得更好?
  - 全局对象往往引入初始化顺序问题
  - 单件模式（Singleton Pattern）
  - 在之后的设计模式中会介绍

文件输入输出流

- 以文件输入流作为例子
- ifstream是istream的子类
- 功能是从文件中读入数据

- 打开文件
  - ifstream ifs("input.txt");
  - ifstream ifs("binary.bin", ifstream::binary);
    //以二进制形式打开文件
  - ifstream ifs;
    ifs.open("file")
    //do something
    ifs.close()

读入示例

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

int main() {
    ifstream ifs("input.txt");
    while(ifs) {          //判断文件是否到末尾 利用了重载的bool运算符
        ifs >> ws;        //除去前导空格 ws也是流操纵算子
        int c = ifs.peek();       //检查下一个字符，但不读取
        if (c == EOF) break;
        if (isdigit(c))           //<cctype>库函数
        {
            int n;
            ifs >> n;
            cout << "Read a number: " << n << endl;
        } else {
            string str;
            ifs >> str;
            cout << "Read a word: " << str << endl;
        }
    }
}
```

其他操作

- **getline(cin, str)**
  - ifstream是istream的子类
  - 故getline(ifs, str)仍然有效

- **其他操作**
  - get()　　读取一个字符
  - ignore(int n=1, int delim=EOF)
    丢弃n个字符，或者直至遇到delim分隔符
  - peek()　　查看下一个字符
  - putback(char c) 返还一个字符
  - unget()　　返还一个字符
  - ……

istream与scanf

- 为什么C++使用流输入取代了scanf
  - scanf不友好，不同类型要使用不同的标识符

```
scanf("%d %hd %f %lf %s", &i, &s, &f, &d, name);
cin >> i >> s >> f >> d >> name;
```

  - 安全性

```
scanf("%d %d", &a);  //可能写入非法内存
```

  - 可拓展性

```
MyClass obj;
cin >> obj;
```

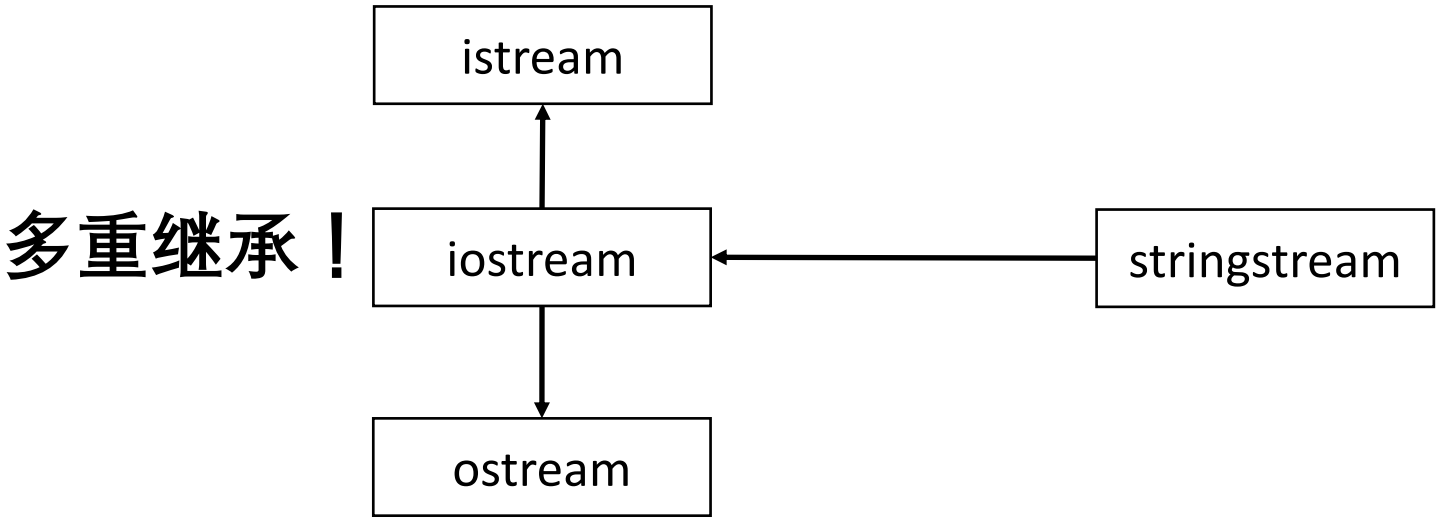
  - 性能

scanf在运行期间需要对格式字符串进行解析

istream在编译期间已经解析完毕

字符串输入输出流

- 以输入输出流作为例子
- stringstream是iostream的子类
- iostream继承于istream和ostream
- stringstream实现了输入输出流双方的接口



stringstream

- **stringstream**
  - 它在对象内部维护了一个buffer
  - 使用流输出函数可以将数据写入buffer
  - 使用流输入函数可以从buffer中读出数据
- **一般用于程序内部的字符串操作**

- **构造方式**
  - `stringstream ss; //空字符串流`
  - `stringstream ss(str); //以字符串初始化流`

使用示例

```
#include <sstream>
using namespace std;

int main() {
    stringstream ss;
    ss << "10";
    ss << "0 200";

    int a, b;
    ss >> a >> b;    //a=100 b=200
}
```

- 可以连接字符串
- 可以将字符串转换为其他类型的数据
- 配合流操作算子，可以达到格式化输出效果

获取stringstream的buffer

- ss.str()
  - 返回一个string对象
  - 内容为stringstream的buffer
- 注意buffer内容并不是未读取的内容

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

int main() {
    stringstream ss;
    ss << "100 200";
    cout << ss.str() << endl;  //输出"100 200"
    int a;
    ss >> a;                   // a = 100
    cout << ss.str() << endl;  //输出"100 200"
    return 0;
}
```

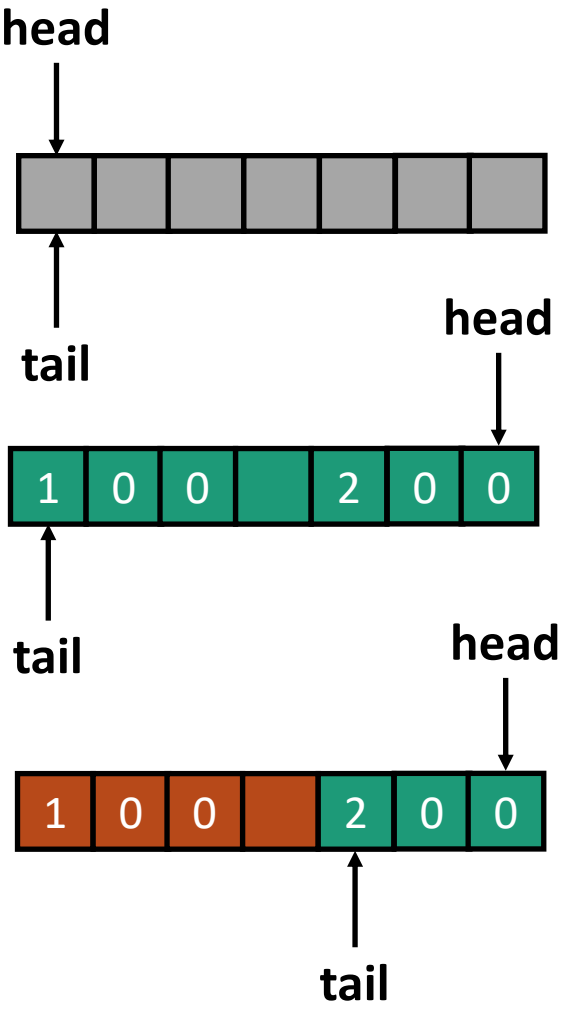
获取stringstream的buffer

```
int main()
{
    stringstream ss;

    ss << "100 200";
    cout << ss.str() << endl;
     //"100 200"

    int a, b;
    ss >> a;  // a = 100
    cout << ss.str() << endl;
    //"100 200"

    ss >> b; // b = 200
    return 0;
}
```



实现一个类型转换函数

- 如何实现字符串与整数的互相转换?
  - to_string 转换为字符串
  - stoi 转换为整数
- 其他类型呢? 可以使用一个函数实现吗?

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

int main() {
    string x = convert<string>(123);
    int y = convert<int>("456");
    cout << x << endl;
    cout << y << endl;
    return 0;
}
```

实现一个类型转换函数

```
template<class outtype, class intype>
outtype convert(intype val)
{
    static stringstream ss;
                        //使用静态变量避免重复初始化
    ss.str("");         //清空缓冲区
    ss.clear();         //清空状态位（不是清空内容）
    ss << val;
    outtype res;
    ss >> res;
    return res;
}
```

关于状态位：
http://www.cplusplus.com/reference/ios/ios/setstate/

字符串处理与正则表达式

用户名注册

- 场景：用户名注册
  - 只能包含小写字母、数字、下划线，并且限制用户名长度在3~15个字符之间

    合法例子： john_123

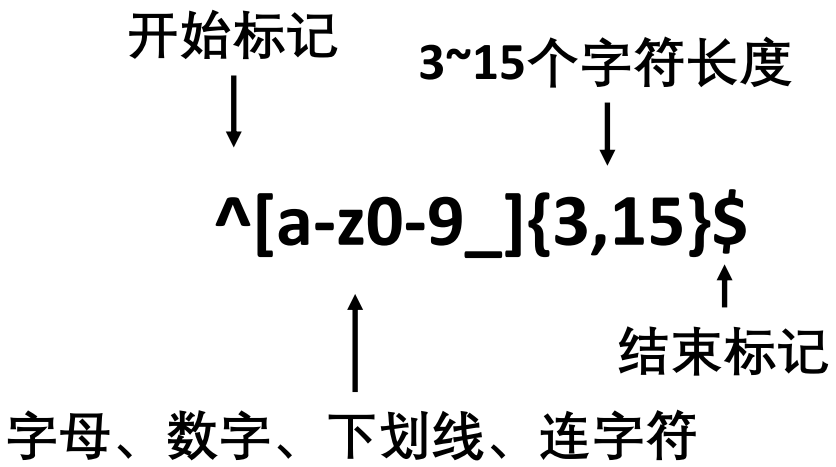
    非法例子： John_123 / jo / @john
  - 如何处理？

```
bool check(string name){
    if (name.length() < 3 || name.length() > 15) return false;
    for(char c: name){
        if(!((c >= 'a' && c <= 'z') || //小写字母
            (c >= '0' && c <= '9') ||  //数字
            c == '_'))  return false;
    }
}
```

太过复杂，不易修改

正则表达式

- 正则表达式：由字母和符号组成的特殊文本，搜索文本时定义的一种规则
- 场景：用户名注册
  - 只能包含小写字母、数字、下划线和连字符，并且限制用户名长度在3~15个字符之间
  - 使用正则表达式表示规则



正则表达式

正则表达式的三种模式

- 匹配：判断整个字符串是否满足条件

  ^[a-z0-9_]{3,15}$

  能与john_123匹配，不能与Jo匹配

- 搜索：符合正则表达式的子串

  在"q123e456w"找出所有数字串 **[0-9]+**

  搜索结果 123，456

- 替换：按规则替换字符串的子串

  给定"q123e456w"将所有数字串替换为(number)

  替换结果 q(123)e(456)w

如何编写正则表达式？

基本模式匹配

- 字符代表其本身
  - 如：使用the进行搜索，可以找到句中所有的"the"

    The car parked in the garage.

- 转义字符
  - \n表示换行、\t表示制表符

- 特殊匹配字符
  - ^代表字符串开头，$代表字符串结尾
  - 如：^\t只能匹配到以制表符开头的内容
  - 如：^bucket$只能匹配到只含bucket的内容

字符簇

- 匹配的单个字符在某个范围中
  - [aeiou] 匹配任意一个元音字符
  - [a-z] 匹配所有单个小写字母
  - [0-9] 匹配所有单个数字
- 范围取反
  - [^a-z] 匹配所有非小写字母的单个字符
  - [^c]ar: The car parked in the garage.
- 连用
  - [a-z][0-9] 匹配所有字母+数字的组合，比如a1、b9
  - ^[^0-9][0-9]$ 匹配长度为2的内容，且第一个不为数字，第二个为数字
  - [Tt]he: The car parked in the garage.

字符簇

特殊字符

- .匹配除换行以外任意字符，[.]或\.可表示匹配句号
  - .ar: The car parked in the garage.
  - ge[.]: The car parked in the garage.

- \d 等价[0-9]，匹配所有单个数字
- \D 等价[^0-9]，匹配所有单个非数字
- \s 匹配所有空白字符，如\t,\n
- \S 匹配所有非空白字符
- \w 匹配字母、数字、下划线，等价[a-zA-Z0-9_]
- \W 匹配非字母、数字、下划线，等价[^a-zA-Z0-9_]

重复模式

- x{n,m}代表前面内容出现次数重复n~m次
  - a{4} 匹配aaaa
  - a{2,4} 匹配aa、aaa、aaaa
  - a{2,} 匹配长度大于等于2的a
- 扩展到字符簇
  - [a-z]{5,12} 代表为长度为5~12的英文字母组合
  - .{5} 所有长度为5的字符
- 特殊字符
  - ? 等价{0,1}
    - [T]?he: **The** car parked in **the** garage.
  - + 等价{1,}
    - c.+e: The **car parked in the garage**.
  - * 等价{0,}
    - [a-z]*: **The car parked in the garage**.

或连接符

- 匹配模式可以使用'|'进行连接
  - (Chapter|Section) [1-9][0-9]?
    可以匹配Chapter 1、Section 10等
  - 0\d{2}-\d{8}|0\d{3}-\d{7}
    可以匹配010-12345678、0376-2233445
  - (c|g|p)ar: The car parked in the garage.

- 使用()改变优先级
  - m|food 可以匹配 m 或者 food
  - (m|f)ood 可以匹配 mood 或者 food
  - (T|t)he|car: The car parked in the garage.

正则表达式辅助工具

- 动态匹配
  - 可以反复测试
  - 以染色区分匹配部分

http://tool.chinaz.com/regex/

正则表达式库 <regex>

- 如何在C++中使用正则表达式
  - <regex>库

- 创建一个正则表达式对象
  - regex re("^[1-9][0-9]{10}$") 11位数

  - 注意：C++的字符串中"\"也是转义字符
    - 如果需要创建正则表达式"\d+"，应该写成
    - regex re("\\d+")

原生字符串

- 原生字符串
  - 原生字符串可以取消转义，保留字面值
  - 语法：R"(str)" 表示str的字面值
  - "\\d+" = R"(\d+)" = \d+
  - 原生字符串能换行，比如

```
string str = R"(Hello
World)";
```

  - 结果str = "hello\nWorld"

正则表达式库 <regex>

- 匹配
  - regex_match(s, re)：询问字符串s是否能完全匹配正则表达式re

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

int main() {
    string s("subject");
    regex e("sub.*");
    smatch sm;
    if(regex_match(s,e))
        cout << "matched" << endl;
}
```

输出：matched

捕获和分组

- 有时我们想要获取匹配每一个部分的细节
  - 例如：在 **\w*\d*** 中，我们想知道 \w*和\d*分别匹配了什么

- 使用()进行标识，每个标识的内容被称作分组
  - 正则表达式匹配后，每个分组的内容将被捕获
  - 用于提取关键信息，例如version(\d+)即可捕获版本号

正则表达式库 <regex>

- 匹配和捕获
  - regex_match(s, result, re)：询问字符串s是否能完全匹配正则表达式re，并将捕获结果储存到result中
  - result需要是smatch类型的对象

```
#include <iostream>
#include <string>
#include <regex>
using namespace std;
int main () {
    string s("version10");
    regex e(R"(version(\d+))"); smatch sm;
    if(regex_match(s,sm,e)) {
        cout << sm.size() << " matches\n";
        cout << "the matches were:" << endl;
        for (unsigned i=0; i<sm.size(); ++i) {
            cout << sm[i] << endl;
        }
    }
    return 0;
}
```

输出：

```
2 matches
the matches were:
version10
10
```

捕获和分组

- 分组会按顺序标号
  - 0号永远是匹配的字符串本身
  - (a)(pple)： 0号为apple，1号为a，2号为pple
  - 用(sub)(.*)匹配subject：0号为subject，1号为sub，2号为ject
- 如果需要括号，又不想捕获该分组，可以使用(?:pattern)
  - 用(?:sub)(.*)匹配subject：0号为subject，1号为ject

正则表达式库 <regex>

- 搜索
  - regex_search(s, result, re)：搜索字符串s中能够匹配正则表达式re的第一个子串，并将结果存储在result中
  - result是一个smatch对象
  - 对于该子串，分组同样会被捕获

搜索的例子

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

int main() {
    string s("this subject has a submarine");
    regex e(R"((sub)([\S]*))");
    smatch sm;
    //每次搜索时当仅保存第一个匹配到的子串
    while(regex_search(s,sm,e)){
        for (unsigned i=0; i<sm.size(); ++i)
            cout << "[" << sm[i] << "] ";
        cout << endl;
        s = sm.suffix().str();
    }
}
```

输出：

```
[subject] [sub] [ject]
[submarine] [sub] [marine]
```

正则表达式库 <regex>

- 替换
  - regex_replace(s, re, s1)：替换字符串s中所有匹配正则表达式re的子串，并替换成s1
  - s1可以是一个普通文本

替换的例子

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

int main() {
    string s("this subject has a submarine");
    regex e(R"(sub[\S]*)");
    //regex_replace返回值即为替换后的字符串
    cout << regex_replace(s,e,"SUB") << "\n";
}
```

输出：
this SUB has a SUB

正则表达式库 <regex>

- 替换
  - regex_replace(s, re, s1)：替换字符串s中所有匹配正则表达式re的子串，并替换成s1
  - s1可以是一个普通文本
  - s1也可以使用一些特殊符号，代表捕获的分组
    - $& 代表re匹配的子串
    - $1，$2 代表re匹配的第1/2个分组

替换的例子

```
输出：
this SUBJECT has a SUBJECT
this [subject] has a [submarine]
this sub has a sub
this ject has a marine
this sub and [ject] has a sub and [marine]
```

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

int main() {
    string s("this subject has a submarine");
    regex e(R"((sub)([\S]*))");
    //regex_replace返回值即为替换后的字符串
    cout << regex_replace(s,e,"SUBJECT") << endl;
    //$&表示所有匹配成功的部分，[$&]表示将其用[]括起来
    cout << regex_replace(s,e,"[$&]") << endl;
    //$i输出e中第i个括号匹配到的值
    cout << regex_replace(s,e,"$1") << endl;
    cout << regex_replace(s,e,"$2") << endl;
    cout << regex_replace(s,e,"$1 and [$2]") << endl;
}
```

更多内容（自学）

- 预查
  - 正向预查(?=pattern) (?!pattern)
  - 反向预查(?<=pattern) (?<!pattern)

- 后向引用
  - \b(\w+)\b\s+\1\b 匹配重复两遍的单词
  - 比如go go 或 kitty kitty

- 贪婪与懒惰
  - 默认多次重复为贪婪匹配，即匹配次数最多
  - 在重复模式后加? 可以变为懒惰匹配，即匹配次数最少

结束