STL 和 字符串处理

directory

string字符串类

变长字符串

- 字符串是char的数组
- 适用于无法提前确认字符串长度的情况

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() //返回值为常量字符指针(const char*)，不能修改
```

函数

```
访问/修改元素：cout << str[1]; str[1]='a';
查询长度：str.size(); 或 str.length();
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'); 或 str.append(s2); 或 str += 'a'
```

输入

- 读取可见字符直到遇到空格: `cin >> firstname;`

    - //Mike

- 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" 拼接
- 直接使用加号 比较
- 直接使用运算符按字典序比较字符串大小 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;  //代表长度的类型 无符号整数
int b = stoi("50 cats", &sz)  //b=50 sz=2 代表读入长度
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
```

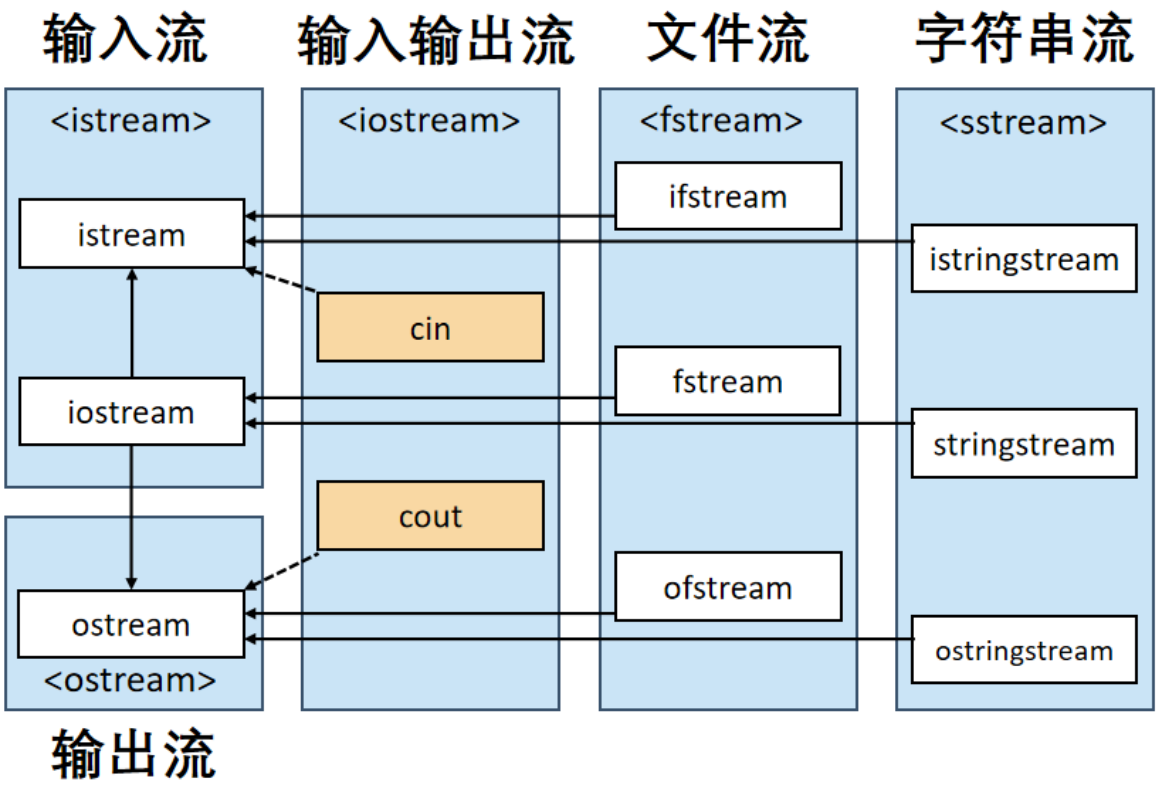
iostream 输入输出流

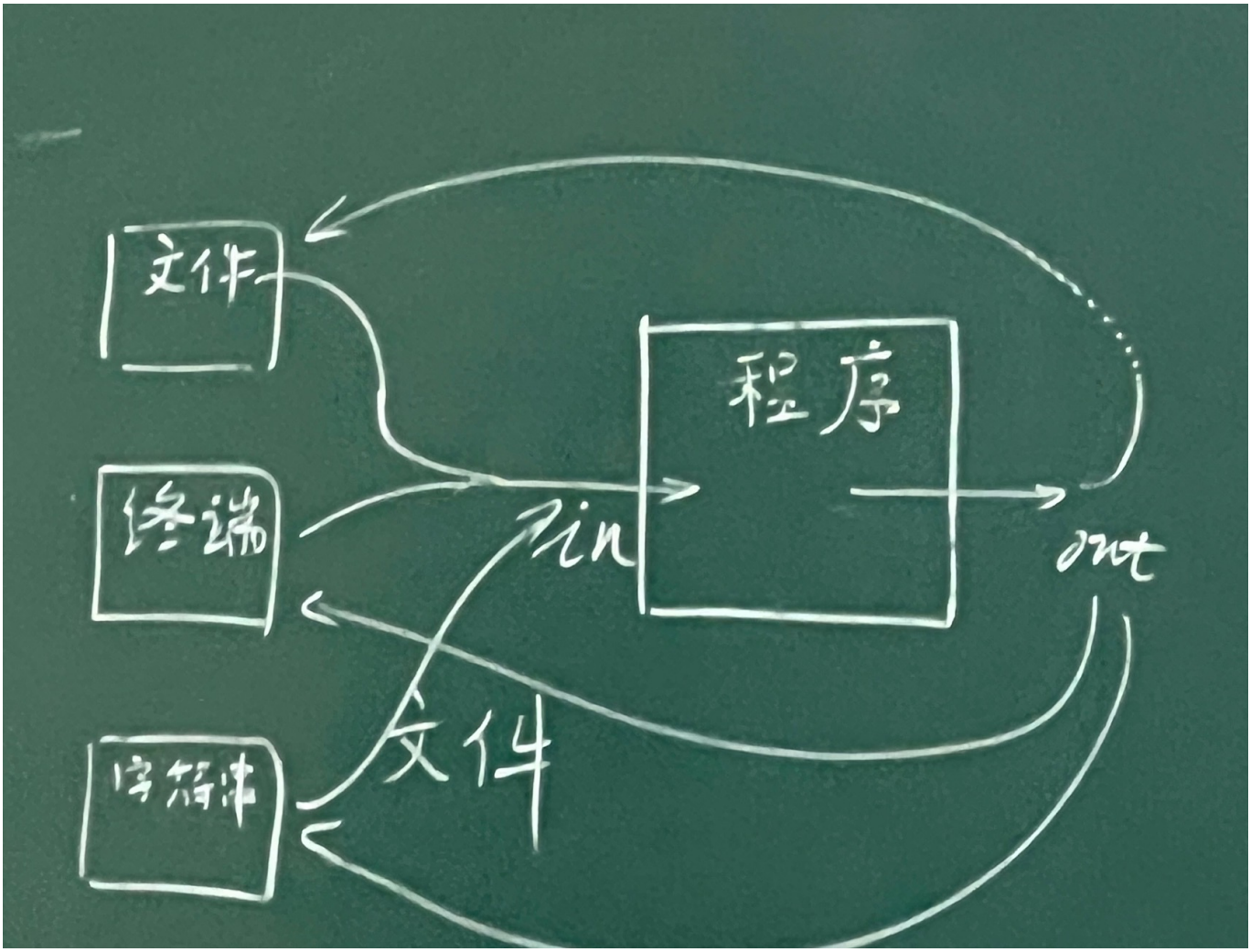
流运算符重载

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

输入输出流

STL输入输出流


输入流
输入输出流
文件流
字符串流
<istream>
<iostream>
<fstream>
<sstream>
istream
iostream
ostream
<ostream>
cin
cout
ifstream
fstream
ofstream
istringstream
stringstream
ostringstream
输出流


文件
终端
字符串
程序
in
out
文件

关系

- ifstream是stream的子类
- ifstream是istream的子类
- 功能是从文件中读入数据
- 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

输入输出流

- 输入流：istream
  - cin
- 输出流：ostream
  - 是STL中所有输出流的基类
  - 流输出运算符<<，接受不同类型的数据，再调用系统函数进行输出
  - cout是STL中内建的一个ostream对象
  - 将数据送到标准输出流（屏幕）

文件流fstream

- 读入文件：ifstream
- 读出文件：ofstream

字符串流 stringstream

- istringstream
- ostringstream

格式化输出

头文件#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 -> 3.1，四舍五入
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
```

流操纵算子(stream manipulator)

- 借助辅助类，设置成员变量
- 这种类叫流操纵算子

```
class ostream {
private:
    int precision; //记录流的状态
public:
    ostream& operator<<(const setprecision &m) {
        precision = m.precision;
        return *this;
    }
} cout;
    cout << setprecision(2);  // setprecision(2) 是一个类的对象
```

流操纵算子：endl

- 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& end(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

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

函数指针

```
#include<iostream>
#include<cstdio>
int max(int x, int y) {return x > y ? x : y; }

int main(void) {    /* p 是函数指针 */
    int (* p)(int, int) = &max; // &可以省略
    int a, b, c, d;
    printf("please input 3 numbers:");
```
```

```
    scanf("%d %d %d", & a, & b, & c); /* 与直接调用函数等价，d = max(max(a, b), c)
*/
    d = p(p(a, b), c);
    printf("the maxumum number is: %d\n", d);
    return 0;
}
```

注意：

- 重载流运算符要返回引用以避免复制
- 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 其他函数

```
getline(cin, str);
getline(ifs,  str);
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

scanf

- 在运行期间需要对格式字符串进行解析
- scanf可能写入非法内存

istream

- istream在编译期间已经解析完毕

stringstream

- 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

构造方式

- stringstream ss; //空字符串流
- stringstream ss(str); //以字符串初始化流

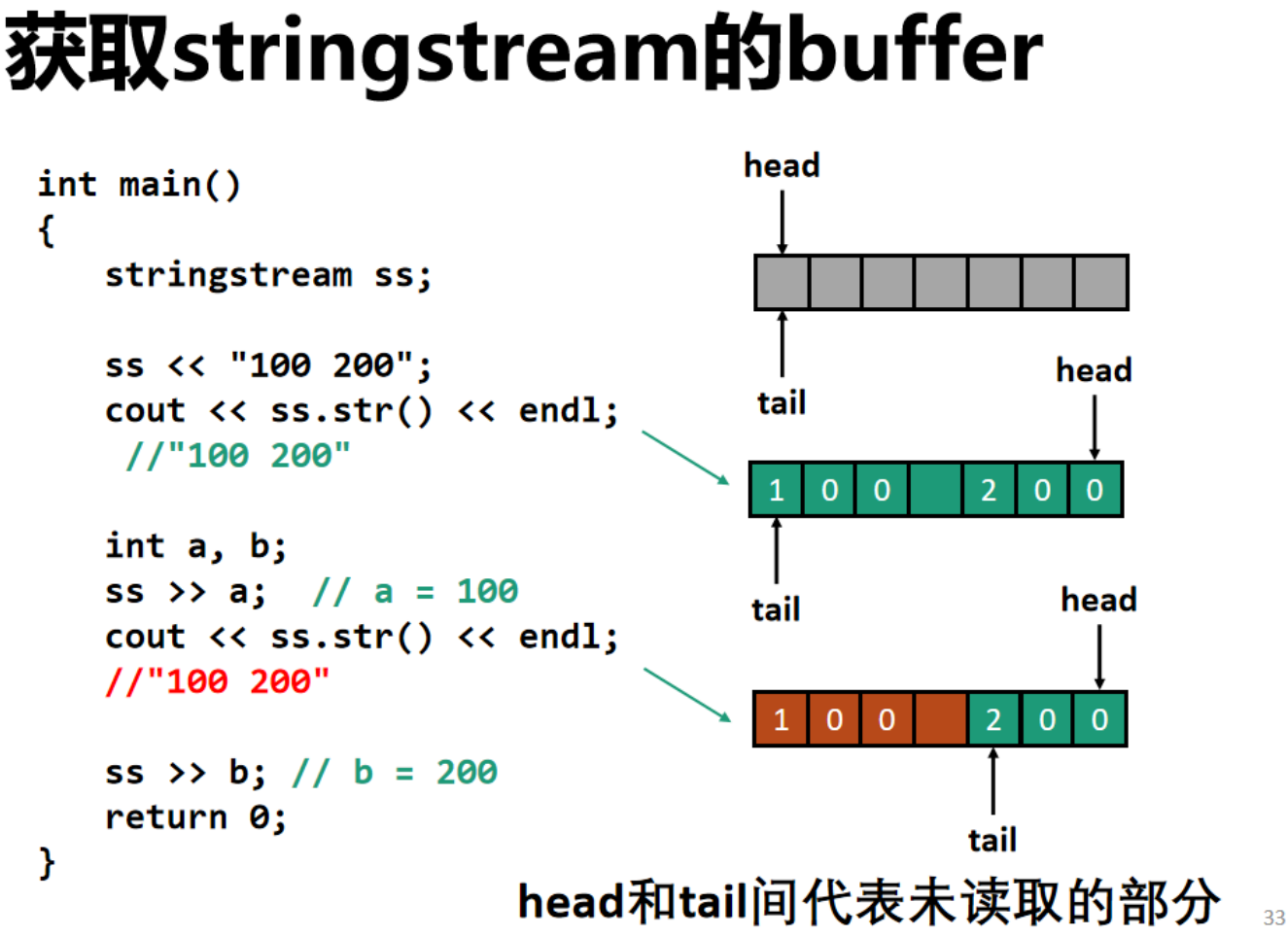
获取stringstream 的 buffer

- ss.str()

- 返回一个string对象
- 内容为stringstream的buffer

```
#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;
}
```



- head是等待读入的最后位置（还没有读过的在开头，已经读过的在尾巴）
- head到tail间表示当前等待读入的缓存区
- head在后面，tail在前面

实现类型转换函数

- to_string 转换为字符串

- stoi 转换为整数

```
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;
}
```

字符串处理与正则表达式

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 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
- 搜索：符合正则表达式的子串
- 替换：按规则替换字符串的子串

```
连用
    [a-z][0-9] //匹配所有字母+数字的组合，比如a1，b9...
特殊字符
    \d 等价于[0-9]，匹配所有单个数字
    \w 匹配字母、数字、下划线，等价[a-zA-Z0-9]
    . 匹配除换行以外的任意字符
    \. 匹配句号
重复模式
    x{n, m} 表示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
特殊字符
    + 前一个字符至少连续出现1次及以上
```

[正则表达式辅助工具][http://tool.chinaz.com/regex] 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

- 原生字符串可以取消转义，保留字面值

- 语法：
  - 1）字符串前加R前缀: R"(str)" 表示str的字面值
    - "\d+" = R"(\d+)" = \d+
  - （2）字符串首尾加上小括号；
- 原生字符串能换行，比如
  - string str = R"(Hello World)";
- 字符串里面可以带双引号

C++中使用匹配

- `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)) // 字符串s是否能完全匹配正则表达式re
        cout << "matched" << endl;
    return 0;
}
```

捕获和分组

- 引入：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

```
#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)) { // 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
        cout << sm.size() << " matches\n";
        cout << "the matches were:" << endl;
        for (unsigned i = 0; i < sm.size(); ++i) {
            cout << sm[i] << endl;
        }
    }
```

```
    return 0;
}
```

- 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

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

```
#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();
    }
    return 0;
}
```

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

```
#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";
    return 0;
}
//输出: this SUB has a SUB
```

- 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

```
#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;
    return 0;
}
/*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]*/

/*分别匹配到两组:
Subject $1 = sub $2=ject
Submarine $1= sub $2=marine
$0=[$&]*/
```

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

void extract(string input) {
    smatch sm;
    regex get_name(R"((My name is |I am )(\w+)\.)");
    regex get_date(R"((\d{4})[\.-](\d{1,2})[\.-](\d{1,2}))");
    regex get_mobile(R"([1-9]\d{10})");
```

```
    regex get_email(R"([\w]+@[\w\.]+)");
    if (std::regex_search(input, sm, get_name))
        cout << sm[2] << endl;
    int date[3] = {0};
    if (regex_search(input, sm, get_date)){
        for (int i = 1; i <= 3; i++)
            date[i - 1] = stoi(sm[i]);
        cout << date[0] << "." << date[1] << "." << date[2] <<endl;
    }
    if (regex_search(input, sm, get_mobile))
    cout << sm[0] << endl;
    if (regex_search(input, sm, get_email))
    cout << sm[0] << endl;
}

int main() {
    string str("I am zhangshuaishuai. \
        I was born on 2000.10.2. \
        My phone number is 18866667777 and you can also \
        reach me by my email: zhangss@tsinghua.edu.cn");
    extract(str);
    return 0;
}
/*
输出:
zhangshuaishuai
2000.10.2
18866667777
zhangss@tsinghua.edu.cn
*/
```

字符簇

```
转义字符
    \n表示换行
    \t表示制表符
范围取反
    [^a-z]: 匹配所有非小写字母的单个字符
    [^c]ar: The car **pa**rked in the garage.
    ^[^0-9][0-9]$: 匹配长度为2的内容，且第一个不为数字，第二个为数字
特殊字符
    \D 等价[^0-9]，匹配所有单个非数字
    \s 匹配所有空白字符，如\t,\n
    \S 匹配所有非空白字符
    \W 匹配非字母、数字、下划线，等价[^a-zA-Z0-9_]
    ^ 代表字符串开头，$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
```

重复模式

x{n,m}代表前面内容出现次数重复n~m次，可扩展到字符簇

- `[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

- `?` 出现0次或1次
  - `[T]?he`: The car parked in the garage.
- `+` 至少连续出现1次及以上 c.+e: The car parked in the garage.
- `*` 至少连续出现0次及以上
  - `[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 par**ked in the **gar**age.
- 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.

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

more

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