

C++程序设计(拾伍)

徐东/计算数学

内容

- 字符串
- 正则表达式

C++中的文本

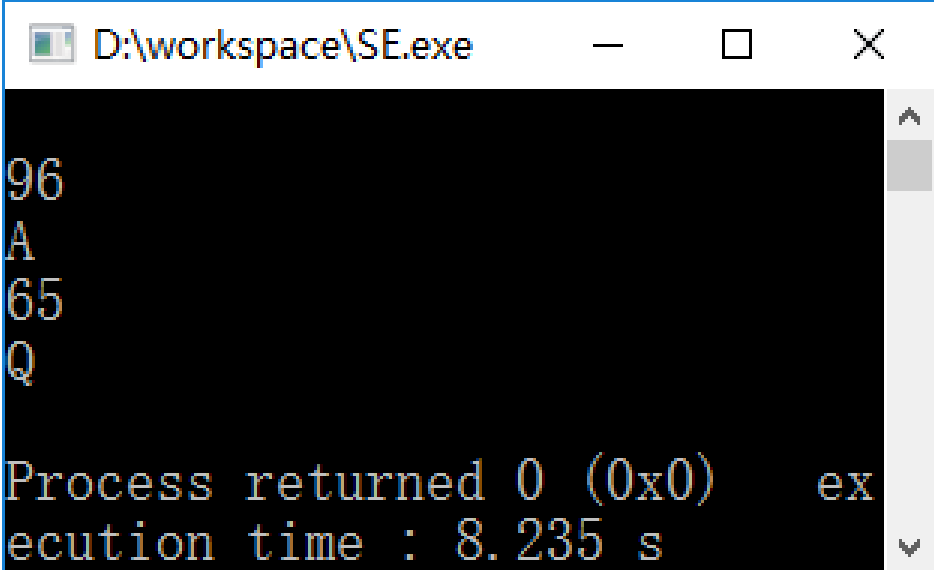
- 文本
 - 由一对双引号包围的一段字符序列
 - `“He has made his weapons his gods. When his weapons win he is defeated himself.”`
- 文本的处理方式
 - `char[]`
 - `string`

C++的文本

- char
 - 字符数据以ASCII码存储(以整数表示)
 - C++的字符数据和整型数据可以相互赋值

C++的文本

```
int main(){  
    int x = 96;  
    char ch = 'A';  
  
    cout << x << endl;  
    cout << ch << endl;  
  
    x = ch ;  
    cout << x << endl;  
  
    ch = 81;  
    cout << ch << endl;  
  
    return 0;  
}
```



```
D:\workspace\SE.exe  
96  
A  
65  
Q  
  
Process returned 0 (0x0)   ex  
ecution time : 8.235 s
```

char类型

- '0'
 - 数字字符
 - ASCII码等于值48或0X30
- 0
 - 整数值
- '\0' 和 NULL 表示整数 0

C++中的文本

- `char[]`
 - 字符数组
- 不安全
 - 必须考虑字符串的长度和数组的大小
 - 字符串的长度 < 数组大小
- 不推荐使用

char[]处理文本

- “shanghai”

's'	'h'	'a'	'n'	'g'	'h'	'a'	'i'	'\0'
-----	-----	-----	-----	-----	-----	-----	-----	------

- C++中的字符串以'\0'结束
- '\0'
 - 字符串结束标志

初始化字符数组

- 语法

- `char 数组名[数组长度]={“文本”};`

- `char 数组名[数组长度]= “文本”;`

- 数组的长度必须大于初始值文本的长度

- 未初始化的字符数组元素被自动赋值 `‘\0’`

- 数组长度可省略

char[]

- `char x[10] = "tom";`
 - 数组大小 10
- `char y[] = "jerry";`
 - 数组大小 6
- 系统自动在数组尾部添加 `'\0'`

char[]

- 错误的声明

- `char name = "tom";`

- `char ch[] = '\0';`

char[]

- **注意**

- 只能在声明语句中，以字符串常量的形式对字符数组赋初值。

- **错误**

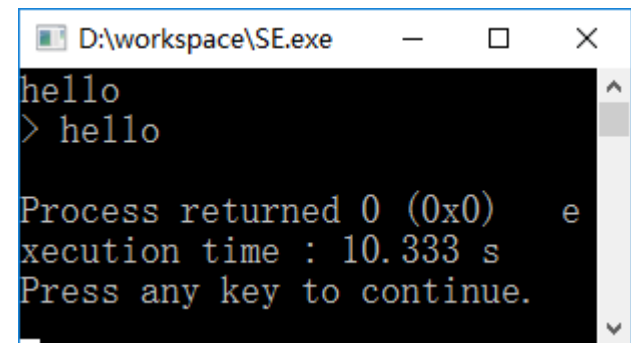
- `char text[100];`
- `text = "shanghai"; // error !!!`

char[]

- **输入整个字符串**
 - 在输入语句中直接使用字符数组名
 - 输入字符串的长度小于目标数组的长度
 - 系统自动向字符串尾部添加 `'\0'`
- **输出整个字符串**
 - 在输出语句中直接使用字符数组名
 - 输出的字符不包括 `'\0'`
 - 遇到第一个 `'\0'` 结束输出

char[]

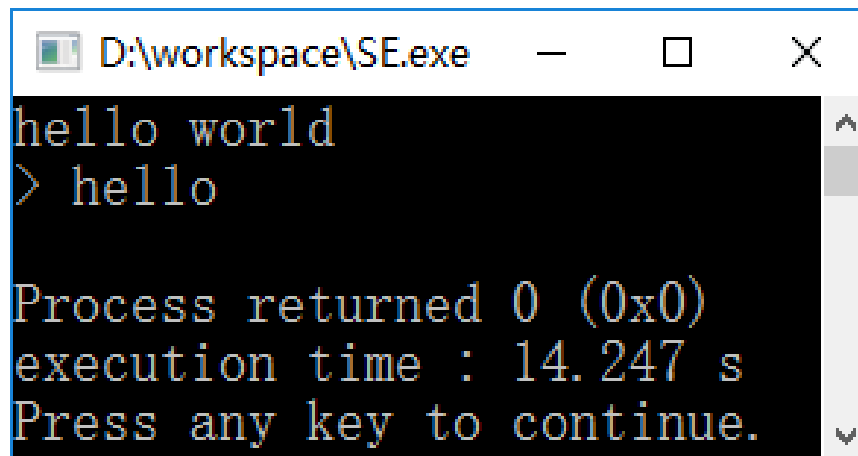
```
int main(){  
    const int NUMBER=256;  
    char text[NUMBER];  
  
    //通过键盘输入字符串  
    cin >> text ;  
  
    //在屏幕上，输出保存在数组中的字符串  
    cout << "> " << text << endl;  
  
    return 0;  
}
```



```
D:\workspace\SE.exe  
hello  
> hello  
  
Process returned 0 (0x0) e  
Execution time : 10.333 s  
Press any key to continue.
```

char[]

```
int main(){  
    const int NUMBER=256;  
    char text[NUMBER];  
  
    //通过键盘输入字符串  
    cin >> text ;  
  
    //在屏幕上，输出保存在数组中的字符串  
    cout << "> " << text << endl;  
  
    return 0;  
}
```



```
D:\workspace\SE.exe  
hello world  
> hello  
  
Process returned 0 (0x0)  
execution time : 14.247 s  
Press any key to continue.
```

- 需要读取包含空格的一行文本

char[]

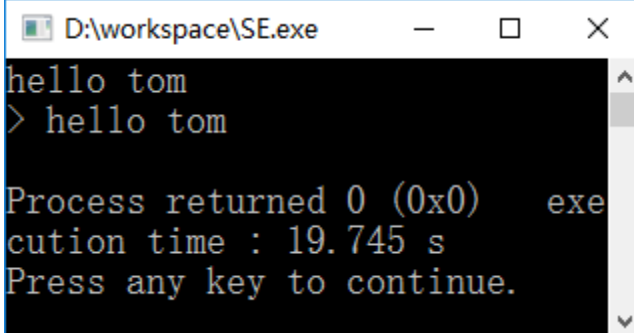
- 读取包含空格的一行文本

- `cin.getline(字符数组名, 字符个数+1, 结束字符);`

char[]

```
int main(){  
    const int NUMBER=256;  
    char text[NUMBER];  
  
    cin.getline(text, 10, '\n');  
    cout << "> " << text << endl;  
  
    return 0;  
}
```

- 遇到结束字符（'\n'）结束输入



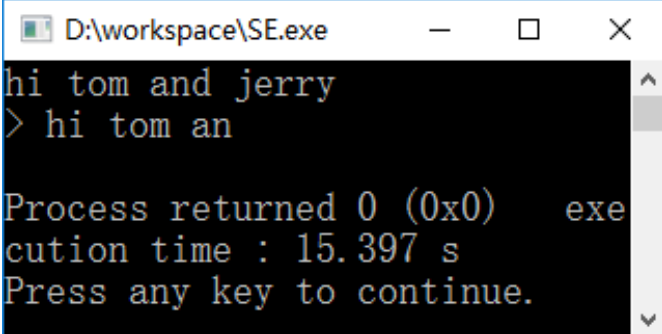
```
D:\workspace\SE.exe  
hello tom  
> hello tom  
  
Process returned 0 (0x0)   exe  
Execution time : 19.745 s  
Press any key to continue.
```

char[]

```
int main(){
    const int NUMBER=256;
    char text[NUMBER];

    cin.getline(text, 10, '\n');
    cout << "> " << text << endl;

    return 0;
}
```



```
D:\workspace\SE.exe
hi tom and jerry
> hi tom an

Process returned 0 (0x0)   exe
cution time : 15.397 s
Press any key to continue.
```

- **读取规定字符数量结束输入**

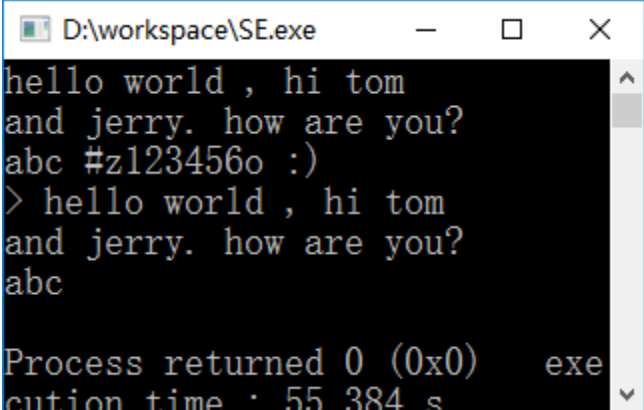
char[]

```
int main(){
    const int NUMBER=256;
    char text[NUMBER];

    cin.getline(text,
                NUMBER - 1, '#');
    cout << "> " << text << endl;

    return 0;
}
```

- 遇到结束字符（'#'）结束输入



```
D:\workspace\SE.exe
hello world , hi tom
and jerry. how are you?
abc #z123456o :)
> hello world , hi tom
and jerry. how are you?
abc

Process returned 0 (0x0)   exe
Execution time : 55.384 s
```

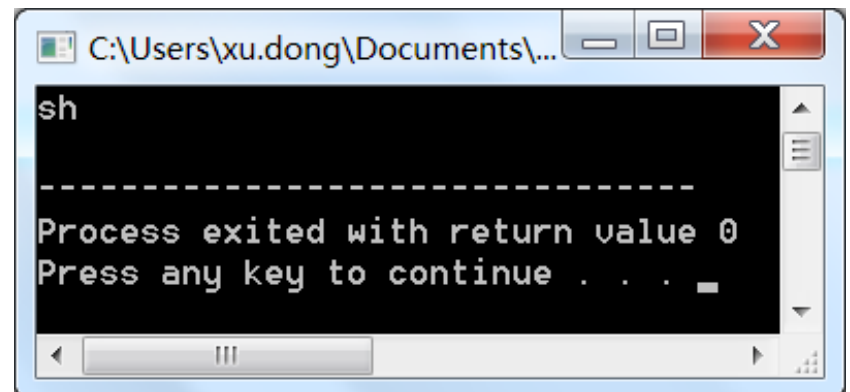
char[]

- 在字符数组中，字符串由 '`\0`' 之前的字符符号组成。

```
char city[] = "shanghai" ;
```

```
city[2] = '\0';
```

```
cout << city << endl;
```



char[]

```
const int size=9;
```

```
char name[size];
```

```
cin >> name;
```

- **name中允许存放的字符串的最大长度**
 - **8个字符**
- **输入字符串大于8个字符(危险)**
 - **改写数组空间以外的内存单元**

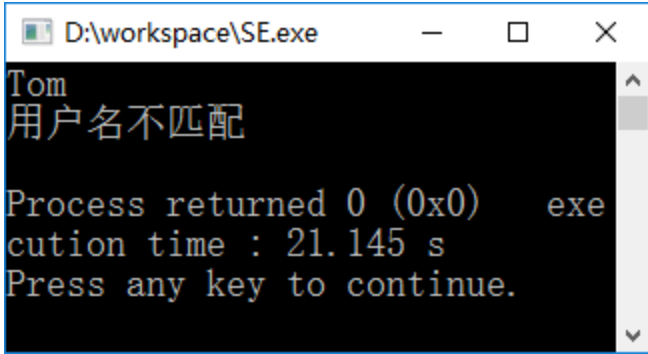
验证用户名是否相同

```
int main(){
    const int NUMBER=256;
    char user[NUMBER] = "Tom";

    char name[NUMBER] ;
    cin >> name ;

    if( name == user ){
        cout << "用户名匹配" << endl;
    }
    else{
        cout << "用户名不匹配" << endl;
    }
    return 0;
}
```

- 指针比较（是否指向同一个地址）



```
D:\workspace\SE.exe
Tom
用户名不匹配

Process returned 0 (0x0)   exe
cution time : 21.145 s
Press any key to continue.
```

验证用户名是否相同

```
int main(){
    const int NUMBER=256;
    char user[NUMBER] = "Tom";

    char name[NUMBER] ;
    cin >> name ;

    //比较字符数组的内容是否相同
    if(                ){
        cout << "用户名匹配" << endl;
    }
    else{
        cout << "用户名不匹配" << endl;
    }
    return 0;
}
```

验证用户名是否相同

```
int main(){
    const int NUMBER=256;
    char user[NUMBER] = "Tom";

    char name[NUMBER] ;
    cin >> name ;

    //比较字符数组的内容是否相同
    if(strcmp(name, user) == 0){
        cout << "用户名匹配" << endl;
    }
    else{
        cout << "用户名不匹配" << endl;
    }
    return 0;
}
```


使用char[]处理字符串

- 部分常用操作

功能	char[]
字符串复制	<code>strcpy(char[], const char[])</code>
字符串连接	<code>strcat(char[], const char[])</code>
字符串比较	<code>strcmp(const char[], const char[])</code>
字符串长度	<code>strlen(const char[])</code>

- `#include<cstring>`

- 确保字符数组的大小足够容纳字符串

C++中的文本

- `string`
 - 自动改变长度以容纳字符串的全部内容
 - 安全处理文本数据
 - 不是C++基本数据类型
- `#include<string>`

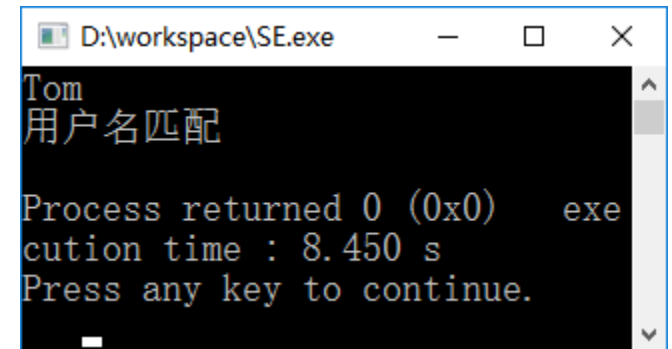
验证用户名是否相同 (string 版本)

```
int main(){
    string name;
    string user = "Tom";

    cin >> name ;

    if( name == user ){
        cout << "用户名匹配" << endl;
    }
    else{
        cout << "用户名不匹配" << endl;
    }

    return 0;
}
```



```
D:\workspace\SE.exe
Tom
用户名匹配

Process returned 0 (0x0)   exe
cution time : 8.450 s
Press any key to continue.
```

安全处理文本的数据类型string

- 声明语句

- `string 变量名 [= 初始值];`

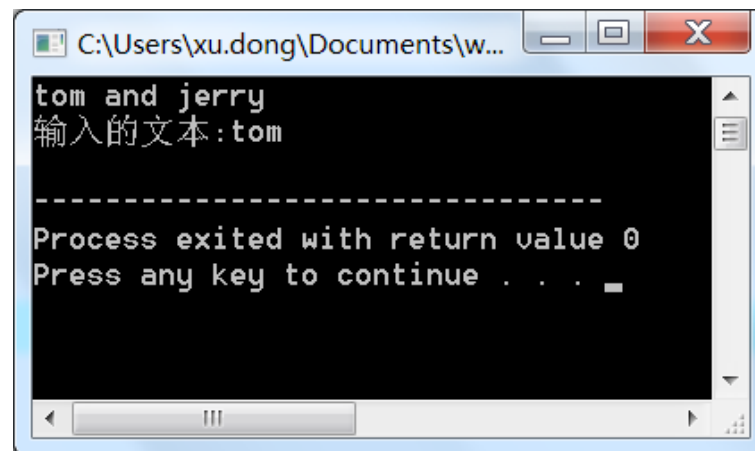
- `#include<string>`

- `string x1;`

- `string x2 = "hello world";`

string: 读取文本

```
string word;  
cin >> word;  
cout << “输入的文本 :”  
      << word  
      << endl;
```



```
tom and jerry  
输入的文本:tom  
-----  
Process exited with return value 0  
Press any key to continue . . .
```

- 空白符作为数据项间的分隔符

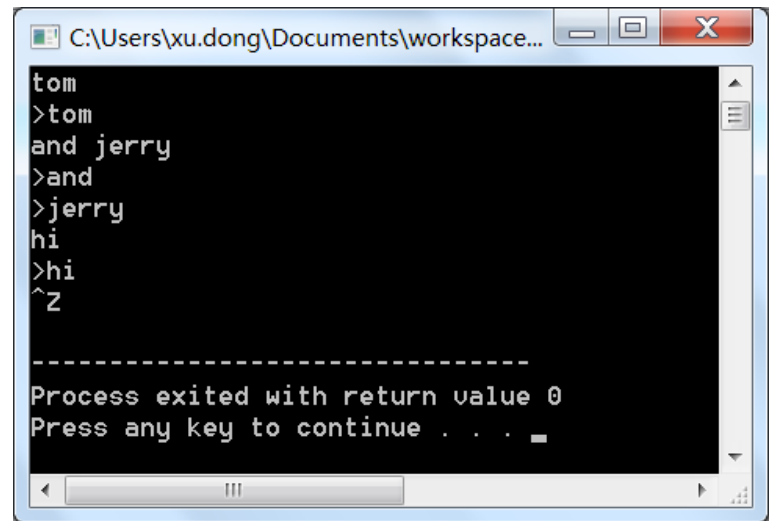
C++中的文本

- 空白符
 - `\t`
 - `\n`
 - `space`

string: 读取文本

```
string word;  
while(cin>>word){  
    cout << ">"  
        << word  
        << endl;  
}
```

- Ctrl + z 结束输入
- 需要读取包含空格的一行文本?



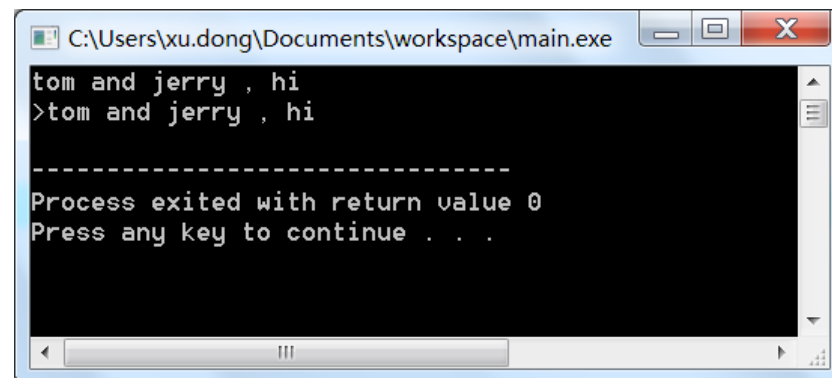
```
C:\Users\xu.dong\Documents\workspace...  
tom  
>tom  
and jerry  
>and  
>jerry  
hi  
>hi  
^Z  
-----  
Process exited with return value 0  
Press any key to continue . . .
```

string

- 读取包含空格的一行文本
 - `getline(in, str_var);`
- `in`
 - 输入流（外部文件等）
 - `cin`
- `str_var`
 - 保存读入文本的string变量

string

```
string line;  
getline(cin, line);  
cout << ">"  
      << line  
      << endl;
```



A screenshot of a Windows command prompt window titled "C:\Users\xu.dong\Documents\workspace\main.exe". The window has a black background with white text. It shows the program's output: "tom and jerry , hi" followed by a prompt ">". The user has entered "tom and jerry , hi". Below this, a dashed line separates the input from the program's exit message: "Process exited with return value 0" and "Press any key to continue . . .".

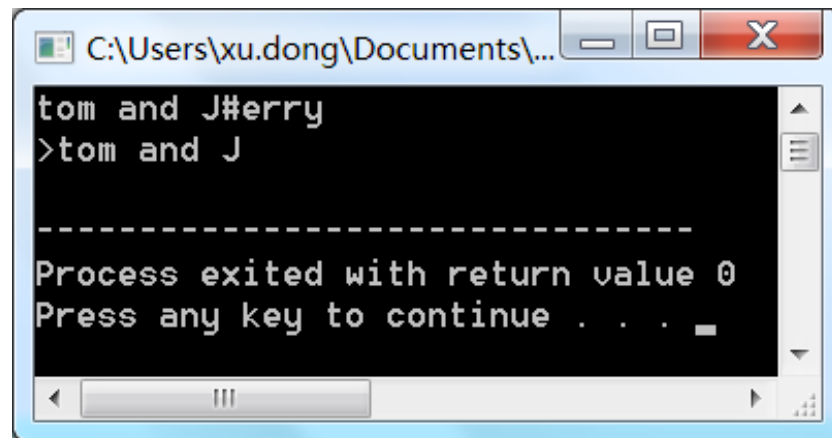
- 读取包含空格的整行文本

string

- 读取以特定符号结尾的一段文本
 - `getline(in, str_var, end_ch);`
- `end_ch`
 - 结束字符
 - 读取的文本不包含结束字符

string

```
string line;  
getline(cin,line,'#');  
cout << ">"  
      << line  
      <<endl;
```



```
C:\Users\xu.dong\Documents\...  
tom and J#erry  
>tom and J  
  
-----  
Process exited with return value 0  
Press any key to continue . . .
```

- 读取以 '#' 为结束标志的一段文本
 - 不包含结束字符

string

部分常用操作

- `s1=s2`
 - 将s2的内容赋予s1
- `s2`
 - 字符串对象
 - C风格字符串

说明

- 错误
 - `string x='a';`

string

部分常用操作

- `s+=x`
 - 将x添加到s的末尾
- `x`
 - 字符
 - 字符串
 - C风格字符串

说明

- `string text="tom";`
- `text += "and jerr";`
- `text += 'y';`

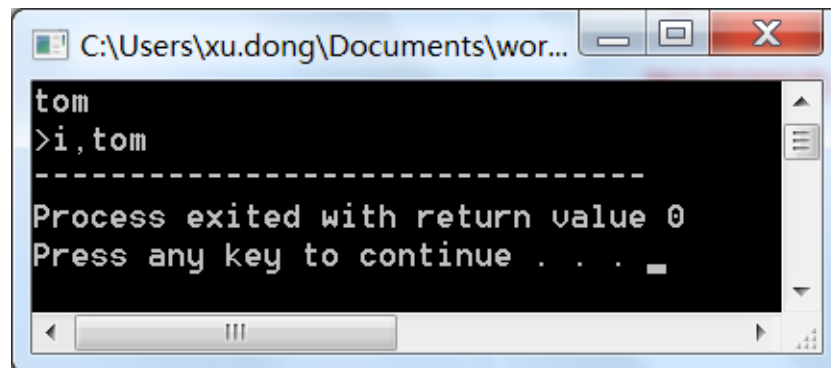
string

部分常用操作

- `s[i]`
 - 数组下标
 - `s`的第*i*个字符
 - 索引值从零开始

说明

- `string text="hi,tom";`
- `for(int i=3;i<6;++i)`
 - `cout<<text[i];`
- `text[0]='>';`
- `cout<<"\n"<<text;`



```
C:\Users\xu.dong\Documents\wor...
tom
>i,tom
-----
Process exited with return value 0
Press any key to continue . . .
```

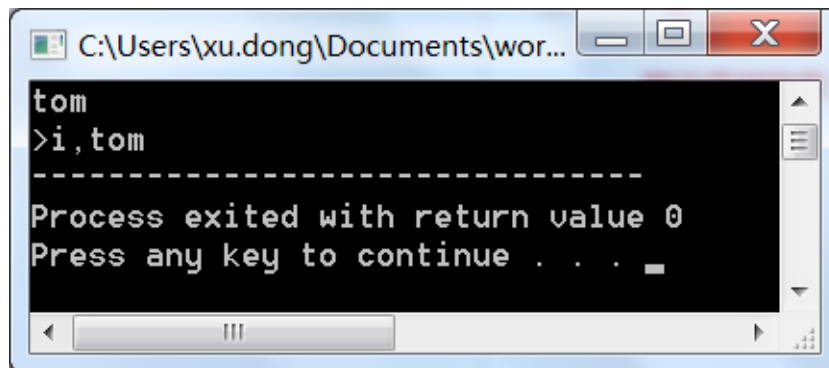
string

部分常用操作

- `s.at(i)`
 - 返回s的第i个字符
 - 索引值从零开始
 - 带边界检查

说明

- `string text="hi,tom";`
- `for(int i=3;i<6;++i)`
 - `cout<<text.at(i);`
- `text[0]='>';`
- `cout<<"\n"<<text;`



```
C:\Users\xu.dong\Documents\wor...
tom
>i,tom
-----
Process exited with return value 0
Press any key to continue . . .
```

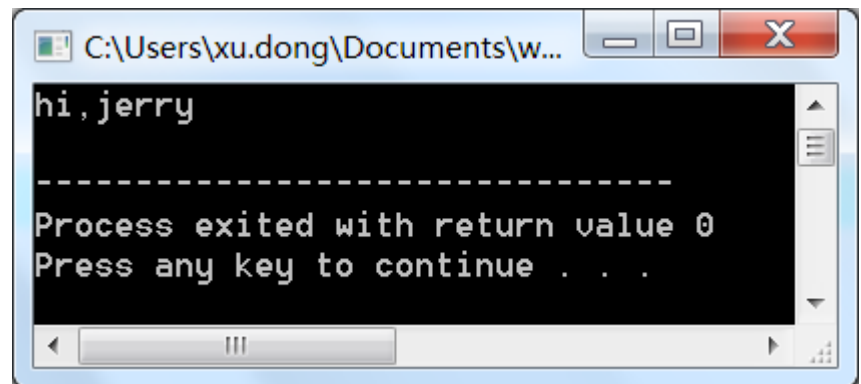
string

部分常用操作

- `s1+s2`
 - 连接两个字符串

说明

- `string x="hi";`
- `string y=",jerry";`
- `cout<< x+y`
`<<endl;`



A screenshot of a Windows command prompt window. The title bar shows the file path "C:\Users\xu.dong\Documents\w...". The window contains the following text: "hi,jerry", a dashed line "-----", "Process exited with return value 0", and "Press any key to continue . . .". The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

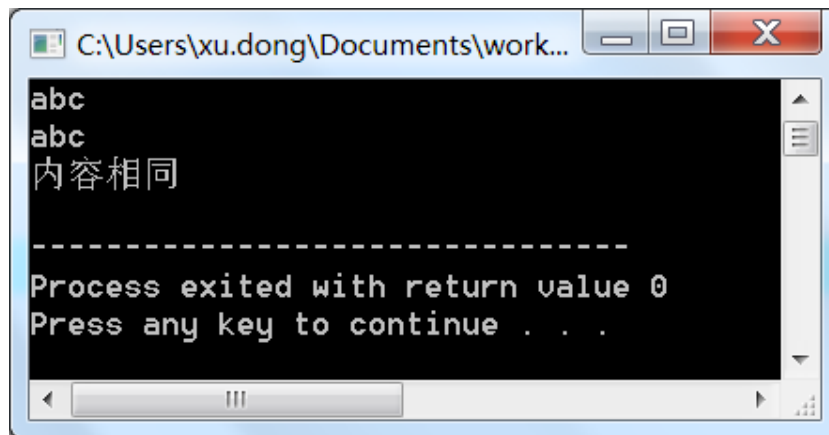
string

部分常用操作

- `s1==s2`
 - 比较字符串的值
 - `s1`或`s2`可以是C风格字符串但不允许两者皆是
- 同 `!=`

说明

- `string x,y;`
- `cin>>x>>y;`
- `if(x==y)`
 - `cout<<“内容相同”;`



```
C:\Users\xu.dong\Documents\work...
abc
abc
内容相同

-----
Process exited with return value 0
Press any key to continue . . .
```

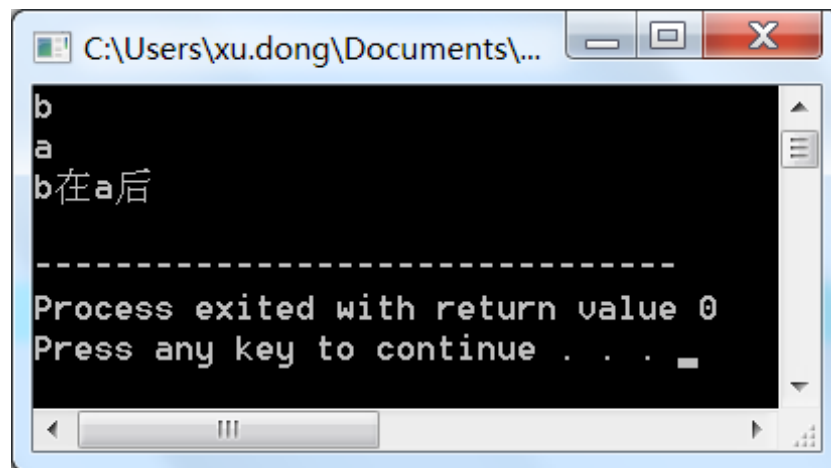
string

部分常用操作

- `s1 < s2`
 - 根据字典比较法比较两个字符串的先后
 - `s1`或`s2`可以是C风格字符串但不允许两者皆是
- 同 `<=` `>=` `>`

说明

- `string x, y;`
- `cin >> x >> y;`
- `if (x > y)`
 - `cout << x << "在" << y << "后";`



```
C:\Users\xu.dong\Documents\...  
b  
a  
b在a后  
-----  
Process exited with return value 0  
Press any key to continue . . .
```

string

部分常用操作

- `s.size()`
 - 返回s中字符的数目
- `s.length()`
 - 返回s中字符的数目

说明

- `string x;`
- `cin>>x;`
- `cout<<x<<"的字符数目是"`
`<<x.length()<<endl;`
- `cout<<x.size()<<endl;`



```
C:\Users\xu.dong\Documents\...  
hello  
hello的字符数目是5  
5  
-----  
Process exited with return value 0  
Press any key to continue . . .
```

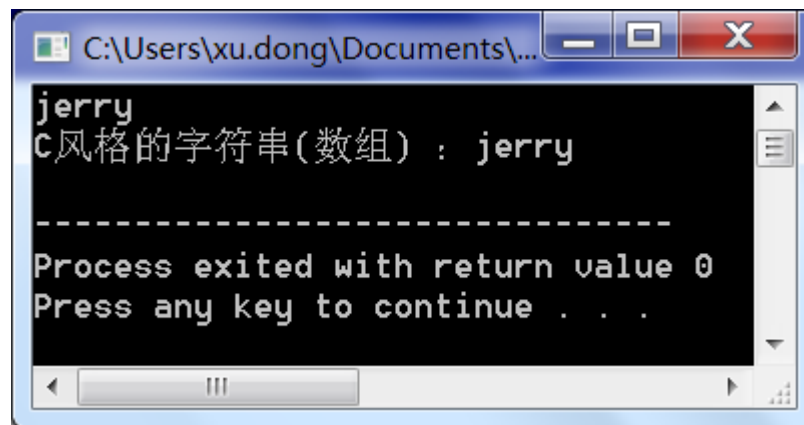
string

部分常用操作

- `s.c_str()`
 - 返回s中字符构成的C风格字符串

说明

- `string x;`
- `cin>>x;`
- `cout<<"C风格的字符串(数组):"<<x.c_str()<<endl;`



```
C:\Users\xu.dong\Documents\...
jerry
C风格的字符串(数组) : jerry
-----
Process exited with return value 0
Press any key to continue . . .
```

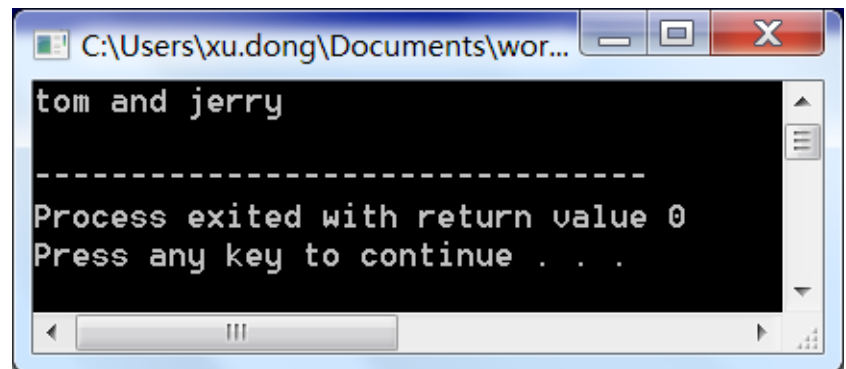
string

部分常用操作

- `s.insert(pos,x)`
 - 将x插入到s[pos]之前的位置
- x
 - 字符
 - 字符串
 - C风格字符串

说明

- `string x="tomjerry";`
- `x.insert(3," and ");`
- `cout<<x<<endl;`



```
C:\Users\xu.dong\Documents\wor...
tom and jerry
-----
Process exited with return value 0
Press any key to continue . . .
```

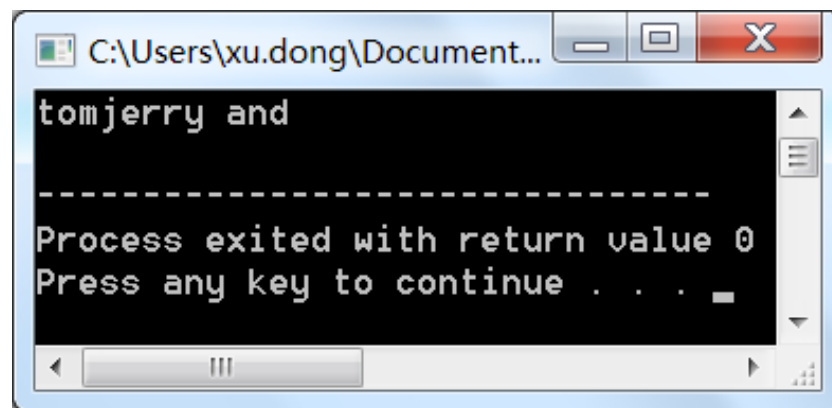
string

部分常用操作

- `s.append(x)`
 - 将x连接到s的尾部
- `x`
 - 字符串
 - C风格字符串

说明

- `string x="tomjerry";`
- `x.append(" and ");`
- `cout<<x<<endl;`



A screenshot of a Windows command prompt window. The title bar shows the file path "C:\Users\xu.dong\Document...". The window contains the following text: "tomjerry and", followed by a dashed line separator, and then "Process exited with return value 0" and "Press any key to continue . . .".

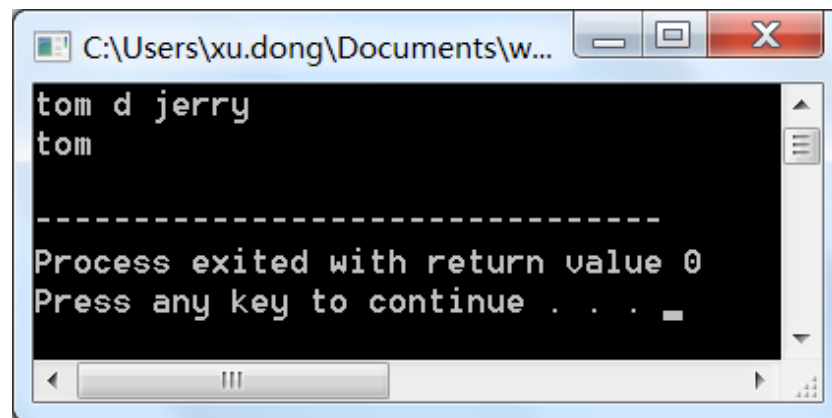
string

部分常用操作

- `s.erase(pos,n)`
 - 删除`s[pos]`起的`n`个字符

说明

- `string x="tom and jerry";`
- `x.erase(4,2);`
- `cout<<x<<endl;`
- `x.erase(3);`
- `cout<<x<<endl;`



```
C:\Users\xu.dong\Documents\w...
tom d jerry
tom
-----
Process exited with return value 0
Press any key to continue . . .
```

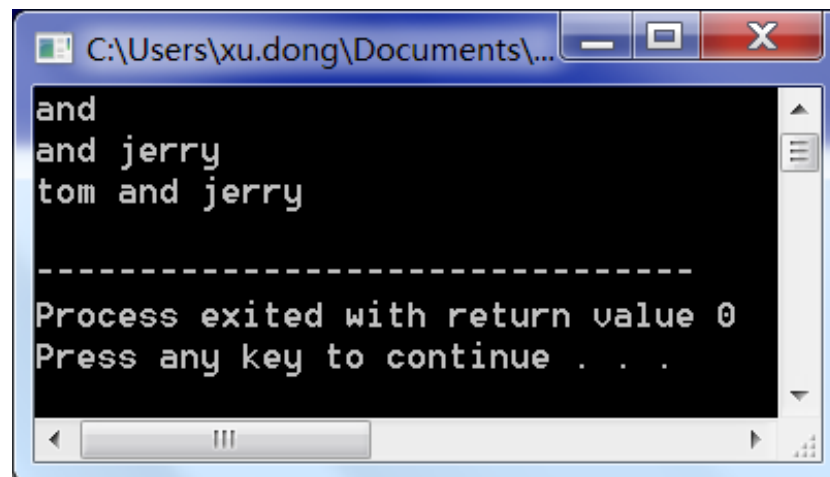
string

部分常用操作

- `s.substr(pos,n)`
 - 返回从pos开始的n个字符组成的字符串(子串)

说明

- `string x="tom and jerry";`
- `cout<<x.substr(4,3)<<endl`
- `<<x.substr(4) <<endl`
- `<<x <<endl;`



```
and
and jerry
tom and jerry

-----
Process exited with return value 0
Press any key to continue . . .
```

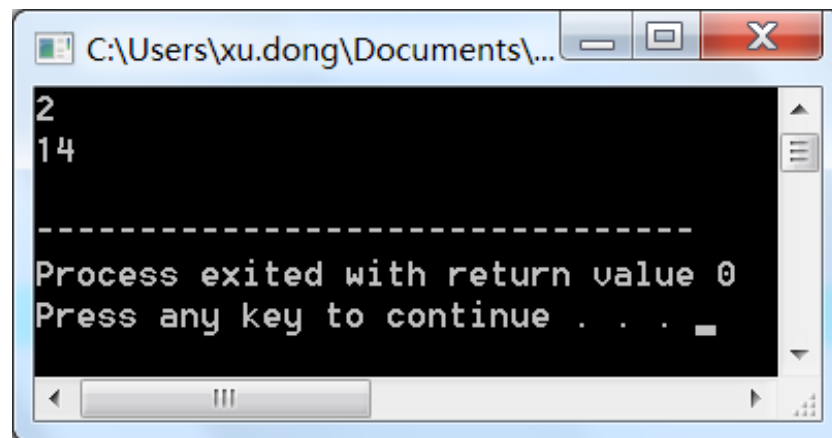

string

部分常用操作

- `s.find(x, pos)`
 - 从pos处开始查找x在当前字符串中的位置
- `x`
 - 字符
 - 字符串
 - C风格字符串

说明

- `string x=`
 - `"tom and jerry and mike";`
 - `cout<<x.find('m')<<endl;`
 - `cout<<x.find("and",6)`
 - `<<endl;`



```
C:\Users\xu.dong\Documents\...
2
14
-----
Process exited with return value 0
Press any key to continue . . .
```

处理文本的两种策略

string

- `#include<string>`
- `string stext;`
- `string st;`
- `cin>>stext ;`
- `st=stext;`
- `cout<< st <<endl;`
- `st=st+stext;`
- `cout<< st <<endl;`
- `if(st>stext) cout<<“abc\n”;`
- `cout<<“字符串长度”<<st.size();`

char[]

- `#include<cstring>`
- `const int size=256;`
- `char ctext[size], ct[size];`
- `cin>>ctext ;`
- `strcpy(ct,ctext);`
- `cout<< ct <<endl;`
- `strcat(ct , ctext);`
- `cout<< ct <<endl;`
- `if(strcmp(ct, ctext)>0)`
 - `cout<< “abc\n”;`
- `cout<<“字符串长度”<<strlen(ct);`

处理回文

string

- `string s;`
- `cin>>s;`
- `bool is_palindrome=true;`
- `int first=0;`
- `int last=s.length()-1;`
- `while(first<last){`
 - `if(s[first]!=s[last]){`
 - `is_palindrome=false;`
 - `break;`
 - `}`
 - `++first;`
 - `--last;`
- `}`

char[]

- `const int n=30;`
- `char ch[n];`
- `cin>>ch;`
- `bool is_palindrome=true;`
- `int first=0, last=n-1;`
- `while(first<last){`
 - `if(ch[first]!=ch[last]){`
 - `is_palindrome=false;`
 - `break;`
 - `}`
 - `++first;`
 - `--last;`
- `}`

正则表达式

- regular expression
 - 允许使用通配符和模式（pattern）查找和替换string中的字符
- Match 将整个输入比对（匹配）某个正则表达式
- Search 查找“与正则表达式吻合”的pattern
- Tokenize 根据“被指定为正则表达式”的切分器（separator）取得语汇单元（token）
- Replace 将与正则表达式吻合的第一个（或后续所有）子序列替换掉

正则表达式

- 检查字符串是否匹配或局部匹配某个正则表达式
- 寻找XML/HTML文档中的标签纸
 - `<tag>value</tag>`

匹配正则表达式

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

void out(bool b){
    cout << (b ? "found" : "not found") << endl;
}

int main(){
    //using default syntax
    regex reg1("<.*>.*</.*>");
    bool found = regex_match("<tag>value</tag>", reg1);
    out(found);

    return 0;
}
```

匹配正则表达式

- 正则表达式

- `<.*>.*</.*>`

- 检验

- `<someChars>someChars</someChars>`

- 点号 (.) 除newline以外的任何字符

- 星号 (*) 0次或多次

匹配正则表达式

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

void out(bool b){
    cout << (b ? "found" : "not found") << endl;
}

int main(){
    //using default syntax
    regex reg1("<.*>.*</.*>");
    bool found = regex_match("<tag>value</tag>", reg1);
    out(found);

    return 0;
}
```


匹配正则表达式

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

void out(bool b){
    cout << (b ? "found" : "not found") << endl;
}

int main(){
    //tags before and after the value must match
    regex reg2("<(.*?)>.*</\\1>");
    bool found = regex_match("<tag>value</tag>", reg2);
    out(found);

    return 0;
}
```

匹配正则表达式

- 正则表达式

- `<(.*>.*</\\1>`

- (...)

- capture grouping

- 正则表达式 `\1` 再次指代它

- 正则表达式是一个普通的字符序列

- `\1` \rightarrow `\\1`

匹配正则表达式

- 正则表达式

- `<(.*)>.*</\1>`

- 在 C++11 中的等价形式

- `R"(<(.*)>.*</\1>)"`

- Raw string

- 以 `R" (` 开始

- 以 `)"` 结束

匹配正则表达式

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

void out(bool b){
    cout << (b ? "found" : "not found") << endl;
}

int main(){
    //using grep syntax
    regex reg3("<\\(.*\\)>.*</\\1>", regex_constants::grep);
    bool found = regex_match("<tag>value</tag>", reg3);
    out(found);

    return 0;
}
```

匹配正则表达式

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

void out(bool b){
    cout << (b ? "found" : "not found") << endl;
}

int main(){
    //needs explicit cast to regex
    bool found = regex_match("<tag>value</tag>",
                             regex("<(.*).*</\\1>"));

    out(found);

    return 0;
}
```

匹配正则表达式

```
regex_match("<tag>value</tag>",           //Error:ambiguous  
            "<(.*).*</\\1>");
```

```
regex_match(string("<tag>value</tag>"), //Error:ambiguous  
            "<(.*).*</\\1>");
```

```
regex_match("<tag>value</tag>",           //OK  
            regex("<(.*).*</\\1>"));
```

匹配正则表达式

- `regex_match()` 和 `regex_search()` 的差异
 - `regex_match()`
 - 检验是否整个字符串匹配（吻合）某个正则表达式
 - `regex_search()`
 - 检验是否部分字符序列匹配（吻合）某个正则表达式

匹配正则表达式

- `regex_search(data, regex(pattern))`
- 等价于
- `regex_match(data,`
- `regex("(.\|\\n)*" + pattern + "(.\|\\n)*")`
- `)`
- 其中, `(.\|\\n)*` 表示任何数量和任何字符。

匹配正则表达式

```
int main(){

    bool found = regex_match("XML tag: <tag>value</tag>",
                             regex("<(.*).*</\\1>"));
    out(found);

    found = regex_search("XML tag: <tag>value</tag>",
                          regex("<(.*).*</\\1>"));
    out(found);

    return 0;
}
```

处理Subexpression (次表达式)

```
int main(){
    string data = "XML tag: <tag-name>the value</tag-name>.";
    cout << "data:          " << data << "\n\n";

    smatch m; // for returned details of the match
    bool found = regex_search(data,
                               m,
                               regex("<(.*?)>(.*?)</(\\1)>"));

    //print match details
    cout << "m.empty():      " << boolalpha << m.empty() << endl;
    cout << "m.size():         " << m.size() << endl;
    if(found){
        cout << "m.str():          " << m.str() << endl;
        cout << "m.length():       " << m.length() << endl;
        cout << "m.position():      " << m.position() << endl;
        cout << "m.prefix().str():  " << m.prefix().str() << endl;
        cout << "m.suffix().str():  " << m.suffix().str() << endl;
        cout << endl;
    }
    return 0;
}
```

处理Subexpression (次表达式)

```
int main(){
    string data = "XML tag: <tag-name>the value</tag-name>.";
    cout << "data:          " << data << "\n\n";

    smatch m; // for returned details of the match
    bool found = regex_search(data,
                               m,
                               regex("<(.*?)>(.*?)</(\\1)>"));

    //print match details
    cout << "m.empty():      " << boolalpha << m.empty() << endl;
    cout << "m.size():        " << m.size() << endl;
    if(found){
        //iterating over all matches(using the match index):
        for(int i = 0; i < m.size(); ++i){
            cout << "m[" << i << "].str():      " << m[i].str() << endl;
            cout << "m.str(" << i << "):          " << m.str(i) << endl;
            cout << "m.position(" << i << "):    " << m.position(i) << endl;
        }
    }
    return 0;
}
```

处理Subexpression (次表达式)

```
int main(){
    string data = "XML tag: <tag-name>the value</tag-name>.";
    cout << "data:          " << data << "\n\n";

    smatch m; // for returned details of the match
    bool found = regex_search(data,
                               m,
                               regex("<(.*?)>(.*?)</(\\1)>"));

    //print match details
    cout << "m.empty():      " << boolalpha << m.empty() << endl;
    cout << "m.size():        " << m.size() << endl;
    if(found){
        //iterating over all matches(using iterators)
        cout << "matches:" << endl;
        for(auto pos = m.begin(); pos != m.end(); ++pos){
            cout << " " << *pos << " ";
            cout << "(length: " << pos->length() << ")" << endl;
        }
    }
    return 0;
}
```

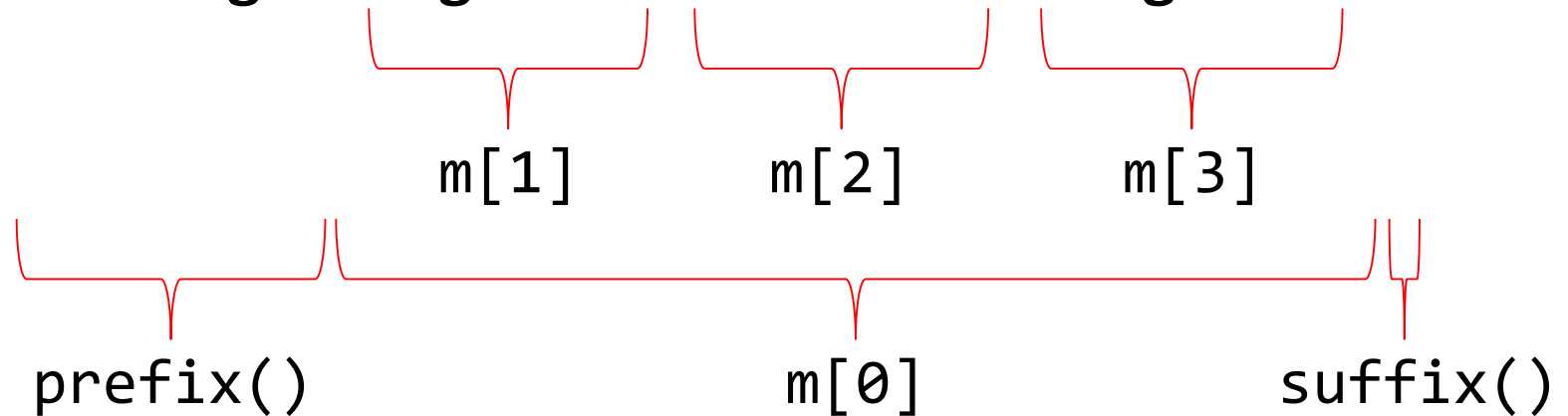
处理Subexpression (次表达式)

- 正则表达式

- `<(.*>(.*</(\1)>`

- 目标字符串

- "XML tag: `<tag-name>the value</tag-name>.`"



处理Subexpression (次表达式)

- `match_results`对象内含：
 - `sub_match`对象 `m[0]`，代表“匹配合格的所有字符”。
 - `sub_match`对象`prefix()`，代表第一个匹配合格的字符的前方所有字符。
 - `sub_match`对象`suffix()`，代表最后一个匹配合格的字符的后方所有字符。

处理Subexpression (次表达式)

- match_results对象内含:
 - 对于任何 capture group, 存在对应的sub_match对象
m[n]

<tag-name>the value</tag-name>

m[1] m[2] m[3]

- 三个 capture group

处理Subexpression（次表达式）

- `match_results`对象内含：
 - `size()` 返回 `sub_match` 对象的个数（包括 `m[0]`）。
 - 所有 `sub_match` 对象都派生自 `pair<>`
 - `first` 成员是第一个字符的位置
 - `Second` 成员是最末字符的下一个位置
 - `str()`返回string形式的字符串
 - `length()`返回字符数量

处理Subexpression (次表达式)

- match_results对象
 - 成员函数 str()
 - str() 或 str(0) 返回 “匹配合格的整体string”
 - str(n) 返回 “第 n 个匹配合格的substring”
 - 如果不存在这样的substring, 那么返回空字符串。

处理Subexpression (次表达式)

- match_results对象
 - 成员函数 length()
 - length() 或 length(0) 返回 “匹配合格的整体 string 的长度”
 - length(n) 返回 “第 n 个匹配合格的substring 的长度”
 - 如果不存在这样的substring, 那么返回0。

处理Subexpression (次表达式)

- match_results对象
 - 成员函数 position()
 - position() 或 position(0) 返回 “匹配合格的整体 string 的位置”
 - position(n) 返回 “第 n 个匹配合格的 substring 的位置”

处理Subexpression (次表达式)

- match_results对象
 - 成员函数 begin()、cbegin()、end() 和 cend() 可用来迭代 sub_match 对象, 从 m[0] 到 m[n]。

```
//iterating over all matches(using iterators)
cout << "matches:" << endl;
for(auto pos = m.begin(); pos != m.end(); ++pos){
    cout << " " << *pos << " ";
    cout << "(length: " << pos->length() << ")" << endl;
}
```

处理Subexpression (次表达式)

- 从 `match_result` 中获得匹配 (整体) 字符串的四种方法
 - `m.str()` //yields whole matches string
 - `m.str(0)`
 - `m[0].str()`
 - `*(m.begin())`

处理Subexpression (次表达式)

- 从 `match_result` 中获得第n个匹配子串的三种方法
 - `m.str(1)` //yields first matches substring
 - `m[1].str()`
 - `*(m.begin() + 1)`

处理Subexpression (次表达式)

```
int main(){
    string data = "<person>\n"
                  "<first>Nico</first>\n"
                  "<last>Josuttis</last>\n"
                  "</person>\n";
    regex reg("<(.*?)>(.*?)</(\\1)>");

    //iterate over all matches
    auto pos = data.cbegin();
    auto end = data.cend();
    smatch m;
    for(; regex_search(pos,end,m,reg); pos = m.suffix().first){
        cout << "match:  " << m.str() << endl;
        cout << " tag:    " << m.str(1) << endl;
        cout << " value:  " << m.str(2) << endl;
    }
    return 0;
}
```

处理Subexpression (次表达式)

```
int main(){
    string data = "<person>\n"
                  " <first>Nico</first>\n"
                  " <last>Josuttis</last>\n"
                  "</person>\n";
    regex reg("<(.*?)>(.*?)</(\\1)>");

    //iterate over all matches
    auto pos = data.cbegin();
    auto end = data.cend();
    smatch m;
    for(; regex_search(pos,end,m,reg); pos = m.suffix().first){
        cout << "match:  " << m.str() << endl;
        cout << " tag:    " << m.str(1) << endl;
        cout << " value:  " << m.str(2) << endl;
    }
    return 0;
}
```


处理Subexpression (次表达式)

```
int main(){
    string data = "<person>\n"
                  " <first>Nico</first>\n"
                  " <last>Josuttis</last>\n"
                  "</person>\n";
    regex reg("<(.*?)>(.*?)</(\\1)>");

    //iterate over all matches
    auto pos = data.cbegin();
    auto end = data.cend();
    smatch m;
    for(; regex_search(pos, end, m, reg); pos = m.suffix().first){
        cout << "match:  " << m.str() << endl;
        cout << " tag:    " << m.str(1) << endl;
        cout << " value:  " << m.str(2) << endl;
    }
    return 0;
}
```

处理Subexpression (次表达式)

```
int main(){
    string data = "<person>\n"
                  "<first>Nico</first>\n"
                  "<last>Josuttis</last>\n"
                  "</person>\n";
    regex reg("<(.*?)>(.*?)</(\\1)>");

    //iterate over all matches
    auto pos = data.cbegin();
    auto end = data.cend();
    smatch m;
    for(; regex_search(pos,end,m,reg); pos = m.suffix().first){
        cout << "match:  " << m.str() << endl;
        cout << " tag:    " << m.str(1) << endl;
        cout << " value:  " << m.str(2) << endl;
    }
    return 0;
}
```

处理Subexpression (次表达式)

```
int main(){
    string data = "<person>\n"
                  "<first>Nico</first>\n"
                  "<last>Josuttis</last>\n"
                  "</person>\n";
    regex reg("<(.*?)>(.*?)</(\\1)>");

    //iterate over all matches
    auto pos = data.cbegin();
    auto end = data.cend();
    smatch m;
    for(; regex_search(pos,end,m,reg); pos = m.suffix().first){
        cout << "match:  " << m.str() << endl;
        cout << " tag:    " << m.str(1) << endl;
        cout << " value:  " << m.str(2) << endl;
    }
    return 0;
}
```

Regex Iterator

- regex iterator 的作用
 - 逐一迭代正则表达式查找的所有匹配结果
- iterator 的类型
 - `regex_iterator<>`
- 针对 string 的 iterator 类型
 - `sregex_iterator`

Regex Iterator

```
int main(){
    string data = "<person>\n"
                  " <first>Nico</first>\n"
                  " <last>Josuttis</last>\n"
                  "</person>\n";

    regex reg("<(.*?)>(.*?)</(\\1)>");

    //iterate over all matches(using a regex_iterator)
    sregex_iterator pos(data.cbegin(), data.cend(), reg);
    sregex_iterator end;
    for( ; pos != end; ++pos){
        cout << "match:   " << pos->str() << endl;
        cout << " tag:    " << pos->str(1) << endl;
        cout << " value:  " << pos->str(2) << endl;
    }

    return 0;
}
```

Regex Iterator

```
int main(){
    string data = "<person>\n"
                  " <first>Nico</first>\n"
                  " <last>Josuttis</last>\n"
                  "</person>\n";

    regex reg("<(.*?)>(.*?)</(\\1)>");

    //iterate over all matches(using a regex_iterator)
    sregex_iterator pos(data.cbegin(), data.cend(), reg);
    sregex_iterator end;
    for( ; pos != end; ++pos){
        cout << "match:   " << pos->str() << endl;
        cout << " tag:    " << pos->str(1) << endl;
        cout << " value:  " << pos->str(2) << endl;
    }

    return 0;
}
```

Regex Iterator

```
int main(){
    string data = "<person>\n"
                  " <first>Nico</first>\n"
                  " <last>Josuttis</last>\n"
                  "</person>\n";

    regex reg("<(.*?)>(.*?)</(\\1)>");

    //iterate over all matches(using a regex_iterator)
    sregex_iterator pos(data.cbegin(), data.cend(), reg);
    sregex_iterator end;
    for( ; pos != end; ++pos){
        cout << "match:   " << pos->str() << endl;
        cout << " tag:    " << pos->str(1) << endl;
        cout << " value:  " << pos->str(2) << endl;
    }

    return 0;
}
```

Regex Iterator

```
int main(){
    string data = "<person>\n"
                  " <first>Nico</first>\n"
                  " <last>Josuttis</last>\n"
                  "</person>\n";

    regex reg("<(.*?)>(.*?)</(\\1)>");

    //iterate over all matches(using a regex_iterator)
    sregex_iterator pos(data.cbegin(), data.cend(), reg);
    sregex_iterator end;
    for( ; pos != end; ++pos){
        cout << "match:   " << pos->str() << endl;
        cout << " tag:    " << pos->str(1) << endl;
        cout << " value:  " << pos->str(2) << endl;
    }

    return 0;
}
```


Regex Iterator

```
int main(){
    string data = "<person>\n"
                  " <first>Nico</first>\n"
                  " <last>Josuttis</last>\n"
                  "</person>\n";
    regex reg("<(.*?)>(.*?)</(\\1)>");

    sregex_iterator end;
    //use a regex_iterator to process each matched substring
    //as element in an algorithm
    sregex_iterator pos(data.cbegin(), data.cend(), reg);
    for_each(pos, end, [](const smatch &m){
        cout << "match:  " << m.str() << endl;
        cout << " tag:    " << m.str(1) << endl;
        cout << " value:  " << m.str(2) << endl;
    });

    return 0;
}
```

Regex Token Iterator

- 作用
 - 处理匹配子序列之间的内容
 - 例如，把 string 拆分成若干 tokens(语汇单元)。
 - 分隔符可以是正则表达式
- Class regex_token_iterator<>
 - 针对 string 的具体实现 sregex_token_iterator

Regex Token Iterator

- 初始化方式
 - 字符序列的起点和终点
 - 正则表达式
 - 特殊含义的整数值
 - -1 匹配每一个“匹配的正则表达式之间”或“语汇切分器之间”的子序列
 - 0 匹配每一个匹配的正则表达式或语汇切分器
 - 任何其他数字n 匹配正则表达式中的第n个匹配次表达式

Regex Token Iterator

```
int main(){
    string data = "<person>\n"
                  " <first>Nico</first>\n"
                  " <last>Josuttis</last>\n"
                  "</person>\n";

    regex reg("<(.*?)>(.*?)</(\\1)>");

    sregex_token_iterator end;
    //iterator over all matches
    sregex_token_iterator pos(data.cbegin(), data.cend(), //sequence
                             reg,      // token separator
                             {0,2}); //0:full match, 2:second substring

    for( ; pos != end ; ++pos){
        cout << "match : " << pos->str() << endl;
    }

    return 0;
}
```

Regex Token Iterator

```
int main(){
    string data = "<person>\n"
                 " <first>Nico</first>\n"
                 " <last>Josuttis</last>\n"
                 "</person>\n";

    regex reg("<(.*?)>(.*?)</(\\1)>");

    sregex_token_iterator end;
    //iterator over all matches
    sregex_token_iterator pos(data.cbegin(), data.cend(), //sequence
                             reg, // token separator
                             {0,2}); //0:full match, 2:second substring

    for( ; pos != end ; ++pos){
        cout << "match : " << pos->str() << endl;
    }

    return 0;
}
```

Regex Token Iterator

```
int main(){

    string names = "nico, jim, helmut, paul, tim, john paul, rita";

    regex sep("[ \\t\\n]*[,;.][ \\t\\n]*"); //separated by , ; or .

    sregex_token_iterator p(names.cbegin(),names.cend(), //sequence
                           sep, //separator
                           -1); //-1: values between separators
    sregex_token_iterator e;

    for( ; p != e; ++p){
        cout << "name : " << *p << endl;
    }

    return 0;
}
```

Regex Token Iterator

```
int main(){

    string names = "nico, jim, helmut, paul, tim, john paul, rita";

    regex sep("[ \\t\\n]*[,;.][ \\t\\n]*"); //separated by , ; or .

    sregex_token_iterator p(names.cbegin(),names.cend(), //sequence
                           sep, //separator
                           -1); //-1: values between separators
    sregex_token_iterator e;

    for( ; p != e; ++p){
        cout << "name : " << *p << endl;
    }

    return 0;
}
```

用于替换的正则表达式

```
int main(){
    string data = "<person>\n"
                  " <first>Nico</first>\n"
                  " <last>Josuttis</last>\n"
                  "</person>\n";

    regex reg("<(.*?)>(.*?)</(\\1)>");

    //print data with replacement for matched patterns
    cout << regex_replace(data, // data
                           reg,   // regular expression
                           "<$1 value = \"$2\"/>") //replacement
          << endl;

    return 0;
}
```


用于替换的正则表达式

```
int main(){
    string data = "<person>\n"
                  " <first>Nico</first>\n"
                  " <last>Josuttis</last>\n"
                  "</person>\n";

    regex reg("<(.*?)>(.*?)</(\1)>");

    //print data with replacement for matched patterns
    cout << regex_replace(data, // data
                           reg,   // regular expression
                           "<$1 value = \"$2\"/>") //replacement
          << endl;

    return 0;
}
```

用于替换的正则表达式

默认的Pattern	说明
\$&	matched pattern
\$n	第 n 个 matched capture group
\$\$	字符 \$

Regex Flag

- Regex 常量
 - 影响 regex 接口的行为
 - `regex_constants::icase` 忽略大小写

```
regex reg("<(.*?)>(.*?)</(\\1)>", regex_constants::icase);
```

```
regex reg("<(.*?)>(.*?)</(\\1)>",  
         regex_constants::icase | regex_constants::egrep);
```

Regex ECMAScript语法

表达式	说明
.	Newline以外的任何字符
[...]	... 字符中的任何一个
[^...]	... 字符之外的任何一个
\n, \t, \v	newline、tabulator 或 vertical tab
\xhh, \uhhh	一个十六进制字符或Unicode字符
\d	一个数字
\D	一个非数字, 相当于 <code>[^[:digit:]]</code>
\s	一个空白字符
\S	一个非空白字符
\w	一个字母、数字或下划线
\W	不是字符、数字和下划线

Regex ECMAScript 文法

表达式	说明
*	前一个字符或群组出现任意次数
?	前一个字符或群组出现零次或一次（可有可无）
+	前一个字符或群组至少出现一次
{n}	前一个字符或群组出现 n 次
{n, }	前一个字符或群组至少出现 n 次
{n, m}	前一个字符或群组至少出现 n 次、至多 m 次。
... ...	在 之前或之后的 pattern
(...)	设定分组 (group)
\1, \2, \3, ...	第 n 个分组 (group) (分组的索引从 1 开始)
\b	字词的起点或终点
\B	字词的非起点或非终点
^	一行的起点 (包括所有字符的起点)
\$	一行的终点 (包括所有字符的终点)

Regex异常

- 当正则表达式被解析 (parsed) 的时候, C++标准库提供了一个用于处理正则表达式异常的 `exception class`。它派生自 `std::runtime_error`。此外, 它还提供一个额外成员 `code()` 用来产生一个差错代码 (error code)。这将有助于找出正则表达式处理过程中的出错环节。不过, `code()` 返回的 `error code` 取决于实现, 所以直接打印出它们没有什么帮助。

待续.....