# STL(string,vector)使用例子
# 更多使用例子

# 分割字符串

```
25 template <typename E>
26 void print_vector(const vector<E> & c) {
27     for (int i = 0; i < c.size(); ++ i) {
28         cout << c[i] << endl;
29     }
30 }
31
32 int main() {
33     string line = "what are we doing ?";
34     vector<string> words = split(line);
35     print_vector(words);
36 }
```

# 分割字符串

```
6  vector<string> split(const string & line,
7                       const string & delimiter=" ") {
8      vector<string> words;
9      int st = 0;
10     while (st < line.size()) {
11         int end = line.find(delimiter, st);
12         if (end == st) st += delimiter.size();
13         else {
14             if (end == -1) {
15                 words.push_back(line.substr(st));
16                 break;
17             }
18             words.push_back(line.substr(st, end-st));
19             st = end + delimiter.size();
20         }
21     }
22     return words;
23 }
```

# 函数式编程

- 只有函数调用，不需要分支和循环的编程方式
- map

应该改为 const vector <A> & x

```
38 template <typename A, typename R>
39 vector<R> map(vector<A> x, R (*map_func)(const A &)) {
40     vector<R> res;
41     for (int i = 0; i < x.size(); ++ i)
42         res.push_back(map_func(x[i]));
43     return res;
44 }
```

# 函数式编程

- map

```
46 #include <ctype.h>
47 string to_upper(const string & text) {
48     string res = text;
49     for (int i = 0; i < res.size(); ++ i) {
50         res[i] = toupper(res[i]);
51     }
52     return res;
53 }
54
55 int main() {
56     string line = "what are we doing ?";
57     vector<string> words = split(line);
58     words = map(words, to_upper);
59     print_vector(words);
60 }
```

# 函数式编程

- map

```
62 #include <sstream>
63 // using namespace std;
64 int to_int(const string & text) {
65     stringstream ss(text);
66     int val;
67     ss >> val;
68     return val;
69 }
70
71 int main() {
72     string line = "123 456 789";
73     vector<string> words = split(line);
74     vector<int> data = map(words, to_int);
75     print_vector(data);
76 }
```

# 函数式编程

- filter

```
78 template <typename E>
79 vector<E> filter(vector<E> x, bool (*filter_func)(const E &)) {
80     vector<E> res;
81     for (int i = 0; i < x.size(); ++ i)
82         if (filter_func(x[i]))
83             res.push_back(x[i]);
84     return res;
85 }
```

# 函数式编程

- filter

```
87 bool is_odd(const int & d) {
88     return d % 2 == 1;
89 }
90
91 int main() {
92     string line = "123 456 789";
93     vector<string> words = split(line);
94     vector<int> data = map(words, to_int);
95     data = filter(data, is_odd);
96     print_vector(data);
97 }
```

# 函数式编程

- reduce

换成另一个类型更好，例如E是学生，返回男生人数(int)

```
 99 template <typename E>
100 E reduce(vector<E> x, E (*reduce_func)(const E &, const E &)) {
101     if (x.size() == 0) return E();
102     E res = x[0];
103     for (int i = 1; i < x.size(); ++ i)
104         res = reduce_func(res, x[i]);
105     return res;
106 }
```

# 函数式编程

- reduce

```
108 string join(const string & str1, const string & str2) {
109     return str1 + " " + str2;
110 }
111
112 int add(const int & data1, const int & data2) {
113     return data1 + data2;
114 }
115
116 int main() {
117     string line = "123 456 789";
118     vector<string> words = split(line);
119     vector<int> data = map(words, to_int);
120     cout << reduce(words, join) << endl;
121     cout << reduce(data, add) << endl;
122 }
```

封装 Encapsulation

# 继承 Inheritance

多态性 Polymorphism

# Inheritance

- C++允许，在一个类的基础上创建一个新的类
  - 新的类将拥有基础类中的一切成员(但不一定能直接使用)
  - 以下方法能完成之前的作业吗？

```
#define String string

public String : public string
{
    ...
};
```

# Inheritance

- + split

```
126 template <typename A, typename R>
127 R convert(const A & a) {
128     return R(a);
129 }
```

```
131 class String : public string // inheritance
132 {
133 public:
134     String(const char * text = "") {
135         assign(text);
136     }
137     String(const string & text) {
138         assign(text);
139     }
140     vector<String> split(const String & delimiter=" ") {
141         vector<string> words = ::split(*this, delimiter);
142         return map(words, convert<string, String>);
143         // return map(words, String); ?
144     }
```

Inheritance

- + join

```
145     String join(const vector<String> & vec) {
146         if (vec.size() == 0) return String();
147         String res = vec[0];
148         for (int i = 1; i < vec.size(); ++ i)
149             res += *this + vec[i];
150         return res;
151     }
152 };
153
154 int main() {
155     // we can use String as if it was a string
156     String line("What");
157     line = "  what,  are, you ,doing , ,?   ";
158     vector<String> words = line.split(", ");
159     for (int i = 0; i < words.size(); ++ i)
160         cout << "\"" << words[i] << "\"" << endl;
161     cout << String(" ").join(words) << endl;
162 }
```

# Inheritance

- Concepts
  - Parent class 父类
    - Base class 基类
    - Super class 超类
  - Children class 子类
    - Derived class 派生类
    - Sub-class 子类

# Inheritance

```
168 class A1
169 {
170 public:
171     int value;
172
173     int getValue() {
174         return value;
175     }
176 };
```

```
178 class B1 : public A1
179 {
180 public:
181     int value2;
182
183     int getValue2() {
184         return value + value2; // value
185     }
186 };
187
188 int main() {
189     B1 obj;
190     obj.value = 10;
191     obj.value2 = 20;
192     cout << obj.getValue() << endl;
193     cout << obj.getValue2() << endl;
194 }
```

# Protected members

```
198 class A2
199 {
200 private:
201     int private_value;
202 protected:
203     int protected_value;
204     int get_private_value() {
205         return private_value;
206     }
207 public:
208     A2() {
209         private_value = 10;
210         protected_value = 20;
211     }
212     int get_protected_value() {
213         return private_value;
214     }
215 };
```

```
217 class B2 : public A2
218 {
219 public:
220     int get_sum() {
221         // this->private_value ?
222         return protected_value +
223             get_private_value();
224     }
225 };
226
227 int main() {
228     B2 obj;
229     // obj.get_private_value() ?
230     cout << obj.get_sum() << endl;
231 }
```

# Protected and private inheritance

```
235 class A3
236 {
237 private:
238     int private_value;
239 protected:
240     int protected_value;
241 public:
242     int public_value;
243 };
244
245 class B3 : protected A3
246 {
247 public:
248     B3() {
249         // private_value
250         protected_value = 1;
251         public_value = 2;
252     }
253 };
```

```
255 class C3 : public B3
256 {
257 public:
258     C3() {
259         // private_value
260         protected_value = 3;
261         public_value = 4;
262     }
263 };
264
265 int main() {
266     B3 obj;
267     // obj.private_value;
268     // obj.protected_value;
269     // obj.public_value;
270 }
```

# Protected and private inheritance

```
275 class A4
276 {
277 private:
278     int private_value;
279 protected:
280     int protected_value;
281 public:
282     int public_value;
283 };
284
285 class B4 : private A4
286 {
287 public:
288     B4() {
289         // private_value
290         protected_value = 1;
291         public_value = 2;
292     }
293 };
```

```
295 class C4 : public B4
296 {
297 public:
298     C4() {
299         // private_value
300         // protected_value = 3;
301         // public_value = 4;
302     }
303 };
304
305 int main() {
306     B3 obj;
307     // obj.private_value;
308     // obj.protected_value;
309     // obj.public_value;
310 }
```

# Code reusing

- Inheritance
- Composition

```
312 class A5
313 {
314 public:
315     int data1;
316     int data2;
317     void code1() {}
318     vode code2() {}
319 };
```

```
321 class B5_1 : private A5
322 {
323 public:
324     int get_data2() {
325         return data2;
326     }
327     void code2() {
328         A5::code2();
329     }
330 };
331
332 class B5_2
333 {
334     A5 a; // composition
335     int get_data2() {
336         return a.data2;
337     }
338     void code2() {
339         a.code2();
340     }
341 };
```