

# 第 8 章 编写泛型函数

刘 卉

huiliu@fudan.edu.cn



# 前言

- 从本章开始，学习如何编写抽象功能
- 泛型函数(generic function)
  - 参数类型直到调用时才能确定.
  - 展示标准库如何实现泛型函数.
  - 解释迭代器，并介绍5种不同的迭代器.
- 第9-13章，学习如何实现抽象的数据类型，以及面向对象编程



## 8.1 什么是泛型函数

`find(b, e, t)` // 库函数. 在任意种类容器中, 查找值为 `t` 的元素

- `b, e`: 迭代器, 支持特定操作.
- 事先并不知道参数/返回值的类型, 直到使用时才能确定.

### □ 如何描述 `find` 函数的行为方式?

- 函数使用参数的方式 → 约束参数的类型. e.g. `x + y`
- 编写 `find` 函数: 只能使用各种迭代器都支持的通用操作.
- 迭代器不是 C++ 特有的, 但它是标准库的基础, 也是使泛型函数有用的核心.



## 8.1.1 未知类型数据的中值

```
double median(vector<double> vec)
```

### □ 模板函数(template function)

- 实现泛型函数的语言设施.
- 不同类型的对象有相同的行为⇒为行为相似的一组函数编写一个单独的定义.
- 定义时并不知道模板参数的具体类型，使用时才能确定.
- 编译和链接程序时，模板参数的类型是明确的.
- 模板参数不同，则对应的函数不同.

□ 改写 `double median(vector<double> vec)`

```
template <class T> // 带1个模板(类型)参数T
T median(vector<T> v) // 带1个函数形参v
{
    // v的元素类型为T, 函数返回值的类型也为T
    typedef typename vector<T>::size_type vec_sz;
    vec_sz size = v.size();
    if (size == 0)
        throw domain_error("median of an empty vector");
    sort(v.begin(), v.end());
    vec_sz mid = size/2;
    return size % 2 == 0 ? (v[mid] + v[mid-1]) / 2 : v[mid];
}
```

□ 模板头: `template <class T>` 或 `template <typename T>`

■ 告诉系统这是一个模板函数, 带有一个类型参数.

■ 类型参数 vs 普通参数

`template <class T>` // T : 类型参数

`T median(vector<T> v)` // v : 普通参数

□ 都定义了可以用在函数生存空间的名字.

□ 类型参数表示类型, 普通参数表示变量.

■ 函数中所有出现T的地方, 系统都认为是一个类型名.

■ `median`函数的参数和返回值都使用了类型参数T

# 模板函数实例化

e.g. `vector<int> vi;`

.....

`median(vi);`     *// 模板函数实例化*

- 系统推断出T是int: 函数中所有出现T的地方都用int取代——实例化.

```
int median(vector<int> v)
{
    typedef vector<int>::size_type vec_sz;
    vec_sz size = v.size();
    .....
    return size%2==0 ? (v[mid]+v[mid-1])/2 : v[mid];
}
```

## □ 类型参数遍及模板的定义

### ■ 虽然很多类型依赖不明显

e.g. 包含v的任何操作都隐式包含了类型参数T

$$(v[mid] + v[mid-1]) / 2$$

为了知道 $v[mid]$ 和 $v[mid-1]$ 的类型，必须知道v的元素类型。  
该类型也决定了操作符 '+' 和 '/' 的类型。



## □ typename保留字

```
typedef typename vector<T>::size_type vec_sz;
```

- 使用依赖于模板参数类型(e.g. vector<T>)的成员, 且该成员也是类型(e.g. size\_type)时, 必须加上typename保留字.



## 8.1.2 模板实例化

### □ C++遵循传统的编辑-编译-链接模型

- 实例化发生在编译或链接期间.
- 直至模板实例化, 系统才会检验模板的代码是否可用于指定类型.

### □ 模板的定义必须是系统可访问的

- 通常在头文件中! 通常在头文件中! 通常在头文件中!





## 8.1.3 泛型函数和类型

□ 困难：理解模板和类型之间的交互作用

[例1] 传递给median函数的vector元素类型必须支持 '+' 和 '/' 。

[例2] `find(s.homework.begin(), s.homework.end(), 0);`  
homework是vector<double>类型，0是int类型。

[例3] `accumulate(v.begin(), v.end(), 0.0);`

第三个参数的类型为累加器的类型。本例中，累加器的类型为double。

[例4] `string::size_type maxlen = 0;`

`maxlen = max(maxlen, name.size());`

- `maxlen`的类型必须与`name.size()`返回的类型精确匹配.

```
template<class T>
```

```
T max(const T& left, const T& right)
```

```
{ return left > right ? left : right; }
```

- `left`和`right`必须是同一类型;
- 如果把一个`int`值和一个`double`值传递给`max`函数→系统无法确定模板函数中`T`的类型→无法实例化.

```
[例]  int i;  
      double d;
```

```
.....
```

```
    cout << max(i, d);
```

[Error] no matching function for call to 'max(int&, double&).'

[Note] template argument deduction/substitution failed: deduced conflicting types for parameter 'const \_Tp' ('int' and 'double')



## 8.2 算法独立于数据结构

```
find(c.begin(), c.end(), val)
```

□ why not `c.find(val)`?

- 问题1. 用到`find`函数的所有类型，都必须定义一个成员函数`find`.
- 问题2. 不能在内置数组上使用`find`函数.

# 算法独立于数据结构

```
find(c.begin(), c.end(), val)
```

□ why not `find(c, val)`?

- 原因1：传递两个值可限定一个区间→查找容器的一部分，而不需要查找整个容器。
- 原因2：使用迭代器作参数，可访问特殊的、不在容器中的元素，e.g. `end`函数。



## 8.2.1 算法和迭代器

### □ 理解模板的直接方法——标准库函数

- 均包含迭代器参数，而所有容器和string都提供迭代器→库函数可操作这些容器的元素.
- 某些容器提供的操作，其它容器不支持.
  - e.g. vector支持索引，list不支持→vector的迭代器支持 '+' 操作(`iter+i`)，list不支持.
- 所有容器迭代器的共同操作，有相同的名字
  - e.g. 使用++得到下一个元素的迭代器.



- 并非所有算法都需要复杂的迭代器操作
  - find函数仅使用很少的迭代器操作.
  - sort函数则使用了大部分迭代器操作→只有vector和string支持sort函数.
- 标准库定义了5种迭代器
  - 每种都对应一个特定的迭代器操作集.
  - 所有容器都包含某种类型的迭代器.
  - 每个标准库算法都需要某种类型的迭代器作参数.
  - 迭代器类型决定了哪些容器可以用哪些算法.

## □ 每种迭代器对应一种访问容器的策略

- 对应某些算法.

e.g. 某些算法只需对输入访问一次→不需要能进行多次访问的迭代器.

e.g. 某些算法要求根据索引随机访问元素→需要能把索引和整数相加的迭代器.

- 描述每种策略, 以及使用该策略的算法.



## 8.2.2 顺序只读访问

### □ 标准库的find函数—<algorithm>

```
template <class In, class X> // 有2个类型参数
In find(In begin, In end, const X& x)
{
    while (begin != end && *begin != x)
        ++begin;
    return begin;
}
```

- 用到的迭代器操作：前缀++，!=，\*。



# 另一个版本的find函数

```
template <class In, class X>
In find(In begin, In end, const X& x)
{
    if (begin == end || *begin == x)
        return begin;
    begin++;
    return find(begin, end, x); // 使用递归方式进行查找
}
```

- 用到的迭代器操作：后缀++，==，\*。



# 输入迭代器

- 支持++(前缀和后缀), ==, !=, \*, ->操作.
- 每种容器的迭代器都支持这些操作⇒find函数可在各种容器上使用.



## 8.2.3 顺序只写访问

### □ 标准库的copy函数

```
template<class In, class Out> // 2个类型参数
Out copy(In begin, In end, Out dest)
{
    输入迭代器      输出迭代器
    while (begin != end)
        *dest++ = *begin++;
    return dest;
}
```

- 类型Out必须支持++(前缀/后缀), \*, =.



# 输出迭代器

## □ 隐式需求——顺序写入

`*it = x; ++it; ++it; *it = y; ?`

- 在对`*it`赋值之间，执行`++it`不能超过1次；也不能对`*it`多次赋值。

## □ 输出迭代器

- 所有标准库容器都提供⇒都能使用`copy`函数。



## 8.2.4 顺序读写访问

### □ replace函数

```
template<class For, class X>
void replace(For beg, For end, const X& x, const X& y)
{
    while (beg != end) {
        if (*beg == x)
            *beg = y;
        ++beg;
    }
}
```

支持输入/输出迭代器的所有操作

满足条件才写入。





# 前向迭代器

- 支持++, ==, !=, \*, ->, = (不必"顺序写入").
- 所有标准库容器都满足这些要求⇒所有标准库容器都能使用replace函数.



## 8.2.5 双向访问

### □ reverse函数

```
template<class Bi> // 1个类型参数
void reverse(Bi begin, Bi end)
{
    while (begin != end) {
        --end; // 从后往前移动
        if (begin != end)
            swap(*begin++, *end);
    }
    // 库算法 从前往后移动
}
```



# 双向迭代器

- 除了满足所有前向迭代器的要求，还支持--(前缀/后缀).
- 标准库容器都支持双向迭代器⇒都能使用reverse函数.



## 8.2.6 随机访问

### □ binary\_search函数

```
template<class Ran, class X>
bool binary_search(Ran begin, Ran end, const X& x)
{
    while (begin < end) {
        Ran mid = begin + (end - begin) / 2;
        if (x < *mid)    end = mid;
        else if (*mid < x) begin = mid + 1;
        else            return true;
    }
    return false;
}
```

Why not  $\text{Ran mid} = (\text{begin} + \text{end}) / 2$  ?

可对迭代器进行算术运算



# 随机访问迭代器

- 除了满足双向迭代器的所有要求外，还需：

$p+n$ ,  $p-n$ ,  $n+p$  //  $p, q$ : 迭代器,  $n$ : 整数

$p-q$

不支持  $p+q$

$p[n]$

$p < q$ ,  $p > q$ ,  $p \leq q$ ,  $p \geq q$

- $\text{vector}$  和  $\text{string}$  支持随机访问迭代器。



## 8.2.7 迭代器区间和越界值

为什么要用末尾元素下一个位置的迭代器？

- 1) 如果是空区间→没有迭代器能标记末尾元素.
  - 以相同方式处理空区间和非空区间，简化程序.
- 2) 只需比较迭代器的相等或不等，无需定义 '<' 或 '>' .
  - 通过比较两个迭代器，可立即知道该区间是否为空.
- 3) 一种自然的方式表示"区间之外".

## 迭代器区间和越界值

- 很多标准库函数以及我们自己编写的函数，通过返回区间的第二个迭代器表示失败。  
e.g. 库函数 `find(b, e, t)` 使用这个惯例表示查找失败。
- 使用半开区间 `[b, e)` 使得程序更为简单和可靠。
- 为了表示区间的末尾，每种容器都为它的迭代器提供一个越界值 (`end` 成员函数)。
- 对越界迭代器的引用：未定义。



## 8.3 输入流/输出流迭代器

### □ 并非所有迭代器都与容器相关

e.g. 与输入/输出流绑定的迭代器，与容器无关.

- 使用流迭代器控制istream/ostream.

### □ 输入流迭代器——istream\_iterator


- 一种输入迭代器.
- <iterator>
- 与vector一样，是模板类.



# 输入流迭代器——istream\_iterator

[例] 从标准输入中读取若干整数存于vector对象

```
// read ints from the standard input and append them to v  
vector<int> v;
```



```
int i;  
while (cin >> i)  
    v.push_back(i);
```

```
//批量输入：把标准输入复制到v中  
copy(istream_iterator<int>(cin), istream_iterator<int>(),  
    back_inserter(v));
```

```
template<class In, class Out>  
Out copy(In begin, In end, Out dest)
```

# 输入流迭代器——istream\_iterator

## □ `istream_iterator<int>(cin)`

- 构造一个输入流迭代器，读取int类型值，并绑定在cin上。

## □ `istream_iterator<int>()`

- 构造了一个默认的流迭代器，一旦遇到文件结束标志或无效输入，就与该值相等。
- 用该默认值为copy函数表示"末尾元素的下一个位置"。



# 迭代器适配器

```
vector<int> v;  
copy(istream_iterator<int>(cin), istream_iterator<int>(),  
      back_inserter(v));
```

## □ 返回输出迭代器的函数

e.g. back\_inserter(v)

- 为容器v生成一个迭代器，可为v添加元素.
- v必须支持push\_back操作，e.g. list, vector, string.



# 输出流迭代器——ostream\_iterator

## □ 一种输出迭代器

[例] 把vector对象中保存的整数输出到屏幕上

```
for (vector<int>::const_iterator iter = v.begin();  
    iter != v.end(); ++iter)  
    cout << *iter << '\t';
```

■ 批量输出：把v复制到标准输出中。

```
// write the elements of v each separated from the other by a space  
copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
```

分隔符

# 输出流迭代器——ostream\_iterator

- `ostream_iterator<int>(cout, " ")`
  - 创建了一个输出流迭代器，输出int类型值，并绑定到cout.
  - 若不指定分隔符，输出时所有值将连在一起.

// no separation between elements!

```
copy(v.begin(), v.end(), ostream_iterator<int>(cout));
```

## 8.4 使用迭代器提高灵活性

□ 重写split函数 `vector<string> split(const string& s)`

- 将找到的单词保存在支持输出迭代器的任意容器中，而不仅仅是vector.
- 改写为模板函数，加入一个输出迭代器类型参数→提高函数的灵活性. `template <class Out>`  
`void split(const string& str, Out os)`
- 除了输入迭代器，前向/双向/随机访问迭代器都满足输出迭代器的要求→可把split函数用于任意种类容器和输出流.

```
split(s, back_inserter(word_list));    // list容器
```

```
split(s, ostream_iterator<string>(cout, "\n")); // 输出流
```

# 小 结

## □ 模板函数

typename

类型名, 在函数定义中使用

```
template<class type-parameter [, class type-parameter]... >  
ret-type function-name (parameter-list)
```

- 定义在头文件中.
- 模板函数中使用依赖类型参数所定义的类型时, 必须用 typename 关键字加以限定.

```
typename T::size_type name;
```

- 实例化: 调用模板函数时, 对于特定类型, 系统会自动创建一个模板函数的实例.

## □ 迭代器

- 通过把迭代器用作算法和容器之间的粘合剂，算法获得了数据结构的独立性。
- 算法使用的迭代器都要求支持某些操作→把容器和能够在这个容器上使用的算法匹配起来。
  - 1) 输入迭代器：按一个方向顺序访问，只能输入。
  - 2) 输出迭代器：按一个方向顺序访问，只能输出。
  - 3) 前向迭代器：按一个方向顺序访问，输入/输出。
  - 4) 双向迭代器：双向顺序访问，输入/输出。
  - 5) 随机访问迭代器：高效访问任意元素，输入/输出。