

The libcstl Library User Guide



The libcstl Library User Guide

for libcstl 2.0

Wangbo

2010-04-22

This file documents the libcstl library.

This is edition 1.0, last updated 2010-04-22, of *The libcstl Library User Guide* for libcstl 2.0.

Copyright (C) 2008, 2009, 2010 Wangbo <activesys.wb@gmail.com>

目录

第一章简介.....	7
第一节关于这本指南	7
第二节关于 libcstl.....	8
第三节 libcstl 的编译和安装.....	10
1.编译和安装.....	10
2.特殊的编译选项.....	10
第二章 libcstl 库的基本概念.....	11
第一节 libcstl 的组成部分.....	11
第二节容器.....	12
1.序列容器.....	14
2.关联容器.....	19
3.容器适配器.....	20
第三节迭代器.....	20
1.使用关联容器的例子.....	23
2.迭代器种类.....	29
第四节算法和函数.....	30
1.数据区间.....	32
2.处理多个数据区间.....	34
3.质变算法.....	36
4.质变算法和关联容器.....	40
5.算法和容器操作函数.....	42
6.用户自定义规则.....	43
7.自定义规则，函数和谓词.....	46
8.libcstl 函数.....	51
第五节 libcstl 容器的使用过程.....	52
1.libcstl 容器的使用过程.....	52
2.数据类型的使用.....	53
第三章 libcstl 工具类型.....	54
第一节 bool_t.....	54
第二节 pair_t.....	54
1.pair_t 的使用过程.....	54
2.pair_t 的主要操作.....	55
3.pair_t 的应用实例	55
第三节 range_t.....	59
第四章 libcstl 容器.....	59
第一节容器的特点和共同的操作.....	59
1.容器的共同特点.....	60
2.容器的共同操作.....	60
第二节 vector_t.....	61
1.vector_t 的能力.....	61
2.vector_t 的操作.....	61

3.将 vector_t 作为数组使用.....	64
4.vector_t 的使用实例.....	65
第三节 deque_t.....	67
1.deque_t 的能力.....	67
2.deque_t 的操作.....	67
3.deque_t 的应用实例.....	69
第四节 list_t.....	70
1.list_t 的能力.....	71
2.list_t 的操作.....	71
3.list_t 的使用实例.....	74
第五节 slist_t.....	76
1.slist_t 的能力.....	76
2.slist_t 的操作.....	76
3.slist_t 使用实例.....	78
第六节 set_t 和 multiset_t.....	81
1.set_t 和 multiset_t 的能力.....	81
2.set_t 和 multiset_t 操作.....	82
3.set_t 和 multiset_t 的使用实例.....	85
第七节 map_t 和 multimap_t.....	90
1.map_t 和 multimap_t 的能力.....	91
2.map_t 和 multimap_t 的操作.....	91
3.将 map_t 作为关联数组使用.....	93
4.map_t 和 multimap_t 的使用实例.....	93
第八节 hash_set_t, hash_multiset_t, hash_map_t, hash_multimap_t.....	100
1.基于哈希结构的关联容器的能力.....	101
2.基于哈希结构的关联容器的操作.....	101
3.将 hash_map_t 作为关联数组使用.....	103
4.基于哈希结构的关联容器的使用实例.....	103
第九节怎样选择容器类型.....	107
第五章 libcstl 迭代器.....	110
第一节迭代器的种类.....	111
1.input_iterator_t.....	113
2.output_iterator_t.....	113
3.forward_iterator_t.....	113
4.bidirectional_iterator_t.....	114
5.random_access_iterator_t.....	114
第二节迭代器的辅助操作.....	117
1.使用 iterator_advance()移动迭代器.....	117
2.使用 iterator_distance()计算迭代器之间的距离.....	119
第六章 libcstl 函数.....	120
第一节函数的作用.....	120
1.作为谓词.....	120
2.作为行为规则.....	123
第二节函数的类型.....	124

1.一元函数.....	124
2.二元函数.....	124
第三节预定义的函数.....	125
1.算术运算函数.....	125
2.关系运算函数.....	125
3.逻辑运算函数.....	125
4.其他函数.....	126
第七章 libcstl 算法.....	126
第一节算法的分类.....	126
第二节非质变算法.....	126
1.查找单一数据.....	128
2.数据区间匹配.....	136
第三节质变算法.....	146
1.拷贝数据.....	147
2.交换数据.....	154
3.数据变换.....	156
4.数据替换.....	159
5.数据填充.....	161
6.移除数据.....	163
7.逆序.....	166
8.旋转数据.....	168
9.随机重排.....	169
10.划分.....	172
第四节排序算法.....	174
1.排序.....	175
2.二分查找.....	188
3.合并.....	191
4.与集合有关的算法.....	193
5.堆算法.....	197
6.最大数据和最小数据.....	199
7.按词典顺序比较.....	202
8.排列组合.....	206
第五节算术算法.....	208
1.依次赋值.....	208
2.求和算法.....	209
3.计算内积.....	211
4.局部总和.....	212
5.相邻数据的差.....	213
6.幂运算.....	215
第八章 libcstl 容器适配器.....	216
第一节 stack_t.....	216
1.stack_t 的操作.....	217
2.stack_t 的使用实例.....	217
第二节 queue_t.....	218

1.queue_t 的操作.....	219
2.queue_t 的使用实例.....	220
第三节 priority_queue_t.....	221
1.priority_queue_t 的操作.....	221
2.priority_queue_t 的使用实例.....	222
第九章 libcstl 字符串.....	223
第一节两个 string_t 的使用实例.....	223
1.第一个例子:提取文件名模版.....	223
2.第二个例子:提取单词并逆序打印.....	226
第二节 string_t 类型的描述.....	228
1.strint_t 的能力.....	228
2.string_t 的操作概览.....	228
3.初始化和销毁.....	229
4.string_t 和 C 字符串.....	229
5.string_t 的大小和容量.....	230
6.string_t 数据访问.....	230
7.string_t 比较操作.....	230
8.赋值.....	231
9.数据交换.....	232
10.添加和连接.....	232
11.插入数据.....	233
12.删除数据.....	233
13.替换.....	233
14.子串.....	234
15.输入输出.....	234
16.查找.....	235
17.NPOS 值.....	236
18.迭代器支持.....	236

第一章 简介

第一节 关于这本指南

这本指南是为介绍 `libcstl` 库的使用方法和使用技巧编写的，全面描述函数接口以及数据结构的书籍，如果想要查询函数接口或者数据结构请参考《The `libcstl` Library Reference Manual》。这本指南只针对 `libcstl` 的 2.0 版本，如果想要了解其他版本请参考相应的用户指南或者参考手册。

在阅读这本指南之前您只需要了解基本的 C 语言编程，了解库的使用方法就可以了。下面简单的介绍了这本书的各个章节的内容：

- 第一章：简介
简单介绍本书的结构和相关的内容，简单介绍 `libcstl` 库。
- 第二章：`libcstl` 库的基本概念
介绍使用 `libcstl` 库需要知道的基本概念，库的组成部分，包含的头文件，要使用到的数据机构等。
- 第三章：工具类型
这一章主要介绍 `libcstl` 提供的小的工具类型，一些辅助的数据结构和宏定义。
- 第四章：容器
这一章解释容器的概念，介绍各个容器的功能以及对各个容器的能力进行对比。
- 第五章：迭代器
详细讨论迭代器的概念，功能，用途以及迭代器的一些辅助函数。
- 第六章：算法
列举了讨论了算法的应用，详细的讨论了算法的适用范围，对各种算法进行了比较。
- 第七章：函数
讨论 `libcstl` 库的一元函数，二元函数以及谓词的用法。
- 第八章：容器适配器
详细描述了特殊的容器 - 容器适配器，介绍了适配器的特点和用法。
- 第九章：字符串
描述了 `libcstl` 提供的字符串类型，它不同于传统的 C 字符串。
- 第十章：类型机制
描述了 `libcstl` 对于用户自定义类型的处理，以及类型的注册和管理机制。
- 附录：对于直接以数据为参数的函数的使用
讨论了一种特殊的接口函数的使用方法和技巧。

本书中使用浅黄色背景表示命令行输入或者输出，例如：

```
[wb@ActiveSys ~]$ tar zxvf libcstl-2.0.0.tar.gz
...
[wb@ActiveSys ~]$ cd libcstl-2.0.0
```

使用蓝灰色表示代码，例如：

```

#include <stdio.h>
#include <cstl/cvector.h>

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);

    if(pvec_v1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);

    vector_push_back(pvec_v1, 1);
    printf("Vector length is %d.\n", vector_size(pvec_v1));

    vector_push_back(pvec_v1, 2);
    printf("Vector length is now %d.\n", vector_size(pvec_v1));

    vector_destroy(pvec_v1);

    return 0;
}

```

第二节 关于 libcstl

libcstl 是使用 ANSI C 编写的通用的数据结构和常用算法的库，它模仿 STL 的接口形式，包括序列容器, 关联容器, 容器适配器, 迭代器, 函数, 算法等. libcstl 为 C 编程中的数据管理提供了方便易用的程序库。

libcstl 分为容器，迭代器，函数和算法四部分，此外 2.0 版本还添加了类型机制，这是一种为用户提供了方便使用自定义类型的机制。

容器一种用于保存数据的类型，按照功能分为序列容器，关联容器和容器适配器。序列容器是按照数据插入的顺序保存数据，关联容器中保存的数据是根据某种规则排序的，容器适配器是在容器的基础上对容器进行封装从而实现特定的功能，容器适配器不支持迭代器操作，因此适配器也不能够用于算法操作。

迭代器表现的是一种指针的语义，它是对位置操作的一种类型，但是迭代器是通用的，通过迭代器可以实现对任何容器的位置操作，同时它也是容器和算法的桥梁，算法通过迭代器对容器中的数据进行操作。

算法是通用的，它通过迭代器来操作数据区间中的数据，这样就可以对任何符合要求的容器以及数据区间应用算法。正式因为通用的关系，相同功能的算法和容器本身的操作函数，后者更高效。

函数以及谓词是规范算法行为的，可以使用特定的函数或者算法来改变算法的行为，带有_if 后缀的算法都要求使用函数或者谓词。

字符串是一种特殊的容器，它只保存字符类型，同时也支持许多针对字符串特有的操作。

类型机制是 2.0 添加的新功能，它为用户使用自定义类型提供了便利，可以让用户像使用基本类型一样使用自定义类型。

libcstl 2.0 与 1.0 相比有了很大的改进，下面列出了不同点：

功能		1.0	2.0	说明
容器	deque_t	支持	支持	
	list_t	支持	支持	
	vector_t	支持	支持	
	slist_t	支持	支持	
	set_t	支持	支持	
	multiset_t	支持	支持	
	map_t	支持	支持	更新了默认的数据比较规则。
	multimap_t	支持	支持	更新了默认的数据比较规则。
	hash_set_t	支持	支持	更新了默认的哈希函数。
	hash_multiset_t	支持	支持	更新了默认的哈希函数。
	hash_map_t	支持	支持	更新的默认的哈希函数和默认的数据比较规则。
	hash_multimap_t	支持	支持	更新的默认的哈希函数和默认的数据比较规则。
	priority_queue_t	支持	支持	
	queue_t	支持	支持	
	stack_t	支持	支持	
迭代器	iterator_t	支持	支持	
	range_t		支持	一种表示数据范围的类型。
算法	数值算法	支持	支持	
	通用算法	支持	支持	
函数	针对基本类型的函数	支持	支持	
	针对 libcstl 内部类型的函数		支持	增加了针对容器以及工具类型的函数和谓词。
字符串	string_t	支持	支持	
工具类型	pair_t	支持	支持	更新了默认的数据比较规则。
	bool_t	支持	支持	
类型机制	支持 c style 字符串		支持	增加了对于 c style 字符串类型的支持。
	支持用户自定义类型	部分支持	支持	通过类型注册机制完善了对用户自定义类型的支持。
	类型注册		支持	增加了类型注册和类型复制功能。

跨平台	支持 Linux	支持	支持	
	支持 Windows		支持	添加了 VS2005 和 VS2008 的编译工程。

第三节 libcstl 的编译和安装

1. 编译和安装

首先需要在 libcstl 的主页中下载源代码压缩包 libcstl-2.0.0.tar.gz，解压后进入目录 libcstl-2.0.0:

```
[wb@ActiveSys ~]$ tar zxvf libcstl-2.0.0.tar.gz
...
[wb@ActiveSys ~]$ cd libcstl-2.0.0
```

然后执行 make 编译 libcstl:

```
[wb@ActiveSys libcstl-2.0.0]$ make
```

接下来安装 libcstl 库(需要 root 权限):

```
[wb@ActiveSys libcstl-2.0.0]# make install
```

这样编译并安装了 libcstl 库.

卸载 libcstl 库同样需要 root 权限:

```
[wb@ActiveSys libcstl-2.0.0]# make uninstall
```

如果以前安装过 libcstl 库, 请使用更新更新命令:

```
[wb@ActiveSys libcstl-2.0.0]# make update
```

上面介绍的是 Linux 下的编译安装方法, libcstl-2.0 也提供了 Windows 下的编译工程, 在解压后的目录中有目录 build-win, 其中两个目录 vc8 和 vc9 分别对应 VS2005 和 VS2008 的工程文件。

2. 特殊的编译选项

在 libcstl-2.0.0 目录下有一个 Makefile 文件, 文件中定义了一个有关编译选项的变量:

```
CFLAGS_EX = -DNDEBUG
```

其中 NDEBUG 都是 libcstl 的默认编译选项, 如果想要修改默认编译选项就请修改 CFLAGS_EX 变量: 想要添加编译选项请在这一行的后面添加 -DXXXX (其中 XXXX 是要添加的编译选项), 如果要删除就直接删除 -DXXXX 选项.

libcstl 编译过程中的可用编译选项有:

- **NDEBUG** 控制断言.

定义这个编译选项可以去掉库中的断言. 当库函数接受到非法参数时, 断言就会报错, 但是有断言的版本执行效率

低(非断言版本效率大约是有断言版本的 20~40 倍)。

- **CSTL_STACK_VECTOR_SEQUENCE** 以 `vector_t` 容器为 `stack_t` 适配器的底层实现。
 - **CSTL_STACK_LIST_SEQUENCE** 以 `list_t` 容器为 `stack_t` 适配器的底层实现。
- 这两个选项都是控制 `stack_t` 适配器的底层实现的。如果不指定则使用 `deque_t` 作为 `stack_t` 的底层实现。

- **CSTL_QUEUE_LIST_SEQUENCE** 以 `list_t` 容器为 `queue_t` 适配器的底层实现。
- 这个选项控制 `queue_t` 的底层实现。如果不指定则使用 `deque_t` 为 `queue_t` 的底层实现。

- **CSTL_SET_AVL_TREE** 以 avl 树作为 `set_t` 的底层实现。
- **CSTL_MULTISSET_AVL_TREE** 以 avl 树作为 `multiset_t` 的底层实现。
- **CSTL_MAP_AVL_TREE** 以 avl 树作为 `map_t` 的底层实现。
- **CSTL_MULTIMAP_AVL_TREE** 以 avl 树作为 `multimap_t` 的底层实现。

以上四个选项是控制关联容器的底层实现的, 如果不指定则对应的关联容器使用红黑树作为底层实现(红黑树比 avl 树快 40%)。默认使用红黑树作为底层实现。

第二章 libcstl 库的基本概念

数据结构和算法是编程的精髓, 有很多数据结构和算法是为我们程序员所熟知的, 也是在平常编程的过程中经常使用到的, 比如链表, 队列, 平衡二叉树等这样的数据结构, 再比如搜索, 排序, 二分查找等这样的算法, `libcstl` 将这些常用的数据结构和算法都封装在一起变成一个通用的库, 这样就不用每次使用的时候都要从头再来重新实现了, 按照编程的需要, 在 `libcstl` 中选择合适的类型和算法就能够满足你的要求。`libcstl` 提供了丰富的数据结构类型和算法。

第一节 libcstl 的组成部分

`libcstl` 由很多部分组成, 其中最主要的有容器, 迭代器, 算法以及函数。

- 容器

容器主要用于保存和管理数据, 它是对数据结构的封装。不同的容器各有特点, 有的保存简单的数据, 有的保存键/值对, 有的可以对数据进行随机访问, 有的是无序的有的排序的, 按照不同的用途选择相应的容器。

- 迭代器

迭代器实现的是位置的语义, 它的作用是对所有容器中的数据位置提供了统一的使用方式, 这样通过迭代器算法就可以操作任何容器中的数据而且不必理会容器的内部结构, 这样就实现了算法和数据的分离。

- 算法和函数

算法就是对如搜索, 排序, 查找等等这样的算法的封装, 通常我们要实现一个算法是要知道数据保存的结构, 但是这里的算法不必知道具体结构, 它通过迭代器和数据区间的概念就可以作用于容器中的数据。函数的作用主要是为算法提供操作准则, 用来改变算法的行为。

libcstl 通过迭代器将数据和算法分离，任何算法都可以和任何容器交互作用。

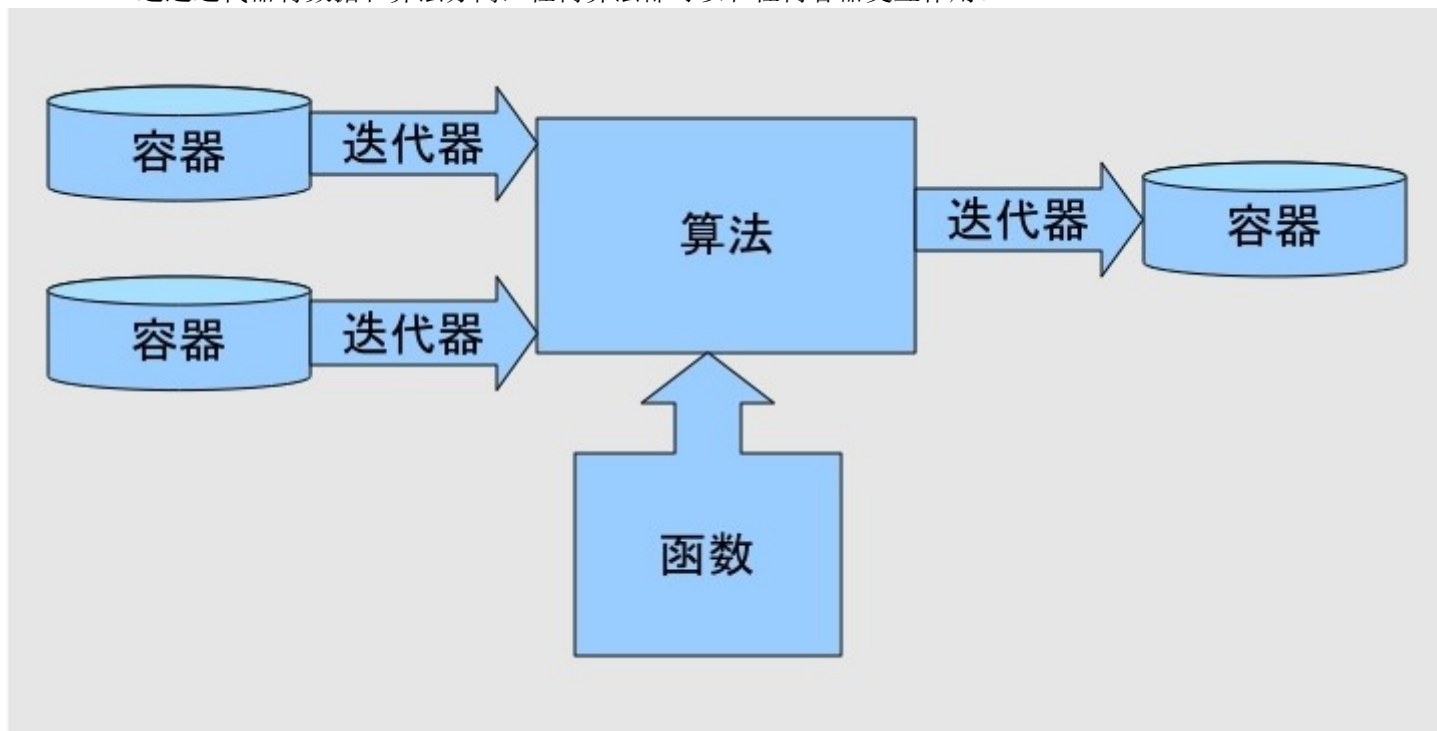


图 2.1 libcstl 组件之间的关系

除了上面基本主要的组成部分，libcstl 还提供了容器适配器，工具类型，字符串和类型机制。

第二节 容器

容器是用来保存和管理数据的，为了满足不同的需要 libcstl 提供了多种容器：

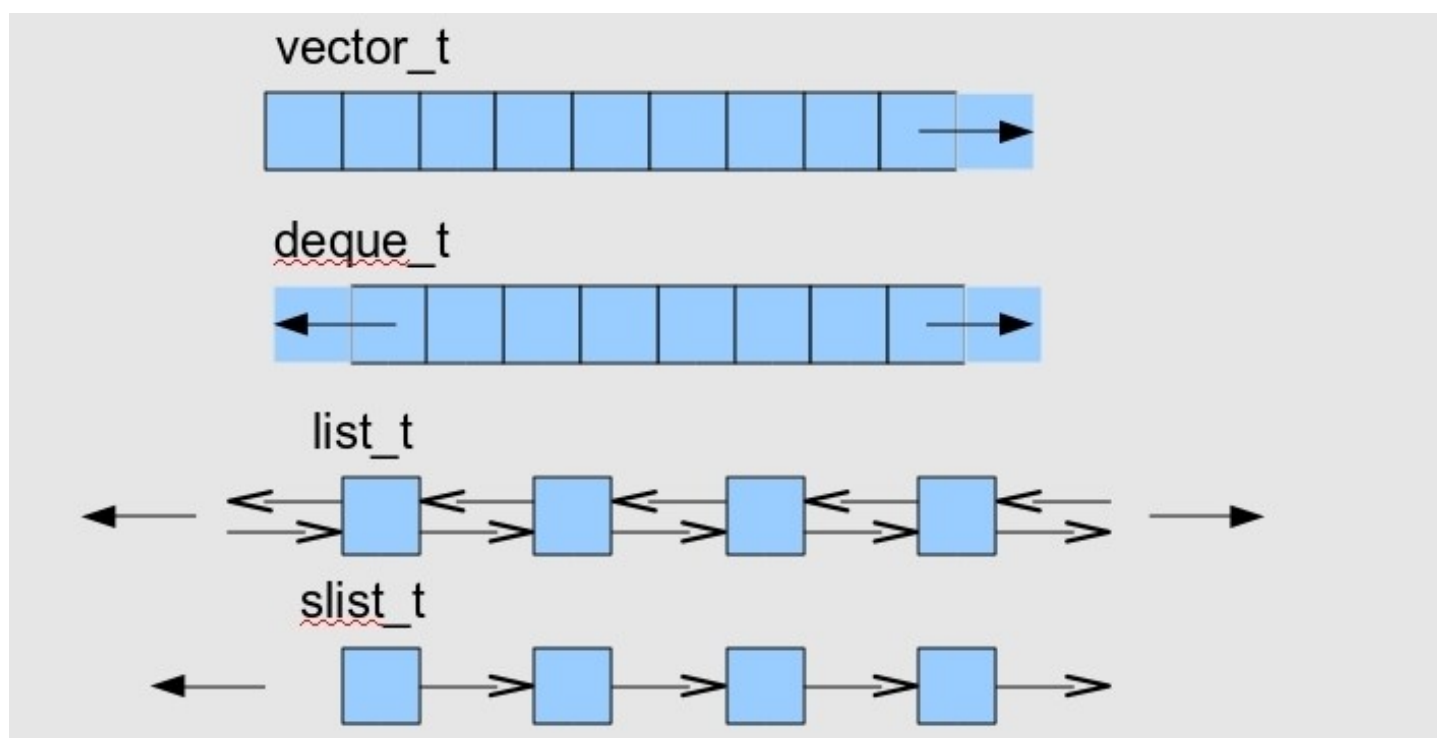


图 2.2 序列容器

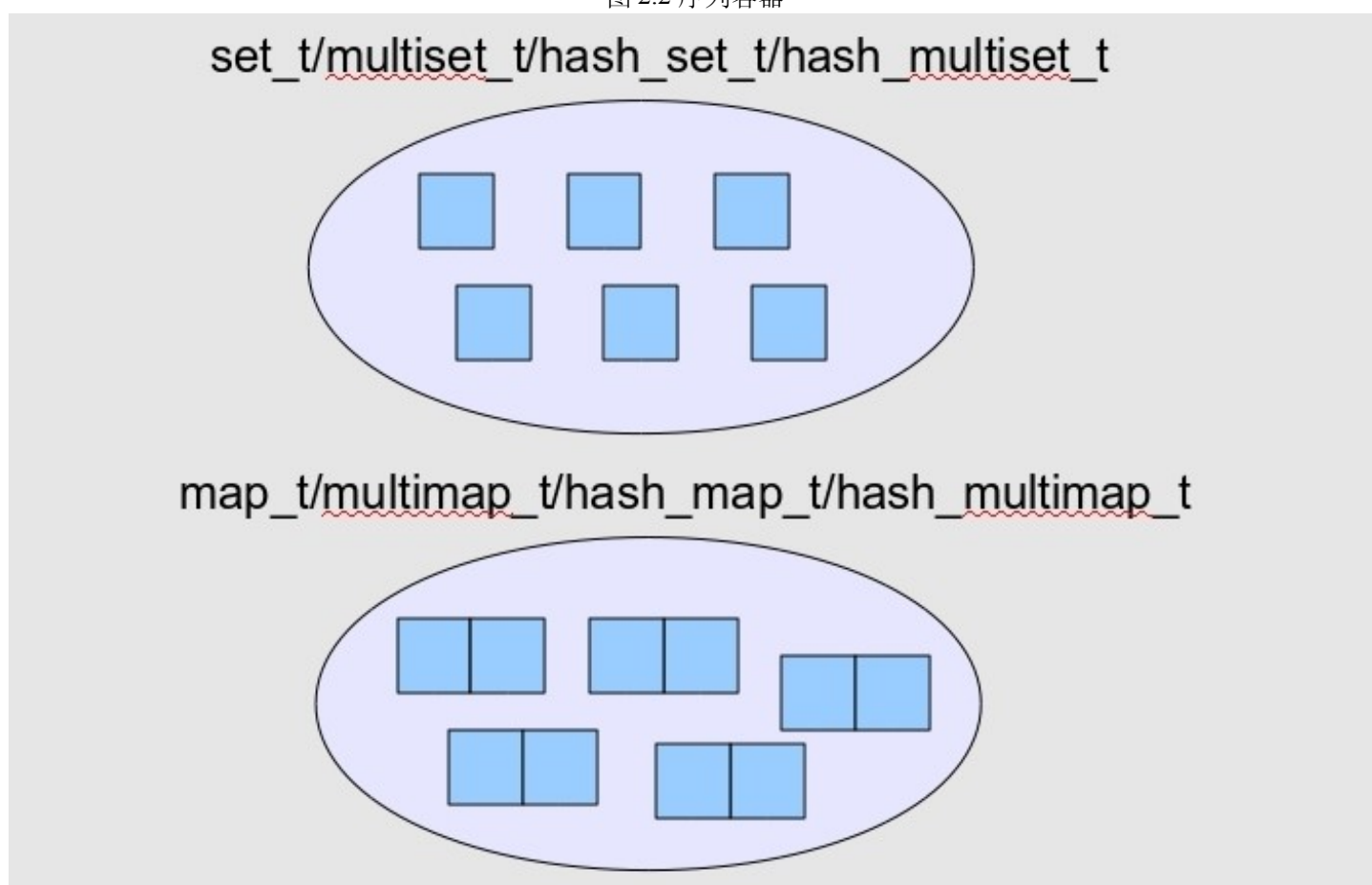


图 2.3 关联容器

总的来说容器分为两大类，序列容器和关联容器：

- 序列容器

序列容器中的数据的顺序与数据插入到容器中的次序有关，而与数据本身的值无关。libcstl 提供的序列容器有：vector_t, list_t, deque_t, slist_t.

- 关联容器

关联容器中数据是有序的，容器中数据的顺序取决于特定的排序规则而与数据插入到容器中的次序无关。libcstl 提供的关联容器有：set_t, map_t, multiset_t, multimap_t, hash_set_t, hash_map_t, hash_multiset_t, hash_multimap_t.

数据是否排序并不是两种容器的主要目的，目的在于对容器中的数据的访问，序列容器中的数据可以在指定的位置快速的插入或者删除，也可以快速的访问指定位置的数据。关联数据中的数据是有序的只可以更快速的查找数据。

1. 序列容器

libcstl 提供四种序列容器：vector_t, list_t, deque_t, slist_t.

- vector_t

vector_t 的行为类似于数组，但是它可以根据需要动态的生长，这样就可以使用下标来随即的访问 vector_t 中的数据。在 vector_t 的末尾插入或者删除数据是非常高效的，但是在开头或者中间插入或者删除数据效率就会很低，因为 vector_t 要移动后面的数据。

下面的例子定义了一个保存整型数据的 vector_t，然后向其中插入 6 个数据并打印出来：

```
/*
 * vector1.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>

int main(int argc, char* argv[])
{
    /* vector container for integer elements */
    vector_t* pvec_coll = create_vector(int);
    int i = 0;

    if(pvec_coll == NULL)
    {
        return -1;
    }

    vector_init(pvec_coll);

    /* append elements with 1 to 6 */
```

```

    for(i = 1; i <= 6; ++i)
    {
        vector_push_back(pvec_coll, i);
    }

    /* print all elements followed by space */
    for(i = 0; i < vector_size(pvec_coll); ++i)
    {
        printf("%d ", *(int*)vector_at(pvec_coll, i));
    }
    printf("\n");

    vector_destroy(pvec_coll);

    return 0;
}

```

使用 vector_t 容器必须包含头文件

```
#include <cstl/cvector.h>
```

创建一个存储 int 类型数据的 vector_t

```
vector_t* pvec_coll = create_vector(int);
```

vector_t 在没有初始化的时候是不能使用的，下面是初始化的代码

```
vector_init(pvec_coll);
```

我们使用 vector_push_back() 函数向 vector_t 中添加数据

```

for(i = 1; i <= 6; ++i)
{
    vector_push_back(pvec_coll, i);
}

```

vector_size() 函数返回 vector_t 容器中数据的个数，vector_at() 是通过下标对 vector_t 容器中数据进行访问，它返回的是指向容器中相应数据的指针，要打印出数据的内容我们使用如下的代码：

```

for(i = 0; i < vector_size(pvec_coll); ++i)
{
    printf("%d ", *(int*)vector_at(pvec_coll, i));
}

```

最后销毁 vector_t 容器，以释放容器本身和为了保存数据申请的资源

```
vector_destroy(pvec_coll);
```

这段代码输出的结果如下：

```
1 2 3 4 5 6
```

● deque_t

deque_t 是双端队列，它也是一个动态数组，但是可以在两端生长，所以在开头和结尾插入和删除数据都很高效，但是在中间插入和删除数据很慢，因为它也要移动数据。

与上面的例子类似，这次向 deque_t 中添加 6 个浮点数并打印出来：

```
/*
 * deque1.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cdeque.h>

int main(int argc, char* argv[])
{
    /* deque container for float-point elements */
    deque_t* pdq_coll = create_deque(float);
    int i = 0;

    if(pdq_coll == NULL)
    {
        return -1;
    }

    deque_init(pdq_coll);

    /* insert elements from 1.1 to 6.6 each at the front */
    for(i = 1; i <= 6; ++i)
    {
        /* insert at the front */
        deque_push_front(pdq_coll, i * 1.1);
    }

    /* print all elements followed by a space */
    for(i = 0; i < deque_size(pdq_coll); ++i)
    {
        printf("%f ", *(float*)deque_at(pdq_coll, i));
    }
    printf("\n");

    deque_destroy(pdq_coll);

    return 0;
}
```

使用 deque_t 需要包含头文件

```
#include <cstl/cdeque.h>
```


创建一个保存浮点数类型数据的 deque_t 容器并初始化

```
deque_t* pdq_coll = create_deque(float);
```

```
...
```

```
deque_init(pdq_coll);
```

在 deque_t 开头插入数据

```
deque_push_front(pdq_coll, i * 1.1);
```

以这种方式插入数据得到的结果是插入顺序相反的序列，结果如下：

```
6.600000 5.500000 4.400000 3.300000 2.200000 1.100000
```

这是因为每一个数据都插入到了第一个数据的前面。

deque_t 容器也提供了 deque_size() 函数和 deque_at() 函数，所以我们可以使用与第一个例子相同的方式打印数据：

```
for(i = 0; i < deque_size(pdq_coll); ++i)
{
    printf("%f ", *(float*)deque_at(pdq_coll, i));
}
```

最后也要销毁 deque_t

```
deque_destroy(pdq_coll);
```

因为 deque_t 是双端队列，所以它也提供了 deque_push_back() 函数向 deque_t 的末尾插入数据，相反 vector_t 并没有提供 vector_push_front() 这样的函数，因为在 vector_t 开头插入数据效率很低。

● list_t

list_t 是一个双向链表。它不具备随即访问数据的能力，如果要访问第十个元素就必须从第一个开始向后遍历，一直找到第十个元素为止，但是对于一个给定的元素找到它的前驱和后继都是非常快的。list_t 的优点是在任意位置删除和插入数据都非常快。

下面的例子创建一个字符类型的空链表，插入从 'a' 到 'z' 的字符，然后从开头删除并打印每一个字符：

```
/*
 * list1.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/clist.h>

int main(int argc, char* argv[])
{
    /* list container for character elements */
    list_t* plist_coll = create_list(char);
    char c = '\0';

    if(plist_coll == NULL)
    {
        return -1;
    }
}
```

```

list_init(plist_coll);

/* append elements from 'a' to 'z' */
for(c = 'a'; c <= 'z'; ++c)
{
    list_push_back(plist_coll, c);
}

/* print all elements
 * - while there are elements
 * - print and remove the first element
 */
while(!list_empty(plist_coll))
{
    printf("%c ", *(char*)list_front(plist_coll));
    list_pop_front(plist_coll);
}
printf("\n");

list_destroy(plist_coll);

return 0;
}

```

使用 list_t 需要包含头文件

```
#include <cstl/clist.h>
```

创建一个字符类型的 list_t 并初始化

```
list_t* plist_coll = create_list(char);
```

```
...
```

```
list_init(plist_coll);
```

向 list_t 中插入字符

```
for(c = 'a'; c <= 'z'; ++c)
```

```

{
    list_push_back(plist_coll, c);
}

```

list_empty() 判断 list_t 是否为空, list_front() 访问第一个元素, list_pop_front() 删除第一个元素, 打印并删除数据的代码:

```

while(!list_empty(plist_coll))
{
    printf("%c ", *(char*)list_front(plist_coll));
    list_pop_front(plist_coll);
}

```

销毁 list_t

```
list_destroy(plist_coll);
```

最后的结果:

```
abcdefghijklmnopqrstuvwxyz
```

向上面这样遍历一个链表后链表中的数据就都被删除了, 如果要保留数据请使用迭代器对链表进行遍历, 迭代器在后面具体介绍。

- **slist_t**

slist_t 是一个单链表(stl 标准中并不包含 slist, 但是 sgi stl 包含一个单链表 slist). 它与 list_t 的特点相同, 对于任意位置的插入和删除数据都是非常快的. 但是 slist_t 寻找前驱的操作效率很低, 因为它必须从第一个元素开始查找.

- **string_t**

string_t 的行为与 vector_t 相似, 但是它主要用来存储字符串, 同时它还提供了很多关于字符串的特殊操作.

2. 关联容器

关联容器对容器中保存的数据自动排序, 这种排序是基于数据本身的值或者数据定义的键值。关联容器对数据排序的默认规则是小于操作, 对于基本类型和 libcstl 内部的容器类型还有一些工具类型等 libcstl 库都提供了默认的小于操作函数, 但是用户自定义类型的默认小于操作函数可以在注册这个类型的时候指定 (类型注册机制在第十章介绍)。

关联容器分集合和映射两种, 集合和映射的不同点在于集合对数据的排序是基于数据本身, 但是映射是以键/值对的方式保存数据, 所以映射对数据的排序是基于键的。

对于集合和映射来说又分为集合和多重集合, 映射和多重映射。集合和多重集合的区别在于前者保证数据的唯一性, 即不允许数据重复, 后者是允许数据重复的。对于映射和多重映射也是同样, 但是区别在于映射只是不允许键重复, 多重映射允许出现重复的键。这样看来可以将集合看作是整值作为键的特殊的映射。

关联容器为了实现对数据进行自动排序的功能一般都采用平衡二叉树的结构, libcstl 采用了两种平衡二叉树来实现关联容器, AVL 树和红黑树, 可以通过在编译的时候添加编译选项来选择关联容器的底层实现 (特殊的编译选项请参考第一章), 建议使用红黑树作为底层实现, 因为红黑树是效率更高的平衡二叉树。除了采用平衡二叉树实现的关联容器为, libcstl 还提供了一套基于哈希结构的关联容器, 这套关联容器同样也包括集合, 多重集合, 映射, 多重映射。这一套关联容器采用的底层实现是哈希表结构, 基于哈希结构的容器在插入, 删除和查找数据可以接近常数时间 (只要哈希函数选择的足够好), 但是基于平衡二叉树结构的关联容器相比它不是完全有序的。

下面列出了 libcstl 库提供的全部关联容器类型:

- **set_t**

set_t 容器根据它保存的数据进行自动排序, 每一个数据在容器中只出现一次, 重复的数据是不允许的。

- **multiset_t**

multiset_t 容器除了允许保存的数据重复, 其他的与 set_t 容器相同。

- **map_t**

map_t 容器中保存的是数据 key/value 对, 数据根据 key 值自动排序, 不允许有 key 重复的数据. 同时 map_t 具有关联数组的能力, 使用 key 值可以随即访问 map_t 中相应的 value 值。

- **multimap_t**

multimap_t 允许有 key 值重复的 key/value 数据, 并且它不具备关联数组的性质, 除此之外与 map_t 特点相同。

- **hash_set_t**

hash_set_t 除了底层采用 hash 表实现外其他的特性与 set_t 相同。

- **hash_multiset_t**

hash_multiset_t 除了底层采用 hash 表实现外其他的特性与 multiset_t 相同。

- **hash_map_t**

hash_map_t 除了底层采用 hash 表实现外其他的特性与 map_t 相同。

- **hash_multimap_t**

hash_multimap_t 除了底层采用 hash 实现外其他的特性与 multimap_t 相同。

由于操作关联容器要用到迭代器，所以关联容器的例子在介绍完迭代器之后给出。

3. 容器适配器

除了以上这些容器之外, libcstl 为了特殊的目的还提供了容器适配器, 它们都是基本的容器实现的。

- **stack_t**

stack_t 是堆栈, 它采用 LIFO (后进先出) 的原则来管理数据。

- **queue_t**

queue_t 是队列, 它采用 FIFO (先进先出) 的原则管理数据。

- **priority_queue_t**

priority_queue_t 是优先队列, 它保存的数据具有优先级, libcstl 默认的数据值越大优先级越大. 每次从队列中取出的都是具有最大优先级的数据。

其中堆栈和队列两种适配器都是可以使用多种容器类型作为底层实现, 通过在编译的时候使用特殊的编译选项来实现, 具体请参考第一章。容器适配器除了底层实现采用容器外, 它们都不支持迭代器, 所以不能通过算法对它们保存的数据进行操作。

第三节 迭代器

迭代器是一种实现的是一种指针语义, 它是指向容器中数据的”智能指针”, 通过迭代器可以访问或者遍历容器中的全部或者部分数据, 同时算法也是通过迭代器对容器中的数据进行操作的。也正是因为有了迭代器才实现了数据和算法分离。

迭代器是将容器的内部结构隐藏, 通过使用它可以对不同的容器进行同样的操作, 同时对于不同的容器迭代器也有特殊的操作, 但是下面的这些操作是迭代器共同的:

- **iterator_get_value()**

获得迭代器所指的数据, 将迭代器所指的数据拷贝到指定的缓冲区中。

- **iterator_get_pointer()**

获得迭代器所指的数据的指针。

- **iterator_next()**

指向下一个数据, 使迭代器向前行进一步。

- **iterator_equal()** 和 **iterator_not_equal()**

判断两个迭代器是否指向相同的位置。

- 赋值运算符 =

在 `libcstl` 中，迭代器类型可以直接使用赋值运算符 `=` 来进行赋值。

上面都是迭代器本身的操作，下面我们来看一看容器是怎样和迭代器建立联系的：

为了实现容器和迭代器的相互操作，所有的容器都提供了基本的函数。

- `xxxx_begin()`

获得指向特定容器中第一个数据的迭代器(如果有第一个数据)。例如 `vector_t` 容器就是 `vector_begin()`。

- `xxxx_end()`

获得指向特定容器末尾的迭代器，这个末尾是指容器中最后一个数据的下一个位置。例如 `vector_t` 容器就是 `vector_end()`。



图 2.4 `xxxx_begin()` 和 `xxxx_end()`

这样的 `xxxx_begin()` 和 `xxxx_end()` 就组成了一个左闭右开区间 `[xxxx_begin(), xxxx_end())`，这样的好处是便利时对结束条件判断比较方便，如果容器为空那么 `xxxx_begin()` 就等于 `xxxx_end()`。

使用一个例子来看看迭代器如何使用。下面的例子是将链表中的数据都打印出来，是对前面的例子的改版：

```
/*
 * list2.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/clist.h>

int main(int argc, char* argv[])
{
    /* list container for character elements */
    list_t* plist_coll = create_list(char);
    list_iterator_t it_pos;
    char c = '\0';

    if(plist_coll == NULL)
    {
```

```

        return -1;
    }

    list_init(plist_coll);

    /* append elements from 'a' to 'z' */
    for(c = 'a'; c <= 'z'; ++c)
    {
        list_push_back(plist_coll, c);
    }

    /* print all elements
     * - iterate over all elements
     */
    for(it_pos = list_begin(plist_coll);
        !iterator_equal(it_pos, list_end(plist_coll));
        it_pos = iterator_next(it_pos))
    {
        printf("%c ", *(char*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    list_destroy(plist_coll);

    return 0;
}

```

首先要定义一个迭代器:

```
list_iterator_t it_pos;
```

迭代器与容器不同它不需要创建和初始化, 定义之后直接就可以使用, 使用之后也无需销毁。

使用 it_pos 遍历 list_t 容器, 并获得每一个数据的指针, 将每一个数据打印出来:

```

for(it_pos = list_begin(plist_coll);
    !iterator_equal(it_pos, list_end(plist_coll));
    it_pos = iterator_next(it_pos))
{
    printf("%c ", *(char*)iterator_get_pointer(it_pos));
}

```

操作过程如图:



图 2.5 使用迭代器遍历链表中的数据

在遍历的开始首先让迭代器 `it_pos` 指向链表的第一个数据：

```
it_pos = list_begin(plist_coll);
```

然后判断当前位置是否为链表的末尾：

```
!iterator_equal(it_pos, list_end(plist_coll));
```

如果不是链表的末尾就或者指向当前数据的指针，并将数据打印出来：

```
printf("%c ", *(char*)iterator_get_pointer(it_pos));
```

然后在移动到下一个数据的位置：

```
it_pos = iterator_next(it_pos)
```

1. 使用关联容器的例子

上面例子中的遍历是通用的，它可以用于任何容器中，下面就将这种便利用在关联容器中。下面这些例子都是使用关联容器的例子：

- `set_t` 和 `multiset_t` 的例子

下面的例子展示了如何向 `set_t` 中插入数据并使用迭代器将数据打印出来：

```
/*
 * set1.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cset.h>

int main(int argc, char* argv[])
{
    /* set container for int values */
    set_t* pset_coll = create_set(int);
    set_iterator_t it_pos;
```

```

if(pset_coll == NULL)
{
    return -1;
}

set_init(pset_coll);

/*
 * insert elements for 1 to 6 in arbitray order
 * - value 1 gets inserted twice
 */
set_insert(pset_coll, 3);
set_insert(pset_coll, 1);
set_insert(pset_coll, 6);
set_insert(pset_coll, 4);
set_insert(pset_coll, 1);
set_insert(pset_coll, 2);
set_insert(pset_coll, 5);

/*
 * print all elements
 * - iterate over all elements
 */
for(it_pos = set_begin(pset_coll);
    !iterator_equal(it_pos, set_end(pset_coll));
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");

set_destroy(pset_coll);

return 0;
}

```

要使用 set_t 必须包含头文件

```
#include <cstl/cset.h>
```

然后创建并初始化一个保存 int 类型的 set_t

set_t 提供了 set_insert() 函数，使用它向容器中插入数据。set_t 并没有提供 set_push_back() 或者 set_push_front() 这样的操作函数，因为在数据插入到容器之前它的位置是未知的。

```
set_insert(pset_coll, 3);
```



```
set_insert(pset_coll, 1);  
set_insert(pset_coll, 6);  
set_insert(pset_coll, 4);  
set_insert(pset_coll, 1);  
set_insert(pset_coll, 2);  
set_insert(pset_coll, 5);
```

向 set_t 中插入了 7 个数据但是实际上容器只保存了 6 个, 第二次插入数据 1 的时候因为容器中已经有数据 1 了所以插入失败, 数据在 set_t 中的可能保存形式:

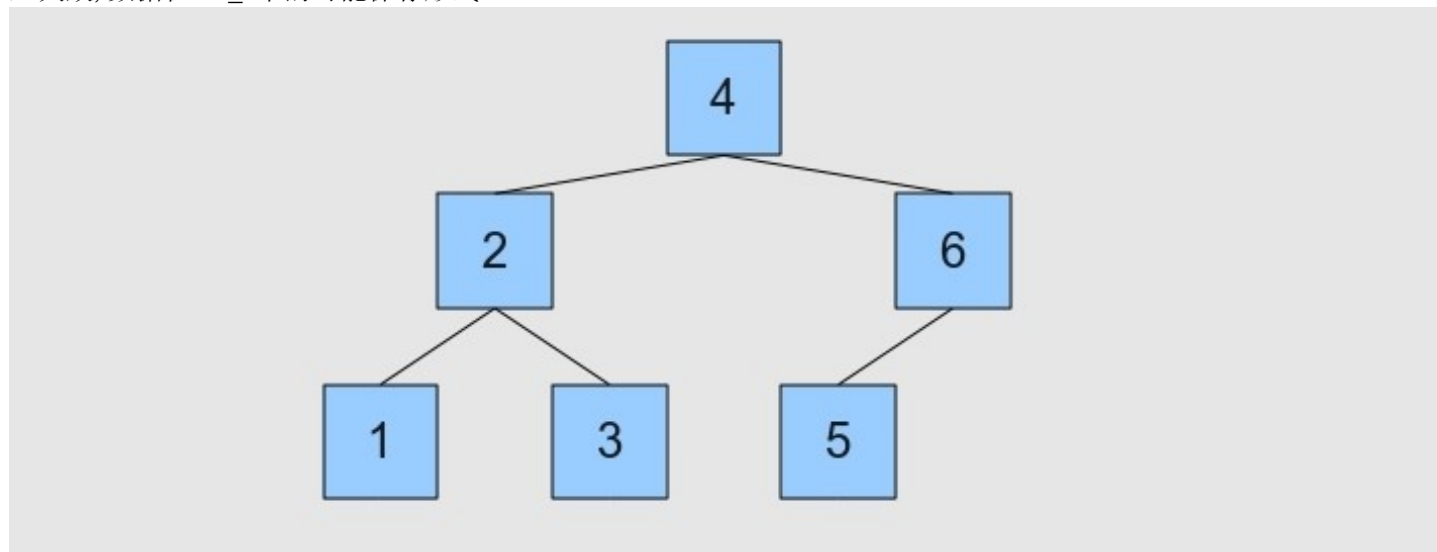


图 2.6 插入数据后的 set_t 容器

接下来是使用迭代器遍历 set_t, 使用了和上面的例子相同的遍历方式:

```
for(it_pos = set_begin(pset_coll);  
    !iterator_equal(it_pos, set_end(pset_coll));  
    it_pos = iterator_next(it_pos))  
{  
    printf("%d ", *(int*)iterator_get_pointer(it_pos));  
}
```

虽然用户在遍历的时候使用的操作是一样的, 但是由于内部结构不同迭代器遍历的过程中内部实现还是要根据各个容器的具体情况, 让我们来看看关联容器使用迭代器遍历时是什么情况:

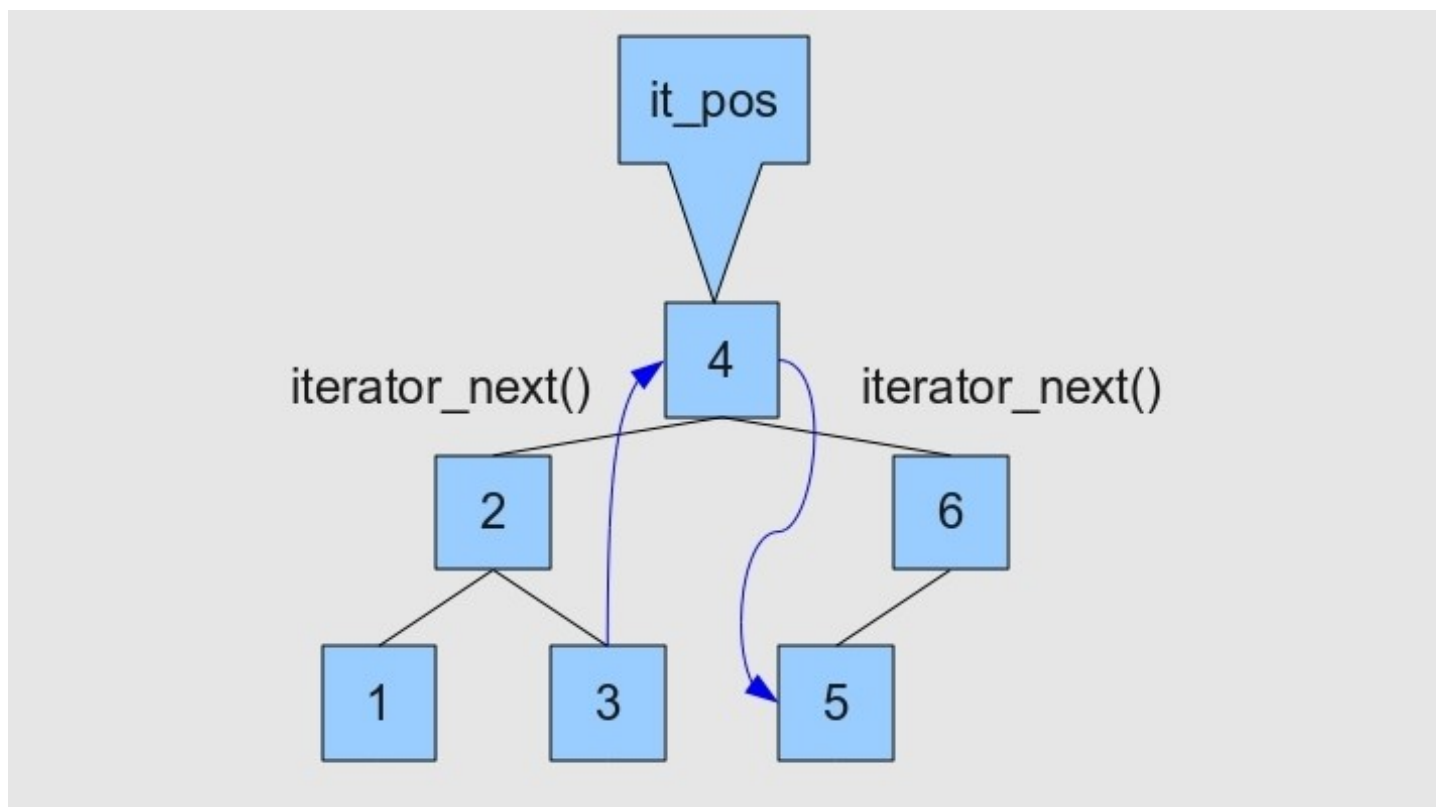


图 2.7 通过迭代器遍历关联容器

上面例子的结果:

1 2 3 4 5 6

如果使用 `multiset_t` 代替 `set_t`, 第二次插入数据 1 就不会失败, 输出结果如下:

1 1 2 3 4 5 6

- 使用 `map_t` 和 `multimap_t` 的例子

`map_t` 和 `multimap_t` 保存的数据都是 key/value 对, 所以对于它们的声明, 插入和访问都有些不同, 下面的例子使用了 `multimap_t`:

```

/*
 * multimap1.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cmap.h>
#include <cstl/cutility.h>

int main(int argc, char* argv[])
{
    /* multimap container for int/c-string value */
    multimap_t* pmmmap_coll = create_multimap(int, char*);
    multimap_iterator_t it_pos;

```

```

pair_t* ppair_elem = create_pair(int, char*);

if(pmmmap_coll == NULL || ppair_elem == NULL)
{
    return -1;
}

multimap_init(pmmmap_coll);
pair_init(ppair_elem);

/*
 * insert some elements in arbitray order
 * - a value with key 1 gets inserted twice
 */
pair_make(ppair_elem, 5, "tagged");
multimap_insert(pmmmap_coll, ppair_elem);
pair_make(ppair_elem, 2, "a");
multimap_insert(pmmmap_coll, ppair_elem);
pair_make(ppair_elem, 1, "this");
multimap_insert(pmmmap_coll, ppair_elem);
pair_make(ppair_elem, 4, "of");
multimap_insert(pmmmap_coll, ppair_elem);
pair_make(ppair_elem, 6, "strings");
multimap_insert(pmmmap_coll, ppair_elem);
pair_make(ppair_elem, 1, "is");
multimap_insert(pmmmap_coll, ppair_elem);
pair_make(ppair_elem, 3, "multimap");
multimap_insert(pmmmap_coll, ppair_elem);

/*
 * print all element values
 * - iterate over all elements
 * - element member second is value
 */
for(it_pos = multimap_begin(pmmmap_coll);
    !iterator_equal(it_pos, multimap_end(pmmmap_coll));
    it_pos = iterator_next(it_pos))
{
    printf("%s ", (char*)pair_second(iterator_get_pointer(it_pos)));
}
printf("\n");

multimap_destroy(pmmmap_coll);

```

```

    pair_destroy(ppair_elem);

    return 0;
}

```

输出的结果是：

this is a multimap of tagged strings

this 和 is 的键都是 1，所以在结果中 this 和 is 有可能顺序颠倒。虽然 this 和 is 具有同样的键但是 multimap_t 允许键重复。

这个例子有点复杂

首先，multiset_t 使用 pair_t 来描述 key/value，所以 multimap_t 中保存的是 pair_t。因此在程序中要声明一个 pair_t 并将数据保存在其中，然后插入到 multimap_t 中。这也是代码中在调用 multimap_insert() 之前都调用了 pair_make()。

其次，由于 multimap_t 中保存的是 pair_t，所以在遍历的过程中获得的数据的指针都是指向 pair_t 类型的，我们要获得的值要通过 pair_t 的操作函数 pair_second() 来获得：

```
printf("%s ", (char*)pair_second(iterator_get_pointer(it_pos)));
```

pair_t 类型第三章中详细的介绍。

- 将 map_t 作为关联数组

map_t 和 multimap_t 的主要区别在于 map_t 不允许键重复。除了这些，map_t 还可以作为关联数组使用：

```

/*
 * map1.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cmap.h>

int main(int argc, char* argv[])
{
    /* map container for c-string/double value */
    map_t* pmap_coll = create_map(char*, double);
    map_iterator_t it_pos;

    if(pmap_coll == NULL)
    {
        return -1;
    }

    map_init(pmap_coll);

    /* insert some element into the collection */
    *(double*)map_at(pmap_coll, "VAT") = 0.15;
    *(double*)map_at(pmap_coll, "Pi") = 3.1415;
    *(double*)map_at(pmap_coll, "an arbitray number") = 445.903;
}

```

```

*(double*)map_at(pmap_coll, "Null") = 0.0;

/*
 * print all element values
 * - iterate over all elements
 * - element member first is key
 * - element member second is value
 */
for(it_pos = map_begin(pmap_coll);
    !iterator_equal(it_pos, map_end(pmap_coll));
    it_pos = iterator_next(it_pos))
{
    printf("key: \"%s\" value: %lf\n",
        (char*)pair_first(iterator_get_pointer(it_pos)),
        *(double*)pair_second(iterator_get_pointer(it_pos)));
}

map_destroy(pmap_coll);

return 0;
}

```

map_t 提供了 map_at() 操作函数，允许使用以键值为下标直接修改 map_t 中的数据。map_at() 与其他容器的 at() 函数不同，它可以以任何类型作为索引，它也没有下标失效的情况。当使用一个容器中不存在的键作下标的时候，map_at() 会首先在容器中生成一个默认的数据，这个数据的键就是当前的下标，值是值类型的默认值。例如：

```
*(double*)map_at(pmap_coll, "VAT") = 0.15;
```

如果像下面这样的写法：

```
map_at(pmap_coll, "VAT");
```

“VAT”键对应的值就是双精度浮点值的默认值 0.0。

执行结果：

```
key: "Null" value: 0.000000
```

```
key: "Pi" value: 3.141500
```

```
key: "VAT" value: 0.150000
```

```
key: "an arbitray number" value: 445.903000
```

2. 迭代器种类

上面已经介绍了迭代器是为了给用户提供一种统一的操作方式，也列举了迭代器的基本操作，但是容器的功能不同内部结构也就不同，所以迭代器除了提供了基本的操作外还针对不同种类的容器提供了不同的操作。

根据迭代已经访问数据的能力，可以将迭代器分为五大类(具体的种类在第五章中介绍)，所有的 libcstl 提供的容器的迭代器都属于其中的三类：

- 单向迭代器(forward_iterator_t)

单向迭代器只能向下一个元素移动:使用 iterator_next()操作, 拥有这类迭代器类型的容器只有 slist_t。

- 双向迭代器(bidirectional_iterator_t)

双向迭代器可以向两个方向迭代, 向下一个元素使用 iterator_next()操作, 向前一个元素使用 iterator_prev()操作, 拥有这类迭代器类型的容器有 list_t, set_t, multiset_t, map_t, multimap_t, hash_set_t, hash_multiset_t, hash_map_t, hash_multimap_t。

- 随机访问迭代器(random_access_iterator_t)

随即访问迭代器具有随即访问的能力, 此为还可以进行关系比较, 可以一次移动多个数据, 还可以计算两个迭代器的差值。拥有这类迭代器类型的容器有 vector_t 和 deque_t。

第四节 算法和函数

libcstl 为了处理容器中的数据提供了许多算法, 例如查找, 排序, 拷贝, 修改还有算术操作。算法不属于任何一种容器, 它是通用的, 可以处理任何容器中的数据。通过迭代器实现的算法和数据的分离, 但是这样做也有不足的地方, 为了做到通用, 必然会忽略容器结构的特殊性, 这样于容器提供的操作函数相比, 实现同样的功能的情况下, 容器提供的操作函数更高效。下面通过一个简单的例子来展示一下如何使用算法:

```
/*
 * algo1.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

int main(int argc, char* argv[])
{
    vector_t* pvec_coll = create_vector(int);
    vector_iterator_t it_pos;

    if(pvec_coll == NULL)
    {
        return -1;
    }

    vector_init(pvec_coll);

    /* insert elements from 1 to 6 in arbitrary order */
    vector_push_back(pvec_coll, 3);
    vector_push_back(pvec_coll, 2);
    vector_push_back(pvec_coll, 5);
    vector_push_back(pvec_coll, 6);
```

```

vector_push_back(pvec_coll, 4);
vector_push_back(pvec_coll, 1);

/* find and print minimum and maximum element */
it_pos = algo_min_element(vector_begin(pvec_coll), vector_end(pvec_coll));
printf("min: %d\n", *(int*)iterator_get_pointer(it_pos));
it_pos = algo_max_element(vector_begin(pvec_coll), vector_end(pvec_coll));
printf("max: %d\n", *(int*)iterator_get_pointer(it_pos));

/* sort all elements */
algo_sort(vector_begin(pvec_coll), vector_end(pvec_coll));

/* find the first element with value 3 */
it_pos = algo_find(vector_begin(pvec_coll), vector_end(pvec_coll), 3);

/*
 * reverse the order of the found element with value 3
 * and all following elements
 */
algo_reverse(it_pos, vector_end(pvec_coll));

/* print all elements */
for(it_pos = vector_begin(pvec_coll);
    !iterator_equal(it_pos, vector_end(pvec_coll));
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");

vector_destroy(pvec_coll);

return 0;
}

```

要使用 libcstl 算法必须包含头文件

```
#include <cstl/calgorithm.h>
```

最开始的两个算法是 algo_min_element()和 algo_max_element()这两个算法都是接受两个迭代器作为参数,并把它视为一个数据区间,在这个区间内分别找出最小的元素和最大的元素,返回指向找到的元素的迭代器。

接下来的算法 algo_sort()是将两个迭代器表示的数据区间中的数据进行排序,结果是:

```
1 2 3 4 5 6
```

然后通过 algo_find()算法找到元素 3 的位置

```
it_pos = algo_find(vector_begin(pvec_coll), vector_end(pvec_coll), 3);
```

成功就返回指向数据 3 的位置的迭代器，如果失败返回数据区间末尾的迭代器。

然后把从数据 3 的位置开始到容器的末尾数据逆序

```
algo_reverse(it_pos, vector_end(pvec_coll));
```

这样例子中程序的输出结果就是：

min: 1

max: 6

1 2 6 5 4 3

1. 数据区间

既然算法是通过迭代器来处理容器中的数据的，那么具体是怎样处理的呢？为例解答这个问题要先给出一个概念：数据区间。数据区间是由两个迭代器组成的，这两个迭代器所表示的位置形成了一个左闭右开区间，这个区间就叫做数据区间，这个区间也是算法要处理的对象。有效数据区间必须去是两个迭代器属于同一个容器，而且第一个迭代器的位置要在第二个迭代器之前或者和第二个迭代器相等。除此之外的迭代器对都是无效的数据区间。

算法处理的就是一个或多个数据区间内的数据，这个数据区间不一定是容器中的所有数据，有可能是容器的子集。算法一般要求一个或者多个数据区间，这样做很方便，但是很危险，调用者必须保证传递给算法的数据区间是有效的，否则算法的行为是未定义的（带有断言的版本会使检测到非法的区间并触发断言）。

虽然这样做会有很多好处，但是也有不足：

```
/*
 * find1.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/clist.h>
#include <cstl/calgorithm.h>

int main(int argc, char* argv[])
{
    list_t* plist_coll = create_list(int);
    list_iterator_t it_pos;
    list_iterator_t it_pos25;
    list_iterator_t it_pos35;
    int i = 0;

    if(plist_coll == NULL)
    {
        return -1;
    }

    list_init(plist_coll);
```



```

/* insert elements from 20 to 40 */
for(i = 20; i <= 40; ++i)
{
    list_push_back(plist_coll, i);
}

/*
 * find position of element with value 3
 * - there is none, so it_pos gets end()
 */
it_pos = algo_find(list_begin(plist_coll), list_end(plist_coll), 3);

/*
 * reverse the order of element between found element and then end
 * - because the it_pos is end(), it reverses an empty range
 */
algo_reverse(it_pos, list_end(plist_coll));

/* find the position of value 25 and 35 */
it_pos25 = algo_find(list_begin(plist_coll), list_end(plist_coll), 25);
it_pos35 = algo_find(list_begin(plist_coll), list_end(plist_coll), 35);

/*
 * print the maximum of corresponding range
 * - note: including pos25 and exculding pos35
 */
printf("max: %d\n",
    *(int*)iterator_get_pointer(algo_max_element(it_pos25, it_pos35)));

list_destroy(plist_coll);

return 0;
}

```

首先使用 20 到 40 的整数填充 list_t，当查找数据 3 的位置时会失败，algo_find()返回要处理的数据区间的末尾，在这个例子中是 list_end()。接下来我们调用 algo_reverse()对 it_pos 和 list_end()范围进行逆序的处理，这是没有问题的，it_pos 等于 list_end()相当于调用

```
algo_reverse(list_end(plist_coll), list_end(plist_coll));
```

这个算法空转，没有任何效果。

然后在 list_t 中查找 25 和 35 两个数据的位置，分别是 it_pos25 和 it_pos35，然后调用 algo_max_element()算法找出范围 [it_pos25, it_pos35)中的最大元素，结果是 34 而不是 35。

max: 34

这是因为指向数据 35 的迭代器 it_pos35 是区间的末端，不会被处理，要想处理数据 35 就必须将 it_pos35 指向下一个

数据:

```
it_pos35 = iterator_next(it_pos35);
```

这次再查找得到了正确的结果:

max: 35

这个例子中有个问题, 如果想要处理区间末端的迭代器所指向的数据, 就必须把迭代器指向下一个数据。另一个问题在上面的例子中我们已经知道了 `it_pos25` 在 `it_pos35` 的前面`[it_pos25, it_pos35)`是有效的区间, 如果 `it_pos25` 在 `it_pos35` 之后, 那么就会使区间无效, `algo_max_element()`会执行未定义的操作, 造成危险。

2. 处理多个数据区间

许多算法要同时处理多个数据区间, 对于这种情况通常需要明确的给出第一个数据区间的起点和终点, 至于其他的区间只需要给出起点就可以了, 重点可以根据第一个区间推算出来。但是这样就要求在使用这样的算法的时候至少要保证第二个及后续的区间的范围至少要和第一个区间一样大, 否则将会产生类似越界的问题, 这种行为也是未定义的。考虑下面这个例子:

```
/*
 * copy1.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/clist.h>
#include <cstl/calgorithm.h>

int main(int argc, char* argv[])
{
    list_t* plist_coll1 = create_list(int);
    vector_t* pvec_coll2 = create_vector(int);
    int i = 0;

    if(plist_coll1 == NULL || pvec_coll2 == NULL)
    {
        return -1;
    }

    list_init(plist_coll1);
    vector_init(pvec_coll2);

    for(i = 1; i <= 9; ++i)
    {
        list_push_back(plist_coll1, i);
    }
}
```

```

/*
 * RUNTIME ERROR.
 * overwrites nonexisting elements in the destination.
 */
algo_copy(list_begin(plist_coll1), list_end(plist_coll1), /* source */
          vector_begin(pvec_coll2));                      /* destination */

list_destroy(plist_coll1);
vector_destroy(pvec_coll2);

return 0;
}

```

在调用算法 `algo_copy` 时将第一个范围内的所有数据 [`list_begin()`, `list_end()`) 拷贝到以 `vector_begin()` 开始的第二个范围内, 算法执行的都是覆盖的操作而不是插入, 所以要求第二个范围有足够的空间, 如果没有足够的空间 (像上面的例子中那样) 那么结果是未定义的。使用断言版本的 `libcstl` 库在这种情况下会触发断言。下面的例子保证了第二个范围有足够的空间:

```

/*
 * copy2.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/clist.h>
#include <cstl/cdeque.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

int main(int argc, char* argv[])
{
    list_t* plist_coll1 = create_list(int);
    vector_t* pvec_coll2 = create_vector(int);
    deque_t* pdeq_coll3 = create_deque(int);
    int i = 0;

    if(plist_coll1 == NULL || pvec_coll2 == NULL || pdeq_coll3 == NULL)
    {
        return -1;
    }

    list_init(plist_coll1);
    vector_init(pvec_coll2);

```

```

/* insert elements from 1 to 9 */
for(i = 1; i <= 9; ++i)
{
    list_push_back(plist_coll1, i);
}

/*
 * resize destination to have enough room for
 * the overwrite algorithm
 */
vector_resize(pvec_coll2, list_size(plist_coll1));

/* copy elements from first to second collection */
algo_copy(list_begin(plist_coll1), list_end(plist_coll1),
          vector_begin(pvec_coll2));

/* initialize third collection with enough room */
deque_init_n(pdeq_coll3, list_size(plist_coll1));

/* copy elements from first to third collection */
algo_copy(list_begin(plist_coll1), list_end(plist_coll1),
          deque_begin(pdeq_coll3));

list_destroy(plist_coll1);
vector_destroy(pvec_coll2);
deque_destroy(pdeq_coll3);

return 0;
}

```

vector_resize()保证了 pvec_coll2 具有和 plist_coll1 相同数目的数据，使得第二个范围足够大。而 pdeq_coll3 在初始化的时候就保证了拥有和 plist_coll1 同样大的空间。

3. 质变算法

有些算法对数据区间内的数据进行修改，这些算法可能移除，或者修改了数据，我们把这样的算法叫质变算法。典型的质变算法 algo_remove()是从一个数据区间内删除指定的数据，下面的例子会产生让你惊奇的结果：

```

/*
 * remove1.c
 * compile with : -lcstl

```

```

*/

#include <stdio.h>
#include <cstl/clist.h>
#include <cstl/calgorithm.h>

int main(int argc, char* argv[])
{
    list_t* plist_coll = create_list(int);
    list_iterator_t it_pos;
    int i = 0;

    if(plist_coll == NULL)
    {
        return -1;
    }

    list_init(plist_coll);

    /* insert element from 1 to 6 and 6 to 1 */
    for(i = 1; i <= 6; ++i)
    {
        list_push_back(plist_coll, i);
        list_push_front(plist_coll, i);
    }

    /* print all elements of the list */
    printf("Pre: ");
    for(it_pos = list_begin(plist_coll);
        !iterator_equal(it_pos, list_end(plist_coll));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    /* remove all elements with value 3 */
    algo_remove(list_begin(plist_coll), list_end(plist_coll), 3);

    printf("Post: ");
    for(it_pos = list_begin(plist_coll);
        !iterator_equal(it_pos, list_end(plist_coll));
        it_pos = iterator_next(it_pos))

```

```

{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");

list_destroy(plist_coll);

return 0;
}

```

您可能认为结果是所有数值等于 3 的数据都被从数据区间中删除了，然而结果却不像想象中那样：

Pre: 6 5 4 3 2 1 1 2 3 4 5 6

Post: 6 5 4 2 1 1 2 4 5 6 5 6

执行 `algo_remove()` 后数据的数目并没有减少，改变的是所有的 3 被后面的数据覆盖，在范围的末尾两个数据并没有被覆盖。这个算法更应该叫做移除而不是删除。

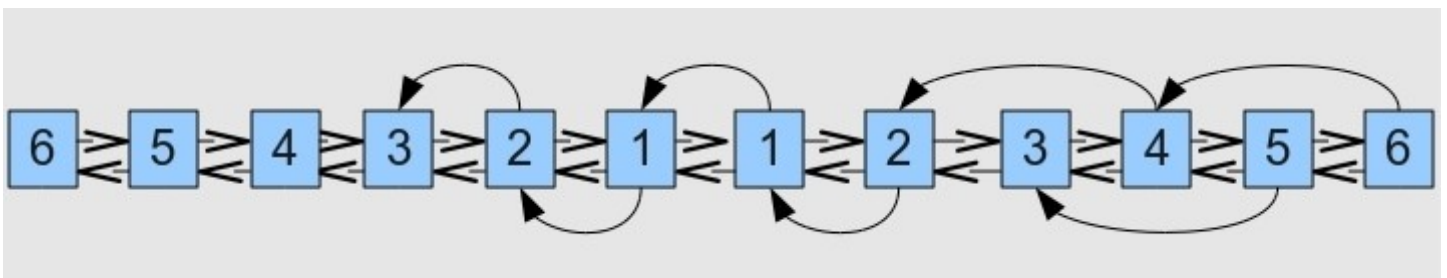


图 2.8 `algo_remove()`

这个算法返回有效数据的结尾，你可利用这个新的结尾来处理残余数据：

```

/*
 * remove2.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/clist.h>
#include <cstl/calgorithm.h>

int main(int argc, char* argv[])
{
    list_t* plist_coll = create_list(int);
    list_iterator_t it_pos;
    list_iterator_t it_end;
    int i = 0;

    if (plist_coll == NULL)

```

```

{
    return -1;
}

list_init(plist_coll);

/* insert element from 1 to 6 and 6 to 1 */
for(i = 1; i <= 6; ++i)
{
    list_push_back(plist_coll, i);
    list_push_front(plist_coll, i);
}

/* print all elements of the list */
for(it_pos = list_begin(plist_coll);
    !iterator_equal(it_pos, list_end(plist_coll));
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");

/* remove all elements with value 3 */
it_end = algo_remove(list_begin(plist_coll), list_end(plist_coll), 3);

/* print resulting elements of the collection */
for(it_pos = list_begin(plist_coll);
    !iterator_equal(it_pos, it_end);
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");

/* print number of resulting elements */
printf("number of removed elements: %d\n",
    iterator_distance(it_end, list_end(plist_coll)));

/* remove "removed" elements */
list_erase_range(plist_coll, it_end, list_end(plist_coll));

/* print all elements of modified collection */
for(it_pos = list_begin(plist_coll);

```

```

        !iterator_equal(it_pos, list_end(plist_coll));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    list_destroy(plist_coll);

    return 0;
}

```

在这个例子中 `algo_remove()` 返回了删除数据 3 后的有效数据区间的结尾迭代器。

```
it_end = algo_remove(list_begin(plist_coll), list_end(plist_coll), 3);
```

可以使用这个新的结尾做很多工作, 如获得这个新结尾和实际结尾之间的距离, 也就是实际删除的数据的数目。

```
iterator_distance(it_end, list_end(plist_coll));
```

还可以把残余的数据从 `plist_coll` 中真正的删除掉。

```
list_erase_range(plist_coll, it_end, list_end(plist_coll));
```

输出结果:

```
6 5 4 3 2 1 1 2 3 4 5 6
```

```
6 5 4 2 1 1 2 4 5 6
```

```
number of removed elements: 2
```

```
6 5 4 2 1 1 2 4 5 6
```

为什么 `algo_remove()` 不真正的删除数据呢, 这是因为算法是独立于容器的, 算法并不知道容器内部数据的保存形式, 由容器自己执行删除操作更容易和高效, 同时 `algo_remove()` 返回新的结尾迭代器这样对于将这个容器用于其他算法也不会产生任何影响。

4. 质变算法和关联容器

对数据进行修改的算法 (像删除, 排序, 修改等) 作用于关联容器时就会发生问题。关联容器是不能作为这类算法的目的容器的, 理由很简单, 关联容器中数据的存储顺序与数据本身有关, 一点数据被修改, 那么这种存储规则就被破坏了, 导致容器的操作和其他算法都会出现错误, 所以可以使用管理容器自己的操作来完成修改数据这样的任务:

```

/*
 * remove3.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cset.h>
#include <cstl/calgorithm.h>

```



```

int main(int argc, char* argv[])
{
    set_t* pset_coll = create_set(int);
    set_iterator_t it_pos;
    size_t t_num = 0;
    int i = 0;

    if(pset_coll == NULL)
    {
        return -1;
    }

    set_init(pset_coll);

    for(i = 1; i <= 9; ++i)
    {
        set_insert(pset_coll, i);
    }

    /* print all elements of set */
    for(it_pos = set_begin(pset_coll);
        !iterator_equal(it_pos, set_end(pset_coll));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    /* remove all element with value 3 */
    t_num = set_erase(pset_coll, 3);
    printf("number of removed elements: %u.\n", t_num);

    /* printf all elements of the modified set */
    for(it_pos = set_begin(pset_coll);
        !iterator_equal(it_pos, set_end(pset_coll));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    set_destroy(pset_coll);
}

```

```
    return 0;
}
```

可见关联容器本身也提供了很多修改数据的操作函数，这个例子的输出结果：

1 2 3 4 5 6 7 8 9

number of removed elements: 1.

1 2 4 5 6 7 8 9

5. 算法和容器操作函数

libcstl 提供了很多的算法，但是有时算法未必是最高效的，如果容器提供了实现同样功能的操作函数的话，那么使用操作函数会更高效。主要的原因是算法是为所有容器提供通用的操作，它是不了解容器的内部结构的，容器的操作函数只是针对特定容器，它更知道容器的内部实现，效率可能更高。所以使用的时候要权衡效率，有的时候容器提供了特定的操作，尽量使用容器操作函数。通过下面的例子来对比一下：

```
/*
 * remove4.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/clist.h>
#include <cstl/calgorithm.h>

int main(int argc, char* argv[])
{
    list_t* plist_coll = create_list(int);
    int i = 0;

    if(plist_coll == NULL)
    {
        return -1;
    }

    list_init(plist_coll);

    for(i = 1; i <= 6; ++i)
    {
        list_push_back(plist_coll, i);
        list_push_front(plist_coll, i);
    }
}
```

```

/*
 * remove all elements with value 3
 * - poor performance
 */
list_erase_range(plist_coll,
                algo_remove(list_begin(plist_coll), list_end(plist_coll), 3),
                list_end(plist_coll));

/*
 * remove all elements with value 4
 * - good performance
 */
list_remove(plist_coll, 4);

list_destroy(plist_coll);

return 0;
}

```

上面的例子中进行了两次删除操作，第一次使用先调用 `algo_remove()` 算法用后面的数据覆盖了值为 3 的数据之后在调用 `list_erase_range()` 来删除后面的”无效数据”。而第二次删除直接在链表结构中将值等于 4 的数据节点移除并销毁，真样就比第一种方法效率高了很多。

6. 用户自定义规则

为了使算法更具有扩展性和功能更强大，`libcstl` 的算法大部分都提供了扩展的版本，允许使用自定义的规则来改变算法的行为。用户可以将自定义的规则作为算法的参数传递给 `algo_xxxx_if()` 这样的算法版本。大部分的算法都提供了这种带有 `_if` 的版本，如 `algo_remove()` 和 `algo_remove_if()`，第一个算法删除与指定数据相等的数据，第二个算法删除符合指定条件的数据。有些算法要求用户必须提供自定义的规则，如 `algo_for_each()` 这个算法对范围内的每一个数据进行用户定义的操作：

```

/*
 * foreach1.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output)

```

```

{
    printf("%d ", *(int*)cpv_input);
}

int main(int argc, char* argv[])
{
    vector_t* pvec_coll = create_vector(int);
    int i = 0;

    if(pvec_coll == NULL)
    {
        return -1;
    }

    vector_init(pvec_coll);

    for(i = 1; i <= 9; ++i)
    {
        vector_push_back(pvec_coll, i);
    }

    algo_for_each(vector_begin(pvec_coll), vector_end(pvec_coll), _print);
    printf("\n");

    vector_destroy(pvec_coll);

    return 0;
}

```

algo_for_each()算法对于[vector_begin(), vector_end())中的每一个数据应用函数_print 输出其值，结果：

1 2 3 4 5 6 7 8 9

在许多方面都可以使用用户自定义规则，如制定排序规则，制定查找规则，在修改或转换数据时使用自定义的重复数
据判断规则等。下面的例子显示了在数据转移个过程中使用用户自定义规则：

```

/*
 * transform1.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/cset.h>
#include <cstl/calgorithm.h>

```

```

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

static void _square(const void* cpv_input, void* pv_output)
{
    *(int*)pv_output = *(int*)cpv_input * *(int*)cpv_input;
}

int main(int argc, char* argv[])
{
    set_t* pset_coll1 = create_set(int);
    vector_t* pvec_coll2 = create_vector(int);
    int i = 0;

    if(pset_coll1 == NULL || pvec_coll2 == NULL)
    {
        return -1;
    }

    set_init(pset_coll1);
    vector_init(pvec_coll2);

    for(i = 1; i <= 9; ++i)
    {
        set_insert(pset_coll1, i);
    }

    printf("initialized: ");
    algo_for_each(set_begin(pset_coll1), set_end(pset_coll1), _print);
    printf("\n");

    /* transform each element from coll1 to coll2 */
    vector_resize(pvec_coll2, set_size(pset_coll1));
    algo_transform(set_begin(pset_coll1), set_end(pset_coll1),
        vector_begin(pvec_coll2), _square);

    printf("squared: ");
    algo_for_each(vector_begin(pvec_coll2), vector_end(pvec_coll2), _print);
    printf("\n");
}

```

```

set_destroy(pset_coll1);
vector_destroy(pvec_coll2);

return 0;
}

```

上面的例子中_square 为了在从 pset_coll1 到 pvec_coll2 的数据传送过程中产生整数的平方。

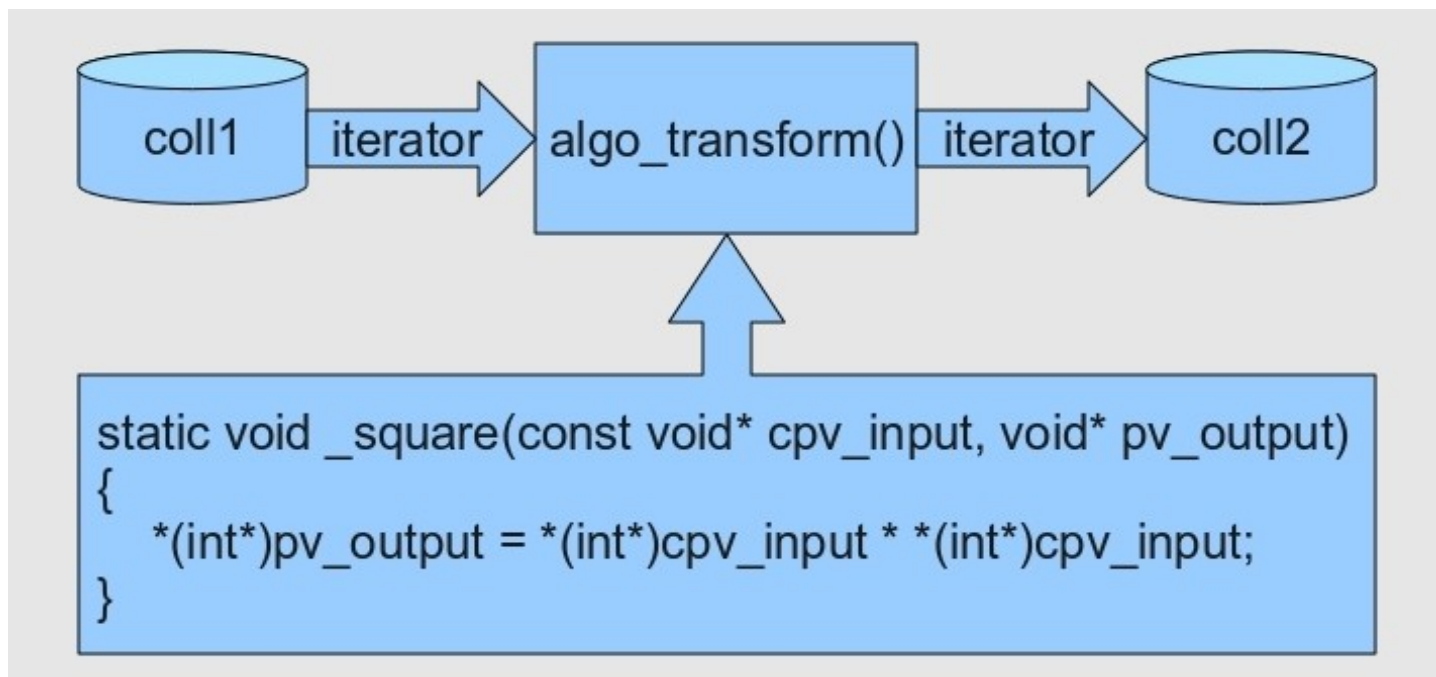


图 2.9 algo_transform()的执行方式

输出结果:

initialized: 1 2 3 4 5 6 7 8 9

squared: 1 4 9 16 25 36 49 64 81

7. 自定义规则，函数和谓词

观察上面两个例子所使用的自定义规则，你会发现这些规则的形式很奇怪，不论是_print 还是_square 它们都是这样的形式：

```
void xxxx(const void* cpv_input, void* pv_output) ;
```

这就是 libcstl 规定的自定义规则的函数形式，只有这样形式的规则才会被算法接受。将规则限定为这种形式是为了能够处理任何的数据类型，无论什么数据类型都可以采用这样的方式来制定规则。

除了上面的形式之外，还有一种形式的规则也是被算法接受的：

```
void xxxx(const void* cpv_first, const void* cpv_second, void* pv_output) ;
```

只有这两种形式被算法接受，我们把前者叫做一元函数，把后者叫做二元函数。它们是不同的规则形式不能混淆，有的函数只接受一元函数作为规则，有的函数只接受二元函数作为规则。

对于一元函数和二元函数来说最后一个参数都是输出，前面的参数是输入。

算法在使用这些自定义规则的时候，大部分的规则的输出参数都是 bool_t 类型的 (bool_t 是辅助类型在第三章介

绍)，还有些输出参数不是 `bool_t` 类型的如上面两个例子中的自定义规则 `_print` 和 `_square`。我们把输出参数是 `bool_t` 类型的函数叫做谓词。因此谓词也有一元谓词和二元谓词。下面的这些例子展示了如何定义和使用谓词：

下面是一个使用一元谓词判断素数的例子：

```
/*
 * prime1.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <stdlib.h>
#include <cstl/clist.h>
#include <cstl/calgorithm.h>

static void _is_prime(const void* cpv_input, void* pv_output)
{
    int number = abs(*(int*)cpv_input);
    int divisor = 0;

    /* 0 and 1 are prime number */
    if(number == 0 || number == 1)
    {
        *(bool_t*)pv_output = true;
        return;
    }

    for(divisor = number / 2; number % divisor != 0; --divisor)
    {
        continue;
    }

    *(bool_t*)pv_output = divisor == 1 ? true : false;
}

int main(int argc, char* argv[])
{
    list_t* plist_coll = create_list(int);
    list_iterator_t it_pos;
    int i = 0;

    if(plist_coll == NULL)
    {
        return -1;
    }
}
```

```

list_init(plist_coll);

for(i = 24; i <= 30; ++i)
{
    list_push_back(plist_coll, i);
}

/* search for prime number */
it_pos = algo_find_if(list_begin(plist_coll), list_end(plist_coll), _is_prime);
if(iterator_equal(it_pos, list_end(plist_coll)))
{
    printf("no prime number found.\n");
}
else
{
    printf("%d is the first prime.\n", *(int*)iterator_get_pointer(it_pos));
}

list_destroy(plist_coll);

return 0;
}

```

在这个例子中 `algo_find_if()` 算法在制定的范围内查找第一个使一元谓词获得 `true` 输出的数据，这个一元谓词就是 `_is_prime` 函数。在一元谓词 `_is_prime` 中计算数据是否为素数，如果是将输出 `pv_output` 设置为 `true` 否则设置为 `false`。输出结果：

29 is the first prime.

下面再来看一个使用二元谓词的例子：

```

/*
 * sort1.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <string.h>
#include <cstl/cdeque.h>
#include <cstl/calgorithm.h>

typedef struct _tagperson
{
    char s_firstname[21];

```



```

    char s_lastname[21];
}person_t;

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%s.%s ",
        ((person_t*)cpv_input)->s_firstname,
        ((person_t*)cpv_input)->s_lastname);
}

static void _person_sort_criterion(
    const void* cpv_first, const void* cpv_second, void* pv_output)
{
    person_t* pt_first = (person_t*)cpv_first;
    person_t* pt_second = (person_t*)cpv_second;
    int n_result1 = strncmp(pt_first->s_firstname, pt_second->s_firstname, 21);
    int n_result2 = strncmp(pt_first->s_lastname, pt_second->s_lastname, 21);

    if(n_result1 < 0 || (n_result1 == 0 && n_result2 < 0))
    {
        *(bool_t*)pv_output = true;
    }
    else
    {
        *(bool_t*)pv_output = false;
    }
}

int main(int argc, char* argv[])
{
    deque_t* pdeq_coll = NULL;
    person_t t_person;

    type_register(person_t, NULL, NULL, NULL, NULL);
    pdeq_coll = create_deque(person_t);
    if(pdeq_coll == NULL)
    {
        return -1;
    }

    deque_init(pdeq_coll);

    memset(t_person.s_firstname, '\\0', 21);

```

```

memset(t_person.s_lastname, '\0', 21);
strcpy(t_person.s_firstname, "Jonh");
strcpy(t_person.s_lastname, "right");
deque_push_back(pdeq_coll, &t_person);
memset(t_person.s_firstname, '\0', 21);
memset(t_person.s_lastname, '\0', 21);
strcpy(t_person.s_firstname, "Bill");
strcpy(t_person.s_lastname, "killer");
deque_push_back(pdeq_coll, &t_person);
memset(t_person.s_firstname, '\0', 21);
memset(t_person.s_lastname, '\0', 21);
strcpy(t_person.s_firstname, "Jonh");
strcpy(t_person.s_lastname, "sound");
deque_push_back(pdeq_coll, &t_person);
memset(t_person.s_firstname, '\0', 21);
memset(t_person.s_lastname, '\0', 21);
strcpy(t_person.s_firstname, "Bin");
strcpy(t_person.s_lastname, "lee");
deque_push_back(pdeq_coll, &t_person);
memset(t_person.s_firstname, '\0', 21);
memset(t_person.s_lastname, '\0', 21);
strcpy(t_person.s_firstname, "Lee");
strcpy(t_person.s_lastname, "bird");
deque_push_back(pdeq_coll, &t_person);

algo_for_each(deque_begin(pdeq_coll), deque_end(pdeq_coll), _print);
printf("\n");

algo_sort_if(deque_begin(pdeq_coll), deque_end(pdeq_coll),
    _person_sort_criterion);

algo_for_each(deque_begin(pdeq_coll), deque_end(pdeq_coll), _print);
printf("\n");

deque_destroy(pdeq_coll);

return 0;
}

```

输出的结果是：

Jonh.right Bill.killer Jonh.sound Bin.lee Lee.bird

Bill.killer Bin.lee Jonh.right Jonh.sound Lee.bird

可以看出是按照_person_sort_criterion()制定的规则排序的。

8. libcstl 函数

通过前面的例子我们知道可以通过自定义规则来扩展或者改变算法的行为，从而增强算法的能力和扩展性。那是不是只要使用自定义规则我们就要自己去编写新的规则呢？答案是否定的，因为 libcstl 库提供了大量的自定义规则，在这里我们把这些 libcstl 库预定义以的自定义规则叫做 libcstl 函数，这些函数中一大部分是谓词。它们的形式都符合上面我们看到的算法接受的函数形式。要使用这些预定义的函数必须包含头文件：

```
#include <cstl/cfunctional.h>
```

下面是一个使用了 libcstl 预定义的二元函数的例子：

```
/*
 * sort2.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cdeque.h>
#include <cstl/calgorithm.h>
#include <cstl/cfunctional.h>

int main(int argc, char* argv[])
{
    deque_t* pdeq_coll = create_deque(int);
    deque_iterator_t it_pos;

    if(pdeq_coll == NULL)
    {
        return -1;
    }

    deque_init(pdeq_coll);

    deque_push_back(pdeq_coll, 34);
    deque_push_back(pdeq_coll, 22);
    deque_push_back(pdeq_coll, 90);
    deque_push_back(pdeq_coll, 51);
    deque_push_back(pdeq_coll, 11);
    deque_push_back(pdeq_coll, 47);
    deque_push_back(pdeq_coll, 33);

    for(it_pos = deque_begin(pdeq_coll);
        !iterator_equal(it_pos, deque_end(pdeq_coll));
        it_pos = iterator_next(it_pos))
```

```

{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");

algo_sort(deque_begin(pdeq_coll), deque_end(pdeq_coll));
for(it_pos = deque_begin(pdeq_coll);
    !iterator_equal(it_pos, deque_end(pdeq_coll));
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");

algo_sort_if(deque_begin(pdeq_coll), deque_end(pdeq_coll), fun_greater_int);
for(it_pos = deque_begin(pdeq_coll);
    !iterator_equal(it_pos, deque_end(pdeq_coll));
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");

deque_destroy(pdeq_coll);

return 0;
}

```

在向 pdeq_coll 中插入了数据后使用 algo_sort() 排序，这个算法其实是 algo_sort_if() 的使用默认关系二元函数的缩写版本，如果容器中的数据是 int 类型那么这个默认的关系二元函数就是 fun_less_int()。最后使用了 algo_sort_if() 使用 fun_greater_int() 作为排序规则从大到小排序，所以结果：

```

34 22 90 51 11 47 33
11 22 33 34 47 51 90
90 51 47 34 33 22 11

```

第五节 libcstl 容器的使用过程

1. libcstl 容器的使用过程

libcstl 提供定义的一系列的新类型，包括容器，容器适配器，迭代器，工具类型，函数类型。其中容器（序列容器

vector_t, deque_t, list_t, slist_t, 关联容器 set_t, map_t, multiset_t, multimap_t, hash_set_t, hash_map_t, hash_multiset_t, 容器适配器 stack_t, queue_t, priority_queue_t, 字符串 string_t) 和工具类型中的 pair_t 在使用的时候都要遵循以下的过程:

首先要创建一个容器, 然后初始化, 使用完成后要销毁。下面以 vector_t 为例:

- 创建

使用 vector_t 之前要根据容器中存储的数据的类型创建容器, 如果要存储 int 类型的数据调用 create_vector() 函数:

```
vector_t* pvec_coll = create_vector(int);
```

如果创建成功 pvec_coll 就是指向新创建的保存 int 类型的 vector_t 容器, 如果创建过程失败那么 pvec_coll 就等于 NULL。所以在创建一个类型之后要判断是否创建成功。每一个需要创建的类型都有一个以 create_ 为前缀的函数, 具体请参考《The libcstl Library Reference Manual》。

- 初始化

当 vector_t 被创建后要对容器进行初始化, 这样才能使用, 如果使用未初始化的容器, 行为是未定义的, 因为有很多事情要在初始化的过程中做。对于一个 libcstl 容器来说往往有多种初始化函数, 这些初始化函数可以以不同的方式初始化这个 libcstl 容器。

例如 vector_t 有 vector_init(), vector_init_n(), vector_init_elem(), vector_init_copy(), vector_init_copy_range() 5 个初始化函数, 它们根据不同的方式对 vector_t 容器进行初始化, 在创建 vector_t 容器后可以调用其中的任意一个, 但是只能调用一次, 如果对一个容器类型初始化多次那么程序的行为是未定义的:

```
vector_init_n(pvec_coll, 100);
```

初始化 vector_t 后容器中有 100 个元素, 每个元素的值都是 0。

- 销毁

vector_t 使用完毕后一定要销毁, 如果未销毁那么容器所占的内存就不会被释放。每一个容器都有销毁函数如:

```
vector_destroy(pvec_coll);
```

2. 数据类型的使用

libcstl 2.0 版本对数据类型的处理做了很大改进, 现在能够处理任何数据类型, libcstl 将数据类型分为三类:

- C 语言内建类型

C 语言支持的基本类型如, int, double, long, char 等, C 字符串也属于这一类。

- libcstl 内建类型

libcstl 库内部的容器类型, 迭代器类型等等。

- 用户自定义类型

用户自己定义的结构, 联合, 枚举或者重定义的类型。

对于前两种类型可以在创建容器中直接使用如:

```
vector_t* pvec_coll = create_vector(int);
```

```
vector_t* pvec_coll = create_vector(double);
```

```
vector_t* pvec_coll = create_vector(char*);
```

```
vector_t* pvec_coll = create_vector(list_t<int>);
```

等等 (这里创建保存 libcstl 内建类型的容器时类型的描述有些特别, 我会在第十章中详细介绍)。但是对于用户自定义类型这样的方式就会创建失败, 如有一种用户自定义类型 abc_t, 现在要创建一个保存 abc_t 类型的 vector_t 容器:

```
vector_t* pvec_coll = create_vector(abc_t);
```

直接这样创建 `pvec_coll` 一定为 `NULL`，因为 `libcstl` 库并不知道关于 `abc_t` 类型的任何信息。要使用这种新类型就必须在使用它之前注册这种类型：

```
type_register(abc_t, abc_init, abc_copy, abc_less, abc_destroy);
```

注册之后 `libcstl` 就知道了该类型的信息如初始化方法 `abc_init`，拷贝方法 `abc_copy`，比较规则 `abc_less`，销毁方法 `abc_destroy`，这样就可以使用这个类型了而且是“一次注册到处使用”。具体在第十章中介绍类型机制。

第三章 `libcstl` 工具类型

为了使用方便，`libcstl` 提供了很多实用的小工具类型。使用这些小工具的时候几乎没有什么开销，有的甚至不需要为了使用它特意的去包含特殊的头文件。这些工具类型包括 `libcstl` 定义的布尔类型 `bool_t`，用于保存对数据的 `pair_t`，用于表示数据区间的 `range_t`。

第一节 `bool_t`

标准 C 语言中是没有布尔类型的，为了使用方便和能够清楚的表达布尔的语义，`libcstl` 引入了一个新的工具类型 `bool_t` 来表示布尔值，同时定义了 `true`，`TRUE`，`false`，`FLASE`。使用 `bool_t` 类型不需要包含特殊的头文件，包含任何一个 `libcstl` 头文件就可以使用 `bool_t` 了。

第二节 `pair_t`

在前面的章节中已经知道映射，多重映射等是以键/值对这样的形式保存数据的，但是实际存储在容器中的值却是 `pair_t` 类型，有它来管理键/值对。不只是键/值对，所有成对的数据可以使用 `pair_t` 类型来描述。`pair_t` 不是容器所以它不能够保存多对数据，一个 `pair_t` 只能保存一对数据。使用 `pair_t` 要求包含头文件：

```
#include <cstl/cutility.h>
```

但是如果程序已经包含了头文件

```
#include <cstl/cmap.h> 或者 #include <cstl/chash_map.h>
```

就不必包含 `<cstl/cutility.h>` 头文件了，这是因为映射容器要用到 `pair_t`，在它的头文件中已经包含了 `<cstl/cutility.h>` 头文件了。

1. `pair_t` 的使用过程

虽然 `pair_t` 不是容器类型，但是使用的过程要与容器类型的使用过程相同（参考第二章. 第五节），也要首先创建，成功后要初始化，然后使用，使用后要销毁。

create_pair	创建 pair_t 类型。
pair_init	使用默认数据初始化 pair_t 类型。
pair_init_copy	使用 pair_t 类型来初始化一个 pair_t 类型。
pair_init_lem	使用指定数据初始化 pair_t 类型。
pair_destroy	销毁 pair_t 类型。

上面的表格列出了 pair_t 创建，初始化和销毁的函数(本书中只类出了讲解相关的函数并且没有具体参数，全面俩解各个操作函数和参数的情况请参考《The libcstl Library Reference Manual》)。

2. pair_t 的主要操作

pair_t 是表示一对数据的，所以通过 pair_t 类型我们应该能够获得和设置这两个数据。依据这样的要求 pair_t 提供了如下操作函数：

pair_first	获得指向第一个数据的指针。
pair_second	获得指向第二个数据的指针。
pair_make	设置 pair_t 中的数据。

通过上面的操作函数我们就能够自由的使用 pair_t 中的数据了。

为了方便两个 pair_t 之间的数据拷贝除了提供在初始化的时候拷贝外还通过了赋值操作函数：

pair_assign	使用源 pair_t 类型为目的 pair_t 类型赋值。
-------------	-------------------------------

除了这些以外 pair_t 还提供了关系操作函数：

pair_equal	判断两个 pair_t 类型是否相等。
pair_not_equal	判断两个 pair_t 类型是否不等。
pair_less	判断第一个 pair_t 类型是否小于第二个 pair_t 类型。
pair_less_equal	判断第一个 pair_t 类型是否小于等于第二个 pair_t 类型。
pair_greater	判断第一个 pair_t 类型是否大于第二个 pair_t 类型。
pair_greater_equal	判断第一个 pair_t 类型是否大于等于第二个 pair_t 类型。

pair_t 的比较过程都是首先比较第一个数据，当第一个数据是大于关系的时候整个关系操作就是大于关系，小于也如此，只有等于的时候再比较第二个数据，只有两个数据都相等的时候两个 pair_t 才相等。

3. pair_t 的应用实例

上面列举了有关 pair_t 的操作函数，下面用例子来演示以下如何使用 pair_t:

```
/*
 * pair1.c
```

```

* compile with : -lcstl
*/

#include <stdio.h>
#include <cstl/cutility.h>

int main(int argc, char* argv[])
{
    pair_t* ppair_p1 = create_pair(int, double);
    pair_t* ppair_p2 = create_pair(int, double);
    pair_t* ppair_p3 = create_pair(int, double);

    if(ppair_p1 == NULL || ppair_p2 == NULL || ppair_p3 == NULL)
    {
        return -1;
    }

    pair_init(ppair_p1);
    pair_init_elem(ppair_p2, 10, 1.1e-2);
    pair_init_copy(ppair_p3, ppair_p2);

    printf("Original pair:\n");
    printf("p1(%d, %lf)\n",
        *(int*)pair_first(ppair_p1),
        *(double*)pair_second(ppair_p1));
    printf("p2(%d, %lf)\n",
        *(int*)pair_first(ppair_p2),
        *(double*)pair_second(ppair_p2));
    printf("p3(%d, %lf)\n",
        *(int*)pair_first(ppair_p3),
        *(double*)pair_second(ppair_p3));

    pair_make(ppair_p1, 10, 2.222);
    pair_assign(ppair_p3, ppair_p1);
    printf("After modifying:\n");
    printf("p1(%d, %lf)\n",
        *(int*)pair_first(ppair_p1),
        *(double*)pair_second(ppair_p1));
    printf("p2(%d, %lf)\n",
        *(int*)pair_first(ppair_p2),
        *(double*)pair_second(ppair_p2));
    printf("p3(%d, %lf)\n",
        *(int*)pair_first(ppair_p3),

```



```

        *(double*)pair_second(ppair_p3));

printf("Compare:\n");
if(pair_equal(ppair_p1, ppair_p3))
{
    printf("p1(%d, %lf) == p3(%d, %lf)\n",
        *(int*)pair_first(ppair_p1),
        *(double*)pair_second(ppair_p1),
        *(int*)pair_first(ppair_p3),
        *(double*)pair_second(ppair_p3));
}
else
{
    printf("p1(%d, %lf) != p3(%d, %lf)\n",
        *(int*)pair_first(ppair_p1),
        *(double*)pair_second(ppair_p1),
        *(int*)pair_first(ppair_p3),
        *(double*)pair_second(ppair_p3));
}
if(pair_not_equal(ppair_p1, ppair_p2))
{
    printf("p1(%d, %lf) != p2(%d, %lf)\n",
        *(int*)pair_first(ppair_p1),
        *(double*)pair_second(ppair_p1),
        *(int*)pair_first(ppair_p2),
        *(double*)pair_second(ppair_p2));
}
else
{
    printf("p1(%d, %lf) == p2(%d, %lf)\n",
        *(int*)pair_first(ppair_p1),
        *(double*)pair_second(ppair_p1),
        *(int*)pair_first(ppair_p2),
        *(double*)pair_second(ppair_p2));
}

pair_make(ppair_p1, 22, 35.2);
pair_make(ppair_p2, 62, 35.2);
pair_make(ppair_p3, 22, 70.0);
if(pair_less(ppair_p1, ppair_p2))
{
    printf("p1(%d, %lf) < p2(%d, %lf)\n",
        *(int*)pair_first(ppair_p1),

```

```

        *(double*)pair_second(ppair_p1),
        *(int*)pair_first(ppair_p2),
        *(double*)pair_second(ppair_p2));
    }
    else
    {
        printf("p1(%d, %lf) >= p2(%d, %lf)\n",
            *(int*)pair_first(ppair_p1),
            *(double*)pair_second(ppair_p1),
            *(int*)pair_first(ppair_p2),
            *(double*)pair_second(ppair_p2));
    }
    if(pair_less(ppair_p1, ppair_p3))
    {
        printf("p1(%d, %lf) < p3(%d, %lf)\n",
            *(int*)pair_first(ppair_p1),
            *(double*)pair_second(ppair_p1),
            *(int*)pair_first(ppair_p3),
            *(double*)pair_second(ppair_p3));
    }
    else
    {
        printf("p1(%d, %lf) >= p3(%d, %lf)\n",
            *(int*)pair_first(ppair_p1),
            *(double*)pair_second(ppair_p1),
            *(int*)pair_first(ppair_p3),
            *(double*)pair_second(ppair_p3));
    }

    pair_destroy(ppair_p1);
    pair_destroy(ppair_p2);
    pair_destroy(ppair_p3);

    return 0;
}

```

这个例子的输出结果:

Original pair:

p1(0, 0.000000)

p2(10, 0.011000)

p3(10, 0.011000)

After modifying:

p1(10, 2.222000)

p2(10, 0.011000)

```
p3(10, 2.222000)
Compare:
p1(10, 2.222000) == p3(10, 2.222000)
p1(10, 2.222000) != p2(10, 0.011000)
p1(22, 35.200000) < p2(62, 35.200000)
p1(22, 35.200000) < p3(22, 70.000000)
```

第三节 range_t

有些函数的返回值是一个数据区间，例如关联容器的 `equal_range` 操作函数，在 `libcstl` 1.0 版本时使用 `pair_t` 类型来表示数据区间，这样用户在获得这个数据区间并使用之后要手动的调用 `pair_destroy()` 函数来销毁 `pair_t` 类型，这样做使用很不方便，况且这时候 `pair_t` 中保存的是两个迭代器类型，所以主要的系统资源还是 `pair_t` 本身占用的。为了解决使用不便的问题 `libcstl` 2.0 版本引入了 `range_t` 类型，使用 `range_t` 类型来表示数据区间就不用再有上面的烦恼了。这个类型本身就是一个结构体，用两个成员来表示数据区间的界限迭代器。使用之后可以直接丢弃，没有手动调用释放资源的烦恼了。

`range_t` 的两个成员分别表示数据区间的上下界限：

<code>it_begin</code>	表示数据区间的开始位置的迭代器。
<code>it_end</code>	表示数据区间的末尾位置的迭代器。

这样通过 `range_t` 的两个成员就可以表示数据区间 `[it_begin, it_end)`。使用 `range_t` 于使用 `bool_t` 一样无需包含特殊的头文件，只要包含任意的 `libcstl` 头文件就能够使用它了。

第四章 libcstl 容器

在接下来的一章中我们将详细讨论 `libcstl` 各个容器的结构，特点，操作性能以及各种操作如何使用。在本章的最后还对比各个容器的优缺点，以便用户按照需求适当的选择容器类型。

第一节 容器的特点和共同的操作

在详细讨论各个容器之前我们先来讨论下一容器的共同特点和共同的操作。

1. 容器的共同特点

各种类型的容器它们都有如下的共同特点：

- 容器中保存的是数据的拷贝

容器中保存的都是数据的拷贝，而不是指向实际数据的指针，这样就要求数据本身都是有明确的拷贝操作函数的。对于 C 内建类型和 `libcstl` 内建类型库本身提供了这些类型的拷贝操作，但是对于用户自定义类型就要求在注册这个类型的时候提供拷贝操作函数，如果没有提供拷贝操作函数，那么 `libcstl` 库会提供一个默认的拷贝操作，这样的话有些自定义类型的拷贝操作就不准确了。

- 数据能够初始化和销毁

这是因为在容器中插入新数据时要求对数据进行初始化，在删除数据的时候要对数据进行销毁。因为对于一个特定的数据类型，只有定义数据类型的人最清除数据内部的情况，`libcstl` 库本身并不知道数据内部的事情，要求有初始化和销毁操作是为了保证数据能够在被创建和销毁的过程中做很多操作。有可能数据在初始化的时候要申请资源在销毁的时候要释放资源。

- 数据的比较操作

保存在容器中的数据都要求有比较规则，无论是保存在序列容器还是关联容器中的数据。因为在容器进行比较和对数据进行排序或者是将容器中的数据应用到算法的时候都要对数据进行比较。这个比较规则通常是小于操作。

- 操作函数并非安全

容器的操作函数并非安全，用户要保证传入操作函数的参数是合法并且有效的，非法或者是无效的参数导致操作函数的行为未定义。

2. 容器的共同操作

无论是那种容器类型都有如下这几种操作：

- 创建容器

容器在使用之前都要创建，创建是为了确定容器中保存数据的类型。

- 初始化和销毁

初始化和销毁是为了申请和释放容器占用的资源，同时初始化和销毁还对容器中的数据进行了初始化和销毁。

- 与数据数量有关的操作

每个容器都有下面三个与数据数量有关的操作函数。

`xxxx_size()`

返回容器中数据的数量。

`xxxx_empty()`

判断容器是否为空。

`xxxx_max_size()`

返回容器可以容纳的数据的最大数量，这个值是一个与系统相关的常量，同时它还与保存在容器中的数据类型相关。

- 比较操作

比较操作实现的是容器之间的比较，包括等于，不等于，小于，小于等于，大于，大于等于。这些操作都要求相比的两个容器保存的是同种类型的数据。同时只有容器中的数据对应相等并且个数相同的情况下才相等，对于等于和不等两个操作函数，数据类型不同也认为是不等。

- 赋值和交换

赋值是使用现有的数据将容器中原有的数据替换掉，交换就是交换两个容器中的内容。

第二节 `vector_t`

`vector_t` 容器类似一个动态的数组，但是它可以根据需要动态的生长。

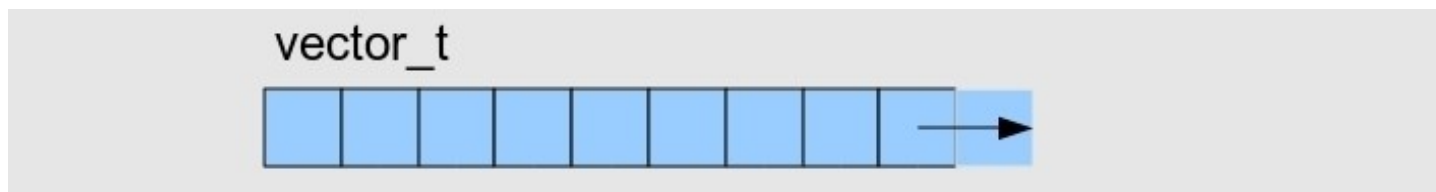


图 4.1 `vector_t` 的结构

要使用 `vector_t` 必须包含头文件 `<cstdlib/cvector.h>`

1. `vector_t` 的能力

`vector_t` 将数据拷贝到内部的动态数组中，数据的顺序与插入的顺序有关与数据本身无关。`vector_t` 是随机访问容器，可以使用下标随机访问容器中的数据，`vector_t` 的迭代器也是随即访问迭代器。任何 `libcstl` 算法都可以作用于 `vector_t` 容器。

在 `vector_t` 末尾插入或删除数据效率很高，在中间或开头插入或删除数据效率很低，因为这样会移动大量数据。

- 尺寸和容量

为数据分配更多的内存往往比分配正好的内存有更好的效率。为了正确和有效的使用 `vector_t` 必须理解尺寸和容量之间的关系。`vector_t` 提供了一个函数 `vector_size()` 获得容器的尺寸，也就是实际保存的数据的个数，同时提供了一个 `vector_capacity()` 函数获得容器的容量，容量就是在 `vector_t` 不重新分配内存的情况下 `vector_t` 中能够保存的数据的总量。如果要保存的数据的个数超过的容器的容量，那么容器就会重新分配内存来容纳下这些数据。

重现分配内存后容器以前的迭代器就失效了，同时重新分配内存也很耗时。为了避免重新分配内存，使用 `vector_reserve()` 函数来指定容量的大小，或者使用 `vector_init_n()` 函数来指定容其中数据的个数，或者使用 `vector_resize()` 函数来改变容器中数据的个数。

我们可以使用 `vector_reserve()` 来增大 `vector_t` 的容量，但是并不能使用 `vector_reserve()` 来缩小 `vector_t` 的容量，及时传递个 `vector_reserve()` 的参数小于当前 `vector_t` 的容量，那么这个容量也是不会改变的。

2. `vector_t` 的操作

- 创建，初始化和销毁操作

上面已经提到了创建是为了确定 `vector_t` 中保存的数据类型，初始化是为 `vector_t` 保存数据申请资源，同时有多种方式初始化一个 `vector_t`，如初始化一个空的 `vector_t`，或者使用多个数据初始化一个 `vector_t` 等等，销毁是释放 `vector_t` 申请的资源。

<code>create_vector</code>	创建一个 <code>vector_t</code> 容器。
<code>vector_init</code>	初始化一个空的 <code>vector_t</code> 容器。
<code>vector_init_n</code>	初始化一个 <code>vector_t</code> 容器，初始化后容器中包含多个默认数据。
<code>vector_init_elem</code>	初始化一个 <code>vector_t</code> 容器，初始化后容器中包含多个指定的数据。
<code>vector_init_copy</code>	初始化一个 <code>vector_t</code> 容器，初始化后的容器中的内容于指定的 <code>vector_t</code> 容器的内容相同。
<code>vector_init_copy_range</code>	初始化一个 <code>vector_t</code> 容器，初始化后的容器中的数据是用户指定的数据区间中的数据。
<code>vector_destroy</code>	销毁一个 <code>vector_t</code> 容器，释放 <code>vector_t</code> 容器占用的资源。

`vector_init_copy_range()`中要求使用一个数据区间来初始化 `vector_t`，这个区间必须属于另一个 `vector_t`，`libcstl` 不支持使用不同种类的数据区间进行初始化。

● 非质变操作

质变操作是指修改容器数据，非质变操作就是指操作后容器种的数据内容和数量不会改变。`vector_t` 容器提供了两类非质变的操作，分别是与容器中数据的数量相关的操作：

<code>vector_size</code>	返回容器中数据的数量。
<code>vector_empty</code>	测试容器是否为空。
<code>vector_max_size</code>	容器能够容纳数据的最大数量。
<code>vector_capacity</code>	容器在不重新分配内存是能够容纳的数据的数量。
<code>vector_reserve</code>	调整容器的容量。

这些函数中前面两个是与容器中数据的实际数量相关的，第三个函数的返回值是一个与系统和保存数据的类型相关的常数，后两个是与容器容量相关的函数。

还有一类非质变操作函数就是比较函数：

<code>vector_equal</code>	测试两个 <code>vector_t</code> 是否相等。
<code>vector_not_equal</code>	测试两个 <code>vector_t</code> 是否不等。
<code>vector_less</code>	测试第一个 <code>vector_t</code> 是否小于第二个 <code>vector_t</code> 。
<code>vector_less_equal</code>	测试第一个 <code>vector_t</code> 是否小于等于第二个 <code>vector_t</code> 。
<code>vector_greater</code>	测试第一个 <code>vector_t</code> 是否大于第二个 <code>vector_t</code> 。
<code>vector_greater_equal</code>	测试第一个 <code>vector_t</code> 是否大于等于第二个 <code>vector_t</code> 。

● 赋值和交换

<code>vector_assign</code>	使用 <code>vector_t</code> 类型为当前的 <code>vector_t</code> 赋值。
<code>vector_assign_elem</code>	使用指定的数据为 <code>vector_t</code> 赋值。
<code>vector_assign_range</code>	使用指定的数据区间为 <code>vector_t</code> 赋值。
<code>vector_swap</code>	将两个 <code>vector_t</code> 类型的内容交换。

`vector_assign_range()`也要求使用属于 `vector_t` 的数据区间。`vector_swap()`要求两个 `vector_t` 保存的数据类型是相同的否则导致函数的行为未定义。

- 数据访问

数据的访问是直接对数据进行操作，这些函数返回的都是 `void*` 可以通过对它进行强制转换来得到指向实际数据类型的指针 (用户是最清楚当前容器中保存的数据类型是什么的)。

<code>vector_at</code>	通过下标对 <code>vector_t</code> 中的数据进行随机访问。
<code>vector_front</code>	访问 <code>vector_t</code> 中的第一个数据。
<code>vector_back</code>	访问 <code>vector_t</code> 中的最后一个数据。

对数据进行访问的时候要确保待访问的数据是存在的，访问不存在的数据结果是未定义的，例如使用了超出范围的下标或者要访问空 `vector_t` 中的第一个或者最后一个数据。

- 与迭代器相关的操作函数

<code>vector_begin</code>	返回指向第一个数据的迭代器。
<code>vector_end</code>	返回指向容器末尾的迭代器。

`vector_t` 的迭代器是随机访问迭代器，同时 `vector_t` 迭代器只有在下面这两种情况下才会失效：第一当在迭代器所指数据的前面出入了数据，第二整个容器的内存重新分配。

插入和删除

调用插入和删除函数是一定要保证迭代器或数据区间是有效的，否则操作函数行为未定义。插入或删除操作在下面的情况下更快一些：

- 在末尾插入或删除数据。
- 容量足够大。
- 一次操作比多次操作更快。

在插入或者删除数据后，指向插入或者删除点后面的数据的迭代器都失效，如果插入数据导致内存重新分配，那么所有的迭代器都失效。

<code>vector_insert</code>	向 <code>vector_t</code> 中插入指定数据。
<code>vector_insert_n</code>	向 <code>vector_t</code> 中插入多个指定的数据。
<code>vector_insert_range</code>	向 <code>vector_t</code> 中插入指定数据区间的数据。
<code>vector_push_back</code>	向 <code>vector_t</code> 末尾添加一个数据。
<code>vector_pop_back</code>	删除 <code>vector_t</code> 的随后一个数据。
<code>vector_erase</code>	删除 <code>vector_t</code> 中指定位置的数据。
<code>vector_erase_range</code>	删除 <code>vector_t</code> 中指定数据区间的数据。
<code>vector_clear</code>	删除 <code>vector_t</code> 中的所有数据。
<code>vector_resize</code>	改变 <code>vector_t</code> 中数据的数量，如果数据的数量变大，用默认数据填充。
<code>vector_resize_elem</code>	改变 <code>vector_t</code> 中数据的数量，如果数据的数量变大，用指定的数据填充。

3. 将 vector_t 作为数组使用

vector_t 内部是连续的内存空间，所以可以当作数组使用，不过使用时一定好注意越界的问题，下面是把 vector_t 当作数组使用的例子：

```
/*
 * vector2.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <string.h>
#include <cstl/cvector.h>

int main(int argc, char* argv[])
{
    vector_t* pvec_coll = create_vector(char);

    if(pvec_coll == NULL)
    {
        return -1;
    }

    vector_init(pvec_coll);
    vector_resize_elem(pvec_coll, 30, '\0');

    strcpy((char*)vector_at(pvec_coll, 0), "Hello World!");
    printf("%s\n", (char*)vector_front(pvec_coll));

    vector_destroy(pvec_coll);

    return 0;
}
```

下面这句将 vector_t 视为数据并将数据拷贝到 vector_t 中

```
strcpy((char*)vector_at(pvec_coll, 0), "Hello World!");
```

结果是：

Hello World!

4. vector_t 的使用实例

下面是 vector_t 的一个简单的使用实例：

```
/*
 * vector3.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%s ", (char*)cpv_input);
}

int main(int argc, char* argv[])
{
    vector_t* pvec_sentence = create_vector(char*);
    vector_iterator_t it_pos;

    if(pvec_sentence == NULL)
    {
        return -1;
    }

    vector_init(pvec_sentence);

    /* reserve memory for five elements to avoid reallocation */
    vector_reserve(pvec_sentence, 5);

    /* append some element */
    vector_push_back(pvec_sentence, "Hello,");
    vector_push_back(pvec_sentence, "how");
    vector_push_back(pvec_sentence, "are");
    vector_push_back(pvec_sentence, "you");
    vector_push_back(pvec_sentence, "?");

    /* print all elements with space */
    algo_for_each(vector_begin(pvec_sentence), vector_end(pvec_sentence), _print);
}
```

```

printf("\n");

/* print technical data */
printf("max size: %u\n", vector_max_size(pvec_sentence));
printf("size      : %u\n", vector_size(pvec_sentence));
printf("capacity: %u\n", vector_capacity(pvec_sentence));

/* swap the second and fourth elements */
algo_iter_swap(iterator_next(vector_begin(pvec_sentence)),
               iterator_next_n(vector_begin(pvec_sentence), 3));

/* insert "always" before "?" */
it_pos = algo_find(vector_begin(pvec_sentence), vector_end(pvec_sentence), "?");
vector_insert(pvec_sentence, it_pos, "always");

/* print all elements with space */
algo_for_each(vector_begin(pvec_sentence), vector_end(pvec_sentence), _print);
printf("\n");

/* print technical data again */
printf("max size: %u\n", vector_max_size(pvec_sentence));
printf("size      : %u\n", vector_size(pvec_sentence));
printf("capacity: %u\n", vector_capacity(pvec_sentence));

vector_destroy(pvec_sentence);

return 0;
}

```

程序的输出可能是这样的：

Hello, how are you ?

max size: 11930464

size : 5

capacity: 5

Hello, you are how always ?

max size: 11930464

size : 6

capacity: 7

为什么可能呢？因为 max size 的值是与系统相关的。

第三节 deque_t

deque_t 与 vector_t 十分相似，同样是使用动态数组管理数据，提供随机访问，大部分操作函数都一样。不同点是 deque_t 的开始和结尾两端都是开放的，所以在开始和结尾插入或删除数据都是高效的。

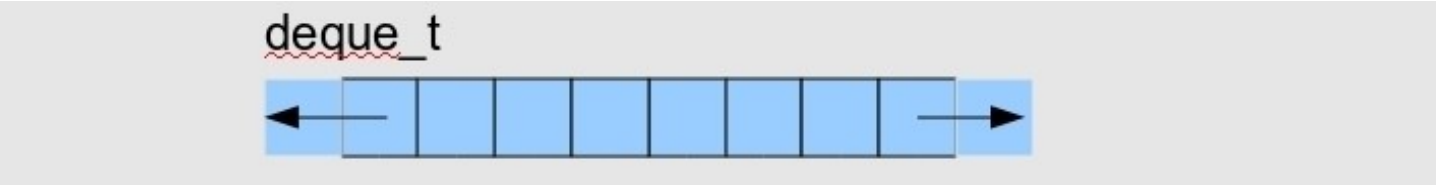


图 4.2 deque_t 的内部结构

为了使用 deque_t，必须包含头文件<csatl/cdeque.h>

1. deque_t 的能力

与 vector_t 相比 deque_t 有很多不同之处：

- deque_t 在开头和结尾插入或删除数据都是高效的。
- 没有容量的概念和内存再分配的概念，所以没有 deque_capacity() 函数。
- 当数据被删除后 deque_t 会成块的释放占用的内存。

deque_t 与 vector_t 相似之处：

- 在中间插入或者删除数据比较慢，因为要移动数据。
- 支持随即访问, 提供随即访问迭代器。

2. deque_t 的操作

- 创建, 初始化和销毁操作

create_deque	创建一个 deque_t 容器。
deque_init	初始化一个空的 deque_t 容器。
deque_init_n	使用多个默认数据初始化 deque_t 容器。
deque_init_elem	使用多个指定数据初始化 deque_t 容器。
deque_init_copy	使用既存的 deque_t 初始化 deque_t 容器。
deque_init_copy_range	使用指定的数据区间初始化 deque_t 容器。
deque_destroy	销毁 deque_t 容器。

- 非质变操作

deque_t 的非质变操作也有两种，一种是与数据数量有关操作函数：

deque_size	返回 deque_t 容器中数据的数量。
------------	----------------------

deque_empty	测试 deque_t 容器是否为空。
deque_max_size	返回 deque_t 中能够保存的数据数量的最大值。

deque_t 没有容量的概念了所以没有就与容量相关操作函数 deque_capacity()和 deque_reserve()。

第二种非质变函数是比较操作函数：

deque_equal	测试第一个 deque_t 是否等于第二个 deque_t。
deque_not_equal	测试第一个 deque_t 是否不等于第二个 deque_t。
deque_less	测试第一个 deque_t 是否小于第二个 deque_t。
deque_less_equal	测试第一个 deque_t 是否小于等于第二个 deque_t。
deque_greater	测试第一个 deque_t 是否大于第二个 deque_t。
deque_greater_equal	测试第一个 deque_t 是否大于等于第二个 deque_t。

● 赋值和交换

deque_assign	使用既存的 deque_t 为 deque_t 赋值。
deque_assign_elem	使用多个指定数据为 deque_t 赋值。
deque_assign_range	使用指定数据区间为 deque_t 赋值。
deque_swap	交换两个 deque_t 中的内容。

● 数据访问

deque_at	使用下标对 deque_t 中的数据进行随机访问。
deque_front	访问 deque_t 中的第一个数据。
deque_back	访问 deque_t 中的最后一个数据。

● 与迭代器相关的操作函数

deque_begin	返回指向 deque_t 中第一个数据的迭代器。
deque_end	返回指向 deque_t 末尾位置的迭代器。

插入和删除

deque_insert	向 deque_t 中插入一个指定的数据。
deque_insert_n	向 deque_t 中插入多个指定的数据。
deque_insert_range	向 deque_t 中插入指定的数据区间。
deque_push_back	在 deque_t 的末尾添加一个数据。
deque_pop_back	删除 deque_t 中的最后一个数据。
deque_push_front	在 deque_t 的开头添加一个数据。
deque_pop_front	删除 deque_t 的第一个数据。

deque_erase	删除 deque_t 中指定位置的数据。
deque_erase_range	删除 deque_t 中指定数据区间的数据。
deque_clear	删除 deque_t 中的所有数据。
deque_resize	重新设置 deque_t 中数据的数量，如果新的数量比现在的数量大，那么使用默认数据填充。
deque_resize_elem	重新设置 deque_t 中数据的数量，如果新的数量比现在的大，那么使用指定数据填充。

deque_t 与 vector_t 不同的是提供了针对第一个数据的插入和删除操作(deque_push_front()和 deque_pop_front())。

3. deque_t 的应用实例

```

/*
 * deque2.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cdeque.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%s\n", (char*)cpv_input);
}

int main(int argc, char* argv[])
{
    deque_t* pdeq_coll = create_deque(char*);

    if(pdeq_coll == NULL)
    {
        return -1;
    }

    deque_init(pdeq_coll);

    deque_assign_elem(pdeq_coll, 3, "string");
    deque_push_back(pdeq_coll, "last string");
    deque_push_front(pdeq_coll, "first string");

    /* print all elements with new line */
    algo_for_each(deque_begin(pdeq_coll), deque_end(pdeq_coll), _print);

```

```

printf("\n");

/* remove first and last elements */
deque_pop_back(pdeq_coll);
deque_pop_front(pdeq_coll);

/* print all elements with new line */
algo_for_each(deque_begin(pdeq_coll), deque_end(pdeq_coll), _print);
printf("\n");

/* change size to four elements */
deque_resize_elem(pdeq_coll, 4, "resized string");

/* print all elements with new line */
algo_for_each(deque_begin(pdeq_coll), deque_end(pdeq_coll), _print);
printf("\n");

deque_destroy(pdeq_coll);

return 0;
}

```

输出的结果如下：

```

first string
string
string
string
last string

```

```

string
string
string

```

```

string
string
string
resized string

```

第四节 list_t

list_t 是双向链表。

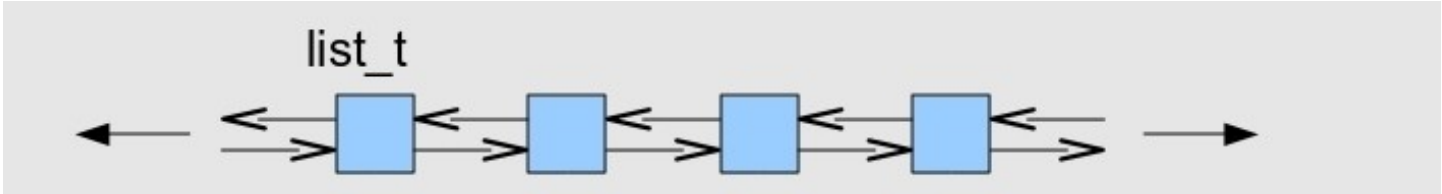


图 4.3 list_t 的结构

要使用 list_t 必须包含头文件 <cstdlib/clist.h>
`#include <cstdlib/clist.h>`

1. list_t 的能力

list_t 的结构与 vector_t 和 deque_t 有很大的不同, 因此 list_t 与它们的能力也有很大的不同:

- list_t 不支持随即访问, 例如要访问的五个数据, 就必须从头开始查找, 一直到的五个数据, 所以要访问 list_t 中的任意数据是低效的。
- list_t 支持在任意位置高效的插入和删除数据, 不仅仅实在开头和结尾。
- 在 list_t 中插入或删除数据不会使迭代器失效。

因此 list_t 也同了与 vector_t 和 deque_t 不同的操作函数:

- list_t 不支持随机访问数据所以不提供 list_at() 函数。
- list_t 不支持容量和内存的重新分配所以不提供 list_capacity() 和 list_reserve() 函数。
- list_t 还提供了和很多特有的函数如 list_sort() 等。

2. list_t 的操作

- 创建, 初始化和销毁操作

list_t 的创建, 初始化和销毁操作函数与其他序列容器相同。

create_list	创建 list_t 容器类型。
list_init	初始化一个空的 list_t。
list_init_n	使用多个默认数据初始化 list_t。
list_init_elem	使用多个指定数据初始化 list_t。
list_init_copy	使用既存的 list_t 初始化当前的 list_t。
list_init_copy_range	使用指定的数据区间初始化 list_t。
list_destroy	销毁 list_t 容器类型。

- 非质变操作

list_t 的非质变操作同样包含与数据数量相关的操作和比较操作:

list_size	返回 list_t 中数据的数量。
list_empty	测试 list_t 是否为空。

list_max_size	返回 list_t 中保存数据数量的最大值。
list_equal	测试两个 list_t 是否相等。
list_not_equal	测试两个 list_t 是否不等。
list_less	测试第一个 list_t 是否小于第二个 list_t。
list_less_equal	测试第一个 list_t 是否小于等于第二个 list_t。
list_greater	测试第一个 list_t 是否大于第二个 list_t。
list_greater_equal	测试第一个 list_t 是否大于等于第二个 list_t。

● 赋值与交换

list_assign	使用既存的 list_t 为 list_t 赋值。
list_assign_elem	使用多个指定的数据为 list_t 赋值。
list_assign_range	使用指定的数据区间为 list_t 赋值。
list_swap	交换两个 list_t 中的内容。

● 数据访问

list_t 没有随机访问能力所以不提供 list_at() 操作函数：

list_front	访问 list_t 中的第一个数据。
list_back	访问 list_t 中的最后一个数据。

● 与迭代器相关的操作函数

list_begin	返回指向 list_t 中第一个数据的迭代器。
list_end	返回指向 list_t 中末尾位置的迭代器。

list_t 提供的是双向迭代器，不是所有的算法都接受双向迭代器，在对 list_t 的数据区间使用算法时请注意。

● 插入和删除

list_insert	向 list_t 中插入一个指定的数据。
list_insert_n	向 list_t 中插入多个指定的数据。
list_insert_range	向 list_t 中插入一个指定的数据区间。
list_push_back	在 list_t 末尾添加一个数据。
list_pop_back	删除 list_t 的最后一个数据。
list_push_front	在 list_t 的开头添加一个数据。
list_pop_front	删除 list_t 的第一个数据。
list_erase	删除 list_t 中指定位置的数据。
list_erase_range	删除 list_t 中指定数据区间的数据。

list_remove	删除 list_t 中与指定数据相等的数据。
list_remove_if	删除 list_t 中符合指定条件的数据。
list_clear	删除 list_t 中的所有数据。
list_resize	重新设置 list_t 中数据的数量，当新数量大于当前数量时使用默认数据填充。
list_resize_elem	重新设置 list_t 中数据的数量，当新数量大于当前数量时使用指定数据填充。

list_t 增加了两个新的删除数据的操作函数：list_remove()和 list_remove_if()这个操作函数是删除满足特定规则的数据，对于 list_t 容器本身来说它们要比算法版本的 algo_remove()和 algo_remove_if()函数好快的多。

● 特有的操作

除了上述操作函数值为 list_t 还提供了很多特有的操作函数，这些操作函数都是基于 list_t 本身结构的特点，list_t 具有在任意位置插入或删除数据都是常量时间的特点。如果从一个 list_t 向另一个 list_t 转移数据那么这个特点就得到了更大的发挥。

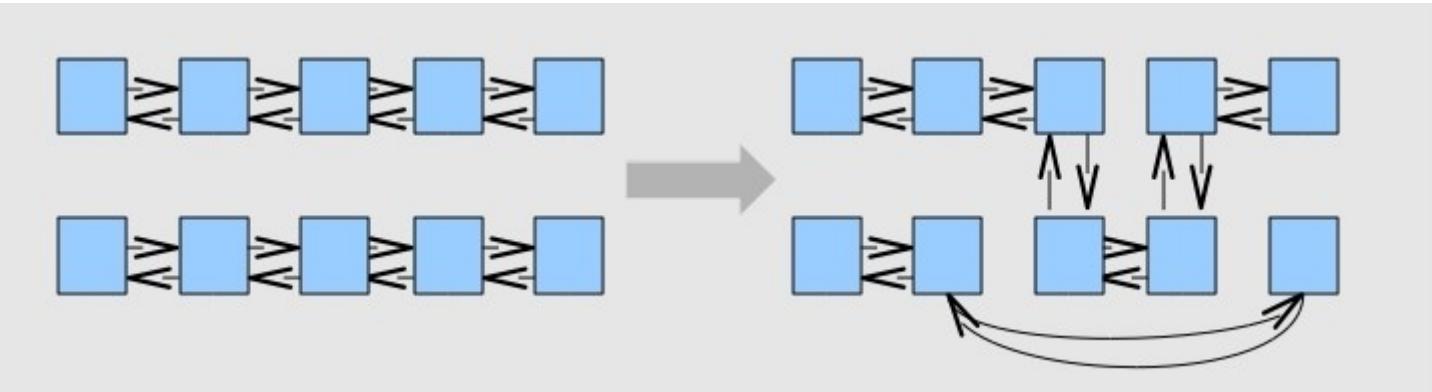


图 4.4 list_t 之间的数据转移

list_unique	删除 list_t 中相邻的重复数据。
list_unique_if	删除 list_t 中相邻的符合指定规则的数据。
list_splice	将第二个 list_t 中的数据转移到第一个 list_t 的指定位置。
list_splice_pos	将第二个 list_t 中指定位置的数据转移到第一个 list_t 的指定位置。
list_splice_range	将第二个 list_t 中指定数据区间的数据转移到第一个 list_t 的指定位置。
list_sort	将 list_t 中的数据排序。
list_sort_if	将 list_t 中的数据按照指定的规则排序。
list_merge	将两个有序的 list_t 合并到第一个 list_t 中。
list_merge_if	将两个按照指定规则排序的 list_t 合并到第一个 list_t 中。
list_reverse	将 list_t 中的数据逆序。

3. list_t 的使用实例

```
/*
 * list3.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/clist.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

static void _print_list(const list_t* cplist_coll1, const list_t* cplist_coll2)
{
    printf("list1: ");
    algo_for_each(list_begin(cplist_coll1), list_end(cplist_coll1), _print);
    printf("\n");
    printf("list2: ");
    algo_for_each(list_begin(cplist_coll2), list_end(cplist_coll2), _print);
    printf("\n");
}

int main(int argc, char* argv[])
{
    list_t* plist_coll1 = create_list(int);
    list_t* plist_coll2 = create_list(int);
    list_iterator_t it_pos;
    int i = 0;

    if(plist_coll1 == NULL || plist_coll2 == NULL)
    {
        return -1;
    }

    list_init(plist_coll1);
    list_init(plist_coll2);
}
```

```

for(i = 0; i < 6; ++i)
{
    list_push_back(plist_coll1, i);
    list_push_front(plist_coll2, i);
}
_print_list(plist_coll1, plist_coll2);

it_pos = algo_find(list_begin(plist_coll1), list_end(plist_coll1), 3);
list_splice(plist_coll1, it_pos, plist_coll2);

_print_list(plist_coll1, plist_coll2);

list_splice_pos(plist_coll1, list_end(plist_coll1),
    plist_coll1, list_begin(plist_coll1));
_print_list(plist_coll1, plist_coll2);

list_sort(plist_coll1);

list_assign(plist_coll2, plist_coll1);
list_unique(plist_coll2);
_print_list(plist_coll1, plist_coll2);

list_merge(plist_coll1, plist_coll2);
_print_list(plist_coll1, plist_coll2);

list_destroy(plist_coll1);
list_destroy(plist_coll2);

return 0;
}

```

输出结果:

list1: 0 1 2 3 4 5

list2: 5 4 3 2 1 0

list1: 0 1 2 5 4 3 2 1 0 3 4 5

list2:

list1: 1 2 5 4 3 2 1 0 3 4 5 0

list2:

list1: 0 0 1 1 2 2 3 3 4 4 5 5

list2: 0 1 2 3 4 5

list1: 0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5

list2:

第五节 slist_t

slist_t 的单链表。

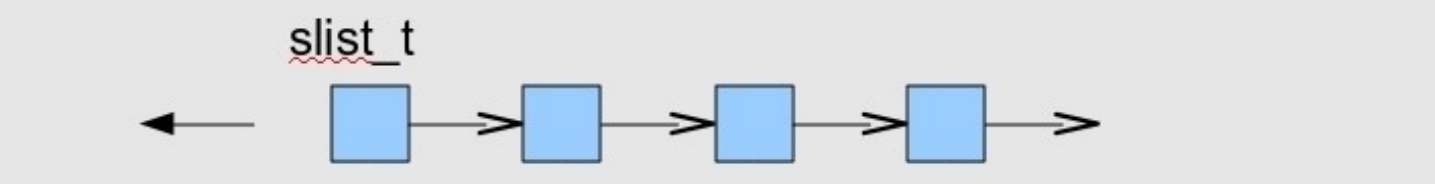


图 4.5 slist_t 的结构

要使用 slist_t 必须包含头文件<cstl/slist_t>

```
#include <cstl/slist_t>
```

1. slist_t 的能力

slist_t 是单链表，所以它的很多功能与 list_t 相比有很大不同。
slist_t 只支持向前访问的能力，所以随即访问能力效率很低，访问某一数据的前驱效率也很低，但是访问后继很高效。例如当前数据是的五个数据，如果要访问第六个数据很快，但是要访问第四个数据就必须从头查找。
由于上面的原因，在 slist_t 的任意位置后面插入或删除效率很高，但是插入或删除任意位置的数据效率很低，所以 slist_t 提供了很多针对任意位置后面的插入和删除操作。
slist_t 提供向前迭代器。

2. slist_t 的操作

● 创建，初始化和销毁操作	
create_slist	创建 slist_t 容器类型。
slist_init	初始化一个空的 slist_t 容器类型。
slist_init_n	使用多个默认数据初始化 slist_t 容器类型。
slist_init_elem	使用多个指定数据初始化 slist_t 容器类型。
slist_init_copy	使用既存的 slist_t 初始化当前的 slist_t 容器类型。
slist_init_copy_range	使用指定的数据区间初始化 slist_t 容器类型。
slist_destroy	销毁 slist_t 容器类型。

● 非质变操作
非质变操作函数同样包含与数据数量有关的操作和比较操作：

slist_size	返回 slist_t 中数据的数量。
slist_empty	测试 slist_t 容器是否为空。
slist_max_size	返回 slist_t 中能够保存数据数量的最大值。
slist_equal	测试两个 slist_t 容器是否相等。
slist_not_equal	测试两个 slist_t 容器是否不等。
slist_less	测试第一个 slist_t 容器是否小于第二个 slist_t 容器。
slist_less_equal	测试第一个 slist_t 容器是否小于等于第二个 slist_t 容器。
slist_greater	测试第一个 slist_t 容器是否大于第二个 slist_t 容器。
slist_greater_equal	测试第一个 slist_t 容器是否大于等于第二个 slist_t 容器。

● 赋值和交换

slist_assign	使用既存 slist_t 为当前 slist_t 赋值。
slist_assign_elem	使用多个指定数据为 slist_t 赋值。
slist_assign_range	使用指定数据区间为 slist_t 赋值。
slist_swap	交换两个 slist_t 中的数据。

● 数据访问

slist_front	访问 slist_t 中的第一个数据。
-------------	---------------------

因为 slist_t 访问最后一个数据效率很低，所以只能访问第一个数据。同样，对最后一个数据的操作函数也没有提供。

● 与迭代器相关的操作函数

slist_begin	返回指向 slist_t 中第一个数据的迭代器。
slist_end	返回指向 slist_t 中末尾数据的迭代器。
slist_previous	返回当前位置的前一个数据的迭代器。

为了简便访问前驱的操作，slist_t 增加了 slist_previous()。

● 插入和删除

slist_push_front	在 slist_t 的开头添加一个数据。
slist_pop_front	删除 slist_t 的第一个数据。
slist_insert	向 slist_t 指定位置插入一个数据。
slist_insert_n	向 slist_t 指定位置插入多个数据。
slist_insert_range	向 slist_t 指定位置插入一个数据区间。
slist_insert_after	向 slist_t 指定位置的后面插入一个数据。
slist_insert_after_n	向 slist_t 指定位置的后面插入多个数据。

slist_insert_after_range	向 slist_t 指定位置的后面插入一个数据区间。
slist_erase	删除 slist_t 中指定位置的数据。
slist_erase_range	删除 slist_t 中指定的数据区间。
slist_erase_after	删除 slist_t 指定位置后面的数据。
slist_erase_after_range	删除 slist_t 指定数据区间后面的数据区间。
slist_remove	删除 slist_t 中指定的数据。
slist_remove_if	删除 slist_t 中符合指定规则的数据。
slist_clear	删除 slist_t 中所有的数据。
slist_resize	重新设置 slist_t 中数据的数量，如果新数量大于现在的数量，那么使用默认数据填充。
slist_resize_elem	重新设置 slist_t 中数据的数量，如果新数量大于现在的数量，那么使用指定的数据填充。

由于在当前位置的后面数据插入和删除效率更高，所以插入和删除操作都增加了 after 版本。

● 特有的操作

slist_unique	删除 slist_t 中相邻的重复数据。
slist_unique_if	删除 slist_t 中相邻的符合指定规则的数据。
slist_splice	将第二个 slist_t 中的数据转移到第一个 slist_t 中的指定位置。
slist_splice_pos	将第二个 slist_t 中指定位置的数据转移到第一个 slist_t 的指定位置。
slist_splice_range	将第二个 slist_t 中指定的数据区间转移到第一个 slist_t 的指定位置。
slist_splice_after_pos	将第二个 slist_t 中指定位置后面的数据转移到第一个 slist_t 的指定位置后面。
slist_splice_after_range	将第二个 slist_t 中指定数据区间后面的数据区间转移到第一个 slist_t 指定位置的后面。
slist_sort	将 slist_t 中的数据排序。
slist_sort_if	将 slist_t 中的数据按照指定规则排序。
slist_merge	将两个有序的 slist_t 合并。
slist_merge_if	将两个按照指定规则排序的 slist_t 合并。
slist_reverse	将 slist_t 中的数据逆序。

3. slist_t 使用实例

```
/*
 * slist1.c
 * compile with : -lcstl
 */
```

```

#include <stdio.h>
#include <cstl/cslist.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

static void _print_slist(const slist_t* cpslist_coll1, const slist_t* cpslist_coll2,
    const slist_t* cpslist_coll3, const slist_t* cpslist_coll4)
{
    printf("slist1: ");
    algo_for_each(slist_begin(cpslist_coll1), slist_end(cpslist_coll1), _print);
    printf("\n");
    printf("slist2: ");
    algo_for_each(slist_begin(cpslist_coll2), slist_end(cpslist_coll2), _print);
    printf("\n");
    printf("slist3: ");
    algo_for_each(slist_begin(cpslist_coll3), slist_end(cpslist_coll3), _print);
    printf("\n");
    printf("slist4: ");
    algo_for_each(slist_begin(cpslist_coll4), slist_end(cpslist_coll4), _print);
    printf("\n\n");
}

int main(int argc, char* argv[])
{
    slist_t* pslist_coll1 = create_slist(int);
    slist_t* pslist_coll2 = create_slist(int);
    slist_t* pslist_coll3 = create_slist(int);
    slist_t* pslist_coll4 = create_slist(int);
    int i = 0;

    if(pslist_coll1 == NULL || pslist_coll2 == NULL ||
        pslist_coll3 == NULL || pslist_coll4 == NULL)
    {
        return -1;
    }

    slist_init(pslist_coll1);
    slist_init(pslist_coll2);
    slist_init(pslist_coll3);

```

```

slist_init(pslist_coll4);

for(i = 0; i < 6; ++i)
{
    slist_push_front(pslist_coll1, i + 1);
    slist_push_front(pslist_coll2, - i - 1);
    slist_push_front(pslist_coll3, i + 1);
    slist_push_front(pslist_coll4, - i - 1);
}
_print_slist(pslist_coll1, pslist_coll2, pslist_coll3, pslist_coll4);

slist_splice_pos(
    pslist_coll1,
    algo_find(slist_begin(pslist_coll1), slist_end(pslist_coll1), 3),
    pslist_coll2,
    algo_find(slist_begin(pslist_coll2), slist_end(pslist_coll2), -3));
slist_splice_after_pos(
    pslist_coll3,
    algo_find(slist_begin(pslist_coll3), slist_end(pslist_coll3), 3),
    pslist_coll4,
    algo_find(slist_begin(pslist_coll4), slist_end(pslist_coll4), -3));

_print_slist(pslist_coll1, pslist_coll2, pslist_coll3, pslist_coll4);

slist_destroy(pslist_coll1);
slist_destroy(pslist_coll2);
slist_destroy(pslist_coll3);
slist_destroy(pslist_coll4);

return 0;
}

```

结果:

slist1: 6 5 4 3 2 1

slist2: -6 -5 -4 -3 -2 -1

slist3: 6 5 4 3 2 1

slist4: -6 -5 -4 -3 -2 -1

slist1: 6 5 4 -3 3 2 1

slist2: -6 -5 -4 -2 -1

slist3: 6 5 4 3 -2 2 1

slist4: -6 -5 -4 -3 -1

第六节 set_t 和 multiset_t

set_t 和 multiset_t 按照某种规则自动排序容器中保存的数据，它们的区别在于 multiset_t 允许数据重复但是 set_t 不允许。

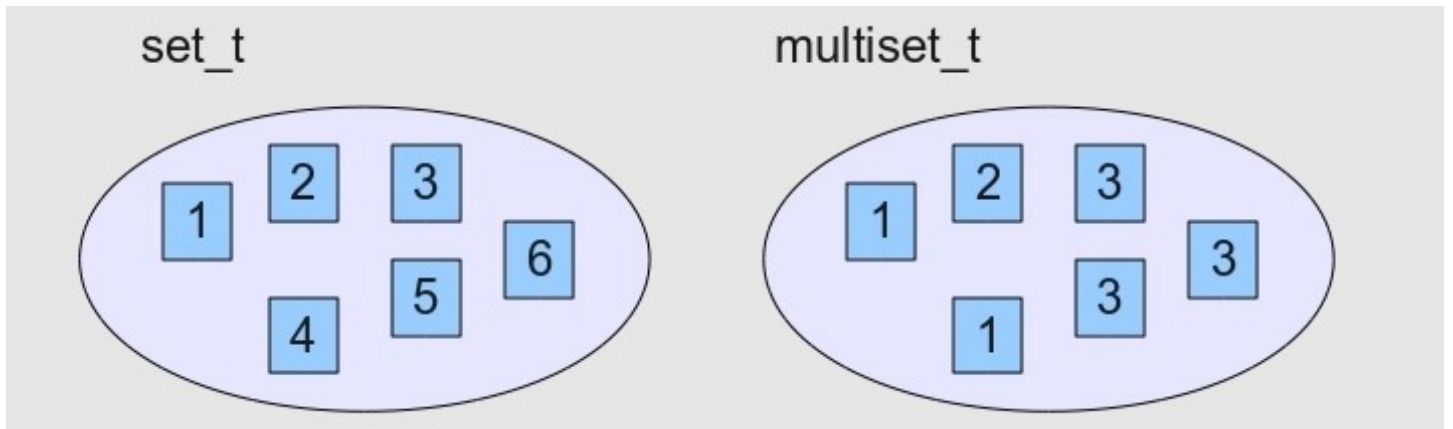


图 4.6 set_t 和 multiset_t

为了使用 set_t 和 multiset_t，必须包含头文件<cstl/cset_t>

```
#include <cstl/cset_t>
```

set_t 和 multiset_t 在初始化的时候可以指定排序规则，如果用户没有指定排序规则，set_t 和 multiset_t 使用默认的排序规则，这个默认的规则是与数据类型关联的小于操作函数。

1. set_t 和 multiset_t 的能力

set_t 和 multiset_t 通常采用平衡二叉树来保存数据 (set_t 和 multiset_t 可以采用 avl 树或红黑树作为底层实现)。

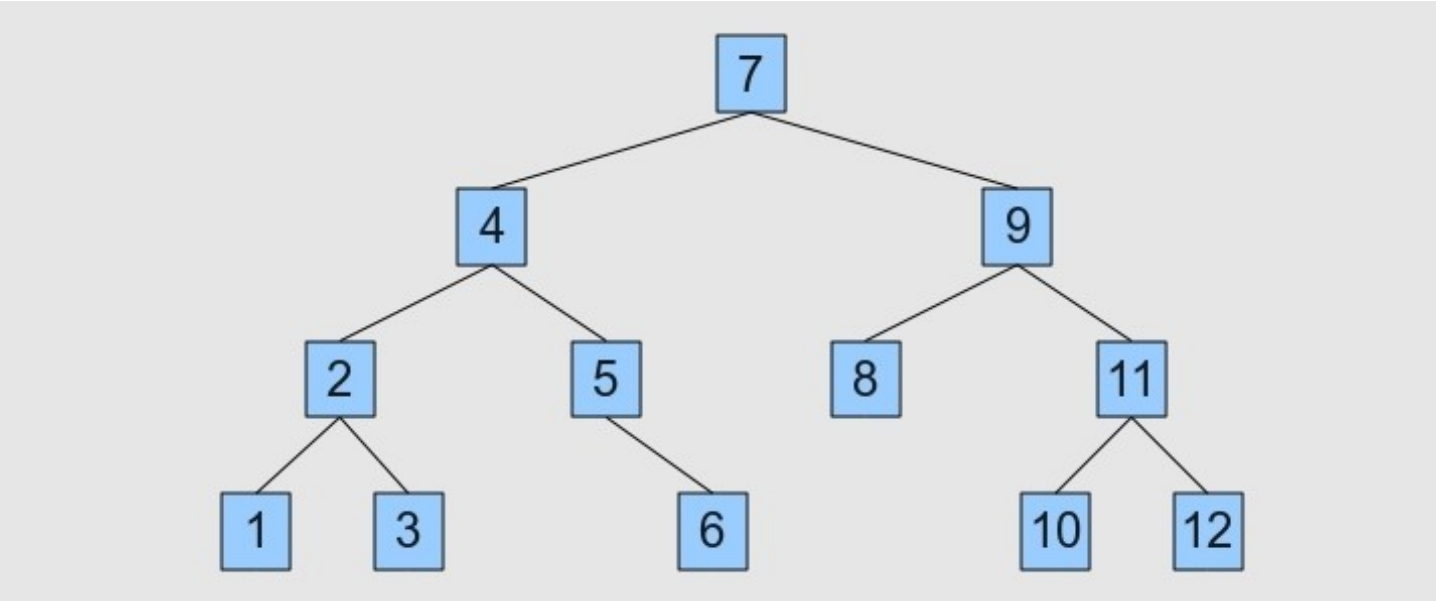


图 4.7 set_t 和 multiset_t 的内部结构

自动排序的好处是当查找某一个数据的时候非常高效，可以达到对数复杂度。但是排序之后就是不能修改容器中的数据，因为修改了容器中的数据就破坏了数据的顺序，造成其他操作的结果错误，只能通过删除就数据然后插入新数据来修改数据。所以 set_t 和 multiset_t 没有提供数据访问的操作，同时要注意的是用户也不能通过迭代器来修改数据。

2. set_t 和 multiset_t 操作

set_t 和 multiset_t 的操作函数完全一样，这里以 set_t 为例介绍，详细的请参考《The libcstl Library Reference Manual》。

● 创建，初始化和销毁操作

create_set	创建 set_t 容器类型。
set_init	使用默认的排序规则初始化一个空的 set_t 容器。
set_init_ex	使用用户指定的排序规则初始化一个空的 set_t 容器。
set_init_copy	通过拷贝来初始化 set_t 容器，新容器的数据和排序规则都来源于原有的 set_t 容器。
set_init_copy_range	使用指定的数据区间和默认的排序规则初始化 set_t 容器。
set_init_copy_range_ex	使用指定的数据区间和指定的排序规则初始化 set_t 容器。
set_destroy	销毁 set_t 容器类型。

● 非质变操作

set_size	返回 set_t 中数据的数量。
set_empty	测试 set_t 容器是否为空。
set_max_size	返回 set_t 中能够保存的数据数量的最大值。

set_equal	测试两个 set_t 是否相等。
set_not_equal	测试两个 set_t 是否不等。
set_less	测试第一个 set_t 是否小于第二个 set_t。
set_less_equal	测试第一个 set_t 是否小于等于第二个 set_t。
set_greater	测试第一个 set_t 是否大于第二个 set_t。
set_greater_equal	测试第一个 set_t 是否大于等于第二个 set_t。

● 特殊的查找函数

set_t 和 multiset_t 的实现都很适合快速的查找，所以它们也提供了许多查找操作，这些查找操作是算法的特殊版本，当在关联容器中查找数据是应该总是使用容器本身提供的查找函数。

set_find	在 set_t 中查找指定的数据。
set_count	统计 set_t 中指定数据的个数。
set_equal_range	返回 set_t 中包含指定数据的数据区间。
set_lower_bound	返回 set_t 中指向第一个等于指定数据的迭代器。
set_upper_bound	返回 set_t 中指向第一个大于指定数据的迭代器。

set_find() 获得第一个与指定值相等的数据的迭代器，如果没有则返回 set_end()。

set_lower_bound() 和 set_upper_bound() 获得一个数据区间的第一个会最后一个迭代器，这个区间中的数据都等于指定的数据，而 set_equal_range() 的结果是与上面两个操作的总体结果相同，但是它获得的是一个两个元素都是迭代器的 range_t 者两个迭代器就是区间的边界。

下面是一个如何使用 set_lower_bound(), set_upper_bound() 和 set_equal_range() 的例子：

```
/*
 * set2.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cset.h>

int main(int argc, char* argv[])
{
    set_t* pset_coll = create_set(int);
    set_iterator_t it_lower;
    set_iterator_t it_upper;
    range_t t_range;

    if(pset_coll == NULL)
    {
        return -1;
    }
}
```

```

}

set_init(pset_coll);

set_insert(pset_coll, 1);
set_insert(pset_coll, 2);
set_insert(pset_coll, 4);
set_insert(pset_coll, 5);
set_insert(pset_coll, 6);

it_lower = set_lower_bound(pset_coll, 3);
it_upper = set_upper_bound(pset_coll, 3);
t_range = set_equal_range(pset_coll, 3);
printf("set_lower_bound(3) : %d\n", *(int*)iterator_get_pointer(it_lower));
printf("set_upper_bound(3) : %d\n", *(int*)iterator_get_pointer(it_upper));
printf("set_equal_range(3) : %d %d\n",
      *(int*)iterator_get_pointer(t_range.it_begin),
      *(int*)iterator_get_pointer(t_range.it_end));

printf("\n");
it_lower = set_lower_bound(pset_coll, 5);
it_upper = set_upper_bound(pset_coll, 5);
t_range = set_equal_range(pset_coll, 5);
printf("set_lower_bound(5) : %d\n", *(int*)iterator_get_pointer(it_lower));
printf("set_upper_bound(5) : %d\n", *(int*)iterator_get_pointer(it_upper));
printf("set_equal_range(5) : %d %d\n",
      *(int*)iterator_get_pointer(t_range.it_begin),
      *(int*)iterator_get_pointer(t_range.it_end));

set_destroy(pset_coll);

return 0;
}

```

输出结果:

```

set_lower_bound(3) : 4
set_upper_bound(3) : 4
set_equal_range(3) : 4 4

```

```

set_lower_bound(5) : 5
set_upper_bound(5) : 6
set_equal_range(5) : 5 6

```

如果使用 `multiset_t` 会得到同样的结果。

- 赋值和交换

set_assign	使用既存的 set_t 为当前的 set_t 赋值。
set_swap	交换两个 set_t 中的内容。

set_t 和 multiset_t 只提供了简单的赋值操作。

- 与迭代器有关的操作函数

set_begin	返回指向 set_t 中第一个数据的迭代器。
set_end	返回指向 set_t 末尾位置的迭代器。

set_t 和 multiset_t 提供双向迭代器。禁止通过迭代器操作间接的修改容器中的数据，同样禁止将质变算法作用于关联容器。

- 插入和删除

set_insert	向 set_t 中插入指定的数据。
set_insert_hint	向 set_t 中插入指定的数据，同时给出被插入的数据在 set_t 中的位置提示。
set_insert_range	向 set_t 中插入指定的数据区间。
set_erase	删除 set_t 中的指定数据。
set_erase_pos	删除 set_t 中指定位置的数据。
set_erase_range	删除 set_t 中指定数据区间的数据。
set_clear	删除 set_t 中的所有数据。

由于 set_t 不允许数据重复而 multiset_t 允许数据重复，那么在插入和删除操作中两个容器对用的操作函数的行为就有区别了，对于插入操作，set_t 中如果已经存在与要插入的数据相等的数据则插入会失败，返回 set_end()但是对于 multiset_t 这样的插入就不会失败，返回的是被插入的数据在 multiset_t 中位置的迭代器。同时对于删除操作 set_t 将容器中与指定数据相等的唯一数据删除，但是 multiset_t 将容器中所有与指定数据相等的数据都删掉了。

3. set_t 和 multiset_t 的使用实例

下面这个例子展示了 set_t 的一些能力：

```
/*
 * set3.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cset.h>
#include <cstl/cfunctional.h> /* for fun_greater_int */
```

```

int main(int argc, char* argv[])
{
    set_t* pset_coll1 = create_set(int);
    set_t* pset_coll2 = create_set(int);
    set_iterator_t it_pos;
    size_t t_count = 0;

    if(pset_coll1 == NULL || pset_coll2 == NULL)
    {
        return -1;
    }

    /* empty set container and descending order */
    set_init_ex(pset_coll1, fun_greater_int);

    /* insert elements in random order */
    set_insert(pset_coll1, 4);
    set_insert(pset_coll1, 3);
    set_insert(pset_coll1, 5);
    set_insert(pset_coll1, 1);
    set_insert(pset_coll1, 6);
    set_insert(pset_coll1, 2);
    set_insert(pset_coll1, 5);

    /* iterate over all elements and print them */
    for(it_pos = set_begin(pset_coll1);
        !iterator_equal(it_pos, set_end(pset_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    /* insert 4 again */
    it_pos = set_insert(pset_coll1, 4);
    if(iterator_equal(it_pos, set_end(pset_coll1)))
    {
        printf("4 already exists\n");
    }
    else
    {
        printf("4 inserted as element %d\n",

```

```

        iterator_distance(set_begin(pset_coll1), it_pos));
    }

    /* initialize set_t with another and ascending order */
    set_init_copy_range(pset_coll2, set_begin(pset_coll1), set_end(pset_coll1));

    /* print all elements of the copy */
    for(it_pos = set_begin(pset_coll2);
        !iterator_equal(it_pos, set_end(pset_coll2));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    /* remove all elements up to element with value 3 */
    set_erase_range(pset_coll2, set_begin(pset_coll2), set_find(pset_coll2, 3));

    /* remove all elements with value 5 */
    t_count = set_erase(pset_coll2, 5);
    printf("%u elements removed\n", t_count);

    /* print all elements of the copy */
    for(it_pos = set_begin(pset_coll2);
        !iterator_equal(it_pos, set_end(pset_coll2));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    set_destroy(pset_coll1);
    set_destroy(pset_coll2);

    return 0;
}

```

在初始化 pset_coll1 的时候使用了自定义的排序规则 fun_greater_int，这是为了让 set_t 中的数据按照从大到小的顺序排序。在第二次插入数据 5 的时候 set_insert(pset_coll1, 5); 操作失败，同样的还有再次插入数据 4 的时候插入操作函数也是失败的，并且 it_pos = set_insert(pset_coll1, 4); 的返回值是 set_end()。在初始化 pset_coll2 的时候我们使用了拷贝初始化操作函数，从 pset_coll1 拷贝数据，但是排序规则使用默认的排序规则。

输出结果：

6 5 4 3 2 1

4 already exists

1 2 3 4 5 6

1 elements removed

3 4 6

同样的程序如果使用 multiset_t 会出现不同的结果:

```
/*
 * multiset2.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cset.h>
#include <cstl/cfunctional.h> /* for fun_greater_int */

int main(int argc, char* argv[])
{
    multiset_t* pmset_coll1 = create_multiset(int);
    multiset_t* pmset_coll2 = create_multiset(int);
    multiset_iterator_t it_pos;
    size_t t_count = 0;

    if(pmset_coll1 == NULL || pmset_coll2 == NULL)
    {
        return -1;
    }

    /* empty multiset container and descending order */
    multiset_init_ex(pmset_coll1, fun_greater_int);

    /* insert elements in random order */
    multiset_insert(pmset_coll1, 4);
    multiset_insert(pmset_coll1, 3);
    multiset_insert(pmset_coll1, 5);
    multiset_insert(pmset_coll1, 1);
    multiset_insert(pmset_coll1, 6);
    multiset_insert(pmset_coll1, 2);
    multiset_insert(pmset_coll1, 5);

    /* iterate over all elements and print them */
    for(it_pos = multiset_begin(pmset_coll1);
        !iterator_equal(it_pos, multiset_end(pmset_coll1));
        it_pos = iterator_next(it_pos))
```



```

{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");

/* insert 4 again */
it_pos = multiset_insert(pmset_coll1, 4);
if(iterator_equal(it_pos, multiset_end(pmset_coll1)))
{
    printf("4 already exists\n");
}
else
{
    printf("4 inserted as element %d\n",
        iterator_distance(multiset_begin(pmset_coll1), it_pos));
}

/* initialize multiset_t with another and ascending order */
multiset_init_copy_range(pmset_coll2,
    multiset_begin(pmset_coll1), multiset_end(pmset_coll1));

/* print all elements of the copy */
for(it_pos = multiset_begin(pmset_coll2);
    !iterator_equal(it_pos, multiset_end(pmset_coll2));
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");

/* remove all elements up to element with value 3 */
multiset_erase_range(pmset_coll2,
    multiset_begin(pmset_coll2), multiset_find(pmset_coll2, 3));

/* remove all elements with value 5 */
t_count = multiset_erase(pmset_coll2, 5);
printf("%u elements removed\n", t_count);

/* print all elements of the copy */
for(it_pos = multiset_begin(pmset_coll2);
    !iterator_equal(it_pos, multiset_end(pmset_coll2));
    it_pos = iterator_next(it_pos))
{

```

```

        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    multiset_destroy(pmset_coll1);
    multiset_destroy(pmset_coll2);

    return 0;
}

```

输出的结果:

6 5 5 4 3 2 1

4 inserted as element 4

1 2 3 4 4 5 5 6

2 elements removed

3 4 4 6

第七节 map_t 和 multimap_t

map_t 和 multimap_t 都是已 key/value 这种形式保存数据，所以 map_t 和 multimap_t 中保存的都是 pair_t 类型。容器使用 key 通过某种规则对保存在其中的数据进行排序。这两个容器的不同点就是 multimap_t 允许 key 重复。

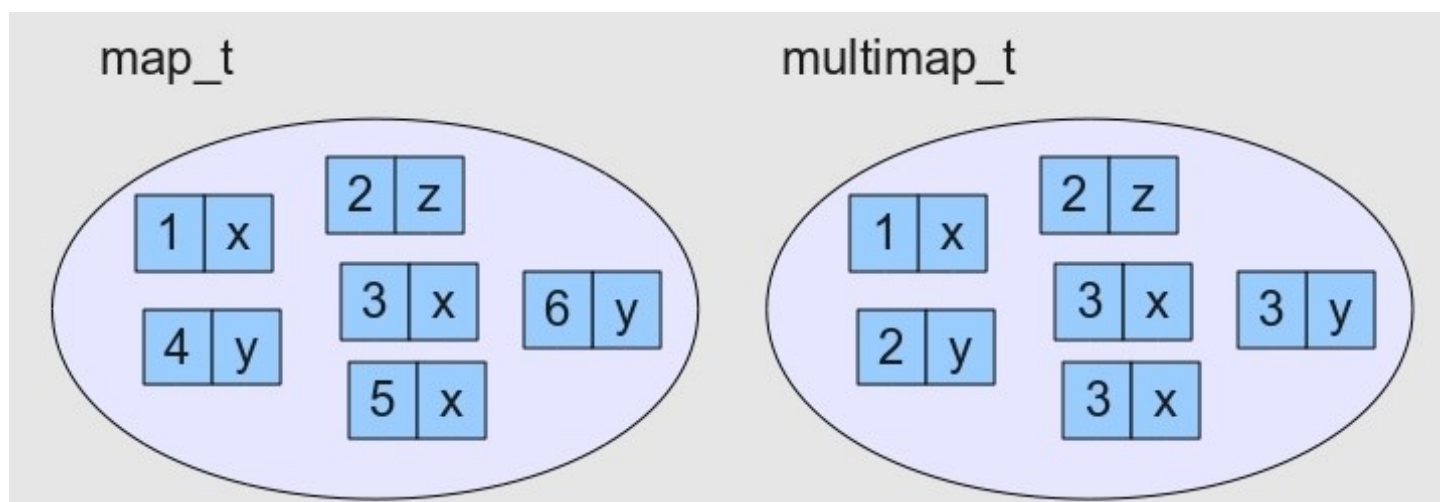


图 4.8 map_t 和 multimap_t

要使用 map_t 和 multimap_t 必须包含头文件 <cstdlib/cmap.h>

```
#include <cstdlib/cmap.h>
```

用户可以在初始化容器的时候指定排序规则，如果没有指定排序规则使用默认的排序规则。map_t 和 multimap_t 类型都是根据 key 排序的，不管是默认的排序规则还是用户指定的排序规则都是针对与 key 的，与 value 无关。

1. map_t 和 multimap_t 的能力

与其他的关联容器一样，map_t 和 multimap_t 也是使用平衡二叉树实现的。其实 set_t, multiset_t, map_t, multimap_t 都是使用统一的内部实现，所以你可以认为 set_t, multiset_t 是特殊 map_t, multimap_t，只是它们的 key 和 value 是一样的都是数据本身。

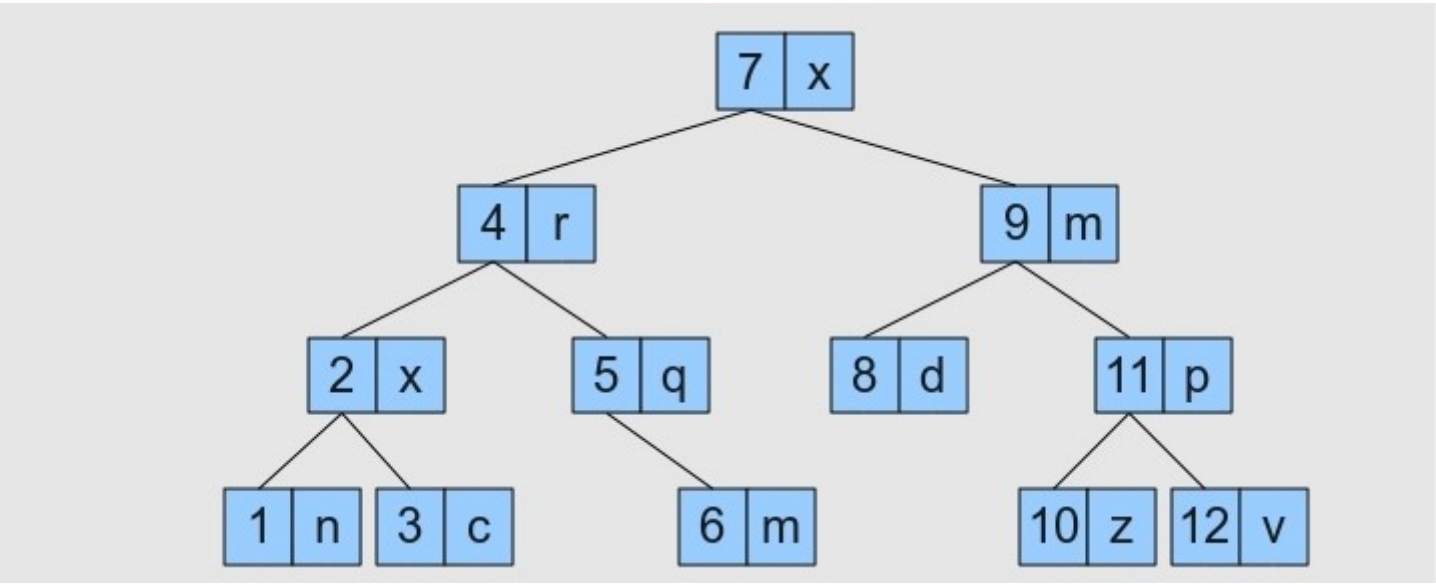


图 4.9 map_t 和 multimap_t 的内部结构

容器根据数据的 key 排序数据，所以根据某个 key 查找数据很高效，但是根据某个 value 查找数据效率很低。map_t 和 multimap_t 具有与 set_t 和 multiset_t 相同的操作函数，同时也具有同样的特定：不能直接或间接修改容器中的数据(map_t 和 multimap_t 中是不能修改数据的 key，但是 value 可以修改)。

2. map_t 和 multimap_t 的操作

map_t 和 multimap_t 的操作都是相同的，只有一个地方不同，会在后面的章节介绍。下面只列出了 map_t 的相关的操作函数，具体的请参考《The libcstl Library Reference Manual》。

● 创建, 初始化和销毁操作	
create_map	创建 map_t 容器类型。
map_init	使用默认的排序规则，初始化一个空的 map_t 容器。
map_init_ex	使用指定的排序规则，初始化一个空的 map_t 容器。
map_init_copy	通过拷贝的方式初始化一个 map_t 容器，数据的排序规则都是来源于源 map_t 容器。
map_init_copy_range	使用指定的数据区间和默认的排序规则初始化一个 map_t 容器。
map_init_copy_range_ex	使用指定的数据区间和指定的排序规则初始化一个 map_t 容器。

map_destroy	销毁 map_t 容器。
-------------	--------------

map_t 和 multimap_t 保存的都是 key/value，所以在创建的时候 key 和 value 的类型都要指明。

● 非质变操作

map_size	返回 map_t 中数据的数量。
map_empty	测试 map_t 是否为空。
map_max_size	返回 map_t 中能够保存的数据数量的最大值。
map_equal	测试两个 map_t 是否相等。
map_not_equal	测试两个 map_t 是否不等。
map_less	测试第一个 map_t 是否小于第二个 map_t。
map_less_equal	测试第一个 map_t 是否小于等于第二个 map_t。
map_greater	测试第一个 map_t 是否大于第二个 map_t。
map_greater_equal	测试第一个 map_t 是否大于等于第二个 map_t。

● 特殊的查找函数

map_t 和 multimap_t 同样提供了特殊查找函数，但是这些函数是基于数据的 key，例如要查找指定的 key 使用 map_find() 操作函数，但是如果查找指定的 value 就不能使用 map_find() 了，使用 algo_find_if() 可用做到，但是没有 map_find() 高效。

map_find	返回指向 map_t 中包含指定 key 的数据的迭代器。
map_count	返回 map_t 中包含指定 key 的数据的个数。
map_equal_range	返回 map_t 中包含指定 key 的数据的数据区间。
map_lower_bound	返回指向 map_t 中第一个包含指定 key 的数据的迭代器。
map_upper_bound	返回指向 map_t 中第一个包含大于指定 key 的数据的迭代器。

● 赋值和交换

map_assign	使用既存的 map_t 为当前的 map_t 赋值。
map_swap	交换两个 map_t 中的内容。

● 与迭代器相关的操作函数

map_begin	返回指向 map_t 中第一个数据的迭代器。
map_end	返回指向 map_t 末尾位置的迭代器。

map_t 和 multimap_t 提供双向迭代器。禁止通过迭代器操作间接的修改容器中数据的 key，但是可以修改数据的 value，同样禁止将质变算法作用于关联容器。

● 插入和删除

map_insert	向 map_t 中插入指定的数据。
------------	-------------------

<code>map_insert_hint</code>	向 <code>map_t</code> 中插入指定的数据，同时给出位置提示。
<code>map_insert_range</code>	向 <code>map_t</code> 中插入指定的数据区间。
<code>map_erase</code>	删除 <code>map_t</code> 中拥有指定 <code>key</code> 的数据。
<code>map_erase_pos</code>	删除 <code>map_t</code> 中指定位置的数据。
<code>map_erase_range</code>	删除 <code>map_t</code> 中指定数据区间的数据。
<code>map_clear</code>	删除 <code>map_t</code> 中所有的数据。

由于 `map_t` 不允许 `key` 重复而 `multimap_t` 允许 `key` 重复，那么在插入和删除操作中两个容器的操作函数的行为就有区别了，对于插入操作，`map_t` 中如果已经存在与要插入的数据的 `key` 相等的 `key` 则插入会失败，返回 `map_end()` 但是对于 `multiset_t` 这样的插入就不会失败，返回的是被插入的数据在 `multiset_t` 中位置的迭代器。同时对于删除操作 `map_t` 将容器中包含指定 `key` 的唯一数据删除，但是 `multiset_t` 将容器中所有包含指定 `key` 的数据都删掉了。

3. 将 `map_t` 作为关联数组使用

关联容器是不提供数据访问操作的，但是 `map_t` 例外，它提供了一个通过下标对数据进行随即访问的操作函数，所以可以将 `map_t` 看作是关联数组。

<code>map_at</code>	通过以 <code>key</code> 为下标对 <code>map_t</code> 中的数据的 <code>value</code> 进行访问。
---------------------	---

这个操作函数使用 `key` 为下标，返回指向数据的 `value` 的指针。如果容器中没有以 `key` 为键值的数据则先向容器中插入一个键值为 `key`，`value` 为该数据类型的默认值的数据然后返回指向 `value` 的指针。

例如：

```
map_t* pmap_coll = create_map(char, double);
...
map_init(pmap_coll);
...
*(double*)map_at(pmap_coll, 'A') = 7.7;
```

这个例子中 `*(double*)map_at(pmap_coll, 'A') = 7.7;` 是对键值为 'A' 的数据的访问，如果容器中包含键值为 'A' 的数据，就将这个数据的 `value` 改为 7.7，如果没有就插入一个 ('A', 0.0) 的数据，然后在将 0.0 改为 7.7。

这个操作有一个缺点，那就是调用者可能偶然或者错误的插了新数据。如：

```
printf("value = %f\n", *(double*)map_at(pmap_coll, 'A'));
```

如果容器中有键值为 'A' 的数据这个操作没有任何错误，但是如果没有那么就无声无息的插入了一个新数据 ('A', 0.0)。

4. `map_t` 和 `multimap_t` 的使用实例

`map_t` 作为关联数组，这个例子使用 `map_t` 来反应股价，其中 `key` 是公司名称，`value` 是股价：

```
/*
 * map2.c
```

```

* compile with : -lcstl
*/

#include <stdio.h>
#include <cstl/cmap.h>

int main(int argc, char* argv[])
{
    map_t* pmap_stocks = create_map(char*, double);
    map_iterator_t it_pos;

    if(pmap_stocks == NULL)
    {
        return -1;
    }

    map_init(pmap_stocks);

    *(double*)map_at(pmap_stocks, "Google") = 834.50;
    *(double*)map_at(pmap_stocks, "IBM") = 431.93;
    *(double*)map_at(pmap_stocks, "Apple") = 557.28;
    *(double*)map_at(pmap_stocks, "MS") = 691.03;
    *(double*)map_at(pmap_stocks, "Oracle") = 670.37;

    /* print all elements */
    for(it_pos = map_begin(pmap_stocks);
        !iterator_equal(it_pos, map_end(pmap_stocks));
        it_pos = iterator_next(it_pos))
    {
        printf("stock:%s\tprice:%lf\n",
            (char*)pair_first((pair_t*)iterator_get_pointer(it_pos)),
            *(double*)pair_second((pair_t*)iterator_get_pointer(it_pos)));
    }
    printf("\n");

    /* all price double */
    for(it_pos = map_begin(pmap_stocks);
        !iterator_equal(it_pos, map_end(pmap_stocks));
        it_pos = iterator_next(it_pos))
    {
        *(double*)pair_second((pair_t*)iterator_get_pointer(it_pos)) *= 2;
    }
}

```

```

/* print all elements */
for(it_pos = map_begin(pmap_stocks);
    !iterator_equal(it_pos, map_end(pmap_stocks));
    it_pos = iterator_next(it_pos))
{
    printf("stock:%s\tprice:%lf\n",
        (char*)pair_first((pair_t*)iterator_get_pointer(it_pos)),
        *(double*)pair_second((pair_t*)iterator_get_pointer(it_pos)));
}
printf("\n");

/* rename the key from "MS" to "Microsoft" */
*(double*)map_at(pmap_stocks, "Microsoft") =
    *(double*)map_at(pmap_stocks, "MS");
map_erase(pmap_stocks, "MS");

/* print all elements */
for(it_pos = map_begin(pmap_stocks);
    !iterator_equal(it_pos, map_end(pmap_stocks));
    it_pos = iterator_next(it_pos))
{
    printf("stock:%s\tprice:%lf\n",
        (char*)pair_first((pair_t*)iterator_get_pointer(it_pos)),
        *(double*)pair_second((pair_t*)iterator_get_pointer(it_pos)));
}
printf("\n");

map_destroy(pmap_stocks);

return 0;
}

```

输出结果:

stock:Apple	price:557.280000
stock:Google	price:834.500000
stock:IBM	price:431.930000
stock:MS	price:691.030000
stock:Oracle	price:670.370000

stock:Apple	price:1114.560000
stock:Google	price:1669.000000
stock:IBM	price:863.860000
stock:MS	price:1382.060000

stock:Oracle price:1340.740000

stock:Apple price:1114.560000

stock:Google price:1669.000000

stock:IBM price:863.860000

stock:Microsoft price:1382.060000

stock:Oracle price:1340.740000

下面这个例子将 `multimap_t` 作为字典来使用:

```
/*
 * multimap2.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <string.h>
#include <cstl/cmap.h>

int main(int argc, char* argv[])
{
    multimap_t* pmmmap_dict = create_multimap(char*, char*);
    pair_t* ppair_item = create_pair(char*, char*);
    multimap_iterator_t it_pos;

    if(pmmmap_dict == NULL || ppair_item == NULL)
    {
        return -1;
    }

    multimap_init(pmmmap_dict);
    pair_init(ppair_item);

    pair_make(ppair_item, "day", "Tag");
    multimap_insert(pmmmap_dict, ppair_item);
    pair_make(ppair_item, "strange", "fremd");
    multimap_insert(pmmmap_dict, ppair_item);
    pair_make(ppair_item, "car", "Auto");
    multimap_insert(pmmmap_dict, ppair_item);
    pair_make(ppair_item, "smart", "elegant");
    multimap_insert(pmmmap_dict, ppair_item);
    pair_make(ppair_item, "trait", "Merkmal");
    multimap_insert(pmmmap_dict, ppair_item);
    pair_make(ppair_item, "strange", "seltsam");
```



```

multimap_insert(pmmmap_dict, ppair_item);
pair_make(ppair_item, "smart", "raffiniert");
multimap_insert(pmmmap_dict, ppair_item);
pair_make(ppair_item, "smart", "klug");
multimap_insert(pmmmap_dict, ppair_item);
pair_make(ppair_item, "clever", "raffiniert");
multimap_insert(pmmmap_dict, ppair_item);

/* print all elements */
printf("english\t\tgerman\n");
printf("-----\n");
for(it_pos = multimap_begin(pmmmap_dict);
    !iterator_equal(it_pos, multimap_end(pmmmap_dict));
    it_pos = iterator_next(it_pos))
{
    printf("%s\t\t%s\n",
        (char*)pair_first((pair_t*)iterator_get_pointer(it_pos)),
        (char*)pair_second((pair_t*)iterator_get_pointer(it_pos)));
}
printf("\n");

/* print all values for key "smart" */
printf("smart:\n");
for(it_pos = multimap_begin(pmmmap_dict);
    !iterator_equal(it_pos, multimap_end(pmmmap_dict));
    it_pos = iterator_next(it_pos))
{
    if(strcmp((char*)pair_first((pair_t*)iterator_get_pointer(it_pos)),
        "smart") == 0)
    {
        printf("\t%s\n",
            (char*)pair_second((pair_t*)iterator_get_pointer(it_pos)));
    }
}
printf("\n");

/* print all keys for value "raffiniert" */
printf("raffiniert:\n");
for(it_pos = multimap_begin(pmmmap_dict);
    !iterator_equal(it_pos, multimap_end(pmmmap_dict));
    it_pos = iterator_next(it_pos))
{
    if(strcmp((char*)pair_second((pair_t*)iterator_get_pointer(it_pos)),

```

```

        "raffiniert") == 0)
    {
        printf("\t%s\n",
            (char*)pair_first((pair_t*)iterator_get_pointer(it_pos)));
    }
}
printf("\n");

multimap_destroy(pmmmap_dict);
pair_destroy(ppair_item);

return 0;
}

```

结果:

english	german

car	Auto
clever	raffiniert
day	Tag
smart	elegant
smart	raffiniert
smart	klug
strange	fremd
strange	seltsam
trait	Merkmal

smart:

elegant
 raffiniert
 klug

raffiniert:

clever
 smart

下面这个例子展示了通过算法来找到 value:

```

/*
 * mapfind.c
 * compile with : -lcstl
 */

#include <stdio.h>

```

```

#include <cstl/cmap.h>
#include <cstl/calgorithm.h>

static void _find_valud(const void* cpv_input, void* pv_output)
{
    if(*(int*)pair_second((pair_t*)cpv_input) == 3)
    {
        *(bool_t*)pv_output = true;
    }
    else
    {
        *(bool_t*)pv_output = false;
    }
}

int main(int argc, char* argv[])
{
    map_t* pmap_coll = create_map(int, int);
    map_iterator_t it_pos;

    if(pmap_coll == NULL)
    {
        return -1;
    }

    map_init(pmap_coll);

    *(int*)map_at(pmap_coll, 1) = 7;
    *(int*)map_at(pmap_coll, 2) = 4;
    *(int*)map_at(pmap_coll, 3) = 2;
    *(int*)map_at(pmap_coll, 4) = 3;
    *(int*)map_at(pmap_coll, 5) = 6;
    *(int*)map_at(pmap_coll, 6) = 1;
    *(int*)map_at(pmap_coll, 7) = 7;

    /* search an element with key 3 */
    it_pos = map_find(pmap_coll, 3);
    if(!iterator_equal(it_pos, map_end(pmap_coll)))
    {
        printf("%d : %d\n",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_pos)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_pos)));
    }
}

```

```

/* search an element with value 3 */
it_pos = algo_find_if(map_begin(pmap_coll), map_end(pmap_coll), _find_valud);
if(!iterator_equal(it_pos, map_end(pmap_coll)))
{
    printf("%d : %d\n",
        *(int*)pair_first((pair_t*)iterator_get_pointer(it_pos)),
        *(int*)pair_second((pair_t*)iterator_get_pointer(it_pos)));
}

map_destroy(pmap_coll);

return 0;
}

```

结果:

3:2

4:3

第八节 hash_set_t, hash_multiset_t, hash_map_t, hash_multimap_t

libcstl 除了提供基于平衡二叉树的关联容器外还提供了基于哈希表结构的关联容器。基于哈希表结构的关联容器在数据的插入和查找方面效率更高，如果哈希函数选择的好的话可以实现操作时间接近常数级别。基于哈希表结构的关联容器中数据是按照哈希函数的计算得到数据在哈希表相应的存储位置。当计算多个数据通过哈希函数计算时得到相同的位置就产生了冲突，为了解决冲突将这些数据按照排序规则组成一个链存放在相应的哈希表存储单元中，好的哈希函数应该尽量减少冲突。

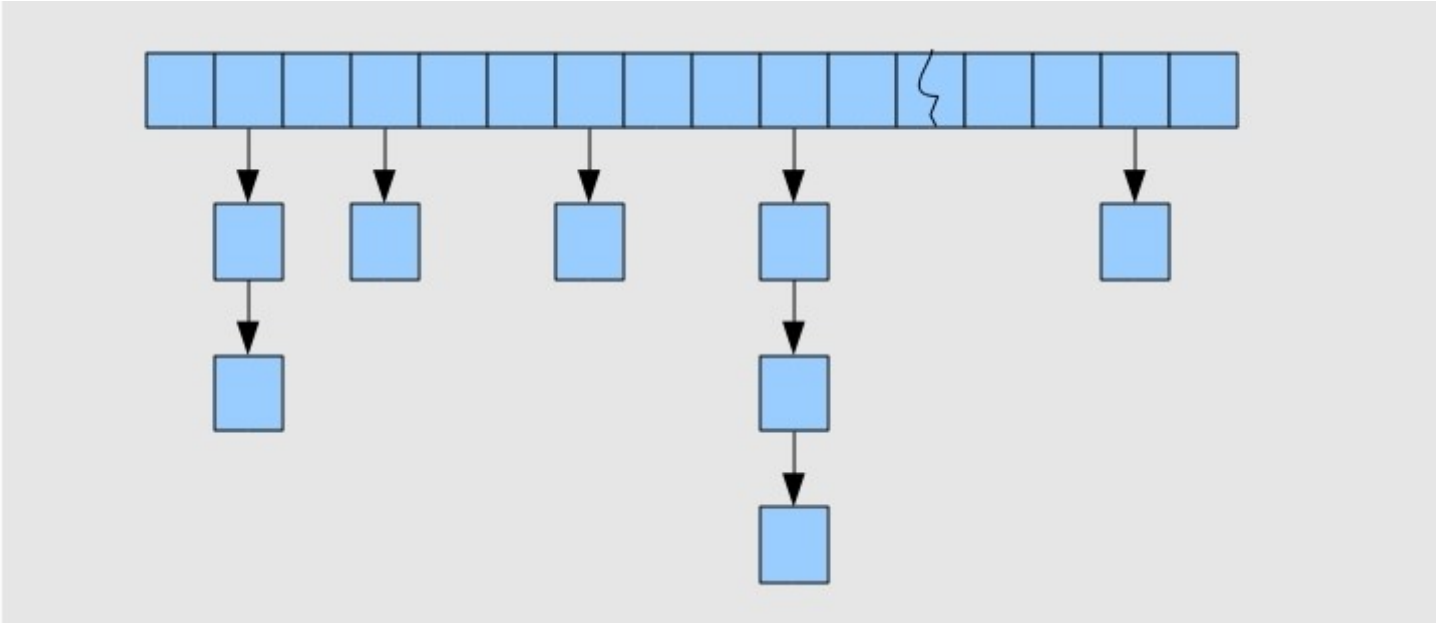


图 4.10 哈希表内部结构

libcstl 提供的基于哈希结构的关联容器包括 `hash_set_t`, `hash_multiset_t`, `hash_map_t`, `hash_multimap_t`。这几种容器与基于平衡二叉树的关联容器的操作基本相同。

要使用 `hash_set_t` 和 `hash_multiset_t` 必须包含头文件 `<cstl/chash_set.h>`

```
#include <cstl/chash_set.h>
```

要使用 `hash_map_t` 和 `hash_multimap_t` 必须包含头文件 `<cstl/chash_map.h>`

```
#include <cstl/chash_map.h>
```

1. 基于哈希结构的关联容器的能力

基于哈希结构的关联容器的采用哈希表结构作为底层实现，这样是容器在数据的插入，删除和查找有更高的效率。此外基于哈希结构的关联容器支持除了同样支持关联容器的操作外还提供了关于哈希表的以下操作函数。

基于哈希结构的关联容器提供双向的迭代器，容器中的数据虽然不是完全有序的但是它的位置是根据数据本身的值和哈希函数计算得到的，所以不要直接或者通过迭代器间接修改数据的值，因为这样会破坏数据位置的正确性。

2. 基于哈希结构的关联容器的操作

基于哈希结构的关联容器的操作大部分与基于平衡二叉树的关联容器操作相同，这里以 `hash_set_t` 的操作为例，其他操作函数请参考《The libcstl Library Reference Manual》。

- 创建，初始化和销毁操作

<code>create_hash_set</code>	创建 <code>hash_set_t</code> 容器类型。
<code>hash_set_init</code>	使用的默认的哈希函数，排序规则和哈希表存储单元数初始化一个空的 <code>hash_set_t</code> 。
<code>hash_set_init_ex</code>	使用指定的哈希函数排序规则和存储单元数初始化一个空的 <code>hash_set_t</code> 。

hash_set_init_copy	通过拷贝的方式初始化一个 hash_set_t。
hash_set_init_copy_range	使用默认的哈希函数排序规则存储单元数目和指定的数据区间初始化 hash_set_t。
hash_set_init_copy_range_ex	使用指定的哈希函数排序规则存储单元数目和指定的数据区间初始化 hash_set_t。
hash_set_destroy	销毁 hash_set_t。

● 非质变操作

hash_set_size	返回 hash_set_t 中数据的数量。
hash_set_empty	测试 hash_set_t 是否为空。
hash_set_max_size	返回 hash_set_t 中能够保存的数据数量的最大值。
hash_set_equal	测试两个 hash_set_t 是否相等。
hash_set_not_equal	测试两个 hash_set_t 是否不等。
hash_set_less	测试第一个 hash_set_t 是否小于第二个 hash_set_t。
hash_set_less_equal	测试第一个 hash_set_t 是否小于等于第二个 hash_set_t。
hash_set_greater	测试第一个 hash_set_t 是否大于第二个 hash_set_t。
hash_set_greater_equal	测试第一个 hash_set_t 是否大于等于第二个 hash_set_t。

● 与哈希表有关的操作函数

hash_set_bucket_count	返回 hash_set_t 中哈希表存储单元的数量。
hash_set_hash	返回 hash_set_t 使用的哈希函数。
hash_set_resize	重新设置 hash_set_t 中哈希表存储单元的个数。

这些操作函数都是与哈希表相关的都是基于哈希结构的关联容器特有的操作函数。

● 特殊的查找函数

基于哈希结构的关联容器同样提供了特殊查找函数，但是没有 set_lower_bound()和 set_upper_bound()。因为只两个操作是基于有序的容器数据的。但是基于哈希结构的关联容器中的数据是无序的。

hash_set_find	在 hash_set_t 中查找指定的数据。
hash_set_count	统计 hash_set_t 中包含指定数据的数量。
hash_set_equal_range	返回 hash_set_t 中包含指定数据的数据区间。

● 赋值和交换

hash_set_assign	使用既存的 hash_set_t 为当前的 hash_set_t 赋值。
hash_set_swap	交换两个 hash_set_t 中的内容。

● 与迭代器相关的操作函数

hash_set_begin	返回指向 hash_set_t 中第一个数据的迭代器。
----------------	-----------------------------

hash_set_end	返回指向 hash_set_t 的末尾位置的迭代器。
--------------	----------------------------

● 插入和删除

hash_set_insert	向 hash_set_t 中插入指定的数据。
hash_set_insert_range	向 hash_set_t 中插入指定的数据区间。
hash_set_erase	将 hash_set_t 指定的数据删除。
hash_set_erase_pos	删除 hash_set_t 中指定位置的数据。
hash_set_erase_range	删除 hash_set_t 中指定数据区间的数据。
hash_set_clear	删除 hash_set_t 中的所有数据。

3. 将 hash_map_t 作为关联数组使用

hash_map_t 与 map_t 相同可以作为关联数组使用，它们的行为也是相同的。

hash_map_at	使用 key 为下标访问 hash_map_t 中相应数据的 value。
-------------	---------------------------------------

4. 基于哈希结构的关联容器的使用实例

第一个例子展示了容器中的数据存储情况：

```
/*
 * hash_set1.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/chash_set.h>

int main(int argc, char* argv[])
{
    hash_set_t* phset_coll = create_hash_set(int);
    hash_set_iterator_t it_pos;
    int i = 0;

    if(phset_coll == NULL)
    {
        return -1;
    }
}
```

```

}

hash_set_init(phset_coll);

hash_set_insert(phset_coll, 7);
hash_set_insert(phset_coll, 99);
hash_set_insert(phset_coll, 63);
hash_set_insert(phset_coll, 2);
hash_set_insert(phset_coll, 108);
hash_set_insert(phset_coll, 444);
hash_set_insert(phset_coll, 53);
hash_set_insert(phset_coll, 55);
hash_set_insert(phset_coll, 255);
hash_set_insert(phset_coll, 55);

printf("bucket count : %u\n", hash_set_bucket_count(phset_coll));
for(it_pos = hash_set_begin(phset_coll);
    !iterator_equal(it_pos, hash_set_end(phset_coll));
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");

for(i = 0; i < 50; ++i)
{
    hash_set_insert(phset_coll, i);
}

printf("bucket count : %u\n", hash_set_bucket_count(phset_coll));
for(it_pos = hash_set_begin(phset_coll);
    !iterator_equal(it_pos, hash_set_end(phset_coll));
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");

hash_set_destroy(phset_coll);

return 0;
}

```


结果:

bucket count : 53

53 2 108 55 7 63 444 255 99

bucket count : 97

0 1 2 99 3 4 5 6 7 8 9 10 108 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 255 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 53 55 63 444

从结果中我们看出容器中的数据并不是排序的而是按照某些规则来计算哈希位置的。

第二个例子将上面的股票的例子使用 hash_map_t 改写:

```
/*
 * hash_map1.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/chash_map.h>

int main(int argc, char* argv[])
{
    hash_map_t* phmap_stocks = create_hash_map(char*, double);
    hash_map_iterator_t it_pos;

    if(phmap_stocks == NULL)
    {
        return -1;
    }

    hash_map_init(phmap_stocks);

    *(double*)hash_map_at(phmap_stocks, "Google") = 834.50;
    *(double*)hash_map_at(phmap_stocks, "IBM") = 431.93;
    *(double*)hash_map_at(phmap_stocks, "Apple") = 557.28;
    *(double*)hash_map_at(phmap_stocks, "MS") = 691.03;
    *(double*)hash_map_at(phmap_stocks, "Oracle") = 670.37;

    /* print all elements */
    for(it_pos = hash_map_begin(phmap_stocks);
        !iterator_equal(it_pos, hash_map_end(phmap_stocks));
        it_pos = iterator_next(it_pos))
    {
        printf("stock:%s\tprice:%lf\n",
            (char*)pair_first((pair_t*)iterator_get_pointer(it_pos)),
            *(double*)pair_second((pair_t*)iterator_get_pointer(it_pos)));
    }
}
```

```

}
printf("\n");

/* all price double */
for(it_pos = hash_map_begin(phmap_stocks);
    !iterator_equal(it_pos, hash_map_end(phmap_stocks));
    it_pos = iterator_next(it_pos))
{
    *(double*)pair_second((pair_t*)iterator_get_pointer(it_pos)) *= 2;
}

/* print all elements */
for(it_pos = hash_map_begin(phmap_stocks);
    !iterator_equal(it_pos, hash_map_end(phmap_stocks));
    it_pos = iterator_next(it_pos))
{
    printf("stock:%s\tprice:%lf\n",
        (char*)pair_first((pair_t*)iterator_get_pointer(it_pos)),
        *(double*)pair_second((pair_t*)iterator_get_pointer(it_pos)));
}
printf("\n");

/* rename the key from "MS" to "Microsoft" */
*(double*)hash_map_at(phmap_stocks, "Microsoft") =
    *(double*)hash_map_at(phmap_stocks, "MS");
hash_map_erase(phmap_stocks, "MS");

/* print all elements */
for(it_pos = hash_map_begin(phmap_stocks);
    !iterator_equal(it_pos, hash_map_end(phmap_stocks));
    it_pos = iterator_next(it_pos))
{
    printf("stock:%s\tprice:%lf\n",
        (char*)pair_first((pair_t*)iterator_get_pointer(it_pos)),
        *(double*)pair_second((pair_t*)iterator_get_pointer(it_pos)));
}
printf("\n");

hash_map_destroy(phmap_stocks);

return 0;

```

```

}

```

```
输出结果：
stock:MS      price:691.030000
stock:IBM     price:431.930000
stock:Oracle  price:670.370000
stock:Apple   price:557.280000
stock:Google  price:834.500000

stock:MS      price:1382.060000
stock:IBM     price:863.860000
stock:Oracle  price:1340.740000
stock:Apple   price:1114.560000
stock:Google  price:1669.000000

stock:IBM     price:863.860000
stock:Oracle  price:1340.740000
stock:Apple   price:1114.560000
stock:Google  price:1669.000000
stock:Microsoft price:1382.060000
```

第九节 怎样选择容器类型

libcstl 提供了很多容器类型，它们拥有不同的能力和特点，能够满足各种不同的需求。怎么选择使用哪种容器类型就成为了一个新问题。下面的列表提供了一个总体概括：

	vector_t	deque_t	list_t	slist_t
典型的数据结构	动态数组	分段的动态数据	双向链表	单向链表
数据形式	value	value	value	value
是否允许数据重复	是	是	是	是
是否支持随机访问	是	是	否	否
迭代器类型	随机	随机	双向	单向
数据查找速度	慢	慢	慢	慢
快速插入删除数据的位置	末尾	开头和末尾	任何位置	任意位置后面
插入删除数据导致迭代器失效	重新分配内存	任何时候	不会失效	不会失效
释放被删除数据的内存	否	有时是	是	是
	set_t	multiset_t	map_t	multimap_t
典型的数据结构	平衡二叉树	平衡二叉树	平衡二叉树	平衡二叉树
数据形式	value	value	key/value	key/value

是否允许数据重复	否	是	不允许 key 重复	是
是否支持随机访问	否	否	使用 key 随机访问	否
迭代器类型	双向	双向	双向	双向
数据查找速度	快	快	查找 key 快	查找 key 快
快速插入删除数据的位置	-	-	-	-
插入删除数据导致迭代器失效	不会失效	不会失效	不会失效	不会失效
释放被删除数据的内存	是	是	是	是
	hash_set_t	hash_multiset_t	hash_map_t	hash_multimap_t
典型的数据结构	哈希表	哈希表	哈希表	哈希表
数据形式	value	value	key/value	key/value
是否允许数据重复	否	是	不允许 key 重复	是
是否支持随机访问	否	否	使用 key 随机访问	否
迭代器类型	双向	双向	双向	双向
数据查找速度	非常快	非常快	查找 key 非常快	查找 key 非常快
快速插入删除数据的位置	-	-	-	-
插入删除数据导致迭代器失效	存储单元变化	存储单元变化	存储单元变化	存储单元变化
释放被删除数据的内存	是	是	是	是

- 在预分配存储的情况下使用 `vector_t`，它简单并且提供了对数据的随即访问，使用起来方便。
- 如果经常在容器的开头和结尾插入数据，那么使用 `deque_t` 它不仅可以在开头和结尾高效的插入和删除数据而且也提供了对数据的随即访问，在删除数据的时候有时内存还会被释放。
- 如果经常在容器的任意位置插入删除或移动数据，那么使用 `list_t` 它可以高效的在任意位置插入或删除数据，并且不会影响已有的迭代器。如果要在任意位置的下一个位置插入或删除数据，那么使用 `slist_t`。
- 如果经常查找或者数据的有序性很重要，那么使用 `set_t` 或 `multiset_t`，如果只关心查找的效率，那么 `hash_set_t` 和 `hash_multiset_t` 也可以使用。如果数据是 `key/value` 形式的那么使用 `map_t`，`multimap_t` 或者 `hash_map_t`，`hash_multimap_t`。如果要使用关联数组，那么使用 `map_t`，`hash_map_t`。

下面是两个相同的例子都是对输入的一系列数据进行排序，使用了两个不同的容器：
使用 `vector_t`：

```
/*
 * sortvec.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>
```

```

#include <cstl/cfunctional.h>

int main(int argc, char* argv[])
{
    vector_t* pvec_coll = create_vector(int);
    int n_random = 0;
    int n_input = 1000000;
    int i = 0;

    if(pvec_coll == NULL)
    {
        return -1;
    }

    vector_init_n(pvec_coll, 1000000);

    for(i = 0; i < 1000000; ++i)
    {
        fun_random_number(&n_input, &n_random);
        *(int*)vector_at(pvec_coll, i) = n_random;
    }

    algo_sort(vector_begin(pvec_coll), vector_end(pvec_coll));

    vector_destroy(pvec_coll);

    return 0;
}

```

使用 time 命令测试执行结果:

```

real    0m51.613s
user    0m49.116s
sys     0m0.371s

```

使用 multiset_t:

```

/*
 * sortmset.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cset.h>
#include <cstl/cfunctional.h>

```

```

int main(int argc, char* argv[])
{
    multiset_t* pmset_coll = create_multiset(int);
    int n_randmo = 0;
    int n_input = 1000000;
    int i = 0;

    if(pmset_coll == NULL)
    {
        return -1;
    }

    multiset_init(pmset_coll);

    for(i = 0; i < 1000000; ++i)
    {
        fun_random_number(&n_input, &n_randmo);
        multiset_insert(pmset_coll, n_randmo);
    }

    multiset_destroy(pmset_coll);

    return 0;
}

```

使用 time 命令测试执行结果：

```

real    0m11.245s
user    0m10.545s
sys     0m0.356s

```

由这个统计数据可以看出一切！

第五章 libcstl 迭代器

libcstl 容器都定义了相关的迭代器类型，所以在使用迭代器的时候并不需要包含特殊的头文件，但是除了各个容器定义的相关的迭代器外，libcstl 还提供了很多类型的迭代器，这些类型的迭代器都定义在<cstl/citerator.h>中。要使用这些额外的迭代器类型就要包含这个头文件。

```
#include <cstl/citerator.h>
```

第一节 迭代器的种类

迭代器是一种表示数据位置的类型，它可以被用来访问数据等其他的一些操作。libcstl 提供了一组统一的迭代器操作函数，通过统一的操作来实现在各种容器或者数据区间上的迭代器。但是由于各个容器的结构不同，迭代器也有了不同的类型，有些算法要求特殊的迭代器类型，例如 algo_sort() 算法要求具有随机访问能力的迭代器，这样算法可以达到最高的小路。下面的列表类出的迭代器的种类：

迭代器类型	能力	相关容器
iterator_t	[1]	-
input_iterator_t	输入，先前迭代	-
output_iterator_t	输出，向前迭代	-
forward_iterator_t	输入，输出，向前迭代	slist_t
bidirectional_iterator_t	输入，输出，双向迭代	list_t, set_t, multiset_t, map_t, multimap_t, hash_set_t, hash_multiset_t, hash_map_t, hash_multimap_t
random_access_iterator_t	输入，输出，随机迭代	vector_t, deque_t, string_t

表中的 iterator_t 是迭代器的基本类型, 下面的所有的类型都是从它演变出来的，它可以表示所有的迭代器类型，所以在实际使用的过程中可以使用 iterator_t 类型取代实际的与迭代器相关的类型。iterator_t 可以接受任何迭代器类型的赋值，赋值后原来的 iterator_t 类型也就变成了相应的类型。如：

```
iterator_t it_i;
bidirectional_iterator_t it_bi;
it_i = it_bi;
```

在赋值之前 it_i 是 iterator_t 类型，它不具有任何迭代器能力，但是赋值之后 it_i 变成了 bidirectional_iterator_t 类型，具有了与 it_bi 同样的能了。所有的迭代器类型都支持赋值操作符，赋值后迭代器类型也相应的改变了。

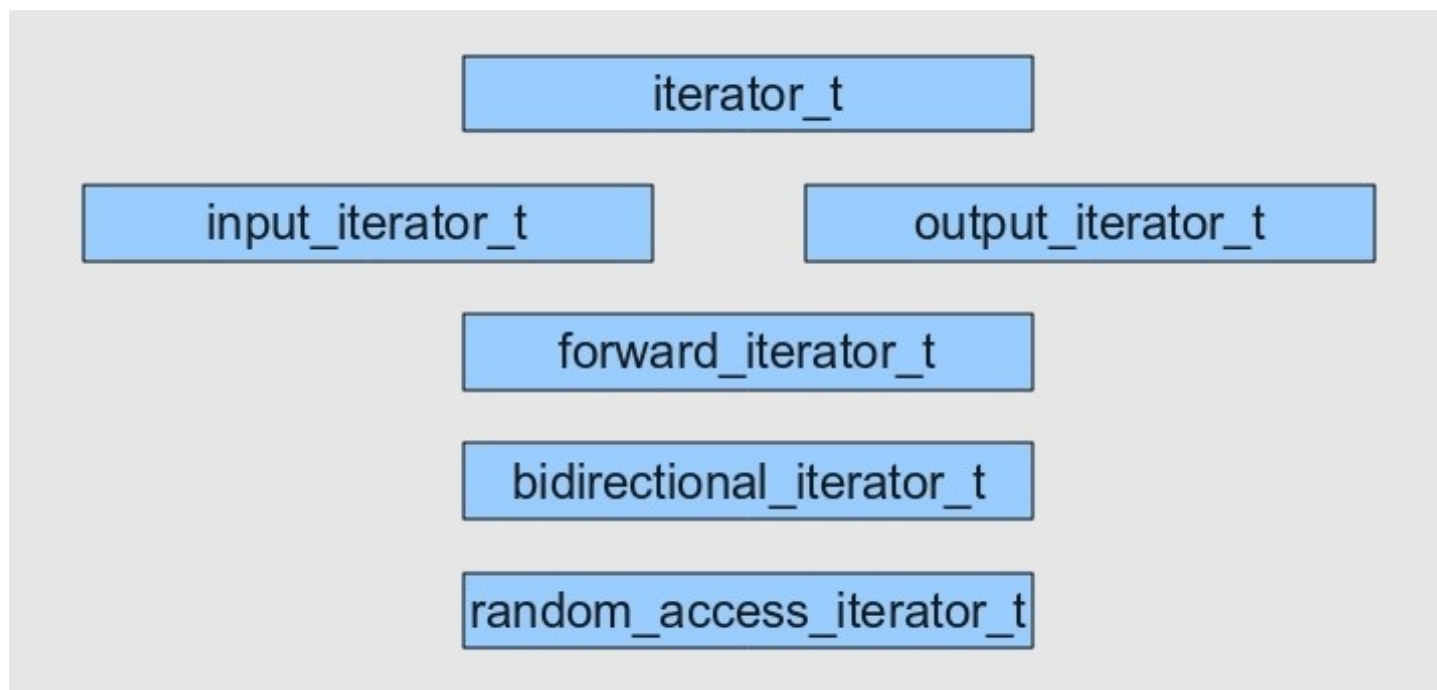


图 5.1 迭代器的种类

上图展示了各个迭代器类型之间的关系，它们的能力是相互包含的,越往下能力越强。除了上面的迭代器类型之外，各个容器也定义了迭代器类型，其实这些类型就是上面迭代器类型的别名，并不是新类型,下面是一个对比表格：

容器定义的迭代器类型	对应的迭代器类型
vector_iterator_t	random_access_iterator_t
deque_iterator_t	random_access_iterator_t
list_iterator_t	bidirectional_iterator_t
slist_iterator_t	forward_iterator_t
string_iterator_t	random_access_iterator_t
set_iterator_t	bidirectional_iterator_t
multiset_iterator_t	bidirectional_iterator_t
map_iterator_t	bidirectional_iterator_t
multimap_iterator_t	bidirectional_iterator_t
hash_set_iterator_t	bidirectional_iterator_t
hash_multiset_iterator_t	bidirectional_iterator_t
hash_map_iterator_t	bidirectional_iterator_t
hash_multimap_iterator_t	bidirectional_iterator_t

1. `input_iterator_t`

`input_iterator_t` 类型的迭代器只能一个数据一个数据的向前移动，同时还支持通过迭代器获得数据，比较两个 `input_iterator_t` 类型的迭代器是否相等。

下面是 `input_iterator_t` 类型的迭代器可以使用的操作函数：

<code>iterator_get_value</code>	获得迭代器指向的数据。
<code>iterator_get_pointer</code>	返回迭代器指向的数据的指针。
<code>iterator_next</code>	返回指向下一个数据的迭代器。
<code>iterator_equal</code>	测试两个迭代器是否相等。
<code>iterator_not_equal</code>	测试两个迭代器是否不等。

2. `output_iterator_t`

`output_iterator_t` 类型的迭代器也支持向前移动，同时它还支持通过迭代器为数据赋值，但是不支持比较两个迭代器是否相等：

<code>iterator_set_value</code>	设置迭代器指向的数据。
<code>iterator_next</code>	返回指向下一个数据的迭代器。

3. `forward_iterator_t`

`forward_iterator_t` 类型的迭代器支持 `input_iterator_t` 类型和 `output_iterator_t` 类型的操作的总和：

<code>iterator_get_value</code>	获得迭代器指向的数据。
<code>iterator_get_pointer</code>	返回迭代器指向的数据的指针。
<code>iterator_set_value</code>	设置迭代器指向的数据。
<code>iterator_next</code>	返回指向下一个数据的迭代器。
<code>iterator_equal</code>	比较两个迭代器是否相等。
<code>iterator_not_equal</code>	比较两个迭代器是否不等。

4. `bidirectional_iterator_t`

`bidirectional_iterator_t` 类型的迭代器是双向迭代器，除了支持 `forward_iterator_t` 类型的迭代器支持的操作外，它还支持向后移动。

<code>iterator_prev</code>	返回指向前一数据的迭代器。
----------------------------	---------------

5. `random_access_iterator_t`

`random_access_iterator_t` 类型的迭代器在 `bidirectional_iterator_t` 类型的基础上又具备了随即访问能力，所以它支持的操作除了 `bidirectional_iterator_t` 类型支持的操作外还支持一次向前移动多个数据，还有比较操作：

<code>iterator_at</code>	通过迭代器随机访问数据。
<code>iterator_next_n</code>	返回指向迭代器向前移动 <code>n</code> 个数据后的位置迭代器。
<code>iterator_prev_n</code>	返回指向迭代器向后移动 <code>n</code> 个数据后的位置迭代器。
<code>iterator_minus</code>	返回两个迭代器的差值。
<code>iterator_less</code>	测试第一个迭代器是否小于第二个迭代器。
<code>iterator_less_equal</code>	测试第一个迭代器是否小于等于第二个迭代器。
<code>iterator_greater</code>	测试第一个迭代器是否大于第二个迭代器。
<code>iterator_greater_equal</code>	测试第一个迭代器是否大于等于第二个迭代器。

下面的例子应用到了 `random_access_iterator_t` 类型的迭代器的特殊操作：

```
/*
 * intercat.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>

int main(int argc, char* argv[])
{
    vector_t* pvec_coll = create_vector(int);
    iterator_t it_pos; /* uses iterator_t instead of vector_iterator_t */
    int i = 0;
    int n_value = 0;

    if(pvec_coll == NULL)
```

```

{
    return -1;
}

vector_init(pvec_coll);

/* insert from -3 to 9 */
for(i = -3; i <= 9; ++i)
{
    vector_push_back(pvec_coll, i);
}

/*
 * print number of elements by processing the distance
 * between vector_begin() and vector_end()
 */
printf("number/distance : %d\n",
       iterator_minus(vector_end(pvec_coll), vector_begin(pvec_coll)));

/*
 * print all elements
 * uses iterator_less instead of !iterator_equal
 */
for(it_pos = vector_begin(pvec_coll);
    iterator_less(it_pos, vector_end(pvec_coll));
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");

/*
 * print all elements
 * uses iterator_at instead of iterator_get_pointer
 */
for(i = 0; i < vector_size(pvec_coll); ++i)
{
    printf("%d ", *(int*)iterator_at(vector_begin(pvec_coll), i));
}
printf("\n");

/* print every second element */
for(it_pos = vector_begin(pvec_coll);

```

```

        iterator_less(it_pos, iterator_prev(vector_end(pvec_coll)));
        it_pos = iterator_next_n(it_pos, 2))
    {
        iterator_get_value(it_pos, &n_value);
        printf("%d ", n_value);
    }
    printf("\n");

    vector_destroy(pvec_coll);

    return 0;
}

```

结果如下:

number/distance : 13

-3 -2 -1 0 1 2 3 4 5 6 7 8 9

-3 -2 -1 0 1 2 3 4 5 6 7 8 9

-3 -1 1 3 5 7

这个例子不能应用于 list_t, map_t, hash_multiset_t 等等这些不是 random_access_iterator_t 的迭代器上。

这个例子中有一段代码:

```

iterator_less(it_pos, iterator_prev(vector_end(pvec_coll)));

```

这是保证迭代器访问数据不越界。这条语句要求 pvec_coll 中至少要包含一个数据，如果 pvec_coll 为空那么 iterator_prev(vector_end(pvec_coll)) 得到的迭代器是指向 vector_begin() 之前的，这样的迭代器行为是未定义的，类似于数组越界的行为，同理如果迭代器指向 vector_end() 之后那么这个迭代器的行为也是未定义的，例如下面的代码:

```

for(it_pos = vector_begin(pvec_coll);
    iterator_less(it_pos, vector_end(pvec_coll));
    it_pos = iterator_next_n(it_pos, 2))
{
    ...
}

```

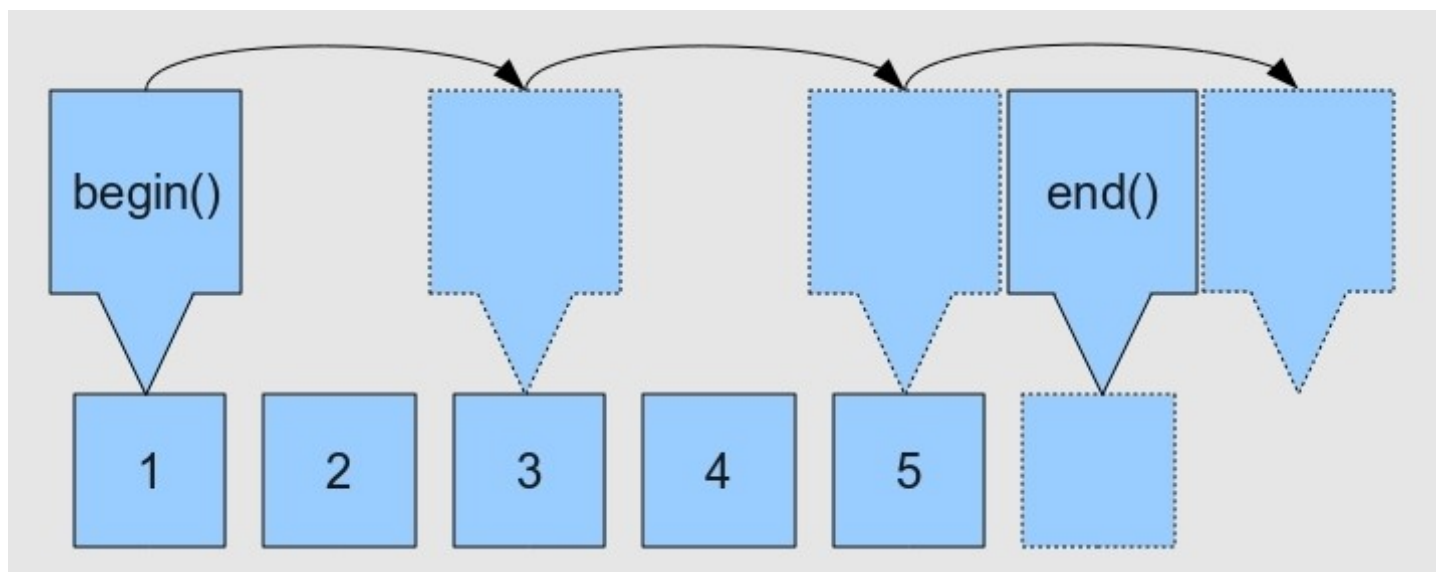


图 5.2 iterator 每次先前移动两个位置

迭代器在向前移动或者向后移动的过程中会发生越界，迭代器移动操作不检测是否越界（带有断言信息的版本会提示越界错误）。所以在移动迭代器时一定要注意，不要越界。

第二节 迭代器的辅助操作

除了 `random_access_iterator_t` 外的迭代器要移动多步或者计算两个迭代器之间的差值只能一个一个的向前或者向后移动吗？答案是否等的，`libcstl` 还提供两个迭代器的辅助操作函数，`iterator_advance()` 和 `iterator_distance()`，这两个操作函数就是为了解决这个问题的。

1. 使用 `iterator_advance()` 移动迭代器

这个辅助函数可以使任何迭代器一次移动多步（向前或向后），当迭代器为双向或随即迭代器时负数步数表示向后移动，否则向前移动，步数为参数的绝对值。

<code>iterator_advance</code>	使迭代器一次向前移动多步。
-------------------------------	---------------

通常来说 `random_access_iterator_t` 的 `iterator_next_n` 和 `iterator_prev_n` 操作要比 `iterator_advance` 效率高很多。

下面是使用 `iterator_advance()` 的一个例子：

```
/*
 * advance1.c
 * compile with : -lcstl
 */

#include <stdio.h>
```

```

#include <cstl/clist.h>

int main(int argc, char* argv[])
{
    list_t* plist_coll = create_list(int);
    iterator_t it_pos;
    int i = 0;

    if(plist_coll == NULL)
    {
        return -1;
    }

    list_init(plist_coll);

    for(i = 1; i <= 9; ++i)
    {
        list_push_back(plist_coll, i);
    }

    it_pos = list_begin(plist_coll);
    /* print actual element */
    printf("%d\n", *(int*)iterator_get_pointer(it_pos));

    /* step three elements forward */
    it_pos = iterator_advance(it_pos, 3);
    /* print actual element */
    printf("%d\n", *(int*)iterator_get_pointer(it_pos));

    /* step one element backward */
    it_pos = iterator_advance(it_pos, -1);
    /* print actual element */
    printf("%d\n", *(int*)iterator_get_pointer(it_pos));

    list_destroy(plist_coll);

    return 0;
}

```

结果:

1
4
3

2. 使用 `iterator_distance()` 计算迭代器之间的距离

这个辅助函数是用来计算任意类型的迭代器之间的距离的，两个迭代器必须属于同一容器。

<code>iterator_distance</code>	计算两个迭代器之间的距离。
--------------------------------	---------------

下面是使用 `iterator_distance()` 的例子：

```
/*
 * distance.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/clist.h>
#include <cstl/calgorithm.h>

int main(int argc, char* argv[])
{
    list_t* plist_coll = create_list(int);
    list_iterator_t it_pos;
    int i = 0;

    if(plist_coll == NULL)
    {
        return -1;
    }

    list_init(plist_coll);

    for(i = -3; i <= 9; ++i)
    {
        list_push_back(plist_coll, i);
    }

    it_pos = algo_find(list_begin(plist_coll), list_end(plist_coll), 5);
    if(iterator_equal(it_pos, list_end(plist_coll)))
    {
        printf("5 not found!\n");
    }
    else
    {
        printf("distance between beginning and 5 : %d\n",
```

```

        iterator_distance(list_begin(plist_coll), it_pos));
    }

    list_destroy(plist_coll);

    return 0;
}

```

结果:
distance between beginning and 5 : 8

第六章 libcstl 函数

第一节 函数的作用

libcstl 提供了大量的函数, 这些函数主要用来为算法提供扩展功能. 算法中每一个算法都有一个后缀为 if 的版本, 这个版本接受函数作为操作的规则.

1. 作为谓词

函数中很大一部分是作为谓词使用的. 例如作为排序的二元谓词:

```

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(int);
    size_t t_limit = 1000;
    size_t t_output = 0;
    int i = 0;

    vector_init(&t_v);

```



```

for(i = 0; i < 10; ++i)
{
    fun_random_number(&t_limit, &t_output);
    vector_push_back(&t_v, t_output);
}
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

algo_sort(vector_begin(&t_v), vector_end(&t_v));
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

algo_random_shuffle(vector_begin(&t_v), vector_end(&t_v));
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

algo_sort_if(vector_begin(&t_v), vector_end(&t_v), fun_great_int);
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

vector_destroy(&t_v);

return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

首先在容器中插入了 10 个随即数. 然后使用 `algo_sort()` 排序, 这个排序算法使用默认的小于规则. 接下来将已经排序的数据大乱, 在使用 `algo_sort_if()` 排序, 这个排序接受一个用户定义排序规则, 例子中使用了 `libcstl` 预定义的二元函数 `fun_great_int`. 则个二元函数的输出是 `bool_t` 类型所以它就是二元谓词. 输出结果:

```

275 388 439 495 412 236 484 548 331 75
75 236 275 331 388 412 439 484 495 548
548 275 75 495 439 331 388 236 412 484
548 495 484 439 412 388 331 275 236 75

```

下面是一个使用一元谓词的例子:

```

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);
static void _mod3(const void* cpv_input, void* pv_output);

```

```

int main(int argc, char* argv[])
{
    vector_t t_v1 = create_vector(int);
    vector_t t_v2 = create_vector(int);
    int i = 0;

    vector_init(&t_v1);
    vector_init(&t_v2);
    for(i = 1; i <= 10; ++i)
    {
        vector_push_back(&t_v1, i);
        vector_push_back(&t_v2, i);
    }
    printf("before remove:\n");
    printf("vector1: ");
    algo_for_each(vector_begin(&t_v1), vector_end(&t_v1), _print);
    printf("\n");
    printf("vector2: ");
    algo_for_each(vector_begin(&t_v2), vector_end(&t_v2), _print);
    printf("\n");

    algo_remove(vector_begin(&t_v1), vector_end(&t_v1), 3);
    algo_remove_if(vector_begin(&t_v2), vector_end(&t_v2), _mod3);
    printf("after remove:\n");
    printf("vector1: ");
    algo_for_each(vector_begin(&t_v1), vector_end(&t_v1), _print);
    printf("\n");
    printf("vector2: ");
    algo_for_each(vector_begin(&t_v2), vector_end(&t_v2), _print);
    printf("\n");

    vector_destroy(&t_v1);
    vector_destroy(&t_v2);

    return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

static void _mod3(const void* cpv_input, void* pv_output)
{

```

```

    if(*(int*)cpv_input % 3 == 0)
    {
        *(bool_t*)pv_output = true;
    }
    else
    {
        *(bool_t*)pv_output = false;
    }
}

```

在 `algo_remove_if()` 中 `_mod3` 就是用户定义的一元谓词. 结果如下:

before remove:

vector1: 1 2 3 4 5 6 7 8 9 10

vector2: 1 2 3 4 5 6 7 8 9 10

after remove:

vector1: 1 2 4 5 6 7 8 9 10 10

vector2: 1 2 4 5 7 8 10 8 9 10

2. 作为行为规则

这种应用最典型的的就是 `algo_for_each()`, 它接受一元函数作为用户自定义的行为, 对容器中每一个数据执行用户自定义的行为, 上面例子的 `_print` 就是用户定义的对每一个数据输出的行为. 还有一个算法 `algo_generate()` 它使用用户定义的函数来产生数据.

```
#include <stdio.h>
```

```
#include <cstdlib/cvector.h>
```

```
#include <cstdlib/calgorithm.h>
```

```
static void _print(const void* cpv_input, void* pv_output);
```

```
static void _generate(const void* cpv_input, void* pv_output);
```

```
int main(int argc, char* argv[])
```

```
{
```

```
    vector_t t_v = create_vector(int);
```

```
    vector_init_n(&t_v, 10);
```

```
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
```

```
    printf("\n");
```

```
    algo_generate(vector_begin(&t_v), vector_end(&t_v), _generate);
```

```
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
```

```
    printf("\n");
```

```
    vector_destroy(&t_v);
```

```

    return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

static void _generate(const void* cpv_input, void* pv_output)
{
    *(int*)pv_output = 100;
}

```

结果:

```

0 0 0 0 0 0 0 0 0 0
100 100 100 100 100 100 100 100 100 100

```

第二节 函数的类型

libcstl 函数分为两种, 一元函数和二元函数, 它们的形式不同, 应用在不同的算法中.

1. 一元函数

一元函数具有如下形式

```
void unary_function(const void* cpv_input, void* pv_output);
```

第一参数是输入数据, 第二个参数是输出数据. 前面的章节已经见过了许多一元函数, 它的实现并没有什么特别之处, `pv_output` 代表函数的输出类型的指针, 这个类型可以是 C 的内部类型, 也可以是用户自定义的类型等, 甚至还可以忽略输出类型 (`algo_for_each()` 中使用的一元函数没有输出, 所以忽略了输出类型). 当输出类型是 `bool_t` 类型时, 一元函数就变成了一元谓词, 一元谓词是特殊的一元函数.

2. 二元函数

二元函数具有如下形式

```
void binary_function(
    const void* cpv_first, const void* cpv_second, void* pv_output);
```

前两个参数是输入数据, 最后一个参数是输出数据. 同一元函数的输出类型相同二元函数的输出也会有很多类型. 当输出类型是 `bool_t` 是就是二元谓词.

第三节 预定义的函数

libcstl 提供了很多预定义的函数, 具体分为几大类, 每一类中都针对 C 内部类型分别定义了具体的函数. 要使用函数必须包含头文件<cstl/cfunctional.h>
#include <cstl/cfunctional.h>

1. 算术运算函数

libcstl 提供了一系列的算术运算函数:

fun_plus_xxxx()	加法操作函数.
fun_minus_xxxx()	减法操作函数.
fun_multiplies_xxxx()	乘法操作函数.
fun_divides_xxxx()	除法操作函数.
fun_modulus_xxxx()	求余操作函数.
fun_negate_xxxx()	求反操作函数.

其中的 xxxx 代表 C 的内部类型, 例如针对 int 类型的除法操作函数为 fun_divides_int().

2. 关系运算函数

fun_equal_xxxx()	判断两个数据是否相等.
fun_not_equal_xxxx()	判断两个数据是否不等.
fun_great_xxxx()	判断第一个数据是否大于第二个数据.
fun_great_equal_xxxx()	判断第一个数据是否大于等于第二个数据.
fun_less_xxxx()	判断第一个数据是否小于第二个数据.
fun_less_equal()	判断第一个数据是否小于等于第二个数据.

3. 逻辑运算函数

fun_logical_and_xxxx()	逻辑与操作.
fun_logical_or_xxxx()	逻辑或操作.
fun_logical_not_xxxx()	逻辑非操作.

4. 其他函数

fun_random_number()	产生随机数.
fun_default_unary()	默认的一元函数.
fun_default_binary()	默认的二元函数.

第七章 libcstl 算法

libcstl 提供了大量的算法来满足不同的需求, 算法大致分为, 非质变算法, 质变算法, 排序算法, 算术算法. 这些种类的之间并没有明确的界限, 因为如果非质变算法调用了修改数据的用户定义规则函数, 那么这个算法就变成了质变的了, 所以这只是大概的划分. 要使用算法要包含头文件<cstl/calgorithm.h>
#include <cstl/calgorithm.h>
其中算术算法不在这个头文件中, 要使用算术算法必须包含头文件<cstl/cnumeric.h>
#include <cstl/cnumeric.h>

第一节 算法的分类

这里所说的分类不是上面说的将算法分为质变, 非质变的意思, 这里的分类是指同一个算法在为了满足不同的需求会有多种不同的形式, 这种形式是通过后缀表现出来的.
一个算法通常会有基本形式, 后缀为_if 的形式, 后缀为_copy 的形式. 但并不是所有的函数都是这样的.
后缀为_if 的形式:
后缀为_if 的形式表示这个算法版本需要用户自定义的规则函数. 但并不是说有_if 后缀的就不能调用用户自定义的规则函数了如 algo_for_each(), algo_generate() 等函数没有_if 后缀但是必须调用用户自定义的规则函数.
后缀为_copy 的形式:
后缀为_copy 的形式是将算法的执行结果拷贝到另一个数据区间中. 对于输出的数据区间来说这些算法都是质变算法.

第二节 非质变算法

非质变算法就是算法执行后不会修改数据区间中的数据. 但是这并不是绝对的.

algo_for_each()	对数据区间内的所有数据都执行一次指定的操作.
algo_find()	在指定的数据区间查找指定的数据.
algo_find_if()	使用特定的规则在指定的数据区间查找指定的数据.
algo_adjacent_find()	找到第一组满足相邻两个数据相等的数据.
algo_adjacent_find_if()	找到第一组满足指定条件的相邻的数据.

<code>algo_find_first_of()</code>	找到第二个数据区间中任意数据在第一个区间中出现的位置.
<code>algo_find_first_of_if()</code>	使用指定规则找到第二个区间中任意数据在第一个区间中出现的位置.
<code>algo_count()</code>	统计指定数据在数据区间中的数量.
<code>algo_count_if()</code>	统计满足指定规则的数据在数据区间中的数量.
<code>algo_mismatch()</code>	找到两个区间中数据不相同的第一个位置.
<code>algo_mismatch_if()</code>	找到两个区间中数据不满足规则的第一个位置.
<code>algo_equal()</code>	判断两个区间的数据是否相等.
<code>algo_equal_if()</code>	判断两个区间的数据是否满足特定规则.
<code>algo_search()</code>	查找第一个与第二个区间匹配的位置.
<code>algo_search_if()</code>	查找第一个与第二个区间满足特定规则的位置.
<code>algo_search_n()</code>	查找连续 n 个指定数据.
<code>algo_search_n_if()</code>	查找连续 n 个满足特定规则的数据.
<code>algo_search_end()</code>	查找最后一个与第二个区间匹配的位置.
<code>algo_search_end_if()</code>	查找最后一个与第二个区间满足特定规则的位置.
<code>algo_find_end()</code>	同 <code>algo_search_end()</code>
<code>algo_find_end_if()</code>	同 <code>algo_search_end_if()</code>

`algo_for_each()` 并不完全属于非质变算法, 当用户指定的规则改变了数据, 那么 `algo_for_each()` 就变成了质变算法. `algo_for_each()` 使用的是一元函数, 但是这个函数不要求返回数据所以不能使用参数 `pv_output`.

```
#include <stdio.h>
#include <cstdlib/cvector.h>
#include <cstdlib/calgorithm.h>
```

```
static void _print(const void* cpv_input, void* pv_output);
static void _plus100(const void* cpv_input, void* pv_output);
```

```
int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(int);
    int i = 0;

    vector_init(&t_v);
    for(i = 0; i < 10; ++i)
    {
        vector_push_back(&t_v, i);
    }
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _plus100);
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");

    vector_destroy(&t_v);
}
```

```

        return 0;
    }

    static void _print(const void* cpv_input, void* pv_output)
    {
        printf("%d ", *(int*)cpv_input);
    }

    static void _plus100(const void* cpv_input, void* pv_output)
    {
        *(int*)cpv_input += 100;
    }

```

结果:

```

0 1 2 3 4 5 6 7 8 9
100 101 102 103 104 105 106 107 108 109

```

当 `algo_for_each()` 算法使用 `_print` 时, 用户定义规则内没有修改数据, 所以这时候 `algo_for_each()` 算法是非质变算法. 当 `algo_for_each()` 算法使用 `_plus100` 时, 用户自定义规则内修改了数据的内容, 所以这个时候 `algo_for_each()` 算法就是质变算法.

建议: 在使用 `algo_for_each()` 算法时不要使用修改数据的规则, 如果要修改数据请使用 `algo_generate()` 算法.

除了 `algo_for_each()` 算法外, 非质变算法都是与查找相关的. 但是查找分为两类, 一类似查找单一数据, 另一类时查找一个区间.

1. 查找单一数据

统计指定数据个数: `algo_count()`, `algo_count_if()`

带有 `if` 后缀的版本接收一个一元谓词为特定规则, 但是这个谓词不能修改数据内容. 关联容器提供了获得指定数据的操作, 对于关联容器本身的操作函数更快速.

```

#include <stdio.h>
#include <cstdlib/cvector.h>
#include <cstdlib/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);
static void _is_even(const void* cpv_input, void* pv_output);
static void _mod3(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{

```



```

vector_t t_v = create_vector(int);
int i = 0;

vector_init(&t_v);
for(i = 1; i <= 10; ++i)
{
    vector_push_back(&t_v, i);
}
printf("vector: ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");
printf("number of elements equal to 4:          %u\n",
       algo_count(vector_begin(&t_v), vector_end(&t_v), 4));
printf("number of elements with even value:      %u\n",
       algo_count_if(vector_begin(&t_v), vector_end(&t_v), _is_even));
printf("number of elements divided exactly by 3: %u\n",
       algo_count_if(vector_begin(&t_v), vector_end(&t_v), _mod3));

vector_destroy(&t_v);

return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

static void _is_even(const void* cpv_input, void* pv_output)
{
    if(*(int*)cpv_input % 2 == 0)
    {
        *(bool_t*)pv_output = true;
    }
    else
    {
        *(bool_t*)pv_output = false;
    }
}

static void _mod3(const void* cpv_input, void* pv_output)
{
    if(*(int*)cpv_input % 3 == 0)
    {
        *(bool_t*)pv_output = true;
    }
}

```

```

    else
    {
        *(bool_t*)pv_output = false;
    }
}

```

结果:

```

vector: 1 2 3 4 5 6 7 8 9 10
number of elements equal to 4:          1
number of elements with even value:      5
number of elements divided exactly by 3: 3

```

查找数据: algo_find(), algo_find_if()

带有 if 后缀的版本接收一个一元谓词为特定规则,但是这个谓词不能修改数据内容. 关联容器提供了查找指定数据的操作,对于关联容器本身的操作函数更快速. 如果数据区间是有序的使用 algo_equal_range(), algo_lower_bound(), algo_upper_bound(), algo_binary_search()速度更快.

```

#include <stdio.h>
#include <cstl/clist.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    list_t t_l = create_list(int);
    iterator_t t_pos1, t_pos2;
    int i = 0;

    list_init(&t_l);
    for(i = 1; i <= 10; ++i)
    {
        list_push_back(&t_l, i);
    }
    for(i = 1; i <= 10; ++i)
    {
        list_push_back(&t_l, i);
    }
    printf("list: ");
    algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
    printf("\n");

    t_pos1 = algo_find(list_begin(&t_l), list_end(&t_l), 4);
    if(!iterator_equal(&t_pos1, list_end(&t_l)))
    {
        iterator_next(&t_pos1);
    }
}

```

```

        t_pos2 = algo_find(t_pos1, list_end(&t_l), 4);
    }

    if(!iterator_equal(&t_pos1, list_end(&t_l)) &&
        !iterator_equal(&t_pos2, list_end(&t_l)))
    {
        iterator_prev(&t_pos1);
        iterator_next(&t_pos2);
        algo_for_each(t_pos1, t_pos2, _print);
        printf("\n");
    }

    list_destroy(&t_l);

    return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

```

list: 1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 10
4 5 6 7 8 9 10 1 2 3 4

```

下面是使用 algo_find_if() 的例子:

```

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);
static void _greater3(const void* cpv_input, void* pv_output);
static void _mod3(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(int);
    int i = 0;
    iterator_t t_pos;

    vector_init(&t_v);
    for(i = 1; i <= 10; ++i)
    {
        vector_push_back(&t_v, i);
    }
}

```

```

    }
    printf("vector: ");
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");

    t_pos = algo_find_if(vector_begin(&t_v), vector_end(&t_v), _greater3);
    printf("the %d. element is the first greater than 3.\n",
        iterator_distance(vector_begin(&t_v), t_pos) + 1);
    t_pos = algo_find_if(vector_begin(&t_v), vector_end(&t_v), _mod3);
    printf("the %d. element is the first divisible by 3.\n",
        iterator_distance(vector_begin(&t_v), t_pos) + 1);

    vector_destroy(&t_v);

    return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

static void _greater3(const void* cpv_input, void* pv_output)
{
    if(*(int*)cpv_input > 3)
    {
        *(bool_t*)pv_output = true;
    }
    else
    {
        *(bool_t*)pv_output = false;
    }
}

static void _mod3(const void* cpv_input, void* pv_output)
{
    if(*(int*)cpv_input % 3 == 0)
    {
        *(bool_t*)pv_output = true;
    }
    else
    {
        *(bool_t*)pv_output = false;
    }
}

```

结果:

vector: 1 2 3 4 5 6 7 8 9 10

the 4. element is the first greater then 3.

the 3. element is the first divisible by 3.

找到第一组满足指定条件的相邻的数据: `algo_adjacent_find()`, `algo_adjacent_find_if()`

带有 `if` 后缀的版本接收一个二元谓词为特定规则, 但是这个谓词用来指定前后两个数据的关系但是不能修改数据内容.

```
#include <stdio.h>
```

```
#include <cstdlib/cvector.h>
```

```
#include <cstdlib/calgorithm.h>
```

```
static void _print(const void* cpv_input, void* pv_output);
```

```
static void _doubled(const void* cpv_first, const void* cpv_second, void* pv_output);
```

```
int main(int argc, char* argv[])
```

```
{
```

```
    vector_t t_v = create_vector(int);
```

```
    iterator_t t_pos;
```

```
    vector_init(&t_v);
```

```
    vector_push_back(&t_v, 1);
```

```
    vector_push_back(&t_v, 3);
```

```
    vector_push_back(&t_v, 2);
```

```
    vector_push_back(&t_v, 4);
```

```
    vector_push_back(&t_v, 5);
```

```
    vector_push_back(&t_v, 5);
```

```
    vector_push_back(&t_v, 0);
```

```
    printf("vector: ");
```

```
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
```

```
    printf("\n");
```

```
    t_pos = algo_adjacent_find(vector_begin(&t_v), vector_end(&t_v));
```

```
    if(!iterator_equal(&t_pos, vector_end(&t_v)))
```

```
    {
```

```
        printf("first two elements with equal value have position %d\n",
```

```
            iterator_distance(vector_begin(&t_v), t_pos) + 1);
```

```
    }
```

```
    t_pos = algo_adjacent_find_if(vector_begin(&t_v), vector_end(&t_v), _doubled);
```

```
    if(!iterator_equal(&t_pos, vector_end(&t_v)))
```

```
    {
```

```
        printf("first two elements with second value twice the first have pos. %d\n",
```

```
            iterator_distance(vector_begin(&t_v), t_pos) + 1);
```

```
    }
```

```

    vector_destroy(&t_v);

    return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

static void _doubled(const void* cpv_first, const void* cpv_second, void* pv_output)
{
    if(*(int*)cpv_first * 2 == *(int*)cpv_second)
    {
        *(bool_t*)pv_output = true;
    }
    else
    {
        *(bool_t*)pv_output = false;
    }
}

```

结果:

vector: 1 3 2 4 5 5 0

first two elements with equal value have position 5

first two elements with second value twice the first have pos. 3

使用指定规则找到第二个区间中任意数据在第一个区间中出现的位置:

`algo_find_first_of()`, `algo_find_first_of_if()`

带有 if 后缀的版本接收一个二元谓词为特定规则, 但是这个谓词用来指定前后两个数据的关系但是不能修改数据内容.

```
#include <stdio.h>
```

```
#include <cstl/cvector.h>
```

```
#include <cstl/clist.h>
```

```
#include <cstl/calgorithm.h>
```

```
static void _print(const void* cpv_input, void* pv_output);
```

```
int main(int argc, char* argv[])
```

```
{
```

```
    vector_t t_v = create_vector(int);
```

```
    list_t t_l = create_list(int);
```

```
    int i = 0;
```

```
    iterator_t t_pos;
```

```
    vector_init(&t_v);
```

```

list_init(&t_l);
for(i = 0; i < 10; ++i)
{
    vector_push_back(&t_v, i);
}
for(i = 3; i < 6; ++i)
{
    list_push_back(&t_l, i);
}
printf("vector: ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");
printf("search: ");
algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
printf("\n");

t_pos = algo_find_first_of(
    vector_begin(&t_v), vector_end(&t_v), list_begin(&t_l), list_end(&t_l));
printf("first element of search in vector is element %d\n",
    iterator_distance(vector_begin(&t_v), t_pos) + 1);

algo_reverse(vector_begin(&t_v), vector_end(&t_v));
printf("vector: ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");
printf("search: ");
algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
printf("\n");
t_pos = algo_find_first_of(
    vector_begin(&t_v), vector_end(&t_v), list_begin(&t_l), list_end(&t_l));
printf("first element of search in vector is element %d\n",
    iterator_distance(vector_begin(&t_v), t_pos) + 1);

vector_destroy(&t_v);
list_destroy(&t_l);

return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

```
vector: 0 1 2 3 4 5 6 7 8 9
search: 3 4 5
first element of search in vector is element 4
vector: 9 8 7 6 5 4 3 2 1 0
search: 3 4 5
first element of search in vector is element 5
```

2. 数据区间匹配

比较数据区间: `algo_equal()`, `algo_equal_if()`

带有 `if` 后缀的版本接收一个二元谓词为特定规则, 但是这个谓词用来指定前后两个数据的关系但是不能修改数据内容.

```
#include <stdio.h>
#include <cstdlib/cvector.h>
#include <cstdlib/clist.h>
#include <cstdlib/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);
static void _both_even_or_odd(
    const void* cpv_first, const void* cpv_second, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(int);
    list_t t_l = create_list(int);
    int i = 0;

    vector_init(&t_v);
    list_init(&t_l);
    for(i = 1; i <= 7; ++i)
    {
        vector_push_back(&t_v, i);
    }
    for(i = 3; i <= 9; ++i)
    {
        list_push_back(&t_l, i);
    }
    printf("vector: ");
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");
    printf("list: ");
    algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
    printf("\n");
```



```

if(algo_equal(vector_begin(&t_v), vector_end(&t_v), list_begin(&t_l)))
{
    printf("vector == list\n");
}
else
{
    printf("vector != list\n");
}
if(algo_equal_if(
    vector_begin(&t_v), vector_end(&t_v), list_begin(&t_l), _both_even_or_odd))
{
    printf("even and odd elements correspond.\n");
}
else
{
    printf("even and odd elements is not correspond.\n");
}

vector_destroy(&t_v);
list_destroy(&t_l);

return 0;
}

```

```

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

static void _both_even_or_odd(
    const void* cpv_first, const void* cpv_second, void* pv_output)
{
    if(*(int*)cpv_first % 2 == *(int*)cpv_second % 2)
    {
        *(bool_t*)pv_output = true;
    }
    else
    {
        *(bool_t*)pv_output = false;
    }
}

```

结果:

```

vector: 1 2 3 4 5 6 7
list:   3 4 5 6 7 8 9

```

```
vector != list
even and odd elements correspond.
```

查找数据区间的不同: `algo_mismatch()`, `algo_mismatch_if()`

带有 if 后缀的版本接收一个二元谓词为特定规则, 但是这个谓词用来指定前后两个数据的关系但是不能修改数据内容. 这两个算法函数返回 `pair_t` 类型, 使用之后要销毁.

```
#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/clist.h>
#include <cstl/cutility.h>
#include <cstl/calgorithm.h>
```

```
static void _print(const void* cpv_input, void* pv_output);
```

```
int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(int);
    list_t t_l = create_list(int);
    pair_t t_p;
    int i = 0;

    vector_init(&t_v);
    list_init(&t_l);
    for(i = 1; i <= 6; ++i)
    {
        vector_push_back(&t_v, i);
    }
    for(i = 1; i <= 16; i *= 2)
    {
        list_push_back(&t_l, i);
    }
    list_push_back(&t_l, 3);
    printf("vector: ");
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");
    printf("list:   ");
    algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
    printf("\n");

    t_p = algo_mismatch(vector_begin(&t_v), vector_end(&t_v), list_begin(&t_l));
    if(iterator_equal(t_p.first, vector_end(&t_v)))
    {
        printf("no mismatch!\n");
    }
}
```

```

else
{
    printf("first mismatch: %d and %d\n",
        *(int*)iterator_get_pointer(t_p.first),
        *(int*)iterator_get_pointer(t_p.second));
}
pair_destroy(&t_p);

t_p = algo_mismatch_if(
    vector_begin(&t_v), vector_end(&t_v),
    list_begin(&t_l), fun_less_equal_int);
if(iterator_equal(t_p.first, vector_end(&t_v)))
{
    printf("always less-or-equal\n");
}
else
{
    printf("not less-or-equal: %d and %d\n",
        *(int*)iterator_get_pointer(t_p.first),
        *(int*)iterator_get_pointer(t_p.second));
}
pair_destroy(&t_p);

vector_destroy(&t_v);
list_destroy(&t_l);

return 0;
}

```

```

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

```

vector: 1 2 3 4 5 6
list:   1 2 4 8 16 3
first mismatch: 3 and 4
not less-or-equal: 6 and 3

```

查找子数据区间: algo_search(), algo_search_if()

带有 if 后缀的版本接收一个二元谓词为特定规则, 但是这个谓词用来指定前后两个数据的关系但是不能修改数据内容.

```

#include <stdio.h>
#include <cstl/cdeque.h>
#include <cstl/clist.h>

```

```

#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    deque_t t_dq = create_deque(int);
    list_t t_l = create_list(int);
    int i = 0;
    iterator_t t_pos;

    deque_init(&t_dq);
    list_init(&t_l);
    for(i = 1; i <= 7; ++i)
    {
        deque_push_back(&t_dq, i);
    }
    for(i = 1; i <= 7; ++i)
    {
        deque_push_back(&t_dq, i);
    }
    for(i = 3; i <= 6; ++i)
    {
        list_push_back(&t_l, i);
    }
    printf("deque: ");
    algo_for_each(deque_begin(&t_dq), deque_end(&t_dq), _print);
    printf("\n");
    printf("sub:   ");
    algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
    printf("\n");

    t_pos = algo_search(
        deque_begin(&t_dq), deque_end(&t_dq), list_begin(&t_l), list_end(&t_l));
    while(!iterator_equal(&t_pos, deque_end(&t_dq)))
    {
        printf("sub find with element %d\n",
            iterator_distance(deque_begin(&t_dq), t_pos) + 1);
        iterator_next(&t_pos);
        t_pos = algo_search(
            t_pos, deque_end(&t_dq), list_begin(&t_l), list_end(&t_l));
    }

    deque_destroy(&t_dq);

```

```

        list_destroy(&t_l);

    return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

```

deque: 1 2 3 4 5 6 7 1 2 3 4 5 6 7
sub:   3 4 5 6
sub find with element 3
sub find with element 10

```

下面是一个使用 algo_search_if() 的例子:

```

#include <stdio.h>
#include <cstdlib/cvector.h>
#include <cstdlib/cdeque.h>
#include <cstdlib/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);
static void _check_even(
    const void* cpv_first, const void* cpv_second, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(int);
    deque_t t_dq = create_deque(bool_t);
    int i = 0;
    iterator_t t_pos;

    vector_init(&t_v);
    deque_init(&t_dq);
    for(i = 1; i <= 9; ++i)
    {
        vector_push_back(&t_v, i);
    }
    deque_push_back(&t_dq, true);
    deque_push_back(&t_dq, false);
    deque_push_back(&t_dq, true);
    printf("vector: ");
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");
}

```

```

t_pos = algo_search_if(
    vector_begin(&t_v), vector_end(&t_v),
    deque_begin(&t_dq), deque_end(&t_dq),
    _check_even);
while(!iterator_equal(&t_pos, vector_end(&t_v)))
{
    printf("subrange found starting with element %d\n",
        iterator_distance(vector_begin(&t_v), t_pos) + 1);

    iterator_next(&t_pos);
    t_pos = algo_search_if(
        t_pos, vector_end(&t_v),
        deque_begin(&t_dq), deque_end(&t_dq),
        _check_even);
}

vector_destroy(&t_v);
deque_destroy(&t_dq);

return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

static void _check_even(
    const void* cpv_first, const void* cpv_second, void* pv_output)
{
    if(*(bool_t*)cpv_second)
    {
        if(*(int*)cpv_first %2 == 0)
        {
            *(bool_t*)pv_output = true;
        }
        else
        {
            *(bool_t*)pv_output = false;
        }
    }
    else
    {
        if(*(int*)cpv_first % 2 == 1)

```

```

        {
            *(bool_t*)pv_output = true;
        }
        else
        {
            *(bool_t*)pv_output = false;
        }
    }
}

```

结果:

```

vector: 1 2 3 4 5 6 7 8 9
subrange found starting with element 2
subrange found starting with element 4
subrange found starting with element 6

```

查找连续 n 个满足特定规则的数据: `algo_search_n()`, `algo_search_n_if()`

带有 if 后缀的版本接收一个二元谓词为特定规则,但是这个谓词用来指定前后两个数据的关系但是不能修改数据内容.

```

#include <stdio.h>
#include <cstl/cdeque.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    deque_t t_dq = create_deque(int);
    int i = 0;
    iterator_t t_pos;

    deque_init(&t_dq);
    for(i = 1; i <= 9; ++i)
    {
        deque_push_back(&t_dq, i);
    }
    printf("deque: ");
    algo_for_each(deque_begin(&t_dq), deque_end(&t_dq), _print);
    printf("\n");

    t_pos = algo_search_n(deque_begin(&t_dq), deque_end(&t_dq), 4, 3);
    if(iterator_equal(&t_pos, deque_end(&t_dq)))
    {
        printf("no four consecutive elements with value 3 found\n");
    }
    else
    {

```

```

        printf("four consecutive elements with value 3 start with %d. element\n",
            iterator_distance(deque_begin(&t_dq), t_pos) + 1);
    }
    t_pos = algo_search_n_if(
        deque_begin(&t_dq), deque_end(&t_dq), 4, 3, fun_great_int);
    if(iterator_equal(&t_pos, deque_end(&t_dq)))
    {
        printf("no four consecutive elements with value > 3 found\n");
    }
    else
    {
        printf("four consecutive elements with value > 3 start with %d. element\n",
            iterator_distance(deque_begin(&t_dq), t_pos) + 1);
    }
    deque_destroy(&t_dq);

    return 0;
}

```

```

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

deque: 1 2 3 4 5 6 7 8 9

no four consecutive elements with value 3 found

four consecutive elements with value > 3 start with 4. element

查找最后一个与第二个区间满足特定规则的位置:

`algo_search_end()`, `algo_search_end_if()`, `algo_find_end()`, `algo_find_end_if()`

带有 if 后缀的版本接收一个二元谓词为特定规则, 但是这个谓词用来指定前后两个数据的关系但是不能修改数据内容.

这个算法与 `algo_search()` 功能相似, 只是它查找的是最后一个子区间. `algo_find_end()` 和

`algo_find_end_if()` 分别与 `algo_search_end()` 和 `algo_search_end_if()` 功能相同, 只不过是模仿 SGI STL 相应的算法名称, 其实更确切的命名应该是 `algo_search_end()` 和 `algo_search_end_if()` 这两个名字.

```
#include <stdio.h>
```

```
#include <cstl/cdeque.h>
```

```
#include <cstl/clist.h>
```

```
#include <cstl/calgorithm.h>
```

```
static void _print(const void* cpv_input, void* pv_output);
```

```
int main(int argc, char* argv[])
```

```
{
```

```
    deque_t t_dq = create_deque(int);
```



```

list_t t_l = create_list(int);
int i = 0;
iterator_t t_pos, t_end;

deque_init(&t_dq);
list_init(&t_l);
for(i = 1; i <= 7; ++i)
{
    deque_push_back(&t_dq, i);
}
for(i = 1; i <= 7; ++i)
{
    deque_push_back(&t_dq, i);
}
for(i = 3; i <= 6; ++i)
{
    list_push_back(&t_l, i);
}
printf("deque: ");
algo_for_each(deque_begin(&t_dq), deque_end(&t_dq), _print);
printf("\n");
printf("sub: ");
algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
printf("\n");
t_pos = algo_search_end(
    deque_begin(&t_dq), deque_end(&t_dq), list_begin(&t_l), list_end(&t_l));
t_end = deque_end(&t_dq);
while(!iterator_equal(&t_pos, t_end))
{
    printf("sub found starting with element %d\n",
        iterator_distance(deque_begin(&t_dq), t_pos) + 1);
    t_end = t_pos;
    t_pos = algo_find_end(
        deque_begin(&t_dq), t_end, list_begin(&t_l), list_end(&t_l));
}
deque_destroy(&t_dq);
list_destroy(&t_l);

return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);

```

```
}
```

结果:

```
deque: 1 2 3 4 5 6 7 1 2 3 4 5 6 7
```

```
sub:    3 4 5 6
```

```
sub found starting with element 10
```

```
sub found starting with element 3
```

第三节 质变算法

质变算法就是算法执行后修改数据区间中的数据. 修改数据的方式有两种:

当迭代器通过数据时修改.

将算法执行的结果拷贝到其他数据区间中.

带有_copy 后缀的算法函数都是属于后者, 有些算法两个版本都提供.

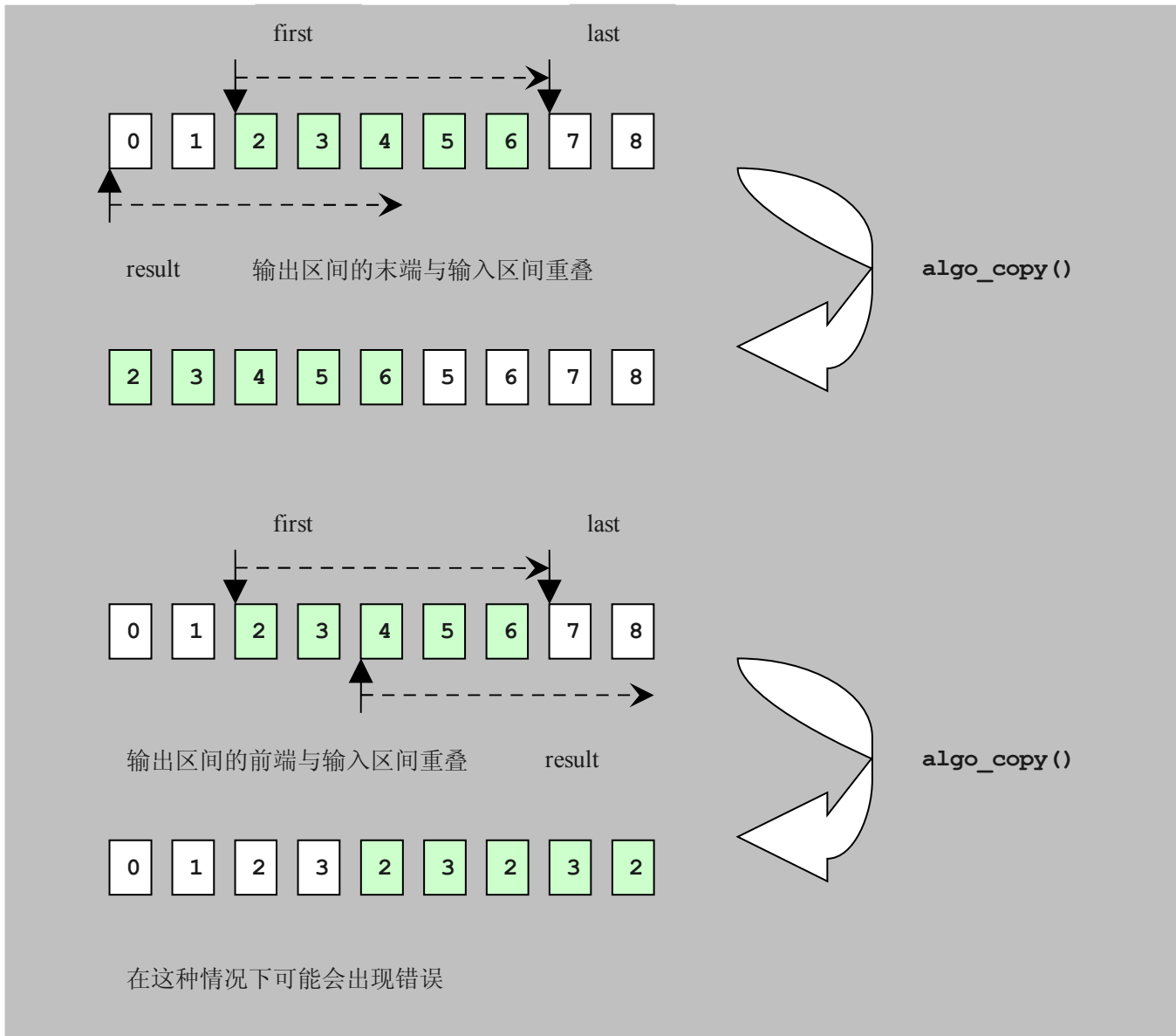
在使用质变算法时, 一定不要使用管理容器作为目的数据区间, 如果使用就会修改容器中的数据, 破坏了关联容器的自动排序的性质, 这样有关该管理容器的后续操作都会出现错误或为定义的结果.

algo_copy()	拷贝数据.
algo_copy_n()	拷贝 n 个数据.
algo_copy_backward()	从后往前拷贝数据.
algo_swap()	交换两个迭代器所指的数据.
algo_iter_swap()	交换两个迭代器所指的数据.
algo_swap_ranges()	交换两个数据区间中的数据.
algo_transform()	将数据从第一个数据区间通过特定的规则转换到第二个区间.
algo_transform_binary()	将数据从第一个数据区间通过特定的二元规则转换到第二个区间.
algo_replace()	使用新数据替换数据区间内所有指定的旧数据.
algo_replace_if()	使用新数据替换数据区间内所有符合条件的旧数据.
algo_replace_copy()	将数据区间中的数据拷贝到新的数据区间, 拷贝过程中使用新数据替换指定的旧数据.
algo_replace_copy_if()	将数据区间中的数据拷贝到新的数据区间, 拷贝过程中使用新数据替换所有符合条件的旧数据.
algo_fill()	向数据区间中填充新数据.
algo_fill_n()	向数据区间中填充 n 个新数据.
algo_generate()	使用一元函数产生的结果填充数据区间.
algo_generate_n()	使用一元函数产生的结果向数据区间中填充 n 个值.
algo_remove()	移除指定数据.
algo_remove_if()	移除符合条件的数据.
algo_remove_copy()	移除指定数据并将结果拷贝到另一区间中.
algo_remove_copy_if()	移除符合条件的数据并将结果拷贝到另一个区间中.
algo_unique()	移除相邻的重复数据.
algo_unique_if()	移除相邻的符合条件的数据.
algo_unique_copy()	移除相邻的重复数据并将结果拷贝到另一个区间中.
algo_unique_copy_if()	移除相邻的符合条件的数据并将结果拷贝到另一个区间中.
algo_reverse()	将数据区间中的数据逆序.
algo_reverse_copy()	将数据区间中的数据逆序并拷贝到另一区间中.

<code>algo_rotate()</code>	将数据区间中的两部分数据互换.
<code>algo_rotate_copy()</code>	将数据区间中的两部分数据互换并将结果拷贝到另一个区间中
<code>algo_random_shuffle()</code>	将数据区间中的数据随机重排.
<code>algo_random_shuffle_if()</code>	使用特殊的随机函数将数据区间中的数据随机重排.
<code>algo_random_sample()</code>	随机的将第一个区间中的数据拷贝到第二个区间.
<code>algo_random_sample_if()</code>	使用特殊的随机函数将第一个区间中的数据拷贝到第二个区间中.
<code>algo_random_sample_n()</code>	随机的将第一个区间中的 n 个数据拷贝到第二个区间.
<code>algo_random_sample_n_if()</code>	使用特殊的随机函数将第一个区间中的 n 个数据拷贝到第二个区间.
<code>algo_partition()</code>	将符合条件的数据放到数据区间的前部, 不符合条件的放到后部.
<code>algo_stable_partition()</code>	顺序稳定的 <code>algo_partition()</code> .

1. 拷贝数据

拷贝数据有三个算法 `algo_copy()`, `algo_copy_n()`, `algo_copy_backward()` 前两个算法的不同点在于第一个算法采用区间来规定拷贝数据的个数, 第二个直接给出了拷贝数据的个数, 它们两个都是从前往后拷贝数据. 第三个算法是从后往前拷贝数据. 如果两个区间没有重叠那么这两种拷贝方式都是没有任何问题的, 但是如果区间有重叠那么不同的方式在重叠不一样的时候结果会出现不同.



上面的图显示了当区间重叠时使用 `std::algo_copy()` 可能会出现的结果, 第一种情况是不会出现任何问题的, 第二种情况就会出现拷贝错误. 为什么说可能呢, 因为如果数据区间是连续的内存(如 `vector_t`, `string_t`)就不会出现这样的情况, 算法内部对连续的内存拷贝使用了 `memmove()` 函数而不是逐一拷贝所以不会出现这种情况, 但是其他容器就会出现这样的错误了.

```
#include <stdio.h>
#include <cstdlib/cvector.h>
#include <cstdlib/clist.h>
#include <cstdlib/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t t_v1 = create_vector(int);
    vector_t t_v2 = create_vector(int);
```

```

list_t t_l1 = create_list(int);
list_t t_l2 = create_list(int);
int i = 0;
iterator_t t_first, t_last, t_result;

vector_init(&t_v1);
vector_init(&t_v2);
list_init(&t_l1);
list_init(&t_l2);
for(i = 0; i < 9; ++i)
{
    vector_push_back(&t_v1, i);
    vector_push_back(&t_v2, i);
    list_push_back(&t_l1, i);
    list_push_back(&t_l2, i);
}
printf("before copy:\n");
printf("vector1: ");
algo_for_each(vector_begin(&t_v1), vector_end(&t_v1), _print);
printf("\n");
printf("list1:   ");
algo_for_each(list_begin(&t_l1), list_end(&t_l1), _print);
printf("\n");
printf("vector2: ");
algo_for_each(vector_begin(&t_v2), vector_end(&t_v2), _print);
printf("\n");
printf("list2:   ");
algo_for_each(list_begin(&t_l2), list_end(&t_l2), _print);
printf("\n");

/* case 1 */
t_first = vector_begin(&t_v1);
iterator_advance(&t_first, 2);
t_last = t_first;
iterator_advance(&t_last, 5);
t_result = vector_begin(&t_v1);
algo_copy(t_first, t_last, t_result);

t_first = list_begin(&t_l1);
iterator_advance(&t_first, 2);
t_last = t_first;
iterator_advance(&t_last, 5);
t_result = list_begin(&t_l1);
algo_copy(t_first, t_last, t_result);

```

```

/* case 2 */
t_first = vector_begin(&t_v2);
iterator_advance(&t_first, 2);
t_last = t_first;
iterator_advance(&t_last, 5);
t_result = vector_begin(&t_v2);
iterator_advance(&t_result, 4);
algo_copy(t_first, t_last, t_result);

t_first = list_begin(&t_l2);
iterator_advance(&t_first, 2);
t_last = t_first;
iterator_advance(&t_last, 5);
t_result = list_begin(&t_l2);
iterator_advance(&t_result, 4);
algo_copy(t_first, t_last, t_result);

printf("after copy:\n");
printf("vector1: ");
algo_for_each(vector_begin(&t_v1), vector_end(&t_v1), _print);
printf("\n");
printf("list1:   ");
algo_for_each(list_begin(&t_l1), list_end(&t_l1), _print);
printf("\n");
printf("vector2: ");
algo_for_each(vector_begin(&t_v2), vector_end(&t_v2), _print);
printf("\n");
printf("list2:   ");
algo_for_each(list_begin(&t_l2), list_end(&t_l2), _print);
printf("\n");

vector_destroy(&t_v1);
list_destroy(&t_l1);
vector_destroy(&t_v2);
list_destroy(&t_l2);

return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

before copy:

vector1: 0 1 2 3 4 5 6 7 8

list1: 0 1 2 3 4 5 6 7 8

vector2: 0 1 2 3 4 5 6 7 8

list2: 0 1 2 3 4 5 6 7 8

after copy:

vector1: 2 3 4 5 6 5 6 7 8

list1: 2 3 4 5 6 5 6 7 8

vector2: 0 1 2 3 2 3 4 5 6

list2: 0 1 2 3 2 3 2 3 2

从结果种可以看出, vector1 和 list1 它们的输出区间的末端与输入区间重叠(图中的第一种情况), 拷贝后结果正确. vector2 和 list2 它们的输出区间的前端与输入区间重叠(图中的第二种情况), 拷贝后 vector2 的结果正确, list2 的结果不正确, 这是因为 vector2 是连续的内存结果拷贝使用了 memmove() 函数而 list2 不是连续的内存结构是按照逐个拷贝的结果. 如果使用 algo_copy_backward() 就不同了.

```
#include <stdio.h>
```

```
#include <cstl/cvector.h>
```

```
#include <cstl/clist.h>
```

```
#include <cstl/calgorithm.h>
```

```
static void _print(const void* cpv_input, void* pv_output);
```

```
int main(int argc, char* argv[])
```

```
{
```

```
    vector_t t_v1 = create_vector(int);
```

```
    vector_t t_v2 = create_vector(int);
```

```
    list_t t_l1 = create_list(int);
```

```
    list_t t_l2 = create_list(int);
```

```
    int i = 0;
```

```
    iterator_t t_first, t_last, t_result;
```

```
    vector_init(&t_v1);
```

```
    vector_init(&t_v2);
```

```
    list_init(&t_l1);
```

```
    list_init(&t_l2);
```

```
    for(i = 0; i < 9; ++i)
```

```
    {
```

```
        vector_push_back(&t_v1, i);
```

```
        vector_push_back(&t_v2, i);
```

```
        list_push_back(&t_l1, i);
```

```
        list_push_back(&t_l2, i);
```

```
    }
```

```
    printf("before copy:\n");
```

```
    printf("vector1: ");
```

```

algo_for_each(vector_begin(&t_v1), vector_end(&t_v1), _print);
printf("\n");
printf("list1:  ");
algo_for_each(list_begin(&t_l1), list_end(&t_l1), _print);
printf("\n");
printf("vector2: ");
algo_for_each(vector_begin(&t_v2), vector_end(&t_v2), _print);
printf("\n");
printf("list2:  ");
algo_for_each(list_begin(&t_l2), list_end(&t_l2), _print);
printf("\n");

```

```

/* case 1 */
t_first = vector_begin(&t_v1);
iterator_advance(&t_first, 2);
t_last = t_first;
iterator_advance(&t_last, 5);
t_result = vector_begin(&t_v1);
iterator_advance(&t_result, 5);
algo_copy_backward(t_first, t_last, t_result);

```

```

t_first = list_begin(&t_l1);
iterator_advance(&t_first, 2);
t_last = t_first;
iterator_advance(&t_last, 5);
t_result = list_begin(&t_l1);
iterator_advance(&t_result, 5);
algo_copy_backward(t_first, t_last, t_result);

```

```

/* case 2 */
t_first = vector_begin(&t_v2);
iterator_advance(&t_first, 2);
t_last = t_first;
iterator_advance(&t_last, 5);
t_result = vector_end(&t_v2);
algo_copy_backward(t_first, t_last, t_result);

```

```

t_first = list_begin(&t_l2);
iterator_advance(&t_first, 2);
t_last = t_first;
iterator_advance(&t_last, 5);
t_result = list_end(&t_l2);
algo_copy_backward(t_first, t_last, t_result);

```



```

    printf("after copy:\n");
    printf("vector1: ");
    algo_for_each(vector_begin(&t_v1), vector_end(&t_v1), _print);
    printf("\n");
    printf("list1:  ");
    algo_for_each(list_begin(&t_l1), list_end(&t_l1), _print);
    printf("\n");
    printf("vector2: ");
    algo_for_each(vector_begin(&t_v2), vector_end(&t_v2), _print);
    printf("\n");
    printf("list2:  ");
    algo_for_each(list_begin(&t_l2), list_end(&t_l2), _print);
    printf("\n");

    vector_destroy(&t_v1);
    list_destroy(&t_l1);
    vector_destroy(&t_v2);
    list_destroy(&t_l2);

    return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

before copy:

vector1: 0 1 2 3 4 5 6 7 8

list1: 0 1 2 3 4 5 6 7 8

vector2: 0 1 2 3 4 5 6 7 8

list2: 0 1 2 3 4 5 6 7 8

after copy:

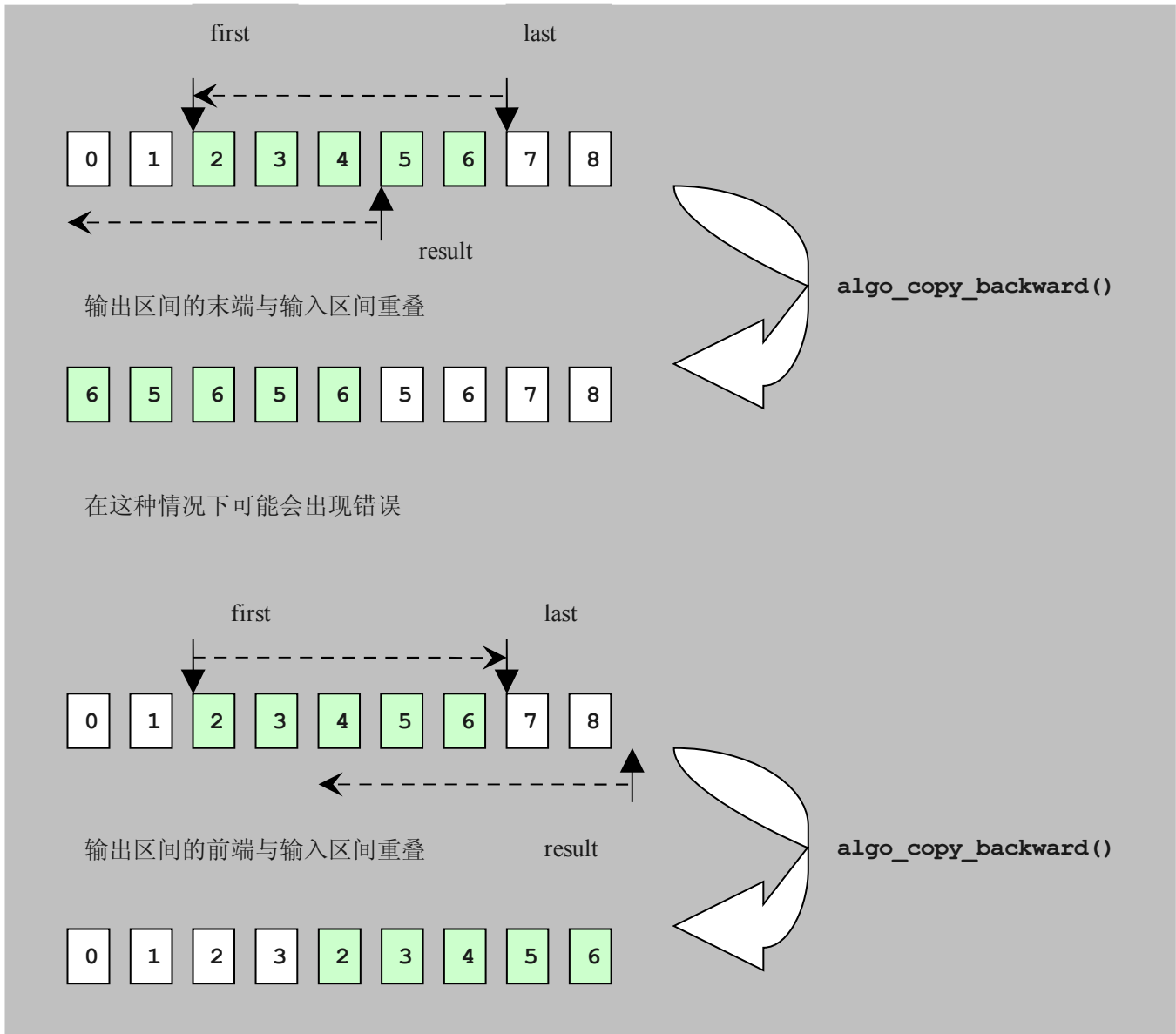
vector1: 2 3 4 5 6 5 6 7 8

list1: 6 5 6 5 6 5 6 7 8

vector2: 0 1 2 3 2 3 4 5 6

list2: 0 1 2 3 2 3 4 5 6

使用 `algo_copy_backward()` 的具体过程:



`algo_copy_backward()` 最需要注意的是最后一个参数是输出区间的末端, 而不是前端.

2. 交换数据

数据交换有 3 个算法: `algo_swap()`, `algo_iter_swap()`, `algo_swap_ranges()` 其中前两个交换单独的数据, 最后一个交换两个数据区间的数据.

```
#include <stdio.h>
#include <cstl/cdeque.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>
```

```
static void _print(const void* cpv_input, void* pv_output);
```

```

int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(int);
    deque_t t_dq = create_deque(int);
    int i = 0;

    vector_init(&t_v);
    deque_init(&t_dq);
    for(i = 1; i <= 9; ++i)
    {
        vector_push_back(&t_v, i);
    }
    for(i = 11; i <= 23; ++i)
    {
        deque_push_back(&t_dq, i);
    }
    printf("vector: ");
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");
    printf("deque:  ");
    algo_for_each(deque_begin(&t_dq), deque_end(&t_dq), _print);
    printf("\n\n");

    algo_swap(vector_begin(&t_v), deque_begin(&t_dq));
    printf("vector: ");
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");
    printf("deque:  ");
    algo_for_each(deque_begin(&t_dq), deque_end(&t_dq), _print);
    printf("\n\n");

    algo_swap_ranges(vector_begin(&t_v), vector_end(&t_v), deque_begin(&t_dq));
    printf("vector: ");
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");
    printf("deque:  ");
    algo_for_each(deque_begin(&t_dq), deque_end(&t_dq), _print);
    printf("\n\n");

    algo_iter_swap(vector_begin(&t_v), deque_begin(&t_dq));
    printf("vector: ");
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");
    printf("deque:  ");

```

```

    algo_for_each(deque_begin(&t_dq), deque_end(&t_dq), _print);
    printf("\n\n");

    vector_destroy(&t_v);
    deque_destroy(&t_dq);

    return 0;
}

```

```

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

```

vector: 1 2 3 4 5 6 7 8 9
deque:  11 12 13 14 15 16 17 18 19 20 21 22 23

```

```

vector: 11 2 3 4 5 6 7 8 9
deque:  1 12 13 14 15 16 17 18 19 20 21 22 23

```

```

vector: 1 12 13 14 15 16 17 18 19
deque:  11 2 3 4 5 6 7 8 9 20 21 22 23

```

```

vector: 11 12 13 14 15 16 17 18 19
deque:  1 2 3 4 5 6 7 8 9 20 21 22 23

```

3. 数据变换

数据变换有 2 个算法: `algo_transform()`, `algo_transform_binary()` 这两个算法的区别在于变换规则不同, 第一个算法是从一个区间到另一个区间的变换, 使用一元函数作为变换规则. 第二个算法从两个区间变换到第三个区间, 使用二元函数作为变换规则.

下面是一个如何使用 `algo_transform()` 的例子:

```

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/clist.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);
static void _multipiles10(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{

```

```

vector_t t_v = create_vector(int);
list_t t_l = create_list(int);
int i = 0;

vector_init(&t_v);
list_init_n(&t_l, 10);
for(i = 1; i <= 10; ++i)
{
    vector_push_back(&t_v, i);
}
printf("vector:  ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");
algo_transform(
    vector_begin(&t_v), vector_end(&t_v), vector_begin(&t_v), fun_negate_int);
printf("negated: ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

algo_transform(
    vector_begin(&t_v), vector_end(&t_v), list_begin(&t_l), _multiples10);
printf("list:  ");
algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
printf("\n");

vector_destroy(&t_v);
list_destroy(&t_l);

return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

static void _multiples10(const void* cpv_input, void* pv_output)
{
    *(int*)pv_output = *(int*)cpv_input * 10;
}

```

执行结果:

```

vector:  1 2 3 4 5 6 7 8 9 10
negated: -1 -2 -3 -4 -5 -6 -7 -8 -9 -10
list:    -10 -20 -30 -40 -50 -60 -70 -80 -90 -100

```

下面是关于 `algo_transform_binary()` 的例子:

```
#include <stdio.h>
#include <cstdlib/cvector.h>
#include <cstdlib/clist.h>
#include <cstdlib/cdeque.h>
#include <cstdlib/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(int);
    list_t t_l = create_list(int);
    deque_t t_dq = create_deque(int);
    int i = 0;

    vector_init(&t_v);
    list_init_n(&t_l, 10);
    deque_init_n(&t_dq, 10);
    for(i = 1; i <= 10; ++i)
    {
        vector_push_back(&t_v, i);
    }
    printf("vector:  ");
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");

    algo_transform_binary(
        vector_begin(&t_v), vector_end(&t_v),
        vector_begin(&t_v), vector_begin(&t_v),
        fun_multiplies_int);
    printf("squared: ");
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");

    algo_transform_binary(
        vector_begin(&t_v), vector_end(&t_v),
        vector_begin(&t_v), list_begin(&t_l),
        fun_plus_int);
    printf("list:  ");
    algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
    printf("\n");

    algo_transform_binary(
```

```

        vector_begin(&t_v), vector_end(&t_v),
        list_begin(&t_l), deque_begin(&t_dq),
        fun_minus_int);
printf("deque:   ");
algo_for_each(deque_begin(&t_dq), deque_end(&t_dq), _print);
printf("\n");

vector_destroy(&t_v);
list_destroy(&t_l);

return 0;
}

```

```

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

```

vector:  1 2 3 4 5 6 7 8 9 10
squared: 1 4 9 16 25 36 49 64 81 100
list:    2 8 18 32 50 72 98 128 162 200
deque:   -1 -4 -9 -16 -25 -36 -49 -64 -81 -100

```

4. 数据替换

数据替换有 4 个算法函数:

`algo_replace()`, `algo_replace_if()`, `algo_replace_copy()`, `algo_replace_copy_if()`

从算法的后缀就可以看出算法之间的区别.

下面是数据替换的例子:

```

#include <stdio.h>
#include <cstdlib/cvector.h>
#include <cstdlib/clist.h>
#include <cstdlib/calgorithm.h>

```

```

static void _print(const void* cpv_input, void* pv_output);
static void _mod3(const void* cpv_input, void* pv_output);
static void _greater3(const void* cpv_input, void* pv_output);

```

```

int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(int);
    list_t t_l = create_list(int);
}

```

```

int i = 0;

vector_init(&t_v);
list_init_n(&t_l, 20);
for(i = 1; i <= 10; ++i)
{
    vector_push_back(&t_v, i);
}
for(i = 1; i <= 10; ++i)
{
    vector_push_back(&t_v, i);
}
printf("vector: ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

printf("      ");
algo_replace(vector_begin(&t_v), vector_end(&t_v), 6, 42);
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

printf("      ");
algo_replace_if(vector_begin(&t_v), vector_end(&t_v), _mod3, -3);
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

printf("list:  ");
algo_replace_copy_if(
    vector_begin(&t_v), vector_end(&t_v), list_begin(&t_l), _greater3, 0);
algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
printf("\n");

vector_destroy(&t_v);

return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

static void _mod3(const void* cpv_input, void* pv_output)
{
    if(*(int*)cpv_input % 3 == 0)

```



```

    {
        *(bool_t*)pv_output = true;
    }
    else
    {
        *(bool_t*)pv_output = false;
    }
}
static void _greater3(const void* cpv_input, void* pv_output)
{
    if(*(int*)cpv_input > 3)
    {
        *(bool_t*)pv_output = true;
    }
    else
    {
        *(bool_t*)pv_output = false;
    }
}

```

结果:

```

vector: 1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 10
        1 2 3 4 5 42 7 8 9 10 1 2 3 4 5 42 7 8 9 10
        1 2 -3 4 5 -3 7 8 -3 10 1 2 -3 4 5 -3 7 8 -3 10
list:   1 2 -3 0 0 -3 0 0 -3 0 1 2 -3 0 0 -3 0 0 -3 0

```

5. 数据填充

数据填充有 4 个算法: `algo_fill()`, `algo_fill_n()`, `algo_generate()`, `algo_generate_n()` 前两个算法使用固定的数据填充数据区间, 两个使用数据产生函数来产生数据填充数据区间.

```

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);
static void _fibonacci(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(int);
    iterator_t t_first, t_last;

    vector_init_n(&t_v, 20);

```

```

algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

t_first = t_last = vector_begin(&t_v);
iterator_advance(&t_last, 5);
algo_fill(t_first, t_last, 2);
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");
algo_fill_n(t_last, 3, 5);
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

iterator_advance(&t_first, 10);
iterator_advance(&t_last, 10);
algo_generate(t_first, t_last, _fibonacci);
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");
algo_generate_n(t_last, 2, _fibonacci);
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

vector_destroy(&t_v);

return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

static void _fibonacci(const void* cpv_input, void* pv_output)
{
    static int i = 1;
    static int j = 0;

    *(int*)pv_output = j + i;
    i = j;
    j = *(int*)pv_output;
}

```

结果:

```

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 5 5 5 0 0 0 0 0 0 0 0 0 0
2 2 2 2 2 5 5 5 0 0 1 1 2 3 5 0 0 0 0

```

2 2 2 2 2 5 5 5 0 0 1 1 2 3 5 8 13 0 0 0

6. 移除数据

移除数据算法只是将符合条件的数据使用后面的数据覆盖掉,但是并不会真正的删除数据所占的内存,操作后在数据区间的后部留下了垃圾数据.移除数据算法有两种:第一种是移除符合条件的数据,第二种是移除相邻的符合条件的数据.

移除符合条件的数据:

`algo_remove()`, `algo_remove_if()`, `algo_remove_copy()`, `algo_remove_copy_if()`

这4个函数是移除符合条件的数据算法的不同版本,从它们的后缀就可以看出它们的区别.

```
#include <stdio.h>
```

```
#include <cstl/cvector.h>
```

```
#include <cstl/clist.h>
```

```
#include <cstl/calgorithm.h>
```

```
static void _print(const void* cpv_input, void* pv_output);
```

```
static void _less4(const void* cpv_input, void* pv_output);
```

```
int main(int argc, char* argv[])
```

```
{
```

```
    vector_t t_v = create_vector(int);
```

```
    list_t t_l = create_list(int);
```

```
    int i = 0;
```

```
    iterator_t t_pos;
```

```
    vector_init(&t_v);
```

```
    list_init_n(&t_l, 20);
```

```
    for(i = 2; i <= 6; ++i)
```

```
    {
```

```
        vector_push_back(&t_v, i);
```

```
    }
```

```
    for(i = 4; i <= 9; ++i)
```

```
    {
```

```
        vector_push_back(&t_v, i);
```

```
    }
```

```
    for(i = 1; i <= 7; ++i)
```

```
    {
```

```
        vector_push_back(&t_v, i);
```

```
    }
```

```
    printf("vector:          ");
```

```
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
```

```

printf("\n");

t_pos = algo_remove(vector_begin(&t_v), vector_end(&t_v), 5);
printf("size not changed:  ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");
vector_erase_range(&t_v, t_pos, vector_end(&t_v));
printf("size changed:      ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

list_erase_range(
    &t_l,
    algo_remove_copy_if(
        vector_begin(&t_v), vector_end(&t_v), list_begin(&t_l), _less4),
    list_end(&t_l));
printf("list(< 4 removed): ");
algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
printf("\n");

vector_destroy(&t_v);
list_destroy(&t_l);

return 0;
}

```

```

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

```

static void _less4(const void* cpv_input, void* pv_output)
{
    if(*(int*)cpv_input < 4)
    {
        *(bool_t*)pv_output = true;
    }
    else
    {
        *(bool_t*)pv_output = false;
    }
}

```

结果:

```

vector:          2 3 4 5 6 4 5 6 7 8 9 1 2 3 4 5 6 7
size not changed: 2 3 4 6 4 6 7 8 9 1 2 3 4 6 7 5 6 7

```

```
size changed:      2 3 4 6 4 6 7 8 9 1 2 3 4 6 7
list(< 4 removed): 4 6 4 6 7 8 9 4 6 7
```

移除相邻的符合条件的数据:

`algo_unique()`, `algo_unique_if()`, `algo_unique_copy()`, `algo_unique_copy_if()`

这4个算法是移除相邻满足条件的数据, 不相邻的不会受到影响.

```
#include <stdio.h>
```

```
#include <cstl/cvector.h>
```

```
#include <cstl/clist.h>
```

```
#include <cstl/calgorithm.h>
```

```
static void _print(const void* cpv_input, void* pv_output);
```

```
int main(int argc, char* argv[])
```

```
{
```

```
    vector_t t_v1 = create_vector(int);
```

```
    vector_t t_v2 = create_vector(int);
```

```
    list_t t_l = create_list(int);
```

```
    iterator_t t_pos;
```

```
    vector_init(&t_v1);
```

```
    vector_push_back(&t_v1, 1);
```

```
    vector_push_back(&t_v1, 4);
```

```
    vector_push_back(&t_v1, 4);
```

```
    vector_push_back(&t_v1, 6);
```

```
    vector_push_back(&t_v1, 1);
```

```
    vector_push_back(&t_v1, 2);
```

```
    vector_push_back(&t_v1, 2);
```

```
    vector_push_back(&t_v1, 3);
```

```
    vector_push_back(&t_v1, 1);
```

```
    vector_push_back(&t_v1, 6);
```

```
    vector_push_back(&t_v1, 6);
```

```
    vector_push_back(&t_v1, 6);
```

```
    vector_push_back(&t_v1, 5);
```

```
    vector_push_back(&t_v1, 7);
```

```
    vector_push_back(&t_v1, 5);
```

```
    vector_push_back(&t_v1, 4);
```

```
    vector_push_back(&t_v1, 4);
```

```
    vector_init_copy(&t_v2, &t_v1);
```

```
    list_init_n(&t_l, vector_size(&t_v1));
```

```
    printf("vector:      ");
```

```
    algo_for_each(vector_begin(&t_v1), vector_end(&t_v1), _print);
```

```
    printf("\n");
```

```

    t_pos = algo_unique(vector_begin(&t_v1), vector_end(&t_v1));
    printf("size not changed: ");
    algo_for_each(vector_begin(&t_v1), vector_end(&t_v1), _print);
    printf("\n");
    vector_erase_range(&t_v1, t_pos, vector_end(&t_v1));
    printf("size changed:      ");
    algo_for_each(vector_begin(&t_v1), vector_end(&t_v1), _print);
    printf("\n");

    list_erase_range(
        &t_l,
        algo_unique_copy_if(
            vector_begin(&t_v2), vector_end(&t_v2),
            list_begin(&t_l), fun_great_int),
        list_end(&t_l));
    printf("list:                ");
    algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
    printf("\n");

    vector_destroy(&t_v1);
    vector_destroy(&t_v2);
    list_destroy(&t_l);

    return 0;
}

```

```

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

```

vector:      1 4 4 6 1 2 2 3 1 6 6 6 5 7 5 4 4
size not changed: 1 4 6 1 2 3 1 6 5 7 5 4 5 7 5 4 4
size changed:   1 4 6 1 2 3 1 6 5 7 5 4
list:          1 4 4 6 6 6 6 7

```

7. 逆序

逆序算法是将数据区间内的数据逆序: `algo_reverse()`, `algo_reverse_copy()`

```

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/clist.h>

```

```

#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(int);
    list_t t_l = create_list(int);
    int i = 0;
    iterator_t t_first, t_last;

    vector_init(&t_v);
    for(i = 1; i <= 9; ++i)
    {
        vector_push_back(&t_v, i);
    }
    list_init_n(&t_l, vector_size(&t_v));

    printf("vector:          ");
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");
    algo_reverse(vector_begin(&t_v), vector_end(&t_v));
    printf("reverse:          ");
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");
    t_first = vector_begin(&t_v);
    t_last = vector_end(&t_v);
    iterator_next(&t_first), iterator_prev(&t_last);
    algo_reverse(t_first, t_last);
    printf("reverse subrange: ");
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");

    algo_reverse_copy(vector_begin(&t_v), vector_end(&t_v), list_begin(&t_l));
    printf("list:          ");
    algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
    printf("\n");

    vector_destroy(&t_v);
    list_destroy(&t_l);

    return 0;
}

```

```
static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}
```

结果:

```
vector:          1 2 3 4 5 6 7 8 9
reverse:         9 8 7 6 5 4 3 2 1
reverse subrange: 9 2 3 4 5 6 7 8 1
list:            1 8 7 6 5 4 3 2 9
```

8. 旋转数据

旋转数据是在同一个数据区间中将两部分数据互相掉换, 它与 `algo_swap_ranges()` 不同, `algo_swap_range()` 相互掉换的数据区间大小相同, 并且可以不同属于一个区间, 旋转数据算法是将一个数据区间的两部分互相掉换, 并且两部分大小可以不同.

```
#include <stdio.h>
```

```
#include <cstl/cvector.h>
```

```
#include <cstl/clist.h>
```

```
#include <cstl/calgorithm.h>
```

```
static void _print(const void* cpv_input, void* pv_output);
```

```
int main(int argc, char* argv[])
```

```
{
```

```
    vector_t t_v = create_vector(int);
```

```
    list_t t_l = create_list(int);
```

```
    int i = 0;
```

```
    iterator_t t_pos;
```

```
    vector_init(&t_v);
```

```
    for(i = 0; i < 10; ++i)
```

```
    {
```

```
        vector_push_back(&t_v, i);
```

```
    }
```

```
    list_init_n(&t_l, vector_size(&t_v));
```

```
    printf("vector:          ");
```

```
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
```

```
    printf("\n");
```

```
    t_pos = vector_begin(&t_v);
```

```
    iterator_next(&t_pos);
```



```

algo_rotate(vector_begin(&t_v), t_pos, vector_end(&t_v));
printf("one left:      ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

t_pos = vector_end(&t_v);
iterator_prev_n(&t_pos, 2);
algo_rotate(vector_begin(&t_v), t_pos, vector_end(&t_v));
printf("two right:     ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

algo_rotate_copy(
    vector_begin(&t_v),
    algo_find(vector_begin(&t_v), vector_end(&t_v), 4),
    vector_end(&t_v),
    list_begin(&t_l));
printf("list(4 first): ");
algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
printf("\n");

vector_destroy(&t_v);
list_destroy(&t_l);

return 0;
}

```

```

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

```

vector:      0 1 2 3 4 5 6 7 8 9
one left:    1 2 3 4 5 6 7 8 9 0
two right:   9 0 1 2 3 4 5 6 7 8
list(4 first): 4 5 6 7 8 9 0 1 2 3

```

9. 随机重排

随机算法包括两种,第一种是将数据区间中的数据随机重排,第二种是对数据区间中的数据随机取样.

随机重排: `algo_random_shuffle()`, `algo_random_shuffle_if()`

```
#include <stdio.h>
```

```

#include <cstdlib/cvector.h>
#include <cstdlib/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(int);
    int i = 0;

    vector_init(&t_v);
    for(i = 0; i < 10; ++i)
    {
        vector_push_back(&t_v, i);
    }
    printf("vector:  ");
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");

    algo_random_shuffle(vector_begin(&t_v), vector_end(&t_v));
    printf("shuffled: ");
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");

    algo_sort(vector_begin(&t_v), vector_end(&t_v));
    printf("sorted:  ");
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");

    algo_random_shuffle(vector_begin(&t_v), vector_end(&t_v));
    printf("shuffled: ");
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");

    vector_destroy(&t_v);

    return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

```
vector:    0 1 2 3 4 5 6 7 8 9
shuffled:  4 1 6 3 7 9 2 5 0 8
sorted:    0 1 2 3 4 5 6 7 8 9
shuffled:  2 1 9 5 7 0 4 3 8 6
```

随机抽样:

随机采样是将第一个数据区间中的数据随机的取出放在第二个数据区间中.

```
algo_random_sample(), algo_random_sample_if(),
algo_random_sample_n(), algo_random_sample_n_if()
```

在第一个区间中的数据在第二个区间中最多只出现一次, 第二个区间中的顺序是任意的, 但不会影响第一个区间中的数据.

```
#include <stdio.h>
#include <cstdlib/cvector.h>
#include <cstdlib/cdeque.h>
#include <cstdlib/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(int);
    deque_t t_dq = create_deque(int);
    int i = 0;

    vector_init(&t_v);
    for(i = 0; i < 15; ++i)
    {
        vector_push_back(&t_v, i);
    }
    printf("vector: ");
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");

    deque_init_n(&t_dq, 10);
    algo_random_sample(
        vector_begin(&t_v), vector_end(&t_v),
        deque_begin(&t_dq), deque_end(&t_dq));
    printf("deque:   ");
    algo_for_each(deque_begin(&t_dq), deque_end(&t_dq), _print);
    printf("\n");

    deque_resize(&t_dq, 20);
    algo_random_sample(
        vector_begin(&t_v), vector_end(&t_v),
```

```

        deque_begin(&t_dq), deque_end(&t_dq));
printf("deque:   ");
algo_for_each(deque_begin(&t_dq), deque_end(&t_dq), _print);
printf("\n");

vector_destroy(&t_v);
deque_destroy(&t_dq);
return 0;
}

```

```

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

```

vector: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
deque:   14 12 10 3 4 5 6 11 8 9
deque:   0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 0 0 0 0 0

```

10. 划分

划分算法是将符合条件的数据移动到数据区间的前部, 剩余的移动到数据区间的后部. 划分分为稳定划分和不稳定的划分: `algo_partition()`, `algo_stable_partition()`

```

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/clist.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);
static void _is_even(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t t_v1 = create_vector(int);
    vector_t t_v2 = create_vector(int);
    int i = 0;
    iterator_t t_pos;

    vector_init(&t_v1);
    vector_init(&t_v2);
    for(i = 1; i <= 9; ++i)
    {

```

```

        vector_push_back(&t_v1, i);
        vector_push_back(&t_v2, i);
    }
    printf("vector1: ");
    algo_for_each(vector_begin(&t_v1), vector_end(&t_v1), _print);
    printf("\n");
    printf("vector2: ");
    algo_for_each(vector_begin(&t_v2), vector_end(&t_v2), _print);
    printf("\n");

    t_pos = algo_partition(vector_begin(&t_v1), vector_end(&t_v1), _is_even);
    printf("vector1: ");
    algo_for_each(vector_begin(&t_v1), vector_end(&t_v1), _print);
    printf("\n");
    printf("first odd element: %d\n", *(int*)iterator_get_pointer(&t_pos));

    t_pos = algo_stable_partition(
        vector_begin(&t_v2), vector_end(&t_v2), _is_even);
    printf("vector2: ");
    algo_for_each(vector_begin(&t_v2), vector_end(&t_v2), _print);
    printf("\n");
    printf("first odd element: %d\n", *(int*)iterator_get_pointer(&t_pos));

    vector_destroy(&t_v1);
    vector_destroy(&t_v2);
    return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

static void _is_even(const void* cpv_input, void* pv_output)
{
    if(*(int*)cpv_input % 2 == 0)
    {
        *(bool_t*)pv_output = true;
    }
    else
    {
        *(bool_t*)pv_output = false;
    }
}

```

结果:

```
vector1: 1 2 3 4 5 6 7 8 9
vector2: 1 2 3 4 5 6 7 8 9
vector1: 8 2 6 4 5 3 7 1 9
first odd element: 5
vector2: 2 4 6 8 1 3 5 7 9
first odd element: 1
```

第四节 排序算法

排序算法是将数据区间排序或者是基于有序的数据区间的操作。

<code>algo_sort()</code>	排序.
<code>algo_sort_if()</code>	使用特定的规则排序.
<code>algo_stable_sort()</code>	稳定排序.
<code>algo_stable_sort_if()</code>	使用特定规则的稳定排序.
<code>algo_partial_sort()</code>	局部排序.
<code>algo_partial_sort_if()</code>	使用特定规则局部排序.
<code>algo_partial_sort_copy()</code>	将局部排序的结果拷贝到两个区间内.
<code>algo_partial_sort_copy_if()</code>	使用特定规则局部排序, 并将结果拷贝到另一个区间内.
<code>algo_is_sorted()</code>	判断数据区间是否已排序.
<code>algo_is_sorted_if()</code>	判断数据区间是否根据特定规则已经排序.
<code>algo_nth_element()</code>	获得排序后的第 n 个数据.
<code>algo_nth_element_if()</code>	获得使用特定规则排序后的第 n 个数据.
<code>algo_lower_bound()</code>	获得第一个不小于指定数据的迭代器.
<code>algo_lower_bound_if()</code>	同上, 但是使用特定规则.
<code>algo_upper_bound()</code>	获得第一个大于指定数据的迭代器.
<code>algo_upper_bound_if()</code>	同上, 但是使用特定规则.
<code>algo_equal_range()</code>	获得包含指定数据的数据区间.
<code>algo_equal_range_if()</code>	同上, 但是使用特定规则.
<code>algo_binary_search()</code>	二叉查找.
<code>algo_binary_search_if()</code>	使用特定规则进行二叉查找.
<code>algo_merge()</code>	合并.
<code>algo_merge_if()</code>	使用特定规则合并.
<code>algo_inplace_merge()</code>	将一个数据区间的两个已排序的部分合并.
<code>algo_inplace_merge_if()</code>	使用特定规则将一个数据区间的两个已排序的部分合并.
<code>algo_includes()</code>	判断第一个数据区间中是否包含第二个区间中的所有数据.
<code>algo_includes_if()</code>	使用特定规则, 判断第一个数据区间中是否包含第二个区间中的所有数据.
<code>algo_set_union()</code>	求并集.
<code>algo_set_union_if()</code>	使用特定规则, 求并集.
<code>algo_set_intersection()</code>	求交集.
<code>algo_set_intersection_if()</code>	使用特定规则, 求交集.
<code>algo_set_difference()</code>	求差集.

<code>algo_set_difference_if()</code>	使用特定规则, 求差集.
<code>algo_set_symmetric_difference()</code>	求对称差集.
<code>algo_set_symmetric_difference_if()</code>	使用特定规则, 求对称差集.
<code>algo_push_heap()</code>	向堆中插入一个数据.
<code>algo_push_heap_if()</code>	使用特定规则向堆中插入一个数据.
<code>algo_pop_heap()</code>	从堆中弹出一个数据.
<code>algo_pop_heap_if()</code>	使用特定规则从堆中弹出一个数据.
<code>algo_make_heap()</code>	将数据区间转变成一个堆.
<code>algo_make_heap_if()</code>	使用特定规则将数据区间转变成一个堆.
<code>algo_sort_heap()</code>	堆排序.
<code>algo_sort_heap_if()</code>	使用特定规则进行堆排序.
<code>algo_is_heap()</code>	判断一个数据区间是否是堆.
<code>algo_is_heap_if()</code>	使用特定规则判断一个数据区间是否是堆.
<code>algo_min()</code>	获得最小数据的迭代器.
<code>algo_min_if()</code>	使用特定规则获得最小数据的迭代器.
<code>algo_max()</code>	获得最大数据的迭代器.
<code>algo_max_if()</code>	使用特定规则获得最大数据的迭代器.
<code>algo_min_element()</code>	获得数据区间中最小数据的迭代器.
<code>algo_min_element_if()</code>	使用特定规则获得数据区间中最小数据的迭代器.
<code>algo_max_element()</code>	获得数据区间中最大数据的迭代器.
<code>algo_max_element_if()</code>	使用特定规则获得数据区间中最大数据的迭代器.
<code>algo_lexicographical_compare()</code>	按字典顺序比较.
<code>algo_lexicographical_compare_if()</code>	使用特定规则按字典顺序比较.
<code>algo_lexicographical_compare_3way()</code>	按字典顺序比较, 返回 3 种结果.
<code>algo_lexicographical_compare_3way_if()</code>	使用特定规则按字典顺序比较, 返回 3 种结果.
<code>algo_next_permutation()</code>	将数据区间中的数据转变到下一个排序组合.
<code>algo_next_permutation_if()</code>	使用特定规则将数据区间中的数据转变到下一个排列组合.
<code>algo_prev_permutation()</code>	将数据区间中的数据转变到上一个排序组合.
<code>algo_prev_permutation_if()</code>	使用特定规则将数据区间中的数据转变到上一个排列组合.

1. 排序

排序算法非常重要, 所以 `libcstl` 提供多种排序算法:

基本排序: `algo_sort()`, `algo_sort_if()`.

```
#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
```

```

vector_t t_v = create_vector(int);
int i = 0;

vector_init(&t_v);
for(i = 1; i <= 9; ++i)
{
    vector_push_back(&t_v, i);
    vector_push_back(&t_v, i);
}
algo_random_shuffle(vector_begin(&t_v), vector_end(&t_v));
printf("vector:   ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

algo_sort(vector_begin(&t_v), vector_end(&t_v));
printf("sorted:   ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

algo_sort_if(vector_begin(&t_v), vector_end(&t_v), fun_great_int);
printf("sorted >: ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

vector_destroy(&t_v);
return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

```

vector:   9 8 7 6 9 5 6 2 1 5 3 4 2 7 8 1 3 4
sorted:   1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
sorted >: 9 9 8 8 7 7 6 6 5 5 4 4 3 3 2 2 1 1

```

稳定排序: `algo_stable_sort()`, `algo_stable_sort_if()`.

```

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/cdeque.h>
#include <cstl/calgorithm.h>

```

```

struct abc_t

```



```

{
    int n_value;
    long l_value;
    double d_key;
};

static void _print_abc(const void* cpv_input, void* pv_output);
static void _abcgreat(
    const void* cpv_first, const void* cpv_second, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(struct abc_t);
    deque_t t_dq = create_deque(struct abc_t);
    struct abc_t t_abc;

    vector_init(&t_v);
    deque_init(&t_dq);
    t_abc.n_value = 70;
    t_abc.l_value = 934000;
    t_abc.d_key = 0.24;
    vector_push_back(&t_v, t_abc);
    deque_push_back(&t_dq, t_abc);
    t_abc.n_value = 100;
    t_abc.l_value = 3000;
    t_abc.d_key = 2.09;
    vector_push_back(&t_v, t_abc);
    deque_push_back(&t_dq, t_abc);
    t_abc.n_value = 2;
    t_abc.l_value = -18;
    t_abc.d_key = 110.00;
    vector_push_back(&t_v, t_abc);
    deque_push_back(&t_dq, t_abc);
    t_abc.n_value = -902;
    t_abc.l_value = 88000;
    t_abc.d_key = -10.007;
    vector_push_back(&t_v, t_abc);
    deque_push_back(&t_dq, t_abc);
    t_abc.n_value = 302;
    t_abc.l_value = 800;
    t_abc.d_key = 0.067;
    vector_push_back(&t_v, t_abc);
    deque_push_back(&t_dq, t_abc);
    t_abc.n_value = 2;

```

```
t_abc.l_value = 7;
t_abc.d_key = 0.067;
vector_push_back(&t_v, t_abc);
deque_push_back(&t_dq, t_abc);
t_abc.n_value = 1;
t_abc.l_value = 89;
t_abc.d_key = 1618.9;
vector_push_back(&t_v, t_abc);
deque_push_back(&t_dq, t_abc);
t_abc.n_value = -10000;
t_abc.l_value = -18;
t_abc.d_key = -8.9;
vector_push_back(&t_v, t_abc);
deque_push_back(&t_dq, t_abc);
t_abc.n_value = 0;
t_abc.l_value = 0;
t_abc.d_key = 0.0;
vector_push_back(&t_v, t_abc);
deque_push_back(&t_dq, t_abc);
t_abc.n_value = 2340;
t_abc.l_value = 890;
t_abc.d_key = 0.009;
vector_push_back(&t_v, t_abc);
deque_push_back(&t_dq, t_abc);
t_abc.n_value = 2340;
t_abc.l_value = 890;
t_abc.d_key = 0.067;
vector_push_back(&t_v, t_abc);
deque_push_back(&t_dq, t_abc);
t_abc.n_value = 2;
t_abc.l_value = 7;
t_abc.d_key = 0.067;
vector_push_back(&t_v, t_abc);
deque_push_back(&t_dq, t_abc);
t_abc.n_value = 1111;
t_abc.l_value = 11189;
t_abc.d_key = 1618.9;
vector_push_back(&t_v, t_abc);
deque_push_back(&t_dq, t_abc);
t_abc.n_value = -2340;
t_abc.l_value = -890;
t_abc.d_key = 0.009;
vector_push_back(&t_v, t_abc);
deque_push_back(&t_dq, t_abc);
```

```

t_abc.n_value = 992;
t_abc.l_value = 918;
t_abc.d_key = 110.00;
vector_push_back(&t_v, t_abc);
deque_push_back(&t_dq, t_abc);
t_abc.n_value = 40;
t_abc.l_value = 90;
t_abc.d_key = 0.009;
vector_push_back(&t_v, t_abc);
deque_push_back(&t_dq, t_abc);
t_abc.n_value = 2;
t_abc.l_value = 8;
t_abc.d_key = 110.00;
vector_push_back(&t_v, t_abc);
deque_push_back(&t_dq, t_abc);

printf("vector:\n");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print_abc);
printf("\n");
printf("deque:\n");
algo_for_each(deque_begin(&t_dq), deque_end(&t_dq), _print_abc);
printf("\n");

algo_sort_if(vector_begin(&t_v), vector_end(&t_v), _abcgreat);
printf("sorted vector:\n");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print_abc);
printf("\n");

algo_stable_sort_if(deque_begin(&t_dq), deque_end(&t_dq), _abcgreat);
printf("stable sorted deque:\n");
algo_for_each(deque_begin(&t_dq), deque_end(&t_dq), _print_abc);
printf("\n");

vector_destroy(&t_v);
deque_destroy(&t_dq);
return 0;
}

static void _abcgreat(const void* cpv_first, const void* cpv_second, void* pv_output)
{
    if(((struct abc_t*)cpv_first)->d_key > ((struct abc_t*)cpv_second)->d_key)
    {
        *(bool_t*)pv_output = true;
    }
}

```

```

else
{
    *(bool_t*)pv_output = false;
}
}
static void _print_abc(const void* cpv_input, void* pv_output)
{
    pv_output = NULL;
    printf("(%d, %ld, %f)\n",
        ((struct abc_t*)cpv_input)->n_value,
        ((struct abc_t*)cpv_input)->l_value,
        ((struct abc_t*)cpv_input)->d_key);
}

```

结果:

vector:

```

(70, 934000, 0.240000)
(100, 3000, 2.090000)
(2, -18, 110.000000)
(-902, 88000, -10.007000)
(302, 800, 0.067000)
(2, 7, 0.067000)
(1, 89, 1618.900000)
(-10000, -18, -8.900000)
(0, 0, 0.000000)
(2340, 890, 0.009000)
(2340, 890, 0.067000)
(2, 7, 0.067000)
(1111, 11189, 1618.900000)
(-2340, -890, 0.009000)
(992, 918, 110.000000)
(40, 90, 0.009000)
(2, 8, 110.000000)

```

deque:

```

(70, 934000, 0.240000)
(100, 3000, 2.090000)
(2, -18, 110.000000)
(-902, 88000, -10.007000)
(302, 800, 0.067000)
(2, 7, 0.067000)
(1, 89, 1618.900000)
(-10000, -18, -8.900000)
(0, 0, 0.000000)
(2340, 890, 0.009000)

```

(2340, 890, 0.067000)
(2, 7, 0.067000)
(1111, 11189, 1618.900000)
(-2340, -890, 0.009000)
(992, 918, 110.000000)
(40, 90, 0.009000)
(2, 8, 110.000000)

sorted vector:

(1111, 11189, 1618.900000)
(1, 89, 1618.900000)
(992, 918, 110.000000)
(2, 8, 110.000000)
(2, -18, 110.000000)
(100, 3000, 2.090000)
(70, 934000, 0.240000)
(2340, 890, 0.067000)
(302, 800, 0.067000)
(2, 7, 0.067000)
(2, 7, 0.067000)
(-2340, -890, 0.009000)
(40, 90, 0.009000)
(2340, 890, 0.009000)
(0, 0, 0.000000)
(-10000, -18, -8.900000)
(-902, 88000, -10.007000)

stable sorted deque:

(1, 89, 1618.900000)
(1111, 11189, 1618.900000)
(2, -18, 110.000000)
(992, 918, 110.000000)
(2, 8, 110.000000)
(100, 3000, 2.090000)
(70, 934000, 0.240000)
(302, 800, 0.067000)
(2, 7, 0.067000)
(2340, 890, 0.067000)
(2, 7, 0.067000)
(2340, 890, 0.009000)
(-2340, -890, 0.009000)
(40, 90, 0.009000)
(0, 0, 0.000000)
(-10000, -18, -8.900000)

`(-902, 88000, -10.007000)`

从结果中可以看出使用 `algo_sort_if()` 排序的 `vector_t` 没有保持键值相同的数据的原有顺序, 但是使用 `algo_stable_sort_if()` 排序的 `deque_t` 排序后保持键值相同的数据的原有顺序. 这就是稳定排序和不稳定排序, 的不同点.

部分排序: `algo_partial_sort()`, `algo_partial_sort_if()`, `algo_partial_sort_copy()`, `algo_partial_sort_copy_if()`.

前两个算法实现的是一个数据区间的前部排序, 后部顺序是未知的. 后两个算法是将排序后的数据拷贝到新的数据区间, 拷贝的数据个数是由两个区间中个数少的一方决定的.

下面是前两个算法的例子:

```
#include <stdio.h>
#include <cstl/cdeque.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    deque_t t_dq = create_deque(int);
    int i = 0;
    iterator_t t_pos;

    deque_init(&t_dq);
    for(i = 3; i <= 7; ++i)
    {
        deque_push_back(&t_dq, i);
    }
    for(i = 2; i <= 6; ++i)
    {
        deque_push_back(&t_dq, i);
    }
    for(i = 1; i <= 5; ++i)
    {
        deque_push_back(&t_dq, i);
    }
    algo_for_each(deque_begin(&t_dq), deque_end(&t_dq), _print);
    printf("\n");

    t_pos = deque_begin(&t_dq);
    iterator_advance(&t_pos, 5);
    algo_partial_sort(deque_begin(&t_dq), t_pos, deque_end(&t_dq));
    algo_for_each(deque_begin(&t_dq), deque_end(&t_dq), _print);
    printf("\n");
}
```

```

algo_partial_sort_if(
    deque_begin(&t_dq), t_pos, deque_end(&t_dq), fun_great_int);
algo_for_each(deque_begin(&t_dq), deque_end(&t_dq), _print);
printf("\n");

algo_partial_sort(deque_begin(&t_dq), deque_end(&t_dq), deque_end(&t_dq));
algo_for_each(deque_begin(&t_dq), deque_end(&t_dq), _print);
printf("\n");

deque_destroy(&t_dq);
return 0;
}

```

```

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

```

3 4 5 6 7 2 3 4 5 6 1 2 3 4 5
1 2 2 3 3 7 6 5 5 6 4 4 3 4 5
7 6 6 5 5 1 2 2 3 3 4 4 3 4 5
1 2 2 3 3 3 4 4 4 5 5 5 6 6 7

```

下面是后两个算法的例子:

```

#include <stdio.h>
#include <cstdlib/cdeque.h>
#include <cstdlib/cvector.h>
#include <cstdlib/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    deque_t t_dq = create_deque(int);
    vector_t t_v6 = create_vector(int); /* initialized with 6 elements */
    vector_t t_v30 = create_vector(int); /* initialized with 30 elements */
    int i = 0;
    iterator_t t_pos;

    deque_init(&t_dq);
    vector_init_n(&t_v6, 6);
    vector_init_n(&t_v30, 30);
    for(i = 3; i <= 7; ++i)
    {

```

```

        deque_push_back(&t_dq, i);
    }
    for(i = 2; i <= 6; ++i)
    {
        deque_push_back(&t_dq, i);
    }
    for(i = 1; i <= 5; ++i)
    {
        deque_push_back(&t_dq, i);
    }
    algo_for_each(deque_begin(&t_dq), deque_end(&t_dq), _print);
    printf("\n");

    t_pos = algo_partial_sort_copy(
        deque_begin(&t_dq), deque_end(&t_dq),
        vector_begin(&t_v6), vector_end(&t_v6));
    algo_for_each(vector_begin(&t_v6), t_pos, _print);
    printf("\n");

    t_pos = algo_partial_sort_copy_if(
        deque_begin(&t_dq), deque_end(&t_dq),
        vector_begin(&t_v30), vector_end(&t_v30),
        fun_great_int);
    algo_for_each(vector_begin(&t_v30), t_pos, _print);
    printf("\n");

    deque_destroy(&t_dq);
    vector_destroy(&t_v6);
    vector_destroy(&t_v30);
    return 0;
}

```

```

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

```

3 4 5 6 7 2 3 4 5 6 1 2 3 4 5
1 2 2 3 3 3
7 6 6 5 5 5 4 4 4 3 3 3 2 2 1

```

判断是否排序: `algo_is_sorted()`, `algo_is_sorted_if()`.

这两个算法是用来判断数据区间是否已经排序了, 后缀为 `if` 的版本接受一个特殊的规则, 它只能判断数据区间是否通过该规则排序. 下面是这两个算法的例子:


```

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(int);
    int i = 0;

    vector_init(&t_v);
    for(i = 0; i < 20; ++i)
    {
        vector_push_back(&t_v, i);
    }
    algo_random_shuffle(vector_begin(&t_v), vector_end(&t_v));
    printf("vector: ");
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");
    if(algo_is_sorted(vector_begin(&t_v), vector_end(&t_v)))
    {
        printf("sorted\n");
    }
    else
    {
        printf("not sorted\n");
    }
    if(algo_is_sorted_if(vector_begin(&t_v), vector_end(&t_v), fun_great_int))
    {
        printf("sorted with >\n");
    }
    else
    {
        printf("not sorted with >\n");
    }

    algo_sort(vector_begin(&t_v), vector_end(&t_v));
    printf("vector: ");
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");
    if(algo_is_sorted(vector_begin(&t_v), vector_end(&t_v)))
    {
        printf("sorted\n");
    }

```

```

    }
    else
    {
        printf("not sorted\n");
    }
    if(algo_is_sorted_if(vector_begin(&t_v), vector_end(&t_v), fun_great_int))
    {
        printf("sorted with >\n");
    }
    else
    {
        printf("not sorted with >\n");
    }

    algo_sort_if(vector_begin(&t_v), vector_end(&t_v), fun_great_int);
    printf("vector: ");
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");
    if(algo_is_sorted(vector_begin(&t_v), vector_end(&t_v)))
    {
        printf("sorted\n");
    }
    else
    {
        printf("not sorted\n");
    }
    if(algo_is_sorted_if(vector_begin(&t_v), vector_end(&t_v), fun_great_int))
    {
        printf("sorted with >\n");
    }
    else
    {
        printf("not sorted with >\n");
    }

    vector_destroy(&t_v);
    return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

```

vector: 10 16 4 7 15 2 11 13 5 6 14 18 12 1 9 19 17 8 0 3
not sorted
not sorted with >
vector: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
sorted
not sorted with >
vector: 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
not sorted
sorted with >

```

根据数据位置排序: `algo_nth_element()`, `algo_nth_element_if()`

这两个算法保证执行后第 n 个位置的数据是整个区间排序后的数据, 并且前面的数据都小于该数据但是不保证顺序, 后面的数据都大于这个数据但是不保证顺序. 下面是两个算法的例子:

```

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t t_v1 = create_vector(int);
    vector_t t_v2 = create_vector(int);
    int i = 0;
    iterator_t t_pos;

    vector_init(&t_v1);
    vector_init(&t_v2);
    for(i = 0; i < 10; ++i)
    {
        vector_push_back(&t_v1, i);
        vector_push_back(&t_v2, i);
    }

    algo_random_shuffle(vector_begin(&t_v1), vector_end(&t_v1));
    algo_random_shuffle(vector_begin(&t_v2), vector_end(&t_v2));

    algo_sort(vector_begin(&t_v1), vector_end(&t_v1));
    t_pos = vector_begin(&t_v2);
    iterator_advance(&t_pos, 5);
    algo_nth_element(vector_begin(&t_v2), t_pos, vector_end(&t_v2));

    printf("vector1: ");
    algo_for_each(vector_begin(&t_v1), vector_end(&t_v1), _print);
}

```

```

printf("\n");
printf("vector2: ");
algo_for_each(vector_begin(&t_v2), vector_end(&t_v2), _print);
printf("\n");

algo_sort_if(vector_begin(&t_v1), vector_end(&t_v1), fun_great_int);
t_pos = vector_begin(&t_v2);
iterator_advance(&t_pos, 4);
algo_nth_element_if(
    vector_begin(&t_v2), t_pos, vector_end(&t_v2), fun_great_int);

printf("vector1: ");
algo_for_each(vector_begin(&t_v1), vector_end(&t_v1), _print);
printf("\n");
printf("vector2: ");
algo_for_each(vector_begin(&t_v2), vector_end(&t_v2), _print);
printf("\n");

vector_destroy(&t_v1);
vector_destroy(&t_v2);
return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

```

vector1: 0 1 2 3 4 5 6 7 8 9
vector2: 0 1 4 2 3 5 6 7 9 8
vector1: 9 8 7 6 5 4 3 2 1 0
vector2: 8 9 7 6 5 3 2 4 1 0

```

2. 二分查找

二分查找算法都是基于有序的数据区间, 相对于顺序查找它可以获得对数级别的效率.

获得数据在数据区间中的上限或下限:

```

algo_lower_bound(), algo_lower_bound_if(),
algo_upper_bound(), algo_upper_bound_if().

```

前两个算法获得数据在区间中的下限迭代器, 后两个算法获得容器在区间中的上限迭代器. 关联容器已经提供获得数据上下限的操作函数, 如果要在关联容器中查找数据的上下限请使用容器本身的操作函数. 下面是这 4 个算法的例子:

```
#include <stdio.h>
```

```

#include <cstl/clist.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    list_t t_l = create_list(int);
    int i = 0;
    iterator_t t_lower, t_upper;

    list_init(&t_l);
    for(i = 0; i < 10; ++i)
    {
        list_push_back(&t_l, i);
        list_push_back(&t_l, i);
    }
    algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
    printf("\n");

    t_lower = algo_lower_bound(list_begin(&t_l), list_end(&t_l), 5);
    t_upper = algo_upper_bound(list_begin(&t_l), list_end(&t_l), 5);
    printf("5 could get position %d up to %d without break the sorting\n",
        iterator_distance(list_begin(&t_l), t_lower) + 1,
        iterator_distance(list_begin(&t_l), t_upper) + 1);

    list_insert(&t_l, algo_lower_bound(list_begin(&t_l), list_end(&t_l), 3), 3);
    list_insert(&t_l, algo_upper_bound(list_begin(&t_l), list_end(&t_l), 7), 7);
    algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
    printf("\n");

    list_destroy(&t_l);
    return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

```

0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
5 could get position 11 up to 13 without break the sorting
0 0 1 1 2 2 3 3 3 4 4 5 5 6 6 7 7 7 8 8 9 9

```

获得数据在数据区间中的上限和下限: `algo_equal_range()`, `algo_equal_range_if()`.

这两个算法返回由上限和下限组成的 `pair_t`, 所以这个函数的返回值不能忽略, 并且 `pair_t` 使用后要销毁. 下面是算法的例子:

```
#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(int);
    int i = 0;
    pair_t t_p;

    vector_init(&t_v);
    for(i = 0; i < 10; ++i)
    {
        vector_push_back(&t_v, i);
        vector_push_back(&t_v, i);
    }
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");

    t_p = algo_equal_range(vector_begin(&t_v), vector_end(&t_v), 5);
    printf("5 could get position %d up to %d without break the sorting\n",
        iterator_distance(vector_begin(&t_v), *(iterator_t*)t_p.first) + 1,
        iterator_distance(vector_begin(&t_v), *(iterator_t*)t_p.second) + 1);

    pair_destroy(&t_p);
    vector_destroy(&t_v);
    return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

结果:
0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
5 could get position 11 up to 13 without break the sorting
```

二分查找: `algo_binary_search()`, `algo_binary_search_if()`

使用二分查找法判断指定数据是否在数据区间中. 下面是使用这两个算法的例子:

```

#include <stdio.h>
#include <cstl/clist.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    list_t t_l = create_list(int);
    int i = 0;

    list_init(&t_l);
    for(i = 0; i < 10; ++i)
    {
        list_push_back(&t_l, i);
    }
    algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
    printf("\n");

    if(algo_binary_search(list_begin(&t_l), list_end(&t_l), 5))
    {
        printf("5 is present\n");
    }
    else
    {
        printf("5 isn't present\n");
    }
    if(algo_binary_search(list_begin(&t_l), list_end(&t_l), 42))
    {
        printf("42 is present\n");
    }
    else
    {
        printf("42 isn't present\n");
    }

    list_destroy(&t_l);
    return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

```
0 1 2 3 4 5 6 7 8 9
```

```
5 is present
```

```
42 isn't present
```

3. 合并

合并必须满足合并双方都是有序的, 并且排序规则相同. 合并包括将两个数据区间合并, 将一个数据区间的两个部分合并. 合并算法一共有 4 个:

```
algo_merge(), algo_merge_if(), algo_inplace_merge(), algo_inplace_merge_if().
```

前两个算法是用来合并两个不同的数据区间. 后两个算法是合并同一数据区间的两个部分. 下面是前两个算法的例子:

```
#include <stdio.h>
#include <cstl/clist.h>
#include <cstl/cdeque.h>
#include <cstl/cset.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    list_t t_l = create_list(int);
    set_t t_s = create_set(int);
    deque_t t_dq = create_deque(int);
    int i = 0;

    list_init(&t_l);
    for(i = 1; i <=6; ++i)
    {
        list_push_back(&t_l, i);
    }
    set_init(&t_s);
    for(i = 3; i <=8; ++i)
    {
        set_insert(&t_s, i);
    }
    deque_init_n(&t_dq, list_size(&t_l) + set_size(&t_s));

    printf("list: ");
    algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
```



```

printf("\n");
printf("set:  ");
algo_for_each(set_begin(&t_s), set_end(&t_s), _print);
printf("\n");

algo_merge(
    list_begin(&t_l), list_end(&t_l),
    set_begin(&t_s), set_end(&t_s),
    deque_begin(&t_dq));
printf("deque: ");
algo_for_each(deque_begin(&t_dq), deque_end(&t_dq), _print);
printf("\n");

list_destroy(&t_l);
set_destroy(&t_s);
deque_destroy(&t_dq);
return 0;
}

```

```

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

```

list: 1 2 3 4 5 6
set:  3 4 5 6 7 8
deque: 1 2 3 3 4 4 5 5 6 6 7 8

```

下面是后两个算法的例子:

```

#include <stdio.h>
#include <cstl/clist.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    list_t t_l = create_list(int);
    int i = 0;
    iterator_t t_pos;

    list_init(&t_l);
    for(i = 1; i <= 7; ++i)
    {

```

```

        list_push_back(&t_l, i);
    }
    for(i = 1; i <=8; ++i)
    {
        list_push_back(&t_l, i);
    }
    algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
    printf("\n");

    t_pos = algo_find(list_begin(&t_l), list_end(&t_l), 7);
    iterator_next(&t_pos);
    algo_inplace_merge(list_begin(&t_l), t_pos, list_end(&t_l));
    algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
    printf("\n");

    list_destroy(&t_l);
    return 0;
}

```

```

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

```

1 2 3 4 5 6 7 1 2 3 4 5 6 7 8
1 1 2 2 3 3 4 4 5 5 6 6 7 7 8

```

4. 与集合有关的算法

libcstl 提供了一些与集合有关的算法, 包括判断子集, 求并集, 求交集, 求差集, 求对称差集.

判断是否为子集: `algo_includes()`, `algo_includes_if()`

判断第二个数据区间中的数据是否全部包含于第一个数据区间中, 同时要求两个数据区间内的数据必须是有序的.

下面是使用这两个算法的例子:

```

#include <stdio.h>
#include <cstl/clist.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{

```

```

list_t t_l = create_list(int);
vector_t t_v = create_vector(int);
int i = 0;

list_init(&t_l);
for(i = 0; i < 10; ++i)
{
    list_push_back(&t_l, i);
}
printf("list:  ");
algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
printf("\n");

vector_init(&t_v);
vector_push_back(&t_v, 3);
vector_push_back(&t_v, 4);
vector_push_back(&t_v, 7);
printf("vector: ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

if(algo_includes(
    list_begin(&t_l), list_end(&t_l), vector_begin(&t_v), vector_end(&t_v)))
{
    printf("all elements of vector are also in list\n");
}
else
{
    printf("not all elements of vector are also in list\n");
}

list_destroy(&t_l);
vector_destroy(&t_v);
return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

list: 0 1 2 3 4 5 6 7 8 9

vector: 3 4 7

all elements of vector are also in list

求并集: algo_set_union(), algo_set_union_if()

求交集: algo_set_intersection(), algo_set_intersection_if()

求差集: algo_set_difference(), algo_set_difference_if()

求对称差集: algo_set_symmetric_difference(), algo_set_symmetric_difference_if()

下面是关于这些算法的例子:

```
#include <stdio.h>
#include <cstl/clist.h>
#include <cstl/cvector.h>
#include <cstl/cdeque.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    list_t t_l = create_list(int);
    vector_t t_v = create_vector(int);
    deque_t t_dq = create_deque(int);
    iterator_t t_pos;

    list_init(&t_l);
    list_push_back(&t_l, 1);
    list_push_back(&t_l, 2);
    list_push_back(&t_l, 2);
    list_push_back(&t_l, 4);
    list_push_back(&t_l, 6);
    list_push_back(&t_l, 7);
    list_push_back(&t_l, 7);
    list_push_back(&t_l, 9);
    vector_init(&t_v);
    vector_push_back(&t_v, 2);
    vector_push_back(&t_v, 2);
    vector_push_back(&t_v, 2);
    vector_push_back(&t_v, 3);
    vector_push_back(&t_v, 6);
    vector_push_back(&t_v, 6);
    vector_push_back(&t_v, 8);
    vector_push_back(&t_v, 9);
    deque_init_n(&t_dq, list_size(&t_l) + vector_size(&t_v));

    printf("list:                ");
    algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
    printf("\n");
```

```

printf("vector:                ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

```

```

t_pos = algo_merge(
    list_begin(&t_l), list_end(&t_l),
    vector_begin(&t_v), vector_end(&t_v),
    deque_begin(&t_dq));
printf("merge:                ");
algo_for_each(deque_begin(&t_dq), t_pos, _print);
printf("\n");

```

```

t_pos = algo_set_union(
    list_begin(&t_l), list_end(&t_l),
    vector_begin(&t_v), vector_end(&t_v),
    deque_begin(&t_dq));
printf("union:                ");
algo_for_each(deque_begin(&t_dq), t_pos, _print);
printf("\n");

```

```

t_pos = algo_set_intersection(
    list_begin(&t_l), list_end(&t_l),
    vector_begin(&t_v), vector_end(&t_v),
    deque_begin(&t_dq));
printf("intersection:        ");
algo_for_each(deque_begin(&t_dq), t_pos, _print);
printf("\n");

```

```

t_pos = algo_set_difference(
    list_begin(&t_l), list_end(&t_l),
    vector_begin(&t_v), vector_end(&t_v),
    deque_begin(&t_dq));
printf("difference:            ");
algo_for_each(deque_begin(&t_dq), t_pos, _print);
printf("\n");

```

```

t_pos = algo_set_symmetric_difference(
    list_begin(&t_l), list_end(&t_l),
    vector_begin(&t_v), vector_end(&t_v),
    deque_begin(&t_dq));
printf("symmetric difference:  ");
algo_for_each(deque_begin(&t_dq), t_pos, _print);
printf("\n");

```

```

    list_destroy(&t_l);
    vector_destroy(&t_v);
    deque_destroy(&t_dq);
    return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

```

list:          1 2 2 4 6 7 7 9
vector:        2 2 2 3 6 6 8 9
merge:         1 2 2 2 2 2 3 4 6 6 6 7 7 8 9 9
union:         1 2 2 2 3 4 6 6 7 7 8 9
intersection:  2 2 6 9
difference:    1 4 7 7
symmetric difference: 1 2 3 4 6 7 7 8

```

从结果中可以看出, 集合的并集并不是简单的合并, 而是去掉了相同的数据.

5. 堆算法

堆的主要用途是用来堆排序, 堆可以认为是一个完全二叉树, 但是通常是使用序列集合实现的, 具体关于堆的概念和结构请参考《算法导论》. 堆的特点是堆顶总是优先级(最大或最小或其他优先级)最高的元素, 向堆中插入或删除元素都可以在对数时间内完成.

libcstl 中的优先队列也是使用堆算法来实现的. libcstl 提供下面 5 个堆算法:

向堆中插入数据: `algo_push_heap()`, `algo_push_heap_if()`.

从堆中弹出数据: `algo_pop_heap()`, `algo_pop_heap_if()`.

将数据区间转换成一个堆: `algo_make_heap()`, `algo_make_heap_if()`

堆排序: `algo_sort_heap()`, `algo_sort_heap_if()`.

判断一个数据区间是不是堆: `algo_is_heap()`, `algo_is_heap_if()`

其中向堆中插入数据假设数据区间的最后一个数据是待插入的数据, 前面已经是一个堆, 从堆中弹出数据将堆顶(数据区间的第一个数据)弹出数据并放在区间的最后位置, 然后将数据区间前面的数据再转化成一个堆. 堆算法默认使用小于操作. 下面是有关堆算法的例子:

```

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{

```

```

vector_t t_v = create_vector(int);
int i = 0;

vector_init(&t_v);
for(i = 3; i <= 7; ++i)
{
    vector_push_back(&t_v, i);
}
for(i = 5; i <= 9; ++i)
{
    vector_push_back(&t_v, i);
}
for(i = 1; i <= 4; ++i)
{
    vector_push_back(&t_v, i);
}
printf("on entry:                ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

algo_make_heap(vector_begin(&t_v), vector_end(&t_v));
printf("after algo_make_heap(): ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

algo_pop_heap(vector_begin(&t_v), vector_end(&t_v));
vector_pop_back(&t_v);
printf("after algo_pop_heap(): ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

vector_push_back(&t_v, 17);
algo_push_heap(vector_begin(&t_v), vector_end(&t_v));
printf("after algo_push_heap(): ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

algo_sort_heap(vector_begin(&t_v), vector_end(&t_v));
printf("after algo_sort_heap(): ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

vector_destroy(&t_v);
return 0;

```

```

}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

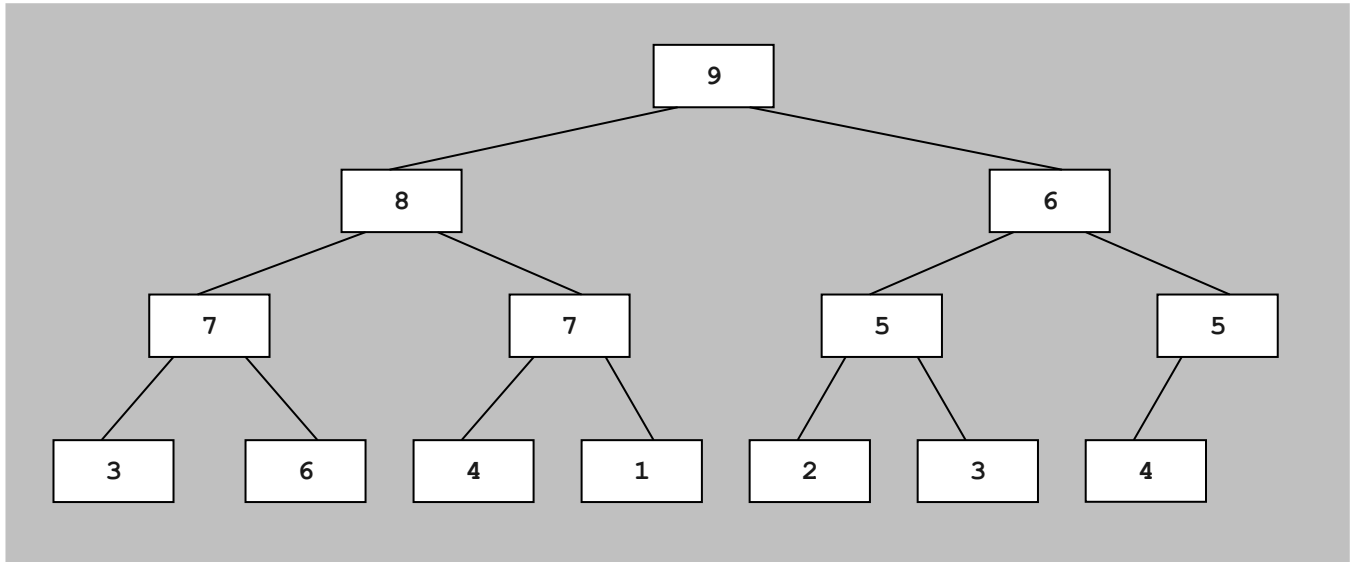
结果:

```

on entry:          3 4 5 6 7 5 6 7 8 9 1 2 3 4
after algo_make_heap(): 9 8 6 7 7 5 5 3 6 4 1 2 3 4
after algo_pop_heap():  8 7 6 7 4 5 5 3 6 4 1 2 3
after algo_push_heap(): 17 7 8 7 4 5 6 3 6 4 1 2 3 5
after algo_sort_heap(): 1 2 3 3 4 4 5 5 6 6 7 7 8 17

```

在使用算法 `algo_make_heap()` 后, 数据区间中的数据排序: 9 8 6 7 7 5 5 3 6 4 1 2 3 4 将这些数据转换成二叉树就可以看出每一个数据都小于等于它父节点的数据:



6. 最大数据和最小数据

最大数据和最小数据算法是在两个数据中或者一个数据区间中查找最大或者最小数据. 包括如下算法:

- 获得两个数据中的最小数据: `algo_min()`, `algo_min_if()`.
- 获得两个数据中的最大数据: `algo_max()`, `algo_max_if()`.
- 获得数据区间中的最小数据: `algo_min_element()`, `algo_min_element_if()`.
- 获得数据区间中的最大数据: `algo_max_element()`, `algo_max_element_if()`.

下面是有关算法的例子:

```

#include <stdio.h>
#include <stdlib.h>
#include <cstl/cvector.h>
#include <cstl/clist.h>
#include <cstl/calgorithm.h>

```



```

static void _print(const void* cpv_input, void* pv_output);
static void _absless(const void* cpv_first, const void* cpv_second, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(int);
    list_t t_l = create_list(int);
    int i = 0;
    iterator_t t_min, t_max;

    vector_init(&t_v);
    list_init(&t_l);
    for(i = -2; i <= 7; ++i)
    {
        vector_push_back(&t_v, i);
    }
    for(i = 0; i >= -9; --i)
    {
        list_push_back(&t_l, i);
    }
    printf("vector: ");
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");

    printf("list:  ");
    algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
    printf("\n");

    t_min = algo_min(vector_begin(&t_v), list_begin(&t_l));
    printf("minmum between two first: %d\n", *(int*)iterator_get_pointer(&t_min));
    t_max = algo_max(vector_begin(&t_v), list_begin(&t_l));
    printf("maxmum between two first: %d\n", *(int*)iterator_get_pointer(&t_max));

    t_min = algo_min_if(vector_begin(&t_v), list_begin(&t_l), _absless);
    printf("minmum of absolute value between two first: %d\n",
        *(int*)iterator_get_pointer(&t_min));
    t_max = algo_max_if(vector_begin(&t_v), list_begin(&t_l), _absless);
    printf("maxmum of absolute value between two first: %d\n",
        *(int*)iterator_get_pointer(&t_max));

    t_min = algo_min_element(vector_begin(&t_v), vector_end(&t_v));
    printf("minmum in vector: %d\n",
        *(int*)iterator_get_pointer(&t_min));

```

```

t_max = algo_max_element(vector_begin(&t_v), vector_end(&t_v));
printf("maxmum in vector: %d\n",
      *(int*)iterator_get_pointer(&t_max));

t_min = algo_min_element(list_begin(&t_l), list_end(&t_l));
printf("minmum in list: %d\n",
      *(int*)iterator_get_pointer(&t_min));
t_max = algo_max_element(list_begin(&t_l), list_end(&t_l));
printf("maxmum in list: %d\n",
      *(int*)iterator_get_pointer(&t_max));

t_min = algo_min_element_if(vector_begin(&t_v), vector_end(&t_v), _absless);
printf("minmum of absolute value in vector: %d\n",
      *(int*)iterator_get_pointer(&t_min));
t_max = algo_max_element_if(vector_begin(&t_v), vector_end(&t_v), _absless);
printf("maxmum of absolute value in vector: %d\n",
      *(int*)iterator_get_pointer(&t_max));

t_min = algo_min_element_if(list_begin(&t_l), list_end(&t_l), _absless);
printf("minmum of absolute value in list: %d\n",
      *(int*)iterator_get_pointer(&t_min));
t_max = algo_max_element_if(list_begin(&t_l), list_end(&t_l), _absless);
printf("maxmum of absolute value in list: %d\n",
      *(int*)iterator_get_pointer(&t_max));

vector_destroy(&t_v);
list_destroy(&t_l);
return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

static void _absless(const void* cpv_first, const void* cpv_second, void* pv_output)
{
    if(abs(*(int*)cpv_first) < abs(*(int*)cpv_second))
    {
        *(bool_t*)pv_output = true;
    }
    else
    {
        *(bool_t*)pv_output = false;
    }
}

```

```

}
结果:
vector: -2 -1 0 1 2 3 4 5 6 7
list:   0 -1 -2 -3 -4 -5 -6 -7 -8 -9
minmum between two first: -2
maxmum between two first: 0
minmum of absolute value between two first: 0
maxmum of absolute value between two first: -2
minmum in vector: -2
maxmum in vector: 7
minmum in list: -9
maxmum in list: 0
minmum of absolute value in vector: 0
maxmum of absolute value in vector: 7
minmum of absolute value in list: 0
maxmum of absolute value in list: -9

```

7. 按词典顺序比较

按词典顺序比较包含 4 个算法:

```

algo_lexicographical_compare(), algo_lexicographical_compare_if().
algo_lexicographical_compare_3way(), algo_lexicographical_compare_3way_if().

```

算法通常采用小于操, 但也可以接受自定义规则. 前两个算法返回 `bool_t` 类型的值, 表示比较是否成功. 后两个算法返回 3 种值, 类似于 `memcpy()` 函数的返回值, `>0 ==0, <0`. 以 `algo_lexicographical_compare_3way()` 如果第一个区间的数据小于第二个区间的数据返回`<0` 的值, 如果相等则返回`==0` 的值, 大于返回`>0` 的值.

按字典顺序比较的意思是, 如果第一个区间只要遇到一个小于第二个区间的数据就表示小于, 如果遇到大于的数据就表示大于, 如果等于还要比较两个区间的数据个数, 如果第一个区间中数据少则小于, 多则大于, 否则等于. 下面是算法的例子:

```

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/clist.h>
#include <cstl/calgorithm.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(int);
    list_t t_l = create_list(int);

    vector_init(&t_v);
    list_init(&t_l);

```

```

vector_push_back(&t_v, 3);
vector_push_back(&t_v, 2);
vector_push_back(&t_v, 8);
vector_push_back(&t_v, 7);

/* case 1 */
list_push_back(&t_l, 3);
list_push_back(&t_l, 3);
list_push_back(&t_l, 3);
list_push_back(&t_l, 3);
printf("vector: ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");
printf("list:   ");
algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
printf("\n");
if(algo_lexicographical_compare(
    vector_begin(&t_v), vector_end(&t_v),
    list_begin(&t_l), list_end(&t_l)))
{
    printf("algo_lexicographical_compare(): true\n");
}
else
{
    printf("algo_lexicographical_compare(): false\n");
}
printf("algo_lexicographical_compare_3way(): %d\n",
    algo_lexicographical_compare_3way(
        vector_begin(&t_v), vector_end(&t_v),
        list_begin(&t_l), list_end(&t_l)));
/* case 2 */
list_clear(&t_l);
list_push_back(&t_l, 3);
list_push_back(&t_l, 2);
list_push_back(&t_l, 8);
list_push_back(&t_l, 7);
list_push_back(&t_l, 7);
printf("vector: ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");
printf("list:   ");
algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
printf("\n");
if(algo_lexicographical_compare(

```

```

        vector_begin(&t_v), vector_end(&t_v),
        list_begin(&t_l), list_end(&t_l)))
{
    printf("algo_lexicographical_compare(): true\n");
}
else
{
    printf("algo_lexicographical_compare(): false\n");
}
printf("algo_lexicographical_compare_3way(): %d\n",
    algo_lexicographical_compare_3way(
        vector_begin(&t_v), vector_end(&t_v),
        list_begin(&t_l), list_end(&t_l)));
/* case 3 */
list_pop_back(&t_l);
printf("vector: ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");
printf("list:   ");
algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
printf("\n");
if(algo_lexicographical_compare(
    vector_begin(&t_v), vector_end(&t_v),
    list_begin(&t_l), list_end(&t_l)))
{
    printf("algo_lexicographical_compare(): true\n");
}
else
{
    printf("algo_lexicographical_compare(): false\n");
}
printf("algo_lexicographical_compare_3way(): %d\n",
    algo_lexicographical_compare_3way(
        vector_begin(&t_v), vector_end(&t_v),
        list_begin(&t_l), list_end(&t_l)));
/* case 4 */
list_pop_back(&t_l);
printf("vector: ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");
printf("list:   ");
algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
printf("\n");
if(algo_lexicographical_compare(

```

```

        vector_begin(&t_v), vector_end(&t_v),
        list_begin(&t_l), list_end(&t_l)))
{
    printf("algo_lexicographical_compare(): true\n");
}
else
{
    printf("algo_lexicographical_compare(): false\n");
}
printf("algo_lexicographical_compare_3way(): %d\n",
    algo_lexicographical_compare_3way(
        vector_begin(&t_v), vector_end(&t_v),
        list_begin(&t_l), list_end(&t_l)));
/* case 5 */
list_push_back(&t_l, 5);
printf("vector: ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");
printf("list:   ");
algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
printf("\n");
if(algo_lexicographical_compare(
    vector_begin(&t_v), vector_end(&t_v),
    list_begin(&t_l), list_end(&t_l)))
{
    printf("algo_lexicographical_compare(): true\n");
}
else
{
    printf("algo_lexicographical_compare(): false\n");
}
printf("algo_lexicographical_compare_3way(): %d\n",
    algo_lexicographical_compare_3way(
        vector_begin(&t_v), vector_end(&t_v),
        list_begin(&t_l), list_end(&t_l)));

vector_destroy(&t_v);
list_destroy(&t_l);
return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);

```

```
}
```

结果:

```
vector: 3 2 8 7
list:   3 3 3 3
algo_lexicographical_compare(): true
algo_lexicographical_compare_3way(): -1
vector: 3 2 8 7
list:   3 2 8 7 7
algo_lexicographical_compare(): true
algo_lexicographical_compare_3way(): -1
vector: 3 2 8 7
list:   3 2 8 7
algo_lexicographical_compare(): false
algo_lexicographical_compare_3way(): 0
vector: 3 2 8 7
list:   3 2 8
algo_lexicographical_compare(): false
algo_lexicographical_compare_3way(): 1
vector: 3 2 8 7
list:   3 2 8 5
algo_lexicographical_compare(): false
algo_lexicographical_compare_3way(): 1
```

8. 排列组合

libcstl 提供了一组算法来获得下一组合或者上一组合. 什么是排列组合呢, 考虑由三个字符组成的集合 {a, b, c}, 这个集合就有六种排列组合形式: abc, acb, bac, bca, cab, cba. 这些组合也是由小到大排列的. 下一个组合就是针对与当前数据的排序组合形式的, 假设当前的排列组合形式为 acb, 那么下一个组合就是 bac, 同理前一个组合形式就是 abc. 对于开头和末尾的组合形式: abc 没有前一个组合, 同理 cba 没有下一个组合.

组合算法:

```
algo_next_permutation(), algo_next_permutation_if().
algo_prev_permutation(), algo_prev_permutation_if().
```

下面四关于排列组合算法的例子:

```
#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>
```

```
static void _print(const void* cpv_input, void* pv_output);
```

```
int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(int);
```

```

vector_init(&t_v);
vector_push_back(&t_v, 1);
vector_push_back(&t_v, 2);
vector_push_back(&t_v, 3);

printf("on entry: ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

while(algo_next_permutation(vector_begin(&t_v), vector_end(&t_v)))
{
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");
}
printf("afterward: ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

while(algo_prev_permutation(vector_begin(&t_v), vector_end(&t_v)))
{
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");
}
printf("now: ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");
while(algo_prev_permutation(vector_begin(&t_v), vector_end(&t_v)))
{
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");
}
printf("afterward: ");
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

vector_destroy(&t_v);
return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```



```
结果:
on entry: 1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
afterward: 1 2 3
now: 3 2 1
3 1 2
2 3 1
2 1 3
1 3 2
1 2 3
afterward: 3 2 1
```

第五节 算术算法

算术算法是将数据区间进行算术运算操作. 要使用算术算法必须包含头文件<cstl/cnumeric.h>

```
#include <cstl/cnumeric.h>
```

algo_iota()	将数据区间的数据从指定的值开始逐一赋值, 每次加一.
algo_accumulate()	将数据区间中的数据求和.
algo_accumulate_if()	将数据区间中的数据依次应用于特定操作.
algo_inner_product()	求两个数据区间的内积.
algo_inner_product_if()	使用特定操作求两个数据区间的内积.
algo_partial_sum()	计算局部总和.
algo_partial_sum_if()	使用特定操作计算局部总和.
algo_adjacent_difference()	计算相邻数据的差.
algo_adjacent_difference_if()	使用特定操作计算相邻数据的差.
algo_power()	计算数据的n次幂.
algo_power_if()	使用特殊操作计算数据的n次幂.

1. 依次赋值

```
algo_iota()

这个算法将数据区间中的数据依次赋新值, 每次递增. 具体赋值过程是
*first = value, *(first + 1) = value + 1, *(first + 2) = value + 2, ...
下面是这个算法的使用实例:
#include <stdio.h>
#include <cstl/cvector.h>
```

```

#include <cstl/calgorithm.h>
#include <cstl/cnumeric.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(int);

    vector_init_n(&t_v, 10);
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");

    algo_iota(vector_begin(&t_v), vector_end(&t_v), 100);
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");

    vector_destroy(&t_v);
    return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

0 0 0 0 0 0 0 0 0 0

100 101 102 103 104 105 106 107 108 109

2. 求和算法

`algo_accumulate()`, `algo_accumulate_if()`.

求和算法的执行过程:

`initvalue + a1 + a2 + a3 + ...`

或者

`initvalue op a1 op a2 op a3 op ...`

关于算法的例子:

```

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>
#include <cstl/cnumeric.h>

```

```

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(int);
    int i = 0;
    int n_result;

    vector_init(&t_v);
    for(i = 1; i <= 9; ++i)
    {
        vector_push_back(&t_v, i);
    }
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");

    algo_accumulate(vector_begin(&t_v), vector_end(&t_v), 0, &n_result);
    printf("sum: %d\n", n_result);
    algo_accumulate(vector_begin(&t_v), vector_end(&t_v), -100, &n_result);
    printf("sum: %d\n", n_result);

    algo_accumulate_if(
        vector_begin(&t_v), vector_end(&t_v), 1, fun_multiplies_int, &n_result);
    printf("product: %d\n", n_result);
    algo_accumulate_if(
        vector_begin(&t_v), vector_end(&t_v), 0, fun_multiplies_int, &n_result);
    printf("product: %d\n", n_result);

    vector_destroy(&t_v);
    return 0;
}

```

```

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

1 2 3 4 5 6 7 8 9

sum: 45

sum: -55

product: 362880

product: 0

3. 计算内积

`algo_inner_product()`, `algo_inner_product_if()`.

内积的计算过程是:

`initvalue + (a1 * b2) + (a2 * b2) + (a3 * b3) + ...`

或者

`initvalue op1 (a1 op2 b1) op1 (a2 op2 b2) op1 (a3 op2 b3) op1 ...`

这个算法的例子:

```
#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/clist.h>
#include <cstl/calgorithm.h>
#include <cstl/numeric.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(int);
    list_t t_l = create_list(int);
    int i = 0;
    int n_result = 0;

    vector_init(&t_v);
    list_init(&t_l);
    for(i = 1; i <= 6; ++i)
    {
        vector_push_back(&t_v, i);
        list_push_back(&t_l, i);
    }
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");
    algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
    printf("\n");

    algo_inner_product(
        vector_begin(&t_v), vector_end(&t_v), list_begin(&t_l), 0, &n_result);
    printf("inner product: %d\n", n_result);
    list_reverse(&t_l);
    algo_inner_product(
        vector_begin(&t_v), vector_end(&t_v), list_begin(&t_l), 0, &n_result);
    printf("inner reverse product: %d\n", n_result);
}
```

```

    algo_inner_product_if(
        vector_begin(&t_v), vector_end(&t_v), vector_begin(&t_v),
        1, fun_multiplies_int, fun_plus_int, &n_result);
    printf("product of sums: %d\n", n_result);

    vector_destroy(&t_v);
    list_destroy(&t_l);
    return 0;
}

```

```

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

1 2 3 4 5 6

1 2 3 4 5 6

inner product: 91

inner reverse product: 56

product of sums: 46080

4. 局部总和

`algo_partial_sum()`, `algo_partial_sum_if()`

这两个算法用来计算数据区间的局部总和, 具体过程如下:

$b_1 = a_1$, $b_2 = a_1 + a_2$, $b_3 = a_1 + a_2 + a_3$, ...

或者

$b_1 = a_1$, $b_2 = a_1 \text{ op } a_2$, $b_3 = a_1 \text{ op } a_2 \text{ op } a_3$, ...

算法的例子:

```

#include <stdio.h>
#include <cstdlib/cvector.h>
#include <cstdlib/clist.h>
#include <cstdlib/calgorithm.h>
#include <cstdlib/cnumeric.h>

```

```

static void _print(const void* cpv_input, void* pv_output);

```

```

int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(int);
    list_t t_l = create_list(int);
    int i = 0;

```

```

vector_init(&t_v);
for(i = 1; i <= 6; ++i)
{
    vector_push_back(&t_v, i);
}
list_init_n(&t_l, vector_size(&t_v));
algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
printf("\n");

algo_partial_sum(vector_begin(&t_v), vector_end(&t_v), list_begin(&t_l));
algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
printf("\n");

algo_partial_sum_if(
    vector_begin(&t_v), vector_end(&t_v), list_begin(&t_l),
    fun_multiplies_int);
algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
printf("\n");

vector_destroy(&t_v);
list_destroy(&t_l);
return 0;
}

static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}

```

结果:

1 2 3 4 5 6

1 3 6 10 15 21

1 2 6 24 120 720

5. 相邻数据的差

algo_adjacent_difference(), algo_adjacent_difference_if().

这两个算法用来计算相邻数据的差, 它们是局部求和算法的逆运算, 具体过程:

$b_1 = a_1$, $b_2 = a_2 - a_1$, $b_3 = a_3 - a_2$, ...

或者

$b_1 = a_1$, $b_2 = a_2 \text{ op } a_1$, $b_3 = a_3 \text{ op } a_2$, ...

下面是关于这两个算法的例子:

```

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/clist.h>
#include <cstl/calgorithm.h>
#include <cstl/cnumeric.h>

static void _print(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t t_v = create_vector(int);
    list_t t_l = create_list(int);
    int i = 0;

    vector_init(&t_v);
    for(i = 1; i <= 6; ++i)
    {
        vector_push_back(&t_v, i);
    }
    list_init_n(&t_l, vector_size(&t_v));
    algo_for_each(vector_begin(&t_v), vector_end(&t_v), _print);
    printf("\n");

    algo_adjacent_difference(
        vector_begin(&t_v), vector_end(&t_v), list_begin(&t_l));
    algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
    printf("\n");

    algo_adjacent_difference_if(
        vector_begin(&t_v), vector_end(&t_v), list_begin(&t_l), fun_plus_int);
    algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
    printf("\n");

    algo_adjacent_difference_if(
        vector_begin(&t_v), vector_end(&t_v), list_begin(&t_l),
        fun_multiplies_int);
    algo_for_each(list_begin(&t_l), list_end(&t_l), _print);
    printf("\n");

    vector_destroy(&t_v);
    list_destroy(&t_l);
    return 0;
}

```

```
static void _print(const void* cpv_input, void* pv_output)
{
    printf("%d ", *(int*)cpv_input);
}
```

结果:

```
1 2 3 4 5 6
1 1 1 1 1 1
1 3 5 7 9 11
1 2 6 12 20 30
```

6. 幂运算

`algo_power()`, `algo_power_if()`.

这两个算法计算数据的 n 次幂, 下面是运用实例:

```
#include <stdio.h>
#include <cstl/cdeque.h>
#include <cstl/numeric.h>

int main(int argc, char* argv[])
{
    deque_t t_d = create_deque(int);
    int n_output = 0;
    deque_init(&t_d);
    deque_push_back(&t_d, 2);

    algo_power(deque_begin(&t_d), 3, &n_output);
    printf("output:%d\n", n_output);
    algo_power_if(deque_begin(&t_d), 3, fun_plus_int, &n_output);
    printf("output:%d\n", n_output);

    deque_destroy(&t_d);
    return 0;
}
```

结果:

```
output:8
output:6
```


第八章 libcstl 容器适配器

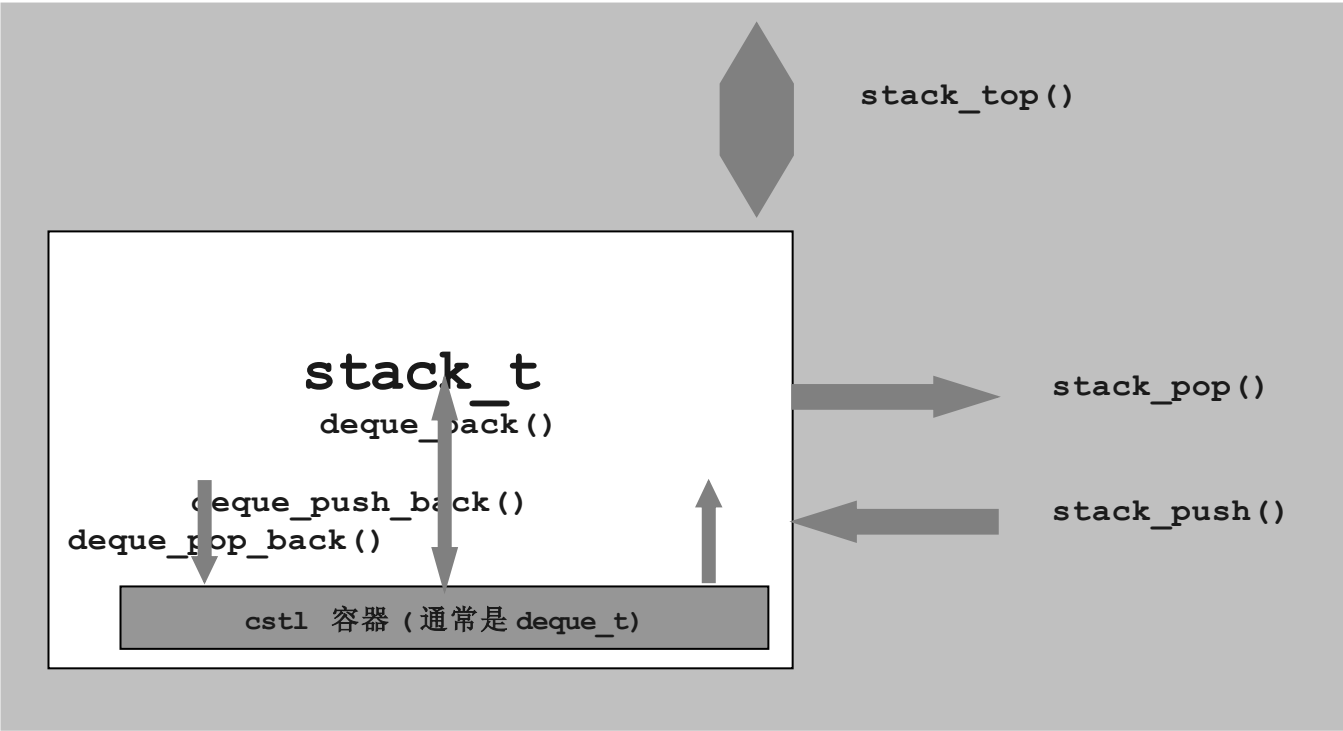
libcstl 除了提供第五章中描述的容器外, 还提供了一些为了满足特定需要接口简单的类型, 这些类型都是使用基本容器实现的, 所以这种类型叫做容器适配器. 容器适配器包括:

```
stack_t, queue_t, priority_queue_t.
```

第一节 stack_t

stack_t 是堆栈类型, 实现后进先出 (LIFO) 规则. 要使用 stack_t 必须包含头文件 `<cstl/cstack.h>`
`#include <cstl/cstack.h>`

stack_t 是使用普通的容器实现的, 它只不过是在普通容器上面又包装了一层, 通常采用 deque_t 来实现, 但是通过修改编译选项可以改变底层实现, 具体参考第二章.



stack_t 的操作函数很简单最主要的就是三个数据操作函数:

- `stack_push()`: 将数据压栈.
- `stack_top()`: 获得栈顶数据.
- `stack_pop()`: 数据出栈.

1. `stack_t` 的操作

创建, 初始化和销毁操作:

<code>create_stack()</code>	创建一个指定类型的 <code>stack_t</code> .
<code>stack_init()</code>	初始化一个空的 <code>stack_t</code> .
<code>stack_init_copy()</code>	使用另一个 <code>stack_t</code> 初始化 <code>stack_t</code> .
<code>stack_destroy()</code>	销毁 <code>stack_t</code> .

非质变操作:

<code>stack_size()</code>	获得 <code>stack_t</code> 中数据的数目.
<code>stack_empty()</code>	判断 <code>stack_t</code> 是否为空.
<code>stack_equal()</code>	判断两个 <code>stack_t</code> 是否相等.
<code>stack_not_equal()</code>	判断两个 <code>stack_t</code> 是否不等.
<code>stack_less()</code>	判断第一个 <code>stack_t</code> 是否小于第二个 <code>stack_t</code> .
<code>stack_less_equal()</code>	判断第一个 <code>stack_t</code> 是否小于等于第二个 <code>stack_t</code> .
<code>stack_great()</code>	判断第一个 <code>stack_t</code> 是否大于第二个 <code>stack_t</code> .
<code>stack_great_equal()</code>	判断第一个 <code>stack_t</code> 是否大于等于第二个 <code>stack_t</code> .

赋值:

<code>stack_assign()</code>	使用另一个 <code>stack_t</code> 为当前 <code>stack_t</code> 赋值.
-----------------------------	---

数据操作:

<code>stack_push()</code>	将数据压栈.
<code>stack_top()</code>	获得栈顶数据.
<code>stack_pop()</code>	数据出栈.

`stack_t` 不支持迭代器, 所以没有迭代器相关的操作函数.

2. `stack_t` 的使用实例

```
#include <stdio.h>
#include <cstl/cstack.h>

int main(int argc, char* argv[])
{
    stack_t t_st = create_stack(int);

    stack_init(&t_st);
    stack_push(&t_st, 1);
    stack_push(&t_st, 2);
    stack_push(&t_st, 3);
```

```

printf("%d ", *(int*)stack_top(&t_st));
stack_pop(&t_st);
printf("%d ", *(int*)stack_top(&t_st));
stack_pop(&t_st);

*(int*)stack_top(&t_st) = 77;
stack_push(&t_st, 4);

while(!stack_empty(&t_st))
{
    printf("%d ", *(int*)stack_top(&t_st));
    stack_pop(&t_st);
}
printf("\n");

stack_destroy(&t_st);
return 0;
}

```

结果:

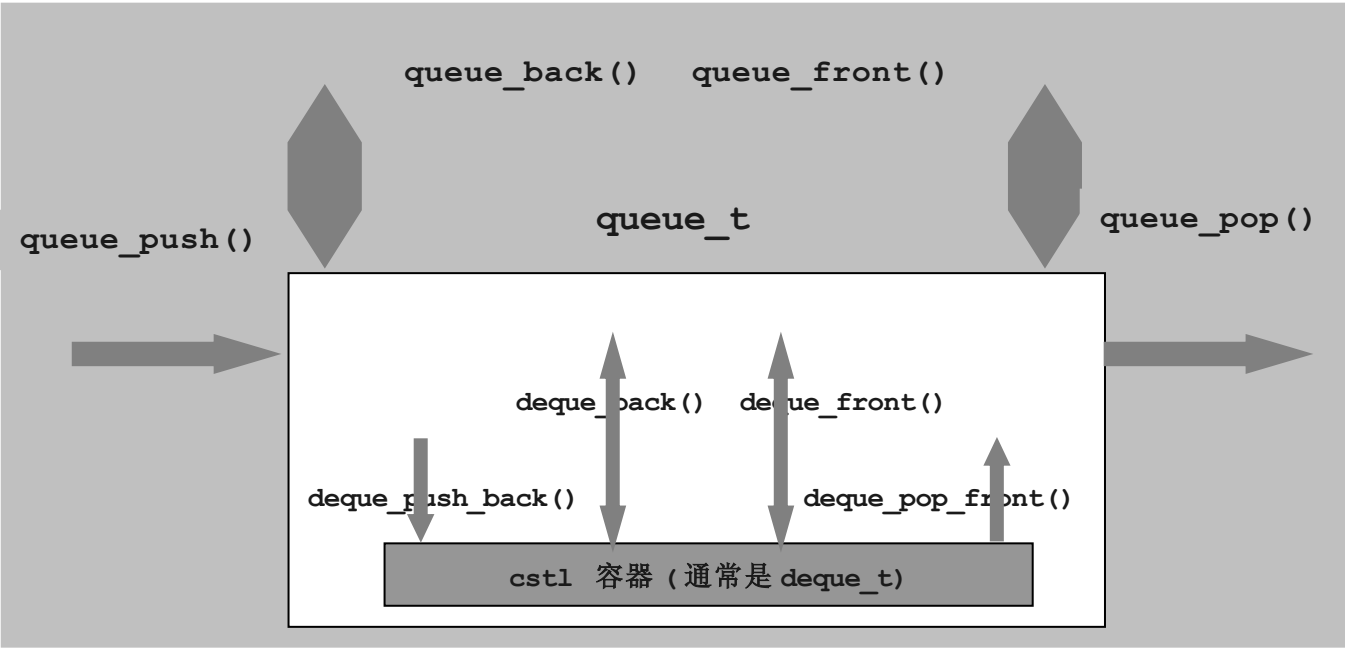
3 2 4 77

第二节 queue_t

queue_t 是队列类型, 实现先进先出 (FIFO) 规则. 要使用 queue_t 必须包含头文件 **<cstl/cqueue.h>**

```
#include <cstl/cqueue.h>
```

queue_t 是使用普通的容器实现的, 它只不过是在普通容器上面又包装了一层, 通常采用 deque_t 来实现, 但是通过修改编译选项可以改变底层实现, 具体参考第二章.



`queue_t` 的操作函数很简单最主要的就是四个数据操作函数：

- `queue_push()` : 向队列中插入数据.
- `queue_front()` : 获得队列开头数据.
- `queue_back()` : 获得队列末尾的数据.
- `queue_pop()` : 从队列中弹出数据.

1. queue_t 的操作

创建, 初始化和销毁操作:

<code>create_queue()</code>	创建一个指定类型的 <code>queue_t</code> .
<code>queue_init()</code>	初始化一个空的 <code>queue_t</code> .
<code>queue_init_copy()</code>	使用另一个 <code>queue_t</code> 初始化 <code>queue_t</code> .
<code>queue_destroy()</code>	销毁 <code>queue_t</code> .

非质变操作:

<code>queue_size()</code>	获得 <code>queue_t</code> 中数据的数目.
<code>queue_empty()</code>	判断 <code>queue_t</code> 是否为空.
<code>queue_equal()</code>	判断两个 <code>queue_t</code> 是否相等.
<code>queue_not_equal()</code>	判断两个 <code>queue_t</code> 是否不等.
<code>queue_less()</code>	判断第一个 <code>queue_t</code> 是否小于第二个 <code>queue_t</code> .
<code>queue_less_equal()</code>	判断第一个 <code>queue_t</code> 是否小于等于第二个 <code>queue_t</code> .
<code>queue_great()</code>	判断第一个 <code>queue_t</code> 是否大于第二个 <code>queue_t</code> .
<code>queue_great_equal()</code>	判断第一个 <code>queue_t</code> 是否大于等于第二个 <code>queue_t</code> .

赋值:

<code>queue_assign()</code>	使用另一个 <code>queue_t</code> 为当前 <code>queue_t</code> 赋值.
-----------------------------	---

数据操作：

<code>queue_push()</code>	向队列中插入数据。
<code>queue_front()</code>	获得队列开头数据。
<code>queue_back()</code>	获得队列末尾的数据。
<code>queue_pop()</code>	从队列中弹出数据。

`queue_t` 不支持迭代器, 所以没有迭代器相关的操作函数。

2. `queue_t` 的使用实例

```
#include <stdio.h>
#include <cstl/cqueue.h>

int main(int argc, char* argv[])
{
    queue_t t_q = create_queue(int);

    queue_init(&t_q);
    printf("queue - size:%u, empty:%d\n", queue_size(&t_q), queue_empty(&t_q));

    queue_push(&t_q, 100);
    queue_push(&t_q, 200);
    queue_push(&t_q, 300);
    printf("queue - size:%u, empty:%d\n", queue_size(&t_q), queue_empty(&t_q));
    printf("front:%d, back:%d\n",
        *(int*)queue_front(&t_q), *(int*)queue_back(&t_q));
    *(int*)queue_front(&t_q) = 5000;
    *(int*)queue_back(&t_q) = -10000;
    printf("queue - size:%u, empty:%d\n", queue_size(&t_q), queue_empty(&t_q));
    printf("front:%d, back:%d\n",
        *(int*)queue_front(&t_q), *(int*)queue_back(&t_q));

    while(!queue_empty(&t_q))
    {
        printf("%d ", *(int*)queue_front(&t_q));
        queue_pop(&t_q);
    }
    printf("\n");

    queue_destroy(&t_q);
    return 0;
}
```

```
结果:
queue - size:0, empty:1
queue - size:3, empty:0
front:100, back:300
queue - size:3, empty:0
front:5000, back:-10000
5000 200 -10000
```

第三节 priority_queue_t

priority_queue_t 是优先队列类型. 要使用 priority_queue_t 必须包含头文件 `#include <cstdlib/cqueue.h>`

priority_queue_t 是使用普通的容器实现的, 它只不过是在普通容器上面又包装了一层, 通常采用 `priority_deque_t` 来实现.

priority_queue_t 是带有优先级的队列, 每次从队列中取出的数据总是优先级最高的数据.

priority_queue_t 的操作函数很简单最主要的就是三个数据操作函数:

- `priority_queue_push()`: 将数据插入优先队列.
- `priority_queue_top()`: 获得优先级最高的数据.
- `priority_queue_pop()`: 弹出优先级最高的数据.

既然使用 `priority_queue_top()` 操作函数获得的总是优先级最高的数据, 那么说明优先队列内部是已经对数据进行某种排序的, 所以不能通过 `priority_queue_top()` 来修改数据的值, 否则就会影响队列中的优先级规则.

1. priority_queue_t 的操作

创建, 初始化和销毁操作:	
<code>create_priority_queue()</code>	创建一个指定类型的 <code>priority_queue_t</code> .
<code>priority_queue_init()</code>	初始化一个空的 <code>priority_queue_t</code> .
<code>priority_queue_init_op()</code>	使用指定的优先级规则初始化一个空的 <code>priority_queue_t</code> .
<code>priority_queue_init_copy()</code>	使用另一个 <code>priority_queue_t</code> 初始化 <code>priority_queue_t</code> .
<code>priority_queue_init_copy_range()</code>	使用一个数据区间初始化 <code>priority_queue_t</code> .
<code>priority_queue_init_copy_range_op()</code>	使用一个数据区间和指定的优先级规则初始化 <code>priority_queue_t</code> .
<code>priority_queue_destroy()</code>	销毁 <code>priority_queue_t</code> .

非质变操作:	
<code>priority_queue_size()</code>	获得 <code>priority_queue_t</code> 中数据的数目.
<code>priority_queue_empty()</code>	判断 <code>priority_queue_t</code> 是否为空.

赋值:

<code>priority_queue_assign()</code>	使用另一个 <code>priority_queue_t</code> 为当前 <code>priority_queue_t</code> 赋值。
--------------------------------------	---

数据操作：

<code>priority_queue_push()</code>	将数据插入优先队列。
<code>priority_queue_top()</code>	获得优先级最高的数据。
<code>priority_queue_pop()</code>	弹出优先级最高的数据。

`priority_queue_t` 不支持迭代器, 所以没有迭代器相关的操作函数。

2. `priority_queue_t` 的使用实例

```
#include <stdio.h>
#include <cstl/cqueue.h>

int main(int argc, char* argv[])
{
    priority_queue_t t_pq = create_priority_queue(double);

    priority_queue_init(&t_pq);
    priority_queue_push(&t_pq, 66.6);
    priority_queue_push(&t_pq, 22.2);
    priority_queue_push(&t_pq, 44.4);

    printf("%f ", *(double*)priority_queue_top(&t_pq));
    priority_queue_pop(&t_pq);
    printf("%f ", *(double*)priority_queue_top(&t_pq));
    priority_queue_pop(&t_pq);
    printf("\n");

    priority_queue_push(&t_pq, 11.1);
    priority_queue_push(&t_pq, 55.5);
    priority_queue_push(&t_pq, 33.3);
    priority_queue_pop(&t_pq);

    while(!priority_queue_empty(&t_pq))
    {
        printf("%f ", *(double*)priority_queue_top(&t_pq));
        priority_queue_pop(&t_pq);
    }
    printf("\n");

    priority_queue_destroy(&t_pq);
}
```

```
    return 0;
}
```

结果:

```
66.600000 44.400000
33.300000 22.200000 11.100000
```

第九章 libcstl 字符串

这里说的 libcstl 字符串并不是 C 语言中的 `char*` 或 `const char*` (统称为 `c-str`), 而是 libcstl 定义的 `string_t` 类型. `string_t` 类型可以像 `c-str` 一样拷贝, 赋值, 比较而不必考虑是否有足够的内存来保存字符串, 会不会越界等等. 因为 `string_t` 可以动态增长, 并且易于使用, 你可很方便的插入删除字符或子串, 方便的替换等等.

第一节 两个 `string_t` 的使用实例

1. 第一个例子: 提取文件名模版

这个例子使用命令行参数获得文件名模版, 例如输入:

```
./test prog.dat mydir hello. oops.tmp end.dat
```

输出结果如下:

```
prog.dat => prog.tmp
mydir => mydir.tmp
hello. => hello.tmp
oops.tmp => oops.xxx
end.dat => end.tmp
```

通常文件的后缀替换为 .tmp, 如果是模版文件” 后缀为 .tmp” 替换为 .xxx

程序如下:

```
#include <stdio.h>
#include <cstl/cstring.h>

int main(int argc, char* argv[])
{
    string_t t_basename, t_filename, t_extname, t_tmpname, t_sffuix;
    int i = 0;
    size_t t_pos = 0;

    string_init_cstr(&t_sffuix, "tmp");
    string_init(&t_basename);
```



```

string_init(&t_filename);
string_init(&t_extname);
string_init(&t_tmpname);

for(i = 1; i < argc; ++i)
{
    string_assign_cstr(&t_filename, argv[i]);

    t_pos = string_find_char(&t_filename, '.', 0);
    if(t_pos == NPOS)
    {
        string_assign(&t_tmpname, &t_filename);
        string_connect_char(&t_tmpname, '.');
        string_connect(&t_tmpname, &t_sffuix);
    }
    else
    {
        string_assign_substring(&t_basename, &t_filename, 0, t_pos);
        string_assign_substring(&t_extname, &t_filename, t_pos + 1, NPOS);

        string_assign(&t_tmpname, &t_filename);
        if(string_empty(&t_extname))
        {
            string_connect(&t_tmpname, &t_sffuix);
        }
        else if(string_equal(&t_extname, &t_sffuix))
        {
            string_replace_cstr(&t_tmpname, t_pos + 1, NPOS, "xxx");
        }
        else
        {
            string_replace(&t_tmpname, t_pos + 1, NPOS, &t_sffuix);
        }
    }

    printf("%s => %s\n",
           string_c_str(&t_filename), string_c_str(&t_tmpname));
}

string_destroy(&t_sffuix);
string_destroy(&t_basename);
string_destroy(&t_filename);
string_destroy(&t_extname);
string_destroy(&t_tmpname);

```

```
    return 0;
}
```

首先是包含头文件

```
#include <cstl/cstring.h>
```

要使用 **string_t** 类型就必须包含上面的头文件.

接下来是创建 **string_t** 类型的变量:

```
string_t t_basename, t_filename, t_extname, t_tmpname, t_sffuix;
```

创建 **string_t** 类型的变量不需要使用 **create_string()** 这样的函数, 因为 **string_t** 类型中保存的就是 **char** 类型, 所以不用指定数据类型, 因此 **libcstl** 也没有提供 **create_string()** 操作函数.

下面是初始化 **string_t** 类型的变量:

```
string_init_cstr(&t_sffuix, "tmp");
string_init(&t_basename);
string_init(&t_filename);
string_init(&t_extname);
string_init(&t_tmpname);
```

你可以直接使用字符串常量直接对 **string_t** 类型的变量进行初始化如:

```
string_init_cstr(&t_sffuix, "tmp");
```

也可以将 **string_t** 类型初始化为一个空字符串. 如其余 4 个 **string_t** 类型的变量.

为 **string_t** 类型变量赋值:

```
string_assign_cstr(&t_filename, argv[i]);
```

直接使用来自于命令行的参数的字符串为 **string_t** 类型的变量赋值.

```
string_assign(&t_tmpname, &t_filename);
```

使用另一个 **string_t** 类型赋值.

```
string_assign_substring(&t_basename, &t_filename, 0, t_pos);
```

使用 **string_t** 的子串赋值.

在 **string_t** 类型中查找字符:

```
t_pos = string_find_char(&t_filename, '.', 0);
```

这个操作函数返回的是 **size_t** 类型的值表示字符在 **string_t** 类型中的位置. 如果没有找到则返回 **NPOS** 表示查找失败:

```
if(t_pos == NPOS)
```

连接 **string_t**:

```
string_connect_char(&t_tmpname, '.');
string_connect(&t_tmpname, &t_sffuix);
```

可以将单个字符连接到 **string_t** 类型, 或者将两个 **string_t** 类型连接在一起.

判断 **string_t** 类型是否为空:

```
if(string_empty(&t_extname))
```

判断两个 **string_t** 是否相等:

```
if(string_equal(&t_extname, &t_sffuix))
```

对 string_t 类型进行替换:

```
string_replace_cstr(&t_tmpname, t_pos + 1, NPOS, "xxx");
```

```
string_replace(&t_tmpname, t_pos + 1, NPOS, &t_sffuix);
```

可以使用字符串常量进行替换也可以使用另一个 string_t 类型进行替换.

获得C字符串:

```
string_c_str(&t_filename);
```

这样可以获得C字符串.

2. 第二个例子:提取单词并逆序打印

```
#include <stdio.h>
```

```
#include <cstl/cstring.h>
```

```
int main(int argc, char* argv[])
```

```
{
```

```
    string_t t_delims, t_str;
```

```
    string_init_cstr(&t_delims, " \\t,.:");
```

```
    string_init(&t_str);
```

```
    while(string_getline(&t_str, stdin))
```

```
    {
```

```
        size_t t_beginx, t_endinx;
```

```
        int i = 0;
```

```
        t_beginx = string_find_first_not_of(&t_str, &t_delims, 0);
```

```
        while(t_beginx != NPOS)
```

```
        {
```

```
            t_endinx = string_find_first_of(&t_str, &t_delims, t_beginx);
```

```
            if(t_endinx == NPOS)
```

```
            {
```

```
                t_endinx = string_length(&t_str);
```

```
            }
```

```
            for(i = t_endinx - 1; i >= (int)t_beginx; --i)
```

```
            {
```

```
                printf("%c", *string_at(&t_str, i));
```

```
            }
```

```

        printf(" ");

        t_beginx = string_find_first_not_of(&t_str, &t_delims, t_endinx);
    }
    printf("\n");
}

string_destroy(&t_delims);

return 0;
}

```

在程序的开始,所有自定义的分隔符都定义在 t_delims 中

```
string_init_cstr(&t_delims, " \t,.:");
```

换行符当然也是分隔符,但是没有必要特殊处理换行符,因为程序是按行读取数据的.

```
while(string_getline(&t_str, stdin))
```

这个函数从指定的流中一行一行的读取数据.接下来使用分隔符来区分单词,首先找到单词的开头.

```
t_beginx = string_find_first_not_of(&t_str, &t_delims, 0);
```

这是从一行的开始位置查找第一个不是分隔符的字符即为第一个单词的开头.接下来检查查找的结果:

```
while(t_beginx != NPOS)
```

NPOS 表示查找失败,如果查找没有失败我们通过找到下一个分隔符字符来确定一个单词的末尾.

```
t_endinx = string_find_first_of(&t_str, &t_delims, t_beginx);
```

如果没有找到就使用这一行的行尾作为单词的结尾.

```
if(t_endinx == NPOS)
```

```
{
    t_endinx = string_length(&t_str);
}
```

确定了单词的开始和结尾然后逆序打印单词.

```
for(i = t_endinx - 1; i >= (int)t_beginx; --i)
```

```
{
    printf("%c", *string_at(&t_str, i));
}
```

```
printf(" ");
```

然后重复上面的步骤.

如果输入的语句为

```
pots & pans
```

```
I saw a reed
```

数据结果为

```
stop & snap
```

```
I was a deer
```

第二节 string_t 类型的描述

1. string_t 的能力

string_t 类型是一个序列容器, 专门用来存储 char 类型, 它包含了所有的序列容器的操作, 同时它有包含了很多针对字符串的操作函数.

要使用 string_t 类型必须包含头文件<cstl/cstring.h>
#include <cstl/cstring.h>

2. string_t 的操作概览

string_t 是序列容器同时它保存的是 char 类型的数据, 所以它还具有很多针对字符串的特殊操作. 因此 string_t 的操作函数即支持迭代器, 同时又支持字符串. 下面列出了 string_t 操作函数参数的典型形式.

参数形式	参数含义	典型后缀
const string_t* cpt_string	整个 string_t 类型.	--
const string_t* cpt_string, size_t t_pos, size_t t_len	cpt_string 中从 t_pos 开始的最多 t_len 个字符.	substring
const char* s_cstr	整个 C 字符串.	cstr
const char* s_cstr, size_t t_len	C 字符串中的前 t_len 个字符.	substr
size_t t_count, char c_char	t_count 个字符.	char
string_iterator_t t_begin, string_iterator_t t_end	数据区间[t_begin, t_end)	range

下面是各种操作对字符串参数的支持情况(使用后缀表示上表的各种参数情况):

	string	substring	cstr	substr	char	range
初始化	yes	yes	yes	yes	yes	yes
赋值	yes	yes	yes	yes	yes	yes
添加	yes	yes	yes	yes	yes	yes
比较	yes	--	yes	--	--	--
compare	yes	yes	yes	yes	--	--
connect	yes	--	yes	--	yes	--
find	yes	--	yes	yes	yes	--
push back	--	--	--	--	yes	--
插入(下标版本)	yes	yes	yes	yes	yes	--
插入(迭代器版本)	--	--	--	--	yes	yes
替换(下标	yes	yes	yes	yes	yes	--

版本)						
替换 (迭代器版本)	yes	yes	yes	yes	yes	yes

3. 初始化和销毁

string_init()	初始化一个空的 <code>string t</code> 容器类型.
string_init_cstr()	使用一个字符串常量初始化一个 <code>string t</code> 容器类型.
string_init_subcstr()	使用子字符串初始化一个 <code>string t</code> 容器类型.
string_init_char()	使用 <code>n</code> 个指定字符初始化一个 <code>string t</code> 容器类型.
string_init_copy()	使用另一个 <code>string t</code> 类型来初始化一个 <code>string t</code> 容器类型.
string_init_copy_substring()	使用另一个 <code>string t</code> 类型的子串来初始化一个 <code>string t</code> 容器类型.
string_init_copy_range()	使用一个数据区间来初始化一个 <code>string t</code> 容器类型.
string_destroy()	销毁 <code>string t</code> 容器类型.

`string_t` 容器中只能保存 `char` 类型的数据, 所以对于 `string_t` 容器类型就确定下来了, 所以不需要创建类型的函数, 因此 `libcstl` 没有提供 `create_string()` 函数.

4. string_t 和 C 字符串

string_c_str()	返回一个指向以 <code>'\0'</code> 结尾的字符数组的指针.
string_data()	返回一个指向字符数组的指针.
string_copy()	将 <code>string t</code> 的子串拷贝到用户指定的缓冲区中.

这三个操作函数都是提供了从 `string_t` 到 C 字符串的转换, 它们的结果都是普通的 C 字符串或 C 字符数组, 但是它们之间是有区别的.

`string_c_str()` 返回一个指向以 `'\0'` 结尾的字符数组的指针, 但是 `string_data()` 返回一个指向字符数组的指针并没有明确说明数组是不是以 `'\0'` 结尾的. 同时 `string_copy()` 只是将 `string_t` 的子串拷贝到用户指定的缓冲区中, 并不会在拷贝的字符后面自动的添加 `'\0'`.

```

string_t t_str;
string_init_cstr(&t_str, "12345");

atoi(string_c_str(&t_str)); /* 将字符串转变成整数 */
f(string_data(&t_str), string_length(&t_str)); /* 使用字符数组及其长度最为参数
调用函数 f() */

char ac_buf[100];
string_copy(&t_str, ac_buff, 100, 2); /* 从第二个字符开始拷贝最多 100 个字符 */

```

`string_c_str()` 和 `string_data()` 返回的指针并不是持续有效的, 当 `string_t` 中的数据修改之后, 内部可能重新分配内存, 那么在修改之前获得的指针就失效了, 必须重新获得指针.

```

string_t t_str;
string_init_cstr(&t_str, "12345");
f(string_c_str(&t_str));           /* 通过这种方式使用总是有效的 */

char* p = string_c_str(&t_str);    /* 获得指向C字符串的指针 */
foo(p);                           /* 在t_str没有修改之前,p是有效的 */
string_connect_cstr(&t_str, "abc"); /* 修改了t_str的内容 */
foo(p);                           /* t_str被修改, p可能会失效 */

```

5. string_t 的大小和容量

string_size()	返回 string_t 容器中字符的个数.
string_length()	返回 string_t 容器字符串的长度,与 string_size() 相同.
string_max_size()	返回 string_t 容器能够保存的字符的最大数目.
string_capacity()	返回 string_t 容器的容量.
string_empty()	判断 string_t 容器是否为空.
string_reserve()	设置 string_t 容器的容量.
string_resize()	设置 string_t 容器中保存的字符的数目.

这些操作函数与 vector_t 容器的相应操作功能相似,只是多了一个 string_length() 函数,它与 string_size() 函数功能相同.

6. string_t 数据访问

string_at()	通过下标随机访问 string_t 容器中的字符.
-------------	---------------------------

通过 string_at() 函数可以对 string_t 中的数据随机访问,它返回 char* 类型,如果 string_t 中的数据修改后,先前获得指向字符的指针可能无效.

```

string_t t_str;
char* p;
string_init_cstr(&t_str, "abcde");

p = string_at(&t_str, 2);    /* *p == 'c' */
*p = 'X';                   /* string_t 中的数据变成了" abXde" */
string_assign_cstr(&t_str, "new long value");
*p = 'Y';                   /* string_t 中的数据被修改,p 可能已经无效 */

```

7. string_t 比较操作

string_equal()	判断两个 string_t 容器是否相等.
string_not_equal()	判断两个 string_t 容器是否不等.

string_less()	判断第一个 string_t 容器是否小于第二个 string_t 容器.
string_less_equal()	判断第一个 string_t 是否小于等于第二个 string_t.
string_great()	判断第一个 string_t 是否大于第二个 string_t.
string_great_equal()	判断第一个 string_t 是否大于等于第二个 string_t.
string_equal_cstr()	判断 string_t 是否等于字符串常量.
string_not_equal_cstr()	判断 string_t 是否不等于字符串常量.
string_less_cstr()	判断 string_t 是否小于字符串常量.
string_less_equal_cstr()	判断 string_t 是否小于等于字符串常量.
string_great_cstr()	判断 string_t 是否大于字符串常量.
string_great_equal_cstr()	判断 string_t 是否大于等于字符串常量.
string_compare()	两个 string_t 比较.
string_compare_substring_string()	第一个 string_t 的子串与第二个 string_t 比较.
string_compare_substring_substring()	两个 string_t 的子串比较.
string_compare_cstr()	string_t 与字符串常量比较.
string_compare_substring_cstr()	string_t 的子串与字符串常量比较.
string_compare_substring_subcstr()	string_t 的子串与字符串常量的子串比较.

string_t 的比较操作函数与其他容器的比较操作函数相同,除此之外 string_t 还有另外的比较函数,这些函数可以比较子串,而且与 strcmp 相似返回 3 种状态的值.

```
string_t t_str;
string_init_cstr(&t_str, "abcd");
string_compare_cstr(&t_str, "abcd"); /* return 0 */
string_compare_cstr(&t_str, "dcba"); /* return a value < 0 */
string_compare_cstr(&t_str, "ab"); /* return a value > 0 */
string_compare(&t_str, &t_str); /* return 0 */
string_compare_substring_substring(&t_str, 0, 2, &t_str, 2, 2);
/* return a value < 0 "ab" < cd " */
string_compare_substring_subcstr(&t_str, 1, 2, "bcx", 2);
/* return 0 "bc" == "bc" */
```

8. 赋值

string_assign()	使用一个 string_t 为 string_t 赋值.
string_assign_substring()	使用一个 string_t 的子串为 string_t 赋值.
string_assign_cstr()	使用一个字符串常量为 string_t 赋值.
string_assign_subcstr()	使用一个字符串常量的子串为 string_t 赋值.
string_assign_char()	使用 n 个指定的字符为 string_t 赋值.
string_assign_range()	使用数据区间为 string_t 赋值.

为了修改 string_t 中的字符串的值可以使用赋值操作函数,你可以使用 string_t, C 字符串, 以及它们的子串, 或者单个字符为 string_t 赋值.

```
string_t t_s, t_srt;
string_init_cstr(&t_s, "othello");
```



```
string_init(&t_str);
string_assign(&t_str, &t_s); /* assign "othello" */
string_assign_substring(&t_str, &t_s, 1, 3); /* assign "the" */
string_assign_substring(&t_str, &t_s, 2, NPOS); /* assign "hello" */
string_assign_cstr(&t_str, "two\nlines"); /* assign C-str */
string_assign_subcstr(&t_str, "nice", 3); /* assign "nic" */
string_assign_char(&t_str, 5, 'x'); /* assign "xxxxx" */
```

9. 数据交换

string_swap()	交换两个 string_t 的数据.
---------------	--------------------

这个操作函数与其他的容器类型相应的函数功能相同.

10. 添加和连接

string_append()	向 string t 后面添加一个 string t.
string_append_substring()	向 string t 后面添加一个 string t 的子串.
string_append_cstr()	向 string t 后面添加一个字符串常量.
string_append_subcstr()	向 string t 后面添加一个字符串常量的子串.
string_append_char()	向 string t 后面添加 n 个指定的字符.
string_append_range()	向 string t 后面添加一个数据区间.
string_connect()	将两个 string_t 连接在一起.
string_connect_cstr()	将一个 string_t 和字符串常量连接在一起.
string_connect_char()	将一个 string_t 和一个字符连接在一起.
string_push_back()	在 string t 最后添加一个字符.

这类操作一共包括三种操作,添加,连接和 push_back,添加操作使用更广泛,可以向 string_t 后面添加 string_t, 字符串常量,以及它们的子串,字符和数据区间.连接操作不能连接子串,字符也只能连接一个字符.string_push_back() 函数更简单,它的功能只是向 string_t 后面添加一个字符.

```
string_t t_s, t_str;
string_init_cstr(&t_s, "othello");
string_init(&t_str);
string_connect(&t_str, &t_s); /* append "othello" */
string_connect_cstr(&t_str, "two\nlines"); /* append C-str */
string_connect_char(&t_str, '\n'); /* append single character */
string_append(&t_str, &t_s); /* append "othello" */
string_append_substring(&t_str, &t_s, 1, 3); /* append "the" */
string_append_substring(&t_str, &t_s, 2, NPOS); /* append "hello" */
string_append_cstr(&t_str, "two\nlines"); /* append C-str */
string_append_subcstr(&t_str, "nice", 3); /* append "nic" */
string_append_char(&t_str, 5, 'x'); /* append "xxxxx" */
string_push_back(&t_str, '\n'); /* append signal character */
```

11. 插入数据

string_insert()	在 string t 指定位置插入字符 (迭代器版本)。
string_insert_n()	在 string t 指定位置插入 n 个字符 (迭代器版本)。
string_insert_range()	在 string t 指定位置插入一个数据区间 (迭代器版本)。
string_insert_string()	在 string t 的指定位置插入 string t。
string_insert_substring()	在 string t 的指定位置插入 string t 的子串。
string_insert_cstr()	在 string t 的指定位置插入字符串常量。
string_insert_subcstr()	在 string t 的指定位置插入字符串常量的子串。
string_insert_char()	在 string t 的指定位置插入 n 个字符。

string_t 的插入函数分为迭代器版本和下标版本, 迭代器版本是指使用迭代器来表示插入的位置, 下标版本是使用下标来表示插入的位置。

```
string_t t_s, t_srt;  
string_init_cstr(&t_s, "age");  
string_init_cstr(&t_str, "p");  
string_insert_string(&t_str, 1, &t_s); /* "page" */  
string_insert_cstr(&t_str, 1, "ersifl"); /* "persiflage" */
```

12. 删除数据

string_erase()	删除指定位置的字符。
string_erase_range()	删除指定数据区间内的字符。
string_erase_substring()	删除 string t 的子串。
string_clear()	清空 string t。

string_t 的删除操作与其他容器的相应函数功能相同, 此为还有一个使用下标的删除函数用来删除子串, string_erase_substring()。

```
string_t t_str;  
string_init_cstr(&t_str, "internationalization");  
string_erase_substring(&t_str, 13, NPOS); /* "international" */  
string_erase_substring(&t_str, 7, 5); /* "internal" */
```

13. 替换

string_replace()	使用 string t 替换。
string_replace_substring()	使用 string t 的子串替换。
string_replace_cstr()	使用字符串常量替换。
string_replace_subcstr()	使用字符串常量的子串替换。
string_replace_char()	使用 n 个指定字符替换。
string_range_replace()	使用 string t 替换 (迭代器版本)。

<code>string_range_replace_substring()</code>	使用 <code>string_t</code> 的子串替换 (迭代器版本) .
<code>string_range_replace_cstr()</code>	使用字符串常量替换 (迭代器版本) .
<code>string_range_replace_subcstr()</code>	使用字符串常量的子串替换 (迭代器版本) .
<code>string_range_replace_char()</code>	使用 <code>n</code> 个指定字符替换 (迭代器版本) .
<code>string_replace_range()</code>	使用数据区间替换 (迭代器版本) .

`string_t` 的替换操作也分为下标版本和迭代器版本, 下标版本是使用下标来指定被替换的范围, 迭代器版本是使用迭代器来指定被替换的范围.

14. 子串

<code>string_substr()</code>	返回 <code>string_t</code> 的子串.
------------------------------	-------------------------------

```
这个函数通过下标来确定 string_t 类型的子串, 子串的类型也是 string_t.  
string_t t_s, t_str;  
string_init_cstr(&t_s, "interchangeability");  
t_str = string_substr(&t_s, 0, NPOS); /* the copy of t_s */  
string_destroy(&t_str);  
t_str = string_substr(&t_s, 11, NPOS); /* "ability" */  
string_destroy(&t_str);  
t_str = string_substr(&t_s, 5, 6); /* "change" */  
string_destroy(&t_str);
```

注意: `string_substr()` 函数的返回值是一个已经初始化的 `string_t` 类型, 所以这个函数的返回值不能忽略, 同时必须使用为初始化的 `string_t` 变量去接收返回值, 使用之后要销毁. 这个函数与 `xxxx_equal_range()` 函数一样.

15. 输入输出

<code>string_output()</code>	将 <code>string_t</code> 输出指定的流中.
<code>string_input()</code>	从指定的流中获得输入.
<code>string_getline()</code>	从指定的流中获得一行输入.
<code>string_getline_delimiter()</code>	从指定的流中获得一行输入, 使用用户自定义的换行符.

`string_t` 类型的输入输出函数, 使 `string_t` 类型与标准输入输出或文件的互操作更简便. 例如使用下面的操作方式:

```
while(string_getline(&t_str, stdin))  
{  
...  
}  
或者调用者可以自定义换行符, 例如使用 ':' 作为换行符:
```

```

while(string_getline_delimiter(&t_str, stdin, `:`))
{
...
}

```

16. 查找

string_find()	从指定位置开始查找 string_t 的第一个位置.
string_find_cstr()	从指定位置开始查找字符串常量的第一个位置.
string_find_subcstr()	从指定位置开始查找字符串常量的子串的第一个位置.
string_find_char()	从指定位置开始查找指定的字符第一个位置.
string_rfind()	从指定位置开始向前查找 string_t 的最后一个位置.
string_rfind_cstr()	从指定位置开始向前查找字符串常量的最后一个位置.
string_rfind_subcstr()	从指定位置开始向前查找字符串的子串的最后一个位置.
string_rfind_char()	从指定位置开始向前查找指定字符的最后一个位置.
string_find_first_of()	从指定位置开始查找 string_t 包含的任意一个字符的第一个位置.
string_find_first_of_cstr()	从指定位置开始查找字符串常量中包含的任意一个字符的第一个位置.
string_find_first_of_subcstr()	从指定位置开始查找字符串常量的子串包含的任意一个字符的第一个位置.
string_find_first_of_char()	从指定位置开始查找指定字符的第一个位置.
string_find_last_of()	从指定位置开始向前查找 string_t 包含的任意一个字符的最后一个位置.
string_find_last_of_cstr()	从指定位置开始向前查找字符串常量包含的任意一个字符的最后一个位置.
string_find_last_of_subcstr()	从指定位置开始向前查找字符串常量的子串中包含的任意一个字符的最后一个位置.
string_find_last_of_char()	从指定位置开始向前查找指定字符的最后一个位置.
string_find_first_not_of()	从指定位置开始查找不包含在 string_t 中的任意字符的第一个位置.
string_find_first_not_of_cstr()	从指定位置开始查找不包含在字符串常量中的任意字符的第一个位置.
string_find_first_not_of_subcstr()	从指定位置开始查找不包含在字符串常量的子串中的任意字符的第一个位置.
string_find_first_not_of_char()	从指定位置开始查找不是指定字符的任意字符的第一个位置.
string_find_last_not_of()	从指定位置开始向前查找不包含在 string_t 中的任意字符的最后一个位置.
string_find_last_not_of_cstr()	从指定位置开始向前查找不包含在字符串常量中的任意字符的最后一个位置.
string_find_last_not_of_subcstr()	从指定位置开始向前查找不包含在字符串常量的子串中的任意字符的最后一个位置.
string_find_last_not_of_char()	从指定位置开始向前查找不是指定字符的任意字符的最后一个位置.

总体来说, 查找函数分为三类: 第一类是查找子串, 如 find 和 rfind 类的函数, 第二类是查找指定的字符串中的任意字符, 如 find_first, find_last 类的函数, 第三类是查找出指定字符串包含的字符以为的字符, 如 find_first_not, find_last_not 类函数. 例如:

```
string_t t_str;
string_init_cstr(&t_str, "Hi Bill, I'm ill, so please pay the bill");
string_find_cstr(&t_str, "il", 0);           /* return 4 */
string_find_cstr(&t_str, "il", 10);          /* return 13 */
string_rfind_cstr(&t_str, "il", NPOS);       /* return 37 */
string_find_first_of_cstr(&t_str, "il", 0);  /* return 1 */
string_find_last_of_cstr(&t_str, "il", NPOS); /* return 39 */
string_find_first_not_of_cstr(&t_str, "il", 0); /* return 0 */
string_find_last_not_of_cstr(&t_str, "il", NPOS); /* return 36 */
```

17. NPOS 值

上一节的查找函数都是使用下标位置为参数, 同时返回结果也是下标位置, 那么当查找失败时怎么区分呢? <cstring.h>中定义了值 NPOS 表示查找失败, 例如:

```
size_t t_pos = string_find_cstr(&t_str, "not found");
if(t_pos == NPOS)
{
    ...
}
```

此外 NPOS 用在接受下标为参数的操作函数中表示长度到达字符串的末尾, 这样可以节省计算字符串常的时间, 同时使用更方便, 如在 10.1.1 的例子中

```
string_replace_cstr(&t_tmpname, t_pos + 1, NPOS, "xxx");
```

表示将从 t_pos+1 开始的一直到字符串末尾的子串替换成 "xxx".

18. 迭代器支持

string_begin()	获得指向第一个字符的迭代器.
string_end()	获得指向最后一个字符的下一个位置的迭代器.

string_t 类型同样支持迭代器, 并且 string_t 类型的迭代器时随机访问迭代器. 通过迭代器操作函数, 所有的算法都可以应用与 string_t 类型. 同时 string_t 类型本身的很多操作函数也都是有迭代器版本的.