The libcstl Library User Guide

The libcstl Library User Guide

for libcstl 2.0

Wangbo

2010-04-22

目录

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清参考《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清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
```

如果哦以前安装过 libcstl 库，清使用更新更新命令:

```
[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_MULTISET_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_MULTISET_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。

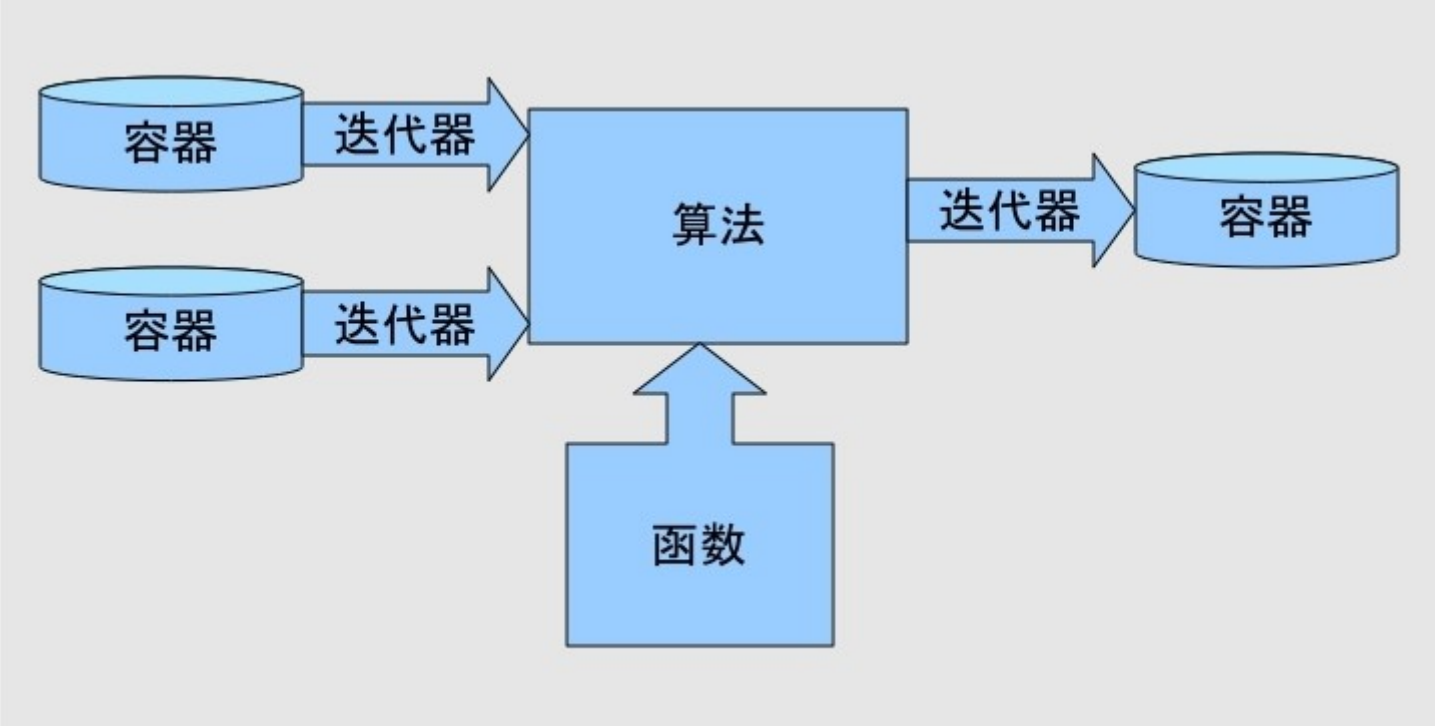

图 2.1 libcstl 组件之间的关系

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：

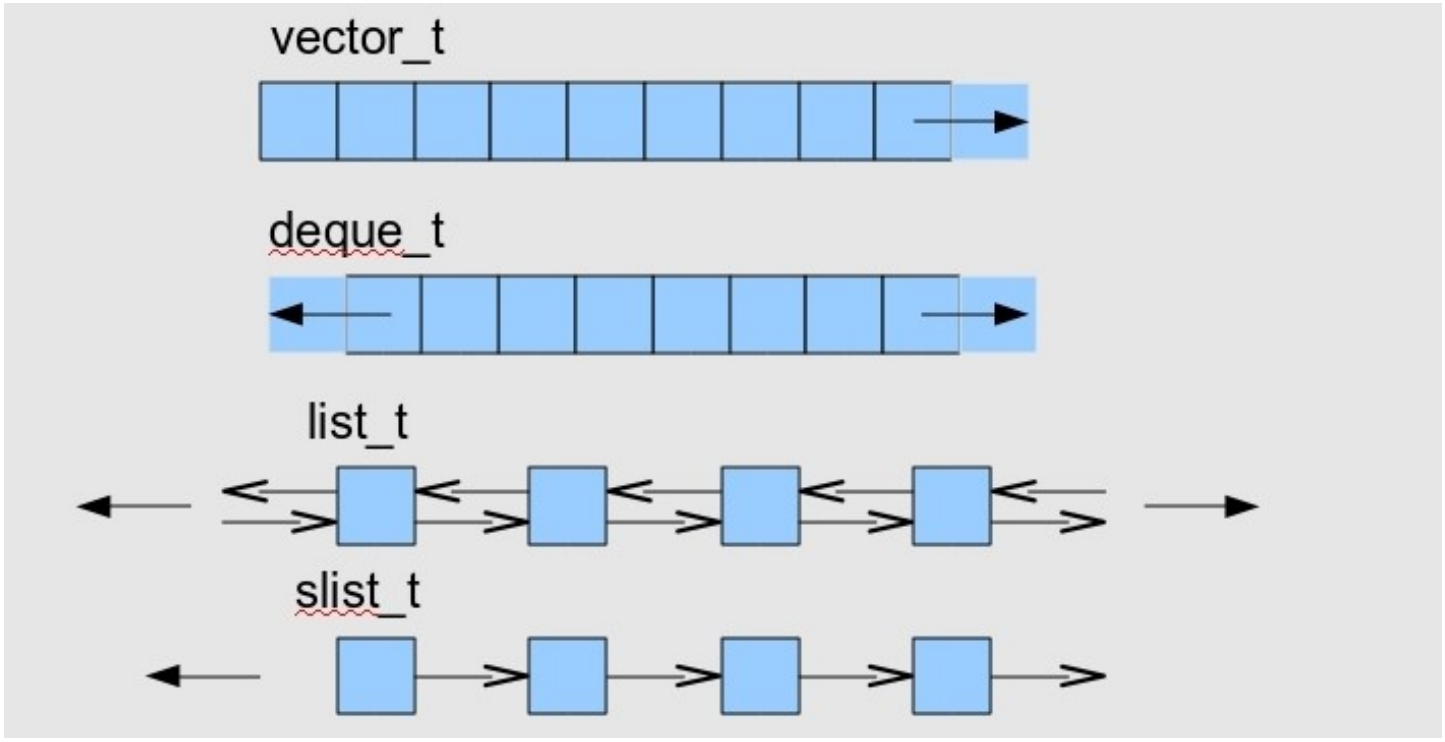

图 2.2 序列容器

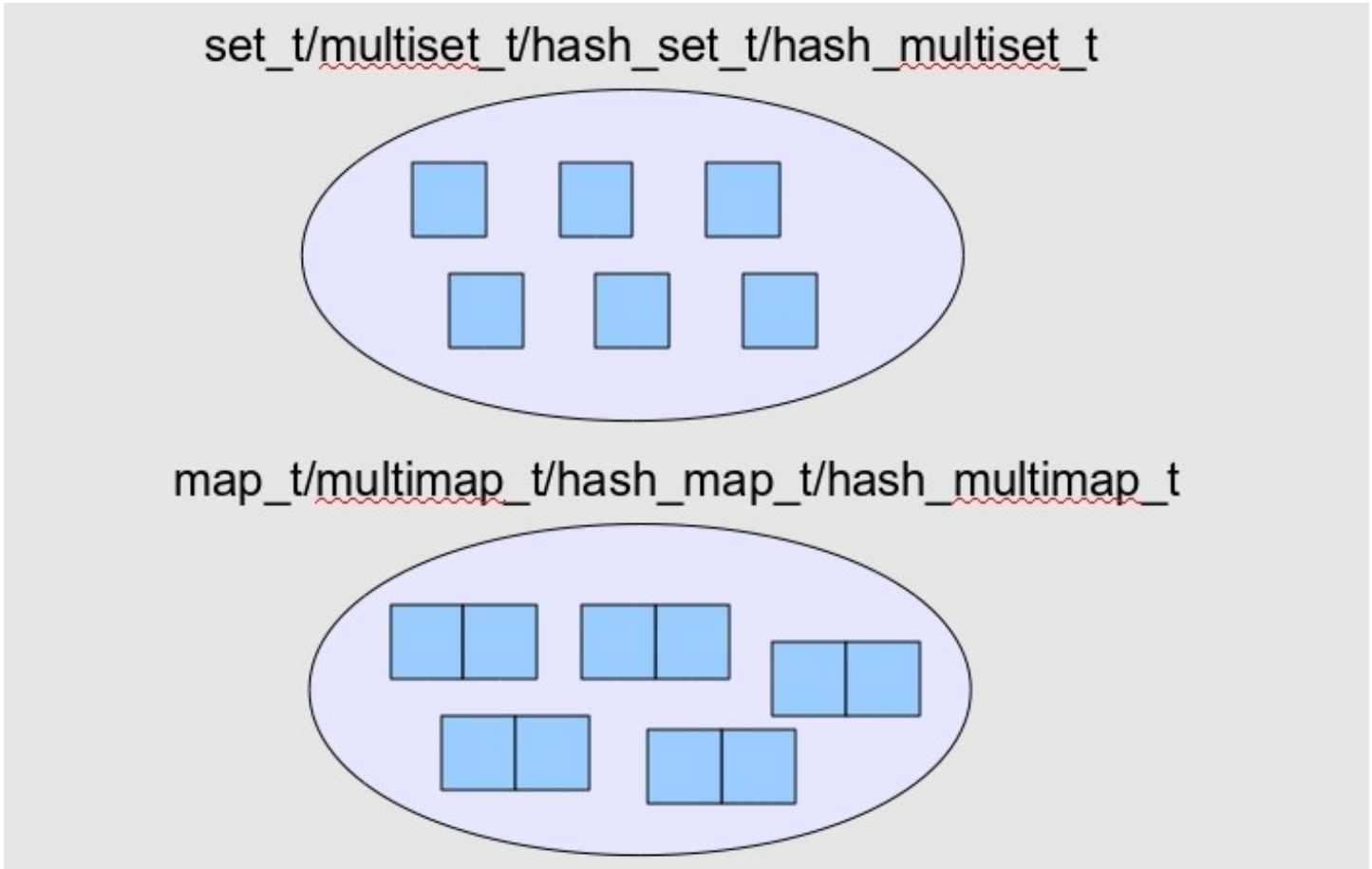

图 2.3 关联容器

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 conatiner 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与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，所以我们也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:
a b c d e f g h i j k l m n o p q r s t u v w x y z

向上面这样遍历一个链表后链表中的数据就都被删除了，如果要保留数据清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个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)，但是于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。

- **mulitset_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()。

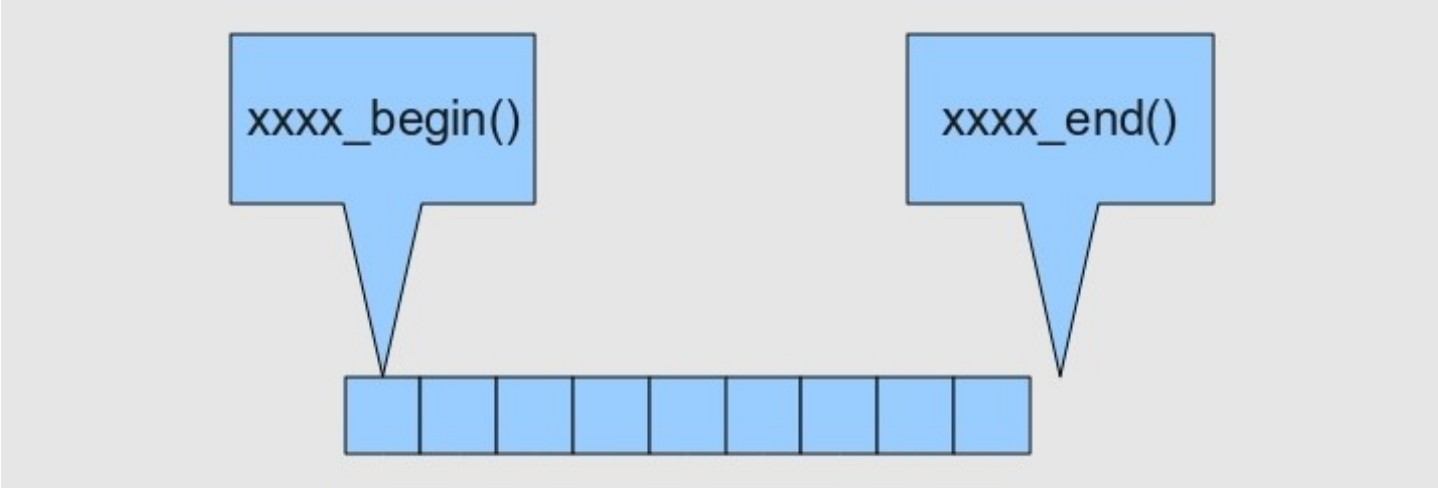

图 2.4 xxxx_begin() 和 xxxx_end()

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:

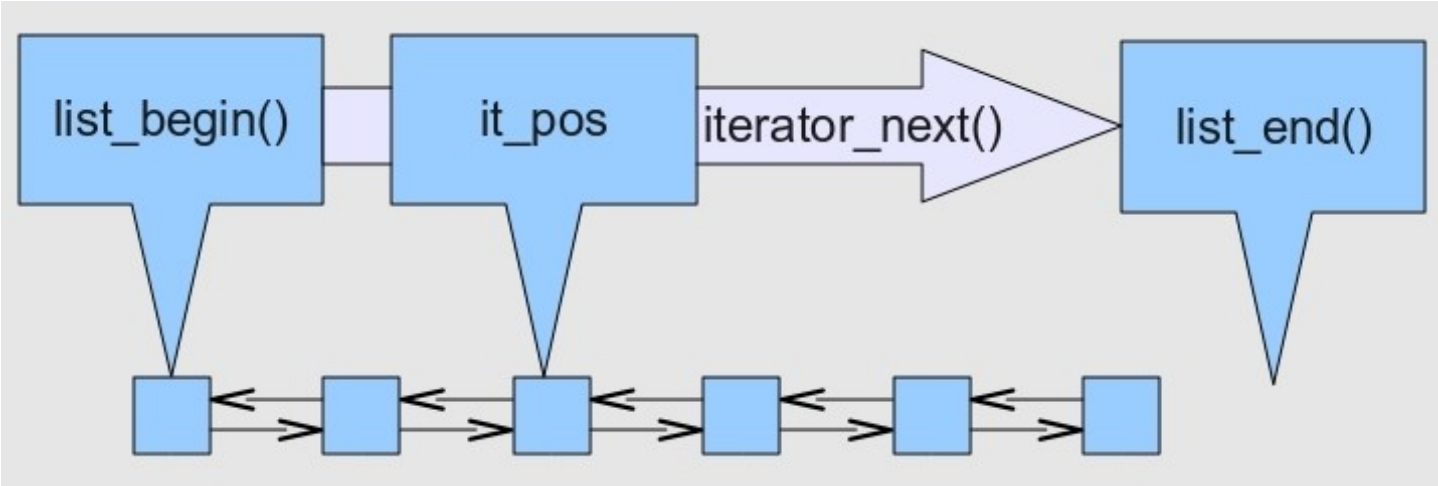

图 2.5 使用迭代器遍历链表中的数据

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，它可以用在任何容器中，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:

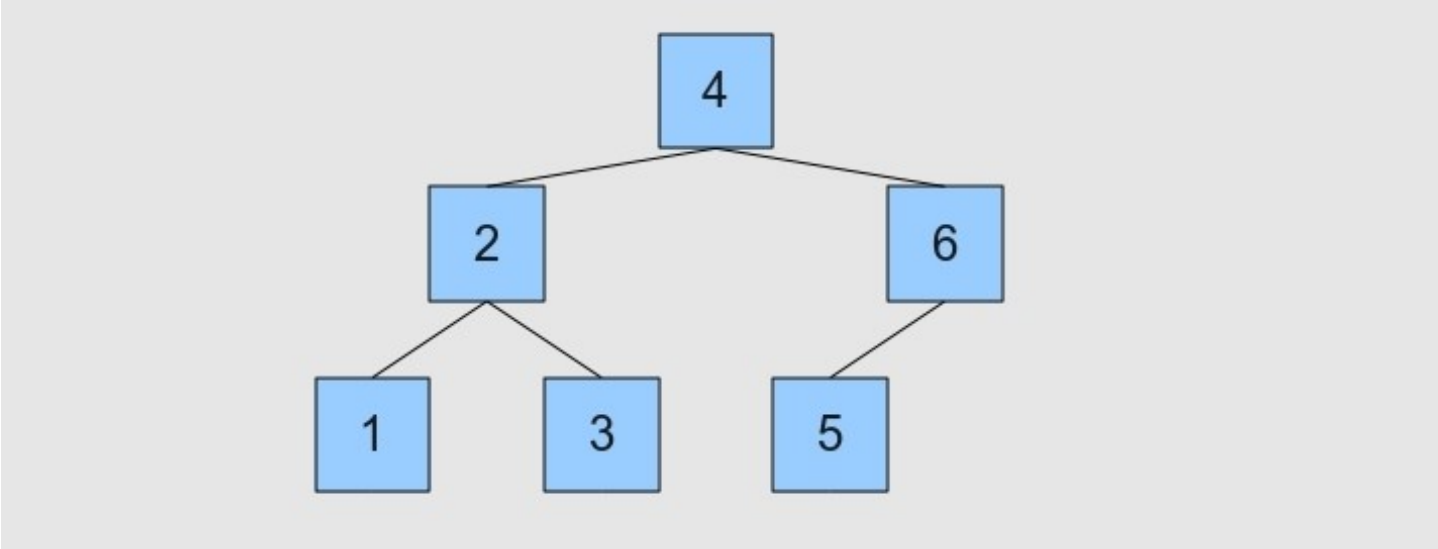

图 2.6 插入数据后的 set_t 容器

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：

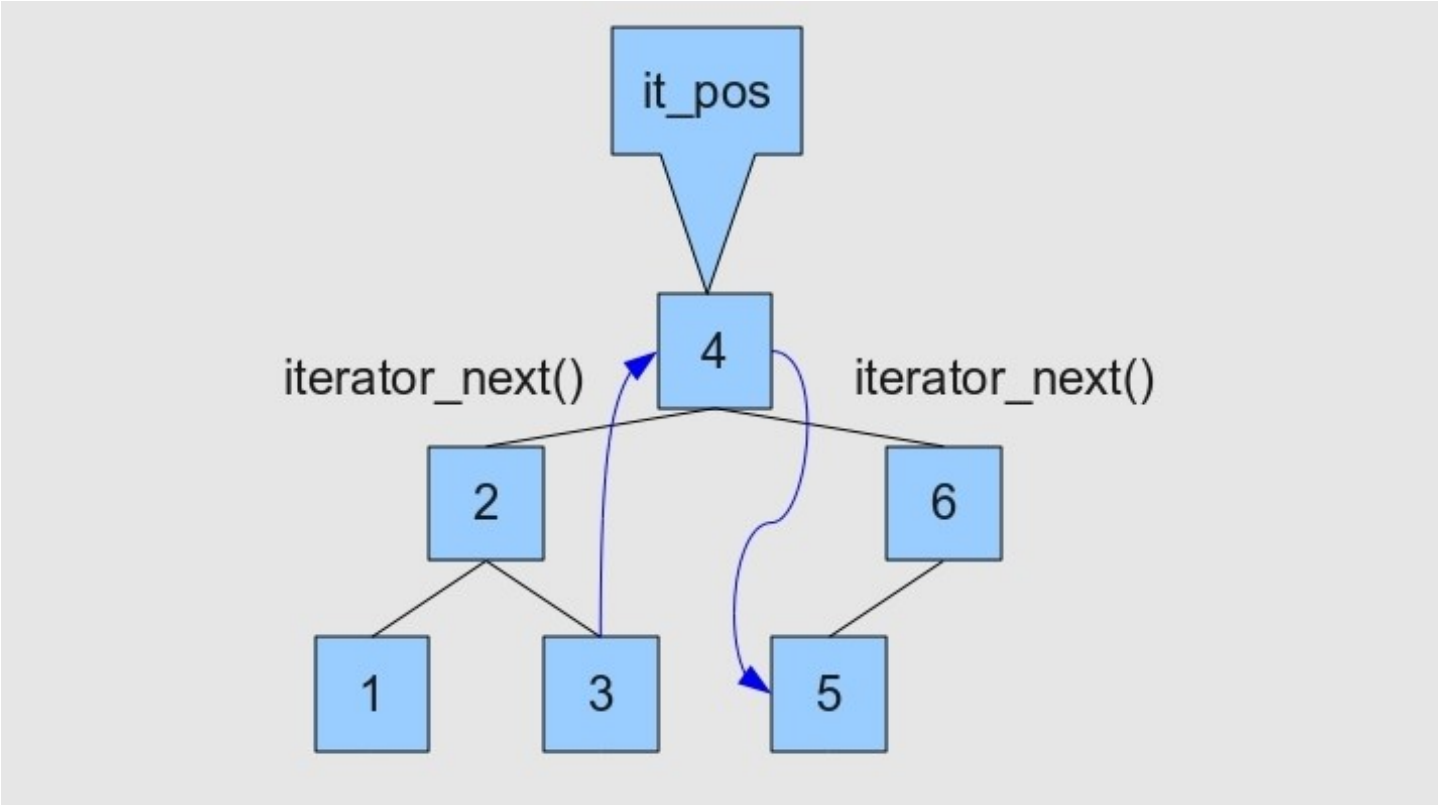

图 2.7 通过迭代器遍历关联容器

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* pmmap_coll = create_multimap(int, char*);
    multimap_iterator_t it_pos;
```

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

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

    multimap_init(pmmap_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(pmmap_coll, ppair_elem);
    pair_make(ppair_elem, 2, "a");
    multimap_insert(pmmap_coll, ppair_elem);
    pair_make(ppair_elem, 1, "this");
    multimap_insert(pmmap_coll, ppair_elem);
    pair_make(ppair_elem, 4, "of");
    multimap_insert(pmmap_coll, ppair_elem);
    pair_make(ppair_elem, 6, "strings");
    multimap_insert(pmmap_coll, ppair_elem);
    pair_make(ppair_elem, 1, "is");
    multimap_insert(pmmap_coll, ppair_elem);
    pair_make(ppair_elem, 3, "multimap");
    multimap_insert(pmmap_coll, ppair_elem);

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

    multimap_destroy(pmmap_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，否则算法的行为是位定义的(带有断言的版本会侦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）那么结果是为定义的。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大的空间：

```
/*
 * 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 distination 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。

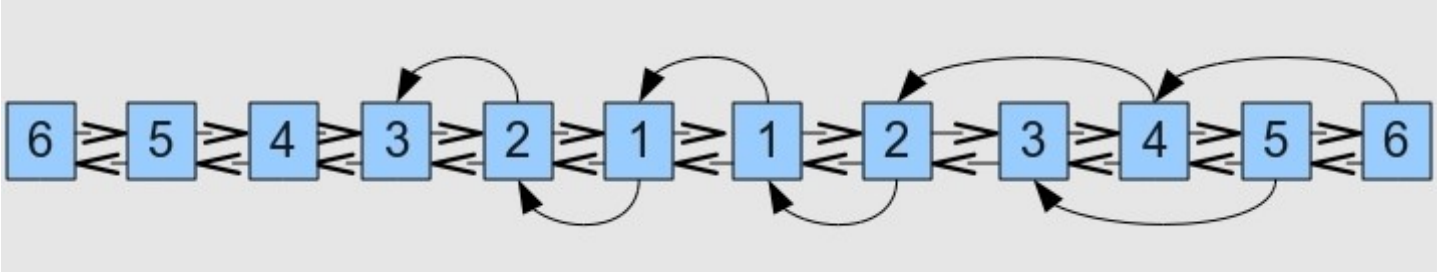

图 2.8 algo_remove()

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。

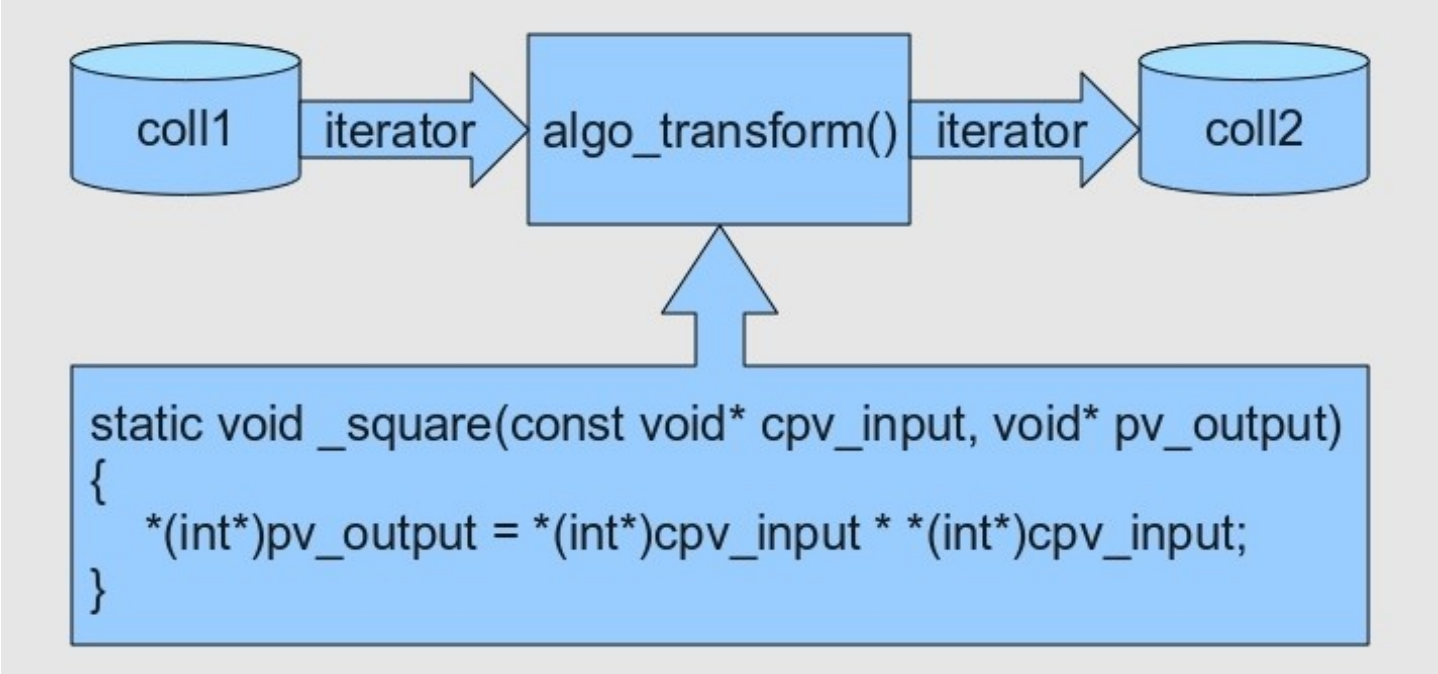

图 2.9 algo_transform()的执行方式

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_elem	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：

it_begin	表示数据区间的开始位置的迭代器。
it_end	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，对于等于和不等于两个操作函数，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。

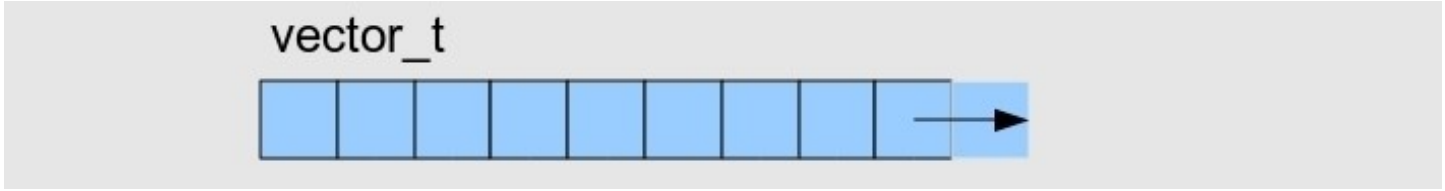

图 4.1 vector_t 的结构

要使用 vector_t 必须包含头文件**<cstl/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。

create_vector	创建一个 vector_t 容器。
vector_init	初始化一个空的 vector_t 容器。
vector_init_n	初始化一个 vector_t 容器，初始化后容器中包含多个默认数据。
vector_init_elem	初始化一个 vector_t 容器，初始化后容器中包含多个指定的数据。
vector_init_copy	初始化一个 vector_t 容器，初始化后的容器中的内容于指定的 vector_t 容器的内容相同。
vector_init_copy_range	初始化一个 vector_t 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。
vector_destroy	销毁一个 vector_t 容器，释放 vector_t 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：

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

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

- 赋值和交换

vector_assign	使用 vector_t 类型为当前的 vector_t 赋值。
vector_assign_elem	使用指定的数据为 vector_t 赋值。
vector_assign_range	使用指定的数据区间为 vector_t 赋值。
vector_swap	将两个 vector_t 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)。

vector_at	通过下标对 vector_t 中的数据进行随机访问。
vector_front	访问 vector_t 中的第一个数据。
vector_back	访问 vector_t 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

vector_begin	返回指向第一个数据的迭代器。
vector_end	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。

vector_insert	向 vector_t 中插入指定数据。
vector_insert_n	向 vector_t 中插入多个指定的数据。
vector_insert_range	向 vector_t 中插入指定数据区间的数据。
vector_push_back	向 vector_t 末尾添加一个数据。
vector_pop_back	删除 vector_t 的随后一个数据。
vector_erase	删除 vector_t 中指定位置的数据。
vector_erase_range	删除 vector_t 中指定数据区间的数据。

vector_clear	删除 vector_t 中的所有数据。
vector_resize	改变 vector_t 中数据的数量，如果数据的数量变大，用默认数据填充。
vector_resize_elem	改变 vector_t 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 technial 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 technial 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。

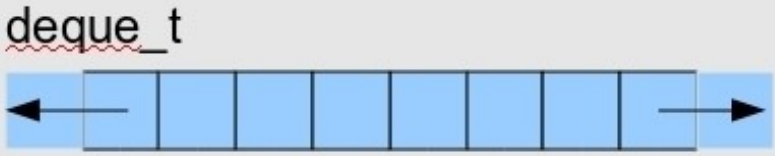

图 4.2 deque_t 的内部结构

为了使用 deque_t，必须包含头文件**<cstl/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。

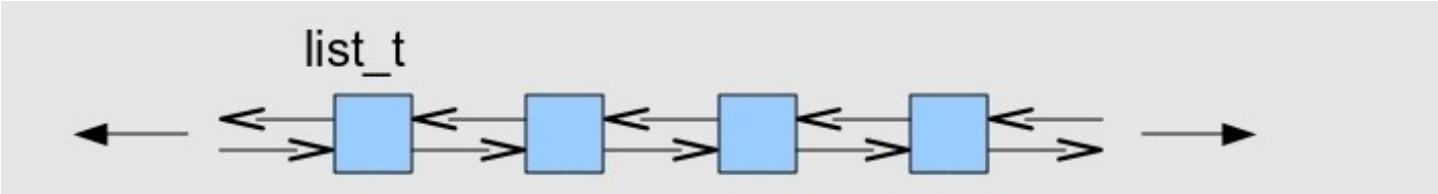

图 4.3 list_t 的结构

要使用 list_t 必须包含头文件<cstl/clist.h>

```
#include <cstl/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 添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。

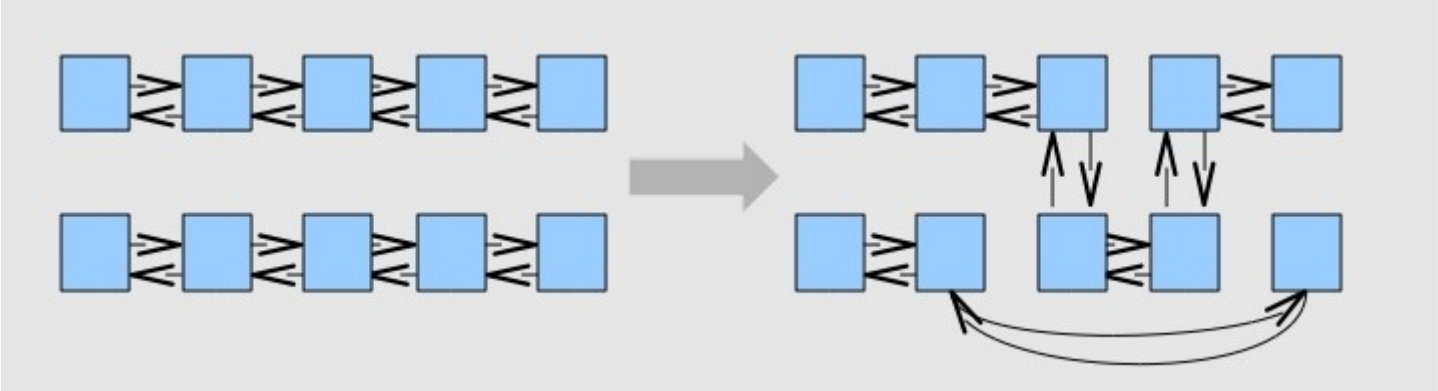

图 4.4 list_t 之间的数据转移

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。

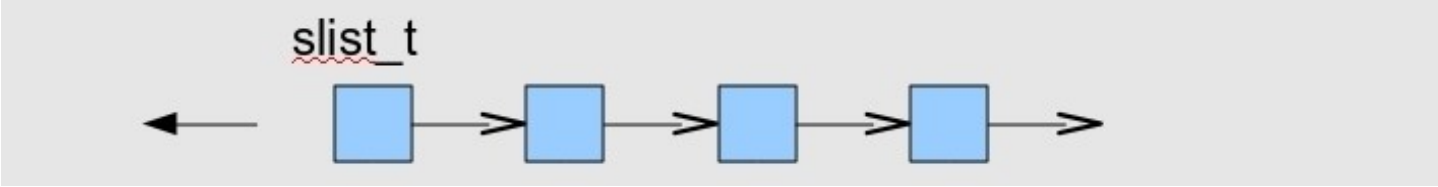

图 4.5 slist_t 的结构

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对数据进行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。

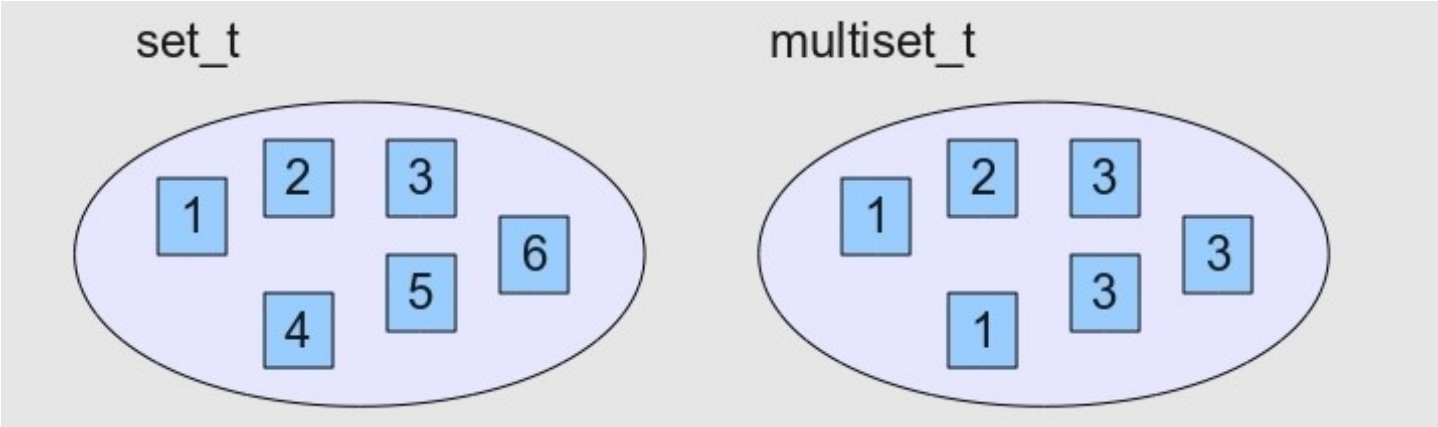

图 4.6 set_t 和 multiset_t

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)。

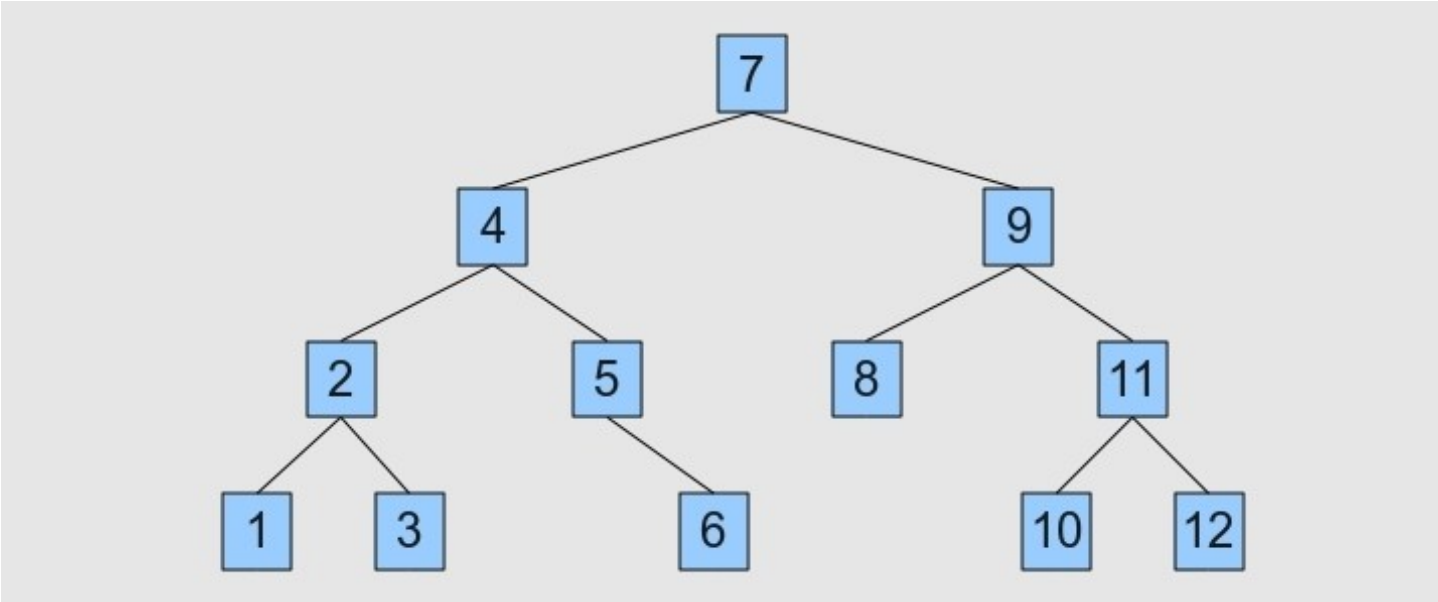

图 4.7 set_t 和 multiset_t 的内部结构

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，详细的清参考《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。

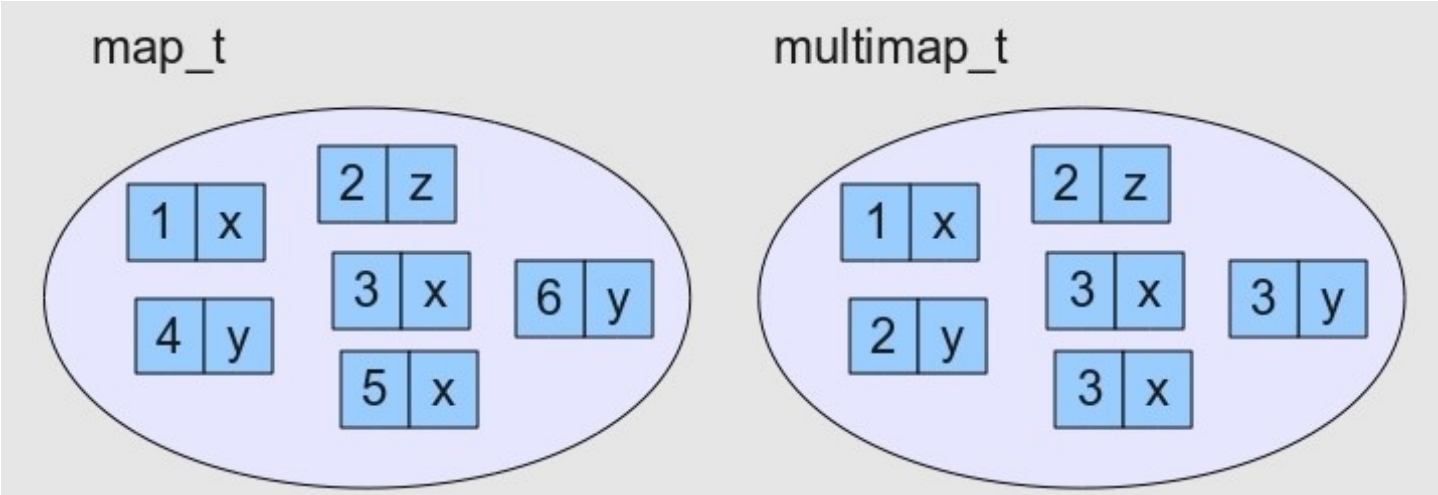

图 4.8 map_t 和 multimap_t

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

```
#include <cstl/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。

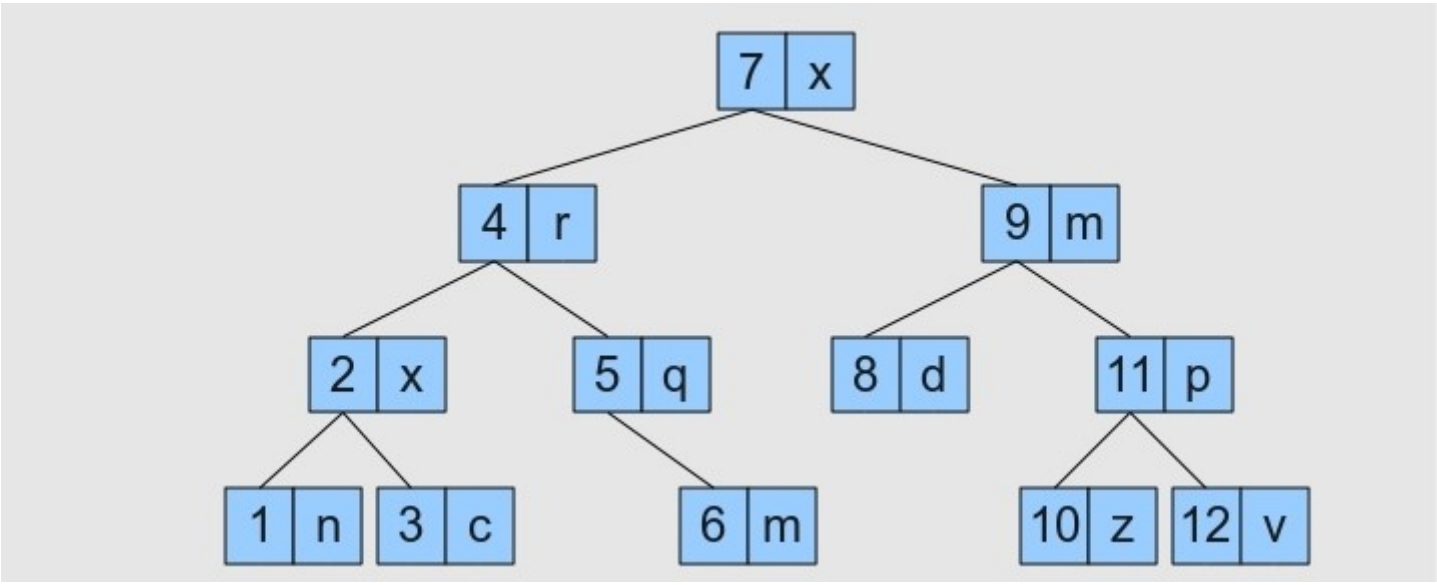

图 4.9 map_t 和 multimap_t 的内部结构

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，具体的清参考《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，但是如果要查找指定的 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。
map_insert_hint	向 map_t 中插入指定的数据，同时给出位置提示。
map_insert_range	向 map_t 中插入指定的数据区间。
map_erase	删除 map_t 中拥有指定 key 的数据。
map_erase_pos	删除 map_t 中指定位置的数据。
map_erase_range	删除 map_t 中指定数据区间的数据。
map_clear	删除 map_t 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。

map_at	通过以 key 为下标对 map_t 中的数据的 value 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* pmmap_dict = create_multimap(char*, char*);
    pair_t* ppair_item = create_pair(char*, char*);
    multimap_iterator_t it_pos;

    if(pmmap_dict == NULL || ppair_item == NULL)
    {
```

```
    return -1;
}

multimap_init(pmmap_dict);
pair_init(ppair_item);

pair_make(ppair_item, "day", "Tag");
multimap_insert(pmmap_dict, ppair_item);
pair_make(ppair_item, "strange", "fremd");
multimap_insert(pmmap_dict, ppair_item);
pair_make(ppair_item, "car", "Auto");
multimap_insert(pmmap_dict, ppair_item);
pair_make(ppair_item, "smart", "elegant");
multimap_insert(pmmap_dict, ppair_item);
pair_make(ppair_item, "trait", "Merkmal");
multimap_insert(pmmap_dict, ppair_item);
pair_make(ppair_item, "strange", "seltsam");
multimap_insert(pmmap_dict, ppair_item);
pair_make(ppair_item, "smart", "raffiniert");
multimap_insert(pmmap_dict, ppair_item);
pair_make(ppair_item, "smart", "klug");
multimap_insert(pmmap_dict, ppair_item);
pair_make(ppair_item, "clever", "raffiniert");
multimap_insert(pmmap_dict, ppair_item);

/* print all elements */
printf("english\t\tgerman\n");
printf("-------------------------------\n");
for(it_pos = multimap_begin(pmmap_dict);
    !iterator_equal(it_pos, multimap_end(pmmap_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(pmmap_dict);
    !iterator_equal(it_pos, multimap_end(pmmap_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(pmmap_dict);
        !iterator_equal(it_pos, multimap_end(pmmap_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(pmmap_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。

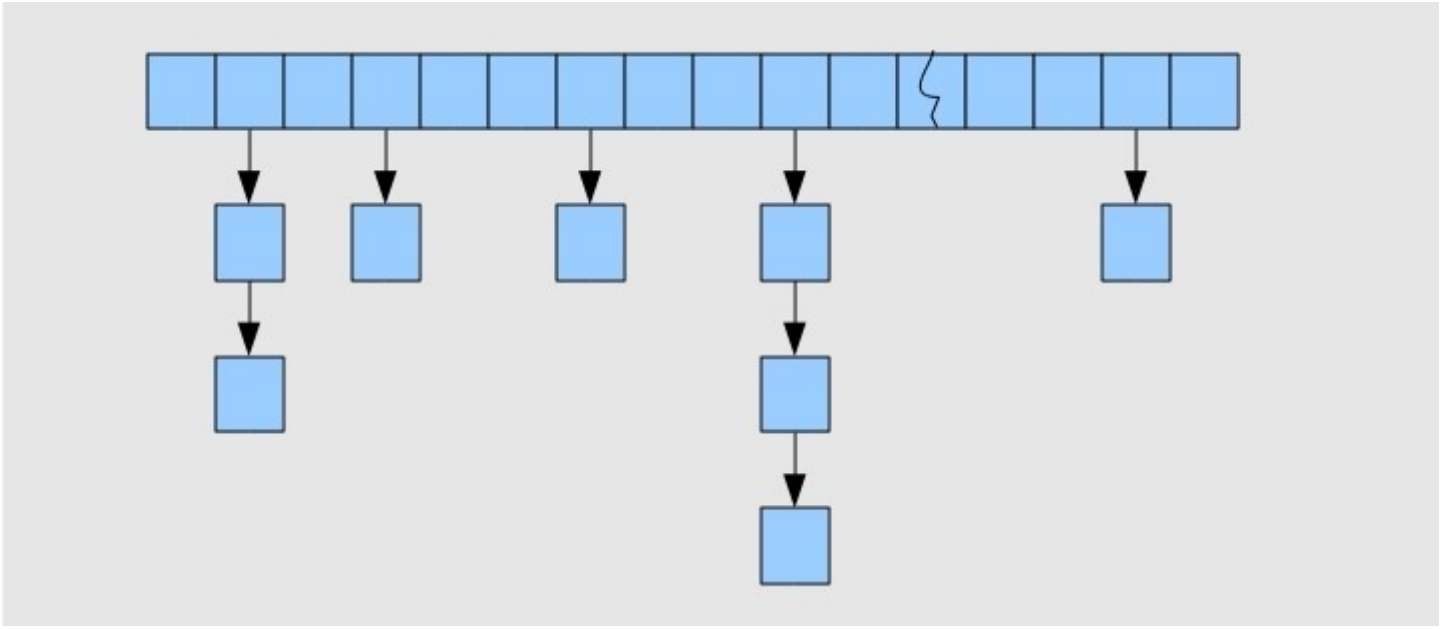

图 4.10 哈希表内部结构

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

create_hash_set	创建 hash_set_t 容器类型。

hash_set_init	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 hash_set_t。
hash_set_init_ex	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 hash_set_t。
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 中的所有的数据。

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。

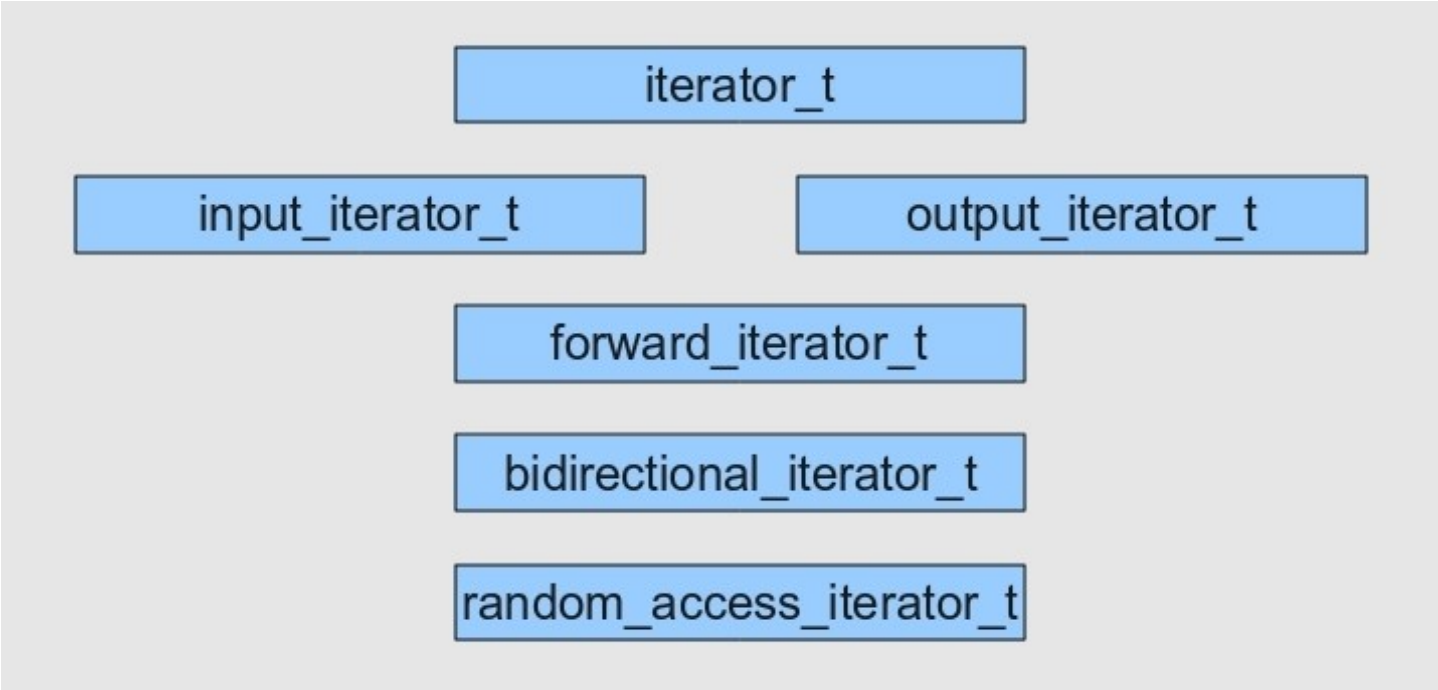

图 5.1 迭代器的种类

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：

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

iterator_set_value	设置迭代器指向的数据。
iterator_next	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：

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

iterator_prev	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：

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

```
/*
 * itercat.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))
{
    ...
}
```

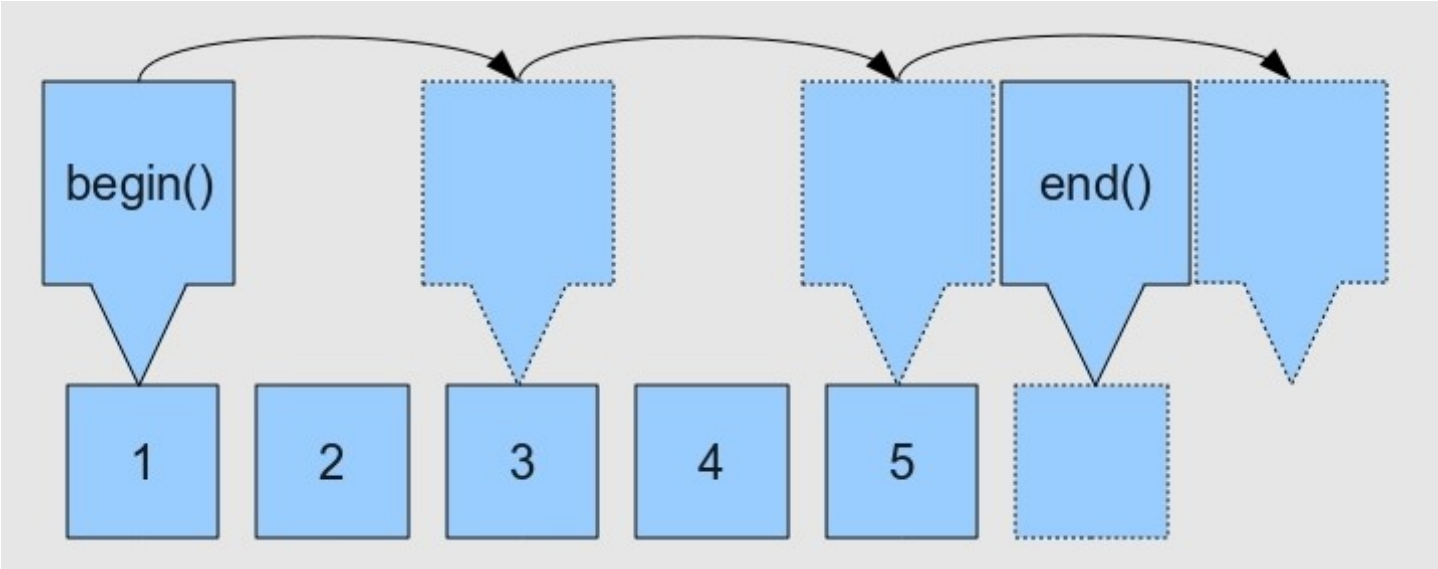

图 5.2 iterator 每次先前移动两个位置

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。

iterator_advance	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，两个迭代器必须数以同一容器。

iterator_distance	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 算法

这一章详细的讨论 libcstl 提供的算法，算法的一般概念，种类，用法，并且给出使用的实例。

第一节 算法的概述

这一节讲述一些关于算法的概念，让你了解它们的能力并且更好的使用它们来解决问题。

1. 算法头文件

算法大体上分为两种，第一种是普通的算法，这类算法提供了诸如排序，删除，查找等等这样的算法，要使用这类算法必须包含头文件<cstl/calgorithm.h>：

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

此外 libcstl 还为算数运算提供了算数算法，要使用算数算法要包含头文件<cstl/cnumeric.h>：

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

2. 算法的共同特点

在第二章中已经介绍了一些算法的概念和特点，算法通常处理一个或者多个数据区间，第一个数据区间要给出开始和末尾，其他的数据区间只给出开始就可以了，因为算法可以通过第一个区间来计算后面数据区间的长度，所以要求调用算法时要保证数据区间是有效的，也就是说数据区间的开始必须和末尾属于同一个容器并且开始位置在末尾之前或者和末尾相等，需要计算长度的数据区间一定要有足够的数据，因为算法只是覆盖已有的数据而不是插入新数据。

为了提高算法的可扩展性和能力，在调用算法的时候还可以传递自定义的函数，这些函数在算法内部调用，用户可以使用它们来改变算法的默认行为。输出是 bool_t 类型的函数我们叫做谓词，函数和谓词我们在下一章中介绍。

3. 算法的种类

不同的算法满足了不同的功能，可以根据算法的功能将算法划分为不同的类型，有的只是读取数据，有的要修改数据，有的只是改变数据的顺序。

从算法的命名上也能够看出算法的主要功能，算法的名字中有两种特殊的后最：

- _if 后缀

带有_if 后缀的算法是提供给用户扩展算法使用的，一个算法通常提供两个版本普通版本和带有_if 后缀的版本，普通的版本使用的默认的算法规则，带有_if 后缀的版本允许用户输入一个自定义的函数来改变算法的行为，从而增强了算法的扩展性。例如 algo_find()算法是在数据区间中查找指定的数据，algo_find_if()是在数据区间中查找符合指定规则的数据。

- _copy 后缀

带有_copy 后缀的算法是将算法处理的数据拷贝到一个新的数据区间，而不是覆盖原有的数据区间，这样的算法通常也有不带有_copy 后缀的算法。例如 algo_reverse()是将数据区间中的数据逆序，是直接在当前数据区间中修改的，algo_reverse_if()是将逆序后的结构拷贝到其他数据区间中，原有的数据区间不受到破坏。

上面是按照名字的后缀简单的区分一下算法，下面按照功能区分算法可以分为四类：

- 非质变算法
- 质变算法
- 排序算法
- 算数算法

接下来就按照上面的分类详细介绍每一种算法。

第二节 非质变算法

非质变算法既不修改数据本身也不修改数据的顺序。非质变算法要求使用 input_iterator_t 或者 forward_iterator_t 类型的迭代器。所以所有的容器都可以应用非质变算法。下面的表格列出所有 libcstl 提供的非质变算法：

algo_for_each	对数据区间中的每一个数据都应用指定的函数。
algo_find	在数据区间中查找指定的数据。
algo_find_if	在数据区间中查找符合指定函数的数据。
algo_adjacent_find	查找数据区间中相邻的相等数据。
algo_adjacent_find_if	查找数据区间中相邻并符合指定函数规则的数据。
algo_find_first_of	在数据区间中查找第一个等于指定某个指定数据的数据。
algo_find_first_of_if	在数据区间中查找第一个与某个指定数据符合特定函数规则的数据。
algo_count	统计数据区间中包含指定数据的个数。
algo_count_if	统计数据区间中符合指定函数规则的数据个数。
algo_mismatch	返回数据区间中第一个与指定的数据不同的数据。
algo_mismatch_if	返回数据区间中第一个不符合指定函数规则的数据。
algo_equal	测试两个数据区间中的数据是否相等。
algo_equal_if	测试两个数据区间中的数据是否符合指定的函数规则。
algo_search	在数据区间中查找第一个指定的子区间。

algo_search_if	在数据区间中查找第一个符合指定函数规则的子区间。
algo_search_n	在数据区间中查找连续 n 个指定数据。
algo_search_n_if	在数据区间中查找联系 n 个符合指定函数规则的数据。
algo_search_end	在数据区间中查找最后一个指定的子区间。
algo_search_end_if	在数据区间中查找最后一个符合指定函数规则的子区间。
algo_find_end	同 algo_search_end
algo_find_end_if	同 algo_search_end_if

其中 algo_for_each()不能严格算是非质变算法。

1. algo_for_each 算法

algo_for_each 算法可扩展性非常强，它接受用户的函数，对数据区间中的所有数据执行指定的函数，例如指定的函数是显示数据信息，那么这个算法的效果就是显示出数据区间中所有的数据信息，但是如果指定的函数是修改数据的值，那么这个算法的效果就是修改了数据区间中所有数据的值，这样看来 algo_for_each()算法可能是非质变算法也可能是质变算法，这主要看指定的函数。建议不要使用 algo_for_each()算法改变数据，因为算法提供了改变每一个数据的算法，在下一节介绍。下面的例子使用 algo_for_each 算法打印每一个数据：

```
/*
 * 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;
}
```

程序输出的结果：
1 2 3 4 5 6 7 8 9

algo_for_each 算法对 pvec_coll 中的每一个数据都应用了_print 函数，结果就是打印出了 pvec_coll 中的所有数据。

下面这个例子就是演示一下如何使用 algo_for_each 算法修改数据区间中的数据：

```
/*
 * foreach2.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);
}

static void _plus_10(const void* cpv_input, void* pv_output)
{
    *(int*)cpv_input += 10;
}

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");


    algo_for_each(vector_begin(pvec_coll), vector_end(pvec_coll), _plus_10);

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

    vector_destroy(pvec_coll);

    return 0;
}
```

输出结果：
1 2 3 4 5 6 7 8 9
11 12 13 14 15 16 17 18 19

algo_for_each 算法将每一个数据都加上 10。

通过上面两个例子看出 algo_for_each 算法只用到函数的输入参数，并没有用到输出参数，所以在使用 algo_for_each 算法的时候输出的函数不要使用输出参数。

2. 搜索数据

- algo_find　algo_find_if 搜索第一个匹配的数据

1. algo_find 返回数据区间中与指定数据相等的第一个数据。
2. algo_find_if 返回数据区间中符合指定函数的第一个数据。
3. 如果没有符合条件的数据，两个算法都返回数据区间的末尾。

4. algo_find_if 中的函数一定不能够修改数据。
5. 关联容器提供了搜索的操作函数。

第一例子展示了如何使用 algo_find 算法：

```
/*
 * find2.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_pos1;
    list_iterator_t it_pos2;
    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);
    }
    for(i = 1; i <= 9; ++i)
    {
        list_push_back(plist_coll, i);
    }
    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");
```

```
    /* find first element with value 4 */
    it_pos1 = algo_find(list_begin(plist_coll), list_end(plist_coll), 4);

    /* find second element with value 4 */
    if(!iterator_equal(it_pos1, list_end(plist_coll)))
    {
        it_pos2 = algo_find(iterator_next(it_pos1), list_end(plist_coll), 4);
    }

    /* print all elements from first to second 4 */
    if(!iterator_equal(it_pos1, list_end(plist_coll)) &&
       !iterator_equal(it_pos2, list_end(plist_coll)))
    {
        for(it_pos = it_pos1;
            !iterator_equal(it_pos, iterator_next(it_pos2));
            it_pos = iterator_next(it_pos))
        {
            printf("%d ", *(int*)iterator_get_pointer(it_pos));
        }
        printf("\n");
    }

    list_destroy(plist_coll);

    return 0;
}
```

为了找到第二个 4，必须将指向第一个 4 的迭代器向前移动一个数据，在每次查找之后要比较一下，返回的迭代器是否和数据区间的末尾相等。这个例子的结果：

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

4 5 6 7 8 9 1 2 3 4

下面这个例子展示了如何使用 algo_find_if 算法查找符合条件的数据：

```
/*
 * find3.c
 * compile with : -lcstl
 */

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

static void _greater_3(const void* cpv_input, void* pv_output)
{
```

```
    *(bool_t*)pv_output = *(int*)cpv_input > 3 ? true : false;
}

static void _divisible_3(const void* cpv_input, void* pv_output)
{
    *(bool_t*)pv_output = *(int*)cpv_input % 3 == 0 ? true : false;
}

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

    if(pvec_coll == 0)
    {
        return -1;
    }

    vector_init(pvec_coll);

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

    /* find first element greater than 3 */
    it_pos = algo_find_if(vector_begin(pvec_coll),
        vector_end(pvec_coll), _greater_3);

    /* print its posititon */
    printf("the %d. element is the first greater than 3.\n",
        iterator_distance(vector_begin(pvec_coll), it_pos) + 1);

    /* find first element divisible by 3 */
    it_pos = algo_find_if(vector_begin(pvec_coll),
```

```
        vector_end(pvec_coll), _divisible_3);

    /* print its position */
    printf("the %d. element is the first divisible by 3.\n",
        iterator_distance(vector_begin(pvec_coll), it_pos) + 1);

    vector_destroy(pvec_coll);

    return 0;
}
```

例子中分别使用了两函数_greater_3 和_divisible_3，在数据区间中查找第一个大于 3 和第一个能够被三整除的数据。程序的结果：

1 2 3 4 5 6 7 8 9

the 4. element is the first greater than 3.

the 3. element is the first divisible by 3.

- algo_adjacent_find　algo_adjacent_find_if 搜索相邻的符合规则的数据

1. algo_adjacent_find 算法返回指向数据区间中第一组相邻且相等的数据的第一个数据的迭代器。
2. algo_adjacent_find_if 算法发挥指向数据区间中第一组相邻且符合规则的数据的第一个数据的迭代器。
3. 如果不成功两个算法都返回数据区间末尾的迭代器。
4. algo_adjacent_find_if 算法使用的函数一定不能修改该传入的数据。

下面的例子演示了这两个算法的用法：

```
/*
 * adjfind1.c
 * compile with : -lcstl
 */

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

/*
 * return whether the second data has double the value of the first.
 */
static void _doubled(const void* cpv_first, const void* cpv_second, void* pv_output)
{
    *(bool_t*)pv_output = *(int*)cpv_first * 2 == *(int*)cpv_second ? true : false;
}

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);

vector_push_back(pvec_coll, 1);
vector_push_back(pvec_coll, 3);
vector_push_back(pvec_coll, 2);
vector_push_back(pvec_coll, 4);
vector_push_back(pvec_coll, 5);
vector_push_back(pvec_coll, 5);
vector_push_back(pvec_coll, 0);

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");

/* search first two elements with equal values */
it_pos = algo_adjacent_find(vector_begin(pvec_coll), vector_end(pvec_coll));
if(!iterator_equal(it_pos, vector_end(pvec_coll)))
{
    printf("first two elements with equal value have position %d\n",
        iterator_distance(vector_begin(pvec_coll), it_pos) + 1);
}

/*
 * search first two elements for which
 * the second has double the value of the first
 */
it_pos = algo_adjacent_find_if(vector_begin(pvec_coll),
    vector_end(pvec_coll), _doubled);
if(!iterator_equal(it_pos, vector_end(pvec_coll)))
{
    printf("first two elements with second value twice the first has pos %d\n",
        iterator_distance(vector_begin(pvec_coll), it_pos) + 1);
}
```

```
    vector_destroy(pvec_coll);

    return 0;
}
```

使用 algo_adjacent_find 算法是为了查找相等的数据，使用 algo_adjacent_find_if 算法是为了查找满足_double 函数的数据，程序的输出结果如下：

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 has pos 3

- algo_find_first_of　algo_find_first_of_if 在数据区间中查找多个可能数据中的第一个

1. algo_find_first_of 算法返回第一个出现在第一个数据区间中同时也出现在第二个数据区间中的数据。
2. algo_find_first_of_if 算法返回第一个出现在第一个数据区间中并且和第二个数据区间中的数据满足指定规则的数据。
3. 如果不成功两个算法都返回第一个数据区间末尾的迭代器。
4. algo_find_first_of_if 算法使用的函数不能修改使用的数据。

下面的例子展示如何使用这两个算法：

```
/*
 * findof1.c
 * compile with : -lcstl
 */

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

static void _doubled(const void* cpv_first, const void* cpv_second, void* pv_output)
{
    *(bool_t*)pv_output = *(int*)cpv_first == *(int*)cpv_second * 2 ? true : false;
}

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

    if(pvec_coll == NULL || plist_searchcoll == NULL)
    {
        return -1;
```

```
}

vector_init(pvec_coll);
list_init(plist_searchcoll);

for(i = 1; i <= 11; ++i)
{
    vector_push_back(pvec_coll, i);
}
for(i = 3; i <= 5; ++i)
{
    list_push_back(plist_searchcoll, i);
}

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

/* search first occurrence of an element of searchcoll in coll */
it_pos = algo_find_first_of(vector_begin(pvec_coll), vector_end(pvec_coll),
    list_begin(plist_searchcoll), list_end(plist_searchcoll));
if(!iterator_equal(it_pos, vector_end(pvec_coll)))
{
    printf("first element of searchcoll in coll is element %d\n",
        iterator_distance(vector_begin(pvec_coll), it_pos) + 1);
}

it_pos = algo_find_first_of_if(vector_begin(pvec_coll), vector_end(pvec_coll),
    list_begin(plist_searchcoll), list_end(plist_searchcoll), _doubled);
if(!iterator_equal(it_pos, vector_end(pvec_coll)))
{
    printf("first doubled element of searchcoll in coll is element %d\n",
```

```
            iterator_distance(vector_begin(pvec_coll), it_pos) + 1);
    }

    vector_destroy(pvec_coll);
    list_destroy(plist_searchcoll);

    return 0;
}
```

这个程序的结果如下：
1 2 3 4 5 6 7 8 9 10 11
3 4 5
first element of searchcoll in coll is element 3
first doubled element of searchcoll in coll is element 6

- algo_search　algo_search_if 搜索第一个子区间

1. algo_search 返回第二个数据区间在第一个数据区间中第一次出现的位置。
2. algo_search_if 返回第一个数据区间中第一个与第二个数据区间中的全部数据满足指定规则的位置。
3. 如果失败两个算法都返回第一个数据区间的末尾。
4. algo_search_if 算法中使用的函数不能修改数据。

下面是展示如果使用 algo_search 算法的例子：

```
/*
 * search1.c
 * compile with : -lcstl
 */

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

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

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

    deque_init(pdeq_coll);
```

```
list_init(plist_subcoll);

for(i = 1; i <= 7; ++i)
{
    deque_push_back(pdeq_coll, i);
}
for(i = 1; i <= 7; ++i)
{
    deque_push_back(pdeq_coll, i);
}
for(i = 3; i <= 6; ++i)
{
    list_push_back(plist_subcoll, i);
}

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

it_pos = algo_search(deque_begin(pdeq_coll), deque_end(pdeq_coll),
    list_begin(plist_subcoll), list_end(plist_subcoll));
while(!iterator_equal(it_pos, deque_end(pdeq_coll)))
{
    printf("subcoll found starting with element %d\n",
        iterator_distance(deque_begin(pdeq_coll), it_pos) + 1);

    it_pos = algo_search(iterator_next(it_pos), deque_end(pdeq_coll),
        list_begin(plist_subcoll), list_end(plist_subcoll));
}

deque_destroy(pdeq_coll);
list_destroy(plist_subcoll);
```

```
    return 0;
}
```

这个例子的输出结果：

1 2 3 4 5 6 7 1 2 3 4 5 6 7

3 4 5 6

subcoll found starting with element 3

subcoll found starting with element 10

接下来的例子展示了 algo_search_if 算法的使用：

```
/*
 * search2.c
 * compile with : -lcstl
 */

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

static void _check_even(const void* cpv_first,
    const void* cpv_second, void* pv_output)
{
    if(*(bool_t*)cpv_second)
    {
        *(bool_t*)pv_output = *(int*)cpv_first % 2 == 0 ? true : false;
    }
    else
    {
        *(bool_t*)pv_output = *(int*)cpv_first % 2 == 1 ? true : false;
    }
}

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

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

```
    vector_init(pvec_coll);
    vector_init(pvec_subcoll);

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

    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");

    it_pos = algo_search_if(vector_begin(pvec_coll), vector_end(pvec_coll),
        vector_begin(pvec_subcoll), vector_end(pvec_subcoll), _check_even);
    while(!iterator_equal(it_pos, vector_end(pvec_coll)))
    {
        printf("subcoll found starting with element %d\n",
            iterator_distance(vector_begin(pvec_coll), it_pos) + 1);

        it_pos = algo_search_if(iterator_next(it_pos), vector_end(pvec_coll),
            vector_begin(pvec_subcoll), vector_end(pvec_subcoll), _check_even);
    }

    vector_destroy(pvec_coll);
    vector_destroy(pvec_subcoll);

    return 0;
}
```

这个例子的输出结果：

1 2 3 4 5 6 7 8 9

subcoll found starting with element 2

subcoll found starting with element 4

subcoll found starting with element 6

- algo_search_n　algo_search_n_if 在数据区间中查找第一个连续 n 个符合规则的数据

1. algo_search_n 返回数据区间中第一个出现连续 n 个指定数据的位置迭代器。
2. algo_search_n_if 返回数据区间中第一个出现连续 n 个满足指定规则的位置迭代器。
3. 如果失败，这个算法都返回数据区间的末尾。
4. algo_search_n_if 算法使用的函数不能修改数据。

下面的例子展示了这两个算法的用法：

```
/*
 * searchn1.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;
    int i = 0;

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

    deque_init(pdeq_coll);

    for(i = 1; i <= 9; ++i)
    {
        deque_push_back(pdeq_coll, i);
    }
    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");

    it_pos = algo_search_n(deque_begin(pdeq_coll), deque_end(pdeq_coll), 4, 3);
    if(iterator_equal(it_pos, deque_end(pdeq_coll)))
    {
```

```
        printf("no four consecutive elements with value 3 found.\n");
    }
    else
    {
        printf("four consecutive elements with value 3 start with %d\n",
            iterator_distance(deque_begin(pdeq_coll), it_pos) + 1);
    }

    it_pos = algo_search_n_if(deque_begin(pdeq_coll),
        deque_end(pdeq_coll), 4, 3, fun_greater_int);
    if(iterator_equal(it_pos, deque_end(pdeq_coll)))
    {
        printf("no four consecutive elements with value > 3 found.\n");
    }
    else
    {
        printf("four consecutive elements with value > 3 start with %d\n",
            iterator_distance(deque_begin(pdeq_coll), it_pos) + 1);
    }

    deque_destroy(pdeq_coll);

    return 0;
}
```

程序的输出结果如下：

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

- algo_search_end　algo_search_end_if　algo_find_end　algo_find_end_if 查找最有一个子区间

这四个算法中 algo_search_end 和 algo_find_end 的功能是一样的，algo_search_end_if 和 algo_find_end_if 的功能是一样的，从功能上来看，这些算法实现的是 search 的功能，但是为什么有名字为 find 的算法呢，这是因为为了与 SGI STL 算法的名字兼容(以后的版本这个算法的的 find 命名版本可能要去掉)。

1. algo_search_end 和 algo_find_end 算法返回第二个数据区间在第一个数据区间中最后一次出现的位置。
2. algo_search_end_if 和 algo_find_end_if 算法返回第一个数据区间中与第二个数据区间共同满足规则的最后一个指定子区间的位置。
3. 如果失败四个算法都返回第一个数据区间的末尾。
4. algo_search_end_if 和 algo_find_end_if 使用的函数不能修改数据。

下面所展示这四个算法用法的例子：

```
/*
 * searchend1.c
 * compile with : -lcstl
 */
```

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

int main(int argc, char* argv[])
{
    deque_t* pdeq_coll = create_deque(int);
    list_t* plist_subcoll = create_list(int);
    iterator_t it_pos;
    iterator_t it_end;
    int i = 0;

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

    deque_init(pdeq_coll);
    list_init(plist_subcoll);

    for(i = 1; i <= 7; ++i)
    {
        deque_push_back(pdeq_coll, i);
    }
    for(i = 1; i <= 7; ++i)
    {
        deque_push_back(pdeq_coll, i);
    }
    for(i = 3; i <= 5; ++i)
    {
        list_push_back(plist_subcoll, i);
    }

    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");
    for(it_pos = list_begin(plist_subcoll);
```

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

    it_pos = algo_search_end(deque_begin(pdeq_coll), deque_end(pdeq_coll),
        list_begin(plist_subcoll), list_end(plist_subcoll));
    it_end = deque_end(pdeq_coll);
    while(!iterator_equal(it_pos, it_end))
    {
        printf("subcoll found starting with element %d\n",
            iterator_distance(deque_begin(pdeq_coll), it_pos) + 1);

        it_end = it_pos;
        it_pos = algo_find_end(deque_begin(pdeq_coll), it_end,
            list_begin(plist_subcoll), list_end(plist_subcoll));
    }

    deque_destroy(pdeq_coll);
    list_destroy(plist_subcoll);

    return 0;
}
```

程序的输出：
1 2 3 4 5 6 7 1 2 3 4 5 6 7
3 4 5
subcoll found starting with element 10
subcoll found starting with element 3

3. 统计数据个数

- algo_count　algo_count_if 统计数据区间中符合规则的数据个数

1. algo_count 返回数据区间中指定数据的个数。
2. algo_count_if 返回数据区间中符合指定规则的数据的个数。
3. algo_count_if 使用的函数不能修改数据。
4. 关联容器提供的统计数据个数的操作函数。

下面的例子展示了如何使用这两个算法：

```
/*
```

```
 * count1.c
 * compile with : -lcstl
 */

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

static void _is_even(const void* cpv_input, void* pv_output)
{
    *(bool_t*)pv_output = *(int*)cpv_input % 2 == 0 ? true : false;
}

int main(int argc, char* argv[])
{
    vector_t* pvec_coll = create_vector(int);
    iterator_t it_pos;
    size_t t_count = 0;
    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);
    }

    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");

    t_count = algo_count(vector_begin(pvec_coll), vector_end(pvec_coll), 4);
    printf("number of elements equal to 4: %u.\n", t_count);
```

```
    t_count = algo_count_if(vector_begin(pvec_coll),
        vector_end(pvec_coll), _is_even);
    printf("number of elements with even values: %u.\n", t_count);

    vector_destroy(pvec_coll);

    return 0;
}
```

程序的输出结果：
1 2 3 4 5 6 7 8 9
number of elements equal to 4: 1.
number of elements with even values: 4.

4. 数据区间的比较

- algo_equal　algo_equal_if 测试两个数据区间是否相等

1. algo_equal 测试两个数据区间是否相等。
2. algo_equal_if 测试两个数据区间是否符合指定的规则。
3. algo_equal_if 使用的函数不能修改数据。
4. 用户必须保证第二个数据区间要有足够的数据。

下面的例子展示了如何使用这两个算法：

```
/*
 * equal1.c
 * compile with : -lcstl
 */

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

static void _both_even_or_odd(const void* cpv_first,
    const void* cpv_second, void* pv_output)
{
    *(bool_t*)pv_output = *(int*)cpv_first % 2 == *(int*)cpv_second % 2 ?
        true : false;
}

int main(int argc, char* argv[])
{
```

```
vector_t* pvec_coll1 = create_vector(int);
list_t* plist_coll2 = create_list(int);
iterator_t it_pos;
int i = 0;

if(pvec_coll1 == NULL || plist_coll2 == NULL)
{
    return -1;
}

vector_init(pvec_coll1);
list_init(plist_coll2);

for(i = 1; i <= 7; ++i)
{
    vector_push_back(pvec_coll1, i);
}
for(i = 3; i <= 9; ++i)
{
    list_push_back(plist_coll2, i);
}

for(it_pos = vector_begin(pvec_coll1);
    !iterator_equal(it_pos, vector_end(pvec_coll1));
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");
for(it_pos = list_begin(plist_coll2);
    !iterator_equal(it_pos, list_end(plist_coll2));
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");

if(algo_equal(vector_begin(pvec_coll1),
    vector_end(pvec_coll1), list_begin(plist_coll2)))
{
    printf("coll1 == coll2\n");
}
else
```

```
    {
        printf("coll1 != coll2\n");
    }

    if(algo_equal_if(vector_begin(pvec_coll1), vector_end(pvec_coll1),
        list_begin(plist_coll2), _both_even_or_odd))
    {
        printf("even and odd elements correspond\n");
    }
    else
    {
        printf("even and odd elements do not correspond\n");
    }

    vector_destroy(pvec_coll1);
    list_destroy(plist_coll2);

    return 0;
}
```

程序的输出结果：
1 2 3 4 5 6 7
3 4 5 6 7 8 9
coll1 != coll2
even and odd elements correspond

algo_mismatch　algo_mismatch_if 返回第一个不匹配的位置
algo_mismatch 返回第一处两个数据区间不匹配的位置。
algo_mismatch_if 返回第一处两个数据区间不符合规则的位置。
如果两个数据区间完全匹配，返回两个区间的末尾。
algo_mismatch_if 使用的函数不能修改数据。
用户必须保证第二个数据区间足够大。
下面的例子展示怎样使用这两算法：

```
/*
 * misma1.c
 * compile with : -lcstl
 */

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

```
int main(int argc, char* argv[])
{
    vector_t* pvec_coll1 = create_vector(int);
    list_t* plist_coll2 = create_list(int);
    iterator_t it_pos;
    range_t t_result;
    int i = 0;

    if(pvec_coll1 == NULL || plist_coll2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_coll1);
    list_init(plist_coll2);

    for(i = 1; i <= 6; ++i)
    {
        vector_push_back(pvec_coll1, i);
    }
    for(i = 1; i <= 16; i *= 2)
    {
        list_push_back(plist_coll2, i);
    }
    list_push_back(plist_coll2, 3);

    for(it_pos = vector_begin(pvec_coll1);
        !iterator_equal(it_pos, vector_end(pvec_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");
    for(it_pos = list_begin(plist_coll2);
        !iterator_equal(it_pos, list_end(plist_coll2));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    t_result = algo_mismatch(vector_begin(pvec_coll1),
        vector_end(pvec_coll1), list_begin(plist_coll2));
```

```
    if(iterator_equal(t_result.it_begin, vector_end(pvec_coll1)))
    {
        printf("no mismatch\n");
    }
    else
    {
        printf("first mismatch: %d and %d\n",
            *(int*)iterator_get_pointer(t_result.it_begin),
            *(int*)iterator_get_pointer(t_result.it_end));
    }

    t_result = algo_mismatch_if(vector_begin(pvec_coll1),
        vector_end(pvec_coll1), list_begin(plist_coll2), fun_less_equal_int);
    if(iterator_equal(t_result.it_begin, vector_end(pvec_coll1)))
    {
        printf("always less-or-equal\n");
    }
    else
    {
        printf("not less-or-equal: %d and %d\n",
            *(int*)iterator_get_pointer(t_result.it_begin),
            *(int*)iterator_get_pointer(t_result.it_end));
    }

    vector_destroy(pvec_coll1);
    list_destroy(plist_coll2);

    return 0;
}
```

程序的输出结果：

1 2 3 4 5 6

1 2 4 8 16 3

first mismatch: 3 and 4

not less-or-equal: 6 and 3

第三节 质变算法

质变算法通常要修改数据区间中的数据，修改数据的方式有两种，一种就是算法直接修改了数据区间中的数据，另一种就是将修改后的数据拷贝到新的数据区间中，后一种质变算法都是带有_copy 后缀的。

关联容器不可以作为质变算法的目的容器，因为关联容器中的数据和存储位置是有直接关系的，修改了数据就破坏了关联容器中数据的存储顺序。

下面列出了所有的质变算法：

algo_copy	将源数据区间中的数据拷贝到目的数据区间。
algo_copy_n	将 n 个数据拷贝到目的数据区间。
algo_copy_backward	从后向前将源数据区间中的数据拷贝到目的数据区间中。
algo_swap	交换两个迭代器指向的数据。
algo_iter_swap	交换两个迭代器指向的数据。
algo_swap_ranges	交换两个数据区间中的数据。
algo_transform	将源数据区间中的数据经过转换后拷贝到目的数据区间中。
algo_transform_binary	将来自于两个源数据区间中的数据经过转换拷贝到目的数据区间中。
algo_replace	将数据区间中的指定数据替换成新数据。
algo_replace_if	将数据区间中的符合规则的数据替换成新数据。
algo_replace_copy	将数据区间中的指定数据替换成新数据，将替换结果拷贝到新数据区间中。
algo_replace_copy_if	将数据区间中的符合规则的数据替换成新数据，将替换结果拷贝到新区间中。
algo_fill	向数据区间中填充指定的数据。
algo_fill_n	将 n 个数据填充成指定的数据。
algo_generate	使用指定的规则产生的数据填充数据区间。
algo_generate_n	使用指定的规则产生的数据填充 n 个数据。
algo_remove	移除数据区间中指定的数据。
algo_remove_if	移除数据区间中符合规则的数据。
algo_remove_copy	移除数据区间中的指定数据，把结果拷贝到新数据区间。
algo_remove_copy_if	移除数据区间中的符合规则的数据，把结果拷贝到新数据区间中。
algo_unique	移除数据区间中相邻的重复的数据。
algo_unique_if	移除数据区间中相邻的符合指定规则的数据。
algo_unique_copy	移除数据区间中相邻的重复数据，将结果拷贝到新数据区间。
algo_unique_copy_if	移除数据区间中相邻的符合规则的数据，将结果拷贝到新数据区间中。
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，将结果拷贝到新数据区间中。
algo_rotate	交换同一数据区间中的两部分。
algo_rotate_copy	交换同一数据区间中的两部分，将结果拷贝到新数据区间中。
algo_random_shuffle	将数据区间中的数据随机重排。
algo_random_shuffle_if	将数据区间中的数据使用指定的规则随机重排。
algo_random_sample	将源数据区间中的数据随机的抽出填充到目的数据区间中。

algo_random_sample_if	将源数据区间中的数据使用指定的规则随机的抽出填充到目的数据区间中。
algo_random_sample_n	将源数据区间中的数据随机的抽出 n 个数据填充到目的数据区间中。
algo_random_sample_n_if	将源数据区间中的数据使用指定的规则随机的抽出 n 个数据填充到目的区间中。
algo_partition	将数据区间中的数据分为符合规则和不符合规则的两部分。
algo_stable_partition	将数据区间中的数据按照稳定的方式分为符合规则和不符合规则的两部分。

1. 拷贝数据

- algo_copy algo_copy_n algo_copy_backward 将数据从源数据区间拷贝到目的数据区间

1. algo_copy 和 algo_copy_backward 都是将源数据区间中的数据拷贝到目的数据区间。
2. algo_copy 从源数据区间的第一个数据开始逐个拷贝到目的数据区间。
3. algo_copy_backward 从源数据区间的最后一个数据开始逐个拷贝到目的数据区间。
4. algo_copy_n 是将源数据区间中的 n 个数据拷贝到目的数据区间中。
5. 三个算法都返回拷贝后目的数据区间中被拷贝过来的数据的末尾。
6. 用户在使用这个三个算法是要保证目的数据区间足够大。

下面的例子展示了如何使用 algo_copy 和 algo_copy_n 算法：

```
/*
 * copy3.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[])
{
    vector_t* pvec_coll1 = create_vector(int);
    list_t* plist_coll2 = create_list(int);
    list_t* plist_coll3 = create_list(int);
    iterator_t it_pos;
    int i = 0;

    if(pvec_coll1 == NULL || plist_coll2 == NULL || plist_coll3 == NULL)
    {
        return -1;
    }
```

```
    vector_init(pvec_coll1);
    list_init(plist_coll2);
    list_init(plist_coll3);

    for(i = 1; i <= 9; ++i)
    {
        vector_push_back(pvec_coll1, i);
    }

    list_resize(plist_coll2, vector_size(pvec_coll1));
    list_resize(plist_coll3, vector_size(pvec_coll1));

    algo_copy(vector_begin(pvec_coll1), vector_end(pvec_coll1),
        list_begin(plist_coll2));
    algo_copy_n(vector_begin(pvec_coll1), 5, list_begin(plist_coll3));

    for(it_pos = vector_begin(pvec_coll1);
        !iterator_equal(it_pos, vector_end(pvec_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");
    for(it_pos = list_begin(plist_coll2);
        !iterator_equal(it_pos, list_end(plist_coll2));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");
    for(it_pos = list_begin(plist_coll3);
        !iterator_equal(it_pos, list_end(plist_coll3));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    vector_destroy(pvec_coll1);
    list_destroy(plist_coll2);
    list_destroy(plist_coll3);

    return 0;
```

```
}
```

在这个例子中第一次使用 algo_copy 来拷贝数据，第二次使用了 algo_copy_n 拷贝数据，输出的结果如下：

1 2 3 4 5 6 7 8 9

1 2 3 4 5 6 7 8 9

1 2 3 4 5 0 0 0 0

algo_copy 和 algo_copy_backward 都是将源数据区间中的数据拷贝到目的数据区间中去，但是两个算法是有区别的，首先看看下面这个例子：

```
/*
 * copy4.c
 * compile with : -lcstl
 */

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

int main(int argc, char* argv[])
{
    vector_t* pvec_coll1 = create_vector(int);
    list_t* plist_coll2 = create_list(int);
    deque_t* pdeq_coll3 = create_deque(int);
    deque_t* pdeq_coll4 = create_deque(int);
    iterator_t it_pos;
    int i = 0;

    if(pvec_coll1 == NULL || plist_coll2 == NULL ||
       pdeq_coll3 == NULL || pdeq_coll4 == NULL)
    {
        return -1;
    }

    vector_init(pvec_coll1);
    list_init(plist_coll2);
    deque_init(pdeq_coll3);
    deque_init(pdeq_coll4);

    for(i = 0; i <= 8; ++i)
    {
        vector_push_back(pvec_coll1, i);
        list_push_back(plist_coll2, i);
```

```
}
deque_resize(pdeq_coll3, vector_size(pvec_coll1));
deque_resize(pdeq_coll4, vector_size(pvec_coll1));

algo_copy(vector_begin(pvec_coll1), vector_end(pvec_coll1),
    deque_begin(pdeq_coll3));
algo_copy_backward(list_begin(plist_coll2), list_end(plist_coll2),
    deque_end(pdeq_coll4));

printf("coll1: ");
for(it_pos = vector_begin(pvec_coll1);
    !iterator_equal(it_pos, vector_end(pvec_coll1));
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");
printf("coll2: ");
for(it_pos = list_begin(plist_coll2);
    !iterator_equal(it_pos, list_end(plist_coll2));
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");
printf("coll3: ");
for(it_pos = deque_begin(pdeq_coll3);
    !iterator_equal(it_pos, deque_end(pdeq_coll3));
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");
printf("coll4: ");
for(it_pos = deque_begin(pdeq_coll4);
    !iterator_equal(it_pos, deque_end(pdeq_coll4));
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");

algo_copy(iterator_advance(vector_begin(pvec_coll1), 2),
```

```
        iterator_advance(vector_begin(pvec_coll1), 7),
        vector_begin(pvec_coll1));
    algo_copy(iterator_advance(list_begin(plist_coll2), 2),
        iterator_advance(list_begin(plist_coll2), 7),
        iterator_advance(list_begin(plist_coll2), 4));
    algo_copy_backward(iterator_advance(deque_begin(pdeq_coll3), 2),
        iterator_advance(deque_begin(pdeq_coll3), 7),
        iterator_advance(deque_begin(pdeq_coll3), 5));
    algo_copy_backward(iterator_advance(deque_begin(pdeq_coll4), 2),
        iterator_advance(deque_begin(pdeq_coll4), 7),
        deque_end(pdeq_coll4));

    printf("\n");
    printf("coll1: ");
    for(it_pos = vector_begin(pvec_coll1);
        !iterator_equal(it_pos, vector_end(pvec_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");
    printf("coll2: ");
    for(it_pos = list_begin(plist_coll2);
        !iterator_equal(it_pos, list_end(plist_coll2));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");
    printf("coll3: ");
    for(it_pos = deque_begin(pdeq_coll3);
        !iterator_equal(it_pos, deque_end(pdeq_coll3));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");
    printf("coll4: ");
    for(it_pos = deque_begin(pdeq_coll4);
        !iterator_equal(it_pos, deque_end(pdeq_coll4));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
```

```
    }
    printf("\n");

    vector_destroy(pvec_coll1);
    list_destroy(plist_coll2);
    deque_destroy(pdeq_coll3);
    deque_destroy(pdeq_coll4);

    return 0;
}
```

这个程序的输出：

coll1: 0 1 2 3 4 5 6 7 8
coll2: 0 1 2 3 4 5 6 7 8
coll3: 0 1 2 3 4 5 6 7 8
coll4: 0 1 2 3 4 5 6 7 8

coll1: 2 3 4 5 6 5 6 7 8
coll2: 0 1 2 3 2 3 2 3 2
coll3: 6 5 6 5 6 5 6 7 8
coll4: 0 1 2 3 2 3 4 5 6

第一组输出是将 coll1 和 coll2 分别使用 algo_copy 和 algo_copy_backward 算法拷贝到 coll3 和 coll4 中，从输出看来这两个算法的结果是相同的，这是因为两个算法的输入数据区间和输出数据区间是没有重叠的，所以这两个算法的效果是相同的。但是第二组输出就不同了，这一组输出都是输入和输出的数据区间是有重叠的。对于 coll1 和 coll2 来说，我们使用了 algo_copy 进行拷贝，但是输入和输出的数据区间重叠方式不同导致结果不同，coll1 的输出数据区间的末尾和输入数据区间重叠，也就是输出数据区间在输入数据区间的前面，而 coll2 的输出数据区间的开头和输入数据区间重叠，也就是输出数据区间在输入数据区间的后面，它们的拷贝过程如下：

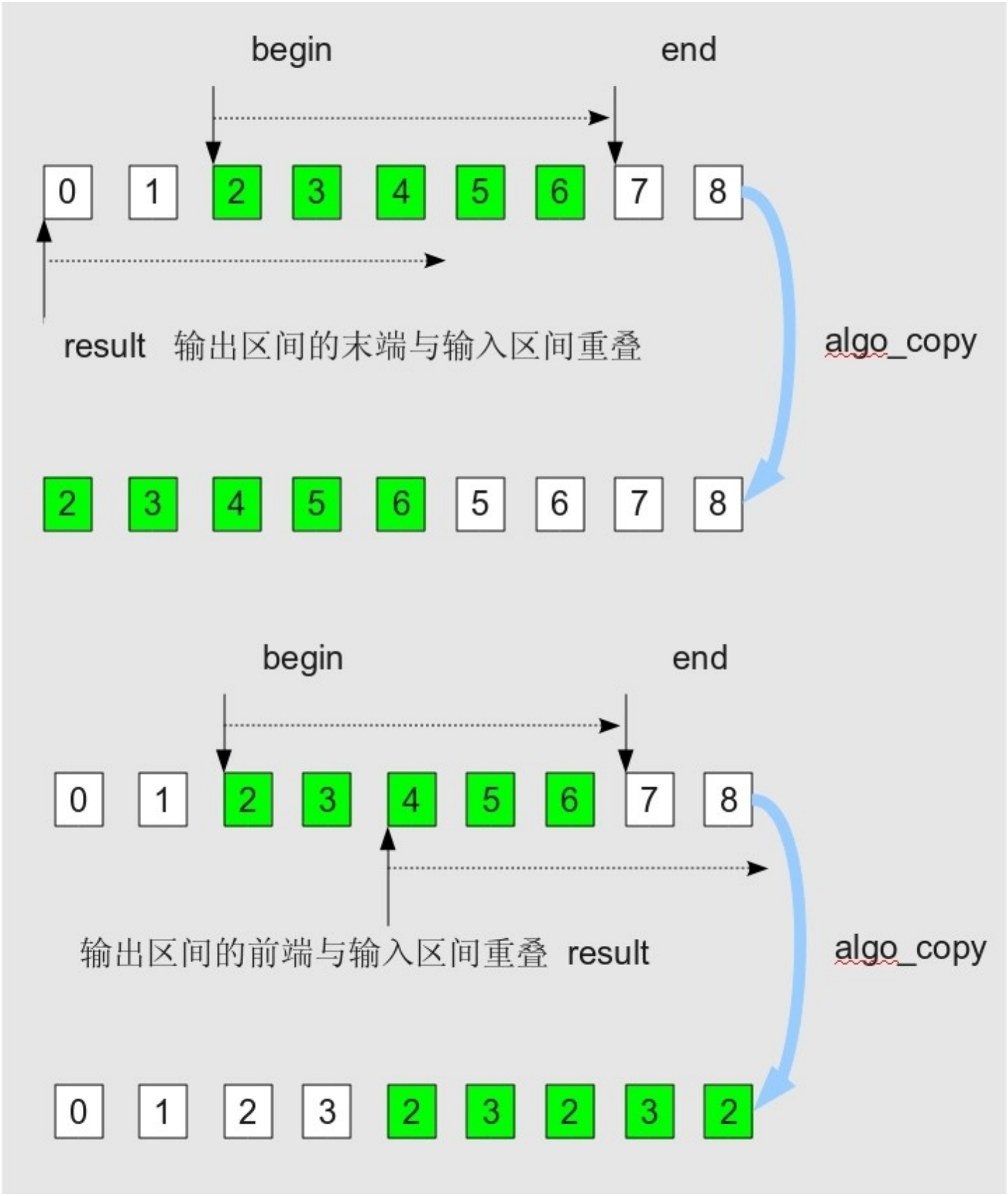

图 6.1 algo_copy 的执行过程

可见，当输出数据区间在输入数据区间的前面的时候，algo_copy 算法执行后获得的结果是正确的，但是输出数据区间在输入数据区间后面的时候，algo_copy 算法在执行的过程中把重叠部分的数据覆盖掉了，导致结果的错误。

对于 coll3 和 coll4 来说，它们的输出数据区间和输入数据区间的重叠情况分别和 coll1 和 coll2 相同，但是使用了算法 algo_copy_backward 执行拷贝，输出结果就与 coll1 和 coll2 不同了：

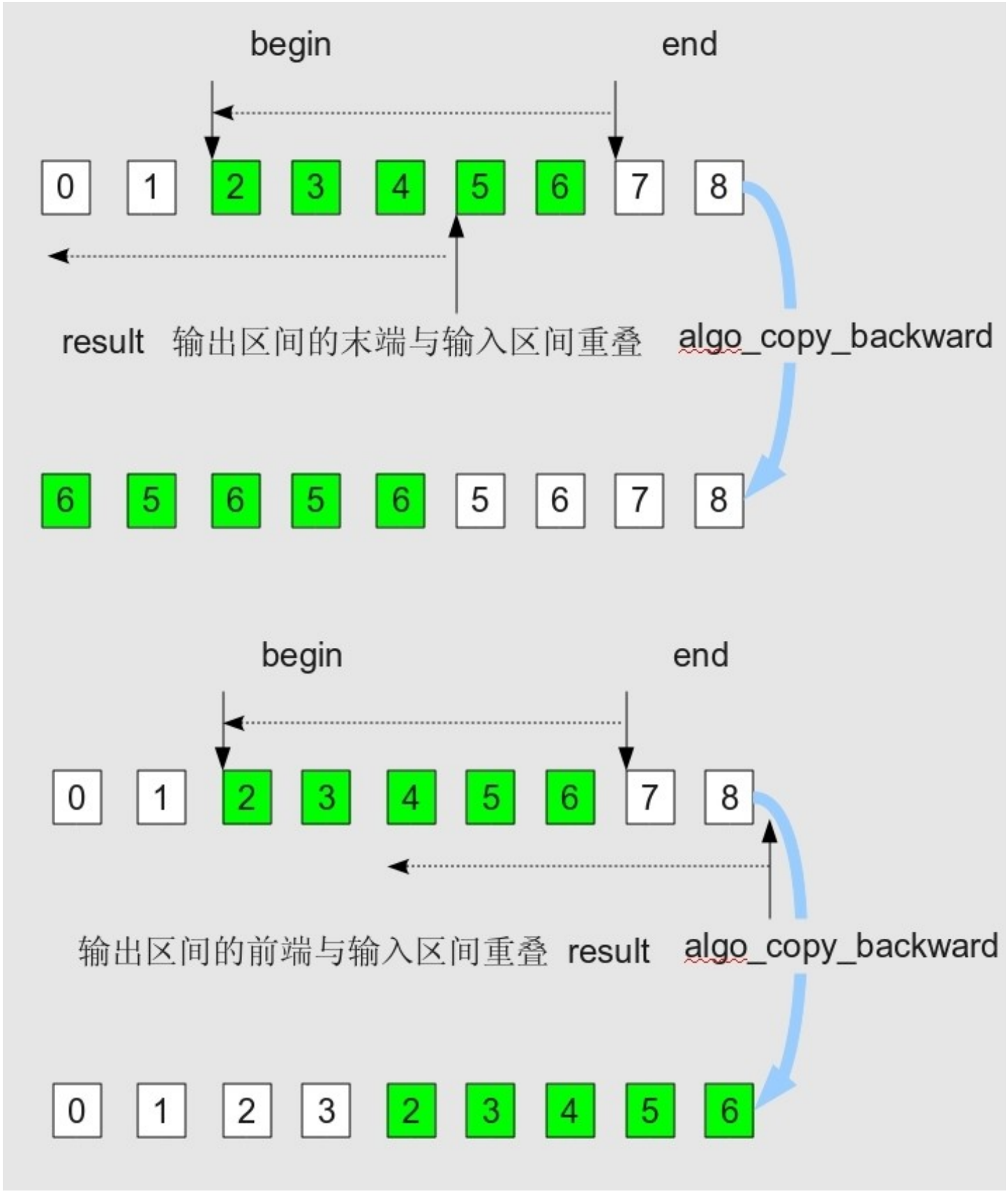

图 6.2 algo_copy_backward 执行过程

使用 algo_copy_backward 的情况与 algo_copy 的情况正好相反，输出数据区间在输入数据区间后面的情况下的结果是正确的。
由上面的例子可以总结出：

1. 输入数据区间和输出数据区间没有重叠的情况下 algo_copy 和 algo_copy_backward 是等效并正确的。
2. 有重叠并且输出数据区间在输入数据区间的前面时使用 algo_copy 是正确的。
3. 有重叠并且输出数据区间在输入数据区间的后面时使用 algo_copy_backward 是正确的。

2. 交换数据

- algo_iter_swap　algo_swap 交换迭代器指向的数据。

下面这个例子展示了如何使用这两个算法：

```
/*
 * swap1.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);
    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);
    }

    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");

    algo_iter_swap(list_begin(plist_coll),
        iterator_next(list_begin(plist_coll)));
    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");

    algo_swap(list_begin(plist_coll),
        iterator_prev(list_end(plist_coll)));
    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;
}
```

程序的输出结果：
1 2 3 4 5 6 7 8 9
2 1 3 4 5 6 7 8 9
9 1 3 4 5 6 7 8 2

- algo_swap 交换数据区间中的数据

1. 交换源数据区间和目的数据区间中的数据。
2. 返回目的数据区间交换后的数据末尾。
3. 用户必须保证用户数据区间足够大。

下面是展示这个算法的例子：

```
/*
 * swap2.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[])
{
    vector_t* pvec_coll1 = create_vector(int);
    list_t* plist_coll2 = create_list(int);
    iterator_t it_pos;
    iterator_t it_end;
    int i = 0;

    if(pvec_coll1 == NULL || plist_coll2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_coll1);
    list_init(plist_coll2);

    for(i = 1; i <= 9; ++i)
    {
        vector_push_back(pvec_coll1, i);
    }
    for(i = 11; i <= 23; ++i)
    {
        list_push_back(plist_coll2, i);
    }

    printf("coll1: ");
    for(it_pos = vector_begin(pvec_coll1);
        !iterator_equal(it_pos, vector_end(pvec_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");
    printf("coll2: ");
    for(it_pos = list_begin(plist_coll2);
        !iterator_equal(it_pos, list_end(plist_coll2));
```

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

    it_end = algo_swap_ranges(vector_begin(pvec_coll1),
        vector_end(pvec_coll1), list_begin(plist_coll2));
    printf("coll1: ");
    for(it_pos = vector_begin(pvec_coll1);
        !iterator_equal(it_pos, vector_end(pvec_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");
    printf("coll2: ");
    for(it_pos = list_begin(plist_coll2);
        !iterator_equal(it_pos, list_end(plist_coll2));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n\n");

    if(!iterator_equal(it_end, list_end(plist_coll2)))
    {
        printf("first element not modified: %d\n",
            *(int*)iterator_get_pointer(it_end));
    }

    vector_destroy(pvec_coll1);
    list_destroy(plist_coll2);

    return 0;
}
```

algo_swap 算法将源数据区间与目的数据区间中相应的数据交换，目的数据区间中剩余的数据不会被修改，算法返回目的数据区间中第一个没有被修改的数据的迭代器。

下面是程序输出的结果：

coll1: 1 2 3 4 5 6 7 8 9

coll2: 11 12 13 14 15 16 17 18 19 20 21 22 23

coll1: 11 12 13 14 15 16 17 18 19

coll2: 1 2 3 4 5 6 7 8 9 20 21 22 23

first element not modified: 20

3. 转换和合并

- algo_transform 是将源数据区间中的数据按照指定的转换函数转换到目的数据区间中

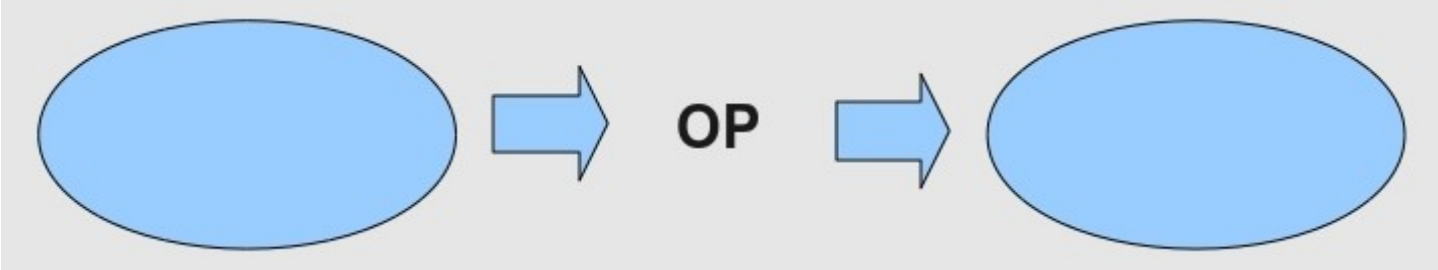

图 6.3 algo_transform 执行过程

1. algo_transform 对于源数据区间中的每一个数据都调用指定的函数产生新数据保存到目的数据区间中。
2. algo_transform 返回目的数据区间被转换数据填充的数据区间的末尾。
3. 代用这必须保证目的数据区间足够大。

下面这个例子展示了如何使用 algo_transform 算法：

```
/*
 * transform2.c
 * compile with : -lcstl
 */

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

static void _multiplies_10(const void* cpv_input, void* pv_output)
{
    *(int*)pv_output = *(int*)cpv_input * 10;
}

int main(int argc, char* argv[])
{
    vector_t* pvec_coll1 = create_vector(int);
    list_t* plist_coll2 = create_list(int);
    iterator_t it_pos;
```

```
    int i = 0;

    if(pvec_coll1 == NULL || plist_coll2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_coll1);
    list_init(plist_coll2);

    for(i = 1; i <= 9; ++i)
    {
        vector_push_back(pvec_coll1, i);
    }
    printf("coll1: ");
    for(it_pos = vector_begin(pvec_coll1);
        !iterator_equal(it_pos, vector_end(pvec_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    /* negate all elements in coll1 */
    algo_transform(vector_begin(pvec_coll1), vector_end(pvec_coll1),
        vector_begin(pvec_coll1), fun_negate_int);
    printf("negated: ");
    for(it_pos = vector_begin(pvec_coll1);
        !iterator_equal(it_pos, vector_end(pvec_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    list_resize(plist_coll2, vector_size(pvec_coll1));
    /* transform elements of coll1 into coll2 with ten times their value */
    algo_transform(vector_begin(pvec_coll1), vector_end(pvec_coll1),
        list_begin(plist_coll2), _multiplies_10);
    printf("coll2: ");
    for(it_pos = list_begin(plist_coll2);
        !iterator_equal(it_pos, list_end(plist_coll2));
        it_pos = iterator_next(it_pos))
```

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

    vector_destroy(pvec_coll1);
    list_destroy(plist_coll2);

    return 0;
}
```

程序的输出结果：

coll1: 1 2 3 4 5 6 7 8 9

negated: -1 -2 -3 -4 -5 -6 -7 -8 -9

coll2: -10 -20 -30 -40 -50 -60 -70 -80 -90

- algo_transform_binary 是将两个源数据区间中的数据通过函数合并到目的数据区间中

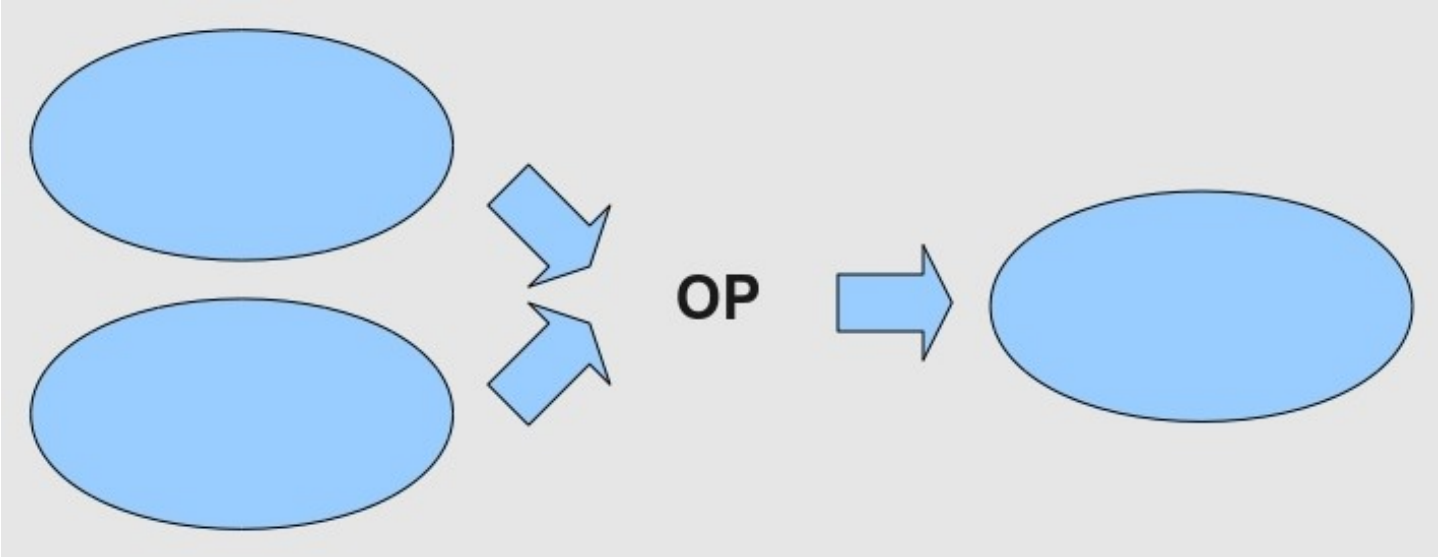

图 6.4 algo_transform_binary 执行过程

1. algo_transform_binary 对两个源数据区间中的每一个数据都调用指定的函数生成新数据并赋值到目的数据区间中。
2. algo_transform_binary 返回目的数据区间中被合并的数据填充的数据区间的末尾。
3. 用户要保证第二个源数据区间要足够大。
4. 用户要保证目的数据区间要足够大。

下面是展示这个算法使用的例子：

```
/*
 * transform3.c
 * compile with : -lcstl
 */
```

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

int main(int argc, char* argv[])
{
    vector_t* pvec_coll1 = create_vector(int);
    list_t* plist_coll2 = create_list(int);
    iterator_t it_pos;
    int i = 0;

    if(pvec_coll1 == NULL || plist_coll2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_coll1);
    list_init(plist_coll2);

    for(i = 1; i <= 9; ++i)
    {
        vector_push_back(pvec_coll1, i);
    }
    printf("coll1: ");
    for(it_pos = vector_begin(pvec_coll1);
        !iterator_equal(it_pos, vector_end(pvec_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    /* square each element */
    algo_transform_binary(vector_begin(pvec_coll1), vector_end(pvec_coll1),
        vector_begin(pvec_coll1), vector_begin(pvec_coll1), fun_multiplies_int);
    printf("squared: ");
    for(it_pos = vector_begin(pvec_coll1);
        !iterator_equal(it_pos, vector_end(pvec_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
```

```
    }
    printf("\n");

    list_resize(plist_coll2, vector_size(pvec_coll1));
    algo_transform_binary(vector_begin(pvec_coll1), vector_end(pvec_coll1),
        vector_begin(pvec_coll1), list_begin(plist_coll2), fun_plus_int);
    printf("coll2: ");
    for(it_pos = list_begin(plist_coll2);
        !iterator_equal(it_pos, list_end(plist_coll2));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    vector_destroy(pvec_coll1);
    list_destroy(plist_coll2);

    return 0;
}
```

程序的输出结果：
coll1: 1 2 3 4 5 6 7 8 9
squared: 1 4 9 16 25 36 49 64 81
coll2: 2 8 18 32 50 72 98 128 162

4. 替换数据

- algo_replace algo_replace_if 替换数据区间中符合规则的数据

1. algo_replace 替换数据区间中与指定数据相等的数据。
2. algo_replace_if 替换数据区间中符合指定规则的数据。
3. algo_replace_if 使用的函数不能修改数据。

下面展示了 algo_replace 和 algo_replace_if 的用法：

```
/*
 * replace1.c
 * compile with : -lcstl
 */

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

```
static void _less_5(const void* cpv_input, void* pv_output)
{
    *(bool_t*)pv_output = *(int*)cpv_input < 5 ? true : false;
}

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 = 2; i <= 7; ++i)
    {
        list_push_back(plist_coll, i);
    }
    for(i = 4; i <= 9; ++i)
    {
        list_push_back(plist_coll, i);
    }

    printf("coll: ");
    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");

    /* replace all elements from value 6 to value 42 */
    algo_replace(list_begin(plist_coll), list_end(plist_coll), 6, 42);
    printf("coll: ");
    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");

    /* replace all elemens from value less than 5 to value 0 */
    algo_replace_if(list_begin(plist_coll), list_end(plist_coll), _less_5, 0);
    printf("coll: ");
    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;
}
```

程序的输出结果：

coll: 2 3 4 5 6 7 4 5 6 7 8 9

coll: 2 3 4 5 42 7 4 5 42 7 8 9

coll: 0 0 0 5 42 7 0 5 42 7 8 9

- algo_replace_copy　algo_replace_copy_if 将源数据区间中符合规则的数据替换并将结果拷贝到目的数据区间中

1. algo_replace_copy 将源数据区间中与指定数据相等的数据替换并将结果拷贝到目的数据区间中。
2. algo_replace_copy_if 将源数据区间中符合指定规则的数据替换并将结果拷贝到目的数据区间中。
3. 两个算法都返回目的数据区间中拷贝过来的数据区间的末尾。
4. algo_replace_copy_if 中使用的函数不能修改源数据区间中的数据。
5. 用户要保证目的数据区间足够大。

下面展示了 algo_replace_copy 和 algo_replace_copy_if 两个算法的使用：

```
/*
 * replace2.c
 * compile with : -lcstl
 */

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

static void _less_5(const void* cpv_input, void* pv_output)
{
```

```
    *(bool_t*)pv_output = *(int*)cpv_input < 5 ? true : false;
}

int main(int argc, char* argv[])
{
    list_t* plist_coll1 = create_list(int);
    list_t* plist_coll2 = create_list(int);
    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 = 2; i <= 7; ++i)
    {
        list_push_back(plist_coll1, i);
    }
    for(i = 4; i <= 9; ++i)
    {
        list_push_back(plist_coll1, i);
    }

    printf("coll1: ");
    for(it_pos = list_begin(plist_coll1);
        !iterator_equal(it_pos, list_end(plist_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    list_resize(plist_coll2, list_size(plist_coll1));
    /* replace all elements of coll1 from value 6 to value 42 and copy to coll2*/
    algo_replace_copy(list_begin(plist_coll1), list_end(plist_coll1),
        list_begin(plist_coll2), 6, 42);
    printf("coll2: ");
    for(it_pos = list_begin(plist_coll2);
        !iterator_equal(it_pos, list_end(plist_coll2));
```

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

    /*
     * replace all elemens of coll1 from value less than 5 to value 0
     * and copy to coll2
     */
    algo_replace_copy_if(list_begin(plist_coll1), list_end(plist_coll1),
        list_begin(plist_coll2), _less_5, 0);
    printf("coll2: ");
    for(it_pos = list_begin(plist_coll2);
        !iterator_equal(it_pos, list_end(plist_coll2));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    list_destroy(plist_coll1);
    list_destroy(plist_coll2);

    return 0;
}
```

程序的输出结果：
coll1: 2 3 4 5 6 7 4 5 6 7 8 9
coll2: 2 3 4 5 42 7 4 5 42 7 8 9
coll2: 0 0 0 5 6 7 0 5 6 7 8 9

5. 赋值新数据

- algo_fill　algo_fill_n 向数据区间中填充数据

1. algo_fill 向数据区间中填充指定的数据。
2. algo_fill_n 向指定的数据区间填充 n 个指定的数据。
3. algo_fill_n 中的数据区间必须足够大。

下面是 algo_fill 和 algo_fill_n 的使用实例：

```
/*
 * fill1.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(char*);
    iterator_t it_pos;

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

    list_init_n(plist_coll, 10);

    algo_fill(list_begin(plist_coll), list_end(plist_coll), "hello");
    printf("coll: ");
    for(it_pos = list_begin(plist_coll);
        !iterator_equal(it_pos, list_end(plist_coll));
        it_pos = iterator_next(it_pos))
    {
        printf("%s ", (char*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    algo_fill_n(list_begin(plist_coll), list_size(plist_coll) - 2, "again");
    printf("coll: ");
    for(it_pos = list_begin(plist_coll);
        !iterator_equal(it_pos, list_end(plist_coll));
        it_pos = iterator_next(it_pos))
    {
        printf("%s ", (char*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    algo_fill(iterator_next(list_begin(plist_coll)),
        iterator_prev(list_end(plist_coll)), "mmmm");
    printf("coll: ");
    for(it_pos = list_begin(plist_coll);
```

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

    list_destroy(plist_coll);

    return 0;
}
```

程序的输出结果：

coll: hello hello hello hello hello hello hello hello hello hello

coll: again again again again again again again again hello hello

coll: again mmmm mmmm mmmm mmmm mmmm mmmm mmmm mmmm hello

algo_generate　algo_generate_n 使用指定函数产生的数据填充数据区间。

algo_generate 使用指定函数产生的数据填充数据区间。

algo_generate_n 使用指定函数产生的数据填充数据区间中的 n 个数据。

algo_gerenate_n 要保证数据区间足够大。

下面是使用 algo_genenrate 和 algo_generate_n 的实例：

```
/*
 * generate1.c
 * compile with : -lcstl
 */

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

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

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

    list_init_elem(plist_coll, 10, 100);
    printf("coll: ");
```

```
    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");

    algo_generate(list_begin(plist_coll),
        list_end(plist_coll), fun_random_number);
    printf("coll: ");
    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");

    algo_generate_n(list_begin(plist_coll),
        list_size(plist_coll) - 3, fun_random_number);
    printf("coll: ");
    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;
}
```

程序输出结果：

coll: 100 100 100 100 100 100 100 100 100 100

coll: 31 60 77 17 54 96 13 43 59 86

coll: 18 38 25 11 15 0 12 43 59 86

6. 移除数据

接下来的算法是将数据区间中符合规则的数据移除，为什么叫做移除而不是删除呢，这是因为它们并不是真正的把数据区间中的数据删掉，而是使用后面的数据覆盖掉应该删掉的数据。

- algo_remove algo_remove_if 移除数据区间中符合指定规则的数据

1. algo_remove 移除数据区间中所有与指定数据相等的数据。
2. algo_remove_if 移除了数据区间中所有满足指定规则的数据。
3. 两个算法都返回移除数据后数据区间的新末尾。
4. 两个算法都是使用后面的数据来覆盖要移除的数据，而并不是将数据真的删除。
5. 数据区间中没有被移除的数据的内容和顺序不会改变。
6. algo_remove_if 算法使用的函数不能修改数据。
7. 有些容器提供了删除的功能，如果要删除这些容器中的数据请使用容器提供的操作函数，它们更高效。

下面的程序展示了如何使用 algo_remove 和 algo_remove_if 算法：

```
/*
 * remove5.c
 * compile with : -lcstl
 */

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

static void _less_4(const void* cpv_input, void* pv_output)
{
    *(bool_t*)pv_output = *(int*)cpv_input < 4 ? true : false;
}

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

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

    list_init(plist_coll);
```

```
    for(i = 2; i <= 6; ++i)
    {
        list_push_back(plist_coll, i);
    }
    for(i = 4; i <= 9; ++i)
    {
        list_push_back(plist_coll, i);
    }
    for(i = 1; i <= 7; ++i)
    {
        list_push_back(plist_coll, i);
    }

    printf("coll: ");
    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 5
     */
    it_end = algo_remove(list_begin(plist_coll), list_end(plist_coll), 5);

    printf("size not changed: ");
    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");

    /* erase the "removed" elements in the container */
    list_erase_range(plist_coll, it_end, list_end(plist_coll));
    printf("size changed: ");
    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 less than 4
     */
    list_erase_range(plist_coll,
        algo_remove_if(list_begin(plist_coll), list_end(plist_coll), _less_4),
        list_end(plist_coll));
    printf("< 4 removed: ");
    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;
}
```

输出结果：

coll: 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

< 4 removed: 4 6 4 6 7 8 9 4 6 7

- algo_remove_copy　algo_remove_copy_if 移除数据区间中符合指定规则的数据并将数据拷贝到目的数据区间

1. algo_remove_copy 移除数据区间中所有与指定数据相等的数据并将结果拷贝到目的数据区间中。
2. algo_remove_copy_if 移除数据区间中所有符合指定规则的数据并将结果拷贝到目的数据区间中。
3. 两个算法都返回目的数据区间中的新的数据区间结尾。
4. 用户要保证目的数据区间要足够大。
5. algo_remove_copy_if 使用的函数不能更改数据。

下面的例子展示了如何使用 algo_remove_copy 和 algo_remove_copy_if：

```
/*
 * remove6.c
 * compile with : -lcstl
 */

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

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

static void _greater_4(const void* cpv_input, void* pv_output)
{
    *(bool_t*)pv_output = *(int*)cpv_input > 4 ? true : false;
}

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

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

    list_init(plist_coll1);
    list_init(plist_coll2);

    for(i = 1; i <= 6; ++i)
    {
        list_push_back(plist_coll1, i);
    }
    for(i = 1; i <= 9; ++i)
    {
        list_push_back(plist_coll1, i);
    }

    list_resize(plist_coll2, list_size(plist_coll1));

    printf("coll1: ");
    for(it_pos = list_begin(plist_coll1);
        !iterator_equal(it_pos, list_end(plist_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");
```

```
    /* remove elements that with value 3 */
    it_end = algo_remove_copy(list_begin(plist_coll1), list_end(plist_coll1),
        list_begin(plist_coll2), 3);
    printf("coll2: ");
    for(it_pos = list_begin(plist_coll2);
        !iterator_equal(it_pos, it_end);
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    /* remove elements that with value greater than 4 */
    it_end = algo_remove_copy_if(list_begin(plist_coll1), list_end(plist_coll1),
        list_begin(plist_coll2), _greater_4);
    printf("coll2: ");
    for(it_pos = list_begin(plist_coll2);
        !iterator_equal(it_pos, it_end);
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    list_destroy(plist_coll1);
    list_destroy(plist_coll2);

    return 0;
}
```

程序的结果：
coll1: 1 2 3 4 5 6 1 2 3 4 5 6 7 8 9
coll2: 1 2 4 5 6 1 2 4 5 6 7 8 9
coll2: 1 2 3 4 1 2 3 4

7. 数据唯一

- algo_unique　algo_unique_if 移除相邻的并且满足指定规则的数据

1. algo_unique 移除数据区间中相邻的并且相等的数据。
2. algo_unique_if 移除数据区间中相邻并且符合指定规则的数据。
3. 两个算法都是是使用后面的数据覆盖要被移除的数据，而不是直接删掉。

4. 两个算法都返回移除数据后新的数据区间的末尾。
5. algo_remove_if 使用的函数不能修改数据。
6. 数据区间中没有被移除的数据的顺序和内容不变。
7. 有些容器提供了相同的功能，如果要使用同样的功能，请优先考虑容器提供的操作函数。

下面的例子展示了如何使用 algo_remove 和 algo_remove_if 算法：

```
/*
 * unique1.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);
    iterator_t it_pos;
    iterator_t it_end;
    int i = 0;

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

    deque_init(pdeq_coll);

    deque_push_back(pdeq_coll, 1);
    deque_push_back(pdeq_coll, 4);
    deque_push_back(pdeq_coll, 4);
    deque_push_back(pdeq_coll, 6);
    deque_push_back(pdeq_coll, 1);
    deque_push_back(pdeq_coll, 2);
    deque_push_back(pdeq_coll, 2);
    deque_push_back(pdeq_coll, 3);
    deque_push_back(pdeq_coll, 1);
    deque_push_back(pdeq_coll, 6);
    deque_push_back(pdeq_coll, 6);
    deque_push_back(pdeq_coll, 6);
    deque_push_back(pdeq_coll, 5);
    deque_push_back(pdeq_coll, 7);
```

```
    deque_push_back(pdeq_coll, 5);
    deque_push_back(pdeq_coll, 4);
    deque_push_back(pdeq_coll, 4);

    printf("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");

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

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

    deque_destroy(pdeq_coll);
```

```
    return 0;
}
```

程序结果：

coll: 1 4 4 6 1 2 2 3 1 6 6 6 5 7 5 4 4

coll: 1 4 6 1 2 3 1 6 5 7 5 4

coll: 1 4 6 1 2 3 1 6 5 7 5 4 5 7 5 4 4

coll: 1 4 6 6 7 7

- algo_unique_copy　algo_unique_copy_if 将数据区间中相邻的符合规则的数据移除并拷贝到目的数据区间中

1. 两个算法都返回拷贝到目的数据区间的末尾。
2. 用户一定要保证目的数据区间足够大。

下面是使用 algo_unique_copy 和 algo_unique_copy_if 的例子：

```
/*
 * unique2.c
 * compile with : -lcstl
 */

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

static void _difference_one(const void* cpv_first,
    const void* cpv_second, void* pv_output)
{
    *(bool_t*)pv_output =
        *(int*)cpv_first + 1 == *(int*)cpv_second ||
        *(int*)cpv_first - 1 == *(int*)cpv_second ?
        true : false;
}

int main(int argc, char* argv[])
{
    deque_t* pdeq_coll1 = create_deque(int);
    deque_t* pdeq_coll2 = create_deque(int);
    deque_t* pdeq_coll3 = create_deque(int);
    deque_t* pdeq_coll4 = create_deque(int);
    iterator_t it_pos;
    iterator_t it_end;
    int i = 0;

    if(pdeq_coll1 == NULL || pdeq_coll2 == NULL ||
```

```
   pdeq_coll3 == NULL || pdeq_coll4 == NULL)
{
    return -1;
}

deque_init(pdeq_coll1);
deque_init(pdeq_coll2);
deque_init(pdeq_coll3);
deque_init(pdeq_coll4);

deque_push_back(pdeq_coll1, 1);
deque_push_back(pdeq_coll1, 4);
deque_push_back(pdeq_coll1, 4);
deque_push_back(pdeq_coll1, 6);
deque_push_back(pdeq_coll1, 1);
deque_push_back(pdeq_coll1, 2);
deque_push_back(pdeq_coll1, 2);
deque_push_back(pdeq_coll1, 3);
deque_push_back(pdeq_coll1, 1);
deque_push_back(pdeq_coll1, 6);
deque_push_back(pdeq_coll1, 6);
deque_push_back(pdeq_coll1, 6);
deque_push_back(pdeq_coll1, 5);
deque_push_back(pdeq_coll1, 7);
deque_push_back(pdeq_coll1, 5);
deque_push_back(pdeq_coll1, 4);
deque_push_back(pdeq_coll1, 4);
deque_assign(pdeq_coll3, pdeq_coll1);
deque_resize(pdeq_coll2, deque_size(pdeq_coll1));
deque_resize(pdeq_coll4, deque_size(pdeq_coll1));

printf("coll1: ");
for(it_pos = deque_begin(pdeq_coll1);
    !iterator_equal(it_pos, deque_end(pdeq_coll1));
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");

it_end = algo_unique_copy(deque_begin(pdeq_coll1),
    deque_end(pdeq_coll1), deque_begin(pdeq_coll2));
printf("coll2: ");
```

```
    for(it_pos = deque_begin(pdeq_coll2);
        !iterator_equal(it_pos, it_end);
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n\n");

    printf("coll3: ");
    for(it_pos = deque_begin(pdeq_coll3);
        !iterator_equal(it_pos, deque_end(pdeq_coll3));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");
    it_end = algo_unique_copy_if(deque_begin(pdeq_coll3), deque_end(pdeq_coll3),
        deque_begin(pdeq_coll4), _difference_one);
    printf("coll4: ");
    for(it_pos = deque_begin(pdeq_coll4);
        !iterator_equal(it_pos, it_end);
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    deque_destroy(pdeq_coll1);
    deque_destroy(pdeq_coll2);
    deque_destroy(pdeq_coll3);
    deque_destroy(pdeq_coll4);

    return 0;
}
```

程序输出结果：

coll1: 1 4 4 6 1 2 2 3 1 6 6 6 5 7 5 4 4

coll2: 1 4 6 1 2 3 1 6 5 7 5 4

coll3: 1 4 4 6 1 2 2 3 1 6 6 6 5 7 5 4 4

coll4: 1 4 4 6 1 3 1 6 6 6 4 4

8. 逆序

- algo_reverse　algo_reverse_copy 将数据区间中的数据逆序

1. algo_reverse 将数据区间中的数据逆序。
2. algo_reverse_copy 将数据区间中的数据逆序并将结果拷贝的目的数据区间。
3. 用户必须保证 algo_reverse_copy 的目的数据区间足够大。
4. 有些容器提供了逆序功能，如果要使用该功能请首先考虑容器提供的算法。

下面的这两个算法的例子：

```
/*
 * reverse1.c
 * compile with : -lcstl
 */

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

int main(int argc, char* argv[])
{
    vector_t* pvec_coll1 = create_vector(int);
    vector_t* pvec_coll2 = create_vector(int);
    iterator_t it_pos;
    iterator_t it_end;
    int i = 0;

    if(pvec_coll1 == NULL || pvec_coll2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_coll1);
    vector_init(pvec_coll2);

    for(i = 1; i <= 9; ++i)
    {
        vector_push_back(pvec_coll1, i);
    }
    printf("coll1: ");
    for(it_pos = vector_begin(pvec_coll1);
        !iterator_equal(it_pos, vector_end(pvec_coll1));
        it_pos = iterator_next(it_pos))
```

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

    algo_reverse(vector_begin(pvec_coll1), vector_end(pvec_coll1));
    printf("after reversing coll1: ");
    for(it_pos = vector_begin(pvec_coll1);
        !iterator_equal(it_pos, vector_end(pvec_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    vector_resize(pvec_coll2, vector_size(pvec_coll1));
    it_end = algo_reverse_copy(iterator_next_n(vector_begin(pvec_coll1), 3),
        iterator_prev(vector_end(pvec_coll1)), vector_begin(pvec_coll2));
    printf("after reversing coll2: ");
    for(it_pos = vector_begin(pvec_coll2);
        !iterator_equal(it_pos, it_end);
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    vector_destroy(pvec_coll1);
    vector_destroy(pvec_coll2);

    return 0;
}
```

输出结果：
coll1: 1 2 3 4 5 6 7 8 9
after reversing coll1: 9 8 7 6 5 4 3 2 1
after reversing coll2: 2 3 4 5 6

9. 数据轮换

- algo_rotate　algo_rotate_copy 是将数据区间的两部分数据调换

1. algo_rotate 将数据区间中的两部分调换，返回新的分界线迭代器。
2. algo_rotate_copy 将数据区间两部分调换并拷贝到目的数据区间中，返回目的数据区间的新末尾。
3. 用户必须保证 algo_rotate_copy 目的数据区间足够大。

下面的例子展示了如何使用 algo_rotate 和 algo_rotate_copy：

```
/*
 * rotate1.c
 * compile with : -lcstl
 */

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

int main(int argc, char* argv[])
{
    vector_t* pvec_coll1 = create_vector(int);
    vector_t* pvec_coll2 = create_vector(int);
    iterator_t it_pos;
    iterator_t it_end;
    int i = 0;

    if(pvec_coll1 == NULL || pvec_coll2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_coll1);
    vector_init(pvec_coll2);

    for(i = 1; i <= 9; ++i)
    {
        vector_push_back(pvec_coll1, i);
    }
    vector_resize(pvec_coll2, vector_size(pvec_coll1));

    printf("coll1: ");
    for(it_pos = vector_begin(pvec_coll1);
        !iterator_equal(it_pos, vector_end(pvec_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");
```

```
    algo_rotate(vector_begin(pvec_coll1),
        iterator_next_n(vector_begin(pvec_coll1), 3),
        vector_end(pvec_coll1));
    printf("coll1: ");
    for(it_pos = vector_begin(pvec_coll1);
        !iterator_equal(it_pos, vector_end(pvec_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    algo_rotate(vector_begin(pvec_coll1),
        iterator_prev_n(vector_end(pvec_coll1), 2),
        vector_end(pvec_coll1));
    printf("coll1: ");
    for(it_pos = vector_begin(pvec_coll1);
        !iterator_equal(it_pos, vector_end(pvec_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    algo_rotate_copy(vector_begin(pvec_coll1),
        algo_find(vector_begin(pvec_coll1), vector_end(pvec_coll1), 6),
        vector_end(pvec_coll1), vector_begin(pvec_coll2));
    printf("coll2: ");
    for(it_pos = vector_begin(pvec_coll2);
        !iterator_equal(it_pos, vector_end(pvec_coll2));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    vector_destroy(pvec_coll1);
    vector_destroy(pvec_coll2);

    return 0;
}
```

程序结果：

coll1: 1 2 3 4 5 6 7 8 9
coll1: 4 5 6 7 8 9 1 2 3
coll1: 2 3 4 5 6 7 8 9 1
coll2: 6 7 8 9 1 2 3 4 5

10. 随机算法

- algo_random_shuffle　algo_random_shuffle_if将数据区间中的数据随机重排

1. 两个算法都是将数据区间中的数据顺序打乱，但是不修改数据内容。
2. algo_random_shuffle_if使用指定的随机数生成算法，但是不能修改数据。

下面是这两个算法的例子：

```
/*
 * random1.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);
    iterator_t it_pos;
    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);
    }
    printf("coll: ");
    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");

    algo_random_shuffle(vector_begin(pvec_coll), vector_end(pvec_coll));
    printf("shuffle: ");
    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");

    algo_sort(vector_begin(pvec_coll), vector_end(pvec_coll));
    printf("sort: ");
    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");

    algo_random_shuffle_if(vector_begin(pvec_coll),
        vector_end(pvec_coll), fun_random_number);
    printf("shuffle: ");
    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;
}
```

输出结果：

coll: 1 2 3 4 5 6 7 8 9
shuffle: 8 5 7 1 9 6 4 2 3
sort: 1 2 3 4 5 6 7 8 9
shuffle: 1 9 8 2 7 6 3 5 4

- algo_random_sample　algo_random_sample_if　algo_random_sample_n　algo_random_sample_n_if 随机抽样

1. 随机抽样是在源数据区间中随机抽取 n 个数据样本到目的数据区间中，保证不会重复抽样。
2. 用户可以指定随机数生成函数。
3. 使用 algo_random_sample_n 和 algo_random_sample_n_if 算法，用户必须保证目的数据区间足够大。
4. 四个算法都返回目的数据区间中抽取的数据区间的末尾。

下面是四个算法的例子：

```
/*
 * random2.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_coll1 = create_vector(int);
    vector_t* pvec_coll2 = create_vector(int);
    iterator_t it_pos;
    iterator_t it_end;
    int i = 0;

    if(pvec_coll1 == NULL || pvec_coll2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_coll1);
    vector_init_n(pvec_coll2, 10);

    for(i = 1; i <= 19; ++i)
    {
        vector_push_back(pvec_coll1, i);
    }
    printf("coll1: ");
    for(it_pos = vector_begin(pvec_coll1);
```

```
        !iterator_equal(it_pos, vector_end(pvec_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    algo_random_sample(vector_begin(pvec_coll1), vector_end(pvec_coll1),
        vector_begin(pvec_coll2), vector_end(pvec_coll2));
    printf("coll2: ");
    for(it_pos = vector_begin(pvec_coll2);
        !iterator_equal(it_pos, vector_end(pvec_coll2));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    algo_random_sample_if(vector_begin(pvec_coll1), vector_end(pvec_coll1),
        vector_begin(pvec_coll2), vector_end(pvec_coll2), fun_random_number);
    printf("coll2: ");
    for(it_pos = vector_begin(pvec_coll2);
        !iterator_equal(it_pos, vector_end(pvec_coll2));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    it_end = algo_random_sample_n(vector_begin(pvec_coll1),
        vector_end(pvec_coll1), vector_begin(pvec_coll2), 5);
    printf("coll2: ");
    for(it_pos = vector_begin(pvec_coll2);
        !iterator_equal(it_pos, it_end);
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

                    it_end     =     algo_random_sample_n_if(vector_begin(pvec_coll1),
vector_end(pvec_coll1),
        vector_begin(pvec_coll2), 8, fun_random_number);
```

```
    printf("coll2: ");
    for(it_pos = vector_begin(pvec_coll2);
        !iterator_equal(it_pos, it_end);
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    vector_destroy(pvec_coll1);
    vector_destroy(pvec_coll2);

    return 0;
}
```

程序的输出：

coll1: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

coll2: 1 2 3 4 5 15 19 12 9 10

coll2: 15 2 3 4 5 6 19 8 17 10

coll2: 1 2 7 12 17

coll2: 4 7 8 10 14 15 17 18

11. 数据划分

- algo_partition　algo_stable_partition 将符合规则的数据放在数据区间的前部，其余的数据放在后部

1. 两个算法都将符合指定规则的数据放到数据区间的前部，其余的数据放在区间的后部。
2. 两个算法都返回两部分数据的边界迭代器。
3. algo_stable_partition 不改变每一个部分中数据的原始顺序。

下面是两个算法的例子：

```
/*
 * partition1.c
 * compile with : -lcstl
 */

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

static void _is_even(const void* cpv_input, void* pv_output)
{
    *(bool_t*)pv_output = *(int*)cpv_input % 2 == 0 ? true : false;
```

```
}

int main(int argc, char* argv[])
{
    vector_t* pvec_coll1 = create_vector(int);
    vector_t* pvec_coll2 = create_vector(int);
    iterator_t it_pos;
    iterator_t it_odd;
    int i = 0;

    if(pvec_coll1 == NULL || pvec_coll2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_coll1);
    vector_init(pvec_coll2);

    for(i = 1; i <= 9; ++i)
    {
        vector_push_back(pvec_coll1, i);
        vector_push_back(pvec_coll2, i);
    }
    printf("coll1: ");
    for(it_pos = vector_begin(pvec_coll1);
        !iterator_equal(it_pos, vector_end(pvec_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");
    printf("coll2: ");
    for(it_pos = vector_begin(pvec_coll2);
        !iterator_equal(it_pos, vector_end(pvec_coll2));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    printf("partition:\n");
    it_odd = algo_partition(vector_begin(pvec_coll1),
        vector_end(pvec_coll1), _is_even);
```

```
    printf("coll1: ");
    for(it_pos = vector_begin(pvec_coll1);
        !iterator_equal(it_pos, vector_end(pvec_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");
    printf("first odd: %d\n", *(int*)iterator_get_pointer(it_odd));
    it_odd = algo_stable_partition(vector_begin(pvec_coll2),
        vector_end(pvec_coll2), _is_even);
    printf("coll2: ");
    for(it_pos = vector_begin(pvec_coll2);
        !iterator_equal(it_pos, vector_end(pvec_coll2));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");
    printf("first odd: %d\n", *(int*)iterator_get_pointer(it_odd));

    vector_destroy(pvec_coll1);
    vector_destroy(pvec_coll2);

    return 0;
}
```

输出结果：
coll1: 1 2 3 4 5 6 7 8 9
coll2: 1 2 3 4 5 6 7 8 9
partition:
coll1: 8 2 6 4 5 3 7 1 9
first odd: 5
coll2: 2 4 6 8 1 3 5 7 9
first odd: 1

第四节 排序算法

libcstl 提供了多种排序算法，这些算法不仅包括将整个数据区间中的数据排序，还包括部分排序，还有与排序相关的算法。有些容器提供了排序操作函数，如果要在这些容器上实现排序请优先考虑容器提供的操作函数。有些容器本身就是有序的，如关联容器，是不允许将排序算法应用到这样的容器上的。下面列出了 libcstl 提供的所有排序算法：

algo_sort	将数据区间中的数据按照默认的比较规则排序。
algo_sort_if	将数据区间中的数据按照指定的比较规则排序。
algo_stable_sort	将数据区间中的数据按照默认比较规则进行稳定排序。
algo_stable_sort_if	将数据区间中的数据按照指定的比较规则进行稳定排序。
algo_partial_sort	将数据区间中的数据按照默认的比较规则进行部分排序。
algo_partial_sort_if	将数据区间中的数据按照指定的比较规则进行部分排序。
algo_partial_sort_copy	将数据区间按照默认比较规则部分排序，将排序结果拷贝到目的区间。
algo_partial_sort_copy_if	将数据区间按照指定比较规则部分排序，将排序结果拷贝到目的区间。
algo_is_sorted	判断数据区间是否是使用默认比较规则排序的。
algo_is_sorted_if	判断数据区间是否是使用指定比较规则排序的。
algo_nth_element	将数据区间按照默认的比较规则分成两部分, 第 n 个数据与排序后的结果相同。
algo_nth_element_if	将数据区间按照指定的比较规则分成两部分, 第 n 个数据与排序后的结果相同。
algo_lower_bound	返回使用默认比较规则排序的数据区间中第一个与指定数据相等的迭代器。
algo_lower_bound_if	返回使用指定比较规则排序的数据区间中第一个与指定数据相等的迭代器。
algo_upper_bound	返回使用默认比较规则排序的数据区间中第一个大于指定数据的迭代器。
algo_upper_bound_if	返回使用指定比较规则排序的数据区间中第一个大于指定数据的迭代器。
algo_equal_range	返回使用默认比较规俄排序的数据区间中包含指定数据的范围。
algo_equal_range_if	返回使用指定比较规则排序的数据区间中包含指定数据的范围。
algo_binary_search	在使用默认比较规则排序的数据区间中实现二分查找。
algo_binary_search_if	在使用指定比较规则排序的数据区间中实现二分查找。
algo_merge	合并两个使用默认比较规则排序的数据区间。
algo_merge_if	合并两个使用指定比较规则排序的数据区间。
algo_inplace_merge	合并同一数据区间中使用默认比较规则排序的两部分。
algo_inplace_merge_if	合并同一数据区间中使用指定比较规则排序的两部分。
algo_includes	判断两个使用默认比较规则排序的数据区间第一个是否包含第二个。
algo_includes_if	判断两个使用指定比较规则排序的数据区间第一个是否包含第二个。
algo_set_union	将两个使用默认比较规则排序的数据区间取并集。
algo_set_union_if	将两个使用指定比较规则排序的数据区间取并集。
algo_set_intersection	将两个使用默认比较规则排序的数据区间取交集。
algo_set_intersection_if	将两个使用指定比较规则排序的数据区间取交集。
algo_set_difference	将两个使用默认比较规则排序的数据区间取差集。
algo_set_difference_if	将两个使用指定比较规则排序的数据区间取差集。

algo_set_symmetric_difference	将两个使用默认比较规则排序的数据区间取对称差集。
algo_set_symmetric_difference_if	将两个使用指定比较规则排序的数据区间取对称差集。
algo_push_heap	向使用默认比较规则的堆中插入一个数据。
algo_push_heap_if	向使用指定比较规则的堆中插入一个数据。
algo_pop_heap	将使用默认比较规则的堆中优先级最高的数据删除。
algo_pop_heap_if	将使用指定比较规则的堆中优先级最高的数据删除。
algo_make_heap	将数据区间变成使用默认比较规则的堆。
algo_make_heap_if	将数据区间变成使用指定比较规则的堆。
algo_sort_heap	将使用默认比较规则的堆进行堆排序。
algo_sort_heap_if	将使用指定比较规则的堆进行堆排序。
algo_is_heap	判断一个数据区间是否是使用默认比较规则的堆。
algo_is_heap_if	判断一个数据区间是否是使用指定比较规则的堆。
algo_min	使用默认比较规则找出两个迭代器所指的数据的小者。
algo_min_if	使用指定比较规则找出两个迭代器所指的数据的小者。
algo_max	使用默认比较规则找出两个迭代器所指的数据的大者。
algo_max_if	使用指定比较规则找出两个迭代器所指的数据的大者。
algo_min_element	使用默认的比较规则找出数据区间中最小的数据。
algo_min_element_if	使用指定的比较规则找出数据区间中最小的数据。
algo_max_element	使用默认的比较规则找出数据区间中最大的数据。
algo_max_element_if	使用指定的比较规则找出数据区间中最大的数据。
algo_lexicographical_compare	使用默认比较规则按照字典方式比较两个数据区间。
algo_lexicographical_compare_if	使用指定比较规则按照字典方式比较两个数据区间。
algo_lexicographical_compare_3way	使用默认比较规则按照字典方式比较两个数据区间，返回 3 种结果。
algo_lexicographical_compare_3way_if	使用指定比较规则按照字典方式比较两个数据区间，返回 3 种结果。
algo_next_permutation	返回使用默认比较规则的当前数据区间的下一排列方式。
algo_next_permutation_if	返回使用指定比较规则的当前数据区间的下一排列方式。
algo_prev_permutation	返回使用默认比较规则的当前数据区间的上一排列方式。
algo_prev_permutation_if	返回使用指定比较规则的当前数据区间的上一排列方式。

1. 排序整个数据区间

- algo_sort algo_sort_if algo_stable_sort algo_stable_sort_if 对整个数据区间进行排序

1. algo_sort 和 algo_stable_sort 默认使用数据类型的小于操作函数作为比较规则进行排序。
2. algo_sort_if 和 algo_stable_sort_if 使用指定的比较规则进行排序。
3. algo_sort_if 和 algo_stable_sort_if 使用的比较规则不能修改数据。
4. algo_sort 和 algo_stable_sort 的不同之处在于 algo_stable_sort 对于相等的数据不改变原有数据的相对位置。
5. 有些容器提供了排序操作操作，排序时应该使用容器提供的排序操作函数。

下面给出了 algo_sort 和 algo_sort_if 的例子：

```
/*
 * sort3.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);
    iterator_t it_pos;
    int i = 0;

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

    deque_init(pdeq_coll);

    for(i = 1; i <= 9; ++i)
    {
        deque_push_back(pdeq_coll, i);
    }
    for(i = 1; i <= 9; ++i)
    {
        deque_push_back(pdeq_coll, i);
    }
```

```
    printf("on entry: ");
    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));
    printf("sorted: ");
    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);
    printf("sorted >: ");
    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;
}
```

输出结果：

on entry: 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9

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

下面这个例子不仅展示了 algo_stable_sort_if 函数的用法而且还展示了 algo_sort_if 和 algo_stable_sort_if 的不同：

```
/*
 * sort4.c
 * compile with : -lcstl
 */
```

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

static void _less_length(const void* cpv_first,
    const void* cpv_second, void* pv_output)
{
    *(bool_t*)pv_output = strlen((char*)cpv_first) < strlen((char*)cpv_second) ?
        true : false;
}

int main(int argc, char* argv[])
{
    vector_t* pvec_coll1 = create_vector(char*);
    vector_t* pvec_coll2 = create_vector(char*);
    iterator_t it_pos;

    if(pvec_coll1 == NULL || pvec_coll2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_coll1);
    vector_init(pvec_coll2);

    vector_push_back(pvec_coll1, "1xxx");
    vector_push_back(pvec_coll1, "2x");
    vector_push_back(pvec_coll1, "3x");
    vector_push_back(pvec_coll1, "4x");
    vector_push_back(pvec_coll1, "5xx");
    vector_push_back(pvec_coll1, "6xxxx");
    vector_push_back(pvec_coll1, "7xx");
    vector_push_back(pvec_coll1, "8xxx");
    vector_push_back(pvec_coll1, "9xx");
    vector_push_back(pvec_coll1, "10xxx");
    vector_push_back(pvec_coll1, "11");
    vector_push_back(pvec_coll1, "12");
    vector_push_back(pvec_coll1, "13");
    vector_push_back(pvec_coll1, "14xx");
    vector_push_back(pvec_coll1, "15");
    vector_push_back(pvec_coll1, "16");
```

```
    vector_push_back(pvec_coll1, "17");
    vector_assign(pvec_coll2, pvec_coll1);

    printf("on entry:\n");
    for(it_pos = vector_begin(pvec_coll1);
        !iterator_equal(it_pos, vector_end(pvec_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%s ", (char*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    algo_sort_if(vector_begin(pvec_coll1),
        vector_end(pvec_coll1), _less_length);
    algo_stable_sort_if(vector_begin(pvec_coll2),
        vector_end(pvec_coll2), _less_length);
    printf("with algo_sort_if():\n");
    for(it_pos = vector_begin(pvec_coll1);
        !iterator_equal(it_pos, vector_end(pvec_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%s ", (char*)iterator_get_pointer(it_pos));
    }
    printf("\n");
    printf("with algo_stable_sort_if():\n");
    for(it_pos = vector_begin(pvec_coll2);
        !iterator_equal(it_pos, vector_end(pvec_coll2));
        it_pos = iterator_next(it_pos))
    {
        printf("%s ", (char*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    vector_destroy(pvec_coll1);
    vector_destroy(pvec_coll2);

    return 0;
}
```

程序的输出结果：

on entry:

1xxx 2x 3x 4x 5xx 6xxxx 7xx 8xxx 9xx 10xxx 11 12 13 14xx 15 16 17

with algo_sort_if():

13 15 11 12 2x 17 16 3x 4x 5xx 7xx 9xx 1xxx 8xxx 14xx 10xxx 6xxxx

with algo_stable_sort_if():
2x 3x 4x 11 12 13 15 16 17 5xx 7xx 9xx 1xxx 8xxx 14xx 6xxxx 10xxx

2. 部分排序

- algo_partial_sort　algo_partial_sort_if 将数据区间的一部分排序。

1. algo_partial_sort 使用数据的小于操作函数作为比较规则排序。
2. algo_partial_sort_if 使用指定的比较规则排序。
3. 这两个算法不像 algo_sort 排序整个数据区间，而是排序后只是前面的一部分数据有序，后面的一部分并未排序。
4. algo_partial_sort_if 使用的函数不能修改数据。

下面的例子展示了如何使用 algo_partial_sort 和 algo_partial_sort_if：

```
/*
 * psort1.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);
    iterator_t it_pos;
    int i = 0;

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

    deque_init(pdeq_coll);

    for(i = 3; i <= 7; ++i)
    {
        deque_push_back(pdeq_coll, i);
    }
    for(i = 2; i <= 6; ++i)
```

```
    {
        deque_push_back(pdeq_coll, i);
    }
    for(i = 1; i <= 5; ++i)
    {
        deque_push_back(pdeq_coll, i);
    }
    printf("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_partial_sort(deque_begin(pdeq_coll),
        iterator_next_n(deque_begin(pdeq_coll), 5), deque_end(pdeq_coll));
    printf("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_partial_sort_if(deque_begin(pdeq_coll),
        iterator_next_n(deque_begin(pdeq_coll), 5),
        deque_end(pdeq_coll), fun_greater_int);
    printf("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");

    deque_destroy(pdeq_coll);

    return 0;
}
```

输出结果：
coll: 3 4 5 6 7 2 3 4 5 6 1 2 3 4 5
coll: 1 2 2 3 3 7 6 5 5 6 4 4 3 4 5
coll: 7 6 6 5 5 1 2 2 3 3 4 4 3 4 5
从结果看来 algo_partial_sort 和 algo_partial_sort_if 执行后只有前 5 个数字是这个数据区间排序的其余的是无序的。

algo_partial_sort_copy　algo_partial_sort_copy_if 对数据区间进行部分排序并将结果拷贝到目的数据区间中
algo_partial_sort_copy 使用数据类型的小于操作函数作为比较规则排序。
algo_partial_sort_copy_if 使用指定的比较规则排序。
algo_partial_sort_copy_if 使用的函数不能修改数据。
两个算法拷贝到目的数据区间中的是最小排序数据区间。
两个算法都返回拷贝到目的数据区间的数据的末尾。
下面是这两个算法的例子：

```
/*
 * psort2.c
 * compile with : -lcstl
 */

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

int main(int argc, char* argv[])
{
    deque_t* pdeq_coll = create_deque(int);
    vector_t* pvec_coll6 = create_vector(int);
    vector_t* pvec_coll30 = create_vector(int);
    iterator_t it_pos;
    iterator_t it_end;
    int i = 0;

    if(pdeq_coll == NULL || pvec_coll6 == NULL || pvec_coll30 == NULL)
    {
        return -1;
    }

    deque_init(pdeq_coll);
    vector_init_n(pvec_coll6, 6);
    vector_init_n(pvec_coll30, 30);

    for(i = 3; i <= 7; ++i)
```

```
{
    deque_push_back(pdeq_coll, i);
}
for(i = 2; i <= 6; ++i)
{
    deque_push_back(pdeq_coll, i);
}
for(i = 1; i <= 5; ++i)
{
    deque_push_back(pdeq_coll, i);
}
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");

it_end = algo_partial_sort_copy(deque_begin(pdeq_coll), deque_end(pdeq_coll),
    vector_begin(pvec_coll6), vector_end(pvec_coll6));
for(it_pos = vector_begin(pvec_coll6);
    !iterator_equal(it_pos, it_end);
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");

it_end = algo_partial_sort_copy_if(deque_begin(pdeq_coll), deque_end(pdeq_coll),
    vector_begin(pvec_coll30), vector_end(pvec_coll30), fun_greater_int);
for(it_pos = vector_begin(pvec_coll30);
    !iterator_equal(it_pos, it_end);
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");

deque_destroy(pdeq_coll);
vector_destroy(pvec_coll6);
vector_destroy(pvec_coll30);
```

```
    return 0;
}
```

程序输出：
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

3. 测试数据区间是否排序

algo_is_sorted　algo_is_sorted_if 判断一个数据区间是否排序
algo_is_sorted 判断一个数据区间是否是以数据类型的小于操作作为比较规则排序的。
algo_is_sorted_if 判断一个数据区间是否是以指定的比较规则排序的。
algo_is_sorted_if 使用的函数不能修改数据。
下面是这个两个算法的例子：

```
/*
 * issorted1.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);
    iterator_t it_pos;
    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_random_shuffle(vector_begin(pvec_coll), vector_end(pvec_coll));
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");
if(algo_is_sorted(vector_begin(pvec_coll), vector_end(pvec_coll)))
{
    printf("coll is sorted with <.\n");
}
else
{
    printf("coll is not sorted with <.\n");
}
if(algo_is_sorted_if(vector_begin(pvec_coll),
    vector_end(pvec_coll), fun_greater_int))
{
    printf("coll is sorted with >.\n");
}
else
{
    printf("coll is not sorted with >.\n");
}

algo_sort(vector_begin(pvec_coll), vector_end(pvec_coll));
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");
if(algo_is_sorted(vector_begin(pvec_coll), vector_end(pvec_coll)))
{
    printf("coll is sorted with <.\n");
}
else
{
    printf("coll is not sorted with <.\n");
```

```
    }
    if(algo_is_sorted_if(vector_begin(pvec_coll),
        vector_end(pvec_coll), fun_greater_int))
    {
        printf("coll is sorted with >.\n");
    }
    else
    {
        printf("coll is not sorted with >.\n");
    }

    algo_sort_if(vector_begin(pvec_coll), vector_end(pvec_coll), fun_greater_int);
    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");
    if(algo_is_sorted(vector_begin(pvec_coll), vector_end(pvec_coll)))
    {
        printf("coll is sorted with <.\n");
    }
    else
    {
        printf("coll is not sorted with <.\n");
    }
    if(algo_is_sorted_if(vector_begin(pvec_coll),
        vector_end(pvec_coll), fun_greater_int))
    {
        printf("coll is sorted with >.\n");
    }
    else
    {
        printf("coll is not sorted with >.\n");
    }

    vector_destroy(pvec_coll);

    return 0;
}
```

输出结果：

8 3 9 4 7 2 5 1 6

coll is not sorted with <.
coll is not sorted with >.
1 2 3 4 5 6 7 8 9
coll is sorted with <.
coll is not sorted with >.
9 8 7 6 5 4 3 2 1
coll is not sorted with <.
coll is sorted with >.

4. 根据第 n 个数据排序

- algo_nth_element　algo_nth_element_if 根据第 n 个数据排序，排序后第 n 个数据是排序后的数据，前面的数据都是小于 n 的数据后面的数据都是大于 n 的数据但是不保证第 n 个数据前后都是有序的

1. algo_nth_element 使用数据的小于操作函数作为默认的比较规则排序。
2. algo_nth_element_if 使用指定的比较规则排序。
3. algo_nth_element_if 使用的函数不能修改数据。

下面是 algo_nth_element 和 algo_nth_element_if 的使用实例：

```
/*
 * nth1.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);
    iterator_t it_pos;
    int i = 0;

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

    deque_init(pdeq_coll);
```

```
for(i = 3; i <= 7; ++i)
{
    deque_push_back(pdeq_coll, i);
}
for(i = 2; i <= 6; ++i)
{
    deque_push_back(pdeq_coll, i);
}
for(i = 1; i <= 5; ++i)
{
    deque_push_back(pdeq_coll, i);
}
printf("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_nth_element(deque_begin(pdeq_coll),
    iterator_next_n(deque_begin(pdeq_coll), 3), deque_end(pdeq_coll));
printf("the four last elements are: ");
for(it_pos = deque_begin(pdeq_coll);
    !iterator_equal(it_pos, iterator_next_n(deque_begin(pdeq_coll), 4));
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");

algo_nth_element(deque_begin(pdeq_coll),
    iterator_prev_n(deque_end(pdeq_coll), 4), deque_end(pdeq_coll));
printf("the four highest elements are: ");
for(it_pos = iterator_prev_n(deque_end(pdeq_coll), 4);
    !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_nth_element_if(deque_begin(pdeq_coll),
        iterator_next_n(deque_begin(pdeq_coll), 3),
        deque_end(pdeq_coll), fun_greater_int);
    printf("the four highest elements are: ");
    for(it_pos = deque_begin(pdeq_coll);
        !iterator_equal(it_pos, iterator_next_n(deque_begin(pdeq_coll), 4));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    deque_destroy(pdeq_coll);

    return 0;
}
```

输出结果：

coll: 3 4 5 6 7 2 3 4 5 6 1 2 3 4 5

the four last elements are: 2 1 2 3

the four highest elements are: 5 6 7 6

the four highest elements are: 6 7 6 5

5. 二分查找

- algo_lower_bound　algo_lower_bound_if 在有序的数据区间中查找第一个大于等于指定的数据。

1. algo_upper_bound　algo_upper_bound_if 在有序的数据区间中查找第一个大于指定的数据。
2. algo_lower_bound 和 algo_upper_bound 都要求数据区间使用数据类型的小于操作函数作为默认比较规则排序。
3. algo_lower_bound_if 和 algo_upper_bound_if 都要求数据区间使用指定的比较规则排序。
4. 如果没有这样的数据，四个算法都返回数据区间的末尾。
5. algo_lower_bound_if 和 algo_upper_bound_if 使用的函数都不能修改数据。
6. 关联容器提供了功能相同的操作函数，首先考虑使用关联容器提供的操作。

下面的例子展示了 algo_lower_bound 和 algo_upper_bound 的用法：

```
/*
 * bounds1.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);
    iterator_t it_pos;
    iterator_t it_lower;
    iterator_t it_upper;
    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);
    }
    for(i = 1; i <= 9; ++i)
    {
        list_push_back(plist_coll, i);
    }

    list_sort(plist_coll);
    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");

    it_lower = algo_lower_bound(list_begin(plist_coll), list_end(plist_coll), 5);
    it_upper = algo_upper_bound(list_begin(plist_coll), list_end(plist_coll), 5);
    printf("5 could get position %d up to %d without breaking the sorting.\n",
        iterator_distance(list_begin(plist_coll), it_lower) + 1,
        iterator_distance(list_begin(plist_coll), it_upper) + 1);

    list_insert(plist_coll,
        algo_lower_bound(list_begin(plist_coll), list_end(plist_coll), 3),
        3);
```

```
    list_insert(plist_coll,
        algo_upper_bound(list_begin(plist_coll), list_end(plist_coll), 7),
        7);

    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;
}
```

输出结果：

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

5 could get position 9 up to 11 without breaking the sorting.

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

- algo_equal_range　algo_equal_range_if 返回有序数据区间中包含指定数据的范围

1. algo_equal_range 要求数据区间是以数据类型的小于操作函数为默认比较规则排序的。
2. algo_equal_range_if 要求数据区间使用指定的比较规则排序。
3. 这两个算法返回的范围的上下界分别与 lower_bound 和 upper_bound 返回的位置相同。
4. algo_equal_range_if 使用的函数不能修改数据。
5. 关联容器提供了具有相同功能的操作函数，优先考虑使用容器提供的操作函数。

下面是使用 algo_equal_range 和 algo_equal_range_if 的例子：

```
/*
 * eqrange1.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);
    iterator_t it_pos;
    range_t r_range;
    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);
    }
    for(i = 1; i <= 9; ++i)
    {
        list_push_back(plist_coll, i);
    }

    list_sort(plist_coll);
    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");

    r_range = algo_equal_range(list_begin(plist_coll), list_end(plist_coll), 5);
    printf("5 could get position %d up to %d without breaking the sorting.\n",
        iterator_distance(list_begin(plist_coll), r_range.it_begin),
        iterator_distance(list_begin(plist_coll), r_range.it_end));

    list_destroy(plist_coll);

    return 0;
}
```

输出结果：

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

5 could get position 8 up to 10 without breaking the sorting.

algo_binary_search　algo_binary_search_if 在有序的数据区间中查找指定的数据。

algo_binary_search 在使用数据类型的小于操作为默认比较规则排序的数据区间中查找指定的数据。

algo_binary_search_if 在使用指定的比较规则排序的数据区间中查找指定的数据。

两个算法使用的二分查找方式。

algo_binary_search_if 使用的函数不能修改数据。
下面是这两个算法的例子：

```
/*
 * bsearch1.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);
    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);
    }

    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");

    if(algo_binary_search(list_begin(plist_coll), list_end(plist_coll), 5))
    {
        printf("5 is present\n");
    }
    else
    {
```

```
        printf("5 is not present\n");
    }

    if(algo_binary_search(list_begin(plist_coll), list_end(plist_coll), 43))
    {
        printf("43 is present\n");
    }
    else
    {
        printf("43 is not present\n");
    }

    list_destroy(plist_coll);

    return 0;
}
```

程序的输出结果：
1 2 3 4 5 6 7 8 9
5 is present
43 is not present

6. 合并

- algo_merge　algo_merge_if 将两个有序的数据区间合并

1. algo_merge 将使用数据类型的小于操作函数作为默认比较规则排序的两个数据区间合并。
2. algo_merge_if 将使用指定比较规则排序的两个数据区间合并。
3. 两个算法都返回目的数据区间中合并的数据的末尾，源数据区间的数据不改变。
4. algo_merge_if 使用的函数不能修改数据。
5. 用户必须保证目的数据区间足够大。

下面是使用 algo_merge 和 algo_merge_if 的实例：

```
/*
 * merge1.c
 * compile with : -lcstl
 */

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

```
int main(int argc, char* argv[])
{
    list_t* plist_coll1 = create_list(int);
    set_t* pset_coll2 = create_set(int);
    vector_t* pvec_merge = create_vector(int);
    iterator_t it_pos;
    int i = 0;

    if(plist_coll1 == NULL || pset_coll2 == NULL || pvec_merge == NULL)
    {
        return -1;
    }

    list_init(plist_coll1);
    set_init(pset_coll2);
    vector_init(pvec_merge);

    for(i = 1; i <= 6; ++i)
    {
        list_push_back(plist_coll1, i);
    }
    for(i = 3; i <= 8; ++i)
    {
        set_insert(pset_coll2, i);
    }
    vector_resize(pvec_merge, list_size(plist_coll1) + set_size(pset_coll2));

    printf("coll1: ");
    for(it_pos = list_begin(plist_coll1);
        !iterator_equal(it_pos, list_end(plist_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");
    printf("coll2: ");
    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");

    algo_merge(list_begin(plist_coll1), list_end(plist_coll1),
        set_begin(pset_coll2), set_end(pset_coll2), vector_begin(pvec_merge));
    printf("merge: ");
    for(it_pos = vector_begin(pvec_merge);
        !iterator_equal(it_pos, vector_end(pvec_merge));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    list_destroy(plist_coll1);
    set_destroy(pset_coll2);
    vector_destroy(pvec_merge);

    return 0;
}
```

输出的结果：
coll1: 1 2 3 4 5 6
coll2: 3 4 5 6 7 8
merge: 1 2 3 3 4 4 5 5 6 6 7 8

- algo_inplace_merge　algo_inplace_merge_if 合并同一数据区间的两个有序部分

1. algo_inplace_merge 合并同一数据区间中两个使用数据类型的小于操作作为默认比较规则排序的有序部分。
2. algo_inplace_merge_if 合并同一数据区间中两个使用指定比较规则排序的有序部分。
3. 合并后整个数据区间有序。
4. algo_inplace_merge_if 使用的函数不能修改数据。

下面是这两个算法的例子：

```
/*
 * merge2.c
 * compile with : -lcstl
 */

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

int main(int argc, char* argv[])
{
```

```
vector_t* pvec_coll1 = create_vector(int);
list_t* plist_coll2 = create_list(int);
iterator_t it_pos;
iterator_t it_bound;
int i = 0;

if(pvec_coll1 == NULL || plist_coll2 == NULL)
{
    return -1;
}

vector_init(pvec_coll1);
list_init(plist_coll2);

for(i = 4; i <= 9; ++i)
{
    vector_push_back(pvec_coll1, i);
}
for(i = 1; i <= 7; ++i)
{
    vector_push_back(pvec_coll1, i);
}

for(i = 6; i >= 2; --i)
{
    list_push_back(plist_coll2, i);
}
for(i = 9; i >= 4; --i)
{
    list_push_back(plist_coll2, i);
}

printf("coll1: ");
for(it_pos = vector_begin(pvec_coll1);
    !iterator_equal(it_pos, vector_end(pvec_coll1));
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");
printf("coll2: ");
for(it_pos = list_begin(plist_coll2);
    !iterator_equal(it_pos, list_end(plist_coll2));
```

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

    it_bound = algo_find(vector_begin(pvec_coll1), vector_end(pvec_coll1), 1);
    algo_inplace_merge(vector_begin(pvec_coll1), it_bound, vector_end(pvec_coll1));
    printf("coll1: ");
    for(it_pos = vector_begin(pvec_coll1);
        !iterator_equal(it_pos, vector_end(pvec_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    it_bound = algo_find(list_begin(plist_coll2), list_end(plist_coll2), 9);
    algo_inplace_merge_if(list_begin(plist_coll2), it_bound,
        list_end(plist_coll2), fun_greater_int);
    printf("coll2: ");
    for(it_pos = list_begin(plist_coll2);
        !iterator_equal(it_pos, list_end(plist_coll2));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    vector_destroy(pvec_coll1);
    list_destroy(plist_coll2);

    return 0;
}
```

程序的输出结果：

coll1: 4 5 6 7 8 9 1 2 3 4 5 6 7

coll2: 6 5 4 3 2 9 8 7 6 5 4

coll1: 1 2 3 4 4 5 5 6 6 7 7 8 9

coll2: 9 8 7 6 6 5 5 4 4 3 2

7. 集合算法

- algo_includes　algo_includes_if 判断第一个有序的数据区间是否包含第二个有序的数据区间。

1. algo_includes 要求两个数据区间都是以数据类型的小于操作为默认比较规则排序的。
2. algo_includes_if 要求两个数据区间都是以指定的比较规则排序的。
3. algo_includes_if 使用的函数不能修改数据。

下面是这两个算法的例子：

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

#include <stdio.h>
#include <cstl/clist.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);
    iterator_t it_pos;
    int i = 0;

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

    list_init(plist_coll1);
    vector_init(pvec_coll2);

    for(i = 1; i <= 9; ++i)
    {
        list_push_back(plist_coll1, i);
    }
    vector_push_back(pvec_coll2, 3);
    vector_push_back(pvec_coll2, 4);
    vector_push_back(pvec_coll2, 7);

    printf("coll1: ");
```

```
    for(it_pos = list_begin(plist_coll1);
        !iterator_equal(it_pos, list_end(plist_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");
    printf("coll2: ");
    for(it_pos = vector_begin(pvec_coll2);
        !iterator_equal(it_pos, vector_end(pvec_coll2));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    if(algo_includes(list_begin(plist_coll1), list_end(plist_coll1),
        vector_begin(pvec_coll2), vector_end(pvec_coll2)))
    {
        printf("all elements of coll2 are also in coll1\n");
    }
    else
    {
        printf("not all elements of coll2 are also in coll1\n");
    }

    list_destroy(plist_coll1);
    vector_destroy(pvec_coll2);

    return 0;
}
```

程序的结果：
coll1: 1 2 3 4 5 6 7 8 9
coll2: 3 4 7
all elements of coll2 are also in coll1

- algo_set_union　algo_set_union_if 获得两个有序数据区间的并集
- algo_set_intersection　algo_set_intersection_if 获得两个有序数据区间的交集。
- algo_set_difference　algo_set_difference_if 获得两个有序数据区间的差集。
- algo_set_symmetric_difference　algo_set_symmetric_difference_if 获得两个有序数据区间的对称差集。

不带_if 后缀的算法要求两个数据区间是使用数据类型的小于操作函数作为默认比较规则进行排序的。
带有_if 后缀的算法要求两个数据区间使用指定的比较规则进行排序。
带有_if 后缀的算法使用的函数不能修改数据。

目的数据区间要足够大，源数据区间中的数据不会被修改。
上面的算法都返回目的数据区间中结果数据的末尾。
下面是关于集合算法的例子：

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

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

int main(int argc, char* argv[])
{
    list_t* plist_coll1 = create_list(int);
    deque_t* pdeq_coll2 = create_deque(int);
    vector_t* pvec_result = create_vector(int);
    iterator_t it_pos;
    iterator_t it_end;
    int i = 0;

    if(plist_coll1 == NULL || pdeq_coll2 == NULL || pvec_result == NULL)
    {
        return -1;
    }

    list_init(plist_coll1);
    deque_init(pdeq_coll2);
    vector_init(pvec_result);

    list_push_back(plist_coll1, 1);
    list_push_back(plist_coll1, 2);
    list_push_back(plist_coll1, 2);
    list_push_back(plist_coll1, 4);
    list_push_back(plist_coll1, 6);
    list_push_back(plist_coll1, 7);
    list_push_back(plist_coll1, 7);
    list_push_back(plist_coll1, 9);
    deque_push_back(pdeq_coll2, 2);
    deque_push_back(pdeq_coll2, 2);
    deque_push_back(pdeq_coll2, 2);
```

```
deque_push_back(pdeq_coll2, 3);
deque_push_back(pdeq_coll2, 6);
deque_push_back(pdeq_coll2, 6);
deque_push_back(pdeq_coll2, 8);
deque_push_back(pdeq_coll2, 9);
vector_resize(pvec_result, list_size(plist_coll1) + deque_size(pdeq_coll2));

printf("coll1: ");
for(it_pos = list_begin(plist_coll1);
    !iterator_equal(it_pos, list_end(plist_coll1));
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");
printf("coll2: ");
for(it_pos = deque_begin(pdeq_coll2);
    !iterator_equal(it_pos, deque_end(pdeq_coll2));
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");

it_end = algo_merge(list_begin(plist_coll1), list_end(plist_coll1),
    deque_begin(pdeq_coll2), deque_end(pdeq_coll2), vector_begin(pvec_result));
printf("merge: ");
for(it_pos = vector_begin(pvec_result);
    !iterator_equal(it_pos, it_end);
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
printf("\n");
it_end = algo_set_union(list_begin(plist_coll1), list_end(plist_coll1),
    deque_begin(pdeq_coll2), deque_end(pdeq_coll2), vector_begin(pvec_result));
printf("union: ");
for(it_pos = vector_begin(pvec_result);
    !iterator_equal(it_pos, it_end);
    it_pos = iterator_next(it_pos))
{
    printf("%d ", *(int*)iterator_get_pointer(it_pos));
}
```

```
    printf("\n");
    it_end = algo_set_intersection(list_begin(plist_coll1), list_end(plist_coll1),
        deque_begin(pdeq_coll2), deque_end(pdeq_coll2), vector_begin(pvec_result));
    printf("intersection: ");
    for(it_pos = vector_begin(pvec_result);
        !iterator_equal(it_pos, it_end);
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");
    it_end = algo_set_difference(list_begin(plist_coll1), list_end(plist_coll1),
        deque_begin(pdeq_coll2), deque_end(pdeq_coll2), vector_begin(pvec_result));
    printf("difference: ");
    for(it_pos = vector_begin(pvec_result);
        !iterator_equal(it_pos, it_end);
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");
    it_end = algo_set_symmetric_difference(list_begin(plist_coll1),
        list_end(plist_coll1), deque_begin(pdeq_coll2), deque_end(pdeq_coll2),
        vector_begin(pvec_result));
    printf("symmetric difference: ");
    for(it_pos = vector_begin(pvec_result);
        !iterator_equal(it_pos, it_end);
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    list_destroy(plist_coll1);
    deque_destroy(pdeq_coll2);
    vector_destroy(pvec_result);

    return 0;
}
```

程序的输出结果：

coll1: 1 2 2 4 6 7 7 9

coll2: 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

8. 堆算法

- algo_make_heap algo_make_heap_if 将一个数据区间转换成堆
- algo_push_heap algo_push_heap_if 向堆中添加一个数据
- algo_pop_heap algo_pop_heap_if 将堆中优先级最高的数据删除
- algo_sort_heap algo_sort_heap_if 将一个堆转换成有序的数据区间
- algo_is_heap algo_is_heap_if 判断一个数据区间是否是堆

1. 没有_if 后缀的算法要求使用数据类型的小于操作函数作为默认的比较规则。
2. 带有_if 后缀的算法要求使用指定的比较规则。
3. 带有_if 后缀的算法使用的函数不能修改数据。

下面是关于堆算法的例子：

```
/*
 * heap1.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);
    iterator_t it_pos;
    int i = 0;

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

    vector_init(pvec_coll);

    for(i = 3; i <= 7; ++i)
    {
```

```
    vector_push_back(pvec_coll, i);
}
for(i = 5; i <= 9; ++i)
{
    vector_push_back(pvec_coll, i);
}
for(i = 1; i <= 4; ++i)
{
    vector_push_back(pvec_coll, i);
}

printf("on entry: ");
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");

algo_make_heap(vector_begin(pvec_coll), vector_end(pvec_coll));
printf("after algo_make_heap(): ");
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");

algo_pop_heap(vector_begin(pvec_coll), vector_end(pvec_coll));
vector_pop_back(pvec_coll);
printf("after algo_pop_heap(): ");
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_push_back(pvec_coll, 17);
algo_push_heap(vector_begin(pvec_coll), vector_end(pvec_coll));
```

```
    printf("after algo_push_heap(): ");
    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");

    algo_sort_heap(vector_begin(pvec_coll), vector_end(pvec_coll));
    printf("after algo_sort_heap(): ");
    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;
}
```

程序的输出：

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

将 pvec_coll 中的数据转换成堆后，数据的顺序是：9 8 6 7 7 5 5 3 6 4 1 2 3 4，将这个序列转换成二叉树可以看出字节点的值都是小于或者等于父节点的。

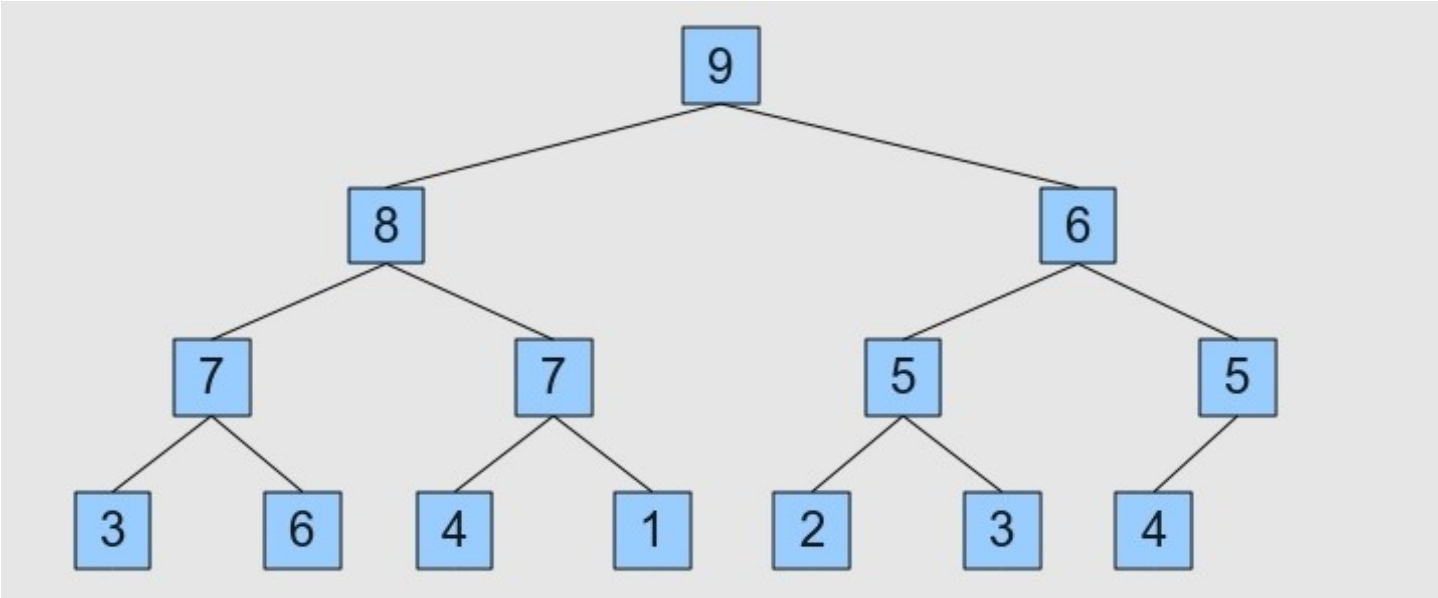

图 6.5 以二叉树的形式显示堆数据

9. 最大值与最小值

- algo_min algo_min_if 返回两个迭代器中值小的迭代器
- algo_max algo_max_if 返回两个迭代器中值大的迭代器

1. 没有_if 后缀的算法使用数据类型的小于操作作为默认的比较规则。
2. 带有_if 后缀的算法使用指定的比较规则。
3. 带有_if 后缀的算法使用的函数不能修改数据。

下面给出了这些算法的例子：

```
/*
 * minmax1.c
 * compile with : -lcstl
 */

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

static void _abs_less(const void* cpv_first,
    const void* cpv_second, void* pv_output)
{
    *(bool_t*)pv_output = abs(*(int*)cpv_first) < abs(*(int*)cpv_second) ?
        true : false;
```

```
}

int main(int argc, char* argv[])
{
    deque_t* pdeq_coll = create_deque(int);
    iterator_t it_first;
    iterator_t it_second;
    iterator_t it_min;
    iterator_t it_max;

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

    deque_init(pdeq_coll);

    deque_push_back(pdeq_coll, 3);
    deque_push_back(pdeq_coll, -5);

    it_first = deque_begin(pdeq_coll);
    it_second = iterator_next(deque_begin(pdeq_coll));
    it_min = algo_min(it_first, it_second);
    it_max = algo_max(it_first, it_second);
    printf("min: %d\n", *(int*)iterator_get_pointer(it_min));
    printf("max: %d\n", *(int*)iterator_get_pointer(it_max));

    it_min = algo_min_if(it_first, it_second, _abs_less);
    it_max = algo_max_if(it_first, it_second, _abs_less);
    printf("absolute min: %d\n", *(int*)iterator_get_pointer(it_min));
    printf("absolute max: %d\n", *(int*)iterator_get_pointer(it_max));

    deque_destroy(pdeq_coll);

    return 0;
}
```

程序输出：

min: -5

max: 3

absolute min: 3

absolute max: -5

algo_min_element　algo_min_element_if 返回数据区间中最小数据的迭代器

algo_max_element　algo_max_element_if返回数据区间中最大数据的迭代器
没有_if后缀的算法要求使用数据类型的小于草组函数作为默认的比较规则。
带有_if后缀的算法要求使用指定的比较规则。
带有_if后缀的算法使用的函数不能修改数据。
下面是这四个算法的例子：

```
/*
 * minmax2.c
 * compile with : -lcstl
 */

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

static void _abs_less(const void* cpv_first,
    const void* cpv_second, void* pv_output)
{
    *(bool_t*)pv_output = abs(*(int*)cpv_first) < abs(*(int*)cpv_second) ?
        true : false;
}

int main(int argc, char* argv[])
{
    deque_t* pdeq_coll = create_deque(int);
    iterator_t it_pos;
    iterator_t it_min;
    iterator_t it_max;
    int i = 0;

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

    deque_init(pdeq_coll);

    for(i = 2; i <= 8; ++i)
    {
        deque_push_back(pdeq_coll, i);
    }
    for(i = -3; i <= 5; ++i)
    {
```

```
        deque_push_back(pdeq_coll, i);
    }

    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");

    it_min = algo_min_element(deque_begin(pdeq_coll), deque_end(pdeq_coll));
    it_max = algo_max_element(deque_begin(pdeq_coll), deque_end(pdeq_coll));
    printf("min: %d\n", *(int*)iterator_get_pointer(it_min));
    printf("max: %d\n", *(int*)iterator_get_pointer(it_max));

    it_min = algo_min_element_if(deque_begin(pdeq_coll),
        deque_end(pdeq_coll), _abs_less);
    it_max = algo_max_element_if(deque_begin(pdeq_coll),
        deque_end(pdeq_coll), _abs_less);
    printf("absolute min: %d\n", *(int*)iterator_get_pointer(it_min));
    printf("absolute max: %d\n", *(int*)iterator_get_pointer(it_max));

    deque_destroy(pdeq_coll);

    return 0;
}
```

程序输出：
2 3 4 5 6 7 8 -3 -2 -1 0 1 2 3 4 5
min: -3
max: 8
absolute min: 0
absolute max: 8

10. 按照辞典顺序比较

- algo_lexicographical_compare　algo_lexicographical_compare_if 按照辞典顺序比较
- algo_lexicographical_compare_3wap　algo_lexicographical_compare_3way_if 按照辞典顺序比较，返回 3 种结果

1. 没有_if 后缀的算法使用数据类型的小于操作函数作为默认的比较规则。
2. 带有_if 后缀的算法使用指定的比较规则。

3. 带有_if后缀的算法使用的函数不能修改数据。

下面是这四个算法的例子：

```
/*
 * lexico1.c
 * compile with : -lcstl
 */

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

static void _print_list(const void* cpv_input, void* pv_output)
{
    list_t* plist_coll = (list_t*)cpv_input;
    iterator_t it_pos;
    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");
}

static void _less_for_coll(const void* cpv_first,
    const void* cpv_second, void* pv_output)
{
    list_t* plist_coll1 = (list_t*)cpv_first;
    list_t* plist_coll2 = (list_t*)cpv_second;

    if(algo_lexicographical_compare(
        list_begin(plist_coll1), list_end(plist_coll1),
        list_begin(plist_coll2), list_end(plist_coll2)))
    {
        *(bool_t*)pv_output = true;
    }
    else
    {
        *(bool_t*)pv_output = false;
    }
}
```

```
int main(int argc, char* argv[])
{
    list_t* plist_c1 = create_list(int);
    list_t* plist_c2 = create_list(int);
    list_t* plist_c3 = create_list(int);
    list_t* plist_c4 = create_list(int);
    vector_t* pvec_cc = create_vector(list_t<int>);
    int i = 0;

    if(plist_c1 == NULL || plist_c2 == NULL || plist_c3 == NULL ||
       plist_c4 == NULL || pvec_cc == NULL)
    {
        return -1;
    }

    list_init(plist_c1);
    list_init(plist_c2);
    list_init(plist_c3);
    list_init(plist_c4);
    vector_init(pvec_cc);

    for(i = 1; i <= 5; ++i)
    {
        list_push_back(plist_c1, i);
    }
    list_assign(plist_c2, plist_c1);
    list_assign(plist_c3, plist_c1);
    list_assign(plist_c4, plist_c1);
    list_push_back(plist_c1, 7);
    list_push_back(plist_c3, 2);
    list_push_back(plist_c3, 0);
    list_push_back(plist_c4, 2);

    vector_push_back(pvec_cc, plist_c1);
    vector_push_back(pvec_cc, plist_c2);
    vector_push_back(pvec_cc, plist_c3);
    vector_push_back(pvec_cc, plist_c4);
    vector_push_back(pvec_cc, plist_c3);
    vector_push_back(pvec_cc, plist_c1);
    vector_push_back(pvec_cc, plist_c4);
    vector_push_back(pvec_cc, plist_c2);

    algo_for_each(vector_begin(pvec_cc), vector_end(pvec_cc), _print_list);
```

```
    printf("\n");
    algo_sort_if(vector_begin(pvec_cc), vector_end(pvec_cc), _less_for_coll);
    algo_for_each(vector_begin(pvec_cc), vector_end(pvec_cc), _print_list);

    list_destroy(plist_c1);
    list_destroy(plist_c2);
    list_destroy(plist_c3);
    list_destroy(plist_c4);
    vector_destroy(pvec_cc);

    return 0;
}
```

程序的输出结果：

1 2 3 4 5 7
1 2 3 4 5
1 2 3 4 5 2 0
1 2 3 4 5 2
1 2 3 4 5 2 0
1 2 3 4 5 7
1 2 3 4 5 2
1 2 3 4 5

1 2 3 4 5
1 2 3 4 5
1 2 3 4 5 2
1 2 3 4 5 2
1 2 3 4 5 2 0
1 2 3 4 5 2 0
1 2 3 4 5 7
1 2 3 4 5 7

11. 数据排列

algo_next_permutation　algo_next_permutation_if 获得当前数据区间的下一个排序
algo_prev_permutation　algo_prev_permutation_if 获得当前数据区间的上一个排序
没有_if 后缀的版本使用数据类型的小于操作函数作为默认的比较规则。
带有_if 后缀的版本使用指定的比较规则。
带有_if 后缀的版本使用的函数不能修改数据。
如果当前数据区间不是最后一个排列，算法返回 true，如果是最后一个排序则返回 false 同时生成第一个排列。
下面是这四个算法的例子：

```
/*
 * perm1.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);

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

    vector_init(pvec_coll);

    vector_push_back(pvec_coll, 1);
    vector_push_back(pvec_coll, 2);
    vector_push_back(pvec_coll, 3);

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

    while(algo_next_permutation(vector_begin(pvec_coll), vector_end(pvec_coll)))
    {
        algo_for_each(vector_begin(pvec_coll), vector_end(pvec_coll), _print);
        printf("\n");
    }
    printf("afterward: ");
    algo_for_each(vector_begin(pvec_coll), vector_end(pvec_coll), _print);
    printf("\n");

    while(algo_prev_permutation(vector_begin(pvec_coll), vector_end(pvec_coll)))
```

```
    {
        algo_for_each(vector_begin(pvec_coll), vector_end(pvec_coll), _print);
        printf("\n");
    }
    printf("now: ");
    algo_for_each(vector_begin(pvec_coll), vector_end(pvec_coll), _print);
    printf("\n");

    while(algo_prev_permutation(vector_begin(pvec_coll), vector_end(pvec_coll)))
    {
        algo_for_each(vector_begin(pvec_coll), vector_end(pvec_coll), _print);
        printf("\n");
    }
    printf("afterward: ");
    algo_for_each(vector_begin(pvec_coll), vector_end(pvec_coll), _print);
    printf("\n");

    vector_destroy(pvec_coll);

    return 0;
}
```

输出结果：

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 算数算法

libcstl 提供了很多数值算法，它们主要对数据区间中的数值进行各种运算，要使用这些数值算法要包含头文件：

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

下面给出了 libcstl 提供的数值算法：

algo_iota	根据初始值将数据区间中的数据依次加一。
algo_accumulate	计算数据区间中数据的和。
algo_accumulate_if	使用指定函数计算数据区间中数据的和。
algo_inner_product	计算两个数据区间的内积。
algo_inner_product_if	使用指定函数计算两个数据区间的内积。
algo_partial_sum	计算数据区间的局部和。
algo_partial_sum_if	使用指定函数计算数据区间的局部和。
algo_adjacent_difference	计算数据区间中相邻数据的差。
algo_adjacent_difference_if	使用指定函数计算数据区间中相邻数据的差。
algo_power	计算数据的 n 次幂。
algo_power_if	使用指定函数计算数据的 n 次幂。

1. 对整个区间计算

- algo_iota 根据初始值对将数据区间一次赋予一个比上一个数大于一的值

下面是这个算法的使用实例：

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

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

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

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

```
    vector_init_n(pvec_coll, 10);

    algo_iota(vector_begin(pvec_coll), vector_end(pvec_coll), 7);
    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;
}
```

程序输出的结果：
7 8 9 10 11 12 13 14 15 16

- algo_accumulate　algo_accumulate_if 计算初始值和数据区间的和

1. algo_accumulate 计算初始值和数据区间中所有数据的和：initValue + elem1 + elem2 + elem3 + ...。
2. algo_accumulate_if 使用指定的函数计算初始值和数据区间中的所有数据的和：initValue op elem1 op elem2 op ...。
3. 如果数据区间为空返回初始值。
4. algo_accumulate_if 使用的函数不能修改数据。

下面是这两算法的使用实例：

```
/*
 * accu1.c
 * compile with : -lcstl
 */

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

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

    if(pvec_coll == NULL)
    {
```

```
        return -1;
    }

    vector_init_n(pvec_coll, 9);
    algo_iota(vector_begin(pvec_coll), vector_end(pvec_coll), 1);

    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");

    algo_accumulate(vector_begin(pvec_coll), vector_end(pvec_coll),
        0, &n_result);
    printf("sum: %d\n", n_result);
    algo_accumulate(vector_begin(pvec_coll), vector_end(pvec_coll),
        -100, &n_result);
    printf("sum: %d\n", n_result);
    algo_accumulate_if(vector_begin(pvec_coll), vector_end(pvec_coll),
        1, fun_multiplies_int, &n_result);
    printf("product: %d\n", n_result);
    algo_accumulate_if(vector_begin(pvec_coll), vector_end(pvec_coll),
        0, fun_multiplies_int, &n_result);
    printf("product: %d\n", n_result);

    vector_destroy(pvec_coll);

    return 0;
}
```

程序输出：

1 2 3 4 5 6 7 8 9

sum: 45

sum: -55

product: 362880

product: 0

- algo_inner_product　algo_inner_product_if 计算两个数据区间的内积

1. algo_inner_product 计算两个数据区间的内积：initValue + (a1 * b1) + (a2 * b2) + (a3 * b3) + ...。
2. algo_inner_product_if 使用指定的函数计算两个数据区间的内积：initValue op1 (a1 op2 b1) op1 (a2 op2 b2) op1 ...。
3. 如果数据区间为空，两个算法返回 initValue。

4. 必须保证第二个数据区间足够大。
5. op1 和 op2 不能修改数据。

下面是这两个算法的例子：

```
/*
 * inner1.c
 * compile with : -lcstl
 */

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

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

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

    list_init_n(plist_coll, 6);
    algo_iota(list_begin(plist_coll), list_end(plist_coll), 1);

    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");

    algo_inner_product(list_begin(plist_coll), list_end(plist_coll),
        list_begin(plist_coll), 0, &n_result);
    printf("inner product: %d\n", n_result);

    algo_inner_product_if(list_begin(plist_coll), list_end(plist_coll),
        list_begin(plist_coll), 1, fun_multiplies_int, fun_plus_int, &n_result);
    printf("product of sums: %d\n", n_result);
```

```
    list_destroy(plist_coll);

    return 0;
}
```

程序的输出：
1 2 3 4 5 6
inner product: 91
product of sums: 46080

2. 相对值和绝对值转换

- algo_partial_sum　algo_partial_sum_if 计算局部和

1. algo_partial_sum 计算局部和后的数据区间：a1　a1+a2　a1+a2+a3　a1+a2+a3+a4　...。
2. algo_partial_sum_if 使用指定函数计算局部和后的数据区间：a1　a1 op a2　a1 op a2 op a3　a1 op a2 op a3 op a4　...。
3. 两个算法返回目的区间中被覆盖的数据的末尾。
4. 要确保目的数据区间足够大。
5. op 不能修改数据。

下面是两个算法的例子：

```
/*
 * partsum1.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/clist.h>
#include <cstl/cvector.h>
#include <cstl/cnumeric.h>
#include <cstl/cfunctional.h>

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

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

```
    list_init_n(plist_coll1, 6);
    vector_init_n(pvec_coll2, list_size(plist_coll1));

    algo_iota(list_begin(plist_coll1), list_end(plist_coll1), 1);
    for(it_pos = list_begin(plist_coll1);
        !iterator_equal(it_pos, list_end(plist_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    algo_partial_sum(list_begin(plist_coll1),
        list_end(plist_coll1), vector_begin(pvec_coll2));
    for(it_pos = vector_begin(pvec_coll2);
        !iterator_equal(it_pos, vector_end(pvec_coll2));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    algo_partial_sum_if(list_begin(plist_coll1), list_end(plist_coll1),
        vector_begin(pvec_coll2), fun_multiplies_int);
    for(it_pos = vector_begin(pvec_coll2);
        !iterator_equal(it_pos, vector_end(pvec_coll2));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    list_destroy(plist_coll1);
    vector_destroy(pvec_coll2);

    return 0;
}
```

程序的输出结果：

1 2 3 4 5 6

1 3 6 10 15 21

1 2 6 24 120 720

- algo_adjacent_difference　algo_adjacent_difference_if 相邻数据的差

1. algo_adjacent_difference 计算相邻数据的差后数据区间：a1　a2-a1　a3-a2　a4-a3 ...。
2. algo_adjacent_difference_if 使用指定函数计算相邻数据的差后数据区间：a1　a2 op a1　a3 op a2　a4 op a3 ...。
3. 两算法都返回目的数据区间被覆盖的数据的末尾。
4. 必须保证目的数据区间足够大。
5. op 不能修改数据。

下面是这两个算法的使用实例：

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

#include <stdio.h>
#include <cstl/cdeque.h>
#include <cstl/clist.h>
#include <cstl/cnumeric.h>
#include <cstl/cfunctional.h>

int main(int argc, char* argv[])
{
    deque_t* pdeq_coll1 = create_deque(int);
    list_t* plist_coll2 = create_list(int);
    iterator_t it_pos;

    if(pdeq_coll1 == NULL || plist_coll2 == NULL)
    {
        return -1;
    }

    deque_init_n(pdeq_coll1, 6);
    list_init_n(plist_coll2, deque_size(pdeq_coll1));

    algo_iota(deque_begin(pdeq_coll1), deque_end(pdeq_coll1), 1);
    for(it_pos = deque_begin(pdeq_coll1);
        !iterator_equal(it_pos, deque_end(pdeq_coll1));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    algo_adjacent_difference(deque_begin(pdeq_coll1),
        deque_end(pdeq_coll1), list_begin(plist_coll2));
```

```
    for(it_pos = list_begin(plist_coll2);
        !iterator_equal(it_pos, list_end(plist_coll2));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    algo_adjacent_difference_if(deque_begin(pdeq_coll1), deque_end(pdeq_coll1),
        list_begin(plist_coll2), fun_plus_int);
    for(it_pos = list_begin(plist_coll2);
        !iterator_equal(it_pos, list_end(plist_coll2));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    algo_adjacent_difference_if(deque_begin(pdeq_coll1), deque_end(pdeq_coll1),
        list_begin(plist_coll2), fun_multiplies_int);
    for(it_pos = list_begin(plist_coll2);
        !iterator_equal(it_pos, list_end(plist_coll2));
        it_pos = iterator_next(it_pos))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    deque_destroy(pdeq_coll1);
    list_destroy(plist_coll2);

    return 0;
}
```

程序输出的结果：

1 2 3 4 5 6

1 1 1 1 1 1

1 3 5 7 9 11

1 2 6 12 20 30

- algo_power　algo_power_if 计算数据的幂值

1. algo_power 计算数据的 n 次幂。
2. algo_power_if 使用指定的函数计算数据的 n 次幂。

下面是两个算法的例子：

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

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

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

    if(plist_coll == 0)
    {
        return -1;
    }

    list_init_elem(plist_coll, 1, -2);

    algo_power(list_begin(plist_coll), 4, &n_result);
    printf("-2 * -2 * -2 * -2 = %d\n", n_result);
    algo_power_if(list_begin(plist_coll), 4, fun_plus_int, &n_result);
    printf("-2 + -2 + -2 + -2 = %d\n", n_result);

    list_destroy(plist_coll);

    return 0;
}
```

程序的输出：

-2 * -2 * -2 * -2 = 16

-2 + -2 + -2 + -2 = -8

下面给出一个相对数据和绝对数据相互转换的例子：

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

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

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

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

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

    vector_init(pvec_coll);

    vector_push_back(pvec_coll, 17);
    vector_push_back(pvec_coll, -3);
    vector_push_back(pvec_coll, 22);
    vector_push_back(pvec_coll, 13);
    vector_push_back(pvec_coll, 13);
    vector_push_back(pvec_coll, -9);

    printf("coll: ");
    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");

    algo_adjacent_difference(vector_begin(pvec_coll),
        vector_end(pvec_coll), vector_begin(pvec_coll));
    printf("relative: ");
    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");

    algo_partial_sum(vector_begin(pvec_coll),
        vector_end(pvec_coll), vector_begin(pvec_coll));
```

```
    printf("absolute: ");
    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;
}
```

程序的输出结果：

coll: 17 -3 22 13 13 -9

relative: 17 -20 25 -9 0 -22

absolute: 17 -3 22 13 13 -9

第七章 libcstl 函数

这一节主要讨论 libcstl 函数的概念，种类和用法。libcstl 提供的函数主要用来扩展算法的功能，有很多算法都提供了_if 后缀的版本，通过向这样的算法提供特殊的函数来改变和扩展算法原有的功能。

第一节 函数的种类

1. 函数的例子

为了说明函数的种类我们先看一个使用函数的例子，这个例子将函数的两种类型都用到了：

```
/*
 * fun1.c
 * compile with : -lcstl
 */

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

static void _print_name(const void* cpv_input, void* pv_output)
{
    printf("%s. %s\n", (char*)pair_first((pair_t*)cpv_input),
        (char*)pair_second((pair_t*)cpv_input));
}

static void _less_name(const void* cpv_first,
    const void* cpv_second, void* pv_output)
{
    if(strcmp((char*)pair_second((pair_t*)cpv_first),
        (char*)pair_second((pair_t*)cpv_second)) < 0)
    {
        *(bool_t*)pv_output = true;
    }
    else
```

```
    {
        *(bool_t*)pv_output = false;
    }
}

int main(int argc, char* argv[])
{
    vector_t* pvec_coll = create_vector(pair_t<char*, char*>);
    pair_t* ppair_item = create_pair(char*, char*);

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

    vector_init(pvec_coll);
    pair_init(ppair_item);

    pair_make(ppair_item, "Douglas E", "Comer");
    vector_push_back(pvec_coll, ppair_item);
    pair_make(ppair_item, "David L", "Stevens");
    vector_push_back(pvec_coll, ppair_item);
    pair_make(ppair_item, "Behdad", "Esfahbod");
    vector_push_back(pvec_coll, ppair_item);
    pair_make(ppair_item, "Guilherme de S", "Pastve");
    vector_push_back(pvec_coll, ppair_item);
    pair_make(ppair_item, "Havoc", "Pennington");
    vector_push_back(pvec_coll, ppair_item);
    pair_make(ppair_item, "Christian", "Persch");
    vector_push_back(pvec_coll, ppair_item);
    pair_make(ppair_item, "Mariano", "Suarez-Alvarez");
    vector_push_back(pvec_coll, ppair_item);

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

    algo_sort_if(vector_begin(pvec_coll), vector_end(pvec_coll), _less_name);

    algo_for_each(vector_begin(pvec_coll), vector_end(pvec_coll), _print_name);

    vector_destroy(pvec_coll);
    pair_destroy(ppair_item);
```

```
    return 0;
}
```

在这个例子中我们在 vector_t 中保存 pair_t，用 pair_t 的来保存人的名字，然后使用 algo_for_each 算法将这些名字打印出来，然后使用 algo_sort_if 算法依旧这些人名的 second name 进行排序，再次打印出来。这个程序的输出为：

Douglas E. Comer
David L. Stevens
Behdad. Esfahbod
Guilherme de S. Pastve
Havoc. Pennington
Christian. Persch
Mariano. Suarez-Alvarez

Douglas E. Comer
Behdad. Esfahbod
Guilherme de S. Pastve
Havoc. Pennington
Christian. Persch
David L. Stevens
Mariano. Suarez-Alvarez

2. 一元函数和二元函数

在上面的例子中使用 algo_for_each 算法打印名字的时候，我们使用了函数_print_name，这个函数具
void _print_name(const void* cpv_input, void* pv_output) ;
这样的声明形式。在使用 algo_sort_if 对名字进行排序的时候，我们使用了函数_less_name，这个函数具有
void _less_name(const void* cpv_first, const void* cpv_second, void* pv_output);
这样的声明形式。

以上这两种形式就是 libcstl 规定的函数规范形式，这两种形式以外的其他形式都不被认为是 libcstl 函数，libcstl 也不会接受其他形式的函数。libcstl 为函数定义的标准形式如下：

```
typedef void (*unary_function_t)(const void*, void*);
typedef void (*binary_function_t)(const void*, const void*, void*);
```

把第一种形式的函数叫做一元函数，使用 unary_function_t 类型表示，它的第一个参数是输入参数，第二个参数是输出参数。把第二种形式叫做二元函数，使用 binary_function_t 类型表示，它的前两个参数是输入参数，第三个参数是输出参数。

虽然一元函数和二元函数都是 libcstl 的函数形式但是二者是不能够通用的，也就是在要求使用一元函数的地方不能使用二元函数，反过来也一样。要求使用函数的的算法都明确的只是使用一元或者二元函数，如 algo_for_each 要求使用一元函数，而 algo_sort_if 要求使用二元函数。还有就是要求使用函数的算法中可以传递 NULL 表示使用默认函数。

3. 谓词

我们再次观察上面使用的两个函数，_print_name 的输出参数 pv_output 根本没有使用，而_less_name 的输出参数 pv_output 作为 bool_t 类型来使用，说明_print_name 主要起到了执行某种动作的作用，而_less_name 起点了判断条件的作用。我们把像_less_name 这样输出参数为 bool_t 类型的函数叫做谓词，而_print_name 这样输出参数不是 bool_t 类型(包括输出参数没有使用)的叫做非谓词。

函数分为一元函数和二元函数，那么谓词也就分为一元谓词和二元谓词了。一元谓词就是包含一个输入参数，输出参数是 bool_t 类型的函数，二元谓词就是包含两个输入参数，输出参数是 bool_t 类型的函数。前面例子中的_less_name 就是二元谓词。下面在列举一个一元谓词的例子：

```
/*
 * pre1.c
 * compile with : -lcstl
 */

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

static void _mod3(const void* cpv_input, void* pv_output)
{
    *(bool_t*)pv_output = *(int*)cpv_input % 3 == 0 ? true : false;
}

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

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

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

   list_init_n(plist_coll, 10);
```

```
    algo_iota(list_begin(plist_coll), list_end(plist_coll), 1);
    algo_for_each(list_begin(plist_coll), list_end(plist_coll), _print);
    printf("\n");

    it_end = algo_remove_if(list_begin(plist_coll), list_end(plist_coll), _mod3);
    algo_for_each(list_begin(plist_coll), it_end, _print);
    printf("\n");

    list_destroy(plist_coll);

    return 0;
}
```

程序的输出结果：
1 2 3 4 5 6 7 8 9 10
1 2 4 5 7 8 10

这个例子中的函数_mode3 就是一元谓词的例子。一般的说谓词主要是作为算法的判断条件出现，而函数则主要用于指向实际的任务。

第二节 预定义函数

libcstl 提供了大量的常用的函数，这些函数极大的方便的算法的使用，用户在扩展算法的时候大部分情况下是不用自己再重新设计和实现函数的。libcstl 的函数都定义在<cstl/cfunctional.h>中。下面将分类描述这些预定义函数，这些类型的函数对数据类型的支持成都不同，在每个类型的函数中都给出类这一类函数支持的数据类型，同时我们使用如 fun_plus_xxxx 这样的形式表形一个功能的函数，其中 fun_plus 表示两个数据相加的功能，xxxx 是某一具体的数据类型的名字，例如 int，这样真正的函数就是 fun_plus_int 表示两个 int 类型相加的函数，要全面的俩解形式如 fun_plus_xxxx 这样的函数的具体细节请参考《The libcstl Library Reference Manual》。

1. 算数函数

算数函数支持除了 C 字符串以外的所有 C 语言内建类型。

fun_plus_xxxx	求两个数据的和。
fun_minus_xxxx	求两个数据的差。
fun_multiplies_xxxx	求两个数据的积。
fun_divides_xxxx	求两个数据的商。
fun_modulus_xxxx	求两个数据的余数。
fun_negation_xxxx	对数据取反。

上面这些函数都是普通的函数，不是谓词，其中只有 fun_negation_xxxx 是一元函数，其余的都是二元函数。

2. 关系函数

关系函数支持所有的 C 语言内建类型和除了 bool_t，range_t，priority_queue_t 和迭代器类型以外的 libcstl 内建类型。

fun_equal_xxxx	测试两个数据是否相等。
fun_not_equal_xxxx	测试两个数据是否不等。
fun_greater_xxxx	测试第一个数据是否大于第二个数据。
fun_greater_equal_xxxx	测试第一个数据是否大于等于第二个数据。
fun_less_xxxx	测试第一个数据是否小于第二个数据。
fun_less_equal_xxxx	测试第一个数据是否小于等于第二个数据。

这些函数都是二元谓词。

3. 逻辑函数

逻辑函数只支持 bool_t 类型。

fun_logical_and_bool	逻辑与函数。
fun_logical_or_bool	逻辑或函数。
fun_logical_not_bool	逻辑非函数。

逻辑非函数是一元的其余两个是二元的，这一组函数即可以认为是谓词，又可以认为不是谓词。

4. 其他函数

这是一些未归类的函数，它们可以支持任意类型。

fun_random_number	产生随机数。
fun_default_uniary	默认的一元函数。
fun_default_binary	默认的二元函数。

第一个函数是一个一元函数，用于产生一个小于等于输入数据的随机数，输入数据是 0 时，随机数也是 0。

5. 预定义函数的例子

我们将本章第一节中的例子改造一些，使用预定义的函数进行排序：

```
/*
 * fun1.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <string.h>
#include <cstl/cvector.h>
#include <cstl/cutility.h>
#include <cstl/calgorithm.h>
#include <cstl/cfunctional.h>

static void _print_name(const void* cpv_input, void* pv_output)
{
    printf("%s. %s\n", (char*)pair_first((pair_t*)cpv_input),
        (char*)pair_second((pair_t*)cpv_input));
}

static void _less_name(const void* cpv_first,
    const void* cpv_second, void* pv_output)
{
    if(strcmp((char*)pair_second((pair_t*)cpv_first),
        (char*)pair_second((pair_t*)cpv_second)) < 0)
    {
        *(bool_t*)pv_output = true;
    }
    else
    {
        *(bool_t*)pv_output = false;
    }
}

int main(int argc, char* argv[])
{
    vector_t* pvec_coll = create_vector(pair_t<char*, char*>);
    pair_t* ppair_item = create_pair(char*, char*);

    if(pvec_coll == NULL || ppair_item == NULL)
```

```
    {
        return -1;
    }

    vector_init(pvec_coll);
    pair_init(ppair_item);

    pair_make(ppair_item, "Douglas E", "Comer");
    vector_push_back(pvec_coll, ppair_item);
    pair_make(ppair_item, "David L", "Stevens");
    vector_push_back(pvec_coll, ppair_item);
    pair_make(ppair_item, "Behdad", "Esfahbod");
    vector_push_back(pvec_coll, ppair_item);
    pair_make(ppair_item, "Guilherme de S", "Pastve");
    vector_push_back(pvec_coll, ppair_item);
    pair_make(ppair_item, "Havoc", "Pennington");
    vector_push_back(pvec_coll, ppair_item);
    pair_make(ppair_item, "Christian", "Persch");
    vector_push_back(pvec_coll, ppair_item);
    pair_make(ppair_item, "Mariano", "Suarez-Alvarez");
    vector_push_back(pvec_coll, ppair_item);

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

    algo_sort_if(vector_begin(pvec_coll), vector_end(pvec_coll), _less_name);
    algo_for_each(vector_begin(pvec_coll), vector_end(pvec_coll), _print_name);
    printf("------------------\n");

    algo_sort_if(vector_begin(pvec_coll), vector_end(pvec_coll), fun_less_pair);
    algo_for_each(vector_begin(pvec_coll), vector_end(pvec_coll), _print_name);

    vector_destroy(pvec_coll);
    pair_destroy(ppair_item);

    return 0;
}
```

程序的输出：

Douglas E. Comer
David L. Stevens
Behdad. Esfahbod
Guilherme de S. Pastve
Havoc. Pennington

Christian. Persch
Mariano. Suarez-Alvarez

Douglas E. Comer
Behdad. Esfahbod
Guilherme de S. Pastve
Havoc. Pennington
Christian. Persch
David L. Stevens
Mariano. Suarez-Alvarez

Behdad. Esfahbod
Christian. Persch
David L. Stevens
Douglas E. Comer
Guilherme de S. Pastve
Havoc. Pennington
Mariano. Suarez-Alvarez

第一次使用自定义的函数作为比较规则，这个规则主要是比较 second name，而第二次排序使用 fun_less_pair 作为排序的比较规则，它首先比较 first name，如果 first name 相等再比较 second name。从结果可以看出第一次排序是根据 second name 排序的，第二次是根据 first name 排序的。

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：

1. stack_t
2. queue_t
3. priority_queue_t

priority_queue_t 是一种队列，队列中优先级最高的数据总是在队列的开头，外部也只是能够访问优先级最高的数据。

第一节 堆栈 stack_t

堆栈类型 stack_t，实现后进先出(LIFO)规则。使用 push()操作向堆栈中添加数据，使用 pop()操作以反向的顺删除堆栈中的数据。

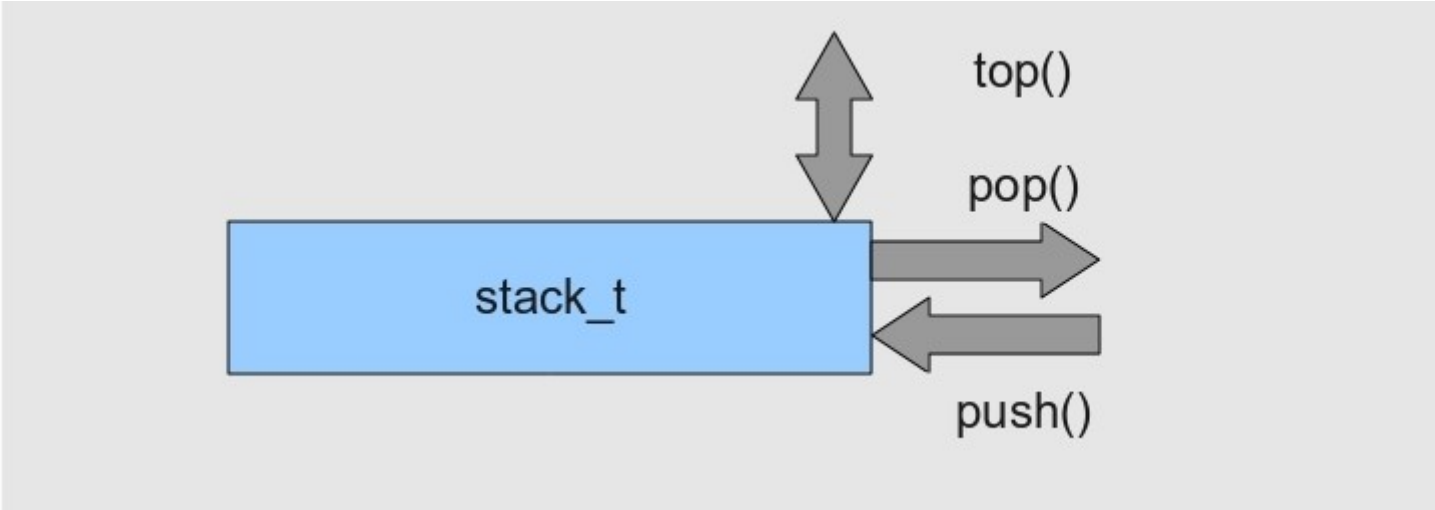

图 8.1 stack_t 的接口

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，默认采用 deque_t 来实现，但是通过修改编译选项可以改变底层实现，具体参考第一章第三节。实现 stack_t 的接口就是将容器对数据的操作限制在了末端，简单调用了容器的 push_back(), pop_back()和 back()操作函数。

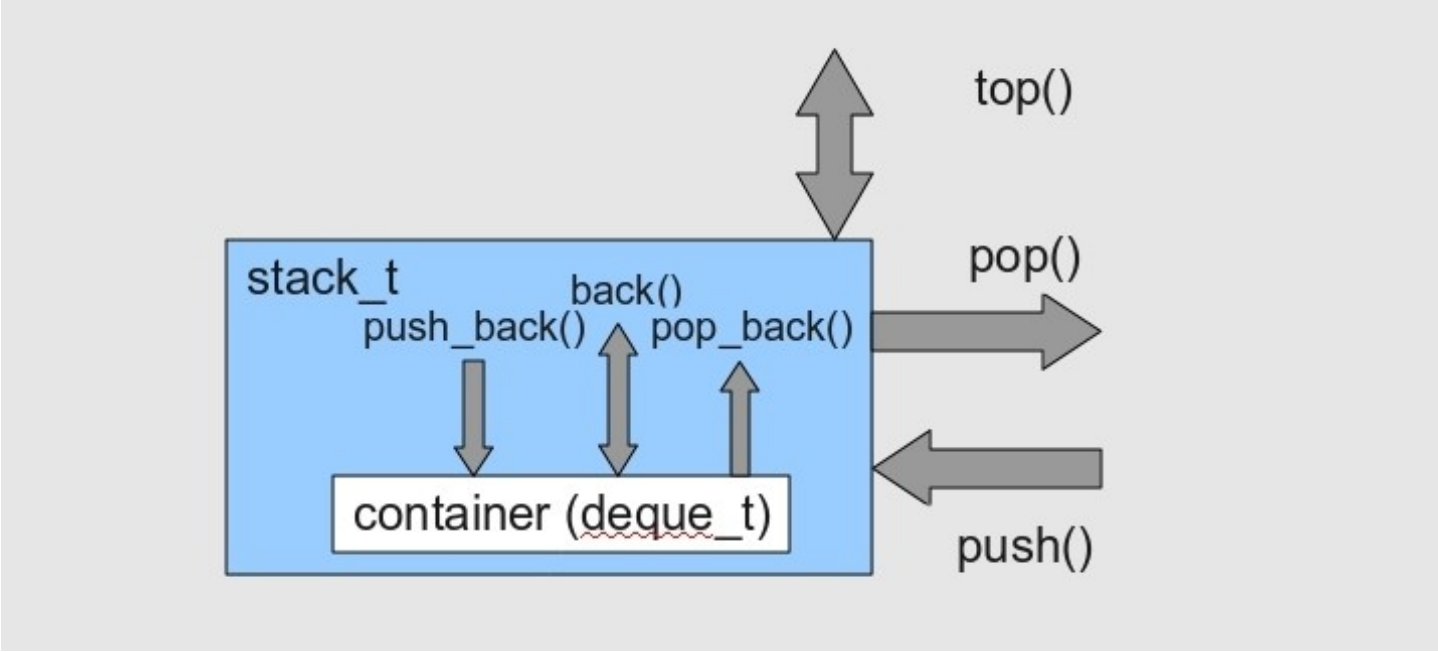

图 8.2 stack_t 接口的内部

1. 核心操作

stack_t 的核心操作函数很简单，就是三个数据操作函数：

- stack_push()：将数据压栈.

- stack_top()：访问栈顶数据.
- stack_pop()：数据出栈.

注意 stack_pop()只是将栈顶数据删除并不返回该数据，而 stack_top()只是访问栈顶数据但是并不会将栈顶数据删除。如果栈中没有数据，那么 stack_pop()和 stack_top()的行为是未定义的，可以使用 stack_empty()或者 stack_size()来确定栈中是否存在数据。

2. stack_t 的使用实例

下面的例子展示了如何使用堆栈：

```
/*
 * stack1.c
 * compile with : -lcstl
 */

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

int main(int argc, char* argv[])
{
    stack_t* pstk_coll = create_stack(int);

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

    stack_init(pstk_coll);

    stack_push(pstk_coll, 1);
    stack_push(pstk_coll, 2);
    stack_push(pstk_coll, 3);

    printf("%d ", *(int*)stack_top(pstk_coll));
    stack_pop(pstk_coll);
    printf("%d ", *(int*)stack_top(pstk_coll));
    stack_pop(pstk_coll);

    *(int*)stack_top(pstk_coll) = 77;
    stack_push(pstk_coll, 4);
    stack_push(pstk_coll, 5);
```

```
    stack_pop(pstk_coll);

    while(!stack_empty(pstk_coll))
    {
        printf("%d ", *(int*)stack_top(pstk_coll));
        stack_pop(pstk_coll);
    }
    printf("\n");

    stack_destroy(pstk_coll);

    return 0;
}
```

程序的输出结果：
3 2 4 77

第二节 队列 queue_t

队列类型 queue_t，实现先进先出(FIFO)规则。使用 push()向队列中添加一个数据，使用 pop()以相同的顺序从队列中删除一个数据。队列就是一个典型的缓冲区。

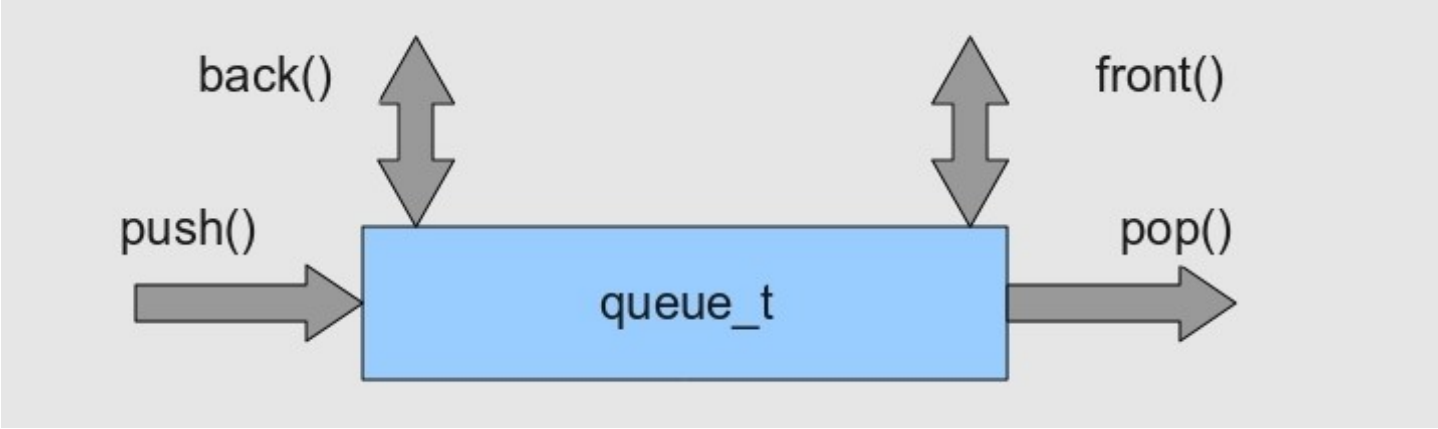

图 8.3 queue_t 的接口

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，默认采用 deque_t 来实现，但是通过修改编译选项可以改变底层实现，具体参考第一章第三节。实现 queue_t 的接口就是将容器对数据的操作限制在了末端，简单调用了容器的 push_back(), pop_front(),front()和 back() 操作函数。

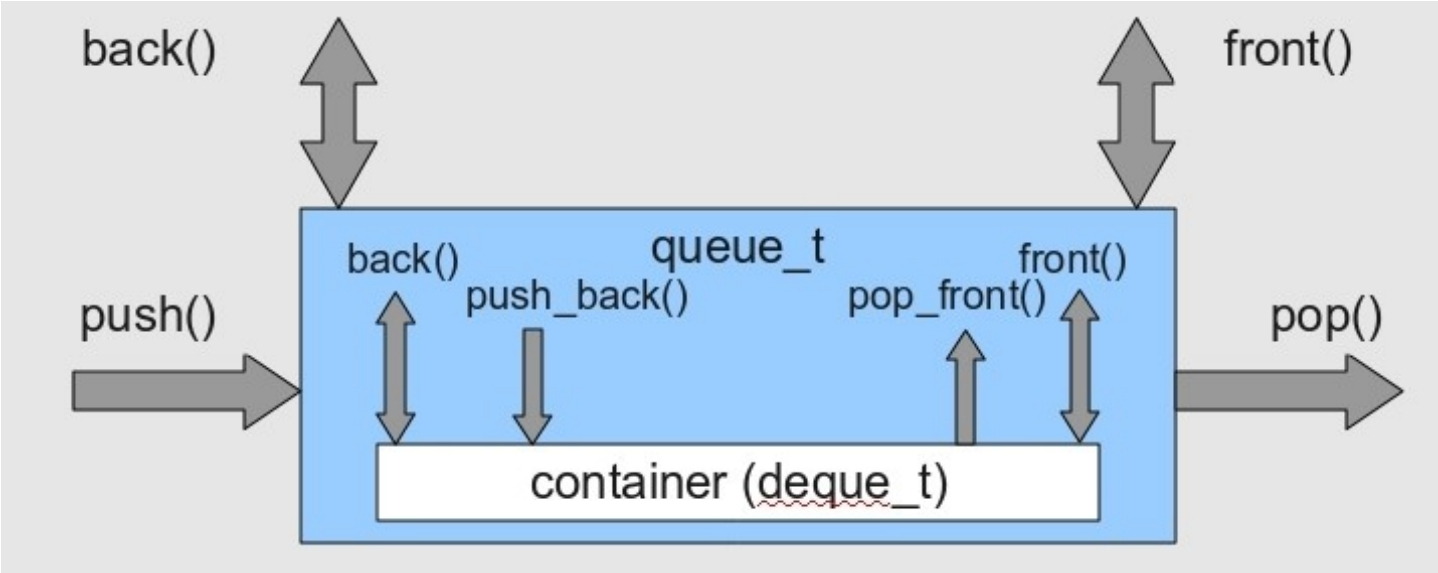

图 8.4 queue_t 接口的内部

1. 核心操作

queue_t 的操作函数很简单，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。

注意 queue_pop()只是删除数据，并不返回该数据，而 queue_front()和 queue_back()也只是访问数据，但不会删除被访问的数据。当 queue_t 为空的时候 queue_pop()，queue_front()和 queue_back()是未定义的，可以使用 queue_size()和 queue_empty()来测试 queue_t 是否为空。

2. queue_t 的使用实例

下面的例子展示了如何使用 queue_t：

```
/*
 * queue1.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cqueue.h>
```

```
int main(int argc, char* argv[])
{
    queue_t* pque_coll = create_queue(char*);

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

    queue_init(pque_coll);

    queue_push(pque_coll, "There ");
    queue_push(pque_coll, "are ");
    queue_push(pque_coll, "more than ");

    printf("%s ", (char*)queue_front(pque_coll));
    queue_pop(pque_coll);
    printf("%s ", (char*)queue_front(pque_coll));
    queue_pop(pque_coll);

    queue_push(pque_coll, "four ");
    queue_push(pque_coll, "words!");
    queue_pop(pque_coll);

    printf("%s ", (char*)queue_front(pque_coll));
    queue_pop(pque_coll);
    printf("%s ", (char*)queue_front(pque_coll));
    queue_pop(pque_coll);
    printf("\n");

    printf("numbers of elements in the queue : %u\n", queue_size(pque_coll));

    queue_destroy(pque_coll);

    return 0;
}
```

程序的输出：

There are four words!

numbers of elements in the queue : 0

第三节 优先队列 priority_queue_t

优先队列类型 priority_queue_t 实现了一个带有优先级的队列类型，位于队列开头的总是优先级最高的数据。使用 priority_queue_push() 向队列中添加数据，使用 priority_queue_pop() 将队列中优先级最高的数据删除，使用 priority_queue_top()访问队列中优先级最高的数据。在插入或者删除数据之后，队列按照一定的规则重新排序，将优先级最高的数据放在队列的开头。

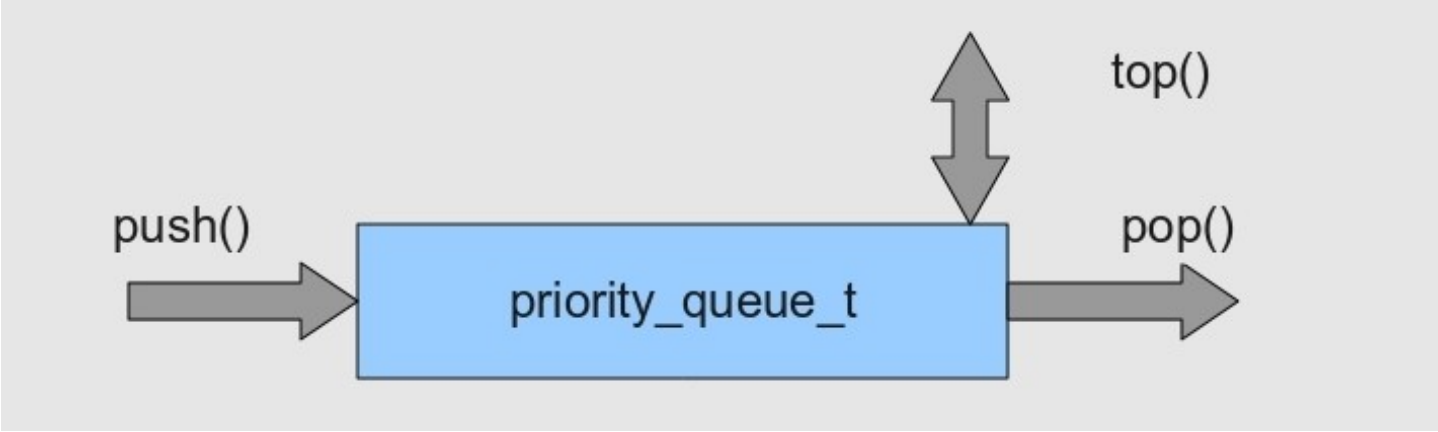

图 8.5 priorioty_queue_t 的接口

要使用 priority_queue_t 必须包含头文件<cstl/cqueue.h>

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

1. 核心操作

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：

1. priority_queue_push()：将数据插入优先队列.
2. priority_queue_top()：访问优先级最高的数据.
3. priority_queue_pop()：弹出优先级最高的数据.

与其他的容器适配器一样 priority_queue_top()只是访问队列中优先级最高的数据，但是并不删除该数据，同时 priority_queue_pop()只是删除队列中优先级最高的数据，但是不能对其进行访问。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。

2. priority_queue_t 的使用实例

下面的例子展示了如何使用 priority_queue_t：

```
/*
```

```
 * pqueue1.c
 * compile with : -lcstl
 */

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

int main(int argc, char* argv[])
{
    priority_queue_t* ppque_coll = create_priority_queue(double);

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

    priority_queue_init(ppque_coll);

    priority_queue_push(ppque_coll, 66.6);
    priority_queue_push(ppque_coll, 22.2);
    priority_queue_push(ppque_coll, 44.4);

    printf("%lf ", *(double*)priority_queue_top(ppque_coll));
    priority_queue_pop(ppque_coll);
    printf("%lf\n", *(double*)priority_queue_top(ppque_coll));
    priority_queue_pop(ppque_coll);

    priority_queue_push(ppque_coll, 11.1);
    priority_queue_push(ppque_coll, 55.5);
    priority_queue_push(ppque_coll, 33.3);

    priority_queue_pop(ppque_coll);

    while(!priority_queue_empty(ppque_coll))
    {
        printf("%lf ", *(double*)priority_queue_top(ppque_coll));
        priority_queue_pop(ppque_coll);
    }
    printf("\n");

    priority_queue_destroy(ppque_coll);

    return 0;
```

```
}
```

程序的输出结果：
66.600000 44.400000
33.300000 22.200000 11.100000

我们可以看出在插入了 66.6，22.2 和 44.4 后输出的是 66.6 和 44.4。在插入三个数据之后 priority_queue_t 中保存着 22.2，11.1，55.5 和 33.3，使用 priority_queue_pop()跳过一个数据之后，打印的结果是 33.3，22.2 和 11.1。

第九章 libcstl 字符串

在这一章中主要讲解 libcstl 提供的字符串类型 string_t，在这一章中使用 string_t 表示 libcstl 提供的字符串类型，对于 C 语言的字符串 char*或者 const char*类型主要使用”字符串”表示，同时向”Hello World”也表示 const char*类型。

第一节 目的和作用

string_t 类型可以向普通容器类型一样使用不必再有烦恼，你可以任意的拷贝，赋值，比较同时不必在担心是否为它分配了足够的内存或者是多长的你存才是有效的。总之你可以向 C 字符串一样使用 string_t 但是不会有 C 字符串使用时出现的烦恼。

这一节通过两个例子来展示 string_t 的这种能力。

1. 第一个例子：提取模板文件名

这个例子使用命令行参数获得文件名模版, 例如输入:
./string1 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：

```
/*
```

```
 * string1.c
 * compile with : -lcstl
 */

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

int main(int argc, char* argv[])
{
    string_t* pstr_filename = create_string();
    string_t* pstr_basename = create_string();
    string_t* pstr_extname = create_string();
    string_t* pstr_tmpname = create_string();
    string_t* pstr_suffix = create_string();
    size_t t_index = 0;
    int i = 0;

    if(pstr_filename == NULL || pstr_basename == NULL ||
       pstr_extname == NULL || pstr_tmpname == NULL ||
       pstr_suffix == NULL)
    {
        return -1;
    }

    string_init(pstr_filename);
    string_init(pstr_basename);
    string_init(pstr_extname);
    string_init(pstr_tmpname);
    string_init_cstr(pstr_suffix, "tmp");

    for(i = 1; i < argc; ++i)
    {
        string_assign_cstr(pstr_filename, argv[i]);

        t_index = string_find_char(pstr_filename, '.', 0);
        if(t_index == NPOS)
        {
            string_assign(pstr_tmpname, pstr_filename);
            string_connect_char(pstr_tmpname, '.');
            string_connect(pstr_tmpname, pstr_suffix);
        }
        else
        {
```

```
            string_assign_substring(pstr_basename, pstr_filename, 0, t_index);
            string_assign_substring(pstr_extname, pstr_filename, t_index + 1, NPOS);
            string_assign(pstr_tmpname, pstr_filename);
            if(string_empty(pstr_extname))
            {
                string_append(pstr_tmpname, pstr_suffix);
            }
            else if(string_equal(pstr_extname, pstr_suffix))
            {
                string_replace_cstr(pstr_tmpname, t_index + 1,
                    string_size(pstr_extname), "xxx");
            }
            else
            {
                string_replace(pstr_tmpname, t_index + 1, NPOS, pstr_suffix);
            }
        }

        printf("%s => %s\n", string_c_str(pstr_filename),
            string_c_str(pstr_tmpname));
    }

    string_destroy(pstr_filename);
    string_destroy(pstr_basename);
    string_destroy(pstr_extname);
    string_destroy(pstr_tmpname);
    string_destroy(pstr_suffix);

    return 0;
}
```

首先是包含头文件

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

接下来是创建创建了四个名字和一个后缀：

```
string_t* pstr_filename = create_string();
string_t* pstr_basename = create_string();
string_t* pstr_extname = create_string();
string_t* pstr_tmpname = create_string();
string_t* pstr_suffix = create_string();
```

然后要判断这个些 string_t 类型是否创建成功了：

```
if(pstr_filename == NULL || pstr_basename == NULL ||
   pstr_extname == NULL || pstr_tmpname == NULL ||
   pstr_suffix == NULL)
```

```
{
    return -1;
}
```

对创建的名字和后缀进行了初始化：

```
string_init(pstr_filename);
string_init(pstr_basename);
string_init(pstr_extname);
string_init(pstr_tmpname);
string_init_cstr(pstr_suffix, "tmp");
```

对 pstr_suffix 的初始化方式与其他的不同，它使用了传统的 C 字符串进行初始化，初始化之后 pstr_suffix 具有了值 tmp 。其他的 string_t 类型初始化后为空串。

接下来对于命令行中的每一个参数进行处理，优先将参数赋值给 pstr_filename。

```
string_assign_cstr(pstr_filename, argv[i]);
```

命名行参数都是 C 字符串类型，这里看到使用 C 字符串类型可以直接给 string_t 类型赋值。

表达式：

```
t_index = string_find_char(pstr_filename, '.', 0);
```

是从 pstr_filename 的开头查找字符'.'出现的第一个位置，返回这个索引。最后的 0 表示从 pstr_filename 的第一个字符差事查找，string_t 中的字符下标也是从 0 开始的。然后通过判断返回值就可以知道 pstr_filename 是否包含'.'字符了：

```
if(t_index == NPOS)
```

这里 NPOS 表示比 string_t 中保存的字符范围以外，这是 string_t 类型定义的一个常量，用它来表示字符最大的长度或者是查找失败的情况。应该总是使用 size_t 类型的数据与这个值进行比较，否则行为是未定义的。

如果在 pstr_filename 中不包含'.'，那么使用下面的语句来生成 pstr_tmpname：

```
string_assign(pstr_tmpname, pstr_filename);
string_connect_char(pstr_tmpname, '.');
string_connect(pstr_tmpname, pstr_suffix);
```

可以看到使用 string_t 类型互相赋值，同时还可以在 string_t 类型后面链接 string_t 类型，甚至是单个字符'.'。

如果参数中包含'.'那么就使用下面的操作：

```
string_assign_substring(pstr_basename, pstr_filename, 0, t_index);
string_assign_substring(pstr_extname, pstr_filename, t_index + 1, NPOS);
string_assign(pstr_tmpname, pstr_filename);
```

在这里我们看到除了使用 string_t 类型互相赋值值为，还可以使用 string_t 的一部分来赋值。上面的操作函数都是使用字符下标和长度表示 string_t 类型的一部分，我们看到了 NPOS 的身影，它除了可以表示最大的下标外还可以表示最大的长度，使用 NPOS 表示 string_t 从某一下标开始到结尾所组成的 string_t 的部分。

接下来判断了以下 pstr_extname 是否为空：

```
if(string_empty(pstr_extname))
```

如果为空我们就将后缀直接添加在 pstr_tmpname 的后面：

```
string_append(pstr_tmpname, pstr_suffix);
```

可以看到这里使用了另一种链接字符串的操作函数。

如果参数带有后缀并且后缀和我们使用的后缀相同，那么我们就用”xxx”来代替它。

```
else if(string_equal(pstr_extname, pstr_suffix))
{
    string_replace_cstr(pstr_tmpname, t_index + 1, string_size(pstr_extname), "xxx");
}
```

这里使用了 string_equal(pstr_extname, pstr_suffix)比较了两个 string_t 类型是否相等，有使用了 string_replace_cstr 用 C 字符串直接替换 string_t 中的一部分字符。
第三种情况就是参数使用了后缀，但是不是我们要使用的后缀，我们要用我们使用的后缀将现存的后缀替换：

```
string_replace(pstr_tmpname, t_index + 1, NPOS, pstr_suffix);
```

这里再次看到了 NPOS 的身影，它表示将 pstr_tmpname 中从 t_index+1 开始的一直到末尾的全部字符都替换成 pstr_suffix 中的字符。
最后将转换前后的效果对比输出：

```
printf("%s => %s\n", string_c_str(pstr_filename), string_c_str(pstr_tmpname));
```

这里使用了 string_c_str(pstr_filename)以 C 字符串的方式返回 string_t 中保存的字符串。

2. 第二个例子：提取单词并逆序打印

第二个例子是从标准输入中提取出单词并使用逆序的方式输出，单词都是使用标准的空白 (空格，换行或 tab) 和逗号，点号或者分号分割的：

```
/*
 * string2.c
 * compile with : -lcstl
 */

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

int main(int argc, char* argv[])
{
    string_t* pstr_delims = create_string();
    string_t* pstr_line = create_string();

    if(pstr_delims == NULL || pstr_line == NULL)
    {
        return -1;
    }

    string_init_cstr(pstr_delims, " \t,.;");
    string_init(pstr_line);

    while(string_getline(pstr_line, stdin))
    {
        size_t t_begin = 0;
        size_t t_end = 0;
        int i = 0;
```

```
        t_begin = string_find_first_not_of(pstr_line, pstr_delims, 0);
        while(t_begin != NPOS)
        {
            t_end = string_find_first_of(pstr_line, pstr_delims, t_begin);
            if(t_end == NPOS)
            {
                t_end = string_length(pstr_line);
            }

            for(i = t_end - 1; i >= (int)t_begin; --i)
            {
                printf("%c", *string_at(pstr_line, i));
            }
            printf(" ");

            t_begin = string_find_first_not_of(pstr_line, pstr_delims, t_end);
        }
        printf("\n");
    }

    string_destroy(pstr_delims);
    string_destroy(pstr_line);

    return 0;
}
```

在这个例子中所有作为单词的字符串都是使用下面的字符分割的：

```
string_init_cstr(pstr_delims, " \t,.;");
```

换行也用来分割单词，但是由于我们的程序是安行读取的，所以不用特殊的指明：

```
string_init(pstr_line);
while(string_getline(pstr_line, stdin))
```

string_getline 是一个特殊的函数，它将输入读取到 string_t 中，默认使用换行作为结束。使用 string_getline_delimiter 还可以自己指定结束符。在循环中，将每个单词分离出来并打印。第一个语句：

```
t_begin = string_find_first_not_of(pstr_line, pstr_delims, 0);
```

查找第一个单词的开头，string_find_first_not_of 找到第一个不是 pstr_delims 中的分隔符的字符的位置。与其他的查找操作函数一样，如果没有符合条件的字符就返回 NPOS，我们以这个作为内存循环的结束标志：

```
while(t_begin != NPOS)
```

在内层循环中首先要找到的就是一个单词的结尾：

```
t_end = string_find_first_of(pstr_line, pstr_delims, t_begin);
```

string_find_first_of 返回从 t_begin 开始的第一个出现的分割字符，同样如果没有符合条件的字符，那就说明已经是最后一个单词了：

```
if(t_end == NPOS)
```

```
{
    t_end = string_length(pstr_line);
}
```
这是我们就要将 t_end 指向 string_t 中字符的实际末尾。
接下来就要以逆序的形式打印出单词了：
```
for(i = t_end - 1; i >= (int)t_begin; --i)
{
    printf("%c", *string_at(pstr_line, i));
}
```
我们使用了 string_at 操作函数它返回 string_t 中指定下标的字符的指针。同时还要注意这个循环的判断条件：
```
i >= (int)t_begin;
```
要将 t_begin 强制转换成 int 类型，否则这个表达式永远是 true。
下面要查找第二个单词了：
```
t_begin = string_find_first_not_of(pstr_line, pstr_delims, t_end);
```

使用下列输入来测试这段代码：
pots & pans
I saw a reed
程序的输出为：
stop & snap
I was a deer

第二节 string_t 的操作函数

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

下面列出了 string_t 类型提供的所有操作：

- 初始化和销毁
- C 字符串操作

- 字符数目和存储能力
- 字符访问
- 关系操作和比较操作
- 赋值
- 交换
- 添加和链接
- 插入
- 删除
- 替换
- 子串
- 输入和输出
- 查找
- 迭代器支持

string_t 提供的操作不仅支持 string_t 参数，同时还自持部分 string_t，C 字符串，和部分 C 字符串，可以从操作函数的后缀中看出它支持的参数类型：

后缀	参数形式	参数含义
--	const string_t* cpstr_string	整个 string_t 类型。
substring	const string_t* cpstr_string, size_t t_pos, size_t t_len	string_t 中从 t_pos 开始的长 t_len 的一部分。
cstr	const char* s_cstr	整个 C 字符串。
subcstr	const char* s_cstr, size_t t_len	C 字符串中前 t_len 个字符。
char	size_t t_count, char c_char	t_count 个字符。
range	string_iterator_t it_begin, string_iterator_t it_end	数据区间[it_begin, it_end)中的字符。

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	subcstr	char	range
初始化和销毁	Yes	Yes	Yes	Yes	Yes	Yes
C 字符串操作	No	No	No	No	No	No
数目和能力	No	No	No	No	No	No
字符访问	No	No	No	No	No	No
关系和比较	Yes	Yes	Yes	Yes	No	No
赋值	Yes	Yes	Yes	Yes	Yes	Yes
交换	Yes	No	No	No	No	No

添加和链接	Yes	Yes	Yes	Yes	Yes	Yes
插入	Yes	Yes	Yes	Yes	Yes	Yes
删除	No	Yes	No	No	No	Yes
替换	Yes	Yes	Yes	Yes	Yes	Yes
子串	No	No	No	No	No	No
输入和输出	No	No	No	No	No	No
查找	Yes	No	Yes	Yes	Yes	No

3. 初始化和销毁

string_t 类型与其他容器类型一致，在使用前也要创建和初始化，使用之后要销毁。但是 string_t 的创建函数与其他的类型不同，它不需要参数，因为 string_t 中保存的类型就是 char 字符类型。string_t 有多种多样的初始化函数，下面类出了所有的初始化函数：

string_init	初始化一个空的 string_t。
string_init_cstr	使用 C 字符串初始化 string_t。
string_init_subcstr	使用部分 C 字符串初始化 string_t。
string_init_char	使用多个字符初始化 string_t。
string_init_copy	使用另一个 string_t 来初始化 string_t。
string_init_copy_substring	使用另一个 string_t 的一部分初始化 string_t。
string_init_copy_range	使用数据区间初始化 string_t。

4. string_t 与 C 字符串

string_t 保存的是 char 类型的数据，本身又实现的是字符串的语义，所以它要和传统的 C 字符串之间能够互相转换，string_t 类型提供了这样的操作函数。

从上面的章节可以看到，从 C 字符串可以很轻松的转换到 string_t 类型，可以通过使用 C 字符串的初始化和一切支持 C 字符串和支持 C 字串的函数来实现。反过来，从 string_t 转换到 C 字符串可以使用下面三个操作函数：

string_c_str	返回一个带有'\0'结尾的 C 字符串。
string_data	返回 string_t 中保存的字符数组。
string_copy	将子串拷贝到指定的缓冲区中。

string_c_str 和 string_data 都是返回 string_t 中保存的数据，只不过形式不同，例如：

func1(string_c_str());
这样调用是可行的，因为 string_c_str()返回的是带有'\0'结束符的 C 字符串，但是
func1(string_data());
这样就不可行了，因为 string_data()返回的是不带有'\0'的字符数组，如果运气好，这个字符数组后面跟的就是'\0'，那么就没什么问题，要不是这样那么这个结果就未定义了，所以要像下面这样使用 string_data()操作函数：
func2(string_dat(), string_length());

此外要注意的是用户一定不能修改这个数据或者释放掉这块内存。另外在 string_t 中的字符增加之后，以前通过 string_c_str()和 string_data()获得的指针可能失效：

```
string_init_cstr(pstr_sample, "abcde");
const char* p = string_c_str();
func(p);
string_append_cstr(pstr_sample, "ABCDE");
func(p);
```

上面代码中第二次 func(p);中 p 可能已经失效了，因为在 string_append_cstr(pstr_sample, "ABCDEF");之后，保存数据的内存可能已经重新分配，那么这个 p 就是无效的。

5. 大小和容量

与 vector_t 类型相似，string_t 也提供关于数据个数和容量的操作函数：

string_size	返回 string_t 中保存的字符的个数。
string_size	返回 string_t 中保存的字符的个数。
string_empty	测试 string_t 是否为空。
string_max_size	返回 string_t 中能够保存字符的最大个数。
string_capacity	返回 string_t 在不重新分配内存的情况下能够保存字符的个数。
string_reserve	重新设置 string_t 的容量。
string_resize	重新设置 string_t 中字符的个数。

与 vector_t 相比 string_t 多出了 string_length()操作函数，但是这个函数与 string_size()是相同的。

6. 数据访问

string_at	对 string_t 中的数据进行随机访问。

与 vector_at()相同，使用超出范围的下标进行访问，函数的行为是未定义的。string_at()也会遇到上面介绍的 string_c_str()和 string_data()的问题，就是当 string_t 中字符增加之后，原有的指针可能已经失效了。

7. 关系操作和比较操作

string_t 不仅提供了两个 string_t 类型之间的关系操作函数，还提供了 string_t 与 C 字符串之间的关系操作。此外 string_t 还提供了比较操作函数，它的返回值是三态的：

string_equal	测试两个 string_t 是否相等。
string_not_equal	测试两个 string_t 是否不等。
string_less	测试第一个 string_t 是否小于第二个 string_t。
string_less_equal	测试第一个 string_t 是否小于等于第二个 string_t。
string_greater	测试第一个 string_t 是否大于第二个 string_t。
string_greater_equal	测试第一个 string_t 是否大于等于第二个 string_t。
string_equal_cstr	测试 string_t 是否等于 C 字符串。
string_not_equal_cstr	测试 string_t 是否不等于 C 字符串。
string_less_cstr	测试 string_t 是否小于 C 字符串。
string_less_equal_cstr	测试 string_t 是否小于等于 C 字符串。
string_greater_cstr	测试 string_t 是否大于 C 字符串。
string_greater_equal_cstr	测试 string_t 是否大于等于 C 字符串。
string_compare	比较两个 string_t。
string_compare_substring_string	比较子 string_t 和 string_t。
string_compare_substring_substring	比较两个子 string_t。
string_compare_cstr	比较 string_t 和 C 字符串。
string_compare_substring_cstr	比较子 string_t 和 C 字符串。
string_compare_substring_subcstr	比较子 string_t 和 C 子字符串。

关系操作和比较操作的主要区别在于比较操作通过返回值来判断两个比较对象之间的关系，同时比较函数还提供了子串和 C 子字符串的比较功能，这是关系操作没有提供的。

8. 赋值

可以通过赋值操作函数来改变 string_t 的值，而 string_t 提供了多种赋值操作函数：

string_assign	使用 string_t 赋值。
string_assign_substring	使用子 string_t 赋值。
string_assign_cstr	使用 C 字符串赋值。

string_assign_subcstr	使用 C 子字符串赋值。
string_assign_char	使用多个字符赋值。
string_assign_range	使用指定的数据区间赋值。

9. 数据交换

这个操作同其他容器的数据交换相同，但是 string_t 并没有提供针对子 string_t，C 字符串和数据区间的交换操作函数：

string_swap	交换两个 string_t。

10. 添加和链接

string_t 最常用的操作函数就是两个将 string_t 合并或者向一个 string_t 末尾添加字符串。string_t 提供了多种多样的添加和链接的操作函数，不仅可以实现两个 string_t 的链接，还可以向 string_t 中添加子 string_t，C 字符串，多个字符或者数据区间，甚至还可以一次只添加一个字符：

string_append	向 string_t 末尾添加 string_t。
string_append_substring	向 string_t 末尾添加子 string_t。
string_append_cstr	向 string_t 末尾添加 C 字符串。
string_append_subcstr	向 string_t 末尾添加 C 子字符串。
string_append_char	向 string_t 末尾添加多个字符。
string_append_range	向 string_t 末尾添加指定数据区间中的字符。
string_connect	将两个 string_t 链接起来。
string_connect_cstr	将 string_t 和 C 字符串链接起来。
string_connect_char	将 string_t 和单个字符链接起来。
string_push_back	向 string_t 末尾添加一个字符。

11. 插入数据

向 string_t 中插入数据，可以使用下标表示插入位置，还可以使用迭代器来表示插入位置。不仅可以插入 string_t 还可以插入子 string_t，C 字符串和数据区间：

string_insert	向 string_t 指定位置插入一个字符。
string_insert_n	向 string_t 指定位置插入多个字符。
string_insert_string	向 string_t 指定位置插入一个 string_t。
string_insert_substring	向 string_t 指定位置插入一个子 string_t。
string_insert_cstr	向 string_t 指定位置插入 C 字符串。
string_insert_subcstr	向 string_t 指定位置插入 C 子字符串。
string_insert_char	向 string_t 指定位置插入多个字符。
string_insert_range	向 string_t 指定位置插入一个数据区间中的字符。

12. 数据删除

向插入操作一样，你也可以在删除操作中使用下标和迭代器表示删除数据的位置：

string_erase	删除 string_t 中指定位置的字符。
string_erase_range	删除 string_t 中指定的数据区间中的字符。
string_erase_substring	删除 string_t 中指定的子 string_t。
string_clear	清空 string_t 中所有字符。

13. 替换

string_t 提供了功能全面，使用方便的替换函数：

string_replace	使用 string_t 替换指定的子 string_t。
string_replace_substring	使用子 string_t 替换指定的子 string_t。
string_replace_cstr	使用 C 字符串替换指定的子 string_t。
string_replace_subcstr	使用 C 子字符串替换指定的子 string_t。
string_replace_char	使用多个字符替换指定的子 string_t。
string_range_replace	使用 string_t 替换指定的数据区间。
string_range_replace_substring	使用子 string_t 替换指定的数据区间。
string_range_replace_cstr	使用 C 字符串替换指定的数据区间。
string_range_replace_subcstr	使用 C 子字符串替换指定的数据区间。
string_range_replace_char	使用多个字符替换指定的数据区间。

string_replace_range	使用数据区间替换指定的数据区间。

14. 子串

这个操作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_substr	返回子串。

这个操作返回的是已经初始化的 string_t 的指针，你可以直接使用它，不用初始化，但是在使用之后用户要负责销毁，否则后造成内存泄漏。

15. 输入输出

string_output	将 string_t 中的字符串输出到指定的流中。
string_input	将流中的数据保存在 string_t 中。
string_getline	从指定的流中获得一行数据。
string_getline_delimiter	从指定的流中获取一行数据，使用用户指定的换行符。

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(pstr_sample, 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(pstr_sample, stdin, ':'))
{
...
}
```

16. 查找

查找操作是 string_t 提供的功能最强大的操作函数，包含多种版本，接受多种参数形式：

string_find	从指定位置开始查找 string_t 出现的第一个位置。
string_find_cstr	从指定位置开始查找 C 字符串出现的第一个位置。
string_find_subcstr	从指定位置开始查找 C 子字符串出现的第一个位置。

string_find_char	从指定位置开始查找字符出现的第一个位置。
string_rfind	从指定位置开始向前查找 string_t 出现的最后一个位置。
string_rfind_cstr	从指定位置开始向前查找 C 字符串出现的最后一个位置。
string_rfind_subcstr	从指定位置开始向前查找 C 子字符串出现的最后一个位置。
string_rfind_char	从指定位置开始向前查找字符出现的最后一个位置。
string_find_first_of	从指定位置开始查找 string_t 中任意字符出现的第一个位置。
string_find_first_of_cstr	从指定位置开始查找 C 字符串中任意字符出现的第一个位置。
string_find_first_of_subcstr	从指定位置开始查找 C 子字符串中任意字符出现的第一个位置。
string_find_first_of_char	从指定位置开始查找字符出现的第一个位置。
string_find_first_not_of	从指定位置开始查找 string_t 以外的任意字符出现的第一个位置。
string_find_first_not_of_cstr	从指定位置开始查找 C 字符串以外的任意字符出现的第一个位置。
string_find_first_not_of_subcstr	从指定位置开始查找 C 子字符串以外的任意字符出现的第一个位置。
string_find_first_not_of_char	从指定位置开始查找指定字符以外的任意字符出现的第一个位置。
string_find_last_of	从指定位置开始向前查找 string_t 中任意字符出现的最后一个位置。
string_find_last_of_cstr	从指定位置开始向前查找 C 字符串中任意字符出现的最后一个位置。
string_find_last_of_subcstr	从指定位置开始向前查找 C 子字符串中任意字符出现的最后一个位置。
string_find_last_of_char	从指定位置开始向前查找字符出现的最后一个位置。
string_find_last_not_of	从指定位置开始向前查找 string_t 以外的任意字符出现的最后一个位置。
string_find_last_not_of_cstr	从指定位置开始向前查找 C 字符串以外的任意字符出现的最后一个位置。
string_find_last_not_of_subcstr	从指定位置开始向前查找 C 子字符串以外的任意字符出现的最后一个位置。
string_find_last_not_of_char	从指定位置开始向前查找指定字符以外的任意字符出现的最后一个位置。

总体上来说，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_of，find_last_not_of 类函数。

17. 迭代器支持

string_begin	返回 string_t 开始位置的迭代器。
string_end	返回 string_t 末尾位置的迭代器。

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。

18. NPOS

查找操作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？<cstl/cstring.h>中定义了值NPOS表示查找失败，例如：

```
size_t t_pos = string_find_cstr(pstr_sample, "not found");
if(t_pos == NPOS)
{
...
}
```

此外NPOS用在接受下标为参数的操作函数中时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的时间，同时使用更方便，如本章中的第一个例子：

```
string_replace_cstr(pstr_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”。

第三节 string_t的使用实例

我们使用两个了例子开始了本章的讨论，在这两个例子中使用都是的基于下标的位置表示，下面给出几个使用迭代器来表示位置的例子，通过讨论这几个例子来结束本章。

通过string_t和算法很容器实现对于大小写字符的转换：

```
/*
 * string3.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <ctype.h>
#include <cstl/cstring.h>
#include <cstl/calgorithm.h>

static void _to_lower(const void* cpv_input, void* pv_output)
{
    *(char*)pv_output = tolower(*(char*)cpv_input);
}

static void _to_upper(const void* cpv_input, void* pv_output)
{
```

```
    *(char*)pv_output = toupper(*(char*)cpv_input);
}

int main(int argc, char* argv[])
{
    string_t* pstr_s = create_string();

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

    string_init_cstr(pstr_s, "The tell of Office is 123456789.\n");
    printf("original: ");
    string_output(pstr_s, stdout);

    algo_transform(string_begin(pstr_s), string_end(pstr_s),
        string_begin(pstr_s), _to_lower);

    printf("lowered: ");
    string_output(pstr_s, stdout);

    algo_transform(string_begin(pstr_s), string_end(pstr_s),
        string_begin(pstr_s), _to_upper);

    printf("uppered: ");
    string_output(pstr_s, stdout);

    string_destroy(pstr_s);

    return 0;
}
```

程序的输出结果：

original: The tell of Office is 123456789.

lowered: the tell of office is 123456789.

uppered: THE TELL OF OFFICE IS 123456789.

下面的例子让你可以自定义规则进行查找：

```
/*
 * string4.c
 * compile with : -lcstl
 */
```

```
#include <stdio.h>
#include <ctype.h>
#include <cstl/cstring.h>
#include <cstl/calgorithm.h>

static void _nocase_compare(const void* cpv_first,
    const void* cpv_second, void* pv_output)
{
    *(bool_t*)pv_output =
        toupper(*(char*)cpv_first) == toupper(*(char*)cpv_second) ?
        true : false;
}

int main(int argc, char* argv[])
{
    string_t* pstr_s1 = create_string();
    string_t* pstr_s2 = create_string();
    string_iterator_t it_pos;

    if(pstr_s1 == NULL || pstr_s2 == NULL)
    {
        return -1;
    }

    string_init_cstr(pstr_s1, "This is a string");
    string_init_cstr(pstr_s2, "STRING");

    if(string_size(pstr_s1) == string_size(pstr_s2) &&
       algo_equal_if(string_begin(pstr_s1), string_end(pstr_s1),
            string_begin(pstr_s2), _nocase_compare))
    {
        printf("the strings are equal.\n");
    }
    else
    {
        printf("the strings are not equal.\n");
    }

    it_pos = algo_search_if(string_begin(pstr_s1), string_end(pstr_s1),
        string_begin(pstr_s2), string_end(pstr_s2), _nocase_compare);
    if(iterator_equal(it_pos, string_end(pstr_s1)))
    {
        printf("s2 is not a substring of s1.\n");
```

```
    }
    else
    {
        printf("\"%s\" is a substring of \"%s\" (at index %d).\n",
            string_c_str(pstr_s2), string_c_str(pstr_s1),
            iterator_distance(string_begin(pstr_s1), it_pos));
    }

    string_destroy(pstr_s1);
    string_destroy(pstr_s2);

    return 0;
}
```

这个程序的输出：

the strings are not equal.

"STRING" is a substring of "This is a string" (at index 10).

下面这个例子展示了将某些算法应用到 string_t 类型的效果：

```
/*
 * string5.c
 * compile with : -lcstl
 */

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

int main(int argc, char* argv[])
{
    string_t* pstr_hello = create_string();
    string_t* pstr_s = create_string();
    string_iterator_t it_pos;

    if(pstr_hello == NULL || pstr_s == NULL)
    {
        return -1;
    }

    string_init_cstr(pstr_hello, "Hello, how are you?");
    string_init_copy_range(pstr_s,
        string_begin(pstr_hello), string_end(pstr_hello));

    for(it_pos = string_begin(pstr_s);
```

```
        !iterator_equal(it_pos, string_end(pstr_s));
        it_pos = iterator_next(it_pos))
    {
        printf("%c", *(char*)iterator_get_pointer(it_pos));
    }
    printf("\n");

    algo_reverse(string_begin(pstr_s), string_end(pstr_s));
    printf("reverse: %s\n", string_c_str(pstr_s));

    algo_sort(string_begin(pstr_s), string_end(pstr_s));
    printf("ordered: %s\n", string_c_str(pstr_s));

    string_erase_range(pstr_s,
        algo_unique(string_begin(pstr_s), string_end(pstr_s)),
        string_end(pstr_s));
    printf("uniqued: %s\n", string_c_str(pstr_s));

    string_destroy(pstr_hello);
    string_destroy(pstr_s);

    return 0;
}
```

这个程序的输出：

Hello, how are you?
reverse: ?uoy era woh ,olleH
ordered: ,?Haeehlloooruwy
uniqued: ,?Haehloruwy

第十章 类型机制

libcstl 提供了类型机制，它大大提升了 libcstl 对与类型的处理能力，通过类型机制可以处理任何类型的数据。本章详细的介绍的类型机制中类型的注册和赋值，容器等类型的创建函数的类型描述机制。

第一节 类型注册和复制

在编程的过程中大部分处理的并不是如 int，double 这样的类型，而是很多自己定义的结构体，联合等等这些自定义类型。如果要在容器中保存这样的类型会是什么样的情况呢：

```
/*
 * type1.c
 * compile with : -lcstl
 */

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

typedef struct _tagabc
{
    int n_elem;
}abc_t;

static void _abc_init(const void* cpv_input, void* pv_output)
{
    ((abc_t*)cpv_input)->n_elem = 0;
    *(bool_t*)pv_output = true;
}

static void _abc_copy(const void* cpv_first,
    const void* cpv_second, void* pv_output)
{
    ((abc_t*)cpv_first)->n_elem = ((abc_t*)cpv_second)->n_elem;
    *(bool_t*)pv_output = true;
}

static void _abc_less(const void* cpv_first,
    const void* cpv_second, void* pv_output)
{
```

```
    *(bool_t*)pv_output =
        ((abc_t*)cpv_first)->n_elem < ((abc_t*)cpv_second)->n_elem ?
        true : false;
}

static void _abc_destroy(const void* cpv_input, void* pv_output)
{
    ((abc_t*)cpv_input)->n_elem = 0;
    *(bool_t*)pv_output = true;
}

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

    printf("before type register: ");
    if(plist_coll == NULL)
    {
        printf("creation of abc_t type container failed!\n");
    }
    else
    {
        printf("creation of abc_t type container success!\n");
    }

    type_register(abc_t, _abc_init, _abc_copy, _abc_less, _abc_destroy);
    plist_coll = create_list(abc_t);

    printf("after type register: ");
    if(plist_coll == NULL)
    {
        printf("creation of abc_t type container failed!\n");
    }
    else
    {
        printf("creation of abc_t type container success!\n");
    }

    return 0;
}
```

这个程序输出：

before type register: creation of abc_t type container failed!

after type register: creation of abc_t type container success!

看看上面这个例子，我们自定义了一个类型 abc_t，然后要创建一个保存这个类型的 list_t：

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

然后检查这个 list_t 是否创建成功了，很遗憾这次检查的结果是 plist_coll == NULL，接下来我们使用类型注册函数将这个类型注册：

```
type_register(abc_t, _abc_init, _abc_copy, _abc_less, _abc_destroy);
```

然后再使用同样的方式创建这个 list_t，这次的结果是创建成功了 plist_coll != NULL。从上面的例子可看的出来，当要在容器等类型中保存一个新的用户自定义类型时必须首先将这个注册，这样在创建保存该自定义类型的容器时不会失败，如果保存没有注册的数据类型，容器的创建函数就会失败。

在具体介绍类型注册函数之前首先来看看 libcstl 对类型的分类。

1. 类型分类和类型机制

libcstl 对保存的数据类型进行了分类：

- C 内建类型：包括 int, double 等 C 语言基本类型和 C 字符串类型。
- 用户自定义类型：用户定义的结构体，枚举，联合还有重定义的类型。
- libcstl 内建类型：libcstl 提供的容器，迭代器，工具类型等等。

其中 libcstl 内建类型属于用户自定义类型的特殊形式。这些类型也是 libcstl 所接受的说有数据类型。

libcstl 库内部有一个类型注册表，记录所有已经注册的类型。当创建一个容器是，创建函数接受一个类型的描述(类型的描述语法在下一节中介绍)。通过分析类型的描述并获得具体的数据类型，然后在注册表中查找相应的类型，如果类型已经注册，那么在类型注册表中找到该类型并获得该类型的相关信息如初始化函数，拷贝函数，比较函数，销毁函数等。如果一个类型并没有注册，那么创建函数就失败了。这个就是 libcstl 的类型机制，当第一个创建函数被调用的时候，这时候注册表中没有任何注册类型，libcstl 调用注册表的初始化函数将默认注册的类型注册到注册表中，然后再执行查询，在后续的创建函数被调用的时候，注册表已经初始化所以不用再初始化就可以直接查询的。

在初始化注册表的时候有默认注册的类型被注册了，那么什么样的类型被注册了呢，C 内建类型和出去 range_t 的 libcstl 内建类型都被注册了。因为 libcstl 内建类型也都是用户自定义类型但是它们是默认注册的，所以说 libcstl 内建类型是特殊的自定义类型。

通过上面的解释就很好理解例子中的现象，在第一次创建 list_t 类型时，abc_t 类型并没有注册，所以创建函数在注册表中找不到相应的信息，所以创建失败。当使用 type_register()将 abc_t 注册之后再次创建时，创建函数就在注册表中找到了 abc_t 相应的信息，创建就成功了。

那么为什么不在初始化注册表的同时将用户自定义类型也注册到注册表中呢？这是因为对于用户自定义类型来说，libcstl 并不知道它的任何信息包括名字，初始化函数，拷贝函数，小于比较函数和销毁函数。对于 range_t 类型，libcstl 只是知道它的名字，但是不知道它的操作函数，所以也么有注册。这样一来用户自定义类型就必须由用户自己注册，在注册的时候必须提供名字，初始化函数，拷贝函数，小于比较函数和销毁函数。

2. 类型注册函数 type_register

用户自定义类型的函数只能用户自己注册，其实最了解这些类型的也就是用户自身。可以通过调用类型注册函数 type_register 来注册自定义类型。这个函数需要 5 个参数，那就是注册一个类型的 5 个要素：

- 类型名字
- 初始化函数
- 拷贝函数
- 小于比较函数
- 销毁函数

类型名字必须在注册表中是唯一的，如果将一个已经注册的类型再次注册将导致注册失败，但是并不会影响已经注册的类型的使用，也就是说一个类型一旦注册就不能再更改了。如果可以更改的话，一个类型在更改前后保存它的容器的操作函数的行为就可能不同，应用在其上的算法的行为也有可能不同。

接下来就是四个操作函数，这个四个函数都必须是谓词，初始化函数和销毁函数是一元谓词，拷贝函数和小于操作函数是二元谓词。而且参数的顺序必须是第一个是初始化函数，第二个是拷贝函数，第三个是小于比较函数，第四个是销毁函数。例如例子中的注册函数：

```
type_register(abc_t, _abc_init, _abc_copy, _abc_less, _abc_destroy);
```

需要注意的是如果在自定义类型中包含需要分配你存的指针时，那么这些操作函数就必须十分小心，你必须在初始化函数中分配内存，因为只是唯一可以分配内存的地方也是对于一个数据只调用一次的函数，类似于 C++中的构造函数。在销毁函数中将内存释放掉，类似与 C++的析构函数。同样在拷贝函数中也要注意有类似 C++浅拷贝的问题。

当注册成功 type_register()返回 true，否则返回 false。

3. 类型复制函数 type_duplicate

首先让我们来看看下面的例子：

```
/*
 * type2.c
 * compile with : -lcstl
 */

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

typedef unsigned int uint_t;

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

    printf("before type duplication: ");
    if(plist_coll == NULL)
    {
        printf("creation of uint_t type container failed!\n");
    }
    else
```

```
    {
        printf("creation of uint_t type container success!\n");
    }

    type_duplicate(unsigned int, uint_t);
    plist_coll = create_list(uint_t);

    printf("after type duplication: ");
    if(plist_coll == NULL)
    {
        printf("creation of uint_t type container failed!\n");
    }
    else
    {
        printf("creation of uint_t type container success!\n");
    }

    return 0;
}
```

这个程序的输出结果：

before type duplication: creation of uint_t type container failed!

after type duplication: creation of uint_t type container success!

在程序的开头我们重定义了类型 unsigned int，让后创建一个保存 uint_t 类型的 list_t，但是创建失败了。在调用了类型复制函数后再次创建就成功了。

为什么会这样，unsigned int 和 uint_t 不是用一种类型吗？unsigned int 不是已经注册了吗？是的 unsigned int 已经注册了，但是 unsigned int 和 uint_t 在 libcstl 看来并不是同一种类型，因为在注册表中有 unsigned int 的注册信息，但是没有 uint_t 的注册信息。如果我们使用 type_register()将 uint_t 注册那么 libcstl_t 将这两个类型认为是不同的类型，即使它们操作函数都百分之百的相同。这样以来，保存 unsigned int 和保存 uint_t 的容器中的数据就不能够通用了。

类型复制函数 type_duplicate()就是用来解决这个问题的。将两个数据类型使用 type_duplicate()复制之后，libcstl 就认为这两个类型就是相同的类型但是就有不同的名字。分别使用这两个类型创建的容器中的数据也是可以通用的。

type_duplicate()接受两个数据类型作为参数，其中一个是已经注册的类型，一个未注册的类型，如果两个类型都是已经注册的或者是未注册的，type_duplicate()将失败。两个参数的顺序没有限制。对于一个类型来说，我们可以不只一次的复制这个类型，如：

type_duplicate(unsigned int, uint_t1);

type_duplicate(unsigned int, uint_t2);

type_duplicate(uint_t2, uint_t3);

type_duplicate(uint_t1, uint_t4);

这样 unsigned int，uint_t1，uint_t2，uint_t3，uint_t4 都是相同类型。

第二节 类型描述

我们看一看典型的容器创建的例子：

```
/*
 * type3.c
 * compile with : -lcstl
 */

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

int main(int argc, char* argv[])
{
    list_t* pt_list = create_list(vector_t<int>);
    vector_t* pt_vec = create_vector(int);
    iterator_t t_it_list;
    iterator_t t_it_vec;
    size_t t_i = 0;
    size_t t_j = 0;
    size_t t_count = 0;
    if(pt_list == NULL || pt_vec == NULL)
    {
        return -1;
    }

    list_init(pt_list);
    vector_init(pt_vec);

    srand((unsigned)time(NULL));
    for(t_i = 0; t_i < 10; ++t_i)
    {
        t_count = rand() % 10;
        vector_clear(pt_vec);
        for(t_j = 0; t_j < t_count; ++t_j)
        {
            vector_push_back(pt_vec, rand() - rand());
        }
        list_push_back(pt_list, pt_vec);
    }
```

```
    printf("before sorting:\n");
    for(t_it_list = list_begin(pt_list);
        !iterator_equal(t_it_list, list_end(pt_list));
        t_it_list = iterator_next(t_it_list))
    {
        for(t_it_vec = vector_begin(iterator_get_pointer(t_it_list));
            !iterator_equal(t_it_vec, vector_end(iterator_get_pointer(t_it_list)));
            t_it_vec = iterator_next(t_it_vec))
        {
            printf("%d, ", *(int*)iterator_get_pointer(t_it_vec));
        }
        printf("\n");
    }
    printf("\n");

    list_sort(pt_list);

    printf("before sorting:\n");
    for(t_it_list = list_begin(pt_list);
        !iterator_equal(t_it_list, list_end(pt_list));
        t_it_list = iterator_next(t_it_list))
    {
        for(t_it_vec = vector_begin(iterator_get_pointer(t_it_list));
            !iterator_equal(t_it_vec, vector_end(iterator_get_pointer(t_it_list)));
            t_it_vec = iterator_next(t_it_vec))
        {
            printf("%d, ", *(int*)iterator_get_pointer(t_it_vec));
        }
        printf("\n");
    }
    printf("\n");

    list_destroy(pt_list);
    vector_destroy(pt_vec);

    return 0;
}
```

这个例子的输出结果：

before sorting:

-54357362, 240555660, -525993312,

-223941381, -867172692, -468189852,

-210926579, 599657030, 414434680, -487753937, 696647328,

440735560,
-585479922, -491816986, 493460215, -658107222, 276576336, -256474479, 1500353387, 1345081139, 117588419,
717245449, 431848120, 1058917247,

11083055,
-521239713, -9901799, -247600637, -1264318026, -1696458330, -685665428,
-1332795689, 23126792, -953917801, 447767044, 556468888, -396338671,

before sorting:

-1332795689, 23126792, -953917801, 447767044, 556468888, -396338671,
-585479922, -491816986, 493460215, -658107222, 276576336, -256474479, 1500353387, 1345081139, 117588419,
-521239713, -9901799, -247600637, -1264318026, -1696458330, -685665428,
-223941381, -867172692, -468189852,
-210926579, 599657030, 414434680, -487753937, 696647328,
-54357362, 240555660, -525993312,
11083055,
440735560,
717245449, 431848120, 1058917247,

这个例子的创建函数：

```
list_t* pt_list = create_list(vector_t<int>);
vector_t* pt_vec = create_vector(int);
```

这两的函数的不同点是 pt_vec 是保存 int 类型的 vector_t 而，pt_list 则是保存 vector_t 类型，这个 vector_t 类型又是保存 int 类型的。也就是说 pt_vec 是保存 C 内建类型的数据，而 pt_list 保存的是 libcstl 容器类型。对于保存的数据类型不同，创建函数的参数就是不同的，有一整套的语法来描述如果描述容器保存的数据类型，详细的语法请参考《The libcstl Library Reference Manual》第八章。

上面的例子中有这样的操作：

```
list_push_back(pt_list, pt_vec);
```

将 pt_vec 作为 pt_list 的数据插入其中，这是因为 pt_list 中的数据类型是保存 int 类型的 vector_t，而 pt_vec 就是保存 int 类型的 vector_t 类型，所以类型是相等的。但是如果 pt_vec 是保存其他类型而非 int 类型以及非 int 类型的复制类型，那么 pt_vec 就不能作为 pt_list 的数据如：

```
vector_t* pt_vec2 = create_vector(double);
list_push_back(pt_list, pt_vec2);
```

list_push_back()行为是未定义的，断言版本将给出类型不匹配的断言。

但是下面的程序是允许的：

```
typedef int myint;
type_duplicate(int, myint);
vector_t* pt_vec3 = create_vector(myint);
list_push_back(pt_list, pt_vec3);
```

这里不会出现任何问题，因为 int 和 myint 是同一种类型，相互兼容。

附录：对于直接使用数据的函数的说明

libcstl 提供的接口函数中有一类函数很特别，它们使用数据直接作为参数。我们以 list_push_back()为例，首先看看它的原型参考《The libcstl Library Reference Manual》：
void list_push_back(list_t* pt_list, elem);
我们注意到第二个参数使用 elem 来描述，这不是合法的 C 函数声明，但是这里只是使用 elem 表示在调用这个函数的时候要直接使用数据作为参数。通过三个例子来展示在 libcstl 中如何直接使用数据作为参数。

- 使用 C 内建类型的数据

下面的例子展示了如何使用 C 内建类型数据：

```
/*
 * elem1.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 = 0; i < 10; ++i)
    {
        list_push_back(plist_coll, i);
    }
    list_push_back(plist_coll, 45);
    list_push_back(plist_coll, 0);
    list_push_back(plist_coll, -34);
    list_push_back(plist_coll, 995);
```

```
    list_push_back(plist_coll, 34);
    list_push_back(plist_coll, -2);

    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;
}
```

从例子中我们看到在使用 list_push_back()时直接将 int 类型的变量作为参数传递进去，还可以使用 int 类型的数值作为参数传递进去。

当使用浮点数的时候建议使用 double 代替 float，float 数据可能会丢失精度。

- 使用用户自定义数据

下面的例子展示如何将用户自定义的数据作为参数使用：

```
/*
 * elem2.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;
}
```

在这个例子中我们看到，将用户自定义类型数据作为参数直接使用的时候必须将数据的指针传递个函数：`deque_push_back(pdeq_coll, &t_person);`

- 使用 libcstl 内建类型数据

下面的例子展示了如何使用 libcstl 内建类型数据作为参数：

```
/*
 * elem3.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <stdlib.h>
#include <cstl/clist.h>
#include <cstl/cvector.h>
```

```
int main(int argc, char* argv[])
{
    list_t* pt_list = create_list(vector_t<int>);
    vector_t* pt_vec = create_vector(int);
    iterator_t t_it_list;
    iterator_t t_it_vec;
    size_t t_i = 0;
    size_t t_j = 0;
    size_t t_count = 0;
    if(pt_list == NULL || pt_vec == NULL)
    {
        return -1;
    }

    list_init(pt_list);
    vector_init(pt_vec);

    srand((unsigned)time(NULL));
    for(t_i = 0; t_i < 10; ++t_i)
    {
        t_count = rand() % 10;
        vector_clear(pt_vec);
        for(t_j = 0; t_j < t_count; ++t_j)
        {
            vector_push_back(pt_vec, rand() - rand());
        }
        list_push_back(pt_list, pt_vec);
    }

    printf("before sorting:\n");
    for(t_it_list = list_begin(pt_list);
        !iterator_equal(t_it_list, list_end(pt_list));
        t_it_list = iterator_next(t_it_list))
    {
        for(t_it_vec = vector_begin(iterator_get_pointer(t_it_list));
            !iterator_equal(t_it_vec, vector_end(iterator_get_pointer(t_it_list)));
            t_it_vec = iterator_next(t_it_vec))
        {
            printf("%d, ", *(int*)iterator_get_pointer(t_it_vec));
        }
        printf("\n");
    }
```

```
    printf("\n");

    list_sort(pt_list);

    printf("before sorting:\n");
    for(t_it_list = list_begin(pt_list);
        !iterator_equal(t_it_list, list_end(pt_list));
        t_it_list = iterator_next(t_it_list))
    {
        for(t_it_vec = vector_begin(iterator_get_pointer(t_it_list));
            !iterator_equal(t_it_vec, vector_end(iterator_get_pointer(t_it_list)));
            t_it_vec = iterator_next(t_it_vec))
        {
            printf("%d, ", *(int*)iterator_get_pointer(t_it_vec));
        }
        printf("\n");
    }
    printf("\n");

    list_destroy(pt_list);
    vector_destroy(pt_vec);

    return 0;
}
```

从这个例子中也可以看出，libcstl 内建数据类型作为参数直接传递给函数时也要使用指针，这与 libcstl 内建类型是用户自定义类型的特例是相符的。

综合上面的例子可以看出，直接使用数据作为参数的函数在使用是，如果容器中保存的是 C 内建类型，那么可以使用保存该类型的变量或者直接使用该类型的数据，如果是用户自定义类型和 libcstl 内建类型就必须使用数据的指针。