The libcstl Library
Reference Manual

The libcstl Library Reference Manual

for libcstl 2.0

Wangbo

2010-04-23

This file documents the libcstl library.
This is edition 1.0, last updated 2010-04-23, of *The libcstl Library Reference Manual* for libcstl 2.0.

目录

第一章 简介

第一节 关于这本手册

这本手册详细的描述了 libcstl 的全部接口和数据结构，详细的介绍了每个函数和算法的参数返回值等。这本手册并没有介绍关于函数的使用技巧方面的内容，如果想要了解关于使用技巧方面的内容请参考《The libcstl Library User Guide》。这本手册是针对 libcstl 的 2.0 版本，如果想了解其他版本请参考相应的用户指南或者参考手册。

以下是本书的结构和阅读约定：

- 第一章：简介
  简单介绍本手册的结构和内容，简单介绍 libcstl。
- 第二章：容器
  详细描述各种容器的概念，用法以及接口函数。
- 第三章：迭代器
  详细描述迭代器的概念，类型，用法。
- 第四章：算法
  详细描述算法的概念，算法的种类以及用法。
- 第五章：函数
  详细描述函数以及谓词的概念用法。
- 第六章：字符串
  详细描述了字符串类型的的概念和用法。
- 第七章：工具类型
  详细描述工具类型的概念和用法。
- 第八章：类型机制
  描述类型机制的概念和方法。
- 附录：类型描述
  描述类型时使用的方法和范式。

第二节 如何阅读这本手册

这是一本关于 libcstl 库的手册，按照库的各个部分介绍，读者可以通读，也可以按照需要来查阅相应的主题。下面是这本书的约定：

下面是本书中用到的所有主题：

- **Typedefs**
  相关的类型定义，宏定义等。
- **Operation Functions**
  与类型相关的操作函数。
- **Algorithm Functions**
  算法函数。
- **Parameters**
  函数参数的说明。
- **Remarks**
  函数相关的说明。
- **Example**
  函数的使用示例。
- **Output**
  示例的输出结果。
- **Requirements**
  要使用函数所需要的条件，如头文件等。

本书的所有范例程序都可以在 libcstl 的主页中下载到 http://code.google.com/p/libcstl/downloads/list

第三节　关于 libcstl

libcstl 为 C 语言编程提供了通用的数据结构和算法，它模仿了 SGI STL 的接口和实现。主要分为容器，迭代器，算法，函数等四个部分，此外 libcstl 2.0 提供了类型机制，为用户提供更方便的自定义类型数据管理。

所有 libcstl 容器，迭代器，函数，算法等都定义在下面列出的头文件中，要使用 libcstl 就要包含相应的头文件，下面是所有的头文件以及简要的描述：

calgorithm.h	定义了除了算术算法以为外的所有算法。
cdeque.h	定义了双端队列容器及其操作函数。
cfunctional.h	定义函数和谓词。
chash_map.h	定义了基于哈希结构的映射和多重映射容器及其操作函数。
chash_set.h	定义了基于哈希结构的集合和多重集合容器及其操作函数。
citerator.h	定义了迭代器和迭代器的辅助函数。
clist.h	定义了双向链表容器及其操作函数。
cmap.h	定义了映射和多重映射容器及其操作函数。
cnumeric.h	定义数值算法。
cqueue.h	定义了队列和优先队列容器适配器及其操作函数。
cset.h	定义了集合和多重集合容器及其操作函数。
cslist.h	定义了单向列表及其操作函数。
cstack.h	定义了堆栈容器适配器及其操作函数。
cstring.h	定义了字符串类型及其操作函数。
cutility.h	定义了工具类型及其操作函数。
cvector.h	定义了向量类型及其操作函数。

第二章 容器

为了保存数据 libcstl 库提供了多种类型的容器，这些容器都是通用的，可以用来保存任何类型的数据。这一章主要介绍各种容器以及操作函数，帮助用户选择适当的容器。

容器可以分为三种类型：序列容器，关联容器，和容器适配器。下面简要的描述了三种容器的特点，详细的信息请参考后面的章节：

- 序列容器：

序列容器按照数据的插入顺序保存数据，同时也允许用户指定在什么位置插入数据。

deque_t	双端队列允许在队列的两端快速的插入或者删除数据，同时也可以随机的访问队列内的数据。
list_t	双向链表允许在链表的任何位置快速的插入或者删除数据，但是不能够随机的访问链表内的数据。
vector_t	向量类似于数组，但是可以根据需要自动生长。
slist_t	单向链表这是一个弱化的链表，只允许在链表开头快速的插入或者删除数据，也不支持随机访问数据。

- 关联容器：

关联容器就是将插入的数据按照规则自动排序。关联容器可以分为两大类，映射和集合。映射保存的数据是键/值对，映射中的数据是按照键来排序的。集合就是保存着有序的数据，数据值本身就是键。映射和集合中的数据的键都是不能重复的，要保存重复的键就要使用多重映射和多重集合。libcstl 库还提供了基于哈希结构的映射和集合容器。

map_t	映射容器，保存有序的键/值对，键不能重复。
multimap_t	多重映射容器，保存有序的键/值对，键可以重复。
set_t	集合容器，保存有序数据，数据不能重复。
multiset_t	多重集合容器，保存有序数据，数据可以重复。
hash_map_t	基于哈希结构的映射容器，保存键/值对，键不能重复。
hash_multimap_t	基于哈希结构的多重映射容器，保存键/值对，键可以重复。
hash_set_t	基于哈希结构的集合，保存的数据不能重复。
hash_multiset_t	基于哈希结构的多重集合，保存的数据可以重复。

- 容器适配器：

容器适配器是对容器的行为进行了简单的封装，它们的底层都是容器，但是容器适配器不支持迭代器。

priority_queue_t	它是被优化的队列，优先级最高的数据总是在队列的最前面。
queue_t	它实现了一个先入先出(FIFO)的语义，第一个被插入的数据也第一个被删除。
stack_t	它实现了一个后入先出(LIFO)的语义，最后被插入的数据第一个被删除。

由于容器适配器都不支持迭代器，所以不能够在算法中使用它们。

第一节 双端队列 deque_t

双端队列使用线性的方式保存数据，像向量(vector_t)一样，它允许随机的访问数据，以及在末尾高效的插入和删除数据，与 vector_t 不同的是 deque_t 也允许在队列的开头高效的插入和删除数据。当添加或者删除实际时，deque_t 的迭代器会失效。

- **Typedefs**

deque_t	双端队列容器。
deque_iterator_t	双端队列容器的迭代器。

- **Operation Functions**

create_deque	创建一个双端队列。
deque_assign	将原始的数据删除并将新的双端队列中的数据拷贝到原来的双端队列中。
deque_assign_elem	将原始的数据删除并将指定个数的数据拷贝到原来的双端队列中。
deque_assign_range	将原始的数据删除并将指定范围内的数据拷贝到原来的双端队列中。
deque_at	访问双端队列中指定位置的数据。
deque_back	访问双端队列中最后一个数据。
deque_begin	返回指向双端队列中第一个数据的迭代器。
deque_clear	删除双端队列中的所有数据。
deque_destroy	销毁双端队列。
deque_empty	测试双端队列是否为空。
deque_end	返回指向双端队列末尾的迭代器。
deque_equal	测试两个双端队列是否相等。
deque_erase	删除双端队列中指定位置的数据。
deque_erase_range	删除双端队列中指定范围的数据。
deque_front	访问双端队列的第一个数据。
deque_greater	测试第一个双端队列是否大于第二个双端队列。
deque_greater_equal	测试第一个双端队列是否大于等于第二个双端队列。
deque_init	初始化一个空的双端队列。
deque_init_copy	使用一个双端队列初始化另一个双端队列。
deque_init_copy_range	使用指定范围内的数据初始化双端队列。
deque_init_elem	使用指定数据初始化双端队列。
deque_init_n	使用指定个数的默认数据初始化双端队列。
deque_insert	在指定位置插入数据。
deque_insert_range	在指定位置插入一个指定数据区间的数据。
deque_insert_n	在指定位置插入多个数据。
deque_less	测试第一个双端队列是否小于第二个双端队列。
deque_less_equal	测试第一个双端队列是否小于等于第二个双端队列。
deque_max_size	返回双端队列的最大可能长度。
deque_not_equal	测试两个双端队列是否不等。
deque_pop_back	删除双端队列的最后一个数据。
deque_pop_front	删除双端队列的第一个数据。
deque_push_back	在双端队列的末尾添加一个数据。
deque_push_front	在双端队列的开头添加一个数据。
deque_resize	指定双端队列的新的长度。

deque_resize_elem	指定双端队列的新的长度，并用指定数据填充。
deque_size	返回双端队列的数据个数。
deque_swap	交换两个双端队列中的数据。

1. deque_t

deque_t 是双端队列类型。

- **Requirements**
  头文件 <cstl/cdeque.h>

- **Example**
  请参考 deque_t 类型的其他操作函数。

2. deque_iterator_t

双端队列的迭代器类型。

- **Remarks**
  deque_iterator_t 是随机访问迭代器类型，可以通过迭代器来修改容器中的数据。

- **Requirements**
  头文件 <cstl/cdeque.h>

- **Example**
  请参考 deque_t 类型的其他操作函数。

3. create_deque

创建一个双端队列。

```
deque_t* create_deque(
    type
);
```

- **Parameters**
  type: 数据类型的描述。

- **Remarks**
  创建成功返回指向 deque_t 类型的指针，失败返回 NULL。

- **Requirements**
  头文件 <cstl/cdeque.h>

- **Example**
  请参考 deque_t 类型的其他操作函数。

4. deque_assign　deque_assign_elem　deque_assign_range

使用另一个 deque_t 或者多个数据或者一个数据区间为 deque_t 赋值。

```
void deque_assign(
    deque_t* pdeq_dest,
    const deque_t* cpdeq_src
);

void deque_assign_elem(
    deque_t* pdeq_dest,
    size_t t_count,
    element
);

void deque_assign_range(
    deque_t* pdeq_dest,
    deque_iterator_t it_begin,
    deque_iterator_t it_end
);
```

- **Parameters**

  pdeq_dest: 指向被赋值的 deque_t 的指针。
  cpdeq_src: 指向赋值的 deque_t 的指针。
  t_count: 赋值数据的个数。
  element: 赋值的数据。
  it_begin: 赋值的数据区间的开始位置的迭代器。
  it_end: 赋值的数据区间的末尾的迭代器。

- **Remarks**

  赋值是将原始的 deque_t 中的数据全部删除之后将新的数据复制到原始 deque_t 中。

- **Requirements**

  头文件 <cstl/cdeque.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q1 = create_deque(int);
    deque_t* pdq_q2 = create_deque(int);
    deque_iterator_t it_q;

    if(pdq_q1 == NULL || pdq_q2 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);
```

```
    deque_init(pdq_q2);

    deque_push_back(pdq_q1, 10);
    deque_push_back(pdq_q1, 20);
    deque_push_back(pdq_q1, 30);
    deque_push_back(pdq_q2, 40);
    deque_push_back(pdq_q2, 50);
    deque_push_back(pdq_q2, 60);

    printf("q1 =");
    for(it_q = deque_begin(pdq_q1);
        !iterator_equal(it_q, deque_end(pdq_q1));
        it_q = iterator_next(it_q))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_q));
    }
    printf("\n");

    deque_assign(pdq_q1, pdq_q2);
    printf("q1 =");
    for(it_q = deque_begin(pdq_q1);
        !iterator_equal(it_q, deque_end(pdq_q1));
        it_q = iterator_next(it_q))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_q));
    }
    printf("\n");

    deque_assign_range(pdq_q1, iterator_next(deque_begin(pdq_q2)),
         deque_end(pdq_q2));
    printf("q1 =");
    for(it_q = deque_begin(pdq_q1);
        !iterator_equal(it_q, deque_end(pdq_q1));
        it_q = iterator_next(it_q))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_q));
    }
    printf("\n");

    deque_assign_elem(pdq_q1, 7, 4);
    printf("q1 =");
    for(it_q = deque_begin(pdq_q1);
        !iterator_equal(it_q, deque_end(pdq_q1));
        it_q = iterator_next(it_q))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_q));
    }
    printf("\n");

    deque_destroy(pdq_q1);
    deque_destroy(pdq_q2);

    return 0;
}
```

- **Output**

```
q1 = 10 20 30
q1 = 40 50 60
q1 = 50 60
q1 = 4 4 4 4 4 4 4
```

5. deque_at

返回指向 deque_t 中指定位置的数据的指针。

```
void* deque_at(
    const deque_t* cpdeq_deque,
    size_t t_pos
);
```

- **Parameters**

  cpdeq_deque: 指向 deque_t 类型的指针。
  t_pos: 数据在 deque_t 中的位置下标。

- **Remarks**

  如果指定的位置下标有效，函数返回指向数据的指针，如果下标无效返回 NULL。

- **Requirements**

  头文件 <cstl/cdeque.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q1 = create_deque(int);
    int* pn_i = NULL;
    int n_j = 0;

    if(pdq_q1 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);

    deque_push_back(pdq_q1, 10);
    deque_push_back(pdq_q1, 20);

    pn_i = (int*)deque_at(pdq_q1, 0);
    n_j = *(int*)deque_at(pdq_q1, 1);
    printf("The first element is %d\n", *pn_i);
    printf("The second element is %d\n", n_j);

    deque_destroy(pdq_q1);

    return 0;
}
```

- **Output**

```
The first element is 10
```

```
The second element is 20
```

6. deque_back

返回指向 deque_t 中最后一个数据的指针。

```
void* deque_back(
    const deque_t* cpdeq_deque
);
```

- **Parameters**

  cpdeq_deque: 指向 deque_t 类型的指针。

- **Remarks**

  deque_t 中数据不为空则返回指向最有一个数据的指针，如果为空返回 NULL。

- **Requirements**

  头文件 <cstl/cdeque.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q1 = create_deque(int);
    int* pn_i = NULL;
    int* pn_j = NULL;

    if(pdq_q1 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);

    deque_push_back(pdq_q1, 10);
    deque_push_back(pdq_q1, 11);

    pn_i = (int*)deque_back(pdq_q1);
    pn_j = (int*)deque_back(pdq_q1);
    printf("The last integer of q1 is %d\n", *pn_i);
    (*pn_i)++;
    printf("The modified last integer of q1 is %d\n", *pn_j);

    deque_destroy(pdq_q1);

    return 0;
}
```

- **Output**

```
The last integer of q1 is 11
```

```
The modified last integer of q1 is 12
```

7. deque_begin

返回指向 deque_t 中第一个数据的迭代器。

```
deque_iterator_t deque_begin(
    const deque_t* cpdeq_deque
);
```

- **Parameters**

  cpdeq_deque: 指向 deque_t 类型的指针。

- **Remarks**

  如果 deque_t 为空，这个迭代器和指向数据末尾的迭代器相等。

- **Requirements**

  头文件 <cstl/cdeque.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q1 = create_deque(int);
    deque_iterator_t it_q;

    if(pdq_q1 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);

    deque_push_back(pdq_q1, 1);
    deque_push_back(pdq_q1, 2);

    it_q = deque_begin(pdq_q1);
    printf("The first element of q1 is %d\n", *(int*)iterator_get_pointer(it_q));

    *(int*)iterator_get_pointer(it_q) = 20;
    printf("The first element of q1 is now %d\n",
        *(int*)iterator_get_pointer(it_q));

    deque_destroy(pdq_q1);

    return 0;
}
```

- **Output**

```
The first element of q1 is 1
```

```
The first element of q1 is now 20
```

8. deque_clear

删除 deque_t 中的所有数据。

```
void deque_clear(
    deque_t* pdeq_deque
);
```

- **Parameters**

  pdeq_deque: 指向 deque_t 类型的指针。

- **Requirements**

  头文件 <cstl/cdeque.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q1 = create_deque(int);

    if(pdq_q1 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);

    deque_push_back(pdq_q1, 10);
    deque_push_back(pdq_q1, 20);
    deque_push_back(pdq_q1, 30);

    printf("The size of the deque is initially %d\n", deque_size(pdq_q1));
    deque_clear(pdq_q1);
    printf("The size of the deque after clearing is %d\n", deque_size(pdq_q1));

    deque_destroy(pdq_q1);

    return 0;
}
```

- **Output**

```
The size of the deque is initially 3
The size of the deque after clearing is 0
```

9. deque_destroy

销毁 deque_t，释放申请的资源。

```
void deque_destroy(
    deque_t* pdeq_deque
);
```

- **Parameters**
  pdeq_deque: 指向 deque_t 类型的指针。

- **Remarks**
  如果在 deque_t 类型在使用之后没有调用销毁函数，申请的资源不能够被释放。

- **Requirements**
  头文件 <cstl/cdeque.h>

- **Example**
  请参考 deque_t 类型的其他操作函数。

10. deque_empty

测试 deque_t 是否为空。

```
bool_t deque_empty(
    const deque_t* cpdeq_deque
);
```

- **Parameters**
  pdeq_deque: 指向 deque_t 类型的指针。

- **Remarks**
  deque_t 为空返回 true，否则返回 false。

- **Requirements**
  头文件 <cstl/cdeque.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q1 = create_deque(int);

    if(pdq_q1 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);

    deque_push_back(pdq_q1, 10);
    if(deque_empty(pdq_q1))
```

```
    {
        printf("The deque is emtpy.\n");
    }
    else
    {
        printf("The deque is not empty.\n");
    }

    deque_destroy(pdq_q1);

    return 0;
}
```

- **Output**

```
The deque is not empty.
```

11. deque_end

返回指向 deque_t 末尾的迭代器。

```
deque_iterator_t deque_end(
    const deque_t* cpdeq_deque
);
```

- **Parameters**

  cpdeq_deque: 指向 deque_t 类型的指针。

- **Remarks**

  当 deque_t 为空的时候返回的迭代器与指向第一个数据的迭代器相等。

- **Requirements**

  头文件 <cstl/cdeque.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q1 = create_deque(int);
    deque_iterator_t it_q;

    if(pdq_q1 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);

    deque_push_back(pdq_q1, 10);
    deque_push_back(pdq_q1, 20);
    deque_push_back(pdq_q1, 30);
```

```
    it_q = deque_end(pdq_q1);
    it_q = iterator_prev(it_q);
    printf("The last integer of q1 is %d\n", *(int*)iterator_get_pointer(it_q));

    it_q = iterator_prev(it_q);
    *(int*)iterator_get_pointer(it_q) = 400;
    printf("The new next-to-last integer of q1 is %d\n",
        *(int*)iterator_get_pointer(it_q));

    printf("The deque is now:");
    for(it_q = deque_begin(pdq_q1);
        !iterator_equal(it_q, deque_end(pdq_q1));
        it_q = iterator_next(it_q))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_q));
    }
    printf("\n");

    deque_destroy(pdq_q1);

    return 0;
}
```

- **Output**

```
The last integer of q1 is 30
The new next-to-last integer of q1 is 400
The deque is now: 10 400 30
```

12. deque_equal

测试两个 deque_t 是否相等。

```
bool_t deque_equal(
    const deque_t* cpdeq_first,
    const deque_t* cpdeq_second
);
```

- **Parameters**

cpdeq_first: 指向第一个 deque_t 类型的指针。
cpdeq_second: 指向第二个 deque_t 类型的指针。

- **Remarks**

两个 deque_t 中的每个数据都对应相等，并且数据的个数相等返回 true，否则返回 false，两个 deque_t 中保存的数据类型不同也被认为两个 deque_t 不等。

- **Requirements**

头文件 <cstl/cdeque.h>

- **Example**

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

#include <stdio.h>
```

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q1 = create_deque(int);
    deque_t* pdq_q2 = create_deque(int);

    if(pdq_q1 == NULL || pdq_q2 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);
    deque_init(pdq_q2);

    deque_push_back(pdq_q1, 1);
    deque_push_back(pdq_q2, 1);

    if(deque_equal(pdq_q1, pdq_q2))
    {
        printf("The deques are equal.\n");
    }
    else
    {
        printf("The deques are not equal.\n");
    }

    deque_push_back(pdq_q1, 1);
    if(deque_equal(pdq_q1, pdq_q2))
    {
        printf("The deques are equal.\n");
    }
    else
    {
        printf("The deques are not equal.\n");
    }

    deque_destroy(pdq_q1);
    deque_destroy(pdq_q2);

    return 0;
}
```

- **Output**

```
The deques are equal.
The deques are not equal.
```

13. deque_erase　deque_erase_range

删除指定位置的数据或者指定数据区间中的数据。

```
deque_iterator_t deque_erase(
    deque_t* pdeq_deque,
    deque_iterator_t it_pos
);

deque_iterator_t deque_erase_range(
    deque_t* pdeq_deque,
```

```
    deque_iterator_t it_begin,
    deque_iterator_t it_end
);
```

- **Parameters**

  pdeq_deque: 指向 deque_t 类型的指针。
  it_pos: 指向被删除的数据的迭代器。
  it_begin: 被删除的数据区间的开始。
  it_end: 被删除的数据区间的末尾。

- **Remarks**

  返回指向被删除的数据的下一个数据的迭代器，或者数据区间的末尾。

- **Requirements**

  头文件 <cstl/cdeque.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q1 = create_deque(int);
    deque_iterator_t it_q;

    if(pdq_q1 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);

    deque_push_back(pdq_q1, 10);
    deque_push_back(pdq_q1, 20);
    deque_push_back(pdq_q1, 30);
    deque_push_back(pdq_q1, 40);
    deque_push_back(pdq_q1, 50);

    printf("The initial deque is: ");
    for(it_q = deque_begin(pdq_q1);
        !iterator_equal(it_q, deque_end(pdq_q1));
        it_q = iterator_next(it_q))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_q));
    }
    printf("\n");

    deque_erase(pdq_q1, deque_begin(pdq_q1));
    printf("After erasing the first element, the deque becomes: ");
    for(it_q = deque_begin(pdq_q1);
        !iterator_equal(it_q, deque_end(pdq_q1));
        it_q = iterator_next(it_q))
```

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

    deque_erase_range(pdq_q1,
        iterator_next(deque_begin(pdq_q1)),
        deque_end(pdq_q1));
    printf("After erasing all elements but the first, the deque becomes: ");
    for(it_q = deque_begin(pdq_q1);
        !iterator_equal(it_q, deque_end(pdq_q1));
        it_q = iterator_next(it_q))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_q));
    }
    printf("\n");

    deque_destroy(pdq_q1);

    return 0;
}
```

- **Output**

```
The initial deque is: 10 20 30 40 50
After erasing the first element, the deque becomes: 20 30 40 50
After erasing all elements but the first, the deque becomes: 20
```

14. deque_front

返回指向第一个数据的指针。

```
void* deque_front(
    const deque_t* cpdeq_deque
);
```

- **Parameters**

  cpdeq_deque: 指向 deque_t 类型的指针。

- **Remarks**

  如果 deque_t 为空，返回 NULL。

- **Requirements**

  头文件 <cstl/cdeque.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q1 = create_deque(int);
    int* pn_i = NULL;
```

```
    int* pn_j = NULL;

    if(pdq_q1 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);

    deque_push_back(pdq_q1, 10);
    deque_push_back(pdq_q1, 11);

    pn_i = (int*)deque_front(pdq_q1);
    pn_j = (int*)deque_front(pdq_q1);
    printf("The first integer of q1 is %d\n", *pn_i);
    (*pn_i)--;
    printf("The modified first integer of q1 is %d\n", *pn_j);

    deque_destroy(pdq_q1);

    return 0;
}
```

- **Output**

```
The first integer of q1 is 10
The modified first integer of q1 is 9
```

15. deque_greater

测试第一个 deque_t 是否大于第二个 deque_t。

```
bool_t deque_greater(
    const deque_t* cpdeq_first,
    const deque_t* cpdeq_second
);
```

- **Parameters**

  cpdeq_first: 指向第一个 deque_t 类型的指针。
  cpdeq_second: 指向第二个 deque_t 类型的指针。

- **Remarks**

  要求两个 deque_t 保存的数据类型相同。

- **Requirements**

  头文件 <cstl/cdeque.h>

- **Example**

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

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

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

```
{
    deque_t* pdq_q1 = create_deque(int);
    deque_t* pdq_q2 = create_deque(int);

    if(pdq_q1 == NULL || pdq_q2 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);
    deque_init(pdq_q2);

    deque_push_back(pdq_q1, 1);
    deque_push_back(pdq_q1, 3);
    deque_push_back(pdq_q1, 1);

    deque_push_back(pdq_q2, 1);
    deque_push_back(pdq_q2, 2);
    deque_push_back(pdq_q2, 2);

    if(deque_greater(pdq_q1, pdq_q2))
    {
        printf("Deque q1 is greater than deque q2.\n");
    }
    else
    {
        printf("Deque q1 is not greater than deque q2.\n");
    }

    deque_destroy(pdq_q1);
    deque_destroy(pdq_q2);

    return 0;
}
```

- **Output**

```
Deque q1 is greater than deque q2.
```

16. deque_greater_equal

测试第一个 deque_t 是否大于等于第二个 deque_t。

```
bool_t deque_greater_equal(
    const deque_t* cpdeq_first,
    const deque_t* cpdeq_second
);
```

- **Parameters**

  cpdeq_first: 指向第一个 deque_t 类型的指针。
  cpdeq_second: 指向第二个 deque_t 类型的指针。

- **Remarks**

  要求两个 deque_t 保存的数据类型相同。

- **Requirements**

  头文件 <cstl/cdeque.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q1 = create_deque(int);
    deque_t* pdq_q2 = create_deque(int);

    if(pdq_q1 == NULL || pdq_q2 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);
    deque_init(pdq_q2);

    deque_push_back(pdq_q1, 1);
    deque_push_back(pdq_q1, 3);
    deque_push_back(pdq_q1, 1);

    deque_push_back(pdq_q2, 1);
    deque_push_back(pdq_q2, 2);
    deque_push_back(pdq_q2, 2);

    if(deque_greater_equal(pdq_q1, pdq_q2))
    {
        printf("Deque q1 is greater than or equal to deque q2.\n");
    }
    else
    {
        printf("Deque q1 is less than deque q2.\n");
    }

    deque_destroy(pdq_q1);
    deque_destroy(pdq_q2);

    return 0;
}
```

- **Output**

```
Deque q1 is greater than or equal to deque q2.
```

17. deque_init deque_init_copy deque_init_copy_range deque_init_elem deque_init_n

初始化 deque_t 容器。

```
void deque_init(
    deque_t* pdeq_deque
);

void deque_init_copy(
```

```
    deque_t* pdeq_deque,
    const deque_t* cpdeq_src
);

void deque_init_copy_range(
    deque_t* pdeq_deque,
    deque_iterator_t it_begin,
    deque_iterator_t it_end
);

void deque_init_elem(
    deque_t* pdeq_deque,
    size_t t_count,
    element
);

void deque_init_n(
    deque_t* pdeq_deque,
    size_t t_count
);
```

- **Parameters**

pdeq_deque: 指向被初始化的 deque_t 类型。
cpdeq_src: 指向用来初始化 deque_t 的 deque_t 类型。
it_begin: 用于初始化的数据区间的开始位置。
it_end: 用于初始化的数据区间的末尾。
t_count: 用于初始化的数据的个数。
element: 用于初始化的数据。

- **Remarks**

第一个函数初始化一个空 deque_t 类型。
第二个函数通过拷贝的方式初始化一个 deque_t 类型。
第三个函数使用一个数据区间初始化一个 deque_t 类型。
第四个函数使用多个指定数据初始化一个 deque_t 类型。
第五个函数使用多个默认数据初始化一个 deque_t 类型。

- **Requirements**

头文件 <cstl/cdeque.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q0 = create_deque(int);
    deque_t* pdq_q1 = create_deque(int);
    deque_t* pdq_q2 = create_deque(int);
    deque_t* pdq_q3 = create_deque(int);
    deque_t* pdq_q4 = create_deque(int);
```

```
    deque_iterator_t it_q;

    if(pdq_q0 == NULL || pdq_q1 == NULL || pdq_q2 == NULL ||
       pdq_q3 == NULL || pdq_q4 == NULL)
    {
        return -1;
    }

    /* Create an empty deque q0 */
    deque_init(pdq_q0);

    /* Create a deque q1 with 3 elements of default value 0 */
    deque_init_n(pdq_q1, 3);

    /* Create a deque q2 with 5 elements of value 2 */
    deque_init_elem(pdq_q2, 5, 2);

    /* Create a copy, deque q3, of deque q2 */
    deque_init_copy(pdq_q3, pdq_q2);

    /* Create a deque q4 by copying the range q3[first, last) */
    deque_init_copy_range(pdq_q4, deque_begin(pdq_q3),
        iterator_advance(deque_begin(pdq_q3), 2));

    printf("q1 = ");
    for(it_q = deque_begin(pdq_q1);
        !iterator_equal(it_q, deque_end(pdq_q1));
        it_q = iterator_next(it_q))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_q));
    }
    printf("\n");

    printf("q2 = ");
    for(it_q = deque_begin(pdq_q2);
        !iterator_equal(it_q, deque_end(pdq_q2));
        it_q = iterator_next(it_q))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_q));
    }
    printf("\n");

    printf("q3 = ");
    for(it_q = deque_begin(pdq_q3);
        !iterator_equal(it_q, deque_end(pdq_q3));
        it_q = iterator_next(it_q))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_q));
    }
    printf("\n");

    printf("q4 = ");
    for(it_q = deque_begin(pdq_q4);
        !iterator_equal(it_q, deque_end(pdq_q4));
        it_q = iterator_next(it_q))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_q));
    }
    printf("\n");
```

```
    deque_destroy(pdq_q0);
    deque_destroy(pdq_q1);
    deque_destroy(pdq_q2);
    deque_destroy(pdq_q3);
    deque_destroy(pdq_q4);

    return 0;
}
```

- **Output**

```
q1 = 0 0 0
q2 = 2 2 2 2 2
q3 = 2 2 2 2 2
q4 = 2 2
```

18. deque_insert deque_insert_range deque_insert_n

向 deque_t 中插入数据。

```
deque_iterator_t deque_insert(
    deque_t* pdeq_deque,
    deque_iterator_t it_pos,
    element
);

void deque_insert_range(
    deque_t* pdeq_deque,
    deque_iterator_t it_pos,
    deque_iterator_t it_begin,
    deque_iterator_t it_end
);

deque_iterator_t deque_insert_n(
    deque_t* pdeq_deque,
    deque_iterator_t it_pos,
    size_t t_count,
    element
);
```

- **Parameters**

pdeq_deque: 指向被初始化的 deque_t 类型。
it_pos: 数据插入的位置。
it_begin: 插入的数据区间的开始位置。
it_end: 插入的数据区间的末尾。
t_count: 插入的数据的个数。
element: 插入的数据。

- **Remarks**

第一个函数向指定位置插入一个数据并返回这个数据插入后的位置迭代器。
第二个函数向指定位置插入一个数据区间。
第三个函数向指定位置插入多个数据并返回被插入的第一个数据的位置迭代器。

- **Requirements**

头文件 <cstl/cdeque.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q1 = create_deque(int);
    deque_t* pdq_q2 = create_deque(int);
    deque_iterator_t it_q;

    if(pdq_q1 == NULL || pdq_q2 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);
    deque_init(pdq_q2);

    deque_push_back(pdq_q1, 10);
    deque_push_back(pdq_q1, 20);
    deque_push_back(pdq_q1, 30);
    deque_push_back(pdq_q2, 40);
    deque_push_back(pdq_q2, 50);
    deque_push_back(pdq_q2, 60);

    printf("q1 = ");
    for(it_q = deque_begin(pdq_q1);
        !iterator_equal(it_q, deque_end(pdq_q1));
        it_q = iterator_next(it_q))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_q));
    }
    printf("\n");

    deque_insert(pdq_q1, iterator_next(deque_begin(pdq_q1)), 100);
    printf("q1 = ");
    for(it_q = deque_begin(pdq_q1);
        !iterator_equal(it_q, deque_end(pdq_q1));
        it_q = iterator_next(it_q))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_q));
    }
    printf("\n");

    deque_insert_n(pdq_q1, iterator_advance(deque_begin(pdq_q1), 2), 2, 200);
    printf("q1 = ");
    for(it_q = deque_begin(pdq_q1);
        !iterator_equal(it_q, deque_end(pdq_q1));
        it_q = iterator_next(it_q))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_q));
    }
    printf("\n");
```

```
    deque_insert_range(pdq_q1, iterator_next(deque_begin(pdq_q1)),
        deque_begin(pdq_q2), iterator_prev(deque_end(pdq_q2)));
    printf("q1 = ");
    for(it_q = deque_begin(pdq_q1);
        !iterator_equal(it_q, deque_end(pdq_q1));
        it_q = iterator_next(it_q))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_q));
    }
    printf("\n");

    deque_destroy(pdq_q1);
    deque_destroy(pdq_q2);

    return 0;
}
```

- **Output**

```
q1 = 10 20 30
q1 = 10 100 20 30
q1 = 10 100 200 200 20 30
q1 = 10 40 50 100 200 200 20 30
```

19. deque_less

测试第一个 deque_t 类型是否小于第二个 deque_t 类型。

```
bool_t deque_less(
    const deque_t* cpdeq_first,
    const deque_t* cpdeq_second
);
```

- **Parameters**

cpdeq_first: 指向第一个 deque_t 类型的指针。
cpdeq_second: 指向第二个 deque_t 类型的指针。

- **Remarks**

要求两个 deque_t 保存的数据类型相同。

- **Requirements**

头文件 <cstl/cdeque.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q1 = create_deque(int);
    deque_t* pdq_q2 = create_deque(int);
```

```
    if(pdq_q1 == NULL || pdq_q2 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);
    deque_init(pdq_q2);

    deque_push_back(pdq_q1, 1);
    deque_push_back(pdq_q1, 2);
    deque_push_back(pdq_q1, 4);

    deque_push_back(pdq_q2, 1);
    deque_push_back(pdq_q2, 3);

    if(deque_less(pdq_q1, pdq_q2))
    {
        printf("Deque q1 is less than deque q2.\n");
    }
    else
    {
        printf("Deque q1 is not less than deque q2.\n");
    }

    deque_destroy(pdq_q1);
    deque_destroy(pdq_q2);

    return 0;
}
```

- **Output**

```
Deque q1 is less than deque q2.
```

20. deque_less_equal

测试第一个 deque_t 类型是否小于等于第二个 deque_t 类型。

```
bool_t deque_less_equal(
    const deque_t* cpdeq_first,
    const deque_t* cpdeq_second
);
```

- **Parameters**

  cpdeq_first: 指向第一个 deque_t 类型的指针。
  cpdeq_second: 指向第二个 deque_t 类型的指针。

- **Remarks**

  要求两个 deque_t 保存的数据类型相同。

- **Requirements**

  头文件 <cstl/cdeque.h>

- **Example**

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

```
 */

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q1 = create_deque(int);
    deque_t* pdq_q2 = create_deque(int);

    if(pdq_q1 == NULL || pdq_q2 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);
    deque_init(pdq_q2);

    deque_push_back(pdq_q1, 1);
    deque_push_back(pdq_q1, 2);
    deque_push_back(pdq_q1, 4);

    deque_push_back(pdq_q2, 1);
    deque_push_back(pdq_q2, 3);

    if(deque_less_equal(pdq_q1, pdq_q2))
    {
        printf("Deque q1 is less than or equal to deque q2.\n");
    }
    else
    {
        printf("Deque q1 is greater than deque q2.\n");
    }

    deque_destroy(pdq_q1);
    deque_destroy(pdq_q2);

    return 0;
}
```

- **Output**

```
Deque q1 is less than or equal to deque q2.
```

21. deque_max_size

返回 deque_t 类型保存数据可能的最大数量。

```
size_t deque_max_size(
    const deque_t* cpdeq_deque
);
```

- **Parameters**

cpdeq_deque: 指向 deque_t 类型的指针。

- **Remarks**

返回 deque_t 类型保存数据可能的最大数量。这是一个与系统相关的常数。

- **Requirements**

  头文件 <cstl/cdeque.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q1 = create_deque(int);

    if(pdq_q1 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);

    printf("The maxmum possible length of the deque is %d\n",
        deque_max_size(pdq_q1));

    deque_destroy(pdq_q1);

    return 0;
}
```

- **Output**

```
The maxmum possible length of the deque is 1073741823
```

22. deque_not_equal

测试两个 deque_t 类型是否不等。

```
bool_t deque_not_equal(
    const deque_t* cpdeq_first,
    const deque_t* cpdeq_second
);
```

- **Parameters**

  cpdeq_first: 指向第一个 deque_t 类型的指针。
  cpdeq_second: 指向第二个 deque_t 类型的指针。

- **Remarks**

  两个 deque_t 中保存的数据类型不同也被认为两个 deque_t 不等。

- **Requirements**

  头文件 <cstl/cdeque.h>

- **Example**

```
/*
 * deque_not_equal.c
```

```
 * compile with : -lcstl
 */

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q1 = create_deque(int);
    deque_t* pdq_q2 = create_deque(int);

    if(pdq_q1 == NULL || pdq_q2 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);
    deque_init(pdq_q2);

    deque_push_back(pdq_q1, 1);
    deque_push_back(pdq_q2, 2);

    if(deque_not_equal(pdq_q1, pdq_q2))
    {
        printf("The deques are not equal.\n");
    }
    else
    {
        printf("The deques are equal.\n");
    }

    deque_destroy(pdq_q1);
    deque_destroy(pdq_q2);

    return 0;
}
```

- **Output**

```
The deques are not equal.
```

23. deque_pop_back

删除 deque_t 最后一个数据。

```
void deque_pop_back(
    deque_t* pdeq_deque
);
```

- **Parameters**

  pdeq_deque: 指向 deque_t 类型的指针。

- **Remarks**

  deque_t 中数据为空函数的行为是未定义的。

- **Requirements**

  头文件 <cstl/cdeque.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q1 = create_deque(int);

    if(pdq_q1 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);

    deque_push_back(pdq_q1, 1);
    deque_push_back(pdq_q1, 2);

    printf("The first element is: %d\n", *(int*)deque_front(pdq_q1));
    printf("The last element is: %d\n", *(int*)deque_back(pdq_q1));

    deque_pop_back(pdq_q1);
    printf("After deleting the element at the end of the deque,"
           " the last element is %d\n",
           *(int*)deque_back(pdq_q1));

    deque_destroy(pdq_q1);

    return 0;
}
```

- **Output**

```
The first element is: 1
The last element is: 2
After deleting the element at the end of the deque, the last element is 1
```

24. deque_pop_front

删除 deque_t 中的第一个数据。

```
void deque_pop_front(
    deque_t* pdeq_deque
);
```

- **Parameters**

  pdeq_deque: 指向 deque_t 类型的指针。

- **Remarks**

  deque_t 中数据为空函数的行为是未定义的。

- **Requirements**

  头文件 <cstl/cdeque.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q1 = create_deque(int);

    if(pdq_q1 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);

    deque_push_back(pdq_q1, 1);
    deque_push_back(pdq_q1, 2);

    printf("The first element is: %d\n", *(int*)deque_front(pdq_q1));
    printf("The second element is: %d\n", *(int*)deque_back(pdq_q1));

    deque_pop_front(pdq_q1);
    printf("After deleting the element at the beginning of the deque,"
           " the first element is: %d\n", *(int*)deque_front(pdq_q1));

    deque_destroy(pdq_q1);

    return 0;
}
```

- **Output**

```
The first element is: 1
The second element is: 2
After deleting the element at the beginning of the deque, the first element is: 2
```

25. deque_push_back

向 deque_t 容器的末尾添加一个数据。

```
void deque_push_back(
    deque_t* pdeq_deque,
    element
);
```

- **Parameters**

  pdeq_deque: 指向 deque_t 类型的指针。
  element: 添加到容器末尾的数据。

- **Requirements**

  头文件 <cstl/cdeque.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q1 = create_deque(int);

    if(pdq_q1 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);

    deque_push_back(pdq_q1, 1);
    if(deque_size(pdq_q1) != 0)
    {
        printf("Last element: %d\n", *(int*)deque_back(pdq_q1));
    }

    deque_push_back(pdq_q1, 2);
    if(deque_size(pdq_q1) != 0)
    {
        printf("New last element: %d\n", *(int*)deque_back(pdq_q1));
    }

    deque_destroy(pdq_q1);

    return 0;
}
```

- **Output**

```
Last element: 1
New last element: 2
```

26. deque_push_front

向 deque_t 的开始位置添加数据。

```
void deque_push_front(
    deque_t* pdeq_deque,
    element
);
```

- **Parameters**

pdeq_deque: 指向 deque_t 类型的指针。
element: 添加到容器开始位置的数据。

- **Requirements**

头文件 <cstl/cdeque.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q1 = create_deque(int);

    if(pdq_q1 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);

    deque_push_front(pdq_q1, 1);
    if(deque_size(pdq_q1) != 0)
    {
        printf("First element: %d\n", *(int*)deque_front(pdq_q1));
    }

    deque_push_front(pdq_q1, 2);
    if(deque_size(pdq_q1) != 0)
    {
        printf("New first element: %d\n", *(int*)deque_front(pdq_q1));
    }

    deque_destroy(pdq_q1);

    return 0;
}
```

- **Output**

```
First element: 1
New first element: 2
```

27. deque_resize deque_resize_elem

重新指定 deque_t 中数据的个数，扩充的部分使用默认数据或者指定的数据填充。

```
void deque_resize(
    deque_t* pdeq_deque,
    size_t t_resize
);

void deque_resize_elem(
    deque_t* pdeq_deque,
    size_t t_resize,
    element
);
```

- **Parameters**

pdeq_deque: 指向 deque_t 类型的指针。
t_resize: deque_t 容器中数据的新的个数。
element: 填充数据。

- **Remarks**

当新的数据个数大于当前个数是使用默认数据或者指定的数据填充，当新的数据个数小于当前数据的个数时将容器后面多余的数据删除。

- **Requirements**

头文件 <cstl/cdeque.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q1 = create_deque(int);

    if(pdq_q1 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);

    deque_push_back(pdq_q1, 10);
    deque_push_back(pdq_q1, 20);
    deque_push_back(pdq_q1, 30);

    deque_resize_elem(pdq_q1, 4, 40);
    printf("The size of q1 is: %d\n", deque_size(pdq_q1));
    printf("The value of the last element is %d\n", *(int*)deque_back(pdq_q1));

    deque_resize(pdq_q1, 5);
    printf("The size of q1 is now: %d\n", deque_size(pdq_q1));
    printf("The value of the last element is now %d\n", *(int*)deque_back(pdq_q1));

    deque_resize(pdq_q1, 2);
    printf("The reduced size of q1 is: %d\n", deque_size(pdq_q1));
    printf("The value of the last element is now %d\n", *(int*)deque_back(pdq_q1));

    deque_destroy(pdq_q1);

    return 0;
}
```

- **Output**

```
The size of q1 is: 4
The value of the last element is 40
The size of q1 is now: 5
The value of the last element is now 0
The reduced size of q1 is: 2
```

```
The value of the last element is now 20
```

28. deque_size

返回容器中数据的个数。

```
size_t deque_size(
    const deque_t* cpdeq_deque
);
```

- **Parameters**

  cpdeq_deque: 指向 deque_t 类型的指针。

- **Requirements**

  头文件 <cstl/cdeque.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q1 = create_deque(int);

    if(pdq_q1 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);

    deque_push_back(pdq_q1, 1);
    printf("The deque length is %d\n", deque_size(pdq_q1));

    deque_push_back(pdq_q1, 2);
    printf("The deque length is now %d\n", deque_size(pdq_q1));

    deque_destroy(pdq_q1);

    return 0;
}
```

- **Output**

```
The deque length is 1
The deque length is now 2
```

29. deque_swap

交换两个 deque_t 的内容。

```
void deque_swap(
```

```
    deque_t* pdeq_first,
    deque_t* pdeq_second
);
```

- **Parameters**

  pdeq_first: 指向第一个 deque_t 类型的指针。
  pdeq_second: 指向第二个 deque_t 类型的指针。

- **Remarks**

  要求两个 deque_t 保存的数据类型相同。

- **Requirements**

  头文件 <cstl/cdeque.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    deque_t* pdq_q1 = create_deque(int);
    deque_t* pdq_q2 = create_deque(int);
    deque_iterator_t it_q;

    if(pdq_q1 == NULL || pdq_q2 == NULL)
    {
        return -1;
    }

    deque_init(pdq_q1);
    deque_init(pdq_q2);

    deque_push_back(pdq_q1, 1);
    deque_push_back(pdq_q1, 2);
    deque_push_back(pdq_q1, 3);
    deque_push_back(pdq_q2, 10);
    deque_push_back(pdq_q2, 20);

    printf("The original deque q1 is:");
    for(it_q = deque_begin(pdq_q1);
        !iterator_equal(it_q, deque_end(pdq_q1));
        it_q = iterator_next(it_q))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_q));
    }
    printf("\n");

    deque_swap(pdq_q1, pdq_q2);
    printf("After swapping with q2, deque q1 is:");
    for(it_q = deque_begin(pdq_q1);
        !iterator_equal(it_q, deque_end(pdq_q1));
        it_q = iterator_next(it_q))
    {
```

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

    deque_destroy(pdq_q1);
    deque_destroy(pdq_q2);

    return 0;
}
```

- **Output**

```
The original deque q1 is: 1 2 3
After swapping with q2, deque q1 is: 10 20
```

第二节　双向链表 list_t

双向链表是序列容器的一种，它以线性的方式保存数据，同时允许在任意位置高效的插入或者删除数据，但是不能够随机的访问链表中的数据。当从 list_t 中删除数据的时候，指向被删除数据的迭代器失效。

- **Typedefs**

list_t	双向链表容器类型。
list_iterator_t	双向链表迭代器类型。

- **Operation Functions**

create_list	创建双向链表容器。
list_assign	将另一个双向链表赋值给当前的双向链表。
list_assign_elem	使用指定数据为双向链表赋值。
list_assign_range	使用指定数据区间为双向链表赋值。
list_back	访问最后一个数据。
list_begin	返回指向第一个数据的迭代器。
list_clear	删除所有数据。
list_destroy	销毁双向链表容器。
list_empty	测试容器是否为空。
list_end	返回容器末尾的迭代器。
list_equal	测试两个双向链表是否相等。
list_erase	删除指定位置的数据。
list_erase_range	删除指定数据区间的数据。
list_front	访问容器中的第一个数据。
list_greater	测试第一个双向链表是否大于第二个双向链表。
list_greater_equal	测试第一个双向链表是否大于等于第二个双向链表。
list_init	初始化一个空的双向链表容器。
list_init_copy	使用另一个双向链表初始化当前的双向链表。
list_init_copy_range	使用指定的数据区间初始化双向链表。

list_init_elem	使用指定数据初始化双向链表。
list_init_n	使用指定个数的默认数据初始化双向链表。
list_insert	在指定位置插入一个数据。
list_insert_range	在指定位置插入一个数据区间。
list_insert_n	在指定位置插入多个数据。
list_less	测试第一个双向链表是否小于第二个双向链表。
list_less_equal	测试第一个双向链表是否小于等于第二个双向链表。
list_max_size	返回双向链表能够保存的最大数据个数。
list_merge	合并两个有序的双向链表。
list_merge_if	按照特定规则合并两个有序的双向链表。
list_not_equal	测试两个双向链表是否不等。
list_pop_back	删除最后一个数据。
list_pop_front	删除第一个数据。
list_push_back	在双向链表的末尾添加一个数据。
list_push_front	在双向链表的开头添加一个数据。
list_remove	删除双向链表中与指定的数据相等的数据。
list_remove_if	删除双向链表中符合特定规则的数据。
list_resize	重新设置双向链表中的数据个数，不足的部分采用默认数据填充
list_resize_elem	重新设置双向链表中的数据个数，不足的部分采用指定数据填充。
list_reverse	把双向链表中的数据逆序。
list_size	返回双向链表中数据的个数。
list_sort	排序双向链表中的数据。
list_sort_if	按照规则排序双向链表中的数据。
list_splice	将双向链表中的数据转移到另一个双向链表中。
list_splice_pos	将制定位置的数据转移到另一个双向链表中。
list_splice_range	将制定区间的数据转移到另一个双向链表中。
list_swap	交换两个双向链表的内容。
list_unique	删除相邻的重复数据。
list_unique_if	删除相邻的满足规则的数据。

1. list_t

list_t 是双向链表容器类型。

- **Requirements**

  头文件 <cstl/clist.h>

- **Example**

  请参考 list_t 类型的其他操作函数。

2. list_iterator_t

list_iterator_t 双向链表的迭代器类型。

- **Remarks**

list_iterator_t 是双向迭代器类型，不支持数据的随机访问，可以通过迭代器来修改容器中的数据。

- **Requirements**

头文件 <cstl/clist.h>

- **Example**

请参考 list_t 类型的其他操作函数。

3. create_list

创建一个双向链表容器类型。

```
list_t* create_list(
    type
);
```

- **Parameters**

type: 数据类型描述。

- **Remarks**

函数成功返回指向 list_t 类型的指针，失败返回 NULL。

- **Requirements**

头文件 <cstl/clist.h>

- **Example**

请参考 list_t 类型的其他操作函数。

4. list_assign list_assign_elem list_assign_range

使用双向链表容器，指定数据或者指定的区间为双向链表赋值。

```
void list_assign(
    list_t* plist_dest,
    const list_t* cplist_src
);

void list_assign_elem(
    list_t* plist_dest,
    size_t t_count,
    element
);

void list_assign_range(
    list_t* plist_dest,
    list_iterator_t it_begin,
    list_iterator_t it_end
);
```

- **Parameters**

  plist_dest: 指向被赋值的 list_t。
  cplist_src: 指向赋值的 list_t。
  t_count: 指定数据的个数。
  element: 指定数据。
  it_begin: 指定数据区间的开始。
  it_end: 指定数据区间的末尾。

- **Remarks**

  这三个函数都要求赋值的数据必须与 list_t 中保存的数据类型相同。

- **Requirements**

  头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    list_t* plist_l2 = create_list(int);
    list_iterator_t it_l;

    if(plist_l1 == NULL || plist_l2 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);
    list_init(plist_l2);

    list_push_back(plist_l1, 10);
    list_push_back(plist_l1, 20);
    list_push_back(plist_l1, 30);
    list_push_back(plist_l2, 40);
    list_push_back(plist_l2, 50);
    list_push_back(plist_l2, 60);

    printf("l1 =");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_assign(plist_l1, plist_l2);
    printf("l1 =");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
```

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

    list_assign_range(plist_l1, iterator_next(list_begin(plist_l2)),
        list_end(plist_l2));
    printf("l1 =");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_assign_elem(plist_l1, 7, 4);
    printf("l1 =");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_destroy(plist_l1);
    list_destroy(plist_l2);

    return 0;
}
```

- **Output**

```
l1 = 10 20 30
l1 = 40 50 60
l1 = 50 60
l1 = 4 4 4 4 4 4 4
```

5. list_back

访问双向链表容器中最后一个数据。

```
void* list_back(
    const list_t* cplist_list
);
```

- **Parameters**

  cplist_list: 指向 list_t 的指针。

- **Remarks**

  如果 list_t 不为空，则返指向 list_t 中最后一个数据的指针，如果 list_t 为空返回 NULL。

- **Requirements**

  头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    int* pn_i = NULL;
    int* pn_j = NULL;

    if(plist_l1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);

    list_push_back(plist_l1, 10);
    list_push_back(plist_l1, 20);

    pn_i = (int*)list_back(plist_l1);
    pn_j = (int*)list_back(plist_l1);

    printf("The last integer of l1 is %d\n", *pn_i);
    (*pn_i)++;
    printf("The modified last integer of l1 is %d\n", *pn_j);

    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
The last integer of l1 is 20
The modified last integer of l1 is 21
```

6. list_begin

返回指向 list_t 中第一个数据的迭代器。

```
list_iterator_t list_begin(
    const list_t* cplist_list
);
```

- **Parameters**

cplist_list: 指向 list_t 的指针。

- **Remarks**

如果 list_t 不为空，则返指向 list_t 中第一个数据的迭代器，如果 list_t 为空返回的迭代器与容器末尾的迭代器相等。

- **Requirements**

头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    list_iterator_t it_l;

    if(plist_l1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);

    list_push_back(plist_l1, 1);
    list_push_back(plist_l1, 2);

    it_l = list_begin(plist_l1);
    printf("The first element of l1 is %d\n",
        *(int*)iterator_get_pointer(it_l));

    *(int*)iterator_get_pointer(it_l) = 20;
    printf("The first element of l1 is now %d\n",
        *(int*)iterator_get_pointer(it_l));

    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
The first element of l1 is 1
The first element of l1 is now 20
```

7. list_clear

删除 list_t 中的所有数据。

```
void list_clear(
    list_t* plist_list
);
```

- **Parameters**

plist_list: 指向 list_t 的指针。

- **Requirements**

头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);

    if(plist_l1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);

    list_push_back(plist_l1, 10);
    list_push_back(plist_l1, 20);
    list_push_back(plist_l1, 30);

    printf("The size of the list is initially %d\n",
        list_size(plist_l1));
    list_clear(plist_l1);
    printf("The size of the list after clearing is %d\n",
        list_size(plist_l1));

    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
The size of the list is initially 3
The size of the list after clearing is 0
```

8. list_destroy

销毁 list_t。

```
void list_destroy(
    list_t* plist_list
);
```

- **Parameters**

  plist_list: 指向 list_t 的指针。

- **Remarks**

  当 list_t 使用之后要销毁，否则 list_t 申请的资源就不会被释放。

- **Requirements**

  头文件 <cstl/clist.h>

- **Example**

请参考 list_t 类型的其他操作函数。

9. list_empty

测试 list_t 是否为空。

```
bool_t list_empty(
    const list_t* cplist_list
);
```

- **Parameters**

cplist_list: 指向 list_t 的指针。

- **Remarks**

list_t 为空返回 true，否则返回 false。

- **Requirements**

头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);

    if(plist_l1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);

    list_push_back(plist_l1, 10);
    if(list_empty(plist_l1))
    {
        printf("The list is empty.\n");
    }
    else
    {
        printf("The list is not empty.\n");
    }

    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
The list is not empty.
```

10. list_end

返回指向 list_t 末尾的迭代器。

```
list_iterator_t list_end(
    const list_t* cplist_list
);
```

- **Parameters**

  cplist_list: 指向 list_t 的指针。

- **Remarks**

  返回指向 list_t 末尾的迭代器，如果 list_t 为空则返回的结果和 list_begin()函数的结果相等。

- **Requirements**

  头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    list_iterator_t it_l;

    if(plist_l1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);

    list_push_back(plist_l1, 10);
    list_push_back(plist_l1, 20);
    list_push_back(plist_l1, 30);

    it_l = list_end(plist_l1);
    it_l = iterator_prev(it_l);
    printf("The last integer of l1 is %d\n",
        *(int*)iterator_get_pointer(it_l));

    it_l = iterator_prev(it_l);
    *(int*)iterator_get_pointer(it_l) = 400;
    printf("The new nex-to-last integer of l1 is %d\n",
        *(int*)iterator_get_pointer(it_l));

    printf("The list is now:");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
```

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

    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
The last integer of l1 is 30
The new nex-to-last integer of l1 is 400
The list is now: 10 400 30
```

11. list_equal

测试两个 list_t 是否相等。

```
bool_t list_equal(
    const list_t* cplist_first,
    const list_t* cplist_second
);
```

- **Parameters**

cplist_first: 指向第一个 list_t 的指针。
cplist_second: 指向第二个 list_t 的指针。

- **Remarks**

list_t 中的每个数据都对应相等且个数相等返回 true，否则返回 false，如果 list_t 中保存的数据类型不同则认为两个 list_t 不等。

- **Requirements**

头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    list_t* plist_l2 = create_list(int);

    if(plist_l1 == NULL || plist_l2 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);
    list_init(plist_l2);
```

```
    list_push_back(plist_l1, 1);
    list_push_back(plist_l2, 1);

    if(list_equal(plist_l1, plist_l2))
    {
        printf("The lists are equal.\n");
    }
    else
    {
        printf("The lists are not equal.\n");
    }

    list_destroy(plist_l1);
    list_destroy(plist_l2);

    return 0;
}
```

- **Output**

```
The lists are equal.
```

12. list_erase list_erase_range

删除 list_t 中指定位置或者指定数据区间的数据。

```
list_iterator_t list_erase(
    list_t* plist_list,
    list_iterator_t it_pos
);

list_iterator_t list_erase_range(
    list_t* plist_list,
    list_iterator_t it_begin,
    list_iterator_t it_end
);
```

- **Parameters**

plist_list: 指向 list_t 的指针。
it_pos: 要删除的数据的位置。
it_begin: 要删除的数据区间的开始位置。
it_end: 要删除的数据区间的末尾。

- **Remarks**

两个函数返回的都是被删除的数据后面的位置迭代器。两个函数要求指向被删除数据的迭代器是有效的否则程序的行为是未定义的。

- **Requirements**

头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    list_iterator_t it_l;

    if(plist_l1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);

    list_push_back(plist_l1, 10);
    list_push_back(plist_l1, 20);
    list_push_back(plist_l1, 30);
    list_push_back(plist_l1, 40);
    list_push_back(plist_l1, 50);

    printf("The initial list is:");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_erase(plist_l1, list_begin(plist_l1));
    printf("After erasing the first element, the list becomes:");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_erase_range(plist_l1, iterator_next(list_begin(plist_l1)),
        list_end(plist_l1));
    printf("After erasing all elements but the first, the list becomes:");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
The initial list is: 10 20 30 40 50
After erasing the first element, the list becomes: 20 30 40 50
```

```
After erasing all elements but the first, the list becomes: 20
```

13. list_front

访问 list_t 中的第一个数据。

```
void* list_front(
    const list_t* cplist_list
);
```

- **Parameters**

  cplist_list: 指向 list_t 的指针。

- **Remarks**

  如果 list_t 不为空，则返指向 list_t 中第一个数据的指针，如果 list_t 为空返回 NULL。

- **Requirements**

  头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    int* pn_i = NULL;
    int* pn_j = NULL;

    if(plist_l1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);

    list_push_back(plist_l1, 10);

    pn_i = (int*)list_front(plist_l1);
    pn_j = (int*)list_front(plist_l1);

    printf("The first integer of l1 is %d\n", *pn_i);
    (*pn_i)++;
    printf("The modified first integer of l1 is %d\n", *pn_j);

    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
The first integer of l1 is 10
The modified first integer of l1 is 11
```

14. list_greater

测试第一个 list_t 是否大于第二个 list_t。

```
bool_t list_greater(
    const list_t* cplist_first,
    const list_t* cplist_second
);
```

- **Parameters**

  cplist_first: 指向第一个 list_t 的指针。
  cplist_second: 指向第二个 list_t 的指针。

- **Remarks**

  要求两个 list_t 保存的数据类型相同，否则程序的行为是未定义的。

- **Requirements**

  头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    list_t* plist_l2 = create_list(int);

    if(plist_l1 == NULL || plist_l2 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);
    list_init(plist_l2);

    list_push_back(plist_l1, 1);
    list_push_back(plist_l1, 3);
    list_push_back(plist_l1, 1);

    list_push_back(plist_l2, 1);
    list_push_back(plist_l2, 2);
    list_push_back(plist_l2, 2);

    if(list_greater(plist_l1, plist_l2))
    {
        printf("List l1 is greater than list l2.\n");
    }
    else
```

```
    {
        printf("The l1 is not greater than list l2.\n");
    }

    list_destroy(plist_l1);
    list_destroy(plist_l2);

    return 0;
}
```

- **Output**

```
List l1 is greater than list l2.
```

15. list_greater_equal

测试第一个 list_t 是否大于等于第二个 list_t。

```
bool_t list_greater_equal(
    const list_t* cplist_first,
    const list_t* cplist_second
);
```

- **Parameters**

cplist_first: 指向第一个 list_t 的指针。
cplist_second: 指向第二个 list_t 的指针。

- **Remarks**

要求两个 list_t 保存的数据类型相同，否则程序的行为是未定义的。

- **Requirements**

头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    list_t* plist_l2 = create_list(int);

    if(plist_l1 == NULL || plist_l2 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);
    list_init(plist_l2);

    list_push_back(plist_l1, 1);
    list_push_back(plist_l1, 3);
```

```
    list_push_back(plist_l1, 1);

    list_push_back(plist_l2, 1);
    list_push_back(plist_l2, 2);
    list_push_back(plist_l2, 2);

    if(list_greater_equal(plist_l1, plist_l2))
    {
        printf("List l1 is greater than or equal to list l2.\n");
    }
    else
    {
        printf("The l1 is less than list l2.\n");
    }

    list_destroy(plist_l1);
    list_destroy(plist_l2);

    return 0;
}
```

- **Output**

```
List l1 is greater than or equal to list l2.
```

16. list_init list_init_copy list_init_copy_range list_init_elem list_init_n

初始化 list_t。

```
void list_init(
    list_t* plist_list
);

void list_init_copy(
    list_t* plist_list,
    const list_t* cplist_src
);

void list_init_copy_range(
    list_t* plist_list,
    list_iterator_t it_begin,
    list_iterator_t it_end
);

void list_init_elem(
    list_t* plist_list,
    size_t t_count,
    element
);

void list_init_n(
    list_t* plist_list,
    size_t t_count
);
```

- **Parameters**

plist_list: 指向初始化的 list_t。

cplist_src: 指向用于初始化 list_t 类型的 list_t。
it_begin: 用于初始化 list_t 的数据区间的开始。
it_end: 用于初始化 list_t 的数据区间的末尾。
t_count: 用于初始化 list_t 的数据的个数。
element: 用于初始化 list_t 的数据。

- **Remarks**

第一个函数初始化一个空的 list_t。第二个函数使用一个现有的 list_t 类型初始化 list_t，要求两个 list_t 保存的数据类型相同，如果数据类型不同程序的行为是未定义的。第三个函数使用一个数据区间初始化 list_t，要求数据区间中的数据与 list_t 中保存的数据类型相同，如果数据类型不同那么程序的行为是未定义的。第四个函数使用指定的数据初始化 list_t。第五个数据使用默认的数据初始化 list_t。

- **Requirements**

头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l0 = create_list(int);
    list_t* plist_l1 = create_list(int);
    list_t* plist_l2 = create_list(int);
    list_t* plist_l3 = create_list(int);
    list_t* plist_l4 = create_list(int);
    list_iterator_t it_l;

    if(plist_l0 == NULL || plist_l1 == NULL || plist_l2 == NULL ||
       plist_l3 == NULL || plist_l4 == NULL)
    {
        return -1;
    }

    /* Create an empty list l0 */
    list_init(plist_l0);

    /* Create a list l1 with 3 elements of default value 0 */
    list_init_n(plist_l1, 3);

    /* Create a list l2 with 5 elements of value 2 */
    list_init_elem(plist_l2, 5, 2);

    /* Create a copy, list l3, of list l2 */
    list_init_copy(plist_l3, plist_l2);

    /* Create a list l4 by copying the range l3[first, last) */
    list_init_copy_range(plist_l4,
        iterator_advance(list_begin(plist_l3), 2),
        list_end(plist_l3));

    printf("l1 =");
    for(it_l = list_begin(plist_l1);
```

```
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    printf("l2 =");
    for(it_l = list_begin(plist_l2);
        !iterator_equal(it_l, list_end(plist_l2));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    printf("l3 =");
    for(it_l = list_begin(plist_l3);
        !iterator_equal(it_l, list_end(plist_l3));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    printf("l4 =");
    for(it_l = list_begin(plist_l4);
        !iterator_equal(it_l, list_end(plist_l4));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_destroy(plist_l0);
    list_destroy(plist_l1);
    list_destroy(plist_l2);
    list_destroy(plist_l3);
    list_destroy(plist_l4);

    return 0;
}
```

- **Output**

```
l1 = 0 0 0
l2 = 2 2 2 2 2
l3 = 2 2 2 2 2
l4 = 2 2 2
```

17. list_insert　list_insert_range　list_insert_n

向 list_t 中插入数据。

```
list_iterator_t list_insert(
    list_t* plist_list,
    list_iterator_t it_pos,
    element
);
```

```
void list_insert_range(
    list_t* plist_list,
    list_iterator_t it_pos,
    list_iterator_t it_begin,
    list_iterator_t it_end
);

list_iterator_t _list_insert_n(
    list_t* plist_list,
    list_iterator_t it_pos,
    size_t t_count,
    element
);
```

- **Parameters**

  plist_list: 指向 list_t 类型的指针。
  it_pos: 数据插入位置的迭代器。
  element: 插入 list_t 的数据。
  it_begin: 插入 list_t 的数据区间的开始。
  it_end: 插入 list_t 的数据区间的末尾。
  t_count: 插入 list_t 的数据的个数。

- **Remarks**

第一个函数返回插入后数据在 list_t 中的位置的迭代器，第三个函数返回多个数据插入 list_t 中第一个数据在 list_t 中的位置。三个函数中表示位置的迭代器必须是有效的，否则程序的行为是未定义的。第二个函数的数据区间中的数据类型必须和 list_t 中保存的数据类型相同，否则程序的行为是未定义的。

- **Requirements**

  头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    list_t* plist_l2 = create_list(int);
    list_iterator_t it_l;

    if(plist_l1 == NULL || plist_l2 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);
    list_init(plist_l2);

    list_push_back(plist_l1, 10);
```

```
    list_push_back(plist_l1, 20);
    list_push_back(plist_l1, 30);
    list_push_back(plist_l2, 40);
    list_push_back(plist_l2, 50);
    list_push_back(plist_l2, 60);

    printf("l1 =");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_insert(plist_l1, iterator_next(list_begin(plist_l1)), 100);
    printf("l1 =");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_insert_n(plist_l1, iterator_advance(list_begin(plist_l1), 2), 2, 200);
    printf("l1 =");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_insert_range(plist_l1, iterator_next(list_begin(plist_l1)),
        list_begin(plist_l2), iterator_prev(list_end(plist_l2)));
    printf("l1 =");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_destroy(plist_l1);
    list_destroy(plist_l2);

    return 0;
}
```

- **Output**

```
l1 = 10 20 30
l1 = 10 100 20 30
l1 = 10 100 200 200 20 30
l1 = 10 40 50 100 200 200 20 30
```

18. list_less

测试第一个 list_t 是否小于第二个 list_t。

```
bool_t list_less(
    const list_t* cplist_first,
    const list_t* cplist_second
);
```

- **Parameters**

  cplist_first: 指向第一个 list_t 的指针。
  cplist_second: 指向第二个 list_t 的指针。

- **Remarks**

  要求两个 list_t 保存的数据类型相同，否则程序的行为是未定义的。

- **Requirements**

  头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    list_t* plist_l2 = create_list(int);

    if(plist_l1 == NULL || plist_l2 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);
    list_init(plist_l2);

    list_push_back(plist_l1, 1);
    list_push_back(plist_l1, 2);
    list_push_back(plist_l1, 4);

    list_push_back(plist_l2, 1);
    list_push_back(plist_l2, 3);

    if(list_less(plist_l1, plist_l2))
    {
        printf("List l1 is less than list l2.\n");
    }
    else
    {
        printf("List l1 is not less than list l2.\n");
    }

    list_destroy(plist_l1);
```

```
    list_destroy(plist_l2);

    return 0;
}
```

- **Output**

```
List l1 is less than list l2.
```

19. list_less_equal

测试第一个 list_t 是否小于等于第二个 list_t。

```
bool_t list_less_equal(
    const list_t* cplist_first,
    const list_t* cplist_second
);
```

- **Parameters**

  cplist_first: 指向第一个 list_t 的指针。
  cplist_second: 指向第二个 list_t 的指针。

- **Remarks**

  要求两个 list_t 保存的数据类型相同，否则程序的行为是未定义的。

- **Requirements**

  头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    list_t* plist_l2 = create_list(int);

    if(plist_l1 == NULL || plist_l2 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);
    list_init(plist_l2);

    list_push_back(plist_l1, 1);
    list_push_back(plist_l1, 2);
    list_push_back(plist_l1, 4);

    list_push_back(plist_l2, 1);
    list_push_back(plist_l2, 3);
```

```
    if(list_less_equal(plist_l1, plist_l2))
    {
        printf("List l1 is less than or equal to list l2.\n");
    }
    else
    {
        printf("List l1 is greater than list l2.\n");
    }

    list_destroy(plist_l1);
    list_destroy(plist_l2);

    return 0;
}
```

- **Output**

```
List l1 is less than or equal to list l2.
```

20. list_max_size

返回 list_t 中保存数据的可能的最大数量。

```
size_t list_max_size(
    const list_t* cplist_list
);
```

- **Parameters**

cplist_list: 指向 list_t 的指针。

- **Remarks**

这是一个与系统相关的常量。

- **Requirements**

头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);

    if(plist_l1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);

    printf("Maximum possible length of the list is %d\n",
```

```
        list_max_size(plist_l1));

    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
Maximum possible length of the list is 1073741823
```

21. list_merge　list_merge_if

合并两个 list_t。

```
void list_merge(
    list_t* plist_dest,
    list_t* plist_src
);

void list_merge_if(
    list_t* plist_dest,
     list_t* plist_src,
     binary_function_t bfun_op
);
```

- **Parameters**

plist_dest: 指向合并的目标 list_t。
plist_src: 指向合并的源 list_t。
bfun_op: list_t 中数据的排序规则。

- **Remarks**

这两个函数都要求 list_t 是有序的，第一个函数是要求 list_t 按照默认规则有序，第二个函数要求 list_t 按照指定的规则 bfun_op 有序，如果 list_t 中的数据无效，那么函数的行为是未定义的。两个 list_t 中的数据都合并到 plist_dest 中，plist_src 中为空，并且合并后的数据也是有序的。

- **Requirements**

头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    list_t* plist_l2 = create_list(int);
    list_t* plist_l3 = create_list(int);
    list_iterator_t it_l;
```

```
    if(plist_l1 == NULL || plist_l2 == NULL || plist_l2 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);
    list_init(plist_l2);
    list_init(plist_l3);

    list_push_back(plist_l1, 3);
    list_push_back(plist_l1, 6);
    list_push_back(plist_l2, 2);
    list_push_back(plist_l2, 4);
    list_push_back(plist_l3, 5);
    list_push_back(plist_l3, 1);

    printf("l1 =");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    printf("l2 =");
    for(it_l = list_begin(plist_l2);
        !iterator_equal(it_l, list_end(plist_l2));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    /* Merge l1 into l2 in (default) ascending order */
    list_merge(plist_l2, plist_l1);
    list_sort_if(plist_l2, fun_greater_int);
    printf("After merging l1 with l2 and sorting with >: l2 =");
    for(it_l = list_begin(plist_l2);
        !iterator_equal(it_l, list_end(plist_l2));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    printf("l3 =");
    for(it_l = list_begin(plist_l3);
        !iterator_equal(it_l, list_end(plist_l3));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_merge_if(plist_l2, plist_l3, fun_greater_int);
    printf("After merging l3 with l2 according to the '>' "
           "comparison relation: l2 =");
    for(it_l = list_begin(plist_l2);
        !iterator_equal(it_l, list_end(plist_l2));
```

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

    list_destroy(plist_l1);
    list_destroy(plist_l2);
    list_destroy(plist_l3);

    return 0;
}
```

- **Output**

```
l1 = 3 6
l2 = 2 4
After merging l1 with l2 and sorting with >: l2 = 6 4 3 2
l3 = 5 1
After merging l3 with l2 according to the '>' comparison relation: l2 = 6 5 4 3 2 1
```

22. list_not_equal

测试两个 list_t 是否不等。

```
bool_t list_not_equal(
    const list_t* cplist_first,
    const list_t* cplist_second
);
```

- **Parameters**

cplist_first: 指向第一个 list_t 的指针。
cplist_second: 指向第二个 list_t 的指针。

- **Remarks**

list_t 中的每个数据都对应相等且个数相等返回 false，否则返回 true，如果 list_t 中保存的数据类型不同则认为两个 list_t 不等。

- **Requirements**

头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    list_t* plist_l2 = create_list(int);

    if(plist_l1 == NULL || plist_l2 == NULL)
    {
```

```
        return -1;
    }

    list_init(plist_l1);
    list_init(plist_l2);

    list_push_back(plist_l1, 1);
    list_push_back(plist_l2, 2);

    if(list_not_equal(plist_l1, plist_l2))
    {
        printf("Lists not equal.\n");
    }
    else
    {
        printf("Lists equal.\n");
    }

    list_destroy(plist_l1);
    list_destroy(plist_l2);

    return 0;
}
```

- **Output**

```
Lists not equal.
```

23. list_pop_back

删除 list_t 中最后一个数据。

```
void list_pop_back(
    list_t* plist_list
);
```

- **Parameters**

  plist_list: 指向 list_t 的指针。

- **Remarks**

  如果 list_t 为空，程序行为未定义。

- **Requirements**

  头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
```

```
    if(plist_l1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);

    list_push_back(plist_l1, 1);
    list_push_back(plist_l1, 2);

    printf("The first element is: %d\n",
        *(int*)list_front(plist_l1));
    printf("The last element is: %d\n",
        *(int*)list_back(plist_l1));

    list_pop_back(plist_l1);
    printf("After deleting the element at the end of the list,"
           " the last element is: %d\n",
           *(int*)list_back(plist_l1));

    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
The first element is: 1
The last element is: 2
After deleting the element at the end of the list, the last element is: 1
```

24. list_pop_front

删除 list_t 第一个数据。

```
void list_pop_front(
    list_t* plist_list
);
```

- **Parameters**

  plist_list: 指向 list_t 的指针。

- **Remarks**

  如果 list_t 为空，程序行为未定义。

- **Requirements**

  头文件 <cstl/clist.h>

- **Example**

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

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

```
int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);

    if(plist_l1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);

    list_push_back(plist_l1, 1);
    list_push_back(plist_l1, 2);

    printf("The first element is: %d\n",
        *(int*)list_front(plist_l1));
    printf("The second element is: %d\n",
        *(int*)list_back(plist_l1));

    list_pop_front(plist_l1);
    printf("After deleting the element at the beginning of the list,"
           " the first element is: %d\n",
           *(int*)list_front(plist_l1));

    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
The first element is: 1
The second element is: 2
After deleting the element at the beginning of the list, the first element is: 2
```

25. list_push_back

向 list_t 末尾添加一个数据。

```
void list_push_back(
    list_t* plist_list,
    element
);
```

- **Parameters**

plist_list: 指向 list_t 的指针。
element: 添加的数据。

- **Requirements**

头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);

    if(plist_l1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);

    list_push_back(plist_l1, 1);
    if(list_size(plist_l1) != 0)
    {
        printf("Last element: %d\n", *(int*)list_back(plist_l1));
    }

    list_push_back(plist_l1, 2);
    if(list_size(plist_l1) != 0)
    {
        printf("New last element: %d\n", *(int*)list_back(plist_l1));
    }

    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
Last element: 1
New last element: 2
```

26. list_push_front

向 list_t 开头添加一个数据。

```
void list_push_front(
    list_t* plist_list,
    element
);
```

- **Parameters**

  plist_list: 指向 list_t 的指针。
  element: 添加的数据。

- **Requirements**

  头文件 <cstl/clist.h>

- **Example**

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

```
 */

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);

    if(plist_l1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);

    list_push_front(plist_l1, 1);
    if(list_size(plist_l1) != 0)
    {
        printf("First element: %d\n", *(int*)list_front(plist_l1));
    }

    list_push_front(plist_l1, 2);
    if(list_size(plist_l1) != 0)
    {
        printf("New first element: %d\n", *(int*)list_front(plist_l1));
    }

    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
First element: 1
New first element: 2
```

27. list_remove

删除 list_t 中与指定数据相等的数据。

```
void list_remove(
    list_t* plist_list,
    element
);
```

- **Parameters**

plist_list: 指向 list_t 的指针。
element: 指定的被删除的数据。

- **Requirements**

头文件 <cstl/clist.h>

- **Example**

```
/*
 * list_remove.c
```

```
 * compile with : -lcstl
 */

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    list_iterator_t it_l;

    if(plist_l1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);

    list_push_back(plist_l1, 5);
    list_push_back(plist_l1, 100);
    list_push_back(plist_l1, 5);
    list_push_back(plist_l1, 200);
    list_push_back(plist_l1, 5);
    list_push_back(plist_l1, 300);

    printf("The initial list is l1 =");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_remove(plist_l1, 5);
    printf("After removing elements with value 5, the list becomes l1 =");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
The initial list is l1 = 5 100 5 200 5 300
After removing elements with value 5, the list becomes l1 = 100 200 300
```

28. list_remove_if

删除 list_t 中符合指定规则的数据。

```
void list_remove_if(
```

```
    list_t* plist_list,
    unary_function_t ufun_op
);
```

- **Parameters**

  plist_list: 指向 list_t 的指针。
  ufun_op: 删除数据的规则。

- **Requirements**

  头文件 <cstl/clist.h>

- **Example**

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

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    list_iterator_t it_l;

    if(plist_l1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);

    list_push_back(plist_l1, 3);
    list_push_back(plist_l1, 4);
    list_push_back(plist_l1, 5);
    list_push_back(plist_l1, 6);
    list_push_back(plist_l1, 7);
    list_push_back(plist_l1, 8);

    printf("The initial list is l1 =");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_remove_if(plist_l1, is_odd);
    printf("After removing the odd elements, the list becomes l1 =");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
```

```
    printf("\n");

    list_destroy(plist_l1);

    return 0;
}

static void is_odd(const void* cpv_input, void* pv_output)
{
    assert(cpv_input != NULL && pv_output != NULL);
    if(*(int*)cpv_input % 2 == 1)
    {
        *(bool_t*)pv_output = true;
    }
    else
    {
        *(bool_t*)pv_output = false;
    }
}
```

- **Output**

```
The initial list is l1 = 3 4 5 6 7 8
After removing the odd elements, the list becomes l1 = 4 6 8
```

29. list_resize   list_resize_elem

重设 list_t 中数据的个数，当新的数据个数比当前个数多，多处的数据使用默认数据或者指定数据填充。

```
void list_resize(
    list_t* plist_list,
    size_t t_resize
);

void list_resize_elem(
    list_t* plist_list,
    size_t t_resize,
    element
);
```

- **Parameters**

plist_list: 指向 list_t 的指针。
t_resize: list_t 中数据的新数量。
element: 填充的数据。

- **Remarks**

如果新的数据个数大于当前的数据个数，就采用默认数据或者是指定的数据来填充。如果新的数据个数小于当前数据个数，list_t 末尾的数据被删除一直到等于新数据个数。如果两个数据个数相等那么没有变化。

- **Requirements**

头文件 <cstl/clist.h>

- **Example**

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

```
 */

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);

    if(plist_l1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);

    list_push_back(plist_l1, 10);
    list_push_back(plist_l1, 20);
    list_push_back(plist_l1, 30);

    list_resize_elem(plist_l1, 4, 40);
    printf("The size of l1 is %d\n", list_size(plist_l1));
    printf("The value of the last element is %d\n",
        *(int*)list_back(plist_l1));

    list_resize(plist_l1, 5);
    printf("The size of l1 is now %d\n", list_size(plist_l1));
    printf("The value of the last element is now %d\n",
        *(int*)list_back(plist_l1));

    list_resize(plist_l1, 2);
    printf("The reduced size of l1 is %d\n", list_size(plist_l1));
    printf("The value of the last element is now %d\n",
        *(int*)list_back(plist_l1));

    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
The size of l1 is 4
The value of the last element is 40
The size of l1 is now 5
The value of the last element is now 0
The reduced size of l1 is 2
The value of the last element is now 20
```

30. list_reverse

将 list_t 中的数据逆序。

```
void list_reverse(
    list_t* plist_list
);
```

- **Parameters**

plist_list: 指向 list_t 的指针。

- **Requirements**

  头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    list_iterator_t it_l;

    if(plist_l1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);

    list_push_back(plist_l1, 10);
    list_push_back(plist_l1, 20);
    list_push_back(plist_l1, 30);

    printf("l1 =");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_reverse(plist_l1);
    printf("Reversed l1 =");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
l1 = 10 20 30
Reversed l1 = 30 20 10
```

31. list_size

返回 list_t 中数据的个数。

```
size_t list_size(
    const list_t* cplist_list
);
```

- **Parameters**

cplist_list: 指向 list_t 的指针。

- **Requirements**

头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);

    if(plist_l1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);

    list_push_back(plist_l1, 1);
    printf("List length is %d\n", list_size(plist_l1));

    list_push_back(plist_l1, 2);
    printf("List length is now %d\n", list_size(plist_l1));

    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
List length is 1
List length is now 2
```

32. list_sort list_sort_if

将 list_t 中的数据按照默认规则或者用户指定的规则排序。

```
void list_sort(
    list_t* plist_list
);
```

```
void list_sort_if(
    list_t* plist_list,
    binary_function_t bfun_op
);
```

- **Parameters**

  plist_list: 指向 list_t 的指针。
  bfun_op: 数据排序的规则。

- **Remarks**

  第一个函数使用默认的规则排序，排序后数据的顺序从小到大。
  第二个函数使用指定规则 bfun_op 排序。

- **Requirements**

  头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    list_iterator_t it_l;

    if(plist_l1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);

    list_push_back(plist_l1, 20);
    list_push_back(plist_l1, 10);
    list_push_back(plist_l1, 30);

    printf("Before sorting: l1 =");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_sort(plist_l1);
    printf("After sorting: l1 =");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
```

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

    list_sort_if(plist_l1, fun_greater_int);
    printf("After sorting with 'greater than' operation: l1 =");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
Before sorting: l1 = 20 10 30
After sorting: l1 = 10 20 30
After sorting with 'greater than' operation: l1 = 30 20 10
```

33. list_splice list_splice_pos list_splice_range

将源 list_t 中的数据转移到目的 list_t 的指定位置。

```
void list_splice(
    list_t* plist_list,
    list_iterator_t it_pos,
    list_t* plist_src
);

void list_splice_pos(
    list_t* plist_list,
    list_iterator_t it_pos,
    list_t* plist_src,
    list_iterator_t it_possrc
);

void list_splice_range(
    list_t* plist_list,
    list_iterator_t it_pos,
    list_t* plist_src,
    list_iterator_t it_begin,
    list_iterator_t it_end
);
```

- **Parameters**

plist_list: 指向目的 list_t 的指针。
it_pos: 目的 list_t 中插入数据的位置迭代器。
cplist_src: 指向源 list_t 的指针。
it_possrc: 源 list_t 中转移的数据的位置迭代器。
it_begin: 源 list_t 中转移的数据区间的开始位置迭代器。

it_end: 源 list_t 中转移的数据区间的末尾位置迭代器。

- **Remarks**

第一个函数将源 list_t 中的所有数据都转移到目的 list_t 的指定位置。
第二个函数将源 list_t 中指定位置的数据都转移到目的 list_t 的指定位置。
第三个函数将源 list_t 中指定数据区间中的数据都转移到目的 list_t 的指定位置。

- **Requirements**

头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    list_t* plist_l2 = create_list(int);
    list_t* plist_l3 = create_list(int);
    list_t* plist_l4 = create_list(int);
    list_iterator_t it_l;

    if(plist_l1 == NULL || plist_l2 == NULL ||
       plist_l3 == NULL || plist_l4 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);
    list_init(plist_l2);
    list_init(plist_l3);
    list_init(plist_l4);

    list_push_back(plist_l1, 10);
    list_push_back(plist_l1, 11);
    list_push_back(plist_l2, 12);
    list_push_back(plist_l2, 20);
    list_push_back(plist_l2, 21);
    list_push_back(plist_l3, 30);
    list_push_back(plist_l3, 31);
    list_push_back(plist_l4, 40);
    list_push_back(plist_l4, 41);
    list_push_back(plist_l4, 42);

    printf("l1 =");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    printf("l2 =");
```

```
    for(it_l = list_begin(plist_l2);
        !iterator_equal(it_l, list_end(plist_l2));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_splice(plist_l2, iterator_next(list_begin(plist_l2)), plist_l1);
    printf("After splicing l1 into l2: l2 =");
    for(it_l = list_begin(plist_l2);
        !iterator_equal(it_l, list_end(plist_l2));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_splice_pos(plist_l2, iterator_next(list_begin(plist_l2)),
        plist_l3, list_begin(plist_l3));
    printf("After splicing the first element of l3 into l2: l2 =");
    for(it_l = list_begin(plist_l2);
        !iterator_equal(it_l, list_end(plist_l2));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_splice_range(plist_l2, iterator_next(list_begin(plist_l2)),
        plist_l4, list_begin(plist_l4), iterator_prev(list_end(plist_l4)));
    printf("After splicing a range of l4 into l2: l2 =");
    for(it_l = list_begin(plist_l2);
        !iterator_equal(it_l, list_end(plist_l2));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_destroy(plist_l1);
    list_destroy(plist_l2);
    list_destroy(plist_l3);
    list_destroy(plist_l4);

    return 0;
}
```

- **Output**

```
l1 = 10 11
l2 = 12 20 21
After splicing l1 into l2: l2 = 12 10 11 20 21
After splicing the first element of l3 into l2: l2 = 12 30 10 11 20 21
After splicing a range of l4 into l2: l2 = 12 40 41 30 10 11 20 21
```

34. list_swap

交换两个 list_t 中的内容。

```
void list_swap(
    list_t* plist_first,
    list_t* plist_second
);
```

- **Parameters**
  plist_first: 指向第一个 list_t 的指针。
  plist_second: 指向第二个 list_t 的指针。

- **Remarks**
  要求两个 list_t 保存的数据类型相同，否则程序的行为是未定义的。

- **Requirements**
  头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    list_t* plist_l2 = create_list(int);
    list_iterator_t it_l;

    if(plist_l1 == NULL || plist_l2 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);
    list_init(plist_l2);

    list_push_back(plist_l1, 1);
    list_push_back(plist_l1, 2);
    list_push_back(plist_l1, 3);
    list_push_back(plist_l2, 10);
    list_push_back(plist_l2, 20);

    printf("The original list l1 is:");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_swap(plist_l1, plist_l2);
    printf("After swapping with l2, list l1 is:");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
```

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

    list_destroy(plist_l1);
    list_destroy(plist_l2);

    return 0;
}
```

- **Output**

```
The original list l1 is: 1 2 3
After swapping with l2, list l1 is: 10 20
```

35. list_unique list_unique_if

删除 list_t 中相邻的重复或者是满足指定规则的数据。

```
void list_unique(
    list_t* plist_list
);

void list_unique_if(
    list_t* plist_list,
    binary_function_t bfun_op
);
```

- **Parameters**

plist_list: 指向 list_t 的指针。
bfun_op: 数据的删除规则。

- **Remarks**

第一个函数将相邻的重复数据删除。
第二个函数将相邻的满足 bfun_op 规则的数据删除。

- **Requirements**

头文件 <cstl/clist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    list_t* plist_l2 = create_list(int);
    list_t* plist_l3 = create_list(int);
    list_iterator_t it_l;
```

```
    if(plist_l1 == NULL || plist_l2 == NULL || plist_l3 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);
    list_init(plist_l2);
    list_init(plist_l3);

    list_push_back(plist_l1, -10);
    list_push_back(plist_l1, 10);
    list_push_back(plist_l1, 10);
    list_push_back(plist_l1, 20);
    list_push_back(plist_l1, 20);
    list_push_back(plist_l1, -10);

    list_assign(plist_l2, plist_l1);
    list_assign(plist_l3, plist_l1);

    printf("The initial list is l1 =");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_unique(plist_l2);
    printf("After removing successive duplicate elements, l2 =");
    for(it_l = list_begin(plist_l2);
        !iterator_equal(it_l, list_end(plist_l2));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_unique_if(plist_l3, fun_not_equal_int);
    printf("After removing successive unequal elements, l3 =");
    for(it_l = list_begin(plist_l3);
        !iterator_equal(it_l, list_end(plist_l3));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    list_destroy(plist_l1);
    list_destroy(plist_l2);
    list_destroy(plist_l3);

    return 0;
}
```

- **Output**

```
The initial list is l1 = -10 10 10 20 20 -10
After removing successive duplicate elements, l2 = -10 10 20 -10
```

```
After removing successive unequal elements, l3 = -10 -10
```

第三节　单向链表 slist_t

slist_t 容器是一种单向链表，支持向前遍历但是不支持向后遍历。在任何位置后面插入和删除数据花费常数时间，在前面插入或删除数据花费线性时间。在 slist_t 中插入或删除数据不会使迭代器失效。slist_t 是 list_t 的一种弱化，它不支持随机访问数据，和双向迭代器。当从 slist_t 中删除数据时，指向被删除的数据的迭代器失效。

- **Typedefs**

slist_t	单向链表容器类型。
slist_iterator_t	单向链表迭代器类型。

- **Operation Functions**

create_slist	创建单向链表容器类型。
slist_assign	使用单向链表为当前的单向链表类型赋值。
slist_assign_elem	使用指定的数据为单向链表赋值。
slist_assign_range	使用指定数据区间中的数据为单向链表赋值。
slist_begin	返回指向单向链表第一个数据的迭代器。
slist_clear	删除单向链表中所有数据。
slist_destroy	销毁单向链表。
slist_empty	测试单向链表是否为空。
slist_end	返回单向链表末尾位置的迭代器。
slist_equal	测试两个单向链表是否相等。
slist_erase	删除单向链表中指定位置的数据。
slist_erase_after	删除单向链表中指定位置后面的那个数据。
slist_erase_after_range	删除单向链表中指定数据区间后面数据区间的数据。
slist_erase_range	删除单向链表中指定数据区间的数据。
slist_front	访问单向链表中第一个数据。
slist_greater	测试第一个单向链表是否大于第二个单向链表。
slist_greater_equal	测试第一个单向链表是否大于等于第二个单向链表。
slist_init	初始化一个空的单向链表。
slist_init_copy	使用一个单向链表初始化当前单向链表。
slist_init_copy_range	使用一个指定的数据区间中的数据初始化单向链表。
slist_init_elem	使用指定的数据初始化单向链表。
slist_init_n	使用多个默认数据初始化单向链表。
slist_insert	向单向链表的指定位置插入一个数据。
slist_insert_after	向单向链表的指定位置的下一个位置插入一个数据。
slist_insert_after_n	向单向链表的指定位置的下一个位置插入多个数据。
slist_insert_after_range	向单向链表的指定位置的下一个位置插入数据区间中的数据。

slist_insert_n	向单向链表的指定位置插入多个数据。
slist_insert_range	向单向链表的指定位置插入数据区间中的数据。
slist_less	测试第一个单向链表是否小于第二个单向链表。
slist_less_equal	测试第一个单向链表是否小于等于第二个单向链表。
slist_max_size	返回单向链表中能够保存数据的最大数量。
slist_merge	合并两个单向链表。
slist_merge_if	按照指定规则合并单向链表。
slist_not_equal	测试两个单向链表是否不等。
slist_pop_front	删除单向链表中的第一个数据。
slist_previous	获得指定位置的前一个位置的迭代器。
slist_push_front	在单向链表的开头添加一个数据。
slist_remove	删除单向链表中与指定数据相等的数据。
slist_remove_if	删除单向链表中与满足指定规则的数据。
slist_resize	设置新的数据个数。
slist_resize_elem	设置新的数据个数，如果新的数据个数超过当前数据个数，使用指定数据填充。
slist_reverse	将单向链表中的数据逆序。
slist_size	返回单向链表中数据的个数。
slist_sort	将单向链表中的数据排序。
slist_sort_if	将单向链表中的数据按照指定规则排序。
slist_splice	将源单向链表中的数据转移到目的单向链表中的指定位置。
slist_splice_after_pos	将源单向链表中指定位置后面的那个数据转移到目的单向链表指定位置后面。
slist_splice_after_range	将源单向链表中指定数据区间下面区间中的数据转移到目的单向链表指定位置后面。
slist_splice_pos	将源单向链表中指定位置的数据转移到目标单向链表的指定位置。
slist_splice_range	将源单向链表中指定的数据区间转移到目的单向链表的指定位置。
slist_swap	交换两个单向链表的内容。
slist_unique	删除单向链表中相邻的重复数据。
slist_unique_if	删除单向链表中相邻的满足指定规则的数据。

1. slist_t

slist_t 是单向链表容器类型。

- **Requirements**

  头文件 <cstl/cslist.h>

- **Example**

  请参考 slist_t 类型的其他操作函数。

## 2.	slist_iterator_t

slist_iterator_t 是单向链表迭代器类型。

- **Remarks**

slist_iterator_t 是前向迭代器类型，不支持数据的随机访问，不支持双向迭代器，可以通过迭代器来修改容器中的数据。

- **Requirements**

头文件 <cstl/cslist.h>

- **Example**

请参考 slist_t 类型的其他操作函数。

## 3.	create_slist

创建 slist_t 类型。

```
slist_t* create_slist(
    type
);
```

- **Parameters**

type:	数据类型描述。

- **Remarks**

函数成功返回指向 slist_t 类型的指针，失败返回 NULL。

- **Requirements**

头文件 <cstl/cslist.h>

- **Example**

请参考 slist_t 类型的其他操作函数。

## 4.	slist_assign	slist_assign_elem	slist_assign_range

使用 slist_t 或者指定的数据或者指定的数据区间为 slist_t 赋值。

```
void slist_assign(
    slist_t* pslist_slist,
    const slist_t* cpslist_src
);

void slist_assign_elem(
    slist_t* pslist_slist,
    size_t t_count,
    element
);

void slist_assign_range(
   slist_t* pslist_slist,
    slist_iterator_t it_begin,
    slist_iterator_t it_end
```

```
);
```

- **Parameters**

  pslist_slist: 指向目的 slist_t 的指针。
  cpslist_src: 指向源 slist_t 的指针。
  t_count: 赋值数据的个数。
  element: 指定的赋值数据。
  it_begin: 指定的赋值数据区间的开始位置迭代器。
  it_end: 指定的赋值数据区间的末尾位置迭代器。

- **Remarks**

第一个函数使用源 slist_t 为目的 slist_t 赋值，这两个 slist_t 保存的数据类型必须相同，否则函数的行为是未定义的。

第二个函数使用多个指定数据对 slist_t 赋值。

第三个函数使用指定的数据区间对 slist_t 赋值，区间中的数据类型必须与 slist_t 中的数据类型相同，否则函数的行为是未定义的。

- **Requirements**

  头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);
    slist_t* pslist_l2 = create_slist(int);
    slist_iterator_t it_l;

    if(pslist_l1 == NULL || pslist_l2 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);
    slist_init(pslist_l2);

    slist_push_front(pslist_l1, 10);
    slist_push_front(pslist_l1, 20);
    slist_push_front(pslist_l1, 30);
    slist_push_front(pslist_l2, 40);
    slist_push_front(pslist_l2, 50);
    slist_push_front(pslist_l2, 60);

    printf("l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
```

```
    printf("\n");

    slist_assign(pslist_l1, pslist_l2);
    printf("l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_assign_range(pslist_l1, iterator_next(slist_begin(pslist_l2)),
        slist_end(pslist_l2));
    printf("l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_assign_elem(pslist_l1, 7, 4);
    printf("l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_destroy(pslist_l1);
    slist_destroy(pslist_l2);

    return 0;
}
```

- **Output**

```
l1 = 30 20 10
l1 = 60 50 40
l1 = 50 40
l1 = 4 4 4 4 4 4 4
```

5. slist_begin

返回指向 slist_t 开始位置的迭代器。

```
slist_iterator_t slist_begin(
    const slist_t* cpslist_slist
);
```

- **Parameters**

  cpslist_slist: 指向 slist_t 的指针。

- **Remarks**

如果 slist_t 为空，返回值与指向 slist_t 末尾位置的迭代器相等。

- **Requirements**
  头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);
    slist_iterator_t it_l;

    if(pslist_l1 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);

    slist_push_front(pslist_l1, 1);
    slist_push_front(pslist_l1, 2);

    it_l = slist_begin(pslist_l1);
    printf("The first element of l1 is %d\n",
        *(int*)iterator_get_pointer(it_l));

    *(int*)iterator_get_pointer(it_l) = 20;
    printf("The first element of l1 is now %d\n",
        *(int*)iterator_get_pointer(it_l));

    slist_destroy(pslist_l1);

    return 0;
}
```

- **Output**

```
The first element of l1 is 2
The first element of l1 is now 20
```

6. slist_clear

删除 slist_t 中的所有数据。

```
void slist_clear(
    slist_t* pslist_slist
);
```

- **Parameters**
  pslist_slist: 指向 slist_t 的指针。

- **Requirements**

  头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);

    if(pslist_l1 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);

    slist_push_front(pslist_l1, 10);
    slist_push_front(pslist_l1, 20);
    slist_push_front(pslist_l1, 30);

    printf("The size of the slist is initially %d\n",
        slist_size(pslist_l1));
    slist_clear(pslist_l1);
    printf("The size of slist after clearing is %d\n",
        slist_size(pslist_l1));

    slist_destroy(pslist_l1);

    return 0;
}
```

- **Output**

```
The size of the slist is initially 3
The size of slist after clearing is 0
```

7. slist_destroy

销毁 slist_t 容器类型。

```
void slist_destroy(
    slist_t* pslist_slist
);
```

- **Parameters**

  pslist_slist: 指向 slist_t 的指针。

- **Remarks**

  使用完 slist_t 要销毁，否则 slist_t 申请的资源不会被释放。

● **Requirements**

头文件 <cstl/cslist.h>

● **Example**

请参考 slist_t 类型的其他操作函数。

8. slist_empty

测试 slist_t 是否为空。

```
bool_t slist_empty(
    const slist_t* cpslist_slist
);
```

● **Parameters**

cpslist_slist: 指向 slist_t 的指针。

● **Remarks**

slist_t 为空返回 true，否则返回 false。

● **Requirements**

头文件 <cstl/cslist.h>

● **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);

    if(pslist_l1 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);

    slist_push_front(pslist_l1, 10);
    if(slist_empty(pslist_l1))
    {
        printf("The slist is empty.\n");
    }
    else
    {
        printf("The slist is not empty.\n");
    }

    slist_destroy(pslist_l1);

    return 0;
}
```

- **Output**

```
The slist is not empty.
```

9. slist_end

返回 slist_t 末尾位置的迭代器。

```
slist_iterator_t slist_end(
    const slist_t* cpslist_slist
);
```

- **Parameters**

cpslist_slist: 指向 slist_t 的指针。

- **Remarks**

如果 slist_t 为空，它与 slist_begin()返回值相等。

- **Requirements**

头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);
    slist_iterator_t it_l;

    if(pslist_l1 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);

    slist_push_front(pslist_l1, 10);
    slist_push_front(pslist_l1, 20);
    slist_push_front(pslist_l1, 30);

    printf("The slist is:");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_destroy(pslist_l1);
```

```
    return 0;
}
```

- **Output**

```
The slist is: 30 20 10
```

10. slist_equal

测试两个 slist_t 容器是否相等。

```
bool_t slist_equal(
   const slist_t* cpslist_first,
    const slist_t* cpslist_second
);
```

- **Parameters**

cpslist_first: 指向第一个 slist_t 的指针。
cpslist_second: 指向第二个 slist_t 的指针。

- **Remarks**

两个 slist_t 中每个数据对应相等，并且数据的数量相等时返回 true，否则返回 false。两个 slist_t 保存的数据类型不同是也认为不等。

- **Requirements**

头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);
    slist_t* pslist_l2 = create_slist(int);

    if(pslist_l1 == NULL || pslist_l2 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);
    slist_init(pslist_l2);

    slist_push_front(pslist_l1, 1);
    slist_push_front(pslist_l2, 1);

    if(slist_equal(pslist_l1, pslist_l2))
    {
        printf("The slists are equal.\n");
    }
    else
```

```
    {
        printf("The slists are not equal.\n");
    }

    slist_destroy(pslist_l1);
    slist_destroy(pslist_l2);

    return 0;
}
```

- **Output**

```
The slists are equal.
```

11. slist_erase slist_erase_after slist_erase_after_range slist_erase_range

删除 slist_t 中指定位置或者指定位置后面的数据或者是区间中的数据。

```
slist_iterator_t slist_erase(
    slist_t* pslist_slist,
    slist_iterator_t it_pos
);

slist_iterator_t slist_erase_after(
   slist_t* pslist_slist,
    slist_iterator_t it_prev
);

slist_iterator_t slist_erase_after_range(
   slist_t* pslist_slist,
    slist_iterator_t it_prev,
    slist_iterator_t it_end
);

slist_iterator_t slist_erase_range(
   slist_t* pslist_slist,
    slist_iterator_t it_begin,
    slist_iterator_t it_end
);
```

- **Parameters**

pslist_slist: 指向 slist_t 的指针。
it_pos: 被删除的数据位置迭代器。
it_prev: 被删除的数据的前一个数据的位置迭代器。
it_begin: 被删除的数据区间的开始位置迭代器。
it_end: 被删除的数据区间的末尾位置迭代器。

- **Remarks**

第一个函数删除指定位置的数据并返回下一个数据的位置迭代器。
第二个函数删除指定位置后面的一个数据并返回删除位置后面的数据的位置迭代器。
第三个函数删除[it_prev+1, it_end)数据区间中的数据，并返回 it_end。
第四个函数删除[it_begin, it_end)数据区间中的数据，并返回 it_end。
上面所有的函数都要求位置迭代器和数据区间是有效的，使用无效的迭代器或者数据区间倒是函数的行为未定义。

- **Requirements**

  头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);
    slist_iterator_t it_l;

    if(pslist_l1 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);

    slist_push_front(pslist_l1, 10);
    slist_push_front(pslist_l1, 20);
    slist_push_front(pslist_l1, 30);
    slist_push_front(pslist_l1, 40);
    slist_push_front(pslist_l1, 50);

    printf("The initial slist is:");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_erase(pslist_l1, slist_begin(pslist_l1));
    printf("After erasing the first element, the slist becomes:");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_erase_range(pslist_l1, iterator_next(slist_begin(pslist_l1)),
        slist_end(pslist_l1));
    printf("After erasing all elements but the first, the slist becomes:");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");
```

```
    slist_clear(pslist_l1);
    slist_push_front(pslist_l1, 10);
    slist_push_front(pslist_l1, 20);
    slist_push_front(pslist_l1, 30);
    slist_push_front(pslist_l1, 40);
    slist_push_front(pslist_l1, 50);

    printf("After resetting, the slist becomes:");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_erase_after(pslist_l1, slist_begin(pslist_l1));
    printf("After erasing the element following the first, the slist becomes:");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_erase_after_range(pslist_l1, slist_begin(pslist_l1),
        slist_end(pslist_l1));
    printf("After erasing all elements but the first, the slist becomes:");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_destroy(pslist_l1);

    return 0;
}
```

- **Output**

```
The initial slist is: 50 40 30 20 10
After erasing the first element, the slist becomes: 40 30 20 10
After erasing all elements but the first, the slist becomes: 40
After resetting, the slist becomes: 50 40 30 20 10
After erasing the element following the first, the slist becomes: 50 30 20 10
After erasing all elements but the first, the slist becomes: 50
```

12. slist_front

访问 slist_t 的第一个数据。

```
void* slist_front(
    const slist_t* cpslist_slist
);
```

- **Parameters**

  cpslist_slist: 指向 slist_t 的指针。

- **Remarks**

  如果 slist_t 不为空，返回指向第一个数据的指针，如果 slist_t 为空返回 NULL。

- **Requirements**

  头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);
    int* pn_i = NULL;
    int* pn_j = NULL;

    if(pslist_l1 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);

    slist_push_front(pslist_l1, 10);

    pn_i = (int*)slist_front(pslist_l1);
    pn_j = (int*)slist_front(pslist_l1);

    printf("The first integer of l1 is %d\n", *pn_i);
    (*pn_i)++;
    printf("The modified first integer of l1 is %d\n", *pn_j);

    slist_destroy(pslist_l1);

    return 0;
}
```

- **Output**

```
The first integer of l1 is 10
The modified first integer of l1 is 11
```

13. slist_greater

测试第一个 slist_t 是否大于第二个 slist_t。

```
bool_t slist_greater(
   const slist_t* cpslist_first,
    const slist_t* cpslist_second
);
```

- **Parameters**

  cpslist_first: 指向第一个 slist_t 的指针。
  cpslist_second: 指向第二个 slist_t 的指针。

- **Remarks**

  要求两个 slist_t 保存的数据类型相同，如果数据类型不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);
    slist_t* pslist_l2 = create_slist(int);

    if(pslist_l1 == NULL || pslist_l2 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);
    slist_init(pslist_l2);

    slist_push_front(pslist_l1, 1);
    slist_push_front(pslist_l1, 3);
    slist_push_front(pslist_l1, 1);

    slist_push_front(pslist_l2, 2);
    slist_push_front(pslist_l2, 2);
    slist_push_front(pslist_l2, 1);

    if(slist_greater(pslist_l1, pslist_l2))
    {
        printf("Slist l1 is greater than slist l2.\n");
    }
    else
    {
        printf("The l1 is not greater than slist l2.\n");
    }

    slist_destroy(pslist_l1);
    slist_destroy(pslist_l2);

    return 0;
}
```

- **Output**

```
Slist l1 is greater than slist l2.
```

14. slist_greater_equal

测试第一个 slist_t 是否大于等于第二个 slist_t。

```
bool_t slist_greater_equal(
   const slist_t* cpslist_first,
    const slist_t* cpslist_second
);
```

- **Parameters**

  cpslist_first: 指向第一个 slist_t 的指针。
  cpslist_second: 指向第二个 slist_t 的指针。

- **Remarks**

  要求两个 slist_t 保存的数据类型相同，如果数据类型不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);
    slist_t* pslist_l2 = create_slist(int);

    if(pslist_l1 == NULL || pslist_l2 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);
    slist_init(pslist_l2);

    slist_push_front(pslist_l1, 1);
    slist_push_front(pslist_l1, 3);
    slist_push_front(pslist_l1, 1);

    slist_push_front(pslist_l2, 2);
    slist_push_front(pslist_l2, 2);
    slist_push_front(pslist_l2, 1);

    if(slist_greater_equal(pslist_l1, pslist_l2))
    {
        printf("Slist l1 is greater than or equal to slist l2.\n");
    }
    else
    {
        printf("The l1 is less than slist l2.\n");
```

```
    }

    slist_destroy(pslist_l1);
    slist_destroy(pslist_l2);

    return 0;
}
```

- **Output**

```
Slist l1 is greater than or equal to slist l2.
```

15. slist_init　slist_init_copy　slist_init_copy_range　slist_init_elem　slist_init_n

初始化 slist_t。

```
void slist_init(
    slist_t* pslist_slist
);

void slist_init_copy(
   slist_t* pslist_slist,
    const slist_t* cpslist_src
);

void slist_init_copy_range(
   slist_t* pslist_slist,
    slist_iterator_t it_begin,
    slist_iterator_t it_end
);

void slist_init_elem(
    slist_t* pslist_slist,
    size_t t_count,
    element
);

void slist_init_n(
    slist_t* pslist_slist,
    size_t t_count
);
```

- **Parameters**

pslist_slist: 指向被初始化 slist_t 的指针。
cpslist_src: 指向用来初始化 slist_t 的指针。
it_begin: 用来初始化的数据区间的开始位置的迭代器。
it_end: 用来初始化的数据区间的末尾位置的迭代器。
t_count: 用来初始化的数据个数。
element: 用来初始化的数据。

- **Remarks**

第一个函数初始化一个空的 slist_t 类型。
第二个函数使用一个 slist_t 来初始化，将源 slist_t 中的内容拷贝到目的 slist_t 中。
第三个函数使用指定的数据区间来初始化一个 slist_t。
第四个函数使用多个指定数据初始化 slist_t。

第五个函数使用多个默认数据初始化 slist_t。

- **Requirements**

  头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l0 = create_slist(int);
    slist_t* pslist_l1 = create_slist(int);
    slist_t* pslist_l2 = create_slist(int);
    slist_t* pslist_l3 = create_slist(int);
    slist_t* pslist_l4 = create_slist(int);
    slist_iterator_t it_l;

    if(pslist_l0 == NULL || pslist_l1 == NULL ||
       pslist_l2 == NULL || pslist_l3 == NULL ||
       pslist_l4 == NULL)
    {
        return -1;
    }

    /* Create an empty slist l0 */
    slist_init(pslist_l0);

    /* Create a slist l1 with 3 elements of default value 0 */
    slist_init_n(pslist_l1, 3);

    /* Create a slist l2 with 5 elements of value 2 */
    slist_init_elem(pslist_l2, 5, 2);

    /* Create a copy, slist l3, of slist l3 */
    slist_init_copy(pslist_l3, pslist_l2);

    /* Create a slist l4 by copying the range l3[first, last) */
    slist_init_copy_range(pslist_l4,
        iterator_advance(slist_begin(pslist_l3), 3),
        slist_end(pslist_l3));

    printf("l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    printf("l2 =");
    for(it_l = slist_begin(pslist_l2);
        !iterator_equal(it_l, slist_end(pslist_l2));
```

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

    printf("l3 =");
    for(it_l = slist_begin(pslist_l3);
        !iterator_equal(it_l, slist_end(pslist_l3));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    printf("l4 =");
    for(it_l = slist_begin(pslist_l4);
        !iterator_equal(it_l, slist_end(pslist_l4));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_destroy(pslist_l0);
    slist_destroy(pslist_l1);
    slist_destroy(pslist_l2);
    slist_destroy(pslist_l3);
    slist_destroy(pslist_l4);

    return 0;
}
```

- **Output**

```
l1 = 0 0 0
l2 = 2 2 2 2 2
l3 = 2 2 2 2 2
l4 = 2 2
```

16. slist_insert slist_insert_after slist_insert_after_n slist_insert_after_range slist_insert_n slist_insert_range

向 slist_t 中插入数据。

```
slist_iterator_t slist_insert(
   slist_t* pslist_slist,
    slist_iterator_t it_pos,
    element
);
slist_iterator_t slist_insert_after(
   slist_t* pslist_slist,
    slist_iterator_t it_prev,
    element
);

void slist_insert_after_n(
   slist_t* pslist_slist,
```

```
    slist_iterator_t it_prev,
    size_t t_count,
    element
);

void slist_insert_after_range(
   slist_t* pslist_slist,
    slist_iterator_t it_prev,
   slist_iterator_t it_begin,
    slist_iterator_t it_end
);

void slist_insert_range(
   slist_t* pslist_slist,
    slist_iterator_t it_pos,
   slist_iterator_t it_begin,
    slist_iterator_t it_end
);

void slist_insert_n(
   slist_t* pslist_slist,
    slist_iterator_t it_pos,
    size_t t_count,
    element
);
```

- **Parameters**

pslist_slist: 指向 slist_t 的指针。
it_pos: 被插入的数据位置迭代器。
it_prev: 被插入的数据的前一个数据的位置迭代器。
it_begin: 被插入的数据区间的开始位置迭代器。
it_end: 被插入的数据区间的末尾位置迭代器。
t_count: 插入的数据个数。
element: 插入的数据。

- **Remarks**

第一个函数在指定位置插入一个数据并返回指向插入的数据的迭代器。
第二个函数在指定位置的后面插入一个数据并返回指向插入的数据的迭代器。
第三个函数在指定位置的后面插入多个数据并返回指向被插入的第一个数据的迭代器。
第四个函数在指定的位置后面插入一个数据区间并返回指向被插入的第一个数据的迭代器。
第五个函数在指定的位置插入一个数据区间并返回指向被插入的第一个数据的迭代器。
第六个函数在指定的位置插入多个数据并返回指向被插入的第一个数据的迭代器。
上面所有的函数都要求位置迭代器和数据区间是有效的，使用无效的迭代器或者数据区间倒是函数的行为未定义。

- **Requirements**

头文件 <cstl/cslist.h>

- **Example**

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

```c
#include <stdio.h>
#include <cstl/cslist.h>

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);
    slist_t* pslist_l2 = create_slist(int);
    slist_iterator_t it_l;

    if(pslist_l1 == NULL || pslist_l2 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);
    slist_init(pslist_l2);

    slist_push_front(pslist_l1, 10);
    slist_push_front(pslist_l1, 20);
    slist_push_front(pslist_l1, 30);
    slist_push_front(pslist_l2, 40);
    slist_push_front(pslist_l2, 50);
    slist_push_front(pslist_l2, 60);

    printf("l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_insert(pslist_l1, iterator_next(slist_begin(pslist_l1)), 100);
    printf("l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_insert_n(pslist_l1, iterator_advance(slist_begin(pslist_l1), 2), 2, 200);
    printf("l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_insert_range(pslist_l1, iterator_next(slist_begin(pslist_l1)),
        slist_begin(pslist_l2), slist_end(pslist_l2));
    printf("l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
```

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

    slist_insert_after(pslist_l1, slist_begin(pslist_l1), -100);
    printf("l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_insert_after_n(pslist_l1, slist_begin(pslist_l1), 2, -200);
    printf("l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_insert_after_range(pslist_l1, slist_begin(pslist_l1),
        slist_begin(pslist_l2), slist_end(pslist_l2));
    printf("l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_destroy(pslist_l1);
    slist_destroy(pslist_l2);

    return 0;
}
```

- **Output**

```
l1 = 30 20 10
l1 = 30 100 20 10
l1 = 30 100 200 200 20 10
l1 = 30 60 50 40 100 200 200 20 10
l1 = 30 -100 60 50 40 100 200 200 20 10
l1 = 30 -200 -200 -100 60 50 40 100 200 200 20 10
l1 = 30 60 50 40 -200 -200 -100 60 50 40 100 200 200 20 10
```

17. slist_less

测试第一个 slist_t 是否小于第二个 slist_t。

```
bool_t slist_less(
   const slist_t* cpslist_first,
    const slist_t* cpslist_second
);
```

- **Parameters**

  cpslist_first: 指向第一个 slist_t 的指针。
  cpslist_second: 指向第二个 slist_t 的指针。

- **Remarks**

  要求两个 slist_t 保存的数据类型相同，如果数据类型不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);
    slist_t* pslist_l2 = create_slist(int);

    if(pslist_l1 == NULL || pslist_l2 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);
    slist_init(pslist_l2);

    slist_push_front(pslist_l1, 4);
    slist_push_front(pslist_l1, 2);
    slist_push_front(pslist_l1, 1);

    slist_push_front(pslist_l2, 3);
    slist_push_front(pslist_l2, 1);

    if(slist_less(pslist_l1, pslist_l2))
    {
        printf("Slist l1 is less than slist l2.\n");
    }
    else
    {
        printf("Slist l1 is not less than slist l2.\n");
    }

    slist_destroy(pslist_l1);
    slist_destroy(pslist_l2);

    return 0;
}
```

- **Output**

```
Slist l1 is less than slist l2.
```

18. slist_less_equal

测试第一个 slist_t 是否小于等于第二个 slist_t。

```
bool_t slist_less_equal(
   const slist_t* cpslist_first,
    const slist_t* cpslist_second
);
```

- **Parameters**

  cpslist_first: 指向第一个 slist_t 的指针。
  cpslist_second: 指向第二个 slist_t 的指针。

- **Remarks**

  要求两个 slist_t 保存的数据类型相同，如果数据类型不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);
    slist_t* pslist_l2 = create_slist(int);

    if(pslist_l1 == NULL || pslist_l2 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);
    slist_init(pslist_l2);

    slist_push_front(pslist_l1, 4);
    slist_push_front(pslist_l1, 2);
    slist_push_front(pslist_l1, 1);

    slist_push_front(pslist_l2, 3);
    slist_push_front(pslist_l2, 1);

    if(slist_less_equal(pslist_l1, pslist_l2))
    {
        printf("Slist l1 is less than or equal to slist l2.\n");
    }
    else
    {
        printf("Slist l1 is greater than slist l2.\n");
    }
```

```
    slist_destroy(pslist_l1);
    slist_destroy(pslist_l2);

    return 0;
}
```

- **Output**

```
Slist l1 is less than or equal to slist l2.
```

19. slist_max_size

返回 slist_t 中保存数据的可能最大数量。

```
size_t slist_max_size(
    const slist_t* cpslist_slist
);
```

- **Parameters**

  cpslist_slist: 指向 slist_t 的指针。

- **Remarks**

  这是一个与系统相关的常数。

- **Requirements**

  头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);

    if(pslist_l1 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);

    printf("Maximum possible length of the slist is %d\n",
        slist_max_size(pslist_l1));

    slist_destroy(pslist_l1);

    return 0;
}
```

- **Output**

```
Maximum possible length of the slist is 1073741823
```

20. slist_merge slist_merge_if

合并两个有序的 slist_t。

```
void slist_merge(
    slist_t* pslist_dest,
    slist_t* pslist_src
);

void slist_merge_if(
   slist_t* pt_dest,
    slist_t* pt_src,
    binary_function_t bfun_op
);
```

- **Parameters**

pslist_dest: 指向合并的目标 slist_t。
pslist_src: 指向合并的源 slist_t。
bfun_op: slist_t 中数据的排序规则。

- **Remarks**

这两个函数都要求 slist_t 是有序的，第一个函数是要求 slist_t 按照默认规则有序，第二个函数要求 slist_t 按照指定的规则 bfun_op 有序，如果 slist_t 中的数据无效，那么函数的行为是未定义的。两个 slist_t 中的数据都合并到 pslist_dest 中，pslist_src 中为空，并且合并后的数据也是有序的。

- **Requirements**

头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);
    slist_t* pslist_l2 = create_slist(int);
    slist_t* pslist_l3 = create_slist(int);
    slist_iterator_t it_l;

    if(pslist_l1 == NULL || pslist_l2 == NULL || pslist_l2 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);
    slist_init(pslist_l2);
    slist_init(pslist_l3);
```

```
    slist_push_front(pslist_l1, 6);
    slist_push_front(pslist_l1, 3);
    slist_push_front(pslist_l2, 4);
    slist_push_front(pslist_l2, 2);
    slist_push_front(pslist_l3, 1);
    slist_push_front(pslist_l3, 5);

    printf("l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    printf("l2 =");
    for(it_l = slist_begin(pslist_l2);
        !iterator_equal(it_l, slist_end(pslist_l2));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    /* Merge l1 into l2 in (default) ascending order */
    slist_merge(pslist_l2, pslist_l1);
    slist_sort_if(pslist_l2, fun_greater_int);
    printf("After merging l1 with l2 and sorting with >: l2 =");
    for(it_l = slist_begin(pslist_l2);
        !iterator_equal(it_l, slist_end(pslist_l2));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    printf("l3 =");
    for(it_l = slist_begin(pslist_l3);
        !iterator_equal(it_l, slist_end(pslist_l3));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_merge_if(pslist_l2, pslist_l3, fun_greater_int);
    printf("After merging l3 with l2 according to the'>'comparison relation: l2 =");
    for(it_l = slist_begin(pslist_l2);
        !iterator_equal(it_l, slist_end(pslist_l2));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_destroy(pslist_l1);
    slist_destroy(pslist_l2);
    slist_destroy(pslist_l3);
```

```
    return 0;
}
```

- **Output**

```
l1 = 3 6
l2 = 2 4
After merging l1 with l2 and sorting with >: l2 = 6 4 3 2
l3 = 5 1
After merging l3 with l2 according to the '>' comparison relation: l2 = 6 5 4 3 2 1
```

21. slist_not_equal

测试两个 slist_t 是否不等。

```
bool_t slist_not_equal(
   const slist_t* cpslist_first,
    const slist_t* cpslist_second
);
```

- **Parameters**

cpslist_first: 指向第一个 slist_t 的指针。
cpslist_second: 指向第二个 slist_t 的指针。

- **Remarks**

两个 slist_t 中每个数据对应相等，并且数据的数量相等时返回 false，否则返回 true。两个 slist_t 保存的数据类型不同是也认为不等。

- **Requirements**

头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);
    slist_t* pslist_l2 = create_slist(int);

    if(pslist_l1 == NULL || pslist_l2 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);
    slist_init(pslist_l2);

    slist_push_front(pslist_l1, 1);
    slist_push_front(pslist_l2, 2);

    if(slist_not_equal(pslist_l1, pslist_l2))
```

```
    {
        printf("Slists not equal.\n");
    }
    else
    {
        printf("Slists equal.\n");
    }

    slist_destroy(pslist_l1);
    slist_destroy(pslist_l2);

    return 0;
}
```

- **Output**

```
Slists not equal.
```

22. slist_pop_front

删除 slist_t 中的第一个数据。

```
void slist_pop_front(
    slist_t* pslist_slist
);
```

- **Parameters**

pslist_slist: 指向 slist_t 的指针。

- **Remarks**

如果 slist_t 为空则函数的行为是未定义的。

- **Requirements**

头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);

    if(pslist_l1 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);

    slist_push_front(pslist_l1, 1);
    slist_push_front(pslist_l1, 2);
```

```
    printf("The first element is: %d\n", *(int*)slist_front(pslist_l1));

    slist_pop_front(pslist_l1);
    printf("After deleting the element at the beginning of the slist,"
           " the first element is: %d\n", *(int*)slist_front(pslist_l1));

    slist_destroy(pslist_l1);

    return 0;
}
```

- **Output**

```
The first element is: 2
After deleting the element at the beginning of the slist, the first element is: 1
```

23. slist_previous

返回前一个数据的迭代器。

```
slist_iterator_t slist_previous(
   const slist_t* cpslist_slist,
    slist_iterator_t it_pos
);
```

- **Parameters**

cpslist_first: 指向 slist_t 的指针。
it_pos: 当前位置迭代器。

- **Remarks**

当前位置必须是有限迭代器，如果当前位置无效者函数行为未定义，如果当前位置为 slist_begin()这函数行为未定义。

- **Requirements**

头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);
    slist_iterator_t it_l;

    if(pslist_l1 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);
```

```
    slist_push_front(pslist_l1, 1);
    slist_push_front(pslist_l1, 2);

    it_l = slist_end(pslist_l1);
    it_l = slist_previous(pslist_l1, it_l);
    printf("The last element of list is %d\n",
        *(int*)iterator_get_pointer(it_l));

    slist_destroy(pslist_l1);

    return 0;
}
```

- **Output**

```
The last element of list is 1
```

24. slist_push_front

向 slist_t 开头添加一个数据。

```
void slist_push_front(
    slist_t* pslist_slist,
    element
);
```

- **Parameters**

pslist_first: 指向 slist_t 的指针。
element: 要添加的数据。

- **Requirements**

头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);

    if(pslist_l1 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);

    slist_push_front(pslist_l1, 1);
    if(slist_size(pslist_l1) != 0)
    {
```

```
        printf("First element: %d\n", *(int*)slist_front(pslist_l1));
    }

    slist_push_front(pslist_l1, 2);
    if(slist_size(pslist_l1) != 0)
    {
        printf("New first element: %d\n", *(int*)slist_front(pslist_l1));
    }

    slist_destroy(pslist_l1);

    return 0;
}
```

- **Output**

```
First element: 1
New first element: 2
```

25. slist_remove

删除 slist_t 中与指定数据相等的数据。

```
void slist_remove(
    slist_t* pslist_slist,
    element
);
```

- **Parameters**

pslist_slist: 指向 slist_t 的指针。
element: 要删除的数据。

- **Requirements**

头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);
    slist_iterator_t it_l;

    if(pslist_l1 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);

    slist_push_front(pslist_l1, 5);
```

```
    slist_push_front(pslist_l1, 100);
    slist_push_front(pslist_l1, 5);
    slist_push_front(pslist_l1, 200);
    slist_push_front(pslist_l1, 5);
    slist_push_front(pslist_l1, 300);

    printf("The initial slist is l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_remove(pslist_l1, 5);
    printf("After removing elements with value 5, the slist becomes l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_destroy(pslist_l1);

    return 0;
}
```

- **Output**

```
The initial slist is l1 = 300 5 200 5 100 5
After removing elements with value 5, the slist becomes l1 = 300 200 100
```

26. slist_remove_if

删除 slist_t 中满足指定规则的数据。

```
void slist_remove_if(
    slist_t* pslist_slist,
    unary_function_t ufun_op
);
```

- **Parameters**

pslist_slist: 指向 slist_t 的指针。
ufun_op: 删除数据的规则。

- **Requirements**

头文件 <cstl/cslist.h>

- **Example**

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

```c
#include <stdio.h>
#include <cstl/cslist.h>

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);
    slist_iterator_t it_l;

    if(pslist_l1 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);

    slist_push_front(pslist_l1, 3);
    slist_push_front(pslist_l1, 4);
    slist_push_front(pslist_l1, 5);
    slist_push_front(pslist_l1, 6);
    slist_push_front(pslist_l1, 7);
    slist_push_front(pslist_l1, 8);

    printf("The initial slist is l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_remove_if(pslist_l1, is_odd);
    printf("After removing the odd elements, the slist becomes l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_destroy(pslist_l1);

    return 0;
}

static void is_odd(const void* cpv_input, void* pv_output)
{
    assert(cpv_input != NULL && pv_output != NULL);
    if(*(int*)cpv_input % 2 == 1)
    {
        *(bool_t*)pv_output = true;
    }
    else
    {
        *(bool_t*)pv_output = false;
    }
}
```

- **Output**

```
The initial slist is l1 = 8 7 6 5 4 3
After removing the odd elements, the slist becomes l1 = 8 6 4
```

27. slist_resize slist_resize_elem

重新设置 slist_t 中数据的个数。

```
void slist_resize(
    slist_t* pslist_slist,
    size_t t_resize
);

void slist_resize_elem(
    slist_t* pslist_slist,
    size_t t_resize,
    element
);
```

- **Parameters**

pslist_slist: 指向 slist_t 的指针。
t_resize: slist_t 容器中数据的新个数。
element: 填充数据。

- **Remarks**

当新的数据个数大于当前数据个数的时候，第一个函数使用默认数据填充，第二个函数使用指定数据填充。当新的数据个数小于当前数据个数时，slist_t 中的靠近末尾的数据被删除一直到数据的个数缩减到新的数据个数。

- **Requirements**

头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);
    slist_iterator_t it_l;

    if(pslist_l1 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);

    slist_push_front(pslist_l1, 10);
    slist_push_front(pslist_l1, 20);
```

```
    slist_push_front(pslist_l1, 30);

    slist_resize_elem(pslist_l1, 4, 40);
    it_l = slist_previous(pslist_l1, slist_end(pslist_l1));
    printf("The size of l1 is %d\n", slist_size(pslist_l1));
    printf("The value of the last element is %d\n",
        *(int*)iterator_get_pointer(it_l));

    slist_resize(pslist_l1, 5);
    it_l = slist_previous(pslist_l1, slist_end(pslist_l1));
    printf("The size of l1 is now %d\n", slist_size(pslist_l1));
    printf("The value of the last element is now %d\n",
        *(int*)iterator_get_pointer(it_l));

    slist_resize(pslist_l1, 2);
    it_l = slist_previous(pslist_l1, slist_end(pslist_l1));
    printf("The reduced size of l1 is %d\n", slist_size(pslist_l1));
    printf("The value of the last element is now %d\n",
        *(int*)iterator_get_pointer(it_l));

    slist_destroy(pslist_l1);

    return 0;
}
```

- **Output**

```
The size of l1 is 4
The value of the last element is 40
The size of l1 is now 5
The value of the last element is now 0
The reduced size of l1 is 2
The value of the last element is now 20
```

28. slist_reverse

将 slist_t 中的数据逆序。

```
void slist_reverse(
    slist_t* pslist_slist
);
```

- **Parameters**

  pslist_slist: 指向 slist_t 的指针。

- **Requirements**

  头文件 <cstl/cslist.h>

- **Example**

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

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

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

```
{
    slist_t* pslist_l1 = create_slist(int);
    slist_iterator_t it_l;

    if(pslist_l1 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);

    slist_push_front(pslist_l1, 10);
    slist_push_front(pslist_l1, 20);
    slist_push_front(pslist_l1, 30);

    printf("l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_reverse(pslist_l1);
    printf("Reversed l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_destroy(pslist_l1);

    return 0;
}
```

- **Output**

```
l1 = 30 20 10
Reversed l1 = 10 20 30
```

29. slist_size

返回 slist_t 中数据的个数。

```
size_t slist_size(
    const slist_t* cpslist_slist
);
```

- **Parameters**

cpslist_slist: 指向 slist_t 的指针。

- **Requirements**

头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);

    if(pslist_l1 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);

    slist_push_front(pslist_l1, 1);
    printf("List length is %d\n", slist_size(pslist_l1));

    slist_push_front(pslist_l1, 2);
    printf("List length is now %d\n", slist_size(pslist_l1));

    slist_destroy(pslist_l1);

    return 0;
}
```

- **Output**

```
List length is 1
List length is now 2
```

30. slist_sort slist_sort_if

将 slist_t 中的数据排序。

```
void slist_sort(
    slist_t* pslist_slist
);

void slist_sort_if(
    slist_t* pslist_slist,
    binary_function_t bfun_op
);
```

- **Parameters**

pslist_slist: 指向 slist_t 的指针。
bfun_op: 数据的排序规则。

- **Remarks**

第一个函数使用默认规则(数据的小于操作函数)来排序 slist_t 中的数据，第二个函数使用指定的规则 bfun_op 来排序 slist_t 中的数据。

- **Requirements**

  头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);
    slist_iterator_t it_l;

    if(pslist_l1 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);

    slist_push_front(pslist_l1, 20);
    slist_push_front(pslist_l1, 10);
    slist_push_front(pslist_l1, 30);

    printf("Before sorting: l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_sort(pslist_l1);
    printf("After sorting: l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_sort_if(pslist_l1, fun_greater_int);
    printf("After sorting with 'greater than' operation: l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_destroy(pslist_l1);
```

```
    return 0;
}
```

- **Output**

```
Before sorting: l1 = 30 10 20
After sorting: l1 = 10 20 30
After sorting with 'greater than' operation: l1 = 30 20 10
```

31. slist_splice slist_splice_after_pos slist_splice_after_range slist_splice_pos slist_splice_range

将数据转移到 slist_t 的指定位置。

```
void slist_splice(
    slist_t* pslist_slist,
    slist_iterator_t it_pos,
    slist_t* pslist_src
);

void slist_splice_after_pos(
    slist_t* pslist_slist,
    slist_iterator_t it_prev,
    slist_t* pslist_src,
    slist_iterator_t it_prevsrc
);

void slist_splice_after_range(
    slist_t* pslist_slist,
    slist_iterator_t it_prev,
    slist_t* pslist_src,
    slist_iterator_t it_beforefirst,
    slist_iterator_t it_beforelast
);

void slist_splice_pos(
    slist_t* pslist_slist,
    slist_iterator_t it_pos,
    slist_t* pslist_src,
    slist_iterator_t it_possrc
);

void slist_splice_range(
    slist_t* pslist_slist,
    slist_iterator_t it_pos,
    slist_t* pslist_src,
    slist_iterator_t it_begin,
    slist_iterator_t it_end
);
```

- **Parameters**

pslist_slist: 指向目的 slist_t 的指针。
it_pos: 转移的数据插入的位置迭代器。
pslist_src: 指向源 slist_t 的指针。

it_prev: 转移的数据插入的位置的前一个位置迭代器。
it_prevsrc: 源 slist_t 中被转移的数据位置的前一个位置迭代器。
it_beforefirst: 源 slist_t 中被转移的数据区间的开始位置的前一个位置迭代器。
it_beforelast: 源 slist_t 中被转移的数据区间的末尾位置的前一个位置迭代器。
it_pos: 源 slist_t 中被转移的数据的位置迭代器。
it_begin: 源 slist_t 中被转移的数据区间的开始位置迭代器。
it_end: 源 slist_t 中被转移的数据区间的末尾位置迭代器。

- **Remarks**

第一个函数将源 slist_t 中的所有数据转移到目的 slist_t 的指定位置。
第二个函数将源 slist_t 中 it_prevsrc+1 数据转移到目的 slist_t 的 it_prev+1。
第三个函数将源 slist_t 中[it_beforefirst+1, it_beforelast+1)数据转移到目的 slist_t 的 it_prev+1。
第四个函数将源 slist_t 中 it_possrc 数据转移到目的 slist_t 的 it_pos。
第五个函数将源 slist_t 中[it_begin, it_end)数据转移到目的 slist_t 的 it_pos。
上面所有的函数都要求位置迭代器和数据区间是有效的，使用无效的迭代器或者数据区间倒是函数的行为未定义。

- **Requirements**

头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);
    slist_t* pslist_l2 = create_slist(int);
    slist_t* pslist_l3 = create_slist(int);
    slist_t* pslist_l4 = create_slist(int);
    slist_t* pslist_l5 = create_slist(int);
    slist_t* pslist_l6 = create_slist(int);
    slist_iterator_t it_l;

    if(pslist_l1 == NULL || pslist_l2 == NULL || pslist_l3 == NULL ||
       pslist_l4 == NULL || pslist_l5 == NULL || pslist_l6 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);
    slist_init(pslist_l2);
    slist_init(pslist_l3);
    slist_init(pslist_l4);
    slist_init(pslist_l5);
    slist_init(pslist_l6);

    slist_push_front(pslist_l1, 10);
    slist_push_front(pslist_l1, 11);
    slist_push_front(pslist_l2, 12);
    slist_push_front(pslist_l2, 20);
    slist_push_front(pslist_l2, 21);
```

```
    slist_push_front(pslist_l3, 30);
    slist_push_front(pslist_l3, 31);
    slist_push_front(pslist_l4, 40);
    slist_push_front(pslist_l4, 41);
    slist_push_front(pslist_l4, 42);
    slist_push_front(pslist_l5, 55);
    slist_push_front(pslist_l5, 56);
    slist_push_front(pslist_l5, 57);
    slist_push_front(pslist_l6, 62);
    slist_push_front(pslist_l6, 65);
    slist_push_front(pslist_l6, 66);
    slist_push_front(pslist_l6, 67);

    printf("l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    printf("l2 =");
    for(it_l = slist_begin(pslist_l2);
        !iterator_equal(it_l, slist_end(pslist_l2));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_splice(pslist_l2, iterator_next(slist_begin(pslist_l2)), pslist_l1);
    printf("After splicing l1 into l2: l2 =");
    for(it_l = slist_begin(pslist_l2);
        !iterator_equal(it_l, slist_end(pslist_l2));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_splice_pos(pslist_l2, iterator_next(slist_begin(pslist_l2)),
        pslist_l3, slist_begin(pslist_l3));
    printf("After splicing the first element of l3 into l2: l2 =");
    for(it_l = slist_begin(pslist_l2);
        !iterator_equal(it_l, slist_end(pslist_l2));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_splice_range(pslist_l2, iterator_next(slist_begin(pslist_l2)),
        pslist_l4, slist_begin(pslist_l4), slist_end(pslist_l4));
    printf("After splicing a range of l4 into l2: l2 =");
    for(it_l = slist_begin(pslist_l2);
        !iterator_equal(it_l, slist_end(pslist_l2));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
```

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

    slist_splice_after_pos(pslist_l2, slist_begin(pslist_l2),
        pslist_l5, slist_begin(pslist_l5));
    printf("After splicing the element following the first of l5 into l2: l2 =");
    for(it_l = slist_begin(pslist_l2);
        !iterator_equal(it_l, slist_end(pslist_l2));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_splice_after_range(pslist_l2, slist_begin(pslist_l2),
        pslist_l6, slist_begin(pslist_l6),
        iterator_advance(slist_begin(pslist_l6), 2));
    printf("After splicing a range of l6 into l2: l2 =");
    for(it_l = slist_begin(pslist_l2);
        !iterator_equal(it_l, slist_end(pslist_l2));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_destroy(pslist_l1);
    slist_destroy(pslist_l2);
    slist_destroy(pslist_l3);
    slist_destroy(pslist_l4);
    slist_destroy(pslist_l5);
    slist_destroy(pslist_l6);

    return 0;
}
```

- **Output**

```
l1 = 11 10
l2 = 21 20 12
After splicing l1 into l2:
l2 = 21 11 10 20 12
After splicing the first element of l3 into l2:
l2 = 21 31 11 10 20 12
After splicing a range of l4 into l2:
l2 = 21 42 41 40 31 11 10 20 12
After splicing the element following the first of l5 into l2:
l2 = 21 56 42 41 40 31 11 10 20 12
After splicing a range of l6 into l2:
l2 = 21 66 65 56 42 41 40 31 11 10 20 12
```

32. slist_swap

交换两个 slist_t 的内容。

```
void slist_swap(
    slist_t* pslist_first,
    slist_t* pslist_second
);
```

- **Parameters**
  pslist_first: 指向第一个 slist_t 的指针。
  pslist_second: 指向第二个 slist_t 的指针。

- **Remarks**
  要求两个 slist_t 保存的数据类型相同，如果数据类型不同导致函数的行为未定义。

- **Requirements**
  头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);
    slist_t* pslist_l2 = create_slist(int);
    slist_iterator_t it_l;

    if(pslist_l1 == NULL || pslist_l2 == NULL)
    {
        return -1;
    }

    slist_init(pslist_l1);
    slist_init(pslist_l2);

    slist_push_front(pslist_l1, 1);
    slist_push_front(pslist_l1, 2);
    slist_push_front(pslist_l1, 3);
    slist_push_front(pslist_l2, 10);
    slist_push_front(pslist_l2, 20);

    printf("The original slist l1 is:");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_swap(pslist_l1, pslist_l2);
    printf("After swapping with l2, slist l1 is:");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");
```

```
    slist_destroy(pslist_l1);
    slist_destroy(pslist_l2);

    return 0;
}
```

- **Output**

```
The original slist l1 is: 3 2 1
After swapping with l2, slist l1 is: 20 10
```

33. slist_unique　slist_unique_if

删除 slist_t 中相邻的重复数据或者符合规则的数据。

```
void slist_unique(
    slist_t* pslist_slist
);

void slist_unique_if(
    slist_t* pslist_slist,
    binary_function_t bfun_op
);
```

- **Parameters**

pslist_slist: 指向 slist_t 的指针。
bfun_op: 删除数据的规则。

- **Remarks**

第一个函数删除 slist_t 中相邻的重复数据，第二个函数删除 slist_t 中相邻的满足 bfun_op 的数据。

- **Requirements**

头文件 <cstl/cslist.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l1 = create_slist(int);
    slist_t* pslist_l2 = create_slist(int);
    slist_t* pslist_l3 = create_slist(int);
    slist_iterator_t it_l;

    if(pslist_l1 == NULL || pslist_l2 == NULL || pslist_l3 == NULL)
    {
        return -1;
    }
```

```
    slist_init(pslist_l1);
    slist_init(pslist_l2);
    slist_init(pslist_l3);

    slist_push_front(pslist_l1, -10);
    slist_push_front(pslist_l1, 10);
    slist_push_front(pslist_l1, 10);
    slist_push_front(pslist_l1, 20);
    slist_push_front(pslist_l1, 20);
    slist_push_front(pslist_l1, -10);

    slist_assign(pslist_l2, pslist_l1);
    slist_assign(pslist_l3, pslist_l1);

    printf("The initial slist is l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_unique(pslist_l2);
    printf("After removing successive duplicate elements, l2 =");
    for(it_l = slist_begin(pslist_l2);
        !iterator_equal(it_l, slist_end(pslist_l2));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_unique_if(pslist_l3, fun_not_equal_int);
    printf("After removing successive unequal elements, l3 =");
    for(it_l = slist_begin(pslist_l3);
        !iterator_equal(it_l, slist_end(pslist_l3));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_destroy(pslist_l1);
    slist_destroy(pslist_l2);
    slist_destroy(pslist_l3);

    return 0;
}
```

- **Output**

```
The initial slist is l1 = -10 20 20 10 10 -10
After removing successive duplicate elements, l2 = -10 20 10 -10
After removing successive unequal elements, l3 = -10 -10
```

第四节　向量 vector_t

vector_t 与数组类似，以线性方式保存并管理数据，但是它可以自动生长。vector_t 快速的随机访问任何数据，在 vector_t 末尾插入或删除数据花费常数时间，在开头或者中间插入或者删除花费线性时间。vector_t 的迭代器是随机访问迭代器，可以通过迭代器随机访问数据，获得并修改数据。当插入或者删除数据是，在插入或删除数据位置之后的迭代器失效。

- **Typedefs**

vector_t	向量容器类型。
vector_iterator_t	向量容器迭代器类型。

- **Operation Functions**

create_vector	创建向量容器类型。
vector_assign	使用向量容器类型为当前向量容器赋值。
vector_assign_elem	使用指定数据为向量容器赋值。
vector_assign_range	使用指定的数据区间为向量赋值。
vector_at	使用下标随机访问向量中的数据。
vector_back	访问向量容器的最后一个数据。
vector_begin	返回指向向量容器的开始的迭代器。
vector_capacity	返回向量容器在不重新分配内存的情况下能够保存数据的个数。
vector_clear	删除向量容器中的所有数据。
vector_destroy	销毁向量容器类型。
vector_empty	测试向量容器是否为空。
vector_end	返回指向向量容器末尾位置的迭代器。
vector_equal	测试两个向量容器是否相等。
vector_erase	删除向量容器中指定位置的数据。
vector_erase_range	删除向量容器中指定数据区间中的数据。
vector_front	访问向量容器中第一个数据。
vector_greater	测试第一个向量容器是否大于第二个向量容器。
vector_greater_equal	测试第一个向量容器是否大于等于第二个向量容器。
vector_init	初始化一个空的向量容器。
vector_init_copy	使用一个向量容器类型初始化当前向量容器。
vector_init_copy_range	使用指定数据区间中的数据初始化向量容器。
vector_init_elem	使用指定数据初始化向量容器。
vector_init_n	使用多个默认数据初始化向量容器。
vector_insert	在向量容器的指定位置插入一个数据。
vector_insert_n	在向量容器的指定位置插入多个数据。
vector_insert_range	在向量容器的指定位置插入数据区间中的数据。
vector_less	测试第一个向量容器是否小于第二个向量容器。
vector_less_equal	测试第一个向量容器是否小于等于第二个向量容器。

vector_max_size	向量容器能够保存的数据的可能最大数量。
vector_not_equal	测试两个向量容器是否不等。
vector_pop_back	删除向量容器中的最后一个数据。
vector_push_back	在向量容器的末尾添加一个数据。
vector_reserve	设置向量容器在不分配内存的情况下能够保存数据的个数。
vector_resize	重新设置向量容器中数据的个数。
vector_resize_elem	重新设置向量容器中数据的个数，不足的部分使用指定数据填充。
vector_size	获得向量容器中的数据的个数。
vector_swap	交换两个向量容器中的内容。

1. vector_t

vector_t 向量容器类型。

- **Requirements**

头文件 <cstl/cvector.h>

- **Example**

请参考 vector_t 类型的其他操作函数。

2. vector_iterator_t

vector_iterator_t 是向量容器迭代器类型。

- **Remarks**

vector_iterator_t 是随机访问迭代器类型，支持数据的随机访问，可以通过迭代器来修改容器中的数据。

- **Requirements**

头文件 <cstl/cvector.h>

- **Example**

请参考 vector_t 类型的其他操作函数。

3. create_vector

创建 vector_t 容器类型。

```
vector_t* create_vector(
    type
);
```

- **Parameters**

type: 数据类型描述。

- **Remarks**

函数成功返回指向 vector_t 类型的指针，失败返回 NULL。

- **Requirements**
  头文件 <cstl/cvector.h>

- **Example**
  请参考 vector_t 类型的其他操作函数。

4. vector_assign vector_assign_elem vector_assign_range

使用 vector_t 或者指定的数据或者数据区间为 vector_t 赋值。

```
void vector_assign(
    vector_t* pvec_vector,
    const vector_t* cpvec_src
);

void vector_assign_elem(
    vector_t* pvec_vector,
    size_t t_count,
    element
);

void vector_assign_range(
    vector_t* pvec_vector,
    vector_iterator_t it_begin,
    vector_iterator_t t_end
);
```

- **Parameters**
  pvec_vector: 指向被赋值的 vector_t。
  cpvec_src: 指向赋值的 vector_t。
  t_count: 指定数据的个数。
  element: 指定数据。
  it_begin: 指定数据区间的开始。
  it_end: 指定数据区间的末尾。

- **Remarks**
  这三个函数都要求赋值的数据必须与 vector_t 中保存的数据类型相同。

- **Requirements**
  头文件 <cstl/cvector.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
```

```
    vector_t* pvec_v3 = create_vector(int);
    vector_t* pvec_v4 = create_vector(int);
    vector_iterator_t it_v;

    if(pvec_v1 == NULL || pvec_v2 == NULL ||
       pvec_v3 == NULL || pvec_v4 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);
    vector_init(pvec_v3);
    vector_init(pvec_v4);

    vector_push_back(pvec_v1, 10);
    vector_push_back(pvec_v1, 20);
    vector_push_back(pvec_v1, 30);
    vector_push_back(pvec_v1, 40);
    vector_push_back(pvec_v1, 50);

    printf("v1 =");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_v));
    }
    printf("\n");

    vector_assign(pvec_v2, pvec_v1);
    printf("v2 =");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_v));
    }
    printf("\n");

    vector_assign_range(pvec_v3, vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("v3 =");
    for(it_v = vector_begin(pvec_v3);
        !iterator_equal(it_v, vector_end(pvec_v3));
        it_v = iterator_next(it_v))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_v));
    }
    printf("\n");

    vector_assign_elem(pvec_v4, 7, 4);
    printf("v4 =");
    for(it_v = vector_begin(pvec_v4);
        !iterator_equal(it_v, vector_end(pvec_v4));
        it_v = iterator_next(it_v))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_v));
    }
    printf("\n");
```

```
    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);
    vector_destroy(pvec_v3);
    vector_destroy(pvec_v4);

    return 0;
}
```

● **Output**

```
v1 = 10 20 30 40 50
v2 = 10 20 30 40 50
v3 = 10 20 30 40 50
v4 = 4 4 4 4 4 4 4
```

5. vector_at

使用下标对 vector_t 中的数据进行随机访问。

```
void* vector_at(
    const vector_t* cpvec_vector,
    size_t t_pos
);
```

● **Parameters**

cpvec_vector: 指向 vector_t 类型的指针。
t_pos: 要访问的数据的下标。

● **Remarks**

要访问的数据的小标必须是有效的下标，无效下标导致函数行为未定义。

● **Requirements**

头文件 <cstl/cvector.h>

● **Example**

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

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

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    int* pn_i = NULL;
    int n_j = 0;

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

    vector_init(pvec_v1);

    vector_push_back(pvec_v1, 10);
```

```
    vector_push_back(pvec_v1, 20);

    pn_i = (int*)vector_at(pvec_v1, 0);
    n_j = *(int*)vector_at(pvec_v1, 1);
    printf("The first element is %d\n", *pn_i);
    printf("The second element is %d\n", n_j);

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The first element is 10
The second element is 20
```

6. vector_back

访问 vector_t 中的最后一个数据。

```
void* vector_back(
    const vector_t* cpvec_vector
);
```

- **Parameters**

  cpvec_vector: 指向 vector_t 类型的指针。

- **Remarks**

  vector_t 容器为空时返回 NULL。

- **Requirements**

  头文件 <cstl/cvector.h>

- **Example**

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

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

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

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

    vector_init(pvec_v1);

    vector_push_back(pvec_v1, 10);
```

```
    vector_push_back(pvec_v1, 11);

    pn_i = (int*)vector_back(pvec_v1);
    pn_j = (int*)vector_back(pvec_v1);

    printf("The last integer of v1 is %d\n", *pn_i);
    (*pn_i)++;
    printf("The modified last integer of v1 is %d\n", *pn_j);

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The last integer of v1 is 11
The modified last integer of v1 is 12
```

7. vector_begin

返回指向 vector_t 第一个数据的迭代器。

```
vector_iterator_t vector_begin(
    const vector_t* cpvec_vector
);
```

- **Parameters**

  cpvec_vector: 指向 vector_t 类型的指针。

- **Remarks**

  vector_t 容器时，函数的返回值与 vector_end()相等。

- **Requirements**

  头文件 <cstl/cvector.h>

- **Example**

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

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

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

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

    vector_init(pvec_v1);
```

```
    vector_push_back(pvec_v1, 1);
    vector_push_back(pvec_v1, 2);

    printf("The vector v1 contains elements:");
    it_v = vector_begin(pvec_v1);
    for(; !iterator_equal(it_v, vector_end(pvec_v1)); it_v = iterator_next(it_v))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_v));
    }
    printf("\n");

    printf("The vector v1 now contains elements:");
    it_v = vector_begin(pvec_v1);
    *(int*)iterator_get_pointer(it_v) = 20;
    for(; !iterator_equal(it_v, vector_end(pvec_v1)); it_v = iterator_next(it_v))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_v));
    }
    printf("\n");

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The vector v1 contains elements: 1 2
The vector v1 now contains elements: 20 2
```

8. vector_capacity

返回 vector_t 在不重新分配内存时能够保存的数据的个数。

```
size_t vector_capacity(
    const vector_t* cpvec_vector
);
```

- **Parameters**

cpvec_vector: 指向 vector_t 类型的指针。

- **Remarks**

返回 vector_t 在不重新分配内存时能够保存的数据的个数，这个值不是容器中实际的数据。当容器中插入的数据超过了这个值，vector_t 容器要重新分配足够的内存。

- **Requirements**

头文件 <cstl/cvector.h>

- **Example**

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

#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("The length of storage allocated is %d.\n",
        vector_capacity(pvec_v1));

    vector_push_back(pvec_v1, 2);
    vector_push_back(pvec_v1, 3);
    printf("The length of storage allocated is now %d.\n",
        vector_capacity(pvec_v1));

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The length of storage allocated is 2.
The length of storage allocated is now 4.
```

9. vector_clear

删除 vector_t 中的所有数据。

```
void vector_clear(
    vector_t* pvec_vector
);
```

- **Parameters**

  pvec_vector: 指向 vector_t 类型的指针。

- **Requirements**

  头文件 <cstl/cvector.h>

- **Example**

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

#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, 10);
    vector_push_back(pvec_v1, 20);
    vector_push_back(pvec_v1, 30);

    printf("The size of v1 is %d\n", vector_size(pvec_v1));
    vector_clear(pvec_v1);
    printf("The size of v1 after clearing is %d\n", vector_size(pvec_v1));

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The size of v1 is 3
The size of v1 after clearing is 0
```

10. vector_destroy

销毁 vector_t 类型。

```
void vector_destroy(
    vector_t* pvec_vector
);
```

- **Parameters**

  pvec_vector: 指向 vector_t 类型的指针。

- **Remarks**

  在 vector_t 类型使用完后，一定要销毁，否则 vector_t 占用的资源不会被释放。

- **Requirements**

  头文件 <cstl/cvector.h>

- **Example**

  请参考 vector _t 类型的其他操作函数。

11. vector_empty

测试 vector_t 是否为空。

```
bool_t vector_empty(
    const vector_t* cpvec_vector
);
```

- **Parameters**

  cpvec_vector: 指向 vector_t 类型的指针。

- **Requirements**

头文件 <cstl/cvector.h>

- **Example**

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

#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);

    if(vector_empty(pvec_v1))
    {
        printf("The vector is empty.\n");
    }
    else
    {
        printf("The vector is not empty.\n");
    }

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The vector is not empty.
```

12. vector_end

返回指向 vector_t 末尾的迭代器。

```
vector_iterator_t vector_end(
    const vector_t* cpvec_vector
);
```

- **Parameters**

  cpvec_vector: 指向 vector_t 类型的指针。

- **Remarks**

  vector_t 容器时，函数的返回值与 vector_begin()相等。

- **Requirements**

头文件 <cstl/cvector.h>

- **Example**

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

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

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

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

    vector_init(pvec_v1);

    vector_push_back(pvec_v1, 1);
    vector_push_back(pvec_v1, 2);

    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d\n", *(int*)iterator_get_pointer(it_v));
    }

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
1
2
```

13. vector_equal

测试两个 vector_t 是否相等。

```
bool_t vector_equal(
    const vector_t* cpvec_first,
    const vector_t* cpvec_second
);
```

- **Parameters**

cpvec_first: 指向第一个 vector_t 类型的指针。
cpvec_second: 指向第二个 vector_t 类型的指针。

- **Remarks**

两个 vector_t 中的数据对应相等，并且数量相等，函数返回 true，否则返回 false。如果两个 vector_t 中的数据

类型不同也认为不等。

- **Requirements**
  头文件 <cstl/cvector.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);

    if(pvec_v1 == NULL || pvec_v2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);

    vector_push_back(pvec_v1, 1);
    vector_push_back(pvec_v2, 1);

    if(vector_equal(pvec_v1, pvec_v2))
    {
        printf("Vectors equal.\n");
    }
    else
    {
        printf("Vectors not equal.\n");
    }

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);

    return 0;
}
```

- **Output**

```
Vectors equal.
```

14. vector_erase　vector_erase_range

删除 vector_t 中指定的数据或者是数据区间。

```
vector_iterator_t vector_erase(
    vector_t* pvec_vector,
    vector_iterator_t it_pos
);
```

```
vector_iterator_t vector_erase_range(
    vector_t* pvec_vector,
    vector_iterator_t it_begin,
    vector_iterator_t it_end
);
```

- **Parameters**
  pvec_vector: 指向 vector_t 类型的指针。
  it_pos: 指向被删除数据的迭代器。
  it_begin: 指向被删除数据区间的开始位置迭代器。
  it_end: 指向被删除数据区间的末尾位置迭代器。

- **Remarks**
  函数中的迭代器和数据区间必须是有效的，无效的参数导致函数的行为未定义。

- **Requirements**
  头文件 <cstl/cvector.h>

- **Example**

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

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

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

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

    vector_init(pvec_v1);

    vector_push_back(pvec_v1, 10);
    vector_push_back(pvec_v1, 20);
    vector_push_back(pvec_v1, 30);
    vector_push_back(pvec_v1, 40);
    vector_push_back(pvec_v1, 50);

    printf("v1 =");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_v));
    }
    printf("\n");

    vector_erase(pvec_v1, vector_begin(pvec_v1));
    printf("v1 =");
    for(it_v = vector_begin(pvec_v1);
```

```
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_v));
    }
    printf("\n");

    vector_erase_range(pvec_v1, iterator_next(vector_begin(pvec_v1)),
        iterator_next_n(vector_begin(pvec_v1), 3));
    printf("v1 =");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_v));
    }
    printf("\n");

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
v1 = 10 20 30 40 50
v1 = 20 30 40 50
v1 = 20 50
```

15. vector_front

访问 vector_t 中的第一个数据。

```
void* vector_front(
    const vector_t* cpvec_vector
);
```

- **Parameters**

  cpvec_vector: 指向 vector_t 类型的指针。

- **Remarks**

  vector_t 容器为空时返回 NULL。

- **Requirements**

  头文件 <cstl/cvector.h>

- **Example**

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

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

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

```
    vector_t* pvec_v1 = create_vector(int);
    int* pn_i = NULL;
    int* pn_j = NULL;

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

    vector_init(pvec_v1);

    vector_push_back(pvec_v1, 10);
    vector_push_back(pvec_v1, 11);

    pn_i = (int*)vector_front(pvec_v1);
    pn_j = (int*)vector_front(pvec_v1);

    printf("The first integer of v1 is %d\n", *pn_i);
    (*pn_i)--;
    printf("The Modified first integer of v1 is %d\n", *pn_j);

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The first integer of v1 is 10
The Modified first integer of v1 is 9
```

16. vector_greater

测试第一个 vector_t 是否大于第二个 vector_t。

```
bool_t vector_greater(
    const vector_t* cpvec_first,
    const vector_t* cpvec_second
);
```

- **Parameters**

  cpvec_first: 指向第一个 vector_t 类型的指针。
  cpvec_second: 指向第二个 vector_t 类型的指针。

- **Remarks**

  要求两个 vector_t 保存的数据类型相同，如果数据类型不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cvector.h>

- **Example**

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

#include <stdio.h>
```

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

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);

    if(pvec_v1 == NULL || pvec_v2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);

    vector_push_back(pvec_v1, 1);
    vector_push_back(pvec_v1, 3);
    vector_push_back(pvec_v1, 1);

    vector_push_back(pvec_v2, 1);
    vector_push_back(pvec_v2, 2);
    vector_push_back(pvec_v2, 2);

    if(vector_greater(pvec_v1, pvec_v2))
    {
        printf("Vector v1 is greater than vector v2.\n");
    }
    else
    {
        printf("Vector v1 is not greater than vector v2.\n");
    }

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);

    return 0;
}
```

- **Output**

```
Vector v1 is greater than vector v2.
```

17. vector_greater_equal

测试第一个 vector_t 是否大于等于第二个 vector_t。

```
bool_t vector_greater_equal(
    const vector_t* cpvec_first,
    const vector_t* cpvec_second
);
```

- **Parameters**

  cpvec_first: 指向第一个 vector_t 类型的指针。
  cpvec_second: 指向第二个 vector_t 类型的指针。

- **Remarks**

  要求两个 vector_t 保存的数据类型相同，如果数据类型不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cvector.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);

    if(pvec_v1 == NULL || pvec_v2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);

    vector_push_back(pvec_v1, 1);
    vector_push_back(pvec_v1, 3);
    vector_push_back(pvec_v1, 1);

    vector_push_back(pvec_v2, 1);
    vector_push_back(pvec_v2, 2);
    vector_push_back(pvec_v2, 2);

    if(vector_greater_equal(pvec_v1, pvec_v2))
    {
        printf("Vector v1 is greater than or equal to vector v2.\n");
    }
    else
    {
        printf("Vector v1 is less than vector v2.\n");
    }

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);

    return 0;
}
```

- **Output**

```
Vector v1 is greater than or equal to vector v2.
```

18. vector_init vector_init_copy vector_init_copy_range vector_init_elem vector_init_n

初始化 vector_t 类型。

```
void vector_init(
```

```
    vector_t* pvec_vector
);

void vector_init_copy(
    vector_t* pvec_vector,
    const vector_t* cpvec_src
);

void vector_init_copy_range(
    vector_t* pvec_vector,
    vector_iterator_t it_begin,
    vector_iterator_t it_end
);

void vector_init_elem(
    vector_t* pvec_vector,
    size_t t_count,
    element
);

void vector_init_n(
    vector_t* pvec_vector,
    size_t t_count
);
```

- **Parameters**

pvec_vector: 指向被初始化的 vector_t 类型的指针。
cpvec_src: 指向源 vector_t 类型的指针。
it_begin: 用于初始化的指定数据区间的开始。
it_end: 用于初始化的指定数据区间的末尾。
t_count: 指定数据的个数。
element: 指定数据。

- **Remarks**

第一个函数初始化一个空的 vector_t 类型。
第二个函数使用已经存在的 vector_t 类型初始化 vector_t 类型。
第三个函数使用指定的数据初始化 vector_t 类型。
第四个函数使用指定数据初始化 vector_t 类型。
第五个函数使用多个默认数据初始化 vector_t 类型。
上面这些函数都要求迭代器和数据区间是有效的，否则导致函数行为未定义。

- **Requirements**

头文件 <cstl/cvector.h>

- **Example**

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

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

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

```
{
    vector_t* pvec_v0 = create_vector(int);
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_t* pvec_v3 = create_vector(int);
    vector_t* pvec_v4 = create_vector(int);
    vector_iterator_t it_v;

    if(pvec_v0 == NULL || pvec_v1 == NULL ||
       pvec_v2 == NULL || pvec_v3 == NULL ||
       pvec_v4 == NULL)
    {
        return -1;
    }

    /* Create an empty vector v0 */
    vector_init(pvec_v0);

    /* Create a vector v1 with 3 elements of default value 0 */
    vector_init_n(pvec_v1, 3);

    /* Create a vector v2 with 5 elements of value 2 */
    vector_init_elem(pvec_v2, 5, 2);

    /* Create a copy, vector v3, of vector v2 */
    vector_init_copy(pvec_v3, pvec_v2);

    /* Create a vector v4 by copying the range v4[first, last) */
    vector_init_copy_range(pvec_v4, iterator_next(vector_begin(pvec_v3)),
        iterator_next_n(vector_begin(pvec_v3), 3));

    printf("v1 =");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_v));
    }
    printf("\n");

    printf("v2 =");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_v));
    }
    printf("\n");

    printf("v3 =");
    for(it_v = vector_begin(pvec_v3);
        !iterator_equal(it_v, vector_end(pvec_v3));
        it_v = iterator_next(it_v))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_v));
    }
    printf("\n");

    printf("v4 =");
    for(it_v = vector_begin(pvec_v4);
```

```
        !iterator_equal(it_v, vector_end(pvec_v4));
        it_v = iterator_next(it_v))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_v));
    }
    printf("\n");

    vector_destroy(pvec_v0);
    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);
    vector_destroy(pvec_v3);
    vector_destroy(pvec_v4);

    return 0;
}
```

- **Output**

```
v1 = 0 0 0
v2 = 2 2 2 2 2
v3 = 2 2 2 2 2
v4 = 2 2
```

19. vector_insert vector_insert_n vector_insert_range

在 vector_t 的指定位置插入数据。

```
vector_iterator_t vector_insert(
    vector_t* pvec_vector,
    vector_iterator_t it_pos,
    element
);

vector_iterator_t vector_insert_n(
    vector_t* pvec_vector,
    vector_iterator_t it_pos,
    size_t t_count,
    element
);

void vector_insert_range(
    vector_t* pvec_vector,
    vector_iterator_t it_pos,
    vector_iterator_t it_begin,
    vector_iterator_t it_end
);
```

- **Parameters**

pvec_vector: 指向 vector_t 类型的指针。
it_pos: 插入数据的位置迭代器。
t_count: 指定数据的个数。
element: 指定数据。
it_begin: 指定数据区间的开始。
it_end: 指定数据区间的末尾。

- **Remarks**

上面这些函数都要求迭代器和数据区间是有效的，否则导致函数行为未定义。

- **Requirements**

  头文件 <cstl/cvector.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_iterator_t it_v;

    if(pvec_v1 == NULL || pvec_v2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);

    vector_push_back(pvec_v1, 10);
    vector_push_back(pvec_v1, 20);
    vector_push_back(pvec_v1, 30);

    vector_init_copy(pvec_v2, pvec_v1);

    printf("v1 =");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_v));
    }
    printf("\n");

    vector_insert(pvec_v1, iterator_next(vector_begin(pvec_v1)), 40);
    printf("v1 =");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_v));
    }
    printf("\n");

    vector_insert_n(pvec_v1, iterator_next_n(vector_begin(pvec_v1), 2), 4, 50);
    printf("v1 =");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_v));
```

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

    vector_insert_range(pvec_v1, iterator_next(vector_begin(pvec_v1)),
        vector_begin(pvec_v2), vector_end(pvec_v2));
    printf("v1 =");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_v));
    }
    printf("\n");

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);

    return 0;
}
```

- **Output**

```
v1 = 10 20 30
v1 = 10 40 20 30
v1 = 10 40 50 50 50 50 20 30
v1 = 10 10 20 30 40 50 50 50 50 20 30
```

20. vector_less

测试第一个 vector_t 是否小于第二个 vector_t。

```
bool_t vector_less(
    const vector_t* cpvec_first,
    const vector_t* cpvec_second
);
```

- **Parameters**

  cpvec_first: 指向第一个 vector_t 类型的指针。
  cpvec_second: 指向第二个 vector_t 类型的指针。

- **Remarks**

  要求两个 vector_t 保存的数据类型相同，如果数据类型不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cvector.h>

- **Example**

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

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

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

```
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);

    if(pvec_v1 == NULL || pvec_v2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);

    vector_push_back(pvec_v1, 1);
    vector_push_back(pvec_v1, 2);
    vector_push_back(pvec_v1, 4);

    vector_push_back(pvec_v2, 1);
    vector_push_back(pvec_v2, 3);

    if(vector_less(pvec_v1, pvec_v2))
    {
        printf("Vector v1 is less than vector v2.\n");
    }
    else
    {
        printf("Vector v1 is not less than vector v2.\n");
    }

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);

    return 0;
}
```

- **Output**

```
Vector v1 is less than vector v2.
```

21. vector_less_equal

测试第一个 vector_t 是否小于等于第二个 vector_t。

```
bool_t vector_less_equal(
    const vector_t* cpvec_first,
    const vector_t* cpvec_second
);
```

- **Parameters**

cpvec_first: 指向第一个 vector_t 类型的指针。
cpvec_second: 指向第二个 vector_t 类型的指针。

- **Remarks**

要求两个 vector_t 保存的数据类型相同，如果数据类型不同导致函数的行为未定义。

- **Requirements**

头文件 <cstl/cvector.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);

    if(pvec_v1 == NULL || pvec_v2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);

    vector_push_back(pvec_v1, 1);
    vector_push_back(pvec_v1, 2);
    vector_push_back(pvec_v1, 4);

    vector_push_back(pvec_v2, 1);
    vector_push_back(pvec_v2, 3);

    if(vector_less_equal(pvec_v1, pvec_v2))
    {
        printf("Vector v1 is less than or equal to vector v2.\n");
    }
    else
    {
        printf("Vector v1 is greater than vector v2.\n");
    }

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);

    return 0;
}
```

- **Output**

```
Vector v1 is less than or equal to vector v2.
```

22. vector_max_size

返回 vector_t 中能够保存的数据最大数目的可能值。

```c
size_t vector_max_size(
    const vector_t* cpvec_vector
);
```

- **Parameters**

cpvec_vector: 指向 vector_t 类型的指针。

- **Remarks**

  这是一个与系统相关的常量。

- **Requirements**

  头文件 <cstl/cvector.h>

- **Example**

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

#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);

    printf("The maximum possible length of the vector is %d.\n",
        vector_max_size(pvec_v1));

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The maximum possible length of the vector is 1073741823.
```

23. vector_not_equal

测试两个 vector_t 是否不等。

```
bool_t vector_not_equal(
    const vector_t* cpvec_first,
    const vector_t* cpvec_second
);
```

- **Parameters**

  cpvec_first: 指向第一个 vector_t 类型的指针。
  cpvec_second: 指向第二个 vector_t 类型的指针。

- **Remarks**

  两个 vector_t 中的数据对应相等，并且数量相等，函数返回 false，否则返回 true。如果两个 vector_t 中的数据类型不同也认为不等。

- **Requirements**

头文件 <cstl/cvector.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);

    if(pvec_v1 == NULL || pvec_v2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);

    vector_push_back(pvec_v1, 1);
    vector_push_back(pvec_v2, 2);

    if(vector_not_equal(pvec_v1, pvec_v2))
    {
        printf("Vectors not equal.\n");
    }
    else
    {
        printf("Vectors equal.\n");
    }

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);

    return 0;
}
```

- **Output**

```
Vectors not equal.
```

24. vector_pop_back

删除 vector_t 中的最后一个数据。

```
void vector_pop_back(
    vector_t* pvec_vector
);
```

- **Parameters**

pvec_vector: 指向 vector_t 类型的指针。

- **Remarks**

vector_t 容器为空函数的行为未定义。

- **Requirements**
  头文件 <cstl/cvector.h>

- **Example**

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

#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("%d\n", *(int*)vector_back(pvec_v1));
    vector_push_back(pvec_v1, 2);
    printf("%d\n", *(int*)vector_back(pvec_v1));
    vector_pop_back(pvec_v1);
    printf("%d\n", *(int*)vector_back(pvec_v1));

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
1
2
1
```

25. vector_push_back

向 vector_t 的末尾添加一个数据。

```
void vector_push_back(
    vector_t* pvec_vector,
    element
);
```

- **Parameters**
  pvec_vector: 指向 vector_t 类型的指针。

- **Requirements**
  头文件 <cstl/cvector.h>

- **Example**

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

#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);
    if(vector_size(pvec_v1) != 0)
    {
        printf("Last element: %d\n", *(int*)vector_back(pvec_v1));
    }

    vector_push_back(pvec_v1, 2);
    if(vector_size(pvec_v1) != 0)
    {
        printf("New last element: %d\n", *(int*)vector_back(pvec_v1));
    }

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
Last element: 1
New last element: 2
```

26. vector_reserve

设置 vector_t 在未重新分配内存时能够保存的数据的数量。

```
void vector_reserve(
    vector_t* pvec_vector,
    size_t t_size
);
```

- **Parameters**

pvec_vector: 指向 vector_t 类型的指针。
t_size: 在 vector_t 未重新分配内存时能够保存的数据的数量。

- **Remarks**

当新的数据数量大于当前数据数量时导致 vector_t 重新分配内存，当新的数据数据小于当前数据数量是当前数

量不变。

- **Requirements**

  头文件 <cstl/cvector.h>

- **Example**

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

#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("Current capacity of v1 = %d\n", vector_capacity(pvec_v1));
    vector_reserve(pvec_v1, 20);
    printf("Current capacity of v1 = %d\n", vector_capacity(pvec_v1));

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
Current capacity of v1 = 2
Current capacity of v1 = 20
```

27. vector_resize vector_resize_elem

重新设置 vector_t 中实际数据的数量。

```
void vector_resize(
    vector_t* pvec_vector,
    size_t t_resize
);
```

- **Parameters**

  pvec_vector: 指向 vector_t 类型的指针。
  t_resize: 新的数据的数量。

- **Remarks**

当新的数据数量大于当前数据数量时第一个函数使用默认数据填充，第二个函数使用指定数据填充，新数量大于 vector_capacity()时导致内存重新分配。当新的数据数据小于当前数据数量是当前数量不变。

- **Requirements**

头文件 <cstl/cvector.h>

- **Example**

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

#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, 10);
    vector_push_back(pvec_v1, 20);
    vector_push_back(pvec_v1, 30);

    vector_resize_elem(pvec_v1, 4, 40);
    printf("The size of v1 is %d\n", vector_size(pvec_v1));
    printf("The value of the last object is %d\n", *(int*)vector_back(pvec_v1));

    vector_resize(pvec_v1, 5);
    printf("The size of v1 is now %d\n", vector_size(pvec_v1));
    printf("The value of the last object is now %d\n", *(int*)vector_back(pvec_v1));

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The size of v1 is 4
The value of the last object is 40
The size of v1 is now 5
The value of the last object is now 0
```

28. vector_size

返回 vector_t 中数据的数量。

```
size_t vector_size(
    const vector_t* cpvec_vector
);
```

- **Parameters**

cpvec_vector: 指向 vector_t 类型的指针。

- **Requirements**

  头文件 <cstl/cvector.h>

- **Example**

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

#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;
}
```

- **Output**

```
Vector length is 1.
Vector length is now 2.
```

29. vector_swap

交换两个 vector_t 中的内容。

```
void vector_swap(
    vector_t* pvec_first,
    vector_t* pvec_second
);
```

- **Parameters**

  pvec_first: 指向第一个 vector_t 类型的指针。
  pvec_second: 指向第二个 vector_t 类型的指针。

- **Remarks**

  要求两个 vector_t 保存的数据类型相同，如果数据类型不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cvector.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);

    if(pvec_v1 == NULL || pvec_v2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);

    vector_push_back(pvec_v1, 1);
    vector_push_back(pvec_v1, 2);
    vector_push_back(pvec_v1, 3);

    vector_push_back(pvec_v2, 10);
    vector_push_back(pvec_v2, 20);

    printf("The number of elements in v1 = %d\n", vector_size(pvec_v1));
    printf("The number of elements in v2 = %d\n", vector_size(pvec_v2));
    printf("\n");

    vector_swap(pvec_v1, pvec_v2);

    printf("The number of elements in v1 = %d\n", vector_size(pvec_v1));
    printf("The number of elements in v2 = %d\n", vector_size(pvec_v2));

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);

    return 0;
}
```

- **Output**

```
The number of elements in v1 = 3
The number of elements in v2 = 2

The number of elements in v1 = 2
The number of elements in v2 = 3
```

第五节　集合 set_t

集合容器 set_t 是关联容器，set_t 中的数据按照键和指定的规则自动排序并且保证键是唯一的，set_t 中的键就是数据本身。set_t 中的数据不可以直接或者通过迭代器修改，因为这样会破会 set_t 中数据的有序性，要想修改一个

数据只有先删除它然后插入新的数据。set_t 支持双向迭代器。插入新数据是不会破坏原有的迭代器，删除数据是只有指向被删除的数据的迭代器失效。set_t 对于数据的查找，插入和删除都是高效的。set_t 中的数据根据指定的规则自动排序，默认的排序规则是使用数据的小于操作符，用户可以在初始化时指定自定义的排序规则。

- **Typedefs**

set_t	集合容器类型。
set_iterator_t	集合容器迭代器类型。

- **Operation Functions**

create_set	创建集合容器类型。
set_assign	为集合容器赋值。
set_begin	返回指向集合中第一个数据的迭代器。
set_clear	删除集合容器中的所有数据。
set_count	返回集合容器中包含指定数据的个数。
set_destroy	销毁集合容器。
set_empty	测试集合容器是否为空。
set_end	返回指向集合容器末尾位置的迭代器。
set_equal	测试两个集合容器是否相等。
set_equal_range	返回一个集合容器中包含指定数据的数据区间。
set_erase	删除集合容器中与指定数据相等的数据。
set_erase_pos	删除集合容器中指定位置的数据。
set_erase_range	删除集合容器中指定数据区间的数据。
set_find	在集合容器中查找指定的数据。
set_greater	测试第一个集合是否大于第二个集合。
set_greater_equal	测试第一个集合是否大于等于第二个集合。
set_init	初始化一个空的集合容器。
set_init_copy	使用一个集合容器的内容来初始化当前集合容器。
set_init_copy_range	使用指定的数据区间初始化集合容器。
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集合容器。
set_init_ex	使用指定的排序规则初始化一个空的集合容器。
set_insert	向集合中插入一个数据。
set_insert_hint	向集合中插入一个数据同时给出位置提示。
set_insert_range	向集合中插入指定数据区间的数据。
set_key_comp	返回集合容器的键比较规则。
set_less	测试第一个集合容器是否小于第二个集合容器。
set_less_equal	测试第一个集合容器是否小于等于第二个集合容器。
set_lower_bound	返回集合中与指定数据相等的第一个数据的迭代器。
set_max_size	返回集合中能够保存的数据个数的最大可能值。
set_not_equal	测试两个集合是否不等。

set_size	返回集合中保存的数据的数量。
set_swap	交换两个集合的内容。
set_upper_bound	返回集合中大于指定数据的第一个数据的迭代器。
set_value_comp	获得集合中的数据比较规则。

1. set_t

集合容器类型。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

请参考 set_t 类型的其他操作函数。

2. set_iterator_t

set_t 类型的迭代器类型。

- **Remarks**

set_iterator_t 是双向迭代器类型，不能通过迭代器来修改容器中的数据。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

请参考 set_t 类型的其他操作函数。

3. create_set

创建 set_t 类型。

```
set_t* create_set(
    type
);
```

- **Parameters**

type: 数据类型描述。

- **Remarks**

函数成功返回指向 set_t 类型的指针，失败返回 NULL。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

请参考 set_t 类型的其他操作函数。

4. set_assign

使用 set_t 类型为当前的 set_t 赋值。

```
void set_assign(
    set_t* pset_dest,
    const set_t* cpset_src
);
```

- **Parameters**

pset_dest: 指向被赋值的 set_t 类型的指针。
cpset_src: 指向赋值的 set_t 类型的指针。

- **Remarks**

要求两个 set_t 类型保存的数据具有相同的类型，否则函数的行为未定义。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    set_t* pset_s1 = create_set(int);
    set_t* pset_s2 = create_set(int);
    set_iterator_t it_s;

    if(pset_s1 == NULL || pset_s2 == NULL)
    {
        return -1;
    }

    set_init(pset_s1);
    set_init(pset_s2);

    set_insert(pset_s1, 10);
    set_insert(pset_s1, 20);
    set_insert(pset_s1, 30);
    set_insert(pset_s2, 40);
    set_insert(pset_s2, 50);
    set_insert(pset_s2, 60);

    printf("s1 =");
    for(it_s = set_begin(pset_s1);
        !iterator_equal(it_s, set_end(pset_s1));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");
```

```
    set_assign(pset_s1, pset_s2);
    printf("s1 =");
    for(it_s = set_begin(pset_s1);
        !iterator_equal(it_s, set_end(pset_s1));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");

    set_destroy(pset_s1);
    set_destroy(pset_s2);

    return 0;
}
```

- **Output**

```
s1 = 10 20 30
s1 = 40 50 60
```

5. set_begin

返回指向 set_t 第一个数据的迭代器。

```
set_iterator_t set_begin(
    const set_t* cpset_set
);
```

- **Parameters**

cpset_set: 指向 set_t 类型的指针。

- **Remarks**

如果 set_t 为空，这个函数的返回值和 set_end()的返回值相等。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    set_t* pset_s1 = create_set(int);

    if(pset_s1 == NULL)
    {
        return -1;
    }

    set_init(pset_s1);
```

```
    set_insert(pset_s1, 1);
    set_insert(pset_s1, 2);
    set_insert(pset_s1, 3);

    printf("The first element of s1 is %d\n",
        *(int*)iterator_get_pointer(set_begin(pset_s1)));

    set_erase_pos(pset_s1, set_begin(pset_s1));
    printf("The first element of s1 is now %d\n",
        *(int*)iterator_get_pointer(set_begin(pset_s1)));

    set_destroy(pset_s1);

    return 0;
}
```

- **Output**

```
The first element of s1 is 1
The first element of s1 is now 2
```

6. set_clear

删除 set_t 中的所有数据。

```
void set_clear(
    set_t* pset_set
);
```

- **Parameters**

pset_set: 指向 set_t 类型的指针。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    set_t* pset_s1 = create_set(int);

    if(pset_s1 == NULL)
    {
        return -1;
    }

    set_init(pset_s1);

    set_insert(pset_s1, 1);
    set_insert(pset_s1, 2);
```

```
    printf("The size of the set is initially %d.\n", set_size(pset_s1));

    set_clear(pset_s1);
    printf("The size of the set after clearing is %d.\n", set_size(pset_s1));

    set_destroy(pset_s1);

    return 0;
}
```

- **Output**

```
The size of the set is initially 2.
The size of the set after clearing is 0.
```

7. set_count

返回容器中包含指定数据的个数。

```
size_t _set_count(
    const set_t* cpset_set,
    element
);
```

- **Parameters**

cpset_set: 指向 set_t 类型的指针。
element: 指定的数据。

- **Remarks**

如果容器中不包含指定数据则返回 0，包含则返回指定数据的个数，集合中返回的都是 1。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    set_t* pset_s1 = create_set(int);

    if(pset_s1 == NULL)
    {
        return -1;
    }

    set_init(pset_s1);

    set_insert(pset_s1, 1);
    set_insert(pset_s1, 1);
```

```
    /* Keys must be unique in set, so duplicates are ignored */
    printf("The number of elements in s1 with a sort key of 1 is: %d.\n",
        set_count(pset_s1, 1));
    printf("The number of elements in s1 with a sort key of 2 is: %d.\n",
        set_count(pset_s1, 2));

    set_destroy(pset_s1);

    return 0;
}
```

- **Output**

```
The number of elements in s1 with a sort key of 1 is: 1.
The number of elements in s1 with a sort key of 2 is: 0.
```

8. set_destroy

销毁 set_t 容器。

```
void set_destroy(
    set_t* pset_set
);
```

- **Parameters**

  pset_set: 指向 set_t 类型的指针。

- **Remarks**

  set_t 容器使用之后要销毁，否则 set_t 占用的资源不会被释放。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

  请参考 set_t 类型的其他操作函数。

9. set_empty

测试 set_t 容器是否为空。

```
bool_t set_empty(
    const set_t* cpset_set
);
```

- **Parameters**

  cpset_set: 指向 set_t 类型的指针。

- **Remarks**

  set_t 容器为空则返回 true，否则返回 false。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    set_t* pset_s1 = create_set(int);
    set_t* pset_s2 = create_set(int);

    if(pset_s1 == NULL || pset_s2 == NULL)
    {
        return -1;
    }

    set_init(pset_s1);
    set_init(pset_s2);

    set_insert(pset_s1, 1);

    if(set_empty(pset_s1))
    {
        printf("The set s1 is empty.\n");
    }
    else
    {
        printf("The set s1 is not empty.\n");
    }

    if(set_empty(pset_s2))
    {
        printf("The set s2 is empty.\n");
    }
    else
    {
        printf("The set s2 is not empty.\n");
    }

    set_destroy(pset_s1);
    set_destroy(pset_s2);

    return 0;
}
```

- **Output**

```
The set s1 is not empty.
The set s2 is empty.
```

10. set_end

返回指向 set_t 末尾位置的迭代器。

```c
set_iterator_t set_end(
    const set_t* cpset_set
);
```

- **Parameters**

  cpset_set: 指向 set_t 类型的指针。

- **Remarks**

  如果 set_t 为空，这个函数的返回值和 set_begin()的返回值相等。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    set_t* pset_s1 = create_set(int);
    set_iterator_t it_s;

    if(pset_s1 == NULL)
    {
        return -1;
    }

    set_init(pset_s1);

    set_insert(pset_s1, 1);
    set_insert(pset_s1, 2);
    set_insert(pset_s1, 3);

    it_s = set_end(pset_s1);
    it_s = iterator_prev(it_s);
    printf("The last element of s1 is %d\n",
        *(int*)iterator_get_pointer(it_s));

    set_erase_pos(pset_s1, it_s);

    it_s = set_end(pset_s1);
    it_s = iterator_prev(it_s);
    printf("The last element of s1 is now %d\n",
        *(int*)iterator_get_pointer(it_s));

    set_destroy(pset_s1);

    return 0;
}
```

- **Output**

```
The last element of s1 is 3
The last element of s1 is now 2
```

11. set_equal

测试两个 set_t 是否相等。

```
bool_t set_equal(
    const set_t* cpset_first,
    const set_t* cpset_second
);
```

- **Parameters**

  cpset_first: 指向第一个 set_t 类型的指针。
  cpset_second: 指向第二个 set_t 类型的指针。

- **Remarks**

两个 set_t 中的数据对应相等，并且数量相等，函数返回 true，否则返回 false。如果两个 set_t 中的数据类型不同也认为不等。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    set_t* pset_s1 = create_set(int);
    set_t* pset_s2 = create_set(int);
    set_t* pset_s3 = create_set(int);
    int i = 0;

    if(pset_s1 == NULL || pset_s2 == NULL || pset_s3 == NULL)
    {
        return -1;
    }

    set_init(pset_s1);
    set_init(pset_s2);
    set_init(pset_s3);

    for(i = 0; i < 3; ++i)
    {
        set_insert(pset_s1, i);
        set_insert(pset_s2, i * i);
        set_insert(pset_s3, i);
    }

    if(set_equal(pset_s1, pset_s2))
    {
        printf("The sets s1 and s2 are equal.\n");
    }
    else
    {
```

```
        printf("The sets s1 and s2 are not equal.\n");
    }

    if(set_equal(pset_s1, pset_s3))
    {
        printf("The sets s1 and s3 are equal.\n");
    }
    else
    {
        printf("The sets s1 and s3 are not equal.\n");
    }

    set_destroy(pset_s1);
    set_destroy(pset_s2);
    set_destroy(pset_s3);

    return 0;
}
```

- **Output**

```
The sets s1 and s2 are not equal.
The sets s1 and s3 are equal.
```

12. set_equal_range

返回 set_t 中包含指定数据的数据区间。

```
range_t set_equal_range(
    const set_t* cpset_set,
    element
);
```

- **Parameters**

cpset_set: 指向 set_t 类型的指针。
element: 指定的数据。

- **Remarks**

返回 set_t 中包含指定数据的数据区间[range_t.it_begin, range_t.it_end)，其中 it_begin 是指向等于指定数据的第一个数据的迭代器，it_end 指向的是大于指定数据的第一个数据的迭代器。如果 set_t 中不包含指定数据则 it_begin 与 it_end 相等。如果指定的数据是 set_t 中最大的数据则 it_end 等于 set_end()。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

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

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

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

```
    set_t* pset_s1 = create_set(int);
    set_iterator_t it_s;
    range_t r_r1;

    if(pset_s1 == NULL)
    {
        return -1;
    }

    set_init(pset_s1);

    set_insert(pset_s1, 10);
    set_insert(pset_s1, 20);
    set_insert(pset_s1, 30);

    r_r1 = set_equal_range(pset_s1, 20);

    printf("The upper bound of the element with a key of 20 in the set s1 is:
%d.\n",
        *(int*)iterator_get_pointer(r_r1.it_end));
    printf("The lower bound of the element with a key of 20 in the set s1 is:
%d.\n",
        *(int*)iterator_get_pointer(r_r1.it_begin));

    /* Compare the upper_bound called directly */
    it_s = set_upper_bound(pset_s1, 20);
    printf("A direct call of upper_bound(20), gives %d,\n",
        *(int*)iterator_get_pointer(it_s));
    printf("matching the 2nd element of the range returned by equal_range(20).\n");

    r_r1 = set_equal_range(pset_s1, 40);
    /* If no match is found for the key. both elements of the range return end() */
    if(iterator_equal(r_r1.it_begin, set_end(pset_s1)) &&
       iterator_equal(r_r1.it_end, set_end(pset_s1)))
    {
        printf("The set s1 doesn't have and element with a key less than 40.\n");
    }
    else
    {
        printf("The element of set s1 with a key >= 40 is: %d.\n",
            *(int*)iterator_get_pointer(r_r1.it_begin));
    }

    set_destroy(pset_s1);

    return 0;
}
```

- **Output**

```
The upper bound of the element with a key of 20 in the set s1 is: 30.
The lower bound of the element with a key of 20 in the set s1 is: 20.
A direct call of upper_bound(20), gives 30,
matching the 2nd element of the range returned by equal_range(20).
The set s1 doesn't have and element with a key less than 40.
```

13. set_erase set_erase_pos set_erase_range

删除 set_t 中指定的数据。

```
size_t set_erase(
    set_t* pset_set,
    element
);

void set_erase_pos(
    set_t* pset_set,
    set_iterator_t it_pos
);

void set_erase_range(
    set_t* pset_set,
    set_iterator_t it_begin,
    set_iterator_t it_end
);
```

- **Parameters**

  pset_set: 指向 set_t 类型的指针。
  element: 要删除的数据。
  it_pos: 要删除的数据的位置迭代器。
  it_begin: 要删除的数据区间的开始位置。
  it_end: 要删除的数据区间的末尾位置。

- **Remarks**

  第一个函数删除 set_t 中指定的数据，并返回删除的个数，如果 set_t 中不包含指定的数据就返回 0。
  第二个函数删除指定位置的数据。
  第三个函数删除指定数据区间中的数据。
  后面两个函数要求迭代器和数据区间是有效的，无效的迭代器或数据区间导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    set_t* pset_s1 = create_set(int);
    set_t* pset_s2 = create_set(int);
    set_t* pset_s3 = create_set(int);
    set_iterator_t it_s;
    size_t t_count = 0;
    int i = 0;

    if(pset_s1 == NULL || pset_s2 == NULL || pset_s3 == NULL)
    {
        return -1;
    }
```

```
    set_init(pset_s1);
    set_init(pset_s2);
    set_init(pset_s3);

    for(i = 1; i < 5; ++i)
    {
        set_insert(pset_s1, i);
        set_insert(pset_s2, i * i);
        set_insert(pset_s3, i - 1);
    }

    /* The first function remove an element at a given position */
    set_erase_pos(pset_s1, iterator_next(set_begin(pset_s1)));
    printf("After the second element is deleted, the set s1 is:");
    for(it_s = set_begin(pset_s1);
        !iterator_equal(it_s, set_end(pset_s1));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");

    /* The second function removes elements in the range [first, last) */
    set_erase_range(pset_s2, iterator_next(set_begin(pset_s2)),
        iterator_prev(set_end(pset_s2)));
    printf("After the middlet two elements are deleted, the set s2 is:");
    for(it_s = set_begin(pset_s2);
        !iterator_equal(it_s, set_end(pset_s2));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");

    /* the third function removes elements with a given key */
    t_count = set_erase(pset_s3, 2);
    printf("After the element with a key of 2 is deleted the set s3 is:");
    for(it_s = set_begin(pset_s3);
        !iterator_equal(it_s, set_end(pset_s3));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");

    /* the third function returns the number of elements removed */
    printf("The number of elements removed from s3 is: %d.\n", t_count);

    set_destroy(pset_s1);
    set_destroy(pset_s2);
    set_destroy(pset_s3);

    return 0;
}
```

- **Output**

```
After the second element is deleted, the set s1 is: 1 3 4
After the middlet two elements are deleted, the set s2 is: 1 16
After the element with a key of 2 is deleted the set s3 is: 0 1 3
```

```
The number of elements removed from s3 is: 1.
```

14. set_find

在 set_t 中查找指定的数据。

```
set_iterator_t set_find(
    const set_t* cpset_set,
    element
);
```

- **Parameters**

  cpset_set: 指向 set_t 类型的指针。
  element: 指定的数据。

- **Remarks**

  如果 set_t 中包含指定的数据则返回指向该数据的迭代器，否则返回 set_end()。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    set_t* pset_s1 = create_set(int);
    set_iterator_t it_s;

    if(pset_s1 == NULL)
    {
        return -1;
    }

    set_init(pset_s1);

    set_insert(pset_s1, 10);
    set_insert(pset_s1, 20);
    set_insert(pset_s1, 30);

    it_s = set_find(pset_s1, 20);
    printf("The element of set s1 with a key of 20 is: %d.\n",
        *(int*)iterator_get_pointer(it_s));

    it_s = set_find(pset_s1, 40);
    /* If no match is found for the key, end() is returned */
    if(iterator_equal(it_s, set_end(pset_s1)))
    {
        printf("The set s1 doesn't have an element with a key of 40.\n");
    }
    else
```

```
    {
        printf("The element of set s1 with a key of 40 is: %d.\n",
            *(int*)iterator_get_pointer(it_s));
    }

    /*
     * The element at specific location in the set can be founc
     * by using a dereferenced iterator addressing the location.
     */
    it_s = set_end(pset_s1);
    it_s = iterator_prev(it_s);
    it_s = set_find(pset_s1, *(int*)iterator_get_pointer(it_s));
    printf("The element of s1 with a key matching that"
           " of the last element is: %d.\n",
           *(int*)iterator_get_pointer(it_s));

    set_destroy(pset_s1);

    return 0;
}
```

- **Output**

```
The element of set s1 with a key of 20 is: 20.
The set s1 doesn't have an element with a key of 40.
The element of s1 with a key matching that of the last element is: 30.
```

15. set_greater

测试第一个 set_t 容器是否大于第二个 set_t 容器。

```
bool_t set_greater(
    const set_t* cpset_first,
    const set_t* cpset_second
);
```

- **Parameters**

  cpset_first: 指向第一个 set_t 类型的指针。
  cpset_second: 指向第二个 set_t 类型的指针。

- **Remarks**

  这个函数要求两个 set_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

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

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

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

```
    set_t* pset_s1 = create_set(int);
    set_t* pset_s2 = create_set(int);
    set_t* pset_s3 = create_set(int);
    int i = 0;

    if(pset_s1 == NULL || pset_s2 == NULL || pset_s3 == NULL)
    {
        return -1;
    }

    set_init(pset_s1);
    set_init(pset_s2);
    set_init(pset_s3);

    for(i = 0; i < 3; ++i)
    {
        set_insert(pset_s1, i);
        set_insert(pset_s2, i * i);
        set_insert(pset_s3, i - 1);
    }

    if(set_greater(pset_s1, pset_s2))
    {
        printf("The set s1 is greater than the set s2.\n");
    }
    else
    {
        printf("The set s1 is not greater than the set s2.\n");
    }

    if(set_greater(pset_s1, pset_s3))
    {
        printf("The set s1 is greater than the set s3.\n");
    }
    else
    {
        printf("The set s1 is not greater than the set s3.\n");
    }

    set_destroy(pset_s1);
    set_destroy(pset_s2);
    set_destroy(pset_s3);

    return 0;
}
```

- **Output**

```
The set s1 is not greater than the set s2.
The set s1 is greater than the set s3.
```

16. set_greater_equal

测试第一个 set_t 是否大于等于第二个 set_t。

```
bool_t set_greater_equal(
    const set_t* cpset_first,
    const set_t* cpset_second
);
```

- **Parameters**

  cpset_first: 指向第一个 set_t 类型的指针。
  cpset_second: 指向第二个 set_t 类型的指针。

- **Remarks**

  这个函数要求两个 set_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    set_t* pset_s1 = create_set(int);
    set_t* pset_s2 = create_set(int);
    set_t* pset_s3 = create_set(int);
    set_t* pset_s4 = create_set(int);
    int i = 0;

    if(pset_s1 == NULL || pset_s2 == NULL ||
       pset_s3 == NULL || pset_s4 == NULL)
    {
        return -1;
    }

    set_init(pset_s1);
    set_init(pset_s2);
    set_init(pset_s3);
    set_init(pset_s4);

    for(i = 0; i < 3; ++i)
    {
        set_insert(pset_s1, i);
        set_insert(pset_s2, i * i);
        set_insert(pset_s3, i - 1);
        set_insert(pset_s4, i);
    }

    if(set_greater_equal(pset_s1, pset_s2))
    {
        printf("The set s1 is greater than or equal to the set s2.\n");
    }
    else
    {
        printf("The set s1 is less than the set s2.\n");
    }

    if(set_greater_equal(pset_s1, pset_s3))
    {
```

```
        printf("The set s1 is greater than or equal to the set s3.\n");
    }
    else
    {
        printf("The set s1 is less than the set s3.\n");
    }

    if(set_greater_equal(pset_s1, pset_s4))
    {
        printf("The set s1 is greater than or equal to the set s4.\n");
    }
    else
    {
        printf("The set s1 is less than the set s4.\n");
    }

    set_destroy(pset_s1);
    set_destroy(pset_s2);
    set_destroy(pset_s3);
    set_destroy(pset_s4);

    return 0;
}
```

- **Output**

```
The set s1 is less than the set s2.
The set s1 is greater than or equal to the set s3.
The set s1 is greater than or equal to the set s4.
```

17. set_init set_init_copy set_init_copy_range set_init_copy_range_ex set_init_ex

初始化 set_t 类型。

```
void set_init(
    set_t* pset_set
);

void set_init_copy(
    set_t* pset_set,
    const set_t* cpset_src
);

void set_init_copy_range(
    set_t* pset_set,
    set_iterator_t it_begin,
    set_iterator_t it_end
);

void set_init_copy_range_ex(
    set_t* pset_set,
    set_iterator_t it_begin,
    set_iterator_t it_end,
    binary_function_t bfun_compare
);

void set_init_ex(
```

```
    set_t* pset_set,
    binary_function_t bfun_compare
);
```

- **Parameters**

  pset_set: 指向被初始化 set_t 类型的指针。
  cpset_src: 指向用于初始化的 set_t 类型的指针。
  it_begin: 用于初始化的数据区间的开始位置。
  it_end: 用于初始化的数据区间的末尾位置。
  bfun_compare: 自定义排序规则。

- **Remarks**

  第一个函数初始化一个空的 set_t，使用与数据类型相关的小于操作函数作为默认的排序规则。
  第二个函数使用一个源 set_t 来初始化 set_t，数据的内容和排序规则都从源 set_t 复制。
  第三个函数使用指定的数据区间初始化一个 set_t，使用与数据类型相关的小于操作函数作为默认的排序规则。
  第四个函数使用指定的数据区间初始化一个 set_t，使用用户指定的排序规则。
  第五个函数初始化一个空的 set_t，使用用户指定的排序规则。
  上面的函数要求迭代器和数据区间是有效的，无效的迭代器或数据区间导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    slist_t* pslist_l0 = create_slist(int);
    slist_t* pslist_l1 = create_slist(int);
    slist_t* pslist_l2 = create_slist(int);
    slist_t* pslist_l3 = create_slist(int);
    slist_t* pslist_l4 = create_slist(int);
    slist_iterator_t it_l;

    if(pslist_l0 == NULL || pslist_l1 == NULL ||
       pslist_l2 == NULL || pslist_l3 == NULL ||
       pslist_l4 == NULL)
    {
        return -1;
    }

    /* Create an empty slist l0 */
    slist_init(pslist_l0);

    /* Create a slist l1 with 3 elements of default value 0 */
    slist_init_n(pslist_l1, 3);

    /* Create a slist l2 with 5 elements of value 2 */
    slist_init_elem(pslist_l2, 5, 2);
```

```
    /* Create a copy, slist l3, of slist l3 */
    slist_init_copy(pslist_l3, pslist_l2);

    /* Create a slist l4 by copying the range l3[first, last) */
    slist_init_copy_range(pslist_l4,
        iterator_advance(slist_begin(pslist_l3), 3),
        slist_end(pslist_l3));

    printf("l1 =");
    for(it_l = slist_begin(pslist_l1);
        !iterator_equal(it_l, slist_end(pslist_l1));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    printf("l2 =");
    for(it_l = slist_begin(pslist_l2);
        !iterator_equal(it_l, slist_end(pslist_l2));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    printf("l3 =");
    for(it_l = slist_begin(pslist_l3);
        !iterator_equal(it_l, slist_end(pslist_l3));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    printf("l4 =");
    for(it_l = slist_begin(pslist_l4);
        !iterator_equal(it_l, slist_end(pslist_l4));
        it_l = iterator_next(it_l))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_l));
    }
    printf("\n");

    slist_destroy(pslist_l0);
    slist_destroy(pslist_l1);
    slist_destroy(pslist_l2);
    slist_destroy(pslist_l3);
    slist_destroy(pslist_l4);

    return 0;
}
```

- **Output**

```
s1 = 10 20 30 40
s2 = 20 10
s3 = 10 20 30 40
s4 = 10 20
s5 = 10
```

18. set_insert　set_insert_hint　set_insert_range

向 set_t 中插入数据。

```
set_iterator_t set_insert(
    set_t* pset_set,
    element
);

set_iterator_t set_insert_hint(
    set_t* pset_set,
    set_iterator_t it_hint,
    element
);

void set_insert_range(
    set_t* pset_set,
    set_iterator_t it_begin,
    set_iterator_t it_end
);
```

- **Parameters**

  pset_set: 指向 set_t 类型的指针。
  element: 插入的数据。
  it_hint: 被插入数据的提示位置。
  it_begin: 被插入的数据区间的开始位置。
  it_end: 被插入的数据区间的末尾位置。

- **Remarks**

第一个函数向 set_t 中插入一个指定的数据，成功后返回指向该数据的迭代器，如果 set_t 中包含了该数据那么插入失败，返回 set_end()。

第二个函数向 set_t 中插入一个指定的数据，同时给出一个该数据被插入后的提示位置迭代器，如果这个位置符合 set_t 的排序规则就把这个数据放在提示位置中成功后返回指向该数据的迭代器，如果提示位置不正确则忽略提示位置，当数据插入成功后返回数据的实际位置迭代器，如果 set_t 中包含了该数据那么插入失败，返回 set_end()。

第三个函数插入指定的数据区间。

上面的函数要求迭代器和数据区间是有效的，无效的迭代器或数据区间导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    set_t* pset_s1 = create_set(int);
    set_t* pset_s2 = create_set(int);
```

```
    set_iterator_t it_s;

    if(pset_s1 == NULL || pset_s2 == NULL)
    {
        return -1;
    }

    set_init(pset_s1);
    set_init(pset_s2);

    set_insert(pset_s1, 10);
    set_insert(pset_s1, 20);
    set_insert(pset_s1, 30);
    set_insert(pset_s1, 40);

    printf("The original s1 =");
    for(it_s = set_begin(pset_s1);
        !iterator_equal(it_s, set_end(pset_s1));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");

    it_s = set_insert(pset_s1, 10);
    if(iterator_equal(it_s, set_end(pset_s1)))
    {
        printf("The element 10 already exists in s1.\n");
    }
    else
    {
        printf("The element 10 was inserted in s1 successfully.\n");
    }

    set_insert_hint(pset_s1, iterator_prev(set_end(pset_s1)), 50);
    printf("After the insertions, s1 =");
    for(it_s = set_begin(pset_s1);
        !iterator_equal(it_s, set_end(pset_s1));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");

    set_insert(pset_s2, 100);
    set_insert_range(pset_s2, iterator_next(set_begin(pset_s1)),
        iterator_prev(set_end(pset_s1)));
    printf("s2 =");
    for(it_s = set_begin(pset_s2);
        !iterator_equal(it_s, set_end(pset_s2));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");

    set_destroy(pset_s1);
    set_destroy(pset_s2);

    return 0;
```

```
}
```

- **Output**

```
The original s1 = 10 20 30 40
The element 10 already exists in s1.
After the insertions, s1 = 10 20 30 40 50
s2 = 20 30 40 100
```

19. set_key_comp

返回 set_t 的键比较规则。

```
binary_function_t set_key_comp(
    const set_t* cpset_set
);
```

- **Parameters**

cpset_set: 指向 set_t 类型的指针。

- **Remarks**

由于 set_t 中数据本身就是键，所以这个函数的返回值与 set_value_comp()相同。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    set_t* pset_s1 = create_set(int);
    set_t* pset_s2 = create_set(int);
    binary_function_t bfun_kl = NULL;
    bool_t b_result = false;
    int n_element1 = 0;
    int n_element2 = 0;

    if(pset_s1 == NULL || pset_s2 == NULL)
    {
        return -1;
    }

    set_init(pset_s1);

    bfun_kl = set_key_comp(pset_s1);
    n_element1 = 2;
    n_element2 = 3;
    (*bfun_kl)(&n_element1, &n_element2, &b_result);
    if(b_result)
```

```
    {
        printf("(*bfun_kl)(2, 3) return value of true, "
               "where bfun_kl is the function of s1.\n");
    }
    else
    {
        printf("(*bfun_kl)(2, 3) return value of false, "
               "where bfun_kl is the function of s1.\n");
    }

    set_destroy(pset_s1);

    set_init_ex(pset_s2, fun_greater_int);

    bfun_kl = set_key_comp(pset_s2);
    (*bfun_kl)(&n_element1, &n_element2, &b_result);
    if(b_result)
    {
        printf("(*bfun_kl)(2, 3) return value of true, "
               "where bfun_kl is the function of s2.\n");
    }
    else
    {
        printf("(*bfun_kl)(2, 3) return value of false, "
               "where bfun_kl is the function of s2.\n");
    }

    set_destroy(pset_s2);

    return 0;
}
```

- **Output**

```
(*bfun_kl)(2, 3) return value of true, where bfun_kl is the function of s1.
(*bfun_kl)(2, 3) return value of false, where bfun_kl is the function of s2.
```

20. set_less

测试第一个 set_t 是否小于第二个 set_t。

```
bool_t set_less(
    const set_t* cpset_first,
    const set_t* cpset_second
);
```

- **Parameters**

cpset_first: 指向第一个 set_t 类型的指针。
cpset_second: 指向第二个 set_t 类型的指针。

- **Remarks**

这个函数要求两个 set_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    set_t* pset_s1 = create_set(int);
    set_t* pset_s2 = create_set(int);
    set_t* pset_s3 = create_set(int);
    int i = 0;

    if(pset_s1 == NULL || pset_s2 == NULL || pset_s3 == NULL)
    {
        return -1;
    }

    set_init(pset_s1);
    set_init(pset_s2);
    set_init(pset_s3);

    for(i = 0; i < 3; ++i)
    {
        set_insert(pset_s1, i);
        set_insert(pset_s2, i * i);
        set_insert(pset_s3, i - 1);
    }

    if(set_less(pset_s1, pset_s2))
    {
        printf("The set s1 is less than the set s2.\n");
    }
    else
    {
        printf("The set s1 is not less than the set s2.\n");
    }

    if(set_less(pset_s1, pset_s3))
    {
        printf("The set s1 is less than the set s3.\n");
    }
    else
    {
        printf("The set s1 is not less than the set s3.\n");
    }

    set_destroy(pset_s1);
    set_destroy(pset_s2);
    set_destroy(pset_s3);

    return 0;
}
```

- **Output**

```
The set s1 is less than the set s2.
The set s1 is not less than the set s3.
```

21. set_less_equal

测试第一个 set_t 是否小于等于第二个 set_t。

```
bool_t set_less_equal(
    const set_t* cpset_first,
    const set_t* cpset_second
);
```

- **Parameters**

cpset_first: 指向第一个 set_t 类型的指针。
cpset_second: 指向第二个 set_t 类型的指针。

- **Remarks**

这个函数要求两个 set_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    set_t* pset_s1 = create_set(int);
    set_t* pset_s2 = create_set(int);
    set_t* pset_s3 = create_set(int);
    set_t* pset_s4 = create_set(int);
    int i = 0;

    if(pset_s1 == NULL || pset_s2 == NULL || pset_s3 == NULL || pset_s4 == NULL)
    {
        return -1;
    }

    set_init(pset_s1);
    set_init(pset_s2);
    set_init(pset_s3);
    set_init(pset_s4);

    for(i = 0; i < 3; ++i)
    {
        set_insert(pset_s1, i);
        set_insert(pset_s2, i * i);
        set_insert(pset_s3, i - 1);
        set_insert(pset_s4, i);
    }

    if(set_less_equal(pset_s1, pset_s2))
    {
```

```
        printf("The set s1 is less than or equal to the set s2.\n");
    }
    else
    {
        printf("The set s1 is greater than the set s2.\n");
    }

    if(set_less_equal(pset_s1, pset_s3))
    {
        printf("The set s1 is less than or equal to the set s3.\n");
    }
    else
    {
        printf("The set s1 is greater than the set s3.\n");
    }

    if(set_less_equal(pset_s1, pset_s4))
    {
        printf("The set s1 is less than or equal to the set s4.\n");
    }
    else
    {
        printf("The set s1 is greater than the set s4.\n");
    }

    set_destroy(pset_s1);
    set_destroy(pset_s2);
    set_destroy(pset_s3);
    set_destroy(pset_s4);

    return 0;
}
```

- **Output**

```
The set s1 is less than or equal to the set s2.
The set s1 is greater than the set s3.
The set s1 is less than or equal to the set s4.
```

22. set_lower_bound

获得 set_t 中等于或者大于指定数据的第一个数据的迭代器。

```
set_iterator_t set_lower_bound(
    const set_t* cpset_set,
    element
);
```

- **Parameters**

cpset_set: 指向 set_t 类型的指针。
element: 指定的数据。

- **Remarks**

如果 set_t 中包含指定的数据则返回等于指定数据的第一个数据的迭代器，如果 set_t 中不包含指定的数据则返回大于指定数据的第一个数据的迭代器，如果指定的数据是 set_t 中最大的数据则返回值等于 set_end()。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    set_t* pset_s1 = create_set(int);
    set_iterator_t it_s;

    if(pset_s1 == NULL)
    {
        return -1;
    }

    set_init(pset_s1);

    set_insert(pset_s1, 10);
    set_insert(pset_s1, 20);
    set_insert(pset_s1, 30);

    it_s = set_lower_bound(pset_s1, 20);
    printf("The element of set s1 with a key of 20 is: %d.\n",
        *(int*)iterator_get_pointer(it_s));

    it_s = set_lower_bound(pset_s1, 40);
    /* If no match is found for the key, end() is is returend */
    if(iterator_equal(it_s, set_end(pset_s1)))
    {
        printf("The set s1 doesn't have an element with a key of 40.\n");
    }
    else
    {
        printf("The element of set s1 with a key of 40 is: %d.\n",
            *(int*)iterator_get_pointer(it_s));
    }

    /*
     * The element at a specific location in the set can be found
     * by using a dereferenced iterator that addreses the location.
     */
    it_s = set_end(pset_s1);
    it_s = iterator_prev(it_s);
    it_s = set_lower_bound(pset_s1, *(int*)iterator_get_pointer(it_s));
    printf("The element of s1 with a key matching "
           "that of the last element is: %d.\n",
           *(int*)iterator_get_pointer(it_s));

    set_destroy(pset_s1);

    return 0;
}
```

- **Output**

```
The element of set s1 with a key of 20 is: 20.
The set s1 doesn't have an element with a key of 40.
The element of s1 with a key matching that of the last element is: 30.
```

23. set_max_size

返回 set_t 中能够保存的数据个数的最大可能值。

```
size_t set_max_size(
    const set_t* cpset_set
);
```

- **Parameters**

  cpset_set: 指向 set_t 类型的指针。

- **Remarks**

  这是一个与系统有关的常数。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    set_t* pset_s1 = create_set(int);

    if(pset_s1 == NULL)
    {
        return -1;
    }

    set_init(pset_s1);

    printf("The maximum possible length of the set is %d.\n",
        set_max_size(pset_s1));

    set_destroy(pset_s1);

    return 0;
}
```

- **Output**

```
The maximum possible length of the set is 1073741823.
```

24. set_not_equal

测试两个 set_t 是否不等。

```
bool_t set_not_equal(
    const set_t* cpset_first,
    const set_t* cpset_second
);
```

- **Parameters**

  cpset_first: 指向第一个 set_t 类型的指针。
  cpset_second: 指向第二个 set_t 类型的指针。

- **Remarks**

两个 set_t 中的数据对应相等，并且数量相等，函数返回 false，否则返回 true。如果两个 set_t 中的数据类型不同也认为不等。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    set_t* pset_s1 = create_set(int);
    set_t* pset_s2 = create_set(int);
    set_t* pset_s3 = create_set(int);
    int i = 0;

    if(pset_s1 == NULL || pset_s2 == NULL || pset_s3 == NULL)
    {
        return -1;
    }

    set_init(pset_s1);
    set_init(pset_s2);
    set_init(pset_s3);

    for(i = 0; i < 3; ++i)
    {
        set_insert(pset_s1, i);
        set_insert(pset_s2, i * i);
        set_insert(pset_s3, i);
    }

    if(set_not_equal(pset_s1, pset_s2))
    {
        printf("The sets s1 and s2 are not equal.\n");
    }
    else
    {
```

```
        printf("The sets s1 and s2 are equal.\n");
    }

    if(set_not_equal(pset_s1, pset_s3))
    {
        printf("The sets s1 and s3 are not equal.\n");
    }
    else
    {
        printf("The sets s1 and s3 are equal.\n");
    }

    set_destroy(pset_s1);
    set_destroy(pset_s2);
    set_destroy(pset_s3);

    return 0;
}
```

- **Output**

```
The sets s1 and s2 are not equal.
The sets s1 and s3 are equal.
```

25. set_size

返回 set_t 中保存的数据的数量。

```
size_t set_size(
    const set_t* cpset_set
);
```

- **Parameters**

  cpset_set: 指向 set_t 类型的指针。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    set_t* pset_s1 = create_set(int);

    if(pset_s1 == NULL)
    {
        return -1;
    }

    set_init(pset_s1);
```

```
    set_insert(pset_s1, 1);
    printf("The set length is %d.\n", set_size(pset_s1));

    set_insert(pset_s1, 2);
    printf("The set length is now %d.\n", set_size(pset_s1));

    set_destroy(pset_s1);

    return 0;
}
```

- **Output**

```
The set length is 1.
The set length is now 2.
```

26. set_swap

交换两个 set_t 中的内容。

```
void set_swap(
    set_t* pset_first,
    set_t* pset_second
);
```

- **Parameters**

pset_first: 指向第一个 set_t 类型的指针。
pset_second: 指向第二个 set_t 类型的指针。

- **Remarks**

这个函数要求两个 set_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    set_t* pset_s1 = create_set(int);
    set_t* pset_s2 = create_set(int);
    set_iterator_t it_s;

    if(pset_s1 == NULL || pset_s2 == NULL)
    {
        return -1;
    }

    set_init(pset_s1);
    set_init(pset_s2);
```

```
    set_insert(pset_s1, 10);
    set_insert(pset_s1, 20);
    set_insert(pset_s1, 30);
    set_insert(pset_s2, 100);
    set_insert(pset_s2, 200);

    printf("The original set s1 is:");
    for(it_s = set_begin(pset_s1);
        !iterator_equal(it_s, set_end(pset_s1));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");

    set_swap(pset_s1, pset_s2);
    printf("After swapping with s2, set s1 is:");
    for(it_s = set_begin(pset_s1);
        !iterator_equal(it_s, set_end(pset_s1));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");

    set_destroy(pset_s1);
    set_destroy(pset_s2);

    return 0;
}
```

- **Output**

```
The original set s1 is: 10 20 30
After swapping with s2, set s1 is: 100 200
```

27. set_upper_bound

返回 set_t 中大于指定数据的第一个数据的迭代器。

```
set_iterator_t set_upper_bound(
    const set_t* cpset_set,
    element
);
```

- **Parameters**

cpset_set: 指向 set_t 类型的指针。
element: 指定的数据。

- **Remarks**

如果指定的数据是 set_t 中最大的数据则返回值等于 set_end()。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    set_t* pset_s1 = create_set(int);
    set_iterator_t it_s;

    if(pset_s1 == NULL)
    {
        return -1;
    }

    set_init(pset_s1);

    set_insert(pset_s1, 10);
    set_insert(pset_s1, 20);
    set_insert(pset_s1, 30);

    it_s = set_upper_bound(pset_s1, 20);
    printf("The first element of set s1 with a key greater than 20 is: %d.\n",
        *(int*)iterator_get_pointer(it_s));

    it_s = set_upper_bound(pset_s1, 30);
    /* If no match is found for the key, end() is returned */
    if(iterator_equal(it_s, set_end(pset_s1)))
    {
        printf("The set s1 doesn't have an element with a key greater than 30.\n");
    }
    else
    {
        printf("the element of set s1 with a key > 30 is: %d.\n",
            *(int*)iterator_get_pointer(it_s));
    }

    /*
     * The element at a specific location in the set can be found
     * by using a dereferenced iterator addressing the location.
     */
    it_s = set_begin(pset_s1);
    it_s = set_upper_bound(pset_s1, *(int*)iterator_get_pointer(it_s));
    printf("The first element of s1 with a key greater than that "
           "of the initial element of s1 is: %d.\n",
           *(int*)iterator_get_pointer(it_s));

    set_destroy(pset_s1);

    return 0;
}
```

- **Output**

```
The first element of set s1 with a key greater than 20 is: 30.
The set s1 doesn't have an element with a key greater than 30.
The first element of s1 with a key greater than that of the initial element of s1
```

```
is: 20.
```

28. set_value_comp

返回 set_t 中数据的比较规则。

```
binary_function_t set_value_comp(
    const set_t* cpset_set
);
```

- **Parameters**

cpset_set: 指向 set_t 类型的指针。

- **Remarks**

由于 set_t 中数据本身就是键，所以这个函数的返回值与 set_key_comp()相同。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    set_t* pset_s1 = create_set(int);
    set_t* pset_s2 = create_set(int);
    binary_function_t bfun_vl = NULL;
    int n_element1 = 0;
    int n_element2 = 0;
    bool_t b_result = false;

    if(pset_s1 == NULL || pset_s2 == NULL)
    {
        return -1;
    }

    set_init(pset_s1);

    bfun_vl = set_value_comp(pset_s1);
    n_element1 = 2;
    n_element2 = 3;
    (*bfun_vl)(&n_element1, &n_element2, &b_result);
    if(b_result)
    {
        printf("(*bfun_vl)(2, 3) returns value of true, "
               "where bfun_vl is the function of s1.\n");
    }
    else
    {
        printf("(*bfun_vl)(2, 3) returns value of false, "
```

```
                "where bfun_vl is the function of s1.\n");
    }

    set_destroy(pset_s1);

    set_init_ex(pset_s2, fun_greater_int);

    bfun_vl = set_value_comp(pset_s2);
    (*bfun_vl)(&n_element1, &n_element2, &b_result);
    if(b_result)
    {
        printf("(*bfun_vl)(2, 3) returns value of true, "
               "where bfun_vl is the function of s2.\n");
    }
    else
    {
        printf("(*bfun_vl)(2, 3) returns value of false, "
               "where bfun_vl is the function of s2.\n");
    }

    set_destroy(pset_s2);

    return 0;
}
```

- **Output**

```
(*bfun_vl)(2, 3) returns value of true, where bfun_vl is the function of s1.
(*bfun_vl)(2, 3) returns value of false, where bfun_vl is the function of s2.
```

第六节　多重集合 multiset_t

多重集合容器 multiset_t 是关联容器，multiset_t 中的数据是按照键和指定的规则自动排序但它允许多个相同的键存在，multiset_t 中的键就是数据本身。multiset_t 中的数据不可以直接或者通过迭代器修改，因为这样会破坏 multiset_t 中数据的有序性，要想修改一个数据只有先删除它然后插入新的数据。multiset_t 支持双向迭代器。插入新数据是不会破坏原有的迭代器，删除数据是只有指向被删除的数据的迭代器失效。multiset_t 对于数据的查找，插入和删除都是高效的。multiset_t 中的数据根据指定的规则自动排序，默认的排序规则是使用数据的小于操作符，用户可以在初始化时指定自定义的排序规则。

- **Typedefs**

multiset_t	多重集合容器类型。
multiset_iterator_t	多重集合容器迭代器类型。

- **Operation Functions**

create_multiset	创建多重集合容器类型。
multiset_assign	为多重集合容器赋值。
multiset_begin	返回指向多重集合容器中第一个数据的迭代器。
multiset_clear	删除多重集合中的所有数据。
multiset_count	返回多重集合容器中包含指定数据的个数。
multiset_destroy	销毁多重集合容器。

multiset_empty	测试多重集合容器是否为空。
multiset_end	返回指向多重集合容器末尾的迭代器。
multiset_equal	测试两个多重集合容器是否相等。
multiset_equal_range	获得多重集合容器中包含指定数据的数据区间。
multiset_erase	删除指定数据。
multiset_erase_pos	删除指定位置的数据。
multiset_erase_range	删除指定数据区间的数据。
multiset_find	在多重集合容器中查找指定的数据。
multiset_greater	测试第一个多重集合容器是否大于第二个多重集合容器。
multiset_greater_equal	测试第一个多重集合容器是否大于等于第二个多重集合容器。
multiset_init	初始化一个空的多重集合容器。
multiset_init_copy	使用一个已经存在的多重集合容器来初始化当前的多重集合容器。
multiset_init_copy_range	使用指定区间中的数据初始化多重集合容器。
multiset_init_copy_range_ex	使用指定的数据区间和指定的排序规则初始化多重集合容器。
multiset_init_ex	使用指定的排序规则初始化一个空的多重集合容器。
multiset_insert	向多重集合容器中插入一个指定的数据。
multiset_insert_hint	向多重集合容器中插入一个指定的数据，并给出位置提示。
multiset_insert_range	向多重集合容器中插入一个指定的数据区间。
multiset_key_comp	返回多重集合容器使用的键比较规则。
multiset_less	测试第一个多重集合容器是否小于第二个多重集合容器。
multiset_less_equal	测试第一个多重集合容器是否小于等于第二个多重集合容器。
multiset_lower_bound	返回多重集合容器中等于指定数据的第一个数据的迭代器。
multiset_max_size	返回多重集合容器能够保存的数据数量的最大可能值。
multiset_not_equal	测试两个多重集合容器是否不等。
multiset_size	返回多重集合容器中数据的数量。
multiset_swap	交换两个多重集合容器的内容。
multiset_upper_bound	返回多重集合容器中大于指定数据的第一个数据的迭代器。
multiset_value_comp	返回多重集合容器使用的数据比较规则。

1. multiset_t

多重集合容器类型。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

请参考 multiset_t 类型的其他操作函数。

2. multiset_iterator_t

多重集合容器类型的迭代器类型。

- **Remarks**

multiset_iterator_t 是双向迭代器类型，不能通过迭代器来修改容器中的数据。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

请参考 multiset_t 类型的其他操作函数。

3. create_multiset

创建 multiset_t 类型。

```
multiset_t* create_multiset(
    type
);
```

- **Parameters**

type: 数据类型描述。

- **Remarks**

函数成功返回指向 multiset_t 类型的指针，失败返回 NULL。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

请参考 multiset_t 类型的其他操作函数。

4. multiset_assign

为 multiset_t 赋值。

```
void multiset_assign(
    multiset_t* pmsett_dest,
    const multiset_t* cpmsett_src
);
```

- **Parameters**

pmset_dest: 指向被赋值的 multiset_t 类型的指针。
cpmset_src: 指向赋值的 multiset_t 类型的指针。

- **Remarks**

要求两个 multiset_t 类型保存的数据具有相同的类型，否则函数的行为未定义。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s1 = create_multiset(int);
    multiset_t* pmset_s2 = create_multiset(int);
    multiset_iterator_t it_s;

    if(pmset_s1 == NULL || pmset_s2 == NULL)
    {
        return -1;
    }

    multiset_init(pmset_s1);
    multiset_init(pmset_s2);

    multiset_insert(pmset_s1, 10);
    multiset_insert(pmset_s1, 20);
    multiset_insert(pmset_s1, 30);
    multiset_insert(pmset_s2, 40);
    multiset_insert(pmset_s2, 50);
    multiset_insert(pmset_s2, 60);

    printf("s1 =");
    for(it_s = multiset_begin(pmset_s1);
        !iterator_equal(it_s, multiset_end(pmset_s1));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");

    multiset_assign(pmset_s1, pmset_s2);
    printf("s1 =");
    for(it_s = multiset_begin(pmset_s1);
        !iterator_equal(it_s, multiset_end(pmset_s1));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");

    multiset_destroy(pmset_s1);
    multiset_destroy(pmset_s2);

    return 0;
}
```

- **Output**

```
s1 = 10 20 30
s1 = 40 50 60
```

5. multiset_begin

返回指向 multiset_t 中第一个数据迭代器。

```
multiset_iterator_t multiset_begin(
    const multiset_t* cpmset_multiset
);
```

- **Parameters**

  cpmset_multiset: 指向 multiset_t 类型的指针。

- **Remarks**

  如果 multiset_t 为空，这个函数的返回值和 multiset_end()的返回值相等。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s1 = create_multiset(int);

    if(pmset_s1 == NULL)
    {
        return -1;
    }

    multiset_init(pmset_s1);

    multiset_insert(pmset_s1, 1);
    multiset_insert(pmset_s1, 2);
    multiset_insert(pmset_s1, 3);

    printf("The first element of s1 is %d\n",
        *(int*)iterator_get_pointer(multiset_begin(pmset_s1)));

    multiset_erase_pos(pmset_s1, multiset_begin(pmset_s1));
    printf("The first element of s1 is now %d\n",
        *(int*)iterator_get_pointer(multiset_begin(pmset_s1)));

    multiset_destroy(pmset_s1);

    return 0;
}
```

- **Output**

```
The first element of s1 is 1
The first element of s1 is now 2
```

6. multiset_clear

删除 multiset_t 中的所有数据。

```
void multiset_clear(
    multiset_t* pmset_multiset
);
```

- **Parameters**

  pmset_multiset: 指向 multiset_t 类型的指针。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s1 = create_multiset(int);

    if(pmset_s1 == NULL)
    {
        return -1;
    }

    multiset_init(pmset_s1);

    multiset_insert(pmset_s1, 1);
    multiset_insert(pmset_s1, 2);

    printf("The size of the multiset is initially %d.\n",
        multiset_size(pmset_s1));

    multiset_clear(pmset_s1);
    printf("The size of the multiset after clearing is %d.\n",
        multiset_size(pmset_s1));

    multiset_destroy(pmset_s1);

    return 0;
}
```

- **Output**

```
The size of the multiset is initially 2.
The size of the multiset after clearing is 0.
```

7. multiset_count

返回 multiset_t 中指定数据的个数。

```
size_t multiset_count(
    const multiset_t* cpmset_multiset,
    element
);
```

- **Parameters**

  cpmset_multiset: 指向 multiset_t 类型的指针。
  element: 指定的数据。

- **Remarks**

  如果容器中不包含指定数据则返回 0，包含则返回指定数据的个数。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s1 = create_multiset(int);

    if(pmset_s1 == NULL)
    {
        return -1;
    }

    multiset_init(pmset_s1);

    multiset_insert(pmset_s1, 1);
    multiset_insert(pmset_s1, 1);
    multiset_insert(pmset_s1, 2);

    /*
     * Element do not need to be unique in multiset,
     * so duplicates are allowed and counted.
     */
    printf("The number of element in s1 with a sort key of 1 is: %d.\n",
        multiset_count(pmset_s1, 1));
    printf("The number of element in s1 with a sort key of 2 is: %d.\n",
        multiset_count(pmset_s1, 2));
    printf("The number of element in s1 with a sort key of 3 is: %d.\n",
        multiset_count(pmset_s1, 3));

    multiset_destroy(pmset_s1);

    return 0;
}
```

- **Output**

```
The number of element in s1 with a sort key of 1 is: 2.
```

```
The number of element in s1 with a sort key of 2 is: 1.
The number of element in s1 with a sort key of 3 is: 0.
```

8. multiset_destroy

销毁 multiset_t 容器。

```
void multiset_destroy(
    multiset_t* pmset_multiset
);
```

- **Parameters**

  pmset_multiset: 指向 multiset_t 类型的指针。

- **Remarks**

  multiset_t 容器使用之后要销毁，否则 multiset_t 占用的资源不会被释放。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

  请参考 multiset_t 类型的其他操作函数。

9. multiset_empty

测试 multiset_t 是否为空。

```
bool_t multiset_empty(
    const multiset_t* cpmset_multiset
);
```

- **Parameters**

  cpmset_multiset: 指向 multiset_t 类型的指针。

- **Remarks**

  multiset_t 容器为空则返回 true，否则返回 false。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s1 = create_multiset(int);
    multiset_t* pmset_s2 = create_multiset(int);
```

```
    if(pmset_s1 == NULL || pmset_s2 == NULL)
    {
        return -1;
    }

    multiset_init(pmset_s1);
    multiset_init(pmset_s2);

    multiset_insert(pmset_s1, 1);

    if(multiset_empty(pmset_s1))
    {
        printf("The multiset s1 is empty.\n");
    }
    else
    {
        printf("The multiset s1 is not empty.\n");
    }

    if(multiset_empty(pmset_s2))
    {
        printf("The multiset s2 is empty.\n");
    }
    else
    {
        printf("The multiset s2 is not empty.\n");
    }

    multiset_destroy(pmset_s1);
    multiset_destroy(pmset_s2);

    return 0;
}
```

- **Output**

```
The multiset s1 is not empty.
The multiset s2 is empty.
```

10. multiset_end

返回 multiset_t 的末尾位置的迭代器。

```
multiset_iterator_t multiset_end(
    const multiset_t* cpmset_multiset
);
```

- **Parameters**

cpmset_multiset: 指向 multiset_t 类型的指针。

- **Remarks**

如果 multiset_t 为空，这个函数的返回值和 multiset_begin()的返回值相等。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s1 = create_multiset(int);
    multiset_iterator_t it_s;

    if(pmset_s1 == NULL)
    {
        return -1;
    }

    multiset_init(pmset_s1);

    multiset_insert(pmset_s1, 1);
    multiset_insert(pmset_s1, 2);
    multiset_insert(pmset_s1, 3);

    it_s = iterator_prev(multiset_end(pmset_s1));
    printf("The last element of s1 is %d\n",
        *(int*)iterator_get_pointer(it_s));

    multiset_erase_pos(pmset_s1, it_s);

    it_s = iterator_prev(multiset_end(pmset_s1));
    printf("The last element of s1 is now %d\n",
        *(int*)iterator_get_pointer(it_s));

    multiset_destroy(pmset_s1);

    return 0;
}
```

- **Output**

```
The last element of s1 is 3
The last element of s1 is now 2
```

11. multiset_equal

测试两个 multiset_t 是否相等。

```
bool_t multiset_equal(
    const multiset_t* cpmset_first,
    const multiset_t* cpmset_second
);
```

- **Parameters**

  cpmset_first: 指向第一个 multiset_t 类型的指针。
  cpmset_second: 指向第二个 multiset_t 类型的指针。

- **Remarks**

两个 multiset_t 中的数据对应相等，并且数量相等，函数返回 true，否则返回 false。如果两个 multiset_t 中的数据类型不同也认为不等。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s1 = create_multiset(int);
    multiset_t* pmset_s2 = create_multiset(int);
    multiset_t* pmset_s3 = create_multiset(int);
    int i = 0;

    if(pmset_s1 == NULL || pmset_s2 == NULL || pmset_s3 == NULL)
    {
        return -1;
    }

    multiset_init(pmset_s1);
    multiset_init(pmset_s2);
    multiset_init(pmset_s3);

    for(i = 0; i < 3; ++i)
    {
        multiset_insert(pmset_s1, i);
        multiset_insert(pmset_s2, i * i);
        multiset_insert(pmset_s3, i);
    }

    if(multiset_equal(pmset_s1, pmset_s2))
    {
        printf("The multisets s1 and s2 are equal.\n");
    }
    else
    {
        printf("The multisets s1 and s2 are not equal.\n");
    }

    if(multiset_equal(pmset_s1, pmset_s3))
    {
        printf("The multisets s1 and s3 are equal.\n");
    }
    else
    {
        printf("The multisets s1 and s3 are not equal.\n");
    }

    multiset_destroy(pmset_s1);
    multiset_destroy(pmset_s2);
    multiset_destroy(pmset_s3);
```

```
    return 0;
}
```

● **Output**

```
The multisets s1 and s2 are not equal.
The multisets s1 and s3 are equal.
```

12. multiset_equal_range

返回 multiset_t 中包含指定数据的数据区间。

```
range_t multiset_equal_range(
    const multiset_t* cpmset_multiset,
    element
);
```

● **Parameters**

cpmset_multiset: 指向 multiset_t 类型的指针。
element: 指定的数据。

● **Remarks**

返回 multiset_t 中包含指定数据的数据区间[range_t.it_begin, range_t.it_end)，其中 it_begin 是指向等于指定数据的第一个数据的迭代器，it_end 指向的是大于指定数据的第一个数据的迭代器。如果 multiset_t 中不包含指定数据则 it_begin 与 it_end 相等。如果指定的数据是 multiset_t 中最大的数据则 it_end 等于 multiset_end()。

● **Requirements**

头文件 <cstl/cset.h>

● **Example**

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

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s1 = create_multiset(int);
    range_t r_s;
    multiset_iterator_t it_s;

    if(pmset_s1 == NULL)
    {
        return -1;
    }

    multiset_init(pmset_s1);

    multiset_insert(pmset_s1, 10);
    multiset_insert(pmset_s1, 20);
    multiset_insert(pmset_s1, 30);
```

```
    r_s = multiset_equal_range(pmset_s1, 20);

    printf("The upper bound of the element with a "
           "key of 20 in the multiset s1 is: %d.\n",
           *(int*)iterator_get_pointer(r_s.it_end));
    printf("The lower bound of the element with a "
           "key of 20 in the multiset s1 is: %d.\n",
           *(int*)iterator_get_pointer(r_s.it_begin));

    /* Compare the upper_bound called directly */
    it_s = multiset_upper_bound(pmset_s1, 20);
    printf("A direct call of upper_bound(20) gives %d, matching the 2nd "
           "element of the range returned by equal_range(20).\n",
           *(int*)iterator_get_pointer(it_s));

    r_s = multiset_equal_range(pmset_s1, 40);
    /* If no match is found for the key, both elements of the range return end(). */
    if(iterator_equal(r_s.it_begin, multiset_end(pmset_s1)) &&
       iterator_equal(r_s.it_end, multiset_end(pmset_s1)))
    {
        printf("The multiset s1 doesn't have an "
               "element with a key less than 40.\n");
    }
    else
    {
        printf("The element of multiset s1 with a key >= 40 is: %d.\n",
            *(int*)iterator_get_pointer(r_s.it_begin));
    }

    multiset_destroy(pmset_s1);

    return 0;
}
```

- **Output**

```
The upper bound of the element with a key of 20 in the multiset s1 is: 30.
The lower bound of the element with a key of 20 in the multiset s1 is: 20.
A direct call of upper_bound(20) gives 30, matching the 2nd element of the range
returned by equal_range(20).
The multiset s1 doesn't have an element with a key less than 40.
```

13. multiset_erase multiset_erase_pos multiset_erase_range

删除 multiset_t 中的数据。

```
size_t multiset_erase(
    multiset_t* pmset_multiset,
    element
);

void multiset_erase_pos(
    multiset_t* pmset_multiset,
    multiset_iterator_t it_pos
);

void multiset_erase_range(
    multiset_t* pmset_multiset,
```

```
    multiset_iterator_t it_begin,
    multiset_iterator_t it_end
);
```

- **Parameters**

  pmset_multiset: 指向 multiset_t 类型的指针。
  element: 要删除的数据。
  it_pos: 要删除的数据的位置迭代器。
  it_begin: 要删除的数据区间的开始位置。
  it_end: 要删除的数据区间的末尾位置。

- **Remarks**

  第一个函数删除 multiset_t 中指定的数据，并返回删除的个数，如果 multiset_t 中不包含指定的数据就返回 0。
  第二个函数删除指定位置的数据。
  第三个函数删除指定数据区间中的数据。
  后面两个函数要求迭代器和数据区间是有效的，无效的迭代器或数据区间导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s1 = create_multiset(int);
    multiset_t* pmset_s2 = create_multiset(int);
    multiset_t* pmset_s3 = create_multiset(int);
    multiset_iterator_t it_s;
    int i = 0;
    int n_count = 0;

    if(pmset_s1 == NULL || pmset_s2 == NULL || pmset_s3 == NULL)
    {
        return -1;
    }

    multiset_init(pmset_s1);
    multiset_init(pmset_s2);
    multiset_init(pmset_s3);

    for(i = 1; i < 5; ++i)
    {
        multiset_insert(pmset_s1, i);
        multiset_insert(pmset_s2, i * i);
        multiset_insert(pmset_s3, i - 1);
    }

    /* The first function removes an element at a given position */
    multiset_erase_pos(pmset_s1, iterator_next(multiset_begin(pmset_s1)));
    printf("After the second element is deleted, the multiset s1 is:");
```

```
    for(it_s = multiset_begin(pmset_s1);
        !iterator_equal(it_s, multiset_end(pmset_s1));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");

    /* The second function remove elements in the range[first, last) */
    multiset_erase_range(pmset_s2, iterator_next(multiset_begin(pmset_s2)),
        iterator_prev(multiset_end(pmset_s2)));
    printf("After the middle two elements are deleted, the multiset s2 is:");
    for(it_s = multiset_begin(pmset_s2);
        !iterator_equal(it_s, multiset_end(pmset_s2));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");

    /* The third function removes elements with a given key */
    multiset_insert(pmset_s3, 2);
    n_count = multiset_erase(pmset_s3, 2);
    printf("The number of elements removed from s3 is: %d.\n", n_count);
    printf("After the element with a key of 2 is deleted, the multiset s3 is:");
    for(it_s = multiset_begin(pmset_s3);
        !iterator_equal(it_s, multiset_end(pmset_s3));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");

    multiset_destroy(pmset_s1);
    multiset_destroy(pmset_s2);
    multiset_destroy(pmset_s3);

    return 0;
}
```

- **Output**

```
After the second element is deleted, the multiset s1 is: 1 3 4
After the middle two elements are deleted, the multiset s2 is: 1 16
The number of elements removed from s3 is: 2.
After the element with a key of 2 is deleted, the multiset s3 is: 0 1 3
```

14. multiset_find

在 multiset_t 中查找指定数据。

```
multiset_iterator_t multiset_find(
    const multiset_t* cpmset_multiset,
    element
);
```

- **Parameters**

cpmset_multiset: 指向 multiset_t 类型的指针。

element: 指定的数据。

- **Remarks**

如果 multiset_t 中包含指定的数据则返回指向该数据的迭代器，否则返回 multiset_end()。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s1 = create_multiset(int);
    multiset_iterator_t it_s;

    if(pmset_s1 == NULL)
    {
        return -1;
    }

    multiset_init(pmset_s1);

    multiset_insert(pmset_s1, 10);
    multiset_insert(pmset_s1, 20);
    multiset_insert(pmset_s1, 20);

    it_s = multiset_find(pmset_s1, 20);
    printf("The first element of multiset s1 with a key of 20 is: %d.\n",
        *(int*)iterator_get_pointer(it_s));

    it_s = multiset_find(pmset_s1, 40);
    /* If no match is found for the key, end() is returned. */
    if(iterator_equal(it_s, multiset_end(pmset_s1)))
    {
        printf("The multiset s1 doesn't have an element with a key of 40.\n");
    }
    else
    {
        printf("The element of multiset s1 with a key of 40 is: %d.\n",
            *(int*)iterator_get_pointer(it_s));
    }

    /*
     * The element at a specific location in the multiset can be
     * found using a dereferenced iterator addressing the location.
     */
    it_s = multiset_end(pmset_s1);
    it_s = iterator_prev(it_s);
    it_s = multiset_find(pmset_s1, *(int*)iterator_get_pointer(it_s));
    printf("The first element of s1 with a key matching that of the "
           "last element is %d.\n", *(int*)iterator_get_pointer(it_s));
```

```
    /*
     * Note that the first element with a key equal to tha key of
     * the last element is not the last element.
     */
    if(iterator_equal(it_s, iterator_prev(multiset_end(pmset_s1))))
    {
        printf("This is the last element of multiset s1.\n");
    }
    else
    {
        printf("The is not the last element of multiset s1.\n");
    }

    multiset_destroy(pmset_s1);

    return 0;
}
```

- **Output**

```
The first element of multiset s1 with a key of 20 is: 20.
The multiset s1 doesn't have an element with a key of 40.
The first element of s1 with a key matching that of the last element is 20.
The is not the last element of multiset s1.
```

15. multiset_greater

测试第一个 multiset_t 是否大于第二个 multiset_t。

```
bool_t multiset_greater(
    const multiset_t* cpmset_first,
    const multiset_t* cpmset_second
);
```

- **Parameters**

cpmset_first: 指向第一个 multiset_t 类型的指针。
cpmset_second: 指向第二个 multiset_t 类型的指针。

- **Remarks**

这个函数要求两个 multiset_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s1 = create_multiset(int);
    multiset_t* pmset_s2 = create_multiset(int);
```

```
    multiset_t* pmset_s3 = create_multiset(int);
    int i = 0;

    if(pmset_s1 == NULL || pmset_s2 == NULL || pmset_s3 == NULL)
    {
        return -1;
    }

    multiset_init(pmset_s1);
    multiset_init(pmset_s2);
    multiset_init(pmset_s3);

    for(i = 0; i < 3; ++i)
    {
        multiset_insert(pmset_s1, i);
        multiset_insert(pmset_s2, i * i);
        multiset_insert(pmset_s3, i - 1);
    }

    if(multiset_greater(pmset_s1, pmset_s2))
    {
        printf("The multiset s1 is greater than the multiset s2.\n");
    }
    else
    {
        printf("The multiset s1 is not greater than the multiset s2.\n");
    }

    if(multiset_greater(pmset_s1, pmset_s3))
    {
        printf("The multiset s1 is greater than the multiset s3.\n");
    }
    else
    {
        printf("The multiset s1 is not greater than the multisets s3.\n");
    }

    multiset_destroy(pmset_s1);
    multiset_destroy(pmset_s2);
    multiset_destroy(pmset_s3);

    return 0;
}
```

- **Output**

```
The multiset s1 is not greater than the multiset s2.
The multiset s1 is greater than the multiset s3.
```

16. multiset_greater_equal

测试第一个 multiset_t 是否大于等于第二个 multiset_t。

```
bool_t multiset_greater_equal(
    const multiset_t* cpmset_first,
    const multiset_t* cpmset_second
);
```

- **Parameters**

cpmset_first:　指向第一个 multiset_t 类型的指针。
cpmset_second: 指向第二个 multiset_t 类型的指针。

- **Remarks**

这个函数要求两个 multiset_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s1 = create_multiset(int);
    multiset_t* pmset_s2 = create_multiset(int);
    multiset_t* pmset_s3 = create_multiset(int);
    multiset_t* pmset_s4 = create_multiset(int);
    int i = 0;

    if(pmset_s1 == NULL || pmset_s2 == NULL ||
       pmset_s3 == NULL || pmset_s4 == NULL)
    {
        return -1;
    }

    multiset_init(pmset_s1);
    multiset_init(pmset_s2);
    multiset_init(pmset_s3);
    multiset_init(pmset_s4);

    for(i = 0; i < 3; ++i)
    {
        multiset_insert(pmset_s1, i);
        multiset_insert(pmset_s2, i * i);
        multiset_insert(pmset_s3, i - 1);
        multiset_insert(pmset_s4, i);
    }

    if(multiset_greater_equal(pmset_s1, pmset_s2))
    {
        printf("The multiset s1 is greater than or equal to the multiset s2.\n");
    }
    else
    {
        printf("The multiset s1 is less than the multiset s2.\n");
    }

    if(multiset_greater_equal(pmset_s1, pmset_s3))
    {
        printf("The multiset s1 is greater than or equal to the multiset s3.\n");
    }
    else
```

```
    {
        printf("The multiset s1 is less than the multiset s3.\n");
    }

    if(multiset_greater_equal(pmset_s1, pmset_s4))
    {
        printf("The multiset s1 is greater than or equal to the multiset s4.\n");
    }
    else
    {
        printf("The multiset s1 is less than the multiset s4.\n");
    }

    multiset_destroy(pmset_s1);
    multiset_destroy(pmset_s2);
    multiset_destroy(pmset_s3);
    multiset_destroy(pmset_s4);

    return 0;
}
```

- **Output**

```
The multiset s1 is less than the multiset s2.
The multiset s1 is greater than or equal to the multiset s3.
The multiset s1 is greater than or equal to the multiset s4.
```

17. multiset_init multiset_init_copy multiset_init_copy_range multiset_init_copy_range_ex multiset_init_ex

初始化 multiset_t。

```
void multiset_init(
    multiset_t* pmset_multiset
);

void multiset_init_copy(
    multiset_t* pmset_multiset,
    const multiset_t* cpmset_src
);

void multiset_init_copy_range(
    multiset_t* pmset_multiset,
    multiset_iterator_t it_begin,
    multiset_iterator_t it_end
);

void multiset_init_copy_range_ex(
    multiset_t* pmset_multiset,
    multiset_iterator_t it_begin,
    multiset_iterator_t it_end,
    binary_function_t bfun_compare
);

void multiset_init_ex(
    multiset_t* pmset_multiset,
    binary_function_t bfun_compare
```

```
);
```

- **Parameters**

  pmset_multiset: 指向被初始化 multiset_t 类型的指针。
  cpmset_src: 指向用于初始化的 multiset_t 类型的指针。
  it_begin: 于初始化的数据区间的开始位置。
  it_end: 于初始化的数据区间的末尾位置。
  bfun_compare: 自定义排序规则。

- **Remarks**

  第一个函数初始化一个空的 multiset_t，使用与数据类型相关的小于操作函数作为默认的排序规则。
  第二个函数使用一个源 multiset_t 来初始化 multiset_t，数据的内容和排序规则都从源 multiset_t 复制。
  第三个函数使用指定的数据区间初始化一个 multiset_t，使用与数据类型相关的小于操作函数作为默认的排序规则。
  第四个函数使用指定的数据区间初始化一个 multiset_t，使用用户指定的排序规则。
  第五个函数初始化一个空的 multiset_t，使用用户指定的排序规则。
  上面的函数要求迭代器和数据区间是有效的，无效的迭代器或数据区间导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s0 = create_multiset(int);
    multiset_t* pmset_s1 = create_multiset(int);
    multiset_t* pmset_s2 = create_multiset(int);
    multiset_t* pmset_s3 = create_multiset(int);
    multiset_t* pmset_s4 = create_multiset(int);
    multiset_t* pmset_s5 = create_multiset(int);
    multiset_iterator_t it_s;

    if(pmset_s0 == NULL || pmset_s1 == NULL || pmset_s2 == NULL ||
       pmset_s3 == NULL || pmset_s4 == NULL || pmset_s5 == NULL)
    {
        return -1;
    }

    /* Create an empty multiset s0 of key type integer */
    multiset_init(pmset_s0);

    /*
     * Create an empty multiset s1 with the key comparison
     * function of less than, then insert 4 elements
     */
    multiset_init_ex(pmset_s1, fun_less_int);
    multiset_insert(pmset_s1, 10);
    multiset_insert(pmset_s1, 20);
```

```
    multiset_insert(pmset_s1, 20);
    multiset_insert(pmset_s1, 40);

    /*
     * Create an empty multiset s2 with the key comparison
     * function of greater than, then insert 2 elements.
     */
    multiset_init_ex(pmset_s2, fun_greater_int);
    multiset_insert(pmset_s2, 10);
    multiset_insert(pmset_s2, 20);

    /* Create a copy, multiset s3, of multiset s1 */
    multiset_init_copy(pmset_s3, pmset_s1);

    /* Create a multiset s4 by copy the range s1[first, last) */
    multiset_init_copy_range(pmset_s4, multiset_begin(pmset_s1),
        iterator_advance(multiset_begin(pmset_s1), 2));

    /*
     * Create a multiset s5 by copying the range s3[first, last)
     * and with the key comparison function of less than.
     */
    multiset_init_copy_range_ex(pmset_s5, multiset_begin(pmset_s3),
        iterator_next(multiset_begin(pmset_s3)), fun_less_int);

    printf("s1 =");
    for(it_s = multiset_begin(pmset_s1);
        !iterator_equal(it_s, multiset_end(pmset_s1));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");

    printf("s2 =");
    for(it_s = multiset_begin(pmset_s2);
        !iterator_equal(it_s, multiset_end(pmset_s2));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");

    printf("s3 =");
    for(it_s = multiset_begin(pmset_s3);
        !iterator_equal(it_s, multiset_end(pmset_s3));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");

    printf("s4 =");
    for(it_s = multiset_begin(pmset_s4);
        !iterator_equal(it_s, multiset_end(pmset_s4));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");
```

```
    printf("s5 =");
    for(it_s = multiset_begin(pmset_s5);
        !iterator_equal(it_s, multiset_end(pmset_s5));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");

    multiset_destroy(pmset_s0);
    multiset_destroy(pmset_s1);
    multiset_destroy(pmset_s2);
    multiset_destroy(pmset_s3);
    multiset_destroy(pmset_s4);
    multiset_destroy(pmset_s5);

    return 0;
}
```

- **Output**

```
s1 = 10 20 20 40
s2 = 20 10
s3 = 10 20 20 40
s4 = 10 20
s5 = 10
```

18. multiset_insert multiset_insert_hint multiset_insert_range

向 multiset_t 中插入数据。

```
multiset_iterator_t multiset_insert(
    multiset_t* pmset_multiset,
    element
);

multiset_iterator_t multiset_insert_hint(
    multiset_t* pmset_multiset,
    multiset_iterator_t it_hint,
    element
);

void multiset_insert_range(
    multiset_t* pmset_multiset,
    multiset_iterator_t it_begin,
    multiset_iterator_t it_end
);
```

- **Parameters**

pmset_multiset: 指向 multiset_t 类型的指针。
element: 插入的数据。
it_hint: 被插入数据的提示位置。
it_begin: 被插入的数据区间的开始位置。
it_end: 被插入的数据区间的末尾位置。

- **Remarks**

第一个函数向 multiset_t 中插入一个指定的数据，成功后返回指向该数据的迭代器，如果插入失败，返回 multiset_end()。

第二个函数向 multiset_t 中插入一个指定的数据，同时给出一个该数据被插入后的提示位置迭代器，如果这个位置符合 multiset_t 的排序规则就把这个数据放在提示位置中成功后返回指向该数据的迭代器，如果提示位置不正确则忽略提示位置，当数据插入成功后返回数据的实际位置迭代器，如果插入失败，返回 multiset_end()。

第三个函数插入指定的数据区间。

上面的函数要求迭代器和数据区间是有效的，无效的迭代器或数据区间导致函数的行为未定义。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s1 = create_multiset(int);
    multiset_t* pmset_s2 = create_multiset(int);
    multiset_iterator_t it_s;

    if(pmset_s1 == NULL || pmset_s2 == NULL)
    {
        return -1;
    }

    multiset_init(pmset_s1);
    multiset_init(pmset_s2);

    multiset_insert(pmset_s1, 10);
    multiset_insert(pmset_s1, 20);
    multiset_insert(pmset_s1, 30);
    multiset_insert(pmset_s1, 40);

    printf("The original s1 =");
    for(it_s = multiset_begin(pmset_s1);
        !iterator_equal(it_s, multiset_end(pmset_s1));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");

    multiset_insert(pmset_s1, 20);
    multiset_insert_hint(pmset_s1, iterator_prev(multiset_end(pmset_s1)), 50);
    printf("After the insertions, s1 =");
    for(it_s = multiset_begin(pmset_s1);
        !iterator_equal(it_s, multiset_end(pmset_s1));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
```

```
    printf("\n");

    multiset_insert(pmset_s2, 100);
    multiset_insert_range(pmset_s2, iterator_next(multiset_begin(pmset_s1)),
        iterator_prev(multiset_end(pmset_s1)));
    printf("s2 =");
    for(it_s = multiset_begin(pmset_s2);
        !iterator_equal(it_s, multiset_end(pmset_s2));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");

    multiset_destroy(pmset_s1);
    multiset_destroy(pmset_s2);

    return 0;
}
```

- **Output**

```
The original s1 = 10 20 30 40
After the insertions, s1 = 10 20 20 30 40 50
s2 = 20 20 30 40 100
```

19. multiset_key_comp

返回 multiset_t 使用的键比较规则。

```
binary_function_t multiset_key_comp(
    const multiset_t* cpmset_multiset
);
```

- **Parameters**

cpmset_multiset: 指向 multiset_t 类型的指针。

- **Remarks**

由于 multiset_t 中数据本身就是键，所以这个函数的返回值与 multiset_value_comp()相同。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s1 = create_multiset(int);
    multiset_t* pmset_s2 = create_multiset(int);
```

```
    binary_function_t bfun_kl = NULL;
    bool_t b_result = false;
    int n_element1 = 0;
    int n_element2 = 0;

    if(pmset_s1 == NULL || pmset_s2 == NULL)
    {
        return -1;
    }

    multiset_init(pmset_s1);

    bfun_kl = multiset_key_comp(pmset_s1);
    n_element1 = 2;
    n_element2 = 3;
    (*bfun_kl)(&n_element1, &n_element2, &b_result);
    if(b_result)
    {
        printf("(*bfun_kl)(2, 3) return value of true, "
               "where bfun_kl is the function of s1.\n");
    }
    else
    {
        printf("(*bfun_kl)(2, 3) return value of false, "
               "where bfun_kl is the function of s1.\n");
    }

    multiset_destroy(pmset_s1);

    multiset_init_ex(pmset_s2, fun_greater_int);

    bfun_kl = multiset_key_comp(pmset_s2);
    (*bfun_kl)(&n_element1, &n_element2, &b_result);
    if(b_result)
    {
        printf("(*bfun_kl)(2, 3) return value of true, "
               "where bfun_kl is the function of s2.\n");
    }
    else
    {
        printf("(*bfun_kl)(2, 3) return value of false, "
               "where bfun_kl is the function of s2.\n");
    }

    multiset_destroy(pmset_s2);

    return 0;
}
```

- **Output**

```
(*bfun_kl)(2, 3) return value of true, where bfun_kl is the function of s1.
(*bfun_kl)(2, 3) return value of false, where bfun_kl is the function of s2.
```

20. multiset_less

测试第一个 multiset_t 是否小于第二个 multiset_t。

```
bool_t multiset_less(
```

```
    const multiset_t* cpmset_first,
    const multiset_t* cpmset_second
);
```

- **Parameters**

  cpmset_first:　指向第一个 multiset_t 类型的指针。
  cpmset_second: 指向第二个 multiset_t 类型的指针。

- **Remarks**

  这个函数要求两个 multiset_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s1 = create_multiset(int);
    multiset_t* pmset_s2 = create_multiset(int);
    multiset_t* pmset_s3 = create_multiset(int);
    int i = 0;

    if(pmset_s1 == NULL || pmset_s2 == NULL || pmset_s3 == NULL)
    {
        return -1;
    }

    multiset_init(pmset_s1);
    multiset_init(pmset_s2);
    multiset_init(pmset_s3);

    for(i = 0; i < 3; ++i)
    {
        multiset_insert(pmset_s1, i);
        multiset_insert(pmset_s2, i * i);
        multiset_insert(pmset_s3, i - 1);
    }

    if(multiset_less(pmset_s1, pmset_s2))
    {
        printf("The multiset s1 is less than the multiset s2.\n");
    }
    else
    {
        printf("The multiset s1 is not less than the multiset s2.\n");
    }

    if(multiset_less(pmset_s1, pmset_s3))
    {
        printf("The multiset s1 is less than the multiset s3.\n");
```

```
    }
    else
    {
        printf("The multset s1 is not less than the multiset s3.\n");
    }

    multiset_destroy(pmset_s1);
    multiset_destroy(pmset_s2);
    multiset_destroy(pmset_s3);

    return 0;
}
```

- **Output**

```
The multiset s1 is less than the multiset s2.
The multset s1 is not less than the multiset s3.
```

21. multiset_less_equal

测试第一个 multiset_t 是否小于等于第二个 multiset_t。

```
bool_t multiset_less_equal(
    const multiset_t* cpmset_first,
    const multiset_t* cpmset_second
);
```

- **Parameters**

  cpmset_first: 指向第一个 multiset_t 类型的指针。
  cpmset_second: 指向第二个 multiset_t 类型的指针。

- **Remarks**

  这个函数要求两个 multiset_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s1 = create_multiset(int);
    multiset_t* pmset_s2 = create_multiset(int);
    multiset_t* pmset_s3 = create_multiset(int);
    multiset_t* pmset_s4 = create_multiset(int);
    int i = 0;

    if(pmset_s1 == NULL || pmset_s2 == NULL ||
       pmset_s3 == NULL || pmset_s4 == NULL)
    {
```

```
        return -1;
    }

    multiset_init(pmset_s1);
    multiset_init(pmset_s2);
    multiset_init(pmset_s3);
    multiset_init(pmset_s4);

    for(i = 0; i < 3; ++i)
    {
        multiset_insert(pmset_s1, i);
        multiset_insert(pmset_s2, i * i);
        multiset_insert(pmset_s3, i - 1);
        multiset_insert(pmset_s4, i);
    }

    if(multiset_less_equal(pmset_s1, pmset_s2))
    {
        printf("The multiset s1 is less than or equal to the multiset s2.\n");
    }
    else
    {
        printf("The multiset s1 is greater than the multiset s2.\n");
    }

    if(multiset_less_equal(pmset_s1, pmset_s3))
    {
        printf("The multiset s1 is less than or equal to the multiset s3.\n");
    }
    else
    {
        printf("The multiset s1 is greater than the multiset s3.\n");
    }

    if(multiset_less_equal(pmset_s1, pmset_s4))
    {
        printf("The multiset s1 is less than or equal to the multiset s4.\n");
    }
    else
    {
        printf("The multiset s1 is greater than the multiset s4.\n");
    }

    multiset_destroy(pmset_s1);
    multiset_destroy(pmset_s2);
    multiset_destroy(pmset_s3);
    multiset_destroy(pmset_s4);

    return 0;
}
```

- **Output**

```
The multiset s1 is less than or equal to the multiset s2.
The multiset s1 is greater than the multiset s3.
The multiset s1 is less than or equal to the multiset s4.
```

22. multiset_lower_bound

返回 multiset_t 中等于指定数据的第一个数据的迭代器。

```
multiset_iterator_t multiset_lower_bound(
    const multiset_t* cpmset_multiset,
    element
);
```

- **Parameters**

cpmset_multiset: 指向 multiset_t 类型的指针。
element: 指定的数据。

- **Remarks**

如果 multiset_t 中包含指定的数据则返回等于指定数据的第一个数据的迭代器，如果 multiset_t 中不包含指定的数据则返回大于指定数据的第一个数据的迭代器，如果指定的数据是 multiset_t 中最大的数据则返回值等于 multiset_end()。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s1 = create_multiset(int);
    multiset_iterator_t it_s;

    if(pmset_s1 == NULL)
    {
        return -1;
    }

    multiset_init(pmset_s1);

    multiset_insert(pmset_s1, 10);
    multiset_insert(pmset_s1, 20);
    multiset_insert(pmset_s1, 30);

    it_s = multiset_lower_bound(pmset_s1, 20);
    printf("The element of multiset s1 with a key of 20 is: %d.\n",
        *(int*)iterator_get_pointer(it_s));

    it_s = multiset_lower_bound(pmset_s1, 40);
    /* If no match is found for the key, end() is is returend */
    if(iterator_equal(it_s, multiset_end(pmset_s1)))
    {
        printf("The multiset s1 doesn't have an element with a key of 40.\n");
    }
    else
    {
```

```
        printf("The element of multiset s1 with a key of 40 is: %d.\n",
            *(int*)iterator_get_pointer(it_s));
    }

    /*
     * The element at a specific location in the multiset can be found
     * by using a dereferenced iterator that addreses the location.
     */
    it_s = multiset_end(pmset_s1);
    it_s = iterator_prev(it_s);
    it_s = multiset_lower_bound(pmset_s1, *(int*)iterator_get_pointer(it_s));
    printf("The element of s1 with a key matching"
           " that of the last element is: %d.\n",
           *(int*)iterator_get_pointer(it_s));

    multiset_destroy(pmset_s1);

    return 0;
}
```

- **Output**

```
The element of multiset s1 with a key of 20 is: 20.
The multiset s1 doesn't have an element with a key of 40.
The element of s1 with a key matching that of the last element is: 30.
```

23. multiset_max_size

返回 multiset_t 能够保存的数据数量的最大可能值。

```
size_t multiset_max_size(
    const multiset_t* cpmset_multiset
);
```

- **Parameters**

  cpmset_multiset: 指向 multiset_t 类型的指针。

- **Remarks**

  这是一个与系统有关的常数。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s1 = create_multiset(int);

    if(pmset_s1 == NULL)
```

```
    {
        return -1;
    }

    multiset_init(pmset_s1);

    printf("The maximum possible length of the multiset is %d.\n",
        multiset_max_size(pmset_s1));

    multiset_destroy(pmset_s1);

    return 0;
}
```

- **Output**

```
The maximum possible length of the multiset is 1073741823.
```

24. multiset_not_equal

测试两个 multiset_t 是否不等。

```
bool_t multiset_not_equal(
    const multiset_t* cpmset_first,
    const multiset_t* cpmset_second
);
```

- **Parameters**

cpmset_first: 指向第一个 multiset_t 类型的指针。
cpmset_second: 指向第二个 multiset_t 类型的指针。

- **Remarks**

两个 multiset_t 中的数据对应相等，并且数量相等，函数返回 false，否则返回 true。如果两个 multiset_t 中的数据类型不同也认为不等。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s1 = create_multiset(int);
    multiset_t* pmset_s2 = create_multiset(int);
    multiset_t* pmset_s3 = create_multiset(int);
    int i = 0;

    if(pmset_s1 == NULL || pmset_s2 == NULL || pmset_s3 == NULL)
    {
        return -1;
```

```
    }

    multiset_init(pmset_s1);
    multiset_init(pmset_s2);
    multiset_init(pmset_s3);

    for(i = 0; i < 3; ++i)
    {
        multiset_insert(pmset_s1, i);
        multiset_insert(pmset_s2, i * i);
        multiset_insert(pmset_s3, i);
    }

    if(multiset_not_equal(pmset_s1, pmset_s2))
    {
        printf("The multisets s1 and s2 are not equal.\n");
    }
    else
    {
        printf("The multisets s1 and s2 are equal.\n");
    }

    if(multiset_not_equal(pmset_s1, pmset_s3))
    {
        printf("The multisets s1 and s3 are not equal.\n");
    }
    else
    {
        printf("The multisets s1 and s3 are equal.\n");
    }

    multiset_destroy(pmset_s1);
    multiset_destroy(pmset_s2);
    multiset_destroy(pmset_s3);

    return 0;
}
```

- **Output**

```
The multisets s1 and s2 are not equal.
The multisets s1 and s3 are equal.
```

25. multiset_size

返回 multiset_t 中数据的个数。

```
size_t multiset_size(
    const multiset_t* cpmset_multiset
);
```

- **Parameters**

cpmset_multiset**:** 指向 multiset_t 类型的指针。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s1 = create_multiset(int);

    if(pmset_s1 == NULL)
    {
        return -1;
    }

    multiset_init(pmset_s1);

    multiset_insert(pmset_s1, 1);
    printf("The multiset length is %d.\n", multiset_size(pmset_s1));

    multiset_insert(pmset_s1, 2);
    printf("The multiset length is now %d.\n", multiset_size(pmset_s1));

    multiset_destroy(pmset_s1);

    return 0;
}
```

- **Output**

```
The multiset length is 1.
The multiset length is now 2.
```

26. multiset_swap

交换两个 multiset_t 中的内容。

```
void multiset_swap(
    multiset_t* pmset_first,
    multiset_t* pmset_second
);
```

- **Parameters**

pmset_first: 指向第一个 multiset_t 类型的指针。
pmset_second: 指向第二个 multiset_t 类型的指针。

- **Remarks**

这个函数要求两个 multiset_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

```
/*
 * multiset_swap.c
```

```
 * compile with : -lcstl
 */

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s1 = create_multiset(int);
    multiset_t* pmset_s2 = create_multiset(int);
    multiset_iterator_t it_s;

    if(pmset_s1 == NULL || pmset_s2 == NULL)
    {
        return -1;
    }

    multiset_init(pmset_s1);
    multiset_init(pmset_s2);

    multiset_insert(pmset_s1, 10);
    multiset_insert(pmset_s1, 20);
    multiset_insert(pmset_s1, 30);
    multiset_insert(pmset_s2, 100);
    multiset_insert(pmset_s2, 200);

    printf("The original multiset s1 is:");
    for(it_s = multiset_begin(pmset_s1);
        !iterator_equal(it_s, multiset_end(pmset_s1));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");

    multiset_swap(pmset_s1, pmset_s2);
    printf("After swapping with s2, multiset s1 is:");
    for(it_s = multiset_begin(pmset_s1);
        !iterator_equal(it_s, multiset_end(pmset_s1));
        it_s = iterator_next(it_s))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_s));
    }
    printf("\n");

    multiset_destroy(pmset_s1);
    multiset_destroy(pmset_s2);

    return 0;
}
```

- **Output**

```
The original multiset s1 is: 10 20 30
After swapping with s2, multiset s1 is: 100 200
```

27. multiset_upper_bound

返回 multiset_t 中大于指定数据的第一个数据的迭代器。

```
multiset_iterator_t multiset_upper_bound(
    const multiset_t* cpmset_multiset,
    element
);
```

- **Parameters**

  cpmset_multiset: 指向 multiset_t 类型的指针。
  element: 指定的数据。

- **Remarks**

  如果指定的数据是 multiset_t 中最大的数据则返回值等于 multiset_end()。

- **Requirements**

  头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s1 = create_multiset(int);
    multiset_iterator_t it_s;

    if(pmset_s1 == NULL)
    {
        return -1;
    }

    multiset_init(pmset_s1);

    multiset_insert(pmset_s1, 10);
    multiset_insert(pmset_s1, 20);
    multiset_insert(pmset_s1, 30);

    it_s = multiset_upper_bound(pmset_s1, 20);
    printf("The first element of multiset s1 with a key "
           "greater than 20 is: %d.\n", *(int*)iterator_get_pointer(it_s));

    it_s = multiset_upper_bound(pmset_s1, 30);
    /* If no match is found for the key, end() is returned */
    if(iterator_equal(it_s, multiset_end(pmset_s1)))
    {
        printf("The multiset s1 doesn't have an element "
               "with a key greater than 30.\n");
    }
    else
    {
        printf("the element of multiset s1 with a key > 30 is: %d.\n",
            *(int*)iterator_get_pointer(it_s));
    }

    /*
```

```
     * The element at a specific location in the multiset can be found
     * by using a dereferenced iterator addressing the location.
     */
    it_s = multiset_begin(pmset_s1);
    it_s = multiset_upper_bound(pmset_s1, *(int*)iterator_get_pointer(it_s));
    printf("The first element of s1 with a key greater than that of the "
           "initial element of s1 is: %d.\n", *(int*)iterator_get_pointer(it_s));

    multiset_destroy(pmset_s1);

    return 0;
}
```

- **Output**

```
The first element of multiset s1 with a key greater than 20 is: 30.
The multiset s1 doesn't have an element with a key greater than 30.
The first element of s1 with a key greater than that of the initial element of s1
is: 20.
```

28. multiset_value_comp

返回 multiset_t 中使用的数据比较规则。

```
binary_function_t multiset_value_comp(
    const multiset_t* cpmset_multiset
);
```

- **Parameters**

cpmset_multiset: 指向 multiset_t 类型的指针。

- **Remarks**

由于 multiset_t 中数据本身就是键，所以这个函数的返回值与 multiset_key_comp()相同。

- **Requirements**

头文件 <cstl/cset.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multiset_t* pmset_s1 = create_multiset(int);
    multiset_t* pmset_s2 = create_multiset(int);
    binary_function_t bfun_vl = NULL;
    int n_element1 = 0;
    int n_element2 = 0;
    bool_t b_result = false;

    if(pmset_s1 == NULL || pmset_s2 == NULL)
```

```
    {
        return -1;
    }

    multiset_init(pmset_s1);

    bfun_vl = multiset_value_comp(pmset_s1);
    n_element1 = 2;
    n_element2 = 3;
    (*bfun_vl)(&n_element1, &n_element2, &b_result);
    if(b_result)
    {
        printf("(*bfun_vl)(2, 3) returns value of true,"
               " where bfun_vl is the function of s1.\n");
    }
    else
    {
        printf("(*bfun_vl)(2, 3) returns value of false,"
               " where bfun_vl is the function of s1.\n");
    }

    multiset_destroy(pmset_s1);

    multiset_init_ex(pmset_s2, fun_greater_int);

    bfun_vl = multiset_value_comp(pmset_s2);
    (*bfun_vl)(&n_element1, &n_element2, &b_result);
    if(b_result)
    {
        printf("(*bfun_vl)(2, 3) returns value of true,"
               " where bfun_vl is the function of s2.\n");
    }
    else
    {
        printf("(*bfun_vl)(2, 3) returns value of false,"
               " where bfun_vl is the function of s2.\n");
    }

    multiset_destroy(pmset_s2);

    return 0;
}
```

- **Output**

```
(*bfun_vl)(2, 3) returns value of true, where bfun_vl is the function of s1.
(*bfun_vl)(2, 3) returns value of false, where bfun_vl is the function of s2.
```

第七节　映射 map_t

映射 map_t 是关联容器，容器中保存的数据是 pair_t 类型。pair_t 的第一个数据是键，map_t 中的数据就是根据这个键排序的，在 map_t 中键不允许重复，也不可以直接或者间接修改键。pair_t 的第二个数据是值，值与键没有直接的关系，map_t 中对于值的唯一性没有要求，值对于 map_t 中的数据排序没有影响，可以直接或者间接修改值。

map_t 的迭代器是双向迭代器，插入新的数据不会破坏原有的迭代器，删除一个数据的时候只有指向该数据的迭代器失效。在 map_t 中查找，插入或者删除数据都是高效的，同时还可以使用键作为下标直接访问相应的值。

map_t 中的数据根据键按照指定规则自动排序，默认规则是与键相关的小于操作，用户也可以在初始化时指定自定义的规则。

- **Typedefs**

map_t	映射容器类型。
map_iterator_t	映射容器迭代器类型。

- **Operation Functions**

create_map	创建映射容器类型。
map_assign	为映射容器赋值。
map_at	通过下键直接访问值。
map_begin	返回指向映射中第一个数据的迭代器。
map_clear	删除映射中的所有数据。
map_count	统计映射中拥有指定键的数据的个数。
map_destroy	销毁映射容器。
map_empty	测试映射容器是否为空。
map_end	返回指向容器末尾的迭代器。
map_equal	测试两个映射容器是否相等。
map_equal_range	返回与指定键相等的数据区间。
map_erase	删除映射中与指定键值相等的数据。
map_erase_pos	删除映射中指定位置的数据。
map_erase_range	删除映射中指定的数据区间。
map_find	查找容器中拥有指定键的数据。
map_greater	测试第一个映射是否大于第二个映射。
map_greater_equal	测试第一个映射是否大于等于第二个映射。
map_init	初始化一个空映射。
map_init_copy	使用另一个映射初始化当前映射容器。
map_init_copy_range	使用指定的数据区间初始化映射容器。
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映射容器。
map_init_ex	使用指定的排序规则初始化一个空的映射容器。
map_insert	在映射容器中插入数据。
map_insert_hint	在映射容器中插入数据，同时给出位置提示。
map_insert_range	在映射容器中插入数据区间。
map_key_comp	返回映射容器使用的键比较规则。
map_less	测试第一个映射容器是否小于第二个映射容器。
map_less_equal	测试第一个映射容器是否小于等于第二个映射容器。
map_lower_bound	返回与指定键相等的第一个数据的迭代器。
map_max_size	返回映射容器中能够保存数据的最大数量的可能值。
map_not_equal	测试两个映射容器是否不等。
map_size	返回映射容器中数据的数量。

map_swap	交换两个映射容器的内容。
map_upper_bound	返回大于指定键的第一个数据的迭代器。
map_value_comp	返回映射容器使用的数据比较规则。

1. map_t

映射容器类型。

- **Requirements**
  头文件 <cstl/cmap.h>

- **Example**
  请参考 map_t 类型的其他操作函数。

2. map_iterator_t

映射容器类型的迭代器类型。

- **Remarks**
  map_iterator_t 是双向迭代器类型，不能通过迭代器来修改容器中数据的键，但是可以修改数据的值。

- **Requirements**
  头文件 <cstl/cmap.h>

- **Example**
  请参考 map_t 类型的其他操作函数。

3. create_map

创建 map_t 类型。

```
map_t* create_map(
    type
);
```

- **Parameters**
  type: 数据类型描述。

- **Remarks**
  函数成功返回指向 map_t 类型的指针，失败返回 NULL。

- **Requirements**
  头文件 <cstl/cmap.h>

- **Example**
  请参考 map_t 类型的其他操作函数。

4. map_assign

为 map_t 类型赋值。

```
void map_assign(
    map_t* pmap_dest,
    const map_t* cpmap_src
);
```

- **Parameters**

pmap_dest: 指向被赋值的 map_t 类型的指针。

cpmap_src: 指向赋值的 map_t 类型的指针。

- **Remarks**

要求两个 map_t 类型保存的数据具有相同的类型，否则函数的行为未定义。

- **Requirements**

头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);
    map_t* pmap_m2 = create_map(int, int);
    pair_t* ppair_p = create_pair(int, int);
    map_iterator_t it_m;

    if(pmap_m1 == NULL || pmap_m2 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    map_init(pmap_m1);
    map_init(pmap_m2);

    pair_make(ppair_p, 1, 10);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 2, 20);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 3, 30);
    map_insert(pmap_m1, ppair_p);

    pair_make(ppair_p, 4, 40);
    map_insert(pmap_m2, ppair_p);
    pair_make(ppair_p, 5, 50);
    map_insert(pmap_m2, ppair_p);
    pair_make(ppair_p, 6, 60);
    map_insert(pmap_m2, ppair_p);

    printf("m1 =");
```

```
    for(it_m = map_begin(pmap_m1);
        !iterator_equal(it_m, map_end(pmap_m1));
        it_m = iterator_next(it_m))
    {
        printf(" <%d, %d>",
            *(int*)pair_first(iterator_get_pointer(it_m)),
            *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    map_assign(pmap_m1, pmap_m2);

    printf("m1 =");
    for(it_m = map_begin(pmap_m1);
        !iterator_equal(it_m, map_end(pmap_m1));
        it_m = iterator_next(it_m))
    {
        printf(" <%d, %d>",
            *(int*)pair_first(iterator_get_pointer(it_m)),
            *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    pair_destroy(ppair_p);
    map_destroy(pmap_m1);
    map_destroy(pmap_m2);

    return 0;
}
```

- **Output**

```
m1 = <1, 10> <2, 20> <3, 30>
m1 = <4, 40> <5, 50> <6, 60>
```

5. map_at

通过键作为下标直接访问 map_t 中相应数据的值。

```
void* map_at(
    map_t* pmap_map,
    key
);
```

- **Parameters**

  pmap_map: 指向 map_t 类型的指针。
  key: 指定的键。

- **Remarks**

这个操作函数通过指定的键来访问 map_t 中相应数据的值，如果 map_t 中包含这个键，那么就返回指向相应数据的值的指针，如果 map_t 中不包含这个键，那么首先在 map_t 中插入一个数据，这个数据以指定的键为键，以值的默认数据为值，然后返回指向这个数据的值的指针。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);
    pair_t* ppair_p = create_pair(int, int);
    map_iterator_t it_m;

    if(pmap_m1 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    map_init(pmap_m1);

    /*
     * Insert a data value of 10 with a key of 1
     * into a map using the at() function.
     */
    *(int*)map_at(pmap_m1, 1) = 10;

    /* Insert datas into a map using insert() function. */
    pair_make(ppair_p, 2, 20);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 3, 30);
    map_insert(pmap_m1, ppair_p);

    printf("The keys of the mapped elements are:");
    for(it_m = map_begin(pmap_m1);
        !iterator_equal(it_m, map_end(pmap_m1));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_first(iterator_get_pointer(it_m)));
    }
    printf("\n");
    printf("The values of the mapped elements are:");
    for(it_m = map_begin(pmap_m1);
        !iterator_equal(it_m, map_end(pmap_m1));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    /*
     * If the key alread exists, at() funtiont changes the value
     * of the datum in the element.
     */
    *(int*)map_at(pmap_m1, 2) = 40;

    /*
     * at() function will also insert the value of the data
     * type's default value if the value is unspecified.
```

```
     */
    map_at(pmap_m1, 5);

    printf("The keys of the mapped elements are now:");
    for(it_m = map_begin(pmap_m1);
        !iterator_equal(it_m, map_end(pmap_m1));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_first(iterator_get_pointer(it_m)));
    }
    printf("\n");
    printf("The values of the mapped elements are now:");
    for(it_m = map_begin(pmap_m1);
        !iterator_equal(it_m, map_end(pmap_m1));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    pair_destroy(ppair_p);
    map_destroy(pmap_m1);

    return 0;
}
```

- **Output**

```
The keys of the mapped elements are: 1 2 3
The values of the mapped elements are: 10 20 30
The keys of the mapped elements are now: 1 2 3 5
The values of the mapped elements are now: 10 40 30 0
```

6. map_begin

返回指向 map_t 中第一个数据的迭代器。

```
map_iterator_t map_begin(
    const map_t* cpmap_map
);
```

- **Parameters**

cpmap_map: 指向 map_t 类型的指针。

- **Remarks**

如果 map_t 为空，这个函数的返回值与 map_end()相等。

- **Requirements**

头文件 <cstl/cmap.h>

- **Example**

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

#include <stdio.h>
```

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

int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);
    pair_t* ppair_p = create_pair(int, int);

    if(pmap_m1 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    map_init(pmap_m1);
    pair_init(ppair_p);

    pair_make(ppair_p, 0, 0);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 1, 1);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 2, 4);
    map_insert(pmap_m1, ppair_p);

    printf("The first element of m1 is %d\n",
        *(int*)pair_first(iterator_get_pointer(map_begin(pmap_m1))));

    map_erase_pos(pmap_m1, map_begin(pmap_m1));

    printf("The first element of m1 is now %d\n",
        *(int*)pair_first(iterator_get_pointer(map_begin(pmap_m1))));

    map_destroy(pmap_m1);
    pair_destroy(ppair_p);

    return 0;
}
```

- **Output**

```
The first element of m1 is 0
The first element of m1 is now 1
```

7. map_clear

删除 map_t 中所有的数据。

```
void map_clear(
    map_t* pmap_map
);
```

- **Parameters**

  cpmap_map: 指向 map_t 类型的指针。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

```
/*
 * map_clear.c
```

```
 * compile with : -lcstl
 */

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

int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);
    pair_t* ppair_p = create_pair(int, int);

    if(pmap_m1 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    map_init(pmap_m1);

    pair_make(ppair_p, 1, 1);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 2, 4);
    map_insert(pmap_m1, ppair_p);

    printf("The size of the map is initially %d.\n", map_size(pmap_m1));

    map_clear(pmap_m1);
    printf("The size of the map after clearing is %d.\n", map_size(pmap_m1));

    pair_destroy(ppair_p);
    map_destroy(pmap_m1);

    return 0;
}
```

- **Output**

```
The size of the map is initially 2.
The size of the map after clearing is 0.
```

8. map_count

统计 map_t 中包含指定键的数据的个数。

```
size_t map_count(
    const map_t* cpmap_map,
    key
);
```

- **Parameters**

cpmap_map: 指向 map_t 类型的指针。
key: 指定的键。

- **Remarks**

如果容器中没有包含指定键的数据返回 0， 否这返回包含指定键的数据的个数，map_t 中的值是 1。

- **Requirements**

头文件 <cstl/cmap.h>

● **Example**

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

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

int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);
    pair_t* ppair_p = create_pair(int, int);

    if(pmap_m1 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    map_init(pmap_m1);

    pair_make(ppair_p, 1, 1);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 2, 1);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 1, 4);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 2, 1);
    map_insert(pmap_m1, ppair_p);

    /* Keys must be unique in map, so duplicates are ignored */
    printf("The number of elements in m1 with a sort key of 1 is: %d.\n",
        map_count(pmap_m1, 1));
    printf("The number of elements in m1 with a sort key of 2 is: %d.\n",
        map_count(pmap_m1, 2));
    printf("The number of elements in m1 with a sort key of 3 is: %d.\n",
        map_count(pmap_m1, 3));

    pair_destroy(ppair_p);
    map_destroy(pmap_m1);

    return 0;
}
```

● **Output**

```
The number of elements in m1 with a sort key of 1 is: 1.
The number of elements in m1 with a sort key of 2 is: 1.
The number of elements in m1 with a sort key of 3 is: 0.
```

9. map_destroy

销毁 map_t 容器类型。

```
void map_destroy(
    map_t* pmap_map
);
```

- **Parameters**
  pmap_map: 指向 map_t 类型的指针。

- **Remarks**
  map_t 容器使用之后一定要销毁，否则 map_t 申请的资源不会被释放。

- **Requirements**
  头文件 <cstl/cmap.h>

- **Example**
  请参考 map_t 类型的其他操作函数。

10. map_empty

测试 map_t 是否为空。

```
bool_t map_empty(
    const map_t* cpmap_map
);
```

- **Parameters**
  cpmap_map: 指向 map_t 类型的指针。

- **Remarks**
  map_t 容器为空返回 true，否则返回 false。

- **Requirements**
  头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);
    map_t* pmap_m2 = create_map(int, int);
    pair_t* ppair_p = create_pair(int, int);

    if(pmap_m1 == NULL || pmap_m2 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    map_init(pmap_m1);
    map_init(pmap_m2);

    pair_make(ppair_p, 1, 1);
    map_insert(pmap_m1, ppair_p);
```

```
    if(map_empty(pmap_m1))
    {
        printf("The map m1 is empty.\n");
    }
    else
    {
        printf("The map m1 is not empty.\n");
    }

    if(map_empty(pmap_m2))
    {
        printf("The map m2 is empty.\n");
    }
    else
    {
        printf("The map m2 is not empty.\n");
    }

    pair_destroy(ppair_p);
    map_destroy(pmap_m1);
    map_destroy(pmap_m2);

    return 0;
}
```

- **Output**

```
The map m1 is not empty.
The map m2 is empty.
```

11. map_end

返回指向 map_t 容器末尾的迭代器。

```
map_iterator_t map_end(
    const map_t* cpmap_map
);
```

- **Parameters**

  cpmap_map: 指向 map_t 类型的指针。

- **Remarks**

  如果 map_t 为空，这个函数的返回值与 map_begin()相等。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

```
int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);
    pair_t* ppair_p = create_pair(int, int);
    map_iterator_t it_m;

    if(pmap_m1 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    map_init(pmap_m1);

    pair_make(ppair_p, 1, 10);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 2, 20);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 3, 30);
    map_insert(pmap_m1, ppair_p);

    it_m = map_end(pmap_m1);
    it_m = iterator_prev(it_m);
    printf("the value of the last element of m1 is: %d\n",
        *(int*)pair_second(iterator_get_pointer(it_m)));

    map_erase_pos(pmap_m1, it_m);

    it_m = map_end(pmap_m1);
    it_m = iterator_prev(it_m);
    printf("the value of the last element of m1 is now: %d\n",
        *(int*)pair_second(iterator_get_pointer(it_m)));

    pair_destroy(ppair_p);
    map_destroy(pmap_m1);

    return 0;
}
```

- **Output**

```
the value of the last element of m1 is: 30
the value of the last element of m1 is now: 20
```

12. map_equal

测试两个 map_t 容器是否相等。

```
bool_t map_equal(
    const map_t* cpmap_first,
    const map_t* cpmap_second
);
```

- **Parameters**

  cpmap_first: 指向第一个 map_t 类型的指针。
  cpmap_second: 指向第二个 map_t 类型的指针。

- **Remarks**

如果两个 map_t 容器中的数据都对应相等，并且数据个数相等，则返回 true 否则返回 false，如果两个 map_t 容器中保存的数据类型不同也认为是不等。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);
    map_t* pmap_m2 = create_map(int, int);
    map_t* pmap_m3 = create_map(int, int);
    pair_t* ppair_p = create_pair(int, int);
    int i = 0;

    if(pmap_m1 == NULL || pmap_m2 == NULL || pmap_m3 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    map_init(pmap_m1);
    map_init(pmap_m2);
    map_init(pmap_m3);
    pair_init(ppair_p);

    for(i = 0; i < 3; ++i)
    {
        pair_make(ppair_p, i, i);
        map_insert(pmap_m1, ppair_p);
        map_insert(pmap_m3, ppair_p);
        pair_make(ppair_p, i, i * i);
        map_insert(pmap_m2, ppair_p);
    }

    if(map_equal(pmap_m1, pmap_m2))
    {
        printf("The maps m1 and m2 are equal.\n");
    }
    else
    {
        printf("The maps m1 and m2 are not equal.\n");
    }

    if(map_equal(pmap_m1, pmap_m3))
    {
        printf("The maps m1 and m3 are equal.\n");
    }
    else
    {
        printf("The maps m1 and m3 are not equal.\n");
    }
```

```
    map_destroy(pmap_m1);
    map_destroy(pmap_m2);
    map_destroy(pmap_m3);
    pair_destroy(ppair_p);

    return 0;
}
```

- **Output**

```
The maps m1 and m2 are not equal.
The maps m1 and m3 are equal.
```

13. map_equal_range

返回 map_t 中包含拥有指定键的数据的数据区间。

```
range_t map_equal_range(
    const map_t* cpmap_map,
    key
);
```

- **Parameters**

cpmap_map: 指向 map_t 类型的指针。
key: 指定的键。

- **Remarks**

返回 map_t 中包含拥有指定键的数据的数据区间[range_t.it_begin, range_t.it_end)，其中 it_begin 是指向拥有指定键的第一个数据的迭代器，it_end 指向拥有大于指定键的第一个数据的迭代器。如果 map_t 中不包含拥有指定键的数据则 it_begin 与 it_end 相等。如果指定的键是 map_t 中最大的键则 it_end 等于 map_end()。

- **Requirements**

头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);
    pair_t* ppair_p = create_pair(int, int);
    map_iterator_t it_m;
    range_t r_r;

    if(pmap_m1 == NULL || ppair_p == NULL)
    {
        return -1;
    }
```

```
    pair_init(ppair_p);
    map_init(pmap_m1);

    pair_make(ppair_p, 1, 10);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 2, 20);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 3, 30);
    map_insert(pmap_m1, ppair_p);

    r_r = map_equal_range(pmap_m1, 2);

    printf("The lower bound of the element with a key of 2 in the map m1 is: %d.\n",
        *(int*)pair_second(iterator_get_pointer(r_r.it_begin)));
    printf("The upper bound of the element with a key of 2 in the map m1 is: %d.\n",
        *(int*)pair_second(iterator_get_pointer(r_r.it_end)));

    it_m = map_upper_bound(pmap_m1, 2);
    printf("A direct call of upper_bound(2) gives %d, matching "
           "the second element of the range returned by equal_range(2).\n",
           *(int*)pair_second(iterator_get_pointer(it_m)));

    r_r = map_equal_range(pmap_m1, 4);
    /* If no match is found for the key, both elements of the range return end() */
    if(iterator_equal(r_r.it_begin, map_end(pmap_m1)) &&
       iterator_equal(r_r.it_end, map_end(pmap_m1)))
    {
        printf("The map m1 doesn't have an element with a key less than 40.\n");
    }
    else
    {
        printf("The element of map m1 with a key >= 40 is %d.\n",
            *(int*)pair_first(iterator_get_pointer(r_r.it_begin)));
    }

    pair_destroy(ppair_p);
    map_destroy(pmap_m1);

    return 0;
}
```

- **Output**

```
The lower bound of the element with a key of 2 in the map m1 is: 20.
The upper bound of the element with a key of 2 in the map m1 is: 30.
A direct call of upper_bound(2) gives 30, matching the second element of the range
returned by equal_range(2).
The map m1 doesn't have an element with a key less than 40.
```

14. map_erase map_erase_pos map_erase_range

删除 map_t 容器中的指定数据。

```
size_t map_erase(
    map_t* pmap_map,
    key
);

void map_erase_pos(
```

```
    map_t* pmap_map,
    map_iterator_t it_pos
);

void map_erase_range(
    map_t* pmap_map,
    map_iterator_t it_begin,
    map_iterator_t it_end
);
```

- **Parameters**

pmap_map: 指向 map_t 类型的指针。
key: 被删除的数据的键。
it_pos: 指向被删除的数据的迭代器。
it_begin: 指向被删除的数据区间开始位置的迭代器。
it_end: 指向被删除的数据区间末尾的迭代器。

- **Remarks**

第一个函数删除 map_t 容器中包含指定键的数据，并返回删除数据的个数，如果容器中没有包含指定键的数据则返回 0。

第二个函数删除指定位置的数据。

第三个函数删除指定数据区间中的数据。

上面操作函数中的迭代器和数据区间都要求是有效的，无效的迭代器和数据区间将导致函数行为未定义。

- **Requirements**

头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);
    map_t* pmap_m2 = create_map(int, int);
    map_t* pmap_m3 = create_map(int, int);
    pair_t* ppair_p = create_pair(int, int);
    map_iterator_t it_m;
    int i = 0;
    size_t t_count = 0;

    if(pmap_m1 == NULL || pmap_m2 == NULL || pmap_m3 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    map_init(pmap_m1);
    map_init(pmap_m2);
    map_init(pmap_m3);
```

```
    for(i = 1; i < 5; ++i)
    {
        pair_make(ppair_p, i, i);
        map_insert(pmap_m1, ppair_p);
        pair_make(ppair_p, i, i * i);
        map_insert(pmap_m2, ppair_p);
        pair_make(ppair_p, i, i - 1);
        map_insert(pmap_m3, ppair_p);
    }

    /* The first function removes an element at a given position */
    it_m = map_begin(pmap_m1);
    it_m = iterator_next(it_m);
    map_erase_pos(pmap_m1, it_m);

    printf("After the second element is deleted, the map m1 is:");
    for(it_m = map_begin(pmap_m1);
        !iterator_equal(it_m, map_end(pmap_m1));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    /* The second function remvoes elements in the range [first, last) */
    map_erase_range(pmap_m2, iterator_next(map_begin(pmap_m2)),
        iterator_prev(map_end(pmap_m2)));

    printf("After the middle two elements are deleted, the map m2 is:");
    for(it_m = map_begin(pmap_m2);
        !iterator_equal(it_m, map_end(pmap_m2));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    /* The third function removes elements with a given key */
    t_count = map_erase(pmap_m3, 2);

    printf("After the element with a key of 2 is deleted, the map m3 is:");
    for(it_m = map_begin(pmap_m3);
        !iterator_equal(it_m, map_end(pmap_m3));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");
    /* The third function returns the number of elements remvoed */
    printf("The number of elements removed from m3 is: %d.\n", t_count);

    pair_destroy(ppair_p);
    map_destroy(pmap_m1);
    map_destroy(pmap_m2);
    map_destroy(pmap_m3);

    return 0;
}
```

- **Output**

```
After the second element is deleted, the map m1 is: 1 3 4
After the middle two elements are deleted, the map m2 is: 1 16
After the element with a key of 2 is deleted, the map m3 is: 0 2 3
The number of elements removed from m3 is: 1.
```

15. map_find

查找 map_t 中包含指定键的数据。

```
map_iterator_t map_find(
    const map_t* cpmap_map,
    key
);
```

- **Parameters**

  cpmap_map: 指向 map_t 类型的指针。
  key: 被删除的数据的键。

- **Remarks**

  如果 map_t 中存在包换指定键的数据，返回指向该数据的迭代器，否则返回 map_end()。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);
    pair_t* ppair_p = create_pair(int, int);
    map_iterator_t it_m;

    if(pmap_m1 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    map_init(pmap_m1);

    pair_make(ppair_p, 1, 10);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 2, 20);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 3, 30);
    map_insert(pmap_m1, ppair_p);

    it_m = map_find(pmap_m1, 2);
    printf("The element of map m1 with a key of 2 is: %d.\n",
        *(int*)pair_second(iterator_get_pointer(it_m)));
```

```
    /* If no match is found for the key, end() is returned */
    it_m = map_find(pmap_m1, 4);
    if(iterator_equal(it_m, map_end(pmap_m1)))
    {
        printf("The map m1 doesn't have an element with a key of 4.\n");
    }
    else
    {
        printf("The element of map m1 with a key of 4 is: %d.\n",
            *(int*)pair_second(iterator_get_pointer(it_m)));
    }

    /*
     * The element at a specific location in the map can be found
     * using a dereferenced iterator addressing the location
     */
    it_m = map_end(pmap_m1);
    it_m = iterator_prev(it_m);
    it_m = map_find(pmap_m1, *(int*)pair_first(iterator_get_pointer(it_m)));
    printf("The element of m1 with a key matching "
           "that of the last element is: %d.\n",
           *(int*)pair_second(iterator_get_pointer(it_m)));

    pair_destroy(ppair_p);
    map_destroy(pmap_m1);

    return 0;
}
```

- **Output**

```
The element of map m1 with a key of 2 is: 20.
The map m1 doesn't have an element with a key of 4.
The element of m1 with a key matching that of the last element is: 30.
```

16. map_greater

测试第一个 map_t 是否大于第二个 map_t。

```
bool_t map_greater(
    const map_t* cpmap_first,
    const map_t* cpmap_second
);
```

- **Parameters**

  cpmap_first: 指向第一个 map_t 类型的指针。
  cpmap_second: 指向第二个 map_t 类型的指针。

- **Remarks**

  这个函数要求两个 map_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

```
/*
```

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

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

int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);
    map_t* pmap_m2 = create_map(int, int);
    map_t* pmap_m3 = create_map(int, int);
    pair_t* ppair_p = create_pair(int, int);
    int i = 0;

    if(pmap_m1 == NULL || pmap_m2 == NULL || pmap_m3 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    map_init(pmap_m1);
    map_init(pmap_m2);
    map_init(pmap_m3);
    pair_init(ppair_p);

    for(i = 0; i < 3; ++i)
    {
        pair_make(ppair_p, i, i);
        map_insert(pmap_m1, ppair_p);
        pair_make(ppair_p, i, i * i);
        map_insert(pmap_m2, ppair_p);
        pair_make(ppair_p, i, i - 1);
        map_insert(pmap_m3, ppair_p);
    }

    if(map_greater(pmap_m1, pmap_m2))
    {
        printf("The map m1 is greater than the map m2.\n");
    }
    else
    {
        printf("The map m1 is not greater than the map m2.\n");
    }

    if(map_greater(pmap_m1, pmap_m3))
    {
        printf("The map m1 is greater than the map m3.\n");
    }
    else
    {
        printf("The map m1 is not greater than the map m3.\n");
    }

    map_destroy(pmap_m1);
    map_destroy(pmap_m2);
    map_destroy(pmap_m3);
    pair_destroy(ppair_p);

    return 0;
}
```

- **Output**

```
The map m1 is not greater than the map m2.
The map m1 is greater than the map m3.
```

17. map_greater_equal

测试第一个 map_t 是否大于等于第二个 map_t。

```
bool_t map_greater_equal(
    const map_t* cpmap_first,
    const map_t* cpmap_second
);
```

- **Parameters**

  cpmap_first: 指向第一个 map_t 类型的指针。
  cpmap_second: 指向第二个 map_t 类型的指针。

- **Remarks**

  这个函数要求两个 map_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);
    map_t* pmap_m2 = create_map(int, int);
    map_t* pmap_m3 = create_map(int, int);
    map_t* pmap_m4 = create_map(int, int);
    pair_t* ppair_p = create_pair(int, int);
    int i = 0;

    if(pmap_m1 == NULL || pmap_m2 == NULL || pmap_m3 == NULL ||
       pmap_m4 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    map_init(pmap_m1);
    map_init(pmap_m2);
    map_init(pmap_m3);
    map_init(pmap_m4);
    pair_init(ppair_p);

    for(i = 0; i < 3; ++i)
    {
```

```
        pair_make(ppair_p, i, i);
        map_insert(pmap_m1, ppair_p);
        map_insert(pmap_m4, ppair_p);
        pair_make(ppair_p, i, i * i);
        map_insert(pmap_m2, ppair_p);
        pair_make(ppair_p, i, i - 1);
        map_insert(pmap_m3, ppair_p);
    }

    if(map_greater_equal(pmap_m1, pmap_m2))
    {
        printf("The map m1 is greater than or equal to the map m2.\n");
    }
    else
    {
        printf("The map m1 is less than the map m2.\n");
    }

    if(map_greater_equal(pmap_m1, pmap_m3))
    {
        printf("The map m1 is greater than or equal to the map m3.\n");
    }
    else
    {
        printf("The map m1 is less than the map m3.\n");
    }

    if(map_greater_equal(pmap_m1, pmap_m4))
    {
        printf("The map m1 is greater than or equal to the map m4.\n");
    }
    else
    {
        printf("The map m1 is less than the map m4.\n");
    }

    map_destroy(pmap_m1);
    map_destroy(pmap_m2);
    map_destroy(pmap_m3);
    map_destroy(pmap_m4);
    pair_destroy(ppair_p);

    return 0;
}
```

- **Output**

```
The map m1 is less than the map m2.
The map m1 is greater than or equal to the map m3.
The map m1 is greater than or equal to the map m4.
```

18. map_init map_init_copy map_init_copy_range map_init_copy_range_ex map_init_ex

初始化 map_t 容器类型。

```
void map_init(
    map_t* pmap_map
);
```

```
void map_init_copy(
    map_t* pmap_map,
    const map_t* cpmap_src
);

void map_init_copy_range(
    map_t* pmap_map,
    map_iterator_t it_begin,
    map_iterator_t it_end
);

void map_init_copy_range_ex(
    map_t* pmap_map,
    map_iterator_t it_begin,
    map_iterator_t it_end,
    binary_function_t bfun_keycompare
);

void map_init_ex(
    map_t* pmap_map,
    binary_function_t bfun_keycompare
);
```

- **Parameters**

  pmap_map: 指向被初始化 map_t 类型的指针。
  cpmap_src: 指向用于初始化的 map_t 类型的指针。
  it_begin: 用于初始化的数据区间的开始位置。
  it_end: 用于初始化的数据区间的末尾位置。
  bfun_keycompare: 自定义的键排序规则。

- **Remarks**

  第一个函数初始化一个空的 map_t，使用与键的数据类型相关的小于操作函数作为默认的排序规则。
  第二个函数使用一个源 map_t 来初始化 map_t，数据的内容和排序规则都从源 map_t 复制。
  第三个函数使用指定的数据区间初始化一个 map_t，使用与键的数据类型相关的小于操作函数作为默认的排序规则。
  第四个函数使用指定的数据区间初始化一个 map_t，使用用户指定的排序规则。
  第五个函数初始化一个空的 map_t，使用用户指定的排序规则。
  上面的函数要求迭代器和数据区间是有效的，无效的迭代器或数据区间导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

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

```
{
    map_t* pmap_m0 = create_map(int, int);
    map_t* pmap_m1 = create_map(int, int);
    map_t* pmap_m2 = create_map(int, int);
    map_t* pmap_m3 = create_map(int, int);
    map_t* pmap_m4 = create_map(int, int);
    map_t* pmap_m5 = create_map(int, int);
    pair_t* ppair_p = create_pair(int, int);
    map_iterator_t it_m;

    if(pmap_m0 == NULL || pmap_m1 == NULL || pmap_m2 == NULL ||
       pmap_m3 == NULL || pmap_m4 == NULL || pmap_m5 == NULL ||
       ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);

    /* Create an empty map m0 of key type integer */
    map_init(pmap_m0);

    /*
     * Create an empty map m1 with the key comparison
     * function of less than, then insert 4 elements.
     */
    map_init_ex(pmap_m1, fun_less_int);
    pair_make(ppair_p, 1, 10);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 2, 20);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 3, 30);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 4, 40);
    map_insert(pmap_m1, ppair_p);

    /*
     * Create an empty map m2 with the key comparison
     * function of greater than, then insert 2 elements.
     */
    map_init_ex(pmap_m2, fun_greater_int);
    pair_make(ppair_p, 1, 10);
    map_insert(pmap_m2, ppair_p);
    pair_make(ppair_p, 2, 20);
    map_insert(pmap_m2, ppair_p);

    /* Create a copy, map m3, of map m1 */
    map_init_copy(pmap_m3, pmap_m1);

    /* Create a map m4 by copying the range m1[first, last) */
    map_init_copy_range(pmap_m4, map_begin(pmap_m1),
        iterator_advance(map_begin(pmap_m1), 2));

    /*
     * Create a map m5 by copying the range m3[first, last)
     * and with the key comparison function less than.
     */
    map_init_copy_range_ex(pmap_m5, map_begin(pmap_m3),
        iterator_next(map_begin(pmap_m3)), fun_less_int);
```

```
    printf("m1 =");
    for(it_m = map_begin(pmap_m1);
        !iterator_equal(it_m, map_end(pmap_m1));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    printf("m2 =");
    for(it_m = map_begin(pmap_m2);
        !iterator_equal(it_m, map_end(pmap_m2));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    printf("m3 =");
    for(it_m = map_begin(pmap_m3);
        !iterator_equal(it_m, map_end(pmap_m3));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    printf("m4 =");
    for(it_m = map_begin(pmap_m4);
        !iterator_equal(it_m, map_end(pmap_m4));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    printf("m5 =");
    for(it_m = map_begin(pmap_m5);
        !iterator_equal(it_m, map_end(pmap_m5));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    map_destroy(pmap_m0);
    map_destroy(pmap_m1);
    map_destroy(pmap_m2);
    map_destroy(pmap_m3);
    map_destroy(pmap_m4);
    map_destroy(pmap_m5);
    pair_destroy(ppair_p);

    return 0;
}
```

- **Output**

```
m1 = 10 20 30 40
m2 = 20 10
```

```
m3 = 10 20 30 40
m4 = 10 20
m5 = 10
```

19. map_insert map_insert_hint map_insert_range

向 map_t 中插入数据。

```
map_iterator_t map_insert(
    map_t* pmap_map,
    const pair_t* cppair_pair
);

map_iterator_t map_insert_hint(
    map_t* pmap_map,
    map_iterator_t it_hint,
    const pair_t* cppair_pair
);

void map_insert_range(
    map_t* pmap_map,
    map_iterator_t it_begin,
    map_iterator_t it_end
);
```

- **Parameters**

pmap_map: 指向 map_t 类型的指针。
cppair_pair: 插入的数据。
it_hint: 被插入数据的提示位置。
it_begin: 被插入的数据区间的开始位置。
it_end: 被插入的数据区间的末尾位置。

- **Remarks**

第一个函数向 map_t 中插入一个指定的数据，成功后返回指向该数据的迭代器，如果 map_t 中包含了该数据那么插入失败，返回 map_end()。

第二个函数向 map_t 中插入一个指定的数据，同时给出一个该数据被插入后的提示位置迭代器，如果这个位置符合 map_t 的排序规则就把这个数据放在提示位置中成功后返回指向该数据的迭代器，如果提示位置不正确则忽略提示位置，当数据插入成功后返回数据的实际位置迭代器，如果 map_t 中包含了该数据那么插入失败，返回 map_end()。

第三个函数插入指定的数据区间。

上面的函数要求迭代器和数据区间是有效的，无效的迭代器或数据区间导致函数的行为未定义。

- **Requirements**

头文件 <cstl/cmap.h>

- **Example**

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

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

```
int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);
    map_t* pmap_m2 = create_map(int, int);
    pair_t* ppair_p = create_pair(int, int);
    map_iterator_t it_m;

    if(pmap_m1 == NULL || pmap_m2 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    map_init(pmap_m1);
    map_init(pmap_m2);

    pair_make(ppair_p, 1, 10);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 2, 20);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 3, 30);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 4, 40);
    map_insert(pmap_m1, ppair_p);

    printf("The original key values of m1 =");
    for(it_m = map_begin(pmap_m1);
        !iterator_equal(it_m, map_end(pmap_m1));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_first(iterator_get_pointer(it_m)));
    }
    printf("\n");
    printf("The original mapped values of m1 =");
    for(it_m = map_begin(pmap_m1);
        !iterator_equal(it_m, map_end(pmap_m1));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    pair_make(ppair_p, 1, 10);
    it_m = map_insert(pmap_m1, ppair_p);
    if(!iterator_equal(it_m, map_end(pmap_m1)))
    {
        printf("The element 10 was inserted in m1 successfully.\n");
    }
    else
    {
        printf("The number 1 already exists in m1.\n");
    }

    /* The hint version of insert */
    pair_make(ppair_p, 5, 50);
    map_insert_hint(pmap_m1, iterator_prev(map_end(pmap_m1)), ppair_p);
    printf("After the insertions, the key values of m1 =");
    for(it_m = map_begin(pmap_m1);
        !iterator_equal(it_m, map_end(pmap_m1));
        it_m = iterator_next(it_m))
```

```
    {
        printf(" %d", *(int*)pair_first(iterator_get_pointer(it_m)));
    }
    printf("\n");
    printf("and mapped values of m1 =");
    for(it_m = map_begin(pmap_m1);
        !iterator_equal(it_m, map_end(pmap_m1));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    pair_make(ppair_p, 10, 100);
    map_insert(pmap_m2, ppair_p);
    /* The templatized version inserting a range */
    map_insert_range(pmap_m2, iterator_next(map_begin(pmap_m1)),
        iterator_prev(map_end(pmap_m1)));
    printf("After the insertions, the key values of m2 =");
    for(it_m = map_begin(pmap_m2);
        !iterator_equal(it_m, map_end(pmap_m2));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_first(iterator_get_pointer(it_m)));
    }
    printf("\n");
    printf("and mapped values of m2 =");
    for(it_m = map_begin(pmap_m2);
        !iterator_equal(it_m, map_end(pmap_m2));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    pair_destroy(ppair_p);
    map_destroy(pmap_m1);
    map_destroy(pmap_m2);

    return 0;
}
```

- **Output**

```
The original key values of m1 = 1 2 3 4
The original mapped values of m1 = 10 20 30 40
The number 1 already exists in m1.
After the insertions, the key values of m1 = 1 2 3 4 5
and mapped values of m1 = 10 20 30 40 50
After the insertions, the key values of m2 = 2 3 4 10
and mapped values of m2 = 20 30 40 100
```

20. map_key_comp

返回 map_t 使用的键的比较规则。

```
binary_function_t map_key_comp(
    const map_t* cpmap_map
);
```

- **Parameters**

  cpmap_map: 指向 map_t 类型的指针。

- **Remarks**

  这个排序规则是针对与数据中的键进行排序。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);
    map_t* pmap_m2 = create_map(int, int);
    binary_function_t bfun_kc = NULL;
    int n_element1 = 2;
    int n_element2 = 3;
    bool_t b_result = false;

    if(pmap_m1 == NULL || pmap_m2 == NULL)
    {
        return -1;
    }

    map_init_ex(pmap_m1, fun_less_int);

    bfun_kc = map_key_comp(pmap_m1);
    (*bfun_kc)(&n_element1, &n_element2, &b_result);
    if(b_result)
    {
        printf("(*bfun_kc)(2, 3) returns value of true,"
               " where bfun_kc is the function of m1.\n");
    }
    else
    {
        printf("(*bfun_kc)(2, 3) returns value of false,"
               " where bfun_kc is the function of m1.\n");
    }

    map_destroy(pmap_m1);

    map_init_ex(pmap_m2, fun_greater_int);

    bfun_kc = map_key_comp(pmap_m2);
    (*bfun_kc)(&n_element1, &n_element2, &b_result);
    if(b_result)
    {
        printf("(*bfun_kc)(2, 3) returns value of true,"
               " where bfun_kc is the function of m2.\n");
```

```
    }
    else
    {
        printf("(*bfun_kc)(2, 3) returns value of false,"
               " where bfun_kc is the function of m2.\n");
    }

    map_destroy(pmap_m2);

    return 0;
}
```

- **Output**

```
(*bfun_kc)(2, 3) returns value of true, where bfun_kc is the function of m1.
(*bfun_kc)(2, 3) returns value of false, where bfun_kc is the function of m2.
```

21. map_less

测试第一个 map_t 是否小于第二个 map_t。

```
bool_t map_less(
    const map_t* cpmap_first,
    const map_t* cpmap_second
);
```

- **Parameters**

cpmap_first: 指向第一个 map_t 类型的指针。
cpmap_second: 指向第二个 map_t 类型的指针。

- **Remarks**

这个函数要求两个 map_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);
    map_t* pmap_m2 = create_map(int, int);
    map_t* pmap_m3 = create_map(int, int);
    pair_t* ppair_p = create_pair(int, int);
    int i = 0;

    if(pmap_m1 == NULL || pmap_m2 == NULL || pmap_m3 == NULL || ppair_p == NULL)
    {
        return -1;
    }
```

```
    map_init(pmap_m1);
    map_init(pmap_m2);
    map_init(pmap_m3);
    pair_init(ppair_p);

    for(i = 0; i < 3; ++i)
    {
        pair_make(ppair_p, i, i);
        map_insert(pmap_m1, ppair_p);
        pair_make(ppair_p, i, i * i);
        map_insert(pmap_m2, ppair_p);
        pair_make(ppair_p, i, i - 1);
        map_insert(pmap_m3, ppair_p);
    }

    if(map_less(pmap_m1, pmap_m2))
    {
        printf("The map m1 is less than the map m2.\n");
    }
    else
    {
        printf("The map m1 is not less than the map m2.\n");
    }

    if(map_less(pmap_m1, pmap_m3))
    {
        printf("The map m1 is less than the map m3.\n");
    }
    else
    {
        printf("The map m1 is not less than the map m3.\n");
    }

    map_destroy(pmap_m1);
    map_destroy(pmap_m2);
    map_destroy(pmap_m3);
    pair_destroy(ppair_p);

    return 0;
}
```

- **Output**

```
The map m1 is less than the map m2.
The map m1 is not less than the map m3.
```

22. map_less_equal

测试第一个 map_t 是否小于等于第二个 map_t。

```
bool_t map_less_equal(
    const map_t* cpmap_first,
    const map_t* cpmap_second
);
```

- **Parameters**

cpmap_first: 指向第一个 map_t 类型的指针。

cpmap_second: 指向第二个 map_t 类型的指针。

- **Remarks**

这个函数要求两个 map_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);
    map_t* pmap_m2 = create_map(int, int);
    map_t* pmap_m3 = create_map(int, int);
    map_t* pmap_m4 = create_map(int, int);
    pair_t* ppair_p = create_pair(int, int);
    int i = 0;

    if(pmap_m1 == NULL || pmap_m2 == NULL || pmap_m3 == NULL ||
       pmap_m4 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    map_init(pmap_m1);
    map_init(pmap_m2);
    map_init(pmap_m3);
    map_init(pmap_m4);
    pair_init(ppair_p);

    for(i = 0; i < 3; ++i)
    {
        pair_make(ppair_p, i, i);
        map_insert(pmap_m1, ppair_p);
        map_insert(pmap_m4, ppair_p);
        pair_make(ppair_p, i, i * i);
        map_insert(pmap_m2, ppair_p);
        pair_make(ppair_p, i, i - 1);
        map_insert(pmap_m3, ppair_p);
    }

    if(map_less_equal(pmap_m1, pmap_m2))
    {
        printf("The map m1 is less than or equal to the map m2.\n");
    }
    else
    {
        printf("The map m1 is greater than the map m2.\n");
    }

    if(map_less_equal(pmap_m1, pmap_m3))
```

```
    {
        printf("The map m1 is less than or equal to the map m3.\n");
    }
    else
    {
        printf("The map m1 is greater than the map m3.\n");
    }

    if(map_less_equal(pmap_m1, pmap_m4))
    {
        printf("The map m1 is less than or equal to the map m4.\n");
    }
    else
    {
        printf("The map m1 is greater than the map m4.\n");
    }

    map_destroy(pmap_m1);
    map_destroy(pmap_m2);
    map_destroy(pmap_m3);
    map_destroy(pmap_m4);
    pair_destroy(ppair_p);

    return 0;
}
```

- **Output**

```
The map m1 is less than or equal to the map m2.
The map m1 is greater than the map m3.
The map m1 is less than or equal to the map m4.
```

23. map_lower_bound

返回 map_t 中包含指定键的第一个数据的迭代器。

```
map_iterator_t map_lower_bound(
    const map_t* cpmap_map,
    key
);
```

- **Parameters**

cpmap_map: 指向 map_t 类型的指针。
key: 指定的键。

- **Remarks**

如果 map_t 中不包含拥有指定键的数据则返回 map_t 中指向包含大于指定键的第一个数据的迭代器。如果指定的键是 map_t 中最大的键则返回 map_end()。

- **Requirements**

头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);
    pair_t* ppair_p = create_pair(int, int);
    map_iterator_t it_m;

    if(pmap_m1 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    map_init(pmap_m1);

    pair_make(ppair_p, 1, 10);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 2, 20);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 3, 30);
    map_insert(pmap_m1, ppair_p);

    it_m = map_lower_bound(pmap_m1, 2);
    printf("The first element of map m1 with a key of 2 is: %d.\n",
        *(int*)pair_second(iterator_get_pointer(it_m)));

    /* If no match is found for this key, end() is returned */
    it_m = map_lower_bound(pmap_m1, 4);
    if(iterator_equal(it_m, map_end(pmap_m1)))
    {
        printf("The map m1 doesn't have an element with a key of 4.\n");
    }
    else
    {
        printf("The element of map m1 with key of 4 is: %d.\n",
            *(int*)pair_second(iterator_get_pointer(it_m)));
    }

    /*
     * The element at a specific location in the map can be found
     * using a dereferenced iterator addressing the location.
     */
    it_m = map_end(pmap_m1);
    it_m = iterator_prev(it_m);
    it_m = map_lower_bound(pmap_m1, *(int*)pair_first(iterator_get_pointer(it_m)));
    printf("The element of m1 with a key matching"
           " that of the last element is: %d.\n",
           *(int*)pair_second(iterator_get_pointer(it_m)));

    pair_destroy(ppair_p);
    map_destroy(pmap_m1);

    return 0;
}
```

- **Output**

```
The first element of map m1 with a key of 2 is: 20.
The map m1 doesn't have an element with a key of 4.
The element of m1 with a key matching that of the last element is: 30.
```

24. map_max_size

返回 map_t 中包含数据数量的最大可能值。

```
size_t map_max_size(
    const map_t* cpmap_map
);
```

- **Parameters**

cpmap_map: 指向 map_t 类型的指针。

- **Remarks**

这是一个与系统相关的常数。

- **Requirements**

头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);

    if(pmap_m1 == NULL)
    {
        return -1;
    }

    map_init(pmap_m1);

    printf("The maximum possible length of the map is %d.\n",
        map_max_size(pmap_m1));
    printf("(Magnitude is machine specific.)\n");

    map_destroy(pmap_m1);

    return 0;
}
```

- **Output**

```
The maximum possible length of the map is 7895160.
(Magnitude is machine specific.)
```

25. map_not_equal

测试两个 map_t 是否不等。

```
bool_t map_not_equal(
    const map_t* cpmap_first,
    const map_t* cpmap_second
);
```

- **Parameters**

cpmap_first: 指向第一个 map_t 类型的指针。
cpmap_second: 指向第二个 map_t 类型的指针。

- **Remarks**

如果两个 map_t 容器中的数据都对应相等，并且数据个数相等，则返回 false 否则返回 true，如果两个 map_t 容器中保存的数据类型不同也认为是不等。

- **Requirements**

头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);
    map_t* pmap_m2 = create_map(int, int);
    map_t* pmap_m3 = create_map(int, int);
    pair_t* ppair_p = create_pair(int, int);
    int i = 0;

    if(pmap_m1 == NULL || pmap_m2 == NULL || pmap_m3 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    map_init(pmap_m1);
    map_init(pmap_m2);
    map_init(pmap_m3);
    pair_init(ppair_p);

    for(i = 0; i < 3; ++i)
    {
        pair_make(ppair_p, i, i);
        map_insert(pmap_m1, ppair_p);
        map_insert(pmap_m3, ppair_p);
        pair_make(ppair_p, i, i * i);
        map_insert(pmap_m2, ppair_p);
    }

    if(map_not_equal(pmap_m1, pmap_m2))
    {
```

```
        printf("The maps m1 and m2 are not equal.\n");
    }
    else
    {
        printf("The maps m1 and m2 are equal.\n");
    }

    if(map_not_equal(pmap_m1, pmap_m3))
    {
        printf("The maps m1 and m3 are not equal.\n");
    }
    else
    {
        printf("The maps m1 and m3 are equal.\n");
    }

    map_destroy(pmap_m1);
    map_destroy(pmap_m2);
    map_destroy(pmap_m3);
    pair_destroy(ppair_p);

    return 0;
}
```

- **Output**

```
The maps m1 and m2 are not equal.
The maps m1 and m3 are equal.
```

26. map_size

返回 map_t 中数据的数量。

```
size_t map_size(
    const map_t* cpmap_map
);
```

- **Parameters**

  cpmap_map: 指向 map_t 类型的指针。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);
    pair_t* ppair_p = create_pair(int, int);

    if(pmap_m1 == NULL || ppair_p == NULL)
```

```
    {
        return -1;
    }

    pair_init(ppair_p);
    map_init(pmap_m1);

    pair_make(ppair_p, 1, 1);
    map_insert(pmap_m1, ppair_p);
    printf("The map length is %d.\n", map_size(pmap_m1));

    pair_make(ppair_p, 2, 4);
    map_insert(pmap_m1, ppair_p);
    printf("The map length is now %d.\n", map_size(pmap_m1));

    pair_destroy(ppair_p);
    map_destroy(pmap_m1);

    return 0;
}
```

- **Output**

```
The map length is 1.
The map length is now 2.
```

27. map_swap

交换两个 map_t 中的内容。

```
void map_swap(
    map_t* pmap_first,
    map_t* pmap_second
);
```

- **Parameters**

  pmap_first: 指向第一个 map_t 类型的指针。
  pmap_second: 指向第二个 map_t 类型的指针。

- **Remarks**

  这个函数要求两个 map_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);
```

```
    map_t* pmap_m2 = create_map(int, int);
    pair_t* ppair_p = create_pair(int, int);
    map_iterator_t it_m;

    if(pmap_m1 == NULL || pmap_m2 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    map_init(pmap_m1);
    map_init(pmap_m2);

    pair_make(ppair_p, 1, 10);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 2, 20);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 3, 30);
    map_insert(pmap_m1, ppair_p);

    pair_make(ppair_p, 10, 100);
    map_insert(pmap_m2, ppair_p);
    pair_make(ppair_p, 20, 200);
    map_insert(pmap_m2, ppair_p);

    printf("The original map m1 is:");
    for(it_m = map_begin(pmap_m1);
        !iterator_equal(it_m, map_end(pmap_m1));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    map_swap(pmap_m1, pmap_m2);

    printf("After swapping with m2, map m1 is:");
    for(it_m = map_begin(pmap_m1);
        !iterator_equal(it_m, map_end(pmap_m1));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    pair_destroy(ppair_p);
    map_destroy(pmap_m1);
    map_destroy(pmap_m2);

    return 0;
}
```

- **Output**

```
The original map m1 is: 10 20 30
After swapping with m2, map m1 is: 100 200
```

28. map_upper_bound

返回 map_t 中包含大于指定键的第一个数据的迭代器。

```
map_iterator_t map_upper_bound(
    const map_t* cpmap_map,
    key
);
```

- **Parameters**

  cpmap_map: 指向 map_t 类型的指针。
  key: 指定的键。

- **Remarks**

  如果指定的键是 map_t 中最大的键则返回 map_end()。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);
    pair_t* ppair_p = create_pair(int, int);
    map_iterator_t it_m;

    if(pmap_m1 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    map_init(pmap_m1);

    pair_make(ppair_p, 1, 10);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 2, 20);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 3, 30);
    map_insert(pmap_m1, ppair_p);

    it_m = map_upper_bound(pmap_m1, 2);
    printf("The first element of map m1 with a key greater than 2 is: %d.\n",
        *(int*)pair_second(iterator_get_pointer(it_m)));

    /* If no match is found for the key, end is returned */
    it_m = map_upper_bound(pmap_m1, 4);
    if(iterator_equal(it_m, map_end(pmap_m1)))
    {
        printf("The map m1 doesn't have an element with a key greater than 4.\n");
```

```
    }
    else
    {
        printf("The element of map m1 with a key > 4 is: %d.\n",
            *(int*)pair_second(iterator_get_pointer(it_m)));
    }

    /*
     * The element at a specific location in the map can be found
     * using a dereferenced iterator addressing the location
     */
    it_m = map_begin(pmap_m1);
    it_m = map_upper_bound(pmap_m1, *(int*)pair_first(iterator_get_pointer(it_m)));
    printf("The first element of m1 with a key greater than"
           " that of the initial element of m1 is: %d.\n",
           *(int*)pair_second(iterator_get_pointer(it_m)));

    pair_destroy(ppair_p);
    map_destroy(pmap_m1);

    return 0;
}
```

- **Output**

```
The first element of map m1 with a key greater than 2 is: 30.
The map m1 doesn't have an element with a key greater than 4.
The first element of m1 with a key greater than that of the initial element of m1
is: 20.
```

29. map_value_comp

返回 map_t 使用的数据比较规则。

```
binary_function_t map_value_comp(
    const map_t* cpmap_map
);
```

- **Parameters**

  cpmap_map: 指向 map_t 类型的指针。

- **Remarks**

  这个规则是针对数据本身的比较规则而不是键或者值。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

```
int main(int argc, char* argv[])
{
    map_t* pmap_m1 = create_map(int, int);
    pair_t* ppair_p = create_pair(int, int);
    binary_function_t bfun_vc = NULL;
    bool_t b_result = false;
    map_iterator_t it_m1;
    map_iterator_t it_m2;

    if(pmap_m1 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    map_init_ex(pmap_m1, fun_less_int);

    pair_make(ppair_p, 1, 10);
    map_insert(pmap_m1, ppair_p);
    pair_make(ppair_p, 2, 5);
    map_insert(pmap_m1, ppair_p);

    it_m1 = map_find(pmap_m1, 1);
    it_m2 = map_find(pmap_m1, 2);
    bfun_vc = map_value_comp(pmap_m1);

    (*bfun_vc)(iterator_get_pointer(it_m1), iterator_get_pointer(it_m2), &b_result);
    if(b_result)
    {
        printf("The element (1, 10) precedes the element (2, 5).\n");
    }
    else
    {
        printf("The element (1, 10) does not precedes the element (2, 5).\n");
    }

    (*bfun_vc)(iterator_get_pointer(it_m2), iterator_get_pointer(it_m1), &b_result);
    if(b_result)
    {
        printf("The element (2, 5) precedes the element (1, 10).\n");
    }
    else
    {
        printf("The element (2, 5) does not precedes the element (1, 10).\n");
    }

    pair_destroy(ppair_p);
    map_destroy(pmap_m1);

    return 0;
}
```

- **Output**

```
The element (1, 10) precedes the element (2, 5).
The element (2, 5) does not precedes the element (1, 10).
```

第八节　多重映射 multimap_t

多重映射 multimap_t 是关联容器，容器中保存的数据是 pair_t 类型。pair_t 的第一个数据是键，multimap_t 中的数据就是根据这个键排序的，在 multimap_t 中键允许重复，不可以直接或者间接修改键。pair_t 的第二个数据是值，值与键没有直接的关系，值对于 multimap_t 中的数据排序没有影响，可以直接或者间接修改值。

multimap_t 的迭代器是双向迭代器，插入新的数据不会破坏原有的迭代器，删除一个数据的时候只有指向该数据的迭代器失效。在 multimap_t 中查找，插入或者删除数据都是高效的。

multimap_t 中的数据根据键按照指定规则自动排序，默认规则是与键相关的小于操作，用户也可以在初始化时指定自定义的规则。

- **Typedefs**

multimap_t	多重映射容器类型。
multimap_iterator_t	多重映射容器迭代器类型。

- **Operation Functions**

create_multimap	创建多重映射容器类型。
multimap_assign	为多重映射容器类型赋值。
multimap_begin	返回指向多重映射容器中的第一个数据的迭代器。
multimap_clear	删除多重映射容器中所有的数据。
multimap_count	返回多重映射容器中包含指定键的数据的个数。
multimap_destroy	销毁多重映射容器。
multimap_empty	测试多重映射容器是否为空。
multimap_end	返回指向多重映射容器末尾的迭代器。
multimap_equal	测试两个多重映射容器是否相等。
multimap_equal_range	返回多重映射容器中包含拥有指定键的数据的数据区间。
multimap_erase	删除多重映射容器中包含指定键的数据。
multimap_erase_pos	删除多重映射容器中指定位置的数据。
multimap_erase_range	删除多重映射容器中指定数据区间的数据。
multimap_find	在多重映射容器中查找包含指定键的数据。
multimap_greater	测试第一个多重映射容器是否大于第二个多重映射容器。
multimap_greater_equal	测试第一个多重映射容器是否大于等于第二个多重映射容器。
multimap_init	初始化一个空的多重映射容器。
multimap_init_copy	使用多重映射容器初始化当前多重映射容器。
multimap_init_copy_range	使用指定的数据区间初始化多重映射容器。
multimap_init_copy_range_ex	使用指定的数据区间和指定的排序规则初始化多重映射容器。
multimap_init_ex	使用指定的排序规则初始化一个空的多重映射容器。
multimap_insert	向多重映射容器中插入一个指定的数据。
multimap_insert_hint	向多重映射容器中插入一个指定的数据，同时给出位置提示。
multimap_insert_range	向多重映射容器中插入指定的数据区间。
multimap_key_comp	返回多重映射容器使用的键比较规则。
multimap_less	测试第一个多重映射容器是否小于第二个多重映射容器。

multimap_less_equal	测试第一个多重映射容器是否小于等于第二个多重映射容器。
multimap_lower_bound	返回多重映射容器中包含指定键的第一个数据的迭代器。
multimap_max_size	返回多重映射容器中能够保存的数据数量的最大可能值。
multimap_not_equal	测试两个多重映射容器是否不等。
multimap_size	返回多重映射容器中数据的数量。
multimap_swap	交换两个多重映射容器的内容。
multimap_upper_bound	返回多重映射容器中包含大于指定键的第一个数据的迭代器。
multimap_value_comp	返回多重映射容器中数据的比较规则。

1. multimap_t

多重映射容器类型。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

  请参考 multimap_t 类型的其他操作函数。

2. multimap_iterator_t

多重映射容器类型的迭代器类型。

- **Remarks**

  multimap_iterator_t 是双向迭代器类型，不能通过迭代器来修改容器中数据的键，但是可以修改数据的值。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

  请参考 multimap_t 类型的其他操作函数。

3. create_multimap

创建 multimap_t 类型。

```
multimap_t* create_multimap(
    type
);
```

- **Parameters**

  type: 数据类型描述。

- **Remarks**

  函数成功返回指向 multimap_t 类型的指针，失败返回 NULL。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

  请参考 multimap_t 类型的其他操作函数。

4. multimap_assign

为 multimap_t 赋值。

```
void multimap_assign(
    multimap_t* pmmap_dest,
    const multimap_t* cpmmap_src
);
```

- **Parameters**

  pmmap_dest: 指向被赋值的 multimap_t 类型的指针。
  cpmmap_src: 指向赋值的 multimap_t 类型的指针。

- **Remarks**

  要求两个 multimap_t 类型保存的数据具有相同的类型，否则函数的行为未定义。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multimap_t* pmmap_m1 = create_multimap(int, int);
    multimap_t* pmmap_m2 = create_multimap(int, int);
    pair_t* ppair_p = create_pair(int, int);
    multimap_iterator_t it_m;

    if(pmmap_m1 == NULL || pmmap_m2 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    multimap_init(pmmap_m1);
    multimap_init(pmmap_m2);

    pair_make(ppair_p, 1, 10);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 2, 20);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 3, 30);
    multimap_insert(pmmap_m1, ppair_p);

    pair_make(ppair_p, 4, 40);
    multimap_insert(pmmap_m2, ppair_p);
```

```
    pair_make(ppair_p, 5, 50);
    multimap_insert(pmmap_m2, ppair_p);
    pair_make(ppair_p, 6, 60);
    multimap_insert(pmmap_m2, ppair_p);

    printf("m1 =");
    for(it_m = multimap_begin(pmmap_m1);
        !iterator_equal(it_m, multimap_end(pmmap_m1));
        it_m = iterator_next(it_m))
    {
        printf(" <%d, %d>",
            *(int*)pair_first(iterator_get_pointer(it_m)),
            *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    multimap_assign(pmmap_m1, pmmap_m2);

    printf("m1 =");
    for(it_m = multimap_begin(pmmap_m1);
        !iterator_equal(it_m, multimap_end(pmmap_m1));
        it_m = iterator_next(it_m))
    {
        printf(" <%d, %d>",
            *(int*)pair_first(iterator_get_pointer(it_m)),
            *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    pair_destroy(ppair_p);
    multimap_destroy(pmmap_m1);
    multimap_destroy(pmmap_m2);

    return 0;
}
```

- **Output**

```
m1 = <1, 10> <2, 20> <3, 30>
m1 = <4, 40> <5, 50> <6, 60>
```

5. multimap_begin

返回指向 multimap_t 中第一个数据的迭代器。

```
multimap_iterator_t multimap_begin(
    const multimap_t* cpmmap_multimap
);
```

- **Parameters**

cpmmap_multimap: 指向 multimap_t 类型的指针。

- **Remarks**

如果 multimap_t 为空，这个函数的返回值与 multimap_end()相等。

- **Requirements**

头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multimap_t* pmmap_m1 = create_multimap(int, int);
    pair_t* ppair_p = create_pair(int, int);

    if(pmmap_m1 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    multimap_init(pmmap_m1);
    pair_init(ppair_p);

    pair_make(ppair_p, 0, 0);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 1, 1);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 2, 4);
    multimap_insert(pmmap_m1, ppair_p);

    printf("The first element of m1 is %d\n",
        *(int*)pair_first(iterator_get_pointer(multimap_begin(pmmap_m1))));

    multimap_erase_pos(pmmap_m1, multimap_begin(pmmap_m1));

    printf("The first element of m1 is now %d\n",
        *(int*)pair_first(iterator_get_pointer(multimap_begin(pmmap_m1))));

    multimap_destroy(pmmap_m1);
    pair_destroy(ppair_p);

    return 0;
}
```

- **Output**

```
The first element of m1 is 0
The first element of m1 is now 1
```

6. multimap_clear

删除 multimap_t 中所有的数据。

```
void multimap_clear(
    multimap_t* pmmap_multimap
);
```

- **Parameters**

cpmap_map: 指向 map_t 类型的指针。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multimap_t* pmmap_m1 = create_multimap(int, int);
    pair_t* ppair_p = create_pair(int, int);

    if(pmmap_m1 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    multimap_init(pmmap_m1);

    pair_make(ppair_p, 1, 1);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 2, 4);
    multimap_insert(pmmap_m1, ppair_p);

    printf("The size of the multimap is initially %d.\n",
        multimap_size(pmmap_m1));

    multimap_clear(pmmap_m1);
    printf("The size of the multimap after clearing is %d.\n",
        multimap_size(pmmap_m1));

    pair_destroy(ppair_p);
    multimap_destroy(pmmap_m1);

    return 0;
}
```

- **Output**

```
The size of the multimap is initially 2.
The size of the multimap after clearing is 0.
```

7. multimap_count

返回 multimap_t 中包含指定键的数据的数量。

```
size_t multimap_count(
    const multimap_t* cpmmap_multimap,
    key
);
```

- **Parameters**

  cpmmap_multimap: 指向 multimap_t 类型的指针。
  key: 指定的键。

- **Remarks**

  如果容器中没有包含指定键的数据返回 0， 否这返回包含指定键的数据的个数。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multimap_t* pmmap_m1 = create_multimap(int, int);
    pair_t* ppair_p = create_pair(int, int);

    if(pmmap_m1 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    multimap_init(pmmap_m1);

    pair_make(ppair_p, 1, 1);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 2, 1);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 1, 4);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 2, 1);
    multimap_insert(pmmap_m1, ppair_p);

    /* Keys must be unique in multimap, so duplicates are ignored */
    printf("The number of elements in m1 with a sort key of 1 is: %d.\n",
        multimap_count(pmmap_m1, 1));
    printf("The number of elements in m1 with a sort key of 2 is: %d.\n",
        multimap_count(pmmap_m1, 2));
    printf("The number of elements in m1 with a sort key of 3 is: %d.\n",
        multimap_count(pmmap_m1, 3));

    pair_destroy(ppair_p);
    multimap_destroy(pmmap_m1);

    return 0;
}
```

- **Output**

```
The number of elements in m1 with a sort key of 1 is: 2.
The number of elements in m1 with a sort key of 2 is: 2.
```

```
The number of elements in m1 with a sort key of 3 is: 0.
```

8. multimap_destroy

销毁 multimap_t 类型。

```
void multimap_destroy(
    multimap_t* pmmap_multimap
);
```

- **Parameters**

  pmmap_multimap: 指向 multimap_t 类型的指针。

- **Remarks**

  multimap_t 容器使用之后一定要销毁，否则 multimap_t 申请的资源不会被释放。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

  请参考 multimap_t 类型的其他操作函数。

9. multimap_empty

测试 multimap_t 是否为空。

```
bool_t multimap_empty(
    const multimap_t* cpmmap_multimap
);
```

- **Parameters**

  cpmmap_multimap: 指向 multimap_t 类型的指针。

- **Remarks**

  multimap_t 容器为空返回 true，否则返回 false。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multimap_t* pmmap_m1 = create_multimap(int, int);
    multimap_t* pmmap_m2 = create_multimap(int, int);
    pair_t* ppair_p = create_pair(int, int);
```

```
    if(pmmap_m1 == NULL || pmmap_m2 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    multimap_init(pmmap_m1);
    multimap_init(pmmap_m2);

    pair_make(ppair_p, 1, 1);
    multimap_insert(pmmap_m1, ppair_p);

    if(multimap_empty(pmmap_m1))
    {
        printf("The multimap m1 is empty.\n");
    }
    else
    {
        printf("The multimap m1 is not empty.\n");
    }

    if(multimap_empty(pmmap_m2))
    {
        printf("The multimap m2 is empty.\n");
    }
    else
    {
        printf("The multimap m2 is not empty.\n");
    }

    pair_destroy(ppair_p);
    multimap_destroy(pmmap_m1);
    multimap_destroy(pmmap_m2);

    return 0;
}
```

- **Output**

```
The multimap m1 is not empty.
The multimap m2 is empty.
```

10. multimap_end

返回指向 multimap_t 末尾的迭代器。

```
multimap_iterator_t multimap_end(
    const multimap_t* cpmmap_multimap
);
```

- **Parameters**

  cpmmap_multimap: 指向 multimap_t 类型的指针。

- **Remarks**

  如果 multimap_t 为空，这个函数的返回值与 multimap_begin()相等。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multimap_t* pmmap_m1 = create_multimap(int, int);
    pair_t* ppair_p = create_pair(int, int);
    multimap_iterator_t it_m;

    if(pmmap_m1 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    multimap_init(pmmap_m1);

    pair_make(ppair_p, 1, 10);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 2, 20);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 3, 30);
    multimap_insert(pmmap_m1, ppair_p);

    it_m = multimap_end(pmmap_m1);
    it_m = iterator_prev(it_m);
    printf("the value of the last element of m1 is: %d\n",
        *(int*)pair_second(iterator_get_pointer(it_m)));

    multimap_erase_pos(pmmap_m1, it_m);

    it_m = multimap_end(pmmap_m1);
    it_m = iterator_prev(it_m);
    printf("the value of the last element of m1 is now: %d\n",
        *(int*)pair_second(iterator_get_pointer(it_m)));

    pair_destroy(ppair_p);
    multimap_destroy(pmmap_m1);

    return 0;
}
```

- **Output**

```
the value of the last element of m1 is: 30
the value of the last element of m1 is now: 20
```

11. multimap_equal

测试两个 multimap_t 是否相等。

```
bool_t multimap_equal(
```

```
    const multimap_t* cpmmap_first,
    const multimap_t* cpmmap_second
);
```

- **Parameters**

  cpmmap_first: 指向第一个 multimap_t 类型的指针。
  cpmmap_second: 指向第二个 multimap_t 类型的指针。

- **Remarks**

如果两个 multimap_t 容器中的数据都对应相等，并且数据个数相等，则返回 true 否则返回 false，如果两个 multimap_t 容器中保存的数据类型不同也认为是不等。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multimap_t* pmmap_m1 = create_multimap(int, int);
    multimap_t* pmmap_m2 = create_multimap(int, int);
    multimap_t* pmmap_m3 = create_multimap(int, int);
    pair_t* ppair_p = create_pair(int, int);
    int i = 0;

    if(pmmap_m1 == NULL || pmmap_m2 == NULL || pmmap_m3 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    multimap_init(pmmap_m1);
    multimap_init(pmmap_m2);
    multimap_init(pmmap_m3);
    pair_init(ppair_p);

    for(i = 0; i < 3; ++i)
    {
        pair_make(ppair_p, i, i);
        multimap_insert(pmmap_m1, ppair_p);
        multimap_insert(pmmap_m3, ppair_p);
        pair_make(ppair_p, i, i * i);
        multimap_insert(pmmap_m2, ppair_p);
    }

    if(multimap_equal(pmmap_m1, pmmap_m2))
    {
        printf("The multimaps m1 and m2 are equal.\n");
    }
    else
    {
```

```
        printf("The multimaps m1 and m2 are not equal.\n");
    }

    if(multimap_equal(pmmap_m1, pmmap_m3))
    {
        printf("The multimaps m1 and m3 are equal.\n");
    }
    else
    {
        printf("The multimaps m1 and m3 are not equal.\n");
    }

    multimap_destroy(pmmap_m1);
    multimap_destroy(pmmap_m2);
    multimap_destroy(pmmap_m3);
    pair_destroy(ppair_p);

    return 0;
}
```

- **Output**

```
The multimaps m1 and m2 are not equal.
The multimaps m1 and m3 are equal.
```

12. multimap_equal_range

返回 multimap_t 中包含拥有指定键的数据的数据区间。

```
range_t multimap_equal_range(
    const multimap_t* cpmmap_multimap,
    key
);
```

- **Parameters**

cpmmap_multimap: 指向 multimap_t 类型的指针。
key: 指定的键。

- **Remarks**

返回 multimap_t 中包含拥有指定键的数据的数据区间[range_t.it_begin, range_t.it_end)，其中 it_begin 是指向拥有指定键的第一个数据的迭代器，it_end 指向拥有大于指定键的第一个数据的迭代器。如果 multimap_t 中不包含拥有指定键的数据则 it_begin 与 it_end 相等。如果指定的键是 multimap_t 中最大的键则 it_end 等于 multimap_end()。

- **Requirements**

头文件 <cstl/cmap.h>

- **Example**

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

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

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

```
{
    multimap_t* pmmap_m1 = create_multimap(int, int);
    pair_t* ppair_p = create_pair(int, int);
    multimap_iterator_t it_m;
    range_t r_r;

    if(pmmap_m1 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    multimap_init(pmmap_m1);

    pair_make(ppair_p, 1, 10);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 2, 20);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 3, 30);
    multimap_insert(pmmap_m1, ppair_p);

    r_r = multimap_equal_range(pmmap_m1, 2);

    printf("The lower bound of the element with a key of 2 "
           "in the multimap m1 is: %d.\n",
           *(int*)pair_second(iterator_get_pointer(r_r.it_begin)));
    printf("The upper bound of the element with a key of 2 "
           "in the multimap m1 is: %d.\n",
           *(int*)pair_second(iterator_get_pointer(r_r.it_end)));

    it_m = multimap_upper_bound(pmmap_m1, 2);
    printf("A direct call of upper_bound(2) gives %d, matching "
           "the second element of the range returned by equal_range(2).\n",
           *(int*)pair_second(iterator_get_pointer(it_m)));

    r_r = multimap_equal_range(pmmap_m1, 4);
    /* If no match is found for the key, both elements of the range return end() */
    if(iterator_equal(r_r.it_begin, multimap_end(pmmap_m1)) &&
       iterator_equal(r_r.it_end, multimap_end(pmmap_m1)))
    {
        printf("The multimap m1 doesn't have an element"
               " with a key less than 40.\n");
    }
    else
    {
        printf("The element of multimap m1 with a key >= 40 is %d.\n",
            *(int*)pair_first(iterator_get_pointer(r_r.it_begin)));
    }

    pair_destroy(ppair_p);
    multimap_destroy(pmmap_m1);

    return 0;
}
```

- **Output**

```
The lower bound of the element with a key of 2 in the multimap m1 is: 20.
The upper bound of the element with a key of 2 in the multimap m1 is: 30.
A direct call of upper_bound(2) gives 30, matching the second element of the range
```

```
returned by equal_range(2).
The multimap m1 doesn't have an element with a key less than 40.
```

13. multimap_erase multimap_erase_pos multimap_erase_range

删除 multimap_t 中的数据。

```
size_t multimap_erase(
    multimap_t* pmmap_multimap,
    key
);

void multimap_erase_pos(
    multimap_t* pmmap_multimap,
    multimap_iterator_t it_pos
);

void multimap_erase_range(
    multimap_t* pmmap_multimap,
    multimap_iterator_t it_begin,
    multimap_iterator_t it_end
);
```

- **Parameters**

pmmap_multimap: 指向 multimap_t 类型的指针。
key: 被删除的数据的键。
it_pos: 指向被删除的数据的迭代器。
it_begin: 指向被删除的数据区间开始位置的迭代器。
it_end: 指向被删除的数据区间末尾的迭代器。

- **Remarks**

第一个函数删除 multimap_t 容器中包含指定键的数据，并返回删除数据的个数，如果容器中没有包含指定键的数据则返回 0。
第二个函数删除指定位置的数据。
第三个函数删除指定数据区间中的数据。
上面操作函数中的迭代器和数据区间都要求是有效的，无效的迭代器和数据区间将导致函数行为未定义。

- **Requirements**

头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multimap_t* pmmap_m1 = create_multimap(int, int);
    multimap_t* pmmap_m2 = create_multimap(int, int);
    multimap_t* pmmap_m3 = create_multimap(int, int);
    pair_t* ppair_p = create_pair(int, int);
```

```
    multimap_iterator_t it_m;
    int i = 0;
    size_t t_count = 0;

    if(pmmap_m1 == NULL || pmmap_m2 == NULL || pmmap_m3 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    multimap_init(pmmap_m1);
    multimap_init(pmmap_m2);
    multimap_init(pmmap_m3);

    for(i = 1; i < 5; ++i)
    {
        pair_make(ppair_p, i, i);
        multimap_insert(pmmap_m1, ppair_p);
        pair_make(ppair_p, i, i * i);
        multimap_insert(pmmap_m2, ppair_p);
        pair_make(ppair_p, i, i - 1);
        multimap_insert(pmmap_m3, ppair_p);
    }

    /* The first function removes an element at a given position */
    it_m = multimap_begin(pmmap_m1);
    it_m = iterator_next(it_m);
    multimap_erase_pos(pmmap_m1, it_m);

    printf("After the second element is deleted, the multimap m1 is:");
    for(it_m = multimap_begin(pmmap_m1);
        !iterator_equal(it_m, multimap_end(pmmap_m1));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    /* The second function remvoes elements in the range [first, last) */
    multimap_erase_range(pmmap_m2, iterator_next(multimap_begin(pmmap_m2)),
        iterator_prev(multimap_end(pmmap_m2)));

    printf("After the middle two elements are deleted, the multimap m2 is:");
    for(it_m = multimap_begin(pmmap_m2);
        !iterator_equal(it_m, multimap_end(pmmap_m2));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    /* The third function removes elements with a given key */
    pair_make(ppair_p, 2, 5);
    multimap_insert(pmmap_m3, ppair_p);
    t_count = multimap_erase(pmmap_m3, 2);

    printf("After the element with a key of 2 is deleted, the multimap m3 is:");
    for(it_m = multimap_begin(pmmap_m3);
        !iterator_equal(it_m, multimap_end(pmmap_m3));
        it_m = iterator_next(it_m))
```

```
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");
    /* The third function returns the number of elements remvoed */
    printf("The number of elements removed from m3 is: %d.\n", t_count);

    pair_destroy(ppair_p);
    multimap_destroy(pmmap_m1);
    multimap_destroy(pmmap_m2);
    multimap_destroy(pmmap_m3);

    return 0;
}
```

- **Output**

```
After the second element is deleted, the multimap m1 is: 1 3 4
After the middle two elements are deleted, the multimap m2 is: 1 16
After the element with a key of 2 is deleted, the multimap m3 is: 0 2 3
The number of elements removed from m3 is: 2.
```

14. multimap_find

在 multimap_t 中查找包含指定键的数据。

```
multimap_iterator_t multimap_find(
    const multimap_t* cpmmap_multimap,
    key
);
```

- **Parameters**

cpmmap_multimap: 指向 multimap_t 类型的指针。
key: 被删除的数据的键。

- **Remarks**

如果 multimap_t 中存在包换指定键的数据，返回指向该数据的迭代器，否则返回 multimap_end()。

- **Requirements**

头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multimap_t* pmmap_m1 = create_multimap(int, int);
    pair_t* ppair_p = create_pair(int, int);
    multimap_iterator_t it_m;

    if(pmmap_m1 == NULL || ppair_p == NULL)
```

```
    {
        return -1;
    }

    pair_init(ppair_p);
    multimap_init(pmmap_m1);

    pair_make(ppair_p, 1, 10);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 2, 20);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 3, 20);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 3, 30);
    multimap_insert(pmmap_m1, ppair_p);

    it_m = multimap_find(pmmap_m1, 2);
    printf("The element of multimap m1 with a key of 2 is: %d.\n",
        *(int*)pair_second(iterator_get_pointer(it_m)));

    it_m = multimap_find(pmmap_m1, 3);
    printf("The first element of multimap m1 with a key of 3 is: %d.\n",
        *(int*)pair_second(iterator_get_pointer(it_m)));

    /* If no match is found for the key, end() is returned */
    it_m = multimap_find(pmmap_m1, 4);
    if(iterator_equal(it_m, multimap_end(pmmap_m1)))
    {
        printf("The multimap m1 doesn't have an element with a key of 4.\n");
    }
    else
    {
        printf("The element of multimap m1 with a key of 4 is: %d.\n",
            *(int*)pair_second(iterator_get_pointer(it_m)));
    }

    /*
     * The element at a specific location in the multimap can be found
     * using a dereferenced iterator addressing the location
     */
    it_m = multimap_end(pmmap_m1);
    it_m = iterator_prev(it_m);
    it_m = multimap_find(pmmap_m1, *(int*)pair_first(iterator_get_pointer(it_m)));
    printf("The element of m1 with a key matching "
           "that of the last element is: %d.\n",
           *(int*)pair_second(iterator_get_pointer(it_m)));

    /*
     * Note that the first element with a key equal to
     * the key of the last element is not the last element.
     */
    if(iterator_equal(it_m, iterator_prev(multimap_end(pmmap_m1))))
    {
        printf("This is the last element of multimap m1.\n");
    }
    else
    {
        printf("This is not the last element of multimap m1.\n");
    }
```

```
    pair_destroy(ppair_p);
    multimap_destroy(pmmap_m1);

    return 0;
}
```

- **Output**

```
The element of multimap m1 with a key of 2 is: 20.
The first element of multimap m1 with a key of 3 is: 20.
The multimap m1 doesn't have an element with a key of 4.
The element of m1 with a key matching that of the last element is: 20.
This is not the last element of multimap m1.
```

15. multimap_greater

测试第一个 multimap_t 是否大于第二个 multimap_t。

```
bool_t multimap_greater(
    const multimap_t* cpmmap_first,
    const multimap_t* cpmmap_second
);
```

- **Parameters**

cpmmap_first: 指向第一个 multimap_t 类型的指针。
cpmmap_second: 指向第二个 multimap_t 类型的指针。

- **Remarks**

这个函数要求两个 multimap_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multimap_t* pmmap_m1 = create_multimap(int, int);
    multimap_t* pmmap_m2 = create_multimap(int, int);
    multimap_t* pmmap_m3 = create_multimap(int, int);
    pair_t* ppair_p = create_pair(int, int);
    int i = 0;

    if(pmmap_m1 == NULL || pmmap_m2 == NULL || pmmap_m3 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    multimap_init(pmmap_m1);
    multimap_init(pmmap_m2);
```

```
    multimap_init(pmmap_m3);
    pair_init(ppair_p);

    for(i = 0; i < 3; ++i)
    {
        pair_make(ppair_p, i, i);
        multimap_insert(pmmap_m1, ppair_p);
        pair_make(ppair_p, i, i * i);
        multimap_insert(pmmap_m2, ppair_p);
        pair_make(ppair_p, i, i - 1);
        multimap_insert(pmmap_m3, ppair_p);
    }

    if(multimap_greater(pmmap_m1, pmmap_m2))
    {
        printf("The multimap m1 is greater than the multimap m2.\n");
    }
    else
    {
        printf("The multimap m1 is not greater than the multimap m2.\n");
    }

    if(multimap_greater(pmmap_m1, pmmap_m3))
    {
        printf("The multimap m1 is greater than the multimap m3.\n");
    }
    else
    {
        printf("The multimap m1 is not greater than the multimap m3.\n");
    }

    multimap_destroy(pmmap_m1);
    multimap_destroy(pmmap_m2);
    multimap_destroy(pmmap_m3);
    pair_destroy(ppair_p);

    return 0;
}
```

- **Output**

```
The multimap m1 is not greater than the multimap m2.
The multimap m1 is greater than the multimap m3.
```

16. multimap_greater_equal

测试第一个 multimap_t 是否大于等于第二个 multimap_t。

```
bool_t multimap_greater_equal(
    const multimap_t* cpmmap_first,
    const multimap_t* cpmmap_second
);
```

- **Parameters**

cpmmap_first: 指向第一个 multimap_t 类型的指针。
cpmmap_second: 指向第二个 multimap_t 类型的指针。

- **Remarks**

这个函数要求两个 multimap_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multimap_t* pmmap_m1 = create_multimap(int, int);
    multimap_t* pmmap_m2 = create_multimap(int, int);
    multimap_t* pmmap_m3 = create_multimap(int, int);
    multimap_t* pmmap_m4 = create_multimap(int, int);
    pair_t* ppair_p = create_pair(int, int);
    int i = 0;

    if(pmmap_m1 == NULL || pmmap_m2 == NULL || pmmap_m3 == NULL ||
       pmmap_m4 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    multimap_init(pmmap_m1);
    multimap_init(pmmap_m2);
    multimap_init(pmmap_m3);
    multimap_init(pmmap_m4);
    pair_init(ppair_p);

    for(i = 0; i < 3; ++i)
    {
        pair_make(ppair_p, i, i);
        multimap_insert(pmmap_m1, ppair_p);
        multimap_insert(pmmap_m4, ppair_p);
        pair_make(ppair_p, i, i * i);
        multimap_insert(pmmap_m2, ppair_p);
        pair_make(ppair_p, i, i - 1);
        multimap_insert(pmmap_m3, ppair_p);
    }

    if(multimap_greater_equal(pmmap_m1, pmmap_m2))
    {
        printf("The multimap m1 is greater than or equal to the multimap m2.\n");
    }
    else
    {
        printf("The multimap m1 is less than the multimap m2.\n");
    }

    if(multimap_greater_equal(pmmap_m1, pmmap_m3))
    {
        printf("The multimap m1 is greater than or equal to the multimap m3.\n");
    }
```

```
    else
    {
        printf("The multimap m1 is less than the multimap m3.\n");
    }

    if(multimap_greater_equal(pmmap_m1, pmmap_m4))
    {
        printf("The multimap m1 is greater than or equal to the multimap m4.\n");
    }
    else
    {
        printf("The multimap m1 is less than the multimap m4.\n");
    }

    multimap_destroy(pmmap_m1);
    multimap_destroy(pmmap_m2);
    multimap_destroy(pmmap_m3);
    multimap_destroy(pmmap_m4);
    pair_destroy(ppair_p);

    return 0;
}
```

- **Output**

```
The multimap m1 is less than the multimap m2.
The multimap m1 is greater than or equal to the multimap m3.
The multimap m1 is greater than or equal to the multimap m4.
```

17. multimap_init multimap_init_copy multimap_init_copy_range multimap_init_copy_range_ex multimap_init_ex

初始化 multimap_t。

```
void multimap_init(
    multimap_t* pmmap_multimap
);

void multimap_init_copy(
    multimap_t* pmmap_multimap,
    const multimap_t* cpmmap_src
);

void multimap_init_copy_range(
    multimap_t* pmmap_multimap,
    multimap_iterator_t it_begin,
    multimap_iterator_t it_end
);

void multimap_init_copy_range_ex(
    multimap_t* pmmap_multimap,
    multimap_iterator_t it_begin,
    multimap_iterator_t it_end,
    binary_function_t bfun_keycompare
);

void multimap_init_ex(
```

```
    multimap_t* pmmap_multimap,
    binary_function_t bfun_keycompare
);
```

- **Parameters**

  pmmap_multimap: 指向被初始化 multimap_t 类型的指针。
  cpmmap_src: 指向用于初始化的 multimap_t 类型的指针。
  it_begin: 用于初始化的数据区间的开始位置。
  it_end: 用于初始化的数据区间的末尾位置。
  bfun_keycompare: 自定义的键排序规则。

- **Remarks**

  第一个函数初始化一个空的 multimap_t，使用与键的数据类型相关的小于操作函数作为默认的排序规则。
  第二个函数使用一个源 multimap_t 来初始化 multimap_t，数据的内容和排序规则都从源 multimap_t 复制。
  第三个函数使用指定的数据区间初始化一个 multimap_t，使用与键的数据类型相关的小于操作函数作为默认的排序规则。
  第四个函数使用指定的数据区间初始化一个 multimap_t，使用用户指定的排序规则。
  第五个函数初始化一个空的 multimap_t，使用用户指定的排序规则。
  上面的函数要求迭代器和数据区间是有效的，无效的迭代器或数据区间导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multimap_t* pmmap_m0 = create_multimap(int, int);
    multimap_t* pmmap_m1 = create_multimap(int, int);
    multimap_t* pmmap_m2 = create_multimap(int, int);
    multimap_t* pmmap_m3 = create_multimap(int, int);
    multimap_t* pmmap_m4 = create_multimap(int, int);
    multimap_t* pmmap_m5 = create_multimap(int, int);
    pair_t* ppair_p = create_pair(int, int);
    multimap_iterator_t it_m;

    if(pmmap_m0 == NULL || pmmap_m1 == NULL || pmmap_m2 == NULL ||
       pmmap_m3 == NULL || pmmap_m4 == NULL || pmmap_m5 == NULL ||
       ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);

    /* Create an empty multimap m0 of key type integer */
    multimap_init(pmmap_m0);
```

```
/*
 * Create an empty multimap m1 with the key comparison
 * function of less than, then insert 4 elements.
 */
multimap_init_ex(pmmap_m1, fun_less_int);
pair_make(ppair_p, 1, 10);
multimap_insert(pmmap_m1, ppair_p);
pair_make(ppair_p, 2, 20);
multimap_insert(pmmap_m1, ppair_p);
pair_make(ppair_p, 3, 30);
multimap_insert(pmmap_m1, ppair_p);
pair_make(ppair_p, 4, 40);
multimap_insert(pmmap_m1, ppair_p);

/*
 * Create an empty multimap m2 with the key comparison
 * function of greater than, then insert 2 elements.
 */
multimap_init_ex(pmmap_m2, fun_greater_int);
pair_make(ppair_p, 1, 10);
multimap_insert(pmmap_m2, ppair_p);
pair_make(ppair_p, 2, 20);
multimap_insert(pmmap_m2, ppair_p);

/* Create a copy, multimap m3, of multimap m1 */
multimap_init_copy(pmmap_m3, pmmap_m1);

/* Create a multimap m4 by copying the range m1[first, last) */
multimap_init_copy_range(pmmap_m4, multimap_begin(pmmap_m1),
    iterator_advance(multimap_begin(pmmap_m1), 2));

/*
 * Create a multimap m5 by copying the range m3[first, last)
 * and with the key comparison function less than.
 */
multimap_init_copy_range_ex(pmmap_m5, multimap_begin(pmmap_m3),
    iterator_next(multimap_begin(pmmap_m3)), fun_less_int);

printf("m1 =");
for(it_m = multimap_begin(pmmap_m1);
    !iterator_equal(it_m, multimap_end(pmmap_m1));
    it_m = iterator_next(it_m))
{
    printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
}
printf("\n");

printf("m2 =");
for(it_m = multimap_begin(pmmap_m2);
    !iterator_equal(it_m, multimap_end(pmmap_m2));
    it_m = iterator_next(it_m))
{
    printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
}
printf("\n");

printf("m3 =");
for(it_m = multimap_begin(pmmap_m3);
    !iterator_equal(it_m, multimap_end(pmmap_m3));
    it_m = iterator_next(it_m))
```

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

    printf("m4 =");
    for(it_m = multimap_begin(pmmap_m4);
        !iterator_equal(it_m, multimap_end(pmmap_m4));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    printf("m5 =");
    for(it_m = multimap_begin(pmmap_m5);
        !iterator_equal(it_m, multimap_end(pmmap_m5));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    multimap_destroy(pmmap_m0);
    multimap_destroy(pmmap_m1);
    multimap_destroy(pmmap_m2);
    multimap_destroy(pmmap_m3);
    multimap_destroy(pmmap_m4);
    multimap_destroy(pmmap_m5);
    pair_destroy(ppair_p);

    return 0;
}
```

- **Output**

```
m1 = 10 20 30 40
m2 = 20 10
m3 = 10 20 30 40
m4 = 10 20
m5 = 10
```

18. multimap_insert multimap_insert_hint multimap_insert_range

向 multimap_t 中插入数据。

```
multimap_iterator_t multimap_insert(
    multimap_t* pmmap_multimap,
    const pair_t* cppair_pair
);

multimap_iterator_t multimap_insert_hint(
    multimap_t* pmmap_multimap,
    multimap_iterator_t it_hint,
    const pair_t* cppair_pair
);

void multimap_insert_range(
```

```
    multimap_t* pmmap_multimap,
    multimap_iterator_t it_begin,
    multimap_iterator_t it_end
);
```

- **Parameters**

  pmmap_multimap: 指向 multimap_t 类型的指针。
  cppair_pair: 插入的数据。
  it_hint: 被插入数据的提示位置。
  it_begin: 被插入的数据区间的开始位置。
  it_end: 被插入的数据区间的末尾位置。

- **Remarks**

第一个函数向 multimap_t 中插入一个指定的数据，成功后返回指向该数据的迭代器，如果 multimap_t 中包含了该数据那么插入失败，返回 multimap_end()。

第二个函数向 multimap_t 中插入一个指定的数据，同时给出一个该数据被插入后的提示位置迭代器，如果这个位置符合 multimap_t 的排序规则就把这个数据放在提示位置中成功后返回指向该数据的迭代器，如果提示位置不正确则忽略提示位置，当数据插入成功后返回数据的实际位置迭代器，否则返回 multimap_end()。

第三个函数插入指定的数据区间。

上面的函数要求迭代器和数据区间是有效的，无效的迭代器或数据区间导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multimap_t* pmmap_m1 = create_multimap(int, int);
    multimap_t* pmmap_m2 = create_multimap(int, int);
    pair_t* ppair_p = create_pair(int, int);
    multimap_iterator_t it_m;

    if(pmmap_m1 == NULL || pmmap_m2 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    multimap_init(pmmap_m1);
    multimap_init(pmmap_m2);

    pair_make(ppair_p, 1, 10);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 2, 20);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 3, 30);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 4, 40);
```

```
    multimap_insert(pmmap_m1, ppair_p);

    printf("The original key values of m1 =");
    for(it_m = multimap_begin(pmmap_m1);
        !iterator_equal(it_m, multimap_end(pmmap_m1));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_first(iterator_get_pointer(it_m)));
    }
    printf("\n");
    printf("The original multimapped values of m1 =");
    for(it_m = multimap_begin(pmmap_m1);
        !iterator_equal(it_m, multimap_end(pmmap_m1));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    pair_make(ppair_p, 1, 10);
    it_m = multimap_insert(pmmap_m1, ppair_p);
    if(!iterator_equal(it_m, multimap_end(pmmap_m1)))
    {
        printf("The element 10 was inserted in m1 successfully.\n");
    }
    else
    {
        printf("The number 1 already exists in m1.\n");
    }

    /* The hint version of insert */
    pair_make(ppair_p, 5, 50);
    multimap_insert_hint(pmmap_m1, iterator_prev(multimap_end(pmmap_m1)), ppair_p);
    printf("After the insertions, the key values of m1 =");
    for(it_m = multimap_begin(pmmap_m1);
        !iterator_equal(it_m, multimap_end(pmmap_m1));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_first(iterator_get_pointer(it_m)));
    }
    printf("\n");
    printf("and multimapped values of m1 =");
    for(it_m = multimap_begin(pmmap_m1);
        !iterator_equal(it_m, multimap_end(pmmap_m1));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    pair_make(ppair_p, 10, 100);
    multimap_insert(pmmap_m2, ppair_p);
    /* The templatized version inserting a range */
    multimap_insert_range(pmmap_m2, iterator_next(multimap_begin(pmmap_m1)),
        iterator_prev(multimap_end(pmmap_m1)));
    printf("After the insertions, the key values of m2 =");
    for(it_m = multimap_begin(pmmap_m2);
        !iterator_equal(it_m, multimap_end(pmmap_m2));
        it_m = iterator_next(it_m))
    {
```

```
        printf(" %d", *(int*)pair_first(iterator_get_pointer(it_m)));
    }
    printf("\n");
    printf("and multimapped values of m2 =");
    for(it_m = multimap_begin(pmmap_m2);
        !iterator_equal(it_m, multimap_end(pmmap_m2));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    pair_destroy(ppair_p);
    multimap_destroy(pmmap_m1);
    multimap_destroy(pmmap_m2);

    return 0;
}
```

- **Output**

```
The original key values of m1 = 1 2 3 4
The original multimapped values of m1 = 10 20 30 40
The element 10 was inserted in m1 successfully.
After the insertions, the key values of m1 = 1 1 2 3 4 5
and multimapped values of m1 = 10 10 20 30 40 50
After the insertions, the key values of m2 = 1 2 3 4 10
and multimapped values of m2 = 10 20 30 40 100
```

19. multimap_key_comp

返回 multimap_t 使用的键比较规则。

```
binary_function_t multimap_key_comp(
    const multimap_t* cpmmap_multimap
);
```

- **Parameters**

cpmmap_multimap: 指向 multimap_t 类型的指针。

- **Remarks**

这个排序规则是针对与数据中的键进行排序。

- **Requirements**

头文件 <cstl/cmap.h>

- **Example**

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

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

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

```
{
    multimap_t* pmmap_m1 = create_multimap(int, int);
    multimap_t* pmmap_m2 = create_multimap(int, int);
    binary_function_t bfun_kc = NULL;
    int n_element1 = 2;
    int n_element2 = 3;
    bool_t b_result = false;

    if(pmmap_m1 == NULL || pmmap_m2 == NULL)
    {
        return -1;
    }

    multimap_init_ex(pmmap_m1, fun_less_int);

    bfun_kc = multimap_key_comp(pmmap_m1);
    (*bfun_kc)(&n_element1, &n_element2, &b_result);
    if(b_result)
    {
        printf("(*bfun_kc)(2, 3) returns value of true, "
               "where bfun_kc is the function of m1.\n");
    }
    else
    {
        printf("(*bfun_kc)(2, 3) returns value of false, "
               "where bfun_kc is the function of m1.\n");
    }

    multimap_destroy(pmmap_m1);

    multimap_init_ex(pmmap_m2, fun_greater_int);

    bfun_kc = multimap_key_comp(pmmap_m2);
    (*bfun_kc)(&n_element1, &n_element2, &b_result);
    if(b_result)
    {
        printf("(*bfun_kc)(2, 3) returns value of true, "
               "where bfun_kc is the function of m2.\n");
    }
    else
    {
        printf("(*bfun_kc)(2, 3) returns value of false, "
               "where bfun_kc is the function of m2.\n");
    }

    multimap_destroy(pmmap_m2);

    return 0;
}
```

- **Output**

```
(*bfun_kc)(2, 3) returns value of true, where bfun_kc is the function of m1.
(*bfun_kc)(2, 3) returns value of false, where bfun_kc is the function of m2.
```

20. multimap_less

测试第一个 multimap_t 是否小于第二个 multimap_t。

```
bool_t multimap_less(
```

```
    const multimap_t* cpmmap_first,
    const multimap_t* cpmmap_second
);
```

● **Parameters**

cpmmap_first: 指向第一个 multimap_t 类型的指针。
cpmmap_second: 指向第二个 multimap_t 类型的指针。

● **Remarks**

这个函数要求两个 multimap_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

● **Requirements**

头文件 <cstl/cmap.h>

● **Example**

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

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

int main(int argc, char* argv[])
{
    multimap_t* pmmap_m1 = create_multimap(int, int);
    multimap_t* pmmap_m2 = create_multimap(int, int);
    multimap_t* pmmap_m3 = create_multimap(int, int);
    pair_t* ppair_p = create_pair(int, int);
    int i = 0;

    if(pmmap_m1 == NULL || pmmap_m2 == NULL || pmmap_m3 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    multimap_init(pmmap_m1);
    multimap_init(pmmap_m2);
    multimap_init(pmmap_m3);
    pair_init(ppair_p);

    for(i = 0; i < 3; ++i)
    {
        pair_make(ppair_p, i, i);
        multimap_insert(pmmap_m1, ppair_p);
        pair_make(ppair_p, i, i * i);
        multimap_insert(pmmap_m2, ppair_p);
        pair_make(ppair_p, i, i - 1);
        multimap_insert(pmmap_m3, ppair_p);
    }

    if(multimap_less(pmmap_m1, pmmap_m2))
    {
        printf("The multimap m1 is less than the multimap m2.\n");
    }
    else
    {
        printf("The multimap m1 is not less than the multimap m2.\n");
```

```
    }

    if(multimap_less(pmmap_m1, pmmap_m3))
    {
        printf("The multimap m1 is less than the multimap m3.\n");
    }
    else
    {
        printf("The multimap m1 is not less than the multimap m3.\n");
    }

    multimap_destroy(pmmap_m1);
    multimap_destroy(pmmap_m2);
    multimap_destroy(pmmap_m3);
    pair_destroy(ppair_p);

    return 0;
}
```

- **Output**

```
The multimap m1 is less than the multimap m2.
The multimap m1 is not less than the multimap m3.
```

21. multimap_less_equal

测试第一个 multimap_t 是否小于等于第二个 multimap_t。

```
bool_t multimap_less_equal(
    const multimap_t* cpmmap_first,
    const multimap_t* cpmmap_second
);
```

- **Parameters**

  cpmmap_first: 指向第一个 multimap_t 类型的指针。
  cpmmap_second: 指向第二个 multimap_t 类型的指针。

- **Remarks**

  这个函数要求两个 multimap_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multimap_t* pmmap_m1 = create_multimap(int, int);
    multimap_t* pmmap_m2 = create_multimap(int, int);
    multimap_t* pmmap_m3 = create_multimap(int, int);
```

```
    multimap_t* pmmap_m4 = create_multimap(int, int);
    pair_t* ppair_p = create_pair(int, int);
    int i = 0;

    if(pmmap_m1 == NULL || pmmap_m2 == NULL || pmmap_m3 == NULL ||
       pmmap_m4 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    multimap_init(pmmap_m1);
    multimap_init(pmmap_m2);
    multimap_init(pmmap_m3);
    multimap_init(pmmap_m4);
    pair_init(ppair_p);

    for(i = 0; i < 3; ++i)
    {
        pair_make(ppair_p, i, i);
        multimap_insert(pmmap_m1, ppair_p);
        multimap_insert(pmmap_m4, ppair_p);
        pair_make(ppair_p, i, i * i);
        multimap_insert(pmmap_m2, ppair_p);
        pair_make(ppair_p, i, i - 1);
        multimap_insert(pmmap_m3, ppair_p);
    }

    if(multimap_less_equal(pmmap_m1, pmmap_m2))
    {
        printf("The multimap m1 is less than or equal to the multimap m2.\n");
    }
    else
    {
        printf("The multimap m1 is greater than the multimap m2.\n");
    }

    if(multimap_less_equal(pmmap_m1, pmmap_m3))
    {
        printf("The multimap m1 is less than or equal to the multimap m3.\n");
    }
    else
    {
        printf("The multimap m1 is greater than the multimap m3.\n");
    }

    if(multimap_less_equal(pmmap_m1, pmmap_m4))
    {
        printf("The multimap m1 is less than or equal to the multimap m4.\n");
    }
    else
    {
        printf("The multimap m1 is greater than the multimap m4.\n");
    }

    multimap_destroy(pmmap_m1);
    multimap_destroy(pmmap_m2);
    multimap_destroy(pmmap_m3);
    multimap_destroy(pmmap_m4);
    pair_destroy(ppair_p);
```

```
    return 0;
}
```

- **Output**

```
The multimap m1 is less than or equal to the multimap m2.
The multimap m1 is greater than the multimap m3.
The multimap m1 is less than or equal to the multimap m4.
```

22. multimap_lower_bound

返回 multimap_t 中包含指定键的第一个数据的迭代器。

```
multimap_iterator_t multimap_lower_bound(
    const multimap_t* cpmmap_multimap,
    key
);
```

- **Parameters**

cpmmap_map: 指向 multimap_t 类型的指针。
key: 指定的键。

- **Remarks**

如果 multimap_t 中不包含拥有指定键的数据则返回 multimap_t 中指向包含大于指定键的第一个数据的迭代器。如果指定的键是 multimap_t 中最大的键则返回 multimap_end()。

- **Requirements**

头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multimap_t* pmmap_m1 = create_multimap(int, int);
    pair_t* ppair_p = create_pair(int, int);
    multimap_iterator_t it_m;

    if(pmmap_m1 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    multimap_init(pmmap_m1);

    pair_make(ppair_p, 1, 10);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 2, 20);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 3, 20);
```

```
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 3, 30);
    multimap_insert(pmmap_m1, ppair_p);

    it_m = multimap_lower_bound(pmmap_m1, 2);
    printf("The first element of multimap m1 with a key of 2 is: %d.\n",
        *(int*)pair_second(iterator_get_pointer(it_m)));

    /* If no match is found for this key, end() is returned */
    it_m = multimap_lower_bound(pmmap_m1, 4);
    if(iterator_equal(it_m, multimap_end(pmmap_m1)))
    {
        printf("The multimap m1 doesn't have an element with a key of 4.\n");
    }
    else
    {
        printf("The element of multimap m1 with key of 4 is: %d.\n",
            *(int*)pair_second(iterator_get_pointer(it_m)));
    }

    /*
     * The element at a specific location in the multimap can be found
     * using a dereferenced iterator addressing the location.
     */
    it_m = multimap_end(pmmap_m1);
    it_m = iterator_prev(it_m);
    it_m = multimap_lower_bound(pmmap_m1,
        *(int*)pair_first(iterator_get_pointer(it_m)));
    printf("The element of m1 with a key matching "
           "that of the last element is: %d.\n",
           *(int*)pair_second(iterator_get_pointer(it_m)));

    /*
     * Note that the first element with a key equal to
     * the key of the last element is not the last element
     */
    if(iterator_equal(it_m, iterator_prev(multimap_end(pmmap_m1))))
    {
        printf("This is the last element of multimap m1.\n");
    }
    else
    {
        printf("This is not the last element of multimap m1.\n");
    }

    pair_destroy(ppair_p);
    multimap_destroy(pmmap_m1);

    return 0;
}
```

- **Output**

```
The first element of multimap m1 with a key of 2 is: 20.
The multimap m1 doesn't have an element with a key of 4.
The element of m1 with a key matching that of the last element is: 20.
This is not the last element of multimap m1.
```

23. multimap_max_size

返回 multimap_t 中保存数据数量的最大可能值。

```
size_t multimap_max_size(
    const multimap_t* cpmmap_multimap
);
```

- **Parameters**

  cpmmap_multimap: 指向 multimap_t 类型的指针。

- **Remarks**

  这是一个与系统相关的常数。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multimap_t* pmmap_m1 = create_multimap(int, int);

    if(pmmap_m1 == NULL)
    {
        return -1;
    }

    multimap_init(pmmap_m1);

    printf("The maximum possible length of the multimap is %d.\n",
        multimap_max_size(pmmap_m1));
    printf("(Magnitude is machine specific.)\n");

    multimap_destroy(pmmap_m1);

    return 0;
}
```

- **Output**

```
The maximum possible length of the multimap is 7895160.
(Magnitude is machine specific.)
```

24. multimap_not_equal

测试两个 multimap_t 是否不等。

```
bool_t multimap_not_equal(
    const multimap_t* cpmmap_first,
    const multimap_t* cpmmap_second
```

```
);
```

- **Parameters**

  cpmmap_first: 指向第一个 multimap_t 类型的指针。
  cpmmap_second: 指向第二个 multimap_t 类型的指针。

- **Remarks**

如果两个 multimap_t 容器中的数据都对应相等，并且数据个数相等，则返回 false 否则返回 true，如果两个 multimap_t 容器中保存的数据类型不同也认为是不等。

- **Requirements**

头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multimap_t* pmmap_m1 = create_multimap(int, int);
    multimap_t* pmmap_m2 = create_multimap(int, int);
    multimap_t* pmmap_m3 = create_multimap(int, int);
    pair_t* ppair_p = create_pair(int, int);
    int i = 0;

    if(pmmap_m1 == NULL || pmmap_m2 == NULL || pmmap_m3 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    multimap_init(pmmap_m1);
    multimap_init(pmmap_m2);
    multimap_init(pmmap_m3);
    pair_init(ppair_p);

    for(i = 0; i < 3; ++i)
    {
        pair_make(ppair_p, i, i);
        multimap_insert(pmmap_m1, ppair_p);
        multimap_insert(pmmap_m3, ppair_p);
        pair_make(ppair_p, i, i * i);
        multimap_insert(pmmap_m2, ppair_p);
    }

    if(multimap_not_equal(pmmap_m1, pmmap_m2))
    {
        printf("The multimaps m1 and m2 are not equal.\n");
    }
    else
    {
        printf("The multimaps m1 and m2 are equal.\n");
    }
```

```
    if(multimap_not_equal(pmmap_m1, pmmap_m3))
    {
        printf("The multimaps m1 and m3 are not equal.\n");
    }
    else
    {
        printf("The multimaps m1 and m3 are equal.\n");
    }

    multimap_destroy(pmmap_m1);
    multimap_destroy(pmmap_m2);
    multimap_destroy(pmmap_m3);
    pair_destroy(ppair_p);

    return 0;
}
```

- **Output**

```
The multimaps m1 and m2 are not equal.
The multimaps m1 and m3 are equal.
```

25. multimap_size

返回 multimap_t 中数据的数量。

```
size_t multimap_size(
    const multimap_t* cpmmap_multimap
);
```

- **Parameters**

cpmmap_multimap: 指向 multimap_t 类型的指针。

- **Requirements**

头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multimap_t* pmmap_m1 = create_multimap(int, int);
    pair_t* ppair_p = create_pair(int, int);

    if(pmmap_m1 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    multimap_init(pmmap_m1);
```

```
    pair_make(ppair_p, 1, 1);
    multimap_insert(pmmap_m1, ppair_p);
    printf("The multimap length is %d.\n", multimap_size(pmmap_m1));

    pair_make(ppair_p, 2, 4);
    multimap_insert(pmmap_m1, ppair_p);
    printf("The multimap length is now %d.\n", multimap_size(pmmap_m1));

    pair_destroy(ppair_p);
    multimap_destroy(pmmap_m1);

    return 0;
}
```

- **Output**

```
The multimap length is 1.
The multimap length is now 2.
```

26. multimap_swap

交换两个 multimap_t 中的内容。

```
void multimap_swap(
    multimap_t* pmmap_first,
    multimap_t* pmmap_second
);
```

- **Parameters**

pmmap_first: 指向第一个 multimap_t 类型的指针。
pmmap_second: 指向第二个 multimap_t 类型的指针。

- **Remarks**

这个函数要求两个 multimap_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multimap_t* pmmap_m1 = create_multimap(int, int);
    multimap_t* pmmap_m2 = create_multimap(int, int);
    pair_t* ppair_p = create_pair(int, int);
    multimap_iterator_t it_m;

    if(pmmap_m1 == NULL || pmmap_m2 == NULL || ppair_p == NULL)
    {
        return -1;
```

```
    }

    pair_init(ppair_p);
    multimap_init(pmmap_m1);
    multimap_init(pmmap_m2);

    pair_make(ppair_p, 1, 10);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 2, 20);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 3, 30);
    multimap_insert(pmmap_m1, ppair_p);

    pair_make(ppair_p, 10, 100);
    multimap_insert(pmmap_m2, ppair_p);
    pair_make(ppair_p, 20, 200);
    multimap_insert(pmmap_m2, ppair_p);

    printf("The original multimap m1 is:");
    for(it_m = multimap_begin(pmmap_m1);
        !iterator_equal(it_m, multimap_end(pmmap_m1));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    multimap_swap(pmmap_m1, pmmap_m2);

    printf("After swapping with m2, multimap m1 is:");
    for(it_m = multimap_begin(pmmap_m1);
        !iterator_equal(it_m, multimap_end(pmmap_m1));
        it_m = iterator_next(it_m))
    {
        printf(" %d", *(int*)pair_second(iterator_get_pointer(it_m)));
    }
    printf("\n");

    pair_destroy(ppair_p);
    multimap_destroy(pmmap_m1);
    multimap_destroy(pmmap_m2);

    return 0;
}
```

- **Output**

```
The original multimap m1 is: 10 20 30
After swapping with m2, multimap m1 is: 100 200
```

27. multimap_upper_bound

返回 multimap_t 中包含大于指定键的第一个数据的迭代器。

```
multimap_iterator_t multimap_upper_bound(
    const multimap_t* cpmmap_multimap,
    key
);
```

- **Parameters**

  cpmmap_multimap: 指向 multimap_t 类型的指针。
  key: 指定的键。

- **Remarks**

  如果指定的键是 multimap_t 中最大的键则返回 multimap_end()。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    multimap_t* pmmap_m1 = create_multimap(int, int);
    pair_t* ppair_p = create_pair(int, int);
    multimap_iterator_t it_m;

    if(pmmap_m1 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    multimap_init(pmmap_m1);

    pair_make(ppair_p, 1, 10);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 2, 20);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 3, 30);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 3, 40);
    multimap_insert(pmmap_m1, ppair_p);

    it_m = multimap_upper_bound(pmmap_m1, 1);
    printf("The first element of multimap m1 with a key greater than 1 is: %d.\n",
        *(int*)pair_second(iterator_get_pointer(it_m)));

    it_m = multimap_upper_bound(pmmap_m1, 2);
    printf("The first element of multimap m1 with a key greater than 2 is: %d.\n",
        *(int*)pair_second(iterator_get_pointer(it_m)));

    /* If no match is found for the key, end is returned */
    it_m = multimap_upper_bound(pmmap_m1, 4);
    if(iterator_equal(it_m, multimap_end(pmmap_m1)))
    {
        printf("The multimap m1 doesn't have an "
               "element with a key greater than 4.\n");
    }
    else
    {
```

```
        printf("The element of multimap m1 with a key > 4 is: %d.\n",
            *(int*)pair_second(iterator_get_pointer(it_m)));
    }

    /*
     * The element at a specific location in the multimap can be found
     * using a dereferenced iterator addressing the location
     */
    it_m = multimap_begin(pmmap_m1);
    it_m = multimap_upper_bound(pmmap_m1,
        *(int*)pair_first(iterator_get_pointer(it_m)));
    printf("The first element of m1 with a key greater than"
           " that of the initial element of m1 is: %d.\n",
           *(int*)pair_second(iterator_get_pointer(it_m)));

    pair_destroy(ppair_p);
    multimap_destroy(pmmap_m1);

    return 0;
}
```

- **Output**

```
The first element of multimap m1 with a key greater than 1 is: 20.
The first element of multimap m1 with a key greater than 2 is: 30.
The multimap m1 doesn't have an element with a key greater than 4.
The first element of m1 with a key greater than that of the initial element of m1
is: 20.
```

28. multimap_value_comp

返回 multimap_t 中使用的数据的比较规则。

```
binary_function_t multimap_value_comp(
    const multimap_t* cpmmap_multimap
);
```

- **Parameters**

  cpmmap_multimap: 指向 multimap_t 类型的指针。

- **Remarks**

  这个规则是针对数据本身的比较规则而不是键或者值。

- **Requirements**

  头文件 <cstl/cmap.h>

- **Example**

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

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

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

```
{
    multimap_t* pmmap_m1 = create_multimap(int, int);
    pair_t* ppair_p = create_pair(int, int);
    binary_function_t bfun_vc = NULL;
    bool_t b_result = false;
    multimap_iterator_t it_m1;
    multimap_iterator_t it_m2;

    if(pmmap_m1 == NULL || ppair_p == NULL)
    {
        return -1;
    }

    pair_init(ppair_p);
    multimap_init_ex(pmmap_m1, fun_less_int);

    pair_make(ppair_p, 1, 10);
    multimap_insert(pmmap_m1, ppair_p);
    pair_make(ppair_p, 2, 5);
    multimap_insert(pmmap_m1, ppair_p);

    it_m1 = multimap_find(pmmap_m1, 1);
    it_m2 = multimap_find(pmmap_m1, 2);
    bfun_vc = multimap_value_comp(pmmap_m1);

    (*bfun_vc)(iterator_get_pointer(it_m1), iterator_get_pointer(it_m2), &b_result);
    if(b_result)
    {
        printf("The element (1, 10) precedes the element (2, 5).\n");
    }
    else
    {
        printf("The element (1, 10) does not precedes the element (2, 5).\n");
    }

    (*bfun_vc)(iterator_get_pointer(it_m2), iterator_get_pointer(it_m1), &b_result);
    if(b_result)
    {
        printf("The element (2, 5) precedes the element (1, 10).\n");
    }
    else
    {
        printf("The element (2, 5) does not precedes the element (1, 10).\n");
    }

    pair_destroy(ppair_p);
    multimap_destroy(pmmap_m1);

    return 0;
}
```

- **Output**

```
The element (1, 10) precedes the element (2, 5).
The element (2, 5) does not precedes the element (1, 10).
```

第九节　基于哈希结构的集合 hash_set_t

基于哈希结构的集合容器 hash_set_t 是关联容器，它使用指定的哈希函数计算数据的存储位置，将数据保存在这个位置上。hash_set_t 中的数据位置是根据数据本身计算的，并且保证数据在 hash_set_t 容器中的唯一性，所以在容器中数据也存在着某种有序性，所以也不能通过直接或者间接的方式修改容器中的数据。hash_set_t 提供双向迭代器，插入新的数据不会破坏原有的数据的迭代器，删除一个数据的时候只有指向数据本身的迭代器失效，但是当哈希表重新计算数据位置的时候所有的迭代器都失效。

- **Typedefs**

hash_set_t	基于哈希结构的集合容器类型。
hash_set_iterator_t	基于哈希结构的集合容器迭代器类型。

- **Operation Functions**

create_hash_set	创建基于哈希结构的集合容器类型
hash_set_assign	为基于哈希结构的集合容器赋值。
hash_set_begin	返回指向容器中第一个数据的迭代器。
hash_set_bucket_count	返回哈希表存储单元的数量。
hash_set_clear	删除容器中的所有数据。
hash_set_count	返回容器中指定数据的数量。
hash_set_destroy	销毁基于哈希结构的集合容器。
hash_set_empty	测试基于哈希结构的集合容器是否为空。
hash_set_end	返回指向基于哈希结构的集合容器末尾的迭代器。
hash_set_equal	测试两个基于哈希结构的集合容器是否相等。
hash_set_equal_range	返回容器中包含指定数据的数据区间。
hash_set_erase	删除容器中的指定数据。
hash_set_erase_pos	删除容器中指定位置的数据。
hash_set_erase_range	删除容器中指定数据区间的数据。
hash_set_find	在基于哈希结构的集合容器中查找指定的数据。
hash_set_greater	测试第一个基于哈希结构的集合容器是否大于第二个基于哈希结构的集合容器。
hash_set_greater_equal	测试第一个基于哈希结构的集合容器是否大于等于第二个基于哈希结构的集合容器。
hash_set_hash	返回基于哈希结构的集合容器使用的哈希函数。
hash_set_init	初始化一个空的基于哈希结构的集合容器。
hash_set_init_copy	使用一个基于哈希结构的集合容器初始化当前容器。
hash_set_init_copy_range	使用指定的数据区间初始化基于哈希结构的集合容器。
hash_set_init_copy_range_ex	使用指定的数据区间，哈希函数和比较规则初始化基于哈希结构的集合容器。
hash_set_init_ex	使用指定的哈希函数和比较规则初始化一个空的基于哈希结构的集合容器。
hash_set_insert	向基于哈希结构的集合容器中插入指定的数据。
hash_set_insert_range	向基于哈希结构的集合容器中插入指定的数据区间。
hash_set_key_comp	返回基于哈希结构的集合容器使用的键比较规则。
hash_set_less	测试第一个基于哈希结构的集合容器是否小于第二个基于哈希结构的集合容器。

hash_set_less_equal	测试第一个基于哈希结构的集合容器是否小于等于第二个基于哈希结构的集合容器。
hash_set_max_size	返回基于哈希结构的集合容器保存数据数量的最大可能值。
hash_set_not_equal	测试两个基于哈希结构的集合容器是否不等。
hash_set_resize	重新设置哈希表存储单元的数量。
hash_set_size	返回基于哈希结构的集合容器中数据的数量。
hash_set_swap	交换两个基于哈希结构的集合容器中的内容。
hash_set_value_comp	返回基于哈希结构的集合容器中使用的数据比较规则。

1. hash_set_t

基于哈希结构的集合容器类型。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

  请参考 hash_set_t 类型的其他操作函数。

2. hash_set_iterator_t

基于哈希结构的集合容器的迭代器类型。

- **Remarks**

  hash_set_iterator_t 是双向迭代器类型，不能通过迭代器来修改容器中数据的数据。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

  请参考 hash_set_t 类型的其他操作函数。

3. create_hash_set

创建 hash_set_t 容器类型。

```
hash_set_t* create_hash_set(
    type
);
```

- **Parameters**

  type: 数据类型描述。

- **Remarks**

  函数成功返回指向 hash_set_t 类型的指针，失败返回 NULL。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

  请参考 hash_set_t 类型的其他操作函数。

4. hash_set_assign

为 hash_set_t 容器类型赋值。

```
void hash_set_assign(
    hash_set_t* phset_dest,
    const hash_set_t* cphset_src
);
```

- **Parameters**

  phset_dest: 指向被赋值的 hash_set_t 类型的指针。
  cphset_src: 指向赋值的 hash_set_t 类型的指针。

- **Remarks**

  要求两个 hash_set_t 类型保存的数据具有相同的类型，否则函数的行为未定义。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);
    hash_set_t* phset_hs2 = create_hash_set(int);
    hash_set_iterator_t it_hs;

    if(phset_hs1 == NULL || phset_hs2 == NULL)
    {
        return -1;
    }

    hash_set_init(phset_hs1);
    hash_set_init(phset_hs2);

    hash_set_insert(phset_hs1, 10);
    hash_set_insert(phset_hs1, 20);
    hash_set_insert(phset_hs1, 30);
    hash_set_insert(phset_hs2, 40);
    hash_set_insert(phset_hs2, 50);
    hash_set_insert(phset_hs2, 60);

    printf("hs1 =");
    for(it_hs = hash_set_begin(phset_hs1);
        !iterator_equal(it_hs, hash_set_end(phset_hs1));
        it_hs = iterator_next(it_hs))
    {
```

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

    hash_set_assign(phset_hs1, phset_hs2);
    printf("hs1 =");
    for(it_hs = hash_set_begin(phset_hs1);
        !iterator_equal(it_hs, hash_set_end(phset_hs1));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    hash_set_destroy(phset_hs1);
    hash_set_destroy(phset_hs2);

    return 0;
}
```

- **Output**

```
hs1 = 10 20 30
hs1 = 60 40 50
```

5. hash_set_begin

返回指向 hash_set_t 中第一个数据的迭代器。

```
hash_set_iterator_t hash_set_begin(
    const hash_set_t* cphset_hset
);
```

- **Parameters**

cphset_hset: 指向 hash_set_t 类型的指针。

- **Remarks**

如果 hash_set_t 为空，这个函数的返回值和 hash_set_end()的返回值相等。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);

    if(phset_hs1 == NULL)
    {
```

```
        return -1;
    }

    hash_set_init(phset_hs1);

    hash_set_insert(phset_hs1, 1);
    hash_set_insert(phset_hs1, 2);
    hash_set_insert(phset_hs1, 3);

    printf("The first element of hs1 is %d.\n",
        *(int*)iterator_get_pointer(hash_set_begin(phset_hs1)));

    hash_set_erase_pos(phset_hs1, hash_set_begin(phset_hs1));

    printf("The first element of hs1 is now %d.\n",
        *(int*)iterator_get_pointer(hash_set_begin(phset_hs1)));

    hash_set_destroy(phset_hs1);

    return 0;
}
```

- **Output**

```
The first element of hs1 is 1.
The first element of hs1 is now 2.
```

6. hash_set_bucket_count

返回 hash_set_t 中的哈希表的存储单元个数。

```
size_t hash_set_bucket_count(
    const hash_set_t* cphset_hset
);
```

- **Parameters**

  cphset_hset: 指向 hash_set_t 类型的指针。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);
    hash_set_t* phset_hs2 = create_hash_set(int);

    if(phset_hs1 == NULL || phset_hs2 == NULL)
    {
        return -1;
```

```
    }

    hash_set_init(phset_hs1);
    hash_set_init_ex(phset_hs2, 100, NULL, NULL);

    printf("The default bucket count of hs1 is %d.\n",
        hash_set_bucket_count(phset_hs1));
    printf("The custom bucket count of hs2 is %d.\n",
        hash_set_bucket_count(phset_hs2));

    hash_set_destroy(phset_hs1);
    hash_set_destroy(phset_hs2);

    return 0;
}
```

- **Output**

```
The default bucket count of hs1 is 53.
The custom bucket count of hs2 is 193.
```

7. hash_set_clear

删除 hash_set_t 中的所有数据。

```
void hash_set_clear(
    hash_set_t* phset_hset
);
```

- **Parameters**

  phset_hset: 指向 hash_set_t 类型的指针。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);

    if(phset_hs1 == NULL)
    {
        return -1;
    }

    hash_set_init(phset_hs1);

    hash_set_insert(phset_hs1, 1);
    hash_set_insert(phset_hs1, 2);
```

```
    printf("The size of the hash_set is initially %d.\n",
        hash_set_size(phset_hs1));

    hash_set_clear(phset_hs1);

    printf("The size of the hash_set after clearing is %d.\n",
        hash_set_size(phset_hs1));

    hash_set_destroy(phset_hs1);

    return 0;
}
```

- **Output**

```
The size of the hash_set is initially 2.
The size of the hash_set after clearing is 0.
```

8. hash_set_count

返回 hash_set_t 中指定数据的数量。

```
size_t hash_set_count(
    const hash_set_t* cphset_hset,
    element
);
```

- **Parameters**

cphset_hset: 指向 hash_set_t 类型的指针。
element: 指定的数据。

- **Remarks**

如果容器中不包含指定数据则返回 0，包含则返回指定数据的个数，hash_set_t 中返回的都是 1。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);

    if(phset_hs1 == NULL)
    {
        return -1;
    }

    hash_set_init(phset_hs1);
```

```
    hash_set_insert(phset_hs1, 1);
    hash_set_insert(phset_hs1, 1);

    /* Keys must be unique in hash_set, so duplicates are ignored */
    printf("The number of elements in hs1 with a sort key of 1 is: %d.\n",
        hash_set_count(phset_hs1, 1));

    printf("The number of elements in hs1 with a sort key of 2 is: %d.\n",
        hash_set_count(phset_hs1, 2));

    hash_set_destroy(phset_hs1);

    return 0;
}
```

- **Output**

```
The number of elements in hs1 with a sort key of 1 is: 1.
The number of elements in hs1 with a sort key of 2 is: 0.
```

9. hash_set_destroy

销毁 hash_set_t 容器类型。

```
void hash_set_destroy(
    hash_set_t* phset_hset
);
```

- **Parameters**

phset_hset: 指向 hash_set_t 类型的指针。

- **Remarks**

hash_set_t 容器使用之后要销毁，否则 hash_set_t 占用的资源不会被释放。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

请参考 hash_set_t 类型的其他操作函数。

10. hash_set_empty

测试 hash_set_t 是否为空。

```
bool_t hash_set_empty(
    const hash_set_t* cphset_hset
);
```

- **Parameters**

cphset_hset: 指向 hash_set_t 类型的指针。

- **Remarks**

hash_set_t 容器为空则返回 true，否则返回 false。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);
    hash_set_t* phset_hs2 = create_hash_set(int);

    if(phset_hs1 == NULL || phset_hs2 == NULL)
    {
        return -1;
    }

    hash_set_init(phset_hs1);
    hash_set_init(phset_hs2);

    hash_set_insert(phset_hs1, 1);

    if(hash_set_empty(phset_hs1))
    {
        printf("The hash_set hs1 is empty.\n");
    }
    else
    {
        printf("The hash_set hs1 is not empty.\n");
    }

    if(hash_set_empty(phset_hs2))
    {
        printf("The hash_set hs2 is empty.\n");
    }
    else
    {
        printf("The hash_set hs2 is not empty.\n");
    }

    hash_set_destroy(phset_hs1);
    hash_set_destroy(phset_hs2);

    return 0;
}
```

- **Output**

```
The hash_set hs1 is not empty.
The hash_set hs2 is empty.
```

11. hash_set_end

返回指向 hash_set_t 容器末尾的迭代器。

```
hash_set_iterator_t hash_set_end(
     const hash_set_t* cphset_hset
);
```

- **Parameters**

  cphset_hset: 指向 hash_set_t 类型的指针。

- **Remarks**

  如果 hash_set_t 为空，这个函数的返回值和 hash_set_begin()的返回值相等。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);
    hash_set_iterator_t it_hs;

    if(phset_hs1 == NULL)
    {
        return -1;
    }

    hash_set_init(phset_hs1);

    hash_set_insert(phset_hs1, 1);
    hash_set_insert(phset_hs1, 2);
    hash_set_insert(phset_hs1, 3);

    it_hs = hash_set_end(phset_hs1);
    it_hs = iterator_prev(it_hs);
    printf("The last element of hs1 is %d.\n",
        *(int*)iterator_get_pointer(it_hs));

    hash_set_erase_pos(phset_hs1, it_hs);

    it_hs = hash_set_end(phset_hs1);
    it_hs = iterator_prev(it_hs);
    printf("The last element of hs1 is now %d.\n",
        *(int*)iterator_get_pointer(it_hs));

    hash_set_destroy(phset_hs1);

    return 0;
}
```

- **Output**

```
The last element of hs1 is 3.
```

```
The last element of hs1 is now 2.
```

12. hash_set_equal

测试两个 hash_set_t 是否相等。

```
bool_t hash_set_equal(
    const hash_set_t* cphset_first,
    const hash_set_t* cphset_second
);
```

- **Parameters**

cphset_first: 指向第一个 hash_set_t 类型的指针。
cphset_second: 指向第二个 hash_set_t 类型的指针。

- **Remarks**

两个 hash_set_t 中的数据对应相等，并且数量相等，函数返回 true，否则返回 false。如果两个 hash_set_t 中的数据类型不同也认为不等。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);
    hash_set_t* phset_hs2 = create_hash_set(int);
    hash_set_t* phset_hs3 = create_hash_set(int);
    int i = 0;

    if(phset_hs1 == NULL || phset_hs2 == NULL || phset_hs3 == NULL)
    {
        return -1;
    }

    hash_set_init(phset_hs1);
    hash_set_init(phset_hs2);
    hash_set_init(phset_hs3);

    for(i = 0; i < 3; ++i)
    {
        hash_set_insert(phset_hs1, i);
        hash_set_insert(phset_hs2, i * i);
        hash_set_insert(phset_hs3, i);
    }

    if(hash_set_equal(phset_hs1, phset_hs2))
    {
        printf("The hash_sets hs1 and hs2 are equal.\n");
```

```
    }
    else
    {
        printf("The hash_sets hs1 and hs2 are not equal.\n");
    }

    if(hash_set_equal(phset_hs1, phset_hs3))
    {
        printf("The hash_sets hs1 and hs3 are equal.\n");
    }
    else
    {
        printf("The hash_sets hs1 and hs3 are not equal.\n");
    }

    hash_set_destroy(phset_hs1);
    hash_set_destroy(phset_hs2);
    hash_set_destroy(phset_hs3);

    return 0;
}
```

- **Output**

```
The hash_sets hs1 and hs2 are not equal.
The hash_sets hs1 and hs3 are equal.
```

13. hash_set_equal_range

返回 hash_set_t 中包含指定数据的数据区间。

```
range_t hash_set_equal_range(
    const hash_set_t* cphset_hset,
    element
);
```

- **Parameters**

  cphset_hset: 指向 hash_set_t 类型的指针。
  element: 指定的数据。

- **Remarks**

返回 hash_set_t 中包含指定数据的数据区间[range_t.it_begin, range_t.it_end)，其中 it_begin 是指向等于指定数据的第一个数据的迭代器，it_end 指向的是大于指定数据的第一个数据的迭代器。如果 hash_set_t 中不包含指定数据则 it_begin 与 it_end 相等。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

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

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

```
int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);
    range_t r_r;

    if(phset_hs1 == NULL)
    {
        return -1;
    }

    hash_set_init(phset_hs1);

    hash_set_insert(phset_hs1, 10);
    hash_set_insert(phset_hs1, 20);
    hash_set_insert(phset_hs1, 30);

    r_r = hash_set_equal_range(phset_hs1, 20);
    printf("The upper bound of the element with "
           "a key of 20 in the hash_set hs1 is: %d.\n",
           *(int*)iterator_get_pointer(r_r.it_end));
    printf("The lower bound of the element with "
           "a key of 20 in the hash_set hs1 is: %d.\n",
           *(int*)iterator_get_pointer(r_r.it_begin));

    /*
     * If no match is bound for the key,
     * bouth element of the range returned end().
     */
    r_r = hash_set_equal_range(phset_hs1, 40);
    if(iterator_equal(r_r.it_begin, hash_set_end(phset_hs1)) &&
       iterator_equal(r_r.it_end, hash_set_end(phset_hs1)))
    {
        printf("The hash_set hs1 doesn't have "
               "an element with a key less than 40.\n");
    }
    else
    {
        printf("The element of hash_set hs1 with a key >= 40 is: %d.\n",
            *(int*)iterator_get_pointer(r_r.it_begin));
    }

    hash_set_destroy(phset_hs1);

    return 0;
}
```

- **Output**

```
The upper bound of the element with a key of 20 in the hash_set hs1 is: 30.
The lower bound of the element with a key of 20 in the hash_set hs1 is: 20.
The hash_set hs1 doesn't have an element with a key less than 40.
```

14. hash_set_erase　hash_set_erase_pos　hash_set_erase_range

删除 hash_set_t 中的数据。

```
size_t hash_set_erase(
    hash_set_t* phset_hset,
```

```
    element
);

void hash_set_erase_pos(
    hash_set_t* phset_hset,
    hash_set_iterator_t it_pos
);

void hash_set_erase_range(
    hash_set_t* phset_hset,
    hash_set_iterator_t it_begin,
    hash_set_iterator_t it_end
);
```

- **Parameters**

  phset_hset: 指向 hash_set_t 类型的指针。
  element: 要删除的数据。
  it_pos: 要删除的数据的位置迭代器。
  it_begin: 要删除的数据区间的开始位置。
  it_end: 要删除的数据区间的末尾位置。

- **Remarks**

  第一个函数删除 hash_set_t 中指定的数据，并返回删除的个数，如果 hash_set_t 中不包含指定的数据就返回 0。
  第二个函数删除指定位置的数据。
  第三个函数删除指定数据区间中的数据。
  后面两个函数要求迭代器和数据区间是有效的，无效的迭代器或数据区间导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);
    hash_set_t* phset_hs2 = create_hash_set(int);
    hash_set_t* phset_hs3 = create_hash_set(int);
    hash_set_iterator_t it_hs;
    size_t t_count = 0;
    int i = 0;

    if(phset_hs1 == NULL || phset_hs2 == NULL || phset_hs3 == NULL)
    {
        return -1;
    }

    hash_set_init(phset_hs1);
    hash_set_init(phset_hs2);
```

```
    hash_set_init(phset_hs3);

    for(i = 1; i < 5; ++i)
    {
        hash_set_insert(phset_hs1, i);
        hash_set_insert(phset_hs2, i * i);
        hash_set_insert(phset_hs3, i - 1);
    }

    /* The first function removes an element at a given position */
    it_hs = iterator_next(hash_set_begin(phset_hs1));
    hash_set_erase_pos(phset_hs1, it_hs);

    printf("After the second element is deleted, the hash_set hs1 is: ");
    for(it_hs = hash_set_begin(phset_hs1);
        !iterator_equal(it_hs, hash_set_end(phset_hs1));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    /* The second function removes elements in the range [first, last) */
    hash_set_erase_range(phset_hs2, iterator_next(hash_set_begin(phset_hs2)),
        iterator_prev(hash_set_end(phset_hs2)));

    printf("After the middle two elements are deleted, the hash_set hs2 is: ");
    for(it_hs = hash_set_begin(phset_hs2);
        !iterator_equal(it_hs, hash_set_end(phset_hs2));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    /* The third function removes elements with a given key */
    t_count = hash_set_erase(phset_hs3, 2);

    printf("After the element with a key of 2 is deleted, the hash_set hs3 is: ");
    for(it_hs = hash_set_begin(phset_hs3);
        !iterator_equal(it_hs, hash_set_end(phset_hs3));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    hash_set_destroy(phset_hs1);
    hash_set_destroy(phset_hs2);
    hash_set_destroy(phset_hs3);

    return 0;
}
```

- **Output**

```
After the second element is deleted, the hash_set hs1 is:  1 3 4
After the middle two elements are deleted, the hash_set hs2 is:  1 16
After the element with a key of 2 is deleted, the hash_set hs3 is:  0 1 3
```

15. hash_set_find

在 hash_set_t 中查找指定的数据。

```
hash_set_iterator_t hash_set_find(
    const hash_set_t* cphset_hset,
    element
);
```

- **Parameters**

  cphset_hset: 指向 hash_set_t 类型的指针。
  element: 指定的数据。

- **Remarks**

  如果 hash_set_t 中包含指定的数据则返回指向该数据的迭代器，否则返回 hash_set_end()。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);
    hash_set_iterator_t it_hs;

    if(phset_hs1 == NULL)
    {
        return -1;
    }

    hash_set_init(phset_hs1);

    hash_set_insert(phset_hs1, 10);
    hash_set_insert(phset_hs1, 20);
    hash_set_insert(phset_hs1, 30);

    it_hs = hash_set_find(phset_hs1, 20);
    printf("The element of hash_set hs1 with a key of 20 is: %d.\n",
        *(int*)iterator_get_pointer(it_hs));

    it_hs = hash_set_find(phset_hs1, 40);

    /* If no match is found for the key, end() is returned */
    if(iterator_equal(it_hs, hash_set_end(phset_hs1)))
    {
        printf("The hash_set hs1 doesn't have an element with a key of 40.\n");
    }
    else
    {
        printf("The element of hash_set hs1 with a key of 40 is: %d.\n",
```

```
            *(int*)iterator_get_pointer(it_hs));
    }

    /*
     * The element at a specific location in the hash_set can be found
     * by using a dereferenced iterator addressing the location.
     */
    it_hs = iterator_prev(hash_set_end(phset_hs1));
    it_hs = hash_set_find(phset_hs1, *(int*)iterator_get_pointer(it_hs));
    printf("The element of hs1 with a key "
           "matching that of the last element is: %d.\n",
           *(int*)iterator_get_pointer(it_hs));

    hash_set_destroy(phset_hs1);

    return 0;
}
```

- **Output**

```
The element of hash_set hs1 with a key of 20 is: 20.
The hash_set hs1 doesn't have an element with a key of 40.
The element of hs1 with a key matching that of the last element is: 30.
```

16. hash_set_greater

测试第一个 hash_set_t 是否大于第二个 hash_set_t。

```
bool_t hash_set_greater(
    const hash_set_t* cphset_first,
    const hash_set_t* cphset_second
);
```

- **Parameters**

cphset_first: 指向第一个 hash_set_t 类型的指针。
cphset_second: 指向第二个 hash_set_t 类型的指针。

- **Remarks**

这个函数要求两个 hash_set_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);
    hash_set_t* phset_hs2 = create_hash_set(int);
    hash_set_t* phset_hs3 = create_hash_set(int);
```

```
    int i = 0;

    if(phset_hs1 == NULL || phset_hs2 == NULL || phset_hs3 == NULL)
    {
        return -1;
    }

    hash_set_init(phset_hs1);
    hash_set_init(phset_hs2);
    hash_set_init(phset_hs3);

    for(i = 0; i < 3; ++i)
    {
        hash_set_insert(phset_hs1, i);
        hash_set_insert(phset_hs2, i * i);
        hash_set_insert(phset_hs3, i - 1);
    }

    if(hash_set_greater(phset_hs1, phset_hs2))
    {
        printf("The hash_set hs1 is greater than the hash_set hs2.\n");
    }
    else
    {
        printf("The hash_set hs1 is not greater than the hash_set hs2.\n");
    }

    if(hash_set_greater(phset_hs1, phset_hs3))
    {
        printf("The hash_set hs1 is greater than the hash_set hs3.\n");
    }
    else
    {
        printf("The hash_set hs1 is not greater than the hash_set hs3.\n");
    }

    hash_set_destroy(phset_hs1);
    hash_set_destroy(phset_hs2);
    hash_set_destroy(phset_hs3);

    return 0;
}
```

- **Output**

```
The hash_set hs1 is not greater than the hash_set hs2.
The hash_set hs1 is greater than the hash_set hs3.
```

17. hash_set_greater_equal

测试第一个 hash_set_t 是否大于等于第二个 hash_set_t。

```
bool_t hash_set_greater_equal(
    const hash_set_t* cphset_first,
    const hash_set_t* cphset_second
);
```

- **Parameters**

cphset_first: 指向第一个 hash_set_t 类型的指针。

cphset_second: 指向第二个 hash_set_t 类型的指针。

- **Remarks**

这个函数要求两个 hash_set_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);
    hash_set_t* phset_hs2 = create_hash_set(int);
    hash_set_t* phset_hs3 = create_hash_set(int);
    hash_set_t* phset_hs4 = create_hash_set(int);
    int i = 0;

    if(phset_hs1 == NULL || phset_hs2 == NULL ||
       phset_hs3 == NULL || phset_hs4 == NULL)
    {
        return -1;
    }

    hash_set_init(phset_hs1);
    hash_set_init(phset_hs2);
    hash_set_init(phset_hs3);
    hash_set_init(phset_hs4);

    for(i = 0; i < 3; ++i)
    {
        hash_set_insert(phset_hs1, i);
        hash_set_insert(phset_hs2, i * i);
        hash_set_insert(phset_hs3, i - 1);
        hash_set_insert(phset_hs4, i);
    }

    if(hash_set_greater_equal(phset_hs1, phset_hs2))
    {
        printf("The hash_set hs1 is greater than or equal to the hash_set hs2.\n");
    }
    else
    {
        printf("The hash_set hs1 is less than the hash_set hs2.\n");
    }

    if(hash_set_greater_equal(phset_hs1, phset_hs3))
    {
        printf("The hash_set hs1 is greater than or equal to the hash_set hs3.\n");
    }
    else
    {
```

```
        printf("The hash_set hs1 is less than the hash_set hs3.\n");
    }

    if(hash_set_greater_equal(phset_hs1, phset_hs4))
    {
        printf("The hash_set hs1 is greater than or equal to the hash_set hs4.\n");
    }
    else
    {
        printf("The hash_set hs1 is less than the hash_set hs4.\n");
    }

    hash_set_destroy(phset_hs1);
    hash_set_destroy(phset_hs2);
    hash_set_destroy(phset_hs3);
    hash_set_destroy(phset_hs4);

    return 0;
}
```

- **Output**

```
The hash_set hs1 is less than the hash_set hs2.
The hash_set hs1 is greater than or equal to the hash_set hs3.
The hash_set hs1 is greater than or equal to the hash_set hs4.
```

18. hash_set_hash

返回 hash_set_t 使用的哈希函数。

```
unary_function_t hash_set_hash(
    const hash_set_t* cphset_hset
);
```

- **Parameters**

cphset_hset: 指向 hash_set_t 类型的指针。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);
    hash_set_t* phset_hs2 = create_hash_set(int);

    if(phset_hs1 == NULL || phset_hs2 == NULL)
    {
```

```
        return -1;
    }

    hash_set_init(phset_hs1);
    hash_set_init_ex(phset_hs2, 100, hash_func, NULL);

    if(hash_set_hash(phset_hs1) == hash_func)
    {
        printf("The hash function of hash_set hs1 is hash_func.\n");
    }
    else
    {
        printf("The hash function of hash_set hs1 is not hash_func.\n");
    }

    if(hash_set_hash(phset_hs2) == hash_func)
    {
        printf("The hash function of hash_set hs2 is hash_func.\n");
    }
    else
    {
        printf("The hash function of hash_set hs2 is not hash_func.\n");
    }

    hash_set_destroy(phset_hs1);
    hash_set_destroy(phset_hs2);

    return 0;
}

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

- **Output**

```
The hash function of hash_set hs1 is not hash_func.
The hash function of hash_set hs2 is hash_func.
```

19. hash_set_init hash_set_init_copy hash_set_init_copy_range hash_set_init_copy_range_ex hash_set_init_ex

初始化 hash_set_t 容器类型。

```
void hash_set_init(
    hash_set_t* phset_hset
);

void hash_set_init_copy(
    hash_set_t* phset_hset,
    const hash_set_t* cphset_src
);

void hash_set_init_copy_range(
    hash_set_t* phset_hset,
    hash_set_iterator_t it_begin,
    hash_set_iterator_t it_end
```

```
);

void hash_set_init_copy_range_ex(
    hash_set_t* phset_hset,
    hash_set_iterator_t it_begin,
    hash_set_iterator_t it_end,
    size_t t_bucketcount,
    unary_function_t ufun_hash,
    binary_function_t bfun_compare
);

void hash_set_init_ex(
    hash_set_t* phset_hset,
    size_t t_bucketcount,
    unary_function_t ufun_hash,
    binary_function_t bfun_compare
);
```

- **Parameters**

phset_hset: 指向被初始化 hash_set_t 类型的指针。
cphset_src: 指向用于初始化的 hash_set_t 类型的指针。
it_begin: 用于初始化的数据区间的开始位置。
it_end: 用于初始化的数据区间的末尾位置。
t_bucketcount: 哈希表中的存储单元个数。
ufun_hash: 自定义的哈希函数。
bfun_compare: 自定义比较规则。

- **Remarks**

第一个函数初始化一个空的 hash_set_t，使用默认的哈希函数和与数据类型相关的小于操作函数作为默认的比较规则。

第二个函数使用一个源 hash_set_t 来初始化 hash_set_t，数据的内容，哈希函数和比较规则都从源 hash_set_t 复制。

第三个函数使用指定的数据区间初始化一个 hash_set_t，使用默认的哈希函数和与数据类型相关的小于操作函数作为默认的比较规则。

第四个函数使用指定的数据区间初始化一个 hash_set_t，使用用户指定的哈希表存储单元个数，哈希函数和比较规则。

第五个函数初始化一个空的 hash_set_t，使用用户指定的哈希表存储单元个数，哈希函数和比较规则。

上面的函数要求迭代器和数据区间是有效的，无效的迭代器或数据区间导致函数的行为未定义。初始化函数根据用户指定的哈希表存储单元个数计算一个与用户指定的个数最接近的最佳哈希表存储单元个数。默认个数是 53 个，用户指定的个数小于等于 53 时都使用这个存储单元个数。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

```
static void _hash_function(const void* cpv_input, void* pv_output)
{
    *(size_t*)pv_output = *(int*)cpv_input + 20;
}

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs0 = create_hash_set(int);
    hash_set_t* phset_hs1 = create_hash_set(int);
    hash_set_t* phset_hs2 = create_hash_set(int);
    hash_set_t* phset_hs3 = create_hash_set(int);
    hash_set_t* phset_hs4 = create_hash_set(int);
    hash_set_t* phset_hs5 = create_hash_set(int);
    hash_set_iterator_t it_hs;

    if(phset_hs0 == NULL || phset_hs1 == NULL || phset_hs2 == NULL ||
       phset_hs3 == NULL || phset_hs4 == NULL || phset_hs5 == NULL)
    {
        return -1;
    }

    /* Create an empty hash_set hs0 of key type integer */
    hash_set_init(phset_hs0);

    /*
     * Create an empty hash_set hs1 with the key comparison
     * function of less than, then insert 4 elements.
     */
    hash_set_init_ex(phset_hs1, 10, _hash_function, fun_less_int);
    hash_set_insert(phset_hs1, 10);
    hash_set_insert(phset_hs1, 20);
    hash_set_insert(phset_hs1, 30);
    hash_set_insert(phset_hs1, 40);

    /*
     * Create an empty hash_set hs2 with the key comparison
     * function of greater than, then insert 2 element.
     */
    hash_set_init_ex(phset_hs2, 100, _hash_function, fun_greater_int);
    hash_set_insert(phset_hs2, 10);
    hash_set_insert(phset_hs2, 20);

    /* Create a copy, hash_set hs3, of hash_set hs1 */
    hash_set_init_copy(phset_hs3, phset_hs1);

    /* Create a hash_set hs4 by copying the range hs1[first, last) */
    hash_set_init_copy_range(phset_hs4, hash_set_begin(phset_hs1),
        iterator_advance(hash_set_begin(phset_hs1), 2));

    /*
     * Create a hash_set hs5 by copying the range hs3[first, last)
     * and with the key comparison function of less than.
     */
    hash_set_init_copy_range_ex(phset_hs5, hash_set_begin(phset_hs3),
        iterator_next(hash_set_begin(phset_hs3)), 100,
        _hash_function, fun_less_int);

    printf("hs1 =");
    for(it_hs = hash_set_begin(phset_hs1);
```

```
        !iterator_equal(it_hs, hash_set_end(phset_hs1));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    printf("hs2 =");
    for(it_hs = hash_set_begin(phset_hs2);
        !iterator_equal(it_hs, hash_set_end(phset_hs2));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    printf("hs3 =");
    for(it_hs = hash_set_begin(phset_hs3);
        !iterator_equal(it_hs, hash_set_end(phset_hs3));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    printf("hs4 =");
    for(it_hs = hash_set_begin(phset_hs4);
        !iterator_equal(it_hs, hash_set_end(phset_hs4));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    printf("hs5 =");
    for(it_hs = hash_set_begin(phset_hs5);
        !iterator_equal(it_hs, hash_set_end(phset_hs5));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    hash_set_destroy(phset_hs0);
    hash_set_destroy(phset_hs1);
    hash_set_destroy(phset_hs2);
    hash_set_destroy(phset_hs3);
    hash_set_destroy(phset_hs4);
    hash_set_destroy(phset_hs5);

    return 0;
}
```

- **Output**

```
hs1 = 40 10 20 30
hs2 = 10 20
hs3 = 40 10 20 30
hs4 = 10 40
hs5 = 40
```

20. hash_set_insert　hash_set_insert_range

向 hash_set_t 中插入数据。

```
hash_set_iterator_t hash_set_insert(
    hash_set_t* phset_hset,
    element
);

void hash_set_insert_range(
    hash_set_t* phset_hset,
    hash_set_iterator_t it_begin,
    hash_set_iterator_t it_end
);
```

- **Parameters**

phset_hset: 指向 hash_set_t 类型的指针。
element: 插入的数据。
it_begin: 被插入的数据区间的开始位置。
it_end: 被插入的数据区间的末尾位置。

- **Remarks**

第一个函数向 hash_set_t 中插入一个指定的数据，成功后返回指向该数据的迭代器，如果 hash_set_t 中包含了该数据那么插入失败，返回 hash_set_end()。
第三个函数插入指定的数据区间。
上面的函数要求迭代器和数据区间是有效的，无效的迭代器或数据区间导致函数的行为未定义。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);
    hash_set_t* phset_hs2 = create_hash_set(int);
    hash_set_iterator_t it_hs;

    if(phset_hs1 == NULL || phset_hs2 == NULL)
    {
        return -1;
    }

    hash_set_init(phset_hs1);
    hash_set_init(phset_hs2);

    hash_set_insert(phset_hs1, 10);
    hash_set_insert(phset_hs1, 20);
```

```
    hash_set_insert(phset_hs1, 30);
    hash_set_insert(phset_hs1, 40);

    printf("The original hs1 =");
    for(it_hs = hash_set_begin(phset_hs1);
        !iterator_equal(it_hs, hash_set_end(phset_hs1));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    it_hs = hash_set_insert(phset_hs1, 10);
    if(iterator_equal(it_hs, hash_set_end(phset_hs1)))
    {
        printf("The element 10 already exist in hs1.\n");
    }
    else
    {
        printf("The element 10 was inserted inhs1 successfully.\n");
    }

    hash_set_insert(phset_hs1, 80);
    printf("After the insertions, hs1 =");
    for(it_hs = hash_set_begin(phset_hs1);
        !iterator_equal(it_hs, hash_set_end(phset_hs1));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    hash_set_insert(phset_hs2, 100);
    hash_set_insert_range(phset_hs2, iterator_next(hash_set_begin(phset_hs1)),
        iterator_prev(hash_set_end(phset_hs1)));

    printf("hs2 =");
    for(it_hs = hash_set_begin(phset_hs2);
        !iterator_equal(it_hs, hash_set_end(phset_hs2));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    hash_set_destroy(phset_hs1);
    hash_set_destroy(phset_hs2);

    return 0;
}
```

- **Output**

```
The original hs1 = 10 20 30 40
The element 10 already exist in hs1.
After the insertions, hs1 = 10 20 80 30 40
hs2 = 20 80 30 100
```

21. hash_set_key_comp

返回 hash_set_t 使用的比较规则。

```
binary_function_t hash_set_key_comp(
    const hash_set_t* cphset_hset
);
```

- **Parameters**

cphset_hset: 指向 hash_set_t 类型的指针。

- **Remarks**

由于 hash_set_t 中数据本身就是键，所以这个函数的返回值与 hash_set_value_comp()相同。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);
    hash_set_t* phset_hs2 = create_hash_set(int);
    binary_function_t bfun_kc = NULL;
    int n_first = 2;
    int n_second = 3;
    bool_t b_result = false;

    if(phset_hs1 == NULL || phset_hs2 == NULL)
    {
        return -1;
    }

    hash_set_init_ex(phset_hs1, 0, NULL, fun_less_int);
    hash_set_init_ex(phset_hs2, 0, NULL, fun_greater_int);

    bfun_kc = hash_set_key_comp(phset_hs1);
    (*bfun_kc)(&n_first, &n_second, &b_result);
    if(b_result)
    {
        printf("(*bfun_kc)(2, 3) returns value of true, "
               "where bfun_kc is the compare function of hs1.\n");
    }
    else
    {
        printf("(*bfun_kc)(2, 3) returns value of false, "
               "where bfun_kc is the compare function of hs1.\n");
    }

    bfun_kc = hash_set_key_comp(phset_hs2);
    (*bfun_kc)(&n_first, &n_second, &b_result);
```

```
    if(b_result)
    {
        printf("(*bfun_kc)(2, 3) returns value of true, "
               "where bfun_kc is the compare function of hs2.\n");
    }
    else
    {
        printf("(*bfun_kc)(2, 3) returns value of false, "
               "where bfun_kc is the compare function of hs2.\n");
    }

    hash_set_destroy(phset_hs1);
    hash_set_destroy(phset_hs2);

    return 0;
}
```

- **Output**

```
The original hs1 = 10 20 30 40
The element 10 already exist in hs1.
After the insertions, hs1 = 10 20 80 30 40
hs2 = 20 80 30 100
```

22. hash_set_less

测试第一个 hash_set_t 是否小于第二个 hash_set_t。

```
bool_t hash_set_less(
    const hash_set_t* cphset_first,
    const hash_set_t* cphset_second
);
```

- **Parameters**

  cphset_first: 指向第一个 hash_set_t 类型的指针。
  cphset_second: 指向第二个 hash_set_t 类型的指针。

- **Remarks**

  这个函数要求两个 hash_set_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);
    hash_set_t* phset_hs2 = create_hash_set(int);
    hash_set_t* phset_hs3 = create_hash_set(int);
```

```
    int i = 0;

    if(phset_hs1 == NULL || phset_hs2 == NULL || phset_hs3 == NULL)
    {
        return -1;
    }

    hash_set_init(phset_hs1);
    hash_set_init(phset_hs2);
    hash_set_init(phset_hs3);

    for(i = 0; i < 3; ++i)
    {
        hash_set_insert(phset_hs1, i);
        hash_set_insert(phset_hs2, i * i);
        hash_set_insert(phset_hs3, i - 1);
    }

    if(hash_set_less(phset_hs1, phset_hs2))
    {
        printf("The hash_set hs1 is less than the hash_set hs2.\n");
    }
    else
    {
        printf("The hash_set hs1 is not less than the hash_set hs2.\n");
    }

    if(hash_set_less(phset_hs1, phset_hs3))
    {
        printf("The hash_set hs1 is less than the hash_set hs3.\n");
    }
    else
    {
        printf("The hash_set hs1 is not less than the hash_set hs3.\n");
    }

    hash_set_destroy(phset_hs1);
    hash_set_destroy(phset_hs2);
    hash_set_destroy(phset_hs3);

    return 0;
}
```

- **Output**

```
The hash_set hs1 is less than the hash_set hs2.
The hash_set hs1 is not less than the hash_set hs3.
```

23. hash_set_less_equal

测试第一个 hash_set_t 是否小于等于第二个 hash_set_t。

```
bool_t hash_set_less_equal(
    const hash_set_t* cphset_first,
    const hash_set_t* cphset_second
);
```

- **Parameters**

cphset_first: 指向第一个 hash_set_t 类型的指针。

cphset_second: 指向第二个 hash_set_t 类型的指针。

- **Remarks**

  这个函数要求两个 hash_set_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);
    hash_set_t* phset_hs2 = create_hash_set(int);
    hash_set_t* phset_hs3 = create_hash_set(int);
    hash_set_t* phset_hs4 = create_hash_set(int);
    int i = 0;

    if(phset_hs1 == NULL || phset_hs2 == NULL ||
       phset_hs3 == NULL || phset_hs4 == NULL)
    {
        return -1;
    }

    hash_set_init(phset_hs1);
    hash_set_init(phset_hs2);
    hash_set_init(phset_hs3);
    hash_set_init(phset_hs4);

    for(i = 0; i < 3; ++i)
    {
        hash_set_insert(phset_hs1, i);
        hash_set_insert(phset_hs2, i * i);
        hash_set_insert(phset_hs3, i - 1);
        hash_set_insert(phset_hs4, i);
    }

    if(hash_set_less_equal(phset_hs1, phset_hs2))
    {
        printf("The hash_set hs1 is less than or equal to the hash_set hs2.\n");
    }
    else
    {
        printf("The hash_set hs1 is greater than the hash_set hs2.\n");
    }

    if(hash_set_less_equal(phset_hs1, phset_hs3))
    {
        printf("The hash_set hs1 is less than or equal to the hash_set hs3.\n");
    }
    else
    {
```

```
        printf("The hash_set hs1 is greater than the hash_set hs3.\n");
    }

    if(hash_set_less_equal(phset_hs1, phset_hs4))
    {
        printf("The hash_set hs1 is less than or equal to the hash_set hs4.\n");
    }
    else
    {
        printf("The hash_set hs1 is greater than the hash_set hs4.\n");
    }

    hash_set_destroy(phset_hs1);
    hash_set_destroy(phset_hs2);
    hash_set_destroy(phset_hs3);
    hash_set_destroy(phset_hs4);

    return 0;
}
```

- **Output**

```
The hash_set hs1 is less than or equal to the hash_set hs2.
The hash_set hs1 is greater than the hash_set hs3.
The hash_set hs1 is less than or equal to the hash_set hs4.
```

24. hash_set_max_size

返回 hash_set_t 中保存数据数量的最大可能值。

```
size_t hash_set_max_size(
    const hash_set_t* cphset_hset
);
```

- **Parameters**

cphset_hset: 指向 hash_set_t 类型的指针。

- **Remarks**

这是一个与系统有关的常数。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);

    if(phset_hs1 == NULL)
```

```
    {
        return -1;
    }

    hash_set_init(phset_hs1);

    printf("The maximum possible length of the hash_set hs1 is: %d.\n",
        hash_set_max_size(phset_hs1));

    hash_set_destroy(phset_hs1);

    return 0;
}
```

- **Output**

```
The maximum possible length of the hash_set hs1 is: 1073741823.
```

25. hash_set_not_equal

测试两个 hash_set_t 是否不等。

```
bool_t hash_set_not_equal(
    const hash_set_t* cphset_first,
    const hash_set_t* cphset_second
);
```

- **Parameters**

cphset_first: 指向第一个 hash_set_t 类型的指针。
cphset_second: 指向第二个 hash_set_t 类型的指针。

- **Remarks**

两个 hash_set_t 中的数据对应相等，并且数量相等，函数返回 false，否则返回 true。如果两个 hash_set_t 中的数据类型不同也认为不等。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);
    hash_set_t* phset_hs2 = create_hash_set(int);
    hash_set_t* phset_hs3 = create_hash_set(int);
    int i = 0;

    if(phset_hs1 == NULL || phset_hs2 == NULL || phset_hs3 == NULL)
    {
        return -1;
```

```
    }

    hash_set_init(phset_hs1);
    hash_set_init(phset_hs2);
    hash_set_init(phset_hs3);

    for(i = 0; i < 3; ++i)
    {
        hash_set_insert(phset_hs1, i);
        hash_set_insert(phset_hs2, i * i);
        hash_set_insert(phset_hs3, i);
    }

    if(hash_set_not_equal(phset_hs1, phset_hs2))
    {
        printf("The hash_sets hs1 and hs2 are not equal.\n");
    }
    else
    {
        printf("The hash_sets hs1 and hs2 are equal.\n");
    }

    if(hash_set_not_equal(phset_hs1, phset_hs3))
    {
        printf("The hash_sets hs1 and hs3 are not equal.\n");
    }
    else
    {
        printf("The hash_sets hs1 and hs3 are equal.\n");
    }

    hash_set_destroy(phset_hs1);
    hash_set_destroy(phset_hs2);
    hash_set_destroy(phset_hs3);

    return 0;
}
```

- **Output**

```
The hash_sets hs1 and hs2 are not equal.
The hash_sets hs1 and hs3 are equal.
```

26. hash_set_resize

重新设置 hash_set_t 中哈希表存储单元的数量。

```
void hash_set_resize(
    hash_set_t* phset_hset,
    size_t t_resize
);
```

- **Parameters**

cphset_hset: 指向 hash_set_t 类型的指针。
t_resize: 哈希表存储单元的新数量。

- **Remarks**

当哈希表存储单元数量改变后，哈希表中的数据将被重新计算位置，所有的迭代器失效。当新的存储单元数

量小于当前数量时，不做任何操作。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);

    if(phset_hs1 == NULL)
    {
        return -1;
    }

    hash_set_init(phset_hs1);

    printf("The bucket count of hash_set hs1 is: %d.\n",
        hash_set_bucket_count(phset_hs1));

    hash_set_resize(phset_hs1, 100);

    printf("The bucket count of hash_set hs1 is now: %d.\n",
        hash_set_bucket_count(phset_hs1));

    hash_set_destroy(phset_hs1);

    return 0;
}
```

- **Output**

```
The bucket count of hash_set hs1 is: 53.
The bucket count of hash_set hs1 is now: 193.
```

27. hash_set_size

返回 hash_set_t 中数据的数量。

```
size_t hash_set_size(
    const hash_set_t* cphset_hset
);
```

- **Parameters**

  cphset_hset: 指向 hash_set_t 类型的指针。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);

    if(phset_hs1 == NULL)
    {
        return -1;
    }

    hash_set_init(phset_hs1);

    hash_set_insert(phset_hs1, 1);
    printf("The hash_set hs1 length is %d.\n", hash_set_size(phset_hs1));

    hash_set_insert(phset_hs1, 2);
    printf("The hash_set hs1 length is now %d.\n", hash_set_size(phset_hs1));

    hash_set_destroy(phset_hs1);

    return 0;
}
```

- **Output**

```
The hash_set hs1 length is 1.
The hash_set hs1 length is now 2.
```

28. hash_set_swap

交换两个 hash_set_t 的内容。

```
void hash_set_swap(
    hash_set_t* phset_first,
    hash_set_t* phset_second
);
```

- **Parameters**

  phset_first: 指向第一个 hash_set_t 类型的指针。
  phset_second: 指向第二个 hash_set_t 类型的指针。

- **Remarks**

  这个函数要求两个 hash_set_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

```
/*
```

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);
    hash_set_t* phset_hs2 = create_hash_set(int);
    hash_set_iterator_t it_hs;

    if(phset_hs1 == NULL || phset_hs2 == NULL)
    {
        return -1;
    }

    hash_set_init(phset_hs1);
    hash_set_init(phset_hs2);

    hash_set_insert(phset_hs1, 10);
    hash_set_insert(phset_hs1, 20);
    hash_set_insert(phset_hs1, 30);
    hash_set_insert(phset_hs2, 100);
    hash_set_insert(phset_hs2, 200);

    printf("The original hash_set hs1 is:");
    for(it_hs = hash_set_begin(phset_hs1);
        !iterator_equal(it_hs, hash_set_end(phset_hs1));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    hash_set_swap(phset_hs1, phset_hs2);

    printf("After swapping with hs2, hash_set hs1 is:");
    for(it_hs = hash_set_begin(phset_hs1);
        !iterator_equal(it_hs, hash_set_end(phset_hs1));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    hash_set_destroy(phset_hs1);
    hash_set_destroy(phset_hs2);

    return 0;
}
```

- **Output**

```
The original hash_set hs1 is: 10 20 30
After swapping with hs2, hash_set hs1 is: 200 100
```

29. hash_set_value_comp

返回 hash_set_t 使用的数据比较规则。

```
binary_function_t hash_set_value_comp(
    const hash_set_t* cphset_hset
);
```

- **Parameters**

  cphset_hset: 指向 hash_set_t 类型的指针。

- **Remarks**

  由于 hash_set_t 中数据本身就是键，所以这个函数的返回值与 hash_set_key_comp()相同。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);
    hash_set_t* phset_hs2 = create_hash_set(int);
    binary_function_t bfun_vc = NULL;
    int n_first = 2;
    int n_second = 3;
    bool_t b_result = false;

    if(phset_hs1 == NULL || phset_hs2 == NULL)
    {
        return -1;
    }

    hash_set_init_ex(phset_hs1, 0, NULL, fun_less_int);
    hash_set_init_ex(phset_hs2, 0, NULL, fun_greater_int);

    bfun_vc = hash_set_value_comp(phset_hs1);
    (*bfun_vc)(&n_first, &n_second, &b_result);
    if(b_result)
    {
        printf("(*bfun_vc)(2, 3) returns value of true, "
               "where bfun_vc is the compare function of hs1.\n");
    }
    else
    {
        printf("(*bfun_vc)(2, 3) returns value of false, "
               "where bfun_vc is the compare function of hs1.\n");
    }

    bfun_vc = hash_set_value_comp(phset_hs2);
    (*bfun_vc)(&n_first, &n_second, &b_result);
```

```
    if(b_result)
    {
        printf("(*bfun_vc)(2, 3) returns value of true, "
               "where bfun_vc is the compare function of hs2.\n");
    }
    else
    {
        printf("(*bfun_vc)(2, 3) returns value of false, "
               "where bfun_vc is the compare function of hs2.\n");
    }

    hash_set_destroy(phset_hs1);
    hash_set_destroy(phset_hs2);

    return 0;
}
```

- **Output**

```
(*bfun_vc)(2, 3) returns value of true, where bfun_vc is the compare function of
hs1.
(*bfun_vc)(2, 3) returns value of false, where bfun_vc is the compare function of
hs2.
```

第十节　基于哈希结构的多重集合 hash_multiset_t

基于哈希结构的多重集合容器 hash_multiset_t 是关联容器，它使用指定的哈希函数计算数据的存储位置，将数据保存在这个位置上。hash_multiset_t 中的数据位置是根据数据本身计算的，数据在 hash_multiset_t 容器中是允许重复的，容器中数据也存在着某种有序性，所以也不能通过直接或者间接的方式修改容器中的数据。hash_multiset_t 提供双向迭代器，插入新的数据不会破坏原有的数据的迭代器，删除一个数据的时候只有指向数据本身的迭代器失效，但是当哈希表重新计算数据位置的时候所有的迭代器都失效。

- **Typedefs**

hash_multiset_t	基于哈希结构的多重集合容器类型。
hash_multiset_iterator_t	基于哈希结构的多重集合容器迭代器类型。

- **Operation Functions**

create_hash_multiset	创建基于哈希结构的多重集合容器类型。
hash_multiset_assign	为基于哈希结构的多重集合容器赋值。
hash_multiset_begin	返回基于哈希结构的多重集合中指向第一个数据的迭代器。
hash_multiset_bucket_count	返回基于哈希结构的多重集合使用的哈希表的存储单元个数。
hash_multiset_clear	删除基于哈希结构的多重集合中所有的数据。
hash_multiset_count	统计基于哈希结构的多重集合包含的指定数据的个数。
hash_multiset_destroy	销毁基于哈希结构的多重集合容器类型。
hash_multiset_empty	测试基于哈希结构的多重集合是否为空。
hash_multiset_end	返回基于哈希结构的多重集合的末尾位置的迭代器。
hash_multiset_equal	测试两个基于哈希结构的多重集合是否相等。
hash_multiset_equal_range	返回基于哈希结构的多重集合中包含指定数据的数据区间。

hash_multiset_erase	删除基于哈希结构的多重集合中包含的指定数据。
hash_multiset_erase_pos	删除基于哈希结构的多重集合中指定位置的数据。
hash_multiset_erase_range	删除基于哈希结构的多重集合中指定数据区间的数据。
hash_multiset_find	在基于哈希结构的多重集合中查找指定的数据。
hash_multiset_greater	测试第一个基于哈希结构的多重集合是否大于第二个基于哈希结构的多重集合。
hash_multiset_greater_equal	测试第一个基于哈希结构的多重集合是否大于等于第二个。
hash_multiset_hash	返回基于哈希结构的多重集合使用的哈希函数。
hash_multiset_init	初始化一个空的基于哈希结构的多重集合。
hash_multiset_init_copy	通过拷贝的方式初始化基于哈希结构的多重集合。
hash_multiset_init_copy_range	使用指定的数据区间初始化基于哈希结构的多重集合。
hash_multiset_init_copy_range_ex	使用指定的数据区间，哈希函数，排序规则和存储单元个数进行初始化。
hash_multiset_init_ex	使用指定的哈希函数，排序规则和存储单元个数进行初始化。
hash_multiset_insert	向基于哈希结构的多重集合中插入数据。
hash_multiset_insert_range	向基于哈希结构的多重集合中插入指定的数据区间。
hash_multiset_key_comp	返回基于哈希结构的多重集合使用的键比较规则。
hash_multiset_less	测试第一个基于哈希结构的多重集合是否小于第二个基于哈希结构的多重集合。
hash_multiset_less_equal	测试第一个基于哈希结构的多重集合是否小于等于第二个。
hash_multiset_max_size	返回基于哈希结构的多重集合能够保存数据的最大数量。
hash_multiset_not_equal	测试两个基于哈希结构的多重集合是否不等。
hash_multiset_resize	重新设置基于哈希结构的多重集合使用的哈希表的存储单元个数。
hash_multiset_size	返回基于哈希结构的多重集合中数据的数量。
hash_multiset_swap	交换两个基于哈希结构的多重集合的内容。
hash_multiset_value_comp	返回基于哈希结构的多重集合使用的值比较规则。

1. hash_multiset_t

基于哈希结构的多重集合容器类型。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

请参考 hash_multiset_t 类型的其他操作函数。

2. hash_multiset_iterator_t

集合哈希结构的多重集合容器的迭代器类型。

- **Remarks**

hash_multiset_iterator_t 是双向迭代器类型，不能通过迭代器来修改容器中数据的数据。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

请参考 hash_multiset_t 类型的其他操作函数。

3. create_hash_multiset

创建 hash_multiset_t 容器类型。

```
hash_multiset_t* create_hash_multiset(
    type
);
```

- **Parameters**

type: 数据类型描述。

- **Remarks**

函数成功返回指向 hash_multiset_t 类型的指针，失败返回 NULL。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

请参考 hash_multiset_t 类型的其他操作函数。

4. hash_multiset_assign

为 hash_multiset_t 类型赋值。

```
void hash_multiset_assign(
    hash_multiset_t* phmset_dest,
    const hash_multiset_t* cphmset_src
);
```

- **Parameters**

phmset_dest: 指向被赋值的 hash_multiset_t 类型的指针。
cphmset_src: 指向赋值的 hash_multiset_t 类型的指针。

- **Remarks**

要求两个 hash_multiset_t 类型保存的数据具有相同的类型，否则函数的行为未定义。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

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

```
{
    hash_multiset_t* phmset_hms1 = create_hash_multiset(int);
    hash_multiset_t* phmset_hms2 = create_hash_multiset(int);
    hash_multiset_iterator_t it_hms;

    if(phmset_hms1 == NULL || phmset_hms2 == NULL)
    {
        return -1;
    }

    hash_multiset_init(phmset_hms1);
    hash_multiset_init(phmset_hms2);

    hash_multiset_insert(phmset_hms1, 10);
    hash_multiset_insert(phmset_hms1, 20);
    hash_multiset_insert(phmset_hms1, 30);
    hash_multiset_insert(phmset_hms2, 40);
    hash_multiset_insert(phmset_hms2, 50);
    hash_multiset_insert(phmset_hms2, 60);

    printf("hs1 =");
    for(it_hms = hash_multiset_begin(phmset_hms1);
        !iterator_equal(it_hms, hash_multiset_end(phmset_hms1));
        it_hms = iterator_next(it_hms))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hms));
    }
    printf("\n");

    hash_multiset_assign(phmset_hms1, phmset_hms2);
    printf("hs1 =");
    for(it_hms = hash_multiset_begin(phmset_hms1);
        !iterator_equal(it_hms, hash_multiset_end(phmset_hms1));
        it_hms = iterator_next(it_hms))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hms));
    }
    printf("\n");

    hash_multiset_destroy(phmset_hms1);
    hash_multiset_destroy(phmset_hms2);

    return 0;
}
```

- **Output**

```
hs1 = 10 20 30
hs1 = 60 40 50
```

5. hash_multiset_begin

返回指向 hash_multiset_t 中第一个数据的迭代器。

```
hash_multiset_iterator_t hash_multiset_begin(
    const hash_multiset_t* cphmset_hmset
);
```

- **Parameters**

cphmset_hmset: 指向 hash_multiset_t 类型的指针。

- **Remarks**

  如果 hash_multiset_t 为空，这个函数的返回值和 hash_multiset_end()的返回值相等。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);

    if(phms_hms1 == NULL)
    {
        return -1;
    }

    hash_multiset_init(phms_hms1);

    hash_multiset_insert(phms_hms1, 1);
    hash_multiset_insert(phms_hms1, 2);
    hash_multiset_insert(phms_hms1, 3);

    printf("The first element of hs1 is %d.\n",
        *(int*)iterator_get_pointer(hash_multiset_begin(phms_hms1)));

    hash_multiset_erase_pos(phms_hms1, hash_multiset_begin(phms_hms1));

    printf("The first element of hs1 is now %d.\n",
        *(int*)iterator_get_pointer(hash_multiset_begin(phms_hms1)));

    hash_multiset_destroy(phms_hms1);

    return 0;
}
```

- **Output**

```
The first element of hs1 is 1.
The first element of hs1 is now 2.
```

6. hash_multiset_bucket_count

返回 hash_multiset_t 中哈希表存储单元的个数。

```
size_t hash_multiset_bucket_count(
    const hash_multiset_t* cphmset_hmset
);
```

- **Parameters**

  cphmset_hmset: 指向 hash_multiset_t 类型的指针。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);
    hash_multiset_t* phms_hms2 = create_hash_multiset(int);

    if(phms_hms1 == NULL || phms_hms2 == NULL)
    {
        return -1;
    }

    hash_multiset_init(phms_hms1);
    hash_multiset_init_ex(phms_hms2, 100, NULL, NULL);

    printf("The default bucket count of hs1 is %d.\n",
        hash_multiset_bucket_count(phms_hms1));
    printf("The custom bucket count of hs2 is %d.\n",
        hash_multiset_bucket_count(phms_hms2));

    hash_multiset_destroy(phms_hms1);
    hash_multiset_destroy(phms_hms2);

    return 0;
}
```

- **Output**

```
The default bucket count of hs1 is 53.
The custom bucket count of hs2 is 193.
```

7. hash_multiset_clear

删除 hash_multiset_t 中所有的数据。

```
void hash_multiset_clear(
    hash_multiset_t* phmset_hmset
);
```

- **Parameters**

  phmset_hmset: 指向 hash_multiset_t 类型的指针。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);

    if(phms_hms1 == NULL)
    {
        return -1;
    }

    hash_multiset_init(phms_hms1);

    hash_multiset_insert(phms_hms1, 1);
    hash_multiset_insert(phms_hms1, 2);

    printf("The size of the hash_multiset is initially %d.\n",
        hash_multiset_size(phms_hms1));

    hash_multiset_clear(phms_hms1);

    printf("The size of the hash_multiset after clearing is %d.\n",
        hash_multiset_size(phms_hms1));

    hash_multiset_destroy(phms_hms1);

    return 0;
}
```

- **Output**

```
The size of the hash_multiset is initially 2.
The size of the hash_multiset after clearing is 0.
```

8. hash_multiset_count

统计 hash_multiset_t 中包含指定数据的数量。

```
size_t hash_multiset_count(
    const hash_multiset_t* cphmset_hmset,
    element
);
```

- **Parameters**

cphmset_hmset: 指向 hash_multiset_t 类型的指针。
element: 指定的数据。

- **Remarks**

如果容器中不包含指定数据则返回 0，包含则返回指定数据的个数。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);

    if(phms_hms1 == NULL)
    {
        return -1;
    }

    hash_multiset_init(phms_hms1);

    hash_multiset_insert(phms_hms1, 1);
    hash_multiset_insert(phms_hms1, 1);

    /* Keys must be unique in hash_multiset, so duplicates are ignored */
    printf("The number of elements in hs1 with a sort key of 1 is: %d.\n",
        hash_multiset_count(phms_hms1, 1));

    printf("The number of elements in hs1 with a sort key of 2 is: %d.\n",
        hash_multiset_count(phms_hms1, 2));

    hash_multiset_destroy(phms_hms1);

    return 0;
}
```

- **Output**

```
The number of elements in hs1 with a sort key of 1 is: 2.
The number of elements in hs1 with a sort key of 2 is: 0.
```

9. hash_multiset_destroy

销毁 hash_multiset_t 容器类型。

```
void hash_multiset_destroy(
    hash_multiset_t* phmset_hmset
);
```

- **Parameters**

  phmset_hmset: 指向 hash_multiset_t 类型的指针。

- **Remarks**

  hash_multiset_t 容器使用之后要销毁，否则 hash_multiset_t 占用的资源不会被释放。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

  请参考 hash_multiset_t 类型的其他操作函数。

10. hash_multiset_empty

测试 hash_multiset_t 是否为空。

```
bool_t hash_multiset_empty(
    const hash_multiset_t* cphmset_hmset
);
```

- **Parameters**

  cphmset_hmset:　　指向 hash_multiset_t 类型的指针。

- **Remarks**

  hash_multiset_t 容器为空则返回 true，否则返回 false。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);
    hash_multiset_t* phms_hms2 = create_hash_multiset(int);

    if(phms_hms1 == NULL || phms_hms2 == NULL)
    {
        return -1;
    }

    hash_multiset_init(phms_hms1);
    hash_multiset_init(phms_hms2);

    hash_multiset_insert(phms_hms1, 1);

    if(hash_multiset_empty(phms_hms1))
    {
        printf("The hash_multiset hs1 is empty.\n");
    }
    else
    {
        printf("The hash_multiset hs1 is not empty.\n");
    }

    if(hash_multiset_empty(phms_hms2))
```

```
    {
        printf("The hash_multiset hs2 is empty.\n");
    }
    else
    {
        printf("The hash_multiset hs2 is not empty.\n");
    }

    hash_multiset_destroy(phms_hms1);
    hash_multiset_destroy(phms_hms2);

    return 0;
}
```

- **Output**

```
The hash_multiset hs1 is not empty.
The hash_multiset hs2 is empty.
```

11. hash_multiset_end

返回指向 hash_multiset_t 末尾位置的迭代器。

```
hash_multiset_iterator_t hash_multiset_end(
    const hash_multiset_t* cphmset_hmset
);
```

- **Parameters**

cphmset_hmset: 指向 hash_multiset_t 类型的指针。

- **Remarks**

如果 hash_multiset_t 为空，这个函数的返回值和 hash_multiset_begin()的返回值相等。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);
    hash_multiset_iterator_t it_hs;

    if(phms_hms1 == NULL)
    {
        return -1;
    }

    hash_multiset_init(phms_hms1);
```

```
    hash_multiset_insert(phms_hms1, 1);
    hash_multiset_insert(phms_hms1, 2);
    hash_multiset_insert(phms_hms1, 3);

    it_hs = hash_multiset_end(phms_hms1);
    it_hs = iterator_prev(it_hs);
    printf("The last element of hs1 is %d.\n",
        *(int*)iterator_get_pointer(it_hs));

    hash_multiset_erase_pos(phms_hms1, it_hs);

    it_hs = hash_multiset_end(phms_hms1);
    it_hs = iterator_prev(it_hs);
    printf("The last element of hs1 is now %d.\n",
        *(int*)iterator_get_pointer(it_hs));

    hash_multiset_destroy(phms_hms1);

    return 0;
}
```

- **Output**

```
The last element of hs1 is 3.
The last element of hs1 is now 2.
```

12. hash_multiset_equal

测试两个 hash_multiset_t 是否相等。

```
bool_t hash_multiset_equal(
    const hash_multiset_t* cphmap_first,
    const hash_multiset_t* cphmap_second
);
```

- **Parameters**

cphmset_first: 指向第一个 hash_multiset_t 类型的指针。
cphmset_second: 指向第二个 hash_multiset_t 类型的指针。

- **Remarks**

两个 hash_multiset_t 中的数据对应相等，并且数量相等，函数返回 true，否则返回 false。如果两个 hash_multiset_t 中的数据类型不同也认为不等。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

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

```
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);
    hash_multiset_t* phms_hms2 = create_hash_multiset(int);
    hash_multiset_t* phms_hms3 = create_hash_multiset(int);
    int i = 0;

    if(phms_hms1 == NULL || phms_hms2 == NULL || phms_hms3 == NULL)
    {
        return -1;
    }

    hash_multiset_init(phms_hms1);
    hash_multiset_init(phms_hms2);
    hash_multiset_init(phms_hms3);

    for(i = 0; i < 3; ++i)
    {
        hash_multiset_insert(phms_hms1, i);
        hash_multiset_insert(phms_hms2, i * i);
        hash_multiset_insert(phms_hms3, i);
    }

    if(hash_multiset_equal(phms_hms1, phms_hms2))
    {
        printf("The hash_multisets hs1 and hs2 are equal.\n");
    }
    else
    {
        printf("The hash_multisets hs1 and hs2 are not equal.\n");
    }

    if(hash_multiset_equal(phms_hms1, phms_hms3))
    {
        printf("The hash_multisets hs1 and hs3 are equal.\n");
    }
    else
    {
        printf("The hash_multisets hs1 and hs3 are not equal.\n");
    }

    hash_multiset_destroy(phms_hms1);
    hash_multiset_destroy(phms_hms2);
    hash_multiset_destroy(phms_hms3);

    return 0;
}
```

- **Output**

```
The hash_multisets hs1 and hs2 are not equal.
The hash_multisets hs1 and hs3 are equal.
```

13. hash_multiset_equal_range

返回 hash_multiset_t 中包含指定数据的数据区间。

```
range_t hash_multiset_equal_range(
    const hash_multiset_t* cphmset_hmset,
    element
);
```

- **Parameters**

  cphmset_hmset: 指向 hash_multiset_t 类型的指针。
  element: 指定的数据。

- **Remarks**

  返回 hash_multiset_t 中包含指定数据的数据区间[range_t.it_begin, range_t.it_end)，其中 it_begin 是指向等于指定数据的第一个数据的迭代器，it_end 指向的是大于指定数据的第一个数据的迭代器。如果 hash_multiset_t 中不包含指定数据则 it_begin 与 it_end 相等。

- **Requirements**

  头文件 <cstl/chash_set.h>

- Example

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

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

int main(int argc, char* argv[])
{
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);
    range_t r_r;

    if(phms_hms1 == NULL)
    {
        return -1;
    }

    hash_multiset_init(phms_hms1);

    hash_multiset_insert(phms_hms1, 10);
    hash_multiset_insert(phms_hms1, 20);
    hash_multiset_insert(phms_hms1, 30);

    r_r = hash_multiset_equal_range(phms_hms1, 20);
    printf("The upper bound of the element with "
           "a key of 20 in the hash_multiset hs1 is: %d.\n",
           *(int*)iterator_get_pointer(r_r.it_end));
    printf("The lower bound of the element with a key "
           "of 20 in the hash_multiset hs1 is: %d.\n",
           *(int*)iterator_get_pointer(r_r.it_begin));

    /*If no match is bound for the key, bouth element of the range returned end().*/
    r_r = hash_multiset_equal_range(phms_hms1, 40);
    if(iterator_equal(r_r.it_begin, hash_multiset_end(phms_hms1)) &&
       iterator_equal(r_r.it_end, hash_multiset_end(phms_hms1)))
    {
        printf("The hash_multiset hs1 doesn't have "
               "an element with a key less than 40.\n");
    }
    else
    {
        printf("The element of hash_multiset hs1 with "
               "a key >= 40 is: %d.\n",
```

```
                *(int*)iterator_get_pointer(r_r.it_begin));
    }

    hash_multiset_destroy(phms_hms1);

    return 0;
}
```

- **Output**

```
The upper bound of the element with a key of 20 in the hash_multiset hs1 is: 30.
The lower bound of the element with a key of 20 in the hash_multiset hs1 is: 20.
The hash_multiset hs1 doesn't have an element with a key less than 40.
```

14. hash_multiset_erase hash_multiset_erase_pos hash_multiset_erase_range

删除 hash_multiset_t 中的数据。

```
size_t hash_multiset_erase(
    hash_multiset_t* phmset_hmset,
    element
);

void hash_multiset_erase_pos(
    hash_multiset_t* phmset_hmset,
    hash_multiset_iterator_t it_pos
);

void hash_multiset_erase_range(
    hash_multiset_t* phmset_hmset,
    hash_multiset_iterator_t it_begin,
    hash_multiset_iterator_t it_end
);
```

- **Parameters**

phmset_hmset: 指向 hash_multiset_t 类型的指针。
element: 要删除的数据。
it_pos: 要删除的数据的位置迭代器。
it_begin: 要删除的数据区间的开始位置。
it_end: 要删除的数据区间的末尾位置。

- **Remarks**

第一个函数删除 hash_multiset_t 中指定的数据，并返回删除的个数，如果 hash_multiset_t 中不包含指定的数据就返回 0。

第二个函数删除指定位置的数据。

第三个函数删除指定数据区间中的数据。

后面两个函数要求迭代器和数据区间是有效的，无效的迭代器或数据区间导致函数的行为未定义。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

```
 */

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

int main(int argc, char* argv[])
{
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);
    hash_multiset_t* phms_hms2 = create_hash_multiset(int);
    hash_multiset_t* phms_hms3 = create_hash_multiset(int);
    hash_multiset_iterator_t it_hs;
    size_t t_count = 0;
    int i = 0;

    if(phms_hms1 == NULL || phms_hms2 == NULL || phms_hms3 == NULL)
    {
        return -1;
    }

    hash_multiset_init(phms_hms1);
    hash_multiset_init(phms_hms2);
    hash_multiset_init(phms_hms3);

    for(i = 1; i < 5; ++i)
    {
        hash_multiset_insert(phms_hms1, i);
        hash_multiset_insert(phms_hms2, i * i);
        hash_multiset_insert(phms_hms3, i - 1);
    }

    /* The first function removes an element at a given position */
    it_hs = iterator_next(hash_multiset_begin(phms_hms1));
    hash_multiset_erase_pos(phms_hms1, it_hs);

    printf("After the second element is deleted, the hash_multiset hs1 is: ");
    for(it_hs = hash_multiset_begin(phms_hms1);
        !iterator_equal(it_hs, hash_multiset_end(phms_hms1));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    /* The second function removes elements in the range [first, last) */
    hash_multiset_erase_range(phms_hms2,
        iterator_next(hash_multiset_begin(phms_hms2)),
        iterator_prev(hash_multiset_end(phms_hms2)));

    printf("After the middle two elements are deleted, "
           "the hash_multiset hs2 is: ");
    for(it_hs = hash_multiset_begin(phms_hms2);
        !iterator_equal(it_hs, hash_multiset_end(phms_hms2));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    /* The third function removes elements with a given key */
    t_count = hash_multiset_erase(phms_hms3, 2);
```

```
    printf("After the element with a key of 2 is deleted, "
           "the hash_multiset hs3 is: ");
    for(it_hs = hash_multiset_begin(phms_hms3);
        !iterator_equal(it_hs, hash_multiset_end(phms_hms3));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    hash_multiset_destroy(phms_hms1);
    hash_multiset_destroy(phms_hms2);
    hash_multiset_destroy(phms_hms3);

    return 0;
}
```

- **Output**

```
After the second element is deleted, the hash_multiset hs1 is:  1 3 4
After the middle two elements are deleted, the hash_multiset hs2 is:  1 16
After the element with a key of 2 is deleted, the hash_multiset hs3 is:  0 1 3
```

15. hash_multiset_find

在 hash_multiset_t 中查找指定的数据。

```
hash_multiset_iterator_t _hash_multiset_find(
    const hash_multiset_t* cphmset_hmset,
    element
);
```

- **Parameters**

cphmset_hmset: 指向 hash_multiset_t 类型的指针。
element: 指定的数据。

- **Remarks**

如果 hash_multiset_t 中包含指定的数据则返回指向该数据的迭代器，否则返回 hash_multiset_end()。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);
    hash_multiset_iterator_t it_hs;
```

```
    if(phms_hms1 == NULL)
    {
        return -1;
    }

    hash_multiset_init(phms_hms1);

    hash_multiset_insert(phms_hms1, 10);
    hash_multiset_insert(phms_hms1, 20);
    hash_multiset_insert(phms_hms1, 30);

    it_hs = hash_multiset_find(phms_hms1, 20);
    printf("The element of hash_multiset hs1 with a key of 20 is: %d.\n",
        *(int*)iterator_get_pointer(it_hs));

    it_hs = hash_multiset_find(phms_hms1, 40);

    /* If no match is found for the key, end() is returned */
    if(iterator_equal(it_hs, hash_multiset_end(phms_hms1)))
    {
        printf("The hash_multiset hs1 doesn't have "
               "an element with a key of 40.\n");
    }
    else
    {
        printf("The element of hash_multiset hs1 with a key of 40 is: %d.\n",
            *(int*)iterator_get_pointer(it_hs));
    }

    /*
     * The element at a specific location in the hash_multiset can be found
     * by using a dereferenced iterator addressing the location.
     */
    it_hs = iterator_prev(hash_multiset_end(phms_hms1));
    it_hs = hash_multiset_find(phms_hms1, *(int*)iterator_get_pointer(it_hs));
    printf("The element of hs1 with a key matching "
           "that of the last element is: %d.\n",
           *(int*)iterator_get_pointer(it_hs));

    hash_multiset_destroy(phms_hms1);

    return 0;
}
```

- **Output**

```
The element of hash_multiset hs1 with a key of 20 is: 20.
The hash_multiset hs1 doesn't have an element with a key of 40.
The element of hs1 with a key matching that of the last element is: 30.
```

16. hash_multiset_greater

测试第一个 hash_multiset_t 是否大于第二个 hash_multiset_t。

```
bool_t hash_multiset_greater(
    const hash_multiset_t* cphmset_first,
    const hash_multiset_t* cphmset_second
);
```

- **Parameters**

  cphmset_first: 指向第一个 hash_multiset_t 类型的指针。
  cphmset_second: 指向第二个 hash_multiset_t 类型的指针。

- **Remarks**

  这个函数要求两个 hash_multiset_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);
    hash_set_t* phset_hs2 = create_hash_set(int);
    hash_set_t* phset_hs3 = create_hash_set(int);
    int i = 0;

    if(phset_hs1 == NULL || phset_hs2 == NULL || phset_hs3 == NULL)
    {
        return -1;
    }

    hash_set_init(phset_hs1);
    hash_set_init(phset_hs2);
    hash_set_init(phset_hs3);

    for(i = 0; i < 3; ++i)
    {
        hash_set_insert(phset_hs1, i);
        hash_set_insert(phset_hs2, i * i);
        hash_set_insert(phset_hs3, i - 1);
    }

    if(hash_set_greater(phset_hs1, phset_hs2))
    {
        printf("The hash_set hs1 is greater than the hash_set hs2.\n");
    }
    else
    {
        printf("The hash_set hs1 is not greater than the hash_set hs2.\n");
    }

    if(hash_set_greater(phset_hs1, phset_hs3))
    {
        printf("The hash_set hs1 is greater than the hash_set hs3.\n");
    }
    else
    {
        printf("The hash_set hs1 is not greater than the hash_set hs3.\n");
    }
```

```
    hash_set_destroy(phset_hs1);
    hash_set_destroy(phset_hs2);
    hash_set_destroy(phset_hs3);

    return 0;
}
```

- **Output**

```
The hash_multiset hs1 is not greater than the hash_multiset hs2.
The hash_multiset hs1 is greater than the hash_multiset hs3.
```

17. hash_multiset_greater_equal

测试第一个 hash_multiset_t 是否大于等于第二个 hash_multiset_t。

```
bool_t hash_multiset_greater_equal(
    const hash_multiset_t* cphmset_first,
    const hash_multiset_t* cphmset_second
);
```

- **Parameters**

cphmset_first: 指向第一个 hash_multiset_t 类型的指针。
cphmset_second: 指向第二个 hash_multiset_t 类型的指针。

- **Remarks**

这个函数要求两个 hash_multiset_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_set_t* phset_hs1 = create_hash_set(int);
    hash_set_t* phset_hs2 = create_hash_set(int);
    hash_set_t* phset_hs3 = create_hash_set(int);
    hash_set_t* phset_hs4 = create_hash_set(int);
    int i = 0;

    if(phset_hs1 == NULL || phset_hs2 == NULL ||
       phset_hs3 == NULL || phset_hs4 == NULL)
    {
        return -1;
    }

    hash_set_init(phset_hs1);
    hash_set_init(phset_hs2);
```

```
    hash_set_init(phset_hs3);
    hash_set_init(phset_hs4);

    for(i = 0; i < 3; ++i)
    {
        hash_set_insert(phset_hs1, i);
        hash_set_insert(phset_hs2, i * i);
        hash_set_insert(phset_hs3, i - 1);
        hash_set_insert(phset_hs4, i);
    }

    if(hash_set_greater_equal(phset_hs1, phset_hs2))
    {
        printf("The hash_set hs1 is greater than or equal to the hash_set hs2.\n");
    }
    else
    {
        printf("The hash_set hs1 is less than the hash_set hs2.\n");
    }

    if(hash_set_greater_equal(phset_hs1, phset_hs3))
    {
        printf("The hash_set hs1 is greater than or equal to the hash_set hs3.\n");
    }
    else
    {
        printf("The hash_set hs1 is less than the hash_set hs3.\n");
    }

    if(hash_set_greater_equal(phset_hs1, phset_hs4))
    {
        printf("The hash_set hs1 is greater than or equal to the hash_set hs4.\n");
    }
    else
    {
        printf("The hash_set hs1 is less than the hash_set hs4.\n");
    }

    hash_set_destroy(phset_hs1);
    hash_set_destroy(phset_hs2);
    hash_set_destroy(phset_hs3);
    hash_set_destroy(phset_hs4);

    return 0;
}
```

- **Output**

```
The hash_multiset hs1 is less than the hash_multiset hs2.
The hash_multiset hs1 is greater than or equal to the hash_multiset hs3.
The hash_multiset hs1 is greater than or equal to the hash_multiset hs4.
```

18. hash_multiset_hash

返回 hash_multiset_t 中使用的哈希函数。

```
unary_function_t hash_multiset_hash(
    const hash_multiset_t* cphmset_hmset
);
```

- **Parameters**

  cphmset_hmset: 指向 hash_multiset_t 类型的指针。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

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

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

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

int main(int argc, char* argv[])
{
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);
    hash_multiset_t* phms_hms2 = create_hash_multiset(int);

    if(phms_hms1 == NULL || phms_hms2 == NULL)
    {
        return -1;
    }

    hash_multiset_init(phms_hms1);
    hash_multiset_init_ex(phms_hms2, 100, hash_func, NULL);

    if(hash_multiset_hash(phms_hms1) == hash_func)
    {
        printf("The hash function of hash_multiset hs1 is hash_func.\n");
    }
    else
    {
        printf("The hash function of hash_multiset hs1 is not hash_func.\n");
    }

    if(hash_multiset_hash(phms_hms2) == hash_func)
    {
        printf("The hash function of hash_multiset hs2 is hash_func.\n");
    }
    else
    {
        printf("The hash function of hash_multiset hs2 is not hash_func.\n");
    }

    hash_multiset_destroy(phms_hms1);
    hash_multiset_destroy(phms_hms2);

    return 0;
}

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

- **Output**

```
The hash function of hash_multiset hs1 is not hash_func.
The hash function of hash_multiset hs2 is hash_func.
```

19. hash_multiset_init hash_multiset_init_copy hash_multiset_init_copy_range hash_multiset_init_copy_range_ex hash_multiset_init_ex

初始化 hash_multiset_t 容器类型。

```
void hash_multiset_init(
    hash_multiset_t* phmset_hmset
);

void hash_multiset_init_copy(
    hash_multiset_t* phmset_hmset,
    const hash_multiset_t* cphmset_src
);

void hash_multiset_init_copy_range(
    hash_multiset_t* phmset_hmset,
    hash_multiset_iterator_t it_begin,
    hash_multiset_iterator_t it_end
);

void hash_multiset_init_copy_range_ex(
    hash_multiset_t* phmset_hmset,
    hash_multiset_iterator_t it_begin,
    hash_multiset_iterator_t it_end,
    size_t t_bucketcount,
    unary_function_t ufun_hash,
    binary_function_t bfun_compare
);

void hash_multiset_init_ex(
    hash_multiset_t* phmset_hmset,
    size_t t_bucketcount,
    unary_function_t ufun_hash,
    binary_function_t bfun_compare
);
```

- **Parameters**

pmhset_hmset: 指向被初始化 hash_multiset_t 类型的指针。
cpmhset_src: 指向用于初始化的 hash_multiset_t 类型的指针。
it_begin: 用于初始化的数据区间的开始位置。
it_end: 用于初始化的数据区间的末尾位置。
t_bucketcount: 哈希表中的存储单元个数。
ufun_hash: 自定义的哈希函数。
bfun_compare: 自定义比较规则。

- **Remarks**

第一个函数初始化一个空的 hash_multiset_t，使用默认的哈希函数和与数据类型相关的小于操作函数作为默认的比较规则。

第二个函数使用一个源 hash_multiset_t 来初始化 hash_multiset_t，数据的内容，哈希函数和比较规则都从源 hash_multiset_t 复制。

第三个函数使用指定的数据区间初始化一个 hash_multiset_t，使用默认的哈希函数和与数据类型相关的小于操作函数作为默认的比较规则。

第四个函数使用指定的数据区间初始化一个 hash_multiset_t，使用用户指定的哈希表存储单元个数，哈希函数和比较规则。

第五个函数初始化一个空的 hash_multiset_t，使用用户指定的哈希表存储单元个数，哈希函数和比较规则。

上面的函数要求迭代器和数据区间是有效的，无效的迭代器或数据区间导致函数的行为未定义。初始化函数根据用户指定的哈希表存储单元个数计算一个与用户指定的个数最接近的最佳哈希表存储单元个数。默认个数是 53 个，用户指定的个数小于等于 53 时都使用这个存储单元个数。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

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

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

static void _hash_function(const void* cpv_input, void* pv_output)
{
    *(size_t*)pv_output = *(int*)cpv_input + 20;
}

int main(int argc, char* argv[])
{
    hash_multiset_t* phms_hms0 = create_hash_multiset(int);
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);
    hash_multiset_t* phms_hms2 = create_hash_multiset(int);
    hash_multiset_t* phms_hms3 = create_hash_multiset(int);
    hash_multiset_t* phms_hms4 = create_hash_multiset(int);
    hash_multiset_t* phms_hms5 = create_hash_multiset(int);
    hash_multiset_iterator_t it_hs;

    if(phms_hms0 == NULL || phms_hms1 == NULL || phms_hms2 == NULL ||
       phms_hms3 == NULL || phms_hms4 == NULL || phms_hms5 == NULL)
    {
        return -1;
    }

    /* Create an empty hash_multiset hs0 of key type integer */
    hash_multiset_init(phms_hms0);

    /*
     * Create an empty hash_multiset hs1 with the key comparison
     * function of less than, then insert 4 elements.
     */
    hash_multiset_init_ex(phms_hms1, 10, _hash_function, fun_less_int);
    hash_multiset_insert(phms_hms1, 10);
    hash_multiset_insert(phms_hms1, 20);
    hash_multiset_insert(phms_hms1, 30);
    hash_multiset_insert(phms_hms1, 40);
```

```
    /*
     * Create an empty hash_multiset hs2 with the key comparison
     * function of greater than, then insert 2 element.
     */
    hash_multiset_init_ex(phms_hms2, 100, _hash_function, fun_greater_int);
    hash_multiset_insert(phms_hms2, 10);
    hash_multiset_insert(phms_hms2, 20);

    /* Create a copy, hash_multiset hs3, of hash_multiset hs1 */
    hash_multiset_init_copy(phms_hms3, phms_hms1);

    /* Create a hash_multiset hs4 by copying the range hs1[first, last) */
    hash_multiset_init_copy_range(phms_hms4, hash_multiset_begin(phms_hms1),
        iterator_advance(hash_multiset_begin(phms_hms1), 2));

    /*
     * Create a hash_multiset hs5 by copying the range hs3[first, last)
     * and with the key comparison function of less than.
     */
    hash_multiset_init_copy_range_ex(phms_hms5, hash_multiset_begin(phms_hms3),
        iterator_next(hash_multiset_begin(phms_hms3)),
        100, _hash_function, fun_less_int);

    printf("hs1 =");
    for(it_hs = hash_multiset_begin(phms_hms1);
        !iterator_equal(it_hs, hash_multiset_end(phms_hms1));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    printf("hs2 =");
    for(it_hs = hash_multiset_begin(phms_hms2);
        !iterator_equal(it_hs, hash_multiset_end(phms_hms2));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    printf("hs3 =");
    for(it_hs = hash_multiset_begin(phms_hms3);
        !iterator_equal(it_hs, hash_multiset_end(phms_hms3));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    printf("hs4 =");
    for(it_hs = hash_multiset_begin(phms_hms4);
        !iterator_equal(it_hs, hash_multiset_end(phms_hms4));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    printf("hs5 =");
```

```
    for(it_hs = hash_multiset_begin(phms_hms5);
        !iterator_equal(it_hs, hash_multiset_end(phms_hms5));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    hash_multiset_destroy(phms_hms0);
    hash_multiset_destroy(phms_hms1);
    hash_multiset_destroy(phms_hms2);
    hash_multiset_destroy(phms_hms3);
    hash_multiset_destroy(phms_hms4);
    hash_multiset_destroy(phms_hms5);

    return 0;
}
```

- **Output**

```
hs1 = 40 10 20 30
hs2 = 10 20
hs3 = 40 10 20 30
hs4 = 10 40
hs5 = 40
```

20. hash_multiset_insert hash_multiset_insert_range

向 hash_multiset_t 中插入数据。

```
hash_multiset_iterator_t hash_multiset_insert(
    hash_multiset_t* phmset_hmset,
    element
);

hash_multiset_insert_range(
    hash_multiset_t* phmset_hmset,
    hash_multiset_iterator_t it_begin,
    hash_multiset_iterator_t it_end
);
```

- **Parameters**

phmset_hmset: 指向 hash_multiset_t 类型的指针。
element: 插入的数据。
it_begin: 被插入的数据区间的开始位置。
it_end: 被插入的数据区间的末尾位置。

- **Remarks**

第一个函数向 hash_multiset_t 中插入一个指定的数据，成功后返回指向该数据的迭代器，如果 hash_multiset_t 中包含了该数据那么插入失败，返回 hash_multiset_end()。
第三个函数插入指定的数据区间。
上面的函数要求迭代器和数据区间是有效的，无效的迭代器或数据区间导致函数的行为未定义。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);
    hash_multiset_t* phms_hms2 = create_hash_multiset(int);
    hash_multiset_iterator_t it_hs;

    if(phms_hms1 == NULL || phms_hms2 == NULL)
    {
        return -1;
    }

    hash_multiset_init(phms_hms1);
    hash_multiset_init(phms_hms2);

    hash_multiset_insert(phms_hms1, 10);
    hash_multiset_insert(phms_hms1, 20);
    hash_multiset_insert(phms_hms1, 30);
    hash_multiset_insert(phms_hms1, 40);

    printf("The original hs1 =");
    for(it_hs = hash_multiset_begin(phms_hms1);
        !iterator_equal(it_hs, hash_multiset_end(phms_hms1));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    it_hs = hash_multiset_insert(phms_hms1, 10);
    if(iterator_equal(it_hs, hash_multiset_end(phms_hms1)))
    {
        printf("The element 10 already exist in hs1.\n");
    }
    else
    {
        printf("The element 10 was inserted inhs1 successfully.\n");
    }

    hash_multiset_insert(phms_hms1, 80);
    printf("After the insertions, hs1 =");
    for(it_hs = hash_multiset_begin(phms_hms1);
        !iterator_equal(it_hs, hash_multiset_end(phms_hms1));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    hash_multiset_insert(phms_hms2, 100);
    hash_multiset_insert_range(phms_hms2,
        iterator_next(hash_multiset_begin(phms_hms1)),
```

```
        iterator_prev(hash_multiset_end(phms_hms1)));

    printf("hs2 =");
    for(it_hs = hash_multiset_begin(phms_hms2);
        !iterator_equal(it_hs, hash_multiset_end(phms_hms2));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    hash_multiset_destroy(phms_hms1);
    hash_multiset_destroy(phms_hms2);

    return 0;
}
```

- **Output**

```
The original hs1 = 10 20 30 40
The element 10 was inserted inhs1 successfully.
After the insertions, hs1 = 10 10 20 80 30 40
hs2 = 10 20 80 30 100
```

21. hash_multiset_key_comp

返回 hash_multiset_t 使用的键比较规则。

```
binary_function_t hash_multiset_key_comp(
    const hash_multiset_t* cphmset_hmset
);
```

- **Parameters**

cphmset_hmset: 指向 hash_multiset_t 类型的指针。

- **Remarks**

由于 hash_multiset_t 中数据本身就是键，所以这个函数的返回值与 hash_multiset_value_comp()相同。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);
    hash_multiset_t* phms_hms2 = create_hash_multiset(int);
    binary_function_t bfun_kc = NULL;
    int n_first = 2;
```

```
    int n_second = 3;
    bool_t b_result = false;

    if(phms_hms1 == NULL || phms_hms2 == NULL)
    {
        return -1;
    }

    hash_multiset_init_ex(phms_hms1, 0, NULL, fun_less_int);
    hash_multiset_init_ex(phms_hms2, 0, NULL, fun_greater_int);

    bfun_kc = hash_multiset_key_comp(phms_hms1);
    (*bfun_kc)(&n_first, &n_second, &b_result);
    if(b_result)
    {
        printf("(*bfun_kc)(2, 3) returns value of true, "
               "where bfun_kc is the compare function of hs1.\n");
    }
    else
    {
        printf("(*bfun_kc)(2, 3) returns value of false, "
               "where bfun_kc is the compare function of hs1.\n");
    }

    bfun_kc = hash_multiset_key_comp(phms_hms2);
    (*bfun_kc)(&n_first, &n_second, &b_result);
    if(b_result)
    {
        printf("(*bfun_kc)(2, 3) returns value of true, "
               "where bfun_kc is the compare function of hs2.\n");
    }
    else
    {
        printf("(*bfun_kc)(2, 3) returns value of false, "
               "where bfun_kc is the compare function of hs2.\n");
    }

    hash_multiset_destroy(phms_hms1);
    hash_multiset_destroy(phms_hms2);

    return 0;
}
```

- **Output**

```
(*bfun_kc)(2, 3) returns value of true, where bfun_kc is the compare function of
hs1.
(*bfun_kc)(2, 3) returns value of false, where bfun_kc is the compare function of
hs2.
```

22. hash_multiset_less

测试第一个 hash_multiset_t 是否小于第二个 hash_multiset_t。

```
bool_t hash_multiset_less(
    const hash_multiset_t* cphmset_first,
    const hash_multiset_t* cphmset_second
);
```

- **Parameters**

  cphmset_first: 指向第一个 hash_multiset_t 类型的指针。
  cphmset_second: 指向第二个 hash_multiset_t 类型的指针。

- **Remarks**

  这个函数要求两个 hash_multiset_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);
    hash_multiset_t* phms_hms2 = create_hash_multiset(int);
    hash_multiset_t* phms_hms3 = create_hash_multiset(int);
    int i = 0;

    if(phms_hms1 == NULL || phms_hms2 == NULL || phms_hms3 == NULL)
    {
        return -1;
    }

    hash_multiset_init(phms_hms1);
    hash_multiset_init(phms_hms2);
    hash_multiset_init(phms_hms3);

    for(i = 0; i < 3; ++i)
    {
        hash_multiset_insert(phms_hms1, i);
        hash_multiset_insert(phms_hms2, i * i);
        hash_multiset_insert(phms_hms3, i - 1);
    }

    if(hash_multiset_less(phms_hms1, phms_hms2))
    {
        printf("The hash_multiset hs1 is less than the hash_multiset hs2.\n");
    }
    else
    {
        printf("The hash_multiset hs1 is not less than the hash_multiset hs2.\n");
    }

    if(hash_multiset_less(phms_hms1, phms_hms3))
    {
        printf("The hash_multiset hs1 is less than the hash_multiset hs3.\n");
    }
    else
    {
        printf("The hash_multiset hs1 is not less than the hash_multiset hs3.\n");
    }
```

```
    hash_multiset_destroy(phms_hms1);
    hash_multiset_destroy(phms_hms2);
    hash_multiset_destroy(phms_hms3);

    return 0;
}
```

- **Output**

```
The hash_multiset hs1 is less than the hash_multiset hs2.
The hash_multiset hs1 is not less than the hash_multiset hs3.
```

23. hash_multiset_less_equal

测试第一个 hash_multiset_t 是否小于等于第二个 hash_multiset_t。

```
bool_t hash_multiset_less_equal(
    const hash_multiset_t* cphmset_first,
    const hash_multiset_t* cphmset_second
);
```

- **Parameters**

cphmset_first: 指向第一个 hash_multiset_t 类型的指针。
cphmset_second: 指向第二个 hash_multiset_t 类型的指针。

- **Remarks**

这个函数要求两个 hash_multiset_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);
    hash_multiset_t* phms_hms2 = create_hash_multiset(int);
    hash_multiset_t* phms_hms3 = create_hash_multiset(int);
    hash_multiset_t* phms_hms4 = create_hash_multiset(int);
    int i = 0;

    if(phms_hms1 == NULL || phms_hms2 == NULL ||
       phms_hms3 == NULL || phms_hms4 == NULL)
    {
        return -1;
    }

    hash_multiset_init(phms_hms1);
    hash_multiset_init(phms_hms2);
```

```
    hash_multiset_init(phms_hms3);
    hash_multiset_init(phms_hms4);

    for(i = 0; i < 3; ++i)
    {
        hash_multiset_insert(phms_hms1, i);
        hash_multiset_insert(phms_hms2, i * i);
        hash_multiset_insert(phms_hms3, i - 1);
        hash_multiset_insert(phms_hms4, i);
    }

    if(hash_multiset_less_equal(phms_hms1, phms_hms2))
    {
        printf("The hash_multiset hs1 is less than "
               "or equal to the hash_multiset hs2.\n");
    }
    else
    {
        printf("The hash_multiset hs1 is greater than the hash_multiset hs2.\n");
    }

    if(hash_multiset_less_equal(phms_hms1, phms_hms3))
    {
        printf("The hash_multiset hs1 is less than or "
               "equal to the hash_multiset hs3.\n");
    }
    else
    {
        printf("The hash_multiset hs1 is greater than the hash_multiset hs3.\n");
    }

    if(hash_multiset_less_equal(phms_hms1, phms_hms4))
    {
        printf("The hash_multiset hs1 is less than or "
               "equal to the hash_multiset hs4.\n");
    }
    else
    {
        printf("The hash_multiset hs1 is greater than the hash_multiset hs4.\n");
    }

    hash_multiset_destroy(phms_hms1);
    hash_multiset_destroy(phms_hms2);
    hash_multiset_destroy(phms_hms3);
    hash_multiset_destroy(phms_hms4);

    return 0;
}
```

- **Output**

```
The hash_multiset hs1 is less than or equal to the hash_multiset hs2.
The hash_multiset hs1 is greater than the hash_multiset hs3.
The hash_multiset hs1 is less than or equal to the hash_multiset hs4.
```

24. hash_multiset_max_size

返回 hash_multiset_t 中能够保存数据数量的最大值。

```
size_t hash_multiset_max_size(
```

```
    const hash_multiset_t* cphmset_hmset
);
```

- **Parameters**

  cphmset_hmset: 指向 hash_multiset_t 类型的指针。

- **Remarks**

  这是一个与系统有关的常数。

- **Requirements**

  头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);

    if(phms_hms1 == NULL)
    {
        return -1;
    }

    hash_multiset_init(phms_hms1);

    printf("The maximum possible length of the hash_multiset hs1 is: %d.\n",
        hash_multiset_max_size(phms_hms1));

    hash_multiset_destroy(phms_hms1);

    return 0;
}
```

- **Output**

```
The maximum possible length of the hash_multiset hs1 is: 1073741823.
```

25. hash_multiset_not_equal

测试两个 hash_multiset_t 是否不等。

```
bool_t hash_multiset_not_equal(
    const hash_multiset_t* cphmset_first,
    const hash_multiset_t* cphmset_second
);
```

- **Parameters**

  cphmset_first: 指向第一个 hash_multiset_t 类型的指针。
  cphmset_second: 指向第二个 hash_multiset_t 类型的指针。

- **Remarks**

两个 hash_multiset_t 中的数据对应相等，并且数量相等，函数返回 false，否则返回 true。如果两个 hash_multiset_t 中的数据类型不同也认为不等。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);
    hash_multiset_t* phms_hms2 = create_hash_multiset(int);
    hash_multiset_t* phms_hms3 = create_hash_multiset(int);
    int i = 0;

    if(phms_hms1 == NULL || phms_hms2 == NULL || phms_hms3 == NULL)
    {
        return -1;
    }

    hash_multiset_init(phms_hms1);
    hash_multiset_init(phms_hms2);
    hash_multiset_init(phms_hms3);

    for(i = 0; i < 3; ++i)
    {
        hash_multiset_insert(phms_hms1, i);
        hash_multiset_insert(phms_hms2, i * i);
        hash_multiset_insert(phms_hms3, i);
    }

    if(hash_multiset_not_equal(phms_hms1, phms_hms2))
    {
        printf("The hash_multisets hs1 and hs2 are not equal.\n");
    }
    else
    {
        printf("The hash_multisets hs1 and hs2 are equal.\n");
    }

    if(hash_multiset_not_equal(phms_hms1, phms_hms3))
    {
        printf("The hash_multisets hs1 and hs3 are not equal.\n");
    }
    else
    {
        printf("The hash_multisets hs1 and hs3 are equal.\n");
    }

    hash_multiset_destroy(phms_hms1);
```

```
    hash_multiset_destroy(phms_hms2);
    hash_multiset_destroy(phms_hms3);

    return 0;
}
```

- **Output**

```
The hash_multisets hs1 and hs2 are not equal.
The hash_multisets hs1 and hs3 are equal.
```

26. hash_multiset_resize

重新设置 hash_multiset_t 中哈希表的存储单元数。

```
void hash_multiset_resize(
    hash_multiset_t* phmset_hmset,
    size_t t_resize
);
```

- **Parameters**

cphmset_hmset: 指向 hash_multiset_t 类型的指针。
t_resize: 哈希表存储单元的新数量。

- **Remarks**

当哈希表存储单元数量改变后，哈希表中的数据将被重新计算位置，所有的迭代器失效。当新的存储单元数量小于当前数量时，不做任何操作。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);

    if(phms_hms1 == NULL)
    {
        return -1;
    }

    hash_multiset_init(phms_hms1);

    printf("The bucket count of hash_multiset hs1 is: %d.\n",
        hash_multiset_bucket_count(phms_hms1));

    hash_multiset_resize(phms_hms1, 100);

    printf("The bucket count of hash_multiset hs1 is now: %d.\n",
```

```
        hash_multiset_bucket_count(phms_hms1));

    hash_multiset_destroy(phms_hms1);

    return 0;
}
```

- **Output**

```
The bucket count of hash_multiset hs1 is: 53.
The bucket count of hash_multiset hs1 is now: 193.
```

27. hash_multiset_size

返回 hash_multiset_t 中数据的数量。

```
size_t hash_multiset_size(
    const hash_multiset_t* cphmset_hmset
);
```

- **Parameters**

cphmset_hmset: 指向 hash_multiset_t 类型的指针。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);

    if(phms_hms1 == NULL)
    {
        return -1;
    }

    hash_multiset_init(phms_hms1);

    hash_multiset_insert(phms_hms1, 1);
    printf("The hash_multiset hs1 length is %d.\n",
        hash_multiset_size(phms_hms1));

    hash_multiset_insert(phms_hms1, 2);
    printf("The hash_multiset hs1 length is now %d.\n",
        hash_multiset_size(phms_hms1));

    hash_multiset_destroy(phms_hms1);

    return 0;
}
```

- **Output**

```
The hash_multiset hs1 length is 1.
The hash_multiset hs1 length is now 2.
```

28. hash_multiset_swap

交换两个 hash_multiset_t 中的内容。

```
void hash_multiset_swap(
    hash_multiset_t* phmset_first,
    hash_multiset_t* phmset_second
);
```

- **Parameters**

phmset_first: 指向第一个 hash_multiset_t 类型的指针。
phmset_second: 指向第二个 hash_multiset_t 类型的指针。

- **Remarks**

这个函数要求两个 hash_multiset_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);
    hash_multiset_t* phms_hms2 = create_hash_multiset(int);
    hash_multiset_iterator_t it_hs;

    if(phms_hms1 == NULL || phms_hms2 == NULL)
    {
        return -1;
    }

    hash_multiset_init(phms_hms1);
    hash_multiset_init(phms_hms2);

    hash_multiset_insert(phms_hms1, 10);
    hash_multiset_insert(phms_hms1, 20);
    hash_multiset_insert(phms_hms1, 30);
    hash_multiset_insert(phms_hms2, 100);
    hash_multiset_insert(phms_hms2, 200);

    printf("The original hash_multiset hs1 is:");
    for(it_hs = hash_multiset_begin(phms_hms1);
        !iterator_equal(it_hs, hash_multiset_end(phms_hms1));
```

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

    hash_multiset_swap(phms_hms1, phms_hms2);

    printf("After swapping with hs2, hash_multiset hs1 is:");
    for(it_hs = hash_multiset_begin(phms_hms1);
        !iterator_equal(it_hs, hash_multiset_end(phms_hms1));
        it_hs = iterator_next(it_hs))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_hs));
    }
    printf("\n");

    hash_multiset_destroy(phms_hms1);
    hash_multiset_destroy(phms_hms2);

    return 0;
}
```

- **Output**

```
The original hash_multiset hs1 is: 10 20 30
After swapping with hs2, hash_multiset hs1 is: 200 100
```

29. hash_multiset_value_comp

返回 hash_multiset_t 中使用的值比较规则。

```
binary_function_t hash_multiset_value_comp(
    const hash_multiset_t* cphmset_hmset
);
```

- **Parameters**

cphmset_hmset: 指向 hash_multiset_t 类型的指针。

- **Remarks**

由于 hash_multiset_t 中数据本身就是键，所以这个函数的返回值与 hash_multiset_key_comp()相同。

- **Requirements**

头文件 <cstl/chash_set.h>

- **Example**

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

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

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

```
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);
    hash_multiset_t* phms_hms2 = create_hash_multiset(int);
    binary_function_t bfun_vc = NULL;
    int n_first = 2;
    int n_second = 3;
    bool_t b_result = false;

    if(phms_hms1 == NULL || phms_hms2 == NULL)
    {
        return -1;
    }

    hash_multiset_init_ex(phms_hms1, 0, NULL, fun_less_int);
    hash_multiset_init_ex(phms_hms2, 0, NULL, fun_greater_int);

    bfun_vc = hash_multiset_value_comp(phms_hms1);
    (*bfun_vc)(&n_first, &n_second, &b_result);
    if(b_result)
    {
        printf("(*bfun_vc)(2, 3) returns value of true, "
               "where bfun_vc is the compare function of hs1.\n");
    }
    else
    {
        printf("(*bfun_vc)(2, 3) returns value of false, "
               "where bfun_vc is the compare function of hs1.\n");
    }

    bfun_vc = hash_multiset_value_comp(phms_hms2);
    (*bfun_vc)(&n_first, &n_second, &b_result);
    if(b_result)
    {
        printf("(*bfun_vc)(2, 3) returns value of true, "
               "where bfun_vc is the compare function of hs2.\n");
    }
    else
    {
        printf("(*bfun_vc)(2, 3) returns value of false, "
               "where bfun_vc is the compare function of hs2.\n");
    }

    hash_multiset_destroy(phms_hms1);
    hash_multiset_destroy(phms_hms2);

    return 0;
}
```

- **Output**

```
(*bfun_vc)(2, 3) returns value of true, where bfun_vc is the compare function of
hs1.
(*bfun_vc)(2, 3) returns value of false, where bfun_vc is the compare function of
hs2.
```

第十一节　　基于哈希结构的映射 hash_map_t

基于哈希结构的映射 hash_map_t 是关联容器，容器中保存的数据是 pair_t 类型。pair_t 的第一个数据是键，hash_map_t 中的数据就是根据这个键排序的，在 hash_map_t 中键不允许重复，也不可以直接或者间接修改键。pair_t

的第二个数据是值，值与键没有直接的关系，hash_map_t 中对于值的唯一性没有要求，值对于 hash_map_t 中的数据排序没有影响，可以直接或者间接修改值。

hash_map_t 的迭代器是双向迭代器，插入新的数据不会破坏原有的迭代器，删除一个数据的时候只有指向该数据的迭代器失效。在 hash_map_t 中查找，插入或者删除数据都是高效的，同时还可以使用键作为下标直接访问相应的值。

hash_map_t 中的数据保存在哈希表中，根据数据和指定的哈希函数计算数据在哈希表中的位置，同时根据键按照指定规则自动排序，默认规则是与键相关的小于操作，用户也可以在初始化时指定自定义的规则。hash_map_t 在数据的插入删除查找等操作上与基于平衡二叉树的关联容器相比效率更高，可以达到接近常数级别，但是数据不是完全有序的。

- **Typedefs**

hash_map_t	基于哈希结构的映射容器类型。
hash_map_iterator_t	基于哈希结构的映射容器迭代器类型。

- **Operation Functions**

create_hash_map	创建基于哈希结构的映射容器类型。
hash_map_assign	为基于哈希结构的映射容器迭代器类型赋值。
hash_map_at	使用键为下标随机访问基于哈希结构的映射容器中相应数据的值。
hash_map_begin	返回指向基于哈希结构的映射容器中的第一个数据的迭代器。
hash_map_bucket_count	返回基于哈希结构的映射容器使用的哈希表的存储单元个数。
hash_map_clear	删除基于哈希结构的映射容器中的所有数据。
hash_map_count	统计基于哈希结构的映射容器中包含指定数据的个数。
hash_map_destroy	销毁基于哈希结构的映射容器。
hash_map_empty	测试基于哈希结构的映射容器是否为空。
hash_map_end	返回指向基于哈希结构的映射容器末尾的迭代器。
hash_map_equal	测试两个基于哈希结构的映射容器是否相等。
hash_map_equal_range	返回基于哈希结构的映射容器中包含指定键的数据区间。
hash_map_erase	删除基于哈希结构的映射容器中包含指定键的数据。
hash_map_erase_pos	删除基于哈希结构的映射容器中指定位置的数据。
hash_map_erase_range	删除基于哈希结构的映射容器中指定的数据区间。
hash_map_find	在基于哈希结构的映射容器中查找包含指定键的数据。
hash_map_greater	测试第一个基于哈希结构的映射是否大于第二个基于哈希结构的映射。
hash_map_greater_equal	测试第一个基于哈希结构的映射是否大于等于第二个基于哈希结构的映射。
hash_map_hash	返回基于哈希结构的映射容器使用的哈希函数。
hash_map_init	初始化一个空的基于哈希结构的映射容器。
hash_map_init_copy	使用拷贝的方式初始化一个基于哈希结构的映射容器，所有内容都来自于源容器。
hash_map_init_copy_range	使用指定的数据区间初始化一个基于哈希结构的映射容器。
hash_map_init_copy_range_ex	使用指定的数据区间，哈希函数，比较规则，存储单元数量来初始化容器。
hash_map_init_ex	使用指定的哈希函数，比较规则，存储单元数量初始化一个空的基于哈希结构的映射。
hash_map_insert	向基于哈希结构的映射中插入一个数据。
hash_map_insert_range	向基于哈希结构的映射中插入一个数据区间。

hash_map_key_comp	返回基于哈希结构的映射容器使用的键比较规则。
hash_map_less	测试第一个基于哈希结构的映射是否小于第二个基于哈希结构的映射。
hash_map_less_equal	测试第一个基于哈希结构的映射是否小于等于第二个基于哈希结构的映射。
hash_map_max_size	返回基于哈希结构的映射容器中能够保存数据数量的最大值。
hash_map_not_equal	测试两个基于哈希结构的映射容器是否不等。
hash_map_resize	重新设置基于哈希结构的映射容器的哈希表存储单元个数。
hash_map_size	返回基于哈希结构的映射容器中保存数据的个数。
hash_map_swap	交换两个基于哈希结构的映射容器的内容。
hash_map_value_comp	返回基于哈希结构的映射容器使用的数据比较规则。

1. hash_map_t

基于哈希结构的映射容器类型。

- **Requirements**
  头文件 <cstl/chash_map.h>

- **Example**
  请参考 hash_map_t 类型的其他操作函数。

2. hash_map_iterator_t

基于哈希结构的映射容器的迭代器类型。

- **Remarks**
  hash_map_iterator_t 是双向迭代器类型，不能通过迭代器来修改容器中数据的键，但是可以修改数据的值。

- **Requirements**
  头文件 <cstl/chash_map.h>

- **Example**
  请参考 hash_map_t 类型的其他操作函数。

3. create_hash_map

创建 hash_map_t 容器。

```
hash_map_t* create_hash_map(
    type
);
```

- **Parameters**
  type: 数据类型描述。

- **Remarks**
  函数成功返回指向 hash_map_t 类型的指针，失败返回 NULL。

- **Requirements**
  头文件 <cstl/chash_map.h>

- **Example**
  请参考 hash_map_t 类型的其他操作函数。

4. hash_map_assign

为 hash_map_t 赋值。

```
void hash_map_assign(
    hash_map_t* phmap_dest,
    const hash_map_t* cphmap_src
);
```

- **Parameters**
  phmap_dest: 指向被赋值的 hash_map_t 类型的指针。
  cphmap_src: 指向赋值的 hash_map_t 类型的指针。

- **Remarks**
  要求两个 hash_map_t 类型保存的数据具有相同的类型，否则函数的行为未定义。

- **Requirements**
  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);
    hash_map_t* phm_hm2 = create_hash_map(int, int);
    pair_t* ppr_hm = create_pair(int, int);
    hash_map_iterator_t it_hm;

    if(phm_hm1 == NULL || phm_hm2 == NULL || ppr_hm == NULL)
    {
        return -1;
    }

    hash_map_init(phm_hm1);
    hash_map_init(phm_hm2);
    pair_init(ppr_hm);

    pair_make(ppr_hm, 1, 1);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, 3, 3);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, 5, 5);
    hash_map_insert(phm_hm1, ppr_hm);
```

```
    pair_make(ppr_hm, 100, 500);
    hash_map_insert(phm_hm2, ppr_hm);
    pair_make(ppr_hm, 200, 900);
    hash_map_insert(phm_hm2, ppr_hm);

    printf("hm1 =");
    for(it_hm = hash_map_begin(phm_hm1);
        !iterator_equal(it_hm, hash_map_end(phm_hm1));
        it_hm = iterator_next(it_hm))
    {
        printf("(%d, %d) ",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");

    hash_map_assign(phm_hm1, phm_hm2);
    printf("hm1 =");
    for(it_hm = hash_map_begin(phm_hm1);
        !iterator_equal(it_hm, hash_map_end(phm_hm1));
        it_hm = iterator_next(it_hm))
    {
        printf("(%d, %d) ",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");

    hash_map_destroy(phm_hm1);
    hash_map_destroy(phm_hm2);
    pair_destroy(ppr_hm);

    return 0;
}
```

- **Output**

```
hm1 =(1, 1) (3, 3) (5, 5)
hm1 =(200, 900) (100, 500)
```

5. hash_map_at

使用键作为下标随机访问 hash_map_t 中相应数据的值。

```
void* hash_map_at(
    hash_map_t* phmap_hmap,
    key
);
```

- **Parameters**

phmap_hmap: 指向 hash_map_t 类型的指针。
key: 指定的键。

- **Remarks**

这个操作函数通过指定的键来访问 hash_map_t 中相应数据的值，如果 hash_map_t 中包含这个键，那么就返回指向相应数据的值的指针，如果 hash_map_t 中不包含这个键，那么首先在 hash_map_t 中插入一个数据，这个数据以指定的键为键，以值的默认数据为值，然后返回指向这个数据的值的指针。

● **Requirements**

头文件 <cstl/chash_map.h>

● **Example**

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

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);
    pair_t* ppr_hm = create_pair(int, int);
    hash_map_iterator_t it_hm;

    if(phm_hm1 == NULL || ppr_hm == NULL)
    {
        return -1;
    }

    hash_map_init(phm_hm1);
    pair_init(ppr_hm);

    /*
     * Insert a data value of 10 with a key of 1
     * into a hash_map uing the at function
     */
    *(int*)hash_map_at(phm_hm1, 1) = 10;

    /*
     * Compare other ways to insert data into a hash_map
     */
    pair_make(ppr_hm, 2, 20);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, 3, 30);
    hash_map_insert(phm_hm1, ppr_hm);

    printf("The keys of the mapped elements are:");
    for(it_hm = hash_map_begin(phm_hm1);
        !iterator_equal(it_hm, hash_map_end(phm_hm1));
        it_hm = iterator_next(it_hm))
    {
        printf(" %d", *(int*)pair_first((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");

    printf("The values of the mapped elements are:");
    for(it_hm = hash_map_begin(phm_hm1);
        !iterator_equal(it_hm, hash_map_end(phm_hm1));
        it_hm = iterator_next(it_hm))
    {
        printf(" %d", *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");

    /*
```

```
     * If the key already exist, the at function
     * changes the value of the datum in the element
     */
    *(int*)hash_map_at(phm_hm1, 2) = 40;

    /*
     * The at function will also insert the value of the data
     * type's default if the value is unspecified
     */
    hash_map_at(phm_hm1, 5);

    printf("The keys of the mapped elements are:");
    for(it_hm = hash_map_begin(phm_hm1);
        !iterator_equal(it_hm, hash_map_end(phm_hm1));
        it_hm = iterator_next(it_hm))
    {
        printf(" %d", *(int*)pair_first((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");

    printf("The values of the mapped elements are:");
    for(it_hm = hash_map_begin(phm_hm1);
        !iterator_equal(it_hm, hash_map_end(phm_hm1));
        it_hm = iterator_next(it_hm))
    {
        printf(" %d", *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");

    hash_map_destroy(phm_hm1);
    pair_destroy(ppr_hm);

    return 0;
}
```

- **Output**

```
The keys of the mapped elements are: 1 2 3
The values of the mapped elements are: 10 20 30
The keys of the mapped elements are: 1 2 3 5
The values of the mapped elements are: 10 40 30 0
```

6. hash_map_begin

返回指向 hash_map_t 中第一个数据的迭代器。

```
hash_map_iterator_t hash_map_begin(
    const hash_map_t* cphmap_hmap
);
```

- **Parameters**

  cphmap_hmap: 指向 hash_map_t 类型的指针。

- **Remarks**

  如果 hash_map_t 为空，这个函数的返回值与 hash_map_end()相等。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);
    pair_t* ppr_hm = create_pair(int, int);
    hash_map_iterator_t it_hm;

    if(phm_hm1 == NULL || ppr_hm == NULL)
    {
        return -1;
    }

    hash_map_init(phm_hm1);
    pair_init(ppr_hm);

    pair_make(ppr_hm, 0, 0);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, 1, 1);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, 2, 4);
    hash_map_insert(phm_hm1, ppr_hm);

    it_hm = hash_map_begin(phm_hm1);
    printf("The first element of hm1 is (%d, %d).\n",
        *(int*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
        *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));

    hash_map_erase_pos(phm_hm1, hash_map_begin(phm_hm1));

    it_hm = hash_map_begin(phm_hm1);
    printf("The first element of hm1 is now (%d, %d).\n",
        *(int*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
        *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));

    hash_map_destroy(phm_hm1);
    pair_destroy(ppr_hm);

    return 0;
}
```

- **Output**

```
The first element of hm1 is (0, 0).
The first element of hm1 is now (1, 1).
```

7. hash_map_bucket_count

返回 hash_map_t 中哈希表的存储单元的个数。

```
size_t hash_map_bucket_count(
```

```
    const hash_map_t* cphmap_hmap
);
```

- **Parameters**

  cphmap_hmap: 指向 hash_map_t 类型的指针。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);
    hash_map_t* phm_hm2 = create_hash_map(int, int);

    if(phm_hm1 == NULL || phm_hm2 == NULL)
    {
        return -1;
    }

    hash_map_init(phm_hm1);
    hash_map_init_ex(phm_hm2, 100, NULL, NULL);

    printf("The default bucket count of hm1 is %d.\n",
        hash_map_bucket_count(phm_hm1));
    printf("The custom bucket count of hm2 is %d.\n",
        hash_map_bucket_count(phm_hm2));

    hash_map_destroy(phm_hm1);
    hash_map_destroy(phm_hm2);

    return 0;
}
```

- **Output**

```
The default bucket count of hm1 is 53.
The custom bucket count of hm2 is 193.
```

8. hash_map_clear

删除 hash_map_t 中的所有数据。

```
void hash_map_clear(
    hash_map_t* phmap_hmap
);
```

- **Parameters**

  cphmap_hmap: 指向 hash_map_t 类型的指针。

● **Requirements**

头文件 <cstl/chash_map.h>

● **Example**

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

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);
    pair_t* ppr_hm = create_pair(int, int);

    if(phm_hm1 == NULL || ppr_hm == NULL)
    {
        return -1;
    }

    hash_map_init(phm_hm1);
    pair_init(ppr_hm);

    pair_make(ppr_hm, 1, 1);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, 2, 4);
    hash_map_insert(phm_hm1, ppr_hm);

    printf("The size of the hash_map is initially %d.\n",
        hash_map_size(phm_hm1));

    hash_map_clear(phm_hm1);

    printf("The size of the hash_map after clearing is %d.\n",
        hash_map_size(phm_hm1));

    hash_map_destroy(phm_hm1);
    pair_destroy(ppr_hm);

    return 0;
}
```

● **Output**

```
The size of the hash_map is initially 2.
The size of the hash_map after clearing is 0.
```

9. hash_map_count

统计 hash_map_t 中包含指定键的数据的个数。

```
size_t hash_map_count(
    const hash_map_t* cphmap_hmap,
    key
);
```

- **Parameters**

  cphmap_hmap: 指向 hash_map_t 类型的指针。
  key: 指定的键。

- **Remarks**

  如果容器中没有包含指定键的数据返回 0， 否这返回包含指定键的数据的个数，hash_map_t 中的值是 1。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);
    pair_t* ppr_hm = create_pair(int, int);

    if(phm_hm1 == NULL || ppr_hm == NULL)
    {
        return -1;
    }

    pair_init(ppr_hm);
    hash_map_init(phm_hm1);

    pair_make(ppr_hm, 1, 1);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, 2, 1);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, 1, 4);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, 2, 1);
    hash_map_insert(phm_hm1, ppr_hm);

    /* Key must be unique in hash_map, so duplicates are ignored */
    printf("The number of elements in hm1 with a sort key of 1 is: %d.\n",
        hash_map_count(phm_hm1, 1));
    printf("The number of elements in hm1 with a sort key of 2 is: %d.\n",
        hash_map_count(phm_hm1, 2));
    printf("The number of elements in hm1 with a sort key of 3 is: %d.\n",
        hash_map_count(phm_hm1, 3));

    pair_destroy(ppr_hm);
    hash_map_destroy(phm_hm1);

    return 0;
}
```

- **Output**

```
The number of elements in hm1 with a sort key of 1 is: 1.
```

```
The number of elements in hm1 with a sort key of 2 is: 1.
The number of elements in hm1 with a sort key of 3 is: 0.
```

10. hash_map_destroy

销毁 hash_map_t 容器类型。

```
void hash_map_destroy(
    hash_map_t* phmap_hmap
);
```

- **Parameters**

  phmap_hmap: 指向 hash_map_t 类型的指针。

- **Remarks**

  hash_map_t 容器使用之后一定要销毁，否则 hash_map_t 申请的资源不会被释放。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

  请参考 hash_map_t 类型的其他操作函数。

11. hash_map_empty

测试 hash_map_t 是否为空。

```
bool_t hash_map_empty(
    const hash_map_t* cphmap_hmap
);
```

- **Parameters**

  cphmap_hmap: 指向 hash_map_t 类型的指针。

- **Remarks**

  hash_map_t 容器为空返回 true，否则返回 false。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);
    hash_map_t* phm_hm2 = create_hash_map(int, int);
    pair_t* ppr_hm = create_pair(int, int);
```

```
    if(phm_hm1 == NULL || phm_hm2 == NULL || ppr_hm == NULL)
    {
        return -1;
    }

    hash_map_init(phm_hm1);
    hash_map_init(phm_hm2);
    pair_init(ppr_hm);

    pair_make(ppr_hm, 1, 1);
    hash_map_insert(phm_hm1, ppr_hm);

    if(hash_map_empty(phm_hm1))
    {
        printf("The hash_map hm1 is empty.\n");
    }
    else
    {
        printf("The hash_map hm1 is not empty.\n");
    }

    if(hash_map_empty(phm_hm2))
    {
        printf("The hash_map hm2 is empty.\n");
    }
    else
    {
        printf("The hash_map hm2 is not empty.\n");
    }

    hash_map_destroy(phm_hm1);
    hash_map_destroy(phm_hm2);
    pair_destroy(ppr_hm);

    return 0;
}
```

- **Output**

```
The hash_map hm1 is not empty.
The hash_map hm2 is empty.
```

12. hash_map_end

返回指向 hash_map_t 末尾位置的迭代器。

```
hash_map_iterator_t hash_map_end(
    const hash_map_t* cphmap_hmap
);
```

- **Parameters**

  cphmap_hmap: 指向 hash_map_t 类型的指针。

- **Remarks**

  如果 hash_map_t 为空，这个函数的返回值与 hash_map_begin()相等。

- **Requirements**

头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);
    hash_map_iterator_t it_hm;
    pair_t* ppr_hm = create_pair(int, int);

    if(phm_hm1 == NULL || ppr_hm == NULL)
    {
        return -1;
    }

    hash_map_init(phm_hm1);
    pair_init(ppr_hm);

    pair_make(ppr_hm, 1, 10);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, 2, 20);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, 3, 30);
    hash_map_insert(phm_hm1, ppr_hm);

    it_hm = iterator_prev(hash_map_end(phm_hm1));
    printf("The value of last element of hm1 is (%d, %d).\n",
        *(int*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
        *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));

    hash_map_erase_pos(phm_hm1, it_hm);

    it_hm = iterator_prev(hash_map_end(phm_hm1));
    printf("The value of last element of hm1 is now (%d, %d).\n",
        *(int*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
        *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));

    hash_map_destroy(phm_hm1);
    pair_destroy(ppr_hm);

    return 0;
}
```

- **Output**

```
The value of last element of hm1 is (3, 30).
The value of last element of hm1 is now (2, 20).
```

13. hash_map_equal

测试两个 hash_map_t 是否相等。

```
bool_t hash_map_equal(
    const hash_map_t* cphmap_first,
    const hash_map_t* cphmap_second
);
```

- **Parameters**

  cphmap_first: 指向第一个 hash_map_t 类型的指针。
  cphmap_second: 指向第二个 hash_map_t 类型的指针。

- **Remarks**

  如果两个 hash_map_t 容器中的数据都对应相等，并且数据个数相等，则返回 true 否则返回 false，如果两个 hash_map_t 容器中保存的数据类型不同也认为是不等。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);
    hash_map_t* phm_hm2 = create_hash_map(int, int);
    hash_map_t* phm_hm3 = create_hash_map(int, int);
    pair_t* ppr_hm = create_pair(int, int);
    int i = 0;

    if(phm_hm1 == NULL || phm_hm2 == NULL || phm_hm3 == NULL || ppr_hm == NULL)
    {
        return -1;
    }

    hash_map_init(phm_hm1);
    hash_map_init(phm_hm2);
    hash_map_init(phm_hm3);
    pair_init(ppr_hm);

    for(i = 0; i < 3; ++i)
    {
        pair_make(ppr_hm, i, i);
        hash_map_insert(phm_hm1, ppr_hm);
        hash_map_insert(phm_hm3, ppr_hm);
        pair_make(ppr_hm, i, i * i);
        hash_map_insert(phm_hm2, ppr_hm);
    }

    if(hash_map_equal(phm_hm1, phm_hm2))
    {
        printf("The hash_maps hm1 and hm2 are equal.\n");
    }
    else
```

```
    {
        printf("The hash_maps hm1 and hm2 are not equal.\n");
    }

    if(hash_map_equal(phm_hm1, phm_hm3))
    {
        printf("The hash_maps hm1 and hm3 are equal.\n");
    }
    else
    {
        printf("The hash_maps hm1 and hm3 are not equal.\n");
    }

    hash_map_destroy(phm_hm1);
    hash_map_destroy(phm_hm2);
    hash_map_destroy(phm_hm3);
    pair_destroy(ppr_hm);

    return 0;
}
```

- **Output**

```
The hash_maps hm1 and hm2 are not equal.
The hash_maps hm1 and hm3 are equal.
```

14. hash_map_equal_range

返回 hash_map_t 中包含指定键的数据区间。

```
range_t hash_map_equal_range(
    const hash_map_t* cphmap_hmap,
    key
);
```

- **Parameters**

cphmap_hmap: 指向 hash_map_t 类型的指针。
key: 指定的键。

- **Remarks**

返回 hash_map_t 中包含拥有指定键的数据的数据区间[range_t.it_begin, range_t.it_end)，其中 it_begin 是指向拥有指定键的第一个数据的迭代器，it_end 指向拥有大于指定键的第一个数据的迭代器。如果 hash_map_t 中不包含拥有指定键的数据则 it_begin 与 it_end 相等。

- **Requirements**

头文件 <cstl/chash_map.h>

- **Example**

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

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

```
int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);
    pair_t* ppr_hm = create_pair(int, int);
    range_t r_hm;

    if(phm_hm1 == NULL || ppr_hm == NULL)
    {
        return -1;
    }

    hash_map_init(phm_hm1);
    pair_init(ppr_hm);

    pair_make(ppr_hm, 1, 10);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, 2, 20);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, 3, 30);
    hash_map_insert(phm_hm1, ppr_hm);

    r_hm = hash_map_equal_range(phm_hm1, 2);
    printf("The lower bound of the element with "
           "a key of 2 in the hash_map hm1 is: (%d, %d).\n",
           *(int*)pair_first((pair_t*)iterator_get_pointer(r_hm.it_begin)),
           *(int*)pair_second((pair_t*)iterator_get_pointer(r_hm.it_begin)));
    printf("The upper bound of the element with "
           "a key of 2 in the hash_map hm1 is: (%d, %d).\n",
           *(int*)pair_first((pair_t*)iterator_get_pointer(r_hm.it_end)),
           *(int*)pair_second((pair_t*)iterator_get_pointer(r_hm.it_end)));

    r_hm = hash_map_equal_range(phm_hm1, 4);
    if(iterator_equal(r_hm.it_begin, hash_map_end(phm_hm1)) &&
       iterator_equal(r_hm.it_end, hash_map_end(phm_hm1)))
    {
        printf("The hash_map hm1 doesn't have "
               "an element with a key less than 4.\n");
    }
    else
    {
        printf("The element of hash_map hm1 with a key >= 4 is (%d, %d).\n",
            *(int*)pair_first((pair_t*)iterator_get_pointer(r_hm.it_begin)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(r_hm.it_begin)));
    }

    hash_map_destroy(phm_hm1);
    pair_destroy(ppr_hm);

    return 0;
}
```

- **Output**

```
The lower bound of the element with a key of 2 in the hash_map hm1 is: (2, 20).
The upper bound of the element with a key of 2 in the hash_map hm1 is: (3, 30).
The hash_map hm1 doesn't have an element with a key less than 4.
```

15. hash_map_erase hash_map_erase_pos hash_map_erase_range

删除 hash_map_t 中的指定数据。

```
size_t hash_map_erase(
    hash_map_t* phmap_hmap,
    key
);

void hash_map_erase_pos(
    hash_map_t* phmap_hmap,
    hash_map_iterator_t it_pos
);

void hash_map_erase_range(
    hash_map_t* phmap_hmap,
    hash_map_iterator_t it_begin,
    hash_map_iterator_t it_end
);
```

- **Parameters**

phmap_hmap: 指向 hash_map_t 类型的指针。
key: 被删除的数据的键。
it_pos: 指向被删除的数据的迭代器。
it_begin: 指向被删除的数据区间开始位置的迭代器。
it_end: 指向被删除的数据区间末尾的迭代器。

- **Remarks**

第一个函数删除 hash_map_t 容器中包含指定键的数据，并返回删除数据的个数，如果容器中没有包含指定键的数据则返回 0。

第二个函数删除指定位置的数据。

第三个函数删除指定数据区间中的数据。

上面操作函数中的迭代器和数据区间都要求是有效的，无效的迭代器和数据区间将导致函数行为未定义。

- **Requirements**

头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);
    hash_map_t* phm_hm2 = create_hash_map(int, int);
    hash_map_t* phm_hm3 = create_hash_map(int, int);
    pair_t* ppr_hm = create_pair(int, int);
    hash_map_iterator_t it_hm;
    size_t t_count = 0;
    int i = 0;
```

```
    if(phm_hm1 == NULL || phm_hm2 == NULL || phm_hm3 == NULL || ppr_hm == NULL)
    {
        return -1;
    }

    hash_map_init(phm_hm1);
    hash_map_init(phm_hm2);
    hash_map_init(phm_hm3);
    pair_init(ppr_hm);

    for(i = 1; i < 5; ++i)
    {
        pair_make(ppr_hm, i, i);
        hash_map_insert(phm_hm1, ppr_hm);
        pair_make(ppr_hm, i, i * i);
        hash_map_insert(phm_hm2, ppr_hm);
        pair_make(ppr_hm, i, i - 1);
        hash_map_insert(phm_hm3, ppr_hm);
    }

    /* The first function removes an element at given position */
    hash_map_erase_pos(phm_hm1, iterator_next(hash_map_begin(phm_hm1)));
    printf("After the second element is deleted, the hash_map hm1 is:");
    for(it_hm = hash_map_begin(phm_hm1);
        !iterator_equal(it_hm, hash_map_end(phm_hm1));
        it_hm = iterator_next(it_hm))
    {
        printf(" (%d, %d)",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");

    /* The second function remove elements in the range [first, last) */
    hash_map_erase_range(phm_hm2, iterator_next(hash_map_begin(phm_hm2)),
        iterator_prev(hash_map_end(phm_hm2)));
    printf("After the middle two elements are deleted, the hash_map hm2 is:");
    for(it_hm = hash_map_begin(phm_hm2);
        !iterator_equal(it_hm, hash_map_end(phm_hm2));
        it_hm = iterator_next(it_hm))
    {
        printf(" (%d, %d)",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");

    /* The third function removes elements with a given key */
    t_count = hash_map_erase(phm_hm3, 2);
    printf("After the element with a key of 2 is deleted, the hash_map hm3 is:");
    for(it_hm = hash_map_begin(phm_hm3);
        !iterator_equal(it_hm, hash_map_end(phm_hm3));
        it_hm = iterator_next(it_hm))
    {
        printf(" (%d, %d)",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");
    /* The third function returns the number of elements returned */
```

```
    printf("The number of elements removed from hm3 is %d.\n", t_count);

    hash_map_destroy(phm_hm1);
    hash_map_destroy(phm_hm2);
    hash_map_destroy(phm_hm3);
    pair_destroy(ppr_hm);

    return 0;
}
```

- **Output**

```
After the second element is deleted, the hash_map hm1 is: (1, 1) (3, 3) (4, 4)
After the middle two elements are deleted, the hash_map hm2 is: (1, 1) (4, 16)
After the element with a key of 2 is deleted, the hash_map hm3 is: (1, 0) (3, 2) (4,
3)
The number of elements removed from hm3 is 1.
```

16. hash_map_find

查找 hash_map_t 中包含指定键的数据。

```
hash_map_iterator_t hash_map_find(
    const hash_map_t* cphmap_hmap,
    key
);
```

- **Parameters**

cphmap_hmap: 指向 hash_map_t 类型的指针。
key: 被删除的数据的键。

- **Remarks**

如果 hash_map_t 中存在包含指定键的数据，返回指向该数据的迭代器，否则返回 hash_map_end()。

- **Requirements**

头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);
    pair_t* ppr_hm = create_pair(int, int);
    hash_map_iterator_t it_hm;

    if(phm_hm1 == NULL || ppr_hm == NULL)
    {
        return -1;
    }

```

```
    hash_map_init(phm_hm1);
    pair_init(ppr_hm);

    pair_make(ppr_hm, 1, 10);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, 2, 20);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, 3, 30);
    hash_map_insert(phm_hm1, ppr_hm);

    it_hm = hash_map_find(phm_hm1, 2);
    printf("The element of hash_map hm1 with a key of 2 is: (%d, %d).\n",
        *(int*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
        *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));

    /* If no match is found for the key, end() is returned */
    it_hm = hash_map_find(phm_hm1, 4);
    if(iterator_equal(it_hm, hash_map_end(phm_hm1)))
    {
        printf("The hash_map hm1 doesn't have an element with a key of 4.\n");
    }
    else
    {
        printf("The element of hash_map hm1 with a key of 4 is: (%d, %d).\n",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));
    }

    hash_map_destroy(phm_hm1);
    pair_destroy(ppr_hm);

    return 0;
}
```

- **Output**

```
The element of hash_map hm1 with a key of 2 is: (2, 20).
The hash_map hm1 doesn't have an element with a key of 4.
```

17. hash_map_greater

测试第一个 hash_map_t 是否大于第二个 hash_map_t。

```
bool_t hash_map_greater(
    const hash_map_t* cphmap_first,
    const hash_map_t* cphmap_second
);
```

- **Parameters**

cphmap_first: 指向第一个 hash_map_t 类型的指针。
cphmap_second: 指向第二个 hash_map_t 类型的指针。

- **Remarks**

这个函数要求两个 hash_map_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

头文件 <cstl/chash_map.h>

* **Example**

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

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);
    hash_map_t* phm_hm2 = create_hash_map(int, int);
    hash_map_t* phm_hm3 = create_hash_map(int, int);
    pair_t* ppr_hm = create_pair(int, int);
    hash_map_iterator_t it_hm;
    int i = 0;

    if(phm_hm1 == NULL || phm_hm2 == NULL || phm_hm3 == NULL || ppr_hm == NULL)
    {
        return -1;
    }

    hash_map_init(phm_hm1);
    hash_map_init(phm_hm2);
    hash_map_init(phm_hm3);
    pair_init(ppr_hm);

    for(i = 1; i < 4; ++i)
    {
        pair_make(ppr_hm, i, i);
        hash_map_insert(phm_hm1, ppr_hm);
        pair_make(ppr_hm, i, i + 1);
        hash_map_insert(phm_hm2, ppr_hm);
        pair_make(ppr_hm, i + 1, i);
        hash_map_insert(phm_hm3, ppr_hm);
    }

    printf("The elements of hash_map hm1 are:");
    for(it_hm = hash_map_begin(phm_hm1);
        !iterator_equal(it_hm, hash_map_end(phm_hm1));
        it_hm = iterator_next(it_hm))
    {
        printf("(%d,%d) ",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");

    printf("The elements of hash_map hm2 are:");
    for(it_hm = hash_map_begin(phm_hm2);
        !iterator_equal(it_hm, hash_map_end(phm_hm2));
        it_hm = iterator_next(it_hm))
    {
        printf("(%d,%d) ",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");
```

```
    printf("The elements of hash_map hm3 are:");
    for(it_hm = hash_map_begin(phm_hm3);
        !iterator_equal(it_hm, hash_map_end(phm_hm3));
        it_hm = iterator_next(it_hm))
    {
        printf("(%d,%d) ",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");

    if(hash_map_greater(phm_hm1, phm_hm2))
    {
        printf("The hash_map hm1 is greater than the hash_map hm2.\n");
    }
    else
    {
        printf("The hash_map hm1 is not greater than the hash_map hm2.\n");
    }

    if(hash_map_greater(phm_hm1, phm_hm3))
    {
        printf("The hash_map hm1 is greater than the hash_map hm3.\n");
    }
    else
    {
        printf("The hash_map hm1 is not greater than the hash_map hm3.\n");
    }

    hash_map_destroy(phm_hm1);
    hash_map_destroy(phm_hm2);
    hash_map_destroy(phm_hm3);
    pair_destroy(ppr_hm);

    return 0;
}
```

- **Output**

```
The elements of hash_map hm1 are:(1,1) (2,2) (3,3)
The elements of hash_map hm2 are:(1,2) (2,3) (3,4)
The elements of hash_map hm3 are:(2,1) (3,2) (4,3)
The hash_map hm1 is not greater than the hash_map hm2.
The hash_map hm1 is not greater than the hash_map hm3.
```

18. hash_map_greater_equal

测试第一个 hash_map_t 是否大于等于第二个 hash_map_t。

```
bool_t hash_map_greater_equal(
    const hash_map_t* cphmap_first,
    const hash_map_t* cphmap_second
);
```

- **Parameters**

cphmap_first: 指向第一个 hash_map_t 类型的指针。
cphmap_second: 指向第二个 hash_map_t 类型的指针。

- **Remarks**

  这个函数要求两个 hash_map_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);
    hash_map_t* phm_hm2 = create_hash_map(int, int);
    hash_map_t* phm_hm3 = create_hash_map(int, int);
    hash_map_t* phm_hm4 = create_hash_map(int, int);
    pair_t* ppr_hm = create_pair(int, int);
    int i = 0;

    if(phm_hm1 == NULL || phm_hm2 == NULL || phm_hm3 == NULL || ppr_hm == NULL)
    {
        return -1;
    }

    hash_map_init(phm_hm1);
    hash_map_init(phm_hm2);
    hash_map_init(phm_hm3);
    hash_map_init(phm_hm4);
    pair_init(ppr_hm);

    for(i = 1; i < 3; ++i)
    {
        pair_make(ppr_hm, i, i);
        hash_map_insert(phm_hm1, ppr_hm);
        hash_map_insert(phm_hm4, ppr_hm);
        pair_make(ppr_hm, i, i * i);
        hash_map_insert(phm_hm2, ppr_hm);
        pair_make(ppr_hm, i, i - 1);
        hash_map_insert(phm_hm3, ppr_hm);
    }

    if(hash_map_greater_equal(phm_hm1, phm_hm2))
    {
        printf("The hash_map hm1 is greater than or equal to the hash_map hm2.\n");
    }
    else
    {
        printf("The hash_map hm1 is less than the hash_map hm2.\n");
    }

    if(hash_map_greater_equal(phm_hm1, phm_hm3))
    {
        printf("The hash_map hm1 is greater than or equal to the hash_map hm3.\n");
    }
```

```
    else
    {
        printf("The hash_map hm1 is less than the hash_map hm3.\n");
    }

    if(hash_map_greater_equal(phm_hm1, phm_hm4))
    {
        printf("The hash_map hm1 is greater than or equal to the hash_map hm4.\n");
    }
    else
    {
        printf("The hash_map hm1 is less than the hash_map hm4.\n");
    }

    hash_map_destroy(phm_hm1);
    hash_map_destroy(phm_hm2);
    hash_map_destroy(phm_hm3);
    hash_map_destroy(phm_hm4);
    pair_destroy(ppr_hm);

    return 0;
}
```

- **Output**

```
The hash_map hm1 is less than the hash_map hm2.
The hash_map hm1 is greater than or equal to the hash_map hm3.
The hash_map hm1 is greater than or equal to the hash_map hm4.
```

19. hash_map_hash

返回 hash_map_t 使用的哈希函数。

```
unary_function_t hash_map_hash(
    const hash_map_t* cphmap_hmap
);
```

- **Parameters**

  cphmap_hmap: 指向 hash_map_t 类型的指针。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);
    hash_map_t* phm_hm2 = create_hash_map(int, int);
```

```
    if(phm_hm1 == NULL || phm_hm2 == NULL)
    {
        return -1;
    }

    hash_map_init(phm_hm1);
    hash_map_init_ex(phm_hm2, 100, hash_func, NULL);

    if(hash_map_hash(phm_hm1) == hash_func)
    {
        printf("The hash function of hash_map hm1 is hash_func.\n");
    }
    else
    {
        printf("The hash function of hash_map hm1 is not hash_func.\n");
    }

    if(hash_map_hash(phm_hm2) == hash_func)
    {
        printf("The hash function of hash_map hm2 is hash_func.\n");
    }
    else
    {
        printf("The hash function of hash_map hm2 is not hash_func.\n");
    }

    hash_map_destroy(phm_hm1);
    hash_map_destroy(phm_hm2);

    return 0;
}

static void hash_func(const void* cpv_input, void* pv_output)
{
    *(int*)pv_output = *(int*)pair_first((pair_t*)cpv_input);
}
```

- **Output**

```
The hash function of hash_map hm1 is not hash_func.
The hash function of hash_map hm2 is hash_func.
```

20. hash_map_init hash_map_init_copy hash_map_init_copy_range hash_map_init_copy_range_ex hash_map_init_ex

初始化 hash_map_t 容器。

```
void hash_map_init(
    hash_map_t* phmap_hmap
);

void hash_map_init_copy(
    hash_map_t* phmap_hmap,
    const hash_map_t* cphmap_src
);

void hash_map_init_copy_range(
```

```
    hash_map_t* phmap_hmap,
    hash_map_iterator_t it_begin,
    hash_map_iterator_t it_end
);

void hash_map_init_copy_range_ex(
    hash_map_t* phmap_hmap,
    hash_map_iterator_t it_begin,
    hash_map_iterator_t it_end,
    size_t t_bucketcount,
    unary_function_t ufun_hash,
    binary_function_t bfun_compare
);

void hash_map_init_ex(
    hash_map_t* phmap_hmap,
    size_t t_bucketcount,
    unary_function_t ufun_hash,
    binary_function_t bfun_compare
);
```

- **Parameters**

phmap_hmap: 指向被初始化 hash_map_t 类型的指针。
cphmap_src: 指向用于初始化的 hash_map_t 类型的指针。
it_begin: 用于初始化的数据区间的开始位置。
it_end: 用于初始化的数据区间的末尾位置。
t_bucketcount: 哈希表中的存储单元个数。
ufun_hash: 自定义的哈希函数。
bfun_compare: 自定义比较规则。

- **Remarks**

第一个函数初始化一个空的 hash_map_t，使用默认的哈希函数和与键类型相关的小于操作函数作为默认的比较规则。

第二个函数使用一个源 hash_map_t 来初始化 hash_map_t，数据的内容，哈希函数和比较规则都从源 hash_map_t 复制。

第三个函数使用指定的数据区间初始化一个 hash_map_t，使用默认的哈希函数和与键类型相关的小于操作函数作为默认的比较规则。

第四个函数使用指定的数据区间初始化一个 hash_map_t，使用用户指定的哈希表存储单元个数，哈希函数和比较规则。

第五个函数初始化一个空的 hash_map_t，使用用户指定的哈希表存储单元个数，哈希函数和比较规则。

上面的函数要求迭代器和数据区间是有效的，无效的迭代器或数据区间导致函数的行为未定义。初始化函数根据用户指定的哈希表存储单元个数计算一个与用户指定的个数最接近的最佳哈希表存储单元个数。默认个数是 53 个，用户指定的个数小于等于 53 时都使用这个存储单元个数。

- **Requirements**

头文件 <cstl/chash_map.h>

- **Example**

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

```
#include <stdio.h>
#include <string.h>
#include <cstl/chash_map.h>
#include <cstl/cfunctional.h>

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm0 = create_hash_map(char*, int);
    hash_map_t* phm_hm1 = create_hash_map(char*, int);
    hash_map_t* phm_hm2 = create_hash_map(char*, int);
    hash_map_t* phm_hm3 = create_hash_map(char*, int);
    hash_map_t* phm_hm4 = create_hash_map(char*, int);
    hash_map_t* phm_hm5 = create_hash_map(char*, int);
    hash_map_iterator_t it_hm;
    pair_t* ppr_hm = create_pair(char*, int);

    if(phm_hm0 == NULL || phm_hm1 == NULL || phm_hm2 == NULL ||
       phm_hm3 == NULL || phm_hm4 == NULL || phm_hm5 == NULL ||
       ppr_hm == NULL)
    {
        return -1;
    }

    pair_init(ppr_hm);

    /* Create an empty hash_map hm0 of key type string */
    hash_map_init(phm_hm0);

    /*
     * Create an empty hash_map hm1 with the key comparison
     * function of less than, than insert 4 elements
     */
    hash_map_init_ex(phm_hm1, 0, _default_hash, fun_less_cstr);
    pair_make(ppr_hm, "one", 0);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, "two", 10);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, "three", 20);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, "four", 30);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, "five", 40);
    hash_map_insert(phm_hm1, ppr_hm);

    /*
     * Create an empty hash_map hm2 with the key comparison
     * function of greater than, then insert 2 elements
     */
    hash_map_init_ex(phm_hm2, 100, _default_hash, fun_greater_cstr);
    pair_make(ppr_hm, "one", 10);
    hash_map_insert(phm_hm2, ppr_hm);
    pair_make(ppr_hm, "two", 20);
    hash_map_insert(phm_hm2, ppr_hm);

    /* Create a copy, hash_map hm3, of hash_map hm1 */
    hash_map_init_copy(phm_hm3, phm_hm1);

    /* Create a hash_map hm4 by coping the range hm1[first, last) */
```

```
    hash_map_init_copy_range(phm_hm4,
        iterator_advance(hash_map_begin(phm_hm1), 2), hash_map_end(phm_hm1));

    /*
     * Create a hash_map hm5 by copying the range hm3 [first, last)
     * and with the key comparison function of less than
     */
    hash_map_init_copy_range_ex(phm_hm5, hash_map_begin(phm_hm3),
        hash_map_end(phm_hm3), 100, _default_hash, fun_less_cstr);

    printf("hm1 =");
    for(it_hm = hash_map_begin(phm_hm1);
        !iterator_equal(it_hm, hash_map_end(phm_hm1));
        it_hm = iterator_next(it_hm))
    {
        printf("(%s, %d) ",
            (char*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");

    printf("hm2 =");
    for(it_hm = hash_map_begin(phm_hm2);
        !iterator_equal(it_hm, hash_map_end(phm_hm2));
        it_hm = iterator_next(it_hm))
    {
        printf("(%s, %d) ",
            (char*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");

    printf("hm3 =");
    for(it_hm = hash_map_begin(phm_hm3);
        !iterator_equal(it_hm, hash_map_end(phm_hm3));
        it_hm = iterator_next(it_hm))
    {
        printf("(%s, %d) ",
            (char*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");

    printf("hm4 =");
    for(it_hm = hash_map_begin(phm_hm4);
        !iterator_equal(it_hm, hash_map_end(phm_hm4));
        it_hm = iterator_next(it_hm))
    {
        printf("(%s, %d) ",
            (char*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");

    printf("hm5 =");
    for(it_hm = hash_map_begin(phm_hm5);
        !iterator_equal(it_hm, hash_map_end(phm_hm5));
        it_hm = iterator_next(it_hm))
    {
        printf("(%s, %d) ",
```

```
            (char*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");

    hash_map_destroy(phm_hm0);
    hash_map_destroy(phm_hm1);
    hash_map_destroy(phm_hm2);
    hash_map_destroy(phm_hm3);
    hash_map_destroy(phm_hm4);
    hash_map_destroy(phm_hm5);
    pair_destroy(ppr_hm);

    return 0;
}

static void _default_hash(const void* cpv_input, void* pv_output)
{
    *(size_t*)pv_output = strlen((char*)pair_first((pair_t*)cpv_input));
}
```

- **Output**

```
hm1 =(one, 0) (two, 10) (four, 30) (five, 40) (three, 20)
hm2 =(one, 10) (two, 20)
hm3 =(one, 0) (two, 10) (four, 30) (five, 40) (three, 20)
hm4 =(five, 40) (three, 20) (four, 30)
hm5 =(one, 0) (two, 10) (four, 30) (five, 40) (three, 20)
```

21. hash_map_insert hash_map_insert_range

向 hash_map_t 中插入数据。

```
hash_map_iterator_t hash_map_insert(
    hash_map_t* phmap_hmap,
    const pair_t* cppair_pair
);

void hash_map_insert_range(
    hash_map_t* phmap_hmap,
    hash_map_iterator_t it_begin,
    hash_map_iterator_t it_end
);
```

- **Parameters**

phmap_hmap: 指向 hash_map_t 类型的指针。
cppair_pair: 插入的数据。
it_begin: 被插入的数据区间的开始位置。
it_end: 被插入的数据区间的末尾位置。

- **Remarks**

第一个函数向 hash_map_t 中插入一个指定的数据，成功后返回指向该数据的迭代器，如果 hash_map_t 中包含了该数据那么插入失败，返回 hash_map_end()。

第三个函数插入指定的数据区间。

上面的函数要求迭代器和数据区间是有效的，无效的迭代器或数据区间导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);
    hash_map_t* phm_hm2 = create_hash_map(int, int);
    pair_t* ppr_hm = create_pair(int, int);
    hash_map_iterator_t it_hm;

    if(phm_hm1 == NULL || phm_hm2 == NULL || ppr_hm == NULL)
    {
        return -1;
    }

    hash_map_init(phm_hm1);
    hash_map_init(phm_hm2);
    pair_init(ppr_hm);

    pair_make(ppr_hm, 1, 10);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, 2, 20);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, 3, 30);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, 4, 40);
    hash_map_insert(phm_hm1, ppr_hm);

    printf("The original elements of hm1 are:");
    for(it_hm = hash_map_begin(phm_hm1);
        !iterator_equal(it_hm, hash_map_end(phm_hm1));
        it_hm = iterator_next(it_hm))
    {
        printf(" (%d, %d)",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");

    pair_make(ppr_hm, 1, 10);
    it_hm = hash_map_insert(phm_hm1, ppr_hm);
    if(iterator_not_equal(it_hm, hash_map_end(phm_hm1)))
    {
        printf("The element (1, 10) was inserted in hm1 successfully.\n");
    }
    else
    {
        printf("The element (1, 10) alread exists in hm1.\n");
    }

    pair_make(ppr_hm, 10, 100);
```

```
    hash_map_insert(phm_hm2, ppr_hm);
    hash_map_insert_range(phm_hm2, iterator_next(hash_map_begin(phm_hm1)),
        iterator_prev(hash_map_end(phm_hm1)));
    printf("After the insertions, the elements of hm2 are:");
    for(it_hm = hash_map_begin(phm_hm2);
        !iterator_equal(it_hm, hash_map_end(phm_hm2));
        it_hm = iterator_next(it_hm))
    {
        printf(" (%d, %d)",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");

    hash_map_destroy(phm_hm1);
    hash_map_destroy(phm_hm2);
    pair_destroy(ppr_hm);

    return 0;
}
```

- **Output**

```
The original elements of hm1 are: (1, 10) (2, 20) (3, 30) (4, 40)
The element (1, 10) alread exists in hm1.
After the insertions, the elements of hm2 are: (2, 20) (3, 30) (10, 100)
```

22. hash_map_key_comp

返回 hash_map_t 中使用的键比较规则。

```
binary_function_t hash_map_key_comp(
    const hash_map_t* cphmap_hmap
);
```

- **Parameters**

cphmap_hmap: 指向 hash_map_t 类型的指针。

- **Remarks**

这个排序规则是针对数据中的键进行排序。

- **Requirements**

头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);
```

```
    hash_map_t* phm_hm2 = create_hash_map(int, int);
    binary_function_t bfun_kc = NULL;
    int n_first = 2;
    int n_second = 3;
    bool_t b_result = false;

    if(phm_hm1 == NULL || phm_hm2 == NULL)
    {
        return -1;
    }

    hash_map_init_ex(phm_hm1, 0, NULL, fun_less_int);
    hash_map_init_ex(phm_hm2, 0, NULL, fun_greater_int);

    bfun_kc = hash_map_key_comp(phm_hm1);
    (*bfun_kc)(&n_first, &n_second, &b_result);
    if(b_result)
    {
        printf("(*bfun_kc)(2, 3) returns value of true, "
               "where bfun_kc is the compare function of hm1.\n");
    }
    else
    {
        printf("(*bfun_kc)(2, 3) returns value of false, "
               "where bfun_kc is the compare function of hm1.\n");
    }

    bfun_kc = hash_map_key_comp(phm_hm2);
    (*bfun_kc)(&n_first, &n_second, &b_result);
    if(b_result)
    {
        printf("(*bfun_kc)(2, 3) returns value of true, "
               "where bfun_kc is the compare function of hm2.\n");
    }
    else
    {
        printf("(*bfun_kc)(2, 3) returns value of false, "
               "where bfun_kc is the compare function of hm2.\n");
    }

    hash_map_destroy(phm_hm1);
    hash_map_destroy(phm_hm2);

    return 0;
}
```

- **Output**

```
(*bfun_kc)(2, 3) returns value of true, where bfun_kc is the compare function of
hm1.
(*bfun_kc)(2, 3) returns value of false, where bfun_kc is the compare function of
hm2.
```

23. hash_map_less

测试第一个 hash_map_t 是否小于第二个 hash_map_t。

```
bool_t hash_map_less(
    const hash_map_t* cphmap_first,
```

```
    const hash_map_t* cphmap_second
);
```

- **Parameters**

  cphmap_first: 指向第一个 hash_map_t 类型的指针。
  cphmap_second: 指向第二个 hash_map_t 类型的指针。

- **Remarks**

  这个函数要求两个 hash_map_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);
    hash_map_t* phm_hm2 = create_hash_map(int, int);
    hash_map_t* phm_hm3 = create_hash_map(int, int);
    pair_t* ppr_hm = create_pair(int, int);
    hash_map_iterator_t it_hm;
    int i = 0;

    if(phm_hm1 == NULL || phm_hm2 == NULL || phm_hm3 == NULL || ppr_hm == NULL)
    {
        return -1;
    }

    hash_map_init(phm_hm1);
    hash_map_init(phm_hm2);
    hash_map_init(phm_hm3);
    pair_init(ppr_hm);

    for(i = 1; i < 4; ++i)
    {
        pair_make(ppr_hm, i, i);
        hash_map_insert(phm_hm1, ppr_hm);
        pair_make(ppr_hm, i, i + 1);
        hash_map_insert(phm_hm2, ppr_hm);
        pair_make(ppr_hm, i + 1, i);
        hash_map_insert(phm_hm3, ppr_hm);
    }

    printf("The elements of hash_map hm1 are:");
    for(it_hm = hash_map_begin(phm_hm1);
        !iterator_equal(it_hm, hash_map_end(phm_hm1));
        it_hm = iterator_next(it_hm))
    {
        printf("(%d,%d) ",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
```

```
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");

    printf("The elements of hash_map hm2 are:");
    for(it_hm = hash_map_begin(phm_hm2);
        !iterator_equal(it_hm, hash_map_end(phm_hm2));
        it_hm = iterator_next(it_hm))
    {
        printf("(%d,%d) ",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");

    printf("The elements of hash_map hm3 are:");
    for(it_hm = hash_map_begin(phm_hm3);
        !iterator_equal(it_hm, hash_map_end(phm_hm3));
        it_hm = iterator_next(it_hm))
    {
        printf("(%d,%d) ",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");

    if(hash_map_less(phm_hm1, phm_hm2))
    {
        printf("The hash_map hm1 is less than the hash_map hm2.\n");
    }
    else
    {
        printf("The hash_map hm1 is not less than the hash_map hm2.\n");
    }

    if(hash_map_less(phm_hm1, phm_hm3))
    {
        printf("The hash_map hm1 is less than the hash_map hm3.\n");
    }
    else
    {
        printf("The hash_map hm1 is not less than the hash_map hm3.\n");
    }

    hash_map_destroy(phm_hm1);
    hash_map_destroy(phm_hm2);
    hash_map_destroy(phm_hm3);
    pair_destroy(ppr_hm);

    return 0;
}
```

- **Output**

```
The elements of hash_map hm1 are:(1,1) (2,2) (3,3)
The elements of hash_map hm2 are:(1,2) (2,3) (3,4)
The elements of hash_map hm3 are:(2,1) (3,2) (4,3)
The hash_map hm1 is less than the hash_map hm2.
The hash_map hm1 is less than the hash_map hm3.
```

24. hash_map_less_equal

测试第一个 hash_map_t 是否小于等于 hash_map_t。

```
bool_t hash_map_less_equal(
    const hash_map_t* cphmap_first,
    const hash_map_t* cphmap_second
);
```

- **Parameters**
  cphmap_first: 指向第一个 hash_map_t 类型的指针。
  cphmap_second: 指向第二个 hash_map_t 类型的指针。

- **Remarks**
  这个函数要求两个 hash_map_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**
  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);
    hash_map_t* phm_hm2 = create_hash_map(int, int);
    hash_map_t* phm_hm3 = create_hash_map(int, int);
    hash_map_t* phm_hm4 = create_hash_map(int, int);
    pair_t* ppr_hm = create_pair(int, int);
    int i = 0;

    if(phm_hm1 == NULL || phm_hm2 == NULL || phm_hm3 == NULL || ppr_hm == NULL)
    {
        return -1;
    }

    hash_map_init(phm_hm1);
    hash_map_init(phm_hm2);
    hash_map_init(phm_hm3);
    hash_map_init(phm_hm4);
    pair_init(ppr_hm);

    for(i = 1; i < 3; ++i)
    {
        pair_make(ppr_hm, i, i);
        hash_map_insert(phm_hm1, ppr_hm);
        hash_map_insert(phm_hm4, ppr_hm);
        pair_make(ppr_hm, i, i * i);
        hash_map_insert(phm_hm2, ppr_hm);
        pair_make(ppr_hm, i, i - 1);
        hash_map_insert(phm_hm3, ppr_hm);
    }
```

```
    if(hash_map_less_equal(phm_hm1, phm_hm2))
    {
        printf("The hash_map hm1 is less than or equal to the hash_map hm2.\n");
    }
    else
    {
        printf("The hash_map hm1 is greater than the hash_map hm2.\n");
    }

    if(hash_map_less_equal(phm_hm1, phm_hm3))
    {
        printf("The hash_map hm1 is less than or equal to the hash_map hm3.\n");
    }
    else
    {
        printf("The hash_map hm1 is greater than the hash_map hm3.\n");
    }

    if(hash_map_less_equal(phm_hm1, phm_hm4))
    {
        printf("The hash_map hm1 is less than or equal to the hash_map hm4.\n");
    }
    else
    {
        printf("The hash_map hm1 is greater than the hash_map hm4.\n");
    }

    hash_map_destroy(phm_hm1);
    hash_map_destroy(phm_hm2);
    hash_map_destroy(phm_hm3);
    hash_map_destroy(phm_hm4);
    pair_destroy(ppr_hm);

    return 0;
}
```

- **Output**

```
The hash_map hm1 is less than or equal to the hash_map hm2.
The hash_map hm1 is greater than the hash_map hm3.
The hash_map hm1 is less than or equal to the hash_map hm4.
```

25. hash_map_max_size

返回 hash_map_t 中能够保存数据数量的最大值。

```
size_t hash_map_max_size(
    const hash_map_t* cphmap_hmap
);
```

- **Parameters**

cphmap_hmap: 指向 hash_map_t 类型的指针。

- **Remarks**

这是一个与系统相关的常数。

- **Requirements**

头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);

    if(phm_hm1 == NULL)
    {
        return -1;
    }

    hash_map_init(phm_hm1);

    printf("The maximum possible length of the hash_map hm1 is: %d.\n",
        hash_map_max_size(phm_hm1));

    hash_map_destroy(phm_hm1);

    return 0;
}
```

- **Output**

```
The maximum possible length of the hash_map hm1 is: 7895160.
```

26. hash_map_not_equal

测试两个 hash_map_t 是否不等。

```
bool_t hash_map_not_equal(
    const hash_map_t* cphmap_first,
    const hash_map_t* cphmap_second
);
```

- **Parameters**

cphmap_first: 指向第一个 hash_map_t 类型的指针。
cphmap_second: 指向第二个 hash_map_t 类型的指针。

- **Remarks**

如果两个 hash_map_t 容器中的数据都对应相等，并且数据个数相等，则返回 false 否则返回 true，如果两个 hash_map_t 容器中保存的数据类型不同也认为是不等。

- **Requirements**

头文件 <cstl/chash_map.h>

- **Example**

```
/*
```

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

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);
    hash_map_t* phm_hm2 = create_hash_map(int, int);
    hash_map_t* phm_hm3 = create_hash_map(int, int);
    pair_t* ppr_hm = create_pair(int, int);
    int i = 0;

    if(phm_hm1 == NULL || phm_hm2 == NULL || phm_hm3 == NULL || ppr_hm == NULL)
    {
        return -1;
    }

    hash_map_init(phm_hm1);
    hash_map_init(phm_hm2);
    hash_map_init(phm_hm3);
    pair_init(ppr_hm);

    for(i = 0; i < 3; ++i)
    {
        pair_make(ppr_hm, i, i);
        hash_map_insert(phm_hm1, ppr_hm);
        hash_map_insert(phm_hm3, ppr_hm);
        pair_make(ppr_hm, i, i * i);
        hash_map_insert(phm_hm2, ppr_hm);
    }

    if(hash_map_not_equal(phm_hm1, phm_hm2))
    {
        printf("The hash_maps hm1 and hm2 are not equal.\n");
    }
    else
    {
        printf("The hash_maps hm1 and hm2 are equal.\n");
    }

    if(hash_map_not_equal(phm_hm1, phm_hm3))
    {
        printf("The hash_maps hm1 and hm3 are not equal.\n");
    }
    else
    {
        printf("The hash_maps hm1 and hm3 are equal.\n");
    }

    hash_map_destroy(phm_hm1);
    hash_map_destroy(phm_hm2);
    hash_map_destroy(phm_hm3);
    pair_destroy(ppr_hm);

    return 0;
}
```

- **Output**

```
The hash_maps hm1 and hm2 are not equal.
The hash_maps hm1 and hm3 are equal.
```

27. hash_map_resize

重新设置 hash_map_t 中哈希表的存储单元个数。

```
void hash_map_resize(
    hash_map_t* phmap_hmap,
    size_t t_resize
);
```

- **Parameters**

cphmap_hmap: 指向 hash_map_t 类型的指针。
t_resize: 哈希表存储单元的新数量。

- **Remarks**

当哈希表存储单元数量改变后，哈希表中的数据将被重新计算位置，所有的迭代器失效。当新的存储单元数量小于当前数量时，不做任何操作。

- **Requirements**

头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);

    if(phm_hm1 == NULL)
    {
        return -1;
    }

    hash_map_init(phm_hm1);

    printf("The bucket count of hash_map hm1 is: %d.\n",
        hash_map_bucket_count(phm_hm1));

    hash_map_resize(phm_hm1, 100);

    printf("The bucket count of hash_map hm1 is now: %d.\n",
        hash_map_bucket_count(phm_hm1));

    hash_map_destroy(phm_hm1);

    return 0;
}
```

- **Output**

```
The bucket count of hash_map hm1 is: 53.
The bucket count of hash_map hm1 is now: 193.
```

28. hash_map_size

返回 hash_map_t 中数据的数量。

```
size_t hash_map_size(
    const hash_map_t* cphmap_hmap
);
```

- **Parameters**

  cphmap_hmap: 指向 hash_map_t 类型的指针。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);
    pair_t* ppr_hm = create_pair(int, int);

    if(phm_hm1 == NULL)
    {
        return -1;
    }

    hash_map_init(phm_hm1);
    pair_init(ppr_hm);

    pair_make(ppr_hm, 1, 1);
    hash_map_insert(phm_hm1, ppr_hm);
    printf("The hash_map hm1 length is %d.\n", hash_map_size(phm_hm1));

    pair_make(ppr_hm, 2, 4);
    hash_map_insert(phm_hm1, ppr_hm);
    printf("The hash_map hm1 length is now %d.\n", hash_map_size(phm_hm1));

    hash_map_destroy(phm_hm1);
    pair_destroy(ppr_hm);

    return 0;
}
```

- **Output**

```
The hash_map hm1 length is 1.
The hash_map hm1 length is now 2.
```

29. hash_map_swap

交换两个 hash_map_t 中的内容。

```
void hash_map_swap(
    hash_map_t* phmap_first,
    hash_map_t* phmap_second
);
```

- **Parameters**

  phmap_first: 指向第一个 hash_map_t 类型的指针。
  phmap_second: 指向第二个 hash_map_t 类型的指针。

- **Remarks**

  这个函数要求两个 hash_map_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);
    hash_map_t* phm_hm2 = create_hash_map(int, int);
    pair_t* ppr_hm = create_pair(int, int);
    hash_map_iterator_t it_hm;

    if(phm_hm1 == NULL || phm_hm2 == NULL || ppr_hm == NULL)
    {
        return -1;
    }

    hash_map_init(phm_hm1);
    hash_map_init(phm_hm2);
    pair_init(ppr_hm);

    pair_make(ppr_hm, 1, 10);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, 2, 20);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, 3, 30);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, 10, 100);
    hash_map_insert(phm_hm2, ppr_hm);
    pair_make(ppr_hm, 20, 200);
    hash_map_insert(phm_hm2, ppr_hm);
```

```
    printf("The orighinal hash_map hm1 is:");
    for(it_hm = hash_map_begin(phm_hm1);
        !iterator_equal(it_hm, hash_map_end(phm_hm1));
        it_hm = iterator_next(it_hm))
    {
        printf(" (%d, %d)",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");

    hash_map_swap(phm_hm1, phm_hm2);

    printf("After swapping with hm2, hash_map hm1 is:");
    for(it_hm = hash_map_begin(phm_hm1);
        !iterator_equal(it_hm, hash_map_end(phm_hm1));
        it_hm = iterator_next(it_hm))
    {
        printf(" (%d, %d)",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hm)));
    }
    printf("\n");

    hash_map_destroy(phm_hm1);
    hash_map_destroy(phm_hm2);
    pair_destroy(ppr_hm);

    return 0;
}
```

- **Output**

```
The orighinal hash_map hm1 is: (1, 10) (2, 20) (3, 30)
After swapping with hm2, hash_map hm1 is: (10, 100) (20, 200)
```

30. hash_map_value_comp

返回 hash_map_t 使用的数据比较规则。

```
binary_function_t hash_map_value_comp(
    const hash_map_t* cphmap_hmap
);
```

- **Parameters**

cphmap_hmap: 指向 hash_map_t 类型的指针。

- **Remarks**

这个规则是针对数据本身的比较规则而不是键或者值。

- **Requirements**

头文件 <cstl/chash_map.h>

- **Example**

```
/*
 * hash_map_value_comp.c
```

```
 * compile with : -lcstl
 */

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

int main(int argc, char* argv[])
{
    hash_map_t* phm_hm1 = create_hash_map(int, int);
    pair_t* ppr_hm = create_pair(int, int);
    binary_function_t bfun_vc = NULL;
    bool_t b_result = false;
    hash_map_iterator_t it_hm1;
    hash_map_iterator_t it_hm2;

    if(phm_hm1 == NULL || ppr_hm == NULL)
    {
        return -1;
    }

    pair_init(ppr_hm);
    hash_map_init_ex(phm_hm1, 100, NULL, fun_less_int);

    pair_make(ppr_hm, 1, 10);
    hash_map_insert(phm_hm1, ppr_hm);
    pair_make(ppr_hm, 2, 5);
    hash_map_insert(phm_hm1, ppr_hm);

    it_hm1 = hash_map_find(phm_hm1, 1);
    it_hm2 = hash_map_find(phm_hm1, 2);
    bfun_vc = hash_map_value_comp(phm_hm1);

    (*bfun_vc)(iterator_get_pointer(it_hm1),
        iterator_get_pointer(it_hm2), &b_result);
    if(b_result)
    {
        printf("The element (1, 10) precedes the element (2, 5).\n");
    }
    else
    {
        printf("The element (1, 10) does not precedes the element (2, 5).\n");
    }

    (*bfun_vc)(iterator_get_pointer(it_hm2),
        iterator_get_pointer(it_hm1), &b_result);
    if(b_result)
    {
        printf("The element (2, 5) precedes the element (1, 10).\n");
    }
    else
    {
        printf("The element (2, 5) does not precedes the element (1, 10).\n");
    }

    pair_destroy(ppr_hm);
    hash_map_destroy(phm_hm1);

    return 0;
}
```

- **Output**

```
The element (1, 10) precedes the element (2, 5).
The element (2, 5) does not precedes the element (1, 10).
```

第十二节　　基于哈希结构的多重映射 hash_multimap_t

基于哈希结构的多重映射 hash_multimap_t 是关联容器，容器中保存的数据是 pair_t 类型。pair_t 的第一个数据是键，hash_multimap_t 中的数据就是根据这个键排序的，不可以直接或者间接修改键。pair_t 的第二个数据是值，值与键没有直接的关系，值对于 hash_multimap_t 中的数据排序没有影响，可以直接或者间接修改值。

hash_multimap_t 的迭代器是双向迭代器，插入新的数据不会破坏原有的迭代器，删除一个数据的时候只有指向该数据的迭代器失效。在 hash_multimap_t 中查找，插入或者删除数据都是高效的。

hash_multimap_t 中的数据保存在哈希表中，根据数据和指定的哈希函数计算数据在哈希表中的位置，同时根据键按照指定规则自动排序，默认规则是与键相关的小于操作，用户也可以在初始化时指定自定义的规则。hash_multimap_t 在数据的插入删除查找等操作上与基于平衡二叉树的关联容器相比效率更高，可以达到接近常数级别，但是数据不是完全有序的。

- **Typedefs**

hash_multimap_t	基于哈希结构的多重映射容器类型。
hash_multimap_iterator_t	基于哈希结构的多重映射容器迭代器类型。

- **Operation Functions**

create_hash_multimap	创建基于哈希结构的多重映射容器类型。
hash_multimap_assign	为基于哈希结构的多重映射容器迭代器类型赋值。
hash_multimap_begin	返回指向基于哈希结构的多重映射中第一个数据的迭代器。
hash_multimap_bucket_count	返回基于哈希结构的多重映射使用的哈希表的存储单元个数。
hash_multimap_clear	删除基于哈希结构的多重映射中包含指定键的数据。
hash_multimap_count	统计基于哈希结构的多重映射中包含指定键的数据的个数。
hash_multimap_destroy	销毁基于哈希结构的多重映射容器。
hash_multimap_empty	测试基于哈希结构的多重映射容器是否为空。
hash_multimap_end	返回指向基于哈希结构的多重映射容器末尾位置的迭代器。
hash_multimap_equal	测试两个基于哈希结构的多重映射容器是否相等。
hash_multimap_equal_range	返回基于哈希结构的多重映射容器中包含指定键的数据区间。
hash_multimap_erase	删除基于哈希结构的多重映射中包含指定键的数据。
hash_multimap_erase_pos	删除基于哈希结构的多重映射容器中指定位置的数据。
hash_multimap_erase_range	删除基于哈希结构的多重映射容器中的指定数据区间。
hash_multimap_find	查找基于哈希结构的多重映射容器中包含指定键的数据。
hash_multimap_greater	测试第一个基于哈希结构的多重映射是否大于第二个基于哈希结构的多重映射。
hash_multimap_greater_equal	测试第一个基于哈希结构的多重映射是否大于等于第二个容器。
hash_multimap_hash	返回基于哈希结构的多重映射使用的哈希函数。
hash_multimap_init	初始化一个空的基于哈希结构的多重映射。

hash_multimap_init_copy	使用拷贝的方式初始化一个基于哈希结构的多重映射。
hash_multimap_init_copy_range	使用指定的数据区间初始化一个基于哈希结构的多重映射。
hash_multimap_init_copy_range_ex	使用指定的数据区间，哈希函数，比较规则和存储单元数初始化容器。
hash_multimap_init_ex	使用指定的哈希函数，比较规则和存储单元数初始化一个空的容器。
hash_multimap_insert	向基于哈希结构的多重映射容器中插入数据。
hash_multimap_insert_range	向基于哈希结构的多重映射容器中插入数据区间。
hash_multimap_key_comp	返回基于哈希结构的多重映射容器使用的键比较规则。
hash_multimap_less	测试第一个基于哈希结构的多重映射是否小于第二个容器。
hash_multimap_less_equal	测试第一个基于哈希结构的多重映射是否小于等于第二个容器。
hash_multimap_max_size	返回基于哈希结构的多重映射容器中能够保存的数据数量的最大值。
hash_multimap_not_equal	测试两个基于哈希结构的多重映射容器是否不等。
hash_multimap_resize	重新设置基于哈希结构的多重映射容器使用的哈希表的存储单元个数。
hash_multimap_size	返回基于哈希结构的多重映射容器中保存的数据的个数。
hash_multimap_swap	交换两个基于哈希结构的多重映射容器中的内容。
hash_multimap_value_comp	返回基于哈希结构的多重映射容器使用的数据比较规则。

1. hash_multimap_t

基于哈希结构的多重映射容器类型。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

  请参考 hash_multimap_t 类型的其他操作函数。

2. hash_multimap_iterator_t

基于哈希结构的多重映射容器的迭代器类型。

- **Remarks**

  hash_multimap_iterator_t 是双向迭代器类型，不能通过迭代器来修改容器中数据的键，但是可以修改数据的值。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

  请参考 hash_multimap_t 类型的其他操作函数。

3. create_hash_multimap

创建 hash_multimap_t 容器类型。

```
hash_multimap_t* create_hash_multimap(
    type
```

```
);
```

- **Parameters**

  type: 数据类型描述。

- **Remarks**

  函数成功返回指向 hash_multimap_t 类型的指针，失败返回 NULL。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

  请参考 hash_multimap_t 类型的其他操作函数。

4. hash_multimap_assign

为 hash_multimap_t 赋值。

```
void hash_multimap_assign(
    hash_multimap_t* phmmap_dest,
    const hash_multimap_t* cphmmap_src
);
```

- **Parameters**

  phmmap_dest: 指向被赋值的 hash_multimap_t 类型的指针。
  cphmmap_src: 指向赋值的 hash_multimap_t 类型的指针。

- **Remarks**

  要求两个 hash_multimap_t 类型保存的数据具有相同的类型，否则函数的行为未定义。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multimap_t* phmm_hmm1 = create_hash_multimap(int, int);
    hash_multimap_t* phmm_hmm2 = create_hash_multimap(int, int);
    pair_t* ppr_hmm = create_pair(int, int);
    hash_multimap_iterator_t it_hmm;

    if(phmm_hmm1 == NULL || phmm_hmm2 == NULL || ppr_hmm == NULL)
    {
        return -1;
    }

    hash_multimap_init(phmm_hmm1);
```

```
    hash_multimap_init(phmm_hmm2);
    pair_init(ppr_hmm);

    pair_make(ppr_hmm, 1, 1);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, 3, 3);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, 5, 5);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);

    pair_make(ppr_hmm, 100, 500);
    hash_multimap_insert(phmm_hmm2, ppr_hmm);
    pair_make(ppr_hmm, 200, 900);
    hash_multimap_insert(phmm_hmm2, ppr_hmm);

    printf("hmm1 =");
    for(it_hmm = hash_multimap_begin(phmm_hmm1);
        !iterator_equal(it_hmm, hash_multimap_end(phmm_hmm1));
        it_hmm = iterator_next(it_hmm))
    {
        printf("(%d, %d) ",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));
    }
    printf("\n");

    hash_multimap_assign(phmm_hmm1, phmm_hmm2);
    printf("hmm1 =");
    for(it_hmm = hash_multimap_begin(phmm_hmm1);
        !iterator_equal(it_hmm, hash_multimap_end(phmm_hmm1));
        it_hmm = iterator_next(it_hmm))
    {
        printf("(%d, %d) ",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));
    }
    printf("\n");

    hash_multimap_destroy(phmm_hmm1);
    hash_multimap_destroy(phmm_hmm2);
    pair_destroy(ppr_hmm);

    return 0;
}
```

- **Output**

```
hmm1 =(1, 1) (3, 3) (5, 5)
hmm1 =(200, 900) (100, 500)
```

5. hash_multimap_begin

返回指向 hash_multimap_t 中第一个数据的迭代器。

```
hash_multimap_iterator_t hash_multimap_begin(
    const hash_multimap_t* cphmmap_hmmap
);
```

- **Parameters**

cphmmap_hmmap: 指向 hash_multimap_t 类型的指针。

- **Remarks**

如果 hash_multimap_t 为空，这个函数的返回值与 hash_multimap_end()相等。

- **Requirements**

头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multimap_t* phmm_hmm1 = create_hash_multimap(int, int);
    pair_t* ppr_hmm = create_pair(int, int);
    hash_multimap_iterator_t it_hmm;

    if(phmm_hmm1 == NULL || ppr_hmm == NULL)
    {
        return -1;
    }

    hash_multimap_init(phmm_hmm1);
    pair_init(ppr_hmm);

    pair_make(ppr_hmm, 0, 0);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, 1, 1);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, 2, 4);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);

    it_hmm = hash_multimap_begin(phmm_hmm1);
    printf("The first element of hmm1 is (%d, %d).\n",
        *(int*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
        *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));

    hash_multimap_erase_pos(phmm_hmm1, hash_multimap_begin(phmm_hmm1));

    it_hmm = hash_multimap_begin(phmm_hmm1);
    printf("The first element of hmm1 is now (%d, %d).\n",
        *(int*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
        *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));

    hash_multimap_destroy(phmm_hmm1);
    pair_destroy(ppr_hmm);

    return 0;
}
```

- **Output**

```
The first element of hmm1 is (0, 0).
```

```
The first element of hmm1 is now (1, 1).
```

6. hash_multimap_bucket_count

返回 hash_multimap_t 中哈希表存储单元的个数。

```
size_t hash_multimap_bucket_count(
    const hash_multimap_t* cphmmap_hmmap
);
```

- **Parameters**

cphmmap_hmmap: 指向 hash_multimap_t 类型的指针。

- **Requirements**

头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multimap_t* phmm_hmm1 = create_hash_multimap(int, int);
    hash_multimap_t* phmm_hmm2 = create_hash_multimap(int, int);

    if(phmm_hmm1 == NULL || phmm_hmm2 == NULL)
    {
        return -1;
    }

    hash_multimap_init(phmm_hmm1);
    hash_multimap_init_ex(phmm_hmm2, 100, NULL, NULL);

    printf("The default bucket count of hmm1 is %d.\n",
        hash_multimap_bucket_count(phmm_hmm1));
    printf("The custom bucket count of hmm2 is %d.\n",
        hash_multimap_bucket_count(phmm_hmm2));

    hash_multimap_destroy(phmm_hmm1);
    hash_multimap_destroy(phmm_hmm2);

    return 0;
}
```

- **Output**

```
The default bucket count of hmm1 is 53.
The custom bucket count of hmm2 is 193.
```

7. hash_multimap_clear

删除 hash_multimap_t 中所有的数据。

```
void hash_multimap_clear(
    hash_multimap_t* phmmap_hmmap
);
```

- **Parameters**

  cphmmap_hmmap: 指向 hash_multimap_t 类型的指针。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multimap_t* phmm_hmm1 = create_hash_multimap(int, int);
    pair_t* ppr_hmm = create_pair(int, int);

    if(phmm_hmm1 == NULL || ppr_hmm == NULL)
    {
        return -1;
    }

    hash_multimap_init(phmm_hmm1);
    pair_init(ppr_hmm);

    pair_make(ppr_hmm, 1, 1);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, 2, 4);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);

    printf("The size of the hash_multimap is initially %d.\n",
        hash_multimap_size(phmm_hmm1));

    hash_multimap_clear(phmm_hmm1);

    printf("The size of the hash_multimap after clearing is %d.\n",
        hash_multimap_size(phmm_hmm1));

    hash_multimap_destroy(phmm_hmm1);
    pair_destroy(ppr_hmm);

    return 0;
}
```

- **Output**

```
The size of the hash_multimap is initially 2.
The size of the hash_multimap after clearing is 0.
```

8. hash_multimap_count

统计 hash_multimap_t 中包含指定键的数据个数。

```
size_t hash_multimap_count(
    const hash_multimap_t* cphmap_hmmap,
    key
);
```

- **Parameters**

  cphmmap_hmmap: 指向 hash_multimap_t 类型的指针。
  key: 指定的键。

- **Remarks**

  如果容器中没有包含指定键的数据返回 0， 否这返回包含指定键的数据的个数。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multimap_t* phmm_hmm1 = create_hash_multimap(int, int);
    pair_t* ppr_hmm = create_pair(int, int);

    if(phmm_hmm1 == NULL || ppr_hmm == NULL)
    {
        return -1;
    }

    pair_init(ppr_hmm);
    hash_multimap_init(phmm_hmm1);

    pair_make(ppr_hmm, 1, 1);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, 2, 1);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, 1, 4);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, 2, 1);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);

    /* Key must be unique in hash_multimap, so duplicates are ignored */
    printf("The number of elements in hmm1 with a sort key of 1 is: %d.\n",
        hash_multimap_count(phmm_hmm1, 1));
    printf("The number of elements in hmm1 with a sort key of 2 is: %d.\n",
        hash_multimap_count(phmm_hmm1, 2));
    printf("The number of elements in hmm1 with a sort key of 3 is: %d.\n",
```

```
        hash_multimap_count(phmm_hmm1, 3));

    pair_destroy(ppr_hmm);
    hash_multimap_destroy(phmm_hmm1);

    return 0;
}
```

- **Output**

```
The number of elements in hmm1 with a sort key of 1 is: 2.
The number of elements in hmm1 with a sort key of 2 is: 2.
The number of elements in hmm1 with a sort key of 3 is: 0.
```

9. hash_multimap_destroy

销毁 hash_multimap_t 容器类型。

```
void hash_multimap_destroy(
    hash_multimap_t* phmmap_hmmap
);
```

- **Parameters**

  phmmap_hmmap: 指向 hash_multimap_t 类型的指针。

- **Remarks**

  hash_multimap_t 容器使用之后一定要销毁，否则 hash_multimap_t 申请的资源不会被释放。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

  请参考 hash_multimap_t 类型的其他操作函数。

10. hash_multimap_empty

测试 hash_multimap_t 是否为空。

```
bool_t hash_multimap_empty(
    const hash_multimap_t* cphmmap_hmmap
);
```

- **Parameters**

  cphmmap_hmmap: 指向 hash_multimap_t 类型的指针。

- **Remarks**

  hash_multimap_t 容器为空返回 true，否则返回 false。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

```
/*
 * hash_multimap_empty.c
```

```
 * compile with : -lcstl
 */

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

int main(int argc, char* argv[])
{
    hash_multimap_t* phmm_hmm1 = create_hash_multimap(int, int);
    hash_multimap_t* phmm_hmm2 = create_hash_multimap(int, int);
    pair_t* ppr_hmm = create_pair(int, int);

    if(phmm_hmm1 == NULL || phmm_hmm2 == NULL || ppr_hmm == NULL)
    {
        return -1;
    }

    hash_multimap_init(phmm_hmm1);
    hash_multimap_init(phmm_hmm2);
    pair_init(ppr_hmm);

    pair_make(ppr_hmm, 1, 1);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);

    if(hash_multimap_empty(phmm_hmm1))
    {
        printf("The hash_multimap hmm1 is empty.\n");
    }
    else
    {
        printf("The hash_multimap hmm1 is not empty.\n");
    }

    if(hash_multimap_empty(phmm_hmm2))
    {
        printf("The hash_multimap hmm2 is empty.\n");
    }
    else
    {
        printf("The hash_multimap hmm2 is not empty.\n");
    }

    hash_multimap_destroy(phmm_hmm1);
    hash_multimap_destroy(phmm_hmm2);
    pair_destroy(ppr_hmm);

    return 0;
}
```

- **Output**

```
The hash_multimap hmm1 is not empty.
The hash_multimap hmm2 is empty.
```

11. hash_multimap_end

返回指向 hash_multimap_t 容器末尾位置的迭代器。

```
hash_multimap_iterator_t hash_multimap_end(
```

```
    const hash_multimap_t* cphmmap_hmmap
);
```

- **Parameters**
  cphmmap_hmmap:　　指向 hash_multimap_t 类型的指针。

- **Remarks**
  如果 hash_multimap_t 为空，这个函数的返回值与 hash_multimap_begin()相等。

- **Requirements**
  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multimap_t* phmm_hmm1 = create_hash_multimap(int, int);
    hash_multimap_iterator_t it_hmm;
    pair_t* ppr_hmm = create_pair(int, int);

    if(phmm_hmm1 == NULL || ppr_hmm == NULL)
    {
        return -1;
    }

    hash_multimap_init(phmm_hmm1);
    pair_init(ppr_hmm);

    pair_make(ppr_hmm, 1, 10);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, 2, 20);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, 3, 30);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);

    it_hmm = iterator_prev(hash_multimap_end(phmm_hmm1));
    printf("The value of last element of hmm1 is (%d, %d).\n",
        *(int*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
        *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));

    hash_multimap_erase_pos(phmm_hmm1, it_hmm);

    it_hmm = iterator_prev(hash_multimap_end(phmm_hmm1));
    printf("The value of last element of hmm1 is now (%d, %d).\n",
        *(int*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
        *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));

    hash_multimap_destroy(phmm_hmm1);
    pair_destroy(ppr_hmm);

    return 0;
```

```
}
```

- **Output**

```
The value of last element of hmm1 is (3, 30).
The value of last element of hmm1 is now (2, 20).
```

12. hash_multimap_equal

测试两个 hash_multimap_t 是否相等。

```
bool_t hash_multimap_equal(
    const hash_multimap_t* cphmmap_first,
    const hash_multimap_t* cphmmap_second
);
```

- **Parameters**

cphmmap_first: 指向第一个 hash_multimap_t 类型的指针。
cphmmap_second: 指向第二个 hash_multimap_t 类型的指针。

- **Remarks**

如果两个 hash_multimap_t 容器中的数据都对应相等，并且数据个数相等，则返回 true 否则返回 false，如果两个 hash_multimap_t 容器中保存的数据类型不同也认为是不等。

- **Requirements**

头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multimap_t* phmm_hmm1 = create_hash_multimap(int, int);
    hash_multimap_t* phmm_hmm2 = create_hash_multimap(int, int);
    hash_multimap_t* phmm_hmm3 = create_hash_multimap(int, int);
    pair_t* ppr_hmm = create_pair(int, int);
    int i = 0;

    if(phmm_hmm1 == NULL || phmm_hmm2 == NULL ||
       phmm_hmm3 == NULL || ppr_hmm == NULL)
    {
        return -1;
    }

    hash_multimap_init(phmm_hmm1);
    hash_multimap_init(phmm_hmm2);
    hash_multimap_init(phmm_hmm3);
    pair_init(ppr_hmm);

    for(i = 0; i < 3; ++i)
    {
```

```
        pair_make(ppr_hmm, i, i);
        hash_multimap_insert(phmm_hmm1, ppr_hmm);
        hash_multimap_insert(phmm_hmm3, ppr_hmm);
        pair_make(ppr_hmm, i, i * i);
        hash_multimap_insert(phmm_hmm2, ppr_hmm);
    }

    if(hash_multimap_equal(phmm_hmm1, phmm_hmm2))
    {
        printf("The hash_multimaps hmm1 and hmm2 are equal.\n");
    }
    else
    {
        printf("The hash_multimaps hmm1 and hmm2 are not equal.\n");
    }

    if(hash_multimap_equal(phmm_hmm1, phmm_hmm3))
    {
        printf("The hash_multimaps hmm1 and hmm3 are equal.\n");
    }
    else
    {
        printf("The hash_multimaps hmm1 and hmm3 are not equal.\n");
    }

    hash_multimap_destroy(phmm_hmm1);
    hash_multimap_destroy(phmm_hmm2);
    hash_multimap_destroy(phmm_hmm3);
    pair_destroy(ppr_hmm);

    return 0;
}
```

- **Output**

```
The hash_multimaps hmm1 and hmm2 are not equal.
The hash_multimaps hmm1 and hmm3 are equal.
```

13. hash_multimap_equal_range

返回 hash_multimap_t 中包含指定键的数据区间。

```
range_t _hash_multimap_equal_range(
    const hash_multimap_t* cphmmap_hmmap,
    key
);
```

- **Parameters**

cphmmap_hmmap: 指向 hash_multimap_t 类型的指针。
key: 指定的键。

- **Remarks**

返回 hash_multimap_t 中包含拥有指定键的数据的数据区间[range_t.it_begin, range_t.it_end)，其中 it_begin 是指向拥有指定键的第一个数据的迭代器，it_end 指向拥有大于指定键的第一个数据的迭代器。如果 hash_multimap_t 中不包含拥有指定键的数据则 it_begin 与 it_end 相等。

- **Requirements**

头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multimap_t* phmm_hmm1 = create_hash_multimap(int, int);
    pair_t* ppr_hmm = create_pair(int, int);
    range_t r_hmm;

    if(phmm_hmm1 == NULL || ppr_hmm == NULL)
    {
        return -1;
    }

    hash_multimap_init(phmm_hmm1);
    pair_init(ppr_hmm);

    pair_make(ppr_hmm, 1, 10);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, 2, 20);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, 3, 30);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);

    r_hmm = hash_multimap_equal_range(phmm_hmm1, 2);
    printf("The lower bound of the element with "
           "a key of 2 in the hash_multimap hmm1 is: (%d, %d).\n",
           *(int*)pair_first((pair_t*)iterator_get_pointer(r_hmm.it_begin)),
           *(int*)pair_second((pair_t*)iterator_get_pointer(r_hmm.it_begin)));
    printf("The upper bound of the element with "
           "a key of 2 in the hash_multimap hmm1 is: (%d, %d).\n",
           *(int*)pair_first((pair_t*)iterator_get_pointer(r_hmm.it_end)),
           *(int*)pair_second((pair_t*)iterator_get_pointer(r_hmm.it_end)));

    r_hmm = hash_multimap_equal_range(phmm_hmm1, 4);
    if(iterator_equal(r_hmm.it_begin, hash_multimap_end(phmm_hmm1)) &&
       iterator_equal(r_hmm.it_end, hash_multimap_end(phmm_hmm1)))
    {
        printf("The hash_multimap hmm1 doesn't have "
               "an element with a key less than 4.\n");
    }
    else
    {
        printf("The element of hash_multimap hmm1 with a key >= 4 is (%d, %d).\n",
            *(int*)pair_first((pair_t*)iterator_get_pointer(r_hmm.it_begin)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(r_hmm.it_begin)));
    }

    hash_multimap_destroy(phmm_hmm1);
    pair_destroy(ppr_hmm);

    return 0;
```

```
}
```

- **Output**

```
The lower bound of the element with a key of 2 in the hash_multimap hmm1 is: (2, 20).
The upper bound of the element with a key of 2 in the hash_multimap hmm1 is: (3, 30).
The hash_multimap hmm1 doesn't have an element with a key less than 4.
```

14. hash_multimap_erase hash_multimap_erase_pos hash_multimap_erase_range

删除 hash_multimap_t 中的数据。

```
size_t hash_multimap_erase(
    hash_multimap_t* phmmap_hmmap,
    key
);

void hash_multimap_erase_pos(
    hash_multimap_t* phmmap_hmmap,
    hash_multimap_iterator_t it_pos
);

void hash_multimap_erase_range(
    hash_multimap_t* phmmap_hmmap,
    hash_multimap_iterator_t it_begin,
    hash_multimap_iterator_t it_end
);
```

- **Parameters**

phmmap_hmmap: 指向 hash_multimap_t 类型的指针。
key: 被删除的数据的键。
it_pos: 指向被删除的数据的迭代器。
it_begin: 指向被删除的数据区间开始位置的迭代器。
it_end: 指向被删除的数据区间末尾的迭代器。

- **Remarks**

第一个函数删除 hash_multimap_t 容器中包含指定键的数据，并返回删除数据的个数，如果容器中没有包含指定键的数据则返回 0。
第二个函数删除指定位置的数据。
第三个函数删除指定数据区间中的数据。
上面操作函数中的迭代器和数据区间都要求是有效的，无效的迭代器和数据区间将导致函数行为未定义。

- **Requirements**

头文件 <cstl/chash_map.h>

- **Example**

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

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

```
int main(int argc, char* argv[])
{
    hash_multimap_t* phmm_hmm1 = create_hash_multimap(int, int);
    hash_multimap_t* phmm_hmm2 = create_hash_multimap(int, int);
    hash_multimap_t* phmm_hmm3 = create_hash_multimap(int, int);
    pair_t* ppr_hmm = create_pair(int, int);
    hash_multimap_iterator_t it_hmm;
    size_t t_count = 0;
    int i = 0;

    if(phmm_hmm1 == NULL || phmm_hmm2 == NULL ||
       phmm_hmm3 == NULL || ppr_hmm == NULL)
    {
        return -1;
    }

    hash_multimap_init(phmm_hmm1);
    hash_multimap_init(phmm_hmm2);
    hash_multimap_init(phmm_hmm3);
    pair_init(ppr_hmm);

    for(i = 1; i < 5; ++i)
    {
        pair_make(ppr_hmm, i, i);
        hash_multimap_insert(phmm_hmm1, ppr_hmm);
        pair_make(ppr_hmm, i, i * i);
        hash_multimap_insert(phmm_hmm2, ppr_hmm);
        pair_make(ppr_hmm, i, i - 1);
        hash_multimap_insert(phmm_hmm3, ppr_hmm);
    }

    /* The first function removes an element at given position */
    hash_multimap_erase_pos(phmm_hmm1,
        iterator_next(hash_multimap_begin(phmm_hmm1)));
    printf("After the second element is deleted, the hash_multimap hmm1 is:");
    for(it_hmm = hash_multimap_begin(phmm_hmm1);
        !iterator_equal(it_hmm, hash_multimap_end(phmm_hmm1));
        it_hmm = iterator_next(it_hmm))
    {
        printf(" (%d, %d)",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));
    }
    printf("\n");

    /* The second function remove elements in the range [first, last) */
    hash_multimap_erase_range(phmm_hmm2,
        iterator_next(hash_multimap_begin(phmm_hmm2)),
        iterator_prev(hash_multimap_end(phmm_hmm2)));
    printf("After the middle two elements are deleted, the hash_multimap hmm2 is:");
    for(it_hmm = hash_multimap_begin(phmm_hmm2);
        !iterator_equal(it_hmm, hash_multimap_end(phmm_hmm2));
        it_hmm = iterator_next(it_hmm))
    {
        printf(" (%d, %d)",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));
    }
    printf("\n");
```

```
    /* The third function removes elements with a given key */
    t_count = hash_multimap_erase(phmm_hmm3, 2);
    printf("After the element with a key of 2 is deleted, "
           "the hash_multimap hmm3 is:");
    for(it_hmm = hash_multimap_begin(phmm_hmm3);
        !iterator_equal(it_hmm, hash_multimap_end(phmm_hmm3));
        it_hmm = iterator_next(it_hmm))
    {
        printf(" (%d, %d)",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));
    }
    printf("\n");
    /* The third function returns the number of elements returned */
    printf("The number of elements removed from hmm3 is %d.\n", t_count);

    hash_multimap_destroy(phmm_hmm1);
    hash_multimap_destroy(phmm_hmm2);
    hash_multimap_destroy(phmm_hmm3);
    pair_destroy(ppr_hmm);

    return 0;
}
```

- **Output**

```
After the second element is deleted, the hash_multimap hmm1 is: (1, 1) (3, 3) (4, 4)
After the middle two elements are deleted, the hash_multimap hmm2 is: (1, 1) (4, 16)
After the element with a key of 2 is deleted, the hash_multimap hmm3 is: (1, 0) (3,
2) (4, 3)
The number of elements removed from hmm3 is 1.
```

15. hash_multimap_find

在 hash_multimap_t 中查找包含指定键的数据。

```
hash_multimap_iterator_t _hash_multimap_find(
    const hash_multimap_t* cphmmap_hmmap,
    key
);
```

- **Parameters**

cphmmap_hmmap: 指向 hash_multimap_t 类型的指针。
key: 被删除的数据的键。

- **Remarks**

如果 hash_multimap_t 中存在包含指定键的数据，返回指向该数据的迭代器，否则返回 hash_multimap_end()。

- **Requirements**

头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multimap_t* phmm_hmm1 = create_hash_multimap(int, int);
    pair_t* ppr_hmm = create_pair(int, int);
    hash_multimap_iterator_t it_hmm;

    if(phmm_hmm1 == NULL || ppr_hmm == NULL)
    {
        return -1;
    }

    hash_multimap_init(phmm_hmm1);
    pair_init(ppr_hmm);

    pair_make(ppr_hmm, 1, 10);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, 2, 20);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, 3, 30);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);

    it_hmm = hash_multimap_find(phmm_hmm1, 2);
    printf("The element of hash_multimap hmm1 with a key of 2 is: (%d, %d).\n",
        *(int*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
        *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));

    /* If no match is found for the key, end() is returned */
    it_hmm = hash_multimap_find(phmm_hmm1, 4);
    if(iterator_equal(it_hmm, hash_multimap_end(phmm_hmm1)))
    {
        printf("The hash_multimap hmm1 doesn't have an element with a key of 4.\n");
    }
    else
    {
        printf("The element of hash_multimap hmm1 with a key of 4 is: (%d, %d).\n",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));
    }

    hash_multimap_destroy(phmm_hmm1);
    pair_destroy(ppr_hmm);

    return 0;
}
```

- **Output**

```
The element of hash_multimap hmm1 with a key of 2 is: (2, 20).
The hash_multimap hmm1 doesn't have an element with a key of 4.
```

16. hash_multimap_greater

测试第一个 hash_multimap_t 是否大于第二个 hash_multimap_t。

```
bool_t hash_multimap_greater(
```

```
    const hash_multimap_t* cphmmap_first,
    const hash_multimap_t* cphmmap_second
);
```

- **Parameters**

  cphmmap_first: 指向第一个 hash_multimap_t 类型的指针。
  cphmmap_second: 指向第二个 hash_multimap_t 类型的指针。

- **Remarks**

  这个函数要求两个 hash_multimap_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multimap_t* phmm_hmm1 = create_hash_multimap(int, int);
    hash_multimap_t* phmm_hmm2 = create_hash_multimap(int, int);
    hash_multimap_t* phmm_hmm3 = create_hash_multimap(int, int);
    pair_t* ppr_hmm = create_pair(int, int);
    hash_multimap_iterator_t it_hmm;
    int i = 0;

    if(phmm_hmm1 == NULL || phmm_hmm2 == NULL ||
       phmm_hmm3 == NULL || ppr_hmm == NULL)
    {
        return -1;
    }

    hash_multimap_init(phmm_hmm1);
    hash_multimap_init(phmm_hmm2);
    hash_multimap_init(phmm_hmm3);
    pair_init(ppr_hmm);

    for(i = 1; i < 4; ++i)
    {
        pair_make(ppr_hmm, i, i);
        hash_multimap_insert(phmm_hmm1, ppr_hmm);
        pair_make(ppr_hmm, i, i + 1);
        hash_multimap_insert(phmm_hmm2, ppr_hmm);
        pair_make(ppr_hmm, i + 1, i);
        hash_multimap_insert(phmm_hmm3, ppr_hmm);
    }

    printf("The elements of hash_multimap hmm1 are:");
    for(it_hmm = hash_multimap_begin(phmm_hmm1);
        !iterator_equal(it_hmm, hash_multimap_end(phmm_hmm1));
        it_hmm = iterator_next(it_hmm))
    {
```

```
        printf("(%d,%d) ",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));
    }
    printf("\n");

    printf("The elements of hash_multimap hmm2 are:");
    for(it_hmm = hash_multimap_begin(phmm_hmm2);
        !iterator_equal(it_hmm, hash_multimap_end(phmm_hmm2));
        it_hmm = iterator_next(it_hmm))
    {
        printf("(%d,%d) ",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));
    }
    printf("\n");

    printf("The elements of hash_multimap hmm3 are:");
    for(it_hmm = hash_multimap_begin(phmm_hmm3);
        !iterator_equal(it_hmm, hash_multimap_end(phmm_hmm3));
        it_hmm = iterator_next(it_hmm))
    {
        printf("(%d,%d) ",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));
    }
    printf("\n");

    if(hash_multimap_greater(phmm_hmm1, phmm_hmm2))
    {
        printf("The hash_multimap hmm1 is greater than the hash_multimap hmm2.\n");
    }
    else
    {
        printf("The hash_multimap hmm1 is not "
               "greater than the hash_multimap hmm2.\n");
    }

    if(hash_multimap_greater(phmm_hmm1, phmm_hmm3))
    {
        printf("The hash_multimap hmm1 is greater than the hash_multimap hmm3.\n");
    }
    else
    {
        printf("The hash_multimap hmm1 is not "
               "greater than the hash_multimap hmm3.\n");
    }

    hash_multimap_destroy(phmm_hmm1);
    hash_multimap_destroy(phmm_hmm2);
    hash_multimap_destroy(phmm_hmm3);
    pair_destroy(ppr_hmm);

    return 0;
}
```

- **Output**

```
The elements of hash_multimap hmm1 are:(1,1) (2,2) (3,3)
The elements of hash_multimap hmm2 are:(1,2) (2,3) (3,4)
```

```
The elements of hash_multimap hmm3 are:(2,1) (3,2) (4,3)
The hash_multimap hmm1 is not greater than the hash_multimap hmm2.
The hash_multimap hmm1 is not greater than the hash_multimap hmm3.
```

17. hash_multimap_greater_equal

测试第一个 hash_multimap_t 是否大于等于第二个 hash_multimap_t。

```
bool_t hash_multimap_greater_equal(
    const hash_multimap_t* cphmmap_first,
    const hash_multimap_t* cphmmap_second
);
```

- **Parameters**

  cphmmap_first: 指向第一个 hash_multimap_t 类型的指针。
  cphmmap_second: 指向第二个 hash_multimap_t 类型的指针。

- **Remarks**

  这个函数要求两个 hash_multimap_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multimap_t* phmm_hmm1 = create_hash_multimap(int, int);
    hash_multimap_t* phmm_hmm2 = create_hash_multimap(int, int);
    hash_multimap_t* phmm_hmm3 = create_hash_multimap(int, int);
    hash_multimap_t* phmm_hmm4 = create_hash_multimap(int, int);
    pair_t* ppr_hmm = create_pair(int, int);
    int i = 0;

    if(phmm_hmm1 == NULL || phmm_hmm2 == NULL ||
       phmm_hmm3 == NULL || ppr_hmm == NULL)
    {
        return -1;
    }

    hash_multimap_init(phmm_hmm1);
    hash_multimap_init(phmm_hmm2);
    hash_multimap_init(phmm_hmm3);
    hash_multimap_init(phmm_hmm4);
    pair_init(ppr_hmm);

    for(i = 1; i < 3; ++i)
    {
        pair_make(ppr_hmm, i, i);
        hash_multimap_insert(phmm_hmm1, ppr_hmm);
```

```
        hash_multimap_insert(phmm_hmm4, ppr_hmm);
        pair_make(ppr_hmm, i, i * i);
        hash_multimap_insert(phmm_hmm2, ppr_hmm);
        pair_make(ppr_hmm, i, i - 1);
        hash_multimap_insert(phmm_hmm3, ppr_hmm);
    }

    if(hash_multimap_greater_equal(phmm_hmm1, phmm_hmm2))
    {
        printf("The hash_multimap hmm1 is greater than or "
               "equal to the hash_multimap hmm2.\n");
    }
    else
    {
        printf("The hash_multimap hmm1 is less than the hash_multimap hmm2.\n");
    }

    if(hash_multimap_greater_equal(phmm_hmm1, phmm_hmm3))
    {
        printf("The hash_multimap hmm1 is greater than or "
               "equal to the hash_multimap hmm3.\n");
    }
    else
    {
        printf("The hash_multimap hmm1 is less than the hash_multimap hmm3.\n");
    }

    if(hash_multimap_greater_equal(phmm_hmm1, phmm_hmm4))
    {
        printf("The hash_multimap hmm1 is greater than or "
               "equal to the hash_multimap hmm4.\n");
    }
    else
    {
        printf("The hash_multimap hmm1 is less than the hash_multimap hmm4.\n");
    }

    hash_multimap_destroy(phmm_hmm1);
    hash_multimap_destroy(phmm_hmm2);
    hash_multimap_destroy(phmm_hmm3);
    hash_multimap_destroy(phmm_hmm4);
    pair_destroy(ppr_hmm);

    return 0;
}
```

- **Output**

```
The hash_multimap hmm1 is less than the hash_multimap hmm2.
The hash_multimap hmm1 is greater than or equal to the hash_multimap hmm3.
The hash_multimap hmm1 is greater than or equal to the hash_multimap hmm4.
```

18. hash_multimap_hash

返回 hash_multimap_t 中使用的哈希函数。

```
unary_function_t hash_multimap_hash(
    const hash_multimap_t* cphmmap_hmmap
);
```

- **Parameters**
  cphmmap_hmmap: 指向 hash_multimap_t 类型的指针。

- **Requirements**
  头文件 <cstl/chash_map.h>

- **Example**

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

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

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

int main(int argc, char* argv[])
{
    hash_multimap_t* phmm_hmm1 = create_hash_multimap(int, int);
    hash_multimap_t* phmm_hmm2 = create_hash_multimap(int, int);

    if(phmm_hmm1 == NULL || phmm_hmm2 == NULL)
    {
        return -1;
    }

    hash_multimap_init(phmm_hmm1);
    hash_multimap_init_ex(phmm_hmm2, 100, hash_func, NULL);

    if(hash_multimap_hash(phmm_hmm1) == hash_func)
    {
        printf("The hash function of hash_multimap hmm1 is hash_func.\n");
    }
    else
    {
        printf("The hash function of hash_multimap hmm1 is not hash_func.\n");
    }

    if(hash_multimap_hash(phmm_hmm2) == hash_func)
    {
        printf("The hash function of hash_multimap hmm2 is hash_func.\n");
    }
    else
    {
        printf("The hash function of hash_multimap hmm2 is not hash_func.\n");
    }

    hash_multimap_destroy(phmm_hmm1);
    hash_multimap_destroy(phmm_hmm2);

    return 0;
}

static void hash_func(const void* cpv_input, void* pv_output)
{
    *(int*)pv_output = *(int*)pair_first((pair_t*)cpv_input);
}
```

- **Output**

```
The hash function of hash_multimap hmm1 is not hash_func.
The hash function of hash_multimap hmm2 is hash_func.
```

19. hash_multimap_init hash_multimap_init_copy hash_multimap_init_copy_range hash_multimap_init_copy_range_ex hash_multimap_init_ex

初始化 hash_multimap_t 容器。

```
void hash_multimap_init(
    hash_multimap_t* phmmap_hmmap
);

void hash_multimap_init_copy(
    hash_multimap_t* phmmap_hmmap,
    const hash_multimap_t* cphmmap_src
);

void hash_multimap_init_copy_range(
    hash_multimap_t* phmmap_hmmap,
    hash_multimap_iterator_t it_begin,
    hash_multimap_iterator_t it_end
);

void hash_multimap_init_copy_range_ex(
    hash_multimap_t* phmmap_hmmap,
    hash_multimap_iterator_t it_begin,
    hash_multimap_iterator_t it_end,
    size_t t_bucketcount,
    unary_function_t ufun_hash,
    binary_function_t bfun_compare
);

void hash_multimap_init_ex(
    hash_multimap_t* phmmap_hmmap,
    size_t t_bucketcount,
    unary_function_t ufun_hash,
    binary_function_t bfun_compare
);
```

- **Parameters**

phmmap_hmmap: 指向被初始化 hash_multimap_t 类型的指针。
cphmmap_src: 指向用于初始化的 hash_multimap_t 类型的指针。
it_begin: 用于初始化的数据区间的开始位置。
it_end: 用于初始化的数据区间的末尾位置。
t_bucketcount: 哈希表中的存储单元个数。
ufun_hash: 自定义的哈希函数。
bfun_compare: 自定义比较规则。

- **Remarks**

第一个函数初始化一个空的 hash_multimap_t，使用默认的哈希函数和与键类型相关的小于操作函数作为默认的比较规则。

第二个函数使用一个源 hash_multimap_t 来初始化 hash_multimap_t，数据的内容，哈希函数和比较规则都从源 hash_multimap_t 复制。

第三个函数使用指定的数据区间初始化一个 hash_multimap_t，使用默认的哈希函数和与键类型相关的小于操作函数作为默认的比较规则。

第四个函数使用指定的数据区间初始化一个 hash_multimap_t，使用用户指定的哈希表存储单元个数，哈希函数和比较规则。

第五个函数初始化一个空的 hash_multimap_t，使用用户指定的哈希表存储单元个数，哈希函数和比较规则。

上面的函数要求迭代器和数据区间是有效的，无效的迭代器或数据区间导致函数的行为未定义。初始化函数根据用户指定的哈希表存储单元个数计算一个与用户指定的个数最接近的最佳哈希表存储单元个数。默认个数是 53 个，用户指定的个数小于等于 53 时都使用这个存储单元个数。

- **Requirements**

头文件 <cstl/chash_map.h>

- **Example**

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

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

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

int main(int argc, char* argv[])
{
    hash_multimap_t* phmm_hmm0 = create_hash_multimap(char*, int);
    hash_multimap_t* phmm_hmm1 = create_hash_multimap(char*, int);
    hash_multimap_t* phmm_hmm2 = create_hash_multimap(char*, int);
    hash_multimap_t* phmm_hmm3 = create_hash_multimap(char*, int);
    hash_multimap_t* phmm_hmm4 = create_hash_multimap(char*, int);
    hash_multimap_t* phmm_hmm5 = create_hash_multimap(char*, int);
    hash_multimap_iterator_t it_hmm;
    pair_t* ppr_hmm = create_pair(char*, int);

    if(phmm_hmm0 == NULL || phmm_hmm1 == NULL || phmm_hmm2 == NULL ||
       phmm_hmm3 == NULL || phmm_hmm4 == NULL || phmm_hmm5 == NULL ||
       ppr_hmm == NULL)
    {
        return -1;
    }

    pair_init(ppr_hmm);

    /* Create an empty hash_multimap hmm0 of key type string */
    hash_multimap_init(phmm_hmm0);

    /*
     * Create an empty hash_multimap hmm1 with the key comparison
     * function of less than, than insert 4 elements
     */
    hash_multimap_init_ex(phmm_hmm1, 0, _default_hash, fun_less_cstr);
    pair_make(ppr_hmm, "one", 0);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, "two", 10);
```

```
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, "three", 20);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, "four", 30);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, "five", 40);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);

    /*
     * Create an empty hash_multimap hmm2 with the key comparison
     * function of greater than, then insert 2 elements
     */
    hash_multimap_init_ex(phmm_hmm2, 100, _default_hash, fun_greater_cstr);
    pair_make(ppr_hmm, "one", 10);
    hash_multimap_insert(phmm_hmm2, ppr_hmm);
    pair_make(ppr_hmm, "two", 20);
    hash_multimap_insert(phmm_hmm2, ppr_hmm);

    /* Create a copy, hash_multimap hmm3, of hash_multimap hmm1 */
    hash_multimap_init_copy(phmm_hmm3, phmm_hmm1);

    /* Create a hash_multimap hmm4 by coping the range hmm1[first, last) */
    hash_multimap_init_copy_range(phmm_hmm4,
        iterator_advance(hash_multimap_begin(phmm_hmm1), 2),
        hash_multimap_end(phmm_hmm1));

    /*
     * Create a hash_multimap hmm5 by copying the range hmm3 [first, last)
     * and with the key comparison function of less than
     */
    hash_multimap_init_copy_range_ex(phmm_hmm5, hash_multimap_begin(phmm_hmm3),
        hash_multimap_end(phmm_hmm3), 100, _default_hash, fun_less_cstr);

    printf("hmm1 =");
    for(it_hmm = hash_multimap_begin(phmm_hmm1);
        !iterator_equal(it_hmm, hash_multimap_end(phmm_hmm1));
        it_hmm = iterator_next(it_hmm))
    {
        printf("(%s, %d) ",
            (char*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));
    }
    printf("\n");

    printf("hmm2 =");
    for(it_hmm = hash_multimap_begin(phmm_hmm2);
        !iterator_equal(it_hmm, hash_multimap_end(phmm_hmm2));
        it_hmm = iterator_next(it_hmm))
    {
        printf("(%s, %d) ",
            (char*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));
    }
    printf("\n");

    printf("hmm3 =");
    for(it_hmm = hash_multimap_begin(phmm_hmm3);
        !iterator_equal(it_hmm, hash_multimap_end(phmm_hmm3));
        it_hmm = iterator_next(it_hmm))
    {
```

```
        printf("(%s, %d) ",
            (char*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));
    }
    printf("\n");

    printf("hmm4 =");
    for(it_hmm = hash_multimap_begin(phmm_hmm4);
        !iterator_equal(it_hmm, hash_multimap_end(phmm_hmm4));
        it_hmm = iterator_next(it_hmm))
    {
        printf("(%s, %d) ",
            (char*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));
    }
    printf("\n");

    printf("hmm5 =");
    for(it_hmm = hash_multimap_begin(phmm_hmm5);
        !iterator_equal(it_hmm, hash_multimap_end(phmm_hmm5));
        it_hmm = iterator_next(it_hmm))
    {
        printf("(%s, %d) ",
            (char*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));
    }
    printf("\n");

    hash_multimap_destroy(phmm_hmm0);
    hash_multimap_destroy(phmm_hmm1);
    hash_multimap_destroy(phmm_hmm2);
    hash_multimap_destroy(phmm_hmm3);
    hash_multimap_destroy(phmm_hmm4);
    hash_multimap_destroy(phmm_hmm5);
    pair_destroy(ppr_hmm);

    return 0;
}

static void _default_hash(const void* cpv_input, void* pv_output)
{
    *(size_t*)pv_output = strlen((char*)pair_first((pair_t*)cpv_input));
}
```

- **Output**

```
hmm1 =(one, 0) (two, 10) (four, 30) (five, 40) (three, 20)
hmm2 =(one, 10) (two, 20)
hmm3 =(one, 0) (two, 10) (four, 30) (five, 40) (three, 20)
hmm4 =(five, 40) (three, 20) (four, 30)
hmm5 =(one, 0) (two, 10) (four, 30) (five, 40) (three, 20)
```

20. hash_multimap_insert hash_multimap_insert_range

向 hash_multimap_t 中插入数据。

```
hash_multimap_iterator_t hash_multimap_insert(
    hash_multimap_t* phmmap_hmmap,
    const pair_t* cpppair_pair
```

```
);

void hash_multimap_insert_range(
    hash_multimap_t* phmmap_hmmap,
    hash_multimap_iterator_t it_begin,
    hash_multimap_iterator_t it_end
);
```

- **Parameters**

  phmmap_hmmap: 指向 hash_multimap_t 类型的指针。
  cppair_pair: 插入的数据。
  it_begin: 被插入的数据区间的开始位置。
  it_end: 被插入的数据区间的末尾位置。

- **Remarks**

  第一个函数向 hash_multimap_t 中插入一个指定的数据，成功后返回指向该数据的迭代器。
  第二个函数插入指定的数据区间。
  上面的函数要求迭代器和数据区间是有效的，无效的迭代器或数据区间导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multimap_t* phmm_hmm1 = create_hash_multimap(int, int);
    hash_multimap_t* phmm_hmm2 = create_hash_multimap(int, int);
    pair_t* ppr_hmm = create_pair(int, int);
    hash_multimap_iterator_t it_hmm;

    if(phmm_hmm1 == NULL || phmm_hmm2 == NULL || ppr_hmm == NULL)
    {
        return -1;
    }

    hash_multimap_init(phmm_hmm1);
    hash_multimap_init(phmm_hmm2);
    pair_init(ppr_hmm);

    pair_make(ppr_hmm, 1, 10);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, 2, 20);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, 3, 30);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, 4, 40);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);

    printf("The original elements of hmm1 are:");
```

```
    for(it_hmm = hash_multimap_begin(phmm_hmm1);
        !iterator_equal(it_hmm, hash_multimap_end(phmm_hmm1));
        it_hmm = iterator_next(it_hmm))
    {
        printf(" (%d, %d)",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));
    }
    printf("\n");

    pair_make(ppr_hmm, 1, 10);
    it_hmm = hash_multimap_insert(phmm_hmm1, ppr_hmm);
    if(iterator_not_equal(it_hmm, hash_multimap_end(phmm_hmm1)))
    {
        printf("The element (1, 10) was inserted in hmm1 successfully.\n");
    }
    else
    {
        printf("The element (1, 10) alread exists in hmm1.\n");
    }

    pair_make(ppr_hmm, 10, 100);
    hash_multimap_insert(phmm_hmm2, ppr_hmm);
    hash_multimap_insert_range(phmm_hmm2,
        iterator_next(hash_multimap_begin(phmm_hmm1)),
        iterator_prev(hash_multimap_end(phmm_hmm1)));
    printf("After the insertions, the elements of hmm2 are:");
    for(it_hmm = hash_multimap_begin(phmm_hmm2);
        !iterator_equal(it_hmm, hash_multimap_end(phmm_hmm2));
        it_hmm = iterator_next(it_hmm))
    {
        printf(" (%d, %d)",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));
    }
    printf("\n");

    hash_multimap_destroy(phmm_hmm1);
    hash_multimap_destroy(phmm_hmm2);
    pair_destroy(ppr_hmm);

    return 0;
}
```

- **Output**

```
The original elements of hmm1 are: (1, 10) (2, 20) (3, 30) (4, 40)
The element (1, 10) was inserted in hmm1 successfully.
After the insertions, the elements of hmm2 are: (1, 10) (2, 20) (3, 30) (10, 100)
```

21. hash_multimap_key_comp

返回 hash_multimap_t 使用的键比较规则。

```
binary_function_t hash_multimap_key_comp(
    const hash_multimap_t* cphmmap_hmmap
);
```

- **Parameters**

cphmmap_hmmap: 指向 hash_multimap_t 类型的指针。

- **Remarks**

  这个排序规则是针对数据中的键进行排序。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multimap_t* phmm_hmm1 = create_hash_multimap(int, int);
    hash_multimap_t* phmm_hmm2 = create_hash_multimap(int, int);
    binary_function_t bfun_kc = NULL;
    int n_first = 2;
    int n_second = 3;
    bool_t b_result = false;

    if(phmm_hmm1 == NULL || phmm_hmm2 == NULL)
    {
        return -1;
    }

    hash_multimap_init_ex(phmm_hmm1, 0, NULL, fun_less_int);
    hash_multimap_init_ex(phmm_hmm2, 0, NULL, fun_greater_int);

    bfun_kc = hash_multimap_key_comp(phmm_hmm1);
    (*bfun_kc)(&n_first, &n_second, &b_result);
    if(b_result)
    {
        printf("(*bfun_kc)(2, 3) returns value of true, "
               "where bfun_kc is the compare function of hmm1.\n");
    }
    else
    {
        printf("(*bfun_kc)(2, 3) returns value of false, "
               "where bfun_kc is the compare function of hmm1.\n");
    }

    bfun_kc = hash_multimap_key_comp(phmm_hmm2);
    (*bfun_kc)(&n_first, &n_second, &b_result);
    if(b_result)
    {
        printf("(*bfun_kc)(2, 3) returns value of true, "
               "where bfun_kc is the compare function of hmm2.\n");
    }
    else
    {
        printf("(*bfun_kc)(2, 3) returns value of false, "
               "where bfun_kc is the compare function of hmm2.\n");
```

```
    }

    hash_multimap_destroy(phmm_hmm1);
    hash_multimap_destroy(phmm_hmm2);

    return 0;
}
```

- **Output**

```
(*bfun_kc)(2, 3) returns value of true, where bfun_kc is the compare function of
hmm1.
(*bfun_kc)(2, 3) returns value of false, where bfun_kc is the compare function of
hmm2.
```

22. hash_multimap_less

测试第一个 hash_multimap_t 是否小于第二个 hash_multimap_t。

```
bool_t hash_multimap_less(
    const hash_multimap_t* cphmmap_first,
    const hash_multimap_t* cphmmap_second
);
```

- **Parameters**

  cphmmap_first: 指向第一个 hash_multimap_t 类型的指针。
  cphmmap_second: 指向第二个 hash_multimap_t 类型的指针。

- **Remarks**

  这个函数要求两个 hash_multimap_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multimap_t* phmm_hmm1 = create_hash_multimap(int, int);
    hash_multimap_t* phmm_hmm2 = create_hash_multimap(int, int);
    hash_multimap_t* phmm_hmm3 = create_hash_multimap(int, int);
    pair_t* ppr_hmm = create_pair(int, int);
    hash_multimap_iterator_t it_hmm;
    int i = 0;

    if(phmm_hmm1 == NULL || phmm_hmm2 == NULL ||
       phmm_hmm3 == NULL || ppr_hmm == NULL)
    {
        return -1;
    }
```

```
    hash_multimap_init(phmm_hmm1);
    hash_multimap_init(phmm_hmm2);
    hash_multimap_init(phmm_hmm3);
    pair_init(ppr_hmm);

    for(i = 1; i < 4; ++i)
    {
        pair_make(ppr_hmm, i, i);
        hash_multimap_insert(phmm_hmm1, ppr_hmm);
        pair_make(ppr_hmm, i, i + 1);
        hash_multimap_insert(phmm_hmm2, ppr_hmm);
        pair_make(ppr_hmm, i + 1, i);
        hash_multimap_insert(phmm_hmm3, ppr_hmm);
    }

    printf("The elements of hash_multimap hmm1 are:");
    for(it_hmm = hash_multimap_begin(phmm_hmm1);
        !iterator_equal(it_hmm, hash_multimap_end(phmm_hmm1));
        it_hmm = iterator_next(it_hmm))
    {
        printf("(%d,%d) ",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));
    }
    printf("\n");

    printf("The elements of hash_multimap hmm2 are:");
    for(it_hmm = hash_multimap_begin(phmm_hmm2);
        !iterator_equal(it_hmm, hash_multimap_end(phmm_hmm2));
        it_hmm = iterator_next(it_hmm))
    {
        printf("(%d,%d) ",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));
    }
    printf("\n");

    printf("The elements of hash_multimap hmm3 are:");
    for(it_hmm = hash_multimap_begin(phmm_hmm3);
        !iterator_equal(it_hmm, hash_multimap_end(phmm_hmm3));
        it_hmm = iterator_next(it_hmm))
    {
        printf("(%d,%d) ",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));
    }
    printf("\n");

    if(hash_multimap_less(phmm_hmm1, phmm_hmm2))
    {
        printf("The hash_multimap hmm1 is less than the hash_multimap hmm2.\n");
    }
    else
    {
        printf("The hash_multimap hmm1 is not less than the hash_multimap hmm2.\n");
    }

    if(hash_multimap_less(phmm_hmm1, phmm_hmm3))
    {
```

```
        printf("The hash_multimap hmm1 is less than the hash_multimap hmm3.\n");
    }
    else
    {
        printf("The hash_multimap hmm1 is not less than the hash_multimap hmm3.\n");
    }

    hash_multimap_destroy(phmm_hmm1);
    hash_multimap_destroy(phmm_hmm2);
    hash_multimap_destroy(phmm_hmm3);
    pair_destroy(ppr_hmm);

    return 0;
}
```

- **Output**

```
The elements of hash_multimap hmm1 are:(1,1) (2,2) (3,3)
The elements of hash_multimap hmm2 are:(1,2) (2,3) (3,4)
The elements of hash_multimap hmm3 are:(2,1) (3,2) (4,3)
The hash_multimap hmm1 is less than the hash_multimap hmm2.
The hash_multimap hmm1 is less than the hash_multimap hmm3.
```

23. hash_multimap_less_equal

测试第一个 hash_multimap_t 是否小于等于第二个 hash_multimap_t。

```
bool_t hash_multimap_less_equal(
    const hash_multimap_t* cphmmap_first,
    const hash_multimap_t* cphmmap_second
);
```

- **Parameters**

  cphmmap_first: 指向第一个 hash_multimap_t 类型的指针。
  cphmmap_second: 指向第二个 hash_multimap_t 类型的指针。

- **Remarks**

  这个函数要求两个 hash_multimap_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);
    hash_multiset_t* phms_hms2 = create_hash_multiset(int);
    hash_multiset_t* phms_hms3 = create_hash_multiset(int);
    hash_multiset_t* phms_hms4 = create_hash_multiset(int);
```

```
    int i = 0;

    if(phms_hms1 == NULL || phms_hms2 == NULL ||
       phms_hms3 == NULL || phms_hms4 == NULL)
    {
        return -1;
    }

    hash_multiset_init(phms_hms1);
    hash_multiset_init(phms_hms2);
    hash_multiset_init(phms_hms3);
    hash_multiset_init(phms_hms4);

    for(i = 0; i < 3; ++i)
    {
        hash_multiset_insert(phms_hms1, i);
        hash_multiset_insert(phms_hms2, i * i);
        hash_multiset_insert(phms_hms3, i - 1);
        hash_multiset_insert(phms_hms4, i);
    }

    if(hash_multiset_less_equal(phms_hms1, phms_hms2))
    {
        printf("The hash_multiset hs1 is less than or "
               "equal to the hash_multiset hs2.\n");
    }
    else
    {
        printf("The hash_multiset hs1 is greater than the hash_multiset hs2.\n");
    }

    if(hash_multiset_less_equal(phms_hms1, phms_hms3))
    {
        printf("The hash_multiset hs1 is less than or "
               "equal to the hash_multiset hs3.\n");
    }
    else
    {
        printf("The hash_multiset hs1 is greater than the hash_multiset hs3.\n");
    }

    if(hash_multiset_less_equal(phms_hms1, phms_hms4))
    {
        printf("The hash_multiset hs1 is less than or "
               "equal to the hash_multiset hs4.\n");
    }
    else
    {
        printf("The hash_multiset hs1 is greater than the hash_multiset hs4.\n");
    }

    hash_multiset_destroy(phms_hms1);
    hash_multiset_destroy(phms_hms2);
    hash_multiset_destroy(phms_hms3);
    hash_multiset_destroy(phms_hms4);

    return 0;
}
```

- **Output**

```
The elements of hash_multimap hmm1 are:(1,1) (2,2) (3,3)
The elements of hash_multimap hmm2 are:(1,2) (2,3) (3,4)
The elements of hash_multimap hmm3 are:(2,1) (3,2) (4,3)
The hash_multimap hmm1 is less than the hash_multimap hmm2.
The hash_multimap hmm1 is less than the hash_multimap hmm3.
```

24. hash_multimap_max_size

返回 hash_multimap_t 中能够保存数据数量的最大值。

```
size_t hash_multimap_max_size(
    const hash_multimap_t* cphmmap_hmmap
);
```

- **Parameters**

  cphmmap_hmmap: 指向 hash_multimap_t 类型的指针。

- **Remarks**

  这是一个与系统相关的常数。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multiset_t* phms_hms1 = create_hash_multiset(int);

    if(phms_hms1 == NULL)
    {
        return -1;
    }

    hash_multiset_init(phms_hms1);

    printf("The maximum possible length of the hash_multiset hs1 is: %d.\n",
        hash_multiset_max_size(phms_hms1));

    hash_multiset_destroy(phms_hms1);

    return 0;
}
```

- **Output**

```
The maximum possible length of the hash_multimap hmm1 is: 7895160.
```

25. hash_multimap_not_equal

测试两个 hash_multimap_t 是否不等。

```
bool_t hash_multimap_not_equal(
    const hash_multimap_t* cphmmap_first,
    const hash_multimap_t* cphmmap_second
);
```

- **Parameters**

cphmmap_first: 指向第一个 hash_multimap_t 类型的指针。
cphmmap_second: 指向第二个 hash_multimap_t 类型的指针。

- **Remarks**

如果两个 hash_multimap_t 容器中的数据都对应相等，并且数据个数相等，则返回 false 否则返回 true，如果两个 hash_multimap_t 容器中保存的数据类型不同也认为是不等。

- **Requirements**

头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multimap_t* phmm_hmm1 = create_hash_multimap(int, int);
    hash_multimap_t* phmm_hmm2 = create_hash_multimap(int, int);
    hash_multimap_t* phmm_hmm3 = create_hash_multimap(int, int);
    pair_t* ppr_hmm = create_pair(int, int);
    int i = 0;

    if(phmm_hmm1 == NULL || phmm_hmm2 == NULL ||
       phmm_hmm3 == NULL || ppr_hmm == NULL)
    {
        return -1;
    }

    hash_multimap_init(phmm_hmm1);
    hash_multimap_init(phmm_hmm2);
    hash_multimap_init(phmm_hmm3);
    pair_init(ppr_hmm);

    for(i = 0; i < 3; ++i)
    {
        pair_make(ppr_hmm, i, i);
        hash_multimap_insert(phmm_hmm1, ppr_hmm);
        hash_multimap_insert(phmm_hmm3, ppr_hmm);
        pair_make(ppr_hmm, i, i * i);
        hash_multimap_insert(phmm_hmm2, ppr_hmm);
    }

    if(hash_multimap_not_equal(phmm_hmm1, phmm_hmm2))
```

```
    {
        printf("The hash_multimaps hmm1 and hmm2 are not equal.\n");
    }
    else
    {
        printf("The hash_multimaps hmm1 and hmm2 are equal.\n");
    }

    if(hash_multimap_not_equal(phmm_hmm1, phmm_hmm3))
    {
        printf("The hash_multimaps hmm1 and hmm3 are not equal.\n");
    }
    else
    {
        printf("The hash_multimaps hmm1 and hmm3 are equal.\n");
    }

    hash_multimap_destroy(phmm_hmm1);
    hash_multimap_destroy(phmm_hmm2);
    hash_multimap_destroy(phmm_hmm3);
    pair_destroy(ppr_hmm);

    return 0;
}
```

- **Output**

```
The hash_multimaps hmm1 and hmm2 are not equal.
The hash_multimaps hmm1 and hmm3 are equal.
```

26. hash_multimap_resize

重新设置 hash_multimap_t 中哈希表存储单元的个数。

```
void hash_multimap_resize(
    hash_multimap_t* phmmap_hmmap,
    size_t t_resize
);
```

- **Parameters**

cphmmap_hmmap: 指向 hash_multimap_t 类型的指针。
t_resize: 哈希表存储单元的新数量。

- **Remarks**

当哈希表存储单元数量改变后，哈希表中的数据将被重新计算位置，所有的迭代器失效。当新的存储单元数量小于当前数量时，不做任何操作。

- **Requirements**

头文件 <cstl/chash_map.h>

- **Example**

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

#include <stdio.h>
```

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

int main(int argc, char* argv[])
{
    hash_multimap_t* phmm_hmm1 = create_hash_multimap(int, int);

    if(phmm_hmm1 == NULL)
    {
        return -1;
    }

    hash_multimap_init(phmm_hmm1);

    printf("The bucket count of hash_multimap hmm1 is: %d.\n",
        hash_multimap_bucket_count(phmm_hmm1));

    hash_multimap_resize(phmm_hmm1, 100);

    printf("The bucket count of hash_multimap hmm1 is now: %d.\n",
        hash_multimap_bucket_count(phmm_hmm1));

    hash_multimap_destroy(phmm_hmm1);

    return 0;
}
```

- **Output**

```
The bucket count of hash_multimap hmm1 is: 53.
The bucket count of hash_multimap hmm1 is now: 193.
```

27. hash_multimap_size

返回 hash_multimap_t 中数据的个数。

```
size_t hash_multimap_size(
    const hash_multimap_t* cphmmap_hmmap
);
```

- **Parameters**

  cphmmap_hmmap: 指向 hash_multimap_t 类型的指针。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multimap_t* phmm_hmm1 = create_hash_multimap(int, int);
    pair_t* ppr_hmm = create_pair(int, int);
```

```
    if(phmm_hmm1 == NULL)
    {
        return -1;
    }

    hash_multimap_init(phmm_hmm1);
    pair_init(ppr_hmm);

    pair_make(ppr_hmm, 1, 1);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    printf("The hash_multimap hmm1 length is %d.\n",
        hash_multimap_size(phmm_hmm1));

    pair_make(ppr_hmm, 2, 4);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    printf("The hash_multimap hmm1 length is now %d.\n",
        hash_multimap_size(phmm_hmm1));

    hash_multimap_destroy(phmm_hmm1);
    pair_destroy(ppr_hmm);

    return 0;
}
```

- **Output**

```
The hash_multimap hmm1 length is 1.
The hash_multimap hmm1 length is now 2.
```

28. hash_multimap_swap

交换两个 hash_multimap_t 中的内容。

```
void hash_multimap_swap(
    hash_multimap_t* phmmap_first,
    hash_multimap_t* phmmap_second
);
```

- **Parameters**

  phmmap_first: 指向第一个 hash_multimap_t 类型的指针。
  phmmap_second: 指向第二个 hash_multimap_t 类型的指针。

- **Remarks**

  这个函数要求两个 hash_multimap_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

```c
int main(int argc, char* argv[])
{
    hash_multimap_t* phmm_hmm1 = create_hash_multimap(int, int);
    hash_multimap_t* phmm_hmm2 = create_hash_multimap(int, int);
    pair_t* ppr_hmm = create_pair(int, int);
    hash_multimap_iterator_t it_hmm;

    if(phmm_hmm1 == NULL || phmm_hmm2 == NULL || ppr_hmm == NULL)
    {
        return -1;
    }

    hash_multimap_init(phmm_hmm1);
    hash_multimap_init(phmm_hmm2);
    pair_init(ppr_hmm);

    pair_make(ppr_hmm, 1, 10);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, 2, 20);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, 3, 30);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, 10, 100);
    hash_multimap_insert(phmm_hmm2, ppr_hmm);
    pair_make(ppr_hmm, 20, 200);
    hash_multimap_insert(phmm_hmm2, ppr_hmm);

    printf("The orighinal hash_multimap hmm1 is:");
    for(it_hmm = hash_multimap_begin(phmm_hmm1);
        !iterator_equal(it_hmm, hash_multimap_end(phmm_hmm1));
        it_hmm = iterator_next(it_hmm))
    {
        printf(" (%d, %d)",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));
    }
    printf("\n");

    hash_multimap_swap(phmm_hmm1, phmm_hmm2);

    printf("After swapping with hmm2, hash_multimap hmm1 is:");
    for(it_hmm = hash_multimap_begin(phmm_hmm1);
        !iterator_equal(it_hmm, hash_multimap_end(phmm_hmm1));
        it_hmm = iterator_next(it_hmm))
    {
        printf(" (%d, %d)",
            *(int*)pair_first((pair_t*)iterator_get_pointer(it_hmm)),
            *(int*)pair_second((pair_t*)iterator_get_pointer(it_hmm)));
    }
    printf("\n");

    hash_multimap_destroy(phmm_hmm1);
    hash_multimap_destroy(phmm_hmm2);
    pair_destroy(ppr_hmm);

    return 0;
}
```

- **Output**

```
The orighinal hash_multimap hmm1 is: (1, 10) (2, 20) (3, 30)
After swapping with hmm2, hash_multimap hmm1 is: (10, 100) (20, 200)
```

29. hash_multimap_value_comp

返回 hash_multimap_t 使用的数据比较规则。

```
binary_function_t hash_multimap_value_comp(
    const hash_multimap_t* cphmmap_hmmap
);
```

- **Parameters**

  cphmmap_hmmap: 指向 hash_multimap_t 类型的指针。

- **Remarks**

  这个规则是针对数据本身的比较规则而不是键或者值。

- **Requirements**

  头文件 <cstl/chash_map.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    hash_multimap_t* phmm_hmm1 = create_hash_multimap(int, int);
    pair_t* ppr_hmm = create_pair(int, int);
    binary_function_t bfun_vc = NULL;
    bool_t b_result = false;
    hash_multimap_iterator_t it_hmm1;
    hash_multimap_iterator_t it_hmm2;

    if(phmm_hmm1 == NULL || ppr_hmm == NULL)
    {
        return -1;
    }

    pair_init(ppr_hmm);
    hash_multimap_init_ex(phmm_hmm1, 100, NULL, fun_less_int);

    pair_make(ppr_hmm, 1, 10);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);
    pair_make(ppr_hmm, 2, 5);
    hash_multimap_insert(phmm_hmm1, ppr_hmm);

    it_hmm1 = hash_multimap_find(phmm_hmm1, 1);
    it_hmm2 = hash_multimap_find(phmm_hmm1, 2);
    bfun_vc = hash_multimap_value_comp(phmm_hmm1);

    (*bfun_vc)(iterator_get_pointer(it_hmm1),
```

```
        iterator_get_pointer(it_hmm2), &b_result);
    if(b_result)
    {
        printf("The element (1, 10) precedes the element (2, 5).\n");
    }
    else
    {
        printf("The element (1, 10) does not precedes the element (2, 5).\n");
    }

    (*bfun_vc)(iterator_get_pointer(it_hmm2),
        iterator_get_pointer(it_hmm1), &b_result);
    if(b_result)
    {
        printf("The element (2, 5) precedes the element (1, 10).\n");
    }
    else
    {
        printf("The element (2, 5) does not precedes the element (1, 10).\n");
    }

    pair_destroy(ppr_hmm);
    hash_multimap_destroy(phmm_hmm1);

    return 0;
}
```

- **Output**

```
The element (1, 10) precedes the element (2, 5).
The element (2, 5) does not precedes the element (1, 10).
```

第十三节　　堆栈 stack_t

堆栈 stack_t 是容器适配器，它是以序列容器为底层实现。stack_t 支持后入先出(LIFO)，数据的插入和删除都是在堆栈的顶部进行的，不能够访问堆栈内部的数据。stack_t 不支持迭代器。

- **Typedefs**

stack_t	堆栈容器适配器类型。

- **Operation Functions**

create_stack	创建堆栈容器适配器类型。
stack_assign	为堆栈容器适配器类型赋值。
stack_destroy	销毁堆栈容器适配器类型。
stack_empty	测试堆栈容器适配器类型是否为空。
stack_equal	测试两个堆栈容器适配器类型是否相等。
stack_greater	测试第一个堆栈容器适配器类型是否大于第二个堆栈容器适配器类型。
stack_greater_equal	测试第一个堆栈容器适配器类型是否大于等于第二个堆栈容器适配器类型。
stack_init	初始化一个空的堆栈容器适配器类型。
stack_init_copy	以拷贝的方式初始化一个堆栈容器适配器。

stack_less	测试第一个堆栈容器适配器是否小于第二个堆栈容器适配器。
stack_less_equal	测试第一个堆栈容器适配器是否小于等于第二个堆栈容器适配器。
stack_not_equal	测试两个堆栈容器适配器是否不等。
stack_pop	弹出堆栈容器适配器栈顶的数据。
stack_push	将数据压入堆栈容器适配器。
stack_size	返回堆栈容器适配器中的数据个数。
stack_top	访问堆栈容器适配器栈顶数据。

1. stack_t

堆栈容器适配器类型。

- **Requirements**

头文件 <cstl/cstack.h>

- **Example**

请参考 stack_t 类型的其他操作函数。

2. create_stack

创建 stack_t 类型。

```
stack_t* create_stack(
    type
);
```

- **Parameters**

type: 数据类型描述。

- **Remarks**

函数成功返回指向 stack_t 类型的指针，失败返回 NULL。

- **Requirements**

头文件 <cstl/cstack.h>

- **Example**

请参考 stack_t 类型的其他操作函数。

3. stack_assign

为 stack_t 类型赋值。

```
void stack_assign(
    stack_t* pstack_dest,
    const stack_t* cpstack_src
);
```

- **Parameters**

pstack_dest: 指向被赋值的 stack_t 类型的指针。

cpstack_src:　指向赋值的 stack_t 类型的指针。

- **Remarks**

  要求两个 stack_t 类型保存的数据具有相同的类型，否则函数的行为未定义。

- **Requirements**

  头文件 <cstl/cstack.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    stack_t* psk_sk1 = create_stack(int);
    stack_t* psk_sk2 = create_stack(int);

    if(psk_sk1 == NULL || psk_sk2 == NULL)
    {
        return -1;
    }

    stack_init(psk_sk1);
    stack_init(psk_sk2);

    stack_push(psk_sk1, 1);
    stack_push(psk_sk1, 2);
    stack_push(psk_sk2, 10);
    stack_push(psk_sk2, 20);
    stack_push(psk_sk2, 30);

    printf("The size of stack sk1 is %d and the top element is %d.\n",
        stack_size(psk_sk1), *(int*)stack_top(psk_sk1));

    stack_assign(psk_sk1, psk_sk2);

    printf("After assigning the size of stack sk1 is %d "
           "and the top element is %d.\n",
           stack_size(psk_sk1), *(int*)stack_top(psk_sk1));

    stack_destroy(psk_sk1);
    stack_destroy(psk_sk2);

    return 0;
}
```

- **Output**

```
The size of stack sk1 is 2 and the top element is 2.
After assigning the size of stack sk1 is 3 and the top element is 30.
```

4. stack_destroy

销毁 stack_t 类型。

```
void stack_destroy(
    stack_t* pstack_stack
);
```

- **Parameters**

pstack_stack: 指向 stack_t 类型的指针。

- **Remarks**

stack_t 使用之后一定要销毁，否则 stack_t 申请的资源不会被释放。

- **Requirements**

头文件 <cstl/cstack.h>

- **Example**

请参考 stack_t 类型的其他操作函数。

5. stack_empty

测试 stack_t 是否为空。

```
bool_t stack_empty(
    const stack_t* cpstack_stack
);
```

- **Parameters**

cpstack_stack: 指向 stack_t 类型的指针。

- **Remarks**

stack_t 为空返回 true，否则返回 false。

- **Requirements**

头文件 <cstl/cstack.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    stack_t* psk_sk1 = create_stack(int);
    stack_t* psk_sk2 = create_stack(int);

    if(psk_sk1 == NULL || psk_sk2 == NULL)
    {
        return -1;
    }
```

```
    stack_init(psk_sk1);
    stack_init(psk_sk2);

    stack_push(psk_sk1, 1);

    if(stack_empty(psk_sk1))
    {
        printf("The stack sk1 is empty.\n");
    }
    else
    {
        printf("The stack sk1 is not empty.\n");
    }

    if(stack_empty(psk_sk2))
    {
        printf("The stack sk2 is empty.\n");
    }
    else
    {
        printf("The stack sk2 is not empty.\n");
    }

    stack_destroy(psk_sk1);
    stack_destroy(psk_sk2);

    return 0;
}
```

- **Output**

```
The stack sk1 is not empty.
The stack sk2 is empty.
```

6. stack_equal

测试两个 stack_t 是否相等。

```
bool_t stack_equal(
    const stack_t* cpstack_first,
    const stack_t* cpstack_second
);
```

- **Parameters**

cpstack_first: 指向第一个 stack_t 类型的指针。
cpstack_second: 指向第二个 stack_t 类型的指针。

- **Remarks**

如果两个 stack_t 中的数据都对应相等，并且数据个数相等，则返回 true 否则返回 false，如果两个 stack_t 中保存的数据类型不同也认为是不等。

- **Requirements**

头文件 <cstl/cstack.h>

- **Example**

```
/*
 * stack_equal.c
```

```
 * compile with : -lcstl
 */

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

int main(int argc, char* argv[])
{
    stack_t* psk_sk1 = create_stack(int);
    stack_t* psk_sk2 = create_stack(int);
    stack_t* psk_sk3 = create_stack(int);

    if(psk_sk1 == NULL || psk_sk2 == NULL || psk_sk3 == NULL)
    {
        return -1;
    }

    stack_init(psk_sk1);
    stack_init(psk_sk2);
    stack_init(psk_sk3);

    stack_push(psk_sk1, 1);
    stack_push(psk_sk2, 2);
    stack_push(psk_sk3, 1);

    if(stack_equal(psk_sk1, psk_sk2))
    {
        printf("The stacks sk1 and sk2 are equal.\n");
    }
    else
    {
        printf("The stacks sk1 and sk2 are not equal.\n");
    }

    if(stack_equal(psk_sk1, psk_sk3))
    {
        printf("The stacks sk1 and sk3 are equal.\n");
    }
    else
    {
        printf("The stacks sk1 and sk3 are not equal.\n");
    }

    stack_destroy(psk_sk1);
    stack_destroy(psk_sk2);
    stack_destroy(psk_sk3);

    return 0;
}
```

- **Output**

```
The stacks sk1 and sk2 are not equal.
The stacks sk1 and sk3 are equal.
```

7. stack_greater

测试第一个 stack_t 是否大于第二个 stack_t。

```
bool_t stack_greater(
```

```
    const stack_t* cpstack_first,
    const stack_t* cpstack_second
);
```

- **Parameters**
  cpstack_first: 指向第一个 stack_t 类型的指针。
  cpstack_second: 指向第二个 stack_t 类型的指针。

- **Remarks**
  这个函数要求两个 stack_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**
  头文件 <cstl/cstack.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    stack_t* psk_sk1 = create_stack(int);
    stack_t* psk_sk2 = create_stack(int);
    stack_t* psk_sk3 = create_stack(int);

    if(psk_sk1 == NULL || psk_sk2 == NULL || psk_sk3 == NULL)
    {
        return -1;
    }

    stack_init(psk_sk1);
    stack_init(psk_sk2);
    stack_init(psk_sk3);

    stack_push(psk_sk1, 1);
    stack_push(psk_sk1, 2);
    stack_push(psk_sk1, 3);
    stack_push(psk_sk2, 5);
    stack_push(psk_sk2, 10);
    stack_push(psk_sk3, 1);
    stack_push(psk_sk3, 2);

    if(stack_greater(psk_sk1, psk_sk2))
    {
        printf("The stack sk1 is greater than the stack sk2.\n");
    }
    else
    {
        printf("The stack sk1 is not greater than the stack sk2.\n");
    }

    if(stack_greater(psk_sk1, psk_sk3))
    {
        printf("The stack sk1 is greater than the stack sk3.\n");
```

```
    }
    else
    {
        printf("The stack sk1 is not greater than the stack sk3.\n");
    }

    stack_destroy(psk_sk1);
    stack_destroy(psk_sk2);
    stack_destroy(psk_sk3);

    return 0;
}
```

- **Output**

```
The stack sk1 is not greater than the stack sk2.
The stack sk1 is greater than the stack sk3.
```

8. stack_greater_equal

测试第一个 stack_t 是否大于等于第二个 stack_t。

```
bool_t stack_greater_equal(
    const stack_t* cpstack_first,
    const stack_t* cpstack_second
);
```

- **Parameters**

cpstack_first: 指向第一个 stack_t 类型的指针。
cpstack_second: 指向第二个 stack_t 类型的指针。

- **Remarks**

这个函数要求两个 stack_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

头文件 <cstl/cstack.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    stack_t* psk_sk1 = create_stack(int);
    stack_t* psk_sk2 = create_stack(int);
    stack_t* psk_sk3 = create_stack(int);

    if(psk_sk1 == NULL || psk_sk2 == NULL || psk_sk3 == NULL)
    {
        return -1;
    }
```

```
    stack_init(psk_sk1);
    stack_init(psk_sk2);
    stack_init(psk_sk3);

    stack_push(psk_sk1, 1);
    stack_push(psk_sk1, 2);
    stack_push(psk_sk1, 3);
    stack_push(psk_sk2, 5);
    stack_push(psk_sk2, 10);
    stack_push(psk_sk3, 1);
    stack_push(psk_sk3, 2);

    if(stack_greater_equal(psk_sk1, psk_sk2))
    {
        printf("The stack sk1 is greater than or equal to the stack sk2.\n");
    }
    else
    {
        printf("The stack sk1 is less than the stack sk2.\n");
    }

    if(stack_greater_equal(psk_sk1, psk_sk3))
    {
        printf("The stack sk1 is greater than or equal to the stack sk3.\n");
    }
    else
    {
        printf("The stack sk1 is less than the stack sk3.\n");
    }

    stack_destroy(psk_sk1);
    stack_destroy(psk_sk2);
    stack_destroy(psk_sk3);

    return 0;
}
```

- **Output**

```
The stack sk1 is less than the stack sk2.
The stack sk1 is greater than or equal to the stack sk3.
```

9. stack_init　stack_init_copy

初始化 stack_t 类型。

```
void stack_init(
    stack_t* pstack_stack
);

void stack_init_copy(
    stack_t* pstack_stack,
    const stack_t* cpstack_src
);
```

- **Parameters**

  pstack_stack: 指向被初始化 stack_t 类型的指针。
  cpstack_src: 指向用于初始化的 stack_t 类型的指针。

- **Remarks**
  第一个函数初始化一个空的 stack_t。
  第二个函数使用一个源 stack_t 来初始化 stack_t，数据的内容从源 stack_t 复制。

- **Requirements**
  头文件 <cstl/cstack.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    stack_t* psk_sk1 = create_stack(int);
    stack_t* psk_sk2 = create_stack(int);

    if(psk_sk1 == NULL || psk_sk2 == NULL)
    {
        return -1;
    }

    stack_init(psk_sk1);
    printf("The size fo stack sk1 is %d.\n", stack_size(psk_sk1));

    stack_push(psk_sk1, 10);
    stack_push(psk_sk1, 20);
    stack_push(psk_sk1, 40);
    printf("The size of stack sk1 is now %d.\n", stack_size(psk_sk1));

    stack_init_copy(psk_sk2, psk_sk1);
    printf("The size of stack sk2 is %d.\n", stack_size(psk_sk2));

    stack_destroy(psk_sk1);
    stack_destroy(psk_sk2);

    return 0;
}
```

- **Output**

```
The size fo stack sk1 is 0.
The size of stack sk1 is now 3.
The size of stack sk2 is 3.
```

10. stack_less

测试第一个 stack_t 是否小于第二个 stack_t。

```
bool_t stack_less(
    const stack_t* cpstack_first,
    const stack_t* cpstack_second
);
```

- **Parameters**
  cpstack_first:　指向第一个 stack_t 类型的指针。
  cpstack_second: 指向第二个 stack_t 类型的指针。

- **Remarks**
  这个函数要求两个 stack_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**
  头文件 <cstl/cstack.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    stack_t* psk_sk1 = create_stack(int);
    stack_t* psk_sk2 = create_stack(int);
    stack_t* psk_sk3 = create_stack(int);
    size_t t_count = 0;
    int i = 0;

    if(psk_sk1 == NULL || psk_sk2 == NULL || psk_sk3 == NULL)
    {
        return -1;
    }

    stack_init(psk_sk1);
    stack_init(psk_sk2);
    stack_init(psk_sk3);

    stack_push(psk_sk1, 2);
    stack_push(psk_sk1, 4);
    stack_push(psk_sk1, 6);
    stack_push(psk_sk1, 8);
    stack_push(psk_sk2, 5);
    stack_push(psk_sk2, 10);
    stack_push(psk_sk3, 2);
    stack_push(psk_sk3, 4);
    stack_push(psk_sk3, 6);
    stack_push(psk_sk3, 8);

    if(stack_less(psk_sk1, psk_sk2))
    {
        printf("The stack sk1 is less than the stack s2.\n");
    }
    else
    {
        printf("The stack sk1 is not less than the stack s2.\n");
    }

    if(stack_less(psk_sk1, psk_sk3))
```

```
    {
        printf("The stack sk1 is less than the stack s3.\n");
    }
    else
    {
        printf("The stack sk1 is not less than the stack s3.\n");
    }

    /* to print out the stack sk1 (by unstacking the elements) */
    printf("The stack sk1 from the top down is: ( ");
    t_count = stack_size(psk_sk1);
    for(i = 0; i < t_count; ++i)
    {
        printf("%d ", *(int*)stack_top(psk_sk1));
        stack_pop(psk_sk1);
    }
    printf(").\n");

    stack_destroy(psk_sk1);
    stack_destroy(psk_sk2);
    stack_destroy(psk_sk3);

    return 0;
}
```

- **Output**

```
The stack sk1 is less than the stack s2.
The stack sk1 is not less than the stack s3.
The stack sk1 from the top down is: ( 8 6 4 2 ).
```

11. stack_less_equal

测试第一个 stack_t 是否小于等于第二个 stack_t。

```
bool_t stack_less_equal(
    const stack_t* cpstack_first,
    const stack_t* cpstack_second
);
```

- **Parameters**

  cpstack_first: 指向第一个 stack_t 类型的指针。
  cpstack_second: 指向第二个 stack_t 类型的指针。

- **Remarks**

  这个函数要求两个 stack_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cstack.h>

- **Example**

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

#include <stdio.h>
```

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

int main(int argc, char* argv[])
{
    stack_t* psk_sk1 = create_stack(int);
    stack_t* psk_sk2 = create_stack(int);
    stack_t* psk_sk3 = create_stack(int);

    if(psk_sk1 == NULL || psk_sk2 == NULL || psk_sk3 == NULL)
    {
        return -1;
    }

    stack_init(psk_sk1);
    stack_init(psk_sk2);
    stack_init(psk_sk3);

    stack_push(psk_sk1, 5);
    stack_push(psk_sk1, 10);
    stack_push(psk_sk2, 1);
    stack_push(psk_sk2, 2);
    stack_push(psk_sk3, 5);
    stack_push(psk_sk3, 10);

    if(stack_less_equal(psk_sk1, psk_sk2))
    {
        printf("The stack sk1 is less than or equal to the stack sk2.\n");
    }
    else
    {
        printf("The stack sk1 is greater than the stack sk2.\n");
    }

    if(stack_less_equal(psk_sk1, psk_sk3))
    {
        printf("The stack sk1 is less than or equal to the stack sk3.\n");
    }
    else
    {
        printf("The stack sk1 is greater than the stack sk3.\n");
    }

    stack_destroy(psk_sk1);
    stack_destroy(psk_sk2);
    stack_destroy(psk_sk3);

    return 0;
}
```

- **Output**

```
The stack sk1 is greater than the stack sk2.
The stack sk1 is less than or equal to the stack sk3.
```

12. stack_not_equal

测试两个 stack_t 是否不等。

```
bool_t stack_not_equal(
```

```
    const stack_t* cpstack_first,
    const stack_t* cpstack_second
);
```

- **Parameters**

  cpstack_first: 指向第一个 stack_t 类型的指针。

  cpstack_second: 指向第二个 stack_t 类型的指针。

- **Remarks**

如果两个 stack_t 中的数据都对应相等，并且数据个数相等，则返回 false 否则返回 true，如果两个 stack_t 中保存的数据类型不同也认为是不等。

- **Requirements**

  头文件 <cstl/cstack.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    stack_t* psk_sk1 = create_stack(int);
    stack_t* psk_sk2 = create_stack(int);
    stack_t* psk_sk3 = create_stack(int);

    if(psk_sk1 == NULL || psk_sk2 == NULL || psk_sk3 == NULL)
    {
        return -1;
    }

    stack_init(psk_sk1);
    stack_init(psk_sk2);
    stack_init(psk_sk3);

    stack_push(psk_sk1, 1);
    stack_push(psk_sk2, 2);
    stack_push(psk_sk3, 1);

    if(stack_not_equal(psk_sk1, psk_sk2))
    {
        printf("The stacks sk1 and sk2 are not equal.\n");
    }
    else
    {
        printf("The stacks sk1 and sk2 are equal.\n");
    }

    if(stack_not_equal(psk_sk1, psk_sk3))
    {
        printf("The stacks sk1 and sk3 are not equal.\n");
    }
    else
```

```
    {
        printf("The stacks sk1 and sk3 are equal.\n");
    }

    stack_destroy(psk_sk1);
    stack_destroy(psk_sk2);
    stack_destroy(psk_sk3);

    return 0;
}
```

- **Output**

```
The stacks sk1 and sk2 are not equal.
The stacks sk1 and sk3 are equal.
```

13. stack_pop

弹出 stack_t 中的数据。

```
void stack_pop(
    stack_t* pstack_stack
);
```

- **Parameters**

pstack_stack: 指向 stack_t 类型的指针。

- **Remarks**

stack_t 为空，程序的行为未定义。

- **Requirements**

头文件 <cstl/cstack.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    stack_t* psk_sk1 = create_stack(int);

    if(psk_sk1 == NULL)
    {
        return -1;
    }

    stack_init(psk_sk1);

    stack_push(psk_sk1, 10);
    stack_push(psk_sk1, 20);
```

```
    stack_push(psk_sk1, 30);

    printf("The stack length is %d.\n", stack_size(psk_sk1));
    printf("The element at the top of the stack is %d.\n",
        *(int*)stack_top(psk_sk1));

    stack_pop(psk_sk1);

    printf("After a pop, the stack length is %d.\n", stack_size(psk_sk1));
    printf("After a pop, the element at the top of the stack is %d.\n",
        *(int*)stack_top(psk_sk1));

    stack_destroy(psk_sk1);

    return 0;
}
```

- **Output**

```
The stack length is 3.
The element at the top of the stack is 30.
After a pop, the stack length is 2.
After a pop, the element at the top of the stack is 20.
```

14. stack_push

将数据压入到 stack_t 中。

```
void stack_push(
    stack_t* pstack_stack,
    element
);
```

- **Parameters**

pstack_stack: 指向 stack_t 类型的指针。
element: 压入 stack_t 的数据。

- **Requirements**

头文件 <cstl/cstack.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    stack_t* psk_sk1 = create_stack(int);

    if(psk_sk1 == NULL)
```

```
    {
        return -1;
    }

    stack_init(psk_sk1);

    stack_push(psk_sk1, 10);
    stack_push(psk_sk1, 20);
    stack_push(psk_sk1, 30);

    printf("The stack length is %d.\n", stack_size(psk_sk1));
    printf("The element at the top of the stack is %d.\n",
        *(int*)stack_top(psk_sk1));

    stack_destroy(psk_sk1);

    return 0;
}
```

- **Output**

```
The stack length is 3.
The element at the top of the stack is 30.
```

15. stack_size

返回 stack_t 中的数据个数。

```
size_t stack_size(
    const stack_t* cpstack_stack
);
```

- **Parameters**

  cpstack_stack: 指向 stack_t 类型的指针。

- **Requirements**

  头文件 <cstl/cstack.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    stack_t* psk_sk1 = create_stack(int);

    if(psk_sk1 == NULL)
    {
```

```
        return -1;
    }

    stack_init(psk_sk1);

    stack_push(psk_sk1, 1);
    printf("The stack length is %d.\n", stack_size(psk_sk1));

    stack_push(psk_sk1, 2);
    printf("The stack length is now %d.\n", stack_size(psk_sk1));

    stack_destroy(psk_sk1);

    return 0;
}
```

- **Output**

```
The stack length is 1.
The stack length is now 2.
```

16. stack_top

访问 stack_t 栈顶数据。

```
void* stack_top(
    const stack_t* cpstack_stack
);
```

- **Parameters**

cpstack_stack: 指向 stack_t 类型的指针。

- **Remarks**

返回栈顶数据的指针，如果 stack_t 为空，程序的行为未定义。

- **Requirements**

头文件 <cstl/cstack.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    stack_t* psk_sk1 = create_stack(int);

    if(psk_sk1 == NULL)
    {
        return -1;
```

```
    }

    stack_init(psk_sk1);

    stack_push(psk_sk1, 1);
    stack_push(psk_sk1, 5);

    printf("The top integer of the stack sk1 is %d.\n", *(int*)stack_top(psk_sk1));
    *(int*)stack_top(psk_sk1) -= 1;
    printf("The top integer of the stack sk1 is %d.\n", *(int*)stack_top(psk_sk1));

    stack_destroy(psk_sk1);

    return 0;
}
```

- **Output**

```
The top integer of the stack sk1 is 5.
The top integer of the stack sk1 is 4.
```

第十四节　　队列 queue_t

队列 queue_t 是容器适配器，它是以序列容器为底层实现。queue_t 支持先入先出(FIFO)，只允许在后端插入数据在前端删除数据，不能够访问队列内部的数据。queue_t 不支持迭代器。

- **Typedefs**

queue_t	队列容器适配器类型。

- **Operation Functions**

create_queue	创建队列容器适配器类型。
queue_assign	为队列容器适配器类型赋值。
queue_back	访问队列容器适配器中最后一个数据。
queue_destroy	销毁队列容器适配器。
queue_empty	测试队列容器适配器是否为空。
queue_equal	测试两个队列容器适配器是否相等。
queue_front	访问队列容器适配器中第一个数据。
queue_greater	测试第一个队列容器适配器是否大于第二个队列容器适配器。
queue_greater_equal	测试第一个队列容器适配器是否大于等于第二个队列容器适配器。
queue_init	初始化一个空的队列容器适配器。
queue_init_copy	以拷贝的方式初始化一个队列容器适配器。
queue_less	测试第一个队列容器适配器是否小于第二个队列容器适配器。
queue_less_equal	测试第一个队列容器适配器是否小于等于第二个队列容器适配器。
queue_not_equal	测试两个队列容器适配器是否不等。

queue_pop	删除队列容器适配器中开头的数据。
queue_push	向队列容器适配器的末尾添加一个数据。
queue_size	返回队列容器适配器中的数据的个数。

1. queue_t

队列容器适配器类型。

- **Requirements**
  头文件 <cstl/cqueue.h>

- **Example**
  请参考 queue_t 类型的其他操作函数。

2. create_queue

创建 queue_t 类型。

```
queue_t* create_queue(
    type
);
```

- **Parameters**
  type: 数据类型描述。

- **Remarks**
  函数成功返回指向 queue_t 类型的指针，失败返回 NULL。

- **Requirements**
  头文件 <cstl/cqueue.h>

- **Example**
  请参考 queue_t 类型的其他操作函数。

3. queue_assign

为 queue_t 类型赋值。

```
void queue_assign(
    queue_t* pque_dest,
    const queue_t* cpque_src
);
```

- **Parameters**
  pque_dest: 指向被赋值的 queue_t 类型的指针。
  cpque_src: 指向赋值的 queue_t 类型的指针。

- **Remarks**
  要求两个 queue_t 类型保存的数据具有相同的类型，否则函数的行为未定义。

- **Requirements**

头文件 <cstl/cqueue.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    queue_t* pq_q1 = create_queue(int);
    queue_t* pq_q2 = create_queue(int);

    if(pq_q1 == NULL || pq_q2 == NULL)
    {
        return -1;
    }

    queue_init(pq_q1);
    queue_init(pq_q2);

    queue_push(pq_q1, 1);
    queue_push(pq_q1, 2);
    queue_push(pq_q1, 3);
    queue_push(pq_q2, 10);
    queue_push(pq_q2, 20);

    printf("The length of queue q1 is %d.\n", queue_size(pq_q1));

    queue_assign(pq_q1, pq_q2);

    printf("After assigning, the length of queue q1 is %d.\n", queue_size(pq_q1));

    queue_destroy(pq_q1);
    queue_destroy(pq_q2);

    return 0;
}
```

- **Output**

```
The length of queue q1 is 3.
After assigning, the length of queue q1 is 2.
```

4. queue_back

返回 queue_t 中末端的数据。

```
void* queue_back(
    const queue_t* cpque_queue
);
```

- **Parameters**

  cpque_queue: 指向 queue_t 类型的指针。

- **Remarks**

  返回 queue_t 中最后一个数据的指针，如果 queue_t 为空，程序的行为未定义。

- **Requirements**

  头文件 <cstl/cqueue.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    queue_t* pq_q1 = create_queue(int);

    if(pq_q1 == NULL)
    {
        return -1;
    }

    queue_init(pq_q1);

    queue_push(pq_q1, 10);
    queue_push(pq_q1, 20);

    printf("The element at the back of queue q1 is %d.\n",
        *(int*)queue_back(pq_q1));

    *(int*)queue_back(pq_q1) -= 5;

    printf("The element at the back of queue q1 is now %d.\n",
        *(int*)queue_back(pq_q1));

    queue_destroy(pq_q1);

    return 0;
}
```

- **Output**

```
The element at the back of queue q1 is 20.
The element at the back of queue q1 is now 15.
```

5. queue_destroy

销毁 queue_t 容器适配器类型。

```
void queue_destroy(
    queue_t* pque_queue
);
```

- **Parameters**
  pque_queue: 指向 queue_t 类型的指针。

- **Remarks**
  queue_t 使用之后一定要销毁，否则 queue_t 申请的资源不会被释放。

- **Requirements**
  头文件 <cstl/cqueue.h>

- **Example**
  请参考 queue_t 类型的其他操作函数。

6. queue_empty

测试 queue_t 是否为空。

```
bool_t queue_empty(
    const queue_t* cpque_queue
);
```

- **Parameters**
  cpque_queue: 指向 queue_t 类型的指针。

- **Remarks**
  queue_t 为空返回 true，否则返回 false。

- **Requirements**
  头文件 <cstl/cqueue.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    queue_t* pq_q1 = create_queue(int);
    queue_t* pq_q2 = create_queue(int);

    if(pq_q1 == NULL || pq_q2 == NULL)
    {
        return -1;
```

```
    }

    queue_init(pq_q1);
    queue_init(pq_q2);

    queue_push(pq_q1, 1);

    if(queue_empty(pq_q1))
    {
        printf("The queue q1 is empty.\n");
    }
    else
    {
        printf("The queue q1 is not empty.\n");
    }

    if(queue_empty(pq_q2))
    {
        printf("The queue q2 is empty.\n");
    }
    else
    {
        printf("The queue q2 is not empty.\n");
    }

    queue_destroy(pq_q1);
    queue_destroy(pq_q2);

    return 0;
}
```

- **Output**

```
The queue q1 is not empty.
The queue q2 is empty.
```

7. queue_equal

测试两个 queue_t 是否相等。

```
bool_t queue_equal(
    const queue_t* cpque_first,
    const queue_t* cpque_second
);
```

- **Parameters**

cpque_first: 指向第一个 queue_t 类型的指针。
cpque_second: 指向第二个 queue_t 类型的指针。

- **Remarks**

如果两个 queue_t 中的数据都对应相等，并且数据个数相等，则返回 true 否则返回 false，如果两个 queue_t 中保存的数据类型不同也认为是不等。

- **Requirements**

  头文件 <cstl/cqueue.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    queue_t* pq_q1 = create_queue(int);
    queue_t* pq_q2 = create_queue(int);
    queue_t* pq_q3 = create_queue(int);

    if(pq_q1 == NULL || pq_q2 == NULL || pq_q3 == NULL)
    {
        return -1;
    }

    queue_init(pq_q1);
    queue_init(pq_q2);
    queue_init(pq_q3);

    queue_push(pq_q1, 1);
    queue_push(pq_q2, 2);
    queue_push(pq_q3, 1);

    if(queue_equal(pq_q1, pq_q2))
    {
        printf("The queues q1 and q2 are equal.\n");
    }
    else
    {
        printf("The queues q1 and q2 are not equal.\n");
    }

    if(queue_equal(pq_q1, pq_q3))
    {
        printf("The queues q1 and q3 are equal.\n");
    }
    else
    {
        printf("The queues q1 and q3 are not equal.\n");
    }

    queue_destroy(pq_q1);
    queue_destroy(pq_q2);
    queue_destroy(pq_q3);
```

```
    return 0;
}
```

- **Output**

```
The queues q1 and q2 are not equal.
The queues q1 and q3 are equal.
```

8. queue_front

访问 queue_t 中开头的数据。

```
void* queue_front(
    const queue_t* cpque_queue
);
```

- **Parameters**

cpque_queue: 指向 queue_t 类型的指针。

- **Remarks**

返回 queue_t 中第一个数据的指针，如果 queue_t 为空，程序的行为未定义。

- **Requirements**

头文件 <cstl/cqueue.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    queue_t* pq_q1 = create_queue(int);

    if(pq_q1 == NULL)
    {
        return -1;
    }

    queue_init(pq_q1);

    queue_push(pq_q1, 10);
    queue_push(pq_q1, 20);
    queue_push(pq_q1, 30);

    printf("The element at the front of queue q1 is %d.\n",
        *(int*)queue_front(pq_q1));

    *(int*)queue_front(pq_q1) -= 5;
```

```
    printf("The element at the front of queue q1 is now %d.\n",
        *(int*)queue_front(pq_q1));

    queue_destroy(pq_q1);

    return 0;
}
```

- **Output**

```
The element at the front of queue q1 is 10.
The element at the front of queue q1 is now 5.
```

9. queue_greater

测试第一个 queue_t 是否大于第二个 queue_t。

```
bool_t queue_greater(
    const queue_t* cpque_first,
    const queue_t* cpque_second
);
```

- **Parameters**

cpque_first: 指向第一个 queue_t 类型的指针。
cpque_second: 指向第二个 queue_t 类型的指针。

- **Remarks**

这个函数要求两个 queue_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

头文件 <cstl/cqueue.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    queue_t* pq_q1 = create_queue(int);
    queue_t* pq_q2 = create_queue(int);
    queue_t* pq_q3 = create_queue(int);

    if(pq_q1 == NULL || pq_q2 == NULL || pq_q3 == NULL)
    {
        return -1;
    }
```

```
    queue_init(pq_q1);
    queue_init(pq_q2);
    queue_init(pq_q3);

    queue_push(pq_q1, 1);
    queue_push(pq_q1, 2);
    queue_push(pq_q1, 3);
    queue_push(pq_q2, 5);
    queue_push(pq_q2, 10);
    queue_push(pq_q3, 1);
    queue_push(pq_q3, 2);

    if(queue_greater(pq_q1, pq_q2))
    {
        printf("The queue q1 is greater than the queue q2.\n");
    }
    else
    {
        printf("The queue q1 is not greater than the queue q2.\n");
    }

    if(queue_greater(pq_q1, pq_q3))
    {
        printf("The queue q1 is greater than the queue q3.\n");
    }
    else
    {
        printf("The queue q1 is not greater than the queue q3.\n");
    }

    queue_destroy(pq_q1);
    queue_destroy(pq_q2);
    queue_destroy(pq_q3);

    return 0;
}
```

- **Output**

```
The queue q1 is not greater than the queue q2.
The queue q1 is greater than the queue q3.
```

10. queue_greater_equal

测试第一个 queue_t 是否大于等于第二个 queue_t。

```
bool_t queue_greater_equal(
    const queue_t* cpque_first,
    const queue_t* cpque_second
);
```

- **Parameters**

cpque_first: 指向第一个 queue_t 类型的指针。
cpque_second: 指向第二个 queue_t 类型的指针。

- **Remarks**

  这个函数要求两个 queue_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

  头文件 <cstl/cqueue.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    queue_t* pq_q1 = create_queue(int);
    queue_t* pq_q2 = create_queue(int);
    queue_t* pq_q3 = create_queue(int);

    if(pq_q1 == NULL || pq_q2 == NULL || pq_q3 == NULL)
    {
        return -1;
    }

    queue_init(pq_q1);
    queue_init(pq_q2);
    queue_init(pq_q3);

    queue_push(pq_q1, 1);
    queue_push(pq_q1, 2);
    queue_push(pq_q2, 5);
    queue_push(pq_q2, 10);
    queue_push(pq_q3, 1);
    queue_push(pq_q3, 2);

    if(queue_greater_equal(pq_q1, pq_q2))
    {
        printf("The queue q1 is greater than or equal to the queue q2.\n");
    }
    else
    {
        printf("The queue q1 is less than the queue q2.\n");
    }

    if(queue_greater_equal(pq_q1, pq_q3))
    {
        printf("The queue q1 is greater than or equal to the queue q3.\n");
```

```
    }
    else
    {
        printf("The queue q1 is less than the queue q3.\n");
    }

    queue_destroy(pq_q1);
    queue_destroy(pq_q2);
    queue_destroy(pq_q3);

    return 0;
}
```

- **Output**

```
The queue q1 is less than the queue q2.
The queue q1 is greater than or equal to the queue q3.
```

11. queue_init　queue_init_copy

初始化 queue_t 容器适配器类型。

```
void queue_init(
    queue_t* pque_queue
);

void queue_init_copy(
    queue_t* pque_queue,
    const queue_t* cpque_src
);
```

- **Parameters**

  pque_queue: 指向被初始化 queue_t 类型的指针。
  cpque_src: 指向用于初始化的 queue_t 类型的指针。

- **Remarks**

  第一个函数初始化一个空的 queue_t。
  第二个函数使用一个源 queue_t 来初始化 queue_t，数据的内容从源 queue_t 复制。

- **Requirements**

  头文件 <cstl/cqueue.h>

- **Example**

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

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

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

```
    queue_t* pq_q1 = create_queue(int);
    queue_t* pq_q2 = create_queue(int);

    if(pq_q1 == NULL || pq_q2 == NULL)
    {
        return -1;
    }

    /* Create an empty queue */
    queue_init(pq_q1);

    printf("The length of queue q1 is %d.\n", queue_size(pq_q1));
    /* Then push 3 elements */
    queue_push(pq_q1, 1);
    queue_push(pq_q1, 2);
    queue_push(pq_q1, 3);

    /* Create an copy queue q2 with q1 */
    queue_init_copy(pq_q2, pq_q1);
    printf("The length of queue q2 is %d.\n", queue_size(pq_q2));

    queue_destroy(pq_q1);
    queue_destroy(pq_q2);

    return 0;
}
```

- **Output**

```
The length of queue q1 is 0.
The length of queue q2 is 3.
```

12. queue_less

测试第一个 queue_t 是否小于第二个 queue_t。

```
bool_t queue_less(
    const queue_t* cpque_first,
    const queue_t* cpque_second
);
```

- **Parameters**

cpque_first: 指向第一个 queue_t 类型的指针。
cpque_second: 指向第二个 queue_t 类型的指针。

- **Remarks**

这个函数要求两个 queue_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

头文件 <cstl/cqueue.h>

- **Example**

```
/*
```

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

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

int main(int argc, char* argv[])
{
    queue_t* pq_q1 = create_queue(int);
    queue_t* pq_q2 = create_queue(int);
    queue_t* pq_q3 = create_queue(int);

    if(pq_q1 == NULL || pq_q2 == NULL || pq_q3 == NULL)
    {
        return -1;
    }

    queue_init(pq_q1);
    queue_init(pq_q2);
    queue_init(pq_q3);

    queue_push(pq_q1, 1);
    queue_push(pq_q1, 2);
    queue_push(pq_q2, 5);
    queue_push(pq_q2, 10);
    queue_push(pq_q3, 1);
    queue_push(pq_q3, 2);

    if(queue_less(pq_q1, pq_q2))
    {
        printf("The queue q1 is less than the queue q2.\n");
    }
    else
    {
        printf("The queue q1 is not less than the queue q2.\n");
    }

    if(queue_less(pq_q1, pq_q3))
    {
        printf("The queue q1 is less than the queue q3.\n");
    }
    else
    {
        printf("The queue q1 is not less than the queue q3.\n");
    }

    queue_destroy(pq_q1);
    queue_destroy(pq_q2);
    queue_destroy(pq_q3);

    return 0;
```

```
}
```

- **Output**

```
The queue q1 is less than the queue q2.
The queue q1 is not less than the queue q3.
```

13. queue_less_equal

测试第一个 queue_t 是否小于等于第二个 queue_t。

```
bool_t queue_less_equal(
    const queue_t* cpque_first,
    const queue_t* cpque_second
);
```

- **Parameters**

cpque_first: 指向第一个 queue_t 类型的指针。
cpque_second: 指向第二个 queue_t 类型的指针。

- **Remarks**

这个函数要求两个 queue_t 中保存的数据类型相同，如果不同导致函数的行为未定义。

- **Requirements**

头文件 <cstl/cqueue.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    queue_t* pq_q1 = create_queue(int);
    queue_t* pq_q2 = create_queue(int);
    queue_t* pq_q3 = create_queue(int);

    if(pq_q1 == NULL || pq_q2 == NULL || pq_q3 == NULL)
    {
        return -1;
    }

    queue_init(pq_q1);
    queue_init(pq_q2);
    queue_init(pq_q3);

    queue_push(pq_q1, 5);
    queue_push(pq_q1, 10);
    queue_push(pq_q2, 1);
```

```
    queue_push(pq_q2, 2);
    queue_push(pq_q3, 5);
    queue_push(pq_q3, 10);

    if(queue_less_equal(pq_q1, pq_q2))
    {
        printf("The queue q1 is less than or equal to the queue q2.\n");
    }
    else
    {
        printf("The queue q1 is greater than the queue q2.\n");
    }

    if(queue_less_equal(pq_q1, pq_q3))
    {
        printf("The queue q1 is less than or equal to the queue q3.\n");
    }
    else
    {
        printf("The queue q1 is greater than the queue q3.\n");
    }

    queue_destroy(pq_q1);
    queue_destroy(pq_q2);
    queue_destroy(pq_q3);

    return 0;
}
```

- **Output**

```
The queue q1 is greater than the queue q2.
The queue q1 is less than or equal to the queue q3.
```

14. queue_not_equal

测试两个 queue_t 是否不等。

```
bool_t queue_not_equal(
    const queue_t* cpque_first,
    const queue_t* cpque_second
);
```

- **Parameters**

cpque_first: 指向第一个 queue_t 类型的指针。
cpque_second: 指向第二个 queue_t 类型的指针。

- **Remarks**

如果两个 queue_t 中的数据都对应相等，并且数据个数相等，则返回 false 否则返回 true，如果两个 queue_t 中保存的数据类型不同也认为是不等。

- **Requirements**

头文件 <cstl/cqueue.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    queue_t* pq_q1 = create_queue(int);
    queue_t* pq_q2 = create_queue(int);
    queue_t* pq_q3 = create_queue(int);

    if(pq_q1 == NULL || pq_q2 == NULL || pq_q3 == NULL)
    {
        return -1;
    }

    queue_init(pq_q1);
    queue_init(pq_q2);
    queue_init(pq_q3);

    queue_push(pq_q1, 1);
    queue_push(pq_q2, 1);
    queue_push(pq_q2, 2);
    queue_push(pq_q3, 1);

    if(queue_not_equal(pq_q1, pq_q2))
    {
        printf("The queues q1 and q2 are not equal.\n");
    }
    else
    {
        printf("The queues q1 and q2 are equal.\n");
    }

    if(queue_not_equal(pq_q1, pq_q3))
    {
        printf("The queues q1 and q3 are not equal.\n");
    }
    else
    {
        printf("The queues q1 and q3 are equal.\n");
    }

    queue_destroy(pq_q1);
    queue_destroy(pq_q2);
    queue_destroy(pq_q3);
```

```
    return 0;
}
```

● **Output**

```
The queues q1 and q2 are not equal.
The queues q1 and q3 are equal.
```

15. queue_pop

删除 queue_t 开头的数据。

```
void queue_pop(
    queue_t* pque_queue
);
```

● **Parameters**

pque_queue: 指向 queue_t 类型的指针。

● **Remarks**

queue_t 为空，程序的行为未定义。

● **Requirements**

头文件 <cstl/cqueue.h>

● **Example**

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

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

int main(int argc, char* argv[])
{
    queue_t* pq_q1 = create_queue(int);

    if(pq_q1 == NULL)
    {
        return -1;
    }

    queue_init(pq_q1);

    queue_push(pq_q1, 10);
    queue_push(pq_q1, 20);
    queue_push(pq_q1, 30);

    printf("The queue length is %d.\n", queue_size(pq_q1));
    printf("The element at the front of the queue q1 is %d.\n",
        *(int*)queue_front(pq_q1));
```

```
    queue_pop(pq_q1);

    printf("After a pop, the queue length is %d.\n", queue_size(pq_q1));
    printf("After a pop, the element at the front of the queue q1 is %d.\n",
        *(int*)queue_front(pq_q1));

    queue_destroy(pq_q1);

    return 0;
}
```

- **Output**

```
The queue length is 3.
The element at the front of the queue q1 is 10.
After a pop, the queue length is 2.
After a pop, the element at the front of the queue q1 is 20.
```

16. queue_push

向 queue_t 的末尾插入一个数据。

```
void queue_push(
    queue_t* pque_queue,
    element
);
```

- **Parameters**

pque_queue: 指向 queue_t 类型的指针。
element: 压入 queue_t 的数据。

- **Requirements**

头文件 <cstl/cqueue.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    queue_t* pq_q1 = create_queue(int);

    if(pq_q1 == NULL)
    {
        return -1;
    }

    queue_init(pq_q1);
```

```
    queue_push(pq_q1, 10);
    queue_push(pq_q1, 20);
    queue_push(pq_q1, 30);

    printf("The queue length is %d.\n", queue_size(pq_q1));
    printf("The element at the front of the queue q1 is %d.\n",
        *(int*)queue_front(pq_q1));

    queue_destroy(pq_q1);

    return 0;
}
```

- **Output**

```
The queue length is 3.
The element at the front of the queue q1 is 10.
```

17. queue_size

返回 queue_t 中数据的个数。

```
size_t queue_size(
    const queue_t* cpque_queue
);
```

- **Parameters**

  cpque_queue: 指向 queue_t 类型的指针。

- **Requirements**

  头文件 <cstl/cqueue.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    queue_t* pq_q1 = create_queue(int);

    if(pq_q1 == NULL)
    {
        return -1;
    }

    queue_init(pq_q1);

    queue_push(pq_q1, 1);
```

```
    printf("The queue length is %d.\n", queue_size(pq_q1));

    queue_push(pq_q1, 2);
    printf("The queue length is now %d.\n", queue_size(pq_q1));

    queue_destroy(pq_q1);

    return 0;
}
```

- **Output**

```
The queue length is 1.
The queue length is now 2.
```

第十五节　优先队列 priority_queue_t

优先队列 priority_queue_t 是容器适配器，它是以序列容器为底层实现。它是一种带有优先级的队列，优先级最高的数据总是在顶部。优先队列允许在插入数据并且值允许删除和访问优先级最高的数据，不能够访问队列内部的数据。priority_queue_t 不支持迭代器和关系运算。

- **Typedefs**

priority_queue_t	优先队列容器适配器类型。

- **Operation Functions**

create_priority_queue	创建优先队列容器适配器类型。
priority_queue_assign	为优先队列容器适配器类型赋值。
priority_queue_destroy	销毁优先队列容器适配器类型。
priority_queue_empty	测试优先队列容器适配器是否为空。
priority_queue_init	初始化一个空的优先队列容器适配器类型。
priority_queue_init_copy	以拷贝的方式初始化一个优先队列容器适配器类型。
priority_queue_init_copy_range	使用指定的数据区间初始化一个优先队列容器适配器。
priority_queue_init_copy_range_ex	使用指定的数据区间和比较规则初始化一个优先队列容器适配器。
priority_queue_init_ex	使用指定的比较规则初始化一个优先队列容器适配器。
priority_queue_pop	删除优先队列容器适配器中优先级最高的数据。
priority_queue_push	向优先队列容器适配器中插入一个数据。
priority_queue_size	返回优先队列容器适配器中数据的个数。
priority_queue_top	访问优先队列容器适配器中优先级最高的数据。

1.　priority_queue_t

优先队列容器适配器类型。

- **Requirements**

头文件 <cstl/cqueue.h>

- **Example**
  请参考 priority_queue_t 类型的其他操作函数。

2. create_priority_queue

创建 priority_queue_t 容器适配器类型。

```
priority_queue_t* create_priority_queue(
    type
);
```

- **Parameters**
  type: 数据类型描述。

- **Remarks**
  函数成功返回指向 priority_queue_t 类型的指针，失败返回 NULL。

- **Requirements**
  头文件 <cstl/cqueue.h>

- **Example**
  请参考 priority_queue_t 类型的其他操作函数。

3. priority_queue_assign

为 priority_queue_t 类型赋值。

```
void priority_queue_assign(
    priority_queue_t* ppque_dest,
    const priority_queue_t* cppque_src
);
```

- **Parameters**
  ppque_dest: 指向被赋值的 priority_queue_t 类型的指针。
  cppque_src: 指向赋值的 priority_queue_t 类型的指针。

- **Remarks**
  要求两个 priority_queue_t 类型保存的数据具有相同的类型，否则函数的行为未定义。

- **Requirements**
  头文件 <cstl/cqueue.h>

- **Example**

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

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

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

```
{
    priority_queue_t* ppq_pq1 = create_priority_queue(int);
    priority_queue_t* ppq_pq2 = create_priority_queue(int);

    if(ppq_pq1 == NULL || ppq_pq2 == NULL)
    {
        return -1;
    }

    priority_queue_init(ppq_pq1);
    priority_queue_init(ppq_pq2);

    priority_queue_push(ppq_pq1, 1);
    priority_queue_push(ppq_pq1, 2);
    priority_queue_push(ppq_pq1, 3);
    priority_queue_push(ppq_pq2, 10);
    priority_queue_push(ppq_pq2, 20);

    printf("The length of priority_queue pq1 is %d.\n",
        priority_queue_size(ppq_pq1));

    priority_queue_assign(ppq_pq1, ppq_pq2);

    printf("After assignment, the length of priority_queue pq1 is %d.\n",
        priority_queue_size(ppq_pq1));

    priority_queue_destroy(ppq_pq1);
    priority_queue_destroy(ppq_pq2);

    return 0;
}
```

- **Output**

```
The length of priority_queue pq1 is 3.
After assignment, the length of priority_queue pq1 is 2.
```

4. priority_queue_destroy

销毁 priority_queue_t 类型。

```
void priority_queue_destroy(
    priority_queue_t* ppque_pqueue
);
```

- **Parameters**

  ppque_pqueue: 指向 priority_queue_t 类型的指针。

- **Remarks**

  priority_queue_t 使用之后一定要销毁，否则 priority_queue_t 申请的资源不会被释放。

- **Requirements**

  头文件 <cstl/cqueue.h>

- **Example**

  请参考 priority_queue_t 类型的其他操作函数。

5. priority_queue_empty

测试 priority_queue_t 是否为空。

```
bool_t priority_queue_empty(
    const priority_queue_t* cppque_pqueue
);
```

- **Parameters**

  cppque_pqueue: 指向 priority_queue_t 类型的指针。

- **Remarks**

  priority_queue_t 为空返回 true，否则返回 false。

- **Requirements**

  头文件 <cstl/cqueue.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    priority_queue_t* ppq_pq1 = create_priority_queue(int);
    priority_queue_t* ppq_pq2 = create_priority_queue(int);

    if(ppq_pq1 == NULL || ppq_pq2 == NULL)
    {
        return -1;
    }

    priority_queue_init(ppq_pq1);
    priority_queue_init(ppq_pq2);

    priority_queue_push(ppq_pq1, 1);
    if(priority_queue_empty(ppq_pq1))
    {
        printf("The priority_queue pq1 is empty.\n");
    }
    else
    {
        printf("The priority_queue pq1 is not empty.\n");
    }
```

```
    if(priority_queue_empty(ppq_pq2))
    {
        printf("The priority_queue pq2 is empty.\n");
    }
    else
    {
        printf("The priority_queue pq2 is not empty.\n");
    }

    priority_queue_destroy(ppq_pq1);
    priority_queue_destroy(ppq_pq2);

    return 0;
}
```

- **Output**

```
The priority_queue pq1 is not empty.
The priority_queue pq2 is empty.
```

6. priority_queue_init priority_queue_init_copy priority_queue_init_copy_range priority_queue_init_copy_range_ex priority_queue_init_ex

初始化 priority_queue_t 容器适配器类型。

```
void priority_queue_init(
    priority_queue_t* ppque_pqueue
);

void priority_queue_init_copy(
    priority_queue_t* ppque_pqueue,
    const priority_queue_t* cppque_src
);

void priority_queue_init_copy_range(
    priority_queue_t* ppque_pqueue,
    random_access_iterator_t it_first,
    random_access_iterator_t it_last
);

void priority_queue_init_copy_range_ex(
    priority_queue_t* ppque_pqueue,
    random_access_iterator_t it_first,
    random_access_iterator_t it_last,
    binary_function_t bfun_compare
);

void priority_queue_init_ex(
    priority_queue_t* ppque_pqueue,
    binary_function_t bfun_compare
);
```

- **Parameters**

ppque_pqueue: 指向被初始化 priority_queue_t 类型的指针。

cppque_src: 指向用于初始化的 priority_queue_t 类型的指针。
it_begin: 用于初始化的数据区间的开始位置。
it_end: 用于初始化的数据区间的末尾位置。
bfun_compare: 自定义排序规则。

- **Remarks**

第一个函数初始化一个空的 priority_queue_t，使用与数据类型相关的小于操作函数作为默认的排序规则。

第二个函数使用一个源 priority_queue_t 来初始化 priority_queue_t，数据的内容和排序规则都从源 priority_queue_t 复制。

第三个函数使用指定的数据区间初始化一个 priority_queue_t，使用与数据类型相关的小于操作函数作为默认的排序规则。

第四个函数使用指定的数据区间初始化一个 priority_queue_t，使用用户指定的排序规则。

第五个函数初始化一个空的 priority_queue_t，使用用户指定的排序规则。

上面的函数要求迭代器和数据区间是有效的，无效的迭代器或数据区间导致函数的行为未定义。

- **Requirements**

头文件 <cstl/cqueue.h>

- Example

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

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

int main(int argc, char* argv[])
{
    vector_t* pvec_v5 = create_vector(int);
    priority_queue_t* ppq_pq1 = create_priority_queue(int);
    priority_queue_t* ppq_pq2 = create_priority_queue(int);
    priority_queue_t* ppq_pq3 = create_priority_queue(int);
    priority_queue_t* ppq_pq4 = create_priority_queue(int);
    priority_queue_t* ppq_pq5 = create_priority_queue(int);
    priority_queue_t* ppq_pq6 = create_priority_queue(int);
    vector_iterator_t it_v5;

    if(ppq_pq1 == NULL || ppq_pq2 == NULL || ppq_pq3 == NULL ||
       ppq_pq4 == NULL || ppq_pq5 == NULL || ppq_pq6 == NULL ||
       pvec_v5 == NULL)
    {
        return -1;
    }

    /* Create an empty priority_queue */
    priority_queue_init(ppq_pq1);
    printf("pq1 = ( ");
    while(!priority_queue_empty(ppq_pq1))
    {
```

```
        printf("%d ", *(int*)priority_queue_top(ppq_pq1));
        priority_queue_pop(ppq_pq1);
    }
    printf(")\n");

    /* Create an empty priority_queue and push 3 elements */
    priority_queue_init(ppq_pq2);
    priority_queue_push(ppq_pq2, 5);
    priority_queue_push(ppq_pq2, 15);
    priority_queue_push(ppq_pq2, 10);
    printf("pq2 = ( ");
    while(!priority_queue_empty(ppq_pq2))
    {
        printf("%d ", *(int*)priority_queue_top(ppq_pq2));
        priority_queue_pop(ppq_pq2);
    }
    printf(")\n");
    printf("After printing, pq2 has %d elements.\n",
        priority_queue_size(ppq_pq2));

    /*
     * Create an empty priority_queue with specific comparison function
     * and push 3 elements.
     */
    priority_queue_init_ex(ppq_pq3, fun_greater_int);
    priority_queue_push(ppq_pq3, 2);
    priority_queue_push(ppq_pq3, 1);
    priority_queue_push(ppq_pq3, 3);
    printf("pq3 = ( ");
    while(!priority_queue_empty(ppq_pq3))
    {
        printf("%d ", *(int*)priority_queue_top(ppq_pq3));
        priority_queue_pop(ppq_pq3);
    }
    printf(")\n");

    /* Create an copy priority_queue form pq1 */
    priority_queue_push(ppq_pq1, 100);
    priority_queue_push(ppq_pq1, 200);
    priority_queue_init_copy(ppq_pq4, ppq_pq1);
    printf("pq4 = ( ");
    while(!priority_queue_empty(ppq_pq4))
    {
        printf("%d ", *(int*)priority_queue_top(ppq_pq4));
        priority_queue_pop(ppq_pq4);
    }
    printf(")\n");

    /* Create an auxiliary vector v5 to be used to initialize pq5 */
    vector_init(pvec_v5);
```

```
    vector_push_back(pvec_v5, 10);
    vector_push_back(pvec_v5, 30);
    vector_push_back(pvec_v5, 20);
    printf("v5 = ( ");
    for(it_v5 = vector_begin(pvec_v5);
        !iterator_equal(it_v5, vector_end(pvec_v5));
        it_v5 = iterator_next(it_v5))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v5));
    }
    printf(")\n");

    /* Create a priority_queue pq5 by copying the range v5[first, last) */
    priority_queue_init_copy_range(ppq_pq5,
        vector_begin(pvec_v5), vector_end(pvec_v5));
    printf("pq5 = ( ");
    while(!priority_queue_empty(ppq_pq5))
    {
        printf("%d ", *(int*)priority_queue_top(ppq_pq5));
        priority_queue_pop(ppq_pq5);
    }
    printf(")\n");

    /*
     * Create a priority_queue pq6 by copying the range v5 [first, last) and
     * initialize with a comparison function greater.
     */
    priority_queue_init_copy_range_ex(ppq_pq6, vector_begin(pvec_v5),
        vector_end(pvec_v5), fun_greater_int);
    printf("pq6 = ( ");
    while(!priority_queue_empty(ppq_pq6))
    {
        printf("%d ", *(int*)priority_queue_top(ppq_pq6));
        priority_queue_pop(ppq_pq6);
    }
    printf(")\n");

    vector_destroy(pvec_v5);
    priority_queue_destroy(ppq_pq1);
    priority_queue_destroy(ppq_pq2);
    priority_queue_destroy(ppq_pq3);
    priority_queue_destroy(ppq_pq4);
    priority_queue_destroy(ppq_pq5);
    priority_queue_destroy(ppq_pq6);

    return 0;
}
```

- **Output**

```
pq1 = ( )
```

```
pq2 = ( 15 10 5 )
After printing, pq2 has 0 elements.
pq3 = ( 1 2 3 )
pq4 = ( 200 100 )
v5 = ( 10 30 20 )
pq5 = ( 30 20 10 )
pq6 = ( 10 20 30 )
```

7. priority_queue_pop

删除 priority_queue_t 中优先级最高的数据。

```
void priority_queue_pop(
    priority_queue_t* ppque_pqueue
);
```

- **Parameters**

pque_pqueue: 指向 priority_queue_t 类型的指针。

- **Remarks**

priority_queue_t 为空，程序的行为未定义。

- **Requirements**

头文件 <cstl/cqueue.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    priority_queue_t* ppq_pq1 = create_priority_queue(int);

    if(ppq_pq1 == NULL)
    {
        return -1;
    }

    priority_queue_init(ppq_pq1);

    priority_queue_push(ppq_pq1, 10);
    priority_queue_push(ppq_pq1, 20);
    priority_queue_push(ppq_pq1, 30);

    printf("The priority_queue length is %d.\n",
        priority_queue_size(ppq_pq1));
    printf("The element at the top of the priority_queue is %d.\n",
        *(int*)priority_queue_top(ppq_pq1));
```

```
    priority_queue_pop(ppq_pq1);

    printf("After a pop, the priority_queue length is %d.\n",
        priority_queue_size(ppq_pq1));
    printf("After a pop, the element at the top of the priority_queue is %d.\n",
        *(int*)priority_queue_top(ppq_pq1));

    priority_queue_destroy(ppq_pq1);

    return 0;
}
```

- **Output**

```
The priority_queue length is 3.
The element at the top of the priority_queue is 30.
After a pop, the priority_queue length is 2.
After a pop, the element at the top of the priority_queue is 20.
```

8. priority_queue_push

向 priority_queue_t 中添加一个数据。

```
void priority_queue_push(
    priority_queue_t* ppque_pqueue,
    element
);
```

- **Parameters**

ppque_pqueue: 指向 priority_queue_t 类型的指针。
element: 压入 priority_queue_t 的数据。

- **Requirements**

头文件 <cstl/cqueue.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    priority_queue_t* ppq_pq1 = create_priority_queue(int);

    if(ppq_pq1 == NULL)
    {
        return -1;
    }
```

```
    priority_queue_init(ppq_pq1);

    priority_queue_push(ppq_pq1, 10);
    priority_queue_push(ppq_pq1, 20);
    priority_queue_push(ppq_pq1, 30);

    printf("The priority_queue length is %d.\n",
        priority_queue_size(ppq_pq1));
    printf("The element at the top of the priority_queue is %d.\n",
        *(int*)priority_queue_top(ppq_pq1));

    priority_queue_destroy(ppq_pq1);

    return 0;
}
```

- **Output**

```
The priority_queue length is 3.
The element at the top of the priority_queue is 30.
```

9. priority_queue_size

返回 priority_queue_t 中数据的个数。

```
size_t priority_queue_size(
    const priority_queue_t* cppque_pqueue
);
```

- **Parameters**

cppque_pqueue: 指向 priority_queue_t 类型的指针。

- **Requirements**

头文件 <cstl/cqueue.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    priority_queue_t* ppq_pq1 = create_priority_queue(int);

    if(ppq_pq1 == NULL)
    {
        return -1;
    }
```

```
    priority_queue_init(ppq_pq1);

    priority_queue_push(ppq_pq1, 1);
    printf("The priority_queue length is %d.\n",
        priority_queue_size(ppq_pq1));

    priority_queue_push(ppq_pq1, 2);
    printf("The priority_queue length is now %d.\n",
        priority_queue_size(ppq_pq1));

    priority_queue_destroy(ppq_pq1);

    return 0;
}
```

- **Output**

```
The priority_queue length is 1.
The priority_queue length is now 2.
```

10. priority_queue_top

访问 priority_queue_t 中优先级最高的数据。

```
void* priority_queue_top(
    const priority_queue_t* cppque_pqueue
);
```

- **Parameters**

pque_pqueue: 指向 priority_queue_t 类型的指针。

- **Remarks**

priority_queue_t 为空，程序的行为未定义。

- **Requirements**

头文件 <cstl/cqueue.h>

- **Example**

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

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

int main(int argc, char* argv[])
{
    priority_queue_t* ppq_pq1 = create_priority_queue(int);

    if(ppq_pq1 == NULL)
    {
        return -1;
```

```
    }

    priority_queue_init(ppq_pq1);

    priority_queue_push(ppq_pq1, 10);
    priority_queue_push(ppq_pq1, 30);
    priority_queue_push(ppq_pq1, 20);

    printf("The priority_queue length is %d.\n",
        priority_queue_size(ppq_pq1));
    printf("The element at the top of the priority_queue is %d.\n",
        *(int*)priority_queue_top(ppq_pq1));

    priority_queue_destroy(ppq_pq1);

    return 0;
}
```

- **Output**

```
The priority_queue length is 3.
The element at the top of the priority_queue is 30.
```

第三章 迭代器

迭代器是一种泛化的指针：是指向容器中数据的指针。它通常提供了对数据进行迭代的操作，也提供了通过迭代器来获得数据和设置数据的操作。它是容器中的数据和算法的桥梁，算法通过它来操作容器中的数据，容器中的数据通过它可以使算法应用与该数据。

第一节 迭代器操作函数

由于容器结构的不同，迭代器也分为很多种类。libcstl 提供了多种迭代器操作函数，但是并不是每种操作函数都接受所有类型的迭代器。

- **Typedefs**

iterator_t	迭代器类型。
input_iterator_t	输入迭代器类型。
output_iterator_t	输出迭代器类型。
forward_iterator_t	前向迭代器类型。
bidirectional_iterator_t	双向迭代器类型。
random_access_iterator_t	随机访问迭代器类型。

- **Operation Functions**

iterator_at	使用下标通过迭代器随机访问数据。
iterator_equal	测试两个迭代器是否相等。
iterator_get_pointer	获得迭代器指向的数据的指针。

iterator_get_value	获得迭代器指向的数据。
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。
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_minus	计算两个迭代器的差值。
iterator_next	返回指向下一个数据的迭代器。
iterator_next_n	返回指向下 n 个数据的迭代器。
iterator_not_equal	测试两个迭代器是否不等。
iterator_prev	返回指向上一个数据的迭代器。
iterator_prev_n	返回指向上 n 个数据的迭代器。
iterator_set_value	设置迭代器指向的数据。

1. iterator_t

迭代器类型。

- **Remarks**

最基本的迭代器类型，它可以代替所有的迭代器类型。

- **Requirements**

头文件 <cstl/citerator.h> 或者任何 libcstl 头文件。

- **Example**

请参考 iterator_t 类型的其他操作函数。

2. input_iterator_t

输入迭代器类型。

- **Remarks**

input_iterator_t 迭代器类型支持获取数据，向前迭代，相等测试。

- **Requirements**

头文件 <cstl/citerator.h> 或者任何 libcstl 头文件。

- **Example**

请参考 input_iterator_t 类型的其他操作函数。

3. output_iterator_t

输出迭代器类型。

- **Remarks**

output_iterator_t 迭代器类型支持设置数据，向前迭代。

- **Requirements**
  头文件 <cstl/citerator.h> 或者任何 libcstl 头文件。

- **Example**
  请参考 output_iterator_t 类型的其他操作函数。

4. forward_iterator_t

前向迭代器。

- **Remarks**
  forward_iterator_t 迭代器类型支持获取数据，设置数据，向前迭代，相等测试。

- **Requirements**
  头文件 <cstl/citerator.h> 或者任何 libcstl 头文件。

- **Example**
  请参考 forward_iterator_t 类型的其他操作函数。

5. bidirectional_iterator_t

双向迭代器类型。

- **Remarks**
  bidirectional_iterator_t 迭代器类型支持获取数据，设置数据，双向迭代，相等测试。

- **Requirements**
  头文件 <cstl/citerator.h> 或者任何 libcstl 头文件。

- **Example**
  请参考 bidirectional_iterator_t 类型的其他操作函数。

6. random_access_iterator_t

随机访问迭代器类型。

- **Remarks**
  random_access_iterator_t 迭代器类型支持所有迭代器操作函数。

- **Requirements**
  头文件 <cstl/citerator.h> 或者任何 libcstl 头文件。

- **Example**
  请参考 random_access_iterator_t 类型的其他操作函数。

7. iterator_at

使用下标通过迭代器随机访问数据。

```
void* iterator_at(
    iterator_t it_iter,
```

```
    int n_index
);
```

- **Parameters**
  it_iter: 迭代器类型。
  n_index: 下标。

- **Remarks**
  it_iter 为 random_access_iterator_t 类型，n_index 为有效下标，否则程序的行为未定义。

- **Requirements**
  头文件 <cstl/citerator.h> 或者任何 libcstl 头文件。

- **Example**

```
/*
 * iterator_at.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;
}
```

- **Output**

```
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
```

8. iterator_equal

测试两个迭代器是否相等。

```
bool_t iterator_equal(
    iterator_t it_first,
    iterator_t it_second
);
```

- **Parameters**

it_first: 第一个迭代器类型。

it_second: 第二个迭代器类型。

- **Remarks**

it_first 和 it_second 为 input_iterator_t，forward_iterator_t，bidirectional_iterator_t，random_access_iterator_t 类型，否则程序的行为未定义。如果 it_first 和 it_second 类型不同则认为不等。

- **Requirements**

头文件 <cstl/citerator.h> 或者任何 libcstl 头文件。

- **Example**

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

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

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    iterator_t it_vec;
    iterator_t it_pos1;
    iterator_t it_pos2;
    int i = 0;

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

    vector_init(pvec_v1);

    for(i = 1; i < 6; ++i)
    {
        vector_push_back(pvec_v1, i * 2);
    }

    printf("The vector v1 is ( ");
    for(it_vec = vector_begin(pvec_v1);
        !iterator_equal(it_vec, vector_end(pvec_v1));
        it_vec = iterator_next(it_vec))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_vec));
    }
    printf(")\n");

    /* Initializing iterator_t it_pos1 and it_pos2 to the first element */
    it_pos1 = vector_begin(pvec_v1);
    it_pos2 = vector_begin(pvec_v1);

    printf("The iterator it_pos1 initially points to the first element: %d\n",
```

```
        *(int*)iterator_get_pointer(it_pos1));
    if(iterator_equal(it_pos1, it_pos2))
    {
        printf("The iterators are equal.\n");
    }
    else
    {
        printf("The iterators are not equal.\n");
    }

    it_pos1 = iterator_next(it_pos1);
    printf("The iterator it_pos1 now points to the second element: %d\n",
        *(int*)iterator_get_pointer(it_pos1));
    if(iterator_equal(it_pos1, it_pos2))
    {
        printf("The iterators are equal.\n");
    }
    else
    {
        printf("The iterators are not equal.\n");
    }

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The vector v1 is ( 2 4 6 8 10 )
The iterator it_pos1 initially points to the first element: 2
The iterators are equal.
The iterator it_pos1 now points to the second element: 4
The iterators are not equal.
```

9. iterator_get_pointer

返回迭代器指向的数据的指针。

```
const void* iterator_get_pointer(
    iterator_t it_iter
);
```

- **Parameters**

it_iter: 迭代器类型。

- **Remarks**

it_iter 为 input_iterator_t，forward_iterator_t，bidirectional_iterator_t，random_access_iterator_t 类型，否则程序的行为未定义。不可以通过 iterator_get_pointer 来修改关联容器中的数据。

- **Requirements**

头文件 <cstl/citerator.h> 或者任何 libcstl 头文件。

- **Example**

  请参考 iterator_at 操作函数。

10. iterator_get_value

获得迭代器指向的数据的内容。

```
void iterator_get_value(
    iterator_t it_iter,
    void* pv_value
);
```

- **Parameters**

  it_iter: 迭代器类型。
  pv_value: 获取的数据内容。

- **Remarks**

  it_iter 为 input_iterator_t，forward_iterator_t，bidirectional_iterator_t，random_access_iterator_t 类型，否则程序的行为未定义。

- **Requirements**

  头文件 <cstl/citerator.h> 或者任何 libcstl 头文件。

- **Example**

  请参考 iterator_at 操作函数。

11. iterator_greater

测试第一个迭代器是否大于第二个迭代器。

```
bool_t iterator_greater(
    iterator_t it_first,
    iterator_t it_second
);
```

- **Parameters**

  it_first: 第一个迭代器类型。
  it_second: 第二个迭代器类型。

- **Remarks**

  it_first 和 it_second 为 random_access_iterator_t 类型，否则程序的行为未定义。

- **Requirements**

  头文件 <cstl/citerator.h> 或者任何 libcstl 头文件。

- **Example**

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

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

```
int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    iterator_t it_vec;
    iterator_t it_pos1;
    iterator_t it_pos2;
    int i = 0;

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

    vector_init(pvec_v1);

    for(i = 1; i < 6; ++i)
    {
        vector_push_back(pvec_v1, i * 2);
    }

    printf("The vector v1 is ( ");
    for(it_vec = vector_begin(pvec_v1);
        !iterator_equal(it_vec, vector_end(pvec_v1));
        it_vec = iterator_next(it_vec))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_vec));
    }
    printf(")\n");

    /* Initializing iterator_t it_pos1 and it_pos2 to the first element */
    it_pos1 = vector_begin(pvec_v1);
    it_pos2 = vector_begin(pvec_v1);

    printf("The iterator it_pos1 initially points to the first element: %d\n",
        *(int*)iterator_get_pointer(it_pos1));
    if(iterator_greater(it_pos1, it_pos2))
    {
        printf("The iterator it_pos1 is greater than the iterator it_pos2.\n");
    }
    else
    {
        printf("The iterator it_pos1 is less than or "
               "equal to the iterator it_pos2.\n");
    }

    it_pos1 = iterator_next(it_pos1);
    printf("The iterator it_pos1 now points to the second element: %d\n",
        *(int*)iterator_get_pointer(it_pos1));
    if(iterator_greater(it_pos1, it_pos2))
    {
```

```
        printf("The iterator it_pos1 is greater than the iterator it_pos2.\n");
    }
    else
    {
        printf("The iterator it_pos1 is less than or "
               "equal to the iterator it_pos2.\n");
    }

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The vector v1 is ( 2 4 6 8 10 )
The iterator it_pos1 initially points to the first element: 2
The iterator it_pos1 is less than or equal to the iterator it_pos2.
The iterator it_pos1 now points to the second element: 4
The iterator it_pos1 is greater than the iterator it_pos2.
```

12. iterator_greater_equal

测试第一个迭代器是否大于等于第二个迭代器。

```
bool_t iterator_greater_equal(
    iterator_t it_first,
    iterator_t it_second
);
```

- **Parameters**

it_first: 第一个迭代器类型。
it_second: 第二个迭代器类型。

- **Remarks**

it_first 和 it_second 为 random_access_iterator_t 类型，否则程序的行为未定义。

- **Requirements**

头文件 <cstl/citerator.h> 或者任何 libcstl 头文件。

- **Example**

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

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

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

```
    iterator_t it_pos1;
    iterator_t it_pos2;
    int i = 0;

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

    vector_init(pvec_v1);

    for(i = 1; i < 6; ++i)
    {
        vector_push_back(pvec_v1, i * 2);
    }

    printf("The vector v1 is ( ");
    for(it_vec = vector_begin(pvec_v1);
        !iterator_equal(it_vec, vector_end(pvec_v1));
        it_vec = iterator_next(it_vec))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_vec));
    }
    printf(")\n");

    /* Initializing iterator_t it_pos1 and it_pos2 to the first element */
    it_pos1 = iterator_next_n(vector_begin(pvec_v1), 2);
    it_pos2 = iterator_next(vector_begin(pvec_v1));

    printf("The iterator it_pos1 initially points to the third element: %d\n",
        *(int*)iterator_get_pointer(it_pos1));
    printf("The iterator it_pos2 initially points to the second element: %d\n",
        *(int*)iterator_get_pointer(it_pos2));
    if(iterator_greater_equal(it_pos1, it_pos2))
    {
        printf("The iterator it_pos1 is greater than or "
               "equal to the iterator it_pos2.\n");
    }
    else
    {
        printf("The iterator it_pos1 is less than the iterator it_pos2.\n");
    }

    it_pos1 = iterator_prev(it_pos1);
    printf("The iterator it_pos1 now points to the second element: %d\n",
        *(int*)iterator_get_pointer(it_pos1));
    if(iterator_greater_equal(it_pos1, it_pos2))
    {
        printf("The iterator it_pos1 is greater than or "
               "equal to the iterator it_pos2.\n");
```

```
    }
    else
    {
        printf("The iterator it_pos1 is less than the iterator it_pos2.\n");
    }

    it_pos1 = iterator_prev(it_pos1);
    printf("The iterator it_pos1 now points to the first element: %d\n",
        *(int*)iterator_get_pointer(it_pos1));
    if(iterator_greater_equal(it_pos1, it_pos2))
    {
        printf("The iterator it_pos1 is greater than or "
               "equal to the iterator it_pos2.\n");
    }
    else
    {
        printf("The iterator it_pos1 is less than the iterator it_pos2.\n");
    }

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The vector v1 is ( 2 4 6 8 10 )
The iterator it_pos1 initially points to the third element: 6
The iterator it_pos2 initially points to the second element: 4
The iterator it_pos1 is greater than or equal to the iterator it_pos2.
The iterator it_pos1 now points to the second element: 4
The iterator it_pos1 is greater than or equal to the iterator it_pos2.
The iterator it_pos1 now points to the first element: 2
The iterator it_pos1 is less than the iterator it_pos2.
```

13. iterator_less

测试第一个迭代器是否小于第二个迭代器。

```
bool_t iterator_less(
    iterator_t it_first,
    iterator_t it_second
);
```

- **Parameters**

it_first: 第一个迭代器类型。
it_second: 第二个迭代器类型。

- **Remarks**

it_first 和 it_second 为 random_access_iterator_t 类型，否则程序的行为未定义。

- **Requirements**

头文件 <cstl/citerator.h> 或者任何 libcstl 头文件。

- **Example**

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

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

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    iterator_t it_vec;
    iterator_t it_pos1;
    iterator_t it_pos2;
    int i = 0;

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

    vector_init(pvec_v1);

    for(i = 1; i < 6; ++i)
    {
        vector_push_back(pvec_v1, i * 2);
    }

    printf("The vector v1 is ( ");
    for(it_vec = vector_begin(pvec_v1);
        !iterator_equal(it_vec, vector_end(pvec_v1));
        it_vec = iterator_next(it_vec))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_vec));
    }
    printf(")\n");

    /* Initializing iterator_t it_pos1 and it_pos2 to the first element */
    it_pos1 = vector_begin(pvec_v1);
    it_pos2 = iterator_next(vector_begin(pvec_v1));

    printf("The iterator it_pos1 initially points to the first element: %d\n",
        *(int*)iterator_get_pointer(it_pos1));
    printf("The iterator it_pos2 initially points to the second element: %d\n",
        *(int*)iterator_get_pointer(it_pos2));
    if(iterator_less(it_pos1, it_pos2))
    {
        printf("The iterator it_pos1 is less than the iterator it_pos2.\n");
    }
    else
```

```
    {
        printf("The iterator it_pos1 is greater than or "
               "equal to the iterator it_pos2.\n");
    }

    it_pos1 = iterator_next(it_pos1);
    printf("The iterator it_pos1 now points to the second element: %d\n",
        *(int*)iterator_get_pointer(it_pos1));
    if(iterator_less(it_pos1, it_pos2))
    {
        printf("The iterator it_pos1 is less than the iterator it_pos2.\n");
    }
    else
    {
        printf("The iterator it_pos1 is greater than or "
               "equal to the iterator it_pos2.\n");
    }

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The vector v1 is ( 2 4 6 8 10 )
The iterator it_pos1 initially points to the first element: 2
The iterator it_pos2 initially points to the second element: 4
The iterator it_pos1 is less than the iterator it_pos2.
The iterator it_pos1 now points to the second element: 4
The iterator it_pos1 is greater than or equal to the iterator it_pos2.
```

14. iterator_less_equal

测试第一个迭代器是否小于等于第二个迭代器。

```
bool_t iterator_less_equal(
    iterator_t it_first,
    iterator_t it_second
);
```

- **Parameters**
  it_first: 第一个迭代器类型。
  it_second: 第二个迭代器类型。

- **Remarks**
  it_first 和 it_second 为 random_access_iterator_t 类型，否则程序的行为未定义。

- **Requirements**
  头文件 <cstl/citerator.h> 或者任何 libcstl 头文件。

- **Example**

```
/*
```

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

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

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    iterator_t it_vec;
    iterator_t it_pos1;
    iterator_t it_pos2;
    int i = 0;

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

    vector_init(pvec_v1);

    for(i = 1; i < 6; ++i)
    {
        vector_push_back(pvec_v1, i * 2);
    }

    printf("The vector v1 is ( ");
    for(it_vec = vector_begin(pvec_v1);
        !iterator_equal(it_vec, vector_end(pvec_v1));
        it_vec = iterator_next(it_vec))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_vec));
    }
    printf(")\n");

    /* Initializing iterator_t it_pos1 and it_pos2 to the first element */
    it_pos1 = iterator_next_n(vector_begin(pvec_v1), 2);
    it_pos2 = iterator_next(vector_begin(pvec_v1));

    printf("The iterator it_pos1 initially points to the third element: %d\n",
        *(int*)iterator_get_pointer(it_pos1));
    printf("The iterator it_pos2 initially points to the second element: %d\n",
        *(int*)iterator_get_pointer(it_pos2));
    if(iterator_less_equal(it_pos1, it_pos2))
    {
        printf("The iterator it_pos1 is less than or "
               "equal to the iterator it_pos2.\n");
    }
    else
    {
```

```
        printf("The iterator it_pos1 is greater than the iterator it_pos2.\n");
    }

    it_pos1 = iterator_prev(it_pos1);
    printf("The iterator it_pos1 now points to the second element: %d\n",
        *(int*)iterator_get_pointer(it_pos1));
    if(iterator_less_equal(it_pos1, it_pos2))
    {
        printf("The iterator it_pos1 is less than or "
               "equal to the iterator it_pos2.\n");
    }
    else
    {
        printf("The iterator it_pos1 is greater than the iterator it_pos2.\n");
    }

    it_pos1 = iterator_prev(it_pos1);
    printf("The iterator it_pos1 now points to the first element: %d\n",
        *(int*)iterator_get_pointer(it_pos1));
    if(iterator_less_equal(it_pos1, it_pos2))
    {
        printf("The iterator it_pos1 is less than or "
               "equal to the iterator it_pos2.\n");
    }
    else
    {
        printf("The iterator it_pos1 is greater than the iterator it_pos2.\n");
    }

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The vector v1 is ( 2 4 6 8 10 )
The iterator it_pos1 initially points to the third element: 6
The iterator it_pos2 initially points to the second element: 4
The iterator it_pos1 is greater than the iterator it_pos2.
The iterator it_pos1 now points to the second element: 4
The iterator it_pos1 is less than or equal to the iterator it_pos2.
The iterator it_pos1 now points to the first element: 2
The iterator it_pos1 is less than or equal to the iterator it_pos2.
```

15. iterator_minus

求两个迭代器的差。

```
int iterator_minus(
    iterator_t it_first,
    iterator_t it_second
);
```

- **Parameters**
  it_first: 第一个迭代器类型。
  it_second: 第二个迭代器类型。

- **Remarks**
  it_first 和 it_second 为 random_access_iterator_t 类型，否则程序的行为未定义。

- **Requirements**
  头文件 <cstl/citerator.h> 或者任何 libcstl 头文件。

- **Example**
  请参考 iterator_at 操作函数。

16. iterator_next

获得指向下一个数据的迭代器。

```
iterator_t iterator_next(
    iterator_t it_iter
);
```

- **Parameters**
  it_iter: 迭代器类型。

- **Remarks**
  it_iter 为 input_iterator_t，forward_iterator_t，bidirectional_iterator_t，random_access_iterator_t 类型，否则程序的行为未定义。

- **Requirements**
  头文件 <cstl/citerator.h> 或者任何 libcstl 头文件。

- **Example**

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

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

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    iterator_t it_vec;
    iterator_t it_pos;
    int i = 0;

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

```
    vector_init(pvec_v1);

    for(i = 0; i < 6; ++i)
    {
        vector_push_back(pvec_v1, i * 2);
    }

    printf("The vector v1 is ( ");
    for(it_vec = vector_begin(pvec_v1);
        !iterator_equal(it_vec, vector_end(pvec_v1));
        it_vec = iterator_next(it_vec))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_vec));
    }
    printf(")\n");

    it_pos = vector_begin(pvec_v1);

    printf("The iterator it_pos initially points to the first element: %d\n",
        *(int*)iterator_get_pointer(it_pos));

    it_pos = iterator_next(it_pos);
    printf("The iterator it_pos now points to the second element: %d\n",
        *(int*)iterator_get_pointer(it_pos));

    it_pos = iterator_next_n(it_pos, 3);
    printf("The iterator it_pos now points to the fifth element: %d\n",
        *(int*)iterator_get_pointer(it_pos));

    it_pos = iterator_next_n(it_pos, -2);
    printf("The iterator it_pos now points to the third element: %d\n",
        *(int*)iterator_get_pointer(it_pos));

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The vector v1 is ( 0 2 4 6 8 10 )
The iterator it_pos initially points to the first element: 0
The iterator it_pos now points to the second element: 2
The iterator it_pos now points to the fifth element: 8
The iterator it_pos now points to the third element: 4
```

17. iterator_next_n

获得指向下 n 个数据的迭代器。

```
iterator_t iterator_next_n(
    iterator_t it_iter,
    int n_step
```

```
);
```

- **Parameters**
  it_iter: 迭代器类型。
  n_step: 迭代器向前移动的步数。

- **Remarks**
  it_iter 为 random_access_iterator_t 类型，否则程序的行为未定义。

- **Requirements**
  头文件 <cstl/citerator.h> 或者任何 libcstl 头文件。

- **Example**
  请参考 iterator_next 操作函数。

18. iterator_not_equal

测试两个迭代器是否不等。

```
bool_t iterator_not_equal(
    iterator_t it_first,
    iterator_t it_second
);
```

- **Parameters**
  it_first: 第一个迭代器类型。
  it_second: 第二个迭代器类型。

- **Remarks**
  it_first 和 it_second 为 input_iterator_t，forward_iterator_t，bidirectional_iterator_t，random_access_iterator_t 类型，否则程序的行为未定义。如果 it_first 和 it_second 类型不同则认为不等。

- **Requirements**
  头文件 <cstl/citerator.h> 或者任何 libcstl 头文件。

- **Example**

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

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

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    iterator_t it_vec;
    iterator_t it_pos1;
    iterator_t it_pos2;
    int i = 0;

    if(pvec_v1 == NULL)
```

```
    {
        return -1;
    }

    vector_init(pvec_v1);

    for(i = 1; i < 6; ++i)
    {
        vector_push_back(pvec_v1, i * 2);
    }

    printf("The vector v1 is ( ");
    for(it_vec = vector_begin(pvec_v1);
        !iterator_equal(it_vec, vector_end(pvec_v1));
        it_vec = iterator_next(it_vec))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_vec));
    }
    printf(")\n");

    /* Initializing iterator_t it_pos1 and it_pos2 to the first element */
    it_pos1 = vector_begin(pvec_v1);
    it_pos2 = vector_begin(pvec_v1);

    printf("The iterator it_pos1 initially points to the first element: %d\n",
        *(int*)iterator_get_pointer(it_pos1));
    if(iterator_not_equal(it_pos1, it_pos2))
    {
        printf("The iterators are not equal.\n");
    }
    else
    {
        printf("The iterators are equal.\n");
    }

    it_pos1 = iterator_next(it_pos1);
    printf("The iterator it_pos1 now points to the second element: %d\n",
        *(int*)iterator_get_pointer(it_pos1));
    if(iterator_not_equal(it_pos1, it_pos2))
    {
        printf("The iterators are not equal.\n");
    }
    else
    {
        printf("The iterators are equal.\n");
    }

    vector_destroy(pvec_v1);

    return 0;
}
```

● **Output**

```
The vector v1 is ( 2 4 6 8 10 )
The iterator it_pos1 initially points to the first element: 2
The iterators are equal.
The iterator it_pos1 now points to the second element: 4
The iterators are not equal.
```

19. iterator_prev

获得指向前一个数据的迭代器。

```
iterator_t iterator_prev(
    iterator_t it_iter
);
```

● **Parameters**

it_iter: 迭代器类型。

● **Remarks**

it_iter 为 bidirectional_iterator_t，random_access_iterator_t 类型，否则程序的行为未定义。

● **Requirements**

头文件 <cstl/citerator.h> 或者任何 libcstl 头文件。

● **Example**

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

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

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    iterator_t it_vec;
    iterator_t it_pos;
    int i = 0;

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

    vector_init(pvec_v1);

    for(i = 0; i < 6; ++i)
    {
        vector_push_back(pvec_v1, i * 2);
    }
```

```
    printf("The vector v1 is ( ");
    for(it_vec = vector_begin(pvec_v1);
        !iterator_equal(it_vec, vector_end(pvec_v1));
        it_vec = iterator_next(it_vec))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_vec));
    }
    printf(")\n");

    it_pos = iterator_prev(vector_end(pvec_v1));

    printf("The iterator it_pos initially points to the last element: %d\n",
        *(int*)iterator_get_pointer(it_pos));

    it_pos = iterator_prev(it_pos);
    printf("The iterator it_pos now points to the fifth element: %d\n",
        *(int*)iterator_get_pointer(it_pos));

    it_pos = iterator_prev_n(it_pos, 3);
    printf("The iterator it_pos now points to the second element: %d\n",
        *(int*)iterator_get_pointer(it_pos));

    it_pos = iterator_prev_n(it_pos, -2);
    printf("The iterator it_pos now points to the fourth element: %d\n",
        *(int*)iterator_get_pointer(it_pos));

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The vector v1 is ( 0 2 4 6 8 10 )
The iterator it_pos initially points to the last element: 10
The iterator it_pos now points to the fifth element: 8
The iterator it_pos now points to the second element: 2
The iterator it_pos now points to the fourth element: 6
```

20. iterator_prev_n

返回指向前 n 个数据的迭代器。

```
iterator_t iterator_prev_n(
    iterator_t it_iter,
    int n_step
);
```

- **Parameters**

  it_iter: 迭代器类型。
  n_step: 迭代器向前移动的步数。

- **Remarks**

it_iter 为 random_access_iterator_t 类型，否则程序的行为未定义。

- **Requirements**
  头文件 <cstl/citerator.h> 或者任何 libcstl 头文件。

- **Example**
  请参考 iterator_prev 操作函数。

21. iterator_set_value

设置迭代器指向的数据。

```
void iterator_set_value(
    iterator_t it_iter,
    const void* cpv_value
);
```

- **Parameters**
  it_iter: 迭代器类型。
  cpv_value: 要设置的数据内容。

- **Remarks**

it_iter 为 output_iterator_t，forward_iterator_t，bidirectional_iterator_t，random_access_iterator_t 类型，否则程序的行为未定义。不能使用 iterator_set_value 操作修改关联容器中的数据值。

- **Requirements**
  头文件 <cstl/citerator.h> 或者任何 libcstl 头文件。

- **Example**

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

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

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    iterator_t it_vec;
    iterator_t it_pos;
    int n_value = 0;
    int i = 0;

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

    vector_init(pvec_v1);

    for(i = 0; i < 6; ++i)
```

```
    {
        vector_push_back(pvec_v1, i * 2);
    }

    printf("The vector v1 is ( ");
    for(it_vec = vector_begin(pvec_v1);
        !iterator_equal(it_vec, vector_end(pvec_v1));
        it_vec = iterator_next(it_vec))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_vec));
    }
    printf(")\n");

    it_pos = vector_begin(pvec_v1);
    n_value = 100;
    iterator_set_value(it_pos, &n_value);

    it_pos = iterator_next_n(it_pos, 3);
    n_value = -999;
    iterator_set_value(it_pos, &n_value);

    printf("After setting value, the vector v1 is ( ");
    for(it_vec = vector_begin(pvec_v1);
        !iterator_equal(it_vec, vector_end(pvec_v1));
        it_vec = iterator_next(it_vec))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_vec));
    }
    printf(")\n");

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The vector v1 is ( 0 2 4 6 8 10 )
After setting value, the vector v1 is ( 100 2 4 -999 8 10 )
```

第二节　迭代器辅助函数

迭代器辅助函数为所有类型的迭代器提供了只有 random_access_iterator_t 类型才能使用的操作函数，如一次迭代多步和获得迭代器之间的距离。

- **Operation Functions**

iterator_advance	第一迭代多步。
iterator_distance	获得两个迭代器之间的距离。

1. iterator_advance

一次迭代多步。

```
iterator_t iterator_advance(
    iterator_t it_iter,
    int n_step
);
```

- **Parameters**

  it_iter: 迭代器类型。

  n_step: 迭代器向前移动的步数。

- **Remarks**

it_iter 所有类型的迭代器，但是如果只有 bidirectional_iterator_t 和 random_iterator_t 可以使用负值，其他类型迭代器使用负值将使用绝对值代替。

- **Requirements**

  头文件 <cstl/citerator.h> 或者任何 libcstl 头文件。

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    iterator_t it_l;
    iterator_t it_pos;
    int i = 0;

    if(plist_l1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);

    for(i = 0; i < 10; ++i)
    {
        list_push_back(plist_l1, i);
    }

    printf("The list is ( ");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
```

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

    it_pos = list_begin(plist_l1);
    printf("The iterator it_pos initinally points to the first element: %d\n",
        *(int*)iterator_get_pointer(it_pos));

    it_pos = iterator_advance(it_pos, 4);
    printf("The iterator it_pos is advanced 4 steps forward to "
           "point to the fifth element: %d\n",
           *(int*)iterator_get_pointer(it_pos));

    it_pos = iterator_advance(it_pos, -3);
    printf("The iterator it_pos is moved 3 steps backward to "
           "point to the second element: %d\n",
           *(int*)iterator_get_pointer(it_pos));

    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
The list is ( 0 1 2 3 4 5 6 7 8 9 )
The iterator it_pos initinally points to the first element: 0
The iterator it_pos is advanced 4 steps forward to point to the fifth element: 4
The iterator it_pos is moved 3 steps backward to point to the second element: 1
```

2. iterator_distance

计算两个迭代器的距离。

```
int iterator_distance(
    iterator_t it_first,
    iterator_t it_second
);
```

- **Parameters**

  it_first: 第一个迭代器类型。
  it_second: 第二个迭代器类型。

- **Remarks**

  it_first 和 it_second 为所有迭代器类型。

- **Requirements**

  头文件 <cstl/citerator.h> 或者任何 libcstl 头文件。

- **Example**

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

```
 */

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    iterator_t it_l;
    iterator_t it_pos;
    int i = 0;

    if(plist_l1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);

    for(i = -1; i < 10; ++i)
    {
        list_push_back(plist_l1, i * 2);
    }

    printf("The list is ( ");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_l));
    }
    printf(")\n");

    it_pos = list_begin(plist_l1);
    printf("The iterator it_pos initinally points to the first element: %d\n",
        *(int*)iterator_get_pointer(it_pos));

    it_pos = iterator_advance(it_pos, 7);
    printf("The iterator it_pos is advanced 4 steps forward to "
           "point to the eighth element: %d\n",
           *(int*)iterator_get_pointer(it_pos));

    printf("The distance from list_begin to it_pos is: %d\n",
        iterator_distance(list_begin(plist_l1), it_pos));

    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
The list is ( -2 0 2 4 6 8 10 12 14 16 18 )
```

```
The iterator it_pos initinally points to the first element: -2
The iterator it_pos is advanced 4 steps forward to point to the eighth element: 12
The distance from list_begin to it_pos is: 7
```

第四章 算法

libcstl 提供的算法是通用的，它能够处理多种容器中的数据，这些处理过程是通过操作容器的迭代器来实现的。算法以迭代器组成的数据区间为对象，对其进行相应的操作。对于不同类型的容器，它的迭代器的种类不同，算法对于这些组成数据区间的迭代器类型是有要求的，迭代器的能力关系是：
random_access_iterator_t > bidirectional_iterator_t > forward_iterator_t > input_iterator_t/output_iterator_t
input_iterator_t 和 output_iterator_t 的能力属于同一个等级，但是能力是不同的。能力等级高的迭代器类型支持能力等级低的迭代器类型的所有操作，例如 bidirectional_iterator_t 支持所有 forward_iterator_t 的操作，此外它还支持一些后者不支持的操作，所以对于算法来说，一个算法要求一种类型的迭代器时，所有高于这个迭代器类型的迭代器都可以支持这个算法，例如一个算法支持 input_iterator_t，那么 random_access_iterator_t， bidirectional_iterator_t，forward_iterator_t 同样支持这个算法，但是 output_iterator_t 不支持这个算法。下面给出的算法函数原型的参数都是算法支持的能力最低的迭代器类型。

为了提供算法的可扩展行，同一个算法通常有几个变形，一个算法带有不同的后缀表示它们的功能有所不同：

- _if 后缀，为算法提供一个可扩展的版本，要求使用者提供一个自定义的规则，算法使用这个规则来代替默认的规则。例如 algo_find_if 在数据区间中查找满足指定规则的数据，而不是相等的数据。
- _copy 后缀，将算法的结果拷贝到目的数据区间，而不修改源数据。这样的算法返回目的数据区间中被覆盖的数据的末尾。

根据功能算法可以分为多个组，有些算法修改数据，有些算法不修改数据，有些算法只修改数据的顺序等等。最基本的算法使用默认的规则实现，这些默认规则通常是数据类型的小于操作函数。此外由于某些容器中的数据不能够被修改，如关联容器，那么修改数据的算法不能够应用到这些数据的迭代器构成的数据区间上。有些算法的功能，某些容器些提供了相应的操作函数，而且容器提供的操作函数效率更好，因为它们熟知容器内部结构，所以在使用这些算法的时候要有些考虑能否通过容器操作实现。

libcstl 提供通用的算法和数值算法，它们分别在<cstl/calgorithm.h>和<cstl/cnumeric.h>中声明。

第一节　通用算法

大部分算法使用数据类型的小于操作函数作为默认的比较规则，有些算法要求使用数据类型的等于操作函数作为默认的比较规则，如果数据类型没有提供等于操作函数，那么算法使用小于比较函数代替等于比较函数，如果数据类型没有提供小于比较函数，算法使用默认的比较规则进行比较。当算法要求输入自定的比较规则时，用户输入 NULL，算法使用默认的比较规则。

- **Algorithm Functions**

algo_adjacent_find	查找数据区间中两个相邻相等的数据。
algo_adjacent_find_if	查找数据区间中两个相邻并且符合指定规则的数据。
algo_binary_search	在使用默认比较规则排序的数据区间中查找指定数据。
algo_binary_search_if	在使用指定比较规则排序的数据区间中查找指定数据。
algo_copy	向目的数据区间中拷贝数据。
algo_copy_backward	以逆序的方式向目的数据区间拷贝数据。
algo_copy_n	向目的数据区间拷贝 n 个数据。
algo_count	统计数据区间中指定数据的个数。

algo_count_if	统计数据区间中符合指定规则的数据的个数。
algo_equal	测试两个数据区间是否相等。
algo_equal_if	使用指定规则测试两个数据区间是否相等。
algo_equal_range	返回使用默认比较规则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_fill	使用指定数据填充数据区间。
algo_fill_n	使用指定的数据向数据区间中填充 n 个数据。
algo_find	在数据区间中查找指定的数据。
algo_find_end	在数据区间中查找最后一个出现的子数据区间。
algo_find_end_if	在数据区间中查找最有一个符合指定规则的子数据区间。
algo_find_first_of	在数据区间中查找第一个同时出现在第二个数据区间中的数据。
algo_find_first_of_if	在数据区间中查找第一个与第二个数据区间中任意数据满足指定规则的数据。
algo_find_if	在数据区间中查找满足指定规则的数据。
algo_for_each	对数据区间中的每一个数据执行指定的规则。
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数据区间中的 n 个数据。
algo_includes	测试第一个有序数据区间中是否包含第二个有序数据区间的全部数据。
algo_includes_if	测试第一个使用指定比较规则排序的数据区间是否包含第二个使用指定比较规则排序的数据区间中所有的数据。
algo_inplace_merge	合并一个数据区间中的两个有序部分。
algo_inplace_merge_if	合并一个数据区间中两个使用指定比较规则排序的部分。
algo_is_heap	测试一个数据区间是否为堆。
algo_is_heap_if	测试一个数据区间是否为符合指定规则的堆。
algo_is_sorted	测试一个数据区间是否有序。
algo_is_sorted_if	测试一个数据区间是否是符合指定比较规则的有序区间。
algo_iter_swap	交换两个迭代器所指的数据内容。
algo_lexicographical_compare	将两个数据区间进行字典顺序比较。
algo_lexicographical_compare_3way	将两个数据区间进行字典顺序比较，返回 3 种结果。
algo_lexicographical_compare_3way_if	将两个数据区间依指定规则按照字典顺序比较，返回 3 种结果。
algo_lexicographical_compare_if	将两个数据区间依指定顺序按照字典顺序比较。
algo_lower_bound	在有序的数据区间中查找第一个等于指定数据的位置。
algo_lower_bound_if	在使用指定比较规则排序的数据区间中查找第一个等于指定数据的位置。
algo_make_heap	将一个数据区间转换成堆。
algo_make_heap_if	将一个数据区间转换成符合指定规则的堆。
algo_max	比较两个迭代器指向的数据，返回大的数据的迭代器。
algo_max_element	返回指向数据区间中最大的数据的迭代器。
algo_max_element_if	使用指定规则比较，返回指向数据区间中最大的数据的迭代器。
algo_max_if	使用指定的比较规则比较两个迭代器指向的数据，返回大的数据的迭代器。

algo_merge	合并两个有序数据区间。
algo_merge_if	合并两个使用指定比较规则排序的数据区间。
algo_min	比较两个迭代器所指的数据，返回较小的数据的迭代器。
algo_min_element	返回数据区间中指向最小数据的迭代器。
algo_min_element_if	使用指定的比较规则，返回数据区间中指向最小数据的迭代器。
algo_min_if	使用指定的比较规则比较两个迭代器指向的数据，返回较小的数据的迭代器。
algo_mismatch	返回两个数据区间中不等的数据迭代器对。
algo_mismatch_if	使用指定的比较规则，返回两个数据区间中不等的数据迭代器对。
algo_next_permutation	返回数据区间的下一个的排列。
algo_next_permutation_if	使用指定的比较规则，返回数据区间的下一个排列。
algo_nth_element	以第 n 个数据为界限将数据区间范围小于 n 和大于 n 的两部分。
algo_nth_element_if	以第 n 个数据为界限使用指定的比较规则将数据区间分为小于 n 和大于 n 的两部分。
algo_partial_sort	将数据区间部分排序。
algo_partial_sort_copy	将数据区间部分排序，将结果拷贝到目的数据区间。
algo_partial_sort_copy_if	使用指定比较规则将数据区间部分排序，将结果拷贝到目的数据区间。
algo_partial_sort_if	使用指定比较规则将数据区间部分排序。
algo_partition	按照指定规则将数据分为两部分。
algo_pop_heap	将堆中优先级最高的数据移除。
algo_pop_heap_if	将符合指定规则的堆中优先级最高的数据移除。
algo_prev_permutation	返回当前数据区间的上一个排列。
algo_prev_permutation_if	使用指定规则，返回当前数据区间的上一个排列。
algo_push_heap	向堆中添加一个数据。
algo_push_heap_if	向符合指定规则的堆中添加一个数据。
algo_random_sample	将数据区间中的数据随机抽样。
algo_random_sample_if	使用指定函数产生随机数，将数据区间中的数据随机抽样。
algo_random_sample_n	将数据区间中的数据随机抽出 n 个数据。
algo_random_sample_n_if	使用指定函数产生随机数，将数据区间中的数据随机抽样。
algo_random_shuffle	将数据区间中的数据随机重排。
algo_random_shuffle_if	使用指定的函数产生随机数，将数据区间中的数据随机重排。
algo_remove	移除数据区间中的指定数据。
algo_remove_copy	移除数据区间中的指定数据，将结果拷贝到目的数据区间中。
algo_remove_copy_if	移除数据区间中符合指定规则的数据，将结果拷贝到目的数据区间中。
algo_remove_if	移除数据区间中符合指定规则的数据。
algo_replace	替换数据区间中指定的数据。
algo_replace_copy	替换数据区间中指定的数据，并将结果拷贝到目的数据区间中。
algo_replace_copy_if	替换数据区间中符合指定规则的数据，将结果拷贝到目的数据区间中。
algo_replace_if	替换数据区间中符合指定规则的数据。

algo_reverse	将数据区间中的数据逆序。
algo_reverse_copy	将数据区间中的数据逆序，并将结果拷贝到目的数据区间中。
algo_rotate	将数据区间中的两部分数据调换。
algo_rotate_copy	将数据区间中的两部分数据调换，将结果拷贝到目的数据区间中。
algo_search	在数据区间中查找子数据区间。
algo_search_end	在数据区间中查找最后一个子数据区间。
algo_search_end_if	在数据区间中查找最后一个符合指定规则的子数据区间。
algo_search_if	在数据区间中查找符合指定规则的子数据区间。
algo_search_n	在数据区间中查找连续 n 个指定数据。
algo_search_n_if	在数据区间中查找连续 n 个符合指定规则的数据。
algo_set_difference	求两个数据区间的差集。
algo_set_difference_if	按照指定规则求两个数据区间的差集。
algo_set_intersection	求两个数据区间的交集。
algo_set_intersection_if	按照指定规则求两个数据区间的交集。
algo_set_symmetric_difference	求两个数据区间的对称差集。
algo_set_symmetric_difference_if	按照指定规则求两个数据区间的对称差集。
algo_set_union	求两个数据区间的并集。
algo_set_union_if	按照指定规则求两个数据区间的并集。
algo_sort	将数据区间排序。
algo_sort_heap	将堆转化成有序的数据区间。
algo_sort_heap_if	将符合指定规则的对转化成有序的数据区间。
algo_sort_if	按照指定规则将数据区间中的数据排序。
algo_stable_sort	将数据区间中的数据进行稳定排序。
algo_stable_sort_if	将数据区间中的数据按照指定规则进行稳定排序。
algo_stable_partition	将数据区间中的数据进行稳定的划分。
algo_swap	交换两个迭代器所指的数据内容。
algo_swap_ranges	交换两个数据区间中的数据。
algo_transform	将数据区间中的数据按照指定规则转换到目的数据区间。
algo_transform_binary	将两个数据区间中的数据按照指定规则转换到目的数据区间。
algo_unique	将数据区间中相邻且相等的数据移除。
algo_unique_copy	将数据区间中相邻且相等的数据移除并拷贝到目的数据区间。
algo_unique_copy_if	将数据区间中相邻且符合指定规则的数据移除并将结果拷贝到目的数据区间。
algo_unique_if	将数据区间中相邻且符合指定规则的数据移除。
algo_upper_bound	返回有序数据区间中第一个大于指定数据的位置。
algo_upper_bound_if	返回按照指定规则排序的数据区间中第一个大于指定数据的位置。

1. algo_adjacent_find　algo_adjacent_find_if

查找数据区间中相邻且符合指定规则的数据位置。

```
forward_iterator_t algo_adjacent_find(
    forward_iterator_t it_first,
    forward_iterator_t it_last
);

forward_iterator_t algo_adjacent_find_if(
    forward_iterator_t it_first,
    forward_iterator_t it_last,
    binary_function_t bfun_op
);
```

- **Parameters**

  it_first:　数据区间的开始位置。
  it_last:　数据区间的末尾位置。
  bfun_op:　比较函数。

- **Remarks**

  返回数据区间中第一对相邻且符合规则的第一个数据的迭代器。
  这个算法默认使用数据类型的等于操作函数。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

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

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

static void _twice(const void* cpv_first, const void* cpv_second, void* pv_output);

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    list_iterator_t it_l;

    if(plist_l1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);

    list_push_back(plist_l1, 50);
    list_push_back(plist_l1, 40);
    list_push_back(plist_l1, 10);
```

```
    list_push_back(plist_l1, 20);
    list_push_back(plist_l1, 20);

    printf("list l1 = ( ");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_l));
    }
    printf(")\n");

    it_l = algo_adjacent_find(list_begin(plist_l1), list_end(plist_l1));
    if(iterator_equal(it_l, list_end(plist_l1)))
    {
        printf("There are not two adjacent elements that are equal.\n");
    }
    else
    {
        printf("There are two adjacent elements that are equal.\n"
               "They have a value of %d.\n", *(int*)iterator_get_pointer(it_l));
    }

    it_l = algo_adjacent_find_if(list_begin(plist_l1), list_end(plist_l1), _twice);
    if(iterator_equal(it_l, list_end(plist_l1)))
    {
        printf("There are not two adjacent elements "
               "where the second is twice the first.\n");
    }
    else
    {
        printf("There are two adjacent elements "
               "where the second is twice the first.\n"
               "They have values of %d & %d.\n",
               *(int*)iterator_get_pointer(it_l),
               *(int*)iterator_get_pointer(iterator_next(it_l)));
    }

    list_destroy(plist_l1);

    return 0;
}

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

- **Output**

```
list l1 = ( 50 40 10 20 20 )
There are two adjacent elements that are equal.
```

```
They have a value of 20.
There are two adjacent elements where the second is twice the first.
They have values of 10 & 20.
```

2. algo_binary_search algo_binary_search_if

在有序的数据区间中查找符合规则的数据。

```
bool_t algo_binary_search(
    forward_iterator_t it_first,
    forward_iterator_t it_last,
    element
);

bool_t algo_binary_search_if(
    forward_iterator_t it_first,
    forward_iterator_t it_last,
    element,
    binary_function_t bfun_op
);
```

- **Parameters**

  it_first: 数据区间的开始位置。
  it_last: 数据区间的末尾位置。
  element: 指定的数据。
  bfun_op: 比较函数。

- **Remarks**

  有序的数据区间中包含之地的数据返回 true 否则返回 false。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

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

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

/* Return whether modulus of elem1 is less than modulus of elem2 */
static void _mod_lesser(const void* cpv_first,
    const void* cpv_second, void* pv_output);

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    list_iterator_t it_l;
    vector_t* pvec_v1 = create_vector(int);
```

```
    vector_iterator_t it_v;
    int i = 0;

    if(plist_l1 == NULL || pvec_v1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);
    vector_init(pvec_v1);

    list_push_back(plist_l1, 50);
    list_push_back(plist_l1, 10);
    list_push_back(plist_l1, 30);
    list_push_back(plist_l1, 20);
    list_push_back(plist_l1, 25);
    list_push_back(plist_l1, 5);
    list_sort(plist_l1);

    printf("l1 = ( ");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_l));
    }
    printf(")\n");

    if(algo_binary_search(list_begin(plist_l1), list_end(plist_l1), 10))
    {
        printf("There is an element in list l1 with a value equal to 10.\n");
    }
    else
    {
        printf("There is no element in list l1 with a value equal to 10.\n");
    }

    /* a binary_search under the binary predicate greater */
    list_sort_if(plist_l1, fun_greater_int);
    if(algo_binary_search_if(list_begin(plist_l1),
        list_end(plist_l1), 10, fun_greater_int))
    {
        printf("There is an element in list l1 "
               "with a value equal to 10 under greater than.\n");
    }
    else
    {
        printf("There is no element in list l1 "
               "with a value equal to 10 under greater than.\n");
    }
```

```
    /* a binary_search under the user-defined binary predicate mod_lesser */
    for(i = -2; i <= 4; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
    algo_sort_if(vector_begin(pvec_v1), vector_end(pvec_v1), _mod_lesser);

    printf("Ordred under mod_lesser, vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    if(algo_binary_search_if(vector_begin(pvec_v1),
        vector_end(pvec_v1), -3, _mod_lesser))
    {
        printf("There is an element with a value equal to -3 under mod_lesser.\n");
    }
    else
    {
        printf("There is no element with a value equal to -3 under mod_lesser.\n");
    }

    list_destroy(plist_l1);
    vector_destroy(pvec_v1);

    return 0;
}

static void _mod_lesser(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;
}
```

- **Output**

```
l1 = ( 5 10 20 25 30 50 )
There is an element in list l1 with a value equal to 10.
There is an element in list l1 with a value equal to 10 under greater than.
Ordred under mod_lesser, vector v1 = ( 0 -1 1 -2 2 3 4 )
There is an element with a value equal to -3 under mod_lesser.
```

3. algo_copy

将数据区间中的数据拷贝到目的数据区间中。

```
output_iterator_t algo_copy(
```

```
    input_iterator_t it_first,
    input_iterator_t it_last,
    output_iterator_t it_result
);
```

- **Parameters**

  it_first: 数据区间的开始位置。
  it_last: 数据区间的末尾位置。
  it_result: 目的数据区间开始位置。

- **Remarks**

  返回目的数据区间中拷贝数据的末尾。

  目的数据区间必须至少和源数据区间一样大，否则程序的行为是未定义的。此外关联容器不能作为目的数据区间。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

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

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

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

    if(pvec_v1 == NULL || pvec_v2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);

    for(i = 0; i < 6; ++i)
    {
        vector_push_back(pvec_v1, i * 10);
    }
    for(i = 0; i < 11; ++i)
    {
        vector_push_back(pvec_v2, i * 3);
    }
```

```
    printf("v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("v2 = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* To copy the first 3 elements of v1 into the middle of v2 */
    algo_copy(vector_begin(pvec_v1), iterator_next_n(vector_begin(pvec_v1), 3),
        iterator_next_n(vector_begin(pvec_v2), 4));
    printf("v2 with v1 insert = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* To shift the elements inserted into v2 two positions to the left */
    algo_copy(iterator_next_n(vector_begin(pvec_v2), 4),
        iterator_next_n(vector_begin(pvec_v2), 7),
        iterator_next_n(vector_begin(pvec_v2), 2));
    printf("v2 with shifted insert = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);

    return 0;
}
```

- **Output**

```
v1 = ( 0 10 20 30 40 50 )
```

```
v2 = ( 0 3 6 9 12 15 18 21 24 27 30 )
v2 with v1 insert = ( 0 3 6 9 0 10 20 21 24 27 30 )
v2 with shifted insert = ( 0 3 0 10 20 10 20 21 24 27 30 )
```

4. algo_copy_backward

以逆序的方式向目的数据区间中拷贝数据。

```
bidirectional_iterator_t algo_copy_backward(
    bidirectional_iterator_t it_first,
    bidirectional_iterator_t it_last,
    bidirectional_iterator_t it_result
);
```

- **Parameters**

it_first: 数据区间的开始位置。
it_last: 数据区间的末尾位置。
it_result: 目的数据区间开始位置。

- **Remarks**

返回目的数据区间中拷贝数据的末尾。

目的数据区间必须至少和源数据区间一样大，否则程序的行为是未定义的。此外关联容器不能作为目的数据区间。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

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

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

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

    if(pvec_v1 == NULL || pvec_v2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);

    for(i = 0; i < 6; ++i)
```

```
    {
        vector_push_back(pvec_v1, i * 10);
    }
    for(i = 0; i < 11; ++i)
    {
        vector_push_back(pvec_v2, i * 3);
    }

    printf("v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("v2 = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* To copy the first 3 elements of v1 into the middle of v2 */
    algo_copy_backward(vector_begin(pvec_v1),
        iterator_next_n(vector_begin(pvec_v1), 3),
        iterator_next_n(vector_begin(pvec_v2), 7));
    printf("v2 with v1 insert = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* To shift the elements inserted into v2 two positions to the left */
    algo_copy_backward(iterator_next_n(vector_begin(pvec_v2), 4),
        iterator_next_n(vector_begin(pvec_v2), 7),
        iterator_next_n(vector_begin(pvec_v2), 9));
    printf("v2 with shifted insert = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
```

```
    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);

    return 0;
}
```

- **Output**

```
v1 = ( 0 10 20 30 40 50 )
v2 = ( 0 3 6 9 12 15 18 21 24 27 30 )
v2 with v1 insert = ( 0 3 6 9 0 10 20 21 24 27 30 )
v2 with shifted insert = ( 0 3 6 9 0 10 0 10 20 27 30 )
```

5. algo_copy_n

向目的数据区间中拷贝 n 个数据。

```
output_iterator_t algo_copy_n(
    input_iterator_t it_first,
    size_t t_count,
    output_iterator_t it_result
);
```

- **Parameters**

it_first: 数据区间的开始位置。
t_count: 拷贝的数据的个数。
it_result: 目的数据区间开始位置。

- **Remarks**

返回目的数据区间中拷贝数据的末尾。

源数据区间和目的数据区间必须至少包含 n 个数据，否则程序的行为是未定义的。此外关联容器不能作为目的数据区间。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

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

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

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;
```

```
    if(pvec_v1 == NULL || pvec_v2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);

    for(i = 0; i < 6; ++i)
    {
        vector_push_back(pvec_v1, i * 10);
    }
    for(i = 0; i < 11; ++i)
    {
        vector_push_back(pvec_v2, i * 3);
    }

    printf("v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("v2 = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* To copy the first 3 elements of v1 into the middle of v2 */
    algo_copy_n(vector_begin(pvec_v1), 3,
        iterator_next_n(vector_begin(pvec_v2), 4));
    printf("v2 with v1 insert = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* To shift the elements inserted into v2 two positions to the left */
    algo_copy_n(iterator_next_n(vector_begin(pvec_v2), 4), 3,
        iterator_next_n(vector_begin(pvec_v2), 2));
    printf("v2 with shifted insert = ( ");
```

```
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);

    return 0;
}
```

- **Output**

```
v1 = ( 0 10 20 30 40 50 )
v2 = ( 0 3 6 9 12 15 18 21 24 27 30 )
v2 with v1 insert = ( 0 3 6 9 0 10 20 21 24 27 30 )
v2 with shifted insert = ( 0 3 0 10 20 10 20 21 24 27 30 )
```

6. algo_count

统计数据区间中指定数据的个数。

```
size_t algo_count(
    input_iterator_t it_first,
    input_iterator_t it_last,
    element
);
```

- **Parameters**

it_first: 数据区间的开始位置。
it_last: 数据区间的末尾位置。
element: 指定的数据。

- **Remarks**

返回目的数据区间中包含指定数据的个数。
这个算法默认使用数据类型的等于操作函数。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

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

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

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

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

    vector_init(pvec_v1);

    vector_push_back(pvec_v1, 10);
    vector_push_back(pvec_v1, 20);
    vector_push_back(pvec_v1, 10);
    vector_push_back(pvec_v1, 40);
    vector_push_back(pvec_v1, 10);

    printf("v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    printf("The number of 10s in v1 is: %u.\n",
        algo_count(vector_begin(pvec_v1), vector_end(pvec_v1), 10));

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
v1 = ( 10 20 10 40 10 )
The number of 10s in v1 is: 3.
```

7. algo_count_if

统计数据区间中包含符合指定规则的数据的个数。

```
size_t algo_count_if(
    input_iterator_t it_first,
    input_iterator_t it_last,
    unary_function_t ufun_op
);
```

- **Parameters**

it_first: 数据区间的开始位置。
it_last: 数据区间的末尾位置。

ufun_op: 指定的函数。

- **Remarks**

返回目的数据区间中包含指定数据的个数。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

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

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

static void _greater10(const void* cpv_input, void* pv_output);

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

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

    vector_init(pvec_v1);

    vector_push_back(pvec_v1, 10);
    vector_push_back(pvec_v1, 20);
    vector_push_back(pvec_v1, 10);
    vector_push_back(pvec_v1, 40);
    vector_push_back(pvec_v1, 10);

    printf("v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    printf("The number of elements in v1 greater than 10 is: %u.\n",
        algo_count_if(vector_begin(pvec_v1), vector_end(pvec_v1), _greater10));

    vector_destroy(pvec_v1);

    return 0;
```

```
}

static void _greater10(const void* cpv_input, void* pv_output)
{
    *(bool_t*)pv_output = *(int*)cpv_input > 10 ? true : false;
}
```

- **Output**

```
v1 = ( 10 20 10 40 10 )
The number of elements in v1 greater than 10 is: 2.
```

8. algo_equal algo_equal_if

测试两个数据区间是否相等。

```
bool_t algo_equal(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    input_iterator_t it_first2
);

bool_t algo_equal_if(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    input_iterator_t it_first2,
    binary_function_t bfun_op
);
```

- **Parameters**

it_first1: 第一个数据区间的开始位置。
it_last1: 第一个数据区间的末尾位置。
it_first2: 第二个数据区间的开始位置。
bfun_op: 比较函数。

- **Remarks**

如果第一个数据区间中的数据和第二个数据区间中的对应的数据都满足指定的比较规则，返回 true 否则返回 false。第二个数据区间必须至少和第一个数据区间一样大。这个算法默认使用类型的等于操作函数。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

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

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

static void _twice(const void* cpv_first, const void* cpv_second, void* pv_output);
```

```c
int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_t* pvec_v3 = create_vector(int);
    vector_iterator_t it_v;
    bool_t b_result = false;
    int i = 0;

    if(pvec_v1 == NULL || pvec_v2 == NULL || pvec_v3 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);
    vector_init(pvec_v3);

    for(i = 0; i < 6; ++i)
    {
        vector_push_back(pvec_v1, i * 5);
        vector_push_back(pvec_v2, i * 5);
        vector_push_back(pvec_v3, i * 10);
    }

    printf("v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    printf("v2 = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    printf("v3 = ( ");
    for(it_v = vector_begin(pvec_v3);
        !iterator_equal(it_v, vector_end(pvec_v3));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
```

```
    /* Test v1 and v2 for equality under identity */
    if(algo_equal(vector_begin(pvec_v1), vector_end(pvec_v1),
        vector_begin(pvec_v2)))
    {
        printf("The vectors v1 and v2 are equal under equality.\n");
    }
    else
    {
        printf("The vectors v1 and v2 are not equal under equality.\n");
    }

    /* Test v1 and v3 for equality under identity */
    if(algo_equal(vector_begin(pvec_v1), vector_end(pvec_v1),
        vector_begin(pvec_v3)))
    {
        printf("The vectors v1 and v3 are equal under equality.\n");
    }
    else
    {
        printf("The vectors v1 and v3 are not equal under equality.\n");
    }

    /* Test v1 and v3 for equality under twice */
    if(algo_equal_if(vector_begin(pvec_v1), vector_end(pvec_v1),
        vector_begin(pvec_v3), _twice))
    {
        printf("The vectors v1 and v3 are equal under twice.\n");
    }
    else
    {
        printf("The vectors v1 and v3 are not equal under twice.\n");
    }

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);
    vector_destroy(pvec_v3);

    return 0;
}

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

- **Output**

```
v1 = ( 0 5 10 15 20 25 )
v2 = ( 0 5 10 15 20 25 )
v3 = ( 0 10 20 30 40 50 )
The vectors v1 and v2 are equal under equality.
The vectors v1 and v3 are not equal under equality.
```

```
The vectors v1 and v3 are equal under twice.
```

9. algo_equal_range algo_equal_range_if

返回有序数据区间中等于指定数据的范围。

```
range_t algo_equal_range(
    forward_iterator_t it_first,
    forward_iterator_t it_last,
    element
);

range_t algo_equal_range_if(
    forward_iterator_t it_first,
    forward_iterator_t it_last,
    element
    binary_function_t bfun_op
);
```

- **Parameters**

it_first: 数据区间的开始位置。
it_last: 数据区间的末尾位置。
element: 指定的数据。
bfun_op: 比较函数。

- **Remarks**

返回一个范围，范围的开始是数据区间中第一个等于指定数据的位置，末尾是第一个大于指定数据的位置，如果数据区间中不包含该数据这个范围为空，如果数据区间中没有大于等于指定数据，那么范围的开始和末尾都指向数据区间的末尾。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

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

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

/* Return whether modulus of elem1 is less than modulus of elem2 */
static void _mod_lesser(const void* cpv_first,
    const void* cpv_second, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_t* pvec_v3 = create_vector(int);
```

```
    vector_iterator_t it_v;
    range_t r_result;
    int i = 0;

    if(pvec_v1 == NULL || pvec_v2 == NULL || pvec_v3 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);
    vector_init(pvec_v3);

    for(i = -1; i <= 4; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
    for(i = -3; i <= 0; ++i)
    {
        vector_push_back(pvec_v1, i);
    }

    algo_sort(vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("Original vector v1 with range sorted by the "
           "binary predicate less than is v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_assign(pvec_v2, pvec_v1);
    algo_sort_if(vector_begin(pvec_v2), vector_end(pvec_v2), fun_greater_int);
    printf("Original vector v2 with range sorted by the "
           "binary predicate greater than is v2 = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_assign(pvec_v3, pvec_v1);
    algo_sort_if(vector_begin(pvec_v3), vector_end(pvec_v3), _mod_lesser);
    printf("Original vector v3 with range sorted by the "
           "binary predicate greater than is v3 = ( ");
    for(it_v = vector_begin(pvec_v3);
```

```
        !iterator_equal(it_v, vector_end(pvec_v3));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* equal_range of 3 in v1 with default binary predicate less than */
    r_result = algo_equal_range(vector_begin(pvec_v1), vector_end(pvec_v1), 3);
    printf("The lower_bound in v1 for the element with a value of 3 is: %d.\n",
        *(int*)iterator_get_pointer(r_result.it_begin));
    printf("The upper_bound in v1 for the element with a value of 3 is: %d.\n",
        *(int*)iterator_get_pointer(r_result.it_end));
    printf("The equal_range in v1 for the element with a value of 3 is: ( ");
    for(it_v = r_result.it_begin;
        !iterator_equal(it_v, r_result.it_end);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* equal_range of 3 in v2 with the binary predicate greater than */
    r_result = algo_equal_range_if(vector_begin(pvec_v2),
        vector_end(pvec_v2), 3, fun_greater_int);
    printf("The lower_bound in v2 for the element with a value of 3 is: %d.\n",
        *(int*)iterator_get_pointer(r_result.it_begin));
    printf("The upper_bound in v2 for the element with a value of 3 is: %d.\n",
        *(int*)iterator_get_pointer(r_result.it_end));
    printf("The equal_range in v2 for the element with a value of 3 is: ( ");
    for(it_v = r_result.it_begin;
        !iterator_equal(it_v, r_result.it_end);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* equal_range of 3 in v3 with the binary predicate _mod_lesser */
    r_result = algo_equal_range_if(vector_begin(pvec_v3),
        vector_end(pvec_v3), 3, _mod_lesser);
    printf("The lower_bound in v3 for the element with a value of 3 is: %d.\n",
        *(int*)iterator_get_pointer(r_result.it_begin));
    printf("The upper_bound in v3 for the element with a value of 3 is: %d.\n",
        *(int*)iterator_get_pointer(r_result.it_end));
    printf("The equal_range in v3 for the element with a value of 3 is: ( ");
    for(it_v = r_result.it_begin;
        !iterator_equal(it_v, r_result.it_end);
        it_v = iterator_next(it_v))
    {
```

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

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);
    vector_destroy(pvec_v3);

    return 0;
}

static void _mod_lesser(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;
}
```

- **Output**

```
Original vector v1 with range sorted by the binary predicate less than is v1 = ( -3
-2 -1 -1 0 0 1 2 3 4 )
Original vector v2 with range sorted by the binary predicate greater than is v2 =
( 4 3 2 1 0 0 -1 -1 -2 -3 )
Original vector v3 with range sorted by the binary predicate greater than is v3 =
( 0 0 -1 -1 1 -2 2 -3 3 4 )
The lower_bound in v1 for the element with a value of 3 is: 3.
The upper_bound in v1 for the element with a value of 3 is: 4.
The equal_range in v1 for the element with a value of 3 is: ( 3 )
The lower_bound in v2 for the element with a value of 3 is: 3.
The upper_bound in v2 for the element with a value of 3 is: 2.
The equal_range in v2 for the element with a value of 3 is: ( 3 )
The lower_bound in v3 for the element with a value of 3 is: -3.
The upper_bound in v3 for the element with a value of 3 is: 4.
The equal_range in v3 for the element with a value of 3 is: ( -3 3 )
```

10. algo_fill

向数据区间中填充指定的数据。

```
void algo_fill(
    forward_iterator_t t_first,
    forward_iterator_t t_last,
    element
);
```

- **Parameters**

it_first: 数据区间的开始位置。
it_last: 数据区间的末尾位置。
element: 指定的数据。

- **Remarks**

关联容器不能作为被填充的数据区间。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

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

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

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

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

    vector_init(pvec_v1);

    for(i = 0; i < 10; ++i)
    {
        vector_push_back(pvec_v1, i * 5);
    }

    printf("Vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Fill the last 5 positions with a value of 2 */
    algo_fill(iterator_next_n(vector_begin(pvec_v1), 5), vector_end(pvec_v1), 2);

    printf("Modified v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);
```

```
    return 0;
}
```

- **Output**

```
Vector v1 = ( 0 5 10 15 20 25 30 35 40 45 )
Modified v1 = ( 0 5 10 15 20 2 2 2 2 2 )
```

11. algo_fill_n

向数据区间中填充 n 个数据

```
output_iterator_t algo_fill_n(
    forward_iterator_t it_first,
    size_t t_size,
    element
);
```

- **Parameters**

it_first: 数据区间的开始位置。
n_size: 填充数据的数据。
element: 指定的数据。

- **Remarks**

返回数据区间中被填充的数据的末尾迭代器。
要保证数据区间至少有 n 个数据。关联容器不能作为被填充的数据区间。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

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

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

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

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

    vector_init(pvec_v1);
```

```
    for(i = 0; i < 10; ++i)
    {
        vector_push_back(pvec_v1, i * 5);
    }

    printf("Vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Fill the last 5 positions with a value of 2 */
    algo_fill_n(iterator_next_n(vector_begin(pvec_v1), 5), 5, 2);

    printf("Modified v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
Vector v1 = ( 0 5 10 15 20 25 30 35 40 45 )
Modified v1 = ( 0 5 10 15 20 2 2 2 2 2 )
```

12. algo_find

在数据区间中查找指定的数据。

```
input_iterator_t algo_find(
    input_iterator_t it_first,
    input_iterator_t it_last,
    element
);
```

- **Parameters**

it_first: 数据区间的开始位置。
it_last: 数据区间的末尾位置。
element: 指定的数据。

- **Remarks**

  返回目的数据区间中指定数据的迭代器，如果数据区间中不包含指定的数据，返回数据区间的末尾。这个算法默认使用数据类型的等于操作函数。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

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

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

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    list_iterator_t it_l;

    if(plist_l1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);

    list_push_back(plist_l1, 40);
    list_push_back(plist_l1, 20);
    list_push_back(plist_l1, 10);
    list_push_back(plist_l1, 30);
    list_push_back(plist_l1, 10);

    printf("l1 = ( ");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_l));
    }
    printf(")\n");

    it_l = algo_find(list_begin(plist_l1), list_end(plist_l1), 10);
    if(iterator_equal(it_l, list_end(plist_l1)))
    {
        printf("There is no 10 in list l1.\n");
    }
    else
    {
        printf("There is a 10 in list l1 and it is followed by a %d.\n",
            *(int*)iterator_get_pointer(iterator_next(it_l)));
    }

    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
l1 = ( 40 20 10 30 10 )
There is a 10 in list l1 and it is followed by a 30.
```

13. algo_find_end algo_find_end_if

在数据区间中查找最后一个符合规则的子数据区间。

```
forward_iterator_t algo_find_end(
    forward_iterator_t it_first1,
    forward_iterator_t it_last1,
    forward_iterator_t it_first2,
    forward_iterator_t it_last2
);

forward_iterator_t algo_find_end_if(
    forward_iterator_t it_first1,
    forward_iterator_t it_last1,
    forward_iterator_t it_first2,
    forward_iterator_t it_last2,
    binary_function_t bfun_op
);
```

- **Parameters**

it_first1: 第一个数据区间的开始位置。
it_last1: 第一个数据区间的末尾位置。
it_first2: 子数据区间的开始位置。
it_last2: 子数据区间的末尾位置。
bfun_op: 指定的比较规则。

- **Remarks**

返回目的数据区间中最后一个符合规则的子数据区间的第一个数据的迭代器，如果不包含这个子数据区间，返回数据区间的末尾。

这个算法默认使用数据类型的等于操作函数。

这个算法与 algo_search_end 和 algo_search_end_if 功能相同，为了兼容 SGI STL 的接口保留这两个算法。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

请参考 algo_search_end 和 algo_search_end_if 算法。

14. algo_find_first_of algo_find_first_of_if

在第一个数据区间中查找第一个出现在第二个数据区间中出任意数据。

```
input_iterator_t algo_find_first_of(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    forward_iterator_t it_first2,
    forward_iterator_t it_last2
```

```
);

input_iterator_t algo_find_first_of_if(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    forward_iterator_t it_first2,
    forward_iterator_t it_last2,
    binary_function_t bfun_op
);
```

- **Parameters**

  it_first1: 第一个数据区间的开始位置。
  it_last1: 第一个数据区间的末尾位置。
  it_first2: 第二个数据区间的开始位置。
  it_last2: 第二个数据区间的末尾位置。
  bfun_op: 指定的比较规则。

- **Remarks**

  返回第一个同时出现在第一个和第二个数据区间中的数据的位置，如果没有返回第一个数据区间的末尾。这个算法默认使用数据类型的等于操作函数。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

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

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

static void _twice(const void* cpv_first, const void* cpv_second, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    list_t* plist_l1 = create_list(int);
    vector_iterator_t it_v;
    list_iterator_t it_l;
    int i = 0;

    if(pvec_v1 == NULL || pvec_v2 == NULL || plist_l1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);
    list_init(plist_l1);

    for(i = 0; i <= 5; ++i)
```

```
    {
        vector_push_back(pvec_v1, i * 5);
    }
    for(i = 0; i <= 5; ++i)
    {
        vector_push_back(pvec_v1, i * 5);
    }
    for(i = 2; i <= 4; ++i)
    {
        vector_push_back(pvec_v2, i * 10);
    }
    for(i = 3; i <= 4; ++i)
    {
        list_push_back(plist_l1, i * 5);
    }

    printf("Vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("List l1 = ( ");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_l));
    }
    printf(")\n");
    printf("Vector v2 = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Searching v1 for first match to l1 under identity */
    it_v = algo_find_first_of(vector_begin(pvec_v1), vector_end(pvec_v1),
        list_begin(plist_l1), list_end(plist_l1));

    if(iterator_equal(it_v, vector_end(pvec_v1)))
    {
        printf("There is no match of l1 in v1.\n");
    }
    else
    {
        printf("There is at least one match of l1 in v1\n"
               "and the first one begins at position %d.\n",
               iterator_distance(vector_begin(pvec_v1), it_v));
    }

    /* Searching v1 for a match to v2 under the binary predicate twice */
    it_v = algo_find_first_of_if(vector_begin(pvec_v1), vector_end(pvec_v1),
        vector_begin(pvec_v2), vector_end(pvec_v2), _twice);
    if(iterator_equal(it_v, vector_end(pvec_v1)))
```

```
    {
        printf("There is no match of v2 in v1.\n");
    }
    else
    {
        printf("There is a sequence of elements in v1 that are equivalent\n"
               "to those in v2 under the binary predicate twice\n"
               "and the first one begins at position %d.\n",
                iterator_distance(vector_begin(pvec_v1), it_v));
    }

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);
    list_destroy(plist_l1);

    return 0;
}

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

- **Output**

```
Vector v1 = ( 0 5 10 15 20 25 0 5 10 15 20 25 )
List l1 = ( 15 20 )
Vector v2 = ( 20 30 40 )
There is at least one match of l1 in v1
and the first one begins at position 3.
There is a sequence of elements in v1 that are equivalent
to those in v2 under the binary predicate twice
and the first one begins at position 2.
```

15. algo_find_if

在数据区间中查找符合指定规则的数据。

```
input_iterator_t algo_find_if(
    input_iterator_t it_first,
    input_iterator_t it_last,
    unary_function_t ufun_op
);
```

- **Parameters**

  it_first: 数据区间的开始位置。
  it_last: 数据区间的末尾位置。
  ufun_op: 指定的比较规则。

- **Remarks**

  返回目的数据区间中符合指定规则的迭代器，如果数据区间中不包含这样的数据，返回数据区间的末尾。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

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

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

static void _greater10(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    list_t* plist_l1 = create_list(int);
    list_iterator_t it_l;

    if(plist_l1 == NULL)
    {
        return -1;
    }

    list_init(plist_l1);

    list_push_back(plist_l1, 40);
    list_push_back(plist_l1, 20);
    list_push_back(plist_l1, 10);
    list_push_back(plist_l1, 30);
    list_push_back(plist_l1, 10);

    printf("l1 = ( ");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_l));
    }
    printf(")\n");

    it_l = algo_find_if(list_begin(plist_l1), list_end(plist_l1), _greater10);
    if(iterator_equal(it_l, list_end(plist_l1)))
    {
        printf("There is no element greater than 10 in list l1.\n");
    }
    else
    {
        printf("There is an element greater than 10 in list l1 is %d.\n",
            *(int*)iterator_get_pointer(it_l));
    }

    list_destroy(plist_l1);

    return 0;
}

static void _greater10(const void* cpv_input, void* pv_output)
{
    *(bool_t*)pv_output = *(int*)cpv_input > 10 ? true : false;
}
```

- **Output**

```
l1 = ( 40 20 10 30 10 )
There is an element greater than 10 in list l1 is 40.
```

16. algo_for_each

对于数据区间中的每一个数据都指向指定的操作。

```
void algo_for_each(
    input_iterator_t it_first,
    input_iterator_t it_last,
    unary_function_t ufun_op
);
```

- **Parameters**

  it_first: 数据区间的开始位置。
  it_last: 数据区间的末尾位置。
  ufun_op: 指定的操作规则。

- **Remarks**

这个算法扩展性很强，通过指定的操作函数可以指定不同的功能，但是当指定的操作函数要修改数据的内容时，关联容器不能作为目的数据区间。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_for_each.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 _mult_5(const void* cpv_input, void* pv_output)
{
    *(int*)cpv_input *= 5;
}

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    int i = 0;

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

    vector_init(pvec_v1);
```

```
    for(i = -4; i <= 2; ++i)
    {
        vector_push_back(pvec_v1, i);
    }

    /* Using for_each to print each elements of vector */
    printf("The original vector v1 = ( ");
    algo_for_each(vector_begin(pvec_v1), vector_end(pvec_v1), _print);
    printf(")\n");

    /* Using for_each to multiply each elements */
    algo_for_each(vector_begin(pvec_v1), vector_end(pvec_v1), _mult_5);

    /* Using for_each to print each elements of vector */
    printf("The modified vector v1 = ( ");
    algo_for_each(vector_begin(pvec_v1), vector_end(pvec_v1), _print);
    printf(")\n");

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The original vector v1 = ( -4 -3 -2 -1 0 1 2 )
The modified vector v1 = ( -20 -15 -10 -5 0 5 10 )
```

17. algo_generate

使用指定的函数产生的数据填充数据区间。

```
void algo_generate(
    forward_iterator_t it_first,
    forward_iterator_t it_last,
    unary_function_t ufun_op
);
```

- **Parameters**

  it_first: 数据区间的开始位置。
  it_last: 数据区间的末尾位置。
  ufun_op: 指定的生成数据的函数。

- **Remarks**

  这个算法使用数据生成函数产生数据并填充数据区间，关联容器不能作为目的数据区间。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <cstl/cvector.h>
#include <cstl/cdeque.h>
#include <cstl/calgorithm.h>

static void _random(const void* cpv_input, void* pv_output)
{
    srand((unsigned)time(NULL) + rand());
    *(int*)pv_output = rand();
}

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    deque_t* pdeq_q1 = create_deque(int);
    deque_iterator_t it_q;

    if(pvec_v1 == NULL || pdeq_q1 == NULL)
    {
        return -1;
    }

    vector_init_n(pvec_v1, 5);
    deque_init_n(pdeq_q1, 5);

    algo_generate(vector_begin(pvec_v1), vector_end(pvec_v1), _random);
    printf("Vector v1 is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    algo_generate(deque_begin(pdeq_q1), deque_end(pdeq_q1), _random);
    printf("Deque q1 is ( ");
    for(it_q = deque_begin(pdeq_q1);
        !iterator_equal(it_q, deque_end(pdeq_q1));
        it_q = iterator_next(it_q))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_q));
    }
    printf(")\n");

    vector_destroy(pvec_v1);
    deque_destroy(pdeq_q1);

    return 0;
}
```

- **Output**

```
Vector v1 is ( 317509817 323919728 1152247880 782498064 233499208 )
Deque q1 is ( 1377238285 538400351 1405670055 700144904 162907430 )
```

18. algo_generate_n

使用指定的函数生成的数据填充数据区间中的 n 个数据。

```
output_iterator_t algo_generate_n(
    output_iterator_t it_first,
    size_t t_count,
    unary_function_t ufun_op
);
```

- **Parameters**

it_first: 数据区间的开始位置。
t_count: 填充数据的个数。
ufun_op: 指定的生成数据的函数。

- **Remarks**

返回数据区间中被填充的数据的末尾。
必须保证数据区间至少包含 n 个数据，否则程序的行为是未定义的。
这个算法使用数据生成函数产生数据并填充数据区间，关联容器不能作为目的数据区间。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

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

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

static void _random(const void* cpv_input, void* pv_output)
{
    srand((unsigned)time(NULL) + rand());
    *(int*)pv_output = rand();
}

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    deque_t* pdeq_q1 = create_deque(int);
    deque_iterator_t it_q;

    if(pvec_v1 == NULL || pdeq_q1 == NULL)
    {
        return -1;
    }

    vector_init_n(pvec_v1, 5);
    deque_init_n(pdeq_q1, 5);
```

```
    algo_generate_n(vector_begin(pvec_v1), 5, _random);
    printf("Vector v1 is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    algo_generate_n(deque_begin(pdeq_q1), 3, _random);
    printf("Deque q1 is ( ");
    for(it_q = deque_begin(pdeq_q1);
        !iterator_equal(it_q, deque_end(pdeq_q1));
        it_q = iterator_next(it_q))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_q));
    }
    printf(")\n");

    vector_destroy(pvec_v1);
    deque_destroy(pdeq_q1);

    return 0;
}
```

- **Output**

```
Vector v1 is ( 259608207 1736347665 722267104 687102819 1704715307 )
Deque q1 is ( 1684494256 1801747599 165435036 0 0 )
```

19. algo_includes algo_includes_if

测试第一个有序的数据区间是否包含第二个有序数据区间的全部数据。

```
bool_t algo_includes(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    input_iterator_t it_first2,
    input_iterator_t it_last2
);

bool_t algo_includes_if(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    input_iterator_t it_first2,
    input_iterator_t it_last2,
    binary_function_t bfun_op
);
```

- **Parameters**

it_first1: 第一个数据区间的开始位置。
it_last1: 第一个数据区间的末尾位置。
it_first2: 第二个数据区间的开始位置。
it_last2: 第二个数据区间的末尾位置。
bfun_op: 指定的比较规则。

- **Remarks**

  如果第一个有序的数据区间包含第二个有序的数据区间的全部内容，返回 true，否则返回 false。第二个算法要保证两个数据区间都是按照指定的比较规则排序的。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

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

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

static void _mod_lesser(const void* cpv_first,
    const void* cpv_second, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t* pvec_v1a = create_vector(int);
    vector_t* pvec_v1b = create_vector(int);
    vector_t* pvec_v2a = create_vector(int);
    vector_t* pvec_v2b = create_vector(int);
    vector_t* pvec_v3a = create_vector(int);
    vector_t* pvec_v3b = create_vector(int);
    vector_iterator_t it_v;
    bool_t b_result;
    int i = 0;

    if(pvec_v1a == NULL || pvec_v1b == NULL ||
       pvec_v2a == NULL || pvec_v2b == NULL ||
       pvec_v3a == NULL || pvec_v3b == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1a);
    vector_init(pvec_v1b);
    vector_init(pvec_v2a);
    vector_init(pvec_v2b);
    vector_init(pvec_v3a);
    vector_init(pvec_v3b);

    /* Constructing vectors v1a and v1b with default less than ordering */
    for(i = -2; i <= 4; ++i)
    {
        vector_push_back(pvec_v1a, i);
    }
    for(i = -2; i <= 3; ++i)
    {
        vector_push_back(pvec_v1b, i);
    }
    printf("Original vector v1a with range sorted by the\n"
           "binary predicate less than is v1a = ( ");
```

```
    for(it_v = vector_begin(pvec_v1a);
        !iterator_equal(it_v, vector_end(pvec_v1a));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("Original vector v1b with range sorted by the\n"
           "binary predicate less than is v1b = ( ");
    for(it_v = vector_begin(pvec_v1b);
        !iterator_equal(it_v, vector_end(pvec_v1b));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Constructing vectors v2a and v2b with ranges sorted by greater */
    vector_assign(pvec_v2a, pvec_v1a);
    vector_assign(pvec_v2b, pvec_v1b);
    algo_sort_if(vector_begin(pvec_v2a), vector_end(pvec_v2a), fun_greater_int);
    algo_sort_if(vector_begin(pvec_v2b), vector_end(pvec_v2b), fun_greater_int);
    vector_pop_back(pvec_v2a);
    printf("Original vector v2a with range sorted by the\n"
           "binary predicate greater than is v2a = ( ");
    for(it_v = vector_begin(pvec_v2a);
        !iterator_equal(it_v, vector_end(pvec_v2a));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("Original vector v2b with range sorted by the\n"
           "binary predicate greater than is v2b = ( ");
    for(it_v = vector_begin(pvec_v2b);
        !iterator_equal(it_v, vector_end(pvec_v2b));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Constructing vectors v3a and v3b with ranges sorted by mod_lesser */
    vector_assign(pvec_v3a, pvec_v1a);
    vector_assign(pvec_v3b, pvec_v1b);
    algo_reverse(vector_begin(pvec_v3a), vector_end(pvec_v3a));
    vector_pop_back(pvec_v3a);
    vector_pop_back(pvec_v3a);
    algo_sort_if(vector_begin(pvec_v3a), vector_end(pvec_v3a), _mod_lesser);
    algo_sort_if(vector_begin(pvec_v3b), vector_end(pvec_v3b), _mod_lesser);
    printf("Original vector v3a with range sorted by the\n"
           "binary predicate mod_lesser is v3a = ( ");
    for(it_v = vector_begin(pvec_v3a);
        !iterator_equal(it_v, vector_end(pvec_v3a));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("Original vector v3b with range sorted by the\n"
```

```
        "binary predicate mod_lesser is v3b = ( ");
    for(it_v = vector_begin(pvec_v3b);
        !iterator_equal(it_v, vector_end(pvec_v3b));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /*
     * To test for inclusion under an assceding order
     * with the default binary predicate less
     */
    b_result = algo_includes(vector_begin(pvec_v1a), vector_end(pvec_v1a),
        vector_begin(pvec_v1b), vector_end(pvec_v1b));
    if(b_result)
    {
        printf("All the elements in vector v1b are contained in vector v1a.\n");
    }
    else
    {
        printf("At least one of the elements in vector v1b "
               "is not contained in vector v1a.\n");
    }

    /*
     * To test for inclusion under an assceding order
     * with the default binary predicate greater
     */
    b_result = algo_includes_if(vector_begin(pvec_v2a), vector_end(pvec_v2a),
        vector_begin(pvec_v2b), vector_end(pvec_v2b), fun_greater_int);
    if(b_result)
    {
        printf("All the elements in vector v2b are contained in vector v2a.\n");
    }
    else
    {
        printf("At least one of the elements in vector v2b "
               "is not contained in vector v2a.\n");
    }

    /*
     * To test for inclusion under an assceding order
     * with the default binary predicate mod_lesser
     */
    b_result = algo_includes_if(vector_begin(pvec_v3a), vector_end(pvec_v3a),
        vector_begin(pvec_v3b), vector_end(pvec_v3b), _mod_lesser);
    if(b_result)
    {
        printf("All the elements in vector v3b are contained in vector v3a.\n");
    }
    else
    {
        printf("At least one of the elements in vector v3b "
               "is not contained in vector v3a.\n");
    }

    vector_destroy(pvec_v1a);
    vector_destroy(pvec_v1b);
    vector_destroy(pvec_v2a);
```

```
    vector_destroy(pvec_v2b);
    vector_destroy(pvec_v3a);
    vector_destroy(pvec_v3b);

    return 0;
}

static void _mod_lesser(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;
}
```

- **Output**

```
Original vector v1a with range sorted by the
binary predicate less than is v1a = ( -2 -1 0 1 2 3 4 )
Original vector v1b with range sorted by the
binary predicate less than is v1b = ( -2 -1 0 1 2 3 )
Original vector v2a with range sorted by the
binary predicate greater than is v2a = ( 4 3 2 1 0 -1 )
Original vector v2b with range sorted by the
binary predicate greater than is v2b = ( 3 2 1 0 -1 -2 )
Original vector v3a with range sorted by the
binary predicate mod_lesser is v3a = ( 0 1 2 3 4 )
Original vector v3b with range sorted by the
binary predicate mod_lesser is v3b = ( 0 -1 1 -2 2 3 )
All the elements in vector v1b are contained in vector v1a.
At least one of the elements in vector v2b is not contained in vector v2a.
At least one of the elements in vector v3b is not contained in vector v3a.
```

20. algo_inplace_merge　algo_inplace_merge_if

将一个数据区间的两个连续的有序部分合并成一个有序的数据区间。

```
void algo_inplace_merge(
    bidirectional_iterator_t it_first,
    bidirectional_iterator_t it_middle,
    bidirectional_iterator_t it_last
);

void algo_inplace_merge_if(
    bidirectional_iterator_t it_first,
    bidirectional_iterator_t it_middle,
    bidirectional_iterator_t it_last,
    binary_function_t bfun_op
);
```

- **Parameters**

it_first: 数据区间第一个有序部分的开始位置。
it_middle: 数据区间第一个有序部分的末尾位置，第二个有序部分的开始位置。
it_last: 数据区间第二个有序部分的末尾位置。
bfun_op: 指定的比较规则。

- **Remarks**

第二个算法要保证数据区间的两部分都是按照指定的比较规则排序的。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

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

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

static void _mod_lesser(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[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_t* pvec_v3 = create_vector(int);
    vector_iterator_t it_v;
    vector_iterator_t it_break1;
    vector_iterator_t it_break2;
    vector_iterator_t it_break3;
    int i = 0;

    if(pvec_v1 == NULL || pvec_v2 == NULL || pvec_v3 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);
    vector_init(pvec_v3);

    for(i = 0; i <= 5; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
    for(i = -5; i <= 0; ++i)
    {
        vector_push_back(pvec_v1, i);
    }

    printf("Original vector v1 with subrangs sorted by the\n"
           "binary predicate less than is v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
```

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

    /* Constructing vector v2 with range sorted by greater */
    vector_assign(pvec_v2, pvec_v1);
    it_break2 = algo_find(vector_begin(pvec_v2), vector_end(pvec_v2), -5);
    algo_sort_if(vector_begin(pvec_v2), it_break2, fun_greater_int);
    algo_sort_if(it_break2, vector_end(pvec_v2), fun_greater_int);
    printf("Original vector v2 with subrangs sorted by the\n"
           "binary predicate greater than is v2 = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Constructing vector v3 with range sorted by greater */
    vector_assign(pvec_v3, pvec_v1);
    it_break3 = algo_find(vector_begin(pvec_v3), vector_end(pvec_v3), -5);
    algo_sort_if(vector_begin(pvec_v3), it_break3, _mod_lesser);
    algo_sort_if(it_break3, vector_end(pvec_v3), fun_greater_int);
    printf("Original vector v3 with subrangs sorted by the\n"
           "binary predicate greater than is v3 = ( ");
    for(it_v = vector_begin(pvec_v3);
        !iterator_equal(it_v, vector_end(pvec_v3));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    it_break1 = algo_find(vector_begin(pvec_v1), vector_end(pvec_v1), -5);
    algo_inplace_merge(vector_begin(pvec_v1), it_break1, vector_end(pvec_v1));
    printf("Merged inplace with default order,\nvector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* To merge inplace in desceding order, specify binary predicate greater */
    algo_inplace_merge_if(vector_begin(pvec_v2), it_break2,
        vector_end(pvec_v2), fun_greater_int);
    printf("Merged inplace with default order,\nvector v2 = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Applying a user defined binary predicate mod_lesser */
    algo_inplace_merge_if(vector_begin(pvec_v3), it_break3,
        vector_end(pvec_v3), _mod_lesser);
```

```
    printf("Merged inplace with default order,\nvector v3 = ( ");
    for(it_v = vector_begin(pvec_v3);
        !iterator_equal(it_v, vector_end(pvec_v3));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);
    vector_destroy(pvec_v3);

    return 0;
}
```

- **Output**

```
Original vector v1 with subrangs sorted by the
binary predicate less than is v1 = ( 0 1 2 3 4 5 -5 -4 -3 -2 -1 0 )
Original vector v2 with subrangs sorted by the
binary predicate greater than is v2 = ( 5 4 3 2 1 0 0 -1 -2 -3 -4 -5 )
Original vector v3 with subrangs sorted by the
binary predicate greater than is v3 = ( 0 1 2 3 4 5 0 -1 -2 -3 -4 -5 )
Merged inplace with default order,
vector v1 = ( -5 -4 -3 -2 -1 0 0 1 2 3 4 5 )
Merged inplace with default order,
vector v2 = ( 5 4 3 2 1 0 0 -1 -2 -3 -4 -5 )
Merged inplace with default order,
vector v3 = ( 0 0 1 -1 2 -2 3 -3 4 -4 5 -5 )
```

21. algo_is_heap　algo_is_heap_if

测试一个数据区间是否是符合指定比较规则的堆。

```
bool_t algo_is_heap(
    random_access_iterator_t it_first,
    random_access_iterator_t it_last
);

bool_t algo_is_heap_if(
    random_access_iterator_t it_first,
    random_access_iterator_t it_last,
    binary_function_t bfun_op
);
```

- **Parameters**

it_first:　数据区间的开始位置。
it_last:　数据区间的末尾位置。
bfun_op:　指定的比较规则。

- **Remarks**

如果数据区间是堆，返回 true，否则返回 false。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_is_heap.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_v1 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

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

    vector_init(pvec_v1);

    for(i = 1; i <= 9; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
    algo_random_shuffle(vector_begin(pvec_v1), vector_end(pvec_v1));

    printf("Vector v1 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    if(algo_is_heap(vector_begin(pvec_v1), vector_end(pvec_v1)))
    {
        printf("The range is less-than heap.\n");
    }
    else
    {
        printf("The range is not less-than heap.\n");
    }
    if(algo_is_heap_if(vector_begin(pvec_v1), vector_end(pvec_v1), fun_greater_int))
    {
        printf("The range is greater-than heap.\n");
    }
    else
    {
        printf("The range is not greater-than heap.\n");
    }

    /* Make v1 a heap with default less than ordering */
    algo_make_heap(vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("The heap version of vector v1 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
```

```
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    if(algo_is_heap(vector_begin(pvec_v1), vector_end(pvec_v1)))
    {
        printf("The range is less-than heap.\n");
    }
    else
    {
        printf("The range is not less-than heap.\n");
    }
    if(algo_is_heap_if(vector_begin(pvec_v1), vector_end(pvec_v1), fun_greater_int))
    {
        printf("The range is greater-than heap.\n");
    }
    else
    {
        printf("The range is not greater-than heap.\n");
    }

    /* Make v1 a heap with greater than ordering */
    algo_make_heap_if(vector_begin(pvec_v1), vector_end(pvec_v1), fun_greater_int);
    printf("The greater-than heap version of vector v1 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    if(algo_is_heap(vector_begin(pvec_v1), vector_end(pvec_v1)))
    {
        printf("The range is less-than heap.\n");
    }
    else
    {
        printf("The range is not less-than heap.\n");
    }
    if(algo_is_heap_if(vector_begin(pvec_v1), vector_end(pvec_v1), fun_greater_int))
    {
        printf("The range is greater-than heap.\n");
    }
    else
    {
        printf("The range is not greater-than heap.\n");
    }

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
Vector v1 is:
( 9 8 5 1 7 3 2 6 4 )
The range is not less-than heap.
```

```
The range is not greater-than heap.
The heap version of vector v1 is:
( 9 8 5 6 7 3 2 1 4 )
The range is less-than heap.
The range is not greater-than heap.
The greater-than heap version of vector v1 is:
( 1 4 2 6 7 3 5 9 8 )
The range is not less-than heap.
The range is greater-than heap.
```

22. algo_is_sorted algo_is_sorted_if

测试数据区间是否按照指定的比较规则排序。

```
bool_t algo_is_sorted(
    forward_iterator_t it_first,
    forward_iterator_t it_last
);

bool_t algo_is_sorted_if(
    forward_iterator_t it_first,
    forward_iterator_t it_last,
    binary_function_t bfun_op
);
```

- **Parameters**

  it_first: 数据区间的开始位置。
  it_last: 数据区间的末尾位置。
  bfun_op: 指定的比较规则。

- **Remarks**

  如果数据区间是按照指定规则排序的，返回 true，否则返回 false。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_is_sort.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_v1 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

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

```
    }

    vector_init(pvec_v1);

    for(i = 0; i <= 5; ++i)
    {
        vector_push_back(pvec_v1, i * 2);
    }
    for(i = 0; i <= 5; ++i)
    {
        vector_push_back(pvec_v1, i * 2 + 1);
    }

    printf("Original vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    if(algo_is_sorted(vector_begin(pvec_v1), vector_end(pvec_v1)))
    {
        printf("Vector is sorted with predicate less than.\n");
    }
    else
    {
        printf("Vector is not sorted with predicate less than.\n");
    }
    if(algo_is_sorted_if(vector_begin(pvec_v1),
        vector_end(pvec_v1), fun_greater_int))
    {
        printf("Vector is sorted with predicate greater than.\n");
    }
    else
    {
        printf("Vector is not sorted with predicate greater than.\n");
    }

    algo_sort(vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("Sorted vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    if(algo_is_sorted(vector_begin(pvec_v1), vector_end(pvec_v1)))
    {
        printf("Vector is sorted with predicate less than.\n");
    }
    else
    {
        printf("Vector is not sorted with predicate less than.\n");
    }
    if(algo_is_sorted_if(vector_begin(pvec_v1),
        vector_end(pvec_v1), fun_greater_int))
    {
        printf("Vector is sorted with predicate greater than.\n");
```

```
    }
    else
    {
        printf("Vector is not sorted with predicate greater than.\n");
    }

    /* To sort in descending order. */
    algo_sort_if(vector_begin(pvec_v1), vector_end(pvec_v1), fun_greater_int);
    printf("Resorted (greater) vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    if(algo_is_sorted(vector_begin(pvec_v1), vector_end(pvec_v1)))
    {
        printf("Vector is sorted with predicate less than.\n");
    }
    else
    {
        printf("Vector is not sorted with predicate less than.\n");
    }
    if(algo_is_sorted_if(vector_begin(pvec_v1),
        vector_end(pvec_v1), fun_greater_int))
    {
        printf("Vector is sorted with predicate greater than.\n");
    }
    else
    {
        printf("Vector is not sorted with predicate greater than.\n");
    }

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
Original vector v1 = ( 0 2 4 6 8 10 1 3 5 7 9 11 )
Vector is not sorted with predicate less than.
Vector is not sorted with predicate greater than.
Sorted vector v1 = ( 0 1 2 3 4 5 6 7 8 9 10 11 )
Vector is sorted with predicate less than.
Vector is not sorted with predicate greater than.
Resorted (greater) vector v1 = ( 11 10 9 8 7 6 5 4 3 2 1 0 )
Vector is not sorted with predicate less than.
Vector is sorted with predicate greater than.
```

23. algo_iter_swap

交换两个迭代器指向的数据的内容。

```
void algo_iter_swap(
    forward_iterator_t it_first,
    forward_iterator_t it_second
);
```

- **Parameters**

  it_first: 第一个数据的迭代器。

  it_second: 第二个数据的迭代器。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

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

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

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    deque_t* pdq_dq1 = create_deque(int);
    deque_t* pdq_dq2 = create_deque(int);
    deque_iterator_t it_dq;
    int i = 0;

    if(pvec_v1 == NULL || pdq_dq1 == NULL || pdq_dq2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    deque_init(pdq_dq1);
    deque_init(pdq_dq2);

    deque_push_back(pdq_dq1, 5);
    deque_push_back(pdq_dq1, 1);
    deque_push_back(pdq_dq1, 10);

    printf("The original deque dq1 = ( ");
    for(it_dq = deque_begin(pdq_dq1);
        !iterator_equal(it_dq, deque_end(pdq_dq1));
        it_dq = iterator_next(it_dq))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_dq));
    }
    printf(")\n");

    /* Exchanging first and last elements with iter_swap */
    algo_iter_swap(deque_begin(pdq_dq1), iterator_prev(deque_end(pdq_dq1)));

    printf("The deque dq1 with first and last element swap is = ( ");
    for(it_dq = deque_begin(pdq_dq1);
        !iterator_equal(it_dq, deque_end(pdq_dq1));
        it_dq = iterator_next(it_dq))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_dq));
```

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

    /* Swapping the second and last elements with swap */
    algo_swap(iterator_prev(deque_end(pdq_dq1)),
        iterator_next(deque_begin(pdq_dq1)));

    printf("The deque dq1 with the second and last element swap is = ( ");
    for(it_dq = deque_begin(pdq_dq1);
        !iterator_equal(it_dq, deque_end(pdq_dq1));
        it_dq = iterator_next(it_dq))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_dq));
    }
    printf(")\n");

    /* Swapping a vector element with a deque element */
    for(i = 0; i < 4; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
    for(i = 4; i < 6; ++i)
    {
        deque_push_back(pdq_dq2, i);
    }

    printf("Vector v1 is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("Deque dq2 is ( ");
    for(it_dq = deque_begin(pdq_dq2);
        !iterator_equal(it_dq, deque_end(pdq_dq2));
        it_dq = iterator_next(it_dq))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_dq));
    }
    printf(")\n");

    algo_iter_swap(vector_begin(pvec_v1), deque_begin(pdq_dq2));

    printf("After exchanging first elements:\n");
    printf("Vector v1 is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("Deque dq2 is ( ");
    for(it_dq = deque_begin(pdq_dq2);
        !iterator_equal(it_dq, deque_end(pdq_dq2));
        it_dq = iterator_next(it_dq))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_dq));
```

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

    vector_destroy(pvec_v1);
    deque_destroy(pdq_dq1);
    deque_destroy(pdq_dq2);

    return 0;
}
```

- **Output**

```
The original deque dq1 = ( 5 1 10 )
The deque dq1 with first and last element swap is = ( 10 1 5 )
The deque dq1 with the second and last element swap is = ( 10 5 1 )
Vector v1 is ( 0 1 2 3 )
Deque dq2 is ( 4 5 )
After exchanging first elements:
Vector v1 is ( 4 1 2 3 )
Deque dq2 is ( 0 5 )
```

24. algo_lexicographical_compare algo_lexicographical_compare_if

将两个数据区间中的数据逐个对比。

```
bool_t algo_lexicographical_compare(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    input_iterator_t it_first2,
    input_iterator_t it_last2
);

bool_t algo_lexicographical_compare_if(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    input_iterator_t it_first2,
    input_iterator_t it_last2,
    binary_function_t bfun_op
);
```

- **Parameters**

it_first1: 第一个数据区间的开始位置。
it_last1: 第一个数据区间的末尾位置。
it_first2: 第二个数据区间的开始位置。
it_last2: 第二个数据区间的末尾位置。
bfun_op: 指定的比较规则。

- **Remarks**

如果第一个数据区间中的数据小于第二个数据区间中的对应数据，那么返回 true，如果大于返回 false。

如果第一个数据区间中的数据和第二个数据区间中的数据对应相等，第一个数据区间数据的个数小于第二个数据区间的个数返回 true，否则返回 false。

- **Requirements**

头文件 <cstl/calgorithm.h>。

● **Example**

```
/*
 * algo_lexicographical_compare.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/clist.h>
#include <cstl/calgorithm.h>

static void _twice(const void* cpv_first, const void* cpv_second, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_iterator_t it_v;
    list_t* plist_l1 = create_list(int);
    list_iterator_t it_l;
    int i = 0;

    if(pvec_v1 == NULL || pvec_v2 == NULL || plist_l1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);
    list_init(plist_l1);

    for(i = 0; i < 6; ++i)
    {
        vector_push_back(pvec_v1, i * 5);
        vector_push_back(pvec_v2, i * 10);
    }
    for(i = 0; i < 7; ++i)
    {
        list_push_back(plist_l1, i * 5);
    }

    printf("Vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("Vector v2 = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("List l1 = ( ");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
```

```
        it_l = iterator_next(it_l))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_l));
    }
    printf(")\n");

    /* Self lexicographical_comparison of v1 under identity */
    if(algo_lexicographical_compare(vector_begin(pvec_v1), vector_end(pvec_v1),
        vector_begin(pvec_v1), vector_end(pvec_v1)))
    {
        printf("Vector v1 is lexicographically less than v1.\n");
    }
    else
    {
        printf("Vector v1 is not lexicographically less than v1.\n");
    }

    /* lexicographical_comparison of v1 and l1 under identity */
    if(algo_lexicographical_compare(vector_begin(pvec_v1), vector_end(pvec_v1),
        list_begin(plist_l1), list_end(plist_l1)))
    {
        printf("Vector v1 is lexicographically less than l1.\n");
    }
    else
    {
        printf("Vector v1 is not lexicographically less than l1.\n");
    }

    if(algo_lexicographical_compare_if(vector_begin(pvec_v1), vector_end(pvec_v1),
        vector_begin(pvec_v2), vector_end(pvec_v2), _twice))
    {
        printf("Vector v1 is lexicographically less than v2 under twice.\n");
    }
    else
    {
        printf("Vector v1 is not lexicographically less than v2 under twice.\n");
    }

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);
    list_destroy(plist_l1);

    return 0;
}

static void _twice(const void* cpv_first, const void* cpv_second, void* pv_output)
{
    *(bool_t*)pv_output = *(int*)cpv_first * 2 < *(int*)cpv_second ? true : false;
}
```

- **Output**

```
Vector v1 = ( 0 5 10 15 20 25 )
Vector v2 = ( 0 10 20 30 40 50 )
List l1 = ( 0 5 10 15 20 25 30 )
Vector v1 is not lexicographically less than v1.
Vector v1 is lexicographically less than l1.
Vector v1 is not lexicographically less than v2 under twice.
```

25. algo_lexicographical_compare_3way　algo_lexicographical_compare_3way_if

将两个数据区间中的数据进行逐个比较，返回三种结果。

```
int algo_lexicographical_compare_3way(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    input_iterator_t it_first2,
    input_iterator_t t_last2
);

int algo_lexicographical_compare_3way_if(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    input_iterator_t it_first2,
    input_iterator_t it_last2,
    binary_function_t bfun_op
);
```

- **Parameters**

it_first1:　第一个数据区间的开始位置。
it_last1:　第一个数据区间的末尾位置。
it_first2:　第二个数据区间的开始位置。
it_last2:　第二个数据区间的末尾位置。
bfun_op:　指定的比较规则。

- **Remarks**

如果第一个数据区间中的数据小于第二个数据区间中的对应数据，那么返回值小于 0，如果大于返回值大于 0。

如果第一个数据区间中的数据和第二个数据区间中的数据对应相等，第一个数据区间数据的个数小于第二个数据区间的个数返回值小于 0，数据个数相等返回值等于 0，否则返回值大于 0。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_lexicographical_compare_3way.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/clist.h>
#include <cstl/calgorithm.h>

static void _twice(const void* cpv_first, const void* cpv_second, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_iterator_t it_v;
    list_t* plist_l1 = create_list(int);
    list_iterator_t it_l;
    int n_result = 0;
```

```
    int i = 0;

    if(pvec_v1 == NULL || pvec_v2 == NULL || plist_l1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);
    list_init(plist_l1);

    for(i = 0; i < 6; ++i)
    {
        vector_push_back(pvec_v1, i * 5);
        vector_push_back(pvec_v2, i * 10);
    }
    for(i = 0; i < 7; ++i)
    {
        list_push_back(plist_l1, i * 5);
    }

    printf("Vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("Vector v2 = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("List l1 = ( ");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_l));
    }
    printf(")\n");

    /* Self lexicographical_comparison of v1 under identity */
    n_result = algo_lexicographical_compare_3way(vector_begin(pvec_v1),
        vector_end(pvec_v1), vector_begin(pvec_v1), vector_end(pvec_v1));
    if(n_result < 0)
    {
        printf("Vector v1 is lexicographically less than v1.\n");
    }
    else if(n_result == 0)
    {
        printf("Vector v1 is lexicographically equal to v1.\n");
    }
    else
    {
        printf("Vector v1 is lexicographically greater than v1.\n");
```

```
    }

    /* lexicographical_comparison of v1 and l1 under identity */
    n_result = algo_lexicographical_compare_3way(vector_begin(pvec_v1),
        vector_end(pvec_v1), list_begin(plist_l1), list_end(plist_l1));
    if(n_result < 0)
    {
        printf("Vector v1 is lexicographically less than l1.\n");
    }
    else if(n_result == 0)
    {
        printf("Vector v1 is lexicographically equal to l1.\n");
    }
    else
    {
        printf("Vector v1 is lexicographically greater than l1.\n");
    }

    n_result = algo_lexicographical_compare_if(vector_begin(pvec_v1),
        vector_end(pvec_v1), vector_begin(pvec_v2), vector_end(pvec_v2), _twice);
    if(n_result < 0)
    {
        printf("Vector v1 is lexicographically less than v2 under twice.\n");
    }
    else if(n_result == 0)
    {
        printf("Vector v1 is lexicographically equal to v2 under twice.\n");
    }
    else
    {
        printf("Vector v1 is lexicographically greater than v2 under twice.\n");
    }

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);
    list_destroy(plist_l1);

    return 0;
}

static void _twice(const void* cpv_first, const void* cpv_second, void* pv_output)
{
    *(bool_t*)pv_output = *(int*)cpv_first * 2 < *(int*)cpv_second ? true : false;
}
```

- **Output**

```
Vector v1 = ( 0 5 10 15 20 25 )
Vector v2 = ( 0 10 20 30 40 50 )
List l1 = ( 0 5 10 15 20 25 30 )
Vector v1 is not lexicographically less than v1.
Vector v1 is lexicographically less than l1.
Vector v1 is not lexicographically less than v2 under twice.
```

26. algo_lower_bound   algo_lower_bound_if

在有序的数据区间中查找第一个等于指定数据的位置。

```
forward_iterator_t algo_lower_bound(
```

```
    forward_iterator_t it_first,
    forward_iterator_t it_last,
    element
);

forward_iterator_t algo_lower_bound_if(
    forward_iterator_t it_first,
    forward_iterator_t it_last,
    element,
    binary_function_t bfun_op
);
```

- **Parameters**

  it_first: 数据区间的开始位置。
  it_last: 数据区间的末尾位置。
  element: 指定的数据。
  bfun_op: 指定的比较规则。

- **Remarks**

  算法返回按照指定规则比较规则排序的数据区间中等于指定数据的第一个位置，如果数据区间中不包含指定数据就返回第一个大于指定数据的位置。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_lower_bound.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>
#include <cstl/cfunctional.h>

/* Return whether modulus of elem1 is less than modulus of elem2 */
static void _mod_lesser(const void* cpv_first,
    const void* cpv_second, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_t* pvec_v3 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

    if(pvec_v1 == NULL || pvec_v2 == NULL || pvec_v3 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);
    vector_init(pvec_v3);
```

```
for(i = -1; i <= 4; ++i)
{
    vector_push_back(pvec_v1, i);
}
for(i = -3; i <= 0; ++i)
{
    vector_push_back(pvec_v1, i);
}

algo_sort(vector_begin(pvec_v1), vector_end(pvec_v1));
printf("Original vector v1 with range sorted by the "
       "binary predicate less than is v1 = ( ");
for(it_v = vector_begin(pvec_v1);
    !iterator_equal(it_v, vector_end(pvec_v1));
    it_v = iterator_next(it_v))
{
    printf("%d ", *(int*)iterator_get_pointer(it_v));
}
printf(")\n");

vector_assign(pvec_v2, pvec_v1);
algo_sort_if(vector_begin(pvec_v2), vector_end(pvec_v2), fun_greater_int);
printf("Original vector v2 with range sorted by the "
       "binary predicate greater than is v2 = ( ");
for(it_v = vector_begin(pvec_v2);
    !iterator_equal(it_v, vector_end(pvec_v2));
    it_v = iterator_next(it_v))
{
    printf("%d ", *(int*)iterator_get_pointer(it_v));
}
printf(")\n");

vector_assign(pvec_v3, pvec_v1);
algo_sort_if(vector_begin(pvec_v3), vector_end(pvec_v3), _mod_lesser);
printf("Original vector v3 with range sorted by the "
       "binary predicate greater than is v3 = ( ");
for(it_v = vector_begin(pvec_v3);
    !iterator_equal(it_v, vector_end(pvec_v3));
    it_v = iterator_next(it_v))
{
    printf("%d ", *(int*)iterator_get_pointer(it_v));
}
printf(")\n");

/* lower_bound of 3 in v1 with default binary predicate less than */
it_v = algo_lower_bound(vector_begin(pvec_v1), vector_end(pvec_v1), 3);
printf("The lower_bound in v1 for the element with a value of 3 is: %d.\n",
    *(int*)iterator_get_pointer(it_v));

/* lower_bound of 3 in v2 with the binary predicate greater than */
it_v = algo_lower_bound_if(vector_begin(pvec_v2),
    vector_end(pvec_v2), 3, fun_greater_int);
printf("The lower_bound in v2 for the element with a value of 3 is: %d.\n",
    *(int*)iterator_get_pointer(it_v));

/* lower_bound of 3 in v3 with the binary predicate _mod_lesser */
it_v = algo_lower_bound_if(vector_begin(pvec_v3),
    vector_end(pvec_v3), 3, _mod_lesser);
printf("The lower_bound in v3 for the element with a value of 3 is: %d.\n",
```

```
        *(int*)iterator_get_pointer(it_v));

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);
    vector_destroy(pvec_v3);

    return 0;
}

static void _mod_lesser(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;
}
```

- **Output**

```
Original vector v1 with range sorted by the binary predicate less than is v1 = ( -3
-2 -1 -1 0 0 1 2 3 4 )
Original vector v2 with range sorted by the binary predicate greater than is v2 =
( 4 3 2 1 0 0 -1 -1 -2 -3 )
Original vector v3 with range sorted by the binary predicate greater than is v3 =
( 0 0 -1 -1 1 -2 2 -3 3 4 )
The lower_bound in v1 for the element with a value of 3 is: 3.
The lower_bound in v2 for the element with a value of 3 is: 3.
The lower_bound in v3 for the element with a value of 3 is: -3.
```

27. algo_make_heap algo_make_heap_if

将数据区间转换成符合指定比较规则的堆。

```
void algo_make_heap(
    random_access_iterator_t it_first,
    random_access_iterator_t it_last
);

void algo_make_heap_if(
    random_access_iterator_t it_first,
    random_access_iterator_t it_last,
    binary_function_t bfun_op
);
```

- **Parameters**

it_first: 数据区间的开始位置。
it_last: 数据区间的末尾位置。
bfun_op: 指定的比较规则。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_make_heap.c
 * compile with : -lcstl
 */
```

```c
#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>
#include <cstl/cfunctional.h>

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

    if(pvec_v1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);

    for(i = 1; i <= 9; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
    algo_random_shuffle(vector_begin(pvec_v1), vector_end(pvec_v1));

    printf("Vector v1 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Make v1 a heap with default less than ordering */
    algo_make_heap(vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("The heap version of vector v1 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Make v1 a heap with greater than ordering */
    algo_make_heap_if(vector_begin(pvec_v1), vector_end(pvec_v1), fun_greater_int);
    printf("The greater-than heap version of vector v1 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
Vector v1 is:
( 6 3 2 1 7 8 4 9 5 )
The heap version of vector v1 is:
( 9 7 8 5 6 2 4 1 3 )
The greater-than heap version of vector v1 is:
( 1 3 2 5 6 8 4 9 7 )
```

28. algo_max algo_max_if

按照指定比较规则比较两个迭代器所指的数据，返回数据较大迭代器。

```
input_iterator_t algo_max(
    input_iterator_t it_first,
    input_iterator_t it_second
);

input_iterator_t algo_max_if(
    input_iterator_t it_first,
    input_iterator_t it_last,
    binary_function_t bfun_op
);
```

- **Parameters**

  it_first: 第一个数据的迭代器。
  it_second: 第二个数据的迭代器。
  bfun_op: 指定的比较规则。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_max.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/cset.h>
#include <cstl/calgorithm.h>
#include <cstl/cfunctional.h>

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(set_t<int>);
    vector_iterator_t it_v;
    set_t* pset_s1 = create_set(int);
    set_iterator_t it_s;
    int i = 0;

    if(pvec_v1 == NULL || pset_s1 == NULL)
    {
        return -1;
    }
```

```
vector_init(pvec_v1);
set_init(pset_s1);

set_insert(pset_s1, 0);
set_insert(pset_s1, 1);
set_insert(pset_s1, 2);
vector_push_back(pvec_v1, pset_s1);
vector_push_back(pvec_v1, pset_s1);
set_clear(pset_s1);
set_insert(pset_s1, 0);
set_insert(pset_s1, 2);
set_insert(pset_s1, 4);
vector_push_back(pvec_v1, pset_s1);

for(it_v = vector_begin(pvec_v1), i = 1;
    !iterator_equal(it_v, vector_end(pvec_v1));
    it_v = iterator_next(it_v), ++i)
{
    printf("Set s%d ( ", i);
    for(it_s = set_begin((set_t*)iterator_get_pointer(it_v));
        !iterator_equal(it_s, set_end((set_t*)iterator_get_pointer(it_v)));
        it_s = iterator_next(it_s))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_s));
    }
    printf(")\n");
}

it_v = algo_max(vector_begin(pvec_v1), iterator_next(vector_begin(pvec_v1)));
printf("The max set between the frist and the second is ( ");
for(it_s = set_begin((set_t*)iterator_get_pointer(it_v));
    !iterator_equal(it_s, set_end((set_t*)iterator_get_pointer(it_v)));
    it_s = iterator_next(it_s))
{
    printf("%d ", *(int*)iterator_get_pointer(it_s));
}
printf(")\n");

it_v = algo_max(vector_begin(pvec_v1), iterator_prev(vector_end(pvec_v1)));
printf("The max set between the frist and the third is ( ");
for(it_s = set_begin((set_t*)iterator_get_pointer(it_v));
    !iterator_equal(it_s, set_end((set_t*)iterator_get_pointer(it_v)));
    it_s = iterator_next(it_s))
{
    printf("%d ", *(int*)iterator_get_pointer(it_s));
}
printf(")\n");

it_v = algo_max_if(vector_begin(pvec_v1),
    iterator_next(vector_begin(pvec_v1)), fun_greater_set);
printf("The max set between the frist and the second under greater than is ( ");
for(it_s = set_begin((set_t*)iterator_get_pointer(it_v));
    !iterator_equal(it_s, set_end((set_t*)iterator_get_pointer(it_v)));
    it_s = iterator_next(it_s))
{
    printf("%d ", *(int*)iterator_get_pointer(it_s));
}
printf(")\n");
```

```
    it_v = algo_max_if(vector_begin(pvec_v1),
        iterator_prev(vector_end(pvec_v1)), fun_greater_set);
    printf("The max set between the frist and the third under greater than is ( ");
    for(it_s = set_begin((set_t*)iterator_get_pointer(it_v));
        !iterator_equal(it_s, set_end((set_t*)iterator_get_pointer(it_v)));
        it_s = iterator_next(it_s))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_s));
    }
    printf(")\n");

    vector_destroy(pvec_v1);
    set_destroy(pset_s1);

    return 0;
}
```

- **Output**

```
Set s1 ( 0 1 2 )
Set s2 ( 0 1 2 )
Set s3 ( 0 2 4 )
The max set between the frist and the second is ( 0 1 2 )
The max set between the frist and the third is ( 0 2 4 )
The max set between the frist and the second under greater than is ( 0 1 2 )
The max set between the frist and the third under greater than is ( 0 1 2 )
```

29. algo_max_element algo_max_element_if

返回数据区间中最大数据的迭代器。

```
forward_iterator_t algo_max_element(
    forward_iterator_t it_first,
    forward_iterator_t it_last
);

forward_iterator_t algo_max_element_if(
    forward_iterator_t it_first,
    forward_iterator_t it_last,
    binary_function_t bfun_op
);
```

- **Parameters**

  it_first: 数据区间的开始位置。
  it_last: 数据区间的末尾位置。
  bfun_op: 指定的比较规则。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_max_element.c
 * compile with : -lcstl
 */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

static void _mod_lesser(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[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

    if(pvec_v1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);

    for(i = 0; i <= 3; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
    for(i = 1; i <= 4; ++i)
    {
        vector_push_back(pvec_v1, i * -2);
    }

    printf("Vector v1 is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    it_v = algo_max_element(vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("The largest element in v1 is : %d.\n",
        *(int*)iterator_get_pointer(it_v));

    it_v = algo_max_element_if(vector_begin(pvec_v1),
        vector_end(pvec_v1), _mod_lesser);
    printf("The largest element in v1 under the mod_lesser is : %d.\n",
        *(int*)iterator_get_pointer(it_v));

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
Vector v1 is ( 0 1 2 3 -2 -4 -6 -8 )
```

```
The largest element in v1 is : 3.
The largest element in v1 under the mod_lesser is : -8.
```

30. algo_merge algo_merge_if

将两个有序数据区间合并到目的数据区间中，合并后的数据区间仍然有序。

```
output_iterator_t algo_merge(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    input_iterator_t it_first2,
    input_iterator_t it_last2,
    output_iterator_t it_result
);

output_iterator_t algo_merge_if(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    input_iterator_t it_first2,
    input_iterator_t it_last2,
    output_iterator_t it_result,
    binary_function_t bfun_op
);
```

- **Parameters**

it_first1: 第一个数据区间的开始位置。
it_last1: 第一个数据区间的末尾位置。
it_first2: 第二个数据区间的开始位置。
it_last2: 第二个数据区间的末尾位置。
it_result: 目的数据区间的开始位置。
bfun_op: 指定的比较规则。

- **Remarks**

返回目的数据区间中合并后的有序数据区间的末尾。

两个有序的源数据区间必须是按照相同的比较规则排序的，同事目的数据区间要足够大，至少是两个源数据区间中数据的和。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_merge.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>
#include <cstl/cfunctional.h>

static void _mod_lesser(const void* cpv_first,
    const void* cpv_second, void* pv_output);

int main(int argc, char* argv[])
```

```
{
    vector_t* pvec_v1a = create_vector(int);
    vector_t* pvec_v1b = create_vector(int);
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2a = create_vector(int);
    vector_t* pvec_v2b = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_t* pvec_v3a = create_vector(int);
    vector_t* pvec_v3b = create_vector(int);
    vector_t* pvec_v3 = create_vector(int);
    vector_iterator_t it_v;
    vector_iterator_t it_result;
    int i = 0;

    if(pvec_v1a == NULL || pvec_v1b == NULL || pvec_v1 == NULL ||
       pvec_v2a == NULL || pvec_v2b == NULL || pvec_v2 == NULL ||
       pvec_v3a == NULL || pvec_v3b == NULL || pvec_v3 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1a);
    vector_init(pvec_v1b);
    vector_init(pvec_v1);
    vector_init(pvec_v2a);
    vector_init(pvec_v2b);
    vector_init(pvec_v2);
    vector_init(pvec_v3a);
    vector_init(pvec_v3b);
    vector_init(pvec_v3);

    /* Constructing vectors v1a and v1b with default less than ordering */
    for(i = 0; i <= 5; ++i)
    {
        vector_push_back(pvec_v1a, i);
    }
    for(i = -5; i <= 0; ++i)
    {
        vector_push_back(pvec_v1b, i);
    }
    vector_resize(pvec_v1, 12);
    printf("Original vector v1a with range sorted by the\n"
           "binary predicate less than is v1a = ( ");
    for(it_v = vector_begin(pvec_v1a);
        !iterator_equal(it_v, vector_end(pvec_v1a));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("Original vector v1b with range sorted by the\n"
           "binary predicate less than is v1b = ( ");
    for(it_v = vector_begin(pvec_v1b);
        !iterator_equal(it_v, vector_end(pvec_v1b));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
```

```
    /* Constructing vectors v2a and v2b with ranges sorted by greater */
    vector_assign(pvec_v2a, pvec_v1a);
    vector_assign(pvec_v2b, pvec_v1b);
    vector_assign(pvec_v2, pvec_v1);
    algo_sort_if(vector_begin(pvec_v2a), vector_end(pvec_v2a), fun_greater_int);
    algo_sort_if(vector_begin(pvec_v2b), vector_end(pvec_v2b), fun_greater_int);
    printf("Original vector v2a with range sorted by the\n"
           "binary predicate greater than is v2a = ( ");
    for(it_v = vector_begin(pvec_v2a);
        !iterator_equal(it_v, vector_end(pvec_v2a));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("Original vector v2b with range sorted by the\n"
           "binary predicate greater than is v2b = ( ");
    for(it_v = vector_begin(pvec_v2b);
        !iterator_equal(it_v, vector_end(pvec_v2b));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Constructing vectors v3a and v3b with ranges sorted by mod_lesser */
    vector_assign(pvec_v3a, pvec_v1a);
    vector_assign(pvec_v3b, pvec_v1b);
    vector_assign(pvec_v3, pvec_v1);
    algo_sort_if(vector_begin(pvec_v3a), vector_end(pvec_v3a), _mod_lesser);
    algo_sort_if(vector_begin(pvec_v3b), vector_end(pvec_v3b), _mod_lesser);
    printf("Original vector v3a with range sorted by the\n"
           "binary predicate mod_lesser is v3a = ( ");
    for(it_v = vector_begin(pvec_v3a);
        !iterator_equal(it_v, vector_end(pvec_v3a));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("Original vector v3b with range sorted by the\n"
           "binary predicate greater than is v3b = ( ");
    for(it_v = vector_begin(pvec_v3b);
        !iterator_equal(it_v, vector_end(pvec_v3b));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* To merge inplace in ascending order with the default binary predicate less */
    it_result = algo_merge(vector_begin(pvec_v1a), vector_end(pvec_v1a),
        vector_begin(pvec_v1b), vector_end(pvec_v1b), vector_begin(pvec_v1));
    printf("Merged inplac with default order\n"
           "vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, it_result);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
```

```
    }
    printf(")\n");

    /* To merge inplace in ascending order with the specify binary predicate greater
*/
    it_result = algo_merge_if(vector_begin(pvec_v2a), vector_end(pvec_v2a),
        vector_begin(pvec_v2b), vector_end(pvec_v2b), vector_begin(pvec_v2),
        fun_greater_int);
    printf("Merged inplace with binary predicate greater order\n"
           "vector v2 = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, it_result);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* To merge inplace in ascending order with the user_defined binary predicate
mod_lesser */
    it_result = algo_merge_if(vector_begin(pvec_v3a), vector_end(pvec_v3a),
        vector_begin(pvec_v3b), vector_end(pvec_v3b), vector_begin(pvec_v3),
        _mod_lesser);
    printf("Merged inplace with binary predicate mod_lesser order\n"
           "vector v3 = ( ");
    for(it_v = vector_begin(pvec_v3);
        !iterator_equal(it_v, it_result);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1a);
    vector_destroy(pvec_v1b);
    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2a);
    vector_destroy(pvec_v2b);
    vector_destroy(pvec_v2);
    vector_destroy(pvec_v3a);
    vector_destroy(pvec_v3b);
    vector_destroy(pvec_v3);

    return 0;
}

static void _mod_lesser(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;
}
```

- **Output**

```
Original vector v1a with range sorted by the
binary predicate less than is v1a = ( 0 1 2 3 4 5 )
Original vector v1b with range sorted by the
binary predicate less than is v1b = ( -5 -4 -3 -2 -1 0 )
```

```
Original vector v2a with range sorted by the
binary predicate greater than is v2a = ( 5 4 3 2 1 0 )
Original vector v2b with range sorted by the
binary predicate greater than is v2b = ( 0 -1 -2 -3 -4 -5 )
Original vector v3a with range sorted by the
binary predicate mod_lesser is v3a = ( 0 1 2 3 4 5 )
Original vector v3b with range sorted by the
binary predicate greater than is v3b = ( 0 -1 -2 -3 -4 -5 )
Merged inplac with default order
vector v1 = ( -5 -4 -3 -2 -1 0 0 1 2 3 4 5 )
Merged inplace with binary predicate greater order
vector v2 = ( 5 4 3 2 1 0 0 -1 -2 -3 -4 -5 )
Merged inplace with binary predicate mod_lesser order
vector v3 = ( 0 0 1 -1 2 -2 3 -3 4 -4 5 -5 )
```

31. algo_min algo_min_if

按照指定比较规则比较两个迭代器所指的数据，返回数据较小迭代器。

```
input_iterator_t algo_min(
    input_iterator_t it_first,
    input_iterator_t it_second
);

input_iterator_t algo_min_if(
    input_iterator_t it_first,
    input_iterator_t it_last,
    binary_function_t bfun_op
);
```

- **Parameters**

  it_first: 第一个数据的迭代器。
  it_second: 第二个数据的迭代器。
  bfun_op: 指定的比较规则。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_min.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/cset.h>
#include <cstl/calgorithm.h>
#include <cstl/cfunctional.h>

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(set_t<int>);
    vector_iterator_t it_v;
    set_t* pset_s1 = create_set(int);
    set_iterator_t it_s;
```

```
    int i = 0;

    if(pvec_v1 == NULL || pset_s1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    set_init(pset_s1);

    set_insert(pset_s1, 0);
    set_insert(pset_s1, 1);
    set_insert(pset_s1, 2);
    vector_push_back(pvec_v1, pset_s1);
    vector_push_back(pvec_v1, pset_s1);
    set_clear(pset_s1);
    set_insert(pset_s1, 0);
    set_insert(pset_s1, 2);
    set_insert(pset_s1, 4);
    vector_push_back(pvec_v1, pset_s1);

    for(it_v = vector_begin(pvec_v1), i = 1;
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v), ++i)
    {
        printf("Set s%d ( ", i);
        for(it_s = set_begin((set_t*)iterator_get_pointer(it_v));
            !iterator_equal(it_s, set_end((set_t*)iterator_get_pointer(it_v)));
            it_s = iterator_next(it_s))
        {
            printf("%d ", *(int*)iterator_get_pointer(it_s));
        }
        printf(")\n");
    }

    it_v = algo_min(vector_begin(pvec_v1), iterator_next(vector_begin(pvec_v1)));
    printf("The min set between the frist and the second is ( ");
    for(it_s = set_begin((set_t*)iterator_get_pointer(it_v));
        !iterator_equal(it_s, set_end((set_t*)iterator_get_pointer(it_v)));
        it_s = iterator_next(it_s))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_s));
    }
    printf(")\n");

    it_v = algo_min(vector_begin(pvec_v1), iterator_prev(vector_end(pvec_v1)));
    printf("The min set between the frist and the third is ( ");
    for(it_s = set_begin((set_t*)iterator_get_pointer(it_v));
        !iterator_equal(it_s, set_end((set_t*)iterator_get_pointer(it_v)));
        it_s = iterator_next(it_s))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_s));
    }
    printf(")\n");

    it_v = algo_min_if(vector_begin(pvec_v1),
        iterator_next(vector_begin(pvec_v1)), fun_greater_set);
    printf("The min set between the frist and the second under greater than is ( ");
    for(it_s = set_begin((set_t*)iterator_get_pointer(it_v));
        !iterator_equal(it_s, set_end((set_t*)iterator_get_pointer(it_v)));
```

```
        it_s = iterator_next(it_s))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_s));
    }
    printf(")\n");

    it_v = algo_min_if(vector_begin(pvec_v1),
        iterator_prev(vector_end(pvec_v1)), fun_greater_set);
    printf("The min set between the frist and the third under greater than is ( ");
    for(it_s = set_begin((set_t*)iterator_get_pointer(it_v));
        !iterator_equal(it_s, set_end((set_t*)iterator_get_pointer(it_v)));
        it_s = iterator_next(it_s))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_s));
    }
    printf(")\n");

    vector_destroy(pvec_v1);
    set_destroy(pset_s1);

    return 0;
}
```

- **Output**

```
Set s1 ( 0 1 2 )
Set s2 ( 0 1 2 )
Set s3 ( 0 2 4 )
The min set between the frist and the second is ( 0 1 2 )
The min set between the frist and the third is ( 0 1 2 )
The min set between the frist and the second under greater than is ( 0 1 2 )
The min set between the frist and the third under greater than is ( 0 2 4 )
```

32. algo_min_element algo_min_element_if

返回数据区间中最小数据的迭代器。

```
forward_iterator_t algo_min_element(
    forward_iterator_t it_first,
    forward_iterator_t it_last
);

forward_iterator_t algo_min_element_if(
    forward_iterator_t it_first,
    forward_iterator_t it_last,
    binary_function_t bfun_op
);
```

- **Parameters**

  it_first: 数据区间的开始位置。
  it_last: 数据区间的末尾位置。
  bfun_op: 指定的比较规则。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_min_element.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <stdlib.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

static void _mod_lesser(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[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

    if(pvec_v1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);

    for(i = 0; i <= 3; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
    for(i = 1; i <= 4; ++i)
    {
        vector_push_back(pvec_v1, i * -2);
    }

    printf("Vector v1 is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    it_v = algo_min_element(vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("The smallest element in v1 is : %d.\n",
        *(int*)iterator_get_pointer(it_v));

    it_v = algo_min_element_if(vector_begin(pvec_v1),
        vector_end(pvec_v1), _mod_lesser);
    printf("The smallest element in v1 under the mod_lesser is : %d.\n",
        *(int*)iterator_get_pointer(it_v));

    vector_destroy(pvec_v1);

    return 0;
```

```
}
```

- **Output**

```
Vector v1 is ( 0 1 2 3 -2 -4 -6 -8 )
The smallest element in v1 is : -8.
The smallest element in v1 under the mod_lesser is : 0.
```

33. algo_mismatch algo_mismatch_if

将两个数据区间中的数据逐个比较，找出第一处不匹配的位置。

```
range_t algo_mismatch(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    input_iterator_t it_first2
);

range_t algo_mismatch_if(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    input_iterator_t it_first2,
    binary_function_t bfun_op
);
```

- **Parameters**

it_first1: 第一个数据区间的开始位置。
it_last1: 第一个数据区间的末尾位置。
it_first2: 第二个数据区间的开始位置。
bfun_op: 指定的比较规则。

- **Remarks**

返回两个数据区间中第一处不匹配的位置，it_begin 表示第一个数据区间中的位置，it_end 表示第二个数据区间中的位置。

要求第二个数据区间必须足够大，至少和第一个数据区间包含的数据数目相同。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_mismatch.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/clist.h>
#include <cstl/calgorithm.h>

static void _twice(const void* cpv_first, const void* cpv_second, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
```

```
    vector_t* pvec_v2 = create_vector(int);
    vector_iterator_t it_v;
    list_t* plist_l1 = create_list(int);
    list_iterator_t it_l;
    range_t r_result;
    int i = 0;

    if(pvec_v1 == NULL || pvec_v2 == NULL || plist_l1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);
    list_init(plist_l1);

    for(i = 0; i < 6; ++i)
    {
        vector_push_back(pvec_v1, i * 5);
        vector_push_back(pvec_v2, i * 10);
    }
    for(i = 0; i < 8; ++i)
    {
        list_push_back(plist_l1, i * 5);
    }

    printf("Vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("Vector v2 = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("List l1 = ( ");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_l));
    }
    printf(")\n");

    /* Test v1 and l1 for mismatch under identity */
    r_result = algo_mismatch(vector_begin(pvec_v1),
        vector_end(pvec_v1), list_begin(plist_l1));
    if(iterator_equal(r_result.it_begin, vector_end(pvec_v1)))
    {
        printf("The two ranges do not differ.\n");
    }
    else
    {
```

```
        printf("The first mismatch is between %d and %d.\n",
            *(int*)iterator_get_pointer(r_result.it_begin),
            *(int*)iterator_get_pointer(r_result.it_end));
    }

    /* Modifying l1 */
    list_insert(plist_l1, iterator_advance(list_begin(plist_l1), 2), 100);
    printf("Modified l1 = ( ");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_l));
    }
    printf(")\n");

    /* Testing v1 with modified l1 for mismatch under identity */
    r_result = algo_mismatch(vector_begin(pvec_v1),
        vector_end(pvec_v1), list_begin(plist_l1));
    if(iterator_equal(r_result.it_begin, vector_end(pvec_v1)))
    {
        printf("The two ranges do not differ.\n");
    }
    else
    {
        printf("The first mismatch is between %d and %d.\n",
            *(int*)iterator_get_pointer(r_result.it_begin),
            *(int*)iterator_get_pointer(r_result.it_end));
    }

    /* Test v1 and v2 for mismatch under the binary predicate twice */
    r_result = algo_mismatch_if(vector_begin(pvec_v1), vector_end(pvec_v1),
        vector_begin(pvec_v2), _twice);
    if(iterator_equal(r_result.it_begin, vector_end(pvec_v1)))
    {
        printf("The two ranges do not differ under the binary predicate twice.\n");
    }
    else
    {
        printf("The first mismatch is between %d and %d.\n",
            *(int*)iterator_get_pointer(r_result.it_begin),
            *(int*)iterator_get_pointer(r_result.it_end));
    }

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);
    list_destroy(plist_l1);

    return 0;
}

static void _twice(const void* cpv_first, const void* cpv_second, void* pv_output)
{
    *(bool_t*)pv_output = *(int*)cpv_first * 2 == *(int*)cpv_second ? true : false;
}
```

- **Output**

```
Vector v1 = ( 0 5 10 15 20 25 )
Vector v2 = ( 0 10 20 30 40 50 )
```

```
List l1 = ( 0 5 10 15 20 25 30 35 )
The two ranges do not differ.
Modified l1 = ( 0 5 100 10 15 20 25 30 35 )
The first mismatch is between 10 and 100.
The two ranges do not differ under the binary predicate twice.
```

34. algo_next_permutation algo_next_permutation_if

获得数据区间当前中数据的下一个排序。

```
bool_t algo_next_permutation(
    bidirectional_iterator_t it_first,
    bidirectional_iterator_t it_last
);

bool_t algo_next_permutation_if(
    bidirectional_iterator_t it_first,
    bidirectional_iterator_t it_last,
    binary_function_t bfun_op
);
```

- **Parameters**

it_first: 数据区间的开始位置。
it_last: 数据区间的末尾位置。
bfun_op: 指定的比较规则。

- **Remarks**

如果对于数据区间中当前数据排序存在下一个排列，则返回 true，并将数据调正成下一个排列，否则返回 false，并将数据调整成整个数据区间的第一个排列。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_next_permutation.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <stdlib.h>
#include <cstl/cdeque.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

static void _mod_lesser(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_q1 = create_deque(int);
```

```
    deque_iterator_t it_q;
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    bool_t b_result = false;
    int i = 0;

    if(pdeq_q1 == NULL || pvec_v1 == NULL)
    {
        return -1;
    }

    deque_init(pdeq_q1);
    vector_init(pvec_v1);

    deque_push_back(pdeq_q1, 5);
    deque_push_back(pdeq_q1, 1);
    deque_push_back(pdeq_q1, 10);

    printf("The original deque of q1 = ( ");
    for(it_q = deque_begin(pdeq_q1);
        !iterator_equal(it_q, deque_end(pdeq_q1));
        it_q = iterator_next(it_q))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_q));
    }
    printf(")\n");

    b_result = algo_next_permutation(deque_begin(pdeq_q1), deque_end(pdeq_q1));
    if(b_result)
    {
        printf("The lexicographically next permutation exists and has\n"
               "replaced the original ordering of the sequence in q1.\n");
    }
    else
    {
        printf("The lexicographically next permutation doesn't exists and\n"
               "the lexicographically smallest permutation has replaced the\n"
               "original ordering of the sequence in q1.\n");
    }
    printf("After one application of next_permutation q1 = ( ");
    for(it_q = deque_begin(pdeq_q1);
        !iterator_equal(it_q, deque_end(pdeq_q1));
        it_q = iterator_next(it_q))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_q));
    }
    printf(")\n");

    for(i = -3; i <= 3; ++i)
    {
        vector_push_back(pvec_v1, i);
    }

    printf("Vector v1 is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
```

```
    printf(")\n");

    algo_next_permutation_if(vector_begin(pvec_v1),
        vector_end(pvec_v1), _mod_lesser);
    printf("After the first next_permutation, vector v1 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    for(i = 1; i <= 5; ++i)
    {
        algo_next_permutation_if(vector_begin(pvec_v1),
            vector_end(pvec_v1), _mod_lesser);
        printf("After another next_permutation, vector v1 is:\n( ");
        for(it_v = vector_begin(pvec_v1);
            !iterator_equal(it_v, vector_end(pvec_v1));
            it_v = iterator_next(it_v))
        {
            printf("%d ", *(int*)iterator_get_pointer(it_v));
        }
        printf(")\n");
    }

    deque_destroy(pdeq_q1);
    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The original deque of q1 = ( 5 1 10 )
The lexicographically next permutation exists and has
replaced the original ordering of the sequence in q1.
After one application of next_permutation q1 = ( 5 10 1 )
Vector v1 is ( -3 -2 -1 0 1 2 3 )
After the first next_permutation, vector v1 is:
( -3 -2 -1 0 1 3 2 )
After another next_permutation, vector v1 is:
( -3 -2 -1 0 2 1 3 )
After another next_permutation, vector v1 is:
( -3 -2 -1 0 2 3 1 )
After another next_permutation, vector v1 is:
( -3 -2 -1 0 3 1 2 )
After another next_permutation, vector v1 is:
( -3 -2 -1 0 3 2 1 )
After another next_permutation, vector v1 is:
( -3 -2 -1 1 0 2 3 )
```

35. algo_nth_element algo_nth_element_if

按照指定规则，根据数据区间中第 n 个数据，将数据区间分为小于等于 n 和大于等于 n 两部分。

```
void algo_nth_element(
    random_access_iterator_t it_first,
```

```
    random_access_iterator_t it_nth,
    random_access_iterator_t it_last
);

void algo_nth_element_if(
    random_access_iterator_t it_first,
    random_access_iterator_t it_nth,
    random_access_iterator_t it_last,
    binary_function_t bfun_op
);
```

- **Parameters**

  it_first: 数据区间的开始位置。
  it_nth: 数据区间第 n 个数据的位置。
  it_last: 数据区间的末尾位置。
  bfun_op: 指定的比较规则。

- **Remarks**

  执行后第 n 个数据之前的数据都小于等于第 n 个数据，后面的数据都大于等于第 n 个数据。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_nth_element.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_v1 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

    if(pvec_v1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);

    for(i = 0; i <= 5; ++i)
    {
        vector_push_back(pvec_v1, i * 3);
    }
    for(i = 0; i <= 5; ++i)
    {
        vector_push_back(pvec_v1, i * 3 + 1);
    }
    for(i = 0; i <= 5; ++i)
```

```
    {
        vector_push_back(pvec_v1, i * 3 + 2);
    }

    printf("Original vector:\nv1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    algo_nth_element(vector_begin(pvec_v1),
        iterator_next_n(vector_begin(pvec_v1), 3), vector_end(pvec_v1));
    printf("Position 3 partitioned vector:\nv1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* To sort in descending order, specify binary predicate greater */
    algo_nth_element_if(vector_begin(pvec_v1),
        iterator_next_n(vector_begin(pvec_v1), 4),
        vector_end(pvec_v1), fun_greater_int);
    printf("Position 4 partitioned vector:\nv1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
Original vector:
v1 = ( 0 3 6 9 12 15 1 4 7 10 13 16 2 5 8 11 14 17 )
Position 3 partitioned vector:
v1 = ( 0 2 1 3 4 5 6 8 7 9 13 16 10 15 12 11 14 17 )
Position 4 partitioned vector:
v1 = ( 17 16 15 14 13 10 11 12 9 7 8 6 5 4 3 1 2 0 )
```

36. algo_partial_sort algo_partial_sort_if

按照指定的比较规则将数据区间中的数据进行部分排序。

```
void algo_partial_sort(
    random_access_iterator_t it_first,
    random_access_iterator_t it_middle,
    random_access_iterator_t it_last
```

```
);

void algo_partial_sort_if(
    random_access_iterator_t it_first,
    random_access_iterator_t it_middle,
    random_access_iterator_t it_last,
    binary_function_t bfun_op
);
```

- **Parameters**

  it_first: 数据区间的开始位置。
  it_middle: 数据区间中被排序的数据区间的末尾。
  it_last: 数据区间的末尾位置。
  bfun_op: 指定的比较规则。

- **Remarks**

  执行后数据区间的前半部分被排序，后半部分是无序的。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_partial_sort.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_v1 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

    if(pvec_v1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);

    for(i = 0; i <= 5; ++i)
    {
        vector_push_back(pvec_v1, i * 2);
    }
    for(i = 0; i <= 5; ++i)
    {
        vector_push_back(pvec_v1, i * 2 + 1);
    }

    printf("Original vector:\nv1 = ( ");
    for(it_v = vector_begin(pvec_v1);
```

```
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    algo_partial_sort(vector_begin(pvec_v1),
        iterator_next_n(vector_begin(pvec_v1), 6), vector_end(pvec_v1));
    printf("Partially sorted vector:\nv1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* To partially sort in descending order, specify binary predicate */
    algo_partial_sort_if(vector_begin(pvec_v1),
        iterator_next_n(vector_begin(pvec_v1), 8),
        vector_end(pvec_v1), fun_greater_int);
    printf("Partially resorted (greater) vector:\nv1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
Original vector:
v1 = ( 0 2 4 6 8 10 1 3 5 7 9 11 )
Partially sorted vector:
v1 = ( 0 1 2 3 4 5 10 8 6 7 9 11 )
Partially resorted (greater) vector:
v1 = ( 11 10 9 8 7 6 5 4 0 1 2 3 )
```

37. algo_partial_sort_copy　algo_partial_sort_copy_if

按照指定的比较规则将数据区间中的数据进行部分排序，将结果拷贝到目的数据区间中。

```
random_access_iterator_t algo_partial_sort_copy(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    random_access_iterator_t it_first2,
    random_access_iterator_t t_last2
);

random_access_iterator_t algo_partial_sort_copy_if(
    input_iterator_t it_first1,
```

```
    input_iterator_t it_last1,
    random_access_iterator_t it_first2,
    random_access_iterator_t it_last2,
    binary_function_t bfun_op
);
```

- **Parameters**

  it_first1: 第一个数据区间的开始位置。
  it_last1: 第一个数据区间的末尾位置。
  it_first2: 第二个数据区间的开始位置。
  it_last2: 第二个数据区间的末尾位置。
  bfun_op: 指定的比较规则。

- **Remarks**

  返回第二个数据区间中排序数据的末尾。

  算法根据第二个数据区间的个数来计算第一个数据区间中被排序的数据的个数，将排序的结果拷贝到第二个数据区间中，同时返回被拷贝的数据的末尾。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_partial_sort_copy.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_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_iterator_t it_v;
    vector_iterator_t it_end;
    list_t* plist_l1 = create_list(int);
    list_iterator_t it_l;
    int i = 0;

    if(pvec_v1 == NULL || pvec_v2 == NULL || plist_l1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);
    list_init(plist_l1);

    for(i = 0; i < 10; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
```

```
    algo_random_shuffle(vector_begin(pvec_v1), vector_end(pvec_v1));

    list_push_back(plist_l1, 60);
    list_push_back(plist_l1, 50);
    list_push_back(plist_l1, 20);
    list_push_back(plist_l1, 30);
    list_push_back(plist_l1, 40);
    list_push_back(plist_l1, 10);

    printf("Vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    printf("List l1 = ( ");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_l));
    }
    printf(")\n");

    /* Copying a partially sorted copy of l1 into v1 */
    it_end = algo_partial_sort_copy(list_begin(plist_l1), list_end(plist_l1),
        vector_begin(pvec_v1), iterator_next_n(vector_begin(pvec_v1), 3));
    printf("List l1 vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("The first v1 element one position beyond\n"
           "the first l1 element inserted was %d.\n",
           *(int*)iterator_get_pointer(it_end));

    /* Copying a partially sorted with greater copy of l1 to v2 */
    for(i = 0; i < 10; ++i)
    {
        vector_push_back(pvec_v2, i);
    }
    algo_random_shuffle(vector_begin(pvec_v2), vector_end(pvec_v2));
    it_end = algo_partial_sort_copy_if(list_begin(plist_l1), list_end(plist_l1),
        vector_begin(pvec_v2), iterator_next_n(vector_begin(pvec_v2), 6),
        fun_greater_int);
    printf("List l1 vector v2 = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("The first v2 element one position beyond\n"
```

```
            "the first l1 element inserted was %d.\n",
            *(int*)iterator_get_pointer(it_end));

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);
    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
Vector v1 = ( 5 7 4 3 8 9 6 0 1 2 )
List l1 = ( 60 50 20 30 40 10 )
List l1 vector v1 = ( 10 20 30 3 8 9 6 0 1 2 )
The first v1 element one position beyond
the first l1 element inserted was 3.
List l1 vector v2 = ( 60 50 40 30 20 10 8 6 2 9 )
The first v2 element one position beyond
the first l1 element inserted was 8.
```

38. algo_partition

按照指定规则将数据区间分为满足指定规则的数据和不满足指定规则的数据两部分。

```
forward_iterator_t algo_partition(
    forward_iterator_t it_first,
    forward_iterator_t it_last,
    unary_function_t ufun_op
);
```

- **Parameters**

it_first: 数据区间的开始位置。
it_last: 数据区间的末尾位置。
ufun_op: 指定的比较规则。

- **Remarks**

返回满足指定规则的数据区间的末尾。

算法根据指定的规则将满足指定规则的数据放在数据区间的前部，不满足指定规则的数据放在数据区间的后部。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_partition.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

static void _greater_5(const void* cpv_input, void* pv_output)
{
```

```
    *(bool_t*)pv_output = *(int*)cpv_input > 5 ? true : false;
}

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

    if(pvec_v1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);

    for(i = 0; i <= 10; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
    algo_random_shuffle(vector_begin(pvec_v1), vector_end(pvec_v1));

    printf("Vector v1 is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Partition the range with predicate greater5 */
    algo_partition(vector_begin(pvec_v1), vector_end(pvec_v1), _greater_5);

    printf("The partitioned set of elements in v1 is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
Vector v1 is ( 7 2 10 8 4 5 9 3 1 6 0 )
The partitioned set of elements in v1 is ( 7 6 10 8 9 5 4 3 1 2 0 )
```

39. algo_pop_heap　algo_pop_heap_if

将堆中优先级最高的数据移除。

```
void algo_pop_heap(
    random_access_iterator_t it_first,
```

```
    random_access_iterator_t it_last
);

void algo_pop_heap_if(
    random_access_iterator_t it_first,
    random_access_iterator_t it_last,
    binary_function_t bfun_op
);
```

- **Parameters**

  it_first: 数据区间的开始位置。
  it_last: 数据区间的末尾位置。
  bfun_op: 指定的比较规则。

- **Remarks**

  数据区间必须是满足指定比较规则的堆。算法执行后堆中最高优先级的数据被转移到数据区间的最后一个位置，其余的数据仍然是堆。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_pop_heap.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_v1 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

    if(pvec_v1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);

    for(i = 1; i <= 9; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
    algo_random_shuffle(vector_begin(pvec_v1), vector_end(pvec_v1));

    /* Make v1 a heap with default less than ordering */
    algo_make_heap(vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("The heap version of vector v1 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
```

```
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Add an element to the heap */
    vector_push_back(pvec_v1, 10);
    algo_push_heap(vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("The reheaped v1 with 10 added is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Remove the largest element form the heap */
    algo_pop_heap(vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("The heap v1 with 10 removed is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Make v1 a heap with greater than ordering */
    algo_make_heap_if(vector_begin(pvec_v1), vector_end(pvec_v1), fun_greater_int);
    printf("The greater-than heaped version of vector v1 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_push_back(pvec_v1, 0);
    algo_push_heap_if(vector_begin(pvec_v1), vector_end(pvec_v1), fun_greater_int);
    printf("The greater-than reheaped v1 with 0 added is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    algo_pop_heap_if(vector_begin(pvec_v1), vector_end(pvec_v1), fun_greater_int);
    printf("The greater-than heap v1 with 0 removed is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
```

```
    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The heap version of vector v1 is:
( 9 8 5 7 6 1 2 3 4 )
The reheaped v1 with 10 added is:
( 10 9 5 7 8 1 2 3 4 6 )
The heap v1 with 10 removed is:
( 9 8 5 7 6 1 2 3 4 10 )
The greater-than heaped version of vector v1 is:
( 1 3 2 4 6 5 9 7 8 10 )
The greater-than reheaped v1 with 0 added is:
( 0 1 2 4 3 5 9 7 8 10 6 )
The greater-than heap v1 with 0 removed is:
( 1 3 2 4 6 5 9 7 8 10 0 )
```

40. algo_prev_permutation　algo_prev_permutation_if

返回当前数据区间中数据的上一个排列。

```
bool_t algo_prev_permutation(
    bidirectional_iterator_t it_first,
    bidirectional_iterator_t it_last
);

bool_t algo_prev_permutation_if(
    bidirectional_iterator_t it_first,
    bidirectional_iterator_t it_last,
    binary_function_t bfun_op
);
```

- **Parameters**

it_first: 数据区间的开始位置。
it_last: 数据区间的末尾位置。
bfun_op: 指定的比较规则。

- **Remarks**

如果对于数据区间中当前数据排序存在上一个排列，则返回 true，并将数据调整成上一个排列，否则返回 false，并将数据调整成整个数据区间的最后一个排列。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_prev_permutation.c
 * compile with : -lcstl
 */

#include <stdio.h>
```

```
#include <stdlib.h>
#include <cstl/cdeque.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

static void _mod_lesser(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_q1 = create_deque(int);
    deque_iterator_t it_q;
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    bool_t b_result = false;
    int i = 0;

    if(pdeq_q1 == NULL || pvec_v1 == NULL)
    {
        return -1;
    }

    deque_init(pdeq_q1);
    vector_init(pvec_v1);

    deque_push_back(pdeq_q1, 5);
    deque_push_back(pdeq_q1, 1);
    deque_push_back(pdeq_q1, 10);

    printf("The original deque of q1 = ( ");
    for(it_q = deque_begin(pdeq_q1);
        !iterator_equal(it_q, deque_end(pdeq_q1));
        it_q = iterator_next(it_q))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_q));
    }
    printf(")\n");

    b_result = algo_prev_permutation(deque_begin(pdeq_q1), deque_end(pdeq_q1));
    if(b_result)
    {
        printf("The lexicographically next permutation exists and has\n"
               "replaced the original ordering of the sequence in q1.\n");
    }
    else
    {
        printf("The lexicographically next permutation doesn't exists and\n"
               "the lexicographically smallest permutation has replaced the\n"
               "original ordering of the sequence in q1.\n");
    }
    printf("After one application of prev_permutation q1 = ( ");
    for(it_q = deque_begin(pdeq_q1);
        !iterator_equal(it_q, deque_end(pdeq_q1));
        it_q = iterator_next(it_q))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_q));
```

```
    }
    printf(")\n");

    for(i = -3; i <= 3; ++i)
    {
        vector_push_back(pvec_v1, i);
    }

    printf("Vector v1 is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    algo_prev_permutation_if(vector_begin(pvec_v1),
        vector_end(pvec_v1), _mod_lesser);
    printf("After the first prev_permutation, vector v1 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    for(i = 1; i <= 5; ++i)
    {
        algo_prev_permutation_if(vector_begin(pvec_v1),
            vector_end(pvec_v1), _mod_lesser);
        printf("After another prev_permutation, vector v1 is:\n( ");
        for(it_v = vector_begin(pvec_v1);
            !iterator_equal(it_v, vector_end(pvec_v1));
            it_v = iterator_next(it_v))
        {
            printf("%d ", *(int*)iterator_get_pointer(it_v));
        }
        printf(")\n");
    }

    deque_destroy(pdeq_q1);
    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The original deque of q1 = ( 5 1 10 )
The lexicographically next permutation exists and has
replaced the original ordering of the sequence in q1.
After one application of prev_permutation q1 = ( 1 10 5 )
Vector v1 is ( -3 -2 -1 0 1 2 3 )
After the first prev_permutation, vector v1 is:
( -3 -2 0 3 2 1 -1 )
After another prev_permutation, vector v1 is:
( -3 -2 0 3 -1 2 1 )
After another prev_permutation, vector v1 is:
```

```
( -3 -2 0 3 -1 1 2 )
After another prev_permutation, vector v1 is:
( -3 -2 0 2 3 1 -1 )
After another prev_permutation, vector v1 is:
( -3 -2 0 2 -1 3 1 )
After another prev_permutation, vector v1 is:
( -3 -2 0 2 -1 1 3 )
```

41. algo_push_heap algo_push_heap_if

向堆中添加一个数据。

```
void algo_push_heap(
    random_access_iterator_t it_first,
    random_access_iterator_t it_last
);

void algo_push_heap_if(
    random_access_iterator_t it_first,
    random_access_iterator_t it_last,
    binary_function_t bfun_op
);
```

- **Parameters**

  it_first: 数据区间的开始位置。
  it_last: 数据区间的末尾位置。
  bfun_op: 指定的比较规则。

- **Remarks**

  数据区间的最后一个数据要向堆中添加的数据，除了最后一个数据，数据区间前面的数据是堆。算法执行后，真个数据区间变成堆。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_push_heap.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_v1 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

    if(pvec_v1 == NULL)
    {
```

```
        return -1;
    }

    vector_init(pvec_v1);

    for(i = 1; i <= 9; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
    algo_random_shuffle(vector_begin(pvec_v1), vector_end(pvec_v1));

    printf("Vector v1 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Make v1 a heap with default less than ordering */
    algo_make_heap(vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("The heap version of vector v1 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Add an element to the heap */
    vector_push_back(pvec_v1, 10);
    printf("The heap v1 with 10 pushed back is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    algo_push_heap(vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("The reheaped v1 with 10 added is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
```

```
    printf(")\n");

    /* Make v1 a heap with greater than ordering */
    algo_make_heap_if(vector_begin(pvec_v1), vector_end(pvec_v1), fun_greater_int);
    printf("The greater-than heaped version of vector v1 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_push_back(pvec_v1, 0);
    printf("The greater-than heap v1 with 0 pushed back is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    algo_push_heap_if(vector_begin(pvec_v1), vector_end(pvec_v1), fun_greater_int);
    printf("The greater-than reheaped v1 with 0 added is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
Vector v1 is:
( 8 4 9 6 1 3 7 2 5 )
The heap version of vector v1 is:
( 9 6 8 5 1 3 7 2 4 )
The heap v1 with 10 pushed back is:
( 9 6 8 5 1 3 7 2 4 10 )
The reheaped v1 with 10 added is:
( 10 9 8 5 6 3 7 2 4 1 )
The greater-than heaped version of vector v1 is:
( 1 2 3 4 6 8 7 5 10 9 )
The greater-than heap v1 with 0 pushed back is:
( 1 2 3 4 6 8 7 5 10 9 0 )
The greater-than reheaped v1 with 0 added is:
```

```
( 0 1 3 4 2 8 7 5 10 9 6 )
```

42. algo_random_sample　algo_random_sample_if

从第一个数据区间中随机抽出数据填充第二个数据区间。

```
random_access_iterator_t algo_random_sample(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    random_access_iterator_t it_first2,
    random_access_iterator_t it_last2
);

random_access_iterator_t algo_random_sample_if(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    random_access_iterator_t it_first2,
    random_access_iterator_t it_last2,
    unary_function_t ufun_op
);
```

- **Parameters**

  it_first1: 第一个数据区间的开始位置。
  it_last1: 第一个数据区间的末尾位置。
  it_first2: 第二个数据区间的开始位置。
  it_last2: 第二个数据区间的末尾位置。
  ufun_op: 指定的随机数生成函数。

- **Remarks**

  返回第二个数据区间中被抽出的数据的末尾。
  算法从第一个数据区间中随机抽出数据填充到第二个数据区间中，随机抽出的数据不重复。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_random_sample.c
 * compile with : -lcslt
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/cdeque.h>
#include <cstl/calgorithm.h>

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    deque_t* pdeq_dq1 = create_deque(int);
    deque_iterator_t it_dq;
```

```
int i = 0;

if(pvec_v1 == NULL || pdeq_dq1 == NULL)
{
    return -1;
}

vector_init(pvec_v1);
deque_init_n(pdeq_dq1, 10);

for(i = 0; i < 15; ++i)
{
    vector_push_back(pvec_v1, i);
}

printf("The original vector v1 is: ( ");
for(it_v = vector_begin(pvec_v1);
    !iterator_equal(it_v, vector_end(pvec_v1));
    it_v = iterator_next(it_v))
{
    printf("%d ", *(int*)iterator_get_pointer(it_v));
}
printf(")\n");

algo_random_sample(vector_begin(pvec_v1), vector_end(pvec_v1),
    deque_begin(pdeq_dq1), deque_end(pdeq_dq1));

printf("The random sample dq1 is : ( ");
for(it_dq = deque_begin(pdeq_dq1);
    !iterator_equal(it_dq, deque_end(pdeq_dq1));
    it_dq = iterator_next(it_dq))
{
    printf("%d ", *(int*)iterator_get_pointer(it_dq));
}
printf(")\n");

deque_resize(pdeq_dq1, 20);
algo_random_sample(vector_begin(pvec_v1), vector_end(pvec_v1),
    deque_begin(pdeq_dq1), deque_end(pdeq_dq1));

printf("The random sample dq1 is : ( ");
for(it_dq = deque_begin(pdeq_dq1);
    !iterator_equal(it_dq, deque_end(pdeq_dq1));
    it_dq = iterator_next(it_dq))
{
    printf("%d ", *(int*)iterator_get_pointer(it_dq));
}
printf(")\n");

vector_destroy(pvec_v1);
deque_destroy(pdeq_dq1);
```

```
    return 0;
}
```

- **Output**

```
The original vector v1 is: ( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 )
The random sample dq1 is : ( 0 12 2 3 11 5 6 7 8 9 )
The random sample dq1 is : ( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 0 0 0 0 0 )
```

43. algo_random_sample_n algo_random_sample_n_if

从第一个数据区间中随机抽出 n 个数据填充到目的数据区间。

```
output_iterator_t algo_random_sample_n(
    input_iterator_t it_first,
    input_iterator_t it_last,
    output_iterator_t it_result,
    size_t t_count
);

output_iterator_t algo_random_sample_n_if(
    input_iterator_t it_first,
    input_iterator_t it_last,
    output_iterator_t it_result,
    size_t it_count,
    unary_function_t ufun_op
);
```

- **Parameters**

it_first: 第一个数据区间的开始位置。
it_last: 第一个数据区间的末尾位置。
it_result: 第二个数据区间的开始位置。
t_count: 随机抽取的数据的个数。
ufun_op: 指定的随机数生成函数。

- **Remarks**

返回第二个数据区间中被抽出的数据的末尾。

算法从第一个数据区间中随机抽出 n 个数据填充到第二个数据区间中，随机抽出的数据不重复。必须保证第二个数据区间至少包含 n 个数据。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_random_sample_n.c
 * compile with : -lcslt
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/cdeque.h>
```

```
#include <cstl/calgorithm.h>

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    deque_t* pdeq_dq1 = create_deque(int);
    deque_iterator_t it_dq;
    int i = 0;

    if(pvec_v1 == NULL || pdeq_dq1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    deque_init_n(pdeq_dq1, 20);

    for(i = 0; i < 15; ++i)
    {
        vector_push_back(pvec_v1, i);
    }

    printf("The original vector v1 is: ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    algo_random_sample_n(vector_begin(pvec_v1), vector_end(pvec_v1),
        deque_begin(pdeq_dq1), 10);

    printf("The random sample 10 dq1 is : ( ");
    for(it_dq = deque_begin(pdeq_dq1);
        !iterator_equal(it_dq, deque_end(pdeq_dq1));
        it_dq = iterator_next(it_dq))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_dq));
    }
    printf(")\n");

    algo_random_sample_n(vector_begin(pvec_v1), vector_end(pvec_v1),
        deque_begin(pdeq_dq1), 20);

    printf("The random sample 20 dq1 is : ( ");
    for(it_dq = deque_begin(pdeq_dq1);
        !iterator_equal(it_dq, deque_end(pdeq_dq1));
        it_dq = iterator_next(it_dq))
```

```
    {
        printf("%d ", *(int*)iterator_get_pointer(it_dq));
    }
    printf(")\n");

    vector_destroy(pvec_v1);
    deque_destroy(pdeq_dq1);

    return 0;
}
```

- **Output**

```
The original vector v1 is: ( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 )
The random sample 10 dq1 is : ( 0 1 2 5 7 9 10 12 13 14 0 0 0 0 0 0 0 0 0 0 )
The random sample 20 dq1 is : ( 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 0 0 0 0 0 )
```

44. algo_random_shuffle algo_random_shuffle_if

对数据区间中的数据进行随机重排。

```
void algo_random_shuffle(
    random_access_iterator_t it_first,
    random_access_iterator_t it_last
);

void algo_random_shuffle_if(
    random_access_iterator_t it_first,
    random_access_iterator_t it_last,
    unary_function_t ufun_op
);
```

- **Parameters**

it_first: 数据区间的开始位置。
it_last: 数据区间的末尾位置。
ufun_op: 指定的随机数生成函数。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_random_shuffle.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
```

```
    int i = 0;

    if(pvec_v1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);

    for(i = 0; i <= 9; ++i)
    {
        vector_push_back(pvec_v1, i);
    }

    printf("The original version of vector v1 is: ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Shuffle once */
    algo_random_shuffle(vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("Vector v1 after one shuffle is:       ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Shuffle again */
    algo_random_shuffle(vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("Vector v1 after another shuffle is:   ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The original version of vector v1 is: ( 0 1 2 3 4 5 6 7 8 9 )
Vector v1 after one shuffle is:        ( 7 2 1 8 0 3 4 5 6 9 )
Vector v1 after another shuffle is:    ( 7 9 1 2 0 4 6 8 3 5 )
```

45. algo_remove

移除数据区间中的指定数据。

```
forward_iterator_t algo_remove(
    forward_iterator_t it_first,
    forward_iterator_t it_last,
    element
);
```

- **Parameters**

it_first: 数据区间的开始位置。
it_last: 数据区间的末尾位置。
element: 指定的数据。

- **Remarks**

返回移除数据后有效数据区间的末尾。
这个算法并不是真正的删除数据，而是使用后面的数据覆盖要删除的数据。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_remove.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_remove;
    vector_iterator_t it_v;
    int i = 0;

    if(pvec_v1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);

    for(i = 0; i < 10; ++i)
    {
        vector_push_back(pvec_v1, i);
```

```
    }
    for(i = 0; i < 4; ++i)
    {
        vector_push_back(pvec_v1, 7);
    }

    algo_random_shuffle(vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("The original vector v1 is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    it_remove = algo_remove(vector_begin(pvec_v1), vector_end(pvec_v1), 7);

    printf("Vector v1 with value 7 removed is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* To change the sequence size, use erase */
    vector_erase_range(pvec_v1, it_remove, vector_end(pvec_v1));

    printf("Vector v1 resized with value 7 removed is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The original vector v1 is ( 2 7 8 9 7 3 6 5 0 7 7 7 4 1 )
Vector v1 with value 7 removed is ( 2 8 9 3 6 5 0 4 1 7 7 7 4 1 )
Vector v1 resized with value 7 removed is ( 2 8 9 3 6 5 0 4 1 )
```

46. algo_remove_copy

将数据区间中指定的数据移除，将结果拷贝到目的数据区间。

```
output_iterator_t algo_remove_copy(
    input_iterator_t it_first,
    input_iterator_t it_last,
    output_iterator_t it_result,
    element
);
```

- **Parameters**

it_first: 数据区间的开始位置。
it_last: 数据区间的末尾位置。
it_result: 目的数据区间的开始位置。
element: 指定的数据。

- **Remarks**

返回目的数据区间中拷贝的数据区间的末尾。
这个算法将源数据区间中非指定数据拷贝到目的数据区间。目的数据区间要足够大。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_remove_copy.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

    if(pvec_v1 == NULL || pvec_v2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);

    for(i = 0; i < 10; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
```

```
    for(i = 0; i < 4; ++i)
    {
        vector_push_back(pvec_v1, 7);
    }

    algo_random_shuffle(vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("The original vector v1 is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_resize(pvec_v2, vector_size(pvec_v1));
    it_v = algo_remove_copy(vector_begin(pvec_v1),
        vector_end(pvec_v1), vector_begin(pvec_v2), 7);

    printf("Vector v1 is left unchanged as ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("Vector v2 is a copy of v1 with the value 7 removed ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);

    return 0;
}
```

- **Output**

```
The original vector v1 is ( 8 7 3 9 7 7 6 4 7 2 7 5 1 0 )
Vector v1 is left unchanged as ( 8 7 3 9 7 7 6 4 7 2 7 5 1 0 )
Vector v2 is a copy of v1 with the value 7 removed ( 8 3 9 6 4 2 5 1 0 0 0 0 0 0 )
```

47. algo_remove_copy_if

移除数据区间中符合指定规则的数据，将结果拷贝到目的数据区间。

```
output_iterator_t algo_remove_copy_if(
    input_iterator_t it_first,
    input_iterator_t it_last,
    output_iterator_t it_result,
    unary_function_t ufun_op
);
```

- **Parameters**

  it_first: 数据区间的开始位置。
  it_last: 数据区间的末尾位置。
  it_result: 目的数据区间的开始位置。
  ufun_op: 指定的规则。

- **Remarks**

  返回目的数据区间中拷贝的数据区间的末尾。
  这个算法将源数据区间中不符合指定规则的数据拷贝到目的数据区间。目的数据区间要足够大。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_remove_copy_if.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

static void _greater_6(const void* cpv_input, void* pv_output)
{
    *(bool_t*)pv_output = *(int*)cpv_input > 6 ? true : false;
}

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_iterator_t it_v;
    vector_iterator_t it_end;
    int i = 0;

    if(pvec_v1 == NULL || pvec_v2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);

    for(i = 0; i < 10; ++i)
    {
```

```
        vector_push_back(pvec_v1, i);
    }
    for(i = 0; i < 4; ++i)
    {
        vector_push_back(pvec_v1, 7);
    }

    algo_random_shuffle(vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("The original vector v1 is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_resize(pvec_v2, vector_size(pvec_v1));
    it_end = algo_remove_copy_if(vector_begin(pvec_v1),
        vector_end(pvec_v1), vector_begin(pvec_v2), _greater_6);

    printf("Vector v1 is left unchanged as ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("Vector v2 is a copy of v1 with the value greater 6 removed ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, it_end);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);

    return 0;
}
```

- **Output**

```
The original vector v1 is ( 7 7 7 7 2 3 6 0 8 5 9 1 7 4 )
Vector v1 is left unchanged as ( 7 7 7 7 2 3 6 0 8 5 9 1 7 4 )
Vector v2 is a copy of v1 with the value greater 6 removed ( 2 3 6 0 5 1 4 )
```

48. algo_remove_if

移除数据区间中符合指定规则的数据。

```
forward_iterator_t algo_remove_if(
    forward_iterator_t it_first,
    forward_iterator_t it_last,
    unary_function_t ufun_op
);
```

- **Parameters**

  it_first: 数据区间的开始位置。
  it_last: 数据区间的末尾位置。
  ufun_op: 指定的规则。

- **Remarks**

  返回移除数据后有效数据区间的末尾。
  这个算法并不是真正的删除数据，而是使用后面的数据覆盖要删除的数据。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_remove_if.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

static void _greater_6(const void* cpv_input, void* pv_output)
{
    *(bool_t*)pv_output = *(int*)cpv_input > 6 ? true : false;
}

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    vector_iterator_t it_end;
    int i = 0;

    if(pvec_v1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);

    for(i = 0; i <= 9; ++i)
    {
        vector_push_back(pvec_v1, i);
```

```
    }
    for(i = 0; i <= 3; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
    algo_random_shuffle(vector_begin(pvec_v1), vector_end(pvec_v1));

    printf("Vector v1 is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Remove elements satisfying predicate greater5 */
    it_end = algo_remove_if(vector_begin(pvec_v1), vector_end(pvec_v1), _greater_6);

    printf("Vector v1 with elements satisfying greater6 removed is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, it_end);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* To change the sequence size, use erase */
    vector_erase_range(pvec_v1, it_end, vector_end(pvec_v1));

    printf("Vector v1 resized elements satisfying greater6 removed is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
Vector v1 is ( 0 9 1 8 0 1 2 3 2 7 4 3 6 5 )
Vector v1 with elements satisfying greater6 removed is ( 0 1 0 1 2 3 2 4 3 6 5 )
Vector v1 resized elements satisfying greater6 removed is ( 0 1 0 1 2 3 2 4 3 6 5 )
```

49. algo_replace

将数据区间中指定的数据替换。

```
void algo_replace(
    forward_iterator_t it_first,
    forward_iterator_t it_last,
    old_element,
    new_element
);
```

- **Parameters**

  it_first: 数据区间的开始位置。
  it_last: 数据区间的末尾位置。
  old_element: 被替换的数据。
  new_element: 替换的数据。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_replace.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

    if(pvec_v1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);

    for(i = 0; i < 10; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
    for(i = 0; i < 4; ++i)
    {
        vector_push_back(pvec_v1, 7);
    }

    algo_random_shuffle(vector_begin(pvec_v1), vector_end(pvec_v1));
```

```
    printf("The original vector v1 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Replace elements with a value of 7 with a value of 700 */
    algo_replace(vector_begin(pvec_v1), vector_end(pvec_v1), 7, 700);

    printf("The vector v1 with a value 700 replacing that of 7 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The original vector v1 is:
( 4 8 7 6 0 7 7 9 2 3 7 5 7 1 )
The vector v1 with a value 700 replacing that of 7 is:
( 4 8 700 6 0 700 700 9 2 3 700 5 700 1 )
```

50. algo_replace_copy

将数据区间中指定的数据替换，将结果拷贝到目的数据区间。

```
void algo_replace_copy(
    input_iterator_t it_first,
    input_iterator_t it_last,
    output_iterator_t it_result,
    old_element,
    new_element
);
```

- **Parameters**

it_first: 数据区间的开始位置。
it_last: 数据区间的末尾位置。
it_result: 目的数据区间的开始位置。
old_element: 被替换的数据。
new_element: 替换的数据。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Remarks**

  目的数据区间至少要和源数据区间一样大。

- **Example**

```
/*
 * algo_replace_copy.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_v1 = create_vector(int);
    vector_iterator_t it_v;
    list_t* plist_l1 = create_list(int);
    list_iterator_t it_l;
    int i = 0;

    if(pvec_v1 == NULL || plist_l1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    list_init_n(plist_l1, 15);

    for(i = 0; i < 10; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
    for(i = 0; i < 4; ++i)
    {
        vector_push_back(pvec_v1, 7);
    }

    algo_random_shuffle(vector_begin(pvec_v1), vector_end(pvec_v1));
    for(i = 0; i < 16; ++i)
    {
        vector_push_back(pvec_v1, 1);
    }

    printf("The original vector v1 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
```

```
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /*
     * Replace elements in one part of a vector with a value of 7
     * with a value 70 and copy into another part of the vector
     */
    algo_replace_copy(vector_begin(pvec_v1),
        iterator_next_n(vector_begin(pvec_v1), 14),
        iterator_prev_n(vector_end(pvec_v1), 15), 7, 70);

    printf("The vector v1 with a value 70 replacing that of 7 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /*
     * Replace elements in a vector with a value of 70
     * with a value of 1 and copy into a list
     */
    algo_replace_copy(vector_begin(pvec_v1),
        iterator_next_n(vector_begin(pvec_v1), 14),
        list_begin(plist_l1), 7, 1);

    printf("The list copy l1 of v1 with the value 0 replacing that of 7 is:\n( ");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_l));
    }
    printf(")\n");

    vector_destroy(pvec_v1);
    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
The original vector v1 is:
( 6 1 7 3 2 5 7 9 7 4 7 8 7 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 )
The vector v1 with a value 70 replacing that of 7 is:
( 6 1 7 3 2 5 7 9 7 4 7 8 7 0 1 6 1 70 3 2 5 70 9 70 4 70 8 70 0 1 )
The list copy l1 of v1 with the value 0 replacing that of 7 is:
```

```
( 6 1 1 3 2 5 1 9 1 4 1 8 1 0 0 )
```

51. algo_replace_copy_if

将数据区间中符合指定规则的数据替换，将结果拷贝到目的数据区间。

```
output_iterator_t algo_replace_copy_if(
    input_iterator_t it_first,
    input_iterator_t it_last,
    output_iterator_t it_result,
    unary_function_t ufun_op,
    element
);
```

- **Parameters**

  it_first: 数据区间的开始位置。
  it_last: 数据区间的末尾位置。
  it_result: 目的数据区间的开始位置。
  ufun_op: 指定的规则。
  element: 替换的数据。

- **Remarks**

  返回目的数据区间中拷贝的数据区间的末尾。
  目的数据区间至少要和源数据区间一样大。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_replace_copy_if.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/clist.h>
#include <cstl/calgorithm.h>

static void _greater_6(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    list_t* plist_l1 = create_list(int);
    list_iterator_t it_l;
    int i = 0;

    if(pvec_v1 == NULL || plist_l1 == NULL)
    {
        return -1;
```

```
    }

    vector_init(pvec_v1);
    list_init_n(plist_l1, 15);

    for(i = 0; i < 10; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
    for(i = 0; i < 4; ++i)
    {
        vector_push_back(pvec_v1, 7);
    }

    algo_random_shuffle(vector_begin(pvec_v1), vector_end(pvec_v1));
    for(i = 0; i < 16; ++i)
    {
        vector_push_back(pvec_v1, 1);
    }

    printf("The original vector v1 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /*
     * Replace elements in one part of a vector with a value that greater than 6
     * with a value 70 and copy into another part of the vector
     */
    algo_replace_copy_if(vector_begin(pvec_v1),
        iterator_next_n(vector_begin(pvec_v1), 14),
        iterator_prev_n(vector_end(pvec_v1), 15), _greater_6, 70);

    printf("The vector v1 with a value 70 replacing those greater"
           " than 6 in the 1st half & copied into the 2nd half is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /*
     * Replace elements in a vector with a value that greater than 6
     * with a value of 1 and copy into a list
```

```
     */
    algo_replace_copy_if(vector_begin(pvec_v1),
        iterator_next_n(vector_begin(pvec_v1), 14),
        list_begin(plist_l1), _greater_6, -1);

    printf("The list copy l1 of v1 with the value -1 replacing "
           "those greater than 6 is:\n( ");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_l));
    }
    printf(")\n");

    vector_destroy(pvec_v1);
    list_destroy(plist_l1);

    return 0;
}

static void _greater_6(const void* cpv_input, void* pv_output)
{
    *(bool_t*)pv_output = *(int*)cpv_input > 6 ? true : false;
}
```

- **Output**

```
The original vector v1 is:
( 7 7 4 7 0 3 2 6 7 8 5 9 7 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 )
The vector v1 with a value 70 replacing those greater than 6 in the 1st half &
copied into the 2nd half is:
( 7 7 4 7 0 3 2 6 7 8 5 9 7 1 1 70 70 4 70 0 3 2 6 70 70 5 70 70 1 1 )
The list copy l1 of v1 with the value -1 replacing those greater than 6 is:
( -1 -1 4 -1 0 3 2 6 -1 -1 5 -1 -1 1 0 )
```

52. algo_replace_if

将数据区间中符合指定规则的数据替换。

```
void algo_replace_if(
    forward_iterator_t it_first,
    forward_iterator_t it_last,
    unary_function_t ufun_op,
    element
);
```

- **Parameters**

it_first: 数据区间的开始位置。
it_last: 数据区间的末尾位置。
ufun_op: 指定的规则。
element: 替换的数据。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_replace_if.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

static void _greater_6(const void* cpv_input, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

    if(pvec_v1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);

    for(i = 0; i < 10; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
    for(i = 0; i < 4; ++i)
    {
        vector_push_back(pvec_v1, 7);
    }

    algo_random_shuffle(vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("The original vector v1 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Replace elements satisfying the predicate greater6 with a value of 70 */
    algo_replace_if(vector_begin(pvec_v1), vector_end(pvec_v1), _greater_6, 70);

    printf("The vector v1 with a value 700 replacing those"
           " elements satisfying the greater 6 predicate is:\n( ");
```

```
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);

    return 0;
}

static void _greater_6(const void* cpv_input, void* pv_output)
{
    *(bool_t*)pv_output = *(int*)cpv_input > 6 ? true : false;
}
```

- **Output**

```
The original vector v1 is:
( 7 0 7 7 4 9 8 2 3 7 6 1 7 5 )
The vector v1 with a value 700 replacing those elements satisfying the greater 6
predicate is:
( 70 0 70 70 4 70 70 2 3 70 6 1 70 5 )
```

53. algo_reverse

将数据区间中的数据逆序。

```
void algo_reverse(
    bidirectional_iterator_t it_first,
    bidirectional_iterator_t it_last
);
```

- **Parameters**

it_first: 数据区间的开始位置。
it_last: 数据区间的末尾位置。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_reverse.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

int main(int argc, char* argv[])
{
```

```
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

    if(pvec_v1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);

    for(i = 0; i <= 9; ++i)
    {
        vector_push_back(pvec_v1, i);
    }

    printf("The original vector v1 is: ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Reverse the elements in the vector */
    algo_reverse(vector_begin(pvec_v1), vector_end(pvec_v1));

    printf("The modified vector v1 with values reversed is: ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
The original vector v1 is: ( 0 1 2 3 4 5 6 7 8 9 )
The modified vector v1 with values reversed is: ( 9 8 7 6 5 4 3 2 1 0 )
```

54. algo_reverse_copy

将数据区间中的数据逆序，将结果拷贝到目的数据区间。

```
output_iterator_t algo_reverse_copy(
```

```
    bidirectional_iterator_t it_first,
    bidirectional_iterator_t it_last,
    output_iterator_t it_result
);
```

- **Parameters**

  it_first: 数据区间的开始位置。
  it_last: 数据区间的末尾位置。
  it_result: 目的数据区间的开始位置。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Remarks**

  目的数据区间至少要和源数据区间一样大。

- **Example**

```
/*
 * algo_reverse_copy.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

    if(pvec_v1 == NULL || pvec_v2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init_n(pvec_v2, 10);

    for(i = 0; i <= 9; ++i)
    {
        vector_push_back(pvec_v1, i);
    }

    printf("The original vector v1 is: ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
```

```
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Reverse the elements in the vector */
    algo_reverse_copy(vector_begin(pvec_v1), vector_end(pvec_v1),
        vector_begin(pvec_v2));

    printf("The cppy v2 of the reversed vector v1 is: ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    printf("The original vector v1 remains unmodified as: ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);

    return 0;
}
```

- **Output**

```
The original vector v1 is: ( 0 1 2 3 4 5 6 7 8 9 )
The cppy v2 of the reversed vector v1 is: ( 9 8 7 6 5 4 3 2 1 0 )
The original vector v1 remains unmodified as: ( 0 1 2 3 4 5 6 7 8 9 )
```

55. algo_rotate

交换数据区间中两个相邻的部分。

```
forward_iterator_t algo_rotate(
    forward_iterator_t it_first,
    forward_iterator_t it_middle,
    forward_iterator_t it_last
);
```

- **Parameters**

it_first: 数据区间第一个有序部分的开始位置。
it_middle: 数据区间第一个有序部分的末尾位置，第二个有序部分的开始位置。
it_last: 数据区间第二个有序部分的末尾位置。

● **Requirements**

头文件 <cstl/calgorithm.h>。

● **Example**

```
/*
 * algo_rotate.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/cdeque.h>
#include <cstl/calgorithm.h>

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    deque_t* pdeq_q1 = create_deque(int);
    deque_iterator_t it_q;
    int i = 0;

    if(pvec_v1 == NULL || pdeq_q1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    deque_init(pdeq_q1);

    for(i = -3; i <= 5; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
    for(i = 0; i <= 5; ++i)
    {
        deque_push_back(pdeq_q1, i);
    }

    printf("Vector v1 is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    algo_rotate(vector_begin(pvec_v1),
        iterator_next_n(vector_begin(pvec_v1), 3), vector_end(pvec_v1));
    printf("After rotating, vector v1 is ( ");
```

```
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    printf("The original deque q1 is ( ");
    for(it_q = deque_begin(pdeq_q1);
        !iterator_equal(it_q, deque_end(pdeq_q1));
        it_q = iterator_next(it_q))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_q));
    }
    printf(")\n");

    i = 1;
    while(i <= iterator_distance(deque_begin(pdeq_q1), deque_end(pdeq_q1)))
    {
        algo_rotate(deque_begin(pdeq_q1),
            iterator_next(deque_begin(pdeq_q1)), deque_end(pdeq_q1));
        printf("After the rotation of a single deque element to the back q1 is ( ");
        for(it_q = deque_begin(pdeq_q1);
            !iterator_equal(it_q, deque_end(pdeq_q1));
            it_q = iterator_next(it_q))
        {
            printf("%d ", *(int*)iterator_get_pointer(it_q));
        }
        printf(")\n");
        i++;
    }

    vector_destroy(pvec_v1);
    deque_destroy(pdeq_q1);

    return 0;
}
```

- **Output**

```
Vector v1 is ( -3 -2 -1 0 1 2 3 4 5 )
After rotating, vector v1 is ( 0 1 2 3 4 5 -3 -2 -1 )
The original deque q1 is ( 0 1 2 3 4 5 )
After the rotation of a single deque element to the back q1 is ( 1 2 3 4 5 0 )
After the rotation of a single deque element to the back q1 is ( 2 3 4 5 0 1 )
After the rotation of a single deque element to the back q1 is ( 3 4 5 0 1 2 )
After the rotation of a single deque element to the back q1 is ( 4 5 0 1 2 3 )
After the rotation of a single deque element to the back q1 is ( 5 0 1 2 3 4 )
After the rotation of a single deque element to the back q1 is ( 0 1 2 3 4 5 )
```

56. algo_rotate_copy

交换数据区间中相邻两部分数据，将结果拷贝到目的数据区间。

```
output_iterator_t algo_rotate_copy(
    forward_iterator_t it_first,
    forward_iterator_t it_middle,
    forward_iterator_t it_last,
    output_iterator_t it_result
);
```

- **Parameters**

it_first: 数据区间第一个有序部分的开始位置。
it_middle: 数据区间第一个有序部分的末尾位置，第二个有序部分的开始位置。
it_last: 数据区间第二个有序部分的末尾位置。
it_result: 目的数据区间的开始位置。

- **Remarks**

返回目的数据区间中拷贝的数据的末尾。
要保证目的数据区间至少和源数据区间一样大。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_rotate_copy.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/cdeque.h>
#include <cstl/calgorithm.h>

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_iterator_t it_v;
    deque_t* pdeq_q1 = create_deque(int);
    deque_t* pdeq_q2 = create_deque(int);
    deque_iterator_t it_q;
    int i = 0;

    if(pvec_v1 == NULL || pvec_v2 == NULL ||
       pdeq_q1 == NULL || pdeq_q2 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init_n(pvec_v2, 9);
```

```
    deque_init(pdeq_q1);
    deque_init_n(pdeq_q2, 6);

    for(i = -3; i <= 5; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
    for(i = 0; i <= 5; ++i)
    {
        deque_push_back(pdeq_q1, i);
    }

    printf("Vector v1 is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    algo_rotate_copy(vector_begin(pvec_v1),
        iterator_next_n(vector_begin(pvec_v1), 3),
        vector_end(pvec_v1), vector_begin(pvec_v2));
    printf("After rotating, vector v2 is ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    printf("The original deque q1 is ( ");
    for(it_q = deque_begin(pdeq_q1);
        !iterator_equal(it_q, deque_end(pdeq_q1));
        it_q = iterator_next(it_q))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_q));
    }
    printf(")\n");

    i = 1;
    while(i <= iterator_distance(deque_begin(pdeq_q1), deque_end(pdeq_q1)))
    {
        algo_rotate_copy(deque_begin(pdeq_q1),
            iterator_next(deque_begin(pdeq_q1)),
            deque_end(pdeq_q1), deque_begin(pdeq_q2));
        printf("After the rotation of a single deque element to the back q2 is ( ");
        for(it_q = deque_begin(pdeq_q2);
```

```
            !iterator_equal(it_q, deque_end(pdeq_q2));
            it_q = iterator_next(it_q))
        {
            printf("%d ", *(int*)iterator_get_pointer(it_q));
        }
        printf(")\n");
        i++;
    }

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);
    deque_destroy(pdeq_q1);
    deque_destroy(pdeq_q2);

    return 0;
}
```

- **Output**

```
Vector v1 is ( -3 -2 -1 0 1 2 3 4 5 )
After rotating, vector v2 is ( 0 1 2 3 4 5 -3 -2 -1 )
The original deque q1 is ( 0 1 2 3 4 5 )
After the rotation of a single deque element to the back q2 is ( 1 2 3 4 5 0 )
After the rotation of a single deque element to the back q2 is ( 1 2 3 4 5 0 )
After the rotation of a single deque element to the back q2 is ( 1 2 3 4 5 0 )
After the rotation of a single deque element to the back q2 is ( 1 2 3 4 5 0 )
After the rotation of a single deque element to the back q2 is ( 1 2 3 4 5 0 )
After the rotation of a single deque element to the back q2 is ( 1 2 3 4 5 0 )
```

57. algo_search algo_search_if

在第一个数据区间中查找子数据区间出现的第一个位置。

```
forward_iterator_t algo_search(
    forward_iterator_t it_first1,
    forward_iterator_t it_last1,
    forward_iterator_t it_first2,
    forward_iterator_t it_last2
);

forward_iterator_t algo_search_if(
    forward_iterator_t it_first1,
    forward_iterator_t it_last1,
    forward_iterator_t it_first2,
    forward_iterator_t it_last2,
    binary_function_t bfun_op
);
```

- **Parameters**

it_first1: 第一个数据区间的开始位置。
it_last1: 第一个数据区间的末尾位置。
it_first2: 子数据区间的开始位置。
it_last2: 子数据区间的末尾位置。

bfun_op: 指定的比较规则。

- **Remarks**

返回目的数据区间中第一个符合规则的子数据区间的第一个数据的迭代器，如果不包含这个子数据区间，返回数据区间的末尾。

这个算法默认使用数据类型的等于操作函数。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_search.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/clist.h>
#include <cstl/calgorithm.h>

static void _twice(const void* cpv_first, const void* cpv_second, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    list_t* plist_l1 = create_list(int);
    vector_iterator_t it_v;
    list_iterator_t it_l;
    int i = 0;

    if(pvec_v1 == NULL || pvec_v2 == NULL || plist_l1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);
    list_init(plist_l1);

    for(i = 0; i <= 5; ++i)
    {
        vector_push_back(pvec_v1, i * 5);
    }
    for(i = 0; i <= 5; ++i)
    {
        vector_push_back(pvec_v1, i * 5);
    }
    for(i = 2; i <= 4; ++i)
    {
        vector_push_back(pvec_v2, i * 10);
```

```
    }
    for(i = 4; i <= 5; ++i)
    {
        list_push_back(plist_l1, i * 5);
    }

    printf("Vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("List l1 = ( ");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_l));
    }
    printf(")\n");
    printf("Vector v2 = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Searching v1 for first match to l1 under identity */
    it_v = algo_search(vector_begin(pvec_v1), vector_end(pvec_v1),
        list_begin(plist_l1), list_end(plist_l1));

    if(iterator_equal(it_v, vector_end(pvec_v1)))
    {
        printf("There is no match of l1 in v1.\n");
    }
    else
    {
        printf("There is at least one match of l1 in v1\n"
               "and the first one begins at position %d.\n",
               iterator_distance(vector_begin(pvec_v1), it_v));
    }

    /* Searching v1 for a match to v2 under the binary predicate twice */
    it_v = algo_search_if(vector_begin(pvec_v1), vector_end(pvec_v1),
        vector_begin(pvec_v2), vector_end(pvec_v2), _twice);
    if(iterator_equal(it_v, vector_end(pvec_v1)))
```

```
    {
        printf("There is no match of v2 in v1.\n");
    }
    else
    {
        printf("There is a sequence of elements in v1 that are equivalent\n"
               "to those in v2 under the binary predicate twice\n"
               "and the first one begins at position %d.\n",
                iterator_distance(vector_begin(pvec_v1), it_v));
    }

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);
    list_destroy(plist_l1);

    return 0;
}

static void _twice(const void* cpv_first, const void* cpv_second, void* pv_output)
{
    *(bool_t*)pv_output = *(int*)cpv_first * 2 == *(int*)cpv_second ? true : false;
}
```

- **Output**

```
Vector v1 = ( 0 5 10 15 20 25 0 5 10 15 20 25 )
List l1 = ( 20 25 )
Vector v2 = ( 20 30 40 )
There is at least one match of l1 in v1
and the first one begins at position 4.
There is a sequence of elements in v1 that are equivalent
to those in v2 under the binary predicate twice
and the first one begins at position 2.
```

58. algo_search_end algo_search_end_if

在数据区间中查找最后一个符合规则的子数据区间。

```
forward_iterator_t algo_search_end(
    forward_iterator_t it_first1,
    forward_iterator_t it_last1,
    forward_iterator_t it_first2,
    forward_iterator_t it_last2
);

forward_iterator_t algo_search_end_if(
    forward_iterator_t it_first1,
    forward_iterator_t it_last1,
    forward_iterator_t it_first2,
    forward_iterator_t it_last2,
    binary_function_t bfun_op
);
```

- **Parameters**

it_first1:　　第一个数据区间的开始位置。
it_last1:　　第一个数据区间的末尾位置。
it_first2:　　子数据区间的开始位置。
it_last2:　　子数据区间的末尾位置。
bfun_op:　　指定的比较规则。

- **Remarks**

返回目的数据区间中最后一个符合规则的子数据区间的第一个数据的迭代器，如果不包含这个子数据区间，返回数据区间的末尾。

这个算法默认使用数据类型的等于操作函数。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_search_end.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/clist.h>
#include <cstl/calgorithm.h>

static void _twice(const void* cpv_first, const void* cpv_second, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    list_t* plist_l1 = create_list(int);
    vector_iterator_t it_v;
    list_iterator_t it_l;
    int i = 0;

    if(pvec_v1 == NULL || pvec_v2 == NULL || plist_l1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);
    list_init(plist_l1);

    for(i = 0; i <= 5; ++i)
    {
        vector_push_back(pvec_v1, i * 5);
    }
    for(i = 0; i <= 5; ++i)
    {
        vector_push_back(pvec_v1, i * 5);
```

```
    }
    for(i = 2; i <= 4; ++i)
    {
        vector_push_back(pvec_v2, i * 10);
    }
    for(i = 1; i <= 4; ++i)
    {
        list_push_back(plist_l1, i * 5);
    }

    printf("Vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("List l1 = ( ");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_l));
    }
    printf(")\n");
    printf("Vector v2 = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Searching v1 for first match to l1 under identity */
    it_v = algo_search_end(vector_begin(pvec_v1), vector_end(pvec_v1),
        list_begin(plist_l1), list_end(plist_l1));

    if(iterator_equal(it_v, vector_end(pvec_v1)))
    {
        printf("There is no match of l1 in v1.\n");
    }
    else
    {
        printf("There is at least one match of l1 in v1\n"
               "and the last one begins at position %d.\n",
               iterator_distance(vector_begin(pvec_v1), it_v));
    }
```

```
    /* Searching v1 for a match to v2 under the binary predicate twice */
    it_v = algo_search_end_if(vector_begin(pvec_v1), vector_end(pvec_v1),
        vector_begin(pvec_v2), vector_end(pvec_v2), _twice);
    if(iterator_equal(it_v, vector_end(pvec_v1)))
    {
        printf("There is no match of v2 in v1.\n");
    }
    else
    {
        printf("There is a sequence of elements in v1 that are equivalent\n"
               "to those in v2 under the binary predicate twice\n"
               "and the last one begins at position %d.\n",
                iterator_distance(vector_begin(pvec_v1), it_v));
    }

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);
    list_destroy(plist_l1);

    return 0;
}

static void _twice(const void* cpv_first, const void* cpv_second, void* pv_output)
{
    *(bool_t*)pv_output = *(int*)cpv_first * 2 == *(int*)cpv_second ? true : false;
}
```

- **Output**

```
Vector v1 = ( 0 5 10 15 20 25 0 5 10 15 20 25 )
List l1 = ( 5 10 15 20 )
Vector v2 = ( 20 30 40 )
There is at least one match of l1 in v1
and the last one begins at position 7.
There is a sequence of elements in v1 that are equivalent
to those in v2 under the binary predicate twice
and the last one begins at position 8.
```

59. algo_search_n algo_search_n_if

在数据区间中搜索第一个有连续 n 个指定数据出现的位置。

```
forward_iterator_t algo_search_n(
    forward_iterator_t it_first,
    forward_iterator_t it_last,
    size_t t_count,
    element
);

forward_iterator_t algo_search_n_if(
    forward_iterator_t it_first,
    forward_iterator_t it_last,
    size_t t_count,
    element,
```

```
    binary_function_t bfun_op
);
```

- **Parameters**

  it_first: 数据区间的开始位置。
  it_last: 数据区间的末尾位置。
  t_count: 要搜索的指定数据的个数。
  element: 指定的数据数据。
  bfun_op: 指定的比较规则函数。

- **Remarks**

  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，如果没有出现连续 n 个指定数据，那么返回数据区间的末尾。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_search_n.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

static void _twice(const void* cpv_first, const void* cpv_second, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

    if(pvec_v1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);

    for(i = 0; i <= 5; ++i)
    {
        vector_push_back(pvec_v1, i * 5);
    }
    for(i = 0; i <= 2; ++i)
    {
        vector_push_back(pvec_v1, 5);
    }
    for(i = 0; i <= 5; ++i)
    {
```

```
        vector_push_back(pvec_v1, i * 5);
    }
    for(i = 0; i <= 3; ++i)
    {
        vector_push_back(pvec_v1, 10);
    }
    for(i = 0; i <= 2; ++i)
    {
        vector_push_back(pvec_v1, 5);
    }

    printf("Vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Searching v1 for first match to (5 5 5) under identity */
    it_v = algo_search_n(vector_begin(pvec_v1), vector_end(pvec_v1), 3, 5);
    if(iterator_equal(it_v, vector_end(pvec_v1)))
    {
        printf("There is no match for a sequence (5 5 5) in v1.\n");
    }
    else
    {
        printf("There is at least one match of a sequence (5 5 5)"
               " in v1 and the first one begins at position %d.\n",
               iterator_distance(vector_begin(pvec_v1), it_v));
    }

    /* Searching v1 for first match to (5 5 5 5) under twice */
    it_v = algo_search_n_if(vector_begin(pvec_v1),
        vector_end(pvec_v1), 4, 5, _twice);
    if(iterator_equal(it_v, vector_end(pvec_v1)))
    {
        printf("There is no match for a sequence (5 5 5 5) in v1"
               " under the equivalence predicate twice.\n");
    }
    else
    {
        printf("There is a match of a sequence (5 5 5 5) "
               "under the equivalence predicate twice"
               " in v1 and the first one begins at position %d.\n",
               iterator_distance(vector_begin(pvec_v1), it_v));
    }

    vector_destroy(pvec_v1);
```

```
    return 0;
}

static void _twice(const void* cpv_first, const void* cpv_second, void* pv_output)
{
    *(bool_t*)pv_output = *(int*)cpv_first == *(int*)cpv_second * 2 ? true : false;
}
```

- **Output**

```
Vector v1 = ( 0 5 10 15 20 25 5 5 5 0 5 10 15 20 25 10 10 10 10 5 5 5 )
There is at least one match of a sequence (5 5 5) in v1 and the first one begins at
position 6.
There is a match of a sequence (5 5 5 5) under the equivalence predicate twice in v1
and the first one begins at position 15.
```

60. algo_set_difference algo_set_difference_if

求两个有序数据区间的差集。

```
output_iterator_t algo_set_difference(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    input_iterator_t it_first2,
    input_iterator_t it_last2,
    output_iterator_t it_result
);

output_iterator_t algo_set_difference_if(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    input_iterator_t it_first2,
    input_iterator_t it_last2,
    output_iterator_t it_result,
    binary_function_t bfun_op
);
```

- **Parameters**

it_first1: 第一个数据区间的开始位置。
it_last1: 第一个数据区间的末尾位置。
it_first2: 第二个数据区间的开始位置。
it_last2: 第二个数据区间的末尾位置。
it_result: 目的数据区间的开始位置。
bfun_op: 指定的比较规则。

- **Remarks**

返回目的数据区间中合并后的有序数据区间的末尾。
目的数据区间要足够大。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

```c
/*
 * algo_set_difference.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>
#include <cstl/cfunctional.h>

static void _mod_lesser(const void* cpv_first,
    const void* cpv_second, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t* pvec_v1a = create_vector(int);
    vector_t* pvec_v1b = create_vector(int);
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2a = create_vector(int);
    vector_t* pvec_v2b = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_t* pvec_v3a = create_vector(int);
    vector_t* pvec_v3b = create_vector(int);
    vector_t* pvec_v3 = create_vector(int);
    vector_iterator_t it_v;
    vector_iterator_t it_result;
    int i = 0;

    if(pvec_v1a == NULL || pvec_v1b == NULL || pvec_v1 == NULL ||
       pvec_v2a == NULL || pvec_v2b == NULL || pvec_v2 == NULL ||
       pvec_v3a == NULL || pvec_v3b == NULL || pvec_v3 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1a);
    vector_init(pvec_v1b);
    vector_init(pvec_v1);
    vector_init(pvec_v2a);
    vector_init(pvec_v2b);
    vector_init(pvec_v2);
    vector_init(pvec_v3a);
    vector_init(pvec_v3b);
    vector_init(pvec_v3);

    /* Constructing vectors v1a and v1b with default less than ordering */
    for(i = -1; i <= 4; ++i)
    {
        vector_push_back(pvec_v1a, i);
    }
    for(i = -3; i <= 0; ++i)
    {
```

```
        vector_push_back(pvec_v1b, i);
    }
    vector_resize(pvec_v1, 12);
    printf("Original vector v1a with range sorted by the\n"
           "binary predicate less than is v1a = ( ");
    for(it_v = vector_begin(pvec_v1a);
        !iterator_equal(it_v, vector_end(pvec_v1a));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("Original vector v1b with range sorted by the\n"
           "binary predicate less than is v1b = ( ");
    for(it_v = vector_begin(pvec_v1b);
        !iterator_equal(it_v, vector_end(pvec_v1b));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Constructing vectors v2a and v2b with ranges sorted by greater */
    vector_assign(pvec_v2a, pvec_v1a);
    vector_assign(pvec_v2b, pvec_v1b);
    vector_assign(pvec_v2, pvec_v1);
    algo_sort_if(vector_begin(pvec_v2a), vector_end(pvec_v2a), fun_greater_int);
    algo_sort_if(vector_begin(pvec_v2b), vector_end(pvec_v2b), fun_greater_int);
    printf("Original vector v2a with range sorted by the\n"
           "binary predicate greater than is v2a = ( ");
    for(it_v = vector_begin(pvec_v2a);
        !iterator_equal(it_v, vector_end(pvec_v2a));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("Original vector v2b with range sorted by the\n"
           "binary predicate greater than is v2b = ( ");
    for(it_v = vector_begin(pvec_v2b);
        !iterator_equal(it_v, vector_end(pvec_v2b));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Constructing vectors v3a and v3b with ranges sorted by mod_lesser */
    vector_assign(pvec_v3a, pvec_v1a);
```

```
    vector_assign(pvec_v3b, pvec_v1b);
    vector_assign(pvec_v3, pvec_v1);
    algo_sort_if(vector_begin(pvec_v3a), vector_end(pvec_v3a), _mod_lesser);
    algo_sort_if(vector_begin(pvec_v3b), vector_end(pvec_v3b), _mod_lesser);
    printf("Original vector v3a with range sorted by the\n"
           "binary predicate mod_lesser is v3a = ( ");
    for(it_v = vector_begin(pvec_v3a);
        !iterator_equal(it_v, vector_end(pvec_v3a));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("Original vector v3b with range sorted by the\n"
           "binary predicate greater than is v3b = ( ");
    for(it_v = vector_begin(pvec_v3b);
        !iterator_equal(it_v, vector_end(pvec_v3b));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /*
     * To combine int a difference in ascending order
     * with the default binary predicate less
     */
    it_result = algo_set_difference(vector_begin(pvec_v1a), vector_end(pvec_v1a),
        vector_begin(pvec_v1b), vector_end(pvec_v1b), vector_begin(pvec_v1));
    printf("Difference of source ranges with default order\n"
           "vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, it_result);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /*
     * To combine int a difference in ascending order
     * with the specify binary predicate greater
     */
    it_result = algo_set_difference_if(vector_begin(pvec_v2a), vector_end(pvec_v2a),
        vector_begin(pvec_v2b), vector_end(pvec_v2b), vector_begin(pvec_v2),
        fun_greater_int);
    printf("Difference of source ranges with binary predicate greater order\n"
           "vector v2 = ( ");
    for(it_v = vector_begin(pvec_v2);
```

```
        !iterator_equal(it_v, it_result);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /*
     * To combine int a difference in ascending order
     * with the user_defined binary predicate mod_lesser
     */
    it_result = algo_set_difference_if(vector_begin(pvec_v3a), vector_end(pvec_v3a),
        vector_begin(pvec_v3b), vector_end(pvec_v3b), vector_begin(pvec_v3),
        _mod_lesser);
    printf("Difference of source ranges with binary predicate mod_lesser order\n"
           "vector v3 = ( ");
    for(it_v = vector_begin(pvec_v3);
        !iterator_equal(it_v, it_result);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1a);
    vector_destroy(pvec_v1b);
    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2a);
    vector_destroy(pvec_v2b);
    vector_destroy(pvec_v2);
    vector_destroy(pvec_v3a);
    vector_destroy(pvec_v3b);
    vector_destroy(pvec_v3);

    return 0;
}

static void _mod_lesser(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;
}
```

- **Output**

```
Original vector v1a with range sorted by the
binary predicate less than is v1a = ( -1 0 1 2 3 4 )
Original vector v1b with range sorted by the
binary predicate less than is v1b = ( -3 -2 -1 0 )
Original vector v2a with range sorted by the
binary predicate greater than is v2a = ( 4 3 2 1 0 -1 )
```

```
Original vector v2b with range sorted by the
binary predicate greater than is v2b = ( 0 -1 -2 -3 )
Original vector v3a with range sorted by the
binary predicate mod_lesser is v3a = ( 0 -1 1 2 3 4 )
Original vector v3b with range sorted by the
binary predicate greater than is v3b = ( 0 -1 -2 -3 )
Difference of source ranges with default order
vector v1 = ( 1 2 3 4 )
Difference of source ranges with binary predicate greater order
vector v2 = ( 4 3 2 1 )
Difference of source ranges with binary predicate mod_lesser order
vector v3 = ( 1 4 )
```

61. algo_set_intersection　algo_set_intersection_if

求两个有序数据区间的交集。

```
output_iterator_t algo_set_intersection(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    input_iterator_t it_first2,
    input_iterator_t it_last2,
    output_iterator_t it_result
);

output_iterator_t algo_set_intersection_if(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    input_iterator_t it_first2,
    input_iterator_t it_last2,
    output_iterator_t it_result,
    binary_function_t bfun_op
);
```

- **Parameters**

it_first1:　第一个数据区间的开始位置。
it_last1:　第一个数据区间的末尾位置。
it_first2:　第二个数据区间的开始位置。
it_last2:　第二个数据区间的末尾位置。
it_result:　目的数据区间的开始位置。
bfun_op:　指定的比较规则。

- **Remarks**

返回目的数据区间中合并后的有序数据区间的末尾。
目的数据区间要足够大。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_set_intersection.c
 * compile with : -lcstl
```

```
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>
#include <cstl/cfunctional.h>

static void _mod_lesser(const void* cpv_first,
    const void* cpv_second, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t* pvec_v1a = create_vector(int);
    vector_t* pvec_v1b = create_vector(int);
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2a = create_vector(int);
    vector_t* pvec_v2b = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_t* pvec_v3a = create_vector(int);
    vector_t* pvec_v3b = create_vector(int);
    vector_t* pvec_v3 = create_vector(int);
    vector_iterator_t it_v;
    vector_iterator_t it_result;
    int i = 0;

    if(pvec_v1a == NULL || pvec_v1b == NULL || pvec_v1 == NULL ||
       pvec_v2a == NULL || pvec_v2b == NULL || pvec_v2 == NULL ||
       pvec_v3a == NULL || pvec_v3b == NULL || pvec_v3 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1a);
    vector_init(pvec_v1b);
    vector_init(pvec_v1);
    vector_init(pvec_v2a);
    vector_init(pvec_v2b);
    vector_init(pvec_v2);
    vector_init(pvec_v3a);
    vector_init(pvec_v3b);
    vector_init(pvec_v3);

    /* Constructing vectors v1a and v1b with default less than ordering */
    for(i = -1; i <= 3; ++i)
    {
        vector_push_back(pvec_v1a, i);
    }
    for(i = -3; i <= 1; ++i)
    {
        vector_push_back(pvec_v1b, i);
    }
    vector_resize(pvec_v1, 12);
```

```
printf("Original vector v1a with range sorted by the\n"
       "binary predicate less than is v1a = ( ");
for(it_v = vector_begin(pvec_v1a);
    !iterator_equal(it_v, vector_end(pvec_v1a));
    it_v = iterator_next(it_v))
{
    printf("%d ", *(int*)iterator_get_pointer(it_v));
}
printf(")\n");
printf("Original vector v1b with range sorted by the\n"
       "binary predicate less than is v1b = ( ");
for(it_v = vector_begin(pvec_v1b);
    !iterator_equal(it_v, vector_end(pvec_v1b));
    it_v = iterator_next(it_v))
{
    printf("%d ", *(int*)iterator_get_pointer(it_v));
}
printf(")\n");

/* Constructing vectors v2a and v2b with ranges sorted by greater */
vector_assign(pvec_v2a, pvec_v1a);
vector_assign(pvec_v2b, pvec_v1b);
vector_assign(pvec_v2, pvec_v1);
algo_sort_if(vector_begin(pvec_v2a), vector_end(pvec_v2a), fun_greater_int);
algo_sort_if(vector_begin(pvec_v2b), vector_end(pvec_v2b), fun_greater_int);
printf("Original vector v2a with range sorted by the\n"
       "binary predicate greater than is v2a = ( ");
for(it_v = vector_begin(pvec_v2a);
    !iterator_equal(it_v, vector_end(pvec_v2a));
    it_v = iterator_next(it_v))
{
    printf("%d ", *(int*)iterator_get_pointer(it_v));
}
printf(")\n");
printf("Original vector v2b with range sorted by the\n"
       "binary predicate greater than is v2b = ( ");
for(it_v = vector_begin(pvec_v2b);
    !iterator_equal(it_v, vector_end(pvec_v2b));
    it_v = iterator_next(it_v))
{
    printf("%d ", *(int*)iterator_get_pointer(it_v));
}
printf(")\n");

/* Constructing vectors v3a and v3b with ranges sorted by mod_lesser */
vector_assign(pvec_v3a, pvec_v1a);
vector_assign(pvec_v3b, pvec_v1b);
vector_assign(pvec_v3, pvec_v1);
algo_sort_if(vector_begin(pvec_v3a), vector_end(pvec_v3a), _mod_lesser);
algo_sort_if(vector_begin(pvec_v3b), vector_end(pvec_v3b), _mod_lesser);
```

```
    printf("Original vector v3a with range sorted by the\n"
           "binary predicate mod_lesser is v3a = ( ");
    for(it_v = vector_begin(pvec_v3a);
        !iterator_equal(it_v, vector_end(pvec_v3a));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("Original vector v3b with range sorted by the\n"
           "binary predicate greater than is v3b = ( ");
    for(it_v = vector_begin(pvec_v3b);
        !iterator_equal(it_v, vector_end(pvec_v3b));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /*
     * To combine int a intersection in ascending order
     * with the default binary predicate less
     */
    it_result = algo_set_intersection(vector_begin(pvec_v1a), vector_end(pvec_v1a),
        vector_begin(pvec_v1b), vector_end(pvec_v1b), vector_begin(pvec_v1));
    printf("Intersection of source ranges with default order\n"
           "vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, it_result);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /*
     * To combine int a intersection in ascending order
     * with the specify binary predicate greater
     */
    it_result = algo_set_intersection_if(vector_begin(pvec_v2a),
vector_end(pvec_v2a),
        vector_begin(pvec_v2b), vector_end(pvec_v2b), vector_begin(pvec_v2),
        fun_greater_int);
    printf("Intersection of source ranges with binary predicate greater order\n"
           "vector v2 = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, it_result);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
```

```
    }
    printf(")\n");

    /*
     * To combine int a intersection in ascending order
     * with the user_defined binary predicate mod_lesser
     */
    it_result = algo_set_intersection_if(vector_begin(pvec_v3a),
vector_end(pvec_v3a),
        vector_begin(pvec_v3b), vector_end(pvec_v3b), vector_begin(pvec_v3),
        _mod_lesser);
    printf("Intersection of source ranges with binary predicate mod_lesser order\n"
           "vector v3 = ( ");
    for(it_v = vector_begin(pvec_v3);
        !iterator_equal(it_v, it_result);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1a);
    vector_destroy(pvec_v1b);
    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2a);
    vector_destroy(pvec_v2b);
    vector_destroy(pvec_v2);
    vector_destroy(pvec_v3a);
    vector_destroy(pvec_v3b);
    vector_destroy(pvec_v3);

    return 0;
}

static void _mod_lesser(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;
}
```

- **Output**

```
Original vector v1a with range sorted by the
binary predicate less than is v1a = ( -1 0 1 2 3 )
Original vector v1b with range sorted by the
binary predicate less than is v1b = ( -3 -2 -1 0 1 )
Original vector v2a with range sorted by the
binary predicate greater than is v2a = ( 3 2 1 0 -1 )
Original vector v2b with range sorted by the
binary predicate greater than is v2b = ( 1 0 -1 -2 -3 )
Original vector v3a with range sorted by the
binary predicate mod_lesser is v3a = ( 0 -1 1 2 3 )
```

```
Original vector v3b with range sorted by the
binary predicate greater than is v3b = ( 0 -1 1 -2 -3 )
Intersection of source ranges with default order
vector v1 = ( -1 0 1 )
Intersection of source ranges with binary predicate greater order
vector v2 = ( 1 0 -1 )
Intersection of source ranges with binary predicate mod_lesser order
vector v3 = ( 0 -1 1 2 3 )
```

62. algo_set_symmetric_difference algo_set_symmetric_difference_if

计算两个有序数据区间的对称差集。

```
output_iterator_t algo_set_symmetric_difference(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    input_iterator_t it_first2,
    input_iterator_t it_last2,
    output_iterator_t it_result
);

output_iterator_t algo_set_symmetric_difference_if(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    input_iterator_t it_first2,
    input_iterator_t it_last2,
    output_iterator_t it_result,
    binary_function_t bfun_op
);
```

- **Parameters**

it_first1: 第一个数据区间的开始位置。
it_last1: 第一个数据区间的末尾位置。
it_first2: 第二个数据区间的开始位置。
it_last2: 第二个数据区间的末尾位置。
it_result: 目的数据区间的开始位置。
bfun_op: 指定的比较规则。

- **Remarks**

返回目的数据区间中合并后的有序数据区间的末尾。
目的数据区间要足够大。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_set_symmetric_difference.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
```

```
#include <cstl/calgorithm.h>
#include <cstl/cfunctional.h>

static void _mod_lesser(const void* cpv_first,
    const void* cpv_second, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t* pvec_v1a = create_vector(int);
    vector_t* pvec_v1b = create_vector(int);
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2a = create_vector(int);
    vector_t* pvec_v2b = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_t* pvec_v3a = create_vector(int);
    vector_t* pvec_v3b = create_vector(int);
    vector_t* pvec_v3 = create_vector(int);
    vector_iterator_t it_v;
    vector_iterator_t it_result;
    int i = 0;

    if(pvec_v1a == NULL || pvec_v1b == NULL || pvec_v1 == NULL ||
       pvec_v2a == NULL || pvec_v2b == NULL || pvec_v2 == NULL ||
       pvec_v3a == NULL || pvec_v3b == NULL || pvec_v3 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1a);
    vector_init(pvec_v1b);
    vector_init(pvec_v1);
    vector_init(pvec_v2a);
    vector_init(pvec_v2b);
    vector_init(pvec_v2);
    vector_init(pvec_v3a);
    vector_init(pvec_v3b);
    vector_init(pvec_v3);

    /* Constructing vectors v1a and v1b with default less than ordering */
    for(i = -1; i <= 4; ++i)
    {
        vector_push_back(pvec_v1a, i);
    }
    for(i = -3; i <= 0; ++i)
    {
        vector_push_back(pvec_v1b, i);
    }
    vector_resize(pvec_v1, 12);
    printf("Original vector v1a with range sorted by the\n"
           "binary predicate less than is v1a = ( ");
    for(it_v = vector_begin(pvec_v1a);
```

```
        !iterator_equal(it_v, vector_end(pvec_v1a));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("Original vector v1b with range sorted by the\n"
           "binary predicate less than is v1b = ( ");
    for(it_v = vector_begin(pvec_v1b);
        !iterator_equal(it_v, vector_end(pvec_v1b));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Constructing vectors v2a and v2b with ranges sorted by greater */
    vector_assign(pvec_v2a, pvec_v1a);
    vector_assign(pvec_v2b, pvec_v1b);
    vector_assign(pvec_v2, pvec_v1);
    algo_sort_if(vector_begin(pvec_v2a), vector_end(pvec_v2a), fun_greater_int);
    algo_sort_if(vector_begin(pvec_v2b), vector_end(pvec_v2b), fun_greater_int);
    printf("Original vector v2a with range sorted by the\n"
           "binary predicate greater than is v2a = ( ");
    for(it_v = vector_begin(pvec_v2a);
        !iterator_equal(it_v, vector_end(pvec_v2a));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("Original vector v2b with range sorted by the\n"
           "binary predicate greater than is v2b = ( ");
    for(it_v = vector_begin(pvec_v2b);
        !iterator_equal(it_v, vector_end(pvec_v2b));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Constructing vectors v3a and v3b with ranges sorted by mod_lesser */
    vector_assign(pvec_v3a, pvec_v1a);
    vector_assign(pvec_v3b, pvec_v1b);
    vector_assign(pvec_v3, pvec_v1);
    algo_sort_if(vector_begin(pvec_v3a), vector_end(pvec_v3a), _mod_lesser);
    algo_sort_if(vector_begin(pvec_v3b), vector_end(pvec_v3b), _mod_lesser);
    printf("Original vector v3a with range sorted by the\n"
           "binary predicate mod_lesser is v3a = ( ");
    for(it_v = vector_begin(pvec_v3a);
```

```
        !iterator_equal(it_v, vector_end(pvec_v3a));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("Original vector v3b with range sorted by the\n"
           "binary predicate greater than is v3b = ( ");
    for(it_v = vector_begin(pvec_v3b);
        !iterator_equal(it_v, vector_end(pvec_v3b));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /*
     * To combine int a symmetric_difference in ascending order
     * with the default binary predicate less
     */
    it_result = algo_set_symmetric_difference(vector_begin(pvec_v1a),
vector_end(pvec_v1a),
        vector_begin(pvec_v1b), vector_end(pvec_v1b), vector_begin(pvec_v1));
    printf("Symmetric difference of source ranges with default order\n"
           "vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, it_result);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /*
     * To combine int a symmetric_difference in ascending order
     * with the specify binary predicate greater
     */
    it_result = algo_set_symmetric_difference_if(vector_begin(pvec_v2a),
vector_end(pvec_v2a),
        vector_begin(pvec_v2b), vector_end(pvec_v2b), vector_begin(pvec_v2),
        fun_greater_int);
    printf("Symmetric difference of source ranges with binary predicate greater
order\n"
           "vector v2 = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, it_result);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
```

```
    printf(")\n");

    /*
     * To combine int a symmetric_difference in ascending order
     * with the user_defined binary predicate mod_lesser
     */
    it_result = algo_set_symmetric_difference_if(vector_begin(pvec_v3a),
vector_end(pvec_v3a),
        vector_begin(pvec_v3b), vector_end(pvec_v3b), vector_begin(pvec_v3),
        _mod_lesser);
    printf("Symmetric difference of source ranges with binary predicate mod_lesser
order\n"
           "vector v3 = ( ");
    for(it_v = vector_begin(pvec_v3);
        !iterator_equal(it_v, it_result);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1a);
    vector_destroy(pvec_v1b);
    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2a);
    vector_destroy(pvec_v2b);
    vector_destroy(pvec_v2);
    vector_destroy(pvec_v3a);
    vector_destroy(pvec_v3b);
    vector_destroy(pvec_v3);

    return 0;
}

static void _mod_lesser(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;
}
```

- **Output**

```
Original vector v1a with range sorted by the
binary predicate less than is v1a = ( -1 0 1 2 3 4 )
Original vector v1b with range sorted by the
binary predicate less than is v1b = ( -3 -2 -1 0 )
Original vector v2a with range sorted by the
binary predicate greater than is v2a = ( 4 3 2 1 0 -1 )
Original vector v2b with range sorted by the
binary predicate greater than is v2b = ( 0 -1 -2 -3 )
Original vector v3a with range sorted by the
binary predicate mod_lesser is v3a = ( 0 -1 1 2 3 4 )
Original vector v3b with range sorted by the
```

```
binary predicate greater than is v3b = ( 0 -1 -2 -3 )
Symmetric difference of source ranges with default order
vector v1 = ( -3 -2 1 2 3 4 )
Symmetric difference of source ranges with binary predicate greater order
vector v2 = ( 4 3 2 1 -2 -3 )
Symmetric difference of source ranges with binary predicate mod_lesser order
vector v3 = ( 1 4 )
```

63. algo_set_union algo_set_union_if

计算两个有序数据区间的并集。

```
output_iterator_t algo_set_union(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    input_iterator_t it_first2,
    input_iterator_t it_last2,
    output_iterator_t it_result
);

output_iterator_t algo_set_union_if(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    input_iterator_t it_first2,
    input_iterator_t it_last2,
    output_iterator_t it_result,
    binary_function_t bfun_op
);
```

- **Parameters**

  it_first1: 第一个数据区间的开始位置。
  it_last1: 第一个数据区间的末尾位置。
  it_first2: 第二个数据区间的开始位置。
  it_last2: 第二个数据区间的末尾位置。
  it_result: 目的数据区间的开始位置。
  bfun_op: 指定的比较规则。

- **Remarks**

  返回目的数据区间中合并后的有序数据区间的末尾。
  目的数据区间要足够大。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_set_union.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>
```

```
#include <cstl/cfunctional.h>

static void _mod_lesser(const void* cpv_first,
    const void* cpv_second, void* pv_output);

int main(int argc, char* argv[])
{
    vector_t* pvec_v1a = create_vector(int);
    vector_t* pvec_v1b = create_vector(int);
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2a = create_vector(int);
    vector_t* pvec_v2b = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_t* pvec_v3a = create_vector(int);
    vector_t* pvec_v3b = create_vector(int);
    vector_t* pvec_v3 = create_vector(int);
    vector_iterator_t it_v;
    vector_iterator_t it_result;
    int i = 0;

    if(pvec_v1a == NULL || pvec_v1b == NULL || pvec_v1 == NULL ||
       pvec_v2a == NULL || pvec_v2b == NULL || pvec_v2 == NULL ||
       pvec_v3a == NULL || pvec_v3b == NULL || pvec_v3 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1a);
    vector_init(pvec_v1b);
    vector_init(pvec_v1);
    vector_init(pvec_v2a);
    vector_init(pvec_v2b);
    vector_init(pvec_v2);
    vector_init(pvec_v3a);
    vector_init(pvec_v3b);
    vector_init(pvec_v3);

    /* Constructing vectors v1a and v1b with default less than ordering */
    for(i = -1; i <= 3; ++i)
    {
        vector_push_back(pvec_v1a, i);
    }
    for(i = -3; i <= 1; ++i)
    {
        vector_push_back(pvec_v1b, i);
    }
    vector_resize(pvec_v1, 12);
    printf("Original vector v1a with range sorted by the\n"
           "binary predicate less than is v1a = ( ");
    for(it_v = vector_begin(pvec_v1a);
        !iterator_equal(it_v, vector_end(pvec_v1a));
```

```
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("Original vector v1b with range sorted by the\n"
           "binary predicate less than is v1b = ( ");
    for(it_v = vector_begin(pvec_v1b);
        !iterator_equal(it_v, vector_end(pvec_v1b));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Constructing vectors v2a and v2b with ranges sorted by greater */
    vector_assign(pvec_v2a, pvec_v1a);
    vector_assign(pvec_v2b, pvec_v1b);
    vector_assign(pvec_v2, pvec_v1);
    algo_sort_if(vector_begin(pvec_v2a), vector_end(pvec_v2a), fun_greater_int);
    algo_sort_if(vector_begin(pvec_v2b), vector_end(pvec_v2b), fun_greater_int);
    printf("Original vector v2a with range sorted by the\n"
           "binary predicate greater than is v2a = ( ");
    for(it_v = vector_begin(pvec_v2a);
        !iterator_equal(it_v, vector_end(pvec_v2a));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("Original vector v2b with range sorted by the\n"
           "binary predicate greater than is v2b = ( ");
    for(it_v = vector_begin(pvec_v2b);
        !iterator_equal(it_v, vector_end(pvec_v2b));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Constructing vectors v3a and v3b with ranges sorted by mod_lesser */
    vector_assign(pvec_v3a, pvec_v1a);
    vector_assign(pvec_v3b, pvec_v1b);
    vector_assign(pvec_v3, pvec_v1);
    algo_sort_if(vector_begin(pvec_v3a), vector_end(pvec_v3a), _mod_lesser);
    algo_sort_if(vector_begin(pvec_v3b), vector_end(pvec_v3b), _mod_lesser);
    printf("Original vector v3a with range sorted by the\n"
           "binary predicate mod_lesser is v3a = ( ");
    for(it_v = vector_begin(pvec_v3a);
        !iterator_equal(it_v, vector_end(pvec_v3a));
```

```
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
    printf("Original vector v3b with range sorted by the\n"
           "binary predicate greater than is v3b = ( ");
    for(it_v = vector_begin(pvec_v3b);
        !iterator_equal(it_v, vector_end(pvec_v3b));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* To combine int a union in ascending order with the default binary predicate
less */
    it_result = algo_set_union(vector_begin(pvec_v1a), vector_end(pvec_v1a),
        vector_begin(pvec_v1b), vector_end(pvec_v1b), vector_begin(pvec_v1));
    printf("Union of source ranges with default order\n"
           "vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, it_result);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* To combine int a union in ascending order with the specify binary predicate
greater */
    it_result = algo_set_union_if(vector_begin(pvec_v2a), vector_end(pvec_v2a),
        vector_begin(pvec_v2b), vector_end(pvec_v2b), vector_begin(pvec_v2),
        fun_greater_int);
    printf("Union of source ranges with binary predicate greater order\n"
           "vector v2 = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, it_result);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* To combine int a union in ascending order with the user_defined binary
predicate mod_lesser */
    it_result = algo_set_union_if(vector_begin(pvec_v3a), vector_end(pvec_v3a),
        vector_begin(pvec_v3b), vector_end(pvec_v3b), vector_begin(pvec_v3),
        _mod_lesser);
    printf("Union of source ranges with binary predicate mod_lesser order\n"
```

```
            "vector v3 = ( ");
    for(it_v = vector_begin(pvec_v3);
        !iterator_equal(it_v, it_result);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1a);
    vector_destroy(pvec_v1b);
    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2a);
    vector_destroy(pvec_v2b);
    vector_destroy(pvec_v2);
    vector_destroy(pvec_v3a);
    vector_destroy(pvec_v3b);
    vector_destroy(pvec_v3);

    return 0;
}

static void _mod_lesser(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;
}
```

- **Output**

```
Original vector v1a with range sorted by the
binary predicate less than is v1a = ( -1 0 1 2 3 )
Original vector v1b with range sorted by the
binary predicate less than is v1b = ( -3 -2 -1 0 1 )
Original vector v2a with range sorted by the
binary predicate greater than is v2a = ( 3 2 1 0 -1 )
Original vector v2b with range sorted by the
binary predicate greater than is v2b = ( 1 0 -1 -2 -3 )
Original vector v3a with range sorted by the
binary predicate mod_lesser is v3a = ( 0 -1 1 2 3 )
Original vector v3b with range sorted by the
binary predicate greater than is v3b = ( 0 -1 1 -2 -3 )
Union of source ranges with default order
vector v1 = ( -3 -2 -1 0 1 2 3 )
Union of source ranges with binary predicate greater order
vector v2 = ( 3 2 1 0 -1 -2 -3 )
Union of source ranges with binary predicate mod_lesser order
vector v3 = ( 0 -1 1 2 3 )
```

64. algo_sort　algo_sort_if

将数据区间中的数据按照指定比较规则排序。

```
void algo_sort(
```

```
    random_access_iterator_t it_first,
    random_access_iterator_t it_last
);

void algo_sort_if(
    random_access_iterator_t it_first,
    random_access_iterator_t it_last,
    binary_function_t bfun_op
);
```

- **Parameters**
  it_first: 数据区间的开始位置。
  it_last: 数据区间的末尾位置。
  bfun_op: 指定的比较规则。

- **Requirements**
  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_sort.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_v1 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

    if(pvec_v1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);

    for(i = 0; i <= 5; ++i)
    {
        vector_push_back(pvec_v1, i * 2);
    }
    for(i = 0; i <= 5; ++i)
    {
        vector_push_back(pvec_v1, i * 2 + 1);
    }

    printf("Original vector v1 = ( ");
```

```
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    algo_sort(vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("Sorted vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* To sort in descending order. */
    algo_sort_if(vector_begin(pvec_v1), vector_end(pvec_v1), fun_greater_int);
    printf("Resorted (greater) vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
Original vector v1 = ( 0 2 4 6 8 10 1 3 5 7 9 11 )
Sorted vector v1 = ( 0 1 2 3 4 5 6 7 8 9 10 11 )
Resorted (greater) vector v1 = ( 11 10 9 8 7 6 5 4 3 2 1 0 )
```

65. algo_sort_heap algo_sort_heap_if

将一个堆转换成有序的数据区间。

```
void algo_sort_heap(
    random_access_iterator_t it_first,
    random_access_iterator_t it_last
);

void algo_sort_heap_if(
    random_access_iterator_t it_first,
    random_access_iterator_t it_last,
```

```
    binary_function_t bfun_op
);
```

- **Parameters**

  it_first: 数据区间的开始位置。

  it_last: 数据区间的末尾位置。

  bfun_op: 指定的比较规则。

- **Remarks**

  排序前的数据区间必须是符合指定规则的堆。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_sort_heap.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_v1 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

    if(pvec_v1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);

    for(i = 1; i <= 9; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
    algo_random_shuffle(vector_begin(pvec_v1), vector_end(pvec_v1));

    printf("Vector v1 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
```

```
    /* Make v1 a heap with default less than ordering */
    algo_make_heap(vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("The heap version of vector v1 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Sort heap v1 with default less-than ordering */
    algo_sort_heap(vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("The heap v1 becomes the sorted range:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Make v1 a heap with greater than ordering */
    algo_make_heap_if(vector_begin(pvec_v1), vector_end(pvec_v1), fun_greater_int);
    printf("The greater-than heaped version of vector v1 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    algo_sort_heap_if(vector_begin(pvec_v1), vector_end(pvec_v1), fun_greater_int);
    printf("The greater-than heap v1 becomes the sorted range:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
Vector v1 is:
( 6 3 9 7 2 1 8 5 4 )
The heap version of vector v1 is:
( 9 7 8 5 2 1 6 3 4 )
The heap v1 becomes the sorted range:
( 1 2 3 4 5 6 7 8 9 )
The greater-than heaped version of vector v1 is:
( 1 2 3 4 5 6 7 8 9 )
The greater-than heap v1 becomes the sorted range:
( 9 8 7 6 5 4 3 2 1 )
```

66. algo_stable_sort algo_stable_sort_if

将数据区间按照指定规则进行稳定排序。

```
void algo_stable_sort(
    random_access_iterator_t it_first,
    random_access_iterator_t it_last
);

void algo_stable_sort_if(
    random_access_iterator_t it_first,
    random_access_iterator_t it_last,
    binary_function_t bfun_op
);
```

- **Parameters**

  it_first: 数据区间的开始位置。
  it_last: 数据区间的末尾位置。
  bfun_op: 指定的比较规则。

- **Remarks**

  这个算法保证了相等的数据在排序后顺序保持不便，但是这个算法在排序效率上不如非稳定排序算法。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_stable_sort.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_v1 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;
```

```
    if(pvec_v1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);

    for(i = 0; i <= 5; ++i)
    {
        vector_push_back(pvec_v1, i * 2);
    }
    for(i = 0; i <= 5; ++i)
    {
        vector_push_back(pvec_v1, i * 2);
    }

    printf("Original vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    algo_stable_sort(vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("Sorted vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* To sort in descending order. */
    algo_stable_sort_if(vector_begin(pvec_v1),
        vector_end(pvec_v1), fun_greater_int);
    printf("Resorted (greater) vector v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);

    return 0;
}
```

● **Output**

```
Original vector v1 = ( 0 2 4 6 8 10 0 2 4 6 8 10 )
Sorted vector v1 = ( 0 0 2 2 4 4 6 6 8 8 10 10 )
Resorted (greater) vector v1 = ( 10 10 8 8 6 6 4 4 2 2 0 0 )
```

67. algo_stable_partition

按照指定规则将数据区间分为满足指定规则的数据和不满足指定规则的数据两部分。

```
forward_iterator_t algo_stable_partition(
    forward_iterator_t it_first,
    forward_iterator_t it_last,
    unary_function_t bfun_op
);
```

● **Parameters**

it_first: 数据区间的开始位置。
it_last: 数据区间的末尾位置。
ufun_op: 指定的比较规则。

● **Remarks**

返回满足指定规则的数据区间的末尾。

算法根据指定的规则将满足指定规则的数据放在数据区间的前部，不满足指定规则的数据放在数据区间的后部。这个算法实现的是稳定的划分。

● **Requirements**

头文件 <cstl/calgorithm.h>。

● **Example**

```
/*
 * algo_stable_partition.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>

static void _greater_5(const void* cpv_input, void* pv_output)
{
    *(bool_t*)pv_output = *(int*)cpv_input > 5 ? true : false;
}

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    vector_iterator_t it_result;
    int i = 0;

    if(pvec_v1 == NULL)
    {
```

```
        return -1;
    }

    vector_init(pvec_v1);

    for(i = 0; i <= 10; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
    for(i = 0; i <= 4; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
    algo_random_shuffle(vector_begin(pvec_v1), vector_end(pvec_v1));

    printf("Vector v1 is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Partition the range with predicate greater5 */
    it_result = algo_stable_partition(
        vector_begin(pvec_v1), vector_end(pvec_v1), _greater_5);

    printf("The partitioned set of elements in v1 is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    printf("The first element in v1 to fail to satisfy "
           "the predicate greater 5 is : %d\n",
           *(int*)iterator_get_pointer(it_result));

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
Vector v1 is ( 2 9 4 0 8 3 6 1 7 2 4 5 10 1 3 0 )
The partitioned set of elements in v1 is ( 9 8 6 7 10 2 4 0 3 1 2 4 5 1 3 0 )
The first element in v1 to fail to satisfy the predicate greater 5 is : 2
```

68. algo_swap

交换两个迭代器指向的数据的内容。

```
void algo_swap(
    forward_iterator_t it_first,
    forward_iterator_t it_second
);
```

- **Parameters**

  it_first: 第一个数据的迭代器。
  it_second: 第二个数据的迭代器。

- **Remarks**

  这个算法和 algo_iter_swap 算法功能相同。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

  请参考 algo_iter_swap 算法。

69. algo_swap_ranges

交换两个数据区间中的数据。

```
forward_iterator_t algo_swap_ranges(
    forward_iterator_t it_first1,
    forward_iterator_t it_last1,
    forward_iterator_t it_first2
);
```

- **Parameters**

  it_first1: 第一个数据区间的开始位置。
  it_last1: 第一个数据区间的末尾位置。
  it_first2: 第二个数据区间的开始位置。

- **Remarks**

  返回交换数据后第二个数据区间的末尾。
  第二个数据区间要至少和第一个数据区间一样大。

- **Requirements**

  头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_swap_ranges.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/cdeque.h>
```

```
#include <cstl/calgorithm.h>

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    deque_t* pdeq_q1 = create_deque(int);
    deque_iterator_t it_q;
    int i = 0;

    if(pvec_v1 == NULL || pdeq_q1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    deque_init(pdeq_q1);

    for(i = 0; i <= 5; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
    for(i = 4; i <= 9; ++i)
    {
        deque_push_back(pdeq_q1, 6);
    }

    printf("Vector v1 is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    printf("Deque q1 is ( ");
    for(it_q = deque_begin(pdeq_q1);
        !iterator_equal(it_q, deque_end(pdeq_q1));
        it_q = iterator_next(it_q))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_q));
    }
    printf(")\n");

    algo_swap_ranges(vector_begin(pvec_v1),
        vector_end(pvec_v1), deque_begin(pdeq_q1));

    printf("After the swap_range, vector v1 is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
```

```
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    printf("After the swap_range, deque q1 is ( ");
    for(it_q = deque_begin(pdeq_q1);
        !iterator_equal(it_q, deque_end(pdeq_q1));
        it_q = iterator_next(it_q))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_q));
    }
    printf(")\n");

    vector_destroy(pvec_v1);
    deque_destroy(pdeq_q1);

    return 0;
}
```

- **Output**

```
Vector v1 is ( 0 1 2 3 4 5 )
Deque q1 is ( 6 6 6 6 6 6 )
After the swap_range, vector v1 is ( 6 6 6 6 6 6 )
After the swap_range, deque q1 is ( 0 1 2 3 4 5 )
```

70. algo_transform algo_transform_binary

通过指定的规则将源数据区间中的数据转换到目的数据区间。

```
output_iterator_t algo_transform(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    output_iterator_t it_result,
    unary_function_t ufun_op
);

output_iterator_t algo_transform_binary(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    input_iterator_t it_first2,
    output_iterator_t it_result,
    binary_function_t bfun_op
);
```

- **Parameters**

it_first1: 第一个数据区间的开始位置。
it_last1: 第一个数据区间的末尾位置。
it_first2: 第二个数据区间的开始位置。
it_result: 目的数据区间的开始位置。
ufun_op: 一元转换函数。

bfun_op: 二元转换函数。

- **Remarks**

返回目的数据区间中转换后的数据的末尾。

第一个算法是将一个源数据区间通过转换函数转换到目的数据区间，第二个是将两个源数据区间通过转换函数转换到目的数据区间。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_transform.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>
#include <cstl/cfunctional.h>

static void _mult_valu_2(const void* cpv_input, void* pv_output)
{
    *(int*)pv_output = *(int*)cpv_input * 2;
}

static void _mult_valu_5(const void* cpv_input, void* pv_output)
{
    *(int*)pv_output = *(int*)cpv_input * 5;
}

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_t* pvec_v3 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

    if(pvec_v1 == NULL || pvec_v2 == NULL || pvec_v3 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init_n(pvec_v2, 7);
    vector_init_n(pvec_v3, 7);

    for(i = -4; i <= 2; ++i)
    {
        vector_push_back(pvec_v1, i);
    }

    printf("Original vector v1 = ( ");
```

```
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Modifying the vector v1 in place */
    algo_transform(vector_begin(pvec_v1), vector_end(pvec_v1),
        vector_begin(pvec_v1), _mult_valu_2);
    printf("The elements of the vector v1 multiplied by 2 in place is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Using transform to multiply each element by a factor of 5 */
    algo_transform(vector_begin(pvec_v1), vector_end(pvec_v1),
        vector_begin(pvec_v2), _mult_valu_5);
    printf("Mutiplying the elements of the vector v2 by factor 5 is ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /*
     * The second version of transform used to multiply the
     * elements of the vectors v1 and v2 pairwise
     */
    algo_transform_binary(vector_begin(pvec_v1), vector_end(pvec_v1),
        vector_begin(pvec_v2), vector_begin(pvec_v3), fun_multiplies_int);
    printf("Mutiplying the elements of the vector v3 is ( ");
    for(it_v = vector_begin(pvec_v3);
        !iterator_equal(it_v, vector_end(pvec_v3));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);
    vector_destroy(pvec_v3);
```

```
    return 0;
}
```

● **Output**

```
Original vector v1 = ( -4 -3 -2 -1 0 1 2 )
The elements of the vector v1 multiplied by 2 in place is ( -8 -6 -4 -2 0 2 4 )
Mutiplying the elements of the vector v2 by factor 5 is ( -40 -30 -20 -10 0 10 20 )
Mutiplying the elements of the vector v3 is ( 320 180 80 20 0 20 80 )
```

71. algo_unique　algo_unique_if

将数据区间中相邻且符合条件的重复数据移除。

```
forward_iterator_t algo_unique(
    forward_iterator_t it_first,
    forward_iterator_t it_last
);

forward_iterator_t algo_unique_if(
    forward_iterator_t it_first,
    forward_iterator_t it_last,
    binary_function_t bfun_op
);
```

● **Parameters**

it_first: 数据区间的开始位置。
it_last: 数据区间的末尾位置。
bfun_op: 指定的比较规则函数。

● **Remarks**

返回数据区间中处理后的数据的末尾。
两个算法是将数据区间中符合条件的重复的数据去掉，保留数据的唯一性。

● **Requirements**

头文件 <cstl/calgorithm.h>。

● **Example**

```
/*
 * algo_unique.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <stdlib.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>
#include <cstl/cfunctional.h>

static void _mod_equal(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[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    vector_iterator_t it_end;
    int i = 0;

    if(pvec_v1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);

    for(i = 0; i <= 3; ++i)
    {
        vector_push_back(pvec_v1, 5);
        vector_push_back(pvec_v1, -5);
    }
    for(i = 0; i <= 3; ++i)
    {
        vector_push_back(pvec_v1, 4);
    }
    vector_push_back(pvec_v1, 7);

    printf("Vector v1 is ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    it_end = algo_unique(vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("Removing adjacent duplicates from vector v1 gives\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, it_end);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    it_end = algo_unique_if(vector_begin(pvec_v1), it_end, _mod_equal);
    printf("Removing adjacent duplicates from vector v1 "
           "under the mod_equal gives\n( ");
    for(it_v = vector_begin(pvec_v1);
```

```
        !iterator_equal(it_v, it_end);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    it_end = algo_unique_if(vector_begin(pvec_v1), it_end, fun_greater_int);
    printf("Removing adjacent duplicates from vector v1 "
           "under the greater gives\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, it_end);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
Vector v1 is ( 5 -5 5 -5 5 -5 5 -5 4 4 4 4 7 )
Removing adjacent duplicates from vector v1 gives
( 5 -5 5 -5 5 -5 5 -5 4 7 )
Removing adjacent duplicates from vector v1 under the mod_equal gives
( 5 4 7 )
Removing adjacent duplicates from vector v1 under the greater gives
( 5 7 )
```

72. algo_unique_copy　algo_unique_copy_if

将数据区间中符合指定规则的重复数据移除，将结果拷贝到目的数据区间。

```
output_iterator_t algo_unique_copy(
    input_iterator_t it_first,
    input_iterator_t it_last,
    output_iterator_t it_result
);

output_iterator_t algo_unique_copy_if(
    input_iterator_t it_first,
    input_iterator_t it_last,
    output_iterator_t it_result,
    binary_function_t bfun_op
);
```

- **Parameters**

it_first:　　数据区间的开始位置。
it_last:　　数据区间的末尾位置。

it_result: 目的数据区间的开始位置。
bfun_op: 指定的比较规则函数。

- **Remarks**

返回目的数据区间中拷贝数据的末尾。

两个算法是将数据区间中符合条件的重复的数据去掉，保留数据的唯一性，将结果拷贝到目的数据区间。要保证目的数据区间足够大。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_unique_copy.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <stdlib.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>
#include <cstl/cfunctional.h>

static void _mod_equal(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[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_iterator_t it_v;
    vector_iterator_t it_end;
    int i = 0;

    if(pvec_v1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);

    for(i = 0; i <= 1; ++i)
    {
        vector_push_back(pvec_v1, 5);
        vector_push_back(pvec_v1, -5);
    }
    for(i = 0; i <= 2; ++i)
    {
        vector_push_back(pvec_v1, 4);
    }
```

```
    vector_push_back(pvec_v1, 7);
    for(i = 0; i <= 5; ++i)
    {
        vector_push_back(pvec_v1, 10);
    }

    printf("Vector v1 is\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    it_end = algo_unique_copy(vector_begin(pvec_v1),
        iterator_next_n(vector_begin(pvec_v1), 8),
        iterator_next_n(vector_begin(pvec_v1), 8));
    printf("Copying the first half of the vector to the second half\n"
           "while removing adjacent duplicates gives\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, it_end);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    for(i = 0; i <= 7; ++i)
    {
        vector_push_back(pvec_v1, 10);
    }

    it_end = algo_unique_copy_if(vector_begin(pvec_v1),
        iterator_next_n(vector_begin(pvec_v1), 14),
        iterator_next_n(vector_begin(pvec_v1), 14), _mod_equal);
    printf("Copying the first half of the vector to the second half\n"
           "while removing adjacent duplicates under mod_equal gives\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, it_end);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);

    return 0;
}
```

- **Output**

```
Vector v1 is
( 5 -5 5 -5 4 4 4 7 10 10 10 10 10 10 )
Copying the first half of the vector to the second half
while removing adjacent duplicates gives
( 5 -5 5 -5 4 4 4 7 5 -5 5 -5 4 7 )
Copying the first half of the vector to the second half
while removing adjacent duplicates under mod_equal gives
( 5 -5 5 -5 4 4 4 7 5 -5 5 -5 4 7 5 4 7 5 4 7 )
```

73. algo_upper_bound algo_upper_bound_if

返回数据区间中第一个大于指定数据的位置。

```
forward_iterator_t algo_upper_bound(
    forward_iterator_t it_first,
    forward_iterator_t it_last,
    element
);

forward_iterator_t algo_upper_bound_if(
    forward_iterator_t it_first,
    forward_iterator_t it_last,
    element
    binary_function_t bfun_op
);
```

- **Parameters**

it_first: 数据区间的开始位置。
it_last: 数据区间的末尾位置。
element: 指定的数据。
bfun_op: 指定的比较规则函数。

- **Remarks**

返回数据区间中第一个大于指定数据的位置，如果没有就返回数据区间的末尾。

- **Requirements**

头文件 <cstl/calgorithm.h>。

- **Example**

```
/*
 * algo_upper_bound.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/calgorithm.h>
#include <cstl/cfunctional.h>

/* Return whether modulus of elem1 is less than modulus of elem2 */
static void _mod_lesser(const void* cpv_first,
    const void* cpv_second, void* pv_output);
```

```
int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_t* pvec_v3 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

    if(pvec_v1 == NULL || pvec_v2 == NULL || pvec_v3 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init(pvec_v2);
    vector_init(pvec_v3);

    for(i = -1; i <= 4; ++i)
    {
        vector_push_back(pvec_v1, i);
    }
    for(i = -3; i <= 0; ++i)
    {
        vector_push_back(pvec_v1, i);
    }

    algo_sort(vector_begin(pvec_v1), vector_end(pvec_v1));
    printf("Original vector v1 with range sorted by the "
           "binary predicate less than is v1 = ( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_assign(pvec_v2, pvec_v1);
    algo_sort_if(vector_begin(pvec_v2), vector_end(pvec_v2), fun_greater_int);
    printf("Original vector v2 with range sorted by the "
           "binary predicate greater than is v2 = ( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_assign(pvec_v3, pvec_v1);
    algo_sort_if(vector_begin(pvec_v3), vector_end(pvec_v3), _mod_lesser);
```

```
    printf("Original vector v3 with range sorted by the "
           "binary predicate greater than is v3 = ( ");
    for(it_v = vector_begin(pvec_v3);
        !iterator_equal(it_v, vector_end(pvec_v3));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* upper_bound of 3 in v1 with default binary predicate less than */
    it_v = algo_upper_bound(vector_begin(pvec_v1), vector_end(pvec_v1), 3);
    printf("The upper_bound in v1 for the element with a value of 3 is: %d.\n",
        *(int*)iterator_get_pointer(it_v));

    /* upper_bound of 3 in v2 with the binary predicate greater than */
    it_v = algo_upper_bound_if(vector_begin(pvec_v2),
        vector_end(pvec_v2), 3, fun_greater_int);
    printf("The upper_bound in v2 for the element with a value of 3 is: %d.\n",
        *(int*)iterator_get_pointer(it_v));

    /* upper_bound of 3 in v3 with the binary predicate _mod_lesser */
    it_v = algo_upper_bound_if(vector_begin(pvec_v3),
        vector_end(pvec_v3), 3, _mod_lesser);
    printf("The upper_bound in v3 for the element with a value of 3 is: %d.\n",
        *(int*)iterator_get_pointer(it_v));

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);
    vector_destroy(pvec_v3);

    return 0;
}

static void _mod_lesser(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;
}
```

- **Output**

```
Original vector v1 with range sorted by the binary predicate less than is v1 = ( -3
-2 -1 -1 0 0 1 2 3 4 )
Original vector v2 with range sorted by the binary predicate greater than is v2 =
( 4 3 2 1 0 0 -1 -1 -2 -3 )
Original vector v3 with range sorted by the binary predicate greater than is v3 =
( 0 0 -1 -1 1 -2 2 -3 3 4 )
The upper_bound in v1 for the element with a value of 3 is: 4.
The upper_bound in v2 for the element with a value of 3 is: 2.
The upper_bound in v3 for the element with a value of 3 is: 4.
```

第二节 数值算法

数值算法的主要目的是处理容器中数值类型的数据的计算，所以普通的数值算法只能应用在数据类型是 C 内建类型(除了 C 字符串类型)的容器上，要想在保存 libcstl 内建类型和用户自定义类型的容器上使用数值算法，就必须使用带有_if 后缀的算法版本，同时提供自定义的函数。数值算法都在<cstl/cnumeric.h>中声明。

- **Algorithm Functions**

algo_accumulate	计算数据区间中所有数据的和。
algo_accumulate_if	对数据区间中所有的数据进行指定的计算。
algo_adjacent_difference	计算数据区间中相邻数据的差。
algo_adjacent_difference_if	对数据区间中相邻的数据执行指定的计算。
algo_inner_product	计算两个数据区间的内积。
algo_inner_product_if	根据指定的算法计算两个数据区间的内积。
algo_iota	根据初始值对数据区间中的数据进行填充。
algo_partial_sum	计算数据区间的局部和。
algo_partial_sum_if	根据指定的算法计算数据区间的局部和。
algo_power	计算数据的幂。
algo_power_if	根据指定的算法计算数据的幂。

1. algo_accumulate algo_accumulate_if

计算数据区间中数据的和。

```
void algo_accumulate(
    input_iterator_t it_first,
    input_iterator_t it_last,
    element,
    void* pv_output
);

void algo_accumulate_if(
    input_iterator_t it_first,
    input_iterator_t it_last,
    element,
    binary_function_t bfun_op,
    void* pv_output
);
```

- **Parameters**

it_first: 数据区间的开始位置。
it_last: 数据区间的末尾位置。
element: 初始化值。
bfun_op: 指定的算法函数。
pv_output: 指向输出值的指针。

- **Remarks**

algo_accumulate 的计算过程是:
*pv_output = element + *it_first + *(it_first + 1) + ... + *(it_last – 1)
algo_accumulate_if 的计算过程是:
*pv_output = element bfun_op *it_first bfun_op *(it_first + 1) bfun_op ... bfun_op *(it_last - 1)

- **Requirements**

  头文件 <cstl/cnumeric.h>。

- **Example**

```
/*
 * algo_accumulate.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_vec1 = create_vector(int);
    vector_t* pvec_vec2 = create_vector(int);
    vector_t* pvec_vec3 = create_vector(int);
    vector_t* pvec_vec4 = create_vector(int);
    vector_iterator_t it_vec;
    int n_sum = 0;
    int n_product = 0;
    int i = 0;

    if(pvec_vec1 == NULL || pvec_vec2 == NULL ||
       pvec_vec3 == NULL || pvec_vec4 == NULL)
    {
        return -1;
    }

    /* The first function for the accumulated sums */
    vector_init(pvec_vec1);
    for(i = 1; i < 21; ++i)
    {
        vector_push_back(pvec_vec1, i);
    }
    vector_init_n(pvec_vec2, vector_size(pvec_vec1));

    printf("The original vector vec1 is:\n( ");
    for(it_vec = vector_begin(pvec_vec1);
        !iterator_equal(it_vec, vector_end(pvec_vec1));
        it_vec = iterator_next(it_vec))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_vec));
    }
    printf(")\n");
```

```
algo_accumulate(vector_begin(pvec_vec1), vector_end(pvec_vec1), 0, &n_sum);
printf("The sum of the integers from 1 to 20 is: %d\n", n_sum);

/* Construction a vector of partial sums */
for(it_vec = vector_begin(pvec_vec1), i = 0;
    !iterator_equal(it_vec, vector_end(pvec_vec1));
    it_vec = iterator_next(it_vec), ++i)
{
    algo_accumulate(vector_begin(pvec_vec1), iterator_next(it_vec),
        0, vector_at(pvec_vec2, i));
}

printf("The vector of partial sums is:\n( ");
for(it_vec = vector_begin(pvec_vec2);
    !iterator_equal(it_vec, vector_end(pvec_vec2));
    it_vec = iterator_next(it_vec))
{
    printf("%d ", *(int*)iterator_get_pointer(it_vec));
}
printf(")\n");

vector_destroy(pvec_vec1);
vector_destroy(pvec_vec2);

/* The second function for the accumulated product */
vector_init(pvec_vec3);
for(i = 1; i < 11; ++i)
{
    vector_push_back(pvec_vec3, i);
}
vector_init_n(pvec_vec4, vector_size(pvec_vec3));

printf("The original vector vec3 is:\n( ");
for(it_vec = vector_begin(pvec_vec3);
    !iterator_equal(it_vec, vector_end(pvec_vec3));
    it_vec = iterator_next(it_vec))
{
    printf("%d ", *(int*)iterator_get_pointer(it_vec));
}
printf(")\n");

algo_accumulate_if(vector_begin(pvec_vec3), vector_end(pvec_vec3),
    1, fun_multiplies_int, &n_product);
printf("The product of the integers from 1 to 10 is: %d.\n", n_product);

/* Constructing a vector of partial products */
for(it_vec = vector_begin(pvec_vec3), i = 0;
    !iterator_equal(it_vec, vector_end(pvec_vec3));
    it_vec = iterator_next(it_vec), ++i)
{
```

```
        algo_accumulate_if(vector_begin(pvec_vec3), iterator_next(it_vec),
            1, fun_multiplies_int, vector_at(pvec_vec4, i));
    }

    printf("The vector of partial products is:\n( ");
    for(it_vec = vector_begin(pvec_vec4);
        !iterator_equal(it_vec, vector_end(pvec_vec4));
        it_vec = iterator_next(it_vec))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_vec));
    }
    printf(")\n");

    vector_destroy(pvec_vec3);
    vector_destroy(pvec_vec4);

    return 0;
}
```

- **Output**

```
The original vector vec1 is:
( 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 )
The sum of the integers from 1 to 20 is: 210
The vector of partial sums is:
( 1 3 6 10 15 21 28 36 45 55 66 78 91 105 120 136 153 171 190 210 )
The original vector vec3 is:
( 1 2 3 4 5 6 7 8 9 10 )
The product of the integers from 1 to 10 is: 3628800.
The vector of partial products is:
( 1 2 6 24 120 720 5040 40320 362880 3628800 )
```

2. algo_adjacent_difference algo_adjacent_difference_if

计算相邻数据的差。

```
output_iterator_t algo_adjacent_difference(
    input_iterator_t it_first,
    input_iterator_t it_last,
    output_iterator_t it_result
);

output_iterator_t algo_adjacent_difference_if(
    input_iterator_t it_first,
    input_iterator_t it_last,
    output_iterator_t it_result,
    binary_function_t bfun_op
);
```

- **Parameters**

it_first: 数据区间的开始位置。
it_last: 数据区间的末尾位置。
it_result: 目的数据区间的开始位置。
bfun_op: 指定的算法函数。

- **Remarks**

  返回目的数据区间拷贝的数据的末尾。
  要保证目的数据区间至少和源数据区间一样大。
  algo_adjacent_difference 执行后目的数据区间的数据:
  *it_first, *(it_first + 1) - *it_first, *(it_first + 2) - *(it_first + 1), ..., *(it_last – 1) - *(it_last - 2)
  algo_adjacent_difference_if 执行后目的数据区间的数据:
  *it_first, *(it_first + 1) bfun_op *it_first, *(it_first + 2) bfun_op *(it_first + 1), ..., *(it_last – 1) bfun_op *(it_last - 2)

- **Requirements**

  头文件 <cstl/cnumeric.h>。

- **Example**

```
/*
 * algo_adjacent_difference.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/clist.h>
#include <cstl/cnumeric.h>
#include <cstl/cfunctional.h>

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_iterator_t it_v;
    list_t* plist_l1 = create_list(int);
    list_iterator_t it_l;
    iterator_t it_end;
    int i = 0;

    if(pvec_v1 == NULL || pvec_v2 == NULL || plist_l1 == NULL)
    {
        return -1;
    }

    vector_init_n(pvec_v1, 10);
    vector_init_n(pvec_v2, 10);
    list_init(plist_l1);

    for(i = 1; i <= 10; ++i)
    {
        list_push_back(plist_l1, i * i);
    }

    printf("The input list l1 is\n( ");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, list_end(plist_l1));
        it_l = iterator_next(it_l))
    {
```

```
        printf("%d ", *(int*)iterator_get_pointer(it_l));
    }
    printf(")\n");

    /*
     * The first function for the adjacent_difference of elements
     * in a list output to a vector.
     */
    it_end = algo_adjacent_difference(list_begin(plist_l1),
        list_end(plist_l1), vector_begin(pvec_v1));

    printf("Output vector containing adjacent_differences is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, it_end);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /*
     * The second function for used to compute the adjacent
     * products of the elements in a list.
     */
    it_end = algo_adjacent_difference_if(list_begin(plist_l1), list_end(plist_l1),
        vector_begin(pvec_v2), fun_multiplies_int);

    printf("Output vector with the adjacent products is:\n( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, it_end);
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    /* Computation of adjacent_differences in place */
    it_end = algo_adjacent_difference(list_begin(plist_l1),
        list_end(plist_l1), list_begin(plist_l1));

    printf("In place output adjacent_differences in list l1 is:\n( ");
    for(it_l = list_begin(plist_l1);
        !iterator_equal(it_l, it_end);
        it_l = iterator_next(it_l))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_l));
    }
    printf(")\n");

    vector_destroy(pvec_v1);
```

```
    vector_destroy(pvec_v2);
    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
The input list l1 is
( 1 4 9 16 25 36 49 64 81 100 )
Output vector containing adjacent_differences is:
( 1 3 5 7 9 11 13 15 17 19 )
Output vector with the adjacent products is:
( 1 4 36 144 400 900 1764 3136 5184 8100 )
In place output adjacent_differences in list l1 is:
( 1 3 5 7 9 11 13 15 17 19 )
```

3. algo_inner_product　algo_inner_product_if

计算两个数据区间的内积。

```
void algo_inner_product(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    input_iterator_t it_first2,
    element,
    void* pv_output
);

void algo_inner_product_if(
    input_iterator_t it_first1,
    input_iterator_t it_last1,
    input_iterator_t it_first2,
    element,
    binary_function_t it_binary_op1,
    binary_function_t it_binary_op2,
    void* pv_output
);
```

- **Parameters**

it_first1:　第一个数据区间的开始位置。
it_last1:　第一个数据区间的末尾位置。
it_first2:　第二个数据区间的开始位置。
element:　初始化值。
bfun_op1:　指定的算法函数。
bfun_op2:　指定的算法函数。
pv_output:　指向输出值的指针。

- **Remarks**

要保证第二个数据区间至少和第一个数据区间一样大。

algo_inner_product 执行的过程:

*pv_output = element + (*it_first1 * *it_first2) + [*(it_first1 + 1) * *(it_first2 + 1)] + ... + [*(it_last1 - 1) * *(it_first2 + it_last – it_first - 1)]

algo_inner_product_if 执行后目的数据区间的数据:

*pv_output = element bfun_op1 (*it_first1 bfun_op2 *it_first2) bfun_op1 [*(it_first1 + 1) bfun_op2 *(it_first2 + 1)] bfun_op1 ... bfun_op1 [*(it_last1 - 1) bfun_op2 *(it_first2 + it_last – it_first - 1)]

- **Requirements**

  头文件 <cstl/cnumeric.h>。

- **Example**

```
/*
 * algo_inner_product.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/clist.h>
#include <cstl/cnumeric.h>
#include <cstl/cfunctional.h>

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_t* pvec_v3 = create_vector(int);
    vector_iterator_t it_v;
    list_t* plist_l1 = create_list(int);
    list_iterator_t it_l;
    int i = 0;
    int n_result = 0;

    if(pvec_v1 == NULL || pvec_v2 == NULL || pvec_v3 == NULL || plist_l1 == NULL)
    {
        return -1;
    }

    vector_init(pvec_v1);
    vector_init_n(pvec_v2, 7);
    vector_init_n(pvec_v3, 7);
    list_init(plist_l1);

    for(i = 1; i <= 7; ++i)
    {
        vector_push_back(pvec_v1, i);
        list_push_back(plist_l1, i);
    }

    printf("The original vector v1 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
```

```
printf(")\n");
printf("The original list l1 is:\n( ");
for(it_l = list_begin(plist_l1);
    !iterator_equal(it_l, list_end(plist_l1));
    it_l = iterator_next(it_l))
{
    printf("%d ", *(int*)iterator_get_pointer(it_l));
}
printf(")\n");

/* The first function for the inner product */
algo_inner_product(vector_begin(pvec_v1), vector_end(pvec_v1),
    list_begin(plist_l1), 0, &n_result);

printf("The inner_product of the vector v1 and the list l1 is: %d.\n",
    n_result);

/* Constructing a vector of partial inner_products between v1 and l1 */
for(it_v = vector_begin(pvec_v1), i = 0;
    !iterator_equal(it_v, vector_end(pvec_v1));
    it_v = iterator_next(it_v), ++i)
{
    algo_inner_product(vector_begin(pvec_v1), iterator_next(it_v),
        list_begin(plist_l1), 0, vector_at(pvec_v2, i));
}

printf("The vector of partial inner_products between v1 and l1 is:\n( ");
for(it_v = vector_begin(pvec_v2);
    !iterator_equal(it_v, vector_end(pvec_v2));
    it_v = iterator_next(it_v))
{
    printf("%d ", *(int*)iterator_get_pointer(it_v));
}
printf(")\n");

/* The second function used to compute the product of the element-wise sums */
algo_inner_product_if(vector_begin(pvec_v1), vector_end(pvec_v1),
    list_begin(plist_l1), 1, fun_multiplies_int, fun_plus_int, &n_result);

printf("The sum of the element-wise products of v1 and l1 is: %d.\n",
    n_result);

/* Constructing a vector of partial sums of element-wise products */
for(it_v = vector_begin(pvec_v1), i = 0;
    !iterator_equal(it_v, vector_end(pvec_v1));
    it_v = iterator_next(it_v), ++i)
{
    algo_inner_product_if(vector_begin(pvec_v1), iterator_next(it_v),
        list_begin(plist_l1), 1, fun_multiplies_int,
        fun_plus_int, vector_at(pvec_v3, i));
```

```
    }

    printf("The vector of partial sums of element-wise products "
           "between v1 and l1 is:\n( ");
    for(it_v = vector_begin(pvec_v3);
        !iterator_equal(it_v, vector_end(pvec_v3));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);
    vector_destroy(pvec_v3);
    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
The original vector v1 is:
( 1 2 3 4 5 6 7 )
The original list l1 is:
( 1 2 3 4 5 6 7 )
The inner_product of the vector v1 and the list l1 is: 140.
The vector of partial inner_products between v1 and l1 is:
( 1 5 14 30 55 91 140 )
The sum of the element-wise products of v1 and l1 is: 645120.
The vector of partial sums of element-wise products between v1 and l1 is:
( 2 8 48 384 3840 46080 645120 )
```

4. algo_iota

根据初始值填充数据区间。

```
void algo_iota(
    forward_iterator_t it_first,
    forward_iterator_t it_last,
    element
);
```

- **Parameters**

it_first: 数据区间的开始位置。
it_last: 数据区间的末尾位置。
element: 初始化值。

- **Remarks**

algo_iota 执行之后数据区间中的数据:
*it_first = element, *(it_first + 1) = element + 1, *(it_first + 2) = element + 2, ..., *(it_last – 1) = element + it_last – it_first - 1

- **Requirements**

头文件 <cstl/cnumeric.h>。

- **Example**

```
/*
 * algo_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_vec1 = create_vector(int);
    vector_iterator_t it_vec;

    if(pvec_vec1 == NULL)
    {
        return -1;
    }

    vector_init_n(pvec_vec1, 10);

    algo_iota(vector_begin(pvec_vec1), vector_end(pvec_vec1), 7);
    printf("The vector vec1 is:");
    for(it_vec = vector_begin(pvec_vec1);
        !iterator_equal(it_vec, vector_end(pvec_vec1));
        it_vec = iterator_next(it_vec))
    {
        printf(" %d", *(int*)iterator_get_pointer(it_vec));
    }
    printf("\n");

    vector_destroy(pvec_vec1);

    return 0;
}
```

- **Output**

```
The vector vec1 is: 7 8 9 10 11 12 13 14 15 16
```

5. algo_partial_sum algo_partial_sum_if

计算数据区间局部和。

```
output_iterator_t algo_partial_sum(
    input_iterator_t it_first,
    input_iterator_t it_last,
    output_iterator_t it_result
);

output_iterator_t algo_partial_sum_if(
```

```
    input_iterator_t it_first,
    input_iterator_t it_last,
    output_iterator_t it_result,
    binary_function_t bfun_op
);
```

- **Parameters**

  it_first: 数据区间的开始位置。
  it_last: 数据区间的末尾位置。
  it_result: 目的数据区间的开始位置。
  bfun_op: 指定的算法函数。

- **Remarks**

  返回目的数据区间拷贝的数据的末尾。
  要保证目的数据区间至少和源数据区间一样大。
  algo_partial_sum 执行后目的数据区间的数据:
  *it_first, *it_first + *(it_first + 1), *it_first + *(it_first + 1) + *(it_first + 2), ...
  algo_partial_sum_if 执行后目的数据区间的数据:
  *it_first, *it_first bfun_op *(it_first + 1), *it_first bfun_op *(it_first + 1) bfun_op *(it_first + 2), ...

- **Requirements**

  头文件 <cstl/cnumeric.h>。

- **Example**

```
/*
 * algo_partial_sum.c
 * compile with : -lcstl
 */

#include <stdio.h>
#include <cstl/cvector.h>
#include <cstl/clist.h>
#include <cstl/cnumeric.h>
#include <cstl/cfunctional.h>

int main(int argc, char* argv[])
{
    vector_t* pvec_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_iterator_t it_v;
    list_t* plist_l1 = create_list(int);
    list_iterator_t it_l;
    iterator_t it_end;

    if(pvec_v1 == NULL || pvec_v2 == NULL || plist_l1 == NULL)
    {
        return -1;
    }

    vector_init_n(pvec_v1, 10);
    vector_init_n(pvec_v2, 10);
    list_init_n(plist_l1, 10);
```

```
algo_iota(list_begin(plist_l1), list_end(plist_l1), 1);
printf("The input list l1 is:\n( ");
for(it_l = list_begin(plist_l1);
    !iterator_equal(it_l, list_end(plist_l1));
    it_l = iterator_next(it_l))
{
    printf("%d ", *(int*)iterator_get_pointer(it_l));
}
printf(")\n");

/*
 * The first function for the partial sums of
 * elements in a list output to a vector.
 */
it_end = algo_partial_sum(list_begin(plist_l1), list_end(plist_l1),
    vector_begin(pvec_v1));
printf("The output vector conatining the partial sums is:\n( ");
for(it_v = vector_begin(pvec_v1);
    !iterator_equal(it_v, it_end);
    it_v = iterator_next(it_v))
{
    printf("%d ", *(int*)iterator_get_pointer(it_v));
}
printf(")\n");

/*
 * The second function used to compute
 * the partial product of the elements in a list.
 */
it_end = algo_partial_sum_if(list_begin(plist_l1), list_end(plist_l1),
    vector_begin(pvec_v2), fun_multiplies_int);
printf("The output vector with the partial products is:\n( ");
for(it_v = vector_begin(pvec_v2);
    !iterator_equal(it_v, it_end);
    it_v = iterator_next(it_v))
{
    printf("%d ", *(int*)iterator_get_pointer(it_v));
}
printf(")\n");

/* Computation of partial sums in place */
it_end = algo_partial_sum(list_begin(plist_l1),
    list_end(plist_l1), list_begin(plist_l1));
printf("The in place output partial_sum list l1 is:\n( ");
for(it_l = list_begin(plist_l1);
    !iterator_equal(it_l, it_end);
    it_l = iterator_next(it_l))
{
    printf("%d ", *(int*)iterator_get_pointer(it_l));
```

```
    }
    printf(")\n");

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);
    list_destroy(plist_l1);

    return 0;
}
```

- **Output**

```
The input list l1 is:
( 1 2 3 4 5 6 7 8 9 10 )
The output vector conatining the partial sums is:
( 1 3 6 10 15 21 28 36 45 55 )
The output vector with the partial products is:
( 1 2 6 24 120 720 5040 40320 362880 3628800 )
The in place output partial_sum list l1 is:
( 1 3 6 10 15 21 28 36 45 55 )
```

6. algo_power　algo_power_if

计算数据的幂。

```
void algo_power(
    iterator_t it_iter,
    size_t t_power,
    void* pv_output
);

void algo_power_if(
    iterator_t it_iter,
    size_t t_power,
    binary_function_t bfun_op,
    void* pv_output
);
```

- **Parameters**

it_iter: 底数的迭代器。
t_power: 幂指数。
bfun_op: 指定的算法函数。
pv_output: 指向输出值的指针。

- **Remarks**

algo_power 的计算过程是:
*pv_output = *it_iter * *it_iter * ... * *it_iter
algo_power_if 的计算过程是:
*pv_output = *it_iter bfun_op *it_iter bfun_op ... bfun_op *it_iter

- **Requirements**

头文件 <cstl/cnumeric.h>。

- **Example**

```c
/*
 * algo_power.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_v1 = create_vector(int);
    vector_t* pvec_v2 = create_vector(int);
    vector_iterator_t it_v;
    int i = 0;

    if(pvec_v1 == NULL || pvec_v2 == NULL)
    {
        return -1;
    }

    vector_init_n(pvec_v1, 10);
    vector_init_n(pvec_v2, 10);

    algo_iota(vector_begin(pvec_v1), vector_end(pvec_v1), 1);

    /* The first function for power */
    printf("The original vector v1 is:\n( ");
    for(it_v = vector_begin(pvec_v1);
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    for(it_v = vector_begin(pvec_v1), i = 0;
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v), ++i)
    {
        algo_power(it_v, 3, vector_at(pvec_v2, i));
    }

    printf("The power result vector v2 is:\n( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");
```

```
    /* The second function for multiplus */
    for(it_v = vector_begin(pvec_v1), i = 0;
        !iterator_equal(it_v, vector_end(pvec_v1));
        it_v = iterator_next(it_v), ++i)
    {
        algo_power_if(it_v, 3, fun_plus_int, vector_at(pvec_v2, i));
    }

    printf("The multiplus result vector v2 is:\n( ");
    for(it_v = vector_begin(pvec_v2);
        !iterator_equal(it_v, vector_end(pvec_v2));
        it_v = iterator_next(it_v))
    {
        printf("%d ", *(int*)iterator_get_pointer(it_v));
    }
    printf(")\n");

    vector_destroy(pvec_v1);
    vector_destroy(pvec_v2);

    return 0;
}
```

- **Output**

```
The original vector v1 is:
( 1 2 3 4 5 6 7 8 9 10 )
The power result vector v2 is:
( 1 8 27 64 125 216 343 512 729 1000 )
The multiplus result vector v2 is:
( 3 6 9 12 15 18 21 24 27 30 )
```

第五章 工具类型

第一节 bool_t

TYPE：

bool_t

VALUE:

false

true

FALSE

TRUE

DESCRIPTION:

bool_t 是 libcstl 定义的新类型用来表示布尔值。

DEFINITION:

包含任何一个 libcstl 头文件都可以使用 bool_t 类型。

第二节 pair_t

TYPE:

pair_t

DESCRIPTION:

pair_t 保存两个任意类型的数据，它将两个不同的数据统一在一起，是对的概念。

DEFINITION:

<cstl/cutility.h>

MEMBER:

first	void*类型的指针，用来引用第一个数据。
second	void*类型的指针，用来引用第二个数据。

OPERATION:

pair_t create_pair(first_type, second_type);	创建指定类型的 pair_t，first_type 为第一个数据的类型，second_type 为第二个数据的类型。
void pair_init(pair_t* pt_pair);	初始化 pair_t，值为空。
void pair_init_elem(pair_t* pt_pair, first_element, second_element);	使用两个值来初始化 pair_t。
void pair_init_copy(pair_t* pt_pair, const pair_t* cpt_src);	使用另一个 pair_t 来初始化 pair_t。
void pair_destroy(pair_t* pt_pair);	销毁 pair_t。
void pair_assign(pair_t* pt_pair, const pair_t* cpt_src);	使用另一个 pair_t 赋值。
void pair_make(pair_t* pt_pair, first_element, second_element);	使用两个值 first_element 和 second_element 来构造已经出初始化的 pair_t。
bool_t pair_equal(const pair_t* cpt_first, const pair_t* cpt_second);	判断两个 pair_t 是否相等。
bool_t pair_not_equal(const pair_t* cpt_first, const pair_t* cpt_second);	判断两个 pair_t 是否不等。
bool_t pair_less(const pair_t* cpt_first,	判断第一个 pair_t 是否小于第二个 pair_t。

const pair_t* cpt_second);	
bool_t pair_less_equal(const pair_t* cpt_first, const pair_t* cpt_second);	判断第一个 pair_t 是否小于等于第二个 pair_t。
bool_t pair_great(const pair_t* cpt_first, const pair_t* cpt_second);	判断第一个 pair_t 是否大于第二个 pair_t。
bool_t pair_great_equal(const pair_t* cpt_first, const pair_t* cpt_second);	判断第一个 pair_t 是否大于等于第二个 pair_t。

第六章 函数类型

TYPE：

    unary_function_t
    binary_function_t

DEFINITION:

    所有的函数声明在<cstl/cfunctional.h>

第一节 算术运算函数

1. plus

PROTOTYPE：

```
void fun_plus_char(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_plus_uchar(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_plus_short(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_plus_ushort(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_plus_int(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_plus_uint(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_plus_long(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_plus_ulong(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_plus_float(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_plus_double(const void* cpv_first, const void* cpv_second, void* pv_output);
```

DESCRIPTION:

fun_plus_xxxx()函数是对所有的C语言内部类型进行加法操作的二元函数，cpv_first和cpv_second都是输入参数，计算结果保存在pv_output中。

2. minus

PROTOTYPE：

```
void fun_minus_char(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_minus_uchar(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_minus_short(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_minus_ushort(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_minus_int(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_minus_uint(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_minus_long(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_minus_ulong(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_minus_float(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_minus_double(const void* cpv_first, const void* cpv_second, void* pv_output);
```

DESCRIPTION:

fun_minus_xxxx()函数是对所有的C语言内部类型进行减法操作的二元函数，cpv_first和cpv_second都是输入参数，计算结果保存在pv_output中。

3. multiplies

PROTOTYPE：

```
void fun_multiplies_char(const void* cpv_first, const void* cpv_second, void*
```

pv_output);
void fun_multiplies_uchar(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_multiplies_short(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_multiplies_ushort(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_multiplies_int(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_multiplies_uint(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_multiplies_long(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_multiplies_ulong(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_multiplies_float(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_multiplies_double(const void* cpv_first, const void* cpv_second, void* pv_output);

DESCRIPTION:

fun_multiplies_xxxx()函数是对所有的C语言内部类型进行乘法操作的二元函数，cpv_first和cpv_second都是输入参数，计算结果保存在pv_output中。

4. divides

PROTOTYPE:

void fun_divides_char(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_divides_uchar(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_divides_short(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_divides_ushort(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_divides_int(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_divides_uint(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_divides_long(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_divides_ulong(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_divides_float(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_divides_double(const void* cpv_first, const void* cpv_second, void* pv_output);

DESCRIPTION:

fun_divides_xxxx()函数是对所有的C语言内部类型进行除法操作的二元函数，cpv_first和cpv_second都是输入参数，计算结果保存在pv_output中。

5. modulus

PROTOTYPE：

```
void fun_negate_char(const void* cpv_input, void* pv_output);
void fun_negate_short(const void* cpv_input, void* pv_output);
void fun_negate_int(const void* cpv_input, void* pv_output);
void fun_negate_long(const void* cpv_input, void* pv_output);
void fun_negate_float(const void* cpv_input, void* pv_output);
void fun_negate_double(const void* cpv_input, void* pv_output);
```

DESCRIPTION:

fun_negate_xxxx()函数是对所有的C语言内部类型进行取反操作的一元函数，cpv_input是输入参数，计算结果保存在pv_output中。

6. negate

PROTOTYPE：

```
void fun_modulus_char(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_modulus_uchar(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_modulus_short(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_modulus_ushort(const void* cpv_first, const void* cpv_second, void*
pv_output);
void fun_modulus_int(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_modulus_uint(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_modulus_long(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_modulus_ulong(const void* cpv_first, const void* cpv_second, void* pv_output);
```

DESCRIPTION:

fun_modulus_xxxx()函数是对所有的C语言内部类型进行取余操作的二元函数，cpv_first和cpv_second都是输入参数，计算结果保存在pv_output中。

第二节 关系运算函数

1. equal_to

PROTOTYPE：

```
void fun_equal_char(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_equal_uchar(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_equal_short(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_equal_ushort(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_equal_int(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_equal_uint(const void* cpv_first, const void* cpv_second, void* pv_output);
```

```
void fun_equal_long(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_equal_ulong(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_equal_float(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_equal_double(const void* cpv_first, const void* cpv_second, void* pv_output);
```

DESCRIPTION:

fun_equal_xxxx()函数是对所有的C语言内部类型进行判断是否相等的二元谓词，cpv_first和cpv_second都是输入参数，比较结果保存在pv_output中，pv_output实际上是bool_t*。

2. not_equal_to

PROTOTYPE:

```
void fun_not_equal_char(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_not_equal_uchar(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_not_equal_short(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_not_equal_ushort(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_not_equal_int(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_not_equal_uint(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_not_equal_long(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_not_equal_ulong(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_not_equal_float(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_not_equal_double(const void* cpv_first, const void* cpv_second, void* pv_output);
```

DESCRIPTION:

fun_not_equal_xxxx()函数是对所有的C语言内部类型进行判断是否不相等的二元谓词，cpv_first和cpv_second都是输入参数，比较结果保存在pv_output中，pv_output实际上是bool_t*。

3. less

PROTOTYPE:

```
void fun_less_char(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_less_uchar(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_less_short(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_less_ushort(const void* cpv_first, const void* cpv_second, void* pv_output);
```

```
void fun_less_int(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_less_uint(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_less_long(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_less_ulong(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_less_float(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_less_double(const void* cpv_first, const void* cpv_second, void* pv_output);
```

DESCRIPTION:

fun_less_xxxx()函数是对所有的C语言内部类型进行判断的二元谓词，判断*cpv_first 是否小于*cpv_second， cpv_first和cpv_second都是输入参数，比较结果保存在pv_output中，pv_output实际上是bool_t*。

4. less_equal

PROTOTYPE:

```
void fun_less_equal_char(const void* cpv_first, const void* cpv_second, void*
pv_output);
void fun_less_equal_uchar(const void* cpv_first, const void* cpv_second, void*
pv_output);
void fun_less_equal_short(const void* cpv_first, const void* cpv_second, void*
pv_output);
void fun_less_equal_ushort(const void* cpv_first, const void* cpv_second, void*
pv_output);
void fun_less_equal_int(const void* cpv_first, const void* cpv_second, void*
pv_output);
void fun_less_equal_uint(const void* cpv_first, const void* cpv_second, void*
pv_output);
void fun_less_equal_long(const void* cpv_first, const void* cpv_second, void*
pv_output);
void fun_less_equal_ulong(const void* cpv_first, const void* cpv_second, void*
pv_output);
void fun_less_equal_float(const void* cpv_first, const void* cpv_second, void*
pv_output);
void fun_less_equal_double(const void* cpv_first, const void* cpv_second, void*
pv_output);
```

DESCRIPTION:

fun_less_equal_xxxx()函数是对所有的C语言内部类型进行判断的二元谓词，判断*cpv_first 是否小于等于 *cpv_second， cpv_first和cpv_second都是输入参数，比较结果保存在pv_output中，pv_output实际上是bool_t*。

5. great

PROTOTYPE:

```
void fun_great_char(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_great_uchar(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_great_short(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_great_ushort(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_great_int(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_great_uint(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_great_long(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_great_ulong(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_great_float(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_great_double(const void* cpv_first, const void* cpv_second, void* pv_output);
```

DESCRIPTION:

fun_great_xxxx()函数是对所有的C语言内部类型进行判断的二元谓词，判断*cpv_first 是否大于*cpv_second， cpv_first和cpv_second都是输入参数，比较结果保存在pv_output中，pv_output实际上是bool_t*。

6. great_equal

PROTOTYPE:

```
void fun_great_equal_char(const void* cpv_first, const void* cpv_second, void*
 pv_output);
void fun_great_equal_uchar(const void* cpv_first, const void* cpv_second, void*
 pv_output);
void fun_great_equal_short(const void* cpv_first, const void* cpv_second, void*
 pv_output);
void fun_great_equal_ushort(const void* cpv_first, const void* cpv_second, void*
 pv_output);
void fun_great_equal_int(const void* cpv_first, const void* cpv_second, void*
 pv_output);
void fun_great_equal_uint(const void* cpv_first, const void* cpv_second, void*
 pv_output);
void fun_great_equal_long(const void* cpv_first, const void* cpv_second, void*
 pv_output);
void fun_great_equal_ulong(const void* cpv_first, const void* cpv_second, void*
 pv_output);
void fun_great_equal_float(const void* cpv_first, const void* cpv_second, void*
 pv_output);
void fun_great_equal_double(const void* cpv_first, const void* cpv_second, void*
 pv_output);
```

DESCRIPTION:

fun_great_equal_xxxx()函数是对所有的C语言内部类型进行判断的二元谓词，判断*cpv_first 是否大于等于 *cpv_second， cpv_first和cpv_second都是输入参数，比较结果保存在pv_output中，pv_output实际上是

bool_t*。

第三节 逻辑运算函数

1. logical_and

PROTOTYPE：

```
void fun_logical_and_bool(const void* cpv_first, const void* cpv_second, void* pv_output);
```

DESCRIPTION:

fun_logical_and_bool()函数是对 bool_t 类型的数据进行逻辑与操作的二元函数，cpv_first 和 cpv_second 都是输入参数，操作结果保存在 pv_output 中。

2. logical_or

PROTOTYPE：

```
void fun_logical_or_bool(const void* cpv_first, const void* cpv_second, void* pv_output);
```

DESCRIPTION:

fun_logical_or_bool()函数是对 bool_t 类型的数据进行逻辑或操作的二元函数，cpv_first 和 cpv_second 都是输入参数，操作结果保存在 pv_output 中。

3. logical_not

PROTOTYPE：

```
void fun_logical_not_bool(const void* cpv_input, void* pv_output);
```

DESCRIPTION:

fun_logical_not_bool()函数是对 bool_t 类型的数据进行逻辑非操作的一元函数，cpv_input 是输入参数，操作结果保存在 pv_output 中。

第四节 其他函数

1. random_number

PROTOTYPE：

```
void fun_random_number(const void* cpv_input, void* pv_output);
```

DESCRIPTION:

fun_random_number()函数是产生随机数的一元函数，cpv_input 是输入参数，操作结果保存在 pv_output 中。

2. default

PROTOTYPE：

```
void fun_default_binary(const void* cpv_first, const void* cpv_second, void* pv_output);
void fun_default_unary(const void* cpv_input, void* pv_output);
```

DESCRIPTION:

fun_default_binary()函数是默认的二元函数。
fun_default_unary()函数是默认的一元函数。