The libcstl Library Reference Manual

The libcstl Library Reference Manual

for libcstl 2.0

Wangbo

2010-04-23

Table of Contents

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：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。
- **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。

● 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* pt_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，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

返回这想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当。

- **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校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。

- **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	访问最有一个数据。
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 为空则返回的结构和 list_begin()函数的结构相等。

- **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
 */

#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. slist_t

TYPE:

slist_t

ITERATOR TYPE:

forward_iterator_t
slist_iterator_t

DESCRIPTION:

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数量时间，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。除此之外 slist_t 还提供了许多额外的操作函数。

DEFINITION:

<cstl/cslist.h>

OPERATION:

slist_t create_slist(type);	创建指定类型的 slist_t 容器。
void slist_init(slist_t* pt_slist);	初始化一个空的 slist_t 容器。
void slist_init_n(slist_t* pt_slist, size_t t_count);	初始化一个具有 t_count 个数据的 slist_t 容器，每个数据的值都是 0。
void slist_init_elem(slist_t* pt_slist, size_t t_count, element);	初始化一个具有 t_count 个数据的 slist_t 容器，每个数据的值都是 element。
void slist_init_copy(slist_t* pt_slist, const slist_t* cpt_src);	使用令一个 slist_t 容器初始化 slist_t 容器。
void slist_init_copy_range(slist_t* pt_slist, slist_iterator_t t_begin, slist_iterator_t t_end);	使用数据区间[t_begin, t_end)初始化 slist_t 容器。[1][2]
void slist_destroy(slist_t* pt_slist);	销毁 slist_t 容器。
size_t slist_size(const slist_t* cpt_slist);	获得 slist_t 容器中数据的数目。
size_t slist_max_size(const slist_t* cpt_slist);	获得 slist_t 容器中能够保存的数据的最大数目。
bool_t slist_empty(const slist_t* cpt_slist);	判断 slist_t 容器是否为空。
bool_t slist_equal(const slist_t* cpt_first, const slist_t* cpt_second);	判断两个 slist_t 容器是否相等。
bool_t slist_not_equal(const slist_t* cpt_first, const slist_t* cpt_second);	判断两个 slist_t 容器是否不等。
bool_t slist_less(const slist_t* cpt_first, const slist_t* cpt_second);	判断第一个 slist_t 容器是否小于第二个 slist_t 容器。

bool_t slist_less_equal(const slist_t* cpt_first, const slist_t* cpt_second);	判断第一个 slist_t 容器是否小于等于第二个 slist_t 容器。
bool_t slist_great(const slist_t* cpt_first, const slist_t* cpt_second);	判断第一个 slist_t 容器是否大于第二个 slist_t 容器。
bool_t slist_great_equal(const slist_t* cpt_first, const slist_t* cpt_second);	判断第一个 slist_t 容器是否大于等于第二个 slist_t 容器。
void slist_assign(slist_t* pt_slist, const slist_t* cpt_src);	使用另一个 slist_t 容器为当前 slist_t 容器赋值。
void slist_assign_elem(slist_t* pt_slist, size_t t_count, element);	使用 t_count 个 element 值给 slist_t 容器赋值。
void slist_assign_range(slist_t* pt_slist, slist_iterator_t t_begin, slist_iterator_t t_end);	使用数据区间[t_begin, t_end)为 slist_t 容器赋值。[1][2]
void slist_swap(slist_t* pt_first, slist_t* pt_second);	交换两个 slist_t 容器的内容。
void* slist_front(const slist_t* cpt_slist);	访问 slist_t 容器中的第一个数据。
slist_iterator_t slist_begin(const slist_t* cpt_slist);	返回指向 slist_t 容器开始的迭代器。
slist_iterator_t slist_end(const slist_t* cpt_slist);	返回指向 slist_t 容器结尾的迭代器。
slist_iterator_t slist_previous(const slist_t* cpt_slist, slist_iterator_t t_pos);	获得 t_pos 前驱的迭代器。
slist_iterator_t slist_insert(slist_t* pt_slist, slist_iterator_t t_pos, element);	在 t_pos 前面插入数据 element，并返回指向新数据的迭代器。
void slist_insert_n(slist_t* pt_slist, slist_iterator_t t_pos, size_t t_count, element);	在 t_pos 前面插入 t_count 个数据 element。
void slist_insert_range(slist_t* pt_slist, slist_iterator_t t_pos, slist_iterator_t t_begin, slist_iterator_t t_end);	在 t_pos 前面插入数据区间[t_begin, t_end)。[1][2]
slist_iterator_t slist_insert_after(slist_t* pt_slist, slist_iterator_t t_pos, element);	在 t_pos 后面插入数据 element，并返回指向新数据的迭代器。
void slist_insert_after_n(slist_t* pt_slist, slist_iterator_t t_pos, size_t t_count, element);	在 t_pos 后面插入 t_count 个数据 element。

void slist_insert_after_range(slist_t* pt_slist, slist_iterator_t t_pos, slist_iterator_t t_begin, slist_iterator_t t_end);	在 t_pos 后面插入数据区间[t_begin, t_end)。[1][2]
void slist_push_front(slist_t* pt_slist, element);	将数据 element 插入到 slist_t 容器的开头。
void slist_pop_front(slist_t* pt_slist);	删除 slist_t 容器的第一个数据。
void slist_remove(slist_t* pt_slist, element);	删除 slist_t 容器中所有值为 element 的数据。
void slist_remove_if(slist_t* pt_slist, unary_function_t t_unary_op);	删除 slist_t 容器中所有满足一元谓词 t_unary_op 的数据。
slist_iterator_t slist_erase(slist_t* pt_slist, slist_iterator_t t_pos);	删除 t_pos 位置的数据，并返回指向下一个数据的迭代器。
slist_iterator_t slist_erase_range(slist_t* pt_slist, slist_iterator_t t_begin, slist_iterator_t t_end);	删除数据区间[t_begin, t_end)的数据，并返回指向下一个数据的迭代器。[1]
slist_iterator_t slist_erase_after(slist_t* pt_slist, slist_iterator_t t_pos);	删除 t_pos 位置后面的数据，并返回指向下一个数据的迭代器。
slist_iterator_t slist_erase_after_range(slist_t* pt_slist, slist_iterator_t t_begin, slist_iterator_t t_end);	删除数据区间[t_begin+1, t_end)的数据，并返回指向下一个数据的迭代器。[1]
void slist_resize(slist_t* pt_slist, size_t t_resize);	重置 slist_t 容器中数据的数目，新增的数据为 0。
void slist_resize_elem(slist_t* pt_slist, size_t t_resize, element);	重置 slist_t 容器中数据的数目，新增的数据为 element。
void slist_clear(slist_t* pt_slist);	清空 slist_t 容器。
void slist_unique(slist_t* pt_slist);	删除 slist_t 容器中连续的重复数据。
void slist_unique_if(slist_t* pt_slist, binary_function_t t_binary_op);	删除 slist_t 容器中连续的满足二元谓词 t_binary_op 的数据。
void slist_splice(slist_t* pt_slist, slist_iterator_t t_pos, slist_t* pt_src);	将 pt_src 中的数据转移到 t_pos。
Void slist_splice_pos(slist_t* pt_slist, slist_iterator_t t_pos, slist_t* pt_src, slist_iterator_t t_srcpos);	将 t_srcpos 位置的数据转移到 t_pos。
void slist_splice_range(slist_t* pt_slist, slist_iterator_t t_pos, slist_t* pt_src, slist_iterator_t t_begin, slist_iterator_t t_end);	将数据区间[t_begin, t_end)中的数据转移到 t_pos。[1]

Void slist_splice_after_pos(slist_t* pt_slist, slist_iterator_t t_pos, slist_t* pt_src, slist_iterator_t t_prev);	将 t_prev 位置后面的数据转移到 t_pos 后面。
void slist_splice_after_range(slist_t* pt_slist, slist_iterator_t t_pos, slist_t* pt_src, slist_iterator_t t_begin, slist_iterator_t t_end);	将数据区间[t_begin+1, t_end+1)中的数据转移到 t_pos 后面。[1]
void slist_sort(slist_t* pt_slist);	排序 slist_t 容器中的数据。
void slist_sort_if(slist_t* pt_slist, binary_function_t t_binary_op);	使用二元谓词 t_binary_op 作为排序规则，排序 slist_t 容器中的数据。
void slist_merge(slist_t* pt_first, slist_t* pt_second);	将两个有序的 slist_t 容器合并。
void slist_merge_if(slist_t* pt_first, slist_t* pt_second, binary_function_t t_binary_op);	使用二元谓词 t_binary_op 作为规则，合并两个 slist_t 容器中的数据。
void slist_reverse(slist_t* pt_slist);	将 slist_t 容器中的数据逆序。

NOTE:
[1]:数据区间[t_begin, t_end)必须是有效的数据区间。
[2]:数据区间[t_begin, t_end)必须属于另一个 slist_t 容器。

第三节 关联容器

1. set_t

TYPE:
set_t

ITERATOR TYPE:
bidirectional_iterator_t
set_iterator_t

DESCRIPTION:
关联容器是根据数据中的键值对容器中的数据进行自动排序的。set_t 容器简单的关联容器，容器中数据本身就是数据的键值。set_t 容器中的数据是不允许重复的。关联容器的特定是自动排序，则样查找数据非常方便。所以关联容器都提供了很多查找的操作函数。

DEFINITION:
<cstl/cset.h>

OPERATION:

set_t create_set(type);	创建指定类型的 set_t 容器。
void set_init(set_t* pt_set);	初始化一个空的 set_t 容器。
void set_init_copy(set_t* pt_set, const set_t* cpt_src);	使用令一个 set_t 容器初始化 set_t 容器。

void set_init_copy_range(set_t* pt_set, set_iterator_t t_begin, set_iterator_t t_end);	使用数据区间[t_begin, t_end)初始化 set_t 容器。[1] [2]
void set_destroy(set_t* pt_set);	销毁 set_t 容器。
size_t set_size(const set_t* cpt_set);	获得 set_t 容器中数据的数目。
size_t set_max_size(const set_t* cpt_set);	获得 set_t 容器中能够保存的数据的最大数目。
bool_t set_empty(const set_t* cpt_set);	判断 set_t 容器是否为空。
bool_t set_equal(const set_t* cpt_first, const set_t* cpt_second);	判断两个 set_t 容器是否相等。
bool_t set_not_equal(const set_t* cpt_first, const set_t* cpt_second);	判断两个 set_t 容器是否不等。
bool_t set_less(const set_t* cpt_first, const set_t* cpt_second);	判断第一个 set_t 容器是否小于第二个 set_t 容器。
bool_t set_less_equal(const set_t* cpt_first, const set_t* cpt_second);	判断第一个 set_t 容器是否小于等于第二个 set_t 容器。
bool_t set_great(const set_t* cpt_first, const set_t* cpt_second);	判断第一个 set_t 容器是否大于第二个 set_t 容器。
bool_t set_great_equal(const set_t* cpt_first, const set_t* cpt_second);	判断第一个 set_t 容器是否大于等于第二个 set_t 容器。
size_t set_count(const set_t* cpt_set, element);	返回 set_t 容器中值为 element 的数据的数目。
set_iterator_t set_find(const set_t* cpt_set, element);	返回值为 element 的数据的位置。
set_iterator_t set_lower_bound(const set_t* cpt_set, element);	返回第一个不小于 element 的数据的位置。
set_iterator_t set_upper_bound(const set_t* cpt_set, element);	返回第一个大于 element 的数据的位置。
pair_t set_equal_range(const set_t* cpt_set, element);	返回一个由数据区间的上下限组成的 pair_t，这个数据区间中包含所有的值为 element 的数据。
void set_assign(set_t* pt_set, const set_t* cpt_src);	使用另一个 set_t 容器为当前 set_t 容器赋值。
void set_swap(set_t* pt_first, set_t* pt_second);	交换两个 set_t 容器的内容。
set_iterator_t set_begin(const set_t* cpt_set);	返回指向 set_t 容器开始的迭代器。
set_iterator_t set_end(const set_t* cpt_set);	返回指向 set_t 容器结尾的迭代器。
set_iterator_t set_insert(set_t* pt_set, element);	向 set_t 容器中插入数据 element，成功返回新数据的位置，不成功返回 set_end()。
set_iterator_t set_insert_hint(	向 set_t 容器中插入数据 element 时使用线索位置

set_t* pt_set, set_iterator_t t_hint, element);	t_hint，成功返回新数据的位置，不成功返回set_end()。
void set_insert_range(set_t* pt_set, set_iterator_t t_begin, set_iterator_t t_end);	向 set_t 容器中插入数据区间[t_begin, t_end)。[1][2]
size_t set_erase(set_t* pt_set, element);	删除值为 element 的数据，同时返回删除的数据的数目。
void set_erase_pos(set_t* pt_set, set_iterator_t t_pos);	删除 t_pos 位置的数据。
void set_erase_range(set_t* pt_set, set_iterator_t t_begin, set_iterator_t t_end);	删除数据区间[t_begin, t_end)的数据。[1]
void set_clear(set_t* pt_set);	清空 set_t 容器。

NOTE:
[1]:数据区间[t_begin, t_end)必须是有效的数据区间。
[2]:数据区间[t_begin, t_end)必须属于另一个 set_t 容器。

2. multiset_t

TYPE:
multiset_t

ITERATOR TYPE:
bidirectional_iterator_t
multiset_iterator_t

DESCRIPTION:
multiset_t 和 set_t 是十分相似的,set_t 不允许数据重复,但是 multiset_t 允许数据重复。所以 multiset_t 的插入操作是不会失败的。除此之外没有其他的不同了。

DEFINITION:
<cstl/cset.h>

OPERATION:

multiset_t create_multiset(type);	创建指定类型的 multiset_t 容器。
void multiset_init(multiset_t* pt_multiset);	初始化一个空的 multiset_t 容器。
void multiset_init_copy(multiset_t* pt_multiset, const multiset_t* cpt_src);	使用令一个 multiset_t 容器初始化 multiset_t 容器。
void multiset_init_copy_range(multiset_t* pt_multiset, multiset_iterator_t t_begin, multiset_iterator_t t_end);	使用数据区间[t_begin, t_end)初始化 multiset_t 容器。[1][2]
void multiset_destroy(multiset_t* pt_multiset);	销毁 multiset_t 容器。
size_t multiset_size(	获得 multiset_t 容器中数据的数目。

const multiset_t* cpt_multiset);	
size_t multiset_max_size(const multiset_t* cpt_multiset);	获得 multiset_t 容器中能够保存的数据的最大数目。
bool_t multiset_empty(const multiset_t* cpt_multiset);	判断 multiset_t 容器是否为空。
bool_t multiset_equal(const multiset_t* cpt_first, const multiset_t* cpt_second);	判断两个 multiset_t 容器是否相等。
bool_t multiset_not_equal(const multiset_t* cpt_first, const multiset_t* cpt_second);	判断两个 multiset_t 容器是否不等。
bool_t multiset_less(const multiset_t* cpt_first, const multiset_t* cpt_second);	判断第一个 multiset_t 容器是否小于第二个 multiset_t 容器。
bool_t multiset_less_equal(const multiset_t* cpt_first, const multiset_t* cpt_second);	判断第一个 multiset_t 容器是否小于等于第二个 multiset_t 容器。
bool_t multiset_great(const multiset_t* cpt_first, const multiset_t* cpt_second);	判断第一个 multiset_t 容器是否大于第二个 multiset_t 容器。
bool_t multiset_great_equal(const multiset_t* cpt_first, const multiset_t* cpt_second);	判断第一个 multiset_t 容器是否大于等于第二个 multiset_t 容器。
size_t multiset_count(const multiset_t* cpt_multiset, element);	返回 multiset_t 容器中值为 element 的数据的数目。
multiset_iterator_t multiset_find(const multiset_t* cpt_multiset, element);	返回值为 element 的数据的位置。
multiset_iterator_t multiset_lower_bound(const multiset_t* cpt_multiset, element);	返回第一个不小于 element 的数据的位置。
multiset_iterator_t multiset_upper_bound(const multiset_t* cpt_multiset, element);	返回第一个大于 element 的数据的位置。
pair_t multiset_equal_range(const multiset_t* cpt_multiset, element);	返回一个由数据区间的上下限组成的 pair_t，这个数据区间中包含所有的值为 element 的数据。
void multiset_assign(multiset_t* pt_multiset, const multiset_t* cpt_src);	使用另一个 multiset_t 容器为当前 multiset_t 容器赋值。
void multiset_swap(multiset_t* pt_first, multiset_t* pt_second);	交换两个 multiset_t 容器的内容。
multiset_iterator_t multiset_begin(const multiset_t* cpt_multiset);	返回指向 multiset_t 容器开始的迭代器。
multiset_iterator_t multiset_end(const multiset_t* cpt_multiset);	返回指向 multiset_t 容器结尾的迭代器。
multiset_iterator_t multiset_insert(multiset_t* pt_multiset, element);	向 multiset_t 容器中插入数据 element，成功返回新数据的位置。
multiset_iterator_t multiset_insert_hint(	向 multiset_t 容器中插入数据 element 时使用线索位置

multiset_t* pt_multiset, multiset_iterator_t t_hint, element);	t_hint，返回新数据的位置。
void multiset_insert_range(multiset_t* pt_multiset, multiset_iterator_t t_begin, multiset_iterator_t t_end);	向 multiset_t 容器中插入数据区间[t_begin, t_end)。[1][2]
size_t multiset_erase(multiset_t* pt_multiset, element);	删除值为 element 的数据，同时返回删除的数据的数目。
void multiset_erase_pos(multiset_t* pt_multiset, multiset_iterator_t t_pos);	删除 t_pos 位置的数据。
void multiset_erase_range(multiset_t* pt_multiset, multiset_iterator_t t_begin, multiset_iterator_t t_end);	删除数据区间[t_begin, t_end)的数据。[1]
void multiset_clear(multiset_t* pt_multiset);	清空 multiset_t 容器。

NOTE:
[1]:数据区间[t_begin, t_end)必须是有效的数据区间。
[2]:数据区间[t_begin, t_end)必须属于另一个 multiset_t 容器。

3. map_t

TYPE:
map_t

ITERATOR TYPE:
bidirectional_iterator_t
map_iterator_t

DESCRIPTION:
map_t 和 set_t 是十分相似的,set_t 不允许数据重复,map_t 也是不允许数据重复，但是 map_t 中保存的数据类型是 pair_t，也就是由 key/value 组成的对，map_t 不允许重复的是 key，但 value 是可以重复的。向 map_t 中插入的数据必须都是 pair_t 类型，在插入 pair_t 是 map_t 复制了插入的数据。此外 map_t 还可以作为关联数组使用，通过 key 对 value 进行随机访问。

DEFINITION:
<cstl/cmap.h>

OPERATION:

map_t create_map(key_type, value_type);	创建指定类型的 map_t 容器。
void map_init(map_t* pt_map);	初始化一个空的 map_t 容器。
void map_init_copy(map_t* pt_map, const map_t* cpt_src);	使用令一个 map_t 容器初始化 map_t 容器。
void map_init_copy_range(map_t* pt_map, map_iterator_t t_begin, map_iterator_t t_end);	使用数据区间[t_begin, t_end)初始化 map_t 容器。[1][2]

void map_destroy(map_t* pt_map);	销毁 map_t 容器。
size_t map_size(const map_t* cpt_map);	获得 map_t 容器中数据的数目。
size_t map_max_size(const map_t* cpt_map);	获得 map_t 容器中能够保存的数据的最大数目。
bool_t map_empty(const map_t* cpt_map);	判断 map_t 容器是否为空。
bool_t map_equal(const map_t* cpt_first, const map_t* cpt_second);	判断两个 map_t 容器是否相等。
bool_t map_not_equal(const map_t* cpt_first, const map_t* cpt_second);	判断两个 map_t 容器是否不等。
bool_t map_less(const map_t* cpt_first, const map_t* cpt_second);	判断第一个 map_t 容器是否小于第二个 map_t 容器。
bool_t map_less_equal(const map_t* cpt_first, const map_t* cpt_second);	判断第一个 map_t 容器是否小于等于第二个 map_t 容器。
bool_t map_great(const map_t* cpt_first, const map_t* cpt_second);	判断第一个 map_t 容器是否大于第二个 map_t 容器。
bool_t map_great_equal(const map_t* cpt_first, const map_t* cpt_second);	判断第一个 map_t 容器是否大于等于第二个 map_t 容器。
size_t map_count(const map_t* cpt_map, key_element);	返回 map_t 容器中值为 key_element 的数据的数目。
map_iterator_t map_find(const map_t* cpt_map, key_element);	返回值为 key_element 的数据的位置。
map_iterator_t map_lower_bound(const map_t* cpt_map, key_element);	返回第一个不小于 key_element 的数据的位置。
map_iterator_t map_upper_bound(const map_t* cpt_map, key_element);	返回第一个大于 key_element 的数据的位置。
pair_t map_equal_range(const map_t* cpt_map, key_element);	返回一个由数据区间的上下限组成的 pair_t，这个数据区间中包含所有的值为 key_element 的数据。
void map_assign(map_t* pt_map, const map_t* cpt_src);	使用另一个 map_t 容器为当前 map_t 容器赋值。
void map_swap(map_t* pt_first, map_t* pt_second);	交换两个 map_t 容器的内容。
map_iterator_t map_begin(const map_t* cpt_map);	返回指向 map_t 容器开始的迭代器。
map_iterator_t map_end(const map_t* cpt_map);	返回指向 map_t 容器结尾的迭代器。
map_iterator_t map_insert(map_t* pt_map, const pair_t* cpt_pair);	向 map_t 容器中插入数据对 cpt_pair，成功返回新数据的位置，不成功返回 map_end()。
map_iterator_t map_insert_hint(map_t* pt_map, map_iterator_t t_hint, const pair_t* cpt_pair);	向 map_t 容器中插入数据对 cpt_pair 时使用线索位置 t_hint，成功返回新数据的位置，不成功返回 map_end()。
void map_insert_range(	向 map_t 容器中插入数据区间[t_begin, t_end)。[1][2]

map_t* pt_map, map_iterator_t t_begin, map_iterator_t t_end);	
size_t map_erase(map_t* pt_map, key_element);	删除值为 key_element 的数据，同时返回删除的数据的数目。
void map_erase_pos(map_t* pt_map, map_iterator_t t_pos);	删除 t_pos 位置的数据。
void map_erase_range(map_t* pt_map, map_iterator_t t_begin, map_iterator_t t_end);	删除数据区间[t_begin, t_end)的数据。[1]
void map_clear(map_t* pt_map);	清空 map_t 容器。
void* map_at(map_t* pt_map, key_element);	关联数组操作，通过 key_element 对相应的值进行访问。返回指向值的指针，如果 map_t 容器中没有相应的 key/value 数据，则首先添加 key/0 然后返回指向值的指针。

NOTE:
[1]:数据区间[t_begin, t_end)必须是有效的数据区间。
[2]:数据区间[t_begin, t_end)必须属于另一个 map_t 容器。

4. multimap_t

TYPE:
multimap_t

ITERATOR TYPE:
bidirectional_iterator_t
multimap_iterator_t

DESCRIPTION:
multimap_t 和 map_t 是十分相似的,map_t 不允许数据重复,multimap_t 允许数据重复，所以向 multimap_t 中插入数据时不会失败。此外 map_t 可以作为关联数组使用，但是 multimap_t 不可以。

DEFINITION:
<cstl/cmap.h>

OPERATION:

multimap_t create_multimap(key_type, value_type);	创建指定类型的 multimap_t 容器。
void multimap_init(multimap_t* pt_multimap);	初始化一个空的 multimap_t 容器。
void multimap_init_copy(multimap_t* pt_multimap, const multimap_t* cpt_src);	使用令一个 multimap_t 容器初始化 multimap_t 容器。
void multimap_init_copy_range(multimap_t* pt_multimap, multimap_iterator_t t_begin, multimap_iterator_t t_end);	使用数据区间[t_begin, t_end)初始化 multimap_t 容器。[1][2]
void multimap_destroy(multimap_t* pt_multimap);	销毁 multimap_t 容器。

size_t multimap_size(const multimap_t* cpt_multimap);	获得 multimap_t 容器中数据的数目。
size_t multimap_max_size(const multimap_t* cpt_multimap);	获得 multimap_t 容器中能够保存的数据的最大数目。
bool_t multimap_empty(const multimap_t* cpt_multimap);	判断 multimap_t 容器是否为空。
bool_t multimap_equal(const multimap_t* cpt_first, const multimap_t* cpt_second);	判断两个 multimap_t 容器是否相等。
bool_t multimap_not_equal(const multimap_t* cpt_first, const multimap_t* cpt_second);	判断两个 multimap_t 容器是否不等。
bool_t multimap_less(const multimap_t* cpt_first, const multimap_t* cpt_second);	判断第一个 multimap_t 容器是否小于第二个 multimap_t 容器。
bool_t multimap_less_equal(const multimap_t* cpt_first, const multimap_t* cpt_second);	判断第一个 multimap_t 容器是否小于等于第二个 multimap_t 容器。
bool_t multimap_great(const multimap_t* cpt_first, const multimap_t* cpt_second);	判断第一个 multimap_t 容器是否大于第二个 multimap_t 容器。
bool_t multimap_great_equal(const multimap_t* cpt_first, const multimap_t* cpt_second);	判断第一个 multimap_t 容器是否大于等于第二个 multimap_t 容器。
size_t multimap_count(const multimap_t* cpt_multimap, key_element);	返回 multimap_t 容器中值为 key_element 的数据的数目。
multimap_iterator_t multimap_find(const multimap_t* cpt_multimap, key_element);	返回值为 key_element 的数据的位置。
multimap_iterator_t multimap_lower_bound(const multimap_t* cpt_multimap, key_element);	返回第一个不小于 key_element 的数据的位置。
multimap_iterator_t multimap_upper_bound(const multimap_t* cpt_multimap, key_element);	返回第一个大于 key_element 的数据的位置。
pair_t multimap_equal_range(const multimap_t* cpt_multimap, key_element);	返回一个由数据区间的上下限组成的 pair_t，这个数据区间中包含所有的值为 key_element 的数据。
void multimap_assign(multimap_t* pt_multimap, const multimap_t* cpt_src);	使用另一个 multimap_t 容器为当前 multimap_t 容器赋值。
void multimap_swap(multimap_t* pt_first, multimap_t* pt_second);	交换两个 multimap_t 容器的内容。
multimap_iterator_t multimap_begin(const multimap_t* cpt_multimap);	返回指向 multimap_t 容器开始的迭代器。
multimap_iterator_t multimap_end(const multimap_t* cpt_multimap);	返回指向 multimap_t 容器结尾的迭代器。
multimap_iterator_t multimap_insert(multimap_t* pt_multimap,	向 multimap_t 容器中插入数据对 cpt_pair，返回新数据的位置。

const pair_t* cpt_pair);	
multimap_iterator_t multimap_insert_hint(multimap_t* pt_multimap, multimap_iterator_t t_hint, const pair_t* cpt_pair);	向 multimap_t 容器中插入数据对 cpt_pair 时使用线索位置 t_hint，返回新数据的位置。
void multimap_insert_range(multimap_t* pt_multimap, multimap_iterator_t t_begin, multimap_iterator_t t_end);	向 multimap_t 容器中插入数据区间[t_begin, t_end)。[1][2]
size_t multimap_erase(multimap_t* pt_multimap, key_element);	删除值为 key_element 的数据，同时返回删除的数据的数目。
void multimap_erase_pos(multimap_t* pt_multimap, multimap_iterator_t t_pos);	删除 t_pos 位置的数据。
void multimap_erase_range(multimap_t* pt_multimap, multimap_iterator_t t_begin, multimap_iterator_t t_end);	删除数据区间[t_begin, t_end)的数据。[1]
void multimap_clear(multimap_t* pt_multimap);	清空 multimap_t 容器。

NOTE:
[1]:数据区间[t_begin, t_end)必须是有效的数据区间。
[2]:数据区间[t_begin, t_end)必须属于另一个 multimap_t 容器。

5. hash_set_t

TYPE:
hash_set_t

ITERATOR TYPE:
forward_iterator_t
hash_set_iterator_t

DESCRIPTION:
hash_set_t 也是关联容器的一种，但是它不同于 set_t，它使用 hash 表机制来保存数据，所以 hash_set_t 内部数据并不是排序的，但是它仍然能够提供高效的存取数据和查找。作为集合 hash_set_t 与 set_t 行为类似都是不允许数据重复。

DEFINITION:
<cstl/chash_set.h>

OPERATION:

hash_set_t create_hash_set(type);	创建指定类型的 hash_set_t 容器。
void hash_set_init(hash_set_t* pt_hash_set, int (*pfun_hash)(const void*, size_t, size_t));	初始化一个空的 hash_set_t 容器。[3]
void hash_set_init_n(	初始化一个空的 hash_set_t 容器，容器的 hash 表大小为

hash_set_t* pt_hash_set, size_t t_bucketcount, int (*pfun_hash)(const void*, size_t, size_t));	t_bucketcount。[3]
void hash_set_init_copy(hash_set_t* pt_hash_set, const hash_set_t* cpt_src);	使用令一个 hash_set_t 容器初始化 hash_set_t 容器。
void hash_set_init_copy_range(hash_set_t* pt_hash_set, hash_set_iterator_t t_begin, hash_set_iterator_t t_end, int (*pfun_hash)(const void*, size_t, size_t));	使用数据区间[t_begin, t_end)初始化 hash_set_t 容器。[1][2][3]
void hash_set_init_copy_range_n(hash_set_t* pt_hash_set, hash_set_iterator_t t_begin, hash_set_iterator_t t_end, size_t t_bucketcount, int (*pfun_hash)(const void*, size_t, size_t));	使用数据区间[t_begin, t_end)初始化 hash_set_t 容器。[1][2][3]
void hash_set_destroy(hash_set_t* pt_hash_set);	销毁 hash_set_t 容器。
size_t hash_set_size(const hash_set_t* cpt_hash_set);	获得 hash_set_t 容器中数据的数目。
size_t hash_set_max_size(const hash_set_t* cpt_hash_set);	获得 hash_set_t 容器中能够保存的数据的最大数目。
bool_t hash_set_empty(const hash_set_t* cpt_hash_set);	判断 hash_set_t 容器是否为空。
size_t hash_set_bucket_count(const hash_set_t* cpt_hash_set);	获得 hash_set_t 容器中 hash 表的大小。
int (*hash_set_hash_func(const hash_set_t* cpt_hash_set))(const void*, size_t, size_t);	获得 hash_set_t 容器的 hash 函数。[3]
bool_t hash_set_equal(const hash_set_t* cpt_first, const hash_set_t* cpt_second);	判断两个 hash_set_t 容器是否相等。
bool_t hash_set_not_equal(const hash_set_t* cpt_first, const hash_set_t* cpt_second);	判断两个 hash_set_t 容器是否不等。
bool_t hash_set_less(const hash_set_t* cpt_first, const hash_set_t* cpt_second);	判断第一个 hash_set_t 容器是否小于第二个 hash_set_t 容器。
bool_t hash_set_less_equal(const hash_set_t* cpt_first, const hash_set_t* cpt_second);	判断第一个 hash_set_t 容器是否小于等于第二个 hash_set_t 容器。
bool_t hash_set_great(const hash_set_t* cpt_first, const hash_set_t* cpt_second);	判断第一个 hash_set_t 容器是否大于第二个 hash_set_t 容器。

bool_t hash_set_great_equal(const hash_set_t* cpt_first, const hash_set_t* cpt_second);	判断第一个 hash_set_t 容器是否大于等于第二个 hash_set_t 容器。
size_t hash_set_count(const hash_set_t* cpt_hash_set, element);	返回 hash_set_t 容器中值为 element 的数据的数目。
hash_set_iterator_t hash_set_find(const hash_set_t* cpt_hash_set, element);	返回值为 element 的数据的位置。
pair_t hash_set_equal_range(const hash_set_t* cpt_hash_set, element);	返回一个由数据区间的上下限组成的 pair_t，这个数据区间中包含所有的值为 element 的数据。
void hash_set_assign(hash_set_t* pt_hash_set, const hash_set_t* cpt_src);	使用另一个 hash_set_t 容器为当前 hash_set_t 容器赋值。
void hash_set_swap(hash_set_t* pt_first, hash_set_t* pt_second);	交换两个 hash_set_t 容器的内容。
hash_set_iterator_t hash_set_begin(const hash_set_t* cpt_hash_set);	返回指向 hash_set_t 容器开始的迭代器。
hash_set_iterator_t hash_set_end(const hash_set_t* cpt_hash_set);	返回指向 hash_set_t 容器结尾的迭代器。
void hash_set_resize(hash_set_t* pt_hash_set, size_t t_resize);	修改 hash_set_t 容器的 hash 表的大小。
hash_set_iterator_t hash_set_insert(hash_set_t* pt_hash_set, element);	向 hash_set_t 容器中插入数据 element，成功返回新数据的位置，不成功返回 hash_set_end()。
void hash_set_insert_range(hash_set_t* pt_hash_set, hash_set_iterator_t t_begin, hash_set_iterator_t t_end);	向 hash_set_t 容器中插入数据区间[t_begin, t_end)。[1][2]
size_t hash_set_erase(hash_set_t* pt_hash_set, element);	删除值为 element 的数据，同时返回删除的数据的数目。
void hash_set_erase_pos(hash_set_t* pt_hash_set, hash_set_iterator_t t_pos);	删除 t_pos 位置的数据。
void hash_set_erase_range(hash_set_t* pt_hash_set, hash_set_iterator_t t_begin, hash_set_iterator_t t_end);	删除数据区间[t_begin, t_end)的数据。[1]
void hash_set_clear(hash_set_t* pt_hash_set);	清空 hash_set_t 容器。

NOTE:

[1]:数据区间[t_begin, t_end)必须是有效的数据区间。

[2]:数据区间[t_begin, t_end)必须属于另一个 hash_set_t 容器。

[3]:hash 函数的形式为 int (*pfun_hash)(const void*, size_t, size_t)。第一个参数为容器中的数据，第二个参数是容器中数据的大小，第三个参数是容器中 hash 表的大小，返回值是数据保存在 hash 表中的位置索引。如果调用这不提供自定义的 hash 函数（使用 NULL），libcstl 就使用默认的 hash 函数。

6. hash_multiset_t

TYPE:

hash_multiset_t

ITERATOR TYPE:

forward_iterator_t

hash_multiset_iterator_t

DESCRIPTION:

hash_multiset_t 和 hash_set_t 是十分相似的,hash_set_t 不允许数据重复,但是 hash_multiset_t 允许数据重复。所以 hash_multiset_t 的插入操作是不会失败的。除此之外没有其他的不同了。

DEFINITION:

<cstl/chash_set.h>

OPERATION:

hash_multiset_t create_hash_multiset(type);	创建指定类型的 hash_multiset_t 容器。
void hash_multiset_init(hash_multiset_t* pt_hash_multiset, int (*pfun_hash)(const void*, size_t, size_t));	初始化一个空的 hash_multiset_t 容器。
void hash_multiset_init_n(hash_multiset_t* pt_hash_multiset, size_t t_bucketcount, int (*pfun_hash)(const void*, size_t, size_t));	初始化一个空的 hash_multiset_t 容器，容器的 hash 表大小为 t_bucketcount。
void hash_multiset_init_copy(hash_multiset_t* pt_hash_multiset, const hash_multiset_t* cpt_src);	使用令一个 hash_multiset_t 容器初始化 hash_multiset_t 容器。
void hash_multiset_init_copy_range(hash_multiset_t* pt_hash_multiset, hash_multiset_iterator_t t_begin, hash_multiset_iterator_t t_end, int (*pfun_hash)(const void*, size_t, size_t));	使用数据区间[t_begin, t_end)初始化 hash_multiset_t 容器。[1][2]
void hash_multiset_init_copy_range_n(hash_multiset_t* pt_hash_multiset, hash_multiset_iterator_t t_begin, hash_multiset_iterator_t t_end, size_t t_bucketcount, int (*pfun_hash)(const void*, size_t, size_t));	使用数据区间[t_begin, t_end)初始化 hash_multiset_t 容器。[1][2]
void hash_multiset_destroy(hash_multiset_t* pt_hash_multiset);	销毁 hash_multiset_t 容器。
size_t hash_multiset_size(const hash_multiset_t* cpt_hash_multiset);	获得 hash_multiset_t 容器中数据的数目。
size_t hash_multiset_max_size(const hash_multiset_t* cpt_hash_multiset);	获得 hash_multiset_t 容器中能够保存的数据的最大数目。

bool_t hash_multiset_empty(const hash_multiset_t* cpt_hash_multiset);	判断 hash_multiset_t 容器是否为空。
size_t hash_multiset_bucket_count(const hash_multiset_t* cpt_hash_multiset);	获得 hash_multiset_t 容器中 hash 表的大小。
int (*hash_multiset_hash_func(const hash_multiset_t* cpt_hash_multiset))(const void*, size_t, size_t);	获得 hash_multiset_t 容器的 hash 函数。
bool_t hash_multiset_equal(const hash_multiset_t* cpt_first, const hash_multiset_t* cpt_second);	判断两个 hash_multiset_t 容器是否相等。
bool_t hash_multiset_not_equal(const hash_multiset_t* cpt_first, const hash_multiset_t* cpt_second);	判断两个 hash_multiset_t 容器是否不等。
bool_t hash_multiset_less(const hash_multiset_t* cpt_first, const hash_multiset_t* cpt_second);	判断第一个 hash_multiset_t 容器是否小于第二个 hash_multiset_t 容器。
bool_t hash_multiset_less_equal(const hash_multiset_t* cpt_first, const hash_multiset_t* cpt_second);	判断第一个 hash_multiset_t 容器是否小于等于第二个 hash_multiset_t 容器。
bool_t hash_multiset_great(const hash_multiset_t* cpt_first, const hash_multiset_t* cpt_second);	判断第一个 hash_multiset_t 容器是否大于第二个 hash_multiset_t 容器。
bool_t hash_multiset_great_equal(const hash_multiset_t* cpt_first, const hash_multiset_t* cpt_second);	判断第一个 hash_multiset_t 容器是否大于等于第二个 hash_multiset_t 容器。
size_t hash_multiset_count(const hash_multiset_t* cpt_hash_multiset, element);	返回 hash_multiset_t 容器中值为 element 的数据的数目。
hash_multiset_iterator_t hash_multiset_find(const hash_multiset_t* cpt_hash_multiset, element);	返回值为 element 的数据的位置。
pair_t hash_multiset_equal_range(const hash_multiset_t* cpt_hash_multiset, element);	返回一个由数据区间的上下限组成的 pair_t，这个数据区间中包含所有的值为 element 的数据。
void hash_multiset_assign(hash_multiset_t* pt_hash_multiset, const hash_multiset_t* cpt_src);	使用另一个 hash_multiset_t 容器为当前 hash_multiset_t 容器赋值。
void hash_multiset_swap(hash_multiset_t* pt_first, hash_multiset_t* pt_second);	交换两个 hash_multiset_t 容器的内容。
hash_multiset_iterator_t hash_multiset_begin(const hash_multiset_t* cpt_hash_multiset);	返回指向 hash_multiset_t 容器开始的迭代器。
hash_multiset_iterator_t hash_multiset_end(const hash_multiset_t* cpt_hash_multiset);	返回指向 hash_multiset_t 容器结尾的迭代器。
void hash_multiset_resize(hash_multiset_t* pt_hash_multiset,	修改 hash_multiset_t 容器的 hash 表的大小。

size_t t_resize);	
hash_multiset_iterator_t hash_multiset_insert(hash_multiset_t* pt_hash_multiset, element);	向 hash_multiset_t 容器中插入数据 element，返回新数据的位置。
void hash_multiset_insert_range(hash_multiset_t* pt_hash_multiset, hash_multiset_iterator_t t_begin, hash_multiset_iterator_t t_end);	向 hash_multiset_t 容器中插入数据区间[t_begin, t_end)。[1][2]
size_t hash_multiset_erase(hash_multiset_t* pt_hash_multiset, element);	删除值为 element 的数据，同时返回删除的数据的数目。
void hash_multiset_erase_pos(hash_multiset_t* pt_hash_multiset, hash_multiset_iterator_t t_pos);	删除 t_pos 位置的数据。
void hash_multiset_erase_range(hash_multiset_t* pt_hash_multiset, hash_multiset_iterator_t t_begin, hash_multiset_iterator_t t_end);	删除数据区间[t_begin, t_end)的数据。[1]
void hash_multiset_clear(hash_multiset_t* pt_hash_multiset);	清空 hash_multiset_t 容器。

NOTE:
[1]:数据区间[t_begin, t_end)必须是有效的数据区间。
[2]:数据区间[t_begin, t_end)必须属于另一个 hash_multiset_t 容器。

7. hash_map_t

TYPE:
hash_map_t

ITERATOR TYPE:
forward_iterator_t
hash_map_iterator_t

DESCRIPTION:
hash_map_t 也是关联容器的一种，但是它不同于 map_t，它使用 hash 表机制来保存数据，所以 hash_map_t 内部数据并不是排序的，但是它仍然能够提供高效的存取数据和查找。作为集合 hash_map_t 与 map_t 行为类似都是不允许数据重复，支持通过键值对数据进行随机访问。

DEFINITION:
<cstl/chash_map.h>

OPERATION:

hash_map_t create_hash_map(key_type, value_type);	创建指定类型的 hash_map_t 容器。
void hash_map_init(hash_map_t* pt_hash_map, int (*pfun_hash)(const void*, size_t, size_t));	初始化一个空的 hash_map_t 容器。
void hash_map_init_n(hash_map_t* pt_hash_map,	初始化一个空的 hash_map_t 容器，容器的 hash 表大小为 t_bucketcount。

size_t t_bucketcount, int (*pfun_hash)(const void*, size_t, size_t));	
void hash_map_init_copy(hash_map_t* pt_hash_map, const hash_map_t* cpt_src);	使用令一个 hash_map_t 容器初始化 hash_map_t 容器。
void hash_map_init_copy_range(hash_map_t* pt_hash_map, hash_map_iterator_t t_begin, hash_map_iterator_t t_end, int (*pfun_hash)(const void*, size_t, size_t));	使用数据区间[t_begin, t_end)初始化 hash_map_t 容器。[1][2]
void hash_map_init_copy_range_n(hash_map_t* pt_hash_map, hash_map_iterator_t t_begin, hash_map_iterator_t t_end, size_t t_bucketcount, int (*pfun_hash)(const void*, size_t, size_t));	使用数据区间[t_begin, t_end)初始化 hash_map_t 容器。[1][2]
void hash_map_destroy(hash_map_t* pt_hash_map);	销毁 hash_map_t 容器。
size_t hash_map_size(const hash_map_t* cpt_hash_map);	获得 hash_map_t 容器中数据的数目。
size_t hash_map_max_size(const hash_map_t* cpt_hash_map);	获得 hash_map_t 容器中能够保存的数据的最大数目。
bool_t hash_map_empty(const hash_map_t* cpt_hash_map);	判断 hash_map_t 容器是否为空。
size_t hash_map_bucket_count(const hash_map_t* cpt_hash_map);	获得 hash_map_t 容器中 hash 表的大小。
int (*hash_map_hash_func(const hash_map_t* cpt_hash_map))(const void*, size_t, size_t);	获得 hash_map_t 容器的 hash 函数。
bool_t hash_map_equal(const hash_map_t* cpt_first, const hash_map_t* cpt_second);	判断两个 hash_map_t 容器是否相等。
bool_t hash_map_not_equal(const hash_map_t* cpt_first, const hash_map_t* cpt_second);	判断两个 hash_map_t 容器是否不等。
bool_t hash_map_less(const hash_map_t* cpt_first, const hash_map_t* cpt_second);	判断第一个 hash_map_t 容器是否小于第二个 hash_map_t 容器。
bool_t hash_map_less_equal(const hash_map_t* cpt_first, const hash_map_t* cpt_second);	判断第一个 hash_map_t 容器是否小于等于第二个 hash_map_t 容器。
bool_t hash_map_great(const hash_map_t* cpt_first, const hash_map_t* cpt_second);	判断第一个 hash_map_t 容器是否大于第二个 hash_map_t 容器。

bool_t hash_map_great_equal(const hash_map_t* cpt_first, const hash_map_t* cpt_second);	判断第一个 hash_map_t 容器是否大于等于第二个 hash_map_t 容器。
size_t hash_map_count(const hash_map_t* cpt_hash_map, key_element);	返回 hash_map_t 容器中值为 key_element 的数据的数目。
hash_map_iterator_t hash_map_find(const hash_map_t* cpt_hash_map, key_element);	返回值为 key_element 的数据的位置。
pair_t hash_map_equal_range(const hash_map_t* cpt_hash_map, key_element);	返回一个由数据区间的上下限组成的 pair_t，这个数据区间中包含所有的值为 key_element 的数据。
void hash_map_assign(hash_map_t* pt_hash_map, const hash_map_t* cpt_src);	使用另一个 hash_map_t 容器为当前 hash_map_t 容器赋值。
void hash_map_swap(hash_map_t* pt_first, hash_map_t* pt_second);	交换两个 hash_map_t 容器的内容。
hash_map_iterator_t hash_map_begin(const hash_map_t* cpt_hash_map);	返回指向 hash_map_t 容器开始的迭代器。
hash_map_iterator_t hash_map_end(const hash_map_t* cpt_hash_map);	返回指向 hash_map_t 容器结尾的迭代器。
void hash_map_resize(hash_map_t* pt_hash_map, size_t t_resize);	修改 hash_map_t 容器的 hash 表的大小。
hash_map_iterator_t hash_map_insert(hash_map_t* pt_hash_map, const pair_t* cpt_pair);	向 hash_map_t 容器中插入数据对 cpt_pair，成功返回新数据的位置，不成功返回 hash_map_end()。
void hash_map_insert_range(hash_map_t* pt_hash_map, hash_map_iterator_t t_begin, hash_map_iterator_t t_end);	向 hash_map_t 容器中插入数据区间[t_begin, t_end)。[1][2]
size_t hash_map_erase(hash_map_t* pt_hash_map, key_element);	删除值为 key_element 的数据，同时返回删除的数据的数目。
void hash_map_erase_pos(hash_map_t* pt_hash_map, hash_map_iterator_t t_pos);	删除 t_pos 位置的数据。
void hash_map_erase_range(hash_map_t* pt_hash_map, hash_map_iterator_t t_begin, hash_map_iterator_t t_end);	删除数据区间[t_begin, t_end)的数据。[1]
void hash_map_clear(hash_map_t* pt_hash_map);	清空 hash_map_t 容器。
void* hash_map_at(hash_map_t* pt_hash_map, key_element);	关联数组操作，通过 key_element 对相应的值进行访问。返回指向值的指针，如果 hash_map_t 容器中没有相应的 key/value 数据，则首先添加 key/0 然后返回指向值的指针。

NOTE:

[1]:数据区间[t_begin, t_end)必须是有效的数据区间。

[2]:数据区间[t_begin, t_end)必须属于另一个 hash_map_t 容器。

8. hash_multimap_t

TYPE:

hash_multimap_t

ITERATOR TYPE:

forward_iterator_t

hash_multimap_iterator_t

DESCRIPTION:

hash_multimap_t 和 hash_map_t 是十分相似的,hash_map_t 不允许数据重复,hash_multimap_t 允许数据重复，所以向 hash_multimap_t 中插入数据时不会失败。此外 hash_map_t 可以作为关联数组使用，但是 hash_multimap_t 不可以。

DEFINITION:

<cstl/chash_map.h>

OPERATION:

hash_multimap_t create_hash_multimap(key_type, value_type);	创建指定类型的 hash_multimap_t 容器。
void hash_multimap_init(hash_multimap_t* pt_hash_multimap, int (*pfun_hash)(const void*, size_t, size_t));	初始化一个空的 hash_multimap_t 容器。
void hash_multimap_init_n(hash_multimap_t* pt_hash_multimap, size_t t_bucketcount, int (*pfun_hash)(const void*, size_t, size_t));	初始化一个空的 hash_multimap_t 容器，容器的 hash 表大小为 t_bucketcount。
void hash_multimap_init_copy(hash_multimap_t* pt_hash_multimap, const hash_multimap_t* cpt_src);	使用令一个 hash_multimap_t 容器初始化 hash_multimap_t 容器。
void hash_multimap_init_copy_range(hash_multimap_t* pt_hash_multimap, hash_multimap_iterator_t t_begin, hash_multimap_iterator_t t_end, int (*pfun_hash)(const void*, size_t, size_t));	使用数据区间[t_begin, t_end)初始化 hash_multimap_t 容器。[1][2]
void hash_multimap_init_copy_range_n(hash_multimap_t* pt_hash_multimap, hash_multimap_iterator_t t_begin, hash_multimap_iterator_t t_end, size_t t_bucketcount, int (*pfun_hash)(const void*, size_t, size_t));	使用数据区间[t_begin, t_end)初始化 hash_multimap_t 容器。[1][2]
void hash_multimap_destroy(hash_multimap_t* pt_hash_multimap);	销毁 hash_multimap_t 容器。
size_t hash_multimap_size(const hash_multimap_t* cpt_hash_multimap);	获得 hash_multimap_t 容器中数据的数目。

size_t hash_multimap_max_size(const hash_multimap_t* cpt_hash_multimap);	获得 hash_multimap_t 容器中能够保存的数据的最大数目。
bool_t hash_multimap_empty(const hash_multimap_t* cpt_hash_multimap);	判断 hash_multimap_t 容器是否为空。
size_t hash_multimap_bucket_count(const hash_multimap_t* cpt_hash_multimap);	获得 hash_multimap_t 容器中 hash 表的大小。
int (*hash_multimap_hash_func(const hash_multimap_t* cpt_hash_multimap))(const void*, size_t, size_t);	获得 hash_multimap_t 容器的 hash 函数。
bool_t hash_multimap_equal(const hash_multimap_t* cpt_first, const hash_multimap_t* cpt_second);	判断两个 hash_multimap_t 容器是否相等。
bool_t hash_multimap_not_equal(const hash_multimap_t* cpt_first, const hash_multimap_t* cpt_second);	判断两个 hash_multimap_t 容器是否不等。
bool_t hash_multimap_less(const hash_multimap_t* cpt_first, const hash_multimap_t* cpt_second);	判断第一个 hash_multimap_t 容器是否小于第二个 hash_multimap_t 容器。
bool_t hash_multimap_less_equal(const hash_multimap_t* cpt_first, const hash_multimap_t* cpt_second);	判断第一个 hash_multimap_t 容器是否小于等于第二个 hash_multimap_t 容器。
bool_t hash_multimap_great(const hash_multimap_t* cpt_first, const hash_multimap_t* cpt_second);	判断第一个 hash_multimap_t 容器是否大于第二个 hash_multimap_t 容器。
bool_t hash_multimap_great_equal(const hash_multimap_t* cpt_first, const hash_multimap_t* cpt_second);	判断第一个 hash_multimap_t 容器是否大于等于第二个 hash_multimap_t 容器。
size_t hash_multimap_count(const hash_multimap_t* cpt_hash_multimap, key_element);	返回 hash_multimap_t 容器中值为 key_element 的数据的数目。
hash_multimap_iterator_t hash_multimap_find(const hash_multimap_t* cpt_hash_multimap, key_element);	返回值为 key_element 的数据的位置。
pair_t hash_multimap_equal_range(const hash_multimap_t* cpt_hash_multimap, key_element);	返回一个由数据区间的上下限组成的 pair_t，这个数据区间中包含所有的值为 key_element 的数据。
void hash_multimap_assign(hash_multimap_t* pt_hash_multimap, const hash_multimap_t* cpt_src);	使用另一个 hash_multimap_t 容器为当前 hash_multimap_t 容器赋值。
void hash_multimap_swap(hash_multimap_t* pt_first, hash_multimap_t* pt_second);	交换两个 hash_multimap_t 容器的内容。
hash_multimap_iterator_t hash_multimap_begin(const hash_multimap_t* cpt_hash_multimap);	返回指向 hash_multimap_t 容器开始的迭代器。
hash_multimap_iterator_t hash_multimap_end(const hash_multimap_t* cpt_hash_multimap);	返回指向 hash_multimap_t 容器结尾的迭代器。

void hash_multimap_resize(hash_multimap_t* pt_hash_multimap, size_t t_resize);	修改 hash_multimap_t 容器的 hash 表的大小。
hash_multimap_iterator_t hash_multimap_insert(hash_multimap_t* pt_hash_multimap, const pair_t* cpt_pair);	向 hash_multimap_t 容器中插入数据对 cpt_pair，返回新数据的位置。
void hash_multimap_insert_range(hash_multimap_t* pt_hash_multimap, hash_multimap_iterator_t t_begin, hash_multimap_iterator_t t_end);	向 hash_multimap_t 容器中插入数据区间[t_begin, t_end)。[1][2]
size_t hash_multimap_erase(hash_multimap_t* pt_hash_multimap, key_element);	删除值为 key_element 的数据，同时返回删除的数据的数目。
void hash_multimap_erase_pos(hash_multimap_t* pt_hash_multimap, hash_multimap_iterator_t t_pos);	删除 t_pos 位置的数据。
void hash_multimap_erase_range(hash_multimap_t* pt_hash_multimap, hash_multimap_iterator_t t_begin, hash_multimap_iterator_t t_end);	删除数据区间[t_begin, t_end)的数据。[1]
void hash_multimap_clear(hash_multimap_t* pt_hash_multimap);	清空 hash_multimap_t 容器。

NOTE:
[1]:数据区间[t_begin, t_end)必须是有效的数据区间。
[2]:数据区间[t_begin, t_end)必须属于另一个 hash_multimap_t 容器。

第四节 字符串

1. string_t

TYPE:
string_t

ITERATOR TYPE:
random_access_iterator_t
string_iterator_t

VALUE:
NPOS

DESCRIPTION:
string_t 是一个保存字符型数据的序列容器，它包含所有序列容器的操作，此为它还提供了像查找，连接等常用的字符串操作。string_t 的许多操作函数除了接受迭代器参数外，还提供了接受特殊的下标参数的同等功能的函数。

DEFINITION:

<cstl/cstring.h>

OPERATION:

void string_init(string_t* pt_string);	初始化一个空的 string_t 容器。
void string_init_cstr(string_t* pt_string, const char* s_cstr);	使用字符串常量初始化 string_t 容器。
void string_init_subcstr(string_t* pt_string, const char* s_cstr, size_t t_len);	使用字符串常量的子串初始化 string_t 容器。
void string_init_char(string_t* pt_string, size_t t_count, char c_char);	使用 t_count 个字符 c_char 来初始化 string_t 容器。
void string_init_copy(string_t* pt_string, const string_t* cpt_src);	使用令一个 string_t 容器初始化 string_t 容器。
void string_init_copy_substring(string_t* pt_string, const string_t* cpt_str, size_t t_pos, size_t t_len);	使用子字符串初始化 string_t 容器。
void string_init_copy_range(string_t* pt_string, string_iterator_t t_begin, string_iterator_t t_end);	使用数据区间[t_begin, t_end)初始化 string_t 容器。[1][2]
void string_destroy(string_t* pt_string);	销毁 string_t 容器。
const char* string_c_str(const string_t* cpt_string);	返回一个指向'\0'结尾的字符串的指针。
const char* string_data(const string_t* cpt_string);	返回一个指向字符数组的指针。
size_t string_copy(const string_t* cpt_string, char* s_buffer, size_t t_copysize, size_t t_copypos);	将 string_t 容器中从 t_copypos 开始的最多 t_copysize 个字符拷贝到 s_buffer 缓冲区中，返回实际拷贝的字符数。
size_t string_size(const string_t* cpt_string);	返回 string_t 容器中字符的个数。
size_t string_length(const string_t* cpt_string);	返回 string_t 容器的长度，与 string_size()功能相同。
size_t string_max_size(const string_t* cpt_string);	返回 string_t 容器中能够保存的字符的最大数目。
size_t string_capacity(const string_t* cpt_string);	返回 string_t 容器的容量。
bool_t string_empty(const string_t* cpt_string);	判断 string_t 容器是否为空。
void string_reserve(string_t* pt_string, size_t t_size);	设置 string_t 容器的容量。
void string_resize(string_t* pt_string, size_t t_size, char c_char);	设置 string_t 容器中字符的数目，string_t 容器中字符数目不足时使用 c_char 填充。
char* string_at(const char* cpt_string, size_t t_pos);	使用下标随机访问 string_t 容器中的字符，返回指向该字符的指针。
bool_t string_equal(	判断两个 string_t 容器是否相等。

const string_t* cpt_first, const string_t* cpt_second);	
bool_t string_not_equal(const string_t* cpt_first, const string_t* cpt_second);	判断两个 string_t 容器是否不等。
bool_t string_less(const string_t* cpt_first, const string_t* cpt_second);	判断第一个 string_t 容器是否小于第二个 string_t 容器。
bool_t string_less_equal(const string_t* cpt_first, const string_t* cpt_second);	判断第一个 string_t 容器是否小于等于第二个 string_t 容器。
bool_t string_great(const string_t* cpt_first, const string_t* cpt_second);	判断第一个 string_t 容器是否大于第二个 string_t 容器。
bool_t string_great_equal(const string_t* cpt_first, const string_t* cpt_second);	判断第一个 string_t 容器是否大于等于第二个 string_t 容器。
bool_t string_equal_cstr(const string_t* cpt_string, const char* s_cstr);	判断 string_t 容器是否等于字符串常量 s_cstr。
bool_t string_not_equal_cstr(const string_t* cpt_string, const char* s_cstr);	判断 string_t 容器是否不等于字符串常量 s_cstr。
bool_t string_less_cstr(const string_t* cpt_string, const char* s_cstr);	判断 string_t 容器是否小于字符串常量 s_cstr。
bool_t string_less_equal_cstr(const string_t* cpt_string, const char* s_cstr);	判断 string_t 容器是否小于等于字符串常量 s_cstr。
bool_t string_great_cstr(const string_t* cpt_string, const char* s_cstr);	判断 string_t 容器是否大于字符串常量 s_cstr。
bool_t string_great_equal_cstr(const string_t* cpt_string, const char* s_cstr);	判断 string_t 容器是否大于等于字符串常量 s_cstr。
int string_compare(const string_t* cpt_first, const string_t* cpt_second);	比较两个 string_t 容器，根据比较返回三种结果，第一个 string_t 容器小于第二个 string_t 容器返回负数值，第一个 string_t 容器等于第二个 string_t 容器返回 0，第一个 string_t 容器大于第二个 string_t 容器返回正数值。
int string_compare_substring_string(const string_t* cpt_first, size_t t_pos, size_t t_len, const string_t* cpt_second);	比较第一个 string_t 容器的子串和第二个 string_t 容器，根据比较返回三种结果，第一个 string_t 容器的子串小于第二个 string_t 容器返回负数值，第一个 string_t 容器的子串等于第二个 string_t 容器返回 0，第一个 string_t 容器的子串大于第二个 string_t 容器返回正数值。
int string_compare_substring_substring(	比较第一个 string_t 容器的子串和第二个 string_t 容器

const string_t* cpt_first, size_t t_firstpos, size_t t_firstlen, const string_t* cpt_second, size_t t_secondpos, size_t t_secondlen);	的子串，根据比较返回三种结果，第一个子串小于第二个子串返回负数值，第一个子串等于第二个子串返回0，第一个子串大于第二个子串返回正数值。
int string_compare_cstr(const string_t* cpt_string, const char* s_cstr);	比较string_t容器和字符串常量，根据比较返回三种结果，string_t容器小于字符串常量返回负数值，string_t容器等于字符串常量返回0，string_t容器大于字符串常量返回正数值。
int string_compare_substring_cstr(const string_t* cpt_string, size_t t_pos, size_t t_len, const char* s_cstr);	比较string_t容器的子串和字符串常量，根据比较返回三种结果，string_t容器的子串小于字符串常量返回负数值，string_t容器的子串等于字符串常量返回0，string_t容器的子串大于字符串常量返回正数值。
int string_compare_substring_subcstr(const string_t* cpt_string, size_t t_pos, size_t t_len, const char* s_cstr, size_t t_cstrlen);	比较string_t容器的子串和字符串常量的子串，根据比较返回三种结果，string_t容器的子串小于字符串常量的子串返回负数值，string_t容器的子串等于字符串常量的子串返回0，string_t容器的子串大于字符串常量的子串返回正数值。
void string_assign(string_t* pt_string, const string_t* cpt_src);	使用另一个string_t容器为string_t容器赋值。
void string_assign_substring(string_t* pt_string, const string_t* cpt_src, size_t t_pos, size_t t_len);	使用另一个string_t容器的子串为string_t容器赋值。
void string_assign_cstr(string_t* pt_string, const char* s_cstr);	使用字符串常量为string_t容器赋值。
void string_assign_subcstr(string_t* pt_string, const char* s_cstr, size_t t_len);	使用字符串常量的子串为string_t容器赋值。
void string_assign_char(string_t* pt_string, size_t t_count, char c_char);	使用t_count个字符c_char为string_t容器赋值。
void string_assign_range(string_t* pt_string, string_iterator_t t_begin, string_iterator_t t_end);	使用数据区间[t_begin, t_end)为string_t容器赋值。[1][2]
void string_swap(string_t* pt_first, string_t* pt_second);	交换两个string_t容器的内容。
void string_append(string_t* pt_string, const string_t* cpt_src);	向string_t容器后添加string_t容器。
void string_append_substring(string_t* pt_string, const string_t* cpt_src, size_t t_pos, size_t t_len);	向string_t容器后添加string_t容器的子串。
void string_append_cstr(string_t* pt_string, const char* s_cstr);	向string_t容器后添加字符串常量。
void string_append_subcstr(string_t* pt_string, const char* s_cstr,	向string_t容器后添加字符串常量的子串。

size_t t_len);	
void string_append_char(string_t* pt_string, size_t t_count, char c_char);	向 string_t 容器后添加 t_count 个字符 c_char。
void string_append_range(string_t* pt_string, string_iterator_t t_begin, string_iterator_t t_end);	向 string_t 容器后添加数据区间[t_begin, t_end)。[1][2]
void string_connect(string_t* pt_string, const string_t* cpt_src);	将两个 string_t 容器连接在一起。
void string_connect_cstr(string_t* pt_string, const char* s_cstr);	将一个 string_t 容器和字符串常量连接在一起。
void string_connect_char(string_t* pt_string, char c_char);	将一个 string_t 容器和字符 c_char 连接在一起。
void string_push_back(string_t* pt_string, char c_char);	在 string_t 容器后面添加一个字符 c_char。
string_iterator_t string_insert(string_t* pt_string, string_iterator_t t_pos, char c_char);	在位置 t_pos 插入字符 c_char。
string_iterator_t string_insert_n(string_t* pt_string, string_iterator_t t_pos, size_t t_count, char c_char);	在位置 t_pos 插入 t_count 个字符 c_char。
void string_insert_range(string_t* pt_string, string_iterator_t t_pos, string_iterator_t t_begin, string_iterator_t t_end);	在 t_pos 前面插入数据区间[t_begin, t_end)。[1][2]
void string_insert_string(string_t* pt_string, size_t t_pos, const string_t* cpt_src);	在位置 t_pos 插入 string_t 容器。
void string_insert_substring(string_t* pt_string, size_t t_pos, const string_t* cpt_src, size_t t_startpos, size_t t_len);	在位置 t_pos 插入 string_t 容器的子串。
void string_insert_cstr(string_t* pt_string, size_t t_pos, const char* s_cstr);	在位置 t_pos 插入字符串常量。
void string_insert_subcstr(string_t* pt_string, size_t t_pos, const char* s_cstr, size_t t_len);	在位置 t_pos 插入字符串常量的子串。
void string_insert_char(string_t* pt_string, size_t t_pos, size_t t_count, char c_char);	在位置 t_pos 插入 t_count 个字符 c_char。
string_iterator_t string_erase (string_t* pt_string, string_iterator_t t_pos	删除 t_pos 位置的字符，返回下一个字符的迭代器。

);	
string_iterator_t string_erase_range(string_t* pt_string, string_iterator_t t_begin, string_iterator_t t_end);	删除数据区间[t_begin, t_end)的数据，并返回指向下一个数据的迭代器。[1]
void string_erase_substring(string_t* pt_string, size_t t_pos, size_t t_len);	删除 string_t 容器的子串。
void string_clear(string_t* pt_string);	清空 string_t 容器。
void string_replace(string_t* pt_string, size_t t_pos, size_t t_len, const string_t* cpt_src);	使用 string_t 容器替换子串。
void string_replace_substring(string_t* pt_string, size_t t_pos1, size_t t_len1, string_t* pt_src, size_t t_pos2, size_t t_len2);	使用 string_t 容器的子串替换子串。
void string_replace_cstr(string_t* pt_string, size_t t_pos, size_t t_len, const char* s_cstr);	使用字符串常量替换子串。
void string_replace_subcstr(string_t* pt_string, size_t t_pos, size_t t_len, const char* s_cstr, size_t t_length);	使用字符串常量的子串替换子串。
void string_replace_char(string_t* pt_string, size_t t_pos, size_t t_len, size_t t_count, char c_char);	使用 t_count 个字符 c_char 替换子串。
void string_range_replace(string_t* pt_string, string_iterator_t t_begin, string_iterator_t t_end, const string_t* cpt_src);	使用 string_t 容器替换数据区间[t_begin, t_end)。[1]
void string_range_replace_substring(string_t* pt_string, string_iterator_t t_begin, string_iterator_t t_end, const string_t* cpt_src, size_t t_pos, size_t t_len);	使用 string_t 容器的子串替换数据区间[t_begin, t_end)。[1]
void string_range_replace_cstr(string_t* pt_string, string_iterator_t t_begin, string_iterator_t t_end, const char* s_cstr);	使用字符串常量替换数据区间[t_begin, t_end)。[1]
void string_range_replace_subcstr(string_t* pt_string,	使用字符串常量的子串替换数据区间[t_begin, t_end)。[1]

string_iterator_t t_begin, string_iterator_t t_end, const char* s_cstr, size_t t_len);	
void string_range_replace_char(string_t* pt_string, string_iterator_t t_begin, string_iterator_t t_end, size_t t_count, char c_char);	使用 t_count 个字符 c_char 替换数据区间[t_begin, t_end)。[1]
void string_replace_range(string_t* pt_string, string_iterator_t t_begin1, string_iterator_t t_end1, string_iterator_t t_begin2, string_iterator_t t_end2);	使用数据区间[t_begin2, t_end2)替换数据区间[t_begin1, t_end1)。[1][2]
string_t string_substr(const string_t* cpt_string, size_t t_pos, size_t t_len);	返回 string_t 容器的子串。
void string_output(const string_t* cpt_string, FILE* fp_stream);	将 string_t 容器中的字符输出到流 fp_stream 中。
void string_input(string_t* pt_string, FILE* fp_stream);	从 fp_stream 流中获得字符并保存在 string_t 容器中。
bool_t string_getline(string_t* pt_string, FILE* fp_stream);	从 fp_stream 流中获得一行并保存在 string_t 容器中，返回操作是否成功。
bool_t string_getline_delimiter(string_t* pt_string, FILE* fp_stream, char c_delimiter);	从 fp_stream 流中获得一行并保存在 string_t 容器中，使用调用者定义的换行符 c_delimiter，返回操作是否成功。
size_t string_find(const string_t* cpt_string, const string_t* cpt_find, size_t t_pos);	从位置 t_pos 开始向后查找 cpt_find，成功返回 cpt_find 在 cpt_string 中出现的第一个位置，失败返回 NPOS。
size_t string_find_cstr(const string_t* cpt_string, const char* s_cstr, size_t t_pos);	从位置 t_pos 开始向后查找字符串常量 s_cstr，成功返回 s_cstr 在 cpt_string 中出现的第一个位置，失败返回 NPOS。
size_t string_find_subcstr(const string_t* cpt_string, const char* s_cstr, size_t t_pos, size_t t_len);	从位置 t_pos 开始向后查找字符串常量 s_cstr 的长度为 t_len 的子串，成功返回 s_cstr 的长度为 t_len 的子串在 cpt_string 中出现的第一个位置，失败返回 NPOS。
size_t string_find_char(const string_t* cpt_string, char c_char, size_t t_pos);	从位置 t_pos 开始向后查找字符 c_char，成功返回 c_char 在 cpt_string 中出现的第一个位置，失败返回 NPOS。
size_t string_rfind(const string_t* cpt_string, const string_t* cpt_find, size_t t_pos);	从位置 t_pos 开始向前查找 cpt_find，成功返回 cpt_find 在 cpt_string 中出现的最后一个位置，失败返回 NPOS。
size_t string_rfind_cstr(const string_t* cpt_string, const char* s_cstr, size_t t_pos);	从位置 t_pos 开始向前查找字符串常量 s_cstr，成功返回 s_cstr 在 cpt_string 中出现的最后一个位置，失败返回 NPOS。
size_t string_rfind_subcstr(const string_t* cpt_string,	从位置 t_pos 开始向前查找字符串常量 s_cstr 的长度为 t_len 的子串，成功返回 s_cstr 的长度为 t_len 的子串在

const char* s_cstr, size_t t_pos, size_t t_len);	cpt_string 中出现的最后一个位置，失败返回 NPOS。
size_t string_rfind_char(const string_t* cpt_string, char c_char, size_t t_pos);	从位置 t_pos 开始向前查找字符 c_char，成功返回 c_char 在 cpt_string 中出现的最后一个位置，失败返回 NPOS。
size_t string_find_first_of(const string_t* cpt_string, const string_t* cpt_find, size_t t_pos);	从位置 t_pos 开始向后查找 cpt_find 中包含的任意一个字符，成功返回这个字符在 cpt_string 中出现的第一个位置，失败返回 NPOS。
size_t string_find_first_of_cstr(const string_t* cpt_string, const char* s_cstr, size_t t_pos);	从位置 t_pos 开始向后查找字符串常量 s_cstr 中包含的任意一个字符，成功返回这个字符在 cpt_string 中出现的第一个位置，失败返回 NPOS。
size_t string_find_first_of_subcstr(const string_t* cpt_string, const char* s_cstr, size_t t_pos, size_t t_len);	从位置 t_pos 开始向后查找字符串常量 s_cstr 的长度为 t_len 的子串中包含的任意一个字符，成功返回这个字符在 cpt_string 中出现的第一个位置，失败返回 NPOS。
size_t string_find_first_of_char(const string_t* cpt_string, char c_char, size_t t_pos);	从位置 t_pos 开始向后查找字符 c_char，成功返回 c_char 在 cpt_string 中出现的第一个位置，失败返回 NPOS。
size_t string_find_last_of(const string_t* cpt_string, const string_t* cpt_find, size_t t_pos);	从位置 t_pos 开始向前查找 cpt_find 中包含的任意一个字符，成功返回这个字符在 cpt_string 中出现的最后一个位置，失败返回 NPOS。
size_t string_find_last_of_cstr(const string_t* cpt_string, const char* s_cstr, size_t t_pos);	从位置 t_pos 开始向前查找字符串常量 s_cstr 中包含的任意一个字符，成功返回这个字符在 cpt_string 中出现的最后一个位置，失败返回 NPOS。
size_t string_find_last_of_subcstr(const string_t* cpt_string, const char* s_cstr, size_t t_pos, size_t t_len);	从位置 t_pos 开始向前查找字符串常量 s_cstr 的长度为 t_len 的子串中包含的任意一个字符，成功返回这个字符在 cpt_string 中出现的最后一个位置，失败返回 NPOS。
size_t string_find_last_of_char(const string_t* cpt_string, char c_char, size_t t_pos);	从位置 t_pos 开始向前查找字符 c_char，成功返回 c_char 在 cpt_string 中出现的最后一个位置，失败返回 NPOS。
size_t string_find_first_not_of(const string_t* cpt_string, const string_t* cpt_find, size_t t_pos);	从位置 t_pos 开始向后查找 cpt_find 中不包含的任意一个字符，成功返回这个字符在 cpt_string 中出现的第一个位置，失败返回 NPOS。
size_t string_find_first_not_of_cstr(const string_t* cpt_string, const char* s_cstr, size_t t_pos);	从位置 t_pos 开始向后查找字符串常量 s_cstr 中不包含的任意一个字符，成功返回这个字符在 cpt_string 中出现的第一个位置，失败返回 NPOS。
size_t string_find_first_not_of_subcstr(const string_t* cpt_string, const char* s_cstr, size_t t_pos, size_t t_len);	从位置 t_pos 开始向后查找字符串常量 s_cstr 的长度为 t_len 的子串中不包含的任意一个字符，成功返回这个字符在 cpt_string 中出现的第一个位置，失败返回 NPOS。
size_t string_find_first_not_of_char(const string_t* cpt_string, char c_char, size_t t_pos);	从位置 t_pos 开始向后查找不是字符 c_char 的字符，成功返回这个字符在 cpt_string 中出现的第一个位置，失败返回 NPOS。
size_t string_find_last_not_of(const string_t* cpt_string, const string_t* cpt_find, size_t t_pos);	从位置 t_pos 开始向前查找 cpt_find 中不包含的任意一个字符，成功返回这个字符在 cpt_string 中出现的最后一个位置，失败返回 NPOS。

size_t string_find_last_not_of_cstr(const string_t* cpt_string, const char* s_cstr, size_t t_pos);	从位置 t_pos 开始向前查找字符串常量 s_cstr 中不包含的任意一个字符，成功返回这个字符在 cpt_string 中出现的最后一个位置，失败返回 NPOS。
size_t string_find_last_not_of_subcstr(const string_t* cpt_string, const char* s_cstr, size_t t_pos, size_t t_len);	从位置 t_pos 开始向前查找字符串常量 s_cstr 的长度为 t_len 的子串中不包含的任意一个字符，成功返回这个字符在 cpt_string 中出现的最后一个位置，失败返回 NPOS。
size_t string_find_last_not_of_char(const string_t* cpt_string, char c_char, size_t t_pos);	从位置 t_pos 开始向前查找不是字符 c_char 的字符，成功返回这个字符在 cpt_string 中出现的最后一个位置，失败返回 NPOS。
string_iterator_t string_begin(const string_t* cpt_string);	返回指向 string_t 容器开始的迭代器。
string_iterator_t string_end(const string_t* cpt_string);	返回指向 string_t 容器结尾的迭代器。

NOTE:
[1]:数据区间[t_begin, t_end)必须是有效的数据区间。
[2]:数据区间[t_begin, t_end)必须属于另一个 string_t 容器。

第五节 容器适配器

1. stack_t

TYPE:
stack_t

DESCRIPTION:
stack_t 是一个适配器，它只提供有限的操作，并且不支持迭代器。stack_t 支持插入删除数据，可以访问位于栈顶的数据，它是一个后进先去(LIFO)的数据结构：只能对栈顶进行插入删除和访问操作，栈内的其他数据都不能访问。stack_t 是一个迭代器，它实在容器基础上实现的。

DEFINITION:
<cstl/cstack.h>

OPERATION:

stack_t create_stack(type);	创建指定类型的 stack_t 容器适配器。
void stack_init(stack_t* pt_stack);	初始化一个空的 stack_t 容器适配器。
void stack_init_copy(stack_t* pt_stack, const stack_t* cpt_src);	使用令一个 stack_t 容器适配器初始化 stack_t 容器适配器。
void stack_destroy(stack_t* pt_stack);	销毁 stack_t 容器适配器。
size_t stack_size(const stack_t* cpt_stack);	获得 stack_t 容器适配器中数据的数目。
bool_t stack_empty(const stack_t* cpt_stack);	判断 stack_t 容器适配器是否为空。
bool_t stack_equal(const stack_t* cpt_first, const stack_t* cpt_second);	判断两个 stack_t 容器适配器是否相等。

bool_t stack_not_equal(const stack_t* cpt_first, const stack_t* cpt_second);	判断两个 stack_t 容器适配器是否不等。
bool_t stack_less(const stack_t* cpt_first, const stack_t* cpt_second);	判断第一个 stack_t 容器适配器是否小于第二个 stack_t 容器适配器。
bool_t stack_less_equal(const stack_t* cpt_first, const stack_t* cpt_second);	判断第一个 stack_t 容器适配器是否小于等于第二个 stack_t 容器适配器。
bool_t stack_great(const stack_t* cpt_first, const stack_t* cpt_second);	判断第一个 stack_t 容器适配器是否大于第二个 stack_t 容器适配器。
bool_t stack_great_equal(const stack_t* cpt_first, const stack_t* cpt_second);	判断第一个 stack_t 容器适配器是否大于等于第二个 stack_t 容器适配器。
void stack_assign(stack_t* pt_stack, const stack_t* cpt_src);	使用另一个 stack_t 容器适配器为当前 stack_t 容器适配器赋值。
void stack_push(stack_t* pt_stack, element);	将数据 element 插压入堆栈。
void stack_pop(stack_t* pt_stack);	将栈顶数据弹出堆栈。
void* stack_top(const stack_t* cpt_stack);	访问栈顶数据。

2. queue_t

TYPE:

queue_t

DESCRIPTION:

queue_t 是一个适配器，它只提供有限的操作，并且不支持迭代器。queue_t 是一个先进先去(FIFO)的数据结构：在队列末尾添加数据，从队列开头删除数据，同时可以访问队列开头和结尾两端的数据，队列内的其他数据都不能访问。queue_t 是一个迭代器，它实在容器基础上实现的。

DEFINITION:

<cstl/cqueue.h>

OPERATION:

queue_t create_queue(type);	创建指定类型的 queue_t 容器适配器。
void queue_init(queue_t* pt_queue);	初始化一个空的 queue_t 容器适配器。
void queue_init_copy(queue_t* pt_queue, const queue_t* cpt_src);	使用令一个 queue_t 容器适配器初始化 queue_t 容器适配器。
void queue_destroy(queue_t* pt_queue);	销毁 queue_t 容器适配器。
size_t queue_size(const queue_t* cpt_queue);	获得 queue_t 容器适配器中数据的数目。
bool_t queue_empty(const queue_t* cpt_queue);	判断 queue_t 容器适配器是否为空。
bool_t queue_equal(const queue_t* cpt_first, const queue_t* cpt_second);	判断两个 queue_t 容器适配器是否相等。

bool_t queue_not_equal(const queue_t* cpt_first, const queue_t* cpt_second);	判断两个 queue_t 容器适配器是否不等。
bool_t queue_less(const queue_t* cpt_first, const queue_t* cpt_second);	判断第一个 queue_t 容器适配器是否小于第二个 queue_t 容器适配器。
bool_t queue_less_equal(const queue_t* cpt_first, const queue_t* cpt_second);	判断第一个 queue_t 容器适配器是否小于等于第二个 queue_t 容器适配器。
bool_t queue_great(const queue_t* cpt_first, const queue_t* cpt_second);	判断第一个 queue_t 容器适配器是否大于第二个 deque_t 容器适配器。
bool_t queue_great_equal(const queue_t* cpt_first, const queue_t* cpt_second);	判断第一个 queue_t 容器适配器是否大于等于第二个 queue_t 容器适配器。
void queue_assign(queue_t* pt_queue, const queue_t* cpt_src);	使用另一个 queue_t 容器适配器为当前 queue_t 容器适配器赋值。
void queue_push(queue_t* pt_queue, element);	将数据 element 插入到 queue_t 容器适配器的末尾。
void queue_pop(queue_t* pt_queue);	删除 queue_t 容器适配器的第一个数据。
void* queue_front(const queue_t* cpt_queue);	访问 queue_t 容器适配器中的第一个数据。
void* queue_back(const queue_t* cpt_queue);	访问 queue_t 容器适配器中的最后一个数据。

3. priority_queue_t

TYPE:

priority_queue_t

DESCRIPTION:

priority_queue_t 是一个适配器，它只提供有限的操作，并且不支持迭代器。priority_queue_t 支持在队列末尾添加数据，从队列开头删除数据，同时可以访问队列开头的数据，队列内的其他数据都不能访问。
priority_queue_t 是优先队列，它保证队列开头的数据是优先级最高的数据，它还支持用户自定义的优先级函数。
priority_queue_t 是一个迭代器，它实在容器基础上实现的。

DEFINITION:

<cstl/cqueue.h>

OPERATION:

priority_queue_t create_priority_queue(type);	创建指定类型的 priority_queue_t 容器适配器。
void priority_queue_init(priority_queue_t* pt_priority_queue);	初始化一个空的 priority_queue_t 容器适配器。
void priority_queue_init_op(priority_queue_t* pt_priority_queue, binary_function_t t_binary_op);	使用用户自定义的优先级规则 t_binary_op 初始化一个空的 priority_queue_t 容器适配器。
void priority_queue_init_copy(priority_queue_t* pt_priority_queue,	使用令一个 priority_queue_t 容器适配器初始化 priority_queue_t 容器适配器。

const priority_queue_t* cpt_src);	
void priority_queue_init_copy_range(priority_queue_t* pt_priority_queue, random_access_iterator_t t_begin, random_access_iterator_t t_end);	使用数据区间[t_begin, t_end)初始化 priority_queue_t 容器适配器。
void priority_queue_init_copy_range_op(priority_queue_t* pt_priority_queue, random_access_iterator_t t_begin, random_access_iterator_t t_end, binary_function_t t_binary_op);	使用数据区间[t_begin, t_end)和用户自定义的优先级规则 t_binary_op 初始化 priority_queue_t 容器适配器。
void priority_queue_destroy(priority_queue_t* pt_priority_queue);	销毁 priority_queue_t 容器适配器。
size_t priority_queue_size(const priority_queue_t* cpt_priority_queue);	获得 priority_queue_t 容器适配器中数据的数目。
bool_t priority_queue_empty(const priority_queue_t* cpt_priority_queue);	判断 priority_queue_t 容器适配器是否为空。
void priority_queue_assign(priority_queue_t* pt_priority_queue, const priority_queue_t* cpt_src);	使用另一个 priority_queue_t 容器适配器为当前 priority_queue_t 容器适配器赋值。
void priority_queue_push(priority_queue_t* pt_priority_queue, element);	将数据 element 插入到 priority_queue_t 容器适配器的末尾。
void priority_queue_pop(priority_queue_t* pt_priority_queue);	删除 priority_queue_t 容器适配器的第一个数据(优先级最高)。
void* priority_queue_top(const priority_queue_t* cpt_priority_queue);	访问 priority_queue_t 容器适配器的第一个数据(优先级最高)。

第三章 迭代器

ITERATOR TYPE:

```
iterator_t
input_iterator_t
output_iterator_t
forward_iterator_t
bidirectional_iterator_t
random_access_iterator_t
```

DESCRIPTION:

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。

DEFINITION:

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

OPERATION:

void iterator_get_value(const iterator_t* cpt_iterator, void* pv_value);	获得迭代器 cpt_iterator 所指的数据。
void iterator_set_value(const iterator_t* cpt_iterator,	设置迭代器 cpt_iterator 所指的数据。

const void* cpt_value);	
const void* iterator_get_pointer(const iterator_t* cpt_iterator);	获得迭代器 cpt_iterator 所指的数据的指针。
void iterator_next(iterator_t* pt_iterator);	向下移动迭代器 pt_iterator，使它指向下一个数据。
void iterator_prev(iterator_t* pt_iterator);	向上移动迭代器 pt_iterator，使它指向上一个数据。
void iterator_next_n(iterator_t* pt_iterator, int n_step);	将迭代器 pt_iterator 向下移动 n_step 个数据位置。
void iterator_prev_n(iterator_t* pt_iterator, int n_step);	将迭代器 pt_iterator 向上移动 n_step 个数据位置。
bool_t iterator_equal(const iterator_t* cpt_iterator, iterator_t t_iterator);	判断另个迭代器类型是否相等。
bool_t iterator_less(const iterator_t* cpt_iterator, iterator_t t_iterator);	判断第一个迭代器是否小于第二个迭代器。
int iterator_minus(const iterator_t* cpt_iterator, iterator_t t_iterator);	求另个迭代器之间的距离差。
void* iterator_at(const iterator_t* cpt_iterator, size_t t_index);	通过下表随机访问迭代器指向的数据。
void iterator_advance(iterator_t* pt_iterator, int n_step);	一次移动迭代器 n_step 步。
int iterator_distance(iterator_t t_first, iterator_t t_second);	计算两个迭代器的距离。

第四章 算法

第一节 非质变算法

1. algo_for_each

PROTOTYPE：

```
void algo_for_each(input_iterator_t t_first, input_iterator_t t_last, unary_function_t t_unary_op);
```

DESCRIPTION:

algo_for_each()接受一元函数 t_unary_op 作为参数，对数据区间中[t_first, t_last)中每一个数据都执行这个一元函数，通常它的返回值是忽略的。

DEFINITION:

<cstl/calgorithm.h>

2. algo_find algo_find_if

PROTOTYPE:

```
input_iterator_t algo_find(input_iterator_t t_first, input_iterator_t t_last, element);
```

```
input_iterator_t algo_find_if(
    input_iterator_t t_first, input_iterator_t t_last, unary_function_t t_unary_op);
```

DESCRIPTION:

algo_find()查找数据区间中[t_first, t_last)中第一个数据值为 element 的数据的位置，没找到返回 t_last。

algo_find_if()查找数据区间中[t_first, t_last)中第一个满足一元谓词 t_unary_op 的数据，如果没找到返回 t_last。

DEFINITION:

<cstl/calgorithm.h>

3. algo_adjacent_find algo_adjacent_find_if

PROTOTYPE:

```
forward_iterator_t algo_adjacent_find(forward_iterator_t t_first, forward_iterator_t t_last);
```

```
forward_iterator_t algo_adjacent_find_if(
    forward_iterator_t t_first, forward_iterator_t t_last, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_adjacent_find()查找数据区间中[t_first, t_last)中第一个数据值与相邻的下一个数据相等的位置，没找到返回 t_last。

algo_adjacent_find_if()查找数据区间中[t_first, t_last)中第一个相邻两个数据满足二元谓词 t_binary_op 的位置，如果没找到返回 t_last。

DEFINITION:

<cstl/calgorithm.h>

4. algo_find_first_of algo_find_first_if

PROTOTYPE:

```
input_iterator_t algo_find_first_of(
    input_iterator_t t_first1, input_iterator_t t_last1,
    forward_iterator_t t_first2, forward_iterator_t t_last2);
```

```
input_iterator_t algo_find_first_of_if(
    input_iterator_t t_first1, input_iterator_t t_last1,
    forward_iterator_t t_first2, forward_iterator_t t_last2,
    binary_function_t t_binary_op);
```

DESCRIPTION:

algo_find_first_of()查找数据区间中[t_first1, t_last1)中第一个数据值与数据区间中[t_first2, t_last2)任意值相等的位置，没找到返回 t_last1。

algo_find_first_of_if()查找数据区间中[t_first1, t_last1)中第一个数据值与数据区间中[t_first2, t_last2)任意值满足二元谓词 t_binary_op 的位置，没找到返回 t_last1。

DEFINITION:

<cstl/calgorithm.h>

5. algo_count algo_count_if

PROTOTYPE:

```
size_t algo_count(input_iterator_t t_first, input_iterator_t t_last, element);
```

```
size_t algo_count_if(
    input_iterator_t t_first, input_iterator_t t_last, unary_function_t t_unary_op);
```

DESCRIPTION:

algo_count()返回数据区间中[t_first, t_last)中值等于 element 的数据的个数。

algo_count_if()返回数据区间中[t_first, t_last)中数据的值满足一元谓词 t_unary_op 的数据的个数。

DEFINITION:

<cstl/calgorithm.h>

6. algo_mismatch algo_mismatch_if

PROTOTYPE:

```
pair_t algo_mismatch(
    input_iterator_t t_first1, input_iterator_t t_last1, input_iterator_t t_first2);
```

```
pair_t algo_mismatch_if(
    input_iterator_t t_first1, input_iterator_t t_last1,
    input_iterator_t t_first2, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_mismatch()返回数据区间中[t_first1, t_last1)和[t_firs2, t_first2 + (t_last1 - t_first1))中的数据不相等的位置。

algo_mismatch_if()返回数据区间中[t_first1, t_last1)和[t_firs2, t_first2 + (t_last1 - t_first1))中的数据不符合二元谓词 t_binary_op 的位置。

DEFINITION:

<cstl/calgorithm.h>

7. algo_equal algo_equal_if

PROTOTYPE:

```
bool_t algo_equal(
    input_iterator_t t_first1, input_iterator_t t_last1, input_iterator_t t_first2);
```

```
bool_t algo_equal_if(
    input_iterator_t t_first1, input_iterator_t t_last1,
    input_iterator_t t_first2, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_equal()测试数据区间中[t_first1, t_last1)和[t_firs2, t_first2 + (t_last1 - t_first1))中的数据是否逐个相等。

algo_equal_if()测试数据区间中[t_first1, t_last1)和[t_firs2, t_first2 + (t_last1 - t_first1))中的数据是否逐个符合二元谓词 t_binary_op。

DEFINITION:

<cstl/calgorithm.h>

8. algo_search algo_search_if

PROTOTYPE:

```
forward_iterator_t algo_search(
    forward_iterator_t t_first1, forward_iterator_t t_last1,
    forward_iterator_t t_first2, forward_iterator_t t_last2);
```

```
forward_iterator_t algo_search_if(
    forward_iterator_t t_first1, forward_iterator_t t_last1,
    forward_iterator_t t_first2, forward_iterator_t t_last2,
    binary_function_t t_binary_op);
```

DESCRIPTION:

algo_search()在数据区间中[t_first1, t_last1)查找子串的第一个位置，这个子串和数据区间[t_firs2, t_last2)中的数据否逐个相等。

algo_search_if()在数据区间中[t_first1, t_last1)查找子串的第一个位置，这个子串和数据区间[t_firs2, t_last2)中的数据逐个符合二元谓词 t_binary_op。

DEFINITION:

<cstl/calgorithm.h>

9. algo_search_n algo_search_n_if

PROTOTYPE:

```
forward_iterator_t algo_search_n(
    forward_iterator_t t_first, forward_iterator_t t_last, size_t t_count, element);
```

```
forward_iterator_t algo_search_n_if(
    forward_iterator_t t_first, forward_iterator_t t_last, size_t t_count, element,
    binary_function_t t_binary_op);
```

DESCRIPTION:

algo_search_n()在数据区间中[t_first1, t_last1)查找子串的位置，这个子串由 t_count 个连续的。

algo_search_n_if()在数据区间中[t_first1, t_last1)查找子串的位置，这个子串和数据区间[t_firs2, t_last2)中的数据逐个符合二元谓词 t_binary_op。

DEFINITION:

<cstl/calgorithm.h>

10. algo_search_end algo_search_end_if algo_find_end algo_find_end_if

PROTOTYPE:

```
forward_iterator_t algo_search_end(
    forward_iterator_t t_first1, forward_iterator_t t_last1,
    forward_iterator_t t_first2, forward_iterator_t t_last2);
```

```
forward_iterator_t algo_search_end_if(
    forward_iterator_t t_first1, forward_iterator_t t_last1,
    forward_iterator_t t_first2, forward_iterator_t t_last2,
    binary_function_t t_binary_op);
```

```
forward_iterator_t algo_find_end(
```

```
    forward_iterator_t t_first1, forward_iterator_t t_last1,
    forward_iterator_t t_first2, forward_iterator_t t_last2);
forward_iterator_t algo_find_end_if(
    forward_iterator_t t_first1, forward_iterator_t t_last1,
    forward_iterator_t t_first2, forward_iterator_t t_last2,
    binary_function_t t_binary_op);
```

DESCRIPTION:

algo_search_end()在数据区间中[t_first1, t_last1)查找子串的最后一个位置，这个子串和数据区间[t_firs2, t_last2)中的数据否逐个相等。

algo_search_end_if()在数据区间中[t_first1, t_last1)查找子串的最后一个位置，这个子串和数据区间[t_firs2, t_last2)中的数据逐个符合二元谓词 t_binary_op。

algo_find_end()和 algo_find_end_if()与 algo_search_end()和 algo_search_end_if()功能相同，只是为了兼容 SGI STL 接口。

DEFINITION:

<cstl/calgorithm.h>

第二节 质变算法

1. algo_copy

PROTOTYPE:

```
output_iterator_t algo_copy(
    input_iterator_t t_first, input_iterator_t t_last, output_iterator_t t_result);
```

DESCRIPTION:

algo_copy()从数据区间中[t_first, t_last)将数据逐个拷贝到数据区间[t_result, t_result + (t_last - t_first))，并返回 t_result + (t_last - t_first)。

DEFINITION:

<cstl/calgorithm.h>

2. algo_copy_n

PROTOTYPE:

```
output_iterator_t algo_copy_n(
    input_iterator_t t_first, size_t t_count, output_iterator_t t_result);
```

DESCRIPTION:

algo_copy_n()从数据区间中[t_first, t_first + t_count)将数据逐个拷贝到数据区间[t_result, t_result + t_count)，并返回 t_result + t_count。

DEFINITION:

<cstl/calgorithm.h>

3. algo_copy_backward

PROTOTYPE:

```
bidirectional_iterator_t algo_copy_backward(
    bidirectional_iterator_t t_first, bidirectional_iterator_t t_last,
    bidirectional_iterator_t t_result);
```

DESCRIPTION:

algo_copy_backward()从数据区间中[t_first, t_last)将数据逐个拷贝到数据区间[t_result - (t_last - t_first), t_result)，并返回t_result - (t_last - t_first)。

DEFINITION:

<cstl/calgorithm.h>

4. algo_swap algo_iter_swap

PROTOTYPE:

```
void algo_swap(forward_iterator_t t_first, forward_iterator_t t_last);
```

```
void algo_iter_swap(forward_iterator_t t_first, forward_iterator_t t_last);
```

DESCRIPTION:

algo_swap()和algo_iter_swap()交换两个迭代器指向的数据的值。

DEFINITION:

<cstl/calgorithm.h>

5. algo_swap_ranges

PROTOTYPE:

```
forward_iterator_t algo_swap_ranges(
    forward_iterator_t t_first1, forward_iterator_t t_last1, forward_iterator_t t_first2);
```

DESCRIPTION:

algo_swap_ranges()逐一的交换两个数据区间[t_first1, t_last1)和[t_first2, t_first2 + (t_last1 - t_first1))中的数据，
并返回t_first2 + (t_last1 - t_first1)。

DEFINITION:

<cstl/calgorithm.h>

6. algo_transform algo_transform_binary

PROTOTYPE:

```
output_iterator_t algo_transform(
    input_iterator_t t_first, input_iterator_t t_last,
    output_iterator_t t_result, unary_function_t t_unary_op);
```

```
output_iterator_t algo_transform_binary(
    input_iterator_t t_first1, input_iterator_t t_last1, input_iterator_t t_first2
```

```
    output_iterator_t t_result, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_transform()将数据区间[t_first, t_last)中的数据逐一的通过一元函数 t_unary_op 转换将转换的结果保存在数据区间[t_result, t_result + (t_last - t_first))中，并返回 t_result + (t_last - t_first)。

algo_transform_binary()将数据区间[t_first1, t_last1)和[t_first2, t_first2 + (t_last1 - t_first1))中的数据逐一的通过二元函数 t_binary_op 转换将转换的结果保存在数据区间[t_result, t_result + (t_last1 - t_first1))中，并返回 t_result + (t_last1 - t_first1)。

DEFINITION:

<cstl/calgorithm.h>

7. algo_replace algo_replace_if algo_replace_copy algo_replace_copy_if

PROTOTYPE:

```
void algo_replace(forward_iterator_t t_first, forward_iterator_t t_last, old_element, new_element);
```

```
void algo_replace_if(
    forward_iterator_t t_first, forward_iterator_t t_last,
    unary_function_t t_unary_op, new_element);
```

```
void algo_replace_copy(
    input_iterator_t t_first, input_iterator_t t_last, output_iterator_t t_result,
    old_element, new_element);
```

```
output_iterator_t algo_replace_copy_if(
    input_iterator_t t_first, input_iterator_t t_last, output_iterator_t t_result,
    unary_function_t t_unary_op, new_element);
```

DESCRIPTION:

algo_replace()将数据区间[t_first, t_last)中所有值等于 old_element 的数据都替换成 new_element。

algo_replace_if()将数据区间[t_first, t_last)中所有值满足一元谓词 t_unary_op 的数据都替换成 new_element。

algo_replace_copy()将数据区间[t_first, t_last)中所有值等于 old_element 的数据都替换成 new_element，并将结果拷贝到数据区间[t_result, t_result + (t_last - t_first))。

algo_replace_copy_if()将数据区间[t_first, t_last)中所有值满足一元谓词 t_unary_op 的数据都替换成 new_element，并将结果拷贝到数据区间[t_result, t_result + (t_last - t_first))，同时返回 t_result + (t_last - t_first)。

DEFINITION:

<cstl/calgorithm.h>

8. algo_fill algo_fill_n

PROTOTYPE:

```
void algo_fill(forward_iterator_t t_first, forward_iterator_t t_last, element);
```

```
output_iterator_t algo_fill_n(output_iterator_t t_first, size_t t_count, element);
```

DESCRIPTION:

algo_fill()使用数据 element 填充数据区间[t_first, t_last)。

algo_fill_n()使用数据 element 填充数据区间[t_first, t_first + t_count)，并返回 t_first +

t_count。

DEFINITION:

<cstl/calgorithm.h>

9. algo_generate algo_generate_n

PROTOTYPE:

```
void algo_generate(
    forward_iterator_t t_first, forward_iterator_t t_last, unary_function_t t_unary_op);
```

```
output_iterator_t algo_generate_n(
    output_iterator_t t_first, size_t t_count, unary_function_t t_unary_op);
```

DESCRIPTION:

algo_generate()使用一元函数 t_unary_op 产生的数据填充数据区间[t_first, t_last)。

algo_generate_n()使用一元函数 t_unary_op 产生的数据填充数据区间[t_first, t_first + t_count)，并返回 t_first + t_count。

DEFINITION:

<cstl/calgorithm.h>

10. algo_remove algo_remove_if algo_remove_copy algo_remove_copy_if

PROTOTYPE:

```
forward_iterator_t algo_remove(forward_iterator_t t_first, forward_iterator_t t_last, element);
```

```
forward_iterator_t algo_remove_if(
    forward_iterator_t t_first, forward_iterator_t t_last, unary_function_t t_unary_op);
```

```
output_iterator_t algo_remove_copy(
    input_iterator_t t_first, input_iterator_t t_last, output_iterator_t t_result, element);
```

```
output_iterator_t algo_remove_copy_if(
    input_iterator_t t_first, input_iterator_t t_last, output_iterator_t t_result,
    unary_function_t t_unary_op);
```

DESCRIPTION:

algo_remove()移除数据区间[t_first, t_last)中所有等于 element 的数据，返回新结尾的位置迭代器 t_new_last，数据区间[t_first, t_new_last)中的数据都不等于 element，数据区间[t_new_last, t_last)是移除 element 后留下的垃圾数据。

algo_remove_if()移除数据区间[t_first, t_last)中所有满足一元谓词 t_unary_op 的数据，返回新结尾的位置迭代器 t_new_last，数据区间[t_first, t_new_last)中的数据都不满足一元谓词 t_unary_op，数据区间[t_new_last, t_last)是移除数据后留下的垃圾数据。

algo_remove_copy()将数据区间[t_first, t_last)中不等于 element 的数据拷贝到以 t_result 开始的数据区间，并返结果数据区间的结尾。

algo_remove_copy_if()将数据区间[t_first, t_last)中不满足一元谓词 t_unary_op 的数据拷贝到以 t_result 开始的数据区间，并返结果数据区间的结尾。

DEFINITION:

<cstl/calgorithm.h>

11. algo_unique algo_unique_if algo_unique_copy algo_unique_copy_if

PROTOTYPE:

```
forward_iterator_t algo_unique(forward_iterator_t t_first, forward_iterator_t t_last);
```

```
forward_iterator_t algo_unique_if(
    forward_iterator_t t_first, forward_iterator_t t_last, binary_function_t t_binary_op);
```

```
output_iterator_t algo_unique_copy(
    input_iterator_t t_first, input_iterator_t t_last, output_iterator_t t_result);
```

```
output_iterator_t algo_unique_copy_if(
    input_iterator_t t_first, input_iterator_t t_last, output_iterator_t t_result,
    binary_function_t t_binary_op);
```

DESCRIPTION:

algo_unique()找到数据区间[t_first, t_last)中连续重复的数据，并移除除了第一个以外的所有数据，返回新结尾的位置迭代器 t_new_last，数据区间[t_first, t_new_last)中的数据连续的位置不包含重复的数据，数据区间[t_new_last, t_last)是移除重复数据后留下的垃圾数据。

algo_unique_if()找到数据区间[t_first, t_last)中连复的满足二元谓词 t_binary_op 的数据，并移除除了第一个以外的所有数据 ，返回新结尾的位置迭代器 t_new_last，数据区间[t_first, t_new_last)中的数据连续的位置不包含满足二元谓词 t_binary_op 的数据，数据区间[t_new_last, t_last)是移除数据后留下的垃圾数据。

algo_unique_copy()将数据区间[t_first, t_last)中不是连续重复的数据拷贝到以 t_result 开始的数据区间，当遇到连续重复的数据时只拷贝第一个数据，并返结果数据区间的结尾。

algo_unique_copy_if()将数据区间[t_first, t_last)中不是连续满足二元谓词 t_binary_op 的数据拷贝到以 t_result 开始的数据区间，当遇到连续满足二元谓词 t_binary_op 的数据时只拷贝第一个数据，并返结果数据区间的结尾。

DEFINITION:

<cstl/calgorithm.h>

12. algo_reverse algo_reverse_copy

PROTOTYPE:

```
void algo_reverse(bidirectional_iterator_t t_first, bidirectional_iterator_t t_last);
```

```
output_iterator_t algo_reverse_copy(
    bidirectional_iterator_t t_first, bidirectional_iterator_t t_last, output_iterator_t t_result);
```

DESCRIPTION:

algo_reverse()将数据区间[t_first, t_last)中的数据逆序。

algo_reverse_copy()将数据区间[t_first, t_last)中的数据逆序，将逆序结果拷贝到以 t_result 开头的数据区间，并返回数据区间的结尾。

DEFINITION:

<cstl/calgorithm.h>

13. algo_rotate algo_rotate_copy

PROTOTYPE:

```
forward_iterator_t algo_rotate(
    forward_iterator_t t_first, forward_iterator_t t_middle, forward_iterator_t t_last);
```

```
output_iterator_t algo_rotate_copy(
    forward_iterator_t t_first, forward_iterator_t t_middle, forward_iterator_t t_last,
    output_iterator_t t_result);
```

DESCRIPTION:

algo_rotate()将数据区间[t_first, t_last)的两部分[t_first, t_middle)和[t_middle, t_last)的数据交换，返回新的中间位置。

algo_rotate_copy()将数据区间[t_first, t_last)的两部分[t_first, t_middle)和[t_middle, t_last)的数据交换，将交换后的结果拷贝到以 t_result 开头的数据区间，并返回数据区间的结尾。

DEFINITION:

<cstl/calgorithm.h>

14. algo_random_shuffle algo_random_shuffle_if

PROTOTYPE:

```
void algo_random_shuffle(random_access_iterator_t t_first, random_access_iterator_t t_last);
```

```
void algo_random_shuffle_if(
    random_access_iterator_t t_first, random_access_iterator_t t_last,
    unary_function_t t_unary_op);
```

DESCRIPTION:

algo_random_shuffle()将数据区间[t_first, t_last)中的数据随机重排。

algo_random_shuffle_if()使用一元随机函数 t_unary_op 将数据区间[t_first, t_last)中的数据随机重排。

DEFINITION:

<cstl/calgorithm.h>

15. algo_random_sample algo_random_sample_if algo_random_sample_n algo_random_sample_n_if

PROTOTYPE:

```
random_access_iterator_t algo_random_sample(
    input_iterator_t t_first1, input_iterator_t t_last1,
    random_access_iterator_t t_first2, random_access_iterator_t t_last2);
```

```
random_access_iterator_t algo_random_sample_if(
    input_iterator_t t_first1, input_iterator_t t_last1,
    random_access_iterator_t t_first2, random_access_iterator_t t_last2,
    unary_function_t t_unary_op);
```

```
output_iterator_t algo_random_sample_n(
    input_iterator_t t_first1, input_iterator_t t_last1,
    output_iterator_t t_first2, size_t t_count);
```

```
output_iterator_t algo_random_sample_n_if(
    input_iterator_t t_first1, input_iterator_t t_last1,
    output_iterator_t t_first2, size_t t_count,
    unary_function_t t_unary_op);
```

DESCRIPTION:

algo_random_sample()对数据区间[t_first1, t_last1)进行随机抽样，将结果拷贝到[t_first2, t_last2)

中，[t_first1, t_last1)中的任意一个数据在[t_first2, t_last2)中只出现一次，返回 t_first2 + n 其中 n 是(t_last1 - t_first1)和(t_last2 - t_first2)的最小值。

algo_random_sample_if()使用一元随机函数 t_unary_op 对数据区间[t_first1, t_last1)进行随机抽样，将结果拷贝到[t_first2, t_last2)中，[t_first1, t_last1)中的任意一个数据在[t_first2, t_last2)中只出现一次，返回 t_first2 + n 其中 n 是(t_last1 - t_first1)和(t_last2 - t_first2)的最小值。

algo_random_sample_n()对数据区间[t_first1, t_last1)进行随机抽样，将结果拷贝到[t_first2, t_first2 + t_count)中，[t_first1, t_last1)中的任意一个数据在[t_first2, t_first2 + t_count)中只出现一次，返回 t_first2 + n 其中 n 是(t_last1 - t_first1)和 t_count 的最小值。

algo_random_sample_n_if()使用一元随机函数 t_unary_op 对数据区间[t_first1, t_last1)进行随机抽样，将结果拷贝到[t_first2, t_first2 + t_count)中，[t_first1, t_last1)中的任意一个数据在[t_first2, t_first2 + t_count)中只出现一次，返回 t_first2 + n 其中 n 是(t_last1 - t_first1)和 t_count 的最小值。

DEFINITION:

<cstl/calgorithm.h>

16. algo_partition algo_stable_partition

PROTOTYPE:

```
forward_iterator_t algo_partition(
    forward_iterator_t t_first, forward_iterator_t t_last, unary_function_t t_unary_op);
```

```
forward_iterator_t algo_stable_partition(
    forward_iterator_t t_first, forward_iterator_t t_last, unary_function_t t_unary_op);
```

DESCRIPTION:

algo_partition()将数据区间[t_first, t_last)划分成两个部分[t_first, t_middle)和[t_middle, t_last)，所有满足一元谓词的数据都在[t_first, t_middle)中，其余的数据在[t_middle, t_last)中，并返回 t_middle。

algo_stable_partition()是数据顺序稳定版本的 algo_partition()。

DEFINITION:

<cstl/calgorithm.h>

第三节 排序算法

1. algo_sort algo_sort_if algo_stable_sort algo_stable_sort_if algo_is_sorted algo_is_sorted_if

PROTOTYPE:

```
void algo_sort(random_access_iterator_t t_first, random_access_iterator_t t_last);
```

```
void algo_sort_if(
    random_access_iterator_t t_first, random_access_iterator_t t_last,
    binary_function_t t_binary_op);
```

```
void algo_stable_sort(random_access_iterator_t t_first, random_access_iterator_t t_last);
```

```
void algo_stable_sort_if(
    random_access_iterator_t t_first, random_access_iterator_t t_last,
```

```
    binary_function_t t_binary_op);
bool_t algo_is_sorted(forward_iterator_t t_first, forward_iterator_t t_last);
bool_t algo_is_sorted_if(
    forward_iterator_t t_first, forward_iterator_t t_last, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_sort()将数据区间[t_first, t_last)中的数据排序，默认使用小于关系排序。

algo_sort_if()将数据区间[t_first, t_last)中的数据排序，使用用户定义二元的比较关系函数t_binary_op。

algo_stable_sort()数据顺序稳定版的algo_sort()。

algo_stable_sort_if()数据顺序稳定版的algo_sort_if()。

algo_is_sorted()判断数据区间[t_first, t_last)是否有序。

algo_is_sorted_if()依据用户定义的二元比较关系函数t_binary_op判断数据区间[t_first, t_last)是否有序。

DEFINITION:

<cstl/calgorithm.h>

2. algo_partial_sort algo_partial_sort_if algo_parital_sort_copy algo_partial_sort_copy_if

PROTOTYPE:

```
void algo_partial_sort(
    random_access_iterator_t t_first, random_access_iterator_t t_middle,
    random_access_iterator_t t_last);
void algo_partial_sort_if(
    random_access_iterator_t t_first, random_access_iterator_t t_middle,
    random_access_iterator_t t_last, binary_function_t t_binary_op);
random_access_iterator_t algo_partial_sort_copy(
    input_iterator_t t_first1, input_iterator_t t_last1,
    random_access_iterator_t t_first2, random_access_iterator_t t_last2);
random_access_iterator_t algo_partial_sort_copy_if(
    input_iterator_t t_first1, input_iterator_t t_last1,
    random_access_iterator_t t_first2, random_access_iterator_t t_last2
    binary_function_t t_binary_op);
```

DESCRIPTION:

algo_partial_sort()将数据区间[t_first, t_last)中的重新排序，排序后保证[t_first, t_middle)中的数据与使用algo_sort()排序后的结果相同，[t_middle, t_last)不保证有序。

algo_partial_sort_if()依据用户定义的二元比较关系函数t_binary_op将数据区间[t_first, t_last)中的重新排序，排序后保证[t_first, t_middle)中的数据与使用algo_sort_if()排序后的结果相同，[t_middle, t_last)不保证有序。

algo_partial_sort_copy()将数据区间[t_first1, t_last1)中排序后的n个数据拷贝到数据区间[t_first2, t_first2 + n)中，其中n是(t_last1 - t_first1)和(t_last2 - t_first2)的最小值，并返回t_first2 + n。

algo_partial_sort_copy_if()将数据区间[t_first1, t_last1)中依据用户定义的二元比较关系函数t_binary_op排序后的n个数据拷贝到数据区间[t_first2, t_first2 + n)中，其中n是(t_last1 - t_first1)和(t_last2 - t_first2)的最小值，并返回t_first2 + n。

DEFINITION:

<cstl/calgorithm.h>

3. algo_nth_element algo_nth_element_if

PROTOTYPE:

```
void algo_nth_element(
    random_access_iterator_t t_first, random_access_iterator_t t_nth,
    random_access_iterator_t t_last);
```

```
void algo_nth_element_if(
    random_access_iterator_t t_first, random_access_iterator_t t_nth,
    random_access_iterator_t t_last, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_nth_element()将数据区间[t_first, t_last)中的重新排序，排序后保证 t_nth 所指的数据与使用 algo_sort()排序后的结果相同，同时[t_first, t_nth)都小于 t_nth，[t_nth + 1, t_last)都不小于 t_nth 但是不保证这两个区间有序。

algo_nth_element_if()依据用户定义的二元比较关系函数 t_binary_op 将数据区间[t_first, t_last)中的重新排序，排序后保证 t_nth 所指的数据与使用 algo_sort_if()排序后的结果相同，同时依据用户定义的二元比较关系函数 t_binary_op[t_first, t_nth)都小于 t_nth，[t_nth + 1, t_last)都不小于 t_nth 但是不保证这两个区间有序。

DEFINITION:

<cstl/calgorithm.h>

4. algo_lower_bound algo_lower_bound_if

PROTOTYPE:

```
forward_iterator_t algo_lower_bound(
    forward_iterator_t t_first, forward_iterator_t t_last, element);
```

```
forward_iterator_t algo_lower_bound_if(
    forward_iterator_t t_first, forward_iterator_t t_last, element, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_lower_bound()获得有序的数据区间[t_first, t_last)中第一个不小于 element 的数据迭代器，没找到返回 t_last。

algo_lower_bound_if()获得依据用户定义的二元比较关系函数 t_binary_op 有序的数据区间[t_first, t_last)中第一个不小于 element 的数据迭代器，没找到返回 t_last。

DEFINITION:

<cstl/calgorithm.h>

5. algo_upper_bound algo_upper_bound_if

PROTOTYPE:

```
forward_iterator_t algo_upper_bound(
    forward_iterator_t t_first, forward_iterator_t t_last, element);
```

```
forward_iterator_t algo_upper_bound_if(
    forward_iterator_t t_first, forward_iterator_t t_last, element, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_upper_bound()获得有序的数据区间[t_first, t_last)中第一个大于 element 的数据迭代器，没找到返回 t_last。

algo_upper_bound_if()获得依据用户定义的二元比较关系函数 t_binary_op 有序的数据区间[t_first, t_last)中第一个大于 element 的数据迭代器，没找到返回 t_last。

DEFINITION:

<cstl/calgorithm.h>

6. algo_equal_range algo_equal_range_if

PROTOTYPE:

```
pair_t algo_equal_range(
    forward_iterator_t t_first, forward_iterator_t t_last, element);
```

```
pair_t algo_equal_range_if(
    forward_iterator_t t_first, forward_iterator_t t_last, element, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_equal_range()获得有序的数据区间[t_first, t_last)中所有等于 element 的数据的区间，没有找到返回(t_last, t_last)。

algo_equal_range_if()获得依据用户定义的二元比较关系函数 t_binary_op 有序的数据区间[t_first, t_last)中所有等于 element 的数据的区间，没有找到返回(t_last, t_last)。

DEFINITION:

<cstl/calgorithm.h>

7. algo_binary_search algo_binary_search_if

PROTOTYPE:

```
bool_t algo_binary_search(
    forward_iterator_t t_first, forward_iterator_t t_last, element);
```

```
bool_t algo_binary_search_if(
    forward_iterator_t t_first, forward_iterator_t t_last, element, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_binary_search()在有序的数据区间[t_first, t_last)中查找值为 element 的数据。

algo_binary_search_if()在依据用户定义的二元比较关系函数 t_binary_op 有序的数据区间[t_first, t_last)中查找值为 element 的数据。

DEFINITION:

<cstl/calgorithm.h>

8. algo_merge algo_merge_if

PROTOTYPE:

```
output_iterator_t algo_merge(
    input_iterator_t t_first1, input_iterator_t t_last1,
```

```
    input_iterator_t t_first2, input_iterator_t t_last2, output_iterator_t t_result);
```

```
output_iterator_t algo_merge_if(
    input_iterator_t t_first1, input_iterator_t t_last1,
    input_iterator_t t_first2, input_iterator_t t_last2,
    output_iterator_t t_result, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_merge()将两个有序的数据区间[t_first1, t_last1)和[t_first2, t_last2)合并到[t_result, t_result + (t_last1 - t_first1) + (t_last2 - t_first2))中，合并后的数据区间仍然有序，并返回[t_result, t_result + (t_last1 - t_first1) + (t_last2 - t_first2))。

algo_merge_if()将两个依据用户定义的二元比较关系函数 t_binary_op 有序的数据区间[t_first1, t_last1)和[t_first2, t_last2)合并到[t_result, t_result + (t_last1 - t_first1) + (t_last2 - t_first2))中，合并后的数据区间仍然依据用户定义的二元比较关系函数 t_binary_op 有序，并返回[t_result, t_result + (t_last1 - t_first1) + (t_last2 - t_first2))。

DEFINITION:

<cstl/calgorithm.h>

9. algo_inplace_merge algo_inplace_merge_if

PROTOTYPE:

```
void algo_inplace_merge(
    bidirectional_iterator_t t_first, bidirectional_iterator_t t_middle,
    bidirectional_iterator_t t_last);
```

```
void algo_inplace_merge_if(
    bidirectional_iterator_t t_first, bidirectional_iterator_t t_middle,
    bidirectional_iterator_t t_last, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_inplace_merge()将数据区间[t_first, t_last)的两个有序的部分[t_first, t_middle)和[t_middle, t_las)合并，合并后整个数据区间[t_first, t_last)有序。

algo_inplace_merge_if()将数据区间[t_first, t_last)的两个依据用户定义的二元比较关系函数 t_binary_op 有序的部分[t_first, t_middle)和[t_middle, t_las)合并，合并后整个数据区间[t_first, t_last)依据用户定义的二元比较关系函数 t_binary_op 有序。

DEFINITION:

<cstl/calgorithm.h>

10. algo_includes algo_includes_if

PROTOTYPE:

```
bool_t algo_includes(
    input_iterator_t t_first1, input_iterator_t t_last1,
    input_iterator_t t_first2, input_iterator_t t_last2);
```

```
bool_t algo_includes_if(
    input_iterator_t t_first1, input_iterator_t t_last1,
    input_iterator_t t_first2, input_iterator_t t_last2, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_includes()测试是否第二个有序的数据区间[t_first2, t_last2)中的所有数据都出现在第一个有序的数据区间[t_first1, t_last1)中，两个有序区间都使用默认的小于关系排序。

algo_includes_if()测试是否第二个有序的数据区间[t_first2, t_last2)中的所有数据都出现在第一个有序的数据区间[t_first1, t_last1)中，两个有序区间都使用用户定义的二元比较关系函数 t_binary_op 排序。

DEFINITION:

<cstl/calgorithm.h>

11. algo_set_union algo_set_union_if

PROTOTYPE:

```
output_iterator_t algo_set_union(
    input_iterator_t t_first1, input_iterator_t t_last1,
    input_iterator_t t_first2, input_iterator_t t_last2, output_iterator_t t_result);
```

```
output_iterator_t algo_set_union_if(
    input_iterator_t t_first1, input_iterator_t t_last1,
    input_iterator_t t_first2, input_iterator_t t_last2,
    output_iterator_t t_result, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_set_union()求两个有序区间[t_first1, t_last1)和[t_first2, t_last2)的并集，把结果拷贝到以 t_result 开头的数据区间，并返回数据区间的末尾，两个有序区间都使用默认的小于关系排序。

algo_set_union_if()求两个有序区间[t_first1, t_last1)和[t_first2, t_last2)的并集，把结果拷贝到以 t_result 开头的数据区间，并返回数据区间的末尾，两个有序区间都使用用户定义的二元比较关系函数 t_binary_op 排序。

DEFINITION:

<cstl/calgorithm.h>

12. algo_set_intersection algo_set_intersection_if

PROTOTYPE:

```
output_iterator_t algo_set_intersection(
    input_iterator_t t_first1, input_iterator_t t_last1,
    input_iterator_t t_first2, input_iterator_t t_last2, output_iterator_t t_result);
```

```
output_iterator_t algo_set_intersection_if(
    input_iterator_t t_first1, input_iterator_t t_last1,
    input_iterator_t t_first2, input_iterator_t t_last2,
    output_iterator_t t_result, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_set_intersection()求两个有序区间[t_first1, t_last1)和[t_first2, t_last2)的交集，把结果拷贝到以 t_result 开头的数据区间，并返回数据区间的末尾，两个有序区间都使用默认的小于关系排序。

algo_set_intersection_if()求两个有序区间[t_first1, t_last1)和[t_first2, t_last2)的交集，把结果拷贝到以 t_result 开头的数据区间，并返回数据区间的末尾，两个有序区间都使用用户定义的二元比较关系函数 t_binary_op 排序。

DEFINITION:

<cstl/calgorithm.h>

13. algo_set_difference algo_set_difference_if

PROTOTYPE:

```
output_iterator_t algo_set_difference(
    input_iterator_t t_first1, input_iterator_t t_last1,
    input_iterator_t t_first2, input_iterator_t t_last2, output_iterator_t t_result);
```

```
output_iterator_t algo_set_difference_if(
    input_iterator_t t_first1, input_iterator_t t_last1,
    input_iterator_t t_first2, input_iterator_t t_last2,
    output_iterator_t t_result, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_set_difference()求两个有序区间[t_first1, t_last1)和[t_first2, t_last2)的差集，把结果拷贝到以 t_result 开头的数据区间，并返回数据区间的末尾，两个有序区间都使用默认的小于关系排序。

algo_set_difference_if()求两个有序区间[t_first1, t_last1)和[t_first2, t_last2)的差集，把结果拷贝到以 t_result 开头的数据区间，并返回数据区间的末尾，两个有序区间都使用用户定义的二元比较关系函数 t_binary_op 排序。

DEFINITION:

<cstl/calgorithm.h>

14. algo_set_symmetric_difference algo_set_symmetric_difference_if

PROTOTYPE:

```
output_iterator_t algo_set_symmetric_difference(
    input_iterator_t t_first1, input_iterator_t t_last1,
    input_iterator_t t_first2, input_iterator_t t_last2, output_iterator_t t_result);
```

```
output_iterator_t algo_set_symmetric_difference_if(
    input_iterator_t t_first1, input_iterator_t t_last1,
    input_iterator_t t_first2, input_iterator_t t_last2,
    output_iterator_t t_result, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_set_symmetric_difference()求两个有序区间[t_first1, t_last1)和[t_first2, t_last2)的对称差集，把结果拷贝到以 t_result 开头的数据区间，并返回数据区间的末尾，两个有序区间都使用默认的小于关系排序。

algo_set_symmetric_difference_if()求两个有序区间[t_first1, t_last1)和[t_first2, t_last2)的对称差集，把结果拷贝到以 t_result 开头的数据区间，并返回数据区间的末尾，两个有序区间都使用用户定义的二元比较关系函数 t_binary_op 排序。

DEFINITION:

<cstl/calgorithm.h>

15. algo_push_heap algo_push_heap_if

PROTOTYPE:

```
void algo_push_heap(random_access_iterator_t t_first, random_access_iterator_t t_last);
```

```
void algo_push_heap_if(
    random_access_iterator_t t_first, random_access_iterator_t t_last,
    binary_function_t t_binary_op);
```

DESCRIPTION:

algo_push_heap()将 t_last 指向的数据插入到堆[t_first, t_last - 1)中，使[t_first, t_last)成为一个有效的堆，数据区间[t_first, t_last - 1)是已经使用默认的小于关系建立起来的堆。

algo_push_heap_if()将 t_last 指向的数据插入到堆[t_first, t_last - 1)中，使[t_first, t_last)成为一个有效的堆，数据区间[t_first, t_last - 1)是已经使用用户定义的二元比较关系函数 t_binary_op 建立起来的堆。

DEFINITION:

<cstl/calgorithm.h>

16. algo_pop_heap algo_pop_heap_if

PROTOTYPE:

```
void algo_pop_heap(random_access_iterator_t t_first, random_access_iterator_t t_last);
```

```
void algo_pop_heap_if(
    random_access_iterator_t t_first, random_access_iterator_t t_last,
    binary_function_t t_binary_op);
```

DESCRIPTION:

algo_pop_heap()将堆[t_first, t_last)中优先级最高的数据 t_first 从堆中删除，并放在最后 t_last 的位置，同时调整[t_first, t_last - 1)使它这个数据区间成为一个有效的堆。

algo_pop_heap_if()将堆[t_first, t_last)中优先级最高的数据 t_first 从堆中删除，并放在最后 t_last 的位置，同时调整[t_first, t_last - 1)使它这个数据区间成为一个有效的堆。algo_pop_heap_if()使用用户定义的二元比较关系函数 t_binary_op 建立堆。

DEFINITION:

<cstl/calgorithm.h>

17. algo_make_heap algo_make_heap_if

PROTOTYPE:

```
void algo_make_heap(random_access_iterator_t t_first, random_access_iterator_t t_last);
```

```
void algo_make_heap_if(
    random_access_iterator_t t_first, random_access_iterator_t t_last,
    binary_function_t t_binary_op);
```

DESCRIPTION:

algo_make_heap()使用默认的小于关系把数据区间[t_first, t_last)建立成有效的堆。

algo_make_heap_if()使用用户定义的二元比较关系函数 t_binary_op 把数据区间[t_first, t_last)建立成有效的堆。

DEFINITION:

<cstl/calgorithm.h>

18. algo_sort_heap algo_sort_heap_if

PROTOTYPE:

```
void algo_sort_heap(random_access_iterator_t t_first, random_access_iterator_t t_last);
```

```
void algo_sort_heap_if(
    random_access_iterator_t t_first, random_access_iterator_t t_last,
    binary_function_t t_binary_op);
```

DESCRIPTION:

algo_sort_heap()对数据区间[t_first, t_last)进行堆排序。

algo_sort_heap_if()对数据区间[t_first, t_last)进行堆排序，排序时使用用户定义的二元比较关系函数t_binary_op。

DEFINITION:

<cstl/calgorithm.h>

19. algo_is_heap algo_is_heap_if

PROTOTYPE:

```
bool_t algo_is_heap(random_access_iterator_t t_first, random_access_iterator_t t_last);
```

```
bool_t algo_is_heap_if(
    random_access_iterator_t t_first, random_access_iterator_t t_last,
    binary_function_t t_binary_op);
```

DESCRIPTION:

algo_is_heap()判断数据区间[t_first, t_last)是否是一个有效的堆。

algo_is_heap_if()判断数据区间[t_first, t_last)是否是一个有效的堆，判断时使用用户定义的二元比较关系函数t_binary_op。

DEFINITION:

<cstl/calgorithm.h>

20. algo_min algo_min_if

PROTOTYPE:

```
input_iterator_t algo_min(input_iterator_t t_first, input_iterator_t t_second);
```

```
input_iterator_t algo_min_if(
    input_iterator_t t_first, input_iterator_t t_second, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_min()返回t_first和t_second两个数据中比较小的数据的迭代器。

algo_min_if()返回t_first和t_second两个数据中比较小的数据的迭代器，判断时使用用户定义的二元比较关系函数t_binary_op。

DEFINITION:

<cstl/calgorithm.h>

21. algo_max algo_max_if

PROTOTYPE:

```
input_iterator_t algo_max(input_iterator_t t_first, input_iterator_t t_second);
input_iterator_t algo_max_if(
    input_iterator_t t_first, input_iterator_t t_second, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_max()返回 t_first 和 t_second 两个数据中比较大的数据的迭代器。

algo_max_if()返回 t_first 和 t_second 两个数据中比较大的数据的迭代器，判断时使用用户定义的二元比较关系函数 t_binary_op。

DEFINITION:

<cstl/calgorithm.h>

22. algo_min_element algo_min_element_if

PROTOTYPE:

```
forward_iterator_t algo_min_element(forward_iterator_t t_first, forward_iterator_t t_last);
forward_iterator_t algo_min_element_if(
    forward_iterator_t t_first, forward_iterator_t t_last, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_min_element()返回数据区间[t_first, t_last)中值最小的数据的迭代器。

algo_min_element_if()返回数据区间[t_first, t_last)中值最小的数据的迭代器，判断时使用用户定义的二元比较关系函数 t_binary_op。

DEFINITION:

<cstl/calgorithm.h>

23. algo_max_element algo_max_element_if

PROTOTYPE:

```
forward_iterator_t algo_max_element(forward_iterator_t t_first, forward_iterator_t t_last);
forward_iterator_t algo_max_element_if(
    forward_iterator_t t_first, forward_iterator_t t_last, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_max_element()返回数据区间[t_first, t_last)中值最大的数据的迭代器。

algo_max_element_if()返回数据区间[t_first, t_last)中值最大的数据的迭代器，判断时使用用户定义的二元比较关系函数 t_binary_op。

DEFINITION:

<cstl/calgorithm.h>

24. algo_lexicographical_compare algo_lexicographical_compare_if

PROTOTYPE:

```
bool_t algo_lexicographical_compare(
    input_iterator_t t_first1, input_iterator_t t_last1,
    input_iterator_t t_first2, input_iterator_t t_last2);
```

```
bool_t algo_lexicographical_compare_if(
    input_iterator_t t_first1, input_iterator_t t_last1,
    input_iterator_t t_first2, input_iterator_t t_last2, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_lexicographical_compare()逐个比较两个数据区间[t_first1, t_last1)和[t_first2, t_last2)的数据，如果第一个区间中的数据小于第二个区间中的相应数据返回 true，如果大于返回 false，如果都相等时比较两个区间的长度第一个区间小时返回 true 否则返回 false。

algo_lexicographical_compare_if()与 algo_lexicographical_compare()功能相同只是判断时使用用户定义的二元比较关系函数 t_binary_op。

DEFINITION:

<cstl/calgorithm.h>

25. algo_lexicographical_compare_3wap algo_lexicographical_compare_3way_if

PROTOTYPE:

```
int algo_lexicographical_compare_3way(
    input_iterator_t t_first1, input_iterator_t t_last1,
    input_iterator_t t_first2, input_iterator_t t_last2);
```

```
int algo_lexicographical_compare_3way_if(
    input_iterator_t t_first1, input_iterator_t t_last1,
    input_iterator_t t_first2, input_iterator_t t_last2, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_lexicographical_compare_3way()与 algo_lexicographical_compare()功能相似，只是返回值不同，当第一个区间小于第二个区间时返回负数值，当两个区间相等时返回 0，当第一个区间大于第二个区间时返回正数值。

algo_lexicographical_compare_3way_if()与 algo_lexicographical_compare_3way()功能相同只是判断时使用用户定义的二元比较关系函数 t_binary_op。

DEFINITION:

<cstl/calgorithm.h>

26. algo_next_permutation algo_next_permutation_if

PROTOTYPE:

```
bool_t algo_next_permutation(bidirectional_iterator_t t_first, bidirectional_iterator_t t_last);
```

```
bool_t algo_next_permutation_if(
    bidirectional_iterator_t t_first, bidirectional_iterator_t t_last,
    binary_function_t t_binary_op);
```

DESCRIPTION:

algo_next_permutation()将数据区间[t_first, t_last)中的数据转换到下一个组合形式，如果没有下一个组合形式就回到第一个组合形式并返回 false，否则返回 true。

algo_next_permutation_if()将数据区间[t_first, t_last)中的数据转换到下一个组合形式，如果没有下一

个组合形式就回到第一个组合形式并返回 false，否则返回 true。判断时使用用户定义的二元比较关系函数 t_binary_op。

DEFINITION:

<cstl/calgorithm.h>

27. algo_prev_permutation algo_prev_permutation_if

PROTOTYPE:

```
bool_t algo_prev_permutation(bidirectional_iterator_t t_first, bidirectional_iterator_t t_last);
```

```
bool_t algo_prev_permutation_if(
    bidirectional_iterator_t t_first, bidirectional_iterator_t t_last,
    binary_function_t t_binary_op);
```

DESCRIPTION:

algo_prev_permutation()将数据区间[t_first, t_last)中的数据转换到上一个组合形式，如果没有上一个组合形式就回到最后一个组合形式并返回 false，否则返回 true。

algo_prev_permutation_if()将数据区间[t_first, t_last)中的数据转换到上一个组合形式，如果没有上一个组合形式就回到最后一个组合形式并返回 false，否则返回 true。判断时使用用户定义的二元比较关系函数 t_binary_op。

DEFINITION:

<cstl/calgorithm.h>

第四节 算术算法

1. algo_iota

PROTOTYPE:

```
void algo_iota(forward_iterator_t t_first, forward_iterator_t t_last, element);
```

DESCRIPTION:

algo_iota()为数据区间[t_first, t_last)赋一系列增加的值，如*t_first = element, *(t_first + 1) = element + 1 等等。

DEFINITION:

<cstl/cnumeric.h>

2. algo_accumulate algo_accumulate_if

PROTOTYPE:

```
void algo_accumulate(input_iterator_t t_first, input_iterator_t t_last, element, void* pv_output);
```

```
void algo_accumulate_if(
    input_iterator_t t_first, input_iterator_t t_last, element,
    binary_function_t t_binary_op, void* pv_output);
```

DESCRIPTION:

algo_accumulate()使用 element 作为初始值，将数据区间[t_first, t_last)的数据累加并把结果保存在输出结果*pv_output 中。

algo_accumulate_if()使用 element 作为初始值，将数据区间[t_first, t_last)的数据累加并把结果保存在输出结果*pv_output 中，累加过程使用用户定义的二元累加函数 t_binary_op。

DEFINITION:

<cstl/cnumeric.h>

3. algo_inner_product algo_inner_product_if

PROTOTYPE:

```
void algo_inner_product(
    input_iterator_t t_first1, input_iterator_t t_last1, input_iterator_t t_first2,
    element, void* pv_output);
```

```
void algo_inner_product_if(
    input_iterator_t t_first1, input_iterator_t t_last1, input_iterator_t t_first2,
    element, binary_function_t t_binary_op1, binary_function_t t_binary_op2, void* pv_output);
```

DESCRIPTION:

algo_inner_product()使用初始值 element 和两个数据区间[t_first1, t_last1)和[t_first2, t_first2 + (t_last1 - t_first1))执行内积运算，结果保存在输出结果*pv_output 中，具体的执行过程如下*pv_output = element + *t_first1 × *t_first2 + *(t_first1 + 1) × *(t_first2 + 1) + ...。

algo_inner_product_if()使用初始值 element 和两个数据区间[t_first1, t_last1)和[t_first2, t_first2 + (t_last1 - t_first1))和两个用户定义的二元运算函数 t_binary_op1 和 t_binary_op2 执行内积运算，结果保存在输出结果*pv_output 中，具体的执行过程如下*pv_output = element OP1 (*t_first1 OP2 *t_first2) OP1 (*(t_first1 + 1) OP2 *(t_first2 + 1)) OP1 ...。

DEFINITION:

<cstl/cnumeric.h>

4. algo_partial_sum algo_partial_sum_if

PROTOTYPE:

```
output_iterator_t algo_partial_sum(
    input_iterator_t t_first, input_iterator_t t_last, output_iterator_t t_result);
```

```
output_iterator_t algo_partial_sum_if(
    input_iterator_t t_first, input_iterator_t t_last,
    output_iterator_t t_result, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_partial_sum()计算数据区间[t_first, t_last)的局部总和，保存在以 t_result 开头的数据区间中，同时返回数据区间的结尾。计算的过程如下*t_result = *t_first, *(t_result + 1) = *t_first + *(t_first + 1), *(t_result + 2) = *t_first + *(t_first + 1) + *(t_first + 2) , ...。

algo_partial_sum_if()计算数据区间[t_first, t_last)的局部总和，保存在以 t_result 开头的数据区间中，同时返回数据区间的结尾。计算的过程如下*t_result = *t_first, *(t_result + 1) = *t_first OP *(t_first + 1), *(t_result + 2) = *t_first OP *(t_first + 1) OP *(t_first + 2) , ...。

DEFINITION:

<cstl/cnumeric.h>

5. algo_adjacent_difference algo_adjacent_difference_if

PROTOTYPE:

```
output_iterator_t algo_adjacent_difference(
    input_iterator_t t_first, input_iterator_t t_last, output_iterator_t t_result);
```

```
output_iterator_t algo_adjacent_difference_if(
    input_iterator_t t_first, input_iterator_t t_last,
    output_iterator_t t_result, binary_function_t t_binary_op);
```

DESCRIPTION:

algo_adjacent_difference()计算数据区间[t_first, t_last)中相邻数据的差，保存在以t_result开头的数据区间中，同时返回数据区间的结尾。计算的过程如下*t_result = *t_first, *(t_result + 1) = *(t_first + 1) - *t_first +, *(t_result + 2) = *(t_first + 2) - *(t_first + 1) , ...。这个函数与algo_partial_sum()互为逆函数。

algo_adjacent_difference_if()计算数据区间[t_first, t_last)的相邻数据的差，保存在以t_result开头的数据区间中，同时返回数据区间的结尾。计算的过程如下*t_result = *t_first, *(t_result + 1) = *(t_first + 1) OP *t_first, *(t_result + 2) = *(t_first + 2) OP *(t_first + 1) , ...。这个函数与algo_partial_sum_if()互为逆函数。

DEFINITION:

<cstl/cnumeric.h>

6. algo_power algo_power_if

PROTOTYPE:

```
void algo_power(input_iterator_t t_iter, size_t t_power, void* pv_output);
```

```
void algo_power_if(
    input_iterator_t t_iter, size_t t_power, binary_function_t t_binary_op, void* pv_output);
```

DESCRIPTION:

algo_power()计算t_iter的t_power次幂元算，结果保存在输出结果中，*pv_output = *t_iter × *t_iter × *t_iter × ...。

algo_power_if()计算t_iter的t_power次幂元算，结果保存在输出结果中，*pv_output = *t_iter OP *t_iter OP *t_iter OP ...。

DEFINITION:

<cstl/cnumeric.h>

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(	判断两个 pair_t 是否不等。

const pair_t* cpt_first, const pair_t* cpt_second);	
bool_t pair_less(const pair_t* cpt_first, const pair_t* cpt_second);	判断第一个 pair_t 是否小于第二个 pair_t。
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。