CSTL 参考手册

王博
http://activesys.cublog.cn
activesys@sina.com.cn
沈阳

Table of Contents

1. CSTL 简介

CSTL 模仿 SGI STL 写成的，为 C 语言编程提供了通用的数据结构和算法的库。CSTL 提供的数据结构类型是通用的，它们可以用来保存各种类型的数据。同时 CSTL 还提供了大量的算法用于管理数据结构中的数据。

1.1. 容器和算法

CSTL 容器是结构体类型，可以保存任何类型的数据。如 **`create_vector(int)`**；就创建了一个用于保存 int 类型的 vector_t 容器类型：

```
vector_t t_v = create_vector(int);
```

CSTL 同样包含一系列算法，算法用来管理容器中的数据。你可以使用逆序算法使容器中的数据逆序：

```
algo_reverse(vector_begin(&t_v), vector_end（&t_v));
```

同样这个算法还可以用在其他容器上：

```
deque_t t_dq = create_deque(double);
…
algo_reverse(deque_begin(&t_dq), deque_end(&t_dq));
```

1.2. 迭代器

迭代器是容器和算法的桥梁，算法通过迭代器组成的数据空间就可以管理任何容器，每一容器都提供了与迭代器相关的操作函数，如 **`vector_begin()`**和 **`vector_end()`**。CSTL 还提供了很多与迭代器相关的操作函数，如通过迭代器获得数据，修改数据，获得数据的指针，向前或向后移动迭代器等。

1.3. CSTL 其他组成部分

CSTL 还提供了工具类型：pair_t, bool_t。这些类型是供其他容器类型使用的。此外 CSTL 提供了函数，用来扩展算法的执行规则。

2. 怎样使用这篇文档

参考手册分为如下几个部分：

TYPE：

这部分主要介绍的具体类型。

ITERATOR TYPE:

迭代器类型。

VALUE:

头文件中定义的值。

DESCRIPTION:

类型描述。

DEFINITION:

类型声明的头文件。

OPERATION:

操作函数。有些函数的参数使用 type 和 element 表示，其中 type 表示需要调用该函数时输入具体类型如

```
vector_t create_vector(type);
```

如果要创建一个保存 int 类型数据的 vector_t 容器：

```
vector_t t_v = create_vector(int);
```

如果要创建一个保存自定义类型 struct abc_t 的 vector_t 容器：

```
vector_t t_v = create_vector(struct abc_t);
```

element 表示调用该函数时直接使用常量数据或变量数据如

```
void vector_push_back(vector_t* pt_vector, element);
```

如 vector_t 容器中保存的是 int 类型的数据，并要向数据中插入的值为 12，可以直接使用常量数据 12：

```
vector_push_back(&t_v, 12);
```

也可以传递变量数据 12：

```
int n_value = 12;
vector_push_back(&t_v, n_value);
```

如果 vector_t 容器中保存的是自定义类型，则必须使用变量数据：

```
struct abc_t t_value;
…
vector_push_back(&t_v, t_value);
```

NOTE:

在调用函数是需要注意的事项。当调用函数需要注意时，函数说明后面会有[1]类似的标志。

PROTOTYPE：

函数原型。

MEMBER：

类型的成员，可以通过类型对象直接使用。

3. 容器

3.1. 序列容器

3.1.1. vector_t

TYPE：

```
vector_t
```

ITERATOR TYPE:

```
random_access_iterator_t
vector_iterator_t
```

DESCRIPTION:

vector_t 容器是序列容器，支持对数据的随机访问。在末尾插入或删除数据花费常数时间，在开头或中间插入或删除数据花费线性时间。支持动态增长。vector_t 是 CSTL 中最简单的容器类型。

DEFINITION:

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

OPERATION:

`vector_t create_vector(type);`	创建指定类型的 vector_t 容器。
`void vector_init(vector_t* pt_vector);`	初始化一个空 vector_t 容器。
`void vector_init_n(` `  vector_t* pt_vector, size_t t_count);`	初始化一个具有 t_count 个数据的 vector_t 容器，每个数据的值都是 0。

`void vector_init_elem(`
`  vector_t* pt_vector, size_t t_count, element`
`);`
	初始化一个具有 t_count 个数据的 vector_t 容器，每个数据的值都是 element。

`void vector_init_copy(`
`  vector_t* pt_vector, const vector_t* cpt_src`
`);`
	使用令一个 vector_t 容器初始化 vector_t 容器。

`void vector_init_copy_range(`
`  vector_t* pt_vector,`
`  vector_iterator_t t_begin,`
`  vector_iterator_t t_end);`
	使用数据区间[t_begin, t_end)初始化 vector_t 容器。[1][2]

`void vector_destroy(vector_t* pt_vector);`	销毁 vector_t 容器。
`size_t vector_size(const vector_t* cpt_vector);`	获得 vector_t 容器中数据的数目。
`size_t vector_max_size(` `  const vector_t* cpt_vector);`	获得 vector_t 容器中能够保存的数据的最大数目。
`bool_t vector_empty(const vector_t* cpt_vector);`	判断 vector_t 容器是否为空。
`size_t vector_capacity(` `  const vector_t* cpt_vector);`	获得 vector_t 容器的容量。
`void vector_reserve(` `  vector_t* pt_vector, size_t t_size);`	设置 vector_t 容器的容量。

`bool_t vector_equal(`
`  const vector_t* cpt_first,`
`  const vector_t* cpt_second);`
	判断两个 vector_t 容器是否相等。

`bool_t vector_not_equal(`
`  const vector_t* cpt_first,`
`  const vector_t* cpt_second);`
	判断两个 vector_t 容器是否不等。

`bool_t vector_less(`
`  const vector_t* cpt_first,`
`  const vector_t* cpt_second);`
	判断第一个 vector_t 容器是否小于第二个 vector_t 容器。

`bool_t vector_less_equal(`
`  const vector_t* cpt_first,`
`  const vector_t* cpt_second);`
	判断第一个 vector_t 容器是否小于等于第二个 vector_t 容器。

`bool_t vector_great(`
`  const vector_t* cpt_first,`
`  const vector_t* cpt_second);`
	判断第一个 vector_t 容器是否大于第二个 vector_t 容器。

`bool_t vector_great_equal(`
`  const vector_t* cpt_first,`
`  const vector_t* cpt_second);`
	判断第一个 vector_t 容器是否大于等于第二个 vector_t 容器。

`void vector_assign(`
`  vector_t* pt_vector, const vector_t* cpt_src`
`);`
	使用另一个 vector_t 容器为当前 vector_t 容器赋值。

`void vector_assign_elem(`
`  vector_t* pt_vector, size_t t_count, element`
`);`
	使用 t_count 个 element 值给 vector_t 容器赋值。

`void vector_assign_range(`
`  vector_t* pt_vector,`
`  vector_iterator_t t_begin,`
`  vector_iterator_t t_end);`
	使用数据区间[t_begin, t_end)为 vector_t 容器赋值。[1][2]

`void vector_swap(` `  vector_t* pt_first, vector_t* pt_second);`	交换两个 vector_t 容器的内容。

`void* vector_at(`
`  const vector_t* cpt_vector,`
`  size_t t_subscript);`
	使用下标对 vector_t 容器中的数据进行随机访问。

`void* vector_front(const vector_t* cpt_vector);`	访问 vector_t 容器中的第一个数据。
`void* vector_back(const vector_t* cpt_vector);`	访问 vector_t 容器中的最后一个数据。
`vector_iterator_t vector_begin(` `  const vector_t* cpt_vector);`	返回指向 vector_t 容器开始的迭代器。
`vector_iterator_t vector_end(` `  const vector_t* cpt_vector);`	返回指向 vector_t 容器结尾的迭代器。

`vector_iterator_t vector_insert(`
`  vector_t* pt_vector,`
`  vector_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。

`vector_iterator_t vector_insert_n(`
`  vector_t* pt_vector,`
`  vector_iterator_t t_pos, size_t t_count,`
`  element);`
	在 t_pos 前面插入 t_count 个数据 element，并返回指向第一个新数据的迭代器。

`void vector_insert_range(`
`  vector_t* pt_vector,`
`  vector_iterator_t t_pos,`
`  vector_iterator_t t_begin,`
`  vector_iterator_t t_end);`
	在 t_pos 前面插入数据区间[t_begin, t_end)。[1][2]

`void vector_push_back(` `  vector_t* pt_vector, element);`	将数据 element 插入到 vector_t 容器的末尾。
`void vector_pop_back(vector_t* pt_vector);`	删除 vector_t 容器的最后一个数据。

`vector_iterator_t vector_erase(`
`  vector_t* pt_vector,`
`  vector_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。

`vector_iterator_t vector_erase_range(`
`  vector_t* pt_vector,`
`  vector_iterator_t t_begin,`
`  vector_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 vector_resize(` `  vector_t* pt_vector, size_t t_resize);`	重置 vector_t 容器中数据的数目，新增的数据为 0。
`void vector_resize_elem(`	重置 vector_t 容器中数据的数目，新增的数据为 element。

`vector_t* pt_vector, size_t t_resize, element );`	
`void vector_clear(vector_t* pt_vector);`	清空 vector_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 vector_t 容器。

3.1.2. deque_t

TYPE：

`deque_t`

ITERATOR TYPE:

`random_access_iterator_t`
`deque_iterator_t`

DESCRIPTION:

deque_t 容器与 vector_t 容器十分相似，支持对数据的随机访问。在末尾插入或删除数据花费常数时间，在中间插入或删除数据花费线性时间。支持动态增长。deque_t 与 vector_t 不同的是在开头插入或删除数据也花费线性时间。同时 deque_t 没有容量的概念所以没有提供与容器相关的操作函数。

DEFINITION:

`<cstl/cdeque.h>`

OPERATION:

`deque_t create_deque(type);`	创建指定类型的 deque_t 容器。
`void deque_init(deque_t* pt_deque);`	初始化一个空的 deque_t 容器。
`void deque_init_n( deque_t* pt_deque, size_t t_count);`	初始化一个具有 t_count 个数据的 deque_t 容器，每个数据的值都是 0。
`void deque_init_elem( deque_t* pt_deque, size_t t_count, element);`	初始化一个具有 t_count 个数据的 deque_t 容器，每个数据的值都是 element。
`void deque_init_copy( deque_t* pt_deque, const deque_t* cpt_src);`	使用令一个 deque_t 容器初始化 deque_t 容器。
`void deque_init_copy_range( deque_t* pt_deque, deque_iterator_t t_begin, deque_iterator_t t_end);`	使用数据区间[t_begin, t_end)初始化 deque_t 容器。[1][2]
`void deque_destroy(deque_t* pt_deque);`	销毁 deque_t 容器。
`size_t deque_size(const deque_t* cpt_deque);`	获得 deque_t 容器中数据的数目。
`size_t deque_max_size( const deque_t* cpt_deque);`	获得 deque_t 容器中能够保存的数据的最大数目。
`bool_t deque_empty(const deque_t* cpt_deque);`	判断 deque_t 容器是否为空。
`bool_t deque_equal( const deque_t* cpt_first, const deque_t* cpt_second);`	判断两个 deque_t 容器是否相等。
`bool_t deque_not_equal( const deque_t* cpt_first, const deque_t* cpt_second);`	判断两个 deque_t 容器是否不等。

`bool_t deque_less(`
`  const deque_t* cpt_first,`
`  const deque_t* cpt_second);`
	判断第一个 deque_t 容器是否小于第二个 deque_t 容器。

`bool_t deque_less_equal(`
`  const deque_t* cpt_first,`
`  const deque_t* cpt_second);`
	判断第一个 deque_t 容器是否小于等于第二个 deque_t 容器。

`bool_t deque_great(`
`  const deque_t* cpt_first,`
`  const deque_t* cpt_second);`
	判断第一个 deque_t 容器是否大于第二个 deque_t 容器。

`bool_t deque_great_equal(`
`  const deque_t* cpt_first,`
`  const deque_t* cpt_second);`
	判断第一个 deque_t 容器是否大于等于第二个 deque_t 容器。

`void deque_assign(` `  deque_t* pt_deque, const deque_t* cpt_src);`	使用另一个 deque_t 容器为当前 deque_t 容器赋值。
`void deque_assign_elem(` `  deque_t* pt_deque, size_t t_count, element);`	使用 t_count 个 element 值给 deque_t 容器赋值。

`void deque_assign_range(`
`  deque_t* pt_deque,`
`  deque_iterator_t t_begin,`
`  deque_iterator_t t_end);`
	使用数据区间[t_begin, t_end)为 deque_t 容器赋值。[1][2]

`void deque_swap(` `  deque_t* pt_first, deque_t* pt_second);`	交换两个 deque_t 容器的内容。

`void* deque_at(`
`  const deque_t* cpt_deque,`
`  size_t t_subscript);`
	使用下标对 deque_t 容器中的数据进行随机访问。

`void* deque_front(const deque_t* cpt_deque);`	访问 deque_t 容器中的第一个数据。
`void* deque_back(const deque_t* cpt_deque);`	访问 deque_t 容器中的最后一个数据。
`deque_iterator_t deque_begin(` `  const deque_t* cpt_deque);`	返回指向 deque_t 容器开始的迭代器。
`deque_iterator_t deque_end(` `  const deque_t* cpt_deque);`	返回指向 deque_t 容器结尾的迭代器。

`deque_iterator_t deque_insert(`
`  deque_t* pt_deque,`
`  deque_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。

`deque_iterator_t deque_insert_n(`
`  deque_t* pt_deque,`
`  deque_iterator_t t_pos, size_t t_count,`
`  element);`
	在 t_pos 前面插入 t_count 个数据 element，并返回指向第一个新数据的迭代器。

`void deque_insert_range(`
`  deque_t* pt_deque,`
`  deque_iterator_t t_pos,`
`  deque_iterator_t t_begin,`
`  deque_iterator_t t_end);`
	在 t_pos 前面插入数据区间[t_begin, t_end)。[1][2]

`void deque_push_back(` `  deque_t* pt_deque, element);`	将数据 element 插入到 deque_t 容器的末尾。
`void deque_pop_back(deque_t* pt_deque);`	删除 deque_t 容器的最后一个数据。
`void deque_push_front(` `  deque_t* pt_deque, element);`	将数据 element 插入到 deque_t 容器的开头。
`void deque_pop_front(deque_t* pt_deque);`	删除 deque_t 容器的第一个数据。

`deque_iterator_t deque_erase(`
`  deque_t* pt_deque,`
`  deque_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。

`deque_iterator_t deque_erase_range(`
`  deque_t* pt_deque,`
`  deque_iterator_t t_begin,`
`  deque_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 deque_resize(`	重置 deque_t 容器中数据的数目，新增的数据为 0。

`deque_t* pt_deque, size_t t_resize);`	
`void deque_resize_elem(` `deque_t* pt_deque, size_t t_resize, element);`	重置 deque_t 容器中数据的数目，新增的数据为 element。
`void deque_clear(deque_t* pt_deque);`	清空 deque_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 deque_t 容器。

3.1.3. list_t

TYPE：

`list_t`

ITERATOR TYPE:

`bidirectional_iterator_t`
`list_iterator_t`

DESCRIPTION:

list_t 容器是一种双向链表，支持向前和向后遍历。在任何位置插入和删除数据花费数量时间。在 list_t 中插入或删除数据不会使迭代器失效。除此之外 list_t 还提供了许多额外的操作函数。

DEFINITION:

`<cstl/clist.h>`

OPERATION:

`list_t create_list(type);`	创建指定类型的 list_t 容器。
`void list_init(list_t* pt_list);`	初始化一个空的 list_t 容器。
`void list_init_n(` `list_t* pt_list, size_t t_count);`	初始化一个具有 t_count 个数据的 list_t 容器，每个数据的值都是 0。
`void list_init_elem(` `list_t* pt_list, size_t t_count, element);`	初始化一个具有 t_count 个数据的 list_t 容器，每个数据的值都是 element。
`void list_init_copy(` `list_t* pt_list, const list_t* cpt_src);`	使用令一个 list_t 容器初始化 list_t 容器。

`void list_init_copy_range(`
`list_t* pt_list,`
`list_iterator_t t_begin,`
`list_iterator_t t_end);`
	使用数据区间[t_begin, t_end)初始化 list_t 容器。[1][2]

`void list_destroy(list_t* pt_list);`	销毁 list_t 容器。
`size_t list_size(const list_t* cpt_list);`	获得 list_t 容器中数据的数目。
`size_t list_max_size(` `const list_t* cpt_list);`	获得 list_t 容器中能够保存的数据的最大数目。
`bool_t list_empty(const list_t* cpt_list);`	判断 list_t 容器是否为空。

`bool_t list_equal(`
`const list_t* cpt_first,`
`const list_t* cpt_second);`
	判断两个 list_t 容器是否相等。

`bool_t list_not_equal(`
`const list_t* cpt_first,`
`const list_t* cpt_second);`
	判断两个 list_t 容器是否不等。

`bool_t list_less(`
`  const list_t* cpt_first,`
`  const list_t* cpt_second);`
	判断第一个 list_t 容器是否小于第二个 list_t 容器。

`bool_t list_less_equal(`
`  const list_t* cpt_first,`
`  const list_t* cpt_second);`
	判断第一个 list_t 容器是否小于等于第二个 list_t 容器。

`bool_t list_great(`
`  const list_t* cpt_first,`
`  const list_t* cpt_second);`
	判断第一个 list_t 容器是否大于第二个 list_t 容器。

`bool_t list_great_equal(`
`  const list_t* cpt_first,`
`  const list_t* cpt_second);`
	判断第一个 list_t 容器是否大于等于第二个 list_t 容器。

`void list_assign(` `  list_t* pt_list, const list_t* cpt_src);`	使用另一个 list_t 容器为当前 list_t 容器赋值。
`void list_assign_elem(` `  list_t* pt_list, size_t t_count, element);`	使用 t_count 个 element 值给 list_t 容器赋值。

`void list_assign_range(`
`  list_t* pt_list,`
`  list_iterator_t t_begin,`
`  list_iterator_t t_end);`
	使用数据区间[t_begin, t_end)为 list_t 容器赋值。[1][2]

`void list_swap(` `  list_t* pt_first, list_t* pt_second);`	交换两个 list_t 容器的内容。
`void* list_front(const list_t* cpt_list);`	访问 list_t 容器中的第一个数据。
`void* list_back(const list_t* cpt_list);`	访问 list_t 容器中的最后一个数据。
`list_iterator_t list_begin(` `  const list_t* cpt_list);`	返回指向 list_t 容器开始的迭代器。
`list_iterator_t list_end(` `  const list_t* cpt_list);`	返回指向 list_t 容器结尾的迭代器。

`list_iterator_t list_insert(`
`  list_t* pt_list,`
`  list_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。

`list_iterator_t list_insert_n(`
`  list_t* pt_list,`
`  list_iterator_t t_pos, size_t t_count,`
`  element);`
	在 t_pos 前面插入 t_count 个数据 element，并返回指向第一个新数据的迭代器。

`void list_insert_range(`
`  list_t* pt_list,`
`  list_iterator_t t_pos,`
`  list_iterator_t t_begin,`
`  list_iterator_t t_end);`
	在 t_pos 前面插入数据区间[t_begin, t_end)。[1][2]

`void list_push_back(` `  list_t* pt_list, element);`	将数据 element 插入到 list_t 容器的末尾。
`void list_pop_back(list_t* pt_list);`	删除 list_t 容器的最后一个数据。
`void list_push_front(` `  list_t* pt_list, element);`	将数据 element 插入到 list_t 容器的开头。
`void list_pop_front(list_t* pt_list);`	删除 list_t 容器的第一个数据。
`void list_remove(list_t* pt_list, element);`	删除 list_t 容器中所有值为 element 的数据。

`void list_remove_if(`
`  list_t* pt_list, unary_function_t t_unary_op`
`);`
	删除 list_t 容器中所有满足一元谓词 t_unary_op 的数据。

`list_iterator_t list_erase(`
`  list_t* pt_list,`
`  list_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。

`list_iterator_t list_erase_range(`
`  list_t* pt_list,`
`  list_iterator_t t_begin,`
`  list_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 list_resize(` `  list_t* pt_list, size_t t_resize);`	重置 list_t 容器中数据的数目，新增的数据为 0。
`void list_resize_elem(` `  list_t* pt_list, size_t t_resize, element);`	重置 list_t 容器中数据的数目，新增的数据为 element。
`void list_clear(list_t* pt_list);`	清空 list_t 容器。
`void list_unique(list_t* pt_list);`	删除 list_t 容器中连续的重复数据。

`void list_unique_if(`
`  list_t* pt_list,`
`  binary_function_t t_binary_op);`
	删除 list_t 容器中连续的满足二元谓词 t_binary_op 的数据。

`void list_splice(`
`  list_t* pt_list, list_iterator_t t_pos,`
`  list_t* pt_src);`
	将 pt_src 中的数据转移到 t_pos。

`Void list_splice_pos(`
`  list_t* pt_list, list_iterator_t t_pos,`
`  list_t* pt_src, list_iterator_t t_srcpos);`
	将 t_srcpos 位置的数据转移到 t_pos。

`void list_splice_range(`
`  list_t* pt_list, list_iterator_t t_pos,`
`  list_t* pt_src,`
`  list_iterator_t t_begin,`
`  list_iterator_t t_end);`
	将数据区间[t_begin, t_end)中的数据转移到 t_pos。[1]

`void list_sort(list_t* pt_list);`	排序 list_t 容器中的数据。

`void list_sort_if(`
`  list_t* pt_list,`
`  binary_function_t t_binary_op);`
	使用二元谓词 t_binary_op 作为排序规则，排序 list_t 容器中的数据。

`void list_merge(` `  list_t* pt_first, list_t* pt_second);`	将两个有序的 list_t 容器合并。

`void list_merge_if(`
`  list_t* pt_first, list_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 list_t 容器中的数据。

`void list_reverse(list_t* pt_list);`	将 list_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 list_t 容器。

3.1.4. 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(`	将 t_srcpos 位置的数据转移到 t_pos。

`slist_t* pt_slist, slist_iterator_t t_pos,` `slist_t* pt_src, slist_iterator_t t_srcpos);`	

`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。

3.2. 关联容器

3.2.1. 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(`
`  set_t* pt_set, set_iterator_t t_hint,`
`  element);`
	向 set_t 容器中插入数据 element 时使用线索位置 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。

3.2.2. 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(` `  const multiset_t* cpt_multiset);`	获得 multiset_t 容器中数据的数目。
`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,`	判断第一个 multiset_t 容器是否小于第二个 multiset_t 容器。

`const multiset_t* cpt_second);`	

`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(`
`multiset_t* pt_multiset,`
`multiset_iterator_t t_hint, element);`
	向 multiset_t 容器中插入数据 element 时使用线索位置 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。

3.2.3. 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(`
`  map_t* pt_map,`
`  map_iterator_t t_begin,`
`  map_iterator_t t_end);`
	向 map_t 容器中插入数据区间[t_begin, t_end)。[1][2]

`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。

3.2.4. 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,`
`  const pair_t* cpt_pair);`
	向 multimap_t 容器中插入数据对 cpt_pair，返回新数据的位置。

`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。

3.2.5. 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(`
`  hash_set_t* pt_hash_set,`
`  size_t t_bucketcount,`
`  int (*pfun_hash)(const void*, size_t, size_t)`
`);`
	初始化一个空的 hash_set_t 容器，容器的 hash 表大小为 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(`	返回指向 hash_set_t 容器开始的迭代器。

`const hash_set_t* cpt_hash_set);`	
`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），CSTL 就使用默认的 hash 函数。

3.2.6. 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(`	初始化一个空的 hash_multiset_t 容器，容器的 hash 表大小为

`hash_multiset_t* pt_hash_multiset,`
`size_t t_bucketcount,`
`int (*pfun_hash)(const void*, size_t, size_t)`
`);`
	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,`
`  size_t t_resize);`
	修改 hash_multiset_t 容器的 hash 表的大小。

`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。

3.2.7. 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,`
`  size_t t_bucketcount,`
`  int (*pfun_hash)(const void*, size_t, size_t)`
`);`
	初始化一个空的 hash_map_t 容器，容器的 hash 表大小为 t_bucketcount。

`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。

3.2.8. 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。

3.3. 字符串

3.3.1. 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(`
`  const string_t* cpt_first,`
`  const string_t* cpt_second);`
	判断两个 string_t 容器是否相等。

`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(`
`  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);`
	比较第一个 string_t 容器的子串和第二个 string_t 容器的子串，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,`	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，

`size_t t_pos, size_t t_len,` `const char* s_cstr);`	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,`
`size_t t_len);`
	向 string_t 容器后添加字符串常量的子串。

`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,`
`  string_iterator_t t_begin,`
`  string_iterator_t t_end,`
`  const char* s_cstr, size_t t_len);`
	使用字符串常量的子串替换数据区间[t_begin, t_end)。[1]

`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,`
`  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 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 中出现的最

`size_t t_len);`	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。

3.4. 容器适配器

3.4.1. 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,`	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。

`const stack_t* cpt_second);`	

`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。

3.4.2. 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。

3.4.3. 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,`
`  const priority_queue_t* cpt_src);`
	使用令一个 priority_queue_t 容器适配器初始化 priority_queue_t 容器适配器。

`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)。

4. 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,`
`  const void* cpt_value);`
	设置迭代器 cpt_iterator 所指的数据。

`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。

5. 算法

5.1. 非质变算法

5.1.1. 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>

5.1.2. 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>

5.1.3. 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>

5.1.4. 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>

5.1.5. 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>

5.1.6. 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>

5.1.7. 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>

5.1.8. 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>

5.1.9. 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>

5.1.10. 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>

5.2. 质变算法

5.2.1. 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>

5.2.2. 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>

5.2.3. 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>

5.2.4. 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>

5.2.5. 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>

5.2.6. 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>
```

5.2.7. 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>
```

5.2.8. 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>

5.2.9. 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>

5.2.10. 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>

5.2.11. 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>

5.2.12. 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>

5.2.13. 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>

5.2.14. 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>

5.2.15. 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>

5.2.16. 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>

5.3. 排序算法

5.3.1. 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>

5.3.2. 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>

5.3.3. 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>

5.3.4. 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>

5.3.5. 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>

5.3.6. 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>

5.3.7. 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>

5.3.8. 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>

5.3.9. 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>
```

5.3.10. 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>
```

5.3.11. 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>

5.3.12. 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>

5.3.13. 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>

5.3.14. 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>

5.3.15. 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>

5.3.16. 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>
```

5.3.17. 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>
```

5.3.18. 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>
```

5.3.19. 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>
```

5.3.20. 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>
```

5.3.21. 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>
```

5.3.22. 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>
```

5.3.23. 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>
```

5.3.24. 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>
```

5.3.25. 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>
```

5.3.26. 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>
```

5.3.27. 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>
```

5.4. 算术算法

5.4.1.　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>

5.4.2. 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>

5.4.3. 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>

5.4.4. 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>

5.4.5. 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>

5.4.6. 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>`

6. 工具类型

6.1. bool_t

TYPE：

bool_t

VALUE:

false

true

FALSE

TRUE

DESCRIPTION:

bool_t 是 CSTL 定义的新类型用来表示布尔值。

DEFINITION:

包含任何一个 **cstl** 头文件都可以使用 **bool_t** 类型。

6.2. pair_t

TYPE：

pair_t

DESCRIPTION:

pair_t 保存两个任意类型的数据，它将两个不同的数据统一在一起，是对的概念。

DEFINITION:

<cstl/cutility.h>

MEMBER:

first	void*类型的指针，用来引用第一个数据。
second	void*类型的指针，用来引用第二个数据。

OPERATION:

pair_t create_pair(first_type, second_type);	创建指定类型的 pair_t，first_type 为第一个数据的类型，second_type 为第二个数据的类型。
void pair_init(pair_t* pt_pair);	初始化 pair_t，值为空。
void pair_init_elem(pair_t* pt_pair, first_element, second_element);	使用两个值来初始化 pair_t。
void pair_init_copy(pair_t* pt_pair, const pair_t* cpt_src);	使用另一个 pair_t 来初始化 pair_t。

`void pair_destroy(pair_t* pt_pair);`	销毁 pair_t。
`void pair_assign(` `  pair_t* pt_pair, const pair_t* cpt_src);`	使用另一个 pair_t 赋值。

`void pair_make(`
`  pair_t* pt_pair,`
`  first_element, second_element);`
	使用两个值 first_element 和 second_element 来构造已经出初始化的 pair_t。

`bool_t pair_equal(`
`  const pair_t* cpt_first,`
`  const pair_t* cpt_second);`
	判断两个 pair_t 是否相等。

`bool_t pair_not_equal(`
`  const pair_t* cpt_first,`
`  const pair_t* cpt_second);`
	判断两个 pair_t 是否不等。

`bool_t pair_less(`
`  const pair_t* cpt_first,`
`  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。

7. 函数类型

TYPE：

```
unary_function_t
binary_function_t
```

DEFINITION:

所有的函数声明在**<cstl/cfunctional.h>**

7.1. 算术运算函数

7.1.1. 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。

7.1.2. 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。

7.1.3. 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。

7.1.4. 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。

7.1.5. 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。

7.1.6. 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。

7.2. 关系运算函数

7.2.1. 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*。

7.2.2. 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*。

7.2.3. 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*。

7.2.4. 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。

7.2.5. 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*。

7.2.6. 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*。

7.3. 逻辑运算函数

7.3.1. 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。

7.3.2. 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。

7.3.3. 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。

7.4. 其他函数

7.4.1. 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。

7.4.2. 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。