deque

deque概述

deque是一种双向开口的“连续”线性空间（即可以在头尾两端分别做元素的插入和删除操作）。

deque与vector的差别：

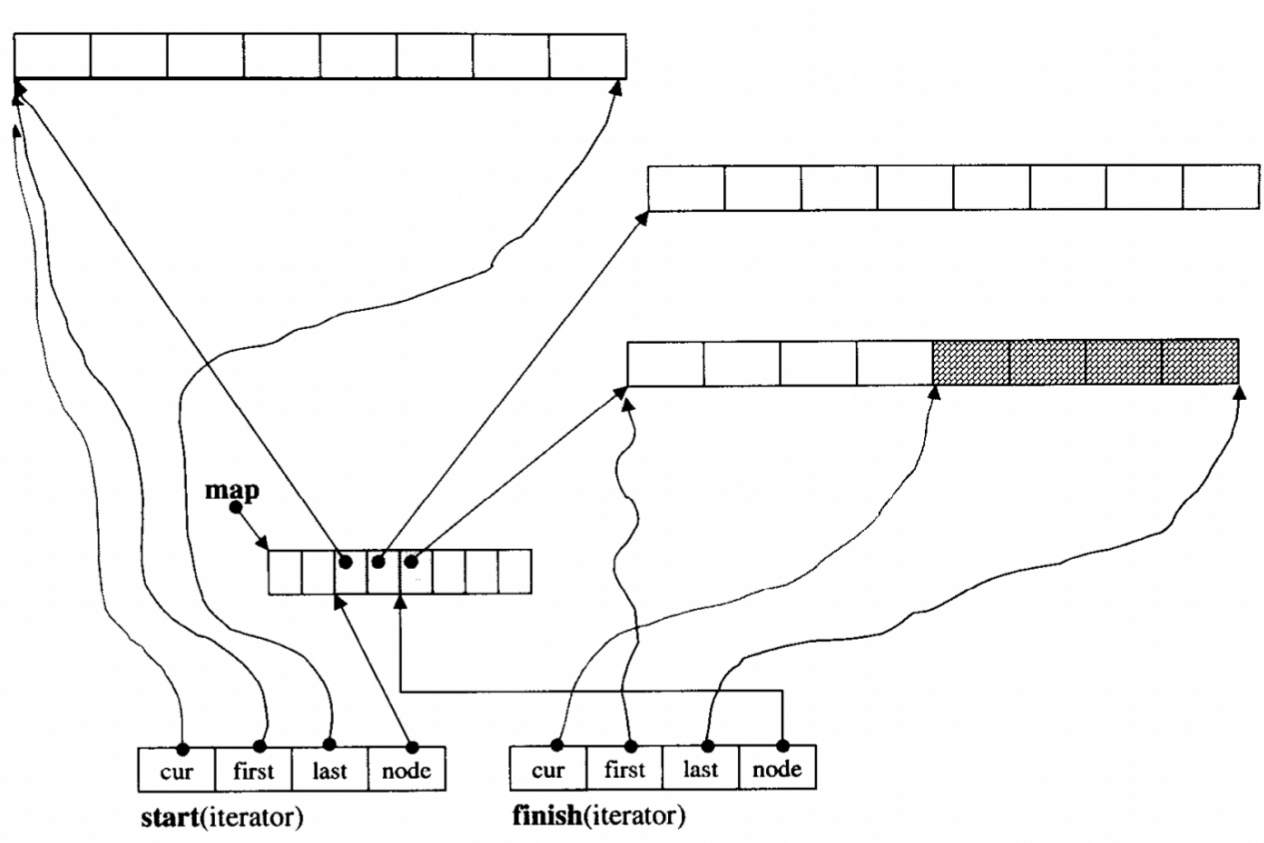
1. deque允许在常数时间内对头端进行元素的插入和删除操作，vector尾部插入和删除常数时间，头部操作O（n）时间
2. deque没有容量概念，不需要和vector一样进行老三样：申请新空间->复制元素->释放旧空间
3. deque的Ramdon Access Iterator进行过特殊设计，复杂度比vector高。

deque的中控器

底层内存分布

deque的“连续”空间只是在逻辑上连续的，实际上deque是由一段一段的定量连续空间构成。一旦有必要在deque的头端或尾端增加新空间，便配置一段定量的连续空间，串接在整个deque的头端或尾端。

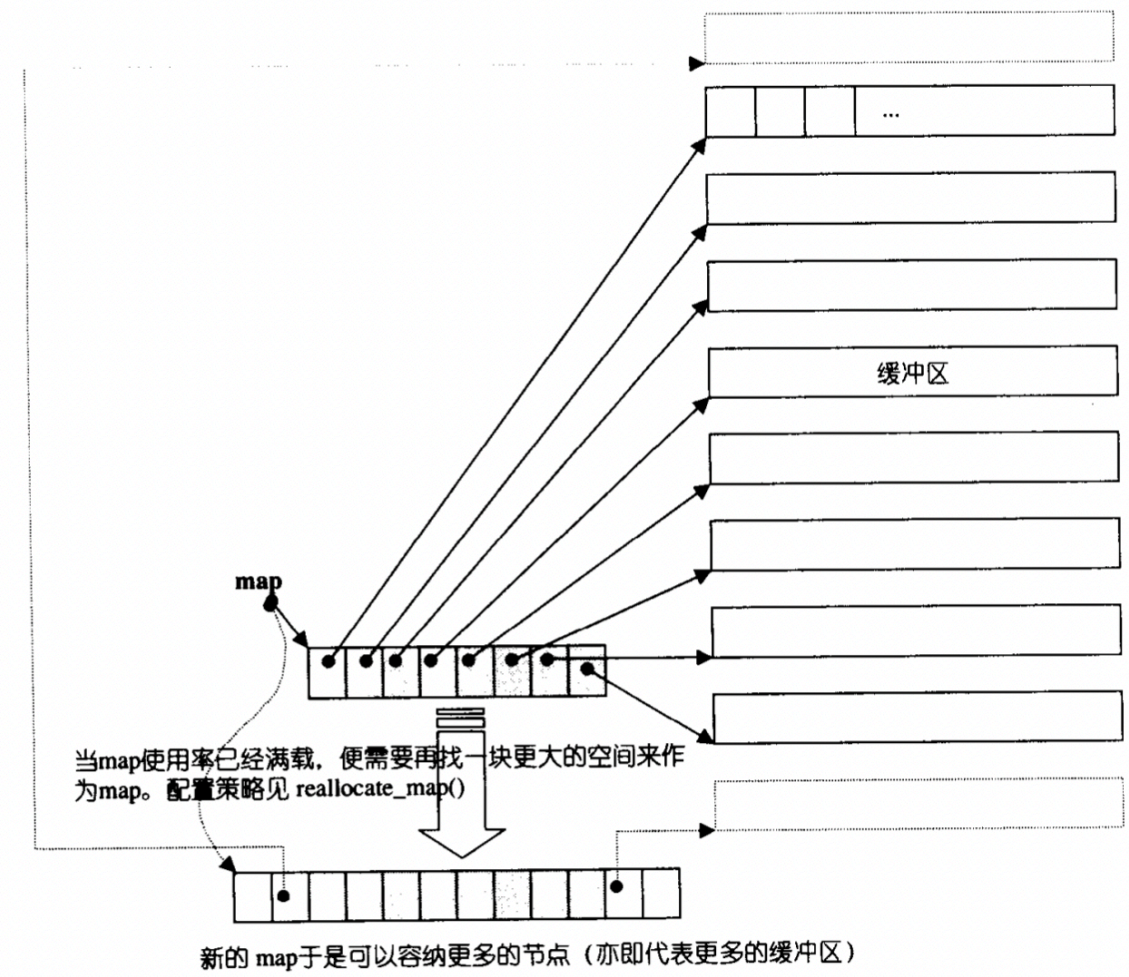
deque的核心任务是维护 **分段的定量连续空间**整体连续的假象，并提供随机存取的接口，避免vector的申请新空间->复制元素->释放旧空间，但其代价是deque的迭代器架构较复杂。



中控器

正如上面那张图看到的，对于**分段的定量连续空间**，我们需要一个map（就是一小段连续空间，和数组类似）来指示他（就像一个包租婆有很多房子，手里得有一个地图，找到他的每个房子）

```
template <class T, class Alloc = alloc, size_t BufSiz = 0>
class deque {
public:                          // Basic types
  typedef T value_type;
  typedef value_type* pointer;
  typedef value_type& reference;
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;
protected:                       // Internal typedefs
  typedef pointer* map_pointer;
protected:
  map_pointer map;//指向指针数组 T** 类似于二维数组 T*指向一个缓冲区，T**就是这个map
  size_type map_size;//指针数组元素个数
}
```



新的 map于是可以容纳更多的节点（亦即代表更多的缓冲区）

deque的迭代器

迭代器结构

deque是**分段连续空间**，维护“整体连续”假象的任务就落在了operator++和operator--两个运算上了。

我们可以想象，一个iterator指向一个缓冲区（buffer）内元素时，当到了一个buffer的末端则需要跳到下一个buffer，到了buffer的头端则需要跳转到前一个buffer，这里则需要我们前面所说的map来调节。如何知道我们是否在buffer的头尾端呢？这就需要iterator保存这个buffer的begin和end了。

综上，deque的iterator需要以下元素：

1. 指向当前元素的指针
2. 指向当前buffer头端的指针
3. 指向当前buffer尾端的指针
4. 指向map中控的指针

SGI STL中源码如下：

```
template <class T, class Ref, class Ptr, size_t BufSiz>
struct __deque_iterator {   // 未继承 std::iterator
  typedef __deque_iterator<T, T&, T*, BufSiz>      iterator;
  typedef __deque_iterator<T, const T&, const T*, BufSiz> const_iterator;
  static size_t buffer_size() {return __deque_buf_size(BufSiz, sizeof(T)); }
#else /* __STL_NON_TYPE_TMPL_PARAM_BUG */
template <class T, class Ref, class Ptr>
struct __deque_iterator {   // 未继承 std::iterator
  typedef __deque_iterator<T, T&, T*>             iterator;
  typedef __deque_iterator<T, const T&, const T*> const_iterator;
  static size_t buffer_size() {return __deque_buf_size(0, sizeof(T)); }
#endif

  // 未继承 std::iterator，所以必须自己写五个必要的迭代器相应型别
  typedef random_access_iterator_tag iterator_category; // (1)
  typedef T value_type;          // (2)
  typedef Ptr pointer;         // (3)
  typedef Ref reference;         // (4)
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;  // (5)
  typedef T** map_pointer;

  typedef __deque_iterator self;

  // 保持与容器的联结
  T* cur; // 此迭代器所指之缓冲区中的现行（current）元素
  T* first; // 此迭代器所指之缓冲区头
  T* last;  // 此迭代器所指之缓冲区尾（含备用空间）
  map_pointer node;

  __deque_iterator(T* x, map_pointer y)
```

```
    : cur(x), first(*y), last(*y + buffer_size()), node(y) {}
  __deque_iterator() : cur(0), first(0), last(0), node(0) {}
  __deque_iterator(const iterator& x)
    : cur(x.cur), first(x.first), last(x.last), node(x.node) {}
}
```

其中用来决定缓冲区大小的函数buffer_size()，调用__deque_buf_size()

```
// 如果 n 不为 0，传回 n，表示 buffer size 由使用者自定。
// 如果 n 为 0，表示buffer size 使用预设值，那么
//   如果 sz（元素大小，sizeof(value_type)）小于 512，传回 512/sz，
//   如果 sz 不小于 512，传回 1。
inline size_t __deque_buf_size(size_t n, size_t sz)
{
  return n != 0 ? n : (sz < 512 ? size_t(512 / sz) : size_t(1));
}
```

有了迭代器，缓存区，中控map，我们的deque的原型就出来啦：

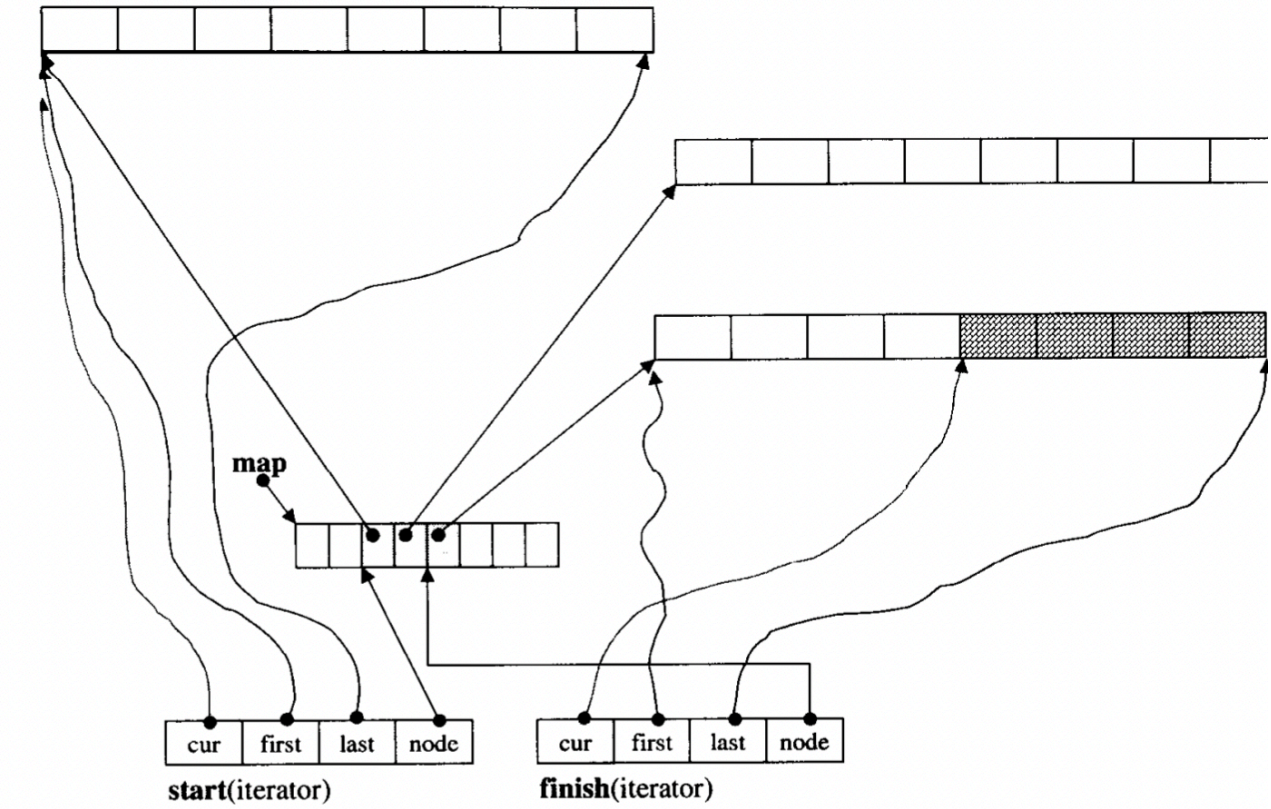


图 4-12 deque::begin() 传回迭代器 start，deque::end() 传回迭代器 finish。这两个迭代器都是 deque 的 data members。图中所示的这个 deque 拥有 20 个 int 元素，以 3 个缓冲区储存之。每个缓冲区 32 bytes，可储存 8 个 int 元素。map 大小为 8（起始值），目前用了 3 个节点。

迭代器操作

迭代器到buffer边缘，则需要请求中控map，看怎么跳转buffer。

```
void set_node(map_pointer new_node) {
    node = new_node;
    first = *new_node;
    last = first + difference_type(buffer_size());
}
```

解引用操作：

```
reference operator*() const { return *cur; }
pointer operator->() const { return &(operator*()); }
```

计算两迭代器之间距离：

```
// 两个iterator相减，计算距离
difference_type operator-(const self& x) const {
    return difference_type(buffer_size()) * (node - x.node - 1) +
           (cur - first) + (x.last - x.cur);
}
```

迭代器的++、--操作：

```
// 参考 More Effective C++, item6: Distinguish between prefix and
// postfix forms of increment and decrement operators.
self& operator++() {
    ++cur;        // 切换至下一个元素。
    if (cur == last) {    // 如果已达所在缓冲区的尾端，
        set_node(node + 1); // 就切换至下一个节点（亦即缓冲区）
        cur = first;      //    的第一个元素。
    }
    return *this;
}

self operator++(int)  {
    self tmp = *this;
    ++*this;
    return tmp;
}

self& operator--() {
    if (cur == first) { // 如果已达所在缓冲区的头端，
        set_node(node - 1); // 就切换至前一个节点（亦即缓冲区）
        cur = last;     //    的最后一个元素。
    }
    --cur;        // 切换至前一个元素。
```

```
    return *this;
}
self operator--(int) {
    self tmp = *this;
    --*this;
    return tmp;
}
```

迭代器+一个数值访问实现：

```
// 參考 More Effective C++, item22: Consider using op= instead of
// stand-alone op.
self operator+(difference_type n) const {
    self tmp = *this;
    return tmp += n; // 调用operator+=
}

self& operator+=(difference_type n) {
    difference_type offset = n + (cur - first);
    if (offset >= 0 && offset < difference_type(buffer_size()))
        // 目标位置在同一缓冲区内
        cur += n;
    else {
        // 目标位置不在同一缓冲区内
        difference_type node_offset =
                offset > 0 ? offset / difference_type(buffer_size())
                           : -difference_type((-offset - 1) / buffer_size()) -
1;
        // 切换至正确的节点（亦即缓冲区）
        set_node(node + node_offset);
        // 切换至正确的元素
        cur = first + (offset - node_offset * difference_type(buffer_size()));
    }
    return *this;
}


// 參考 More Effective C++, item22: Consider using op= instead of
// stand-alone op.
self operator-(difference_type n) const {
    self tmp = *this;
    return tmp -= n; // 调用operator-=
}

self& operator-=(difference_type n) { return *this += -n; }
// 以上利用operator+= 来完成 operator-=
```

随机访问实现，模拟连续空间：

```
//随机存取实现
reference operator[](difference_type n) const { return *(*this + n); }
// 以上调用operator*, operator+
```

迭代器的比较操作：

```
bool operator==(const self& x) const { return cur == x.cur; }
bool operator!=(const self& x) const { return !(*this == x); }
bool operator<(const self& x) const {
    return (node == x.node) ? (cur < x.cur) : (node < x.node);
}
```

deque的数据结构

deque除了维护一个map中控和map中控大小外，还维护了start、finish两个迭代器，分别指向第一个buffer的第一个元素和最后buffer的最后一个元素的下一个位置（左闭右开）。map中控大小的作用是：一旦节点不足，就得配置一块更大的map。

deque的数据结构如下：

```
template <class T, class Alloc = alloc, size_t BufSiz = 0>
class deque {
public:                         // Basic types
  typedef T value_type;
  typedef value_type* pointer;
  typedef value_type& reference;
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;

protected:                      // Internal typedefs
  typedef pointer* map_pointer;

  static size_type buffer_size() {//返回
    return __deque_buf_size(BufSiz, sizeof(value_type));
  }
  static size_type initial_map_size() { return 8; }

protected:                      // Data members
  map_pointer map;//指向指针数组
  size_type map_size;//指针数组元素个数
  iterator start;//开始迭代器，其中cur指向头部元素
  iterator finish;//结束迭代器，其中cur指向尾部元素后面的一个元素
}
```

deque的基本对外接口：

```
public:                          // Basic accessors
iterator begin() { return start; }

iterator end() { return finish; }

const_iterator begin() const { return start; }

const_iterator end() const { return finish; }

reference operator[](size_type n) {
    return start[difference_type(n)]; // 调用 __deque_iterator<>::operator[]
}

const_reference operator[](size_type n) const {
    return start[difference_type(n)];
}

reference front() { return *start; } // 调用 __deque_iterator<>::operator*
reference back() {
    iterator tmp = finish;
    --tmp;    // 调用 __deque_iterator<>::operator--
    return *tmp;    // 调用 __deque_iterator<>::operator*
}

size_type size() const { return finish - start; }

// 以上调用iterator::operator-
size_type max_size() const { return size_type(-1); }

bool empty() const { return finish == start; }
```

deque的构造与内存管理

deque的ctor

deque自行定义了两个空间配置器：

```
protected:                       // Internal typedefs
// 专属之空间配置器，每次配置一个元素大小
typedef simple_alloc<value_type, Alloc> data_allocator;
// 专属之空间配置器，每次配置一个指标大小
typedef simple_alloc<pointer, Alloc> map_allocator;
```

并有如下构造函数：

```
deque(size_type n, const value_type& value)
  : start(), finish(), map(0), map_size(0)
{
    fill_initialize(n, value);
}
```

fill_initialize()负责产生并安排好deque的结构，并将元素的初值设置好：

```
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::fill_initialize(size_type n,
                                               const value_type& value) {
    create_map_and_nodes(n);    // 把deque的结构都产生并安排好
    map_pointer cur;
    __STL_TRY {
            // 为每个节点的缓冲区設定初值
            for (cur = start.node; cur < finish.node; ++cur)
            uninitialized_fill(*cur, *cur + buffer_size(), value);
            // 最后一个节点的设定稍有不同（因为尾端可能有备用空间，不必设初值）
            uninitialized_fill(finish.first, finish.cur, value);
    }
    catch(...) {
    // "commit or rollback" ：若非全部成功，就一个不留。
  }
```

其中create_map_and_nodes()复制产生并安排好deque的结构：

```
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::create_map_and_nodes(size_type num_elements) {
    // 需要节点数=(元素个数/每个缓冲区可容纳的元素个数)+1
    // 如果刚好整除，會多配一个节点。
    size_type num_nodes = num_elements / buffer_size() + 1;

    // 一个 map 要管理几个节点。最少8个，最多是 “所需节点数加2”
    //  （前后各預留一个，扩充時可用）。
    map_size = max(initial_map_size(), num_nodes + 2);
    map = map_allocator::allocate(map_size);
    // 以上配置出一个 “具有 map_size个节点” 的map。

    // 以下令nstart和nfinish指向map所拥有之全部节点的最中央区段。
    // 保持在最中央，可使头尾两端的扩充能量一样大。每个节点可对应一个缓冲区。
    map_pointer nstart = map + (map_size - num_nodes) / 2;
    map_pointer nfinish = nstart + num_nodes - 1;

    map_pointer cur;
    __STL_TRY {
            // 为map內的每个現用节点配置缓冲区。所有缓冲区加起来就是deque的空间
            //  （最后一个缓冲区可能留有一些余裕）。
            for (cur = nstart; cur <= nfinish; ++cur)
```

```
            *cur = allocate_node();
    }
    catch(...) {
    // "commit or rollback" : 若非全部成功，就一个不留。
  }
  // 为deque内的两个迭代器start和end 設定正确的内容。
  start.set_node(nstart);
  finish.set_node(nfinish);
  start.cur = start.first;    // first, cur都是public
  finish.cur = finish.first + num_elements % buffer_size();
  // 前面说过，如果刚好整除，会多配一个节点。
  // 此时即令cur指向這多配的一個节点（所对应之缓冲区）的起点。
}
```

push_back() & push_front()

```
public:                          // push_* and pop_*
void push_back(const value_type& t) {
    if (finish.cur != finish.last - 1) {
        // 最后缓冲区尚有一个以上的备用空间
        construct(finish.cur, t); // 直接在备用空间上建构元素
        ++finish.cur; // 調整最后缓冲区的使用状态
    }
    else  // 最后缓冲区已无（或只剩一个）元素备用空间。
        push_back_aux(t);
}
```

尾端只有一个元素备用空间时，push_back调用push_back_aux()，先设置一整块的buffer，再设置新元素内容，然后更改finish：

```
// 只有当 finish.cur == finish.last – 1 时才会被呼叫。
// 也就是说只有当最后一个缓冲区只剩一个备用元素空間时才会被呼叫。
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::push_back_aux(const value_type& t) {
    value_type t_copy = t;
    reserve_map_at_back();    //  若符合某种条件则必须重换一个map
    *(finish.node + 1) = allocate_node(); // 配置一个新节点（缓冲区）
    __STL_TRY {
            construct(finish.cur, t_copy);    // 设值
            finish.set_node(finish.node + 1); // 改变finish，令其指向新节点
            finish.cur = finish.first;      // 設定 finish 的状态
    }
    __STL_UNWIND(deallocate_node(*(finish.node + 1)));
}
```

push_front()和push_back()同理：

```
void push_front(const value_type& t) {
    if (start.cur != start.first) {   // 第一缓冲区尚有备用空間
        construct(start.cur - 1, t);  // 直接在备用空間上建构元素
        --start.cur;     // 调整第一缓冲区的使用状态
    }
    else // 第一缓冲区已无备用空間
        push_front_aux(t);
}

// 只有当start.cur == start.first時才会被呼叫。
// 也就是说只有当第一个缓冲区沒有任何备用元素時才会被呼叫。
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::push_front_aux(const value_type& t) {
    value_type t_copy = t;
    reserve_map_at_front();   //  若符合某种条件則必須重換一个map
    *(start.node - 1) = allocate_node();  // 配置一个新节点（缓冲区）
    __STL_TRY {
            start.set_node(start.node - 1);   // 改变start，令其指向新节点
            start.cur = start.last - 1;     // 设定 start的状态
            construct(start.cur, t_copy);   // 设值
    }
    catch(...) {
    // "commit or rollback" ：若非全部成功，就一个不留。
        start.set_node(start.node + 1);
        start.cur = start.first;
        deallocate_node(*(start.node - 1));
        throw;
  }
}
```

其中，整治map的操作reserve_map_at_back()和reserve_map_at_front()为调用reallocate_map()：

```
void reserve_map_at_back (size_type nodes_to_add = 1) {
    if (nodes_to_add + 1 > map_size - (finish.node - map))
        // 如果 map 尾端的节点备用空间不足
        // 符合以上条件則必須重換一个map（配置更大的，拷贝原來的，释放原來的）
        reallocate_map(nodes_to_add, false);
}

void reserve_map_at_front (size_type nodes_to_add = 1) {
    if (nodes_to_add > start.node - map)
        // 如果 map 前端的节点备用空间不足
        // 符合以上条件則必須重換一个map（配置更大的，拷贝原來的，释放原來的）
        reallocate_map(nodes_to_add, true);
}
```

reallocate_map()函数实现为：

```
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::reallocate_map(size_type nodes_to_add,
                                              bool add_at_front) {
    size_type old_num_nodes = finish.node - start.node + 1;
    size_type new_num_nodes = old_num_nodes + nodes_to_add;

    map_pointer new_nstart;
    if (map_size > 2 * new_num_nodes) {
        new_nstart = map + (map_size - new_num_nodes) / 2
                     + (add_at_front ? nodes_to_add : 0);
        if (new_nstart < start.node)
            copy(start.node, finish.node + 1, new_nstart);
        else
            copy_backward(start.node, finish.node + 1, new_nstart +
old_num_nodes);
    }
    else {
        size_type new_map_size = map_size + max(map_size, nodes_to_add) + 2;
        // 配置一块空間，准备给新map使用。
        map_pointer new_map = map_allocator::allocate(new_map_size);
        new_nstart = new_map + (new_map_size - new_num_nodes) / 2
                     + (add_at_front ? nodes_to_add : 0);
        // 把原map 内容拷贝过来。
        copy(start.node, finish.node + 1, new_nstart);
        // 释放原map
        map_allocator::deallocate(map, map_size);
        // 设定新map的起始位址与大小
        map = new_map;
        map_size = new_map_size;
    }

    // 重新设定迭代器 start 和 finish
    start.set_node(new_nstart);
    finish.set_node(new_nstart + old_num_nodes - 1);
}
```

deque的元素操作

pop_back() & pop_front()

pop操作和push操作相反，pop是要把元素拿掉，push需要考虑加入buffer，而pop则需要考虑释放buffer。

```
void pop_back() {
    if (finish.cur != finish.first) {
        // 最后缓冲区有一个（或更多）元素
```

```
        --finish.cur;    // 调整指标，相当于排除了最后元素
        destroy(finish.cur);  // 将最后元素析构
    }
    else
        // 最后缓冲区没有任何元素
        pop_back_aux();    // 這里将进行缓冲区的释放工作
}

// 只有当finish.cur == finish.first时才會被呼叫。
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::pop_back_aux() {
    deallocate_node(finish.first);  // 释放最后一个缓冲区
    finish.set_node(finish.node - 1); // 调整 finish 的状态，使指向
    finish.cur = finish.last - 1;    //  上一个缓冲区的最后一个元素
    destroy(finish.cur);         // 将该元素析构。
}

void pop_front() {
    if (start.cur != start.last - 1) {
        // 第一缓冲区有一个（或更多）元素
        destroy(start.cur); // 将第一元素析构
        ++start.cur;       // 调整指标，相当于排除了第一元素
    }
    else
        // 第一缓冲区僅有一个元素
        pop_front_aux();     // 這里将进行缓冲区的释放工作
}

// 只有当start.cur == start.last - 1时才會被呼叫。
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::pop_front_aux() {
    destroy(start.cur);         // 将第一缓冲区的第一个元素析构。
    deallocate_node(start.first);    // 释放第一缓冲区。
    start.set_node(start.node + 1); // 调整 start 的状态，使指向
    start.cur = start.first;       //  下一个缓冲区的第一个元素。
}
```

clear()

clear()用于清空deque，deque在初始的时候有一个buffer，因此clear之后，也应该有一个buffer。

```
// 注意，最終需要保留一個緩衝區。這是deque 的策略，也是deque 的初始狀態。
template <class T, class Alloc, size_t BufSize>
void deque<T, Alloc, BufSize>::clear() {
    // 以下針對頭尾以外的每一個緩衝區（它們一定都是飽滿的）
    for (map_pointer node = start.node + 1; node < finish.node; ++node) {
        // 將緩衝區內的所有元素解構。注意，呼叫的是destroy() 第二版本，見2.2.3節
```

```
        destroy(*node, *node + buffer_size());
        // 釋放緩衝區記憶體
        data_allocator::deallocate(*node, buffer_size());
    }

    if (start.node != finish.node) {  // 至少有頭尾兩個緩衝區
        destroy(start.cur, start.last); // 將頭緩衝區的目前所有元素解構
        destroy(finish.first, finish.cur); // 將尾緩衝區的目前所有元素解構
        // 以下釋放尾緩衝區。注意，頭緩衝區保留。
        data_allocator::deallocate(finish.first, buffer_size());
    }
    else  // 只有一個緩衝區
        destroy(start.cur, finish.cur); // 將此唯一緩衝區內的所有元素解構
    // 注意，並不釋放緩衝區空間。這唯一的緩衝區將保留。

    finish = start; // 調整狀態
}
```

erase()

erase()函数可以清除一个iterator的内容，也可以清除一个范围的内容。

```
// 清除 pos 所指的元素。pos 為清除點。
iterator erase(iterator pos) {
    iterator next = pos;
    ++next;
    difference_type index = pos - start;     // 清除點之前的元素個數
    if (index < (size() >> 1)) {             // 如果清除點之前的元素比較少，
        copy_backward(start, pos, next);     // 就搬移清除點之前的元素
        pop_front();                  // 搬移完畢，最前一個元素贅餘，去除之
    } else {                        // 清除點之後的元素比較少，
        copy(next, finish, pos);     // 就搬移清除點之後的元素
        pop_back();                  // 搬移完畢，最後一個元素贅餘，去除之
    }
    return start + index;
}

template<class T, class Alloc, size_t BufSize>
deque<T, Alloc, BufSize>::iterator deque<T, Alloc, BufSize>::erase(iterator
first, iterator last) {
    if (first == start && last == finish) { // 如果清除區間就是整個 deque
        clear();                            // 直接呼叫 clear() 即可
        return finish;
    } else {
        difference_type n = last - first;             // 清除區間的長度
        difference_type elems_before = first - start;    // 清除區間前方的元素個數
        if (elems_before < (size() - n) / 2) {         // 如果前方的元素比較少，
```

```
            copy_backward(start, first, last);          // 向後搬移前方元素（覆蓋清除
區間）
            iterator new_start = start + n;             // 標記 deque 的新起點
            destroy(start, new_start);                  // 搬移完畢，將贅餘的元素解構
// 以下將贅餘的緩衝區釋放
            for (map_pointer cur = start.node; cur < new_start.node; ++cur)
                data_allocator::deallocate(*cur, buffer_size());
            start = new_start;    // 設定 deque 的新起點
        } else {    // 如果清除區間後方的元素比較少
            copy(last, finish, first);              // 向前搬移後方元素（覆蓋清除區
間）
            iterator new_finish = finish - n;    // 標記 deque 的新尾點
            destroy(new_finish, finish);         // 搬移完畢，將贅餘的元素解構
// 以下將贅餘的緩衝區釋放
            for (map_pointer cur = new_finish.node + 1; cur <= finish.node;
++cur)
                data_allocator::deallocate(*cur, buffer_size());
            finish = new_finish;    // 設定 deque 的新尾點
        }
        return start + elems_before;
    }
}
```

insert()

insert()功能：在某一点之前插入一个元素，并设定其值。

```
// 在position 處安插一個元素，其值為 x
iterator insert(iterator position, const value_type &x) {
    if (position.cur == start.cur) {    // 如果安插點是deque 最前端
        push_front(x);                  // 交給push_front 去做
        return start;
    } else if (position.cur == finish.cur) { // 如果安插點是deque 最尾端
        push_back(x);                       // 交給push_back 去做
        iterator tmp = finish;
        --tmp;
        return tmp;
    } else {
        return insert_aux(position, x);        // 交給 insert_aux 去做
    }
}

template<class T, class Alloc, size_t BufSize>
typename deque<T, Alloc, BufSize>::iterator
deque<T, Alloc, BufSize>::insert_aux(iterator pos, const value_type &x) {
    difference_type index = pos - start;    // 安插點之前的元素個數
    value_type x_copy = x;
```

```
    if (index < size() / 2) {              // 如果安插點之前的元素個數比較少
        push_front(front());               // 在最前端加入與第一元素同值的元素。
        iterator front1 = start;           // 以下標示記號，然後進行元素搬移...
        ++front1;
        iterator front2 = front1;
        ++front2;
        pos = start + index;
        iterator pos1 = pos;
        ++pos1;
        copy(front2, pos1, front1);          // 元素搬移
    } else {                             // 安插點之後的元素個數比較少
        push_back(back());                 // 在最尾端加入與最後元素同值的元素。
        iterator back1 = finish;     // 以下標示記號，然後進行元素搬移...
        --back1;
        iterator back2 = back1;
        --back2;
        pos = start + index;
        copy_backward(pos, back2, back1);     // 元素搬移
    }
    *pos = x_copy;     // 在安插點上設定新值
    return pos;
}
```