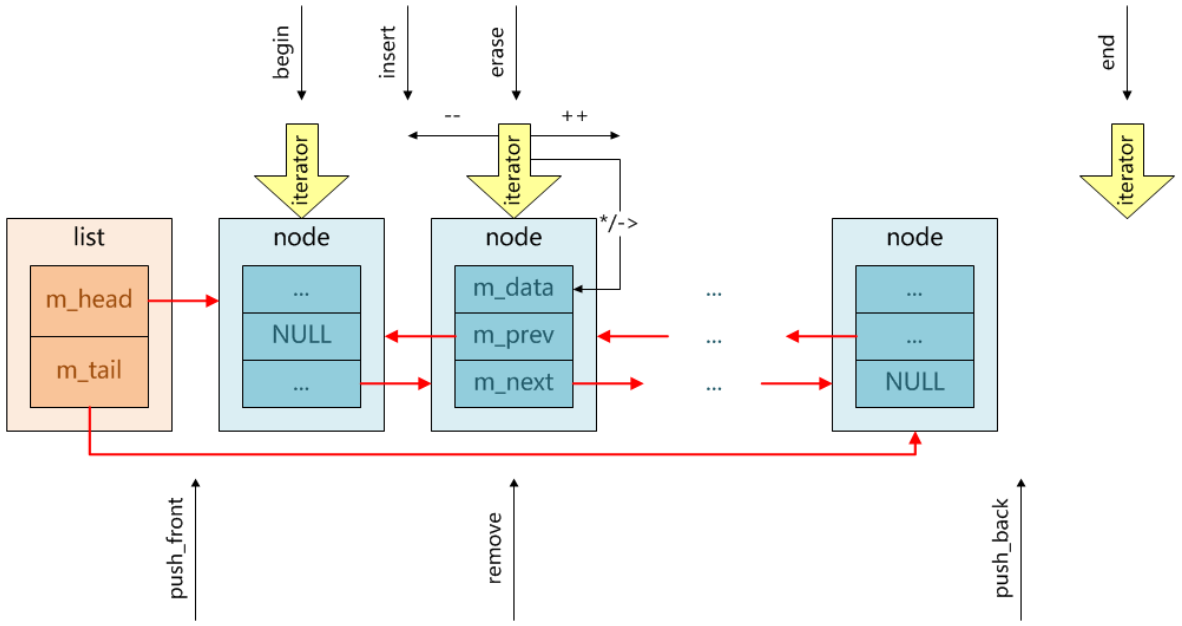
9 STL

9.1 双向线性链表容器



9.1.1 基础设施

节点及其指针。

```cpp
// list.cpp

// 双向线性链表容器

#include <string.h>

#include <iostream>
#include <stdexcept>

using namespace std;

// 链表模板

template<typename T>
class List {
private:
    // 节点及其指针

    class Node {
    public:
        Node(T const& data, Node* prev = NULL, Node* next = NULL)
            : data(data), prev(prev), next(next) {}

        friend ostream& operator<<(ostream& os, Node const& node) {
            return os << '(' << node.data << ')';
        }

        T data;
```

```
        Node *prev, *next;
    };

    Node *head, *tail;
};
```

9.1.2 构造、深拷和析构

构造函数初始化空链表，拷贝构造函数和拷贝赋值运算符函数支持深拷贝，析构函数销毁剩余节点。

```
// list.cpp

// 双向线性链表容器

...

// 链表模板

template<typename T>
class List {
public:
    // 构造、深拷和析构

    List(void) : head(NULL), tail(NULL) {}

    List(List const& list) : head(NULL), tail(NULL) {
        for (Node* node = list.head; node; node = node->next)
            pushBack(node->data);
    }

    List& operator=(List const& list) {
        if (&list != this) {
            List temp = list;

            swap(head, temp.head);
            swap(tail, temp.tail);
        }

        return *this;
    }

    ~List(void) {
        clear();
    }

    ...
};
```

9.1.3 获取首元素

通过普通容器获取其首元素的左值引用，通过常容器获取其首元素的常左值引用，后者应能够复用前者的实现。

```
// list.cpp
```

```
// 双向线性链表容器

...

// 链表模板

template<typename T>
class List {
public:
    ...

    // 获取首元素

    T& front(void) {
        if (empty())
            throw underflow_error("链表下溢");

        return head->data;
    }

    T const& front(void) const {
        return const_cast<List*>(this)->front();
    }

    ...
};
```

9.1.4 向首部压入

在首节点的前面增加新节点，使新节点成为新的首节点。

```
// list.cpp

// 双向线性链表容器

...

// 链表模板

template<typename T>
class List {
public:
    ...

    // 向首部压入

    void pushFront(T const& data) {
        head = new Node(data, NULL, head);

        if (head->next)
            head->next->prev = head;
        else
            tail = head;
    }
```

```
    ...
};
```

9.1.5 从首部弹出

删除首节点，使被删除节点的后节点成为新的首节点。

```
// list.cpp

// 双向线性链表容器

...

// 链表模板

template<typename T>
class List {
public:
    ...

    // 从首部弹出

    void popFront(void) {
        if (empty())
            throw underflow_error("链表下溢");

        Node* next = head->next;
        delete head;
        head = next;

        if (head)
            head->prev = NULL;
        else
            tail = NULL;
    }

    ...
};
```

9.1.6 在链表尾部执行与首部类似的操作

获取尾元素、向尾部压入和从尾部弹出。

```
// list.cpp

// 双向线性链表容器

...

// 链表模板

template<typename T>
class List {
```

```
public:
    ...

    // 获取尾元素

    T& back(void) {
        if (empty())
            throw underflow_error("链表下溢");

        return tail->data;
    }

    T const& back(void) const {
        return const_cast<List*>(this)->back();
    }

    // 向尾部压入

    void pushBack(T const& data) {
        tail = new Node(data, tail);

        if (tail->prev)
            tail->prev->next = tail;
        else
            head = tail;
    }

    // 从尾部弹出

    void popBack(void) {
        if (empty())
            throw underflow_error("链表下溢");

        Node* prev = tail->prev;
        delete tail;
        tail = prev;

        if (tail)
            tail->next = NULL;
        else
            head = NULL;
    }

    ...
};
```

9.1.7 删除所有匹配元素

遍历的同时判断是否相等，删除所有满足条件的节点。

```
// list.cpp

// 双向线性链表容器

...
```

```
6
7  // 链表模板
8
9  template<typename T>
10 class List {
11 public:
12     ...
13
14     // 删除所有匹配元素
15
16     void remove(T const& data) {
17         for (Node *node = head, *next; node; node = next) {
18             next = node->next;
19
20             if (node->data == data) {
21                 if (node->prev)
22                     node->prev->next = node->next;
23                 else
24                     head = node->next;
25
26                 if (node->next)
27                     node->next->prev = node->prev;
28                 else
29                     tail = node->prev;
30
31                 delete node;
32             }
33         }
34     }
35
36     ...
37 };
```

9.1.8 清空、判空、大小和输出

```
1  // list.cpp
2
3  // 双向线性链表容器
4
5  ...
6
7  // 链表模板
8
9  template<typename T>
10 class List {
11 public:
12     ...
13
14     // 清空、判空、大小和输出
15
16     void clear(void) {
17         for (Node* next; head; head = next) {
18             next = head->next;
19             delete head;
20         }
```

```
        tail = NULL;
    }

    bool empty(void) const {
        return !head && !tail;
    }

    size_t size(void) const {
        size_t nodes = 0;

        for (Node* node = head; node; node = node->next)
            ++nodes;

        return nodes;
    }

    friend ostream& operator<<(ostream& os, List const& list) {
        for (Node* node = list.head; node; node = node->next)
            os << *node;

        return os;
    }

    ...
};
```

9.1.9 第一个测试用例

```
// list.cpp

// 双向线性链表容器

...

// 测试用例

void test1(void) {
    // List<int>

    int an[] = {10, 20, 30, 20, 50};

    List<int> ln;
    for (size_t i = 0; i < sizeof(an) / sizeof(an[0]); ++i)
        ln.pushFront(an[i]);
    cout << ln << endl;
    cout << ln.front() << endl;
    ln.popFront();
    cout << ln << endl;
    ln.remove(20);
    cout << ln << endl;

    // List<string>

    char ac[][256] = {
```

```
        "beijing",
        "shanghai",
        "tianjin",
        "shanghai",
        "chongqing"};

    List<string> ls;
    for (size_t i = 0; i < sizeof(ac) / sizeof(ac[0]); ++i)
        ls.pushBack(ac[i]);
    cout << ls << endl;
    cout << ls.back() << endl;
    ls.popBack();
    cout << ls << endl;
    ls.remove("shanghai");
    cout << ls << endl;

    // List<char const*>

    List<char const*> lc;
    for (size_t i = 0; i < sizeof(ac) / sizeof(ac[0]); ++i)
        lc.pushBack(ac[i]);
    cout << lc << endl;
    cout << lc.back() << endl;
    lc.popBack();
    cout << lc << endl;
    lc.remove("shanghai");
    cout << lc << endl;
}

int main(void) {
    test1();

    return 0;
}
```

当用char const*实例化List模板时，其remove函数无法删除与其参数匹配的元素。

9.1.10 针对char const*的特化

将类型相关的操作与类型无关的操作分开。

```
// list.cpp

// 双向线性链表容器

...

// 链表模板

template<typename T>
class List {
public:
    ...

    // 删除所有匹配元素
```

```
15
16      void remove(T const& data) {
17          for (Node *node = head, *next; node; node = next) {
18              next = node->next;
19
20              if (equal(node->data, data)) {
21                  ...
22              }
23          }
24      }
25
26      ...
27  };
28
29  ...
```

分别给出equal函数的通用版本和特化版本。

```
1   // list.cpp
2
3   // 双向线性链表容器
4
5   ...
6
7   // 链表模板
8
9   template<typename T>
10  class List {
11      ...
12
13  private:
14      ...
15
16      // 判等函数通用版本
17
18      bool equal(T const& data1, T const& data2) const {
19          return data1 == data2;
20      }
21
22      ...
23  };
24
25  // 判等函数针对char const*的特化版本
26
27  template<>
28  bool List<char const*>::equal(char const* const& data1, char const* const&
    data2) const {
29      return !strcmp(data1, data2);
30  }
31
32  ...
```

当用char const*实例化List模板时，其remove函数可以删除与其参数匹配的元素。

9.2 迭代器与泛型算法

9.2.1 正向迭代器

9.2.1.1 不用迭代器的遍历

允许用户以类似数组的方式，通过下标访问容器中的元素。

```
// list.cpp

// 双向线性链表容器

...

// 链表模板

template<typename T>
class List {
public:
    ...

    // 下标操作符

    T& operator[](size_t i) {
        for (Node* node = head; node; node = node->next)
            if (i-- == 0)
                return node->data;

        throw out_of_range("下标越界");
    }

    T const& operator[](size_t i) const {
        return const_cast<List&>(*this)[i];
    }

    ...
};

...
```

第二个测试用例。

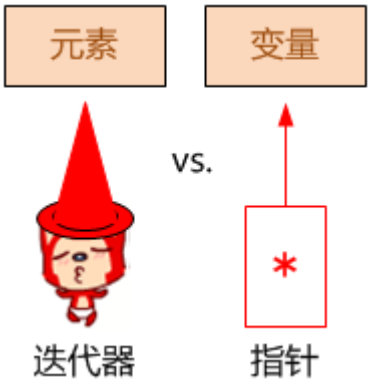
```
// list.cpp

// 双向线性链表容器

...

// 测试用例

...

void test2(void) {
    List<int> ln;
```

```
    for (int n = 1; n < 10; ++n)
        ln.pushBack(n);
    for (size_t i = 0; i < ln.size(); ++i)
        cout << ln[i] << ' ';
    cout << endl;
}

int main(void) {
    ...
    test2();

    return 0;
}
```

通过下标遍历链表的平均时间复杂度高达$O(N^2)$级。

9.2.1.2 正向迭代器内部类

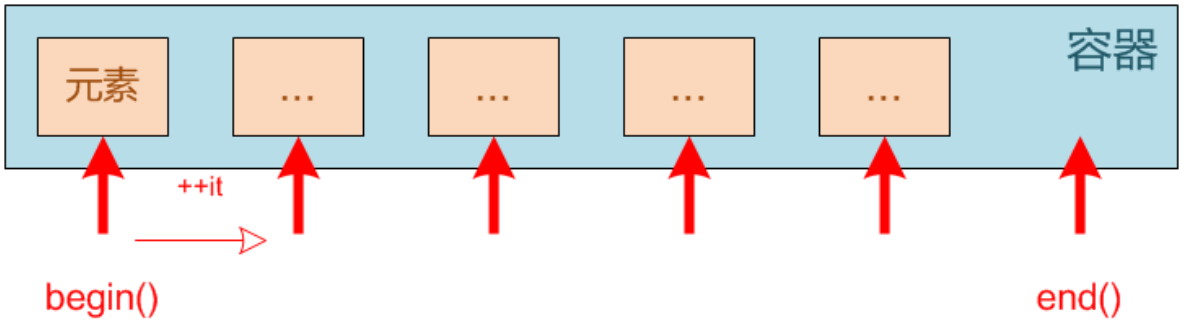
封装节点的指针，扮演指向容器中元素的指针的角色。




```
// list.cpp

// 双向线性链表容器

...

// 链表模板

template<typename T>
class List {
public:
    ...

    // 正向迭代器

    class Iterator {
    public:
        Iterator(Node* head = NULL, Node* tail = NULL, Node* node = NULL)
            : head(head), tail(tail), node(node) {}

    private:
        Node *head, *tail, *node;

        friend class List;
```

```
    };

    ...
};

...
```

通过任何容器的迭代器访问其中的元素，都与通过指针访问数组中元素无异。允许用户在完全不了解容器内部细节的前提下，以一致且透明的方式，访问其中的元素。



象指针一样支持相等和不等操作符。

```cpp
// list.cpp

// 双向线性链表容器

...

// 链表模板

template<typename T>
class List {
public:
    ...

    // 正向迭代器

    class Iterator {
    public:
        ...

        // 相等和不等操作符

        bool operator==(Iterator const& it) const {
            return node == it.node;
        }

        bool operator!=(Iterator const& it) const {
            return !(*this == it);
        }

        ...
    };

    ...
};
```

```

...
```

象指针一样支持前后自增操作符。

```
// list.cpp

// 双向线性链表容器

...

// 链表模板

template<typename T>
class List {
public:
    ...

    // 正向迭代器

    class Iterator {
    public:
        ...

        // 前后自增操作符

        Iterator& operator++(void) {
            if (node)
                node = node->next;
            else
                node = head;

            return *this;
        }

        Iterator const operator++(int) {
            Iterator it = *this;

            ++*this;

            return it;
        }

        ...
    };

    ...
};

...
```

象指针一样支持前后自减操作符。

```
// list.cpp
```

```
// 双向线性链表容器

...

// 链表模板

template<typename T>
class List {
public:
    ...

    // 正向迭代器

    class Iterator {
    public:
        ...

        // 前后自减操作符

        Iterator& operator--(void) {
            if (node)
                node = node->prev;
            else
                node = tail;

            return *this;
        }

        Iterator const operator--(int) {
            Iterator it = *this;

            --*this;

            return it;
        }

        ...
    };

    ...
};

...
```

象指针一样支持解引用和间接成员访问操作符。

```
// list.cpp

// 双向线性链表容器

...

// 链表模板

```

```
template<typename T>
class List {
public:
    ...

    // 正向迭代器

    class Iterator {
    public:
        ...

        // 解引用和间接成员访问操作符

        T& operator*(void) const {
            return node->data;
        }

        T* operator->(void) const {
            return &**this;
        }

        ...
    };

    ...
};

...
```

在实现了所有这些功能以后，一个迭代器在行为和用法上，便与一个指向容器元素的平凡指针无异。

9.2.1.3 获取起始和终止迭代器

起始正向迭代器指向容器中的首元素，而终止正向迭代器则指向容器中尾元素的下一个位置。

```
// list.cpp

// 双向线性链表容器

...

// 链表模板

template<typename T>
class List {
public:
    ...

    // 获取起始和终止迭代器

    Iterator begin(void) {
        return Iterator(head, tail, head);
    }

    Iterator end(void) {
```

```
        return Iterator(head, tail);
    }

    ...
};

...
```

9.2.1.4 基于迭代器的插入

创建新节点，并使迭代器所指向的节点及其前节点成为新建节点的后节点和前节点，返回指向新节点的迭代器。

```cpp
// list.cpp

// 双向线性链表容器

...

// 链表模板

template<typename T>
class List {
public:
    ...

    // 基于迭代器的插入

    Iterator insert(Iterator it, T const& data) {
        if (it == end()) {
            pushBack(data);
            return Iterator(head, tail, tail);
        }

        Node* node = new Node(data, it.node->prev, it.node);

        if (node->prev)
            node->prev->next = node;
        else
            head = node;

        node->next->prev = node;

        return Iterator(head, tail, node);
    }

    ...
};

...
```

9.2.1.5 基于迭代器的删除

删除迭代器所指向的节点，令该节点前后节点的后前指针指向其后前节点，返回指向被删除节点之后的迭代器。

```
// list.cpp

// 双向线性链表容器

...

// 链表模板

template<typename T>
class List {
public:
    ...

    // 基于迭代器的删除

    Iterator erase(Iterator it) {
        if (it == end())
            throw invalid_argument("无效参数");

        if (it.node->prev)
            it.node->prev->next = it.node->next;
        else
            head = it.node->next;

        if (it.node->next)
            it.node->next->prev = it.node->prev;
        else
            tail = it.node->prev;

        Node* next = it.node->next;

        delete it.node;

        return Iterator(head, tail, next);
    }

    ...
};

...
```

9.2.1.6 第三个测试用例

```
// list.cpp

// 双向线性链表容器

...

// 测试用例
```

```
...

void test3(void) {
    List<int> ln;
    ln.pushBack(10);
    ln.pushBack(20);
    ln.pushBack(30);
    ln.pushBack(40);
    ln.pushBack(50);
    ln.pushBack(60);

    List<int>::Iterator it;

    for (it = ln.begin(); it != ln.end(); ++it)
        ++*it;
    cout << ln << endl;

    it = ln.begin();
    it++;
    ++++it;
    it = ln.insert(it, 35);
    cout << ln << endl;
    it--;
    ----it;
    it = ln.erase(it);
    cout << ln << endl;
    cout << *it << endl;
}

int main(void) {
    ...
    test3();

    return 0;
}
```

9.2.2 泛型算法

9.2.2.1 线性查找

基于迭代器的线性查找。

```
// list.cpp

// 双向线性链表容器

...

// 基于迭代器的线性查找

template<typename IT, typename T>
IT find(IT begin, IT end, T const& key) {
    IT it;
    for (it = begin; it != end; ++it)
```

```
        if (*it == key)
            break;

    return it;
}

...
```

第四个测试用例。

```
// list.cpp

// 双向线性链表容器

...

// 测试用例

...

void test4(void) {
    int an[] = {13, 27, 19, 48, 36}, key = 19;
    size_t size = sizeof(an) / sizeof(an[0]);

    int* p = find(an, an + size, key);
    if (p == an + size)
        cout << "没找到" << endl;
    else
        cout << "找到了：" << *p << endl;

    List<int> ln;
    for (size_t i = 0; i < size; ++i)
        ln.pushBack(an[i]);

    List<int>::Iterator it = find(ln.begin(), ln.end(), key);
    if (it == ln.end())
        cout << "没找到" << endl;
    else
        cout << "找到了：" << *it << endl;
}

int main(void) {
    ...
    test4();

    return 0;
}
```

9.2.2.2 快速排序

基于迭代器的快速排序，包括通过小于操作符比大小和通过比较器比大小，两个版本。

```
// list.cpp

// 双向线性链表容器
```

```
...

// 基于迭代器的快速排序

template<typename IT>
void sort(IT begin, IT end) {
    IT pivot = begin;
    IT last = end;
    --last;

    for (IT i = begin, j = last; i != j;) {
        while (!(i == pivot || *pivot < *i))
            ++i;

        if (i != pivot) {
            swap(*i, *pivot);
            pivot = i;
        }

        while (!(j == pivot || *j < *pivot))
            --j;

        if (j != pivot) {
            swap(*j, *pivot);
            pivot = j;
        }
    }

    IT it = begin;
    ++it;
    if (pivot != begin && pivot != it)
        sort(begin, pivot);

    it = pivot;
    ++it;
    if (it != end && it != last)
        sort(it, end);
}

template<typename IT, typename CMP>
void sort(IT begin, IT end, CMP cmp) {
    IT pivot = begin;
    IT last = end;
    --last;

    for (IT i = begin, j = last; i != j;) {
        while (!(i == pivot || cmp(*pivot, *i)))
            ++i;

        if (i != pivot) {
            swap(*i, *pivot);
            pivot = i;
        }

        while (!(j == pivot || cmp(*j, *pivot)))
```

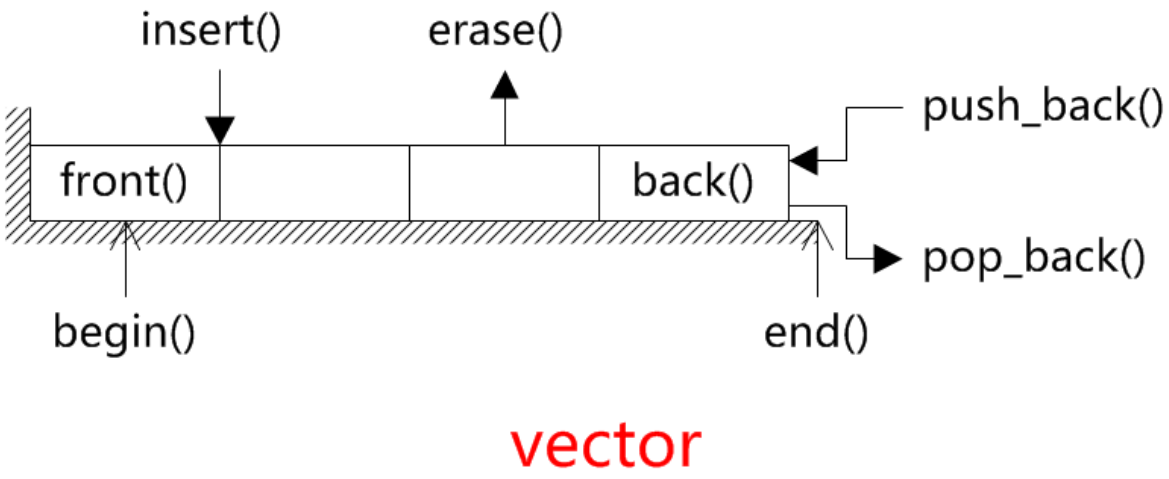
```
            --j;

        if (j != pivot) {
            swap(*j, *pivot);
            pivot = j;
        }
    }

    IT it = begin;
    ++it;
    if (pivot != begin && pivot != it)
        sort(begin, pivot, cmp);

    it = pivot;
    ++it;
    if (it != end && it != last)
        sort(it, end, cmp);
}

...
```

第五个测试用例。

```
// list.cpp

// 双向线性链表容器

...

// 测试用例

...

void test5(void) {
    int an[] = {38, 12, 56, 37, 38};

    List<int> ln;
    for (size_t i = 0; i < sizeof(an) / sizeof(an[0]); ++i)
        ln.pushBack(an[i]);
    sort(ln.begin(), ln.end());
    cout << ln << endl;

    char ac[][256] = {
        "beijing",
        "shanghai",
        "tianjin",
        "shanghai",
        "chongqing"};

    List<string> ls;
    for (size_t i = 0; i < sizeof(ac) / sizeof(ac[0]); ++i)
        ls.pushBack(ac[i]);
    sort(ls.begin(), ls.end());
    cout << ls << endl;

```

```
    List<char const*> lc;
    for (size_t i = 0; i < sizeof(ac) / sizeof(ac[0]); i++)
        lc.pushBack(ac[i]);
    class Comparator {
    public:
        bool operator() (char const* a, char const* b) {
            return strcmp(a, b) < 0;
        }
    };
    sort(lc.begin(), lc.end(), Comparator());
    cout << lc << endl;
}

int main(void) {
    ...
    test5();

    return 0;
}
```

9.3 向量



9.3.1 基本特性与实例化

```
// vector.cpp

// vector的基本用法

#include <iostream>
#include <vector>

using namespace std;

template<class T>
void print(vector<T> const& vec) {
    cout << "size=" << vec.size() << endl;

    for (typename vector<T>::const_iterator it = vec.begin();
        it != vec.end(); ++it)
        cout << *it << ' ';
```

```
17
18      cout << endl;
19  }
20
21  int main(void) {
22      vector<int> vn;
23
24      vn.push_back(34);
25      vn.push_back(23);
26      print(vn);
27
28      vector<int>::iterator it = vn.begin();
29      *it = 68;
30      *(it + 1) = 69;
31      *(it + 2) = 70; // 对不存在的元素赋值，不一定出错，但并未增加新元素
32      print(vn);
33
34      vn.pop_back();
35      print(vn);
36
37      vn.push_back(101);
38      vn.push_back(103);
39      for (size_t i = 0; i < vn.size(); ++i)
40          cout << vn[i] << ' ';
41      cout << endl;
42
43      vn[0] = 1000;
44      vn[1] = 1001;
45      vn[2] = 1002;
46      print(vn);
47
48      return 0;
49  }
```

9.3.1.1 连续内存与下标访问

- 向量中的元素被存储在一段连续的内存空间中
- 通过下标随机访问向量元素的效率与数组相当

9.3.1.2 动态内存分配

- 向量的内存空间会随着新元素的加入而自动增长
- 内存空间的连续性不会妨碍向量元素的持续增加
- 如果当前内存空间无法满足连续存储的需要，向量会自动开辟新的足够的连续内存空间，并在把原空间中的元素复制到新空间中以后，释放原空间
- 向量的增长往往伴随着内存的分配与释放、元素的复制与销毁等额外开销
- 如果能够在创建向量时合理地为它预分配一些空间，将在很大程度上缓解这些额外开销

9.3.1.3 实例化

- vector<元素类型> 向量对象
  - vector<int> vn
    空向量不包含任何元素，也不占任何用于存放元素的内存空间，但向量对象本身仍要占用内存空间
- vector<元素类型> 向量对象(初始大小)
  - vector<int> vn(5)
    基本类型元素，用适当类型的零初始化
  - vector<Student> vs(5)
    类类型的元素，用元素类型的缺省构造函数初始化
- vector<元素类型> 向量对象(初始大小, 元素初值)
  - vector<int> vn(5, 10)
    元素初值用于初始化初始大小范围内所有的元素，而不只是初始化第一个元素
  - vector<Student> vs(5, Student("张飞", 25))
    对于类类型向量而言，初始化的过程其实就是调用元素类型中特定构造函数的过程
- vector<元素类型> 向量对象(起始迭代器, 终止迭代器)
  - vector<int> v1(5)
    vector<int> v2 (v1.begin(), v1.end())
  - int an[] = {10, 20, 30, 40, 50}
    vector<int> vn(an, an+5)
    注意，平凡指针也是迭代器

9.3.2 迭代器

```cpp
// rit.cpp

// 向量的反向迭代器

#include <iostream>
#include <vector>

using namespace std;

int main(void) {
    int an[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    vector<int> vn(an, an + 10);

    for (vector<int>::reverse_iterator it = vn.rbegin(); it != vn.rend();
++it)
        cout << *it << ' ';
    cout << endl;

    return 0;
}
```

9.3.2.1 可写迭代器与只读迭代器

- 可写迭代器
  - 通过可写迭代器既能读取其目标元素的值，亦能修改其值
  - 从不带常限定（const）的容器对象、指针或引用获得的迭代器，通常都是可写迭代器
- 只读迭代器
  - 通过只读迭代器只能读取其目标元素的值，不能修改其值
  - 从带有常限定（const）的容器对象、指针或引用获得的迭代器，通常都是只读迭代器

9.3.2.2 正向迭代器与反向迭代器

- 正向迭代器
  - 起始迭代器指向容器中的第一个元素，终止迭代器指向容器中最后一个元素的下一个位置
  - 增操作向容器的尾端移动，减操作向容器首端移动
- 反向迭代器
  - 起始迭代器指向容器中的最后一个元素，终止迭代器指向容器中第一个元素的前一个位置
  - 增操作向容器的首端移动，减操作向容器尾端移动

9.3.2.3 顺序迭代器与随机迭代器

- 顺序迭代器
  - 一次只能向后或向前迭代一步
  - 只支持“++”和“--”运算
- 随机迭代器
  - 既能一次迭代一步，也能一次迭代多步
  - 除了支持“++”和“--”运算外，还支持对整数的加减运算、迭代器之间的大小比较及相减运算
- 除了向量和双端队列这两个连续内存容器可以提供随机迭代器外，其它STL容器都只提供顺序迭代器

9.3.2.4 迭代器的使用

- 向量容器包含了四个迭代器内部类：

向量容器的四个迭代器内部类	可写/只读	正向/反向	顺序/随机
iterator	可写	正向	随机
const_iterator	只读	正向	随机
reverse_iterator	可写	反向	随机
const_reverse_iterator	只读	反向	随机

- 向量容器提供了八个成员函数以获取特定的迭代器对象：

向量容器的八个迭代器成员函数	迭代器类型	迭代器位置
begin()	iterator	首元素

向量容器的八个迭代器成员函数	迭代器类型	迭代器位置
begin() const	const_iterator	首元素
end()	iterator	尾元素后
end() const	const_iterator	尾元素后
rbegin()	reverse_iterator	尾元素
rbegin() const	const_reverse_iterator	尾元素
rend()	reverse_iterator	首元素前
rend() const	const_reverse_iterator	首元素前

- 任何可能导致容器结构发生变化的函数被调用以后，先前获得的迭代器都可能因此失效，重新初始化以后再使用才是安全的

```
vector<int> vn;
vn.push_back(100);
vector<int>::iterator it = vn.begin(); // it->100
cout << *it << endl;
vn.push_back(200); // it->?
cout << *it << endl;
```

9.3.3 成员函数

9.3.3.1 获取首/尾元素

```
value_type& front(void);              // 返回容器中首元素的引用
value_type const& front(void) const; // 返回容器中首元素的常引用
value_type& back(void);               // 返回容器中尾元素的引用
value_type const& back(void) const;  // 返回容器中尾元素的常引用
```

例如：

```
vector<int> vn(5, 10);
vn.front() += 5;
vector<int> const& cr = vn;
cr.front() -= 5; // 编译错误
cout << cr.front() << endl;
--vn.back();
vector<int> const* cp = &vn;
++cp->back(); // 编译错误
cout << cp->back() << endl;
```

9.3.3.2 压入/弹出元素

```
void push_back(value_type const& val); // 向容器尾端压入元素
void pop_back(void);                   // 从容器尾端弹出元素
```

例如：

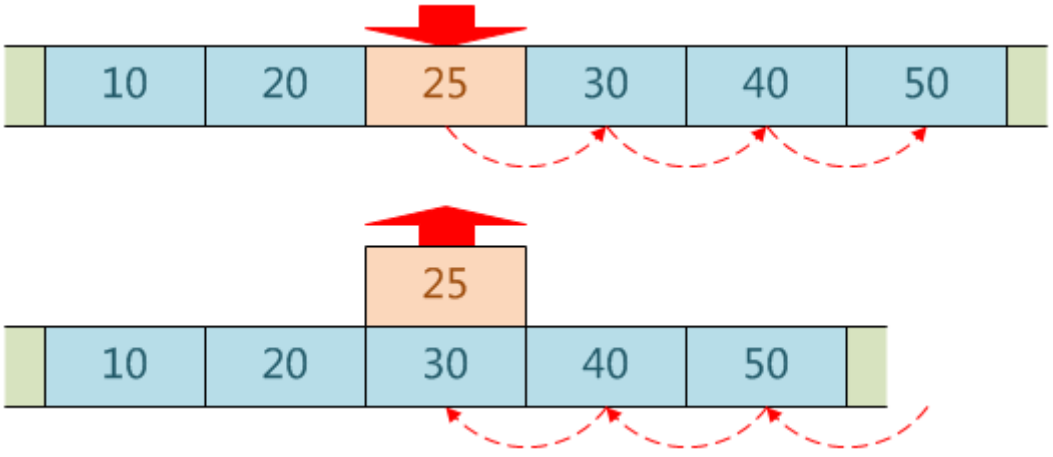
```
vector<string> vs;
vs.push_back("C++");
vs.push_back("喜欢");
vs.push_back("我们");
while (!vs.empty()) {
    cout << vs.back() << flush;
    vs.pop_back();
}
cout << endl;
```

9.3.3.3 插入/删除元素

```
iterator insert(iterator loc, value_type const& val); // 在容器中指定位置之前插入一个元素，并返回指向该元素的迭代器
iterator erase(iterator loc);                         // 删除容器中位于指定位置处的元素，并返回指向该元素下一个位置的迭代器
```

例如：

```
vector<int> vn(1, 55);
vn.insert(vn.insert(vn.insert(vn.insert(vn.begin(), 44), 33), 22), 11);
vector<int>::iterator it = vn.begin();
while (it != vi.end())
    if (*it % 2)
        it = vn.erase(it);
    else
        ++it;
```



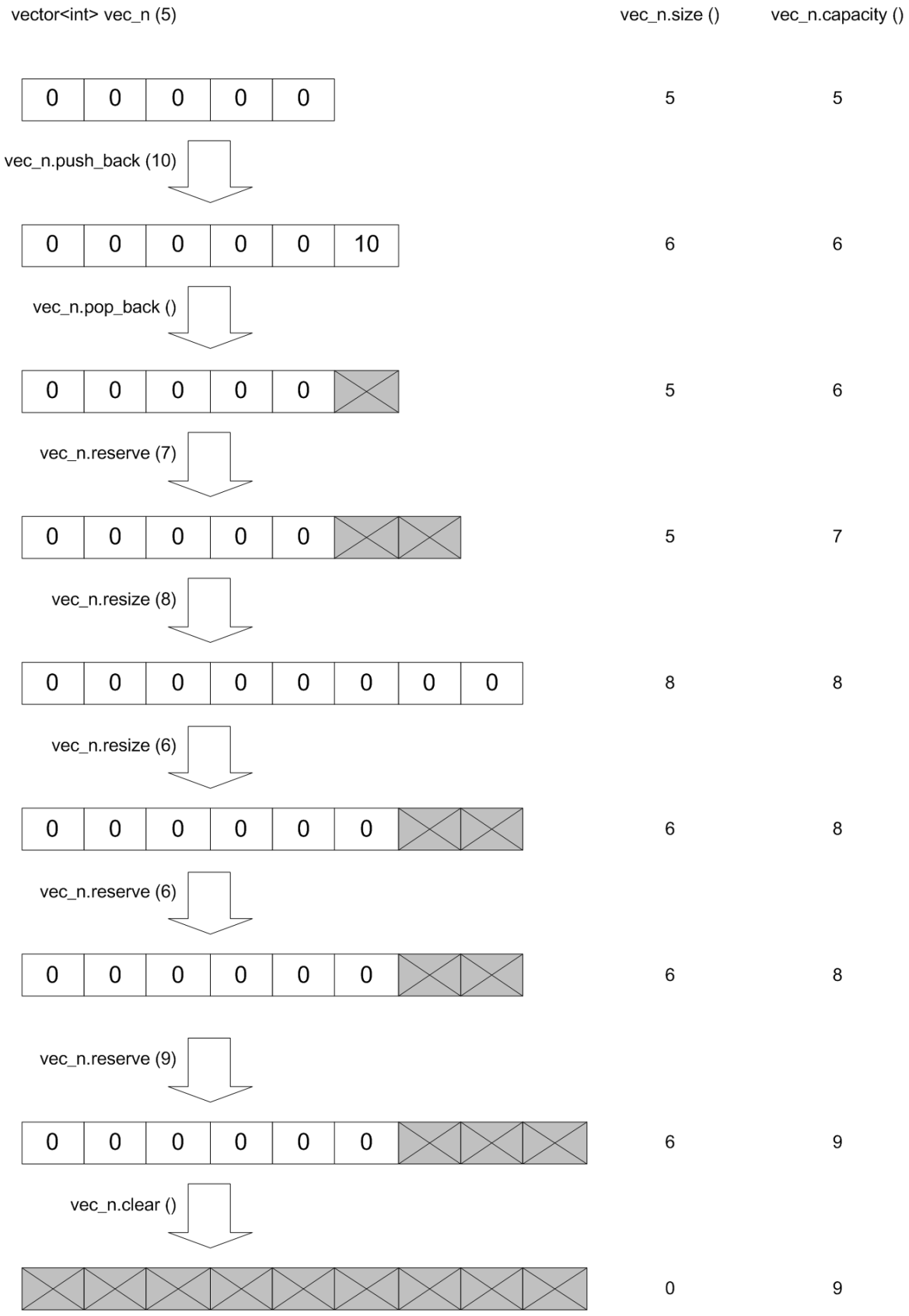
9.3.3.4 大小和容量

vector类模板提供一系列与大小和容量有关的成员函数：

- size函数返回向量的大小，即元素的个数
- resize函数可以改变向量的大小，如有新增元素，这些元素将被初始化为适当（基本）类型的零，或通过相应（类）类型的缺省构造函数初始化
- clear函数可以清空向量中的所有元素，使向量的大小变成0，但它的容量却仍保持不变。resize(0)也可以达到同样的效果
- capacity函数返回向量的容量，即最多能容纳多少个元素

- reserve函数为向量预留一些内存空间，扩充其容量。其间既不做初始化，也不调用构造函数，更不改变向量的大小

```cpp
// vecsize.cpp

// 向量的大小和容量

#include <iostream>
#include <vector>

using namespace std;

void print(vector<int> const& vn) {
    size_t size = vn.size();
    for (size_t i = 0; i < size; ++i)
        cout << vn[i] << ' ';
    cout << "- " << size << endl;

    size_t capacity = vn.capacity();
    for (size_t i = 0; i < capacity; ++i)
        cout << vn[i] << ' ';
    cout << "- " << capacity << endl;
}

int main(void) {
    vector<int> vn(5);
    print(vn);

    vn.push_back(10);
    print(vn);

    vn.pop_back();
    print(vn);

    vn.reserve(11);
    print(vn);

    vn.resize(12);
    print(vn);

    vn.resize(5);
    print(vn);

    vn.reserve(5);
    print(vn);

    vn.clear();
    print(vn);

    return 0;
}
```



由图中不难看出，向量的大小和容量具有如下特性：

- 向量的大小可以增加也可以减少，引起向量大小变化的成员函数除resize外，还有push_back、pop_back、insert、erase和clear等
- 向量的容量只能增加不能减少，只能通过reserve成员函数改变向量的容量
- 向量大小的增加可能会导致其容量的增加，但向量容量的变化并不会影响其大小
- 通过resize成员函数增加向量的大小，新增部分被初始化为适当类型的零或做缺省构造

- 通过reserve成员函数增加向量的容量，新增部分不做任何初始化
- 位于向量的容量范围内但不在其大小范围内的元素，也可以通过下标或迭代器进行访问，但访问结果未定义
- 由resize和reserve成员函数所引起的，向量大小和容量的变化，都发生在该向量的尾部

9.3.4 查找和排序

```cpp
// vecinsert.cpp

// 插入/删除向量中的元素

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

template<typename T>
void print(vector<T> const& vec) {
    for (typename vector<T>::const_iterator it = vec.begin();
        it != vec.end(); ++it)
        cout << *it << ' ';

    cout << endl;
}

int main(void) {
    vector<int> vn(5);
    print(vn); // 0 0 0 0 0

    vector<int>::iterator it = vn.begin();
    cout << *it << endl; // 0

    vn.insert(vn.begin(), 9);
    print(vn); // 9 0 0 0 0 0
    cout << *it << endl; // 未定义值
    it = vn.begin();
    cout << *it << endl; // 9

    vn.erase(vn.begin());
    print(vn); // 0 0 0 0 0

    vn.insert(vn.begin() + 2, 8);
    print(vn); // 0 0 8 0 0 0

    vn.erase(vn.begin() + 2);
    print(vn); // 0 0 0 0 0

    vn.insert(vn.end(), 7);
    print(vn); // 0 0 0 0 0 7

    vn.erase(vn.end() - 1);
    print(vn); // 0 0 0 0 0

```

```
    vn.insert(vn.begin() + 3, 6);
    print(vn); // 0 0 0 6 0 0
    vn.erase(find(vn.begin(), vn.end(), 6));
    print(vn); // 0 0 0 0 0

    vn.insert(vn.begin() + 1, 3);
    vn.insert(vn.begin() + 5, 3);
    print(vn); // 0 3 0 0 0 3 0
    it = find(vn.begin(), vn.end(), 3);
    if (it != vn.end())
        vn.erase(it);
    else
        cout << "Unable to find" << endl;
    print(vn); // 0 0 0 0 3 0
    it = find(vn.begin(), vn.end(), 3);
    if (it != vn.end())
        vn.erase(it);
    else
        cout << "Unable to find" << endl;
    print(vn); // 0 0 0 0 0

    vn[0] = 23;
    vn[1] = 2;
    vn[2] = 16;
    vn[3] = 45;
    vn[4] = 16;
    print(vn); // 23 2 16 45 16
    sort(vn.begin(), vn.end());
    print(vn); // 2 16 16 23 45

    return 0;
}
```

9.3.4.1 在特定区域内查找

```
iterator find(iterator begin, iterator end, value_type const& key);
// 成功返回第一个匹配元素的迭代器，失败返回第二个参数（指向查找范围内最后一个元素的下一个位置的迭代器）
```

例如：

```
template<typename T>
void remove(vector<T>& vec, T const& key) {
    for (typename vector<T>::iterator it = vec.begin();
        (it = find(it, vec.end(), key)) != vec.end(); it = vec.erase(it));
}
```

9.3.4.2 在特定区域内排序

```
void sort(iterator begin, iterator end);           // 通过小于操作符比较元素大小
void sort(iterator begin, iterator end, less cmp); // 通过小于比较器比较元素大小
```

小于比较器可以是形如：

```
bool comparator(int const& a, int const& b) {
    return a < b;
}
```

的函数，也可以是实现了小括号操作符的函数对象：

```
class Comparator {
public:
    bool operator()(int const& a, int const& b) {
        return a < b;
    }
};
```

旨在定义容器中元素间的（小于）比较规则，以供排序之需。

一种类型最多只能定义一个小于操作符函数，因此通过小于操作符定义比较规则的灵活性通常较差。例如：

```
// lesssort.cpp

// 在排序过程中使用小于操作符比较元素大小

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

class Student {
public:
    Student(string const& name, int age) : name(name), age(age) {}

    string const& getName(void) const {
        return name;
    }

    int getAge(void) const {
        return age;
    }

    bool operator<(Student const& student) const {
        if (name == student.name)
            return age < student.age;

        return name < student.name;
    }

private:
    string name;
    int age;
};

ostream& operator<<(ostream& os, Student const& student) {
    return os << '(' << student.getName() << ',' << student.getAge() << ')';
```

```
}

template<typename T>
void print(vector<T> const vec) {
    for (typename vector<T>::const_iterator it = vec.begin();
        it != vec.end(); ++it)
        cout << *it;

    cout << endl;
}

int main(void) {
    vector<Student> students;

    students.push_back(Student("zhangfei", 26));
    students.push_back(Student("guanyu", 43));
    students.push_back(Student("huangzhong", 32));
    students.push_back(Student("zhaoyun", 45));
    students.push_back(Student("huangzhong", 26));

    print(students);

    sort(students.begin(), students.end());

    print(students);

    return 0;
}
```

而小于比较器则可以定义多个，可以根据需要选择适用的比较器版本，因此灵活性往往更好。例如：

```
// funcsort.cpp

// 在排序过程中使用小于比较器比较元素大小

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

class Student {
public:
    Student(string const& name, int age) : name(name), age(age) {}

    string const& getName(void) const {
        return name;
    }

    int getAge(void) const {
        return age;
    }

private:
    string name;
```

```
    int age;
};

ostream& operator<<(ostream& os, Student const& student) {
    return os << '(' << student.getName() << ',' << student.getAge() << ')';
}

bool nameComparator(Student const& a, Student const& b) {
    if (a.getName() == b.getName())
        return a.getAge() < b.getAge();

    return a.getName() < b.getName();
};

class AgeComparator {
public:
    bool operator()(Student const& a, Student const& b) const {
        if (a.getAge() == b.getAge())
            return a.getName() < b.getName();

        return a.getAge() < b.getAge();
    }
};

template<typename T>
void print(vector<T> const vec) {
    for (typename vector<T>::const_iterator it = vec.begin();
        it != vec.end(); ++it)
        cout << *it;

    cout << endl;
}

int main(void) {
    vector<Student> students;

    students.push_back(Student("zhangfei", 26));
    students.push_back(Student("guanyu", 43));
    students.push_back(Student("huangzhong", 32));
    students.push_back(Student("zhaoyun", 45));
    students.push_back(Student("huangzhong", 26));

    print(students);

    sort(students.begin(), students.end(), nameComparator);

    print(students);

    sort(students.begin(), students.end(), AgeComparator());

    print(students);

    return 0;
}
```

9.3.5 类类型元素

```cpp
// veccls.cpp

// 用vector存放自定义类型对象

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

class Integer {
public:
    Integer(int n = 0) : n(new int(n)) {}

    Integer(Integer const& i) : n(new int(*i.n)) {}

    Integer& operator=(Integer const& i) {
        if (&i != this)
            *n = *i.n;

        return *this;
    }

    ~Integer(void) {
        delete n;
    }

    void set(int n) {
        *this->n = n;
    }

    int get(void) const {
        return *n;
    }

    bool operator<(Integer const& i) const {
        return *n < *i.n;
    }

    bool operator==(Integer const& i) const {
        return *n == *i.n;
    }

private:
    int* n;
};

ostream& operator<<(ostream& os, Integer const& i) {
    return os << i.get();
}

class IntegerComparator {
public:
```

```
    bool operator()(Integer const& a, Integer const& b) const {
        return a.get() < b.get();
    }
};

template<typename T>
void print(vector<T> vec) {
    cout << "size=" << vec.size() << endl;

    for (typename vector<T>::const_iterator it = vec.begin();
        it != vec.end(); ++it)
        cout << *it << ' ';

    cout << endl;
}

int main(void) {
    vector<Integer> v1;
    Integer i1(2), i2(3), i3(5);
    v1.push_back(i1);
    v1.push_back(i2);
    v1.push_back(i3);
    print(v1); // 2 3 5

    i2.set(1000);
    print(v1); // 2 3 5
    v1[1].set(1000);
    print(v1); // 2 1000 5

    vector<Integer> v2(5);
    print(v2); // 0 0 0 0 0
    v2.resize(7);
    print(v2); // 0 0 0 0 0 0 0
    v2.reserve(10);
    cout << v2.capacity() << endl; // 10
    print(v2); // 0 0 0 0 0 0 0
    // cout << v2[7].get() << endl; // 未定义

    v2[0] = 12;
    v2[1] = 36;
    v2[2] = 3;
    v2[3] = 56;
    v2[4] = 2;
    sort(v2.begin(), v2.end());
    print(v2); // 0 0 2 3 12 36 56

    v2[0] = 12;
    v2[1] = 36;
    v2[2] = 33;
    v2[3] = 7;
    v2[4] = 22;
    sort(v2.begin(), v2.end(), IntegerComparator());
    print(v2); // 7 12 22 33 36 36 56

    vector<Integer>::iterator it = find(v2.begin(), v2.end(), 33);
    cout << *it << endl; // 33
```

```
110
111     v2.clear();
112     print(v2); // 空
113     cout << v2.capacity() << endl; // 10
114
115     return 0;
116 }
```

9.3.5.1 缺省构造

如果一个类类型对象需要被存储在向量中，那么该类至少应支持缺省构造，以确保向量内存的初始化。

```
1  class Integer {
2      ...
3
4      Integer(int n = 0) : n(new int(n)) {}
5
6      ...
7  };
```

```
1  ...
2  vector<Integer> vi(5);
3  ...
4  vi.resize(7);
5  ...
```

9.3.5.2 拷贝构造和拷贝赋值

该类还需要支持完整意义上的拷贝构造和拷贝赋值。

```
1  class Integer {
2      ...
3
4      Integer(Integer const& i) : n(new int(*i.n)) {}
5
6      Integer& operator=(Integer const& i) {
7          if (&i != this)
8              *n = *i.n;
9
10         return *this;
11     }
12
13     ...
14 };
```

```
1  ...
2  vi.push_back(100);
3  ...
4  vi.erase(vi.begin());
5  ...
```

9.3.5.3 小于和等于

该类可能还需要支持“<”和“==”两个关系操作符，用于元素间的小于和等于判断，以支持排序和查找。

```
class Integer {
    ...

    bool operator<(Integer const& i) const {
        return *n < *i.n;
    }

    bool operator==(Integer const& i) const {
        return *n == *i.n;
    }

    ...
};
```

```
...
sort(vi.begin(), vi.end());
...
vector<Integer>::iterator it = find(vi.begin(), vi.end(), 33);
...
```

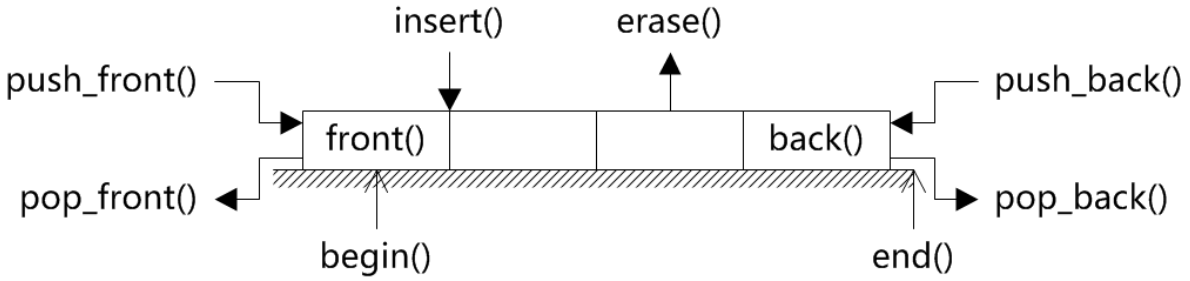
作为小于操作符的替代方案，也可以提供一个定义了小于比较规则的小于比较器（函数或函数对象）。

```
class IntegerComparator {
    ...

    bool operator()(Integer const& a, Integer const& b) const {
        return a.get() < b.get();
    }

    ...
};
```

```
...
sort(vi.begin(), vi.end(), IntegerComparator());
...
```

9.4 双端队列与列表

9.4.1 双端队列



deque

9.4.1.1 具有向量的所有功能

双端队列具有向量的所有功能，除了没有capacity和reserve两个成员函数。

9.4.1.2 首尾增删效率一致

在双端队列的头部与在其尾部进行插入（insert）或删除（erase）操作的效率一样高。

```
vector<int> vn(5);
vn.insert(vn.begin() + 1, 10); // 移动4个元素
vn.insert(vn.end() - 1, 10);   // 移动1个元素
```

```
deque<int> dn(5);
dn.insert(dn.begin() + 1, 10); // 移动1个元素
dn.insert(dn.end() - 1, 10);   // 移动1个元素
```

9.4.1.3 时空复杂度比向量高

- 和向量相比，双端队列的内存开销要大一些，对元素下标访问的效率也要略低一些
- 为了保持连续存储空间首尾两端的开放性，双端对列的动态内存管理比向量要复杂一些
- 双端队列需要预留更多的内存，以应对首尾两端的动态增长，其空间复杂度较向量略高
- 双端队列虽然也提供了下标访问的功能，但其首端的开放性势必会增加计算起始地址的时间复杂度

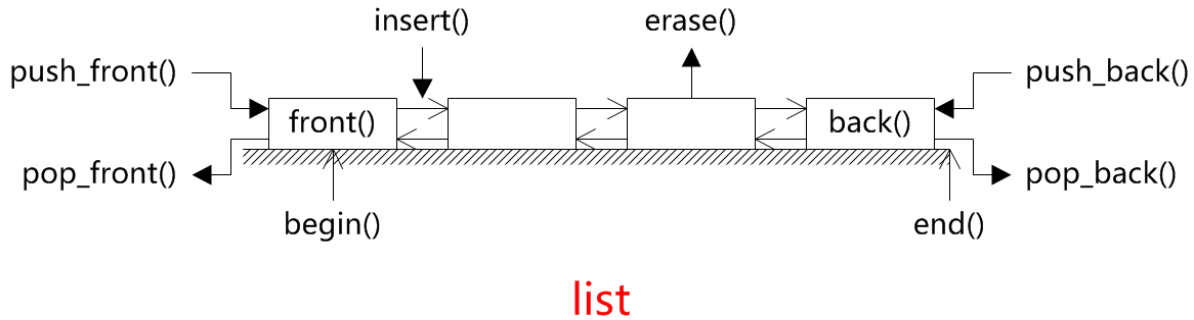
9.4.1.4 可在头部压入和弹出元素

- 双端队列所提供的push_front和pop_front成员函数，可以与push_back和pop_back成员函数相类似的方式，在容器的头部压入和弹出元素，其效率也是相当的
- 双端队列的操作性更接近于列表，但它地址连续的存储结构又与向量别无二致。双端队列在随机访问、插入删除等操作的性能方面往往也介于列表和向量之间
- 相对于强调随机访问的向量和突出增删效率的列表，双端队列的表现更加趋于中庸，在需求尚不明确的情况下选择双端队列，即使算不上是最优的，至少也不至于是最差的

```
// deque.cpp

// 双端队列

#include <iostream>
#include <deque>
#include <algorithm>

using namespace std;
```

```cpp

template<typename T>
void print(deque<T> const& deq) {
    cout << "size=" << deq.size() << endl;

    for (typename deque<T>::const_iterator it = deq.begin();
        it != deq.end(); ++it)
        cout << *it << ' ';

    cout << endl;
}

int main(void) {
    deque<string> ds;
    ds.push_back("广州");
    ds.push_back("深圳");
    ds.push_front("上海");
    ds.push_front("北京");
    print(ds); // 北京 上海 广州 深圳

    ds.pop_front();
    ds.pop_back();
    print(ds); // 上海 广州

    ds.erase(find(ds.begin(), ds.end(), "广州"));
    print(ds); // 上海

    ds.insert(ds.begin(), "北京");
    print(ds); // 北京 上海

    ds.resize(5);
    ds[2] = "杭州";
    ds[3] = "广州";
    ds[4] = "深圳";
    print(ds); // 北京 上海 杭州 广州 深圳

    ds.erase(ds.begin() + 2);
    print(ds); // 北京 上海 广州 深圳

    sort(ds.begin(), ds.end ());
    print(ds); // 上海 北京 广州 深圳

    return 0;
}
```

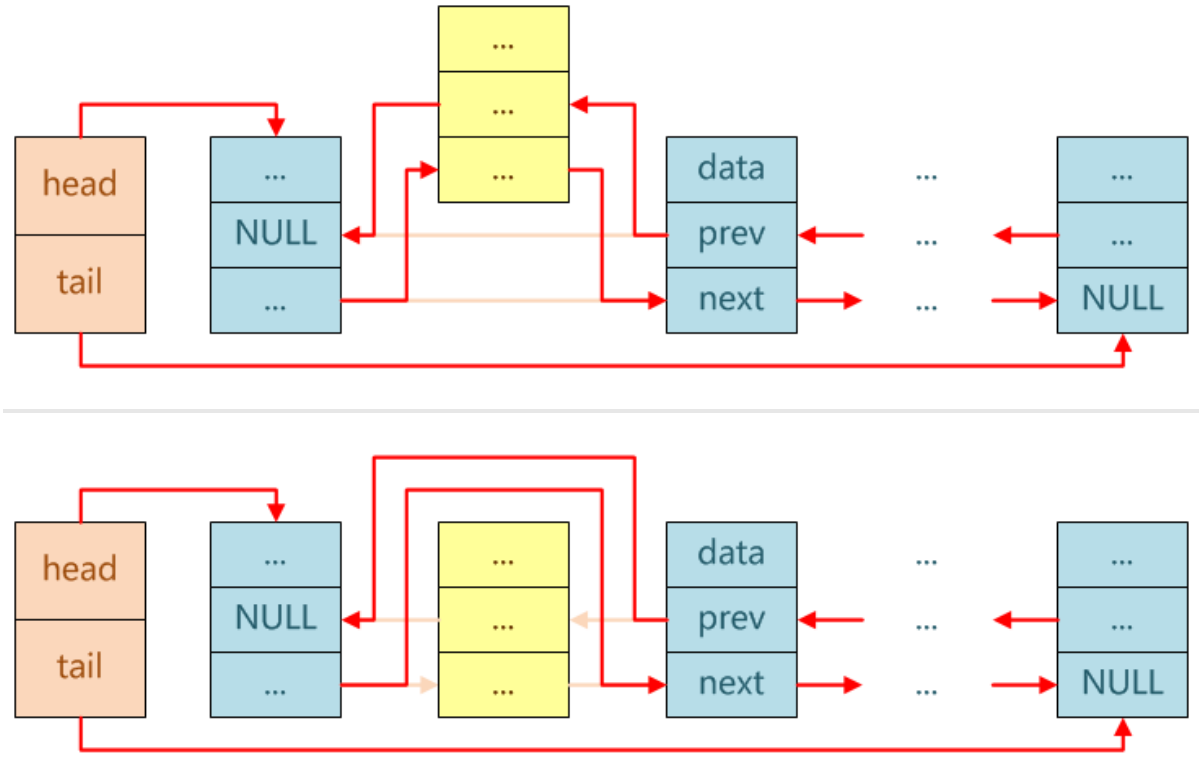
9.4.2 列表



9.4.2.1 链式线性表（链表）

列表是按照链式线性表（链表）的形式进行存储的，元素存放在地址不连续的离散节点中，节点通过前后指针彼此关联，符合数据结构中双向线性链表的结构特征。

9.4.2.2 任意位置增删效率都很高



在列表的任何位置插入或删除元素的效率都很高。离散的链式存储结构决定了在列表中增删元素不需要移动现有节点。所要做的唯一工作仅仅是调整前后节点的指针，以维持链式结构的连续性，而完成这种工作所花费的时间与列表中的元素个数无关，与被增删元素的位置亦无关，即所谓常数时间$O(1)$，这也是选择列表容器的理由之一。

9.4.2.3 禁止随机访问

无法通过下标或迭代器对列表中的元素做随机访问。因为对于列表来说，执行这样的操作至少需要线性时间$O(N)$。STL的设计哲学是只在容器内部实现具有较好性能表现的操作，而把那些时间或者空间复杂度偏高的工作，留给用户根据实际应用的具体需求量力而为。这总比包揽一切却处处挨骂要好。

STL容器中只有向量和双端队列采用连续内存存放数据元素，只有这两个容器的随机访问（指针计算）可以在常数时间内完成，因此只有这两个容器的迭代器支持随机迭代，同时也只有这两个容器提供下标操作符。其它容器，包括列表，都只能做顺序迭代，而且也不支持下标操作符。

9.4.2.4 删匹配

```
1 void remove(const value_type& val); // 删除容器中所有与参数匹配的元素
```

例如：

```
list<int> ln;
ln.push_back(13);
ln.push_back(27);
ln.push_back(13);
ln.push_back(39);
ln.push_back(13);
ln.remove(13);
for (list<int>::const_iterator it = ln.begin(); it != ln.end(); ++it)
    cout << *it << ' '
cout << endl; // 27 39
```

9.4.2.5 唯一化

```
void unique(void); // 连续重复出现的元素只保留一个
```

例如：

```
int an[] = {10, 10, 20, 20, 10, 20, 30, 20, 20, 10, 10};
list<int> ln(an, an + sizeof(an) / sizeof(an[0]));
ln.unique();
for (list<int>::const_iterator it = ln.begin(); it != ln.end(); ++it)
    cout << *it << ' ';
cout << endl; // 10 20 10 20 30 20 10
```

9.4.2.6 拆分

将参数列表的部分或全部元素剪切到调用列表中。

```
void splice(iterator pos, list& lst);                                  // 将lst列表中的全部元素剪切到调用列表的pos处
void splice(iterator pos, list& lst, iterator del);                    // 将lst列表del处的元素剪切到调用列表的pos处
void splice(iterator pos, list& lst, iterator begin, iterator end); // 将lst列表中位于begin和end之间的元素剪切到调用列表的pos处
```

用两个迭代器表示一个容器中的元素范围，其中的上限迭代器，通常指向该范围中最后一个元素的下一个位置。

列表拆分的时间复杂度为常数级$O(1)$。

9.4.2.7 合并

将有序参数列表中的全部元素，合并到有序的调用列表中，合并后的调用列表依然有序，参数列表为空。

```
void merge(list& lst);           // 利用元素类型的“<”操作符比大小
void merge(list& lst, less cmp); // 利用小于比较器比大小
```

列表合并的过程中并没有排序，因此其时间复杂度仅为线性级$O(N)$。

9.4.2.8 排序

列表有自己的排序函数。

```
void sort(void);      // 利用元素类型的“<”操作符比大小
void sort(less cmp); // 利用小于比较器比大小
```

全局函数sort只适用于拥有随机迭代器的容器，列表的迭代器是顺序迭代器，不能使用全局域的sort排序函数。

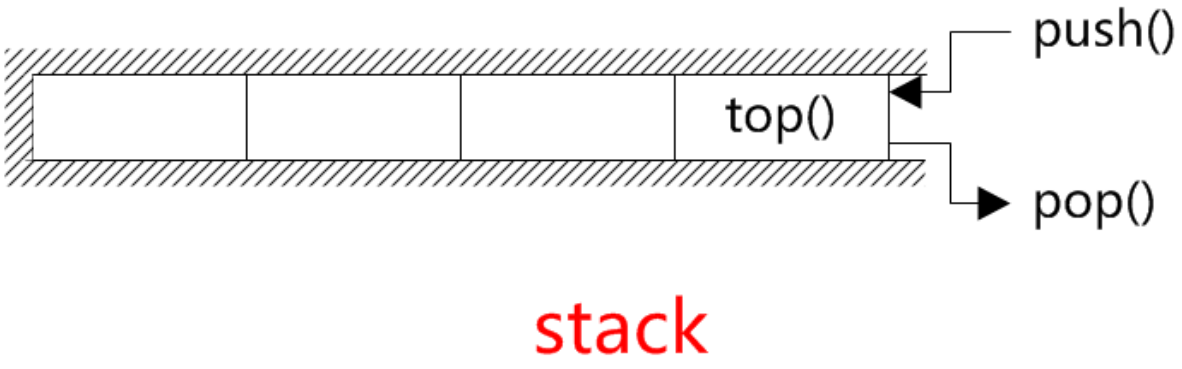
列表利用自身内存不连续的特点，可以更好的性能实现排序算法函数。

```
// stl_list.cpp

// 列表

#include <iostream>
#include <list>

using namespace std;

template<typename T>
void print(list<T> const& lst) {
    cout << "size=" << lst.size() << endl;

    for (typename list<T>::const_iterator it = lst.begin();
        it != lst.end(); ++it)
        cout << *it << ' ';

    cout << endl;
}

int main(void) {
    list<string> l1;
    l1.push_back("华为");
    l1.push_back("腾讯");
    l1.push_back("阿里");
    l1.push_back("腾讯");
    l1.push_back("字节");
    l1.push_back("百度");
    print(l1); // 华为 腾讯 阿里 腾讯 字节 百度

    l1.pop_back();
    print(l1); // 华为 腾讯 阿里 腾讯 字节

    l1.remove("腾讯");
    print(l1); // 华为 阿里 字节

    l1.push_front("腾讯");
    print(l1); // 腾讯 华为 阿里 字节

    l1.pop_front();
    print(l1); // 华为 阿里 字节

    l1.insert(l1.end(), "腾讯");
```

```
    print(l1); // 华为 阿里 字节 腾讯

    l1.sort();
    print(l1); // 华为 字节 腾讯 阿里

    l1.push_front("华为");
    l1.push_front("华为");
    l1.push_back("华为");
    l1.push_back("华为");
    list<string>::const_iterator it = l1.begin();
    ++it;
    ++it;
    ++it;
    ++it;
    l1.insert(l1.insert(it, "字节"), "字节");
    print(l1); // 华为 华为 华为 字节 字节 字节 腾讯 阿里 华为 华为

    l1.unique();
    print(l1); // 华为 字节 腾讯 阿里 华为

    list<string> l2;
    l2.push_back("Microsoft");
    l2.push_back("Oracle");
    l2.push_back("Google");
    l2.push_back("Amazon");
    print(l2); // Microsoft Oracle Google Amazon

    l1.splice(l1.begin(), l2, l2.begin());
    print(l1); // Microsoft 华为 字节 腾讯 阿里 华为
    print(l2); // Oracle Google Amazon

    l1.sort();
    l2.sort();
    l1.merge(l2);
    print(l1); // Amazon Google Microsoft Oracle 华为 华为 字节 腾讯 阿里
    print(l2); // 空

    return 0;
}
```

9.5 堆栈、队列和优先队列

9.5.1 堆栈

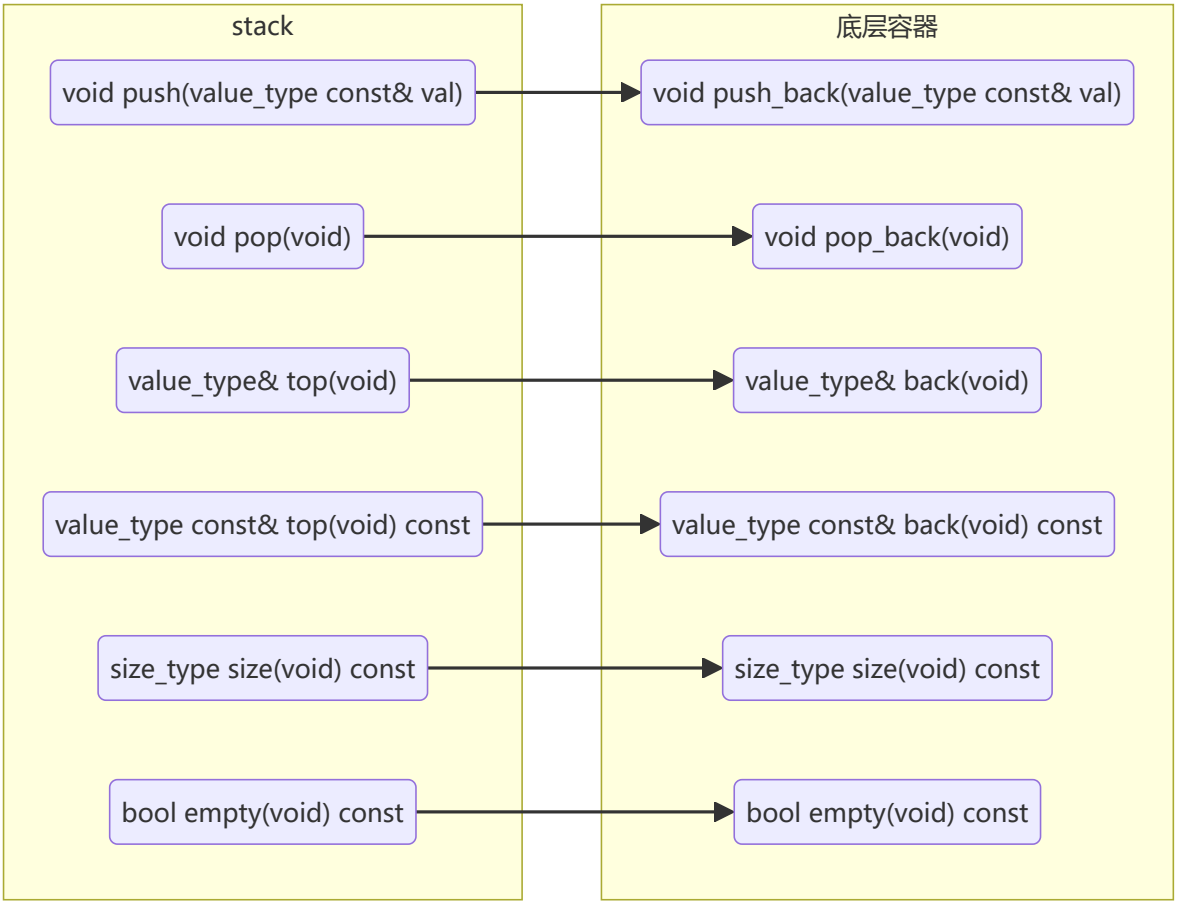


9.5.1.1 对底层容器的要求

堆栈作为一种适配器容器，可以用任何支持push_back、pop_back、back、size和empty等操作的底层容器进行适配。

9.5.1.2 满足要求即可适配

除了STL的三种线性容器外，程序员自定义的容器，只要提供了正确的接口函数，也可用于适配堆栈。



适配器模式的本质就是接口翻译，即将一种形式的接口翻译成另一种形式，以下是堆栈容器的简化实现：

```
template<typename value_type, typename container_type = deque<value_type> >
class stack {
public:
    void push(value_type const& val) {
        container.push_back(val);
    }

    void pop(void) {
        container.pop_back();
    }

    value_type& top(void) {
        return container.back();
    }

    value_type const& top(void) const {
        return const_cast<stack*>(this)->top();
    }
```

```
    size_t size(void) const {
        return container.size();
    }

    bool empty(void) const {
        return container.empty();
    }

private:
    container_type container; // 底层容器
};
```

9.5.1.3 底层容器的类型

定义堆栈容器时可以指定底层容器的类型。

```
stack<string, vector<string> > ss;
```

在C++11以前，两个右尖括号之间至少要留一个空格，否则编译器会把“>>”误解为右移操作符，导致编译错误。

三种线性容器，向量、双端队列和列表中的任何一种都可以作为堆栈的底层容器，其中双端队列是缺省底层容器。

```
stack<int> sn;
```

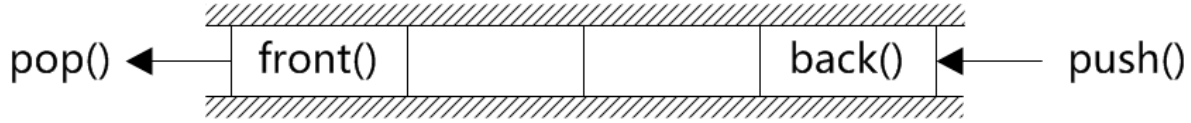
等价于

```
stack<int, deque<int> > sn;
```

```
// stack.cpp

// 堆栈

#include <iostream>
#include <stack>
#include <vector>

using namespace std;

int main(void) {
    stack<string, vector<string> > ss;

    ss.push("C++!");
    ss.push("学习");
    ss.push("我们");

    while (!ss.empty()) {
        cout << ss.top();
        ss.pop();
    }

```

```
23      cout << endl;
24
25      return 0;
26  }
```

9.5.2 队列



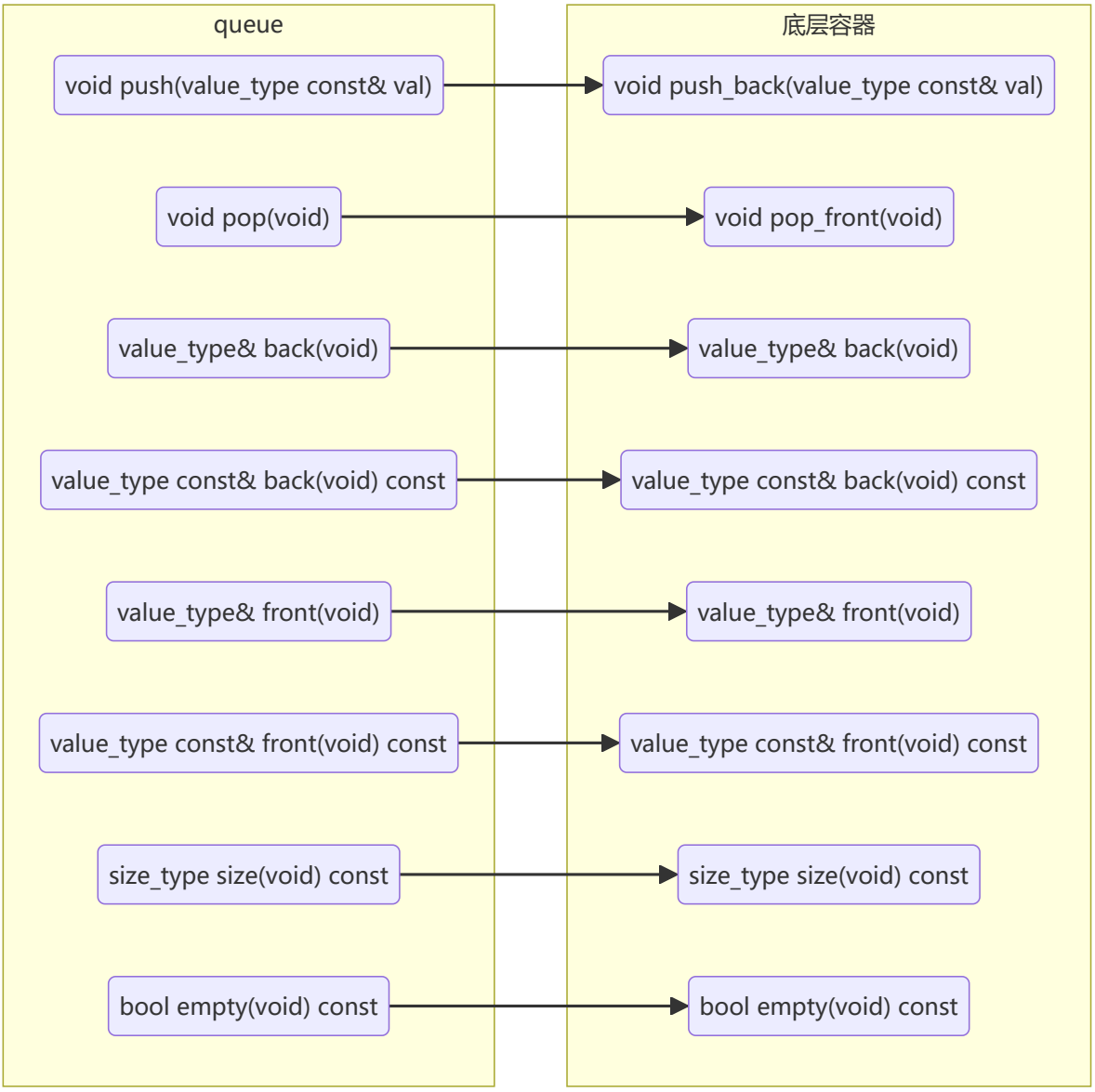
queue

9.5.2.1 对底层容器的要求

队列作为一种适配器容器，可以用任何支持push_back、pop_front、 back、front、 size和empty等操作的底层容器进行适配。

9.5.2.2 满足要求即可适配

除了STL的三种线性容器外，程序员自定义的容器，只要提供了正确的接口函数，也可用于适配队列。



9.5.2.3 底层容器的类型

定义队列容器时可以指定底层容器的类型。

```
queue<string, list<string> > qs;
```

在C++11以前，两个右尖括号之间至少要留一个空格，否则编译器会把“>>”误解为右移操作符，导致编译错误。

三种线性容器中除向量以外的任何一种都可以作为队列的底层容器，其中双端队列是缺省底层容器。

```
queue<int> qn;
```

等价于

```
queue<int, deque<int> > qn;
```

不能用向量适配队列的原因是，它缺少pop_front接口。

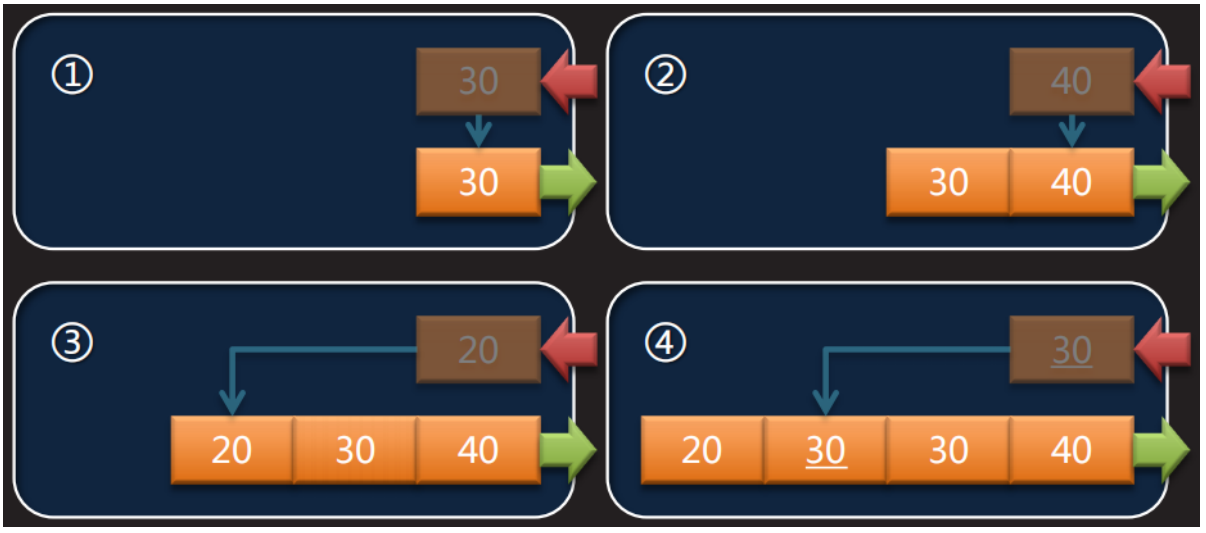
```
// queue.cpp

```

```cpp
// 队列

#include <iostream>
#include <queue>
#include <list>

using namespace std;

int main(void) {
    queue<string, list<string> > qs;

    qs.push("我们");
    qs.push("学习");
    qs.push("C++!");

    while (!qs.empty()) {
        cout << qs.front();
        qs.pop();
    }

    cout << endl;

    return 0;
}
```

9.5.3 优先队列

9.5.3.1 压入元素大者优先

每当向优先队列中压入一个元素，队列中具有最高优先级的元素会被自动排至队首（类似插入排序）。因此从优先队列中弹出元素，既不是先进先出，也不是后进先出，而是优者先出，即谁的优先级高谁先出队。只有那些优先级相同的元素才遵循和队列一样的先进先出规则。



9.5.3.2 用小于操作符比大小

缺省情况下，优先队列利用元素类型的“<”操作符比较大小，谁大谁的优先级高。

```cpp
class Student {
public:
    Student(string const& name = "", int age = 0, int score = 0)
```

```
        : name(name), age(age), score(score) {}

    bool operator<(Student const& student) const {
        return age < student.age; // 年长者优
    }

    ...

private:
    string name;
    int age, score;
};
```

```
priority_queue<Student, vector<Student> > ps;
```

9.5.3.3 用小于比较器比大小

也可以通过小于比较器比较大小，确定优先级高低。

```
class Student {
public:
    Student(string const& name = "", int age = 0, int score = 0)
        : name(name), age(age), score(score) {}

    ...

private:
    string name;
    int age, score;

    friend class PrioritizeByScore;
};
```

```
class PrioritizeByScore {
public:
    bool operator()(Student const& a, Student const& b) const {
        return a.score < b.score; // 分高者优
    }
};
```

```
priority_queue<Student, vector<Student>, PrioritizeByScore> ps;
```

9.5.3.4 底层容器的类型

定义优先队列容器时可以指定底层容器的类型。

```
priority_queue<string, vector<string> > ps;
```

在C++11以前，两个右尖括号之间至少要留一个空格，否则编译器会把“>>”误解为右移操作符，导致编译错误。

三种线性容器中除列表以外的任何一种都可以作为优先队列的底层容器，其中双端队列是缺省底层容器。

```
priority_queue<int> pn;
```

等价于

```
priority_queue<int, deque<int> > pn;
```

不能用列表适配优先队列的原因是，它缺少按优先级调整元素顺序时所需要的随机迭代器。

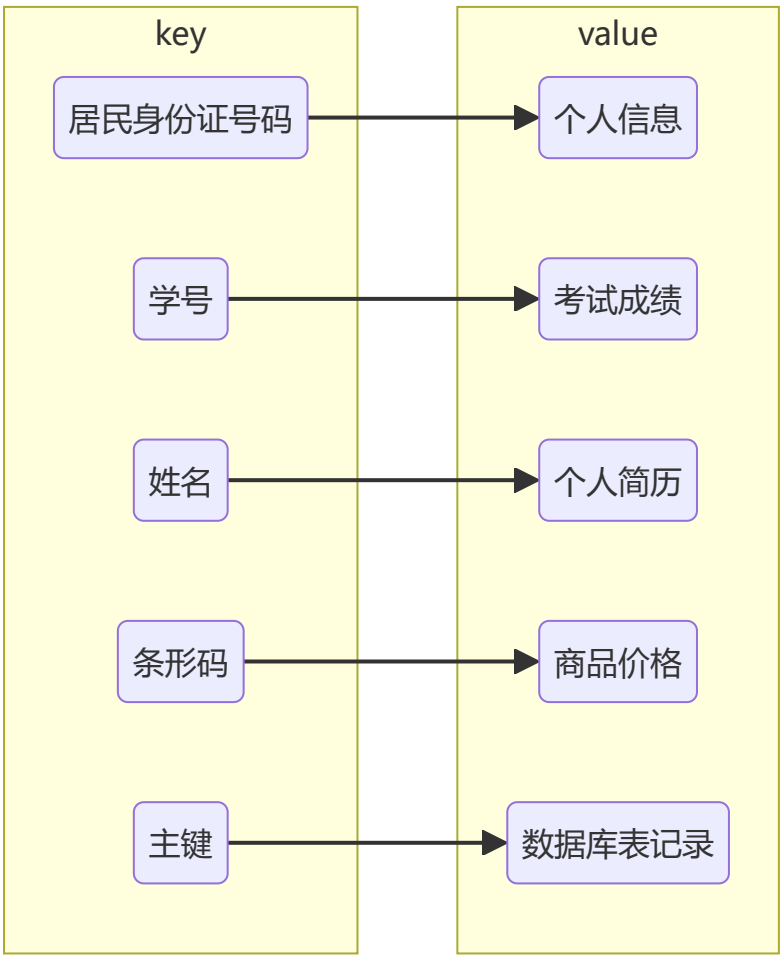
```
// prique.cpp

// 优先队列

#include <iostream>
#include <queue>

using namespace std;

class PrioritizeByTemperature {
public:
    bool operator()(const pair<string, unsigned int>& a,
        const pair<string, unsigned int>& b) const {
        return a.second < b.second;
    }
};

int main(void) {
    priority_queue<string> p1;

    p1.push("changsha");
    p1.push("xian");
    p1.push("nanjing");

    while (!p1.empty()) {
        cout << p1.top() << endl;
        p1.pop();
    }

    priority_queue<pair<string, unsigned int>,
        vector<pair<string, unsigned int> >, PrioritizeByTemperature> p2;

    p2.push(make_pair("changsha", 30));
    p2.push(make_pair("xian", 25));
    p2.push(make_pair("nanjing", 20));

    while (!p2.empty()) {
        cout << p2.top().first << ": " << p2.top().second << endl;
        p2.pop();
    }

    return 0;
}
```

9.6 映射与多重映射

9.6.1 映射

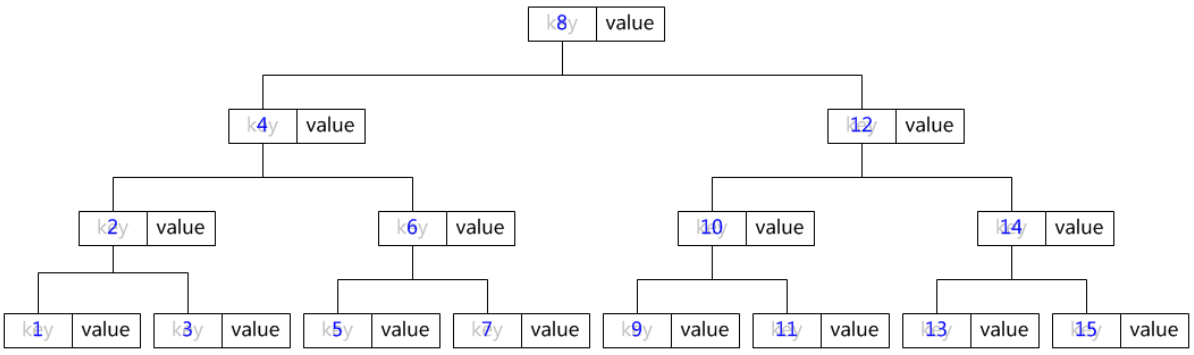
9.6.1.1 key-value对

映射将现实世界中不同类型事物间的对应关系，抽象为由一系列key-value（键——值）对组成的集合，并提供通过key快速检索与之相对应的value的方法。



9.6.1.2 红黑树

在映射内部，所有的key-value对被组织成以key为基准的红黑树（自平衡有序二叉树）的形式。



map

映射中的key必须是唯一的，表达一一对应的逻辑关系。

9.6.1.3 根据key找value

基于有序二叉树的结构特征，映射可以在对数时间$O(logN)$内，根据key找到与之相对应的value。因此映射主要被应用在需要高速检索特性的场合。

具有平衡有序二叉树特征的红黑树，相较于一般有序二叉树的特殊之处在于，其任何节点的左右子树大小之差不会超过1。这样做既有利于保证其搜索性能的均化，又可以有效避免单连枝树向链表退化的风险。

当由于向红黑树中插入或删除节点而导致其平衡性遭到破坏时，必须根据具体情况对树进行一次或多次的单旋或双旋，直至其恢复平衡。因此映射的构建和修改通常较耗时，不适于存储频繁变化的key。

9.6.1.4 以key为索引的下标操作

映射支持以key为索引的下标操作。

```
value_type& operator[](key_type const& key);
```

返回与参数key相对应的value的引用，若key不存在，则新建一个key-value对，其中的value按缺省方式初始化。

映射的下标运算符既可用于插入又可用于检索。

```
map<string, int> sn;
sn["张飞"] = 10000; // 插入
sn["张飞"] = 20000; // 检索
cout << sn["张飞"] << endl; // 20000
```

9.6.1.5 key类型的小于操作符

映射内部为了维护其二叉树结构的有序性，必然需要对key做大小比较。缺省情况下这项工作由key类型的“<”操作符来完成。

```
map<string, int> sn;
++sn["tarena"];
++sn["TARENA"];
++sn["Tarena"];
cout << sn["tarena"] << endl; // 1
```

注意string类型的“<”操作符是大小写敏感的。

9.6.1.6 key类型的小于比较器

也可以通过针对key类型的小于比较器自定义排序规则。

```
class StrCaseCmp {
public:
    bool operator()(string const& a, string const& b) const {
        return strcasecmp(a.c_str(), b.c_str ()) < 0; // 大小写不敏感的字符串比较
    }
};
```

```
map<string, int, StrCaseCmp> sn;
++sn["tarena"];
++sn["TARENA"];
++sn["Tarena"];
cout << sn["tarena"] << endl; // 3
```

9.6.1.7 基本单元

映射的基本访问单元和存储单元都是pair，pair只有两个成员变量，first和second，分别表示key和value。

```
template<typename first_type, typename second_type>
class pair {
public:
    pair(first_type const& first, second_type const& second)
        : first(first), second(second) {}

    first_type first;
    second_type second;
};
```

```
map<string, int> sn;
sn.insert(pair<string, int>("张飞", 10000));
sn.insert(make_pair("赵云", 20000));
```

9.6.1.8 插入元素

```
pair<iterator, bool> insert(pair<key_type, value_type> const& pair);
```

- 插入位置由映射根据key的有序性确定
- 插入成功，作为返回值的pair对象中，second成员为true，first成员为指新插入元素的迭代器
- 插入失败，作为返回值的pair对象中，second成员为false，first成员未定义

例如：

```
map<string, int> sn;
pair<map<string, int>::iterator, bool> res = msi.insert(
    pair<string, int>("张飞", 10000));
if (!res.second)
    cout << "插入失败" << endl;
else
    cout << res.first->first << "->" << res.first->second << endl;
```

9.6.1.9 删除元素

```
void erase(iterator pos);                  // 删除指定位置的元素
void erase(iterator begin, iterator end); // 删除指定范围的元素
size_type erase(key_type const& key);      // 删除与指定key匹配的元素
```

9.6.1.10 查找元素

```
iterator find(key_type const& key);
```

- 查找与key匹配的元素，对数时间复杂度$O(logN)$，成功返回指向匹配元素的迭代器，失败返回终止迭代器
- 与全局线性查找函数find不同，映射采用基于树的查找，并不要求key的类型支持“==”操作符，仅支持“<”操作符或小于比较器即可，其匹配逻辑类似下面这样：

```
if (目标key < 节点key)
    在该节点的左子树中继续查找
else
if (节点key < 目标key)
    在该节点的右子树中继续查找
else
    该节点即为匹配节点，找到了
```

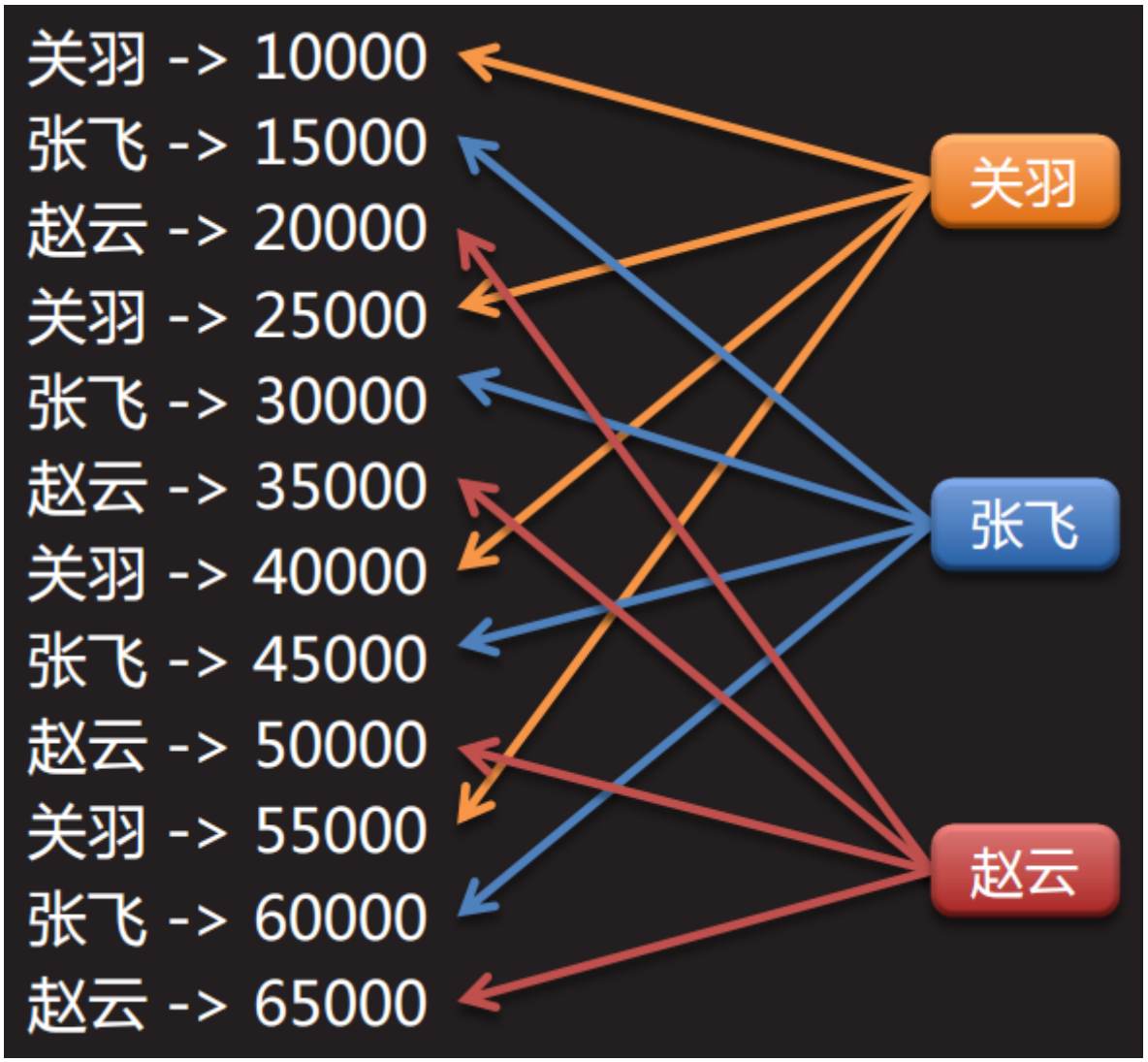
```
// map.cpp

// 映射

#include <iostream>
#include <map>

using namespace std;

class Candidate {
public:
    Candidate(string const& name = "") : name(name), votes(0) {}

    string const& getName(void) const {
        return name;
    }

    size_t getVotes(void) const {
        return votes;
    }

    void vote(void) {
        ++votes;
    }

private:
    string name;
    size_t votes;
};

int main(void) {
    map<char, Candidate> candidates;

    candidates.insert(pair<char, Candidate>('A', Candidate("张飞")));
    candidates.insert(make_pair('B', Candidate("关羽")));
    candidates['C'] = Candidate("赵云");
```

```
    candidates['D'] = Candidate("黄忠");
    candidates['E'] = Candidate("马超");

    for (size_t i = 0; i < 10; ++i) {
        for (map<char, Candidate>::const_iterator it = candidates.begin();
            it != candidates.end(); ++it)
            cout << '(' << it->first << ") " << it->second.getName() << ' ';

        cout << endl << "请投票：" << flush;
        char key;
        cin >> key;

        map<char, Candidate>::iterator it = candidates.find(key);
        if (it == candidates.end())
            cout << "弃权。" << endl;
        else
            it->second.vote();
    }

    map<char, Candidate>::const_iterator win = candidates.begin();

    for (map<char, Candidate>::const_iterator it = candidates.begin();
        it != candidates.end(); ++it) {
        cout << it->second.getName() << "获得" << it->second.getVotes() <<
"票。" << endl;

        if (it->second.getVotes() > win->second.getVotes())
            win = it;
    }

    cout << "热烈祝贺" << win->second.getName() << "成功当选！" << endl;

    return 0;
}
```

9.6.2 多重映射

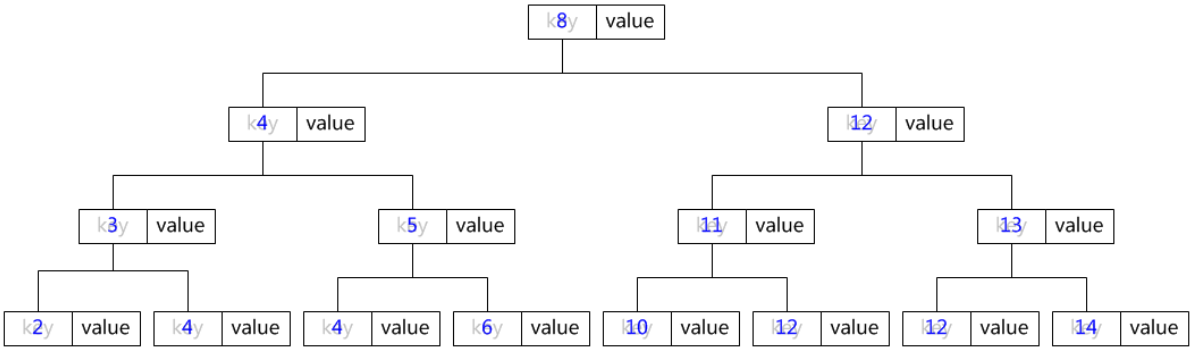
9.6.2.1 允许key重复

允许key重复的映射即为多重映射，表示一对多的逻辑关系。比如下图显示销售员——季度销售额对应关系：



9.6.2.2 不支持下标操作

多重映射无法根据给定的key唯一确定与之对应的value，因此多重映射不支持以key为索引的下标操作。



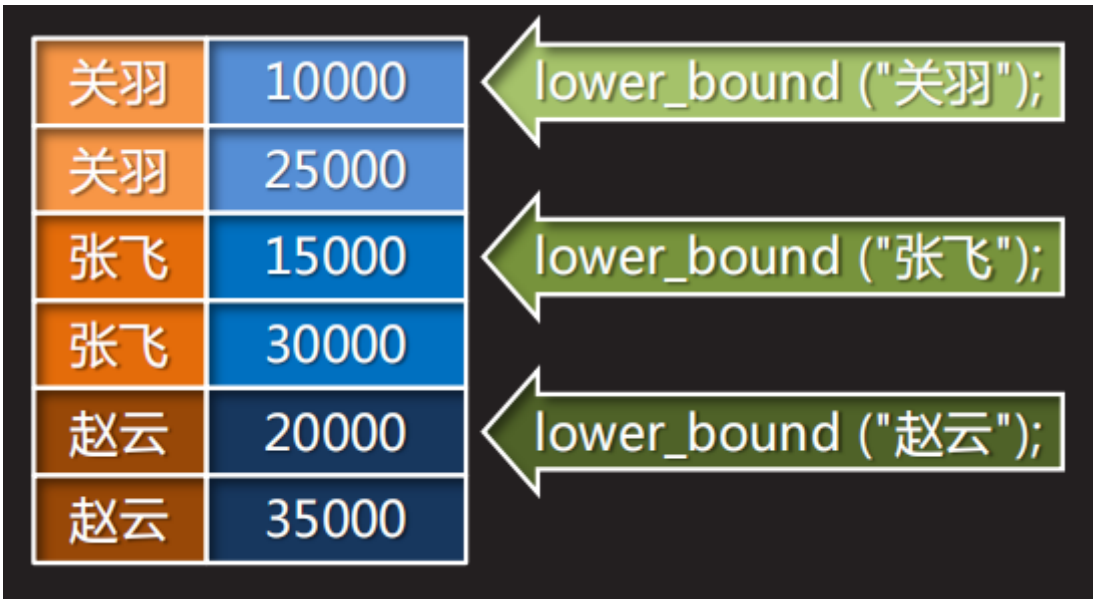
multimap

9.6.2.3 获取匹配下限

```
iterator lower_bound(key_type const& key); // 返回指向中序遍历中第一个键大于或等于key的元素的迭代器
```

例如：

```
multimap<string, int> sn;
sn.insert(make_pair("关羽", 10000));
sn.insert(make_pair("张飞", 15000));
sn.insert(make_pair("赵云", 20000));
sn.insert(make_pair("关羽", 25000));
sn.insert(make_pair("张飞", 30000));
sn.insert(make_pair("赵云", 35000));
```

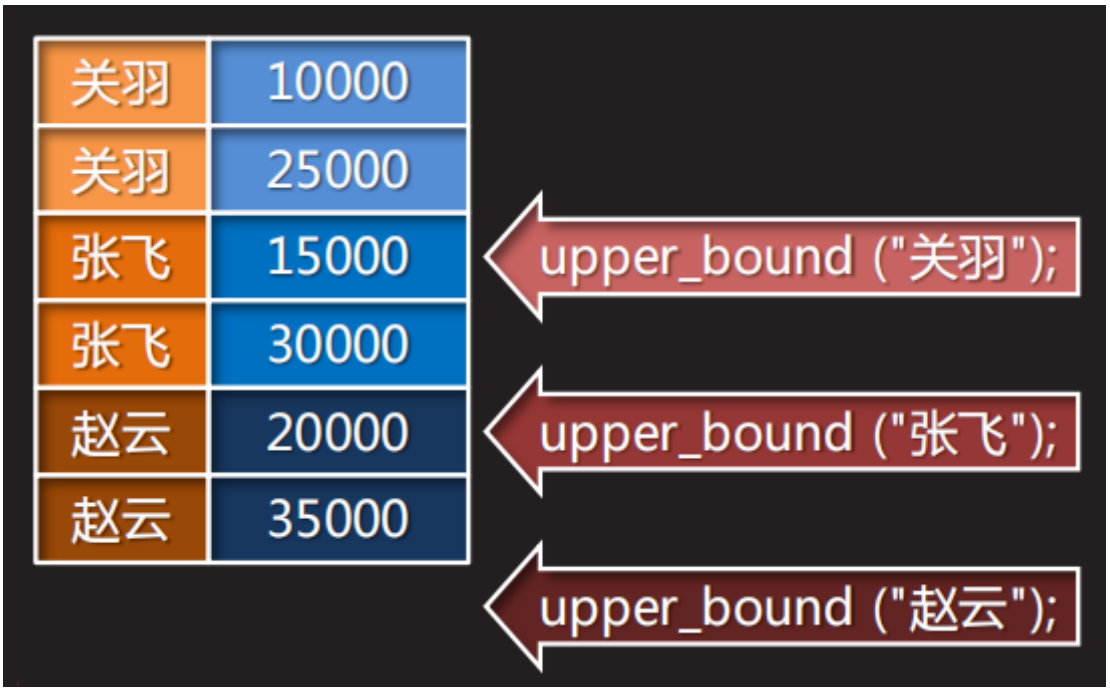


9.6.2.4 获取匹配上限

```
iterator upper_bound(key_type const& key); // 返回指向中序遍历中第一个键大于key的元素的迭代器
```

例如：

```
multimap<string, int> sn;
sn.insert(make_pair("关羽", 10000));
sn.insert(make_pair("张飞", 15000));
sn.insert(make_pair("赵云", 20000));
sn.insert(make_pair("关羽", 25000));
sn.insert(make_pair("张飞", 30000));
sn.insert(make_pair("赵云", 35000));
```



9.6.2.5 获取匹配范围

```
pair<iterator, iterator> equal_range(key_type const& key); // 返回由匹配下限迭代器（first）和匹配上限迭代器（second）组成的pair对象
```

例如：

```
multimap<string, int> sn;
...
pair<multimap<string, int>::iterator, multimap<string, int>::iterator> range =
    sn.equal_range("张飞");
for (multimap<string, int>::iterator it = range.first; it != range.second;
++it)
    cout << it->first << "->" << it->second << endl;
```

```
// mmap.cpp

// 多重映射

#include <iostream>
#include <map>

using namespace std;

int main(void) {
    multimap<string, int> sn;

    sn.insert(make_pair("张飞", 100000));
    sn.insert(make_pair("赵云", 150000));
    sn.insert(make_pair("张飞", 200000));
    sn.insert(make_pair("关羽", 250000));
    sn.insert(make_pair("赵云", 300000));
    sn.insert(make_pair("关羽", 350000));

    for (multimap<string, int>::const_iterator it = sn.begin();
```

```
        it != sn.end(); ++it)
        cout << it->first << "->" << it->second << endl;

    multimap<string, int>::const_iterator it = sn.find("赵云");
    cout << it->first << "--" << it->second << endl;

    for (multimap<string, int>::const_iterator it = sn.begin ();
        it != sn.end(); ++it) {
        string key = it->first;

        multimap<string, int>::const_iterator upper = sn.upper_bound(key);

        int sum = 0;
        for (it; it != upper; ++it)
            sum += it->second;
        cout << key << "=>" << sum << endl;

        --it;
    }

    return 0;
}
```

9.7 集合与多重集合

9.7.1 集合

集合就是没有value的映射，其中的每个元素相当于映射中的key，必须是唯一的。

```
// set.cpp

// 集合

#include <iostream>
#include <set>

using namespace std;

int main(void) {
    cout << "第一个字符串：" << flush;
    string str1;
    cin >> str1;

    set<char> set1;
    for (string::const_iterator it = str1.begin(); it != str1.end(); ++it)
        set1.insert(*it);

    cout << "第二个字符串：" << flush;
    string str2;
    cin >> str2;

    set<char> set2;
    for (string::const_iterator it = str2.begin(); it != str2.end(); ++it)
        set2.insert(*it);

```

```
    cout << "两串共有字符：" << flush;
    for (set<char>::const_iterator it = set1.begin(); it != set1.end();
++it)
        if (!set2.insert(*it).second) // 插不进去的必是相同的
            cout << *it;
    cout << endl;

    return 0;
}
```

9.7.2 多重集合

多重集合就是没有value的多重映射，其中的每个元素相当于多重映射中的key，允许重复。

```
// mset.cpp

// 多重集合

#include <iostream>
#include <set>

using namespace std;

int main(void) {
    multiset<string> mset;

    mset.insert("张飞");
    mset.insert("关羽");
    mset.insert("赵云");
    mset.insert("张飞");
    mset.insert("赵云");
    mset.insert("关羽");
    mset.insert("赵云");

    for (multiset<string>::const_iterator it = mset.begin(); it !=
mset.end(); ++it)
        cout << *it << endl;

    for (multiset<string>::const_iterator it = mset.begin(); it !=
mset.end(); ++it) {
        cout << *it << '(' << mset.count(*it) << ')' << endl;
        it = mset.upper_bound(*it);
        --it;
    }

    return 0;
}
```

9.8 字符串

C++中的string实际上是basic_string模板用char类型实例化后的类型别名。

```
template<typename charT, ...> class basic_string { ... };
```

```
typedef basic_string<char> string;
```

需要包含头文件<string>。

9.8.1 实例化和操作符

9.8.1.1 基本用法

```
// string.cpp

// string的基本用法

#include <iostream>

using namespace std;

int main(void) {
    string s1("hello world"); // 在栈中隐式实例化string对象
    cout << s1 << endl;

    string s2 = "hello world"; // 在栈中隐式实例化string对象
    cout << s2 << endl;

    string s3 = string("hello world"); // 在栈中显示实例化string对象
    cout << s3 << endl;

    string* s4 = new string("hello world"); // 在堆中显示实例化string对象
    cout << *s4 << endl;
    delete s4;

    string s5; // 在栈中隐式实例化空string对象
    cout << '[' << s5 << ']' << endl;

    string s6(""); // 在栈中隐式实例化空string对象
    cout << '[' << s6 << ']' << endl;

    string s7 = ""; // 在栈中隐式实例化空string对象
    cout << '[' << s7 << ']' << endl;

    string s8 = string(""); // 在栈中显示实例化空string对象
    cout << '[' << s8 << ']' << endl;

    string* s9 = new string; // 在堆中显示实例化空string对象
    cout << '[' << *s9 << ']' << endl;
    delete s9;

    // 从C风格字符串到string对象
    char s10[] = "hello world";
    string s11(s10);
    cout << s11 << endl;

    // 从string对象到C风格字符串
    const char* s12 = s11.c_str();
    cout << s12 << endl;
```

```
47
48      return 0;
49  }
```

9.8.1.2 基本运算

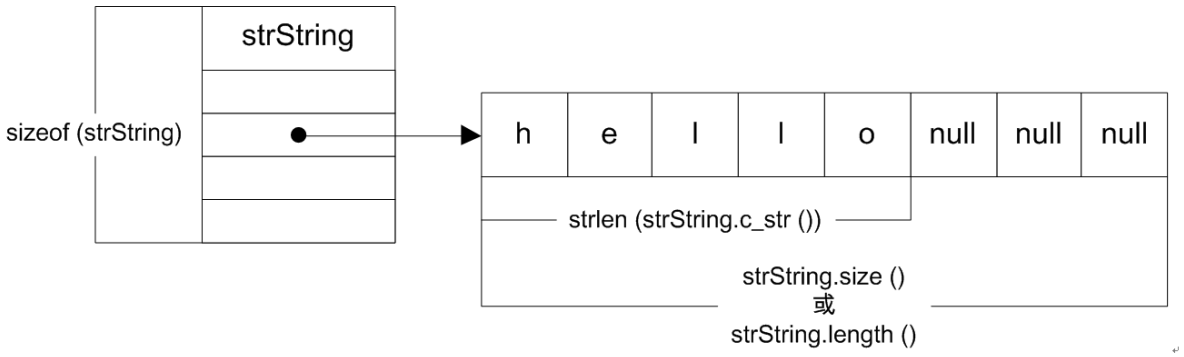
```
1   // strop.cpp
2
3   // string的基本运算
4
5   #include <iostream>
6
7   using namespace std;
8
9   int main(void) {
10      // =
11      string s1("new");
12      string s2("old");
13      cout << s2 << endl;
14      s2 = s1;
15      cout << s2 << endl;
16
17      // +和+=
18      string s3("hello");
19      string s4("world");
20      string s5 = s3 + " " + s4;
21      cout << s5 << endl;
22      (s3 += " ") += s4;
23      cout << s3 << endl;
24
25      // >
26      string s6("abcde");
27      string s7("1bcde");
28      cout << boolalpha << (s6 > s7) << endl;
29
30      // >=
31      string s8("12345");
32      string s9("12345");
33      cout << (s8 >= s9) << endl;
34
35      // <
36      string s10("abcde");
37      string s11("abcdef");
38      cout << (s10 < s11) << endl;
39
40      // <=
41      string s12("!@#$%");
42      string s13("!@#$%");
43      cout << (s12 <= s13) << endl;
44
45      // ==
46      string s14("hello");
47      string s15("hello");
48      cout << (s14 == s15) << endl;
49
```

```
    // !=
    string s16("c++");
    string s17("C++");
    cout << (s16 != s17) << endl;

    return 0;
}
```

9.8.2 大小和长度

- string::size()：以字符为单位的字符串容量
- string::length()：以字符为单位的字符串容量
- strlen()：以字符为单位的空字符结尾字符串长度
- sizeof()：以字节为单位的string对象本身的大小

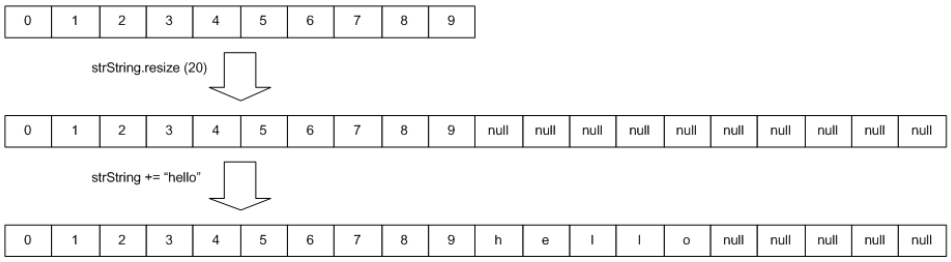
```
string str("hello");
str.resize(8);
```

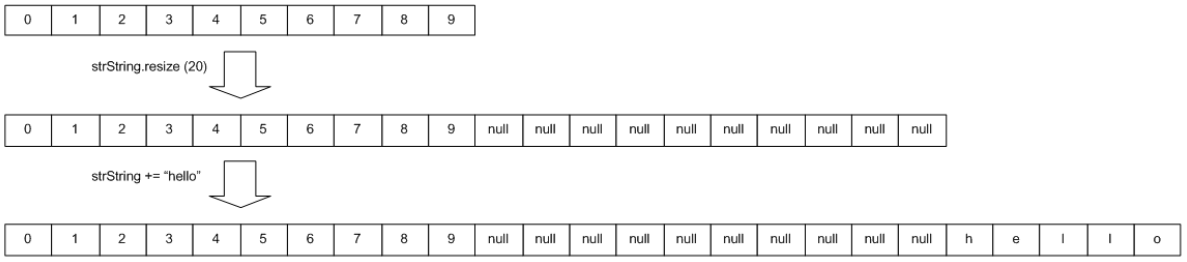



```
// strsize.cpp

// string的容量、长度和大小

#include <stdio.h>
#include <string.h>

#include <iostream>

using namespace std;

void print(string const& s) {
    cout << "容量: " << s.size() << endl;
    cout << "容量: " << s.length() << endl;
    cout << "长度: " << strlen(s.c_str()) << endl;
    cout << "大小: " << sizeof(s) << endl;

    for (size_t i = 0; i < s.size(); ++i)
        printf("|%02X", s[i]);
    cout << '|' << endl;

    cout << s << endl;

    cout << s.c_str() << endl;
```

```
}

int main(void) {
    string s = "0123456789";
    print(s);

    s.resize(5);
    print(s);

    s.resize(10);
    print(s);

    s += "abcdefghij";
    print(s);

    return 0;
}
```

- string的“+”和“+=”运算可能因实现而异





- string的大小可能因实现而异
  - 某些C++实现，string实际上只包含一个char*类型的非静态成员变量，该变量保存一个以'\0'字符结尾的字符数组的地址。如下所示：

    ```
    class string {
        char* pc;
    };
    ```

    因此，一个string对象的大小，实际上就是它唯一的一个非静态成员变量的大小——8个字节（64位系统）
  - 而另一些C++实现，string还包括一个size_t类型的非静态成员变量，用于保存字符串的长度。如下所示：

```
class string {
    char* pc;
    size_t len;
};
```

这种情况下，成员变量pc所指向的字符数组就不必再以'\0'字符结尾了。而这样的string对象的大小，显然是它的两个非静态成员变量大小的总和——16个字节（64位系统）

- 不同C++实现的string对象会有不同的成员，其大小并没有统一标准

9.8.3 拼接、搜索和子串

9.8.3.1 拼接

```
string& append(string const& str);
string& append(string const& str, size_type index, size_type len);
```

```
// strapp.cpp

// 字符串拼接

#include <iostream>

using namespace std;

int main(void) {
    string s1 = "hello";
    string s2 = " the world at large";

    string s3 = s1;
    s3.append(s2); // 将s2追加到s3尾部
    cout << s3 << endl;

    string s4 = s1;
    s4.append(s2, 4, 6); // 从s2的第4个（基零）字符开始，取6个字符追加到s4尾部
    cout << s4 << endl;

    return 0;
}
```

9.8.3.2 搜索

```
size_type find_first_of(string const& str, size_type index = 0);
```

```
// strsearch.cpp

// 在字符串中搜索特定字符

#include <iostream>

using namespace std;

int main(void) {
```

```
    string s1 = "Ah, Why, ye Gods, should two and two make four ? - Alexander Pope";
    cout << s1 << endl;

    string s2 = ",.?:;-";
    cout << s2 << endl;

    string::size_type pos = 0;
    size_t nmarks = 0;

    // 返回s1中从pos处开始的第一个出现在s2中的字符的位置

    while ((pos = s1.find_first_of(s2, pos)) != string::npos) {
            ++pos;
            ++nmarks;
    }

    cout << "Find " << nmarks << " marks" << endl;

    return 0;
}
```

9.8.3.3 子串

```
string substr(size_type index, size_type num = npos);
string(string const& str, size_type index, size_type length = npos);
```

```
// substr.cpp

// 从字符串中提取子字符串

#include <iostream>

using namespace std;

int main(void) {
    string s1 = "The difference between reality and fiction ? Fiction has to make sense. - Tom Clancy";
    cout << s1 << endl;

    // 通过string类的substr成员函数提取子字符串

    string s2 = s1.substr(45); // 从s1的第45个（基零）字符开始取到字符串尾
    cout << s2 << endl;

    s2 = s1.substr(45, 7); // 从s1的第45个（基零）字符开始取7个字符
    cout << s2 << endl;

    // 通过string类的构造函数提取子字符串

    string s3(s1, 45); // 从s1的第45个（基零）字符开始取到字符串尾
    cout << s3 << endl;

    string s4(s1, 45, 7); // 从s1的第45个（基零）字符开始取7个字符
```

```
    cout << s4 << endl;

    return 0;
}
```

9.8.4 单字符访问

```
char& operator[](size_type index);
char const& operator[](size_type index) const;
char& at(size_type loc);
char const& at(size_type loc) const;
```

```
// strchar.cpp

// 通过string类的“[]”操作符和at成员函数访问字符串中的单个字符

#include <stdio.h>

#include <iostream>
#include <stdexcept>

using namespace std;

int main(void) {
    string s = "hello";
    cout << s << endl;

    // 通过“[]”操作符访问字符串中的单个字符

    char c = s[0];
    cout << c << endl;
    s[0] = 'H';
    cout << s << endl;

    // 通过at成员函数访问字符串中的单个字符

    c = s.at(0);
    cout << c << endl;
    s.at(0) = 'h';
    cout << s << endl;

    c = s[1000]; // “[]”操作符不做下标溢出检查
    printf("%02X\n", c);

    try {
        c = s.at(1000); // 下标溢出at成员函数会抛出异常
        printf("%02X\n", c);
    }
    catch (out_of_range const& ex) {
        cout << ex.what() << endl;
    }

    return 0;
}
```

9.8.5 高级操作

9.8.5.1 查找与替换

```
size_type find(string const& str, size_type index);
string& replace(size_type index, size_type num, string const& str);
```

```
// strfind.cpp

// 字符串的查找与替换

#include <iostream>

using namespace std;

int main(void) {
    string s1 = "one hello is like any other hello", s2 = "hello", s3 = "armadillo";

    cout << s1 << endl;

    for (string::size_type pos = 0; (pos = s1.find(s2, pos)) != string::npos; pos += s3.size())
        s1.replace(pos, s2.size(), s3);

    cout << s1 << endl;

    return 0;
}
```

9.8.5.2 比较与排序

```
int compare(string const& str);
```

```
// strcomp.cpp

// string的比较与排序

#include <stdlib.h>

#include <iostream>

using namespace std;

// 比较函数返回零、负数或正数，表示第一个参数等于、小于或大于第二个参数

int comparator(void const* a, void const* b) {
    // string类的compare成员函数返回零、负数或正数，表示调用字符串等于、小于或大于参数字符串
    return (**(string**)a).compare(**(string**)b);
}

int main(void) {
```

```
    string* s[] = {
        new string("world"),
        new string("zoo"),
        new string("Moon"),
        new string("MOON"),
        new string("beautiful"),
        new string("love"),
        new string("apple"),
        new string("knife"),
        new string("friend")};

    size_t n = sizeof(s) / sizeof(s[0]);

    for (size_t i = 0; i < n; ++i)
        cout << *s[i] << ' ';
    cout << endl;

    // 通过自定义比较函数进行快速排序

    qsort(s, n, sizeof(s[0]), comparator);

    for (size_t i = 0; i < n; ++i)
        cout << *s[i] << ' ';
    cout << endl;

    for (size_t i = 0; i < n; ++i)
        delete s[i];

    return 0;
}
```

9.8.5.3 插入与删除

```
string& erase(size_type index = 0, size_type num = npos);
string& insert(size_type index, string const& str);
string& insert(size_type index1, string const& str, size_type index2,
size_type num);
```

```
// strinsert.cpp

// 字符串的插入与删除

#include <iostream>

using namespace std;

int main(void) {
    string s = "Some cause happiness wherever they go, others whenever they
to - Oscar Wilde";
    cout << s << endl;

    string::size_type pos = 0;

    if ((pos = s.find("happiness")) != string::npos) {
```

```
        s.erase(pos, 9);
        s.insert(pos, "excitement");
    }
    cout << s << endl;

    s.erase(pos, 11);
    cout << s << endl;

    s.insert(pos, "infinite happiness in the air", 9, 10);
    cout << s << endl;

    return 0;
}
```

9.8.5.4 交换与复制

```
void swap(string& from);
string& assign(string const& str);
```

```
// strswap.cpp

// 字符串的交换与复制

#include <iostream>

using namespace std;

int main(void) {
    string s1 = "hello", s2 = "world";
    cout << s1 << ' ' << s2 << endl;

    s1.swap(s2);
    cout << s1 << ' ' << s2 << endl;

    s1.assign(s2);
    cout << s1 << ' ' << s2 << endl;

    return 0;
}
```

9.9 内存分配器

```
// allocator.cpp

// 内存分配器

#include <iostream>
#include <vector>
#include <limits>

using namespace std;

template<typename T>
```

```cpp
class MyAllocator {
public:
    typedef size_t    size_type;
    typedef ptrdiff_t difference_type;
    typedef T*        pointer;
    typedef const T*  const_pointer;
    typedef T&        reference;
    typedef const T&  const_reference;
    typedef T         value_type;

    template<typename U>
    class rebind {
    public:
        typedef MyAllocator<U> other;
    };

    MyAllocator(void) throw() {}
    MyAllocator(const MyAllocator& allocator) throw() {}
    template<typename U>
    MyAllocator(const MyAllocator<U>& allocator) throw() {}
    ~MyAllocator(void) throw() {}

    pointer address(reference value) const {
        return &value;
    }

    const_pointer address(const_reference value) const {
        return &value;
    }

    size_type max_size(void) const throw() {
        return numeric_limits<size_t>::max();
    }

    pointer allocate(size_type num, const void* hint = 0) {
        return pointer(::operator new(num * sizeof(T)));
    }

    void construct(pointer p, const_reference value) {
        new(p) T(value);
    }

    void destroy(pointer p) {
        p->~T();
    }

    void deallocate(pointer p, size_type num) {
        ::operator delete(p);
    }
};

template<typename T1, typename T2>
bool operator==(const MyAllocator<T1>& a, const MyAllocator<T2>& b) throw()
{
    return true;
}
```

```
67
68  template<typename T1, typename T2>
69  bool operator!=(const MyAllocator<T1>& a, const MyAllocator<T2>& b) throw()
    {
70      return false;
71  }
72
73  int main(void) {
74      vector<int, MyAllocator<int> > vn(5, 10);
75      for (int n = 11; n < 16; ++n)
76          vn.push_back(n);
77
78      for (vector<int, MyAllocator<int> >::const_iterator it = vn.begin();
79          it != vn.end(); ++it)
80          cout << *it << ' ';
81      cout << endl;
82
83      return 0;
84  }
```

9.10 全局迭代器

9.10.1 I/O流迭代器

```
1   // ioit.cpp
2
3   // I/O流迭代器
4
5   #include <iostream>
6   #include <vector>
7   #include <fstream>
8   #include <iterator>
9
10  using namespace std;
11
12  int main(void) {
13      vector<int> vn(10);
14
15      ifstream ifs("i.dat");
16      copy(istream_iterator<int>(ifs), istream_iterator<int>(), vn.begin());
17
18      for (vector<int>::const_iterator it = vn.begin(); it != vn.end(); ++it)
19          cout << *it << ' ';
20      cout << endl;
21
22      ofstream ofs("o.dat");
23      copy(vn.begin(), vn.end(), ostream_iterator<int>(ofs, " "));
24
25      return 0;
26  }
```

9.10.2 插入迭代器

```cpp
// insert.cpp

// 插入迭代器

#include <iostream>
#include <deque>
#include <iterator>

using namespace std;

template<class Iterator>
void print(Iterator begin, Iterator end) {
    copy(begin, end, ostream_iterator<int>(cout, " "));
    cout << endl;
}

int main(void) {
    int an[] = {2, 3, 6, 7};
    deque<int> dn(an, an + 4);
    print(dn.begin(), dn.end());

    front_insert_iterator<deque<int> > front(dn);
    *front = 1;
    *front = 0;
    print(dn.begin(), dn.end());

    back_insert_iterator<deque<int> > back(dn);
    *back = 8;
    *back = 9;
    print(dn.begin(), dn.end());

    insert_iterator<deque<int> > insert(dn, dn.begin() + 4);
    *insert = 4;
    *insert = 5;
    print(dn.begin(), dn.end());

    return 0;
}
```

9.11 常用泛型算法

除了查找（find）和排序（sort）算法以外，STL还提供了其它一些算法，用于完成诸如复制、遍历、计数、合并、填充、比较、变换、删除、划分等多种任务

- STL算法通常只需要以迭代器作为参数，并不需要知道所操作容器的细节，因此STL算法又称为泛型算法
- 对于泛型算法，一个算法能否被应用于一种容器，完全取决于该容器能否满足这个算法对迭代器的要求
- 在STL中，共有60种不同的算法，其中包括23种非修改算法（如find），和37种修改算法（如sort）

9.11.1 复制算法

```
iterator copy(iterator begin, iterator end, iterator dest);
```

```
// copy.cpp

// 复制算法

#include <iostream>
#include <vector>

using namespace std;

template<class T>
void print(vector<T> const& vec) {
    for (typename vector<T>::const_iterator it = vec.begin();
        it != vec.end(); ++it)
        cout << *it << ' ';

    cout << endl;
}

int main(void) {
    int an[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

    vector<int> v1(10);
    copy(an, an + 10, v1.begin());
    print(v1);

    vector<int> v2(10);
    copy(v1.begin(), v1.end(), v2.begin());
    print(v2);

    return 0;
}
```

9.11.2 遍历算法

```
UnaryFunction for_each(iterator begin, iterator end, UnaryFunction f);
```

```
// foreach.cpp

// 遍历算法

#include <string.h>

#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>

using namespace std;

```

```cpp
class Student {
    string name;
    int age;
    double score;

    friend istream& operator>>(istream& is, Student& student) {
        return is >> student.name >> student.age >> student.score;
    }

    friend ostream& operator<<(ostream& os, Student const& student) {
        return os << student.name << ' ' << student.age << ' ' <<
student.score;
    }
};

class Writer {
public:
    Writer(ostream& os) : os(os) {}

    void operator()(Student const& student) const {
        os << student << endl;
    }

private:
    ostream& os;
};

int main(int argc, char* argv[]) {
    if (argc < 2)
        goto usage;

    if (!strcmp(argv[1], "-w")) {
        ofstream ofs ("students.txt");
        if (!ofs) {
            cerr << "创建学生档案文件失败! " << endl;
            return -1;
        }

        cout << "学生人数: " << flush;
        size_t num;
        cin >> num;

        vector<Student> students(num);

        for (size_t i = 0; i < students.size(); ++i) {
            cout << "第" << i + 1 << "个学生: " << flush;
            cin >> students[i];
        }

        for_each(students.begin(), students.end(), Writer(ofs));

        ofs.close();
    }
    else
    if (!strcmp(argv[1], "-r")) {
        ifstream ifs("students.txt");
```

```
        if (!ifs) {
            cerr << "打开学生档案文件失败！" << endl;
            return -1;
        }

        Student student;
        vector<Student> students;
        while (ifs >> student)
            students.push_back(student);

        ifs.close();

        for_each(students.begin(), students.end(), Writer(cout));
    }
    else
        goto usage;

    return 0;

usage:
    cerr << "用法：" << argv[0] << " -w|-r" << endl;
    return -1;
}
```