C++容器库

string容器

string构造函数

- `string();` //创建一个空的字符串 例如: string str;
  `string(const char* s);` //使用字符串s初始化
- `string(const string& str);` //使用一个string对象初始化另一个string对象
- `string(int n, char c);` //使用n个字符c初始化

string赋值

- `string& operator=(const char* s);` **//char*类型字符串 赋值给当前的字符串**
- `string& operator=(const string &s);` //把字符串s赋给当前的字符串
- `string& operator=(char c);` //字符赋值给当前的字符串

- `string& assign(const char *s);` //把字符串s赋给当前的字符串
- `string& assign(const char *s, int n);` **//把字符串s的前n个字符赋给当前的字符串**
- `string& assign(const string &s);` //把字符串s赋给当前字符串
- `string& assign(int n, char c);` //用n个字符c赋给当前字符串

string字符串拼接

- `string& operator+=(const char* str);` //重载+=操作符

- `string& operator+=(const char c);` //重载+=操作符
- `string& operator+=(const string& str);` //重载+=操作符

- `string& append(const char *s);` //把字符串s连接到当前字符串结尾
- `string& append(const char *s, int n);` **//把字符串s的前n个字符连接到当前字符串结尾**
- `string& append(const string &s);` //同operator+=(const string& str)
- `string& append(const string &s, int pos, int n);` **//字符串s中从pos开始的n个字符连接到字符串结尾**

string查找和替换

- `int find(const string& str, int pos = 0) const;` **//查找str第一次出现位置,从pos开始查找**
- `int find(const char* s, int pos = 0) const;` //查找s第一次出现位置,从pos开始查找
- `int find(const char* s, int pos, int n) const;` //从pos位置查找s的前n个字符第一次位置
- `int find(const char c, int pos = 0) const;` //查找字符c第一次出现位置

- `int rfind(const string& str, int pos = npos) const;` //查找str最后一次位置,从pos开始查找
- `int rfind(const char* s, int pos = npos) const;` //查找s最后一次出现位置,从pos开始查找
- `int rfind(const char* s, int pos, int n) const;` //从pos查找s的前n个字符最后一次位置
- `int rfind(const char c, int pos = 0) const;` //查找字符c最后一次出现位置

- `string& replace(int pos, int n, const string& str);` //替换从pos开始n个字符为字符串str
- `string& replace(int pos, int n,const char* s);` **//替换从pos开始的n个字符为字符串s**

string字符串比较

- `int compare(const string &s) const;` //与字符串s比较
- `int compare(const char *s) const;` //与字符串s比较

string字符存取

- `char& operator[](int n);` **//通过[]方式取字符**
- `char& at(int n);` //通过at方法获取字符

string插入和删除

- `string& insert(int pos, const char* s);` **//插入字符串**
- `string& insert(int pos, const string& str);` //插入字符串
- `string& insert(int pos, int n, char c);` //在指定位置插入n个字符c

- `string& erase(int pos, int n = npos);` **//删除从Pos开始的n个字符**

string子串

- `string substr(int pos = 0, int n = npos) const;` //返回由pos开始的n个字符组成的字符串

Vector容器

vector构造函数

- `vector<T> v;` //用模板默认构造函数
- `vector(v.begin(),v.end());` //将该区间内的元素拷贝给本身
- `vector(n);` //构造函数使容器元素个数为n
- `vector(n,elem);` //构造函数将n个elem拷贝给本身
- `vector(const vector &vec);` //拷贝构造函数

vector赋值操作

- `vector& operator = (const vector &vec);` //重载等号操作符
- `assign(beg,end);` //将[beg,end)区间中的数据拷贝赋值给本身
- `assign(n,elem);` //将n个elem拷贝赋值给本身

vector容量和大小

- `empty();` //判断容器是否为空
- `capacity();` //容器的容量
- `size();` //返回容器元素的个数
- `resize(int num);` //重新指定容器的长度为num，若容器变长，则以默认值0填充新位置，否则删除超出容器长度的部分
- `resize(int num, elem);` //重新指定容器的长度为num，若容器变长，则以elem填充新位置，否则删除超出容器长度的部分

vector插入和删除

- `push_back(ele);` //尾部插入元素ele
- `pop_back;` //删除最后一个元素
- `insert(const_iterator pos, ele);` //迭代器指向位置pos插入元素ele
- `insert(const_iterator pos,int count, ele);` // //迭代器指向位置pos插入count个元素ele
- `erase(const_iterator pos);` //删除迭代器指向的元素
- `erase(const_iterator start,const_iterator end);` //删除迭代器从start到end之间的元素
- `clear();` //删除容器中所有元素

vector数据存取

- `at(int idx);` //返回索引idx所指的数据
- `operator[];` //返回索引idx所指的数据
- `front();` //返回容器第一个数据元素
- `back();` //返回容器最后一个数据集元素

vector互换容器

- `swap(vec);`//将vec与本身的元素互换

> 特殊用途：利用匿名容器完成容器收缩：`vector<int> (v).swap(v);`

vector预留空间

- `reserve(int len)` //提前开辟len长度的空间

vector二维构造

- vector< vector< int> >v; 二维向量//这里最外的<>要有空格。
- ```
  int N=5, M=6;
     vector<vector<int> > obj(N); //定义二维动态数组大小5行
     for(int i =0; i< obj.size(); i++)//动态二维数组为5行6列，值全为0
     {
         obj[i].resize(M);
     }
  ```
- ```
  int N=5, M=6;
      vector<vector<int> > obj(N, vector<int>(M)); //定义二维动态数组5行6列
  ```

deque容器

deque构造函数

- `deque<T> deqT;` //用模板默认构造函数
- `deque(v.begin(),v.end());` //将该区间内的元素拷贝给本身
- `deque(n,elem);`//构造函数将n个elem拷贝给本身
- `deque(const deque &deq);` //拷贝构造函数

deque赋值操作

- `deque& operator = (const deque &deq);` //重载等号操作符
- `assign(beg,end);` //将[beg,end)区间中的数据拷贝赋值给本身
- `assign(n,elem);` //将n个elem拷贝赋值给本身

deque容量和大小

- `empty();` //判断容器是否为空
- `size();` //返回容器元素的个数
- `resize(int num);` //重新指定容器的长度为num，若容器变长，则以默认值0填充新位置，否则删除超出容器长度的部分
- `resize(int num, elem);`//重新指定容器的长度为num，若容器变长，则以elem填充新位置，否则删除超出容器长度的部分

deque插入和删除

两端插入操作

- `push_back(ele);` //尾部插入元素ele
- `push_front(ele);` //头部插入元素ele
- `pop_back();` //删除最后一个元素
- `pop_front();` //删除第一个元素

指定位置操作

- `insert(const_iterator pos, ele);` //迭代器指向位置pos插入元素ele
- `insert(const_iterator pos,int count, ele);` // //迭代器指向位置pos插入count个元素ele
- `erase(const_iterator pos);` //删除迭代器指向的元素
- `erase(const_iterator start,const_iterator end);` //删除迭代器从start到end之间的元素
- `clear();` //删除容器中所有元素

deque数据存取

- `at(int idx);` //返回索引idx所指的数据
- `operator[];` //返回索引idx所指的数据
- `front();` //返回容器第一个数据元素
- `back();` //返回容器最后一个数据集元素

deque排序

- `sort(iterator beg, iterator end)` //对beg和end区间内的元素排序

stack容器

stack构造函数

- `stack<T> stk` //模板默认构造形式
- `stack(const stack &stk)` //拷贝构造函数

stack赋值操作

- `stack& operator = (const stack &stk);` //重载等号操作符

stack数据存储

- `push()` //入栈
- `pop()` //出栈
- `top()` //返回栈顶元素

stack大小操作

- `empty()` //判断栈是否为空
- `size()` //返回栈内元素个数

queue容器

queue构造函数

- `queue<T> que` //模板默认构造形式
- `queue(const queue &que)` //拷贝构造函数

queue赋值操作

- `queue& operator = (const queue &que);` //重载等号操作符

queue数据存储

- `push()` //入队
- `pop()` //出队

queue大小操作

- `empty()` //判断队是否为空
- `size()` //返回队内元素个数

优先队列priority_queue

- top 访问队头元素
- empty 队列是否为空
- size 返回队列内元素个数
- push 插入元素到队尾 (并排序)
- emplace 原地构造一个元素并插入队列
- pop 弹出队头元素
- swap 交换内容
- 升序队列 priority_queue <int,vector,greater > q;
- 降序队列 priority_queue <int,vector,less >q;
- 对结构体排序

```
struct cmp    //另辟struct，排序自定义
{
    bool operator () (const student & a,const student & b) const
    {
        if(a.sc != b.sc)    return b.sc > a.sc;
        else if(a.g != b.g) return a.g > b.g;
        else return a.s > b.s;
    }
};
```

list容器

list构造函数

- `list<T> lst;` //用模板默认构造函数
- `list(v.begin(),v.end());` //将该区间内的元素拷贝给本身
- `list(n,elem);` //构造函数将n个elem拷贝给本身
- `list(const list &lst);` //拷贝构造函数

list赋值与交换

- `list& operator = (const list &lst);` //重载等号操作符
- `assign(beg,end);` //将[beg,end)区间中的数据拷贝赋值给本身
- `assign(n,elem);` //将n个elem拷贝赋值给本身
- `swap(lst);` //将lst与本身元素互换

list容量和大小

- `empty();` //判断容器是否为空
- `size();` //返回容器元素的个数
- `resize(int num);` //重新指定容器的长度为num，若容器变长，则以默认值0填充新位置，否则删除超出容器长度的部分
- `resize(int num, elem);` //重新指定容器的长度为num，若容器变长，则以elem填充新位置，否则删除超出容器长度的部分

list插入和删除

两端插入操作

- `push_back(ele);` //尾部插入元素ele
- `push_front(ele);` //头部插入元素ele
- `pop_back();` //删除最后一个元素
- `pop_front();` //删除第一个元素

指定位置操作

- `insert(const_iterator pos, ele);` //迭代器指向位置pos插入元素ele
- `insert(const_iterator pos,int count, ele);` // //迭代器指向位置pos插入count个元素ele
- `erase(const_iterator pos);` //删除迭代器指向的元素
- `erase(const_iterator start,const_iterator end);` //删除迭代器从start到end之间的元素
- `clear();` //删除容器中所有元素

list数据存取

- `front();` //返回容器第一个数据元素
- `back();` //返回容器最后一个数据集元素

list反转和排序

- `reverse();` //反转链表
- `sort();` //链表排序（是成员算法，不是标准算法）

set容器和multiset容器

set构造函数

- `set<T> st;` //用模板默认构造函数
- `set(const set &st);` //拷贝构造函数

set赋值

- `set& operator = (const set &st);` //重载等号操作符

set大小和交换

- `empty();` //判断容器是否为空
- `size();` //返回容器元素的个数
- `swap(st);` //将st与本身元素互换

set插入和删除

- `insert(ele);` //插入元素ele
- `erase(ele);` //删除元素ele
- `erase(const_iterator pos);` //删除迭代器指向的元素，返回下一个元素的迭代器
- `erase(const_iterator start,const_iterator end);` //删除迭代器从start到end之间的元素，返回下一个元素的迭代器
- `clear();` //删除容器中所有元素

set查找和统计

- `find(key);` //查找key是否存在，存在返回该元素的迭代器，否则返回set.end();
- `count(key);` //统计key的元素的个数

pair对组创建

两种构造形式

- `pair<type type> p ( value1, value2);`
- `pair<type type> p = make_pair( value1, value2);`

set内置类型指定排序规则

- `set<int> s;` //升序
- `set<int,greater<int> >;` //降序

set自定义类型（结构体）指定排序规则

- 重载括号

```
class cmp{
    public:
        bool operator ()(const struct node &p1 ,const struct node &p2)
        {
            if(p1.time != p2.time)  return p1.time > p2.time;
            else return p1.name > p2.name;
        }
};
for(set<struct node,cmp>::iterator it = s.begin();it != s.end();it++)
    {
        cout << (*it).time << ' ' << (*it).name << endl;
    }
```

map/multimap/unordered_map容器

map构造函数

- `map<T1,T2> mp;` //用模板默认构造函数
- `map(const map &mp);` //拷贝构造函数

map赋值

- `map& operator = (const map &mp);` //重载等号操作符

map大小和交换

- `empty();` //判断容器是否为空
- `size();` //返回容器元素的个数
- `swap(mp);` //将st与本身元素互换

map插入和删除

- `insert(ele);` //插入元素ele
- `erase(key);` //删除值为key的元素
- `erase(const_iterator pos);` //删除迭代器指向的元素，返回下一个元素的迭代器
- `erase(const_iterator start,const_iterator end);` //删除迭代器从start到end之间的元素，返回下一个元素的迭代器
- `clear();` //删除容器中所有元素

map查找和统计

- `find(key);` //查找key是否存在，**存在返回该元素的迭代器**，否则返回map.end();
- `count(key);` //统计key的元素的个数

map内置类型指定排序规则

- `map<int,int> m;` //升序
- `map<int,int,greater<int> >;` //降序

map自定义类型（结构体）指定排序规则

- 重载括号

```
class cmp{
    public:
        bool operator ()(const struct node &p1 ,const struct node &p2)
        {
            if(p1.time != p2.time)  return p1.time > p2.time;
            else return p1.name > p2.name;
        }
};
for(map<int,struct node,cmp>::iterator it = m.begin();it != m.end();it++)
    {
        cout << (*it).time << ' ' << (*it).name << endl;
    }
```

函数对象

1. 函数对象在使用的时候，可以像普通函数那样调用，可以有参数，可以有返回值。
2. 函数对象超出了普通函数的概念，函数对象可以有自己的状态。
3. 函数对象可以作为参数进行传递。

谓词

- 返回值类型是bool数据类型的仿函数，称为谓词

一元谓词

```
class cmp{
    public:
        bool operator()(int val)
        {
            ...;
        }
};
```

二元谓词

```
class cmp{
    public:
        bool operator()(int val1, int val2)
        {
            ...;
        }
};
```

内建函数对象

算术仿函数

> 实现四则运算

> 其中negate是一元运算，其余都是二元

- `template<class T> T plus<T>;` //加法仿函数
- `template<class T> T minus<T>;` //减法仿函数
- `template<class T> T multiplies<T>;` //乘法仿函数
- `template<class T> T divides<T>;` //除法仿函数
- `template<class T> T moduls<T>;` //取模仿函数
- `template<class T> T negate<T>;` //取反仿函数

关系仿函数

- `template<class T> bool equal_to<T>;` //等于
- `template<class T> bool not_equal_to<T>;` //不等于
- `template<class T> bool greater<T>;` //大于

- `template<class T> bool greater_equal<T>;` //大于等于
- `template<class T> bool less<T>;` //小于
- `template<class T> bool less_equal<T>;` //小于等于

逻辑仿函数

- `template<class T> bool logical_and<T>;` //逻辑与
- `template<class T> bool logical_or<T>;` //逻辑或
- `template<class T> bool logical_not<T>;` //逻辑非

运算符重载

加减乘除运算符重载（全局或者成员函数）

```
struct node{
    int a,b;
    node(int _a = 0, int _b = 0)
    {
        a = _a;
        b = _b;
    }
    node operator+(const node &o)
    {
        node tmp;
        tmp.a = this->a + o.a;
        tmp.b = this->b + o.b;
        return tmp;
    }
};
node operator-(const node &n1, const node &n2)
{
    node tmp;
    tmp.a = n1.a - n2.a;
    tmp.b = n1.b - n2.b;
    return tmp;
}
```

左移运算符重载（全局）

```
ostream& operator<<(ostream &cout, node &o)
{
    cout<< o.a << o.b ;
    return cout;
}
```

递增运算符重载（成员函数）

```
struct node
{
    int a, b;
    node(int _a = 0, int _b = 0)
    {
        a = _a;
```

```
        b = _b;
    }
    node &operator++()  //重置前置++运算符，返回引用时为了一只对一个数据进行递增操作
    {
        this->a++;
        this->b++;
        return *this;
    }
    node operator++(int)//int代表占位参数，可以用于区分前置和后置递增
    {
        node tmp = *this;
        this->a++;
        this->b++;
        return tmp; //由于返回的是一个临时变量，所以不可以返回其引用
    }
};
```

关系运算符重载（成员函数）

```
struct node
{
    int a, b;
    node(int _a = 0, int _b = 0)
    {
        a = _a;
        b = _b;
    }
    bool operator==(node &o)
    {
        if(this->a == o.a && this->b == o.b)
            return true;
        return false;
    }

};
```