list

- list允许在序列中的任何位置进行插入和删除，并且在前后两个方向进行迭代
- list的底层是双向链表结构，双向链表中每个元素存储在互不相关的独立节点中，在节点中通过指针指向其前一个元素和后一个元素
- list是为补充vector的缺点存在的，
- vector的缺点：
  - 1、头部和中部的插入效率低，O(N)需要挪动数据；
  - 2、插入数据空间不够需要增容，增容时需要开辟新空间、拷贝数据和释放旧空间，会付出很大的代价；
- vector的优点：
  - 1、支持下标的随机访问，简介的就很好的支持排序、二分查找和堆算法；
- list的优点：
  - list头部、中间插入不再需要挪动数据，O(1)效率高
  - list插入数据是新增节点，不需要增容
- list的缺点：
  - 不支持随机访问
- 所以在实际使用过程中vector和list是相辅相成的两个容器

list的构造

- list()
  - 构造空的list
- list(size_type n,const value_type& val=value_type())
  - 构造的list中包含n个值为val的元素
- list (const list& x)
  - 拷贝构造函数
- list(InputIterator first,InputIterator last)
  - 构造一个包含范围为[first，last]的元素的容器，每个元素由该范围内其对应元素以相同顺序构造。

```
#include<iostream>
#include<list>

using namespace std;

int main()
{
    //构造空的list
    //list<int> l;

    //构造l里面有n个为val的元素
    list<int> l(5, 10);

    //拷贝构造
    list<int> l2(l);
```

```
    //区间构造
    list<int> l3(l2.begin(), l2.end());

    //利用数组为构造区间
    int arr[] = { 1,2,3,4 };
    list<int> l4(arr, arr + sizeof(arr) / sizeof(arr[0]));

    //迭代器访问
    list<int>::iterator it = l4.begin();
    while (it != l4.end())
    {
        cout << *it << " ";
        ++it;
    }
    cout << endl;

    //反向迭代器访问
    list<int>::reverse_iterator rit = l4.rbegin();
    while (rit != l4.rend())
    {
        cout << *rit << " ";
        ++rit;
    }
    cout << endl;

    //范围for
    for (auto e : l4)
    {
        cout << e << " ";
    }
    cout << endl;

    return 0;
}
```

list的iterator使用

- iterator begin()+iterator end()
  - begin()函数返回一个迭代器，指向list的第一个元素
  - end()函数返回一个迭代器，指向链表的末尾
- reverse_iterator rbegin()+reverse_iterator rend()
  - rbegin()函数返回一个逆向迭代器，指向链表的末尾
  - rend()函数迭代器指向链表的头部

```
#include<iostream>
#include<list>

using namespace std;

int main()
{
    //利用数组为构造区间
    int arr[] = { 1,2,3,4 };
    list<int> l4(arr, arr + sizeof(arr) / sizeof(arr[0]));
```

```
    //迭代器访问
    list<int>::iterator it = l4.begin();
    while (it != l4.end())
    {
        cout << *it << " ";
        ++it;
    }
    cout << endl;//1 2 3 4

    //反向迭代器访问
    list<int>::reverse_iterator rit = l4.rbegin();
    while (rit != l4.rend())
    {
        cout << *rit << " ";
        ++rit;
    }
    cout << endl;//4 3 2 1

    return 0;
}
```

list的容量

- size_type size() const
  - 返回list中元素的数量
- bool empty() const
  - 如果链表为空返回真(true)，否则返回假(false)

```
#include<iostream>
#include<list>

using namespace std;

int main()
{
    //利用数组为构造区间
    int arr[] = { 1,2,3,4 };
    list<int> l4(arr, arr + sizeof(arr) / sizeof(arr[0]));

    //empty()
    cout << l4.empty() << endl;//0

    //size()
    cout << l4.size() << endl;//4

    return 0;
}
```

list的元素获取

- reference front()
  - front()函数返回一个引用，指向list的第一个元素
- reference back();

- back()函数返回一个引用，指向list的最后一个元素

```
#include<iostream>
#include<list>

using namespace std;

int main()
{
    //利用数组为构造区间
    int arr[] = { 1,2,3,4 };
    list<int> l4(arr, arr + sizeof(arr) / sizeof(arr[0]));

    //front()
    cout << l4.front() << endl;//1

    //back()
    cout << l4.back() << endl;//4

    return 0;
}
```

list的增删查改

- void push_front(const value_type& val)
  - 将val连接到链表的头部
- void pop_front()
  - 删除链表的第一个元素
- void push_back (const value_type& val)
  - 将val连接到链表的最后
- void pop_back()
  - 删除链表的最后一个元素
- iterator insert (iterator position, const value_type& val)
  - 插入元素val到位置pos，返回一个迭代器指向被插入的元素
- iterator erase(iterater pos)
  - 除以pos指示位置的元素，返回一个迭代器，指向被删除元素的下一个元素
- void swap(list& x)
  - 交换链表x和现链表中的元素
- void clear()
  - 删除list的所有元素

list的实现

```
#pragma once
#include<assert.h>

namespace bin
{
    //链表节点list_node
    template<class T>
    struct list_node
```

```
{
    list_node<T>* _next;
    list_node<T>* _prev;
    T _data;

    //构造函数
    list_node(const T& data=T())
        :_data(data)
        ,_next(nullptr)
        ,_prev(nullptr)
    {

    }
};

//迭代器list_iterator
//Ref引用、Ptr指针
template<class T,class Ref,class Ptr>
struct list_iterator
{
    typedef list_iterator<T, Ref, Ptr> my_iterator;
    typedef list_node<T> Node;
    Node* _node;
    //链表迭代的核心还是节点的指针

    //构造函数
    list_iterator(Node* node=nullptr)
        :_node(node)
    {

    }

    //*
    Ref operator*()
    {
        return _node->_data;
    }

    //->
    Ptr operator->()
    {
        return &_node->_data;
    }

    //++it
    my_iterator operator++()
    {
        _node = _node->_next;
        return *this;
    }

    //it++
    my_iterator operator++(int)
    {
        list_iterator<T, Ref, Ptr> tmp(*this);
        //_node = _node->_next;
        ++(*this);
        return tmp;
```

```
        }

        //--it
        my_iterator operator--()
        {
            _node = _node->_prev;
            return *this;
        }

        //it--
        my_iterator operator--(int)
        {
            list_iterator<T, Ref, Ptr> tmp(*this);
            _node = _node->_prev;
            return tmp;
        }

        //!=
        bool operator!=(const my_iterator& it)
        {
            return _node != it._node;
        }

        //==
        bool operator==(const my_iterator& it)
        {
            return _node == it._node;
        }

    };

    //链表list
    template<class T>
    class list
    {
        typedef list_node<T> Node;
    public:
        typedef list_iterator<T, T&, T*> iterator;
        typedef list_iterator<T, const T&, const T*> const_iterator;

        //begin()
        iterator begin()
        {
            return iterator(_head->_next);
        }

        //end()
        iterator end()
        {
            return iterator(_head);
        }

        const_iterator begin() const
        {
            return const_iterator(_head->_next);
        }

        const_iterator end() const
```

```
        {
            return const_iterator(_head);
        }

        //构造函数，带头双向循环链表
        list()
        {
            _head = new Node;
            _head->_next = _head;
            _head->_prev = _head;
        }

        //拷贝构造函数
        list(const list<T>& lt)
        {
            _head = new Node;
            _head->_next = _head;
            _head->_prev = _head;
            for (auto e : lt)
            {
                push_back(e);
            }
        }

        //operator=赋值运算符重载
        //lt1=lt3
        //list<T>& operator=(const list<T>& lt)
        //{
        //  //自己不可以给自己赋值
        //  if (this != &lt)
        //  {
        //      clear();
        //      for (auto e : lt)
        //      {
        //          push_back(e);
        //      }
        //  }
        //  return *this;
        //}
        list<T>& operator=(const list<T>& lt)
        {
            swap(_head, lt._head);
            return *this;
        }

        //析构函数
        ~list()
        {
            clear();
            delete _head;
            _head = nullptr;
        }

        //clear()删除list中所有元素
        void clear()
        {
            //迭代器指向链表头
            iterator it = begin();
```

```
        //遍历链表
        while (it != end())
        {
            //依次删除迭代器指向的元素
            erase(it++);
        }
    }

    //push_back
    void push_back(const T& data)
    {
        //找到链表的尾结点
        Node* tail = _head->_prev;
        //创建新的节点
        Node* newnode = new Node(data);

        //链表尾的后指针指向新结点
        tail->_next = newnode;
        //新结点的前指针指向链表尾结点
        newnode->_prev = tail;
        //新结点的后指针指向链表的头结点
        newnode->_next = _head;
        //头结点的前指针指向新结点
        _head->_prev = newnode;
    }

    //pop_back() 尾删
    void pop_back()
    {
        erase(--end());
    }

    //push_front() 头插
    void push_front(const T& data)
    {
        insert(begin(),data);
    }

    //pop_front() 头删
    void pop_front()
    {
        erase(begin());
    }

    //insert()插入
    void insert(iterator pos, const T& data)
    {
        Node* cur = pos._node;
        Node* prev = cur->_prev;
        Node* newnode = new Node(data);

        //prev newnode cur

        prev->_next = newnode;
        newnode->_prev = prev;
        newnode->_next = cur;
        cur->_prev = newnode;
```

```
        }

        //erase() 删除
        iterator erase(iterator pos)
        {
            assert(pos != end());
            Node* del = pos._node;
            Node* prev = del->_prev;
            Node* next = del->_next;

            prev->_next = next;
            next->_prev = prev;
            delete del;

            return iterator(next);
        }
    private:
        Node* _head;
    };
}
```