Lesson10---list

【本节目标】

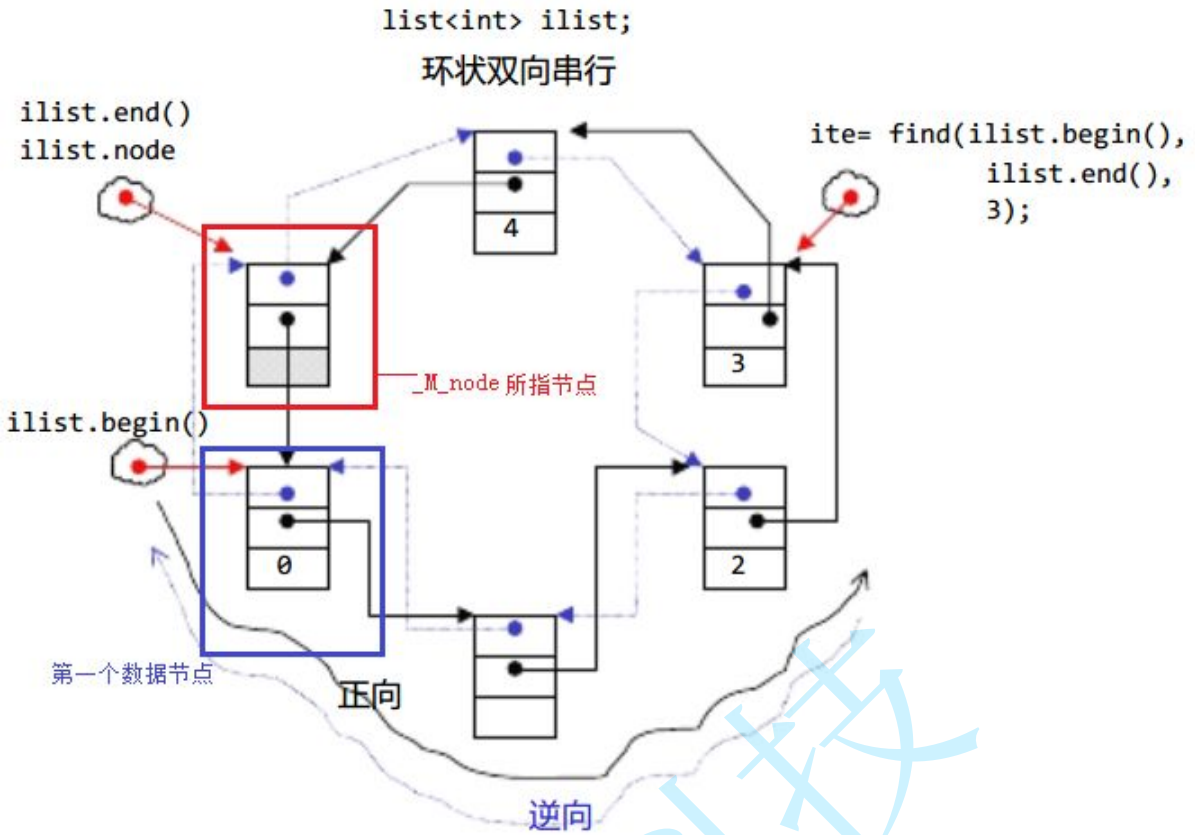
- **1. list的介绍及使用**
- **2. list的深度剖析及模拟实现**
- **3. list与vector的对比**
- **4. 本节作业**

---

1. list的介绍及使用

1.1 list的介绍

> list的文档介绍
>
> 1. list是可以在常数范围内在任意位置进行插入和删除的序列式容器，并且该容器可以前后双向迭代。
> 2. 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。
> 3. list与forward_list非常相似：最主要的不同在于forward_list是单链表，只能朝前迭代，已让其更简单高效。
> 4. 与其他的序列式容器相比(array，vector，deque)，list通常在任意位置进行插入、移除元素的执行效率更好。
> 5. 与其他序列式容器相比，list和forward_list最大的缺陷是不支持任意位置的随机访问，比如：要访问list的第6个元素，必须从已知的位置(比如头部或者尾部)迭代到该位置，在这段位置上迭代需要线性的时间开销；list还需要一些额外的空间，以保存每个节点的相关联信息(对于存储类型较小元素的大list来说这可能是一个重要的因素)



1.2 list的使用

list中的接口比较多，此处类似，只需要掌握如何正确的使用，然后再去深入研究背后的原理，已达到可扩展的能力。以下为list中一些**常见的重要接口**。

1.2.1 list的构造

构造函数（(constructor)）	接口说明
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)	用[first, last)区间中的元素构造list

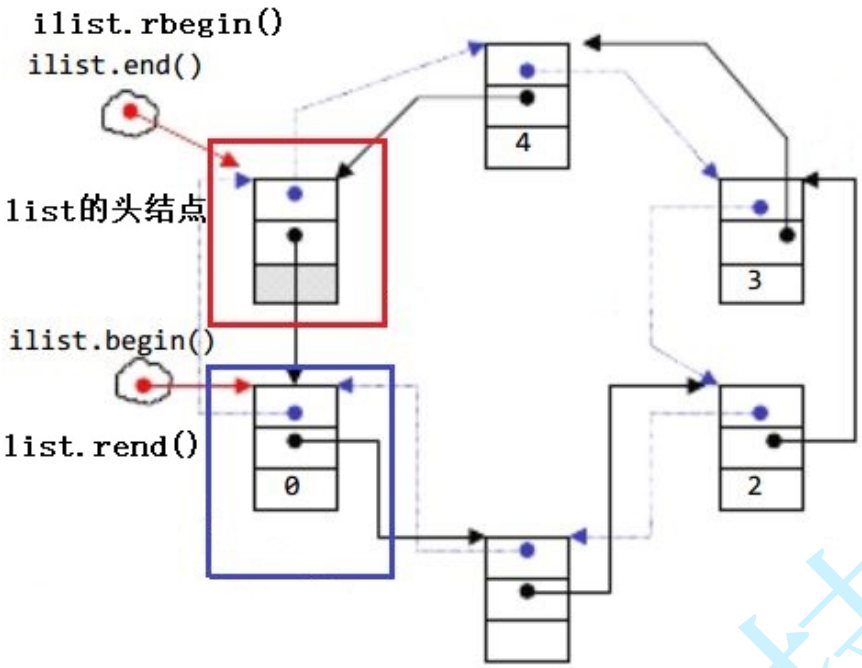
```
// constructing lists
#include <iostream>
#include <list>

int main ()
{
    std::list<int> l1;                         // 构造空的l1
    std::list<int> l2 (4,100);                 // l2中放4个值为100的元素
    std::list<int> l3 (l2.begin(), l2.end());  // 用l2的[begin(), end()) 左闭右开的区间构造l3
    std::list<int> l4 (l3);                    // 用l3拷贝构造l4
```

```
    // 以数组为迭代器区间构造l5
    int array[] = {16,2,77,29};
    std::list<int> l5 (array, array + sizeof(array) / sizeof(int) );

    // 用迭代器方式打印l5中的元素
    for(std::list<int>::iterator it = l5.begin(); it != l5.end(); it++)
        std::cout << *it << " ";
    std::cout<<endl;

    // C++11范围for的方式遍历
    for(auto& e : l5)
        std::cout<< e << " ";

    std::cout<<endl;
    return 0;
}
```

1.2.2 list iterator的使用

此处，大家可暂时**将迭代器理解成一个指针，该指针指向list中的某个节点。**

函数声明	接口说明
begin + end	返回第一个元素的迭代器+返回最后一个元素下一个位置的迭代器
rbegin + rend	返回第一个元素的reverse_iterator,即end位置，返回最后一个元素下一个位置的reverse_iterator,即begin位置



【注意】

1. begin与end为正向迭代器，对迭代器执行++操作，迭代器向后移动
2. rbegin(end)与rend(begin)为反向迭代器，对迭代器执行++操作，迭代器向前移动

```
#include <iostream>
using namespace std;
#include <list>

void print_list(const list<int>& l)
{
    // 注意这里调用的是list的 begin() const，返回list的const_iterator对象
    for (list<int>::const_iterator it = l.begin(); it != l.end(); ++it)
    {
        cout << *it << " ";
        // *it = 10; 编译不通过
    }

    cout << endl;
}

int main()
{
    int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
    list<int> l(array, array + sizeof(array) / sizeof(array[0]));
    // 使用正向迭代器正向list中的元素
    for (list<int>::iterator it = l.begin(); it != l.end(); ++it)

        cout << *it << " ";
```

```
    cout << endl;

    // 使用反向迭代器逆向打印list中的元素
    for (list<int>::reverse_iterator it = l.rbegin(); it != l.rend(); ++it)
        cout << *it << " ";
    cout << endl;

    return 0;
}
```

1.2.3 list capacity

函数声明	接口说明
empty	检测list是否为空，是返回true，否则返回false
size	返回list中有效节点的个数

1.2.4 list element access

函数声明	接口说明
front	返回list的第一个节点中值的引用
back	返回list的最后一个节点中值的引用

1.2.5 list modifiers

函数声明	接口说明
push_front	在list首元素前插入值为val的元素
pop_front	删除list中第一个元素
push_back	在list尾部插入值为val的元素
pop_back	删除list中最后一个元素
insert	在list position 位置中插入值为val的元素
erase	删除list position位置的元素
swap	交换两个list中的元素
clear	清空list中的有效元素

```cpp
#include <list>

void PrintList(list<int>& l)
{
    for (auto& e : l)
        cout << e << " ";
    cout << endl;
}

//==========================================================================================
// push_back/pop_back/push_front/pop_front
void TestList1()
{
    int array[] = { 1, 2, 3 };
    list<int> L(array, array+sizeof(array)/sizeof(array[0]));

    // 在list的尾部插入4，头部插入0
    L.push_back(4);
    L.push_front(0);
    PrintList(L);

    // 删除list尾部节点和头部节点
    L.pop_back();
    L.pop_front();
    PrintList(L);
}

//==========================================================================================
// insert /erase
void TestList3()
{
    int array1[] = { 1, 2, 3 };
    list<int> L(array1, array1+sizeof(array1)/sizeof(array1[0]));

    // 获取链表中第二个节点
    auto pos = ++L.begin();
    cout << *pos << endl;

    // 在pos前插入值为4的元素
    L.insert(pos, 4);
    PrintList(L);

    // 在pos前插入5个值为5的元素
    L.insert(pos, 5, 5);
    PrintList(L);

    // 在pos前插入[v.begin(), v.end)区间中的元素
    vector<int> v{ 7, 8, 9 };
    L.insert(pos, v.begin(), v.end());
    PrintList(L);

    // 删除pos位置上的元素

    L.erase(pos);
```

```
    PrintList(L);

    // 删除list中[begin, end)区间中的元素，即删除list中的所有元素
    L.erase(L.begin(), L.end());
    PrintList(L);
}

// resize/swap/clear
void TestList4()
{
    // 用数组来构造list
    int array1[] = { 1, 2, 3 };
    list<int> l1(array1, array1+sizeof(array1)/sizeof(array1[0]));
    PrintList(l1);

    // 交换l1和l2中的元素
    l1.swap(l2);
    PrintList(l1);
    PrintList(l2);

    // 将l2中的元素清空
    l2.clear();
    cout<<l2.size()<<endl;
}
```

list中还有一些操作，需要用到时大家可参阅list的文档说明。

1.2.6 list的迭代器失效

前面说过，此处大家可将迭代器暂时理解成类似于指针，**迭代器失效即迭代器所指向的节点的无效，即该节点被删除了。**因为**list的底层结构为带头结点的双向循环链表**，因此**在list中进行插入时是不会导致list的迭代器失效的，只有在删除时才会失效，并且失效的只是指向被删除节点的迭代器，其他迭代器不会受到影响。**

```
void TestListIterator1()
{
    int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
    list<int> l(array, array+sizeof(array)/sizeof(array[0]));

    auto it = l.begin();
    while (it != l.end())
    {
        // erase()函数执行后，it所指向的节点已被删除，因此it无效，在下一次使用it时，必须先给其赋值
        l.erase(it);
        ++it;
    }
}

// 改正
void TestListIterator()
{
    int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
    list<int> l(array, array+sizeof(array)/sizeof(array[0]));
```

```
    auto it = l.begin();
    while (it != l.end())
    {
        l.erase(it++);    // it = l.erase(it);
    }
}
```

2. list的模拟实现

2.1 模拟实现list

要模拟实现list，必须要熟悉list的底层结构以及其接口的含义，通过上面的学习，这些内容已基本掌握，现在我们来模拟实现list。

```
namespace bite
{
    // List的节点类
    template<class T>
    struct ListNode
    {
        ListNode(const T& val = T())
            : _pPre(nullptr)
            , _pNext(nullptr)
            , _val(val)
        {}

        ListNode<T>* _pPre;
        ListNode<T>* _pNext;
        T _val;
    };

    /*
    List 的迭代器
    迭代器有两种实现方式，具体应根据容器底层数据结构实现：
      1. 原生态指针，比如：vector
      2. 将原生态指针进行封装，因迭代器使用形式与指针完全相同，因此在自定义的类中必须实现以下方法：
         1. 指针可以解引用，迭代器的类中必须重载operator*()
         2. 指针可以通过->访问其所指空间成员，迭代器类中必须重载oprator->()
         3. 指针可以++向后移动，迭代器类中必须重载operator++()与operator++(int)
            至于operator--()/operator--(int)释放需要重载，根据具体的结构来抉择，双向链表可以向前移动，所以需要重载，如果是forward_list就不需要重载--
         4. 迭代器需要进行是否相等的比较，因此还需要重载operator==()与operator!=()
    */
    template<class T, class Ref, class Ptr>
    class ListIterator
    {
        typedef ListNode<T>* PNode;
        typedef ListIterator<T, Ref, Ptr> Self;
    public:
        ListIterator(PNode pNode = nullptr)
            : _pNode(pNode)
        {}
```

```
        ListIterator(const Self& l)
            : _pNode(l._pNode)
        {}

        T& operator*(){return _pNode->_val;}
        T* operator->(){return &(operator*());}

        Self& operator++()
        {
            _pNode = _pNode->_pNext;
            return *this;
        }

        Self operator++(int)
        {
            Self temp(*this);
            _pNode = _pNode->_pNext;
            return temp;
        }

        Self& operator--();
        Self& operator--(int);

        bool operator!=(const Self& l){return _pNode != l._pNode;}
        bool operator==(const Self& l){return _pNode != l._pNode;}

        PNode _pNode;
    };

    template<class T>
    class list
    {
        typedef ListNode<T> Node;
        typedef Node* PNode;

    public:
        typedef ListIterator<T, T&, T*> iterator;
        typedef ListIterator<T, const T&, const T&> const_iterator;
    public:
        ////////////////////////////////////////////////////////////////
        // List的构造
        list()
        {
            CreateHead();
        }

        list(int n, const T& value = T())
        {
            CreateHead();
            for (int i = 0; i < n; ++i)
                push_back(value);

        }
```

```
template <class Iterator>
list(Iterator first, Iterator last)
{
    CreateHead();
    while (first != last)
    {
        push_back(*first);
        ++first;
    }
}

list(const list<T>& l)
{
    CreateHead();

    // 用l中的元素构造临时的temp,然后与当前对象交换
    list<T> temp(l.cbegin(), l.cend());
    this->swap(temp);
}

list<T>& operator=(const list<T> l)
{
    this->swap(l);
    return *this;
}

~list()
{
    clear();
    delete _pHead;
    _pHead = nullptr;
}

////////////////////////////////////////////////////////////////
// List Iterator
iterator begin(){return iterator(_pHead->_pNext);}
iterator end(){return iterator(_pHead);}
 const_iterator begin(){return const_iterator(_pHead->_pNext);}
const_iterator end(){return const_iterator(_pHead);}
////////////////////////////////////////////////////////////////
// List Capacity
size_t size()const;
bool empty()const;
////////////////////////////////////////////////////////////////
// List Access
T& front();
const T& front()const;
T& back();
const T& back()const;
////////////////////////////////////////////////////////////////
// List Modify

void push_back(const T& val){insert(begin(), val);}
```

```
        void pop_back(){erase(--end());}
        void push_front(const T& val){insert(begin(), val);}
        void pop_front(){erase(begin());}

        // 在pos位置前插入值为val的节点
        iterator insert(iterator pos, const T& val)
        {
            PNode pNewNode = new Node(val);
            PNode pCur = pos._pNode;
            // 先将新节点插入
            pNewNode->_pPre = pCur->_pPre;
            pNewNode->_pNext = pCur;
            pNewNode->_pPre->_pNext = pNewNode;
            pCur->_pPre = pNewNode;
            return iterator(pNewNode);
        }

        // 删除pos位置的节点，返回该节点的下一个位置
        iterator erase(iterator pos)
        {
            // 找到待删除的节点
            PNode pDel = pos._pNode;
            PNode pRet = pDel->_pNext;

            // 将该节点从链表中拆下来并删除
            pDel->_pPre->_pNext = pDel->_pNext;
            pDel->_pNext->_pPre = pDel->_pPre;
            delete pDel;

            return iterator(pRet);
        }

        void clear();
        void swap(List<T>& l);
    private:
        void CreateHead()
        {
            _pHead = new Node;
            _pHead->_pPre = _pHead;
            _pHead->_pNext = _pHead;
        }
    private:
        PNode _pHead;
    };
}
```

2.2 对模拟的bite::list进行测试

```
// 正向打印链表
template<class T>
void PrintList(const bite::list<T>& l)
{
    auto it = l.cbegin();
```

```
    while (it != l.cend())
    {
        cout << *it << " ";
        ++it;
    }

    cout << endl;
}

// 测试List的构造
void TestList1()
{
    bite::list<int> l1;
    bite::list<int> l2(10, 5);
    PrintList(l2);

    int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
    bite::list<int> l3(array, array+sizeof(array)/sizeof(array[0]));
    PrintList(l3);

    bite::list<int> l4(l3);
    PrintList(l4);

    l1 = l4;
    PrintList(l1);
    PrintListReverse(l1);
}

// PushBack()/PopBack()/PushFront()/PopFront()
void TestList2()
{
    // 测试PushBack与PopBack
    bite::list<int> l;
    l.push_back(1);
    l.push_back(2);
    l.push_back(3);
    PrintList(l);

    l.pop_back();
    l.pop_back();
    PrintList(l);

    l.pop_back();
    cout << l.size() << endl;

    // 测试PushFront与PopFront
    l.push_front(1);
    l.push_front(2);
    l.push_front(3);
    PrintList(l);

    l.pop_front();

    l.pop_front();
```

```
    PrintList(l);

    l.pop_front();
    cout << l.size() << endl;
}

void TestList3()
{
    int array[] = { 1, 2, 3, 4, 5 };
    bite::list<int> l(array, array+sizeof(array)/sizeof(array[0]));

    auto pos = l.begin();
    l.insert(l.begin(), 0);
    PrintList(l);

    ++pos;
    l.insert(pos, 2);
    PrintList(l);

    l.erase(l.begin());
    l.erase(pos);
    PrintList(l);

    // pos指向的节点已经被删除，pos迭代器失效
    cout << *pos << endl;

    auto it = l.begin();
    while (it != l.end())
    {
        it = l.erase(it);
    }
    cout << l.size() << endl;
}
```

3. list与vector的对比

vector与list都是STL中非常重要的序列式容器，由于两个容器的底层结构不同，导致其特性以及应用场景不同，其主要不同如下：

	vector	list
底层结构	动态顺序表，一段连续空间	带头结点的双向循环链表
随机访问	支持随机访问，访问某个元素效率O(1)	不支持随机访问，访问某个元素效率O(N)
插入和删除	任意位置插入和删除效率低，需要搬移元素，时间复杂度为O(N)，插入时有可能需要增容，增容：开辟新空间，拷贝元素，释放旧空间，导致效率更低	任意位置插入和删除效率高，不需要搬移元素，时间复杂度为O(1)
空间利用率	底层为连续空间，不容易造成内存碎片，空间利用率高，缓存利用率高	底层节点动态开辟，小节点容易造成内存碎片，空间利用率低，缓存利用率低
迭代器	原生态指针	对原生态指针(节点指针)进行封装
迭代器失效	在插入元素时，要给所有的迭代器重新赋值，因为插入元素有可能会导致重新扩容，致使原来迭代器失效，删除时，当前迭代器需要重新赋值否则会失效	插入元素不会导致迭代器失效，删除元素时，只会导致当前迭代器失效，其他迭代器不受影响
使用场景	需要高效存储，支持随机访问，不关心插入删除效率	大量插入和删除操作，不关心随机访问

4. 作业

本节内容非常重要，对于list的常规接口的应用必须要掌握，弄清楚list底层的实现原理以及vector与list的区别。作业如下：

1. 熟练应用list的常规接口
2. 熟悉list的底层原理并模拟实现
3. 熟悉迭代器的原理
4. 写博客进行总结