天津大學

《高级语言程序设计》实验报告 1

学　　院　理学院

专　　业　应用物理学（强基计划）

年　　级　2020

姓　　名　贺卓诚

2022 年 4 月 9 日

实验八

一、 实验内容

仿照 STL 库里的<list>，编写程序 list.h，完成以下任务:

1.给出模板类 list 的定义,使其可以存储任意类型变量;

2.类需提供构造函数、析构函数;

3.类需定义 iterator 和 const_iterator 类型，并通过 begin()和 end()方法返回相应类型的迭代器;

4.迭代器需要为 Bidirectional 迭代器;

5.类需要以 O(1)时间复杂度实现在头部插入、任意位置插入、尾部插入方法;

6.类需以 O(1)时间复杂度实现删除任意位置节点;

7.类需以 O(1)时间复杂度返回链表长度、判断链表是否为空;

8.完成你认为需要加入的 list 中的方法，可参考 std::list 中的方法;

9.编写驱动函数，测试你编写的 list 类中的每一个方法。

二、 实验任务

1. 这里进行基本架构的搭建。主要包括 Node 类和 List 类的代码。

```
class Node
{
public:
    _N data;
    Node<_N> *next;
    Node<_N> *prev;
    Node(const _N &d, Node<_N> *n = nullptr, Node<_N> *p = nullptr)
    {
        data = d;
        next = n;
        prev = p;
    }
};
```

Node 类的搭建，需要定义 next 和 prev 节点。初始值默认为 nullptr，并用模板类型给一个 data 值。

```
template <class _E>
class hzclist
{
public:
    typedef Node<_E> Node;
    typedef size_t size_type;
```

```
private:
    Node *head;
    size_type _M_len;
```
这里是 list 类的基本结构，它包括一个头结点，还有一个动态维护的长度 _M_len。另外，typedef Node<_E> Node;可以不写，这样会导致该类内的所有函数在调用 Node 时均要写成 Node<_E>，即这样做是提前给定它的类型。

2. 构造函数和析构函数

我给出了三个构造函数

```
    hzclist() : _M_len(0) //构造函数
    {
        head = new Node(0);
        head->next = head;
        head->prev = head;
    }
    hzclist(int n, const _E &value = _E()) //批量构造函数
    {
        head = new Node(0);
        head->next = head;
        head->prev = head;
        while (n--)
        {
            push_back(value);
        }
    }
    hzclist(const hzclist<_E> &l) //拷贝构造函数 需要支持深拷贝，即指针也要跟着走
    {
        head = new Node(0);
        head->next = head;
        head->prev = head;
        hzclist<_E>::const_iterator it = l.cbegin();
        while (it != l.cend())
        {
            this->push_back(*it);
            ++it;
        }
    }
```

这三个构造函数都在 main 函数中进行了测试。

对于析构函数，我先写了一个 clear 函数来清空链表，因为这涉及到指针的深层清理。

```
void Clear()
    {
        Node *p = head->next;
```

```
        while (p != head)
        {
            Node *tmp = p;
            p = p->next;
            delete tmp;
        }
        _M_len = 0;
        head->next = head;
        head->prev = head;
    }
    ~hzclist()
    {
        Clear();
        delete head;
        head = nullptr;
        //全部清空并删除头结点
    }
```

我们在主函数中也可以调用某对象的 clear 函数来主动清空这个 list。

3. 迭代器的实现

```
template <class T, class Ref, class Ptr>
class hzclist_iterator
{
public:
    typedef Node<T> Node; //这一步记得说明一下
    typedef hzclist_iterator<T, T &, T *> iterator;
    typedef hzclist_iterator<T, const T &, const T *> const_iterator;
    typedef hzclist_iterator<T, Ref, Ptr> Self;
    Node *_node;
    hzclist_iterator(Node *n) : _node(n) {}
    hzclist_iterator() : _node(nullptr) {}
};
```

创建了一个迭代器类,在其中定义了 iterator 和 const_iterator,每个 iterator 都指向一个 Node。

```
    typedef hzclist_iterator<_E, _E &, _E *> iterator;
    typedef hzclist_iterator<_E, const _E &, const _E *> const_iterator;
iterator begin() { return iterator(head->next); }
    iterator end() { return iterator(head); }
    const_iterator cbegin() const { return const_iterator(head->next); }
    const_iterator cend() const { return const_iterator(head); }
```

以上为 list 类中迭代器的实现，先定义迭代器类型，然后实现 begin()方法。

4. 双向迭代器的实现

双向迭代器需要重载各个运算符。

```
    Ref operator*() { return _node->data; }
```

```
    Self *operator->() { return &(operator*()); }
    //双向迭代器
    Self &operator++()
    {
        _node = _node->next;
        return *this;
    }
    Self operator++(int)
    {
        Self tmp = *this;
        ++*this;
        return tmp;
    }
    Self &operator--()
    {
        _node = _node->prev;
        return *this;
    }
    Self operator--(int)
    {
        Self tmp = *this;
        --*this;
        return tmp;
    }
    bool operator!=(const Self &rhs) { return _node != rhs._node; } //不等于运算符
    bool operator==(const Self &rhs) { return _node == rhs._node; } //相等运算符
    hzclist_iterator operator=(const Self &rhs)                    //赋值运算符重载
    {
        _node = rhs._node;
        return *this;
    }
```

每个运算符的重载拆开来看都是非常简单的。

5. 插入节点

插入节点全部使用 insert 函数实现。其实只是将一个新节点放进来，并重新连线。这里调用了迭代器。

```
    iterator insert(iterator pos, const _E &x)
    {
        //时间复杂度为O(1)
        Node *cur = pos._node;
        Node *prev = cur->prev;
        Node *newNode = new Node(x);
```

```
        newNode->next = cur;
        cur->prev = newNode;
        prev->next = newNode;
        newNode->prev = prev;
        _M_len++;
        return iterator(newNode);
    }
```

在特殊位置（头，尾）的插入元素，我们直接调用迭代器和 insert 函数实现。

```
void push_front(const _E &x)
    {
        insert(begin(), x);
    }
    void push_back(const _E &x)
    {
        insert(end(), x);
    }
```

6. 删除操作和 insert 操作是类似的。注意这里需要使用 delete 操作符释放内存

```
    iterator erase(iterator pos)
    {
        //时间复杂度为O(1)
        assert(pos != end());
        Node *cur = pos._node;
        Node *prev = cur->prev;
        prev->next = cur->next;
        cur->next->prev = prev;
        delete cur;
        _M_len--;
        return iterator(prev->next);
    }
```

同样的，特殊位置的删除元素我们直接调用 erase 函数

```
    void pop_back()
    {
        erase(--end());
    }
    void pop_front()
    {
        erase(begin());
    }
```

7. 判断列表长度

可以注意到我们在刚才的 insert 函数和 delete 函数中都在维护_M_len 变量。所以，我们需要求长度的时候，直接输出这个长度变量就可以了。这样可以达到 O(1)复杂度。如果不这么做，而是在需要长度的时候不停迭代，就会得到 O(n)复杂度。维护_M_len 变量相当于把复杂度平摊到了平时的每次操作中。

```
    size_t Size()
    {
        return _M_len;
    }
    bool empty()
    {
        return begin() == end();
        // return  _M_len == 0;//这样写也行
    }
```

8. 一些其他的方法：

对 list 重载+运算符

```
template <class _Tp> //两个 list 相加，直接把 list 进行拼接
hzclist<_Tp> &operator+(hzclist<_Tp> &l1, hzclist<_Tp> &l2)
{
    hzclist<int> l3;
    for (auto i = l1.begin(); i != l1.end(); i++)
    {
        l3.push_back(*i);
    }
    for (auto i = l2.begin(); i != l2.end(); i++)
    {
        l3.push_back(*i);
    }
    return l3;
}
```

也就是创建一个新的 list，其他两个链表的所有元素 push 进去。

对 list 重载<<运算符

```
template <class _Tp> //重载流运算符
ostream &operator<<(ostream &out, hzclist<_Tp> &l)
{
    for (auto i = l.begin(); i != l.end(); i++)
    {
        out << *i << " ";
    }
    return out;
}
```

方便调试的时候输出 list 内容用。

取 list 某一位的值而不调用迭代器

```
    //取特殊位置的值，分为常引用和普通引用
    _E &front()
    {
        assert(!empty());
        return head->next->data;
    }
```

```
    const _E &front() const
    {
        assert(!empty());
        return head->next->data;
    }
    _E &back()
    {
        assert(!empty());
        return head->prev->data;
    }
    const _E &back() const
    {
        assert(!empty());
        return head->prev->data;
    }
```

9. 驱动函数测试

主函数比较简单易懂，就不往这里复制了。