我们的普通数组在使用的时候可以定义一个指针指向一个节点，然后使指针 ++ 就可以访问下一个节点，我们想要我们的list也能够使用这个功能，于是就出现了迭代器iterator，通过定义一个iterator，我们可以使用++访问下一个节点，也可以使用\*来找到iterator维护的节点的数据，本文就简单模拟实现iterator。

**iterator的使用**

#include<iostream>

using namespace std;

#include<list>

void TestList()

{

list<int> l;

l.push\_back(1);

l.push\_back(2);

l.push\_back(3);

list<int>::iterator it = l.begin();

while (it != l.end())

{

cout << \*it << endl;

it++;

}

}

这个时候输出的结果就是1 2 3，这样使用list访问数据是不是很方便呢，从上面的使用中，我们也可以终于理解迭代器到底是什么东西了，迭代器其实就是一个维护list中各个节点的一个东西，其中的begin()和end()放回的也是一个迭代器，那么迭代器到底是一个什么东西呢 ，我们呢通过下面的下面的模拟实现可以知道迭代器到底是什么。

**简单模拟实现list和iterator**

#include<iostream>

using namespace std;

#include<list>

template<class T>

struct \_ListNode

{

\_ListNode<T>\* \_next;

\_ListNode<T>\* \_prev;

T \_data;

\_ListNode(T d = 0)

:\_data(d)

, \_next(NULL)

, \_prev(NULL)

{}

};

//这个T标记的是维护的数据类型,Ref是引用，这个引用是对数据的引用，就是这里面data的引用

//Ptr是指针，这个指针也是指向数据data的指针这两个模板类型我们可以在

//List这个类里面使用，因为在编写List\_Iterator的时候，会经常使用Ref和Ptr

template<class T, class Ref, class Ptr>

class List\_Iterator

{

typedef \_ListNode<T> Node;

typedef List\_Iterator<T, Ref, Ptr> Self; //这个地方写的 有点失误了，就是把List\_Iterator的单词写错了

public:

List\_Iterator(Node\* it)

:\_it(it)

{}

List\_Iterator(const List\_Iterator& it)

{

\_it = it.\_it;

}

//List\_Iterator& operator++()

Self& operator++()

{

\_it = \_it->\_next;

return \*this;

}

Self operator++(int) //注意前置++返回的是引用，而后置++返回的是Self

{

Node\* tmp = \_it;

\_it = \_it->\_next;

return tmp;

}

//T& operator\*()

Ref operator\*()

{

return \_it->\_data;

}

bool operator!=(const Self& it)

{

return \_it != it.\_it; //一开始写这个！=运算符重载的时候也出现了错误，就是我一开始是拿this和&it比

//这显然是不对的，两个迭代器不可能是相等的，所以这个时候应该是拿

//这两个迭代器所指向的内容来比较，就是那他们的成员变量来 进行比较才正确

}

private:

Node\* \_it;

};

//这里我们需要做一个修改就是，我们需要把这个链表定义成一个双向的有头的循环链表

//这个工作就是要在构造函数的时候去做了

//所以这里\_tail就不需要了

//有头结点，\_head维护了一个头结点，但是这个节点不放置任何的数据，因为这样在后面的增删查改的时候使用的时候会非常的方便

template<class T>

class \_List

{

typedef \_ListNode<T> Node;

public:

typedef List\_Iterator<T, T&, T\*> Iterator;

//const迭代器

typedef List\_Iterator<T, const T&, const T\*> Const\_Iterator;

//这里为什么传递的是const T&和const T\*呢，如果我们这样传递的话，在实现iterator的时候，里面有个函数是

//Ref operator\*()

//{

// return \_it->\_data;

//}

//这个时候因为我们传递的是const的，所以这个时候的Ref就成了const的了，这个时候的iterator所维护的节点

//里面的数据就不可改变了，这就实现了const的作用了

//这也是我们为什么设计iterator的时候为什么设计的是三个模板参数了

//配合着const\_iterator我们还需要模拟实现返回时const的end和begin

\_List()

:\_head(new Node) //这里报了一个错误就是，没有合适的构造函数可以使用，原因是我上面写\_ListNode的时候

//传参了，但是这里没有传参数，所以我上面的额时候把传递的参数默认为d = 0这个时候就算是我不传递参数

//也没有什么问题了

// , \_tail(\_head)

{

\_head->\_next = \_head;

\_head->\_prev = \_head;

}

//这里开始设计的时候想着返回值是Node\*，然后外面使用的时候，可以拿这个Node\*构造一个迭代器，调用迭代器

//的构造函数，就生成了一个迭代器了，但是问题在于，这样做的话，是不是会引起我们的某些时候的一些误解呢

//我们可以直接返回的就是一个迭代器

//但是如果我们这里设计的返回值是一个迭代器的话，我们需要在这个函数里面定义一个迭代器，然后返回的时候

//生成一个临时的对象，返回给外层的迭代器，这个时候调用了一次拷贝构造函数，返回给外层的迭代器的时候

//又调用了一次拷贝构造函数，这样会不会太复杂了呢

//Iterator begin()

//{

// if (\_head->\_next == \_head) //这里做了一个判断就是当我们的链表是空的时候，就不能直接返回next

// {

// return Iterator(NULL);

// }

// return Iterator(\_head->\_next);

//}

//在使用Const\_Iterator的时候出现了问题就是，为什么我下面的普通的begin函数不注释 的时候，就编译不过呢

Const\_Iterator begin() const //返回值是Const\_iterator为了和上面的匹配，然后后面 的一个const为为了

//this指针所指向的内容

{

if (\_head->\_next == \_head) //这里做了一个判断就是当我们的链表是空的时候，就不能直接返回next

{

return Const\_Iterator(NULL);

}

return Const\_Iterator(\_head->\_next);

}

//Node\* begin() const

//{

// if (\_head->\_next == \_head) //这里做了一个判断就是当我们的链表是空的时候，就不能直接返回next

// {

// return NULL;

// }

// return \_head->\_next;

//}

/\*Iterator end()

{

if (\_head->\_next == \_head)

{

return Iterator(NULL);

}

return Iterator(\_head);

}\*/

Const\_Iterator end() const

{

if (\_head->\_next == \_head)

{

return Const\_Iterator(NULL);

}

return Const\_Iterator(\_head);

}

void Pushback(const T& data)

{

Node\* cur = new Node(data);

cur->\_prev = \_head->\_prev;

\_head->\_prev = cur;

cur->\_next = \_head;

cur->\_prev->\_next = cur;

}

//void Print()

//{

// Const\_Iterator it = begin(); //这个原因是，我们的begin不是 const的，但是我们的const\_ietator是const的

// while (it != end())

// {

// cout << \*it << endl;

// it++;

// }

//}

private:

Node\* \_head;

// Node\* \_tail;

};

void TestList()

{

\_List<int> l;

l.Pushback(0); //一开始的时候，我这里设置的这个参数是引用，这个时候是辩不过的，因为我们的0是在常量区的

l.Pushback(1);

l.Pushback(2);

list<int>::const\_iterator it = l.begin();\*

}

这里我们按照一个类一个类的来进行分析

**\_ListNode**

首先看到的是\_ListNode，这里就不用过多的解释什么了，成员变量是两个指向该节点的指针，和一个存放数据的东西；这里只实现了一个构造函数，构造函数完成的初始化的时候把两个指针置为NULL

**List\_Iterator**

**构造函数**，我们的List\_Iterator底层维护的实际上就是一个指针，指向的就是我们的链表的一个个节点，构造函数中是把一个指针给了迭代器中，这个 函数一般在我们的list中的begin函数中使用到

**拷贝构造函数**。这个是拿一个已经存在的迭代器去初始化 另一个迭代器。

前置 ++ 和后置 ++ 。Self& operator++()和Self operator++(int) 这两个函数是我们实现的重点，因为我们 开始写迭代器 的目的就是希望能够像一个指针使用它，直接使用 ++ 操作就可以实现对下一个节点的访问了。这里我们 应该注意的一个问题就是，我们的前置 ++ 和后置 ++ 的返回值应该是不一样的，因为前置++返回的是先 ++ 后返回，返回的是 ++ 后的结果，所以这个时候应该返回的是Self的引用，但是后置 ++ 是先返回在 ++ ，这个时候是不能够使用引用的，因为我们既然要先返回再 ++ ，我们就要在 函数的里面先将 ++ 前的值记录下来，所以这个时候就需要定义一个变量保存 ++ 前的结果，然后返回，如果这个时候使用的是引用的话，我们的函数结束之后，刚刚的变量就释放掉了，所以这个时候的引用是错误的。

**\*号运算符的重载**。这个重载返回的应该是迭代器 所指向的数据，所以这个时候返回的是T&，然后我们又在上面对这个T&进行了一个封装，所以这个时候返回值直接使用Ref。

**\_List**

关于链表，我们这里维护的是一个双向的循环的有头节点的双向链表

typedef List\_Iterator<T, T&, T\*> Iterator;

//const迭代器

typedef List\_Iterator<T, const T&, const T\*> Const\_Iterator;

这里重点说明一下，为什么要将迭代器声明为三个模板参数的，这里就可以很好的体现了复用性了

这里为什么传递的是const T&和const T呢，如果我们这样传递的话，在实现iterator的时候，里面有个函数是 Ref operator() { return \_it->\_data; } 这个时候因为我们传递的是const的，所以这个时候的Ref就成了const的了，这个时候的iterator所维护的节点 里面的数据就不可改变了，这就实现了const的作用了 这也是我们为什么设计iterator的时候为什么设计的是三个模板参数了 配合着const\_iterator我们还需要模拟实现返回时const的end和begin