类结构设计

链表类的基础组成部分为链表节点以及链表，考虑到为了便于链表对节点的直接操作且用户不会直接使用链表节点，链表节点所有成员使用public修饰符，同时为了便于用户使用，采用迭代器设计对链表节点进行封装。

故链表类的组成为链表节点MyDataStruct::ListNode，链表迭代器MyDataStruct::ListIterator，const修饰的链表迭代器MyDataStruct::ConstListIterator以及链表本身MyDataStruct::List。这些类模板包含在命名空间MyDataStruct，定义于generic\_component目录下的List.h头文件中。

链表节点类（MyDataStruct::ListNode）



链表节点类模板接受模板参数T作为存储数据的类型。

节点类包含三个成员，指向直接前驱和指向直接后继的指针，同时为了避免出现循环包含的问题，使用指针来进行数据存储。

同时节点类包含了三个构造函数，默认构造函数：将所有数据都定义为空指针、接受存储数据为参数的构造函数，为data\_指针开辟一块内存空间存放数据（释放内存的工作交与List类进行）、拷贝构造函数（拷贝三个指针成员的值进行初始化）。

链表迭代器类（MyDataStruct::ListIterator）



链表迭代器类模板支持类模板参数T作为存储数据的类型，为兼容标准库，链表迭代器继承自std::iterator<std::bidirectional\_iterator\_tag, T>这样一个不储存任何数据的类来获得双向迭代器等标签。

为了提高可读性，将pointer作为T\*的类型别名，将reference作为T&的类型别名。

为了便于链表类操作迭代器类将链表类作为友元类。

链表迭代器封装了一个指向链表节点的指针node\_作为protected成员。

迭代器支持的重载运算符操作：

ListIterator operator++(int);

后置++，储存原有迭代器的值，node\_指针指向其直接后继，返回原有迭代器。

ListIterator& operator++();

前置++，node\_指针指向其直接后继，返回自身的引用。

ListIterator operator--(int);

后置--，储存原有迭代器的值，node\_指针指向其直接前驱，返回原有迭代器。

ListIterator& operator--();

前置--，node\_指针指向其直接前驱，返回自身的引用。

bool operator==(const ListIterator<T>& iter) const;

重载==运算符，若两迭代器储存指针的值相同，返回true，反之返回false。

bool operator!=(const ListIterator<T>& iter) const;

重载!=运算符，若两迭代器储存指针的值不相同，返回true，反之返回false。

ListIterator<T>& operator=(const ListIterator<T>& iter);

重载=运算符，将迭代器参数存储指针的值赋予自身存储指针。

pointer operator->() const;

重载->运算符，返回node\_指向的节点内存储数据的指针；

reference operator\*() const;

重载\*运算符，返回node\_指内的节点内存储数据的引用；

迭代器支持的构造函数：

ListIterator() : node\_(nullptr)

{}

默认构造函数，node\_指针初始化为空指针。

ListIterator(const ListIterator<T>& iter) : node\_(iter.node\_)

{}

拷贝构造函数，将参数iter存储指针的值赋予自身存储的指针。

ListIterator(ListNode<T>\* const node) : node\_(node)

{}

接受一个指向链表节点的指针node作为参数，由链表类调用，将node的值赋予this->node\_。

Const链表迭代器类（MyDataStruct::ConstListIterator）



const链表迭代器继承自MyDataStruct::ListIterator<T>。

将pointer作为const T\*的类型别名，将reference作为const T&的类型别名。

const迭代器支持基本操作以及构造函数大部分与普通迭代器相同，不同的是const迭代器重写了->和\*操作符。

pointer operator->() const;

返回指向const T类型的指针，该指针为node\_指向的节点内部存放数据的指针。

reference operator\*() const;

返回const T&类型的引用，该引用为node\_指向的节点内部所存放数据的引用。

链表类



链表类接受一个类模板参数T作为存储数据类型。

链表类以iterator作为ListIterator<T>的类型别名、以const\_iterator作为ConstListIterator<T>的类型别名，以size\_type作为int的类型别名。

链表类private成员有头指针begin\_，尾后指针end\_以及链表长度len\_，初始化条件下begin\_==end\_。尾后节点内存放指向数据的指针为空指针。

链表类定义了SAFE\_MODE和EFFICIENCY\_MODE两个Mode枚举常量，当执行插入或删除时，根据Mode参数的值判断是否要对插入或删除位置的迭代器进行合法性判断。

链表类支持的操作：

iterator begin();const\_iterator begin() const;

返回指向头节点的迭代器或const迭代器。

iterator end(); const\_iterator end() const;

返回指向尾后节点的迭代器或const迭代器。

iterator insert(iterator position, const T& value, Mode mode = EFFICIENCY\_MODE);

将value的值插到position迭代器前面，默认情况下不对迭代器合法性进行判断，之后返回指向插入元素的迭代器。

插入函数核心代码：

ListNode<T>\* new\_node = new ListNode<T>(value);

if (new\_node == nullptr) {

std::cout << "Memory application is failed" << std::endl;

exit(1);

}

if (position.node\_->prev\_) {

position.node\_->prev\_->next\_ = new\_node;

}

new\_node->prev\_ = position.node\_->prev\_;

new\_node->next\_ = position.node\_;

position.node\_->prev\_ = new\_node;

++len\_;

if (position.node\_ == node\_) {

node\_ = new\_node;

}

return iterator(new\_node);

iterator erase(iterator position, Mode mode = EFFICIENCY\_MODE);

删除position迭代器指向的位置，默认情况下不对迭代器合法性进行判断，之后返回被删除元素之后的迭代器。

删除函数核心代码：

position.node\_->next\_->prev\_ = position.node\_->prev\_;

ListNode<T>\* result = position.node\_->next\_;

--len\_;

if (position.node\_ == node\_) {

node\_ = result;

} else {

position.node\_->prev\_->next\_ = position.node\_->next\_;

}

delete position.node\_;

return iterator(result);

void push\_back(const T& value);

在链表尾部插入值为value的节点。

size\_type size() const;

返回链表长度。

T& front(); const T& front() const;

返回链表头节点存放的数据的值。

T& back(); const T& back() const;

返回链表尾节点存放的数据的值。

void clear()

清空链表内所有节点，并对清除的节点进行内存释放，防止内存泄漏。;

iterator find(const T& value) const;

查找链表内第一个值为value的节点，返回指向该节点的迭代器，如不存在这样的节点，返回尾后迭代器。

bool empty() const;

判断该链表是否为空。

List();

默认构造函数，构造一个空链表。

List(const iterator& start, const iterator& end);

接受两个迭代器作为参数的构造函数，构造一个值的范围为[start,end)的链表。

List(const List<T>& list);

拷贝构造函数，构造一个与参数一样的链表。

List<T>& operator=(const List<T>& list);

重载=运算符，令自身存储数据与参数一致，返回指向自身的引用。

~List();

析构函数，调用clear()函数，释放内存，避免内存泄漏。