|  |
| --- |
| 哈尔滨工业大学(深圳) |
| **《数据结构》实验报告** |
|  |
| 实验一  线性结构及其应用  学 院: 计算机科学与技术   |  |  | | --- | --- | | 姓 名: | 梁鑫嵘 | | 学 号: | 200110619 | | 专 业: | 计算机科学与技术 | | 日 期: | 2021-04-01 | |

# 问题分析

**问题描述**

学期结束，辅导员需要收集并整理两个班级同学的C语言课程成绩。请你为辅导员编写一个成绩录入统计程序，帮助辅导员更好地工作。

程序需求：

1. 存储链表：请按头插法将两个班的成绩存储到链表中。由于给出的输入总是升序的，此链表在存储后成绩将是降序的。
2. 反转链表：请将建立的两个降序存储的链表按升序进行反转。
3. 寻找链表交点：编程找出两个链表的第一个公共交点，你只能拿到交叉链表的两个头结点。

**问题分析(见后文)**

# 二、详细设计

## 2.1 设计思想

1. *储存链表*：

链表的储存是使用在计算机的内存中的一段可以不连续的区域，通过储存数据和指向下一个节点的指针的方式进行储存的一种线性结构。

在创建这个链表的时候使用头插法，即每次把新的数据插入到头指针的前面，能够获得和输入数据顺序相反的链表。

1. *反转链表*：

反转链表有两个思路，一是通过改变指针的指向方向，来原地改变链表的前后顺序。这样的方式效率高，也没有额外的内存消耗，是最优的算法。

但是在本实验中本人使用了C++中的智能指针shared\_ptr，使得节点不能储存自身的指针，而且在操作中有可能出现循环引用问题，造成资源无法自动正确释放，所以本人仍然使用了“头插法”来使链表反向。不过在“头插法”运行时，由于智能指针的特性，资源是一边赋值一边释放的，所以消耗了一些数据传输的时间，但是总内存量仍不变。相对于上面第一个算法只是略有不足，理论算法复杂度和空间复杂度仍然相同。

1. 寻找链表交点

使用了逐个搜索的算法，即使用两个循环依次从头到尾检查两个链表指针是否指向同一内存。暂不知道有更加高效的解决方法。

## 2.2 存储结构及操作

(1) 存储结构

链表的储存，我使用了这样的储存方式（见./linked\_list.hpp）

 // 单向链表，C++智能指针实现  
 template<typename T>  
 class linked\_list {  
 public:  
   std::shared\_ptr<linked\_list<T>> next = nullptr;  // 指向下一个节点的指针  
   T data{};                                        // 当前数据  
 // ......  
 };

设计特点：

1. 递归的结构设计，每一个指针p都可以单独表示一个链表。
2. 使用智能指针，防止内存泄露，方便内存管理。
3. 使用模板，链表的特性和数据本身脱离，具有更好的可移植性。
4. 代码规范，使用现代C++代码规范，方便代码管理和阅读。

**特别说明**

本程序运用到的C++标准库仅有std::shared\_ptr，未使用包括std::string在内的STL标准库。chilib::string为自己实现的string。

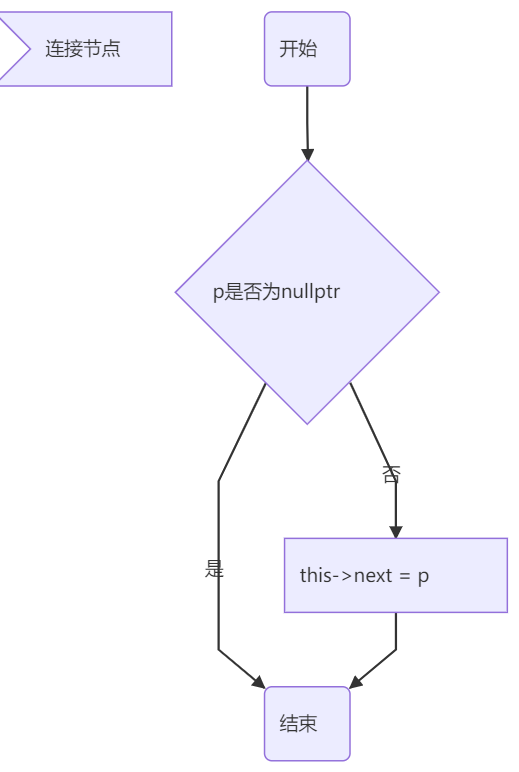
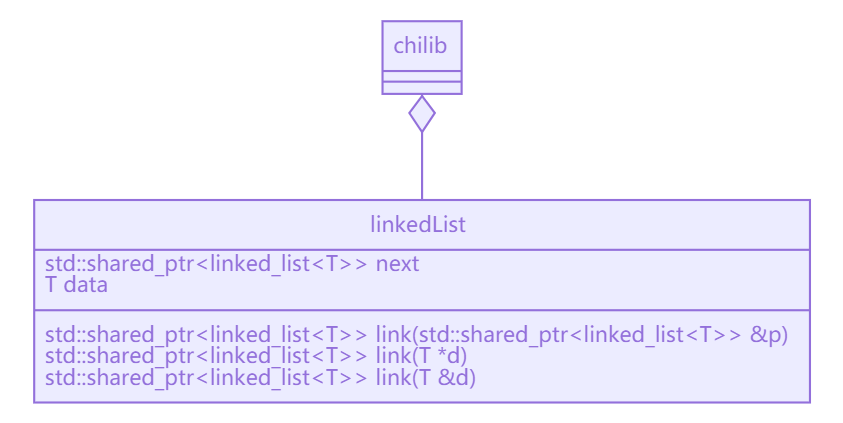
using namespace chilib;  
 using std::ios;  
 // 声明：本程序没有使用 std::string  
 // using std::string;

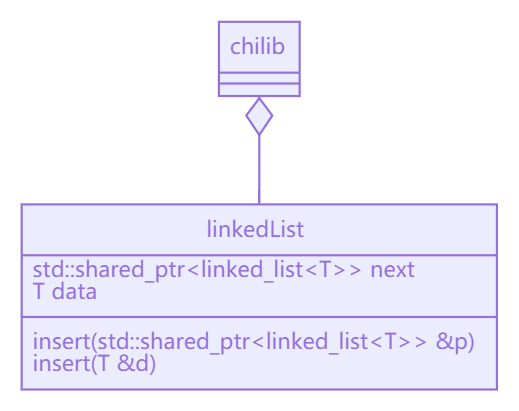
(2) 涉及的操作

**涉及的操作**

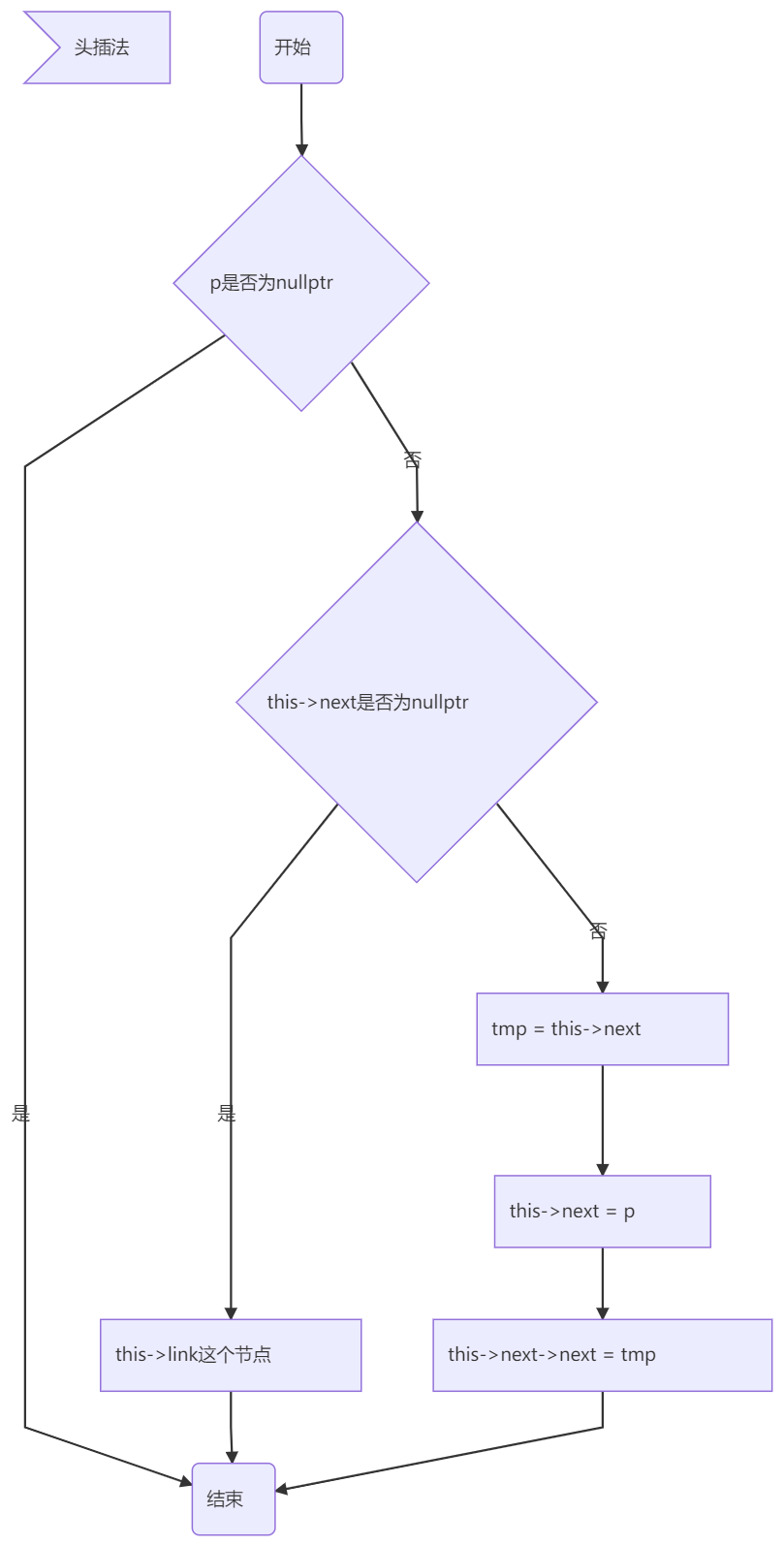
// @prog   : 使用头插法读取文件数据到一个链表  
 // @rets   : 通过引用返回两个链表头指针和交叉节点，直接返回读取文件的offset  
 // @notice : 读取完需要的数据或者EOF就表示文件结束  
 long readFileData(const string &filename, StudentList &class1, StudentList &class2, CrossLink &cross,  
                   size\_t offset = 0);  
 // 创建交叉链表  
 // 相交方法是将b中的前一结点指向a中的首个相交结点  
 // 使用了智能指针使得未被引用的数据自动删除  
 void createCrossLink(StudentList &head1, StudentList &head2, CrossLink &cross);  
 // @prog   : 找到相交的第一个结点  
 // @rets   : 返回找到的交点  
 StudentList findCrossBeginNode(StudentList &head1, StudentList &head2);  
 // @prog   : 输出该表的成绩情况  
 void outputStudentLinkedList(StudentList &head);  
 // @prog   : 翻转链表  
 // @rets   : 通过引用修改链表  
 void reverseLinkedList(StudentList &head);  
 // @prog   : 从对象生成一个节点  
 template<typename T>  
 auto linked\_list\_make(T &d);  
 // @prog   : 检查是否存在环  
 template<typename T>  
 bool linked\_list\_check\_circulation(linked\_list\_p<T> &p);  
 // @prog   : 打印整条链表的数据  
 template<typename T>  
 void linked\_list\_print(linked\_list\_p<T> &head);  
 // ## 下面是 linked\_list 的内部函数  
   // @prog   : 得到链表尾部  
   // @notice : 只有自己一个节点的时候也返回nullptr  
   std::shared\_ptr<linked\_list<T>> get\_tail();  
   // @prog   : 连接一个节点到this节点后面  
   // @notice : 这个传入的节点之后的数据被释放  
   auto link(T &d);  
   // @prog   : 接一个已经存在的链表串到本节点之后  
   // @notice : 本节点之后的数据释放  
   //           参数节点之后的数据保留  
   std::shared\_ptr<linked\_list<T>> link(std::shared\_ptr<linked\_list<T>> &p);  
   // @prog     : 接一个已经存在的链表串到本节点之后  
   // @notice   : 本节点之后的数据接到参数节点之后  
   //             参数节点之后的数据保留到串中  
   std::shared\_ptr<linked\_list<T>> append(std::shared\_ptr<linked\_list<T>> &p);  
   // @prog   : 获取往下几个节点的指针  
   // @notice : 不能返回自己的指针  
   std::shared\_ptr<linked\_list<T>> step(size\_t s);

**连接节点**



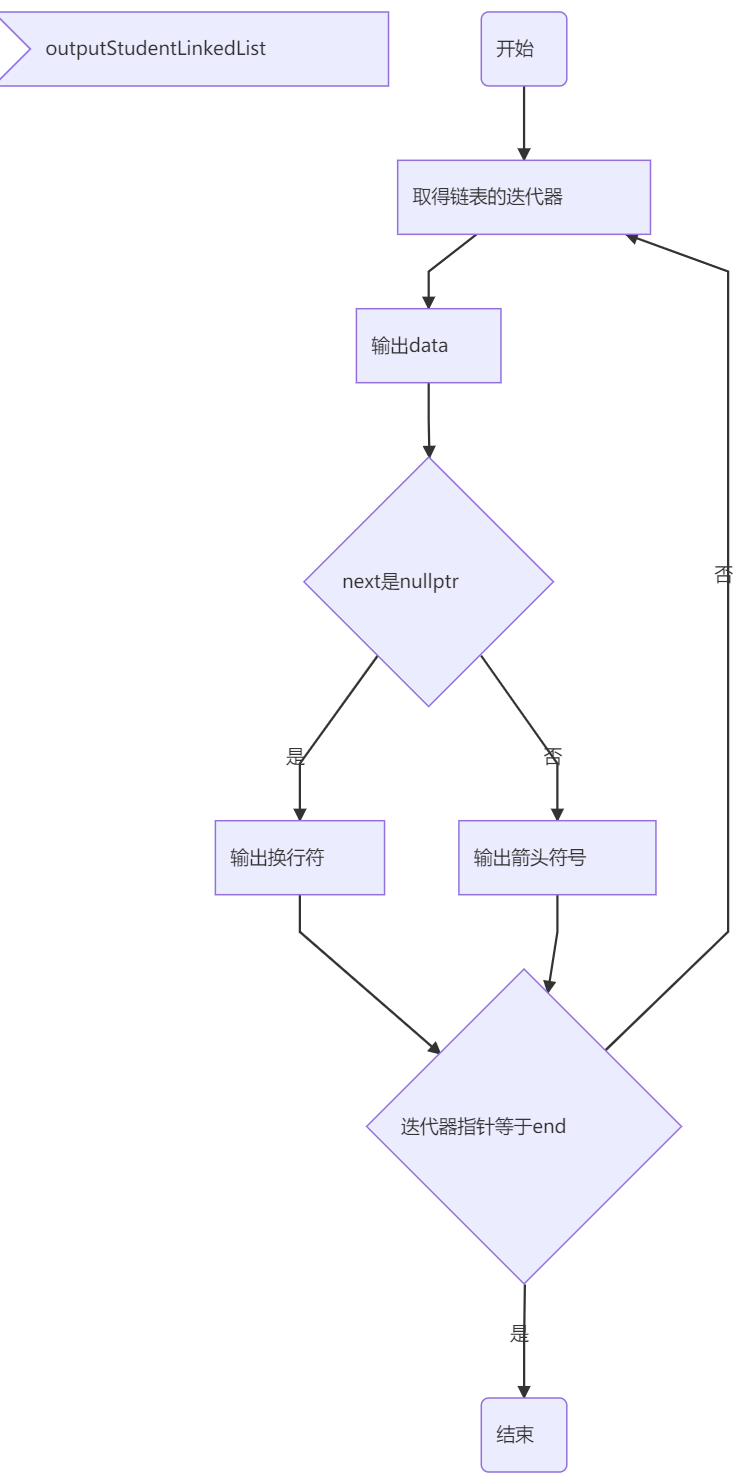


**头插法**

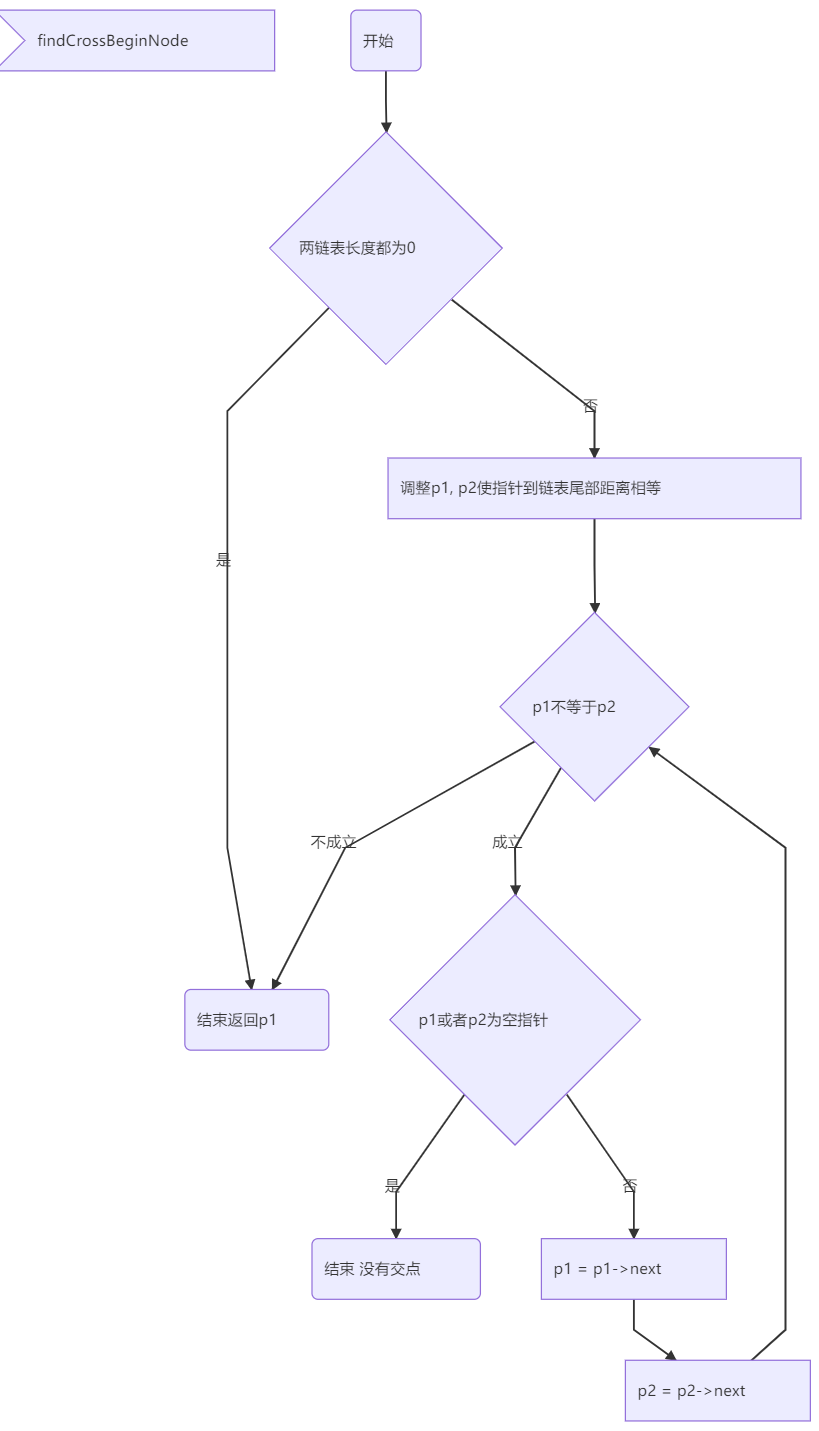


复杂度：O(1)

**遍历链表**



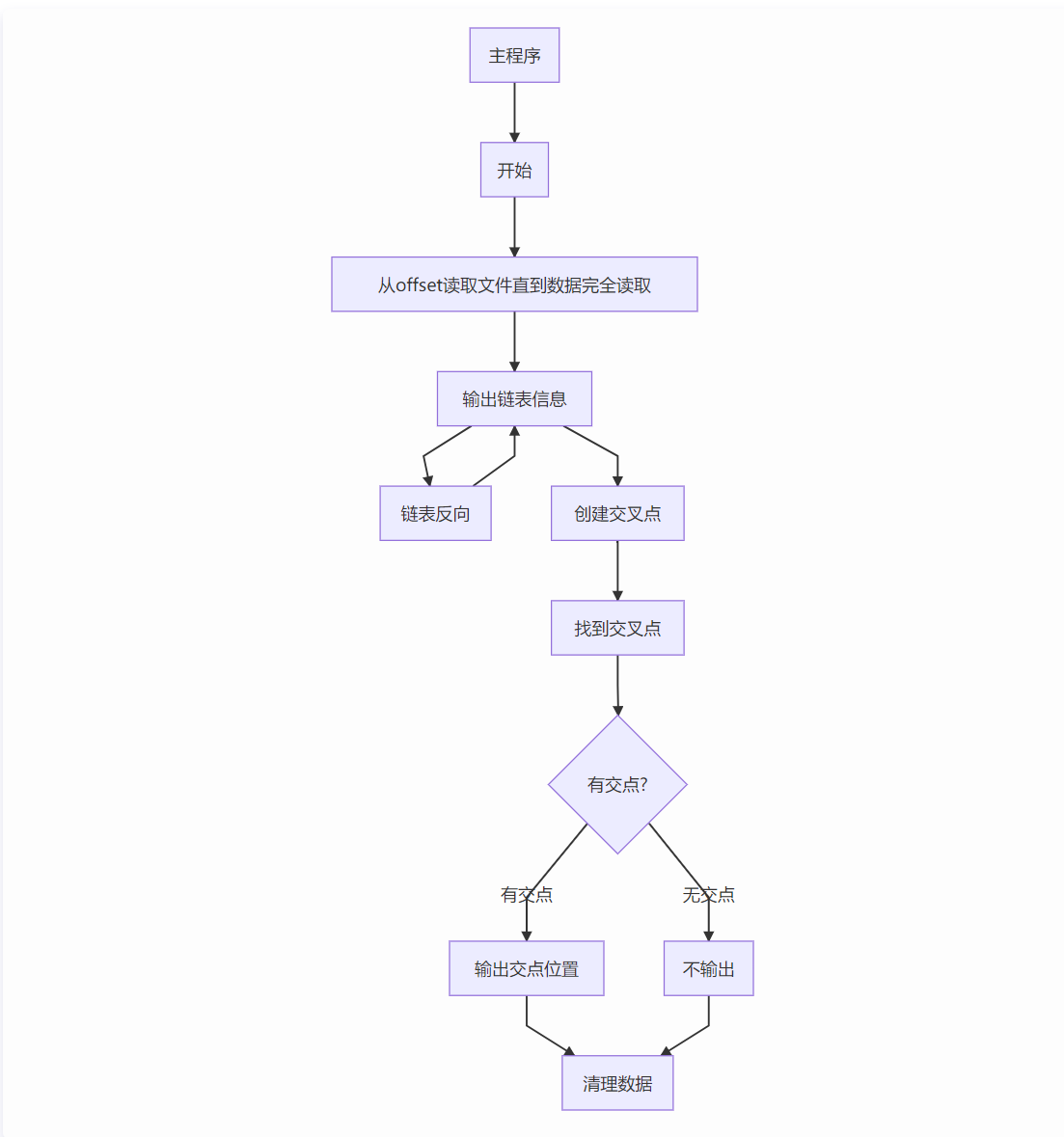
遍历全部的复杂度：O(N)



**找到链表交点**

复杂度：O(N)

## 2.3 程序整体流程



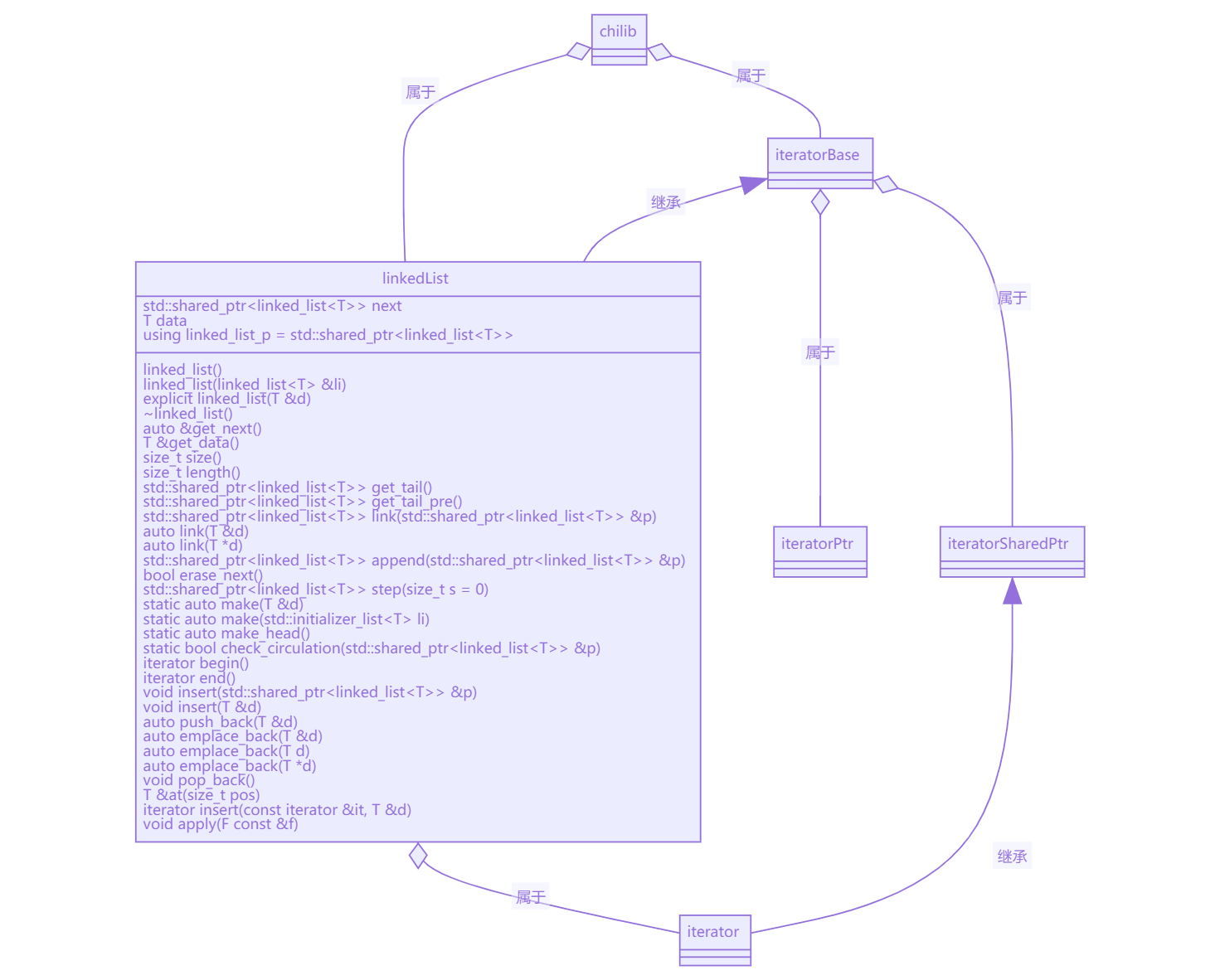
# 三、用户手册

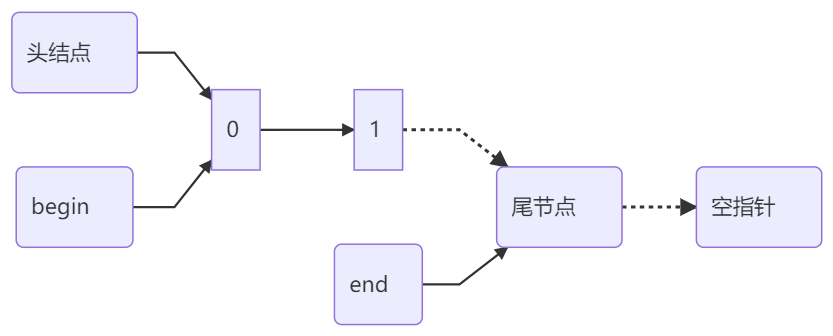
## 使用说明 - chilib::linked\_list

chilib是Chiro([Chiro2001@163.com](mailto:Chiro2001@163.com)) (200110619梁鑫嵘)在数据结构课中，尝试使用C++语言和自己有限的知识实现C++库中部分STL库的功能的一个库。链接：<https://github.com/chiro2001/chilib>。以下是chilib::linked\_list的使用说明。

### 链表结构

本程序的链表数据结构表示如下：





### 获取链表信息

linked\_list::get\_next(); // 获取下一个节点的指针   
linked\_list::get\_data(); // 获取本节点的数据   
linked\_list::size(); // 获取以这个节点为头指针计算的链表长度   
linked\_list::length(); // 同 size()   
linked\_list::get\_tail(); // 得到尾指针   
linked\_list::get\_tail\_pre();// 得到尾指针的前一个指针   
linked\_list::step(s); // 获取往下 s + 1 个节点的指针   
linked\_list::begin(); // 得到首迭代器（如上图）   
linked\_list::end(); // 得到尾迭代器（如上图）

### 建立链表/初始化节点

// 生成链表头指针，头指针不包含数据，仅方便链表管理   
linked\_list\_p<int> head = linked\_list<int>::make\_head();   
// 生成含数据的一个链表节点   
int d0 = 0;   
linked\_list\_p<int> node = linked\_list<int>::make(d0);   
// 直接用数据生成一串链表   
auto head2 = linked\_list<int>::make({9, 7, 5});

### 添加数据

// 方式1：把节点链接到节点之后   
head->link(node); // 此时数据为 head(空) --> node(0)   
// 方式2：直接链接一个数据到这个节点之后   
d0 = 1;   
head->link(d0); // 此时数据为 head(空) --> (1)，原来的node(0)被挤掉   
// 方式3：在节点之后插入一个数据或者节点，原来节点之后的数据还在   
d0 = 3;   
head->insert(d0); // 此时数据为 head(空) --> (3) --> (1)   
head->insert(node); // 此时数据为 head(空) --> node(0) --> (3) --> (1)   
// 方式4：在当前节点之后添加一串链表，原来节点之后的数据还在   
node->append(head2->get\_next());   
head2 = nullptr;   
// 此时数据为 head(空) --> node(0) --> [(9) --> (7) --> (5)] --> (3) --> (1)   
// 方式5：向尾部添加数据   
head->emplace\_back(10);   
// 此时数据为 head(空) --> node(0) --> [(9) --> (7) --> (5)] --> (3) --> (1) --> (10)

### 遍历数据

// 方式1：利用指针遍历   
auto p0 = head->get\_next();   
while (p0 != nullptr) {   
 // Do something   
 p0 = p0->step();   
}   
// 方式2：利用迭代器   
for (auto p : head) {   
 // Do something   
}

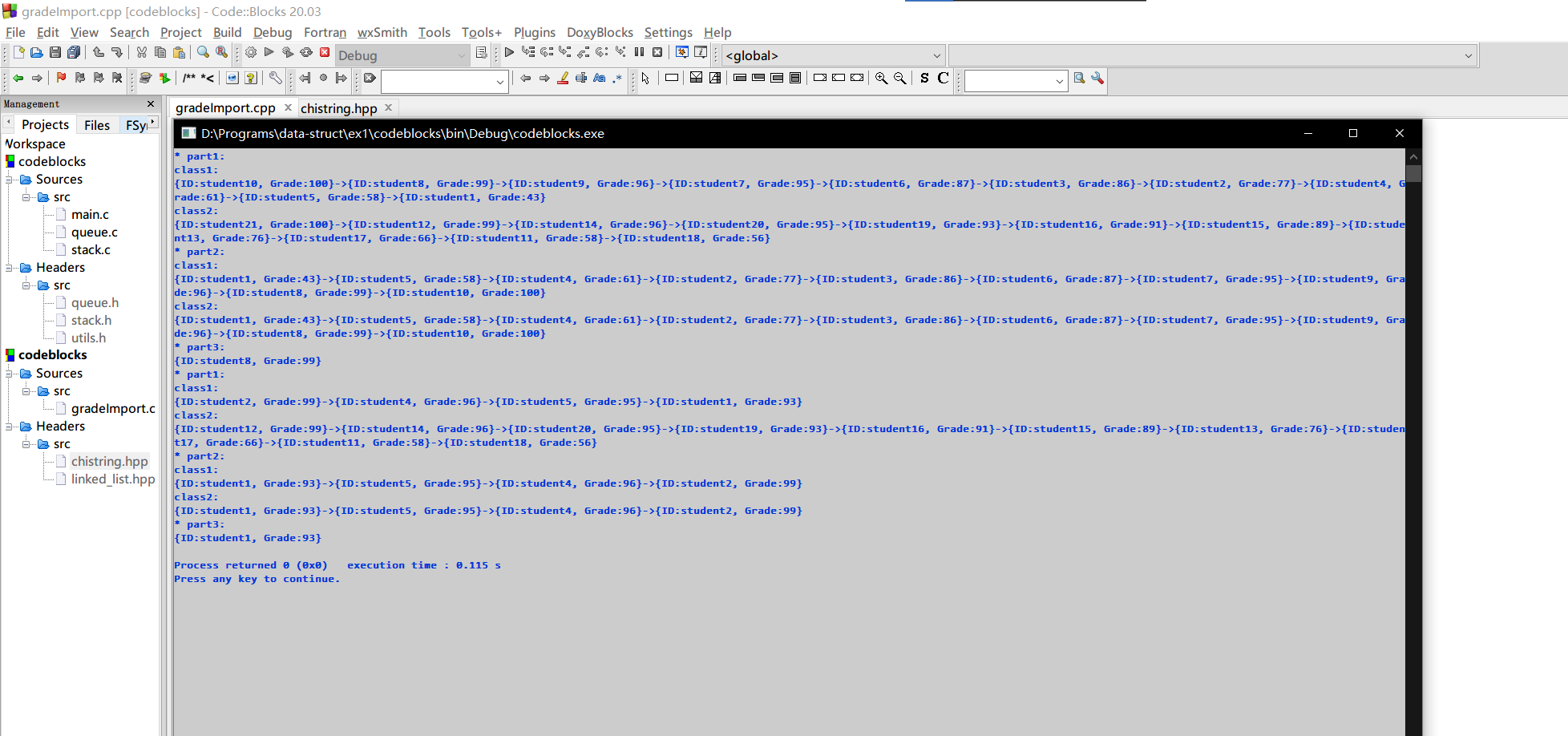
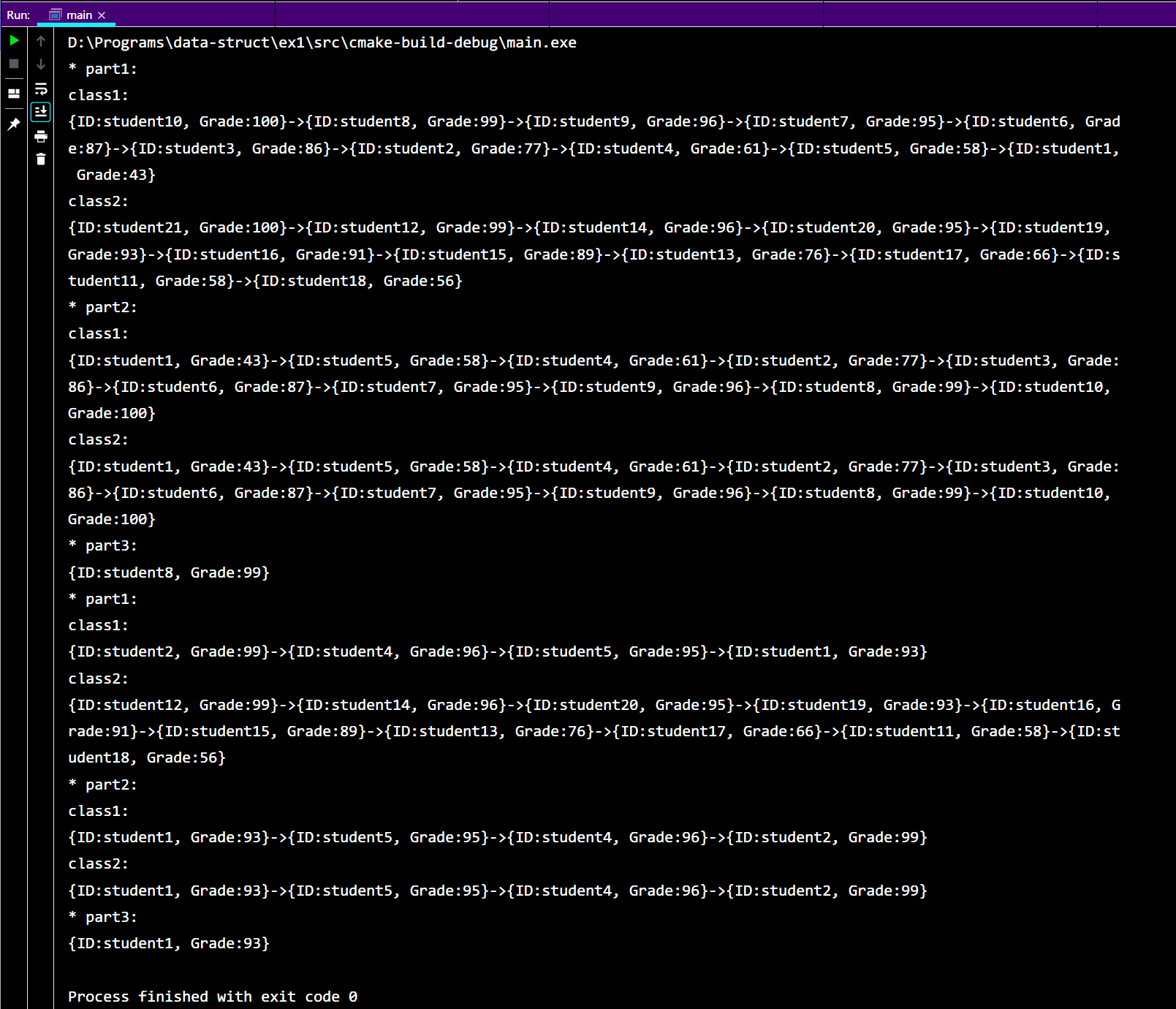
### 修改数据

// 此时数据为 head(空) --> node(0) --> [(9) --> (7) --> (5)] --> (3) --> (1) --> (10)   
// 你可以遍历修改数据   
for (auto &p : head) {   
 p.get\_data() -= 3;   
}   
// 也可以用 lambda 函数执行   
head->apply([](auto &p){   
 p.get\_data() += 6;   
})   
// 此时数据为 head(空) --> [3, 12, 10, 8, 6, 4, 13]   
// 可以修改某一个位置的数据   
head->at(3) = -1;   
head[2] = -3;   
// 此时数据为 head(空) --> [3, 12, -3, -1, 6, 4, 13]   
// 可以删除数据   
head->pop\_back();   
// 此时数据为 head(空) --> [3(node), 12, -3, -1, 6, 4]   
node->erase\_next();   
// 此时数据为 head(空) --> [3(node), -3, -1, 6, 4]

### 删除链表

// 此时数据为 head(空) --> [3(node), -3, -1, 6, 4]   
node = nullptr;   
// 此时内存未释放   
head = nullptr;   
// 此时内存已经被完全释放

# 四、结果

程序正确运行的结果截图。

# 五、总结

该实验涉及到的数据结构和算法，以及遇到的问题和收获。

学习了链表，并且用C++的智能指针对链表进行了实现。

练习了调试方法，增加了工程经验，提高了代码、架构水平。