

C++程序设计课程设计

实验报告

**2021/2022(2)**



实验题目 宾馆客房管理系统

学生姓名 江银

学生学号 202103151211

学生班级 计算机实验班01

任课教师 田贤忠

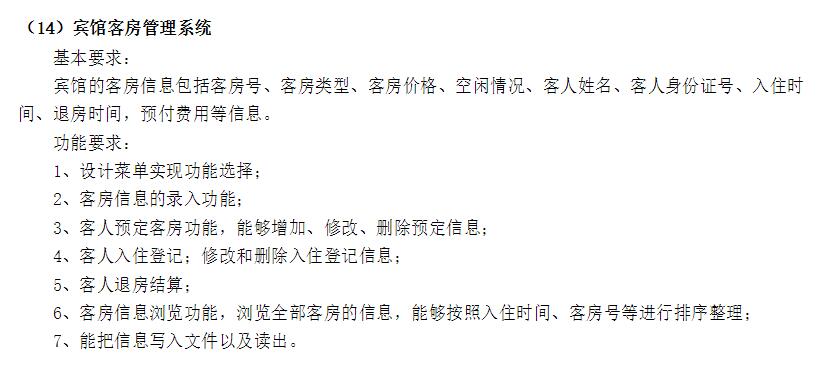
提交日期 2022.6

**计算机科学与技术学院**

**宾馆客房管理系统 实验报告**

1. **大型实验的内容**

宾馆客房管理系统用于宾馆进行客房信息的管理，要求完成的主要的功能包括宾客管理、客房管理、管理人员管理。可以完成客人预订客房、客房入住登记、退房并结算费用、客房信息排序、客房信息浏览等工作。使用学习过的C/C++程序设计的知识完成宾馆客房管理系统的设计与实现。



1. **运行环境**

宾馆客房管理系统在Visual Studio Code平台下开发，操作系统：Windows 10。

硬件环境：

处理器：Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 1.99GHz

内存：8.00GB

系统类型：64位操作系统

1. **实验课题分析（主要的模块功能、流程图）**

**3.1 宾馆客房管理系统的主要功能**

**宾馆客房管理系统主要功能为：**管理员管理、宾客管理、宾馆客房管理，可以完成客房信息录入，预订客房，客房入住登记，退房，客房信息浏览等工作。详细的系统功能结构为图1所示。

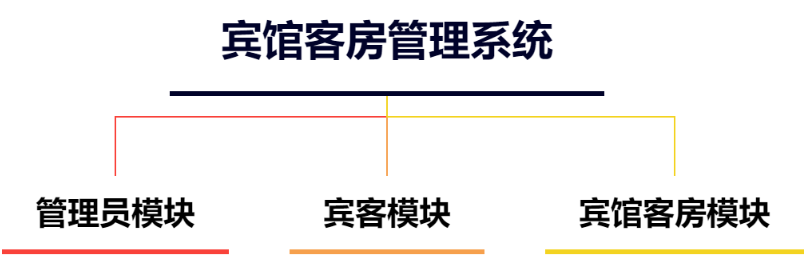


图1 系统结构图

**系统各模块的功能具体描述为：**

**1、管理员模块**

在默认界面选择管理员身份，输入用户名和密码，进行登录，然后将输入的内容与存放管理员用户名密码的文件内容相匹配，登录成功后进入管理员模块。

管理员不仅具有基本的客房信息浏览功能，还具有更高的浏览权限。管理员可以浏览客人登记的信息。

管理员有权限修改宾客入住信息以及客房信息。管理员可以修改客房价格，入住宾客姓名、身份证号、手机号，也可以重置客房的空闲情况（是否预订以及是否入住）。

管理员还可以初始化所有客房信息，将所有的预订、入住情况清零，房间价格数据回到最初的设置。

**2、宾客模块**

在默认界面选择宾客身份，即可直接进入宾客模块。宾客模块具有三个大功能：预订客房功能、客房入住登记功能、退房功能。

首先是预订客房功能。宾客选择预订客房后，跳转到客房信息浏览界面，客人需要选择空闲的客房进行预订。客人选择好客房后，需要填写姓名、身份证号、手机号，预订完成后，可以缴纳一定的预支付费用。同时模块还提供了修改预订信息功能。客人正确输入姓名和房间号后，就可以更改之前所预订的房间以及客人的信息。客人还可以选择取消预订。

然后是客房入住登记功能。宾客入住登记前，需要正确填写房间号、姓名、身份证号和手机号。核实通过以后，宾客就入住成功了，自动将当前时间作为入住时间储存到文件中。但是宾客没有权限直接更改或者删除入住登记信息。

最后是客人退房结算。客人选择退房结算后，正确输入房间号与姓名，就可以退房。自动将当前时间作为退房时间储存到文件中。退房成功后，显示结算页面。

**3、宾馆客房模块**

宾馆客房模块具有客房信息浏览功能。

管理员和宾客都有权限查看全部客房的信息。而宾客又有根据入住时间、客房号等进行排序的额外功能。

**3.2 系统分析及设计**

**系统涉及对象有两个基本类：**客房类和人员类。其中人员类又可以细分为宾客类和管理员类。人员类涉及的功能操作归纳为如下表1所示：

表1 人员类涉及的操作

|  |  |  |
| --- | --- | --- |
| **类和对象** | **涉及的类和对象操作** | |
| 人员基类 | 将房间信息根据房间号排序 | |
| 将房间信息根据入住时间排序 | |
| 管理员类 | 查询读者信息(个人信息和读者借书情况) | |
| 浏览房间信息（高权限） | |
| 更改房间入住信息 | |
| 宾客类 | 预订客房 | 添加预订信息、修改预订信息、删除预订信息 |
| 入住客房 | 添加入住信息 |
| 退房结算 | |
| 浏览房间信息（低权限） | |

可以采用面向对象的方式实现宾馆客房管理系统，根据不同的使用权限，使用对象分为管理员和宾客。分别设计管理员类、宾客类，人员类为基类实现通用的功能。

**3.3系统的实现**

**（1）类的编写**

系统包含了十余个类，Person类作为人员基类，派生出Guest类和Admin类。List类、Listnode类、Iterator类和Const\_iterator类分别都是与数据结构相关的类，Adminlist类和Roomlist类通过继承List类得到。此外，在Guest类和Admin类的部分操作中，使用了嵌套类来使程序结构清晰。最后，另外添加了Menu类和File类，来进行一些菜单界面和文件处理的问题。

具体类结构声明如下：

**Person类：**

class Person

{

public:

    virtual void view(Roomlist) = 0; //纯虚函数，在派生类中实现

    virtual ~Person() {}

    void sort\_by\_time(Roomlist &); //根据入住时间排序

    void sort\_by\_num(Roomlist &); //根据房间号排序

    static string nowTime(); //获取当前时间

};

**Guest类：**

class Guest : public Person

{

private:

    string name;

    string id;

    int cost;

public:

    virtual ~Guest() {}

    void view(Roomlist);

    class Book

    {

    public:

        void add(Roomlist &);

        void modify(Roomlist &);

        void del(Roomlist &);

    } book;

    class Checkin

    {

    public:

        void add(Roomlist &);

    } checkin;

    class Checkout

    {

    public:

        void checkout(Roomlist &);

    } checkout;

};

**Admin类：**

class Admin : public Person

{

private:

    string username;

    string password;

public:

    virtual ~Admin() {}

    void view(Roomlist);

    void set\_username(string);

    void set\_password(string);

    string get\_username();

    string get\_password();

    class Checkin

    {

    public:

        void modify(Roomlist &);

    } checkin;

};

**Room类：**

class Room

{

private:

    int number;

    string type;

    int price;

    bool isBooked;

    bool isChecked;

    string guestName;

    string guestID;

    string guestPhoneNum;

    string bookTime;

    string checkinTime;

    string checkoutTime;

public:

    Room(){};

    Room(int num, string t, int p, bool r, bool c) : number(num), type(t), price(p), isBooked(r), isChecked(c){};

    void set\_room(int, string, int, char, char);

    void set\_room(int, string, int, char, char, string, string, string, string, string, string);

    int get\_number();

    string get\_type();

    int get\_price();

    bool get\_isBooked();

    bool get\_isChecked();

    string get\_guestName();

    string get\_guestID();

    string get\_guestPhoneNum();

    string get\_bookTime();

    string get\_checkinTime();

    string get\_checkoutTime();

    void set\_number(int);

    void set\_type(string);

    void set\_price(int);

    void set\_isBooked(bool);

    void set\_isChecked(bool);

    void set\_guestName(string);

    void set\_guestID(string);

    void set\_guestPhoneNum(string);

    void set\_bookTime(string);

    void set\_checkinTime(string);

    void set\_checkoutTime(string);

    friend class Admin;

    friend class Book;

    friend class Checkin;

    friend class Checkout;

};

**（2）链表的使用**

考虑到管理员账号密码信息和房间的信息的存储，采用了链表的数据结构来实现。链表通过List模板类实现，Listnode也是模板类，相当于List类中一个抽象的结点，而Iterator类和Const\_iterator分别是迭代器模板类和常迭代器模板类，Iterator类通过继承Const\_iterator类实现。具体的结构声明如下：

**Listnode类：**

template <typename T>

class Listnode

{

public:

    T val;

    Listnode<T> \*prev;

    Listnode<T> \*next;

    Listnode(const T &v) : val(v), next(nullptr), prev(nullptr) {}

    Listnode() : next(nullptr), prev(nullptr) {}

};

**List类**

template <typename T>

class List

{

public:

    List();

    ~List();

    List(const List<T> &);

    const T &front();

    const T &back();

    int size();

    bool empty();

    void push\_front(const T &);

    void push\_back(const T &);

    void pop\_front();

    void pop\_back();

    void clear();

    void insert(Iterator<T>, const T &);

    void remove(const T &);

    IteratorC<T> begin() const

    {

        IteratorC<T> it;

        it.now = head->next;

        return it;

    }

    IteratorC<T> end() const

    {

        IteratorC<T> it;

        it.now = tail;

        return it;

    }

    Iterator<T> begin()

    {

        Iterator<T> it;

        it.now = head->next;

        return it;

    }

    Iterator<T> end()

    {

        Iterator<T> it;

        it.now = tail;

        return it;

    }

    Iterator<T> erase(Iterator<T> it)

    {

        if (!empty() && it != end())

        {

            Listnode<T> \*tmp = it.now;

            Listnode<T> \*p = tmp;

            tmp->prev->next = tmp->next;

            tmp = tmp->next;

            delete p;

            Iterator<T> res;

            res.now = tmp;

            len--;

            return res;

        }

    }

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

    T &operator[](const int);

    void display();

    void swap(Iterator<T>, Iterator<T>);

protected:

    int len;

    Listnode<T> \*head;

    Listnode<T> \*tail;

};

**Const\_iterator类**

template <typename T>

class IteratorC

{

public:

    IteratorC() : now(nullptr) {}

    const T &operator\*() const;

    IteratorC operator++(int);

    IteratorC &operator++();

    IteratorC operator--(int);

    IteratorC &operator--();

    bool operator==(const IteratorC &);

    bool operator!=(const IteratorC &);

    friend class List<T>;

protected:

    Listnode<T> \*now;

};

**Iterator类**

template <typename T>

class Iterator : public IteratorC<T>

{

public:

    Iterator() {}

    T &operator\*();

    const T &operator\*() const;

    Iterator &operator++();

    Iterator operator++(int);

    Iterator &operator--();

    Iterator operator--(int);

    friend class List<T>;

};

在编写这个List模板类的时候，我参考并模仿了标准模板库（STL）中的list容器，将它设计为双向链表，在Listnode中同时添加了prev（指向前一个元素）和next（指向后一个元素）指针。并模仿了其中的部分操作，比如begin()，end()，push\_back()等。其中，我为了编写方便，将空List设计为由头尾2个空结点组成，如下所示。

template <typename T>

List<T>::List() : head(new Listnode<T>()), tail(new Listnode<T>()), len(0)

{

    tail->prev = head;

    head->next = tail;

}

template <typename T>

List<T>::List(const List<T> &rhs) : head(new Listnode<T>()), tail(new Listnode<T>()), len(0)

{

    tail->prev = head;

    head->next = tail;

    for (IteratorC<T> it = rhs.begin(); it != rhs.end(); ++it)

        push\_back(\*it);

}

将begin()表示为返回头部空结点之后的那个元素，end()表示为尾部空结点元素。方便通过迭代器遍历（循环）。

    IteratorC<T> begin() const

    {

        IteratorC<T> it;

        it.now = head->next;

        return it;

    }

    IteratorC<T> end() const

    {

        IteratorC<T> it;

        it.now = tail;

        return it;

    }

    Iterator<T> begin()

    {

        Iterator<T> it;

        it.now = head->next;

        return it;

    }

    Iterator<T> end()

    {

        Iterator<T> it;

        it.now = tail;

        return it;

    }

以下是push\_back()方法的实现，其实就相当于将元素插入到所有结点中倒数第二个位置。

**元素的添加：**

template <typename T>

void List<T>::push\_back(const T &elem)

{

    Listnode<T> \*tmp = new Listnode<T>(elem); //创建新结点

    tmp->next = tail; //新结点的next指针指向尾部空结点

    tail->prev->next = tmp; //原本倒数第二个元素的next指针指向新结点

    tmp->prev = tail->prev; //新结点的prev指针指向原本倒数第二个元素

    tail->prev = tmp; //尾部空结点的prev指针指向新结点

    len++; //添加元素后链表总长度+1

}

为了模板类的完整性以及功能的流畅性，我也编写了insert()，remove()等方法和Iterator类以及Const\_iterator类，并通过了我自己的测试。此外，为了与程序本身功能相贴切，比如排序等功能，我又另外编写了[]运算符重载，以及swap方法。宾馆客房系统的信息的管理就具体表现为链表的操作。

**[]运算符重载的实现：**

template <typename T>

T &List<T>::operator[](const int index)

{

    if (index <= len)

    {

        Iterator<T> it = begin();

        int cnt = index;

        while (cnt--) //从头开始循环index次，即一一遍历至下标值为index的链表元素后返回

        {

            if (cnt == 0)

                break;

            it++;

        }

        return \*it;

    }

    Iterator<T> it = begin();

    return \*it; // only to avoid the situation of no return value

}

**链表中两个元素交换的实现：**

template <typename T>

void List<T>::swap(Iterator<T> it1, Iterator<T> it2) //大致分为两种情况：两个元素相邻与否，而我为了方便起见，将两个元素的前后顺序也分别用两个if写了出来。

{

    Listnode<T> \*tmp1, \*tmp2;

    tmp1 = it1.now;

    tmp2 = it2.now;

    if (tmp1->next == tmp2) //相邻且it1在it2前

    {

        tmp1->prev->next = tmp2;

        tmp1->next = tmp2->next;

        tmp2->next = tmp1;

        tmp2->next->prev = tmp1;

        tmp2->prev = tmp1->prev;

        tmp1->prev = tmp2;

    } //若两个元素相邻，则只需取出在前面的元素的前面一个元素，以及在后面的元素的之后一个元素即可，将上述这6个指针所指向的内容重新进行修改。

    else if (tmp1->prev == tmp2) //相邻且it1在it2后

    {

        tmp2->prev->next = tmp1;

        tmp2->next = tmp1->next;

        tmp1->next = tmp2;

        tmp1->next->prev = tmp2;

        tmp1->prev = tmp2->prev;

        tmp2->prev = tmp1;

    }

    else //两个链表元素不相邻

    {

        Listnode<T> \*tmp = tmp1->prev;

        Listnode<T> \*tmp\_ = tmp2->next;

        Listnode<T> \*tmp\_\_ = tmp1->next;

        Listnode<T> \*tmp\_\_\_ = tmp2->prev;

        tmp1->prev->next = tmp2;

        tmp2->next->prev = tmp1;

        tmp1->next->prev = tmp2;

        tmp2->prev->next = tmp1;

        tmp1->prev = tmp\_\_\_;

        tmp2->next = tmp\_\_;

        tmp1->next = tmp\_;

        tmp2->prev = tmp;

    } //如果两个元素不相邻，则这两个元素的前后的相邻元素都需要被取出，也就是有8个指针所指向的内容需要被修改。

}

Adminlist是通过继承List类，将模板类实例化为Admin类型得到的类，而Roomlist是通过继承List类，将模板类实例化为Room类型得到的类。这两个类的结构声明如下：

**Adminlist类**

class Adminlist : public List<Admin>

{

};

**Roomlist类**

class Roomlist : public List<Room>

{

};

为了实现根据房间号或者入住时间来进行房间的排序功能，我编写了如下程序段：

**链表元素选择排序：**

int len = roomlist.size();

    auto beg = roomlist.begin();

    auto posi = beg, posj = beg, tmp = beg;

    for (int i = 1; i < len; i++)

    {

        posj = posi;

        tmp = posi;

        for (int j = i + 1; j <= len; j++)

        {

            posj++;

            Room room1 = \*posi;

            Room room2 = \*posj;

            if (room1.get\_number() > room2.get\_number())

            {

                tmp = posj;

            }

        }

        if (tmp != posi)

        {

            roomlist.swap(tmp, posi);

            posi = tmp;

        }

        posi++;

    }

选择排序的实现本身较为简单，每次循环找到剩余位置中的最值，然后跟当前位置交换。但是和链表结合，感觉并没有那么简单。在每一轮找完剩余位置中的最值后，必须加上if (tmp!=posi)这一句判断，因为我并没有实现两个同一位置的元素进行交换的容错功能。而交换之后，posi=tmp这一句更为重要，具体在下面的“遇到的问题”中会再次提到。

上图就以根据房间号排序为例。

**（3）交互界面的实现**

Menu类的功能操作如下所示：

**Menu类：**

class Menu

{

public:

    void defaultMenu(); //默认初始界面

    void mainMenu(); //主界面

    void bookMenu(); //宾客预订界面

    void loginMenu(); //管理员登录界面

    void adminMenu(); //管理员操作界面

    void viewMenu(); //浏览客房信息界面

};

Menu类完成了六个界面的实现。首先是运行宾馆客房管理系统后的初始默认界面。具体界面如下：

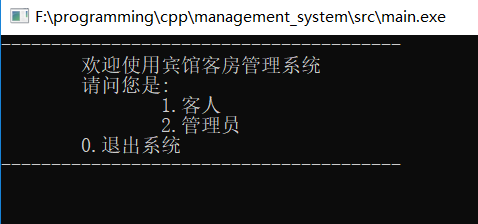
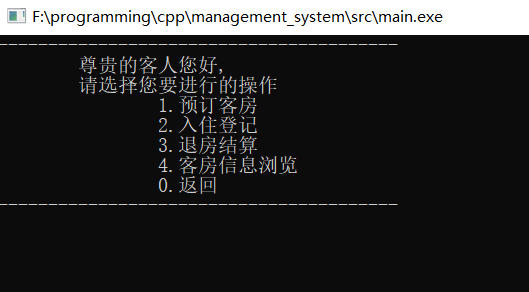


图2 初始默认界面

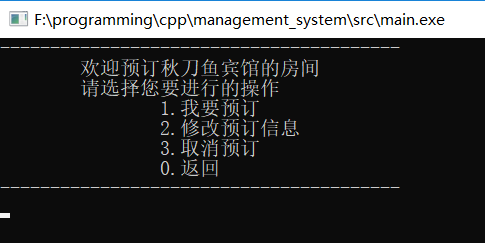
主要通过选择结构和循环结构实现界面的前进和后退，部分程序段采用try-catch进行异常处理。在该初始默认界面出现：1.客人，2.管理员，0.结束。选择当前操作人的身份，然后进入相应的界面。如果选择0，终止程序。

**操作人：客人**

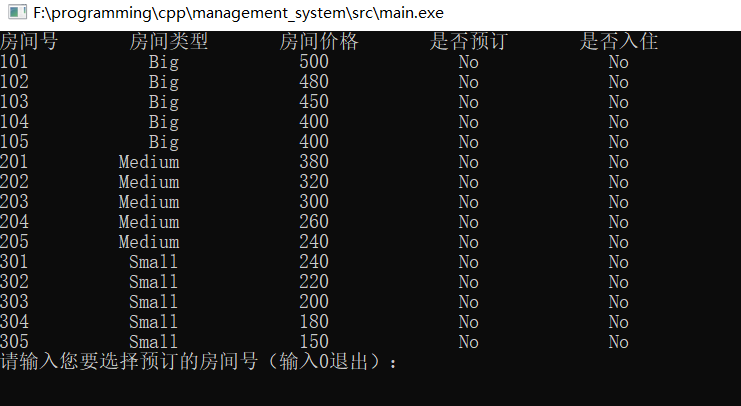
选择1，进入如下界面：



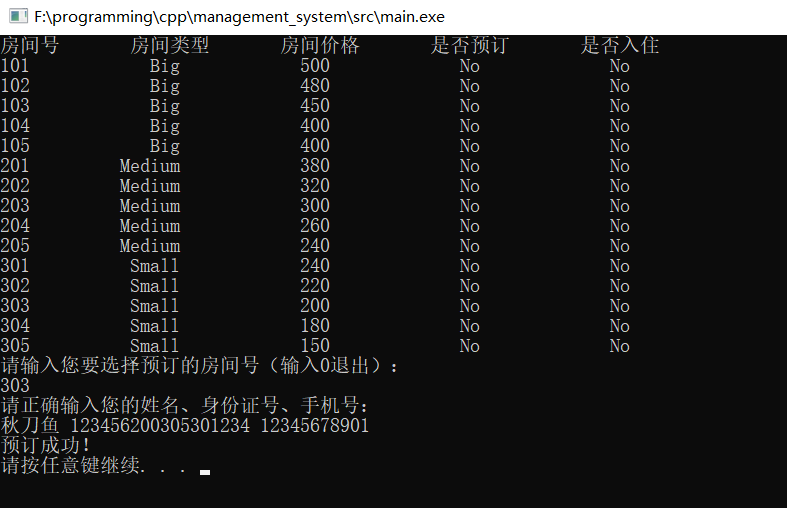
继续选择1，进入如下界面：

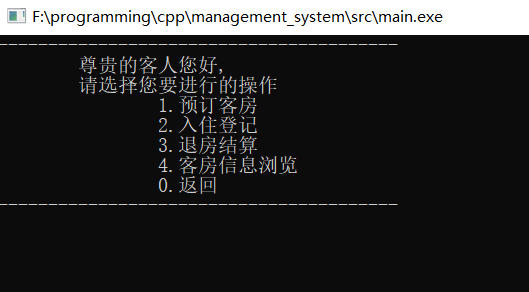


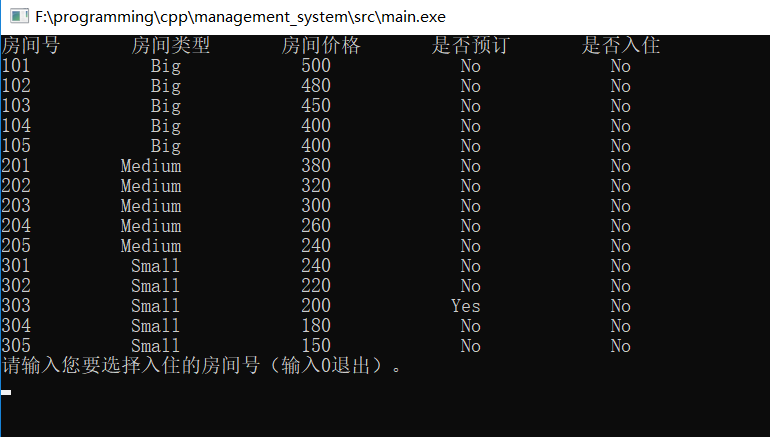
继续选择1，进入如下界面：



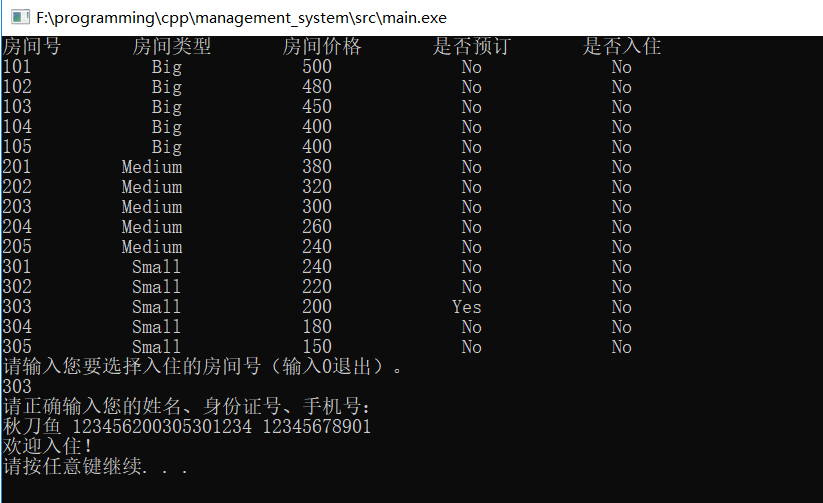
输入房间号以及个人信息后：

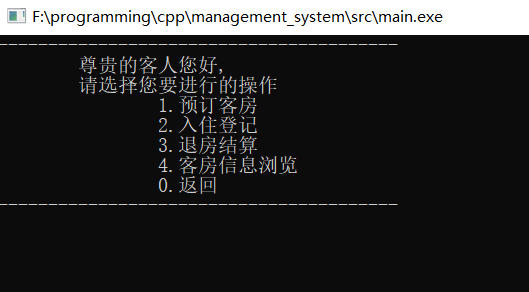


显示预订成功。按任意键继续后，返回宾客的操作界面。

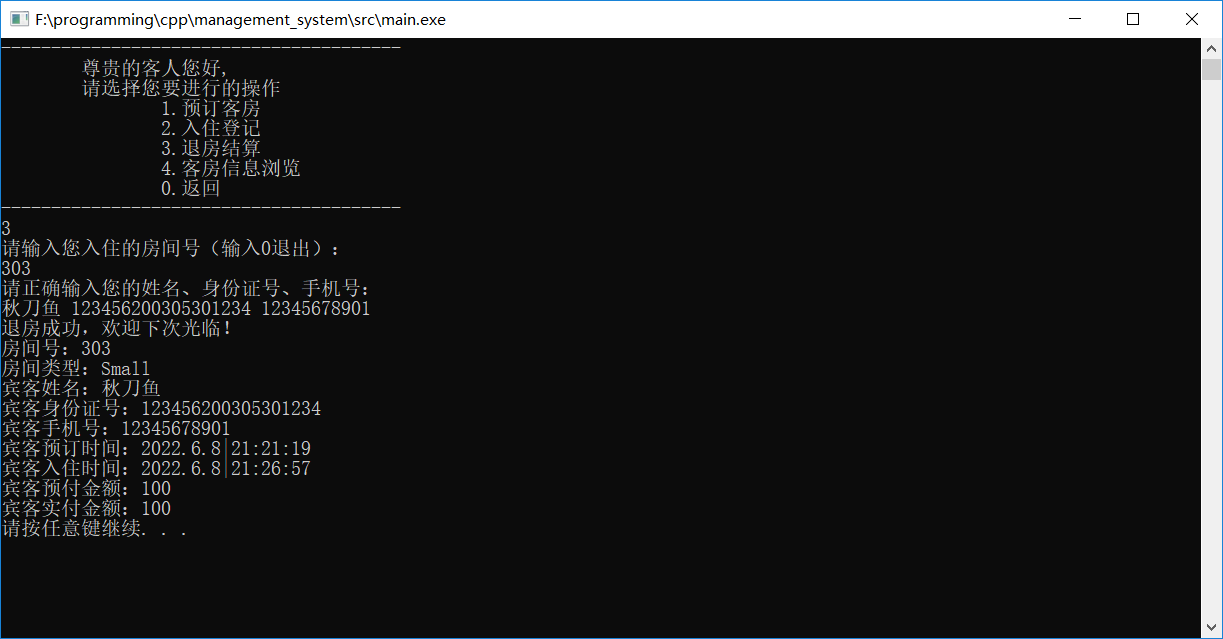
这一次选择入住登记。

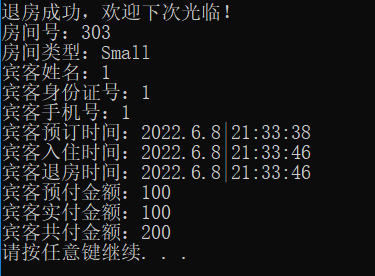
正确输入要入住的房间号和对应的个人信息，要与之前预订所输入的信息相一致。



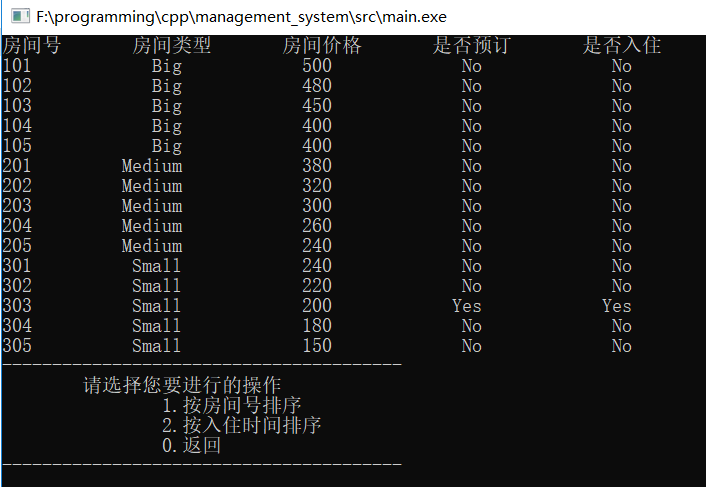
信息匹配成功后，系统反馈“欢迎入住”。按任意键后继续返回宾客的操作界面。

选择3，进行退房结算。输入房间号，以及个人信息后，系统反馈“退房成功，欢迎下次光临！”并且将该宾客的所有客房信息以及个人信息以及预约、入住、退房时间输出。

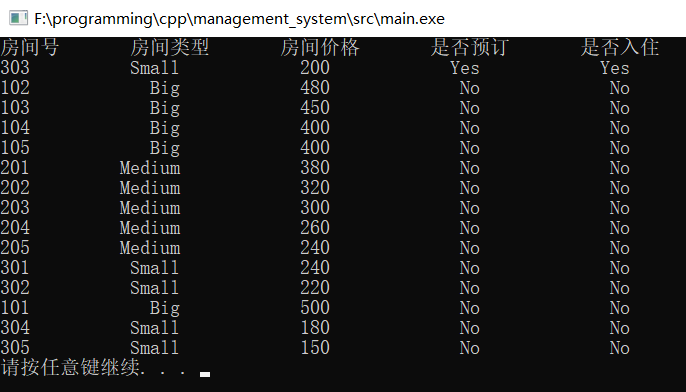




而当客人选择客房信息预览功能时，则会出现如下界面：



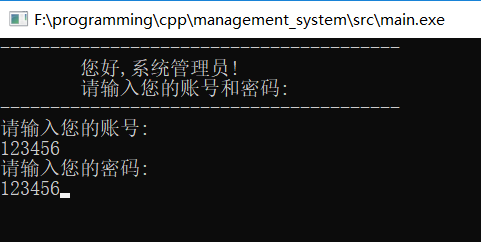
以按照入住时间排序为例，排序后如下图所示：



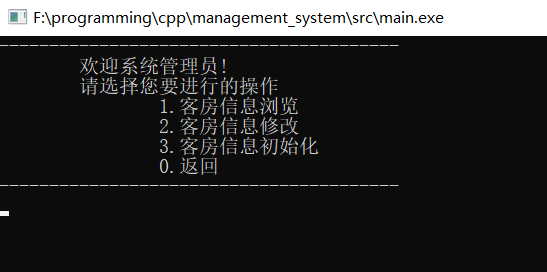
按任意键后退出。

**操作人：管理员**

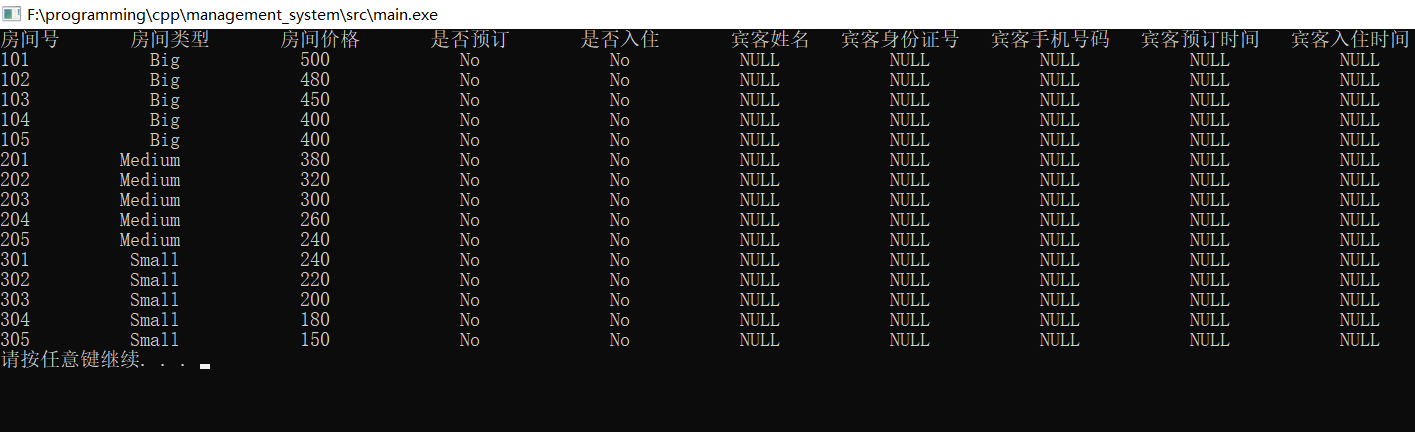
选择2，进入如下界面：



输入正确的账号密码后，进入如下界面：



选择2或选择3，分别实现文字对应功能。选择1，进入客房信息浏览（高权限）界面，如下图所示（初始化后的数据）。



按任意键继续后，返回初始默认界面（选择宾客和管理员身份界面）。

1. **文件读写**

文件操作在这个系统中起到了不容忽视的作用。

**File类：**

class File

{

public:

    Adminlist get\_admin\_data(); //获取管理员账号信息

    Roomlist get\_room\_data(); //获取房间信息

    void roomInit(); //房间信息初始化

    void write(Roomlist &); //信息写入文本文件

};

**main函数中程序开始运行时获取数据的代码段：**

File file;

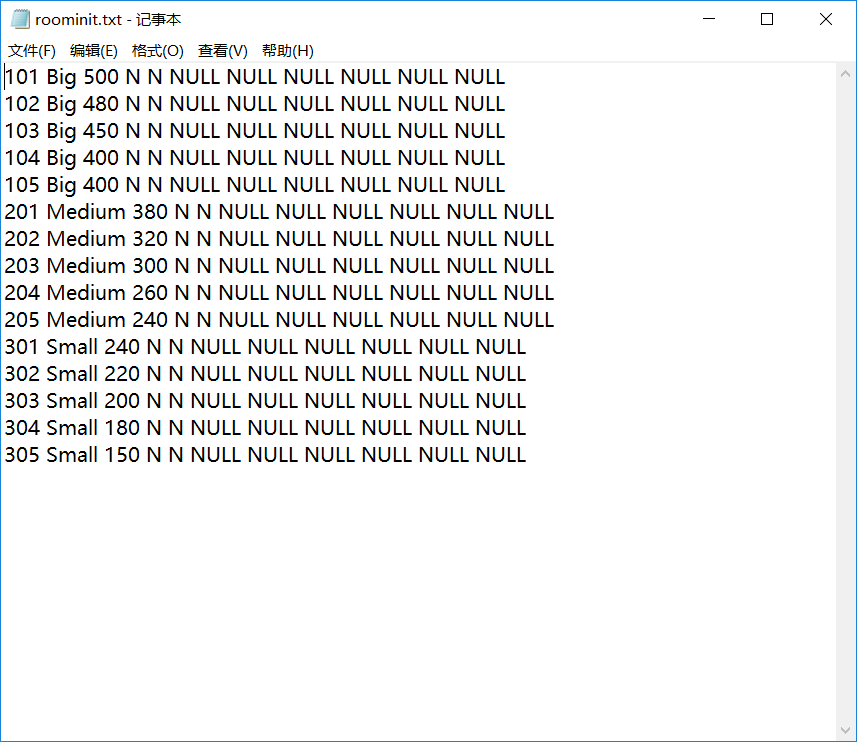
    Adminlist adminlist;

    adminlist = file.get\_admin\_data(); //获取管理员账号密码

    Roomlist roomlist;

    roomlist = file.get\_room\_data(); //获取客房信息

上面五行代码是系统开始运行后main函数最早执行的几条语句。在data文件夹中我创建了admindata.txt，roomdata.txt，roominit.txt三个文本文件。Admindata.txt存放的是管理员的账号和密码，通过file对象的get\_admin\_data()成员函数，使adminlist获取到管理员账号密码数据，以待接下来管理员登录时的账号密码匹配。



而当系统没有使用过时，roomdata.txt和roominit.txt存放的都是最初的房间数据，有着房间号、房间类型、房间价格信息，但其它数值都为空。通过file对象的get\_room\_data()成员函数，使roomlist获取到房间信息。

当接下来客人或者管理员进行一系列对于客房信息（roomlist）的修改操作后，都会执行write()这个成员函数，将roomlist的内容重新写入roomdata.txt文件中。所以每一次关闭系统，重新打开系统后，可以保证获取的数据是实时的，之前最终的结果会保留下来。

而如果要将所有数据初始化，可以用roomInit()将roomlist的值全部变成roominit.txt中设定的初始值。

1. **实验中遇到的问题**

系统在测试过程中遇到若干问题，但经过不断调试，我已经消除了我能找到的所有bug。遇到的部分问题和解决方法如下：

* + **问题1：**

**问题描述：**Iterator类和List类互相引用，即List类中定义了以Iterator类为数据类型的返回值的函数，而Iterator类中要将List类作为友元类。在main.cpp运行代码，发现编译出错，错误信息是说认为List并不是一个模板类。

**解决方法：**我在定义Iterator类之前对List类进行预定义，即加上两行代码：

Template<typename T>

Class List;

经调试，编译通过。

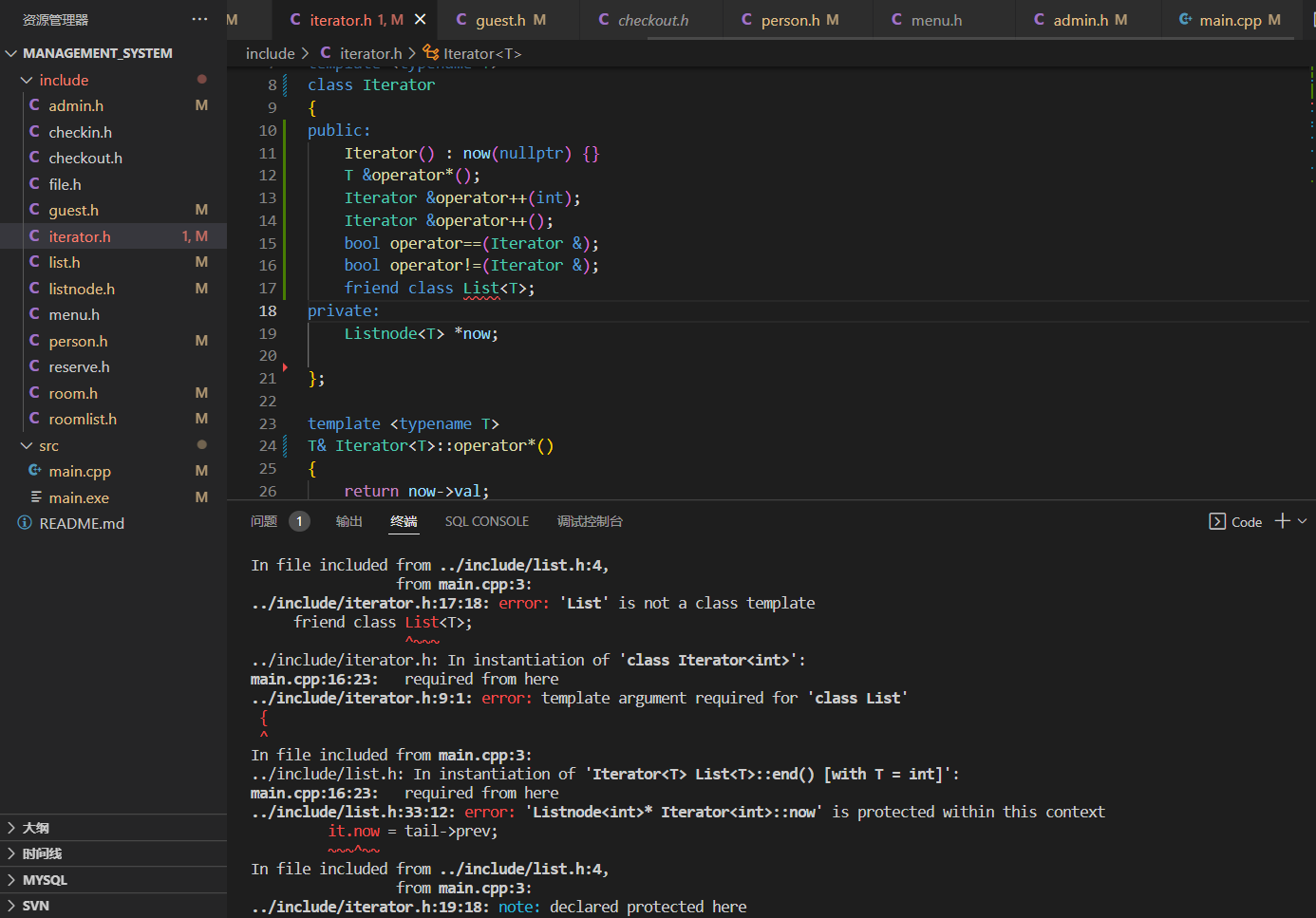


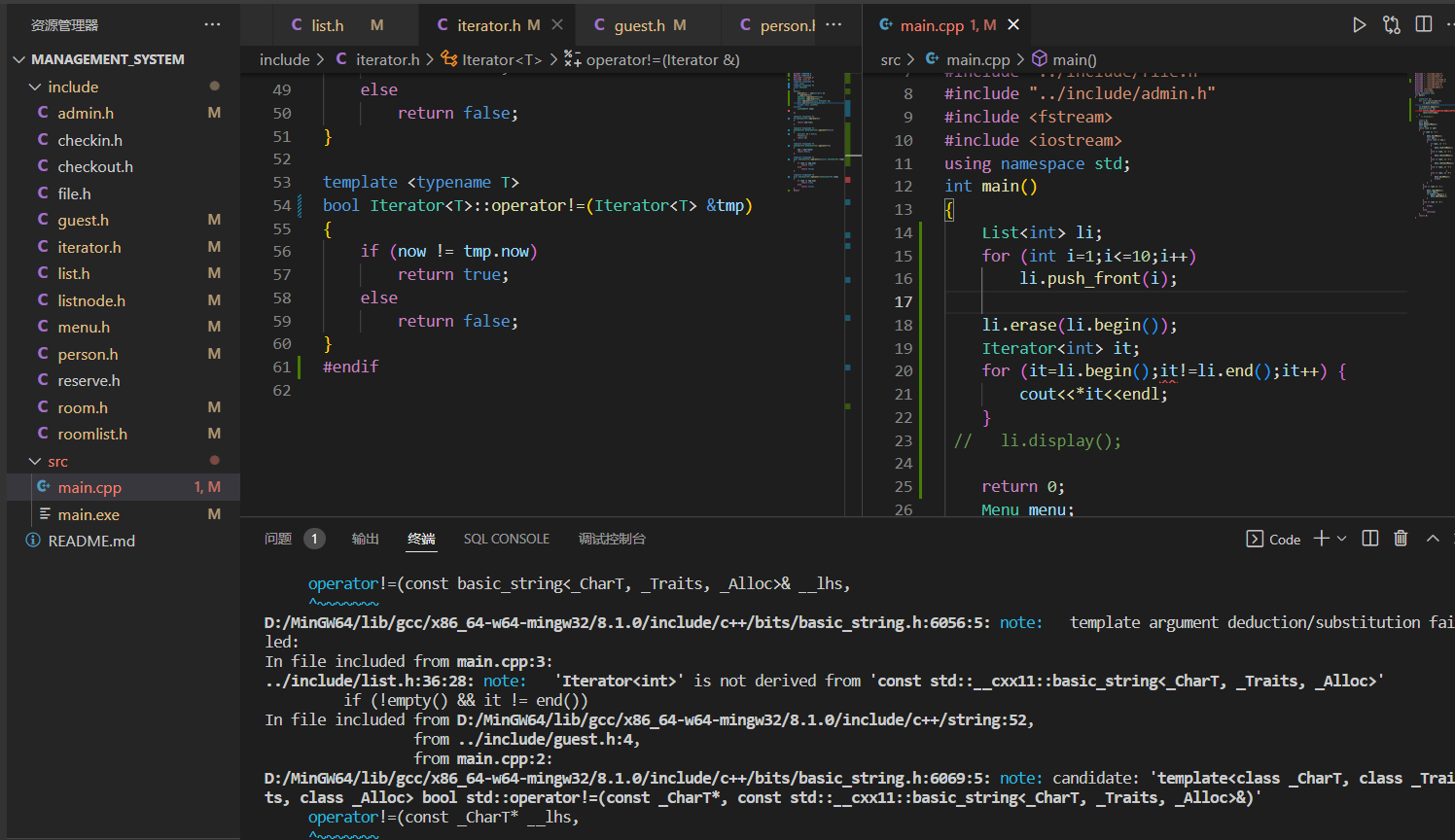
图6 调试测试问题1

* + **问题2：**

**问题描述：**在main函数中尝试遍历链表的值并输出，使用!=运算符作为循环条件，编译器提示没有与!=相匹配的运算符。

**解决方法：**我在Iterator.h中对迭代器!=运算符的重载部分，将参数Iterator<T> &tmp改为了const Iterator<T> &tmp。因为该值是一个临时变量，不能直接引用，应该加上const引用。

经调试，编译通过。



* + **问题3：**

**问题描述：**在编写管理员模块时，用到了迭代器对管理员链表的遍历。我将管理员的数据封装在Admin类中，并创建Adminlist继承List类，结点类型为Admin类，存放Admin的数据。但在编写完成后，进行编译，发现通不过。

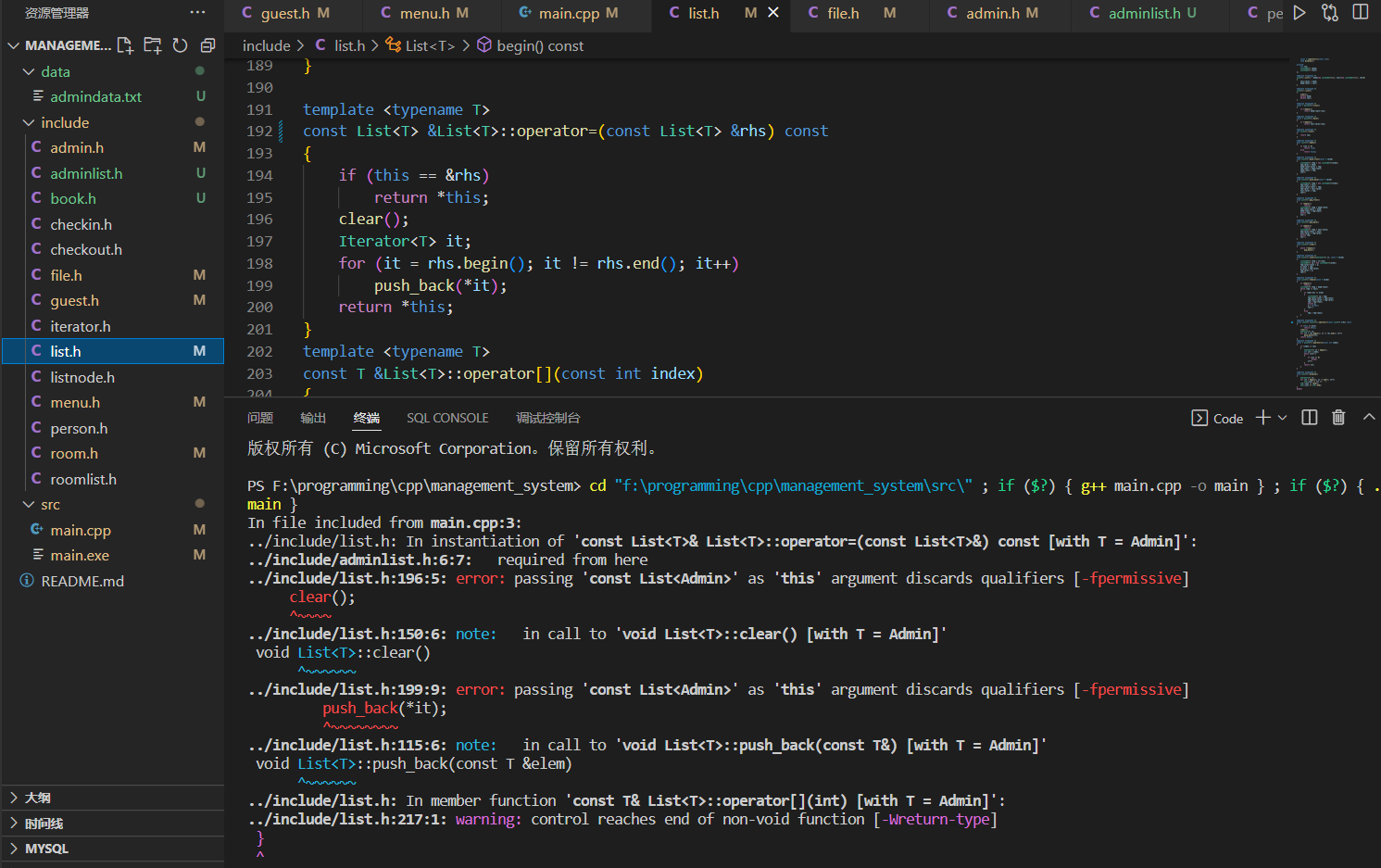
错误信息如下：

error: passing 'const List<Admin>' as 'this' argument discards qualifiers [-fpermissive]

**解决方法：**

在=的运算符重载中，我使用的是清空this指针指向的内容，然后将传入的参数通过一次循环逐个赋值给this指针指向的内容，并最终将\*this作为返回值。可能是在循环的过程中，一系列操作，用到了许多List类中的成员函数，而编译器认为这些成员函数可能会改变=运算符重载函数中const常量的值，从而产生error。而在查阅资料以后，我决定在原有的迭代器基础上重新写一个常迭代器，即const迭代器。

经调试，编译通过。

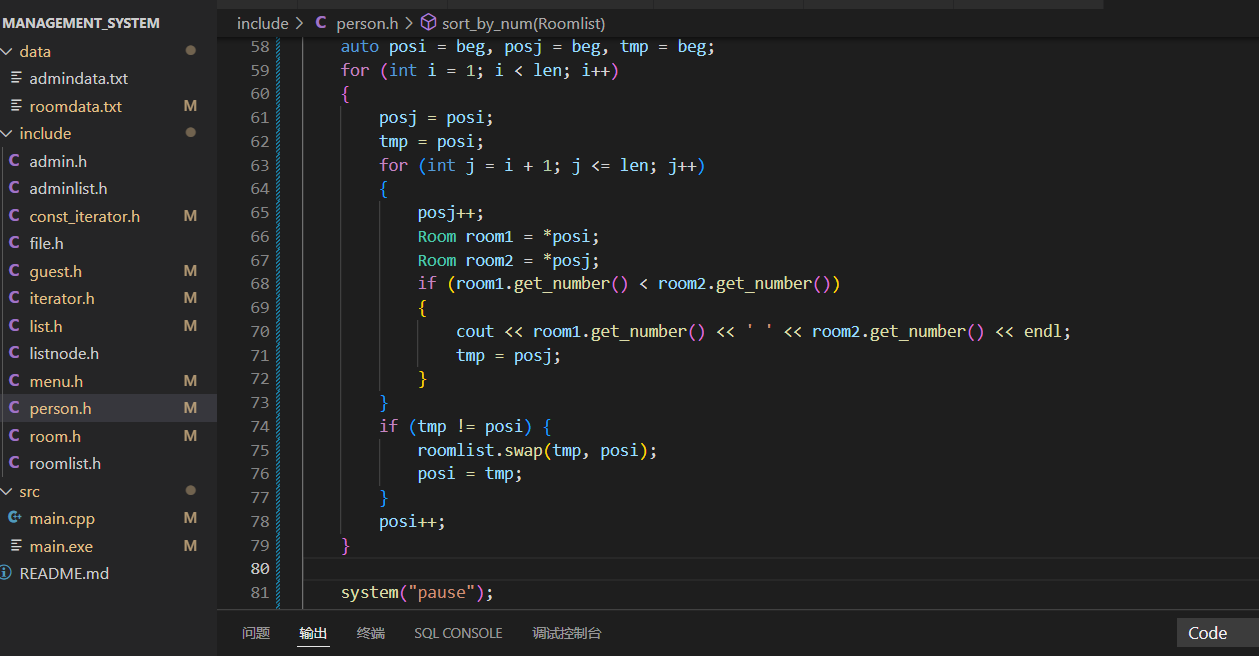


* + **问题4：**

**问题描述：**排序时可能出现死循环，可能无输出跳出程序。

**解决方法：**在75行调用交换链表两个元素的函数有不曾考虑到的情况。交换链表中两个元素需要考虑两个元素相邻或者不相邻的两种情况。修复以后，尝试再次运行程序，依然运行出错。尝试在76行加上posi = tmp这一语句。

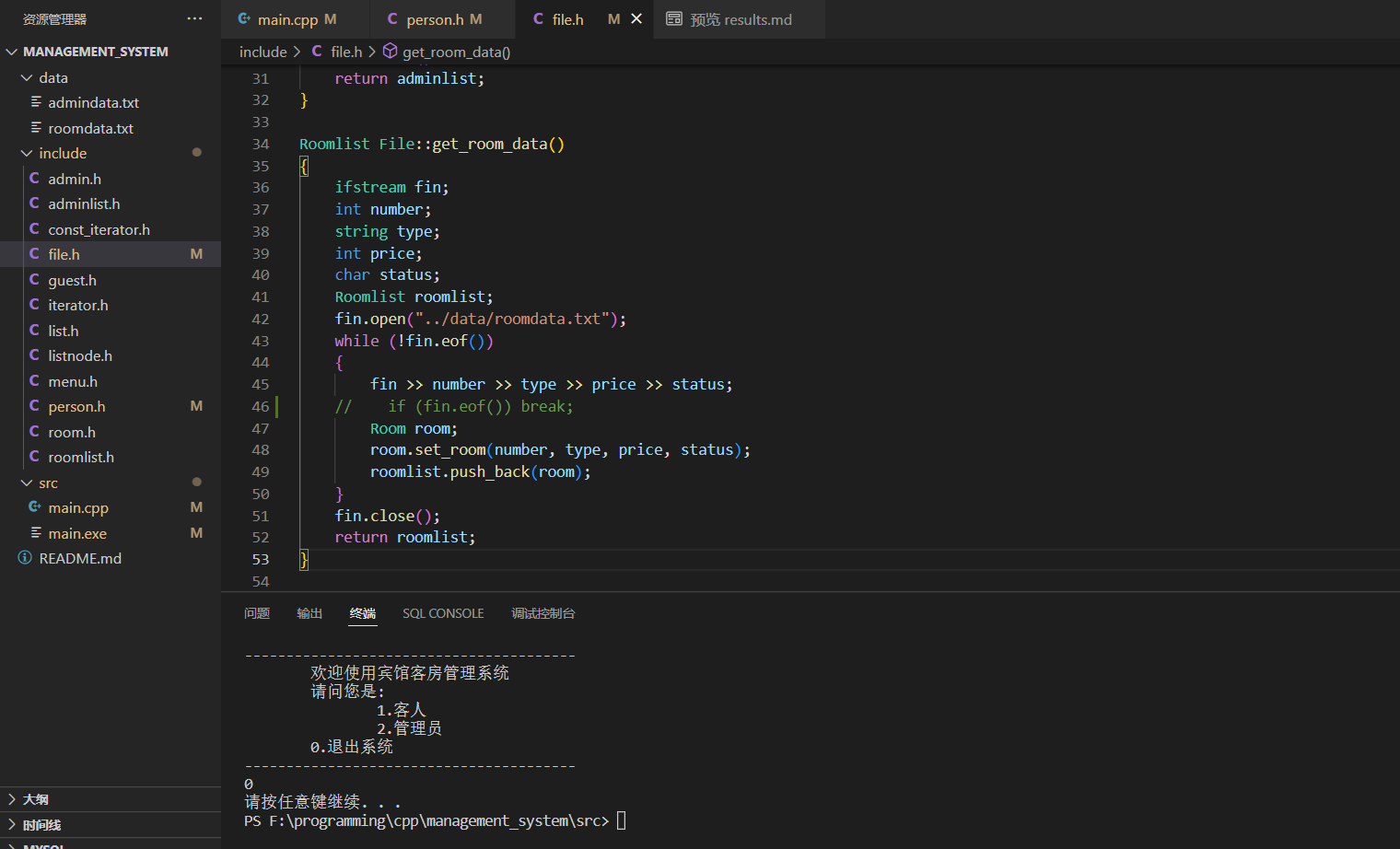
在swap函数中，做到了链表元素的交换，但在这个排序的程序段中，仍然需要将tmp的值赋值给posi，因为这是交换完链表元素后，当前i位置的指针（迭代器）。运行posi++语句后，posi顺利地来到了下一个位置，而不是原来那样，指针飞了。



* + **问题5：**

**问题描述：**从文件中提取客房数据时，最后一个数据提取了两次。

**解决方法：**加上46行的代码，并将循环改为while (1)，运行，最后一个数据只提取了一次，成功达到目标。原因应该是读取最后一行时，fin对象此时并没有end of file，还是读取到了数据，而只有再次读取时，eof才会为true。



之后的问题虽然也不少，但比较简单而且不具有代表性，不一一罗列。

1. **实验总结（优点、不足、收获及体会）**

我设计的宾馆客房管理系统满足了实验要求。我将面向对象的程序设计思想融于其中，设计了许多类，并结合了本学期所授课的各个知识点内容。我设计了链表模板类，并且设计了迭代器类和常迭代器类来实现一些链表中的功能；面向对象程序设计中的封装性、继承性、多态性也都在我设计的系统中随处可见，我写了许多函数重载、运算符重载的程序段，设计了人员基类，并通过基类继承了管理员类和宾客类，并在人员基类中设计了一个纯虚函数；我也通过文件操作完成了许多功能，也运用了异常处理中的try-catch；我也来在用户交互上，我也尽力将数据和文字展示得更加符合自然，比如运用了流格式控制符来控制文字的间距等，“返回”键的操作更是人性化，让许多误操作得以修改，各个层级之间也较为清晰。

但是这个系统也有一些不足。与用户的交互上可以再丰富一些，比如可以实时显示剩余房间的总数。另外，在数据排序上我采用了选择排序，选择排序相对于冒泡排序，交换结点的次数会减少，但相对于归并排序来说，效率就显得低下了。

本次大型试验我也花了许多时间来完成，代码行数1700行左右。在编写链表模板类的时候，我对链表这一数据结构的部分操作和处理有了更深刻的了解，体验了链表的添加元素、删除元素、交换元素、元素自定义排序等操作，并且模拟标准模板库中的一些方法进行了相关实现。也对面向对象的程序设计中的封装性、继承性、多态性有了更深的了解，在实践中得到了成长。

1. **附录：源代码**