项目说明文档

数据结构课程设计

——约瑟夫游戏

作 者 姓 名　林继申

学　　号　2250758

指 导 教 师　张　颖

学 院 专 业　软件学院　软件工程

同济大学 TONGJI UNIVERSITY

二〇二三 年 十二 月 十三 日

目录

1 项目分析

1.1 项目背景分析

约瑟夫生者死者游戏的大意是：30 个旅客同乘一条船，因为严重超载，加上风高浪大危险万分，因此船长告诉乘客，只有将全船一半的旅客投入海中，其余人才能幸免于难。无奈，大家只得统一这种方法，并议定 30 个人围成一圈，由第一个人开始，依次报数，数到第 9 人，便将他投入大海中，然后从他的下一个人数起，数到第 9 人，再将他投入大海，如此循环，直到剩下 15 个乘客为止。问哪些位置是将被扔下大海的位置。

本游戏的数学建模如下：N 个人按顺序排成一个环形，依次顺序编号为 1 到 N，从第 S 号开始，沿环顺序计数，每数到第 M 个人就将其淘汰，且从下一个人开始重新计数，重复这个过程，直到剩下 K 个人为止。

约瑟夫生者死者游戏是一个经典的数学问题，这个问题有着悠久的历史，引发了人们对数学、数据结构和算法的探讨。

1.2 项目需求分析

基于以上背景分析，本项目需要实现需求如下：

(1)实现约瑟夫生者死者游戏的核心逻辑，要求用户输入总人数、起始位置、间隔人数、剩余人数，并输出被淘汰人的位置和剩余人的位置；

(2)设计简单直观的控制台界面，使操作便捷、容易上手，适应不同用户的操作习惯；

(3)选择合适的数据结构，并且将所有人的序号作为一组数据存放在某种数据结构中；

(4)实现异常处理机制，确保系统稳定性和安全性，避免因用户输入错误导致系统崩溃或信息丢失。

1.3 项目功能分析

本项目旨在使用带表头结点的单循环链表的数据结构，实现约瑟夫生者死者游戏的核心逻辑。

本项目需要处理输入、建立带表头结点的单循环链表、游戏核心逻辑和异常处理等功能。

1.3.1 输入功能

允许用户输入以下游戏参数：

(1)总人数 N；

(2)起始位置 S；

(3)间隔人数 M；

(4)剩余人数 K。

这些输入参数将用于配置游戏的规则和开始模拟游戏过程。

1.3.2 建立带表头结点的单循环链表功能

在游戏开始前，需要建立一个带表头结点的单循环链表。该链表存储了所有人的序号（从 1 到 N)。这个链表的特点是最后一个节点指向第一个节点，形成一个循环结构，以符合游戏的环形特性。

1.3.3 游戏核心逻辑功能

游戏核心逻辑是本项目的关键部分。在游戏进行过程中，程序会按照以下步骤进行：

(1)从指定的序号开始；

(2)沿着链表依次数 M 个人，然后将这个人从链表中删除；

(3)游戏继续，从下一个人开始重新计数，继续进行上述步骤，直到只剩下 K 个人。

1.3.4 异常处理功能

实现异常处理机制，处理用户可能输入的非法信息，确保系统的稳定性和安全性。

2 项目设计

2.1 数据结构设计

基于项目分析，在约瑟夫生者死者游戏中，需要管理每个人的序号，同时需要按照一定规则进行淘汰，直到达到剩余人数。

关于数据结构设计有以下两个要点：

(1)每个人都有一个唯一的序号(从 1 到 N),这些序号需要以某种方式存储，以便程序能够按照指定规则删除某一个人；

(2)本游戏中的人是围成一圈的，因此数据结构需要反映这种循环性质，即

最后一个人的下一个人即是第一个人。

基于上述分析，本项目选择带表头结点的单循环链表作为数据结构，带表头结点的单循环链表有如下优点：

⑴循环链表的节点结构非常适合处理这种循环游戏的场景。每个节点都包含一个人的序号，并且指向下一个节点，这使得按规则删除某一个人非常容易，只需调整指向节点的指针；

⑵循环链表自然地反映了游戏的循环性质，因为最后一个节点指向第一个节点，从而实现了人之间的环形关系；

⑶在循环链表中，删除节点的操作是非常高效的，因为只需修改指向节点的指针，而不需要大规模的数据移动；

⑷循环链表节点的结构直观地映射到游戏规则，使得代码更加容易理解和维护，这种数据结构设计有助于简化游戏的核心逻辑实现。

2.2 结构体与类设计

2.2.1 MyCircLinkNode 结构体的设计

2.2.1.1 概述

MyCircLinkNode 结构体是一个用于构建循环链表节点的模板结构体。该结构体用于表示循环链表中的每个节点，其中包括节点存储的数据以及指向下一个节点的指针。经典的循环链表一般包括两个抽象数据类型（ADT）——链表结点类（CircLinkNode）与链表类（CircLinkList）。本项目希望链表结点类可以直接访问链表结点，所以使用 struct 而不是 class 描述链表结点类。

2.2.1.2 结构体定义

```
struct MyCircLinkNode {
    Type data;
    MyCircLinkNode<Type>* link;
    MyCircLinkNode(MyCircLinkNode<Type>* ptr = NULL) { data = 0, link = ptr; }
    MyCircLinkNode(const Type& item, MyCircLinkNode<Type>* ptr = NULL) { data = item; link = ptr; }
};
```

2.2.1.3 数据成员

Type data：数据域，存储节点的数据

MyCircLinkNode<Type>* link：指针域，指向下一个节点的指针

2.2.1.4 构造函数

MyCircLinkNode(MyCircLinkNode<Type>* ptr = NULL);

构造函数，初始化指针域。

MyCircLinkNode(const Type& item, MyCircLinkNode<Type>* ptr = NULL);

构造函数，初始化数据域和指针域。

2.2.2 MyCircList 类的设计

2.2.2.1 概述

该通用模板类 MyCircList 用于表示单循环链表。此循环链表以附加的头节点作为起点，简化了操作和提高了效率。链表节点由 MyCircLinkNode 结构体表示，其中包含数据和指向下一个节点的指针。该循环链表提供了一系列基本操作函数，包括节点的插入、删除、查找、访问等，以及循环链表的构造和析构，满足了常见的循环链表操作需求。

2.2.2.2 类定义

```
template <typename Type>
class MyCircList {
private:
    MyCircLinkNode<Type>* first;
    MyCircLinkNode<Type>* last;
public:
    MyCircList();
    MyCircList(const Type& item);
    MyCircList(MyCircList<Type>& L);
    ~MyCircList();
    void makeEmpty(void);
    int getLength(void) const;
    MyCircLinkNode<Type>* getHead(void) const;
    MyCircLinkNode<Type>* getTail(void) const;
    MyCircLinkNode<Type>* search(Type item) const;
    MyCircLinkNode<Type>* locate(int i) const;
    bool getData(int i, Type& item) const;
    bool setData(int i, Type& item);
```

```
    bool insert(int i, Type& item);
    bool remove(int i, Type& item);
    bool isEmpty(void) const;
    void output(void) const;
    MyCircList<Type>& operator=(MyCircList<Type> L);
};
```

2.2.2.3 私有数据成员

`MyCircLinkNode<Type>* first`：指向循环链表的第一个节点（头节点）的指针

`MyCircLinkNode<Type>* last`：指向循环链表的最后一个节点的指针

2.2.2.4 构造函数

`MyCircList();`

默认构造函数，创建一个空循环链表。

`MyCircList(const Type& item);`

转换构造函数，创建一个只包含一个元素的循环链表。

`MyCircList(MyCircList<Type>& L);`

复制构造函数，通过复制另一个链表创建新循环链表。

2.2.2.5 析构函数

`~MyCircList();`

析构函数，释放循环链表的内存资源，包括所有节点的内存。

2.2.2.6 公有成员函数

`void makeEmpty(void);`

清空循环链表，释放所有节点的内存。

`int getLength(void) const;`

获取循环链表中节点的个数。

`MyCircLinkNode<Type>* getHead(void) const;`

获取循环链表头节点的指针。

`MyCircLinkNode<Type>* getTail(void) const;`

获取循环链表尾节点的指针。

`MyCircLinkNode<Type>* search(Type item) const;`

搜索循环链表中值为 `item` 的节点，返回该节点的指针，若不存在返回 `NULL`。

`MyCircLinkNode<Type>* locate(int i) const;`

返回循环链表中第 i 个节点的指针，若 i 超出循环链表长度或小于 0，则返回 NULL。

```
bool getData(int i, Type& item) const;
```

获取循环链表中第 i 个节点的数据，并通过引用返回。返回值为操作是否成功。

```
bool setData(int i, Type& item);
```

设置循环链表中第 i 个节点的数据。返回值为操作是否成功。

```
bool insert(int i, Type& item);
```

在循环链表中第 i 个节点后插入新节点。返回值为操作是否成功。

```
bool remove(int i, Type& item);
```

删除循环链表中第 i 个节点，并通过引用返回其数据。返回值为操作是否成功。

```
bool isEmpty(void) const;
```

检查循环链表是否为空。

```
void output(void) const;
```

输出循环链表中所有节点的数据。

2.2.2.7 运算符重载

```
MyCircList<Type>& operator=(MyCircList<Type> L);
```

重载赋值运算符，用于将一个循环链表赋值给另一个循环链表。

2.3 项目主体架构设计

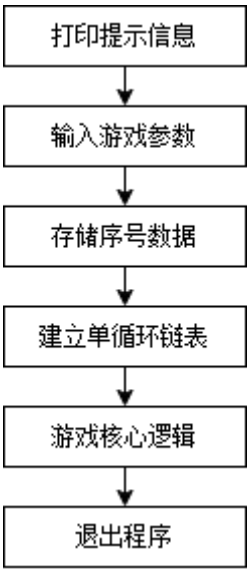

图 2.3.1 项目主体架构设计流程图

项目主体架构设计为：

(1)输出提示信息，并要求用户输入四个参数：总人数(N)、起始位置(S)、间隔人数(M)、剩余人数(K)；

(2)创建一个动态分配的整数数组 order 用于所有人的初始序号；

(3)在游戏开始前，建立一个带表头结点的单循环链表；

(4)执行游戏的核心逻辑功能；

(5)游戏结束后，输出游戏结束的信息，包括剩余人数和剩余人员的位置。

3 项目功能实现

3.1 输入功能的实现

3.1.1 输入功能实现思路

输入功能的实现思路为：

(1)输入总人数 N，N 为正整数，范围在 1 至 INT_MAX（2147483647）之间，小于 1 数据无意义，大于 INT_MAX（2147483647）超过 int 类型所能存储的最大数据；

(2)输入起始位置 S，S 为正整数，范围在 1 至 N 之间，小于 1 数据无意义，大于 N 数据无意义；

(3)输入间隔人数 M，M 为正整数，范围在 1 至 INT_MAX（2147483647）之间，小于 1 数据无意义，大于 INT_MAX（2147483647）超过 int 类型所能存储的最大数据；

(4)输入剩余人数 K，K 为正整数，范围在 1 至 N-1 之间。小于 1 数据无意义，大于 N-1 则全部人员都剩余，无意义；

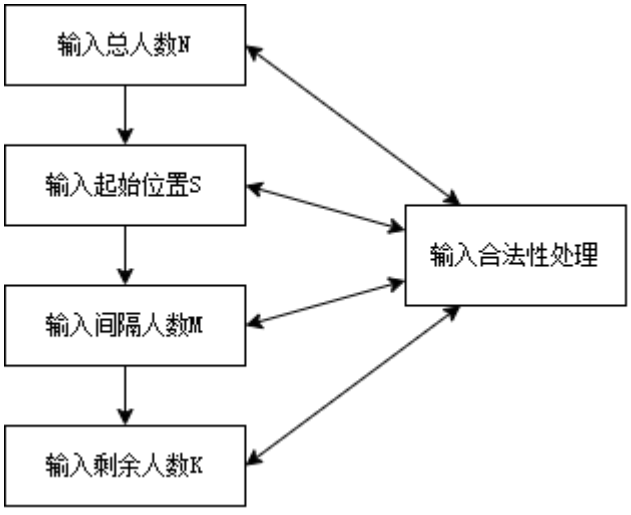

图 3.1.1.1 输入功能实现流程图

输入完成后，首先分配和初始化一个整数数组。代码使用new(std::nothrow)动态分配了一个整数数组，用于存储每个人的序号。每个人的序号在这个数组中按顺序存储，从1开始递增，依次为1，2，3，…，N。

之后代码计算了一个叫做 numDigits 的变量，该变量用于格式化输出。计算的方式是通过减去剩余人数 K 从总人数 N 中得到一个数字 num，然后通过不断地将 num 除 10 并增加 numDigits 的方式来确定 num 的位数。这是为了确保输出格式的对齐性，以便输出淘汰的人的位置。

3.1.2 输入功能核心代码

```
    /* Input */
    printPrompt();
    std::cout << ">>> 请输入总人数、起始位置、间隔人数、剩余人数" << std::endl
<< std::endl;
    int N = inputInteger(1, INT_MAX, "总人数 N ");
    std::cout << std::endl;
    int S = inputInteger(1, N, "起始位置 S ");
    std::cout << std::endl;
    int M = inputInteger(1, INT_MAX, "间隔人数 M ");
    std::cout << std::endl;
    int K = inputInteger(0, N - 1, "剩余人数 K ");

    /* Save the order number of each person */
    int* order = new(std::nothrow) int[N];
    if (order == NULL) {
        std::cerr << "Error: Memory allocation failed." << std::endl;
        exit(MEMORY_ALLOCATION_ERROR);
    }
    for (int count = 0; count < N; count++)
        order[count] = count + 1;

    /* Calculate the number of eliminated people's digits for formatting
output */
    int numDigits = 0, num = N - K;
    while (num != 0) {
        num /= 10;
        numDigits++;
    }
```

3.1.3 输入功能示例

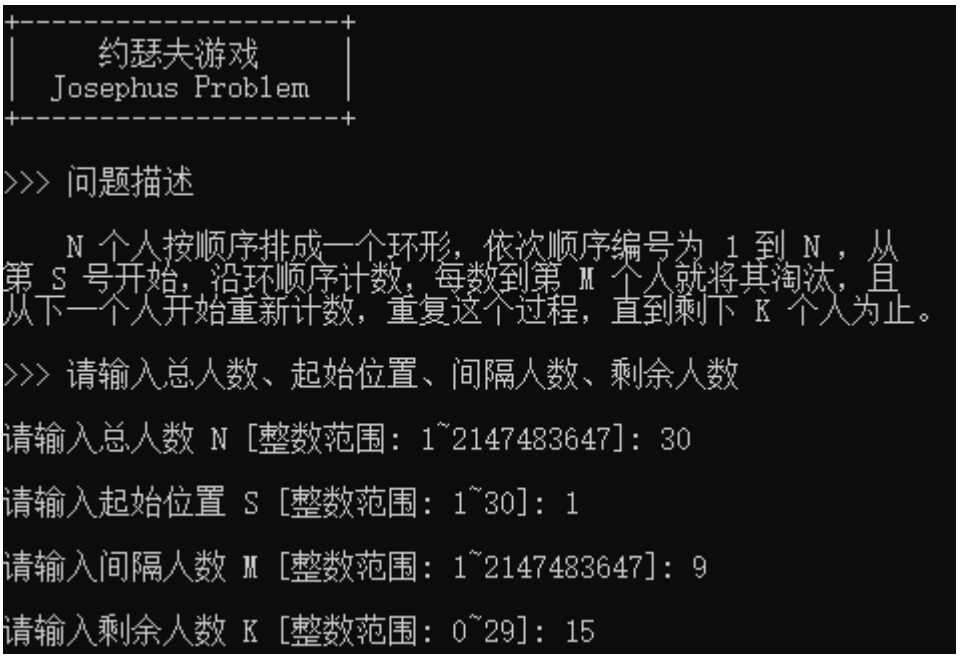

图 3.1.3.1 输入功能示例

3.2 建立带表头结点的单循环链表功能的实现

3.2.1 建立带表头结点的单循环链表功能实现思路

建立带表头结点的单循环链表功能的实现思路为：

(1)构造函数主要用于创建一个空的循环链表，初始化表头结点。在构造函数中，首先分配内存以创建表头结点（first），然后将 first 的 link 指针指向自己，以表示一个空链表。同时，last 指针也指向 first，以方便后续的插入操作；

(2)insert 函数用于在链表中插入新节点。对于带有表头结点的单循环链表，插入节点的操作与普通单链表类似，但需要特别处理尾部节点（last），具体步骤如下：

①首先,使用 locate 函数找到要插入的位置,即第 i 个节点的前一个节点，或者如果 i=0，则表示在表头插入；

②创建新节点，并将数据赋给新节点；

③将新节点的 link 指针指向当前位置节点（前一个节点）的 link，以保持链表的连续性；

④更新当前位置节点（前一个节点）的 link，使其指向新节点，完成插入操作；

⑤如果插入操作发生在尾部（即在 last 之后插入），需要更新 last 指针，

将其指向新节点，以确保链表仍然是循环的。

通过构造函数和 insert 函数的协同作用，带有表头结点的单循环链表得以创建。

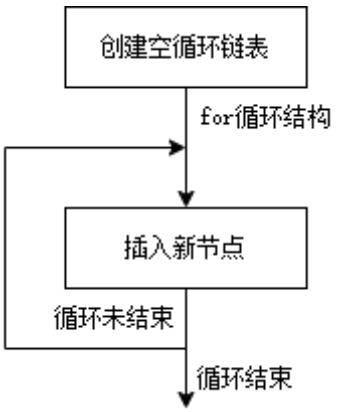

图 3.2.1.1 建立带表头结点的单循环链表功能实现流程图

3.2.2 建立带表头结点的单循环链表功能核心代码

```
/* Initialize a circular linked list */
MyCircList<int> circList;
for (int count = 0; count < N; count++)
    circList.insert(count, order[count]);
```

3.2.3 建立带表头结点的单循环链表功能示例

在上述建立带表头结点的单循环链表功能核心代码后插入语句 system("pause")，在 Microsoft Visual Studio 2022 中使用调试工具在第 483 行（见图 3.2.3.1）设置断点，并开始调试。

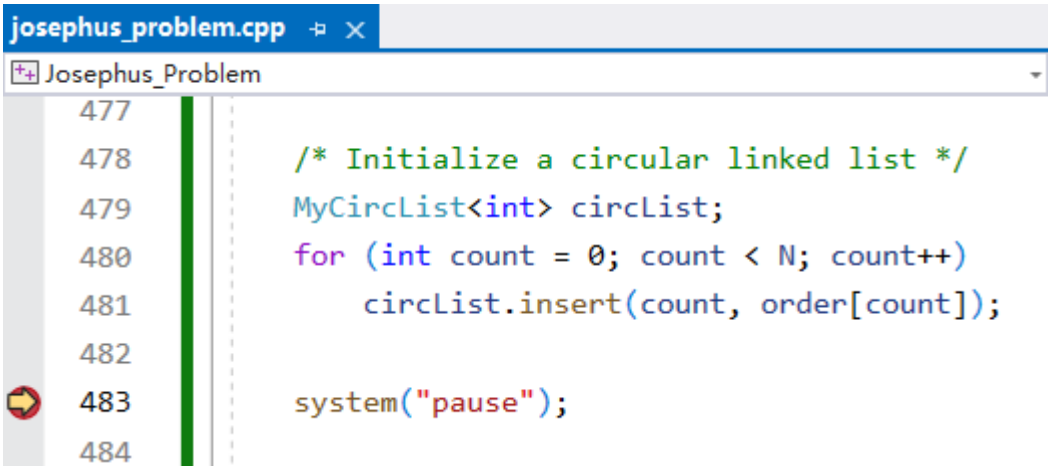

图 3.2.3.1 使用调试工具设置断点并开始调试

在“自动窗口”中，可以查看到对象 circList 中头指针和尾指针的地址和其所指向的值，观察发现 first 和 last 指针指向下一节点的 link 指针和其所

指向的值呈现循环结构，即带表头结点的单循环链表建立成功。

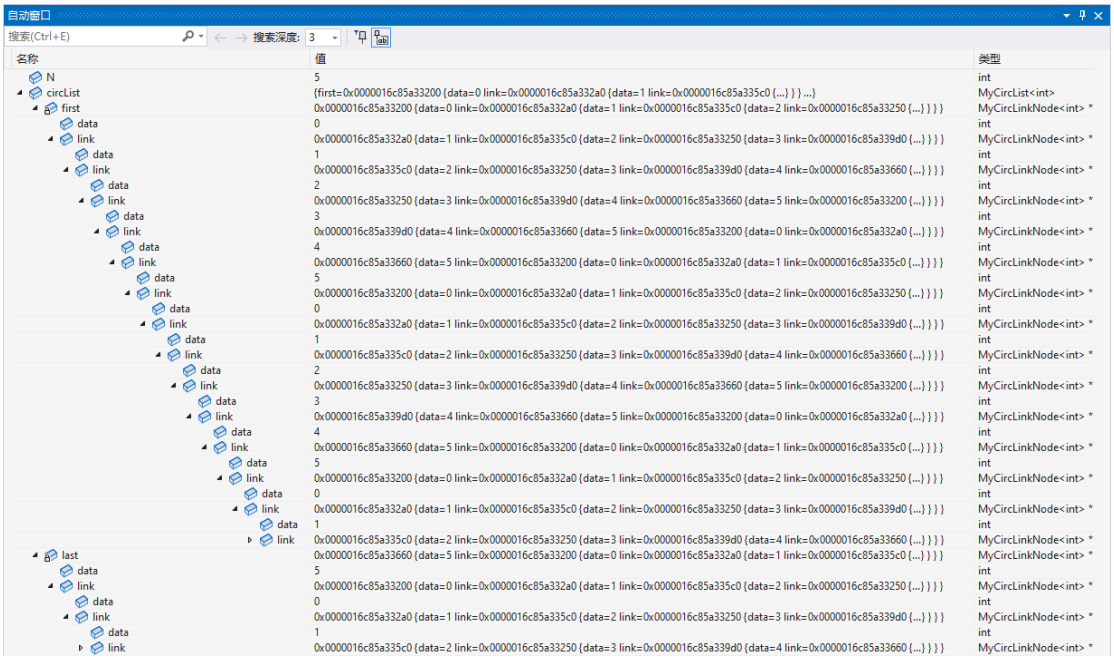

图 3.2.3.2 建立带表头结点的单循环链表功能示例

3.3 游戏核心逻辑功能的实现

3.3.1 游戏核心逻辑功能实现思路

游戏核心逻辑功能的实现思路为：

(1)初始化游戏参数和数据结构：remaining 表示剩余参与游戏的人数，初始值为总人数 N。currentPos 表示当前游戏的位置，初始值为起始位置 S。currentNode 是一个指针，用于表示当前游戏的位置节点。通过 circList.locate(currentPos)找到初始的位置节点；

(2)输出游戏开始的提示信息；

(3)进入循环，只有当剩余人数大于剩余人数目标 K 时才进行游戏，这个循环负责一轮游戏，即按照间隔 M 逐个淘汰人；

(4)在游戏的每一轮中，进行如下步骤：

①使用 for 循环，计数变量为 count，从 1 开始循环计数。如果 count 小于间隔 M，继续向前移动；

②将当前位置节点 currentNode 向后移动，即指向下一个位置。如果当前位置节点已经是链表的尾部，它将绕回到链表的头部；

③更新当前位置 currentPos，以确保它仍在合法范围内。如果当前位置超出了剩余人数范围，它会回绕到 1；

④再次将当前位置节点 currentNode 向后移动一个位置，即指向下一个要淘汰的人；

⑤调用链表的 remove 函数，将当前位置的人从链表中移除，并将其位置存储在 eliminated 中；

⑥输出当前轮游戏淘汰的人的位置，并将 remaining 减一以表示剩余人数；

(5)输出游戏结束的提示信息，显示剩余人数；

(6)如果游戏结束后还有剩余人，输出剩余人的位置；

(7)释放动态分配的内存。

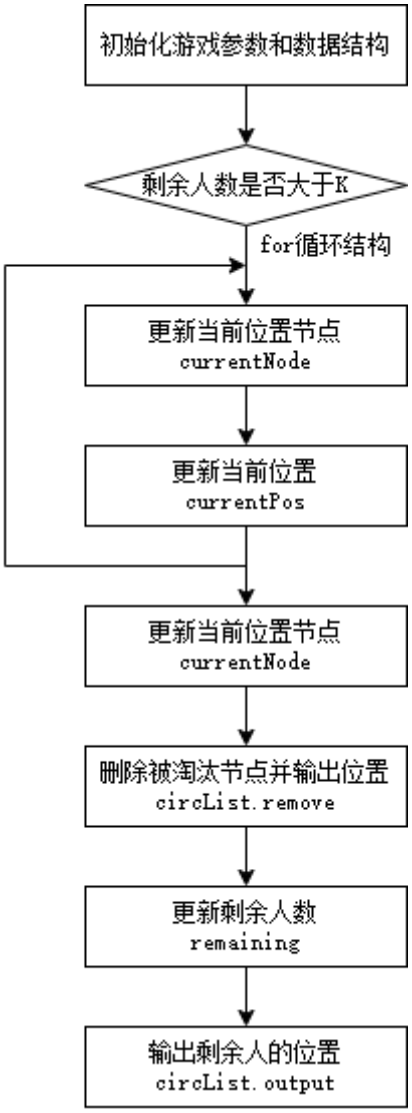

图 3.3.1.1 游戏核心逻辑功能实现流程图

3.3.2 游戏核心逻辑功能核心代码

```
/* Perform Josephus problem */
int remaining = N, currentPos = S, eliminated;
MyCircLinkNode<int>* currentNode = circList.locate(currentPos);
std::cout << std::endl << ">>> 游戏开始" << std::endl << std::endl;
while (remaining > K) {
    for (int count = 1; count < M; count++) {
```

```
            currentNode = currentNode->link;
            if (currentNode == circList.getHead())
                count--;
        }
        currentPos = (currentPos + M - 1) % remaining;
        if (currentPos == 0)
            currentPos = remaining;
        currentNode = currentNode->link;
        if (currentNode == circList.getHead())
            currentNode = currentNode->link;
        circList.remove(currentPos, eliminated);
        std::cout << "第 " << std::setw(numDigits) << N - (--remaining)
<< " 个淘汰的人的位置: " << eliminated << std::endl;
    }
    std::cout << std::endl << ">>> 游戏结束（剩余人数: " << remaining << "）
" << std::endl << std::endl;
    if (remaining > 0) {
        std::cout << "剩余人的位置为: ";
        circList.output();
        std::cout << std::endl << std::endl;
    }

    /* Free dynamic memory */
    delete[] order;
```

3.3.3 游戏核心逻辑功能示例

```
>>> 游戏开始

第  1 个淘汰的人的位置: 9
第  2 个淘汰的人的位置: 18
第  3 个淘汰的人的位置: 27
第  4 个淘汰的人的位置: 6
第  5 个淘汰的人的位置: 16
第  6 个淘汰的人的位置: 26
第  7 个淘汰的人的位置: 7
第  8 个淘汰的人的位置: 19
第  9 个淘汰的人的位置: 30
第 10 个淘汰的人的位置: 12
第 11 个淘汰的人的位置: 24
第 12 个淘汰的人的位置: 8
第 13 个淘汰的人的位置: 22
第 14 个淘汰的人的位置: 5
第 15 个淘汰的人的位置: 23

>>> 游戏结束（剩余人数: 15）

剩余人的位置为: 1 2 3 4 10 11 13 14 15 17 20 21 25 28 29
```

图 3.3.3.1 游戏核心逻辑功能示例

3.4 异常处理功能的实现

3.4.1 动态内存申请失败的异常处理

在进行 MyCircLinkNode 类等的动态内存申请时，程序使用 new(std::nothrow)来尝试分配内存。new(std::nothrow)在分配内存失败时不会引发异常，而是返回一个空指针（NULL 或 nullptr），代码检查指针是否为空指针，如果为空指针，意味着内存分配失败，这时程序将执行以下操作：

(1) 向标准错误流 std::cerr 输出一条错误消息"Error: Memory allocation failed."，指出内存分配失败；

(2)调用 exit 函数，返回错误码 MEMORY_ALLOCATION_ERROR（通过宏定义方式定义为-1），用于指示内存分配错误，并导致程序退出。

下面是动态内存申请的异常处理的一个代码示例：

```
template <typename Type>
MyCircList<Type>::MyCircList()
{
    first = new(std::nothrow) MyCircLinkNode<Type>;
    if (first == NULL) {
        std::cerr << "Error: Memory allocation failed." << std::endl;
        exit(MEMORY_ALLOCATION_ERROR);
    }
    first->link = first;
    last = first;
}
```

3.4.2 MyCircList 类的异常处理

3.4.2.1 MyCircList 类索引越界的异常处理

在调用 MyCircList 类的 getData，setData，insert，remove 等成员函数时，程序会检查传入的索引参数 i 进行检查。若索引越界，函数返回 false，表示操作失败（索引无效）。

在调用 MyCircList 类的 locate 函数时，程序会检查传入的索引参数 i 进行检查。若索引越界，函数返回 NULL，表示操作失败（索引无效）。

3.4.2.2 MyCircList 类自赋值的异常处理

MyCircList 类重载了赋值运算符，用于将一个循环链表赋值给另一个循环链表。自赋值是一种常见的错误操作，可能导致资源泄漏和其他问题，该运算符重载函数对自赋值（将循环链表赋值给自身）的情况进行了处理。

程序会检查 this 指针和传入的 L 循环链表对象是否相同，如果 this 指针与 L 相同，表示自赋值操作，为了防止自赋值，运算符重载函数不会执行赋值操

作，而是直接返回当前对象的引用*this。这个错误处理机制确保了当尝试将循环链表赋值给自身时，不会导致资源泄漏或其他问题。

3.4.3 输入非法的异常处理

程序通过调用 inputInteger 函数输入总人数、起始位置、间隔人数、剩余人数。

inputInteger 函数用于获取用户输入的整数，同时限制输入必须在指定的范围内，函数的代码如下：

```
int inputInteger(int lowerLimit, int upperLimit, const char* prompt)
{
    while (true) {
        std::cout << "请输入" << prompt << "[整数范围: " << lowerLimit
<< "~" << upperLimit << "]: ";
        double tempInput;
        std::cin >> tempInput;
        if (std::cin.good() && tempInput == static_cast<int>(tempInput)
&& tempInput >= lowerLimit && tempInput <= upperLimit) {
            std::cin.clear();

std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
            return static_cast<int>(tempInput);
        }
        else {
            std::cerr << std::endl << ">>> " << prompt << "输入不合法，
请重新输入" << prompt << "! " << std::endl << std::endl;
            std::cin.clear();

std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
        }
    }
}
```

inputInteger 函数对输入非法的情况进行了处理，代码具体执行逻辑如下：

⑴进入一个无限循环，它会一直运行直到用户提供有效的输入；

⑵用户的输入被读取到 tempInput 变量中，这里采用 double 类型来接收输入以便后续检查；

⑶进行输入验证：std::cin.good()检查输入流的状态是否正常，确保没有发生数据类型输入错误，tempInput==static_cast<int>(tempInput)检查用户输入是否为整数，通过将其转换为整数再比较，tempInput>=lowerLimit 和 tempInput<=upperLimit 确保输入在指定的范围内；

⑷合法输入处理：如果用户提供了合法的输入，函数会清除输入流的错误

状态，丢弃输入缓冲区中的任何剩余内容，然后返回转换后的整数值；

(5)非法输入处理：如果用户提供的输入不合法，函数会输出错误消息，清除输入流的错误状态，丢弃输入缓冲区中的内容，并继续循环以等待用户提供合法的输入。

4 项目测试

4.1 输入功能测试

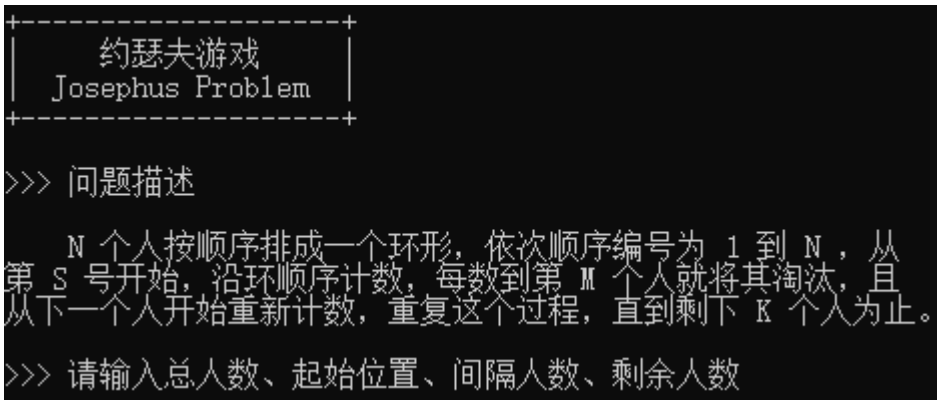

图 4.1.1 输入功能测试

4.1.1 输入总人数功能测试

分别输入超过上下限的整数、浮点数、字符、字符串，可以验证程序对输入非法的情况进行了处理。

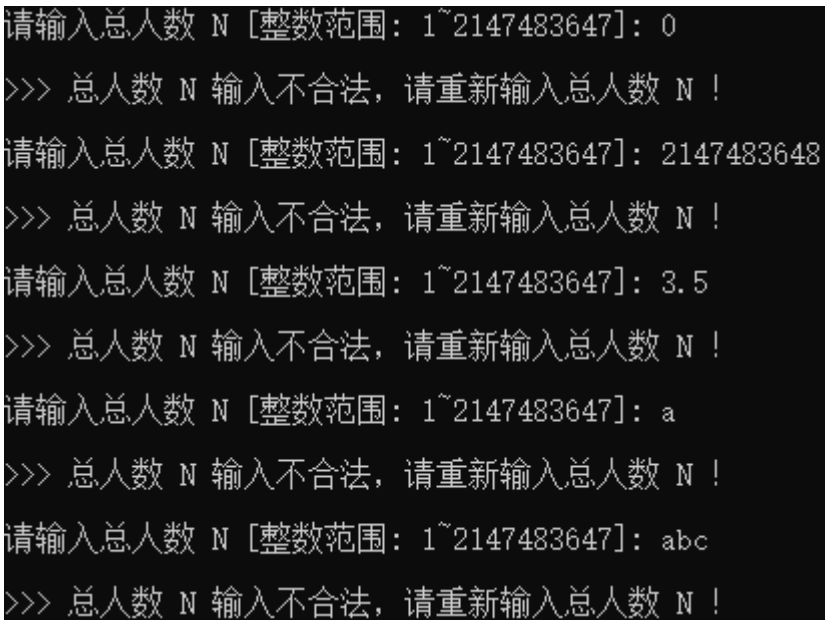

图 4.1.1.1 输入总人数功能测试

当输入合法时，程序继续运行。

4.1.2 输入起始位置功能测试

分别输入超过上下限的整数、浮点数、字符、字符串，可以验证程序对输入非法的情况进行了处理。

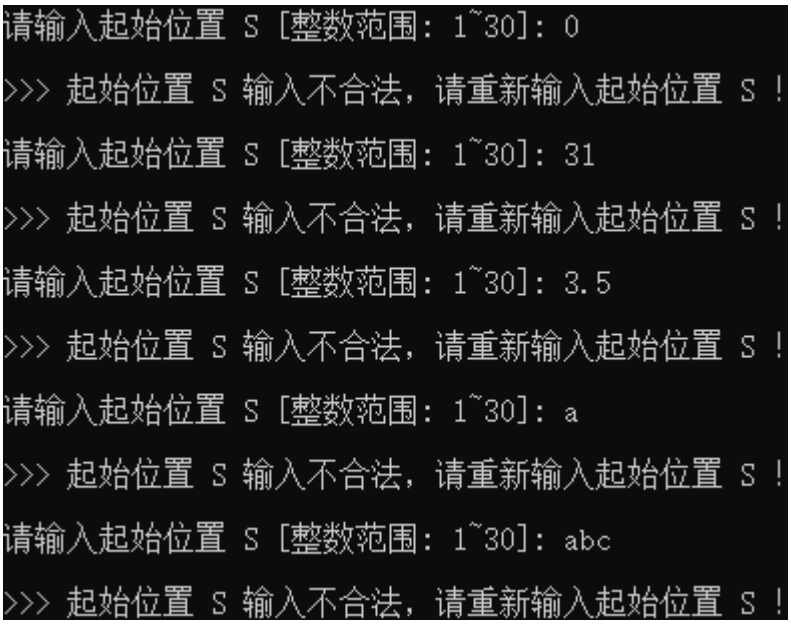

图 4.1.2.1 输入起始位置功能测试

当输入合法时，程序继续运行。

4.1.3 输入间隔人数功能测试

分别输入超过上下限的整数、浮点数、字符、字符串，可以验证程序对输入非法的情况进行了处理。

图 4.1.3.1 输入间隔人数功能测试

当输入合法时，程序继续运行。

4.1.4 输入剩余人数功能测试

分别输入超过上下限的整数、浮点数、字符、字符串，可以验证程序对输入非法的情况进行了处理。

图 4.1.4.1 输入剩余人数功能测试

当输入合法时，程序继续运行。

4.2 建立带表头结点的单循环链表功能测试

建立带表头结点的单循环链表功能测试可以通过在 Microsoft Visual Studio 2022 中使用调试工具设置断点并开始调试的方法进行测试（见图 3.2.3.1）。

在“自动窗口”中，可以查看到对象 `circList` 中头指针和尾指针的地址和其所指向的值，观察发现 `first` 和 `last` 指针指向下一节点的 `link` 指针和其所指向的值呈现循环结构，即带表头结点的单循环链表建立成功（见图 3.2.3.2）。

4.3 游戏核心逻辑功能测试

游戏核心逻辑为：N 个人按顺序排成一个环形，依次顺序编号为 1 到 N，从第 S 号开始，沿环顺序计数，每数到第 M 个人就将其淘汰，且从下一个人开始重新计数，重复这个过程，直到剩下 K 个人为止。

在游戏进行过程中，程序会按照这样的核心逻辑进行：

(1)从指定的序号开始；

(2)沿着链表依次数 M 个人，然后将这个人从链表中删除；

(3)游戏继续，从下一个人开始重新计数，继续进行上述步骤，直到只剩下 K 个人。

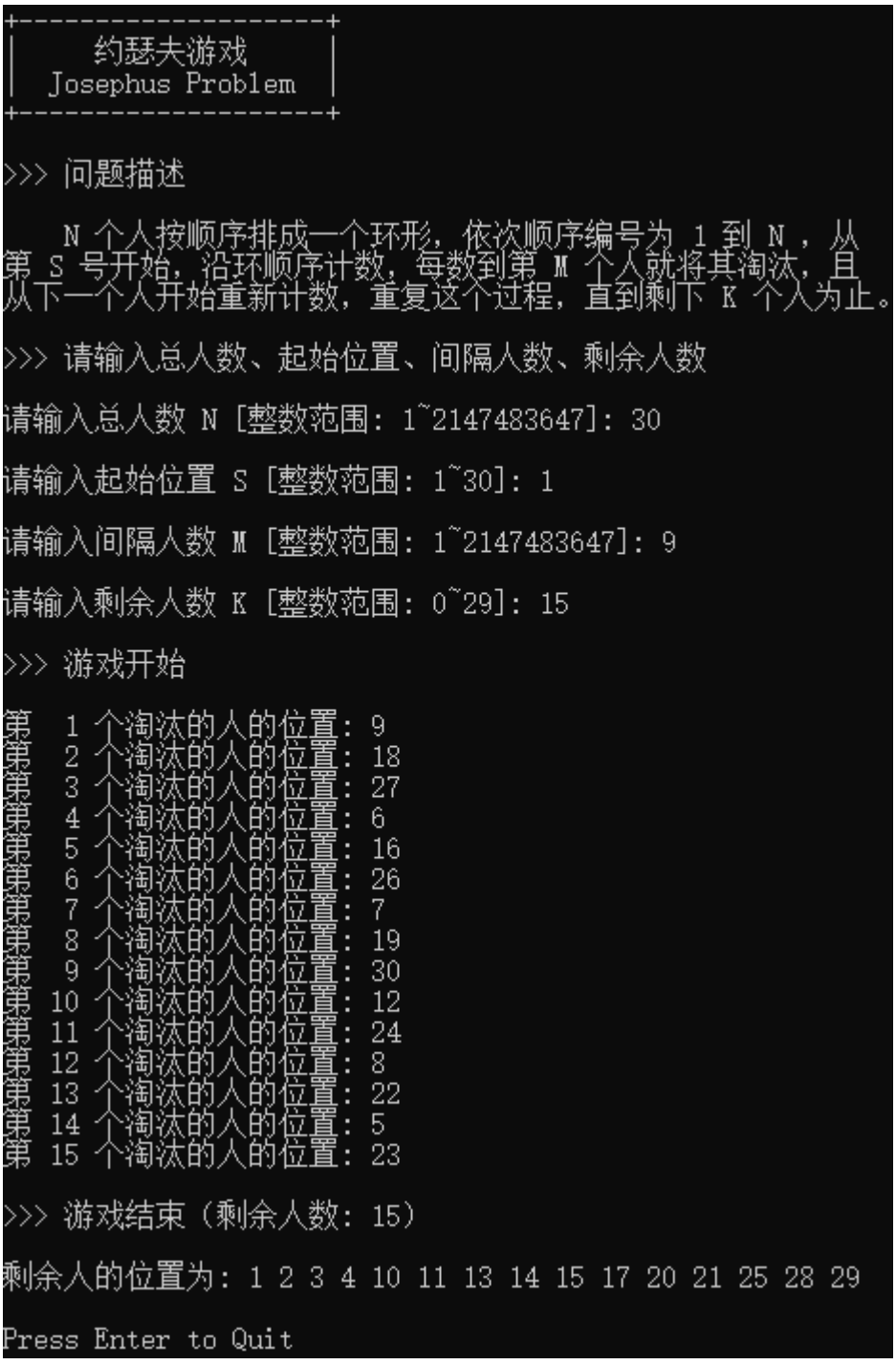

图 4.3.1 游戏核心逻辑功能测试

4.4 退出程序功能测试

为避免直接运行可执行文件在输入完成后会发生闪退的情况，本程序使用如下代码，创建了一个无限循环，等待用户按下回车键，以便退出程序。避免结果

输出结束后程序迅速退出的情况。

```
/* Wait for enter to quit */
std::cout << "Press Enter to Quit" << std::endl;
while (_getch() != '\r')
    continue;
```

5 集成开发环境与编译运行环境

Windows 系统：Windows 11 x64

Windows 集成开发环境：Microsoft Visual Studio 2022 (Release 模式)

Windows 编译运行环境：本项目适用于 x86 架构和 x64 架构

Linux 系统：CentOS 7 x64

Linux 编译命令：

g++ '/root/桌面/Share_Folder/josephus_problem.cpp' -std=c++11 -o '/root/桌面/Share_Folder/josephus_problem' -lncurses

Linux 运行命令：

'/root/桌面/Share_Folder/josephus_problem'

图 5.1 Linux 环境程序运行示例

本项目使用条件编译解决 Windows 系统和 Linux 系统编译环境的差异，示例代码如下。

```
#ifdef _WIN32
#include <conio.h>
#elif __linux__
```

```
#include <ncurses.h>
#endif
```