

# 题目二 约瑟夫生者死者游戏

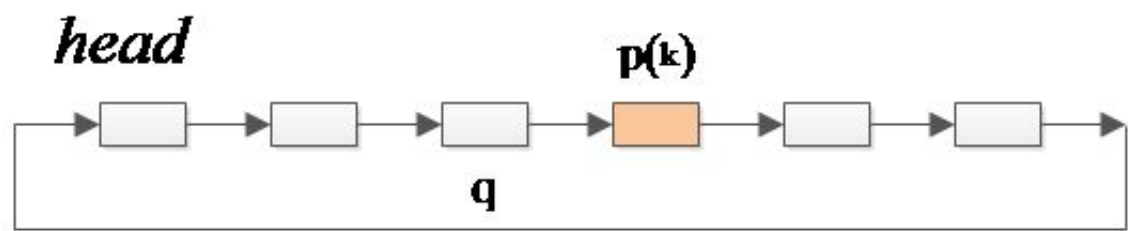
## 一.设计思路

约瑟夫生者死者游戏的要求如下：

- N个旅客排成一个环形，依次顺序编号1, 2, ..., N
- 从某个指定的第S号开始。沿环计数，每数到第M个人就让器出列
- 从下一个人开始重新计数，继续进行下去
- 剩下K个旅客时游戏结束

分析游戏的要求不难发现，我们需要建立一个循环链表。链表中的每一个节点就代表一名乘客，节点的数据域存储乘客的位置编号。我们需要一个指针在链上不断移动，按照游戏规则每次移动相应的步数，指针最终停在哪里就将哪里的乘客杀死，也就是将对应的节点从链表中删除。当杀死足够数量的乘客后就可停止。

其原理可表示如下：



基于这样的思路，我们首先需要建立一个节点类型，代表游戏中的乘客。这个节点需

要像普通单链表一样，具备数据域以及指针域，此外，为便于其他函数使用该结构，还需要定义辅助的构造函数。

此外，我们还需要定义一个循环单链表代表约瑟夫生死环。环中元素的类型就是之前定义的乘客节点。循环单链表同样需要定义相应的构造函数以便于后续操作。并且，我们还需要定义定义内部的移步函数，用于移动指针。还要定义杀人函数，即将环中节点删除的函数。

循环单链表的包括两个成员：一个表示表示单链表的长度，另一个表示单链表的头节点。

## 二.数据结构实现

---

### 1.乘客结点类型（Member）

定义了结构体 `struct Member` 作为循环单链表的结点，即游戏中每个乘客。

#### 1.1类成员

```
int id;           // 位置编号
Member* next;    // 相邻的下一个节点
```

`id` 为 C 语言内置 `int` 类型，代表成员的位置编号。该编号在节点创建时给出，不再改变。

`next` 为 `Member *` 类型，分别代表指向链表中下一个节点的指针。

#### 1.2构造函数

```
Member():next(NULL){}
```

默认构造函数，用于一般新建乘客结点。未初始化乘客位置编号，只将指向邻接节点

的指针置为空。

```
Member(int s){                // 构造函数，传入乘客位置编号
    id = s;
    next = NULL;
}
```

含编号构造函数，传入乘客位置编号进行初始化，同时将后续节点指针置空，免去了后续的赋值过程。

## 2.循环单链表类型（DyingList）

定义了结构体 `struct DyingList` 作为循环单链表类型，即约瑟夫生死环。

### 1.1类成员

```
int id;                // 位置编号
Member* next;          // 相邻的下一个节点
```

`id` 为 C 语言内置 `int` 类型，代表成员的位置编号。该编号在节点创建时给出，不再改变。

`next` 为 `Member *` 类型，分别代表指向链表中下一个节点的指针。

### 1.2构造函数

```
DyingList(){} 
```

默认构造函数，用于建立一个空的约瑟夫生死环，环内没有元素。实际实现中不使用该构造函数。

```
DyingList(int sz);          // 构造函数，传入环的长度
```

含编号构造函数，传入乘客位置编号进行初始化，同时将后续节点指针置空，免去了后续的赋值过程。

### 1.3移步函数

```
Member* step(int n,Member* beg);    // 移步函数，移动指定步数
```

移步函数，传入要移动的步数以及移动起始节点，返回移动后的位置指针。

### 1.4杀人函数

```
Member* kill(Member* target);       // 杀人函数，删除节点
```

杀人函数，即删除函数，传入要删除的节点指针，返回被删节点的下一个节点。

## 3.移步游标（current）

```
Member* current = list.head;
```

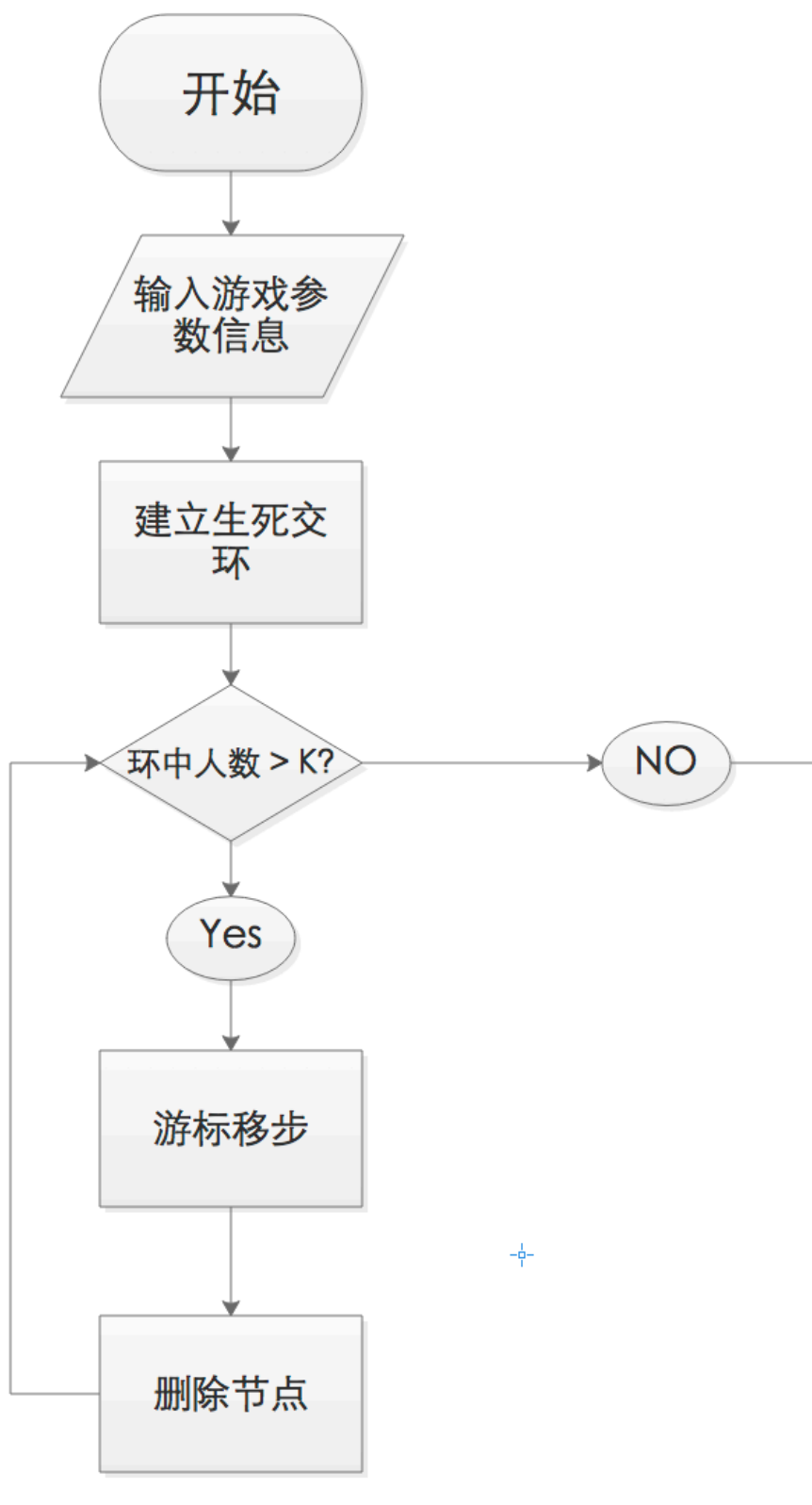
约瑟夫生死环游戏中的遍历游标，根据游戏中输入的移动步数每次向前移动。游标停止位置指向的节点就是被删除的节点位置。

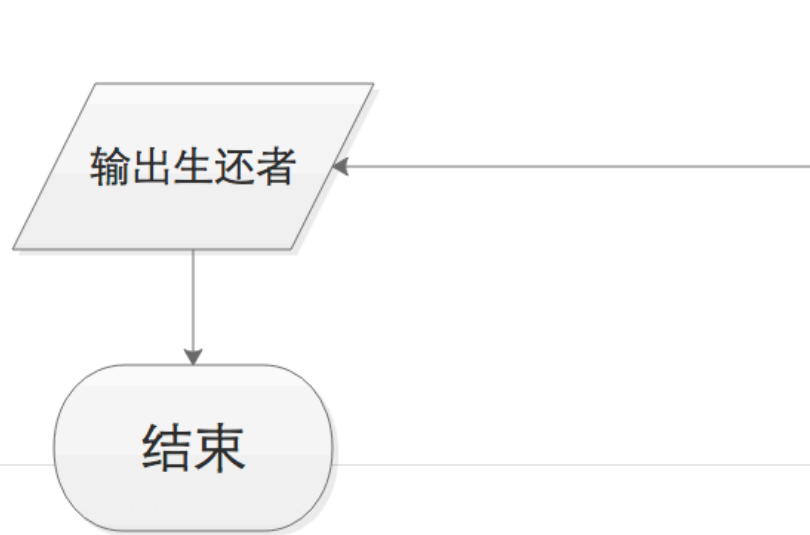
每次删除节点后，游标指向被删除节点的后一个节点。

## 三.系统实现

---

## 1.系统执行框架





首先通过交互信息要求用户输入本次游戏的参数。参数包括：

- 生死游戏的总人数N
- 游戏开始的位置S
- 死亡数字M
- 剩余的生者人数K

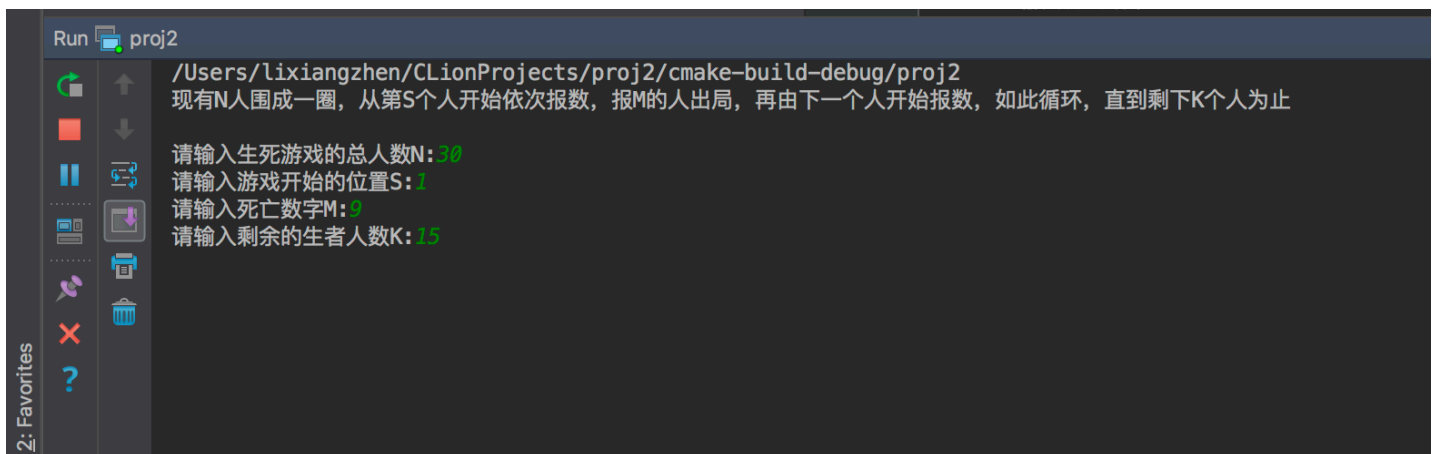
核心代码如下：

```

/*
 * 初始化游戏信息
 * */
cout<<"现有N人围成一圈，从第s个人开始依次报数，报M的人出局，再由下一个人开始报数，如此循环，
直到剩下K个人为止\n\n";
int n,m,s,k;
cout<<"请输入生死游戏的总人数N:";
cin>>n;
cout<<"请输入游戏开始的位置s:";
cin>>s;
cout<<"请输入死亡数字M:";
cin>>m;
cout<<"请输入剩余的生者人数k:";
cin>>k;
cout<<endl;

```

程序执行情况如下：



```

Run proj2
/Users/lixiangzhen/CLionProjects/proj2/cmake-build-debug/proj2
现有N人围成一圈，从第S个人开始依次报数，报M的人出局，再由下一个人开始报数，如此循环，直到剩下K个人为止

请输入生死游戏的总人数N: 30
请输入游戏开始的位置S: 1
请输入死亡数字M: 9
请输入剩余的生者人数K: 15

```

对用户输入数据的合理性进行判断。  
排除游戏人数为0，或者游戏人数过少等非法情况。

核心代码如下：

```
if(n == 0){
    cout<<"游戏人数为0, 非法输入! \n";
    return 0;
}
else if(n < k){
    cout<<"游戏人数过少, 非法输入! \n";
    return 0;
}
else if(n < s){
    cout<<"非法的开始位置! \n";
    return 0;
}
```

然后根据用户输入的生死环游戏总人数 `N` 建立生死环。建立过程调用了

`struct DyingList` 类内部的构造函数 `DyingList(int sz)`

随后进行一系列初始化操作。先建立游标 `Member* current`，其初值即位生死环的头节点 `list.head`

接着根据用户输入的游戏开始位置 `s` 将游标移动到指定的初始位置。移动操作调用了 `struct DyingList` 类内部的移步函数 `Member* step(int n, Member* beg)`

核心代码如下：

```
DyingList list(n);           // 根据输入的乘客数量创建约瑟夫环
Member* current = list.head; // 指针最初指向生死环头节点
current = list.step(s - 1, current); // 根据开始位置的要求跳转到对应位置
```



经过上述的预处理，游戏正式开始。

根据用户输入的死亡数字 `M` 进行游戏。每次将游标移动 `M` 个位置，将游标停留位置的节点删除。

由于每次删除后游标 `current` 都指向被删除节点的下一个位置，因此只需要移动 `M - 1` 个位置。

删除操作调用了 `struct DyingList` 类内部的杀人函数

数 `Member* kill(Member* target)`

移动操作调用了 `struct DyingList` 类内部的移步函数

数 `Member* step(int n, Member* beg)`

核心代码如下：

```
/*
 * 循环执行 n - k 次，每次杀死一个人
 * 最后环中剩余 k 个人
 * */
for(int i = 1; i <= n - k; ++i){
    Member* killed = list.step(m - 1, current);    // 每次跳转 m - 1 个位置
    cout<<"第"<<i<<"个死者的位置是： " <<killed->id<<endl;    // 杀死对应位置的人
    current = list.kill(killed);
}
```

程序执行情况如下：

```
Run proj2
/Users/lixiangzhen/CLionProjects/proj2/cmake-build-debug/proj2
现有N人围成一圈，从第S个人开始依次报数，报M的人出局，再由下一个人开始报数，如此循环，直到剩下K个人为止

请输入生死游戏的总人数N: 30
请输入游戏开始的位置S: 1
请输入死亡数字M: 9
请输入剩余的生者人数K: 15

第1个死者的位置是: 9
第2个死者的位置是: 18
第3个死者的位置是: 27
第4个死者的位置是: 6
第5个死者的位置是: 16
第6个死者的位置是: 26
第7个死者的位置是: 7
第8个死者的位置是: 19
第9个死者的位置是: 30
第10个死者的位置是: 12
第11个死者的位置是: 24
第12个死者的位置是: 8
第13个死者的位置是: 22
第14个死者的位置是: 5
第15个死者的位置是: 23
```

最后，将游戏结果输出。

先输出生者人数

接着一次输出剩余生者的位置编号。

核心代码如下：

```
/*
 * 输出游戏结果
 * */
cout<<"\n最后剩下:   "<<k<<"人\n";

cout<<"剩余生者的位置为:   ";
Member* p = list.head;
for(int i = 0;i < k;++i){           // 循环遍历输出链中剩余的人的位置
    cout<<p->id<<"   ";
    p = p->next;
}
```

程序执行情况如下：

```
Run proj2
/Users/lixiangzhen/CLionProjects/proj2/cmake-build-debug/proj2
现有N人围成一圈，从第S个人开始依次报数，报M的人出局，再由下一个人开始报数，如此循环，直到剩下K个人为止

请输入生死游戏的总人数N: 30
请输入游戏开始的位置S: 1
请输入死亡数字M: 9
请输入剩余的生者人数K: 15

第1个死者的位置是: 9
第2个死者的位置是: 18
第3个死者的位置是: 27
第4个死者的位置是: 6
第5个死者的位置是: 16
第6个死者的位置是: 26
第7个死者的位置是: 7
第8个死者的位置是: 19
第9个死者的位置是: 30
第10个死者的位置是: 12
第11个死者的位置是: 24
第12个死者的位置是: 8
第13个死者的位置是: 22
第14个死者的位置是: 5
第15个死者的位置是: 23

最后剩下: 15人
剩余生者的位置为: 1 2 3 4 10 11 13 14 15 17 20 21 25 28 29
Process finished with exit code 0
```

## 2.生死环建立功能

核心代码如下：

```

/*
 * 生死环构造函数
 * 传入生死环长度
 * 建立对应长度的生死环
 * */
DyingList::DyingList(int sz){
    size = sz;          // 生死环长度赋值
    head = new Member(1);    // 建立头节点
    Member* p = head;

    for(int i = 2; i <= sz; ++i){    // 循环的建立链表节点
        p->next = new Member(i);
        p = p->next;
    }
    p->next = head;
}

```

直接使用 `struct DyingList` 类内部的构造函数 `DyingList(int sz)` 来实现建立单循环链表的功能。

循环链表的长度由参数 `int sz` 传入，函数内部首先建立整个链表的头节点，随后用 `for` 循环不断 `new` 出新的节点，并将其加入到循环链表中。

各个节点的编号在建立节点时就已尽通过 `struct Member` 类内部的构造函数 `Member(int s)` 初始化，其数值就是该节点被建立的顺序。

关联调用情况如下：

```

106      cin >> n;
107      cout<<endl;
108
109      DyingList list(n);          // 根据输入的乘客数量创建约瑟夫环
110      Member* current = list.head;    // 指针最初指向生死环头节点

```

### 3.移步功能

核心代码如下：

```
/*
 * 移步函数
 * 传入要移动的步数以及移动起始节点
 * 返回移动后的位置指针
 * */
Member* DyingList::step(int n, Member* beg){
    for(int i = 0; i < n; ++i){          // 循环移步，每次移动移步
        beg = beg->next;
    }
    return beg;                          // 返回移动后位置
}
```

移步函数被定义为 `struct Member` 类内部的功能函数。

该函数有两个参数：

`int n` 代表游标要移动的步数。

`Member* beg` 代表游标的指针。

进入函数以后，利用 `for` 循环不断将游标移动到它的下一个邻接节点处，直到达到给定移动次数。

函数返回移动后游标指针。

关联调用情况如下：

```

109     DyingList list(n);           // 根据输入的乘各数量创建约瑟夫大环
110     Member* current = list.head; // 指针最初指向生死环头节点
111     current = list.step(s - 1, current); // 根据开始位置的要求跳转到对应位置
112
113     /*
114     * 循环执行 n - k 次，每次杀死一个人

```

程序初始化时第一次调用移步函数，使得游标指向指定起始位置。

```

    */
    for(int i = 1; i <= n - k; ++i){
        Member* killed = list.step(m - 1, current); // 每次跳转 m - 1 个位置
        cout<<"第"<<i<<"个死者的位置是: " <<killed->id<<endl; // 杀死对应位置的人
    }

```

生死环游戏运行过程中，每一轮都调用一次移步函数。

#### 4.杀人功能

核心代码如下：

```

/*
 * 杀人函数
 * 即删除函数
 * 传入要删除的节点指针
 * 返回被删节点的下一个节点
 * */
Member* DyingList::kill(Member* target){
    if(target == head)           // 处理删除头节点的情况
        head = target->next;

    Member* p = head;
    while(p->next != target){      // 寻找被删除节点的前驱节点
        p = p->next;
    }
    p->next = target->next;        // 从链中去除节点
    p = target->next;             // 保留被删节点的下一个节点
    delete target;               // 释放被删空间
    return p;                     // 返回被删节点的下一个
}

```

杀人函数就是循环链表的节点删除函数。

函数有一个参数 `Member * target` 表示被删出的节点的指针。

进入函数首先判断该节点是否为循环链表的头节点，如果是头节点，则将头节点后移一位。

处理好头节点问题后，在链表中找到被删除节点的前驱节点。将前驱节点的下一位指向被删除节点的先一位。

然后把被删除节点占用的内存空间释放。

最后返回被删除节点的下一个节点位置。

关联调用情况如下：

```

    Member* killed = list.step(m - 1, current); // 每次跳转 m - 1 个位置
    cout<<"第"<<i<<"个死者的位置是:   "<<killed->id<<endl; // 杀死对应位置的人
    current = list.kill(killed);
}

/*
 * 输出游戏结果
 */

```

生死环游戏运行过程中，每一轮都调用一次杀人函数，将该轮的出局者删除。

## 四.测试

### 1.基本功能测试

#### 测试用例

生死游戏的总人数N:

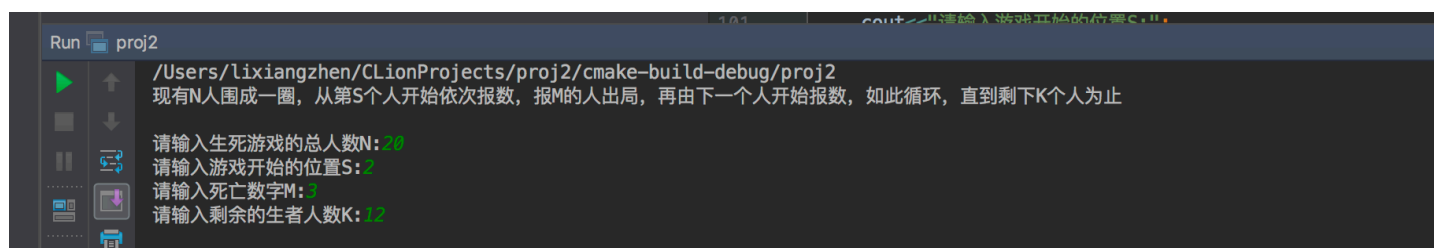
游戏开始的位置S:

死亡数字M:

剩余的生者人数K:

#### 输入游戏参数

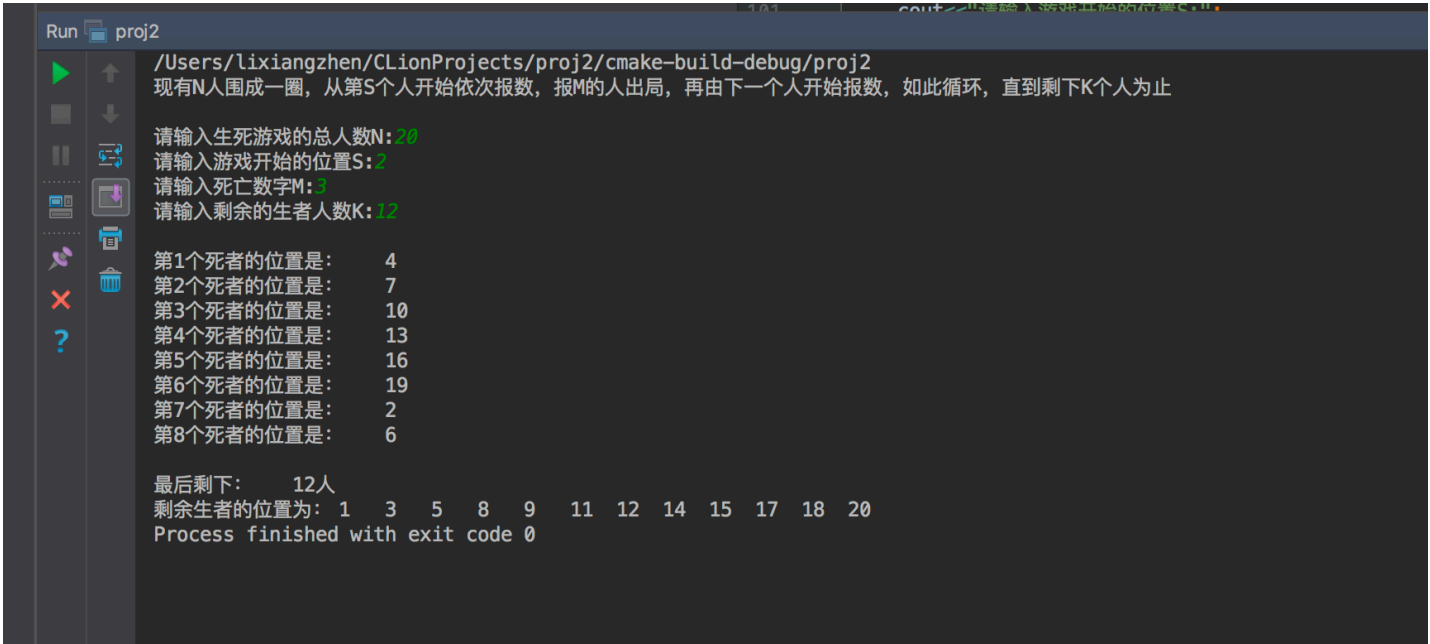
程序执行情况如下：



#### 游戏结果



程序执行情况如下：



2.死亡数字为0

测试用例

生死游戏的总人数N: 15

游戏开始的位置S: 1

死亡数字M: 0

剩余的生者人数K: 8

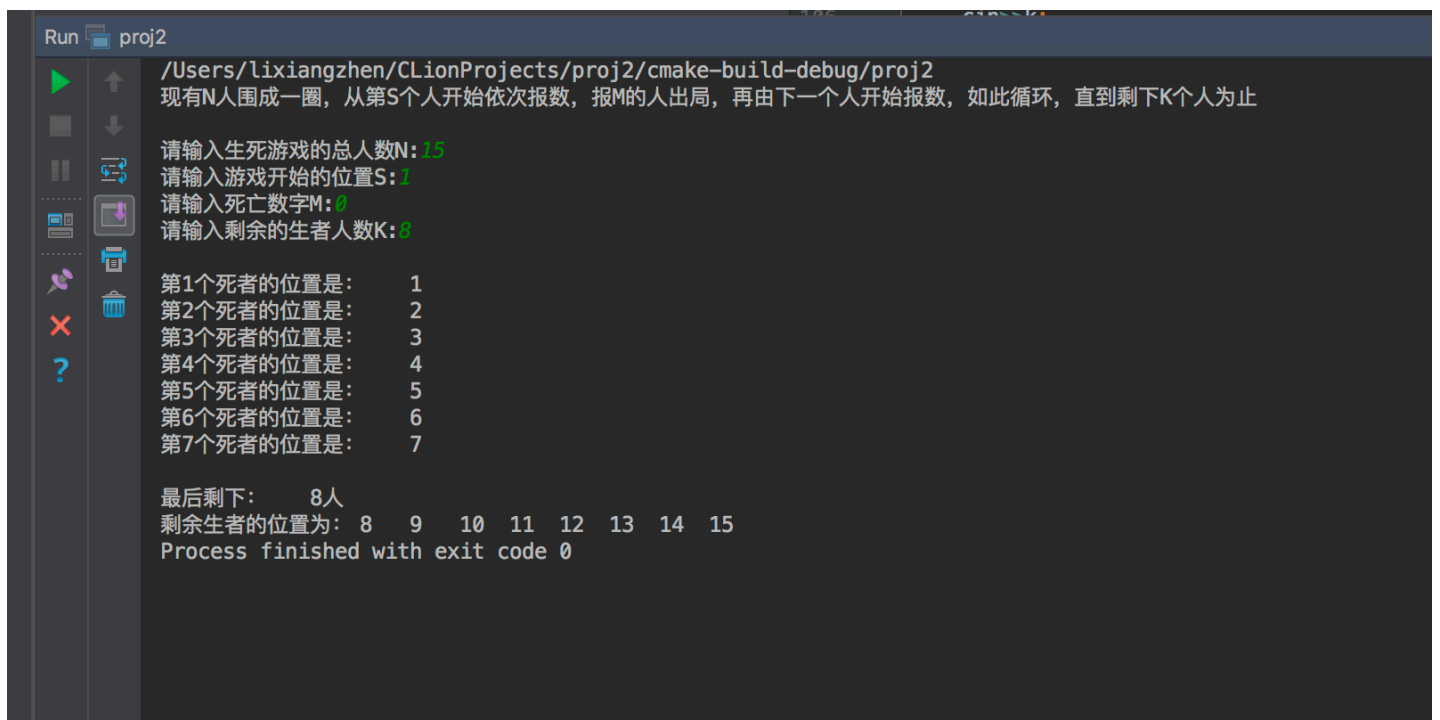
输入游戏参数

程序执行情况如下：



## 游戏结果

程序执行情况如下：



## 2.死亡数字大于游戏总人数

测试用例

生死游戏的总人数N:

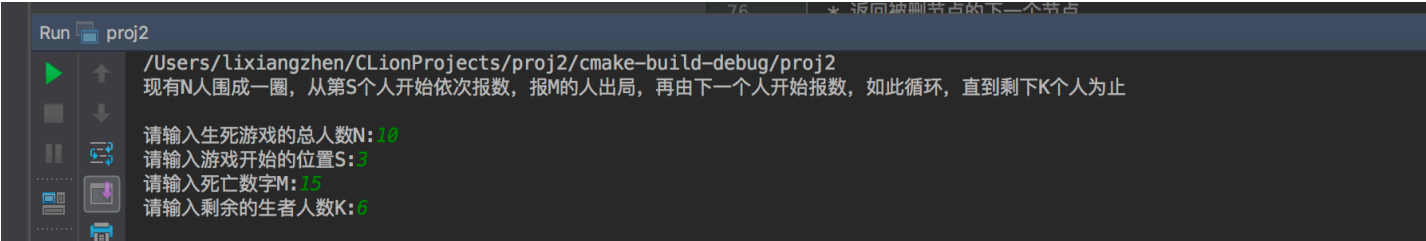
游戏开始的位置S: 3

死亡数字M: 15

剩余的生者人数K: 6

输入游戏参数

程序执行情况如下：



游戏结果

程序执行情况如下：

```
run proj2
/Users/lixiangzhen/CLionProjects/proj2/cmake-build-debug/proj2
现有N人围成一圈，从第S个人开始依次报数，报M的人出局，再由下一个人开始报数，如此循环，直到剩下K个人为止

请输入生死游戏的总人数N: 10
请输入游戏开始的位置S: 3
请输入死亡数字M: 15
请输入剩余的生者人数K: 6

第1个死者的位置是: 7
第2个死者的位置是: 3
第3个死者的位置是: 1
第4个死者的位置是: 2

最后剩下: 6人
剩余生者的位置为: 4 5 6 8 9 10
Process finished with exit code 0
```

### 3.游戏总人数为0

#### 测试用例

生死游戏的总人数N: 0

游戏开始的位置S: 3

死亡数字M: 3

剩余的生者人数K: 6

#### 输入游戏参数

程序执行情况如下：

```
126
127      DyingList list(n); // 根据输入
Run proj2
/Users/lixiangzhen/CLionProjects/proj2/cmake-build-debug/proj2
现有N人围成一圈，从第S个人开始依次报数，报M的人出局，再由下一个人开始报数，如此循环，直到剩下K个人为止

请输入生死游戏的总人数N: 0
请输入游戏开始的位置S: 3
请输入死亡数字M: 3
请输入剩余的生者人数K: 6

游戏人数为0，非法输入！

Process finished with exit code 0
```

## 4.游戏总人数小于生还者人数

### 测试用例

生死游戏的总人数N: 10

游戏开始的位置S: 1

死亡数字M: 3

剩余的生者人数K: 23

### 输入游戏参数

程序执行情况如下：

```
126
127
DyingList list(n); // 根据输入的乘数

Run proj2
/Users/lixiangzhen/CLionProjects/proj2/cmake-build-debug/proj2
现有N人围成一圈，从第S个人开始依次报数，报M的人出局，再由下一个人开始报数，如此循环，直到剩下K个人为止

请输入生死游戏的总人数N: 10
请输入游戏开始的位置S: 1
请输入死亡数字M: 3
请输入剩余的生者人数K: 23

游戏人数过少，非法输入！

Process finished with exit code 0
```

## 5.游戏开始位置大于总人数

### 测试用例

生死游戏的总人数N: 10

游戏开始的位置S: 50

死亡数字M: 4

剩余的生者人数K: 2

### 输入游戏参数

程序执行情况如下：

```
Run proj2
/Users/lixiangzhen/CLionProjects/proj2/cmake-build-debug/proj2
现有N人围成一圈，从第S个人开始依次报数，报M的人出局，再由下一个人开始报数，如此循环，直到剩下K个人为止

请输入生死游戏的总人数N: 10
请输入游戏开始的位置S: 50
请输入死亡数字M: 4
请输入剩余的生者人数K: 2

非法的开始位置!

Process finished with exit code 0
```