# 学习档案（文睿杰）

目录：

A 前言(先看这)

B Level1-1

C Level1-2

D Level1-3

E Level2-1

F Level2-2(普通版)

G Level2-2(工程版)

A 前言：

0、已做完所有题目，Level2\_2【有两个版本，一个是普通版，还有一个是文件夹工程版（第一次尝试工程形式，做得不太好）】是最完整和完善的版本，也是最终版本，拥有较完善的错误输入检定系统，比较有参考价值。

1、这份文档是按照秋招题目中级别由低到高排版的，这也是我做题目的顺序。我在这份文档中详细记录了我在完成各个级别的任务时，所采用的设计思路、遇到的疑难点、以及收获（每个级别不一定三者同时都写了）。

2、设计思路用蓝体标明，疑难用红体标明。

3、我的c语言文件是用vs code编写的，文件的命名规则例如：Levelx\_y\_z.c. , 表示这是级别Levelx-y的第z份文件（对于同一级别任务，我可能会有多份文件，以此来记录我的代码迭代升级的过程）

4、在我的文件Level2\_1\_4(Final).c 之前，很多printf函数里面的内容我都是用字母代码省略的而没有补充完整。

5、由于所有的级别大体上是彼此相承，逐渐升级的，因此在低级别的任务中你可能会发现我也许会有许多地方编写的不完美、对错误的检定不够彻底仔细。不过在后续的代码中我也是不断发现错误并且逐渐解决了问题、完善错误检定过程，优化代码。

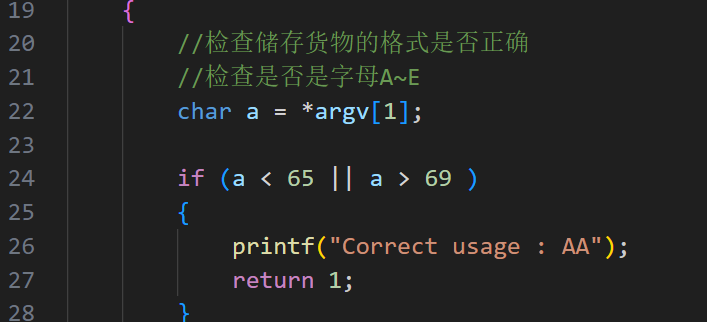
B Level1-1

设计思路

(1)考虑到该level级别只用储存一种货物在一种通道，且无需重复购买，那么最好让用户第一次输入是储存货物而不是购买货物，于是我使用了argc与argv而不是int main(void),并写下部分代码来对用户的第一次输入进行限定。既然程序的第一步应该是对用户的输入进行检定，并判断其是否符合储存货物的格式，于是有了下面的代码：

<Level1_1_1.c>

(2)如下图。首先，考虑到储存货物的格式为 ./xxx A 1 2 3，那么要如何判断argv[1]为大写字母呢？ 我联想到 在ASC II表中A~Z都对应一个数字，于是我选择用数字大小来作判断，但实际上argv[?]是一个指针，因此我要先取出其指向的字母，所以有了下图第22行的代码



图L1-1 (1)

(3)如下图。 接着，我需要检定argv[2]、argv[3]、argv[4]是否为整数。

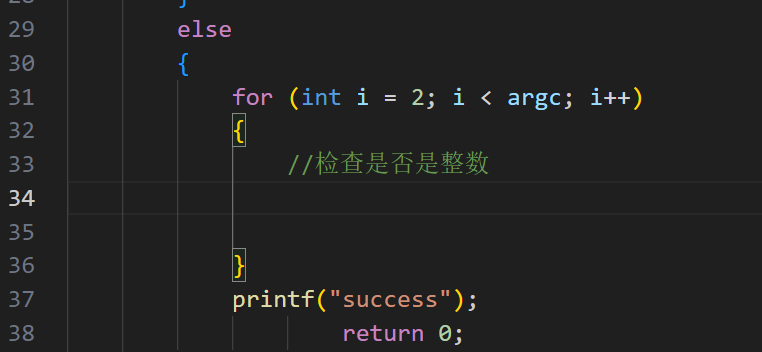


图 L 1-1 (2)

难点 1-1-1

但是在如何检定其是否为整数上我有了疑问，通过上网搜索，发现ctype.h中的isdigit函数可以完成该任务(若为整数，则返回非零值；否则返回0，于是代码变成下图所示

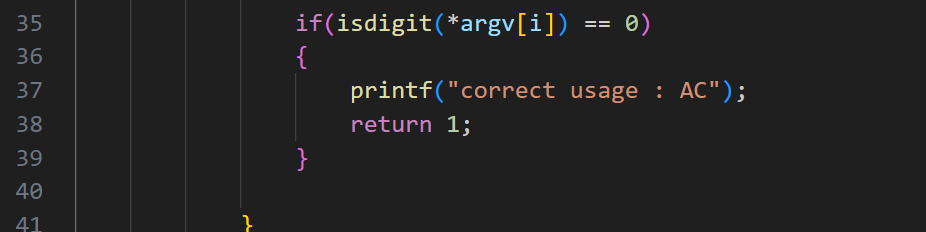


图 L 1-1 (3)

设计思路

1. 之后还要依次判断输入的argv大小是否在给定区间以及输出货物情况，这一块我没有什么疑惑，完成后的部分如下部分链接所示：

<Level1_1_2.c>

最后的完整版：

<Level1_1_3(Final).c>

C LEVEL 1-2

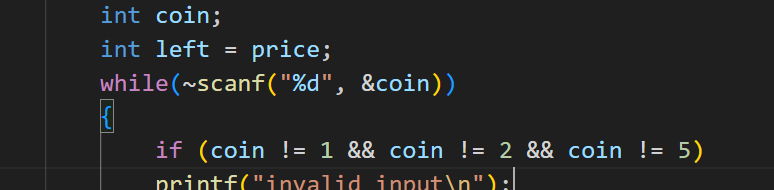
（投币部分）

难点1-2-1

在这一部分，对我个人而言最难的是如何反复监测用户的输入。

从最开始的储存货物，到输入总价以确定购买的商品种类，再到反复投币，让如何记录下用户的多次输入成为了一个难题。

事实上，在做这个任务之前，我只处理过只需要用户输入一次的任务。为了解//AAAAAAA决这个难题，我询问了高年级的学长，并学习到将while和scanf结合起来便可，如下图所示



完整版LEVEL 1-2代码如下链接所示：

<Level1_2_1(Final).c>

D LEVEL 1-3

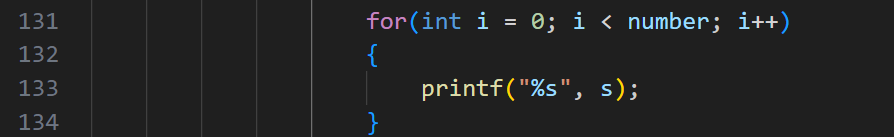
设计思路

1. 在1-2的项目中，用户付款需要反复投币，因此有了个监测投币的while循环。而这个1-3中要求货物售空前可以重复购买，那就意味着要在投币的循环外再套一个监测是否要购买货物的while循环。
2. 既然要反复监测用户是否发出购买货物的指令，那就要反复检查用户的输入是否符合格式和要求，对不合理的输入给出合适的处理和提示。
3. 投币环节只需对1-2的投币环节稍作修改即可。

Level1-3的完整代码如下所示

<Level1_3_1(Final).c>

疑难点1-3-1



在第133行的printf中，我在考虑用&s、\*s还是s 。

为什么会有这种疑虑呢？

1、例如，对于 int n = 1；

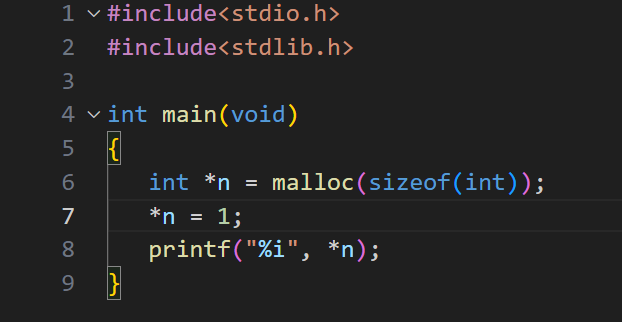
使用printf(“%i”, n);来输出1

使用printf(“%i”, &n);来输出 n的地址；

使用printf(“%i”, \*n)；将直接报错；

这是因为n是一个整数，而不是一个代表着地址的指针

2、又如 int \*n = malloc(sizeof(int));此时n则是一个代表着地址的指针。



上图将输出1，

若将第八行的\*删去将输出n代表的地址，

若使用&n将报错。

综合以上两种情况，我一开始认为string也符合上述规则，即对于char \* s = ABCD，

s作为一个地址，需使用printf(“%s”, \*s)来输出ABCD，但结果是只输出了A,经上网查证学习以及自己敲代码实验，我认识到 对于所谓的char \*s = ABCDEFG... ，无论后面的字符串有多长，s指针只返回了字符串中第一个字符的地址，而由于字符串在计算机内部所占用的memory空间是连续的，只要使用了%s，计算机会自动往后读取所有的字符，直到遇到哨位值’\0’（代表着字符串的结束）才停止。

因此若使用printf(“%s”, \*s)，则只会对字符串中第一个字符进行dereference，从而导致与%s矛盾，是系统无响应

若使用(“%c”, \*s)则只输出A；

使用(“%i”, &s)则只输出A的地址；

综上，我选择使用(“%s”, s)输出完整的字符串。

当然，事实上，这里的s只需指向一个字符字母即货道的种类，完全可以使用(“%c”, s)来输出货道种类。但是对这些东西深入探究后确实收获满满。

E LEVEL 2-1

序言：

Level2-1我在军训期间零零碎碎做了6天才做完。做这一部分的确带给我很多很多体悟和收获，very much！！！

我可以自信的说我的2-1完成的很好！

正如学习文档的前言和我的个人简历上所写，我自学的是一门注重实践的国外课程（没有过多的系统学习所有知识），并且我并没有c语言的书籍，所以对一些比较基础的知识可能没接触过（比如函数 gets 和 goto，这都是我通过和同学相互探讨，上网搜索时学到的，或许这些函数都是很基础的函数）。因此在2-1中，为了完成这个庞大的工程，我一边学习新的函数一边整理思路来设计程序，加上程序大致成型后的后期优化，再加上军训期间每天学习时间实在有限，所以耗费了较长时间在这2-1，不过收获的知识和经验也是相当多的啦！

这一部分的难点我觉得其实也就是设计程序时的思路和逻辑以及对用户输入的检定和错误输入的排查。

该程序大致分为三个环节： 储存货物——选购货物——投币

设计思路（详细）

1. 【储存货物】在最开始，我打算使用while（scanf...）来扫描用户储存货物的指令，这需要使用到%c %d %d %d。然后在设计以输入END来跳转到下一阶段时，我发现我新学的gets函数会更方便（所有扫描到的内容全部录为同一类型 char，可用atoi来将char转化为数字，且用户输入可以无上限），于是我使用while（gets...）来检查用户是否输入END。于是有了下面的代码：

<Level2_1_1.c>

这个时候，我发现，既然对于判断是否输入了END，用gets会很方便，那么我是否可以用gets来处理用户储存货物的指令呢？答案是肯定的，并且我可以将这两者合并，这样极大简化了代码。

于是有了：

<Level2_1_2.c>

正如Level2\_1\_2的代码所示，我统一使用了gets来检验输入。同时我设计了重复储存货物的功能，我认为这也是储存货物这一环节最难的部分，因为

1、我要检查每一次的输入是否规范，数字大小是否在给定区间内......

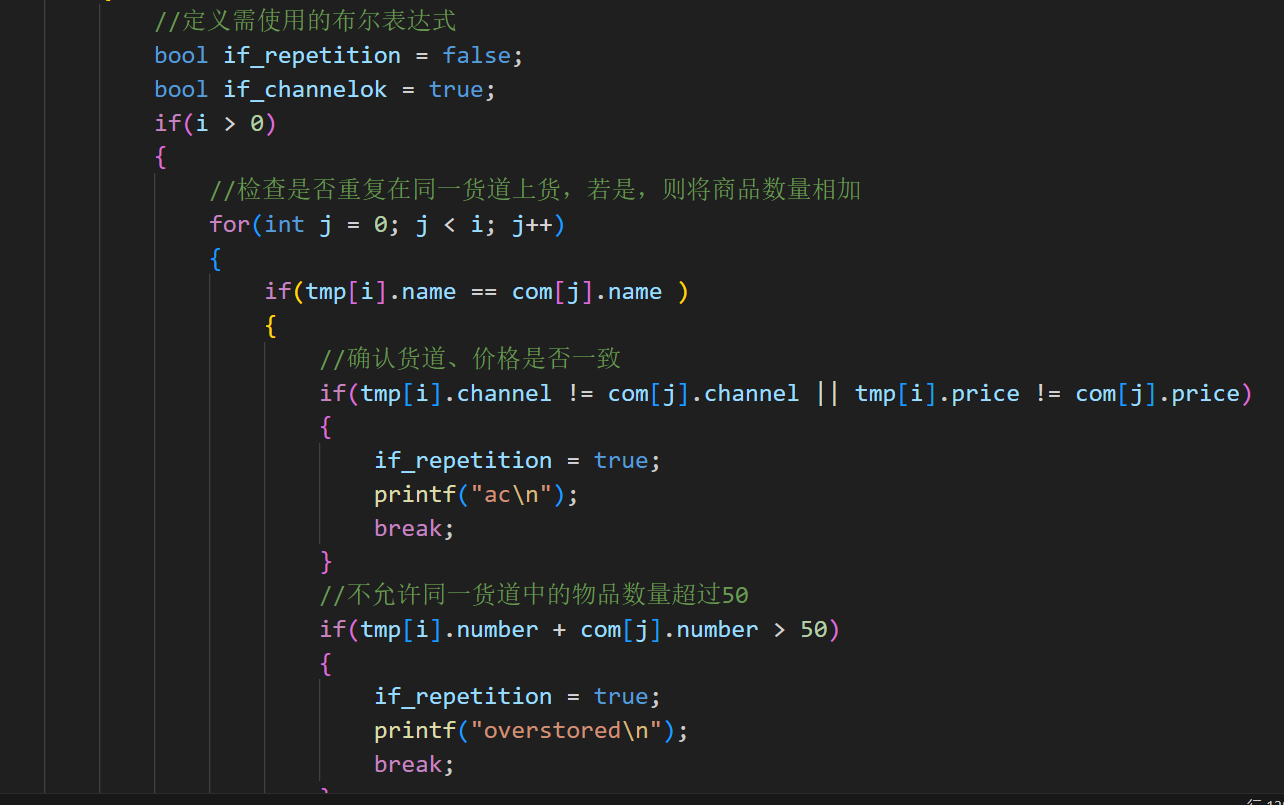
2、重复储存时，我需要考虑前后输入的货物名称是否重复；

3、若重复，那么后面输入的该货物的货道和价格就必须和前面的一致，否则报错；

4、若前后输入的货物名称、货道、价格全都一致，那么将两者的数量叠加，但不允许超过50，否则报错；

5、货物的种类（货道）不允许超过5种A~E，而不是设计成只允许用户输入5次。

下面的截图便展示了进行错误检定的代码（部分）



1. 【选购货物】之后开始设计选择要购买的货物的部分。考虑到储存和选购货物的输入格式较相似，我在这个环节同样使用了while（gets）

在这个环节，设计程序时我认为需要注意：

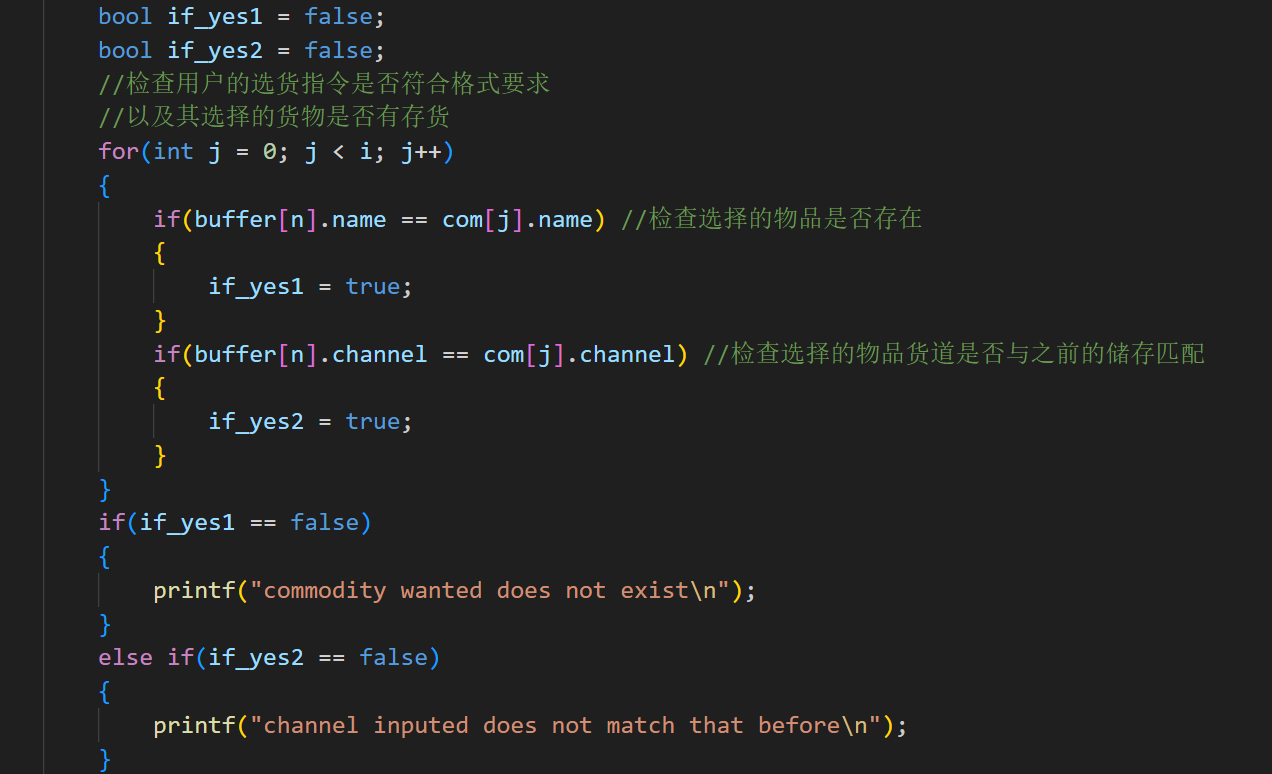
1、用户输入的格式、数字大小正确

2、用户选购的货物必须在上个环节已被储存，否则报错

3、用户输入的某种货物的货道必须吻合上一环节所自定义的货道序号，且用户购买的数量不得大于其储存的量，否则报错

4、若成功选购（且后续的购买成功），则库存相应的减少

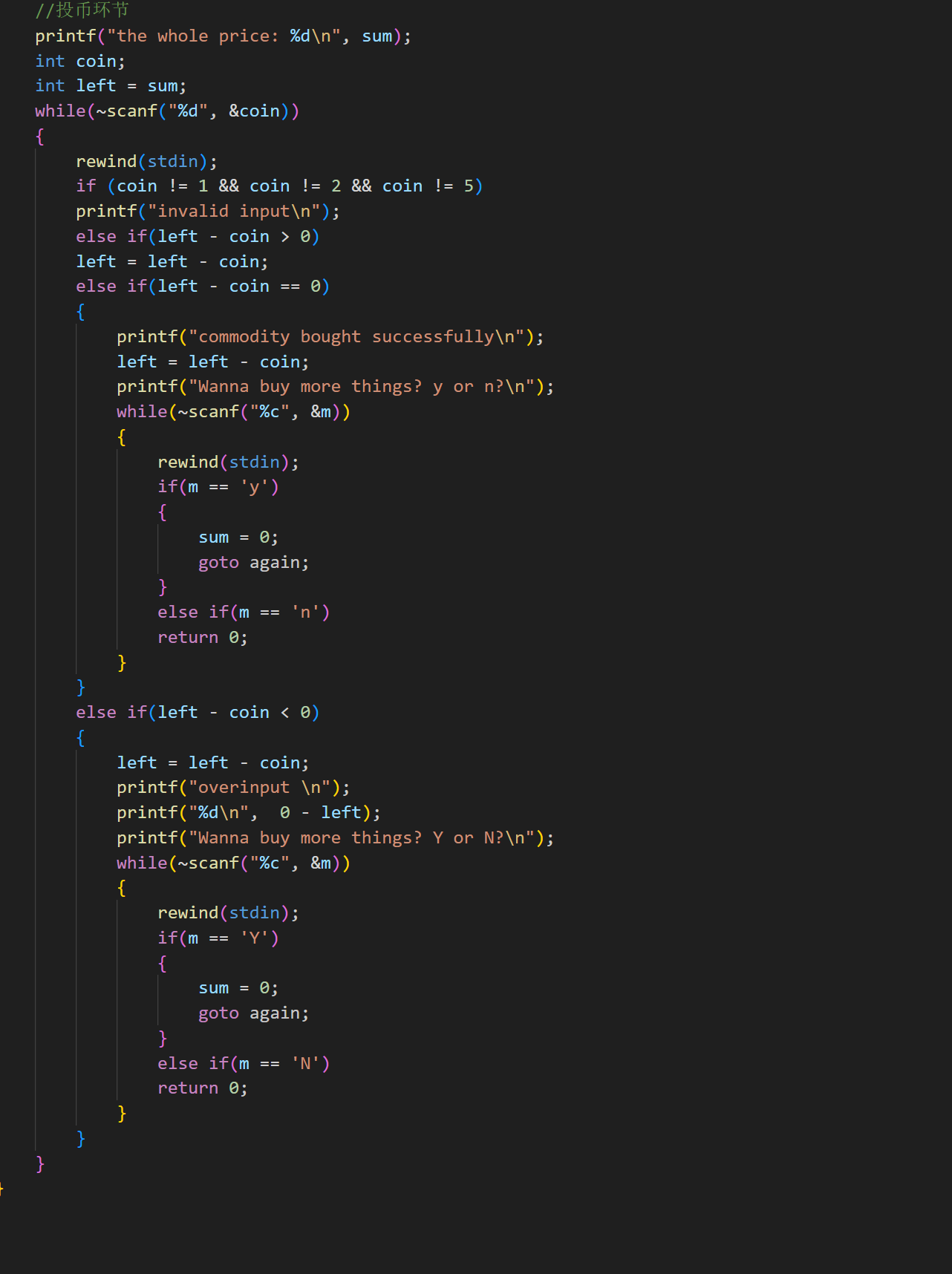
部分功能实现如截图所示



于是有了

<Level2_1_3.c>

1. 【投币】这一部分没有什么难点，可以直接参考Level1-?中的投币部分，稍作调整即可，比如总价格需要把所有选择的货物的价格叠加。

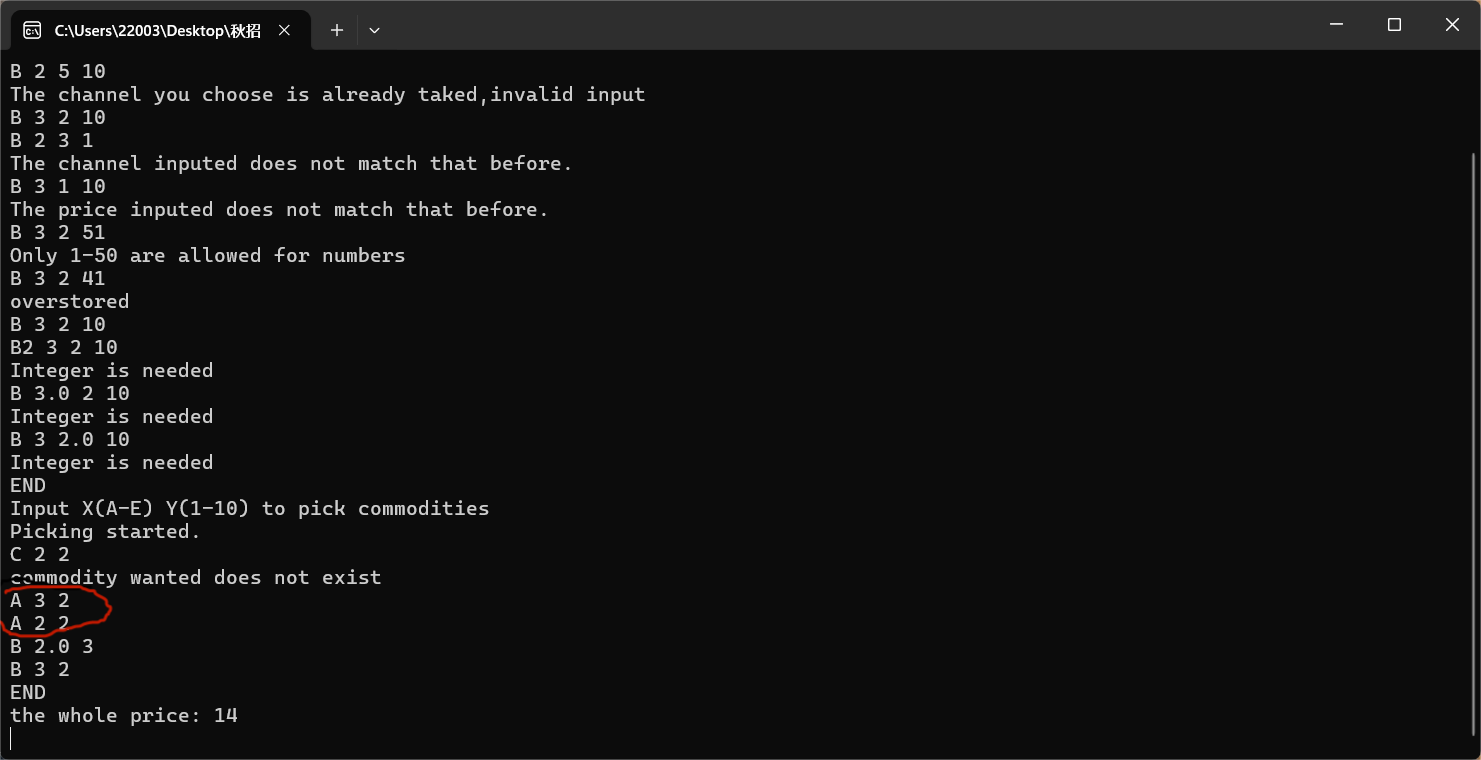


（4） 后期调试、优化与完善

其实我认为这一部分是最繁琐的。

在这一部分，我将printf里面的内容补全，简化合并了部分代码，且多次启动程序、反复试错、发现并补全了很多原来没想到的漏洞，。

例如下述例子：

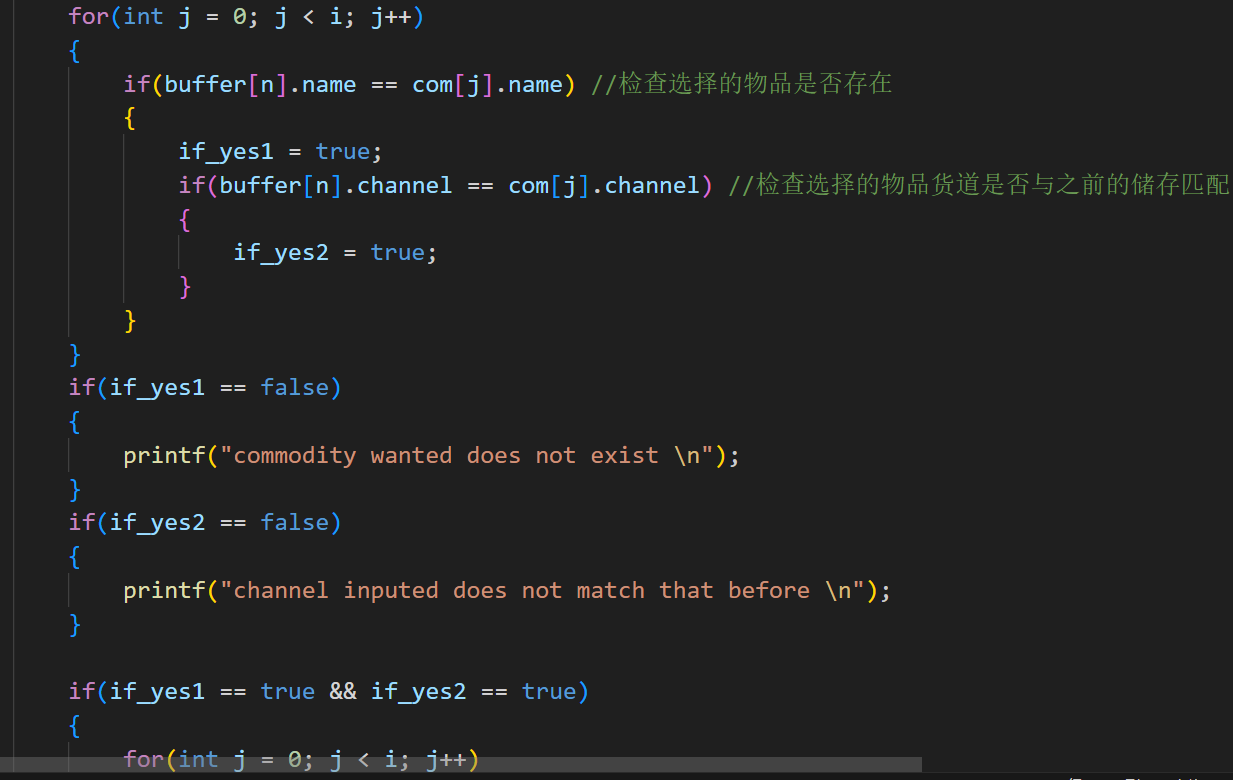


上图是我某次试错时的截图（不完整），如红圈中所示，我发现在第一次定义A的货道为2后（截图没截到），我在picking阶段输入A 3 2后系统只是单纯的忽略这次输入但却没有输出相应的报错提示词，而我记得我已经将这个报错提示写好了，

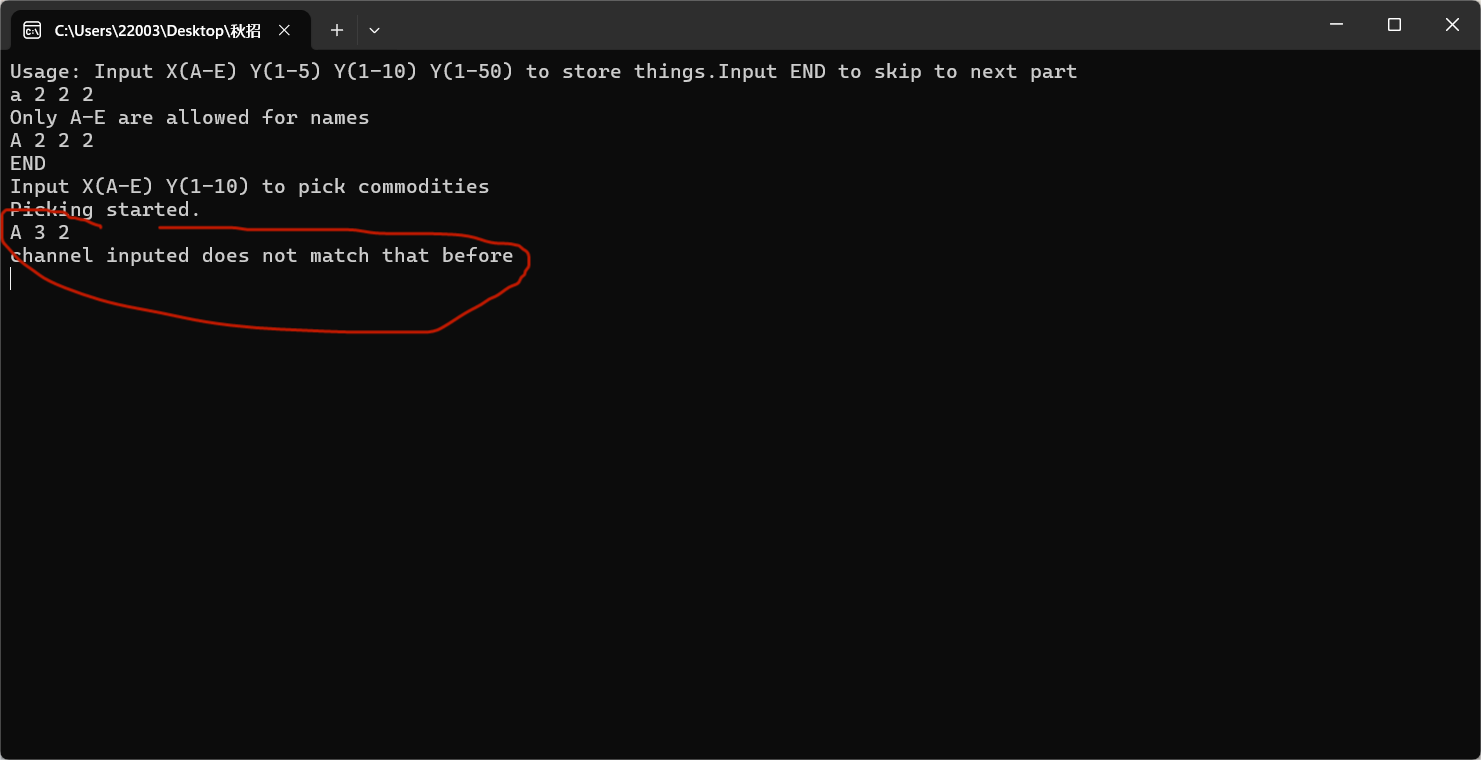
于是我找到了该段代码：



经过分析，我发现我问题出在for循环中的两个if bool表达式从句，他们不应该是并列关系。图片中的代码意味着在遍历之前所有已储存的数据时，name和channel是分开处理的，而不是保证在j为某个值时两者同时成立。而将第二个if bool从句放在第一个里面，就可以保证必须要在同一个j值下，name和channel同时成立才能使两个bool都为true，于是我将代码修改成了下面的截图



在之后的调试过程中，证实了该问题确实解决了。



下面是Level2-1的最终版：

<Level2_1_4(Final).c>

F LEVEL2-2（普通版）

设计思路

2-2大体上相承2-1，但是多了个back功能，我能想到的实现这个功能的办法就是当检查到用户输入了back后，清除其上一次输入的内容所造成的影响。

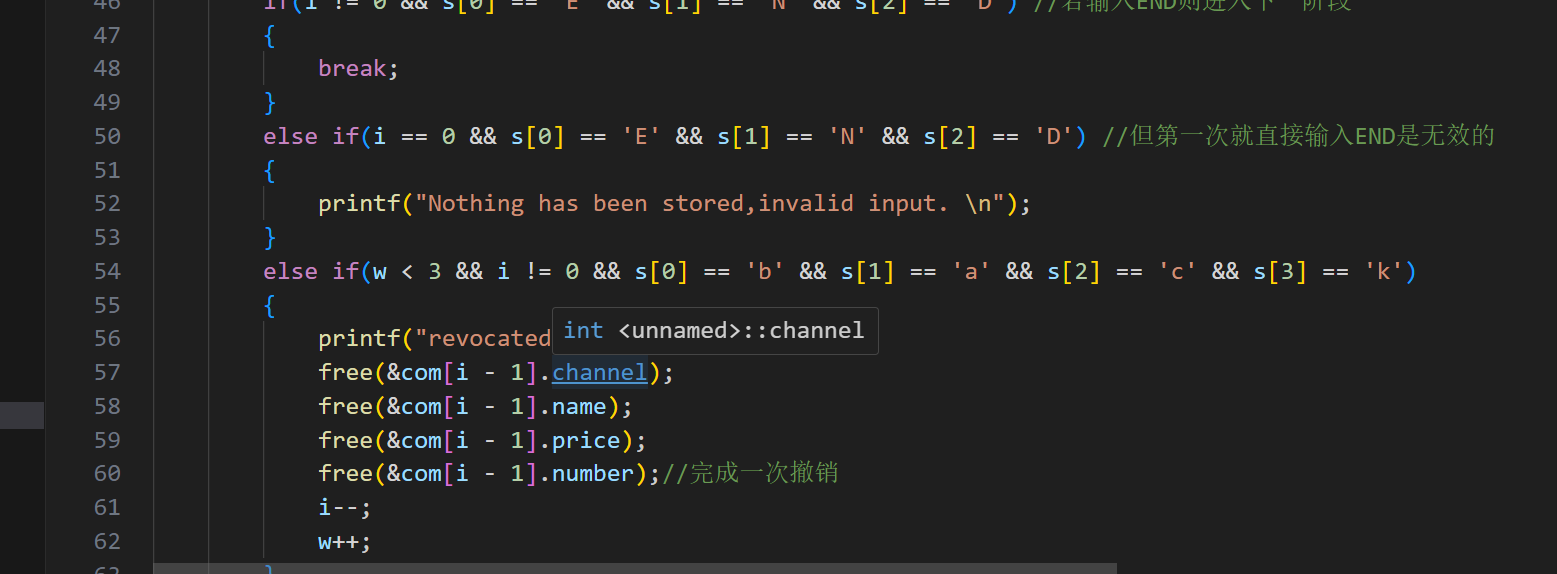
这种影响大致分为两种：一是在某一环节内部的back，另一种是直接back到了上一环节。第一种影响很好实现，直接清除其上一次储存在struct里的数据即可，第二种的话也不难，我选择用goto函数来实现.

完整代码如下所示：

<Level2_2.c>（普通版）

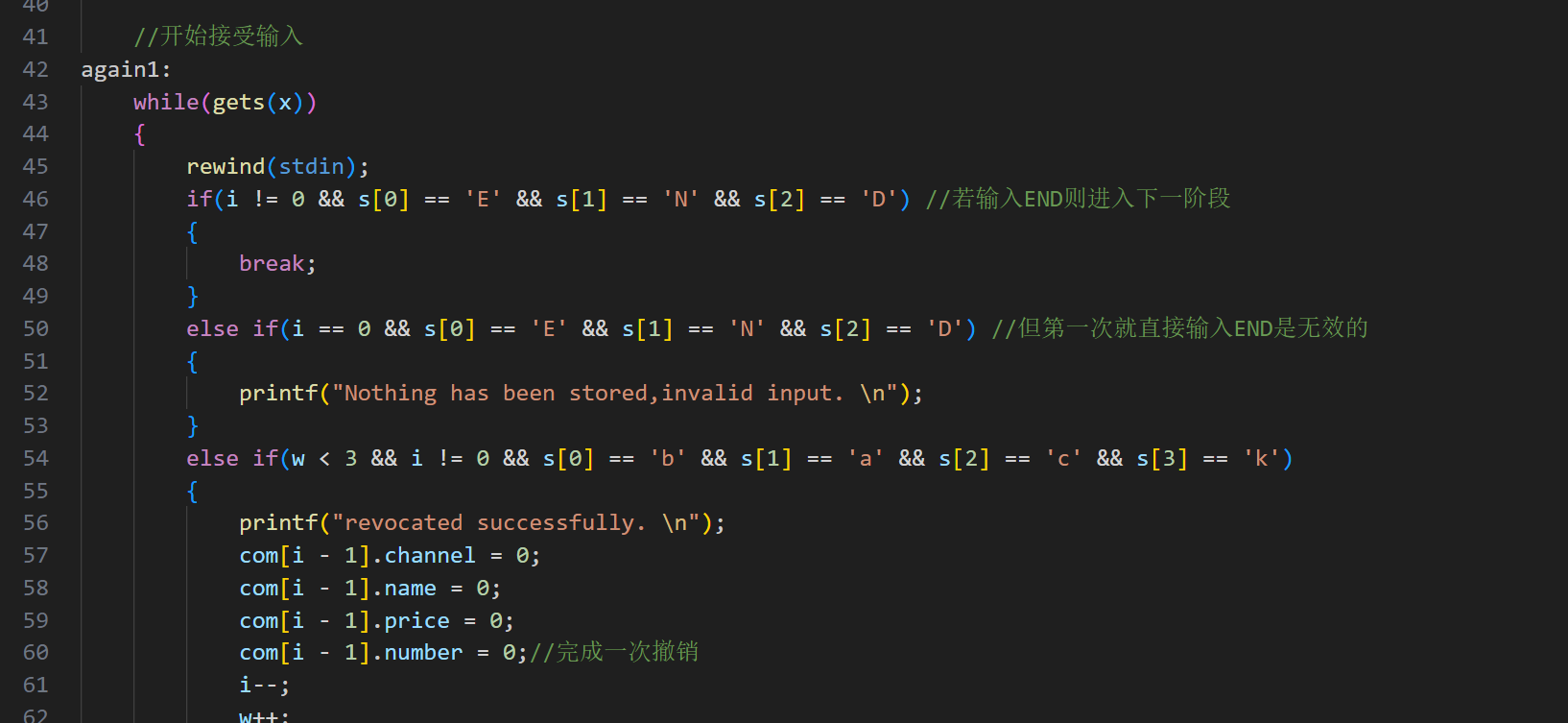
下面我主要想分享一下在做2-2时我认为有波折的点：

1、

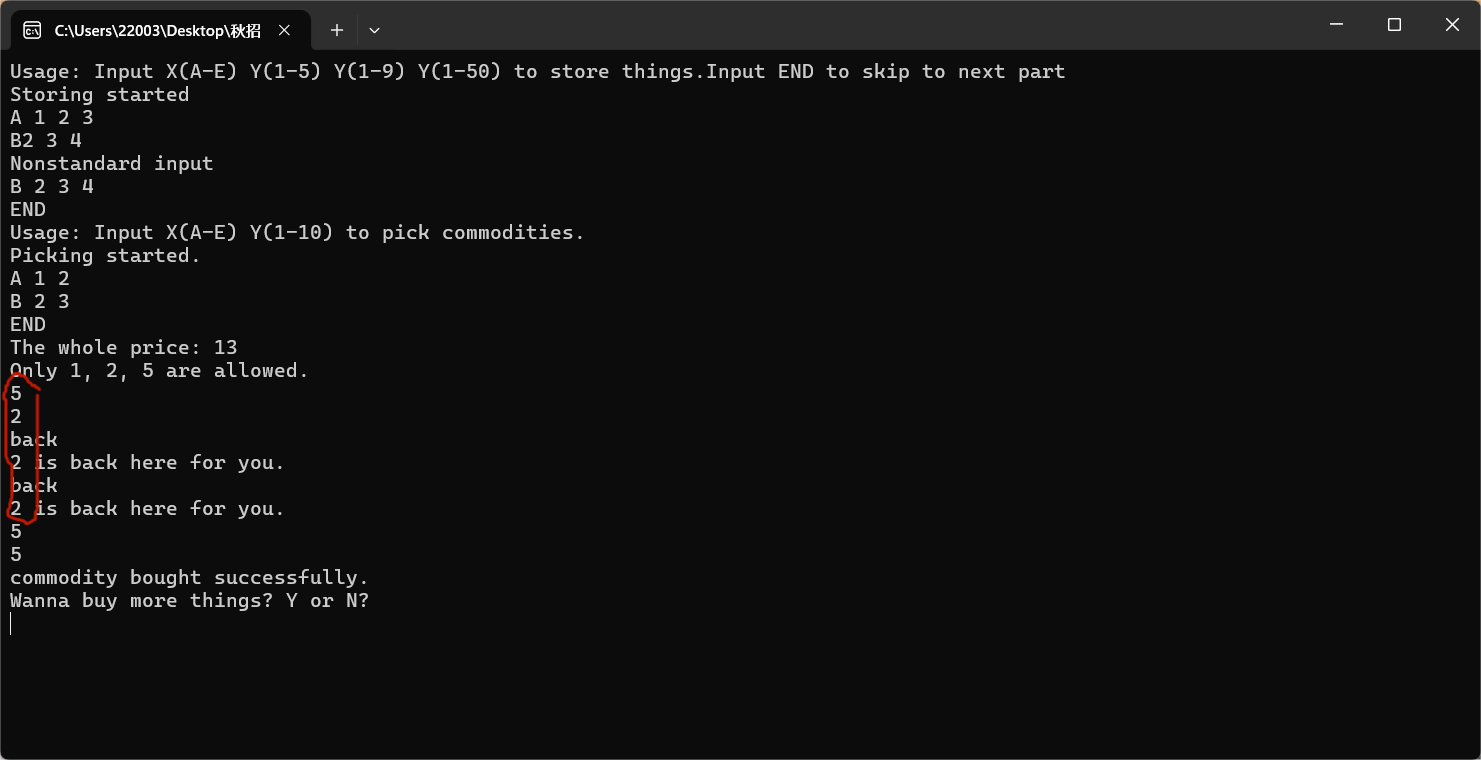


如上图所示，我一开始打算用free来清除数据，结果运行时程序直接崩溃了。

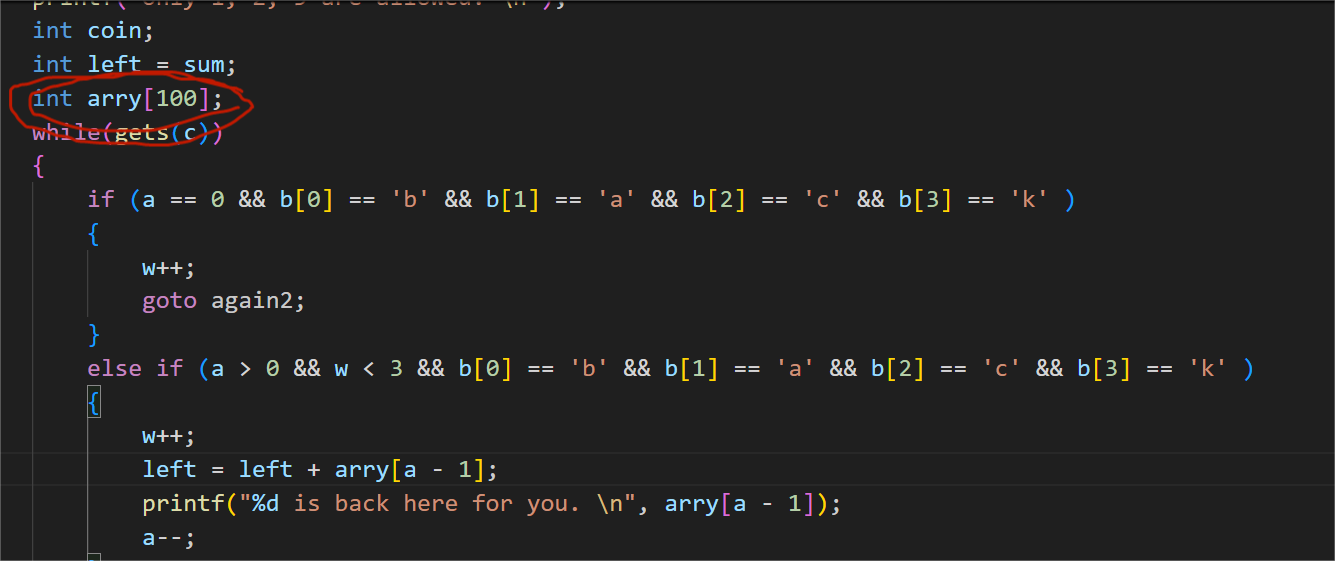
后面我才想起来free是指针动态申请内存释放用的，于是改成了如下所示，直接将其初始化为0，程序便能正常运行了



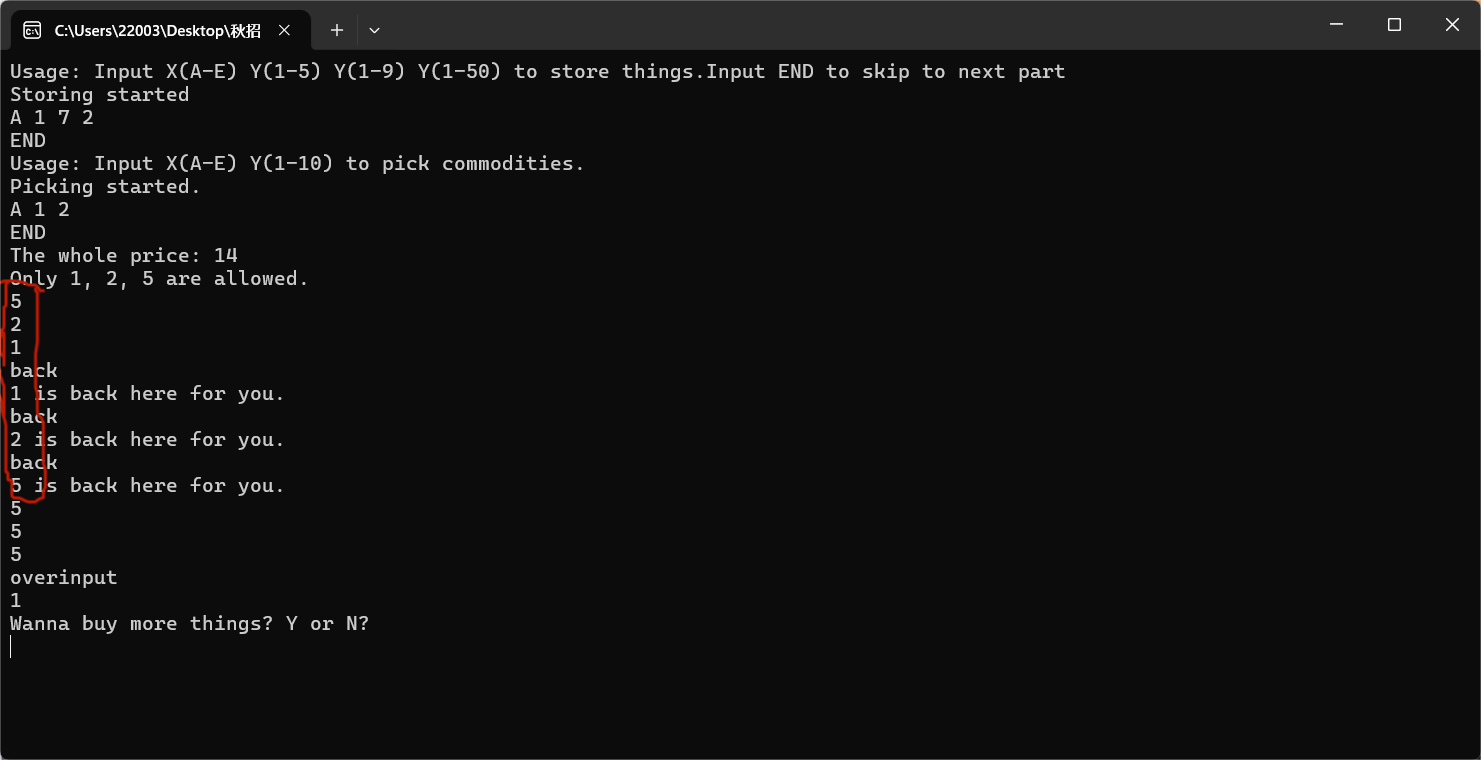
2、第二个事情就是当我测试时，发现投币环节的back有问题，如下所示



我依次投了5、2，但连按两次back，第一次回退2，第二次竟然也是2而不是5，回去看了代码我才发现原来是因为由于我的投币环节是沿用了1-x里面的，故没有将每一次投币的数据都储存起来。为了修改方便，我打算直接用一个数组来储存



再经过相应的代码修改，程序便可以正常运行了，如下所示：



G LEVEL2-2（工程版）

之前在网上看到别人采取了这种工程形式，这次也是我的第一次尝试，实在做得不太好，请见谅

下面是链接：

<Level2_2(Final)>（工程版）