【麦当劳点餐系统-概要设计书】

版本号：v1.0
编制时间：2024 年 4 月 28 日
编制人员：刘南府

1.用户界面设计

1.1 文件版本

文件通信中所需要的文件有两个，分别是 dict.dic 和 input.txt。

dict.dic:菜单文件，文件格式如下
第一行给出 N 和 M，其中 N 代表食物的种类数，M 代表套餐的种类数。
第二行包含 N 个字符串，每个字符串表示第 i 种食物的名称。
接下来 M 行每行包含多个字符串，其中第一个字符串代表第 i 个套餐的名称，后续的第 j 个字符串代表第 i 个套餐中第 j 种食物的名称。

input.txt:数据/需求输入文件，文件格式如下
第一行包含一个整数 n，表示订单个数
第二行包含两个整数 w1 和 w2，其中 w1 代表队列关闭长度，w2 代表队列开放长度，当队列长度大于 w1 时系统关闭，当队列长度小于 w2 时系统开放。
第三行包括 N 个整数，每个整数表示第 i 种食物的制作时长。
第四行包括 N 个整数，每个整数表示第 i 种食物的最大容量。
接下来的 n 行，每行具有一个格式为 00：00：00 的字符串作为第 i 个订单的下单时间，在时间后的每个字符串分别对应订单所需的套餐或食物的名称。

1.2 图形化版本

待完善

2.高层数据结构设计

2.1 全局变量定义

```
int food_kinds_num;          //食物种类数量
int combo_kinds_num;         //套餐种类数量
int order_num;               //订单数量
int queue_low_num;           //队列开放长度（小于则开放）
int queue_high_num;          //队列关闭长度（大于则关闭）
```

2.2 数据结构设计

2.2.1 相关结构体声明

```
struct Node3;
typedef struct Node3 *PtrToNode3;
typedef PtrToNode3 List_combo_food;
typedef PtrToNode3 combo_food_Position;
```

```
struct Node4;
typedef struct Node4 *PtrToNode4;
typedef PtrToNode4 List_order;
typedef PtrToNode4 order_Position;

struct Node5;
typedef struct Node5 *PtrToNode5;
typedef PtrToNode5 List_order_food;
typedef PtrToNode5 order_food_Position;
```

2.2.2 食物信息存储

```
struct Node1
{
    char food_name[50];
    int id;
    int current=0;
    int cap;
    int make_time;
    int schedule=0;
};
struct Node1 food_information[food_kinds_num];
```

我们通过定义一个长度等于食物种类数目的结构体数组来存储食物的各种信息，在结构体 Node1 中，

char food_name[50]	是用于存储食物名称的字符数组。
int id	是用于存储食物在结构体数组位置的变量。
int current	是用于存储食物当前数量的变量。
int cap	是用于存储食物的最大存储数量的变量。
int make_time	是用于存储食物制作所需时间的变量。
int schedule=0	是用于存储食物当前制作进度的变量，单位为秒。

2.2.3 套餐信息存储

```
struct Node2
{
    char combo_name[50];
    int id;
    List_combo_food food_list;
};
struct Node2 combo_information[combo_kinds_num];
```

我们通过定义一个长度等于套餐种类数目的结构体数组来存储套餐的各种信息，在结构体 Node2 中，

char combo_name[50]	是用于存储套餐名称的字符数组
int id	是用于存储套餐在结构体数组位置的变量
List_combo_food food_list	是用于存储套餐中对应食物的链表

而 List_combo_food food_list 的实现如下：

```
struct Node3
{
    int id;
    combo_food_Position next;
};
```

在结构体 Node3 中，

int id　　是用于存储套餐中所包含食物对应的 id，也就是其在存储食物信息结构体数组中 food_information[food_kinds_num]中对应的位置。

2.2.4 订单信息存储

```
struct Node4
{
    int time;
    List_order_food order_food_list;
    int finish_flag=0;
    int receive_flag=0;
    int finish_time;
    order_Position next;
};
List_order L_order=MakeEmpty_order_list(NULL);
```

我们通过定义一张二维链表来存储订单信息，在结构体 Node4 中，

int time　　是用于存储下单时间的变量。

int receive_flag=0　　是用于标记订单接收状态的变量。

int finish_flag=0　　是用于标记订单完成状态的变量。

int finish_time　　是用于存储订单完成时间的变量。

List_order_food order_food_list　　是用于存储订单中所包含食物种类的链表。

而 List_order_food order_food_list 的实现方式如下

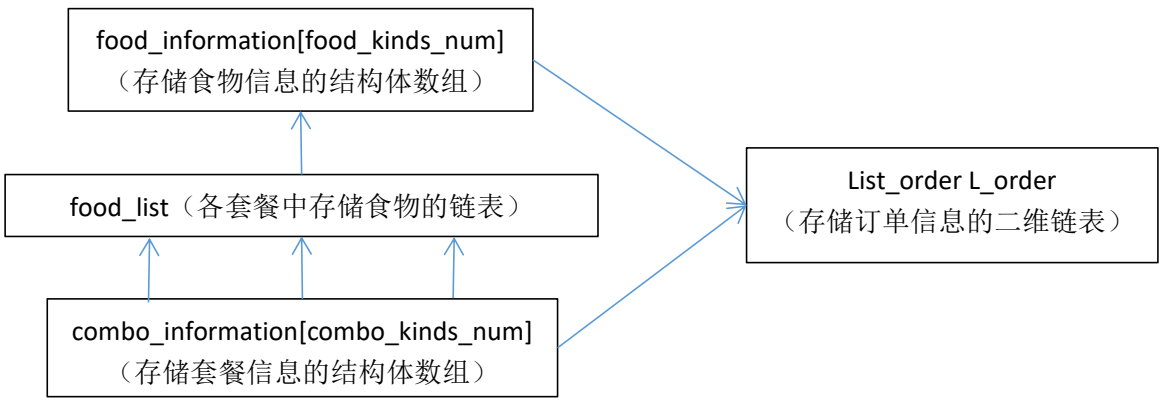
```
struct Node5
{
    int id;
    int finish_flag=0;
    order_food_Position next;
};
```

在结构体 Node5 中，

int id　　是用于存储订单中所包含食物对应的 id，也就是其在存储食物信息结构体数组中 food_information[food_kinds_num]中对应的位置。

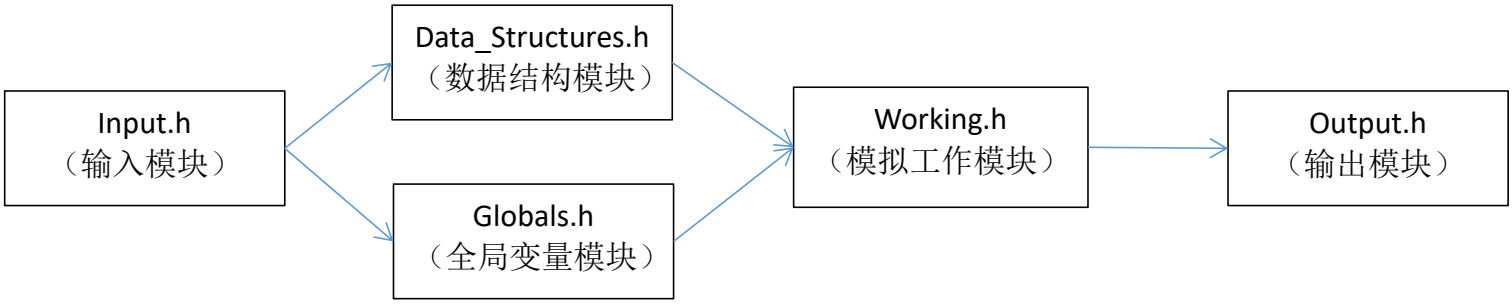
int finish_flag=0　　是用于标记订单中某食物完成状态的变量。

2.3 数据结构示意图



3. 系统模块划分

3.1 系统模块划分示意图



3.2 系统各模块功能描述

3.2.1 输入模块（Input.h）

(1)int food_id_find(struct Node1 *food_array,char *food_name);

通过食物名称查询到其对应的 id（在食物信息结构体数组中的位置）

(2)int combo_id_find(struct Node2 *combo_array,char *combo_name);

通过套餐名称查询到其对应的 id（在套餐信息结构体数组中的位置）

(3) void combo_food_list_insert(char *food_name,List_combo_food L,combo_food_Position P,struct Node1 *food_array);

为某个套餐中存储食物的链表加入新食物

(4)void Delete_combo_food_list(List_combo_food L);

删除某个套餐中存储食物的链表

(5)List_combo_food MakeEmpty_combo_food_list(List_combo_food L);

为某个套餐创建存储食物的链表

(6) void order_list_insert(List_order L,int order_time,order_Position P,List_order_food L_order_food);

为订单链表加入新订单

(7) void Delete_order_list(List_order L);

删除订单链表

(8) List_order MakeEmpty_order_list(List_order L);

创建订单链表

(9) void order_food_list_insert(List_order_food L,int insert_id,order_food_Position P);

为某个订单中存储食物的链表中加入新食物

(10) void Delete_order_food_list(List_order_food L);

删除某个订单中存储食物的链表

(11) List_order_food MakeEmpty_order_food_list(List_order_food L);

为某个订单创建存储食物的链表

(12) void read_data(FILE *fpdict,FILE *fpinput,struct Node1 *food_information,struct Node2 *combo_information);

读取基础数据，将数据读入到 combo_information[combo_kinds_num]

和 food_information[food_kinds_num]中去

(13)void read_order(FILE *fpinput,List_order L,struct Node1 *food_array,struct Node2 *combo_array);

读取订单数据，将数据读入到 L_order 中去

(14)void data_test(struct Node1 *food_information,struct Node2 *combo_information,List_order L_order);

测试数据结构与数据读入的正确性

3.2.2 数据结构模块与全局变量模块（Data_Structures.h,Globals.h）

详见高层数据结构设计

3.2.3 模拟工作模块（Working.h）

(1) int type_conversion_str_to_second(int hour,int minute,int second);
将时：分：秒形式的时间字符串转换为以秒为单位的时间数据

(2) void type_conversion_second_to_str(FILE *fpoutput,int clock);
将以秒为单位的时间数据转换为时：分：秒的形式并打印

(3) void make_food(struct Node1 *food_information,int clock);
模拟食物制作流程

(4) int deal_with_order(List_order L_order,struct Node1 *food_information,int clock);
模拟订单处理流程

(5) void deal_with_this_order(order_Position P,struct Node1 *food_information,int clock);
当接收到新订单时，对其进行一次特殊处理，以判定其是否可以立刻完成

(6) int queue_is_empty(List_order L_order,int clock);
检测实际订单队列是否为空

(7) void working_simulation(struct Node1 *food_information,List_order L_order);
模拟整体工作流程

3.2.4 输出模块（Output.h）

(1) void output(List_order L_order);
扫描整个订单链表状态并给出最终输出结果

(2) void release_all_memory(List_order L_order);
释放程序内存

3.3 核心算法设计

程序的核心处理步骤均集中在模拟工作模块（Working.h）的
void working_simulation(struct Node1 *food_information,List_order L_order)中，
其逻辑为：

初始化时间、订单队列开放状态、订单队列实际长度，设定用于模拟接单的遍历订单链表的指针。

启动循环，循环只有在时间超过十点且订单全部处理完的情况下才会停止，循环的单位是秒。

每一次循环代表一秒内的运行情况，在新的一秒内，程序会执行如下步骤：

(1)对当前实际订单队列进行处理

处理方法为遍历订单链表，检测是否存在已经接单且未完成的订单，如果有，则遍历这一订单的食物需求链表，根据当前食物存量处理订单需求，直至订单链表遍历完成。之后我们再次遍历全部订单，检测是否有已经被接单且全部需求已经被完成的订单，如果有，则标记其为已完成，同时得到处理工作完成后的实际订单队列的长度

(2)对新订单进行判定处理

在这一阶段，程序将检测这一秒是否有新订单进入，如果有新订单，并且同时满足时间在 7：00：00 到 22：00：00 之间与队列状态开放这三个条件，程序将接入新订单，并且判定其是否能立即完成，由于我们已经依次满足了之前的

订单的需求，所以对新订单需求的满足不会与先来的订单产生冲突，若新订单无法立刻处理完成，则实际订单队列长度增加，处理过新订单后，函数内置的用于接收订单的遍历指针将下移；若上述三个条件有一个不满足，则这一时刻的订单将被拒单，也就是保持未被接单的状态并被跳过。如果此时没有新订单，则不会有任何事情发生，直到时间前进到出现新订单。

(3)制作食物

循环从 7：00：00 开始，而制作食物需要切实的一秒，也就是说在 7：00：00 到 7：00：01 的过程中实际上是没有任何食物完成了这一秒的制作流程的，因此我们将制作食物置于处理订单之后，以满足对食物制作时间情况的模拟。

(4)判定订单队列开放状态

在处理完新旧订单后，我们得到了当前确切的实际订单队列长度，根据这一长度，我们判定订单是否开放，若其当前长度大于所规定的上限，则队列状态标记改变为 0，若其当前长度小于所规定的下限，则队列状态标记改变为 1。要注意的是，在从初始状态到队列第一次关闭之前的状态时，若实际订单队列长度进入到了大于下限值但小于上限值的时候，队列依然处于开放状态。