

# 计算机基础与编程综合实验

C++版

实验指导书

武汉理工大学计算机学院

2018 年 2 月

# 1 课程简介

## 1.1 基本信息

课程名称：计算机基础与编程综合实验

学时/学分：32/1

适用专业：计算机类

先修课程：高级语言程序设计

## 1.2 实验目的与要求

编程能力是计算机类专业学生必备的重要专业技能。通过计算机基础与编程综合实验，从无到有地开发一个软件产品，是进行专业能力训练的重要途径。本课程选取实际应用问题，通过精心组织的案例，采用迭代式和增量式开发思想，通过教师引导，使学生逐步实践完成一个系统，并根据学生的不同层次实现不同的扩展功能，体现出差异化和个性化。

本实验课程旨在帮助熟悉软件开发过程，掌握实现一个软件系统的基本方法。在软件系统开发过程中，锻炼学生独立分析问题和解决问题的能力，提高编写程序代码能力，为后续专业领域的学习、研究和开发工作打下基础。

具体来说，通过计算机基础与编程综合实验，学生应该：

- 1) 深入理解 C/C++语言的基本概念和基本原理，如数据类型等，熟练应用顺序选择和循环结构程序设计、函数、结构体、文件读写等基础技术。
- 2) 掌握 C/C++语言的高级知识，如类、vector、链表等技术。
- 3) 掌握模块化开发的具体实现方法，深入领会一些 C/C++程序设计实用开发方法和技巧。
- 4) 了解迭代式软件开发的一般过程，领会系统设计、系统实现以及系统测试的方法。

## 1.3 计费管理系统介绍

**特别说明：本实验指导书只是给学生提供了一种设计参考思路，学生可以根据自己的情况，选择如何完成本系统的基本功能。编程能力较弱的同学可以按本指导书一步一步的实现本系统。编程能力较强的同学，也可以按照实验要求，自行设计系统方案并编写代码，完成基本功能并扩充相关的功能。**

本计费管理系统主要是模拟实现网吧收费的基本功能，提供上机卡管理，每次上机的计费管理，充值退费管理，销卡管理，查询统计，系统管理等功能。

在网吧上机的用户首先在网吧进行开卡，注册一张上机卡。第一次开卡时同时需要向卡

中进行充值。每次上机时，系统根据上机时间和下机时计算费用，从卡中扣除该费用。用户可以要求退回余额，也可以存放在卡中供下次上机使用。系统记载用户每次充值退费的信息，记载每次上机的信息，供查询统计使用。

系统可定义计费策略，提供多种计费方式。例如，会员制；不同时间段收费标准不一样；包天或者月计费等。

### 1.3.1 功能结构

系统功能结构图如下图所示。

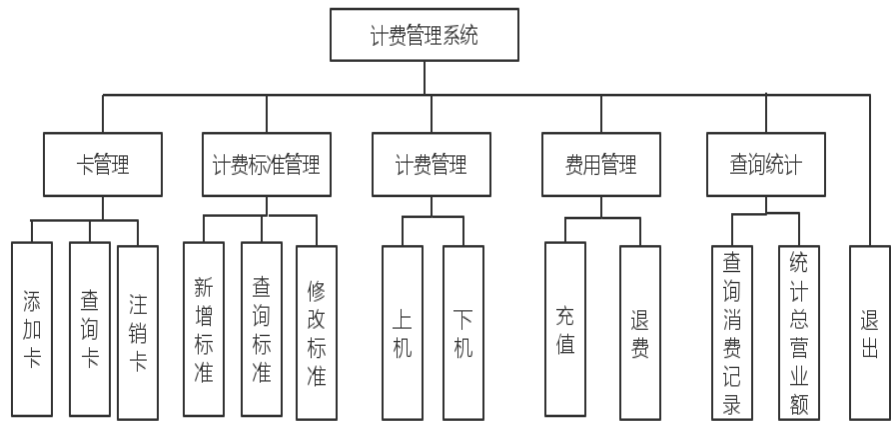


图 1-1 系统功能结构图

### 1.3.2 设计思路

计费管理系统的所涉及到的数据存储文本文件中。本系统有 3 个文本文件，分别是：

- card.txt，卡信息文件，存储所有上机卡
  - billing.txt，计费信息记录文件
  - money.txt，充值退费记录文件
- 1) 计费管理系统启动过程中要完成下列功能：
- 从文本文件 card.txt 中读入所有上机卡信息，每张上机卡片信息存储在一个结构体变量中。
  - 用 vector 容器数组（实验三）或者链表（实验四）存储所有的上机卡。
  - 从文本文件 billing.txt 中读入所有信息，？
- 2) 计费管理系统退出过程中要完成下列功能：
- 保存对上机卡信息的更新，即将所有上机卡信息写入到文本文件 card.txt。

## 1.4 实验项目及学时分配

序号	实 验 项 目 名 称	实验学时	每组人数	实验类别	实验类型
1	计费管理系统的需求分析与人机接口设计	4	1	专业基础	综合性
2	计费管理系统的数据结构设计	4	1	专业基础	综合性
3	计费管理系统的文件存储管理	4	1	专业基础	综合性
4	计费管理系统的链表基本操作	4	1	专业基础	综合性
5	计费管理系统的数据动态存储管理	4	1	专业基础	综合性
6	计费管理系统的基本功能实现	4	1	专业基础	综合性
7	计费管理系统的扩展功能实现	4	1	专业基础	综合性
8	系统验收与报告	4	1	专业基础	综合性

## 2 实验一 人机接口设计

### 2.1 实验目的

- 1) 掌握如何搭建程序框架;学会使用 Project 管理一个应用程序系统的所有涉及到的文件。
- 2) 掌握分支和循环结构的编程方法;掌握函数定义和调用的编程方法。
- 3) 理解将一个程序分解成多个 .cpp 文件,分解成多个函数的编程思想;理解 .cpp 文件和 .h 文件分类的编程思想。掌握条件编译的使用方法。

### 2.2 实验内容

完成类似于图 2-1 的菜单界面。系统能正常运行,界面友好,具有容错功能(即当用户输入不正确数据时,系统不能非正常退出,而是给出友好的提示信息,等待用户的下一步输入选择)。

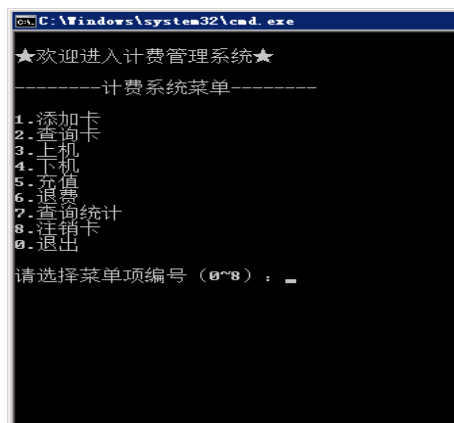


图 2-1 运行结果示例

### 2.3 实验步骤

#### 2.3.1 搭建开发环境

安装 Microsoft Visual Studio 2010。

## 2.3.2 创建 AMS 项目

1. 启动 Microsoft Visual Studio 2010，首次运行会要求开发人员选择默认环境设置，如图所示。在本实验中设置为：Visual C++开发设置。系统启动后会自动创建一个空的解决方案。



图 2-2 选择默认环境设置

2. 在“文件”菜单下，新建“项目”。项目类型为“Win32 控制台程序”，项目名称为“AMS”，默认情况下解决方案名称也为“AMS”，输入项目保存的位置。如下图所示。

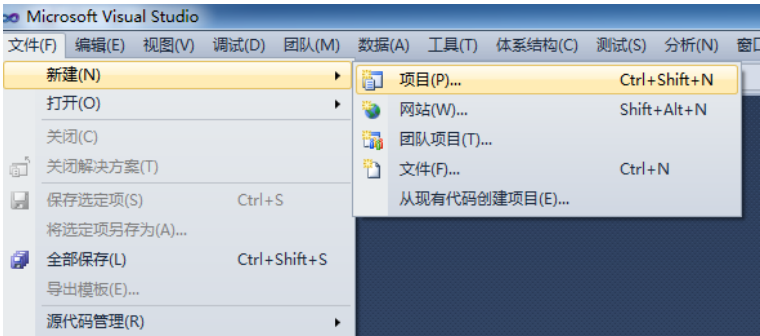


图 2-3 新建项目步骤 1

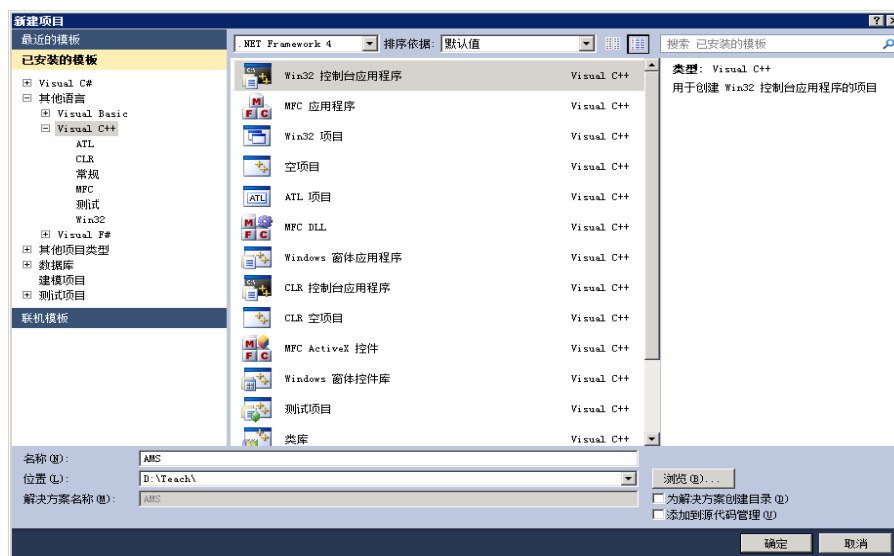


图 2-4 新建项目步骤 2

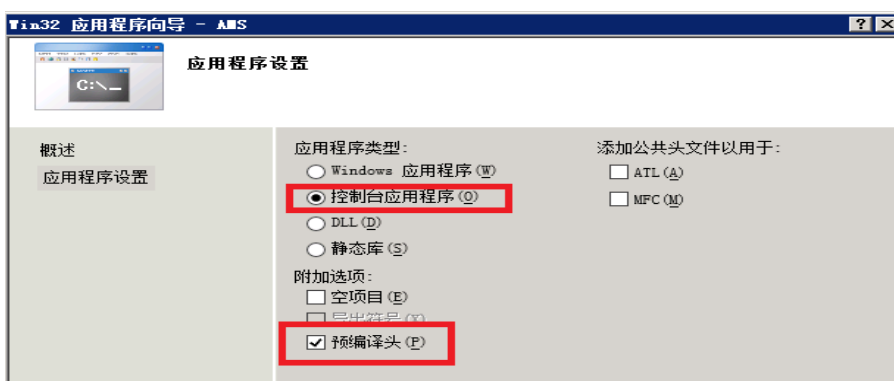


图 2-5 新建项目步骤 3

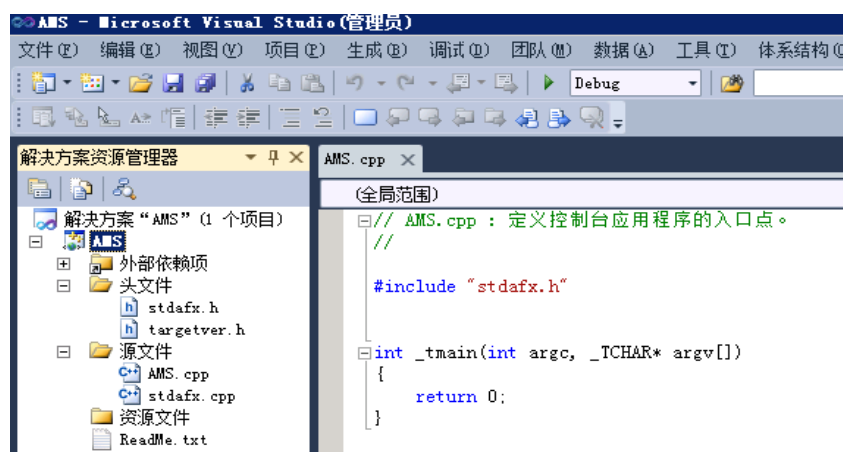


图 2-6 新建项目步骤

### 2.3.3 项目和解决方案的理解

项目是构成某个程序的全部组件的容器。程序通常由一个或多个包含用户代码的源文件，可能还要加上包含其它辅助数据的文件组成。某个项目的所有文件都存储在相应的项目文件夹中，关于项目的详细信息存储在一个扩展名为.vcproj 的 xml 文件中，该文件同样存储在相应的项目文件夹中。项目文件夹还包括其它文件夹，它们用来存储编译及链接项目时所产生的输出。

解决方案的含义是一种将所有程序和其它资源(它们是某个具体的数据处理问题的解决方案)聚集到一起的机制。例如，用于企业经营的分式订单录入系统可能由若干个不同的程序组成，而各个程序是作为同一个解决方案内的项目开发的，因此，解决方案就是存储与一个或多个项目有关的所有信息的文件夹，这样就有一个或多个项目文件夹是解决方案文件夹的子文件夹。与解决方案中项目有关的信息存储在扩展名为.sln 和.suo 的两个文件中。当创建某个项目时，如果没有选择在现有的解决方案中添加该项目，那么系统将自动创建一个新的解决方案。

我们可以在现有的解决方案中添加任意种类的项目，但通常只添加与该解决方案内现有项目相关的项目。一般来说，各个项目都应该有自己的解决方案，除非我们有很好的理由不这样做。

### 2.3.4 代码实现

在 AMS 项目的 AMS.cpp 文件中输入以下的代码。

```
1
2    // AMS.cpp : 定义控制台应用程序的入口点。
3    //
4
5    #include "stdafx.h"
6
7    #include <iostream>
8    using namespace std;
9
10   int _tmain(int argc, _TCHAR* argv[])
11   {
12       char selection;    // 输入菜单项编号
13
14       cout << endl;
15       cout << "★欢迎进入计费管理系统★" << endl;
16       cout << endl;
17
18       do
19       {
20           cout << "-----计费系统菜单-----" << endl << endl;
21           cout << "1. 添加卡" << endl;
22           cout << "2. 查询卡" << endl;
```



```

23         cout << "3. 上机"         << endl;
24         cout << "4. 下机"         << endl;
25         cout << "5. 充值"         << endl;
26         cout << "6. 退费"         << endl;
27         cout << "7. 查询统计"     << endl;
28         cout << "8. 注销卡"       << endl;
29         cout << "0. 退出"         << endl << endl;
30         cout << "请选择菜单项编号 (0~8): ";
31
32         selection = 'a'; // 初始化选择的菜单项编号为'a'
33         cin >> selection; // 输入菜单项编号
34         cin.clear();
35         cin.sync();
36
37         switch(selection) //输出选择的菜单编号
38         {
39             case '1':
40             {
41                 cout << endl << "-----添加卡-----" << endl << endl;
42                 break;
43             }
44             case '2':
45             {
46                 cout << endl << "-----查询卡-----" << endl << endl;
47                 break;
48             }
49             case '3':
50             {
51                 cout << endl << "-----上机-----" << endl << endl;
52                 break;
53             }
54             case '4':
55             {
56                 cout << endl << "-----下机-----" << endl << endl;
57                 break;
58             }
59             case '5':
60             {
61                 cout << endl << "-----充值-----" << endl << endl;
62                 break;
63             }
64             case '6':
65             {
66                 cout << endl << "-----退费-----" << endl << endl;

```

```

67         break;
68     }
69     case '7':
70     {
71         cout << endl << "-----查询统计-----" << endl << endl;
72         break;
73     }
74     case '8':
75     {
76         cout << endl << "-----注销卡-----" << endl << endl;
77         break;
78     }
79     case '0':
80     {
81         cout << endl << "谢谢你使用本系统!" << endl << endl;
82         break;
83     }
84     default:
85     {
86         cout << "输入的菜单编号错误! \n";
87         break;
88     }
89     }
90     cout << endl;
91 }while(selection != '0');
92
93 return 0;
94 }
95

```

## 2.3.5 代码分析

### 1. #include "stdafx.h"

这行代码是.net 开发环境中 C++的项目编译所需要的“预编译头”。开发人员**在后面**添加的.cpp 文件时，都要在第一行加上此语句，这样不会出现编译错误

### 2. int \_tmain(int argc, \_TCHAR\* argv[])

在.net 开发环境的“Win32 控制台应用程序”项目中，默认已经用\_tmain()函数替代了我们熟知的 main()函数。

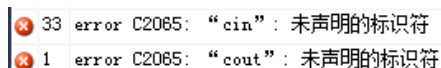
### 3. cin.sync()和 cin.clear()

`cin.sync()`是用来清除缓存区的数据流。`cin.clear()`是用来更改 `cin` 的状态标示符的。如果标示符没有改变那么即使清除了数据流也无法输入。所以 `cin.clear()`和 `cin.sync()`要联合起来使用。

在控制台应用程序中，从键盘输入数据时，用户如果输入过程中无意中多输入了一些数据，这些数据没有被应用程序接收，会缓存在键盘缓冲区。下次需要从键盘接收数据时，会先从键盘缓冲区取数据。为了不让这些数据影响下次的输入，在每次接收键盘输入时，最好时先清除键盘缓存区的数据流。

## 2.3.6 常见错误

### 1. 错误现象 1:



```
33 error C2065: "cin": 未声明的标识符
1 error C2065: "cout": 未声明的标识符
```

解决方法：加上语句

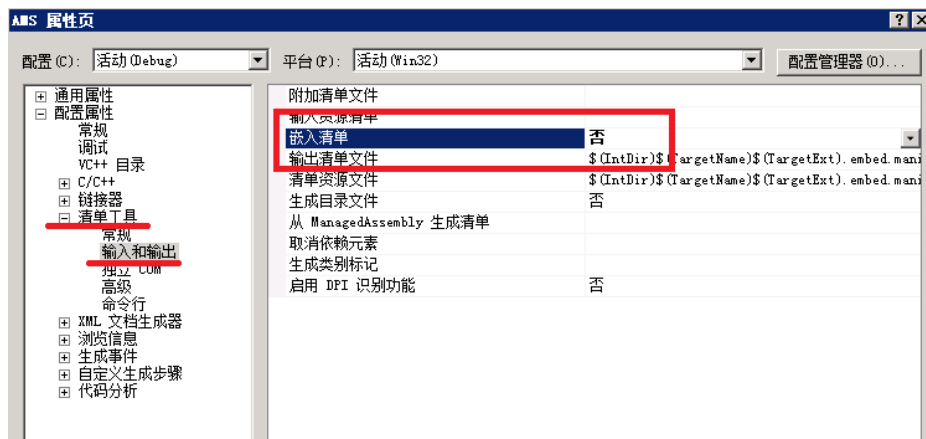
```
#include <iostream>
using namespace std;
```

### 2. 错误现象 2:



```
1 error LNK1123: 转换到 COFF 期间失败: 文件无效或损坏
```

解决方法:



## 2.3.7 代码优化

### 1. 优化原因

- 1) 在程序设计中，一般单个函数中代码一般不能过长。函数代码过长，不便于开发者的阅读和修改。因此，一个项目会分解成若干个函数，每个函数完成一个独立的小功能，函数间通过函数调用来完成大功能。
- 2) 如果把所有的函数放在单个.cpp 文件中，会导致.cpp 文件很大，如代码行数超过万行甚至更多，这样也不便于开发者对代码的管理和维护。因此，通常一个项目中会

有很多个.cpp 文件，每个.cpp 文件中包含若干个函数。单个.cpp 中包含的所有函数基本上是实现同一类型的一些功能。

*注意：函数怎么分类没有绝对的标准，用户可以根据自己对项目的理解，将各个的函数放置到不同的.cpp 文件中。*

## 2. 优化方法

- 1) 由于前面的\_tmain()函数较长，我们可以将界面显示语句写成一个单独的函数，即将优化前的第 20-32 行代码拿出来单独作为一个函数 outputMenu()。将优化前第 20-32 行代码换成调用函数 outputMenu();的形式，见优化后代码的第 22 行。
- 2) 由于 outputMenu()函数的定义在\_tmain()函数后面，因此在调用该函数前，必须先声明该函数，见优化后代码的第 10 行。

## 3. 优化后代码

```
1
2 // AMS.cpp : 定义控制台应用程序的入口点。
3 //
4
5 #include "stdafx.h"
6
7 #include <iostream>
8 using namespace std;
9
10 void outputMenu();
11
12 int _tmain(int argc, _TCHAR* argv[])
13 {
14     char selection; // 输入菜单项编号
15
16     cout << endl;
17     cout << "★欢迎进入计费管理系统★" << endl;
18     cout << endl;
19
20     do
21     {
22         outputMenu();
23
24         selection = 'a'; // 初始化选择的菜单项编号为'a'
25         cin >> selection; // 输入菜单项编号
26         cin.clear();
27         cin.sync();
28
29         switch(selection) //输出选择的菜单编号
30         {
31             case '1':
```

```
32     {
33         cout << endl << "-----添加卡-----" << endl << endl;
34         break;
35     }
36 case '2': // 查询卡
37     {
38         cout << endl << "-----查询卡-----" << endl << endl;
39         break;
40     }
41 case '3':
42     {
43         cout << endl << "-----上机-----" << endl << endl;
44         break;
45     }
46 case '4': // 下机
47     {
48         cout << endl << "-----下机-----" << endl << endl;
49         break;
50     }
51 case '5':
52     {
53         cout << endl << "-----充值-----" << endl << endl;
54         break;
55     }
56 case '6':
57     {
58         cout << endl << "-----退费-----" << endl << endl;
59         break;
60     }
61 case '7':
62     {
63         cout << endl << "-----查询统计-----" << endl << endl;
64         break;
65     }
66 case '8':
67     {
68         cout << endl << "-----注销卡-----" << endl << endl;
69         break;
70     }
71 case '0': // 退出
72     {
73         cout << endl << "谢谢你使用本系统!" << endl << endl;
74         break;
75     }
```

```

76         default:
77         {
78             cout << "输入的菜单编号错误! \n";
79             break;
80         }
81     }
82     cout << endl;
83 }while(selection != '0');
84
85 return 0;
86 }
87
88 // 系统菜单
89 void outputMenu()
90 {
91     cout << "-----计费系统菜单-----" << endl << endl;
92
93     cout << "1. 添加卡" << endl;
94     cout << "2. 查询卡" << endl;
95     cout << "3. 上机" << endl;
96     cout << "4. 下机" << endl;
97     cout << "5. 充值" << endl;
98     cout << "6. 退费" << endl;
99     cout << "7. 查询统计" << endl;
100    cout << "8. 注销卡" << endl;
101    cout << "0. 退出" << endl << endl;
102    cout << "请选择菜单项编号 (0~8) : ";
103 }
104

```

## 2.3.8 代码进一步优化

### 1. 优化原因

在一个项目中，主文件(或者启动文件，在本例中是 AMS.cpp)中一般只放置\_tmain()函数。因此我们需要将 outputMenu()函数放置到另一个文件中。

*注意：用户在编写程序时应该遵循这些通用的编程风格。虽然不是必须的，就像变量的命名一样，但是良好的编程风格有助于项目开发过程中的修改调试和以后的维护。*

### 2. 优化方法

- 1) 向项目添加一个 menu.cpp 文件，该文件中将存放跟用户界面显示和输入有关的一些函数，同时添加一个 menu.h 头文件。

下面示意图演示了如何添加 menu.cpp 文件，添加 menu.h 文件过程类似，只需在步骤 2 中，选择“头文件(.h)”即可。

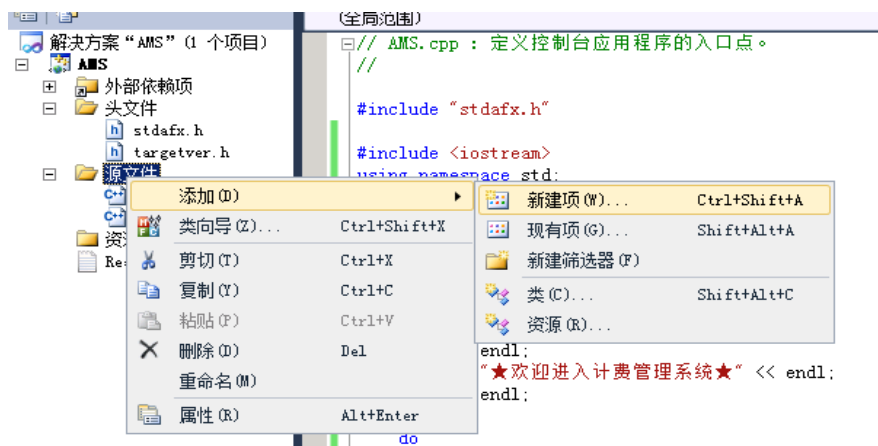


图 2-7 添加 menu.cpp 步骤 1

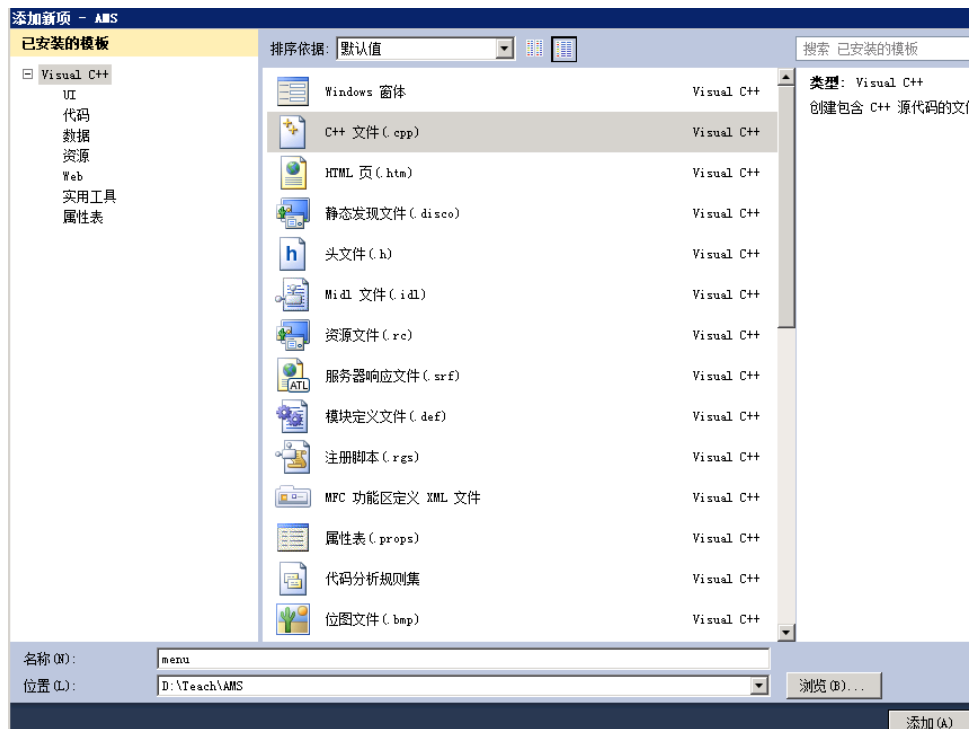
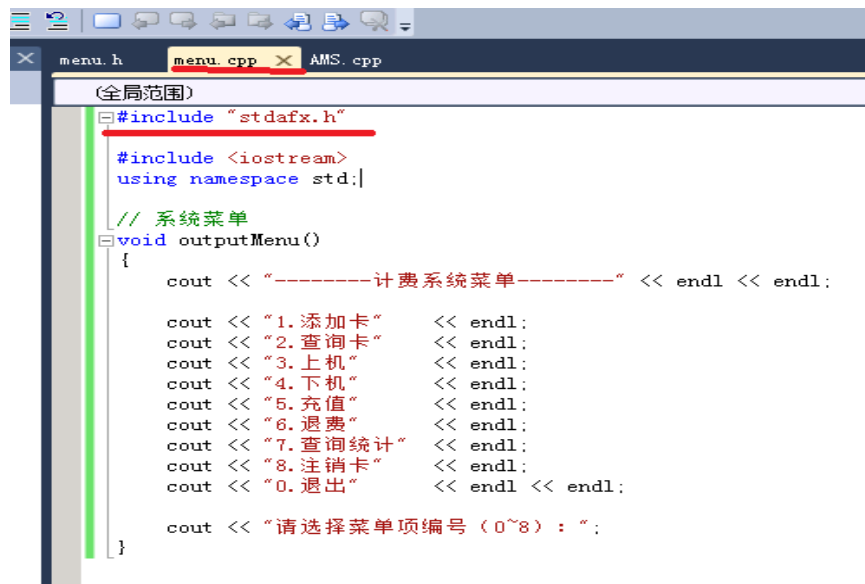


图 2-8 添加 menu.cpp 步骤 2

- 2) 将 outputMenu()函数定义放在 menu.cpp 中，将 outputMenu()函数的声明放在 menu.h。

The screenshot shows the Visual Studio editor with the 'menu.cpp' file open. The code defines a function 'outputMenu()' that prints a menu for a billing system. The menu options are: 1. Add card, 2. Query card, 3. Login, 4. Logout, 5. Recharge, 6. Withdrawal, 7. Query statistics, 8. Cancel card, and 0. Exit. The function prompts the user to select a menu item number (0-8).

```
(全局范围)
#include "stdafx.h"

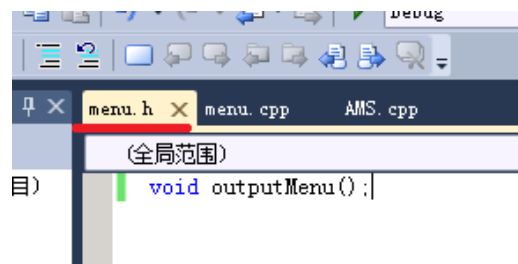
#include <iostream>
using namespace std;

// 系统菜单
void outputMenu()
{
    cout << "-----计费系统菜单-----" << endl << endl;

    cout << "1. 添加卡" << endl;
    cout << "2. 查询卡" << endl;
    cout << "3. 上机" << endl;
    cout << "4. 下机" << endl;
    cout << "5. 充值" << endl;
    cout << "6. 退费" << endl;
    cout << "7. 查询统计" << endl;
    cout << "8. 注销卡" << endl;
    cout << "0. 退出" << endl << endl;

    cout << "请选择菜单项编号 (0~8) : ";
}
```

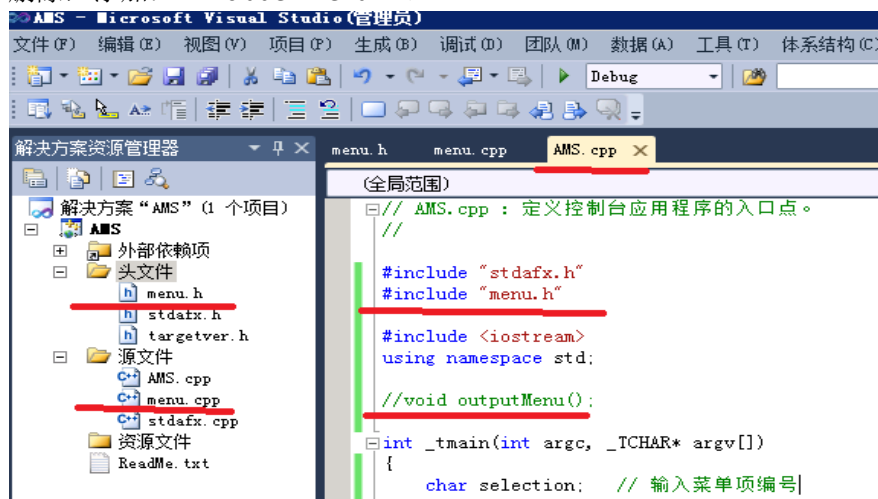
图 2-9 menu.cpp 内容

The screenshot shows the Visual Studio editor with the 'menu.h' file open. It contains a single function declaration 'void outputMenu();' in the global scope.

```
(全局范围)
void outputMenu();
```

图 2-10 menu.h 内容

- 3) 修改 AMS.cpp 文件，将上一部优化代码中的 outputMenu()函数的声明（第 10 行）删除，添加： #include "menu.h"。

The screenshot shows the Visual Studio editor with the 'AMS.cpp' file open. The code has been modified to include 'menu.h' and remove the function declaration. The 'stdafx.h' file is also included. The main function '\_tmain' is shown, which prompts the user to input a menu item number.

```
AMS - Microsoft Visual Studio (管理页)
文件(F) 编辑(E) 视图(V) 项目(P) 生成(B) 调试(D) 团队(M) 数据(A) 工具(T) 体系结构(C)

解决方案资源管理器
解决方案“AMS” (1 个项目)
AMS
  外部依赖项
  头文件
    menu.h
    stdafx.h
    targetver.h
  源文件
    AMS.cpp
    menu.cpp
    stdafx.cpp
  资源文件
    ReadMe.txt

(全局范围)
// AMS.cpp : 定义控制台应用程序的入口点。
//
#include "stdafx.h"
#include "menu.h"

#include <iostream>
using namespace std;

//void outputMenu();

int _tmain(int argc, _TCHAR* argv[])
{
    char selection; // 输入菜单项编号
```

图 2-11 AMS.cpp 的修改

### 3. 特别注意

- ①在项目中每添加一个.cpp 文件，应该同时添加一个同名字的.h 头文件；在.cpp



文件中添加一个函数定义时，应该在同名的.h 头文件中添加该函数的声明。这样在调用该函数的文件中，只需把该函数的.h 头文件#include 进去就可以了。（具体的原因会在后面单独的章节中讲解）

②如果修改了函数定义，同时要在.h 头文件中修改函数声明。

③在 menu.cpp 文件第一行添加：#include "stdafx.h"

## 2.4 实验思考

## 3 实验二 数据结构设计

### 3.1 实验目的

- 1) 掌握结构体的使用方法。
- 2) 掌握容器 `vector` 的使用方法。
- 3) 掌握 `typedef` 的使用方法。

### 3.2 实验内容

- 1) 完成“添加卡”功能。
- 2) 完成“查询卡”功能。

### 3.3 `vector` 数据类型

#### 数组与 `vector` 的比较

- 数组定义的时候必须定义数组的元素个数，而 `vector` 不需要。
- 数组定义后的空间是固定的，不能改变；而 `vector` 要灵活得多，可再加或减。
- 不能将一个数组赋值给另一个数组；但是 `vector` 可以。

#### `vector` 基本操作示例

- 头文件：`#include <vector>`
- 定义 `vector` 对象：`vector<int> vec;`
- 尾部插入数据：`vec.push_back(a);`
- 使用下标访问元素：`vec[0]`等，下标是从 0 开始的。
- 使用迭代器访问元素  
`vector<int>::iterator it;`  
`for(it=vec.begin();it!=vec.end();it++)`  
`cout<<*it<<endl;`
- 插入元素：`vec.insert(vec.begin()+i,a);`在第 `i+1` 个元素前面插入 `a`;
- 删除元素：  
`vec.erase(vec.begin()+2);`删除第 3 个元素  
`vec.erase(vec.begin()+i,vec.end()+j);`删除区间`[i,j-1]`;区间从 0 开始

### 3.4 与时间有关的知识

系统中涉及到时间处理，C++提供了与时间处理相关的一些函数。在了解这些函数前，我们先熟悉与时间处理有关的数据类型 `time_t` 和 `tm`，它们在 `time.h` 中定义。

`time_t` 是 64 位长整数，精确到秒，是从 1970 年零点时间到当前时间经过了多少秒。

tm 是结构体类型，如下所示。

```
1
2  struct tm
3  {
4      int tm_sec; /*秒，0-59*/
5      int tm_min; /*分钟，0-59*/
6      int tm_hour; /*小时，0-23*/
7      int tm_mday; /*日，1-31*/
8      int tm_mon; /*月，0-11*/
9      int tm_year; /*年，从1900至今已经多少年*/
10     int tm_wday; /*星期，从星期日算起，0-6*/
11     int tm_yday; /*天数，0-365*/
12     int tm_isdst; /*日光节约时间的旗标*/
13 };
14
```

常见时间函数：

- **time\_t time(time\_t\* t);** // 取得从 1970 年 1 月 1 日至今的秒数  
time\_t t = time(NULL); // 获取本地时间
- **struct tm \*localtime(const time\_t \*clock);** // 将长整数时间转换为结构体时间，从中  
//得到“年月日，星期，时分秒”等信息  
struct tm\* current\_time = localtime(&t);
- **time\_t mktime(struct tm\* timeptr);** // 将 struct tm 结构的时间转换为长整数  
// 从 1970 年至这个时间点的秒数

由于一般用户能够接受的时间是字符串“2017 年 3 月 8 日 15 时 30 分”这样的形式，因此我们需要编写自己的时间处理函数，将结构体 tm 中相关信息取出来组合成这样的字符串。在本例中，用 tool.cpp 文件存放与时间处理有关的函数，tool.h 文件存放函数的声明。

```
1  #include <time.h> // 包含时间类型头文件
2  #include <stdio.h> // 包含sscanf()函数头文件
3
4  void timeToString(time_t t, char* pBuf)
5  {
6      struct tm * timeinfo;
7
8      timeinfo = localtime(&t);
9      strftime(pBuf, 20, "%Y-%m-%d %H:%M", timeinfo);
10 }
11
12 time_t stringToTime(char* pTime)
13 {
14     struct tm tml;
15     time_t timel;
```

```

16
17     sscanf(pTime, "%d-%d-%d %d:%d",&tml.tm_year, &tml.tm_mon,
18             &tml.tm_mday , &tml.tm_hour, &tml.tm_min);
19
20     tml.tm_year -= 1900; // 年份为从1900年开始
21     tml.tm_mon -= 1;    // 月份为0~11
22     tml.tm_sec = 0;
23     tml.tm_isdst = -1;
24
25     time1 = mktime(&tml);
26
27     return time1;
28 }
29

```

- `size_t strftime( char *str, size_t maxsize, const char *fmt, struct tm *time )` 函数  
功能：将时间格式化，或者说格式化一个时间字符串。即将时间 `time` 格式化为 `fmt` 样式的字符串。  
`fmt`:  
%Y, 用 CCYY 表示的年（如：2004）  
%m, 月份 (1-12)  
%d, 月中的第几天(1-31)  
%H, 小时, 24 小时格式 (0-23)  
%M, 分钟(0-59)
- `int sscanf( const char *str, const char * format,.....)`  函数  
功能：将字符串 `str` 根据参数 `format` 字符串来转换并格式化数据。格式转换形式请参考 `scanf()`。转换后的结果存于对应的参数内。

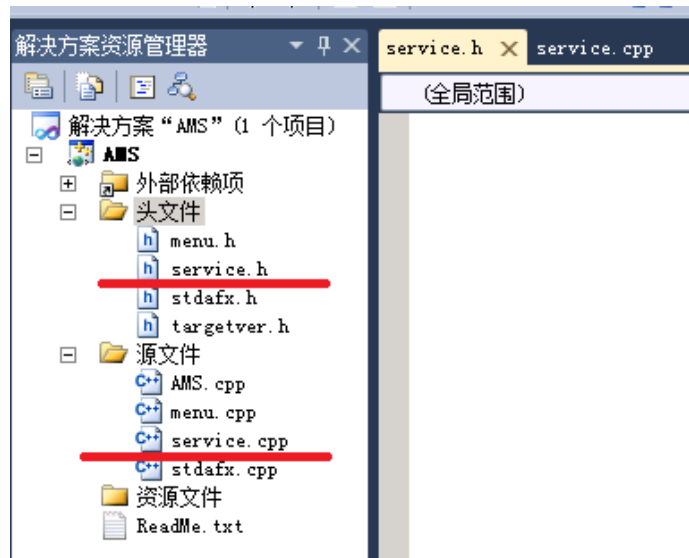
## 3.5 实验步骤（非类版本）

良好的编程风格不应该在 `_tmain()` 函数中编写较多行的代码，因此我们将每个菜单项实现的功能用一个主函数来实现。对于比较复杂的功能（也会有很长的代码），可以细分为更多的子功能，每个子功能用一个独立函数实现，通过在主函数中调用子功能函数来实现各菜单项的功能。

在本示例中，我们使用 `addCard()` 函数，通过在此函数中再调用其它函数完成添加新卡的功能。

### 3.5.1 添加 `service.cpp` 和 `service.h`

前面说过主文件（`AMS.cpp`）中一般只放置 `_tmain()` 函数，因此我们将 `addCard()` 函数放置到另一个文件中。在本例中我们添加一个新文件 `service.cpp`，它所包含的函数都是实现菜单项各功能的主函数（详见表 1）。同时我们添加同名的头文件 `service.h`。



菜单项功能	实现函数
上机	addCard()
查询	queryCard()
上机	
下机	
充值	
退费	
销卡	
统计	
退出	

添加文件完成后，我们要做如下的一些准备工作：

- 根据.net 编程环境编译要求，向 service.cpp 中添加下面语句  
`#include "stdafx.h"`  
 由于一般都会用到标准的输入输出，所以同时添加下面语句  
`#include <iostream>`  
`using namespace std;`  
 在 service.cpp 中添加 addCard()空函数，为下一步编写做准备。  
 见图 1，
- 在 service.h 中添加 addCard()函数的声明，见图 2。  
 注意：函数的声明后面的分号“;”不要漏掉。
- 在 \_tmain()函数调用 addCard()函数，因此要在 AMS.cpp 中添加下面语句，见图 3。  
`#include "service.h"`
- 上述步骤完成后，编译无错误后能运行则继续后面步骤。

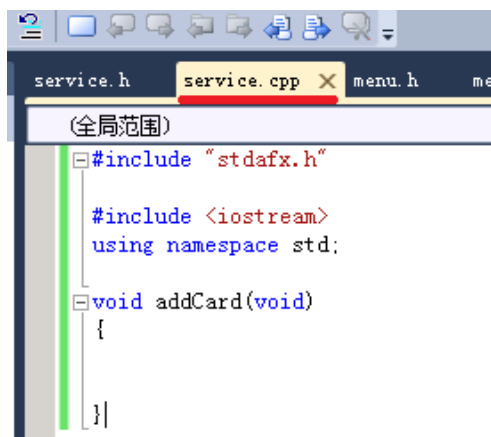


图 1

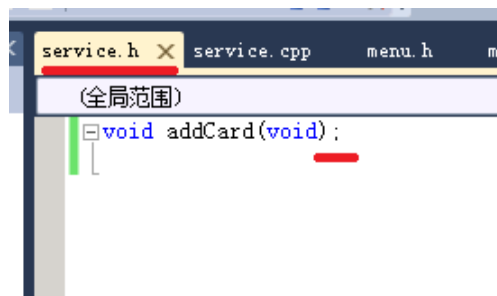


图 2

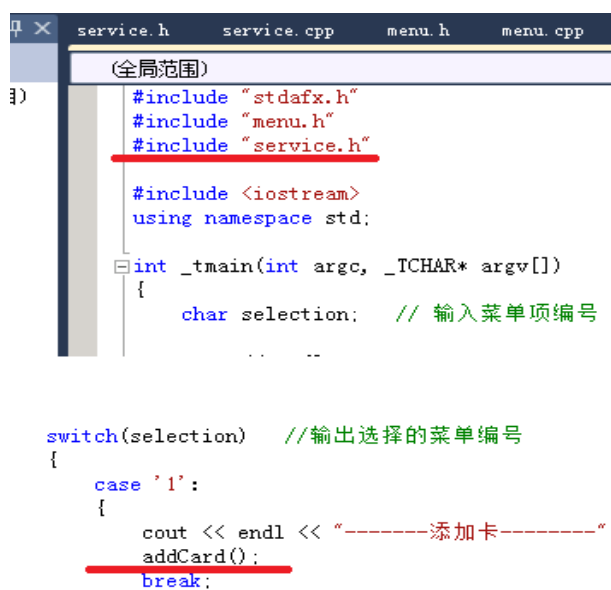


图 3

## 3.6 addCard(): 输入卡号、密码和金额

addCard()这个函数代码比较多，可以在编写过程中慢慢完善。

添加卡需要用户在界面上输入卡号、密码和新卡的金额，先完成这部分功能。

**注意：**在做完每一步工作时，都应该进行测试，只有得到正确的信息，才进行下一步的工作。

### 3.6.1 输入卡号功能实现

只有输入正确格式的卡号才能建立新卡，因此我们的程序必须保证卡号格式正确。如果

不正确就直接退出回到菜单模块，不用再去做后面的事情。实现代码如下。

```
1
2 void addCard(void)
3 {
4     string strNo;
5
6     while(true)
7     {
8         cout << "请输入卡号(长度为1~17): ";
9         cin >> strNo;
10        cin.clear();
11        cin.sync();
12        if(strNo.length() >= 17) // 卡号长度是否正确
13        {
14            cout << "卡号长度超过最大值!" << endl;
15
16            char ch='M';
17            while(ch != 'N' && ch != 'Y')
18            {
19                cout << "重新输入吗? (y/n) ";
20                cin >> ch;
21                ch = toupper(ch);
22                cin.clear();
23                cin.sync();
24            }
25            if(ch == 'N')
26            {
27                cout << "卡号输入错误!" << endl;
28                return ;
29            }
30        }
31        else
32        {
33            break;
34        }
35    }
36    cout << "卡号: " << strNo << endl;
37 }
38
```

其它需要修改的地方：

- 使用到了 string 数据类型，在 service.cpp 中添加语句  
`#include <string>`

进行各种情况测试，运行结果如图，表示功能无误，可以进行下一步骤。

```
请选择菜单项编号 (0~8) : 1
-----添加卡-----
请输入卡号<长度为1~17>: 123
卡号: 123
```

图 1

```
请输入卡号<长度为1~17>: 12345678901234567890
卡号长度超过最大值!
重新输入吗? (y/n) y
请输入卡号<长度为1~17>: 1234567890
卡号: 1234567890
```

图 2

```
请输入卡号<长度为1~17>: 12345678901234567890
卡号长度超过最大值!
重新输入吗? (y/n) n
卡号输入错误!
```

图 3

### 3.6.2 输入卡号功能优化

- 从代码中可以看出实现输入卡号的代码较长，如果再加上输入密码、金额等，代码行数就更多。
- 在整个系统中有很多地方都要输入卡号，如查询卡需要输入卡号。

基于以上 2 点考虑，我们用函数 `inputCardNo()` 实现输入卡号的功能。

`inputCardNo()` 需要将“卡号格式是否正确”和“正确卡号”这两种信息返回给调用它的函数，返回值只能带回一种信息，因此需要通过参数传回另一种信息。

通过参数将数据传回给调用它的函数有两种方式：指针参数和引用参数。本示例中采用引用参数的形式。函数格式如下：

```
bool inputCardNo(string& strNo)
```

返回值表示卡号格式是否正确，参数 `strNo` 存储输入的卡号。

输入卡号也是跟界面有关的，在示例中，我们将函数 `inputCardNo()` 放置在 `menu.cpp` 文件中。

`inputCardNo()` 函数的实现如下。

```
1
2  bool inputCardNo(string& strNo)
3  {
4      while(true)
5      {
6          cout << "请输入卡号(长度为1~17): ";
7          cin >> strNo;
```



```

8      cin.clear();
9      cin.sync();
10     if(strNo.length() >= 17) // 卡号长度是否正确
11     {
12         cout << "卡号长度超过最大值!" << endl;
13
14         char ch='M';
15         while(ch != 'N' && ch != 'Y')
16         {
17             cout << "重新输入吗? (y/n) ";
18             cin >> ch;
19             ch = toupper(ch);
20             cin.clear();
21             cin.sync();
22         }
23         if(ch == 'N')
24         {
25             return false;
26         }
27     }
28     else
29     {
30         break;
31     }
32 }
33 }
34

```

其它需要修改的地方：

- 在 menu.h 文件中要添加 inputCardNo() 函数声明，语句如下  
`bool inputCardNo(string& strNo);`
- 使用到了 string 数据类型，在 menu.cpp 和 menu.h 文件中添加语句  
`#include <string>`

### addCard()函数修改

由于我们将输入卡号功能作为一个函数，因此在 addCard() 函数中要将相应代码改为调用 inputCardNo() 函数。

```

1
2     void addCard(void)
3     {
4         string strNo;
5
6         if(inputCardNo(strNo))

```

```

7      {
8          cout << "卡号: " << strNo << endl;
9      }
10     else
11     {
12         cout << "输入的卡号格式不正确!" << endl;
13     }
14 }
15

```

其它需要修改的地方：

- 在 `service.cpp` 文件中要添加 `inputCardNo()` 函数声明，语句如下

```
#include "menu.h"
```

思考题：

1. 如果采用指针参数的形式来改写 `inputCardNo()` 函数，该如何修改？
2. 为什么添加 `inputCardNo()` 函数声明使用下面语句？

```
#include "menu.h"
```

### 3.6.3 输入卡号、密码和金额最终版本

参照输入卡号功能的编写过程，实现输入密码函数和输入金额函数，函数形式如下，具体代码见系统源码。

```

bool inputCardPwd(string& strPwd) { ..... }
bool inputCardBalance(float& fBalance) { ..... }

```

为了进一步代码优化和复用，本示例在 `menu.cpp` 中编写了函数 `inputNoPwd()`，它调用 `inputCardNo()` 和 `inputCardPwd()` 实现输入卡号和密码；编写了函数 `inputNoPwdBalance()`，它调用 `inputCardNo()`、`inputCardPwd()` 和 `inputCardBalance()` 实现输入卡号、密码和金额。

注意：在 `menu.h` 中添加函数声明。

```

1
2  bool inputNoPwd(string& strNo, string& strPwd)
3  {
4      if(!inputCardNo(strNo))
5      {
6          return false;
7      }
8
9      if(!inputCardPwd(strPwd))
10     {
11         return false;
12     }

```

```

13
14     return true;
15 }
16
17 bool inputNoPwdBalance(string& strNo, string& strPwd, float& fBalance)
18 {
19     if(!inputCardNo(strNo))
20     {
21         return false;
22     }
23
24     if(!inputCardPwd(strPwd))
25     {
26         return false;
27     }
28
29     if(!inputCardBalance(fBalance))
30     {
31         return false;
32     }
33
34     return true;
35 }
36

```

### 3.6.4 addCard()函数的修改

同时，我们需要进一步修改 addCard()函数。

```

1
2 void addCard(void)
3 {
4     string strNo;
5     string strPwd;
6     float fBalance;
7
8     if(inputNoPwdBalance(strNo, strPwd, fBalance))
9     {
10         cout << "卡号: " << strNo << endl;
11         cout << "密码: " << strPwd << endl;
12         cout << "金额: " << fBalance << endl;
13     }
14     else

```

```

15     {
16         cout << "输入的信息格式不正确，添加卡失败!" << endl;
17     }
18 }
19

```

至此，我们完成了在界面上输入卡号、密码和新卡金额的功能，接着我们用这些信息去构造一张卡，并将卡存储起来。

## 3.7 addCard(): 构造上机卡并存储到 vector

### 3.7.1 上机卡结构体

上机卡中包含了很多信息，因此我们使用结构体将卡的相关信息组织在一起。

一般在软件项目中，将用户自定义的结构体数据类型单独组织在一个文件中，本示例使用 `model.h` 文件来存放所有用户定义的结构体。`model.h` 文件内容如下。由于此结构体涉及到时间，所以需要将系统文件 `time.h` 引入。

```

1
2  #include <time.h>
3
4  // 定义卡信息结构体
5  struct Card
6  {
7      char aName[18];    // 卡号
8      char aPwd[8];      // 密码
9      int  nStatus;      // 卡状态(UNUSE-未上机; USING-正在上机; INVALID-已注销)
10     time_t tStart;     // 开卡时间, long
11     float fTotalUse;   // 累计金额
12     time_t tLast;     // 最后使用时间, long
13     int  nUseCount;    // 使用次数
14     float fBalance;    // 余额
15 };
16

```

**注意：**由于原结构体中：卡的截止时间（`tEnd`）和删除标识（`nDel`）在系统中没有起作用，因此在本示例中把这两个信息删除了。

### 3.7.2 vector 数据类型

#### 数组与 vector 的比较

- 数组定义的时候必须定义数组的元素个数，而 `vector` 不需要。

- 数组定义后的空间是固定的，不能改变；而 **vector** 要灵活得多，可再加或减。
- 不能将一个数组赋值给另一个数组；但是 **vector** 可以。

### vector 基本操作示例

- 头文件：#include <vector>
- 定义 vector 对象：vector<int> vec;
- 尾部插入数据：vec.push\_back(a);
- 使用下标访问元素：vec[0]等，下标是从 0 开始的。
- 使用迭代器访问元素  

```
vector<int>::iterator it;
for(it=vec.begin();it!=vec.end();it++)
    cout<<*it<<endl;
```
- 插入元素：vec.insert(vec.begin()+i,a);在第 i+1 个元素前面插入 a;
- 删除元素：  

```
vec.erase(vec.begin()+2);删除第 3 个元素
vec.erase(vec.begin()+i,vec.end()+j);删除区间[i,j-1];区间从 0 开始
```

## 3.7.3 在 \_tmain() 中定义 vector 和传递参数 vector

本例中使用容器 **vector** 存储上机卡。容器 **vector** 中存储了所有的上机卡，这也意味着菜单项的每个功能都要对这个 **vector** 进行操作，即这些函数都应该能访问（读写）这个 **vector**。为了实现这个目的，有 2 种方式：一是将这个 **vector** 定义为全局的；二是在 **\_tmain()** 函数中定义一个 **vector** 容器，将这个 **vector** 作为参数传给所有要用到 **vector** 的函数。

在良好的编程风格中，应尽量少用全局变量，因此本例中使用第 2 种方式，将 **vector** 作为参数传给所有要用到 **vector** 的函数。由于在一些函数中会对这个 **vector** 进行修改，为了能将修改的结果返回给调用它的函数，本例将 **vector** 参数定义为引用参数。

下图显示了在 **\_tmain()** 中如何定义容器 **vector**；在增加上机卡的功能中，如何将定义的容器 **vector** 传递给 **addCard()** 函数。

```
int _tmain(int argc, _TCHAR* argv[])
{
    char selection;    // 菜单项选择

    vector<Card> vec;

    cout << endl;
    cout << "★欢迎进入计费管理系统★" << endl;
    cout << endl;
```

```

switch(selection)
{
    case '1':
    {
        cout << endl << "-----开卡-----" << endl << endl;
        addCard(vec);
        break;
    }
    . . .
}

```

### 3.7.4 addCard()函数的完善

从前面的 addCard()函数实现的功能中，我们可以得到格式正确的卡号、密码和金额。在此我们通过调用函数 addNewCard()实现添加卡，并将操作结果返回，根据返回结果，显示不同的提示信息。addCard()函数代码如下。

```

1
2 void addCard(vector<Card> &vec)
3 {
4     string strNo;
5     string strPwd;
6     float fBalance;
7
8     if(inputNoPwdBalance(strNo, strPwd, fBalance))
9     {
10         int nResult;
11         nResult = addNewCard(strNo, strPwd, fBalance, vec);
12
13         if(nResult == FINDCARD)
14         {
15             cout << "此卡号已使用，添加卡失败!" << endl;
16         }
17         else if(nResult == SUCCESS)
18         {
19             cout << "卡添加卡成功!" << endl;
20         }
21         else
22         {
23             cout << "其它情况" << endl;
24         }
25     }
26     else
27     {
28         cout << "输入的信息格式不正确，添加卡失败!" << endl;
29     }
}

```

```
30 }
31
```

### 3.7.5 addNewCard()函数的实现

添加文件 **VectorCard.cpp** 和 **VectorCard.h**

在项目中添加文件 **VectorCard.cpp** 和 **VectorCard.h**，具体操作与前面“添加 **service.cpp** 和 **service.h**”类似。这个文件中存放与卡有关的具体操作，如添加卡，查询卡，修改卡等。

**addNewCard()函数思路**

在文件 **VectorCard.cpp** 中添加 **addNewCard()** 函数的定义，在 **VectorCard.h** 中添加 **addNewCard()** 函数的声明，具体操作与前面“添加 **addCard()** 函数”类似。

- **addNewCard()** 函数先查找此卡号在 **vector** 中是否存在。如果存在则不能添加卡；如果不存在，则用卡号，密码，金额和当前时间生成一张卡，再把卡放到容器 **vector** 中。返回值代表了不同的状态，用于在调用函数 **addCard()** 中显示不同的提示信息。
- 程序 5-6 行：调用函数 **cardIsExist()** 在 **vector** 中查找卡号是 **strNo** 的卡是否存在。如果不存在，返回值是 **NULL**；如果不存在，返回非 **NULL**。此处先放置一个空函数，下一步再完善。
- 函数 **cardIsExist()** 暂时为空函数，后面再实现。
- 程序 9-16 行：生成一张卡。
- 程序 18 行：把卡放入容器 **vector**。

```
1
2  int addNewCard(string strNo, string strPwd, float fBalance, vector<Card>& vec)
3  {
4      int nCardIndex = 0;
5      if(cardIsExist(strNo, nCardIndex, vec) != NULL)
6          return FINDCARD;
7
8      // 构造一张上机卡
9      Card card;
10     strcpy(card.aName, strNo.c_str());
11     strcpy(card.aPwd, strPwd.c_str());
12     card.fBalance = fBalance;
13     card.fTotalUse = card.fBalance;    // 添加新卡时，累计金额等于开卡金额
14     card.nStatus = UNUSE;             // 卡状态
15     card.nUseCount = 0;               // 使用次数
16     card.tStart = card.tLast = time(NULL); // 开卡时间，最后使用时间
17
18     vec.push_back(card);
19
20     return SUCCESS;
21 }
```

### 3.7.6 常量文件 global.h

卡的状态分为上机, 未使用和已销卡等, 可以用 3 个数字表示这 3 种状态。在编写程序中, 记忆某个数字是哪种状态很不方便 (后面还有更多的状态), 因此在项目开发中, 一般会将这些数字定义成常量, 将常量取个有意义的名字, 通过引用常量来使用这些数字。这种方式在需要修改数字所代表不同含义的时候, 也很方便, 只需修改常量定义, 而不用去修改引用常量的每一个地方。

在本例中所有的常量都在常量文件 `global.h` 中定义。

```

1
2  #ifndef GLOBAL_H_INCLUDED
3  #define GLOBAL_H_INCLUDED
4
5  #include <string>
6  using namespace std;
7
8  const string CARDPATH = "d:\\card.txt";    // 卡信息记录文件
9  const string BILLINGPATH = "d:\\billing.txt"; // 计费信息记录文件
10 const string MONEYPATH = "d:\\money.txt";   // 充值退费记录文件
11
12 const int TIMELENGTH = 20;                // 时间字符数组长度
13
14 const int SUCCESS      = 0;                // 操作成功
15 const int FINDCARD     = 1;                // 找到卡
16 const int NOFINDCARD  = 2;                // 没找到卡
17 const int NOMATCH      = 3;                // 卡号密码不匹配
18 const int ENOUGHMONEY = 4;                // 卡余额不足
19 const int NOFARE       = 5;                // 没有找到计费信息
20
21 const int USING        = 6;                // 卡状态: 正在上机
22 const int UNUSE        = 7;                // 卡状态: 没有上机
23 const int INVALID      = 8;                // 卡状态: 已经注销
24
25 const int NOSETTLEMENT = 0;                // 上机已结算
26 const int YESSETTLEMENT = 1;               // 上机未结算
27
28 const int UNIT         = 15;                // 最小收费单元(分钟)
29 const float CHARGE     = 0.5f;              // 每个计费单元收费(元)
30
31 #endif // GLOBAL_H_INCLUDED
32

```



注意：此文件中采用了 C++ 的常量定义方式。

### 3.7.7 cardIsExist ()函数的实现

在文件 VectorCard.cpp 中添加 cardIsExist() 函数的定义, 在 VectorCard.h 中添加 cardIsExist() 函数的声明, 具体操作与前面 “添加 addCard() 函数” 类似。

函数 Card\* cardIsExist(string strNo, int &nCardIndex, vector<Card>& vec) 通过遍历 vector 来查找卡号 strNo 是否被使用。如果使用, 则返回该卡的地址, 同时通过参数 nCardIndex 返回是第几张卡, 此数据在后面模块要使用; 如果没使用, 则返回 NULL, 表示可以生成新卡。

```
1
2 Card* cardIsExist(string strNo, int &nCardIndex, vector<Card>& vec)
3 {
4     vector<Card>::iterator it;
5
6     nCardIndex = 0;
7     for(it=vec.begin(); it!=vec.end(); it++)
8     {
9         if(strcmp(it->aName, strNo.c_str())==0)
10        {
11            return &(*it);
12        }
13        nCardIndex++;
14    }
15    return NULL;
16 }
17
```

#### 编写函数 displayCard()

新卡是否添加到了 vector 中, 需要验证, 因此我们编写一个 displayCard() 函数去遍历 vector, 将每次添加新卡后的结果, 即所有卡显示出来。

我们可以暂时在 “查询卡” 菜单功能实现显示所有卡的功能。按我们前面的设计安排, 在 service.cpp 中定义所有菜单功能实现的函数, 查询卡对应的是 queryCard() 函数, 因为是查询 vector 类型变量 vec 中的卡, 因此需要将变量 vec 作为参数传给 queryCard() 函数。再在 queryCard() 函数中调用 displayCard() 函数。

```
case '2':
{
    cout << endl << "-----查询-----" << endl << endl;
    queryCard(vec);
    break;
}
```

```

void queryCard(vector<Card> &vec)
{
    displayCard(vec);
}

```

```

1
2 void displayCard(vector<Card>& vec)
3 {
4     vector<Card>::iterator it;
5
6     cout << "卡号\t状态\t余额\t累计使用\t使用次数\t上次使用时间\n";
7     for(it=vec.begin();it!=vec.end();it++)
8     {
9         char aLastTime[TIMELENGTH] = {0};
10        timeToString(it->tLast, aLastTime);
11
12        cout << it->aName << "\t";
13        if(it->nStatus == USING)
14            cout << "上机\t";
15        else if(it->nStatus == UNUSE)
16            cout << "未上机\t";
17        else if(it->nStatus == INVALID)
18            cout << "注销\t";
19        else
20            cout << "错误\t";
21        cout << it->fBalance << "\t";
22        cout << it->fTotalUse << "\t\t";
23        cout << it->nUseCount << "\t\t";
24        cout << aLastTime << endl;
25    }
26 }
27

```

请选择菜单项编号 (0~8) : 2

-----查询卡-----

卡号	状态	余额	累计使用	使用次数	上次使用时间
1	未上机	100	100	0	2017-03-16 14:22
2	未上机	200	200	0	2017-03-16 14:22



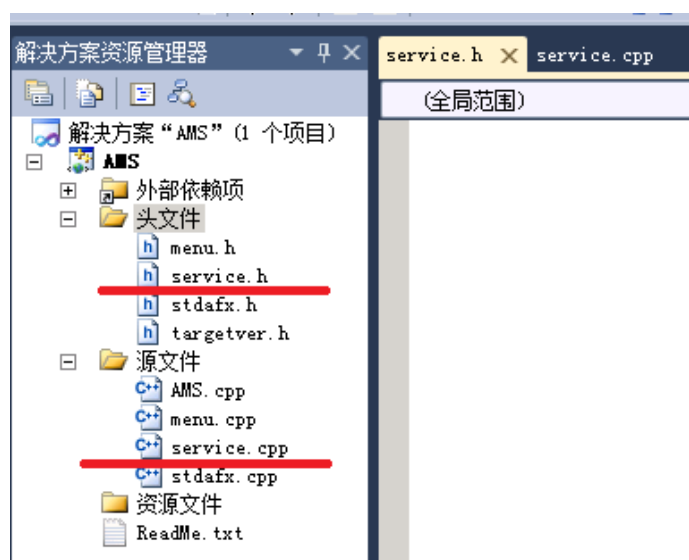
## 3.8 实验步骤（类版本）

良好的编程风格不应该在 `_tmain()` 函数中编写较多行的代码，因此我们将每个菜单项实现的功能用一个主函数来实现。对于比较复杂的功能（也会有很长的代码），可以细分为更多的子功能，每个子功能用一个独立函数实现，通过在主函数中调用子功能函数来实现各菜单项的功能。

在本次实验中，我们使用 `addCard()` 函数完成“添加卡”的功能，在此函数中根据需要再调用其它函数。

### 3.8.1 添加 `service.cpp` 和 `service.h`

前面说过主文件（`AMS.cpp`）中一般只放置 `_tmain()` 函数，因此我们将 `addCard()` 函数放置到另一个文件中。在本例中我们添加一个新文件 `service.cpp`，它所包含的函数都是实现菜单项各功能的主函数（详见表 1）。同时我们添加同名的头文件 `service.h`。



添加文件完成后，我们要做如下的一些准备工作：

- 根据 .net 编程环境编译要求，向 `service.cpp` 中添加下面语句  
`#include "stdafx.h"`  
由于一般都会用到标准的输入输出，所以同时添加下面语句  
`#include <iostream>`  
`using namespace std;`  
在 `service.cpp` 中添加 `addCard()` 空函数，为下一步编写做准备。  
见图 1，
- 在 `service.h` 中添加 `addCard()` 函数的声明，见图 2。  
注意：函数的声明后面的分号“;”不要漏掉。
- 在 `_tmain()` 函数调用 `addCard()` 函数，因此要在 `AMS.cpp` 中添加下面语句，见图 3。  
`#include "service.h"`

- 上述步骤完成后，编译无错误后能运行则继续后面步骤。

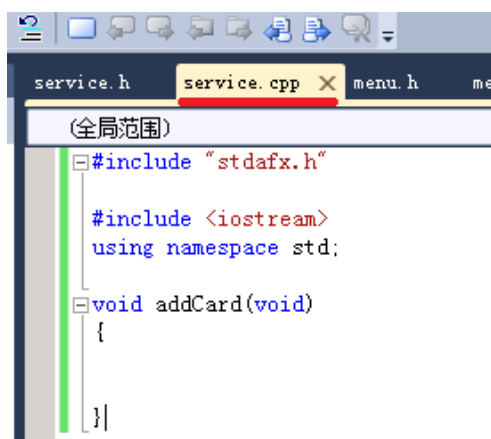


图 1

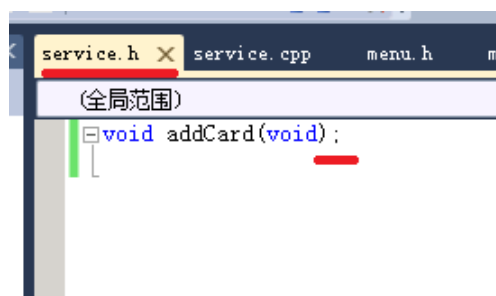


图 2



图 3

### 3.8.2 编写 addCard()函数

由于这个函数代码比较多，在编写过程中需要慢慢完善。添加卡需要用户在界面上输入卡号、密码和新卡的金额，先完成这部分功能。

**注意：在做完每一步工作时，都应该进行测试，只有得到正确的信息，才进行下一步的工作。**

#### 输入卡号功能

由于只有输入正确格式的卡号才能建立新卡，因此我们的程序必须保证卡号格式正确。

如果不正确就直接退出回到菜单模块，不用完成后面的功能。实现代码如下。

```
1
2 void addCard(void)
3 {
4     string strNo;
5
6     while(true)
7     {
8         cout << "请输入卡号(长度为1~17): ";
9         cin >> strNo;
10        cin.clear();
11        cin.sync();
12        if(strNo.length() >= 17) // 卡号长度是否正确
13        {
14            cout << "卡号长度超过最大值!" << endl;
15
16            char ch='M';
17            while(ch != 'N' && ch != 'Y')
18            {
19                cout << "重新输入吗? (y/n) ";
20                cin >> ch;
21                ch = toupper(ch);
22                cin.clear();
23                cin.sync();
24            }
25            if(ch == 'N')
26            {
27                cout << "卡号输入错误!" << endl;
28                return ;
29            }
30        }
31        else
32        {
33            break;
34        }
35    }
36    cout << "卡号: " << strNo << endl;
37 }
38
```

其它需要修改的地方：

- 使用到了 string 数据类型，在 service.cpp 文件中添加语句

```
#include <string>
```

进行各种情况测试，运行结果如图，表示功能无误，可以进行下一步骤。

```
请选择菜单项编号 (0~8) : 1
-----添加卡-----
请输入卡号<长度为1~17>: 123
卡号: 123
```

图 1

```
请输入卡号<长度为1~17>: 12345678901234567890
卡号长度超过最大值!
重新输入吗? (y/n) y
请输入卡号<长度为1~17>: 1234567890
卡号: 1234567890
```

图 2

```
请输入卡号<长度为1~17>: 12345678901234567890
卡号长度超过最大值!
重新输入吗? (y/n) n
卡号输入错误!
```

图 3

### 输入卡号功能优化

- 从代码中可以看出实现输入卡号的代码较长，如果再加上输入密码、金额等，代码行数就更多。
  - 在整个系统中有很多地方都要输入卡号，如查询卡需要输入卡号。
- 基于以上 2 点考虑，我们用函数 `inputCardNo()` 实现输入卡号的功能。

`inputCardNo()` 需要将“卡号格式是否正确”和“正确卡号”这两种信息返回给调用它的函数，返回值只能带回一种信息，因此需要通过参数传回另一种信息。

通过参数将数据传回给调用它的函数有两种方式：指针参数和引用参数。本实验中采用引用参数的形式。函数格式如下：

```
bool inputCardNo(string& strNo)
```

返回值表示卡号格式是否正确，参数 `strNo` 存储输入的卡号。

输入卡号也是跟界面有关的，在示例中，我们将函数 `inputCardNo()` 放置在 `menu.cpp` 文件中。

### `inputCardNo()` 函数

```
1
2  bool inputCardNo(string& strNo)
3  {
4      while(true)
5      {
6          cout << "请输入卡号(长度为1~17): ";
7          cin >> strNo;
```

```

8      cin.clear();
9      cin.sync();
10     if(strNo.length() >= 17) // 卡号长度是否正确
11     {
12         cout << "卡号长度超过最大值!" << endl;
13
14         char ch='M';
15         while(ch != 'N' && ch != 'Y')
16         {
17             cout << "重新输入吗? (y/n) ";
18             cin >> ch;
19             ch = toupper(ch);
20             cin.clear();
21             cin.sync();
22         }
23         if(ch == 'N')
24         {
25             return false;
26         }
27     }
28     else
29     {
30         break;
31     }
32 }
33 }
34

```

其它需要修改的地方：

- 在 menu.h 文件中要添加 inputCardNo() 函数声明，语句如下  
`bool inputCardNo(string& strNo);`
- 使用到了 string 数据类型，在 menu.cpp 和 menu.h 文件中添加语句  
`#include <string>`

### addCard()函数修改

由于我们将输入卡号功能作为一个函数，因此在 addCard() 函数中要将相应代码改为调用 inputCardNo() 函数。

```

1
2     void addCard(void)
3     {
4         string strNo;
5
6         if(inputCardNo(strNo))

```



```

7      {
8          cout << "卡号: " << strNo << endl;
9      }
10     else
11     {
12         cout << "输入的卡号格式不正确!" << endl;
13     }
14 }
15

```

其它需要修改的地方：

- 在 `service.cpp` 文件中要添加 `inputCardNo()` 函数声明，语句如下

```
#include "menu.h"
```

思考题：

3. 如果采用指针参数的形式来改写 `inputCardNo()` 函数，该如何修改？
4. 为什么添加 `inputCardNo()` 函数声明使用下面语句？

```
#include "menu.h"
```

### 3.8.3 输入密码、金额功能

参照输入卡号功能的编写过程，实现输入密码函数和输入金额函数，函数形式如下，具体代码见系统源码。

```

bool inputCardPwd(string& strPwd) { ..... }
bool inputCardBalance(float& fBalance) { ..... }

```

### 3.8.4 输入卡号、密码和新卡金额最终版本

为了进一步代码优化和复用，本实验在 `menu.cpp` 中编写了函数 `inputNoPwd()`，它调用 `inputCardNo()` 和 `inputCardPwd()` 实现输入卡号和密码；编写了函数 `inputNoPwdBalance()`，它调用 `inputCardNo()`、`inputCardPwd()` 和 `inputCardBalance()` 实现输入卡号、密码和金额。

注意：在 `menu.h` 中添加函数声明。

#### inputNoPwd()和 inputNoPwdBalance()函数代码清单

```

1
2  bool inputNoPwd(string& strNo, string& strPwd)
3  {
4      if(!inputCardNo(strNo))
5      {
6          return false;
7      }

```

```

8
9     if(!inputCardPwd(strPwd))
10    {
11        return false;
12    }
13
14    return true;
15 }
16
17 bool inputNoPwdBalance(string& strNo, string& strPwd, float& fBalance)
18 {
19     if(!inputCardNo(strNo))
20     {
21         return false;
22     }
23
24     if(!inputCardPwd(strPwd))
25     {
26         return false;
27     }
28
29     if(!inputCardBalance(fBalance))
30     {
31         return false;
32     }
33
34     return true;
35 }
36

```

同时，我们需要进一步修改 `addCard()` 函数。

#### 修改 `addCard()` 函数

```

1
2 void addCard(void)
3 {
4     string strNo;
5     string strPwd;
6     float fBalance;
7
8     if(inputNoPwdBalance(strNo, strPwd, fBalance))
9     {
10        cout << "卡号: " << strNo << endl;
11        cout << "密码: " << strPwd << endl;

```

```

12         cout << "金额: " << fBalance << endl;
13     }
14     else
15     {
16         cout << "输入的信息格式不正确, 添加卡失败!" << endl;
17     }
18 }
19

```

至此，我们完成了在界面上输入卡号、密码和新卡金额的功能，接着我们用这些信息去构造一张卡，并将卡存储起来。

### 3.8.5 上机卡结构体

上机卡中包含了很多信息，因此我们使用结构体将卡的相关信息组织在一起。

在软件项目中，一般将用户自定义的数据类型组织在一个独立文件中。本实验使用 `model.h` 文件来存放所有用户定义的结构体。`model.h` 文件内容如下。由于此结构体涉及到时间，所以需要将系统文件 `time.h` 引入。

```

1
2     #include <time.h>
3
4     // 定义卡信息结构体
5     struct Card
6     {
7         char aName[18];    // 卡号
8         char aPwd[8];      // 密码
9         int  nStatus;      // 卡状态 (UNUSE-未上机; USING-正在上机; INVALID-已注销)
10        time_t tStart;     // 开卡时间, long
11        time_t tEnd;       // 卡的截止时间, long
12        float fTotalUse;   // 累计金额
13        time_t tLast;      // 最后使用时间, long
14        int  nUseCount;    // 使用次数
15        float fBalance;    // 余额
16    };
17

```

#### 与时间有关的知识

C++提供了与时间处理相关的一些函数。在了解这些函数前，我们先要了解与时间处理有关的数据类型 `time_t` 和 `tm`，它们在 `time.h` 文件中定义。

`time_t` 是 64 位长整数，精确到秒，表示从 1970 年 1 月 1 日的零点开始到当前时间经过了多少秒。

`tm` 是结构体类型，如下所示。

```

1

```

```

2  struct tm
3  {
4      int tm_sec; /*秒, 0-59*/
5      int tm_min; /*分钟, 0-59*/
6      int tm_hour; /*小时, 0-23*/
7      int tm_mday; /*日, 1-31*/
8      int tm_mon; /*月, 0-11*/
9      int tm_year; /*年, 从1900至今已经多少年*/
10     int tm_wday; /*星期, 从星期日算起, 0-6*/
11     int tm_yday; /*天数, 0-365*/
12     int tm_isdst; /*日光节约时间的旗标*/
13 };
14

```

常见时间函数：

- **time\_t time(time\_t\* t);** // 取得从 1970 年 1 月 1 日的零点至今的秒数  
time\_t t = time(NULL); // 获取本地时间
- **struct tm \*localtime(const time\_t \*clock);** // 将长整数时间转换为结构体时间，从中  
// 得到“年月日，星期，时分秒”等信息  
struct tm\* current\_time = localtime(&t);
- **time\_t mktime(struct tm\* timeptr);** // 将 struct tm 结构的时间转换为长整数  
// 从 1970 年至这个时间点的秒数

由于一般用户能够接受的时间是字符串“2017 年 3 月 8 日 15 时 30 分”这样的形式，因此我们需要编写自己的时间处理函数，将结构体 tm 中相关信息取出来组合成这样的字符串。在本例中，用 tool.cpp 文件存放与时间处理有关的函数，tool.h 文件存放函数的声明。

#### 时间处理函数代码清单

```

1  #include <time.h> // 包含时间类型头文件
2  #include <stdio.h> // 包含sscanf()函数头文件
3
4  void timeToString(time_t t, char* pBuf)
5  {
6      struct tm * timeinfo;
7
8      timeinfo = localtime(&t);
9      strftime(pBuf, 20, "%Y-%m-%d %H:%M", timeinfo);
10 }
11
12 time_t stringToTime(char* pTime)
13 {
14     struct tm tml;
15     time_t timel;
16

```

```

17     sscanf(pTime, "%d-%d-%d %d:%d",&tml.tm_year, &tml.tm_mon,
18           &tml.tm_mday, &tml.tm_hour, &tml.tm_min);
19
20     tml.tm_year -= 1900; // 年份为从1900年开始
21     tml.tm_mon -= 1;    // 月份为0~11
22     tml.tm_sec = 0;
23     tml.tm_isdst = -1;
24
25     time1 = mktime(&tml);
26
27     return time1;
28 }
29

```

- `size_t strftime( char *str, size_t maxsize, const char *fmt, struct tm *time )` 函数  
功能：将时间格式化，或者说格式化一个时间字符串。即将时间 `time` 格式化为 `fmt` 样式的字符串。  
fmt:  
%Y, 用 CCYY 表示的年（如：2004）  
%m, 月份 (1-12)  
%d, 月中的第几天(1-31)  
%H, 小时, 24 小时格式 (0-23)  
%M, 分钟(0-59)
- `int sscanf( const char *str,const char * format,.....)`  函数  
功能：将字符串 `str` 根据参数 `format` 字符串来转换并格式化数据。格式转换形式请参考 `scanf()`。转换后的结果存于对应的参数内。

### 3.8.6 卡在 vector 中的存储

我们用类 `CardVector` 来封装存储上机卡的成员变量（容器 `vector`）和操作上机卡的成员函数。因此我们先要向 `project` 中添加一个类，.net 会自动将类的定义与实现分开。如图所示。

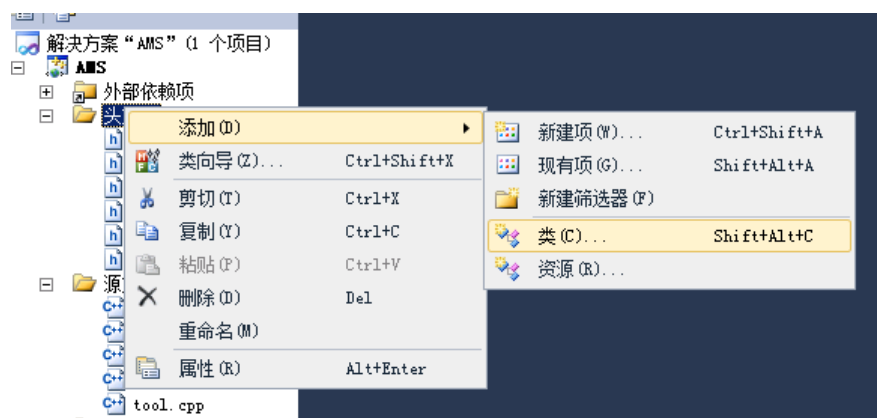


图 1

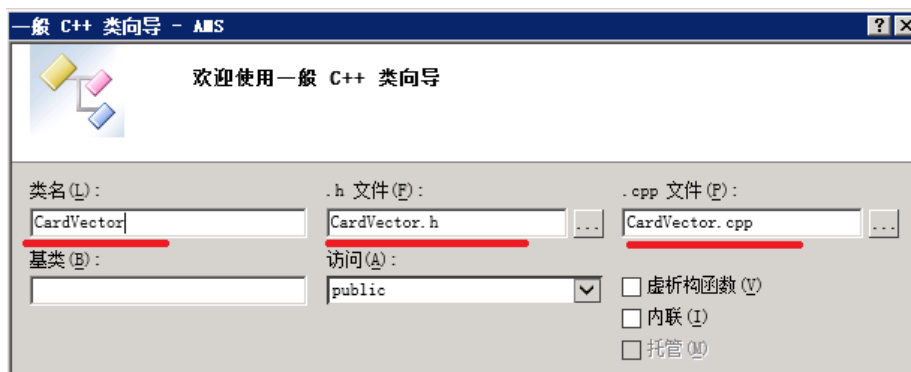


图 2

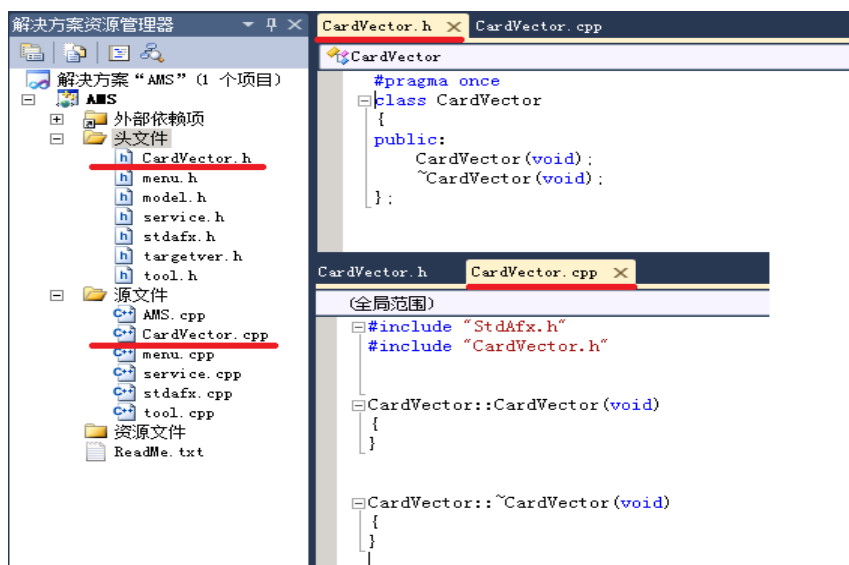


图 3

在类 `CardVector` 中，定义一个 `vector` 类型的容器，用来存放所有的卡。根据面向对象程序设计的封装要求，我将这个容器设置为 `private` 的访问权限，因此我们也要编写一些 `public` 函数来操作容器中的卡片。首先要实现的是类的成员函数 `addNewCard()`，它实现向容器中添加卡的功能。

### CardVector 类定义 (.h 文件)

```

1
2  #pragma once
3
4  #include <string>
5  #include <vector>
6  using namespace std;
7
8  #include "model.h"
9

```

```

10  class CardVector
11  {
12  public:
13      CardVector(void);
14      ~CardVector(void);
15      int addNewCard(string strNo, string strPwd, float fBalance);
16      Card* cardIsExist(string strNo, int &nCardIndex);
17
18  private:
19      vector<Card> vec;
20  };

```

### CardVector 类实现 (.cpp 文件)

```

1
2  #include "StdAfx.h"
3  #include "CardVector.h"
4  #include "global.h"
5
6  CardVector::CardVector(void)
7  {
8  }
9
10 CardVector::~~CardVector(void)
11 {
12 }
13
14 int CardVector::addNewCard(string strNo, string strPwd, float fBalance)
15 {
16     int nCardIndex = 0;
17     if(cardIsExist(strNo, nCardIndex) != NULL)
18         return FINDCARD;
19
20     Card card;
21     struct tm* startTime;    // 开卡时间
22
23     strcpy(card.aName, strNo.c_str());
24     strcpy(card.aPwd, strPwd.c_str());
25     card.fBalance = fBalance;
26     // 添加新卡时, 累计金额等于开卡金额
27     card.fTotalUse = card.fBalance;
28     card.nStatus = UNUSE;    // 卡状态
29     card.nUseCount = 0;    // 使用次数
30     // 开卡时间, 最后使用时间都默认为当前时间

```

```

31     card.tStart = card.tLast = time(NULL);
32
33     vec.push_back(card); // 将新卡放进vector中
34
35     return SUCCESS;
36 }
37
38 Card* CardVector::cardIsExist(string strNo, int &nCardIndex)
39 {
40     return NULL;
41 }
42

```

**注意：**由于原结构体中：卡的截止时间（tEnd）和删除标识（nDel）在系统中没有起作用，因此在本实验中把这两个信息删除了。

### addNewCard()函数思路

- addNewCard()函数先通过卡号查找此卡在 vector 中是否存在。如果存在则不能添加卡；如果不存在，则用卡号，密码，金额和当前时间生成一张卡，再把卡放到容器 vector 中。返回值代表了不同的状态，用于在调用函数中显示不同的提示信息。
- 程序 16-18 行：调用成员函数 cardIsExist()在 vector 中查找卡号是 strNo 的卡是否存在。如果不存在，返回值是 NULL；如果不存在，返回非 NULL。此处先放置一个空函数，下一步再完善。
- 成员函数 cardIsExist()暂时为空函数，后面再实现。
- 程序 20-31 行：生成一张卡。
- 程序 33 行：把卡放入容器 vector。

### 数组与 vector 的比较

- 数组定义的时候必须定义数组的元素个数，而 vector 不需要。
- 数组定义后的空间是固定的，不能改变；而 vector 要灵活得多，可再加或减。
- 一个数组不能用另一个数组初始化，也不能将一个数组赋值给另一个数组；但是 vector 可以。

### vector 基本操作示例

- 头文件：#include <vector>
- 定义 vector 对象：vector<int> vec;
- 尾部插入数据：vec.push\_back(a);
- 使用下标访问元素：vec[0]等，下标是从 0 开始的。
- 使用迭代器访问元素  

```

vector<int>::iterator it;
for(it=vec.begin();it!=vec.end();it++)
    cout<<*it<<endl;

```
- 插入元素：vec.insert(vec.begin()+i,a);在第 i+1 个元素前面插入 a;
- 删除元素：



vec.erase(vec.begin()+2);删除第 3 个元素

vec.erase(vec.begin()+i,vec.end()+j);删除区间[i,j-1];区间从 0 开始

### 常量文件 global.h

卡的状态分为上机，未使用和已销卡等，可以用 3 个数字表示这 3 种状态。在编写程序中，记忆某个数字是哪种状态很不方便（后面还有更多的状态），因此在项目开发中，一般会将这些数字定义成常量，将常量取个有意义的名字，通过引用常量来使用这些数字。这种方式在需要修改数字所代表不同含义的时候，也很方便，只需修改常量定义，而不用去修改引用常量的每一个地方。

在本例中所有的常量都在常量文件 global.h 中定义。

```
1
2  #ifndef GLOBAL_H_INCLUDED
3  #define GLOBAL_H_INCLUDED
4
5  #include <string>
6  using namespace std;
7
8  const string CARDPATH = "d:\\card.txt";    // 卡信息记录文件
9  const string BILLINGPATH = "d:\\billing.txt"; // 计费信息记录文件
10 const string MONEYPATH = "d:\\money.txt";   // 充值退费记录文件
11
12 const int TIMELENGTH = 20;    // 时间字符数组长度
13
14 const int SUCCESS      = 0;    // 操作成功
15 const int FINDCARD     = 1;    // 找到卡
16 const int NOFINDCARD   = 2;    // 没找到卡
17 const int NOMATCH      = 3;    // 卡号密码不匹配
18 const int ENOUGHMONEY  = 4;    // 卡余额不足
19 const int NOFARE       = 5;    // 没有找到计费信息
20
21 const int USING        = 6;    // 卡状态：正在上机
22 const int UNUSE        = 7;    // 卡状态：没有上机
23 const int INVALID      = 8;    // 卡状态：已经注销
24
25 const int NOSETTLEMENT = 0;    // 上机已结算
26 const int YESSETTLEMENT = 1;   // 上机未结算
27
28 const int UNIT         = 15;    // 最小收费单元(分钟)
29 const float CHARGE     = 0.5f;  // 每个计费单元收费(元)
30
31 #endif // GLOBAL_H_INCLUDED
32
```

注意：此文件中采用了 C++ 的常量定义方式。

### 3.8.7 ASM.cpp 文件中函数\_tmain()的修改

在函数\_tmain()中，通过调用函数 addCard()实现添加上机卡。上机卡存放在哪儿呢？前面我们讲过放置在容器 vector 中，而 vector 定义在类 CardVector 中，因此我们需要使用类 CardVector 去定义一个对象（变量）cardVec，让上机卡存放在对象 cardVec 中，确切说，存放在对象 cardVec 的成员变量 vec 中。

```
int _tmain(int argc, _TCHAR* argv[])
{
    char selection; // 输入菜单项编号
    CardVector cardVec;

    cout << endl;
    cout << "★欢迎进入计费管理系统★" << endl;
    cout << endl;

    do
    {
        outputMenu();

        selection = 'a'; // 初始化选择的菜单项编号为'a'
        cin >> selection; // 输入菜单项编号

        cin.clear();
        cin.sync();

        switch(selection) //输出选择的菜单编号
        {
            case '1':
            {
                cout << endl << "-----添加卡-----" << endl << endl;
                addCard(cardVec);
                break;
            }
            case '2': // 查询卡
```

由于每个菜单项的功能都要操作变量 cardVec，因此我们在函数\_tmain()中定义变量 cardVec，然后作为参数传给每个菜单项下面的调用函数，如 addCard(cardVec)。

### 3.8.8 service.cpp 文件中函数 addCard ()的修改

```
1
2 void addCard(CardVector &cardVec)
3 {
4     string strNo;
5     string strPwd;
6     float fBalance;
7
8     if(inputNoPwdBalance(strNo, strPwd, fBalance))
```

```

9      {
10         int nResult;
11         nResult = cardVec.addNewCard(strNo, strPwd, fBalance);
12
13         if(nResult == FINDCARD)
14         {
15             cout << "此卡号已使用，添加卡失败！" << endl;
16         }
17         else if(nResult == SUCCESS)
18         {
19             cout << "卡添加卡成功！" << endl;
20         }
21         else
22         {
23             cout << "其它情况" << endl;
24         }
25     }
26     else
27     {
28         cout << "输入的信息格式不正确，添加卡失败！" << endl;
29     }
30 }
31

```

#### addCard()函数思路

- 在 addCard()函数中，通过调用 inputNoPwdBalance()函数获得卡号，密码和金额；再调用类 CardVector 中的成员函数 addNewCard()来生成一张新卡，并添加到容器 vector 中；最后根据添加卡返回的结果，显示不同的提示信息。
- 在 addCard ()函数中对 cardVec 对象的修改，即添加新卡后的对象，应该返回给调用它的函数中，也就是 \_tmain()中。可以使用 return 语句返回修改后的 cardVec 对象，但为了效率等，一般 return 的值是 C++的基础数据类型。因此在参数中返回修改后的 cardVec 对象，在本例中使用引用参数的方式，见程序第 2 行。
- 由于不能直接操作类 CardVector 中的成员变量 vec(private)，因此需要调用类 CardVector 中的成员函数 addNewCard()来添加新的上机卡，见程序第 11 行。

### 3.8.9 编写成员函数 cardIsExist()

成员函数 Card\* cardIsExist(string strNo, int &nCardIndex)通过遍历 vector 来查找卡号 strNo 是否被使用。如果使用，则返回该卡的地址，同时通过参数 nCardIndex 返回是第几张卡，此数据在后面模块要使用；如果没使用，则返回 NULL，表示可以生成新卡。

```

1
2 Card* CardVector::cardIsExist(string strNo, int &nCardIndex)
3 {

```

```

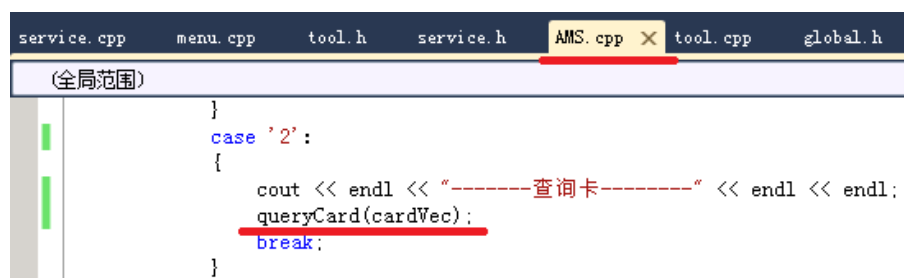
4     vector<Card>::iterator it;
5
6     nCardIndex = 0;
7     for(it=vec.begin();it!=vec.end();it++)
8     {
9         if(strcmp(it->aName, strNo.c_str())==0)
10        {
11            return &(*it);
12        }
13        nCardIndex++;
14    }
15
16    return NULL;
17 }
18

```

### 3.8.10编写成员函数 displayCard()

新卡是否添加到了 vector 中，需要验证。vector 封装在类 CardVector 中，因此我们编写一个 displayCard()成员函数去遍历 vector，将每次添加新卡后的结果，即所有卡显示出来。

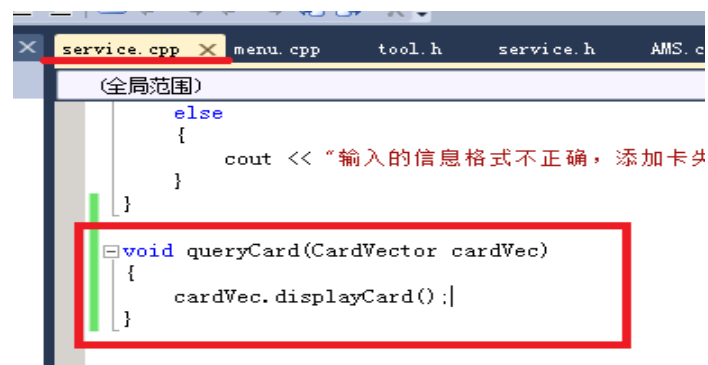
我们可以暂时在“查询卡”菜单功能实现显示所有卡的功能。按我们前面的设计安排，在 service.cpp 中定义所有菜单功能实现的函数，查询卡对应的是 queryCard()函数，因为是查询 CardVector 类对象 cardVec 中的卡，因此需要将对象 cardVec 作为参数传给 queryCard()函数。再在 queryCard()函数中调用 CardVector 类 displayCard()成员函数。



```

service.cpp  menu.cpp  tool.h  service.h  AMS.cpp  tool.cpp  global.h
(全局范围)
}
case '2':
{
    cout << endl << "-----查询卡-----" << endl << endl;
    queryCard(cardVec);
    break;
}

```



```

service.cpp  menu.cpp  tool.h  service.h  AMS.c
(全局范围)
else
{
    cout << "输入的信息格式不正确，添加卡失败" << endl;
}
}

void queryCard(CardVector cardVec)
{
    cardVec.displayCard();
}

```

```

1
2 void CardVector::displayCard()
3 {
4     vector<Card>::iterator it;
5
6     cout << "卡号\t状态\t余额\t累计使用\t使用次数\t上次使用时间\n";
7     for(it=vec.begin();it!=vec.end();it++)
8     {
9         char aLastTime[TIMELENGTH] = {0};
10        timeToString(it->tLast, aLastTime);
11
12        cout << it->aName << "\t" ;
13        if(it->nStatus == USING)
14            cout << "上机\t";
15        else if(it->nStatus == UNUSE)
16            cout << "未上机\t";
17        else if(it->nStatus == INVALID)
18            cout << "注销\t";
19        else
20            cout << "错误\t";
21        cout << it->fBalance << "\t";
22        cout << it->fTotalUse << "\t\t";
23        cout << it->nUseCount << "\t\t";
24        cout << aLastTime << endl;
25    }
26 }
27

```

请选择菜单编号 (0~8) : 2

-----查询卡-----

卡号	状态	余额	累计使用	使用次数	上次使用时间
1	未上机	100	100	0	2017-03-16 14:22
2	未上机	200	200	0	2017-03-16 14:22

## 4 文件存储管理

### 4.1 实验目的

- 1) 掌握文件读写的方法。

### 4.2 实验内容

- 1) 在系统启动过程中，从文本文件 `card.txt` 中读入所有上机卡信息，每张上机卡片信息存储在一个结构体变量中；所有的上机卡存储在 `vector` 容器数组中。
- 2) 在系统退出过程中，将 `vector` 容器数组中所有的上机卡信息存储到文本文件 `card.txt` 中。实现卡信息的更新能永久保存

### 4.3 文件操作知识

前面的实验中，上机卡的信息保存在容器 `vector` 中，`vector` 容器是内存中的变量，当系统退出后，添加的数据就会消失，不能永久保存。要想实现永久保存，必须将卡的信息写入文件中。

#### 文本文件和二进制文件

文本文件保存的是可读的字符（ASCII 码），而二进制文件保存的是二进制数据。

#### 文件打开方式：文本方式与二进制方式

在默认情况下，文件用文本方式打开。

两者的区别在于处理换行符：在以文本模式输出时，若遇到换行符“`\n`”(十进制为 10)则自动扩充为回车换行符(十进制为 13 和 10)；以文本模式输入时，则执行相反操作。

注意：数据存储的是 ASCII 码还是二进制，这个是由文件的读写方式确定，而与文件打开方式无关。

#### 文件打开方式和读写方式

- 打开方式是在定义文件流时指定的。如下面语句中  
`ofstream ofile "d:\\text.bin", ios::binary;`  
`ios::binary` 就是用于指定文件的打开方式是二进制方式。
- 读写方式是由文件采用的读写方法确定的。如  
`read()`和 `write()`方法是采用二进制读写数据。  
`get()`和 `put()`方法是采用 ASCII 码读写数据。
- 用户在存取文件时应该始终以相同的方式打开。
- 文件读写时，怎样写进去，就应该以相同的方式读出来。

#### 文件流

头文件：`#include <fstream>`

提供了三个类，用来实现 c++对文件的操作。

fstream（读写文件）

ifstream（读文件）

ofstream（写文件）

### 按二进制方式读写文件

write ( char \* buffer, streamsize size );

read ( char \* buffer, streamsize size );

参数 buffer 是一块内存的地址，用来存储或读出数据。

参数 size 是一个整数值，表示要从缓存（buffer）中读出或写入的字符数。

### “<<”和“>>”运算符

使用“<<”和“>>”运算符向文件中读写基本数据类型的数据时，是以文本方式操作文件，即将数据转换为字符串储存在文件中。

## 4.4 上机卡的文件存储

前面的实验中，上机卡的信息保存在容器 vector 中，vector 容器是内存中的变量，当系统退出后，添加的数据就会消失，不能永久保存。要想实现永久保存，必须将卡的信息写入文件中。

### 4.4.1 文件操作

#### 文本文件和二进制文件

文本文件保存的是可读的字符（ASCII 码），而二进制文件保存的是二进制数据。

#### 文件打开方式：文本方式与二进制方式

在默认情况下，文件用文本方式打开。

两者的区别在于处理换行符：在以文本模式输出时，若遇到换行符“\n”(十进制为 10)则自动扩充为回车换行符(十进制为 13 和 10)；以文本模式输入时，则执行相反操作。

注意：数据存储的是 ASCII 码还是二进制，这个是由文件的读写方式确定，而与文件打开方式无关。

#### 文件打开方式和读写方式

- 打开方式是在定义文件流时指定的。如下面语句中  
ofstream ofile "d:\\text.bin", ios::binary);  
ios::binary 就是用于指定文件的打开方式是二进制方式。
- 读写方式是由文件采用的读写方法确定的。如  
read()和 write()方法是采用二进制读写数据。

get()和 put()方法是采用 ASCII 码读写数据。

- 用户在存取文件时应该始终以相同的方式打开。
- 文件读写时，怎样写进去，就应该以相同的方式读出来。

## 文件流

头文件：#include <fstream>

提供了三个类，用来实现 c++对文件的操作。

fstream（读写文件）

ifstream（读文件）

ofstream（写文件）

## 按二进制方式读写文件

write ( char \* buffer, streamsize size );

read ( char \* buffer, streamsize size );

参数 buffer 是一块内存的地址，用来存储或读出数据。

参数 size 是一个整数值，表示要从缓存（buffer）中读出或写入的字符数。

## “<<”和“>>”运算符

使用“<<”和“>>”运算符向文件中读写基本数据类型的数据时，是以文本方式操作文件，即将数据转换为字符串储存在文件中。

## 4.4.2 向文件中写入新建的上机卡

每次添加新上机卡时，都是在文件末尾追加这张卡，因此只需在打开文件时使用参数 ios::app，即采用追加模式写文件即可。同时为了节省存储空间，我们都使用二进制方式打开和读写文件。实现代码如下。

```
1
2  bool saveCard(const Card* pCard, const char* pPath)
3  {
4      ofstream ofile(pPath, ios::binary | ios::app);
5
6      ofile.write((char*)pCard, sizeof(Card));
7      ofile.close();
8
9      return true;
10 }
11
```

在 addNewCard()函数中做如下修改。



```

// 开卡时间，最后使用时间都默认为当前时间
card.tStart = card.tLast = time(NULL);

vec.push_back(card);

saveCard(&card, CARDPATH);

return SUCCESS;

```

### 4.4.3 系统启动时从文件中恢复所有上机卡

在系统启动时，需要从文件中恢复所有上机卡信息到容器 `vector` 中，因此我们可以编写一个 `CardVectorInit()` 函数实现此功能，在 `_tmain()` 函数中，定义容器 `vector` 后，调用此函数对 `vector` 进行初始化，即在 `_tmain()` 函数中定义对象 `cardVec` 后执行恢复操作。

```

int _tmain(int argc, _TCHAR* argv[])
{
    char selection;    // 菜单项选择

    vector<Card> vec;
    CardVectorInit(CARDPATH, vec);

    cout << endl;
    cout << "★欢迎进入计费管理系统★" << endl;
    cout << endl;
}

```

### 4.4.4 CardVectorInit()函数的实现

```

1
2 void CardVectorInit(const string filename, vector<Card>& vec)
3 {
4     ifstream ifile(filename);
5     Card card;
6
7     if(!ifile.is_open())
8     {
9         return;
10    }
11
12    while(1)
13    {
14        ifile.read((char*)&card, sizeof(Card));
15
16        if(ifile.eof())

```

```

17         {
18             break;
19         }
20
21         vec.push_back(card);
22     }
23 }
24

```

至此，上机卡的添加、存储的功能就已经比较完善了。

## 4.5 实验步骤（类的实现方法）

### 4.5.1 向文件中写入新建的上机卡

每次添加新上机卡时，都是在文件末尾追加这张卡，因此只需在打开文件时使用参数 `ios::app`，即采用追加模式写文件即可。同时为了节省存储空间，我们都使用二进制方式打开和读写文件。实现代码如下。

```

1
2     bool CardVector::saveCard(const Card* pCard, const char* pPath)
3     {
4         ofstream ofile(pPath, ios::binary | ios::app);
5
6         ofile.write((char*)pCard, sizeof(Card));
7         ofile.close();
8
9         return true;
10    }
11

```

在 `CardVector::addNewCard()` 成员函数中做如下修改。

```

// 开卡时间，最后使用时间都默认为当前时间
card.tStart = card.tLast = time(NULL);

vec.push_back(card);

saveCard(&card, CARDPATH);

return SUCCESS;

```

## 一、系统启动时从文件中恢复所有上机卡

在系统启动时，需要从文件中恢复所有上机卡信息到容器 `vector` 中，而容器 `vector` 是类 `CardVector` 中的成员变量，因此我们可以在类 `CardVector` 的构造函数中执行此操作，即在 `_tmain()` 函数中定义对象 `cardVec` 时执行恢复操作。

```
int _tmain(int argc, _TCHAR* argv[])
{
    char selection; // 输入菜单项编号
    CardVector cardVec(CARDPATH);

    cout << endl;
    cout << "★欢迎进入计费管理系统★" << endl;
    cout << endl;
```

修改类 `CardVector` 的构造函数如下所示。

```
1
2 CardVector::CardVector(const string filename)
3 {
4     ifstream ifile(filename);
5     Card card;
6
7     if(!ifile.is_open())
8     {
9         return;
10    }
11
12    while(1)
13    {
14        ifile.read((char*)&card, sizeof(Card));
15        if(ifile.eof())
16        {
17            break;
18        }
19        vec.push_back(card);
20    }
21 }
22
```

至此，上机卡的添加、存储的功能就已经比较完善了。

## 5 “开卡”功能（链表）

在上次实验中，上机卡是存储在 `vector` 中。在本次实验中，在前面编写的实验代码基础上，实现用链表来存储上机卡。

在本例中，采用的是单向链表，在程序运行过程中，用 `new` 运算符动态创建的。

由于我们将对 `vector` 的操作封装在了类 `CardVector` 中，因此，当我们将 `vector` 改为链表时，我们只需修改类 `CardVector` 即可。（这就是面向对象程序设计的优点）

### 5.1 表头指针和表尾指针

对链表的访问一般是从表头开始，先将链表指针移动到要访问的节点，然后对节点进行访问。因此我们在类 `CardVector` 中，将 `vector` 更换成链表的表头指针。由于我们实现开卡功能时需要在链表的末尾插入一个节点，为了操作简单，在这里我们也定义了一个表尾指针。如下图所示。

```
class CardVector
{
public:
    CardVector(const string filename);
    ~CardVector(void);
    int addNewCard(string strNo, string strPwd, float fBalance);
    Card* cardIsExist(string strNo, int &nCardIndex);
    void displayCard();
    bool saveCard(const Card* pCard, const string pPath);

private:
    //vector<Card> vec;
    CardNode *pCardNodeHead, *pCardNodeTail;
};
```

### 5.2 链表的初始化

上次实验是用文件中所有卡去初始化 `vector`，在链表中，我们需要改为初始化链表，即构造一个链表，用前面定义的表头指针和表尾指针分别指向链表的头和尾。

```
1
2 CardVector::CardVector(const string filename)
3 {
4     pCardNodeHead = NULL;
5     pCardNodeTail = NULL;
6
7     CardNode *pCardNode;
8     Card card;
9
```

```
10     ifstream cardfile(filename);
11     if(!cardfile.is_open())
12     {
13         return ;
14     }
15
16     while(1)
17     {
18         cardfile.read((char*)&card, sizeof(Card));
19
20         if(cardfile.eof())
21         {
22             break;
23         }
24
25         pCardNode = new CardNode;
26         pCardNode->data = card;
27
28         if(pCardNodeHead == NULL)
29         {
30             pCardNodeHead = pCardNode;
31             pCardNodeTail = pCardNode;
32         }
33         else
34         {
35             pCardNodeTail->next = pCardNode;
36             pCardNodeTail = pCardNode;
37         }
38     }
39
40     pCardNodeTail->next = NULL;
41
42     cardfile.close();
43 }
44
```

## 5.3 修改 addNewCard()成员函数

```
//vec.push_back(card); // 将新卡添加到vector中
```

```
CardNode* pCardNode = new CardNode;
pCardNode->data = card;

if(pCardNodeHead == NULL)
{
    pCardNodeHead = pCardNode;
    pCardNodeTail = pCardNode;
}
else
{
    pCardNodeTail->next = pCardNode;
    pCardNodeTail = pCardNode;
}

pCardNodeTail->next = NULL;
```

```
saveCard(&card, CARDPATH); // 将新卡保存到文件中
```

## 5.4 修改 cardIsExist ()成员函数

```
1
2 Card* CardVector::cardIsExist(string strNo, int &nCardIndex)
3 {
4     CardNode *pCardNode = pCardNodeHead;
5
6     nCardIndex = 0;
7     while(pCardNode != NULL)
8     {
9         if(strcmp(pCardNode->data.aName, strNo.c_str())==0)
10        {
11            return &(pCardNode->data);
12        }
13        pCardNode = pCardNode->next;
14        nCardIndex++;
15    }
16
17    return NULL;
18 }
19
```

## 5.5 修改 cardDisplay()成员函数

```
1
2 void CardVector::displayCard()
3 {
4     if(pCardNodeHead == NULL)
5     {
6         cout << endl << endl << "一张上机卡都没有!" << endl << endl;
7         return;
8     }
9
10    cout << "卡号\t状态\t余额\t累计使用\t使用次数\t上次使用时间" << endl;
11
12    CardNode* pCur = pCardNodeHead;
13    while(pCur != NULL)
14    {
15        char aLastTime[TIMELENGTH] = {0};
16        timeToString((pCur->data).tLast, aLastTime);
17        cout << pCur->data.aName << "\t" ;
18        if(pCur->data.nStatus == USING)
19            cout << "上机\t";
20        else if(pCur->data.nStatus == UNUSE)
21            cout << "未上机\t";
22        else if(pCur->data.nStatus == INVALID)
23            cout << "注销\t";
24        else
25            cout << "错误\t";
26        cout << pCur->data.fBalance << "\t";
27        cout << pCur->data.fTotalUse << "\t\t"
28            << pCur->data.nUseCount << "\t\t" << aLastTime << endl;
29
30        pCur = pCur->next;
31    }
32 }
33
```