**北京科技大学实验报告**

学院：计通学院 专业：通信工程 班级：通信1701

姓名： 胡成成 学号：41724260 实验日期：2018年 3月 6-8 日

**实验名称：个人银行账户管理系统**

个人银行账户管理系统

**实验目的：**

用C++实现一个银行账户系统，并完成相关功能的实现：账户存取款，利息计算等。同时掌握创建过程中运用的C++知识及思想，并且熟悉优化过程中的处理办法，达到理论运用到实践的目的。

Step1：建立完善的储蓄账户；

step2：改进储蓄账户，进行日期的处理；

step3：增加信用账户；

step4：管理多个不同的账户；

step5、6：增加并且改进动态只能增添账户的功能；

step7、8：增加保存指令和异常处理的功能。

**实验仪器：**

计算机：ThinkPad

CPU：Intel(R) Core(TM) i7-6498DU @2.50GHz 2.60GHz

内存：8GB

硬盘：1TB

显卡：AMD Radeon R7 M360

操作系统：Windows 10

编译器：TDM-GCC 4.9.2 64-bit Release

集成开发环境:Dev-C++ Version 5.11

**实验原理：**

1、我们首先需要建立简单的储蓄账户，明确类中有哪些数据成员与成员函数。选择正确的计算利息的方法。对数据进行合理的四舍五入处理。

2、我们需要完善储蓄账户，需要增加处理多个账户的功能，使用数组的方式。需要计算多个账户的总额，可以引入静态数据成员。进行精确简便的日期处理。可以定义一个日期类。

3、然后要扩展增加“信用”账户。可以使用继承的方法，来扩展信用账户派生类。信用类和储蓄类存在某些共性，应该增加一个累加类计算某些具有共性的数据。

4、为了方便管理多个账户。可以用一个数组来管理多个储蓄、信用账户，并使用基类指针调用派生类函数。进行日期差的计算。可使用运算符重载对“-”进行重载。

5、接着使程序能够随时添加账户。使用动态数组类模板，以实现添加不同账户类的功能。

6、最后用标准模板STL改进程序。使用文件保存指令。每次运行程序的时候从文件读出指令并自动运行。加入异常处理机制。

**实验内容与步骤：**

**【**步骤1建立简单的储蓄账户**】**

(1)内容：Step1中主要实现简易活期储蓄账户管理系统，其变量存储过程涉及到从主函数中依次读取存储账户的操作日期（date以整数表示），Id，以及存款年利率（rate），而账户余额（balance）和余额按日累加之和（accumulation）默认存储为0，此外，上次变更余额的日期就是此次操作日期，意思为本次操作即为第一次。而该系统功能主要有：给用户显示所有储蓄账户的余额（show），存入现金（deposit），取出现金（先判断余额是否大于输入现金数withdraw），结算利息（每年1月1日结算settle），此处涉及到类中公有函数对类中私有函数的调用，此例有三个函数都调用了record（记录账单），该函数中有一个声明在头文件<cmath>中的floor函数，是向下取整函数，即返回一个double型的不大于一个数的最大整数，以double型浮点数形式存储结果，在本系统中主要功能是使金额（amount）转变为两位小数，方便与余额相加再存入余额。

（2）主要实现

\*创建一个活期储蓄账户 SavingsAccount类 ：

class SavingsAccount {}

主要成员： 账号（id），余额（balance），年利率（rate）等

private:

int id; //账号

double balance; //余额

double rate; //存款的年利率

int lastDate; //上次变更余额的时期

double accumulation; //余额按日累加之和

主要函数操作：显示账户信息（show），存款（deposit），取款（withdraw），结算利息（settle）

void record(int date, double amount);//记录一笔帐，date为日期， amount为金额

void deposit(int date, double amount);//存入现金

void withdraw(int date, double amount);//取出现金

void settle(int date);//结算利息，每年1月1日调用一次该函数

void show() const; //显示账户信息

计算利息：

double accumulate(int date) const {

return accumulation + balance \* (date - lastDate); }

//获得到指定日期为止的存款金额按日累积值

主要函数的代码：

record(int date, double amount) {

accumulation = accumulate(date);lastDate = date;

amount = floor(amount \* 100 + 0.5) / 100; //保留小数点后两位

balance += amount;

cout << date << "\t#" << id << "\t" << amount << "\t" << balance << endl;}

void SavingsAccount::deposit(int date, double amount) {

record(date, amount);}//存款

void SavingsAccount::withdraw(int date, double amount) {

if (amount > getBalance())

cout << "Error: not enough money" << endl;

else

record(date, -amount);}//取款

注：其中取款设判断语句，判断余额和所取现金的大小，从何判断能否正常取款

void SavingsAccount::settle(int date) {

double interest = accumulate(date) \* rate / 365; //计算年息

if (interest != 0)

record(date, interest);

accumulation = 0;}//结算利息

void SavingsAccount::show() const {

cout << "#" << id << "\tBalance: " << balance;//显示账户信息

**【**步骤2改进储蓄账户，进行日期的处理**】**

(1)内容：Step2中主要改善了总账户金额的存储和维护，友好处理日期（方便用户使用），以及用户账号的更全面表示。

1) 首先是总账户金额的存储和维护：在SavingsAccount中增加一个静态数据成员total，使得全体账户对象共享一份，无需copy，既能节省存储空间，又无需担心数据一致性的问题，同时自然要增加公有静态成员函数getTotal来实现对私有数据total的访问，而至于不需要改变对象状态的成员函数可声明为常成员函数，用const 修饰。

2) 其次是对日期的友好处理：在Step1中以整数代表日期，很不直观，大大增加了用户难度，因此在Step2中加入Date类，并将公元元年1月1日作为公共基准日期，将y年m月d日相距这一天的天数记为f(y/m/d,1/1/1),这样可以将其分解为三部分：

f(y/m/d,1/1/1)=f(y/1/1,1/1/1)+ f(y/m/1,y/1/1)+f(y/m/d,y/m/1)

其中，

f(y/1/1,1/1/1)=365\*(y-1)+(y-1)/4-(y-1)/100+(y-1)/400

f(y/m/d,y/m/1) = d – 1

但由于每月天数不同，因此f(y/m/1,y/1/1)难以表示为一个统一的公式，所以分别讨论平闰年，并且方便起见，引入一个数组，用于存储每月一日到一月一日的相差天数，并且为便于getMaxDay函数的实现，该数组多出一项，另外，加一个exit函数，相当于循环中的break，可以有效避免多执行造成的出错。

1. 最后是用户账号全面表示：因为Step1中用int整数来表示账号Id，并不全面完美，若银行账号以“0”开头，或账号超过整数的表示范围，或账号中包含其他字符，都无法实现最初设想功能，甚至报错，所以在Step2中，我们引入字符串String来表示Id，同时可以顺便在record 函数中加上说明文字，增加用户可读性，而为了充分利用String功能，并且使功能更全面，再增加一个报错函数，当其他函数需要输出错误信息时，直接把信息以字符串形式传递给该函数（error）即可，error函数会直接输出该字符串。
2. 主要实现：

\*总账户金额的存储和维护

static double total; //所有账户的总金额

void record(const Date &date, double amount, const std::string &desc);

//记录一笔帐，date为日期，amount为金额，desc为说明

void error(const std::string &msg) const; //报告错误信息

double accumulate(const Date& date) const {

return accumulation + balance \* date.distance(lastDate); }

public:

SavingsAccount(const Date &date, const std::string &id, double rate);

const std::string &getId() const { return id; }

double getBalance() const { return balance; }

double getRate() const { return rate; }

static double getTotal() { return total; }

void deposit(const Date &date, double amount, const std::string &desc); //存入现金

void withdraw(const Date &date, double amount, const std::string &desc);//取出现金

void settle(const Date &date); //结算利息，每年1月1日调用一次该函数

void show() const;//显示账户信息

};

\*日期友好化处理

namespace { //存储平年中某个月1日之前有多少天，为便于getMaxDay函数的实现，该数组多出一项

const int DAYS\_BEFORE\_MONTH[] = { 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365 };}

Date::Date(int year, int month, int day) : year(year), month(month), day(day) {

if (day <= 0 || day > getMaxDay()) {

cout << "Invalid date: "; show(); cout << endl;

exit(1); //用来中止当前程序的执行，并将一个整数返回给系统，通常，0表示正常退出，1表示异常退出}

int years = year - 1;

totalDays = years \* 365 + years / 4 - years / 100 + years / 400

+ DAYS\_BEFORE\_MONTH[month - 1] + day;

if (isLeapYear() && month > 2) totalDays++;}

int Date::getMaxDay() const {

if (isLeapYear() && month == 2)

return 29;

else

return DAYS\_BEFORE\_MONTH[month] - DAYS\_BEFORE\_MONTH[month - 1];}

\*用户账号表示的字符范围扩展

void record(const Date &date, double amount, const std::string &desc);

//报告错误信息

void error(const std::string &msg) const;

//获得到指定日期为止的存款金额按日累积值

【步骤3扩展出申请信用账户的功能】

（1）内容：在Step2中只有活期储蓄的简易功能，但实际上，银行账户还存在信用账户，所以我们需要在此步骤中扩展出申请信用账户的功能。

首先我们得联想银行信用账户都拥有哪些功能，于是，我们知道，我们做的信用账户得允许透支及有信用额度的考虑（总的透支金额应该这个额度之内），利息（信用卡中存钱，不会有利息，但使用信用账户透支则需要支付利息，信用账户的利率一般以日为单位，为了简单期间，我们不去考虑这个免息期），每月结算（与储蓄账户每年结算一次利息不同的是，信用账户每月进行结算，我们假定结算日是每月的1日），年费（信用账户每年需要交一次年费，假定，每年1月1日结算的时候扣年费）。于是我们从基类Account中派生出SavingAccount类和CreditAccount类，又因为计息功能不同，所以得分开算，因此，我们引入Accumulator类，提供计算一项数值的按日累加之和所需的接口，在两个派生类中分别将该类实例化，通过该类的实例来计算利息，这样，可以降低账户类的复杂性，提高计算数值按日累加之和的代码的可复用性。

1. 主要实现：

\*建立Account基类

class Account { //账户类

private:

std::string id; //帐号

double balance; //余额

static double total; //所有账户的总金额

protected:

Account(const Date &date, const std::string &id);

//记录一笔帐，date为日期，amount为金额，desc为说明

void record(const Date &date, double amount, const std::string &desc);

//报告错误信息

void error(const std::string &msg) const;

public:

const std::string &getId() const { return id; }

double getBalance() const { return balance; }

static double getTotal() { return total; }

void show() const;}; //显示账户信息

\*建立CreditAccount类来扩展信用账户功能

class CreditAccount : public Account { //信用账户类

private:

Accumulator acc; //辅助计算利息的累加器

double credit; //信用额度

double rate; //欠款的日利率

double fee; //信用卡年费

double getDebt() const { //获得欠款额

double balance = getBalance();

return (balance < 0 ? balance : 0); }

public:

CreditAccount(const Date &date, const std::string &id, double credit, double rate, double fee);//构造函数

double getCredit() const { return credit; }

double getRate() const { return rate; }

double getFee() const { return fee; }

double getAvailableCredit() const { //获得可用信用

if (getBalance() < 0) return credit + getBalance();

else return credit; }

\*建立Accumulator类

class Accumulator { //将某个数值按日累加

private:

Date lastDate; //上次变更数值的时期

double value; //数值的当前值

double sum; //数值按日累加之和

public:

//构造函数，date为开始累加的日期，value为初始值

Accumulator(const Date &date, double value)

: lastDate(date), value(value), sum(0) { }

//获得到日期date的累加结果

double getSum(const Date &date) const {

return sum + value \* date.distance(lastDate); }

//在date将数值变更为value

void change(const Date &date, double value) {

sum = getSum(date); lastDate = date;

this->value = value; }

//初始化，将日期变为date，数值变为value，累加器清零

void reset(const Date &date, double value) {

lastDate = date; this->value = value;

sum = 0; }};

【步骤4统一类型的数组来存储账户】

（1）内容：Step3中不同派生类的实例不能用统一类型的数组来存储，只能使用独立的对象来声明，对象只能分别操作，例如输出各个账户信息,只能分别调用对象show函数，无法放在循环结构中。而且派生类SavingAccount,CreditAccount虽然具有相同的成员函数deposit, withdraw和settle ，但其实现不同，所以放在派生类中定义，因此不同类的同名实现是彼此独立的。为避免代码冗余，则要使不同类账户放在统一类型数组中，此时就要用到虚函数。

1） 首先将show函数声明为虚函数，通过指向CreditAccount/SavingAccount类实例的Account类型的指针，来调用show函数时，被实际调用的将是为CreditAccount/SavingAccount类定义的show函数；

2） 其次，创建一个Account指针类型的数组，使各个元素分别指向各个账户对象，就可以通过一个循环来调用各个对象的show函数。

3） 最后，在Account类中添加deposit、withdraw、settle这3个函数的声明，且将他们都声明为纯虚函数，基类的指针可以调用派生类的相应函数，无需在基类中实现。

4） 此外，还改善了用户操作区，可以动态存入相关操作，如存储金额，取出金额，显示用户所有信息，改变日期，退出操作，显得更为友好，用while循环和switch来实现读取用户操作并作出相应动作。又使用基类Account指针来调用settle函数时，事先并不知道该指针所指向对象的具体类型，所以无法决定采用何种方式调用settle函数。于是，将二者统一为对每月1日的调用，并且在活期储蓄账户SavingAccount类的settle函数修改后实现，在结算之前判断是否为1月，是才结算。

（2）主要实现：

\*基类中将成员函数deposit, withdraw和settle设为虚函数

//存入现金，date为日期，amount为金额，desc为款项说明

virtual void deposit(const Date &date, double amount, const std::string &desc) = 0;//取出现金，date为日期，amount为金额，desc为款项说明

virtual void withdraw(const Date &date, double amount, const std::string &desc) = 0; //结算（计算利息、年费等），每月结算一次，date为结算日期

virtual void settle(const Date &date) = 0;

\*改善用户操作区

char cmd;

do { date.show();//显示日期和总金额

cout << "\tTotal: " << Account::getTotal() << "\tcommand> "; int index, day;double amount;string desc;

cin >> cmd;

switch (cmd) {

case 'd': //存入现金

cin >> index >> amount; getline(cin, desc);

accounts[index]->deposit(date, amount, desc);break;

case 'w': //取出现金

cin >> index >> amount; getline(cin, desc);

accounts[index]->withdraw(date, amount, desc);break;

case 's': //查询各账户信息

for (int i = 0; i < n; i++) {

cout << "[" << i << "] ";

accounts[i]->show();cout << endl;}

break;

case 'c': //改变日期

cin >> day;

if (day < date.getDay())

cout << "You cannot specify a previous day";

else if (day > date.getMaxDay())

cout << "Invalid day";

else

date = Date(date.getYear(), date.getMonth(), day);

break;

case 'n': //进入下个月

if (date.getMonth() == 12)

date = Date(date.getYear() + 1, 1, 1);

else

date = Date(date.getYear(), date.getMonth() + 1, 1);

for (int i = 0; i < n; i++)

accounts[i]->settle(date);break;}} while (cmd != 'e');

return 0;

\*基类Account指针来调用settle函数

void SavingsAccount::settle(const Date &date) {

if (date.getMonth() == 1) { //每年的一月计算一次利息

double interest = acc.getSum(date) \* rate

/ (date - Date(date.getYear() - 1, 1, 1));

if (interest != 0)

record(date, interest, "interest");

acc.reset(date, getBalance());}}

【步骤5增加并且改进动态只能增添账户的功能】

1. 内容：
2. 适当解决了用户操作友好性问题，我们考虑如何使程序简易易读，即可读性强，于是我们将原来计算两日期相差天数的distance函数改为“-”运算符，此处用到了重载运算符。运算符函数是一种特殊的成员函数或友元函数，成员函数的语句格式为：

**类型 类名：：operator op(参数表){//相对于该类定义的操作}**

1. 其中，“类型”是函数的返回类型，“类名”是重载该运算符的类，“op”表示要重载的运算符，函数名是“operator op”，由关键字operator和被重载的运算符op组成，“参数表”列出该运算符所需要的操作数。
2. 此外，我们还发现Step4中虽然可以在已有账户上做各种操作，但无法添加新的账户，为了实现随时动态添加新的不同类型的账户，使用动态数组类模板Array来代替C++预定义的数组类型，这样既可以完成C++基本数组同样的功能，又因为Array数组允许动态改变大小，因此可以向Array数组中动态添加新的元素，以实现添加不同类型新账户的目标。
3. 模板的功能由此显而易见，强类型的程序设计迫使程序员为逻辑结构相同而具体数据类型不同的对象编写模式一致的代码，而无法抽取其中的共性，这显然不利于程序的扩充和维护，模板则提供了对逻辑结构相同的数据对象通用行为的定义，需通过编译才真正可以运行，称为实例化。
4. 主要实现：

\*运算符重载

int operator op - const（Date &date）const{

return totalDays -date. totalDays;}

\*建立数组类模版

template <class T>

class Array {

private:

T\* list; //T类型指针，用于存放动态分配的数组内存首地址

int size; //数组大小（元素个数）

public:

Array(int sz = 50); //构造函数

Array(const Array<T> &a); //拷贝构造函数

~Array(); //析构函数

Array<T>&operator=(constArray<T>&rhs);

//重载"="使数组对象可以整体赋值

T & operator [] (int i); //重载"[]"，使Array对象可以起到C++普通数组的作用

const T & operator [] (int i) const; //"[]"运算符的const版本

operator T \* (); //重载到T\*类型的转换，使Array对象可以起到C++普通数组的作用

operator const T \* () const; //到T\*类型转换操作符的const版本

int getSize() const; //取数组的大小

void resize(int sz); //修改数组的大小}

【步骤6向量容器vector能够方便地完成自定义的Array类模板管理账户列表的功能】

（1）内容：Step5虽然用了Array模板，并且很好的解决了我们的问题，但敲代码过程甚为繁琐，花费众多时间，于是我们联想到标准模板库STL中提供的向量容器vector能够更加方便地完成自己定义的Array类模板管理账户列表的功能，利用该功能进行泛型编程，可以节省大量的时间和精力，并得到更高质量的代码。

vector向量容器可以随机访问序列中的单个元素，在序列尾快速插入和删除元素，如c.push\_back(elem)就能在容器末端插入数据elem，c.pop\_back则能删除容器末端数据。如果在序列中插入和删除元素，则时间与序列长度成正比。

除此之外，若用户想查询历史账目，则Step5根本无法实现，因此在步骤6中，我们希望能够查询一个指定日期范围内的历史账目。于是，我们考虑用多重映射multimap来高效地查询历史账目，多重映射multimap存储每一笔账目，该映射的键（key）是账目的日期，附加数据（value）是账目的详细内容。调用该多重映射low\_bound和upper\_bound函数，可以得到指定日期范围的所有账目，再利用一个循环将它们输出，就实现了查询历史账目的功能。

做了这个映射以后，我们需要对Date做配合修改来适应这个映射，于是将Date类型的数据做为键，此时需要为Date重载“<”运算符，以支持键值的比较。并且完善起见，在用户查询历史账目时，为了能够接受用户输入的日期，需要为Date类增加一个静态成员函数read专门从cin读入一个日期。需要强调的是，<utility>头文件通过一组函数模板对任意数据类型重载了“<=”, “<”, “>=”, “!=”运算符,且前三个运算符均被转化为对“<”的调用，最后一个被转化为对“==”的调用，因此即使只重载了“<”，但可以实现“<=”的调用。又因为这些函数模板定义在std::rel\_ops命名空间内，因此在Account.cpp中使用它们之前，需要#include <utility>并用using namespace 开启std::rel\_ops命名空间。

(2)主要实现：

\*vector向量容器的使用

struct deleter {

template <class T> void operator () (T\* p) { delete p; }};

vector<Account \*> accounts; //创建账户数组，元素个数为0

\*运算符重载

static Date read();

int getYear() const { return year; }

int getMonth() const { return month; }

int getDay() const { return day; }

int getMaxDay() const; //获得当月有多少天

bool isLeapYear() const { //判断当年是否为闰年

return year % 4 == 0 && year % 100 != 0 || year % 400 == 0; }

void show() const; //输出当前日期

int operator - (const Date& date) const {

return totalDays - date.totalDays;}//计算两个日期之间差多少天

bool operator < (const Date& date) const {

return totalDays < date.totalDays; }};//判断两个日期的前后顺序

【步骤7增加控制器来储存账户列表和处理命令】

1. 内容：
2. 虽然以上步骤六能够实现增加用户的功能，却无法保存账户信息，下次启动程序后还需要重新输入数据，给用户增加了许多不必要的麻烦，因此，在步骤7，我们用一个Controller类作为控制器，用来储存账户列表和处理命令。

2） 具体实现则需要调用fstream头文件，它包含处理文件的有关信息，提供建立文件、读写文件的各种操作接口，以及sstream头文件，为了stringstream、isstringstream、ostringstream类库中定义的istringstream类和ostringstream类支持从string对象输入/输出数据，并且更加安全。如此操作，当账户数据发生改变时，自动存入.txt文件。

（2）主要实现：

\*添加Controller类

class Controller { //控制器，用来储存账户列表和处理命令

private: Date date; //当前日期

vector<Account \*> accounts; //账户列表

bool end; //用户是否输入了退出命令

public:

Controller(const Date &date) : date(date), end(false) { }

~Controller();

const Date &getDate() const { return date; }

bool isEnd() const { return end; }

//执行一条命名，返回该命令是否改变了当前状态（即是否需要保存该命令）

bool runCommand(const string &cmdLine);}

\*调用Controller类控制器

ifstream fileIn(FILE\_NAME); //以读模式打开文件

if (fileIn) { //如果正常打开，就执行文件中的每一条命令

while (getline(fileIn, cmdLine))

controller.runCommand(cmdLine);

fileIn.close(); //关闭文件}

ofstream fileOut(FILE\_NAME, ios\_base::app); //以追加模式

cout << "(a)add account (d)deposit (w)withdraw (s)show (c)change day (n)next month (q)query (e)exit" << endl;

while (!controller.isEnd()) { //从标准输入读入命令并执行，直到退出

cout << controller.getDate() << "\tTotal: " << Account::getTotal() << "\tcommand> ";

string cmdLine;getline(cin, cmdLine);

if (controller.runCommand(cmdLine))

fileOut << cmdLine << endl; //将命令写入文件}

【步骤8 异常的检测与异常处理分离】

1. 内容：

1） 经过步骤7，可以说功能已基本完善，但这只是在正常情况下的运行成功状态，若程序运行期间出现非正常情况，结果就很难办，所以我们需要使用C++提供结构化的异常处理方法，将异常的检测与异常处理分离，因而增加了程序的可读性，这样，在保证逻辑上的正确性时，还拥有了容错能力。

2）非正常情况包括了用户误操作，外部设备或文件的不正确连接，或者内存空间不足等状况，异常通常是错误，不过也可能是某些很少出现的特殊事件。传统的解决方案为使用中断命令，但这样使得错误处理代码掺杂于系统功能实现的代码主线中，降低程序的可读性和可维护性，不适用于组件式的大型软件开发。

3）C++异常处理通过三个关键字实现：throw、try和catch，一般，被调用函数直接检测到异常条件的存在，并用throw语句抛出一个异常；在上层调用函数中，使用try语句检测函数调用是否引发异常，被检测到的各种异常由catch语句捕获并作相应的处理。

4）此外，我们还可以在date重载流插入和流提取运算符，用于传输用户自定义类型的数据，我们都知道，cout是C++输出流ostream的预定义对象，用于连接显示器，cin是输入流istream的预定义对象，用于连接键盘，而“流”是C++基本类库中定义的类，那么，在此例中，我们引用参数in成为cin的别名，out成为cout的别名，函数返回一个对istream的引用，以便插入运算符的连续调用，而ostream则同理。

（2）主要实现：

\*增加异常处理

try { controller.runCommand(cmdLine); }

catch (exception &e) {

cout << "Bad line in " << FILE\_NAME << ": " << cmdLine << endl;

cout << "Error: " << e.what() << endl;

return 1; } }

\*定义抛错类处理异常

class AccountException : public std::runtime\_error {

private: const Account \*account;

public:

AccountException(const Account \*account, const std::string &msg)

: runtime\_error(msg), account(account) { }

const Account \*getAccount() const { return account; } };

\*重载流插入和流提取运算符

istream & operator >> (istream &in, Date &date) {

int year, month, day;

char c1, c2;

in >> year >> c1 >> month >> c2 >> day;

if (c1 != '-' || c2 != '-')

throw runtime\_error("Bad time format");

date = Date(year, month, day);

return in;}

ostream & operator << (ostream &out, const Date &date) {

out << date.getYear() << "-" << date.getMonth() << "-" << date.getDay();

return out;}

**实验数据：**

【步骤1】

测试用例1：a s S3755217 0.015（用户输入正确数据）

测试用例2：100 s S3755217 0.015（用户输入了错误的选项）

测试用例3: a s S3755217 50000（用户输入了错误的利率）

【步骤2】

测试用例1：2008-11-5 #S3755217 5000 5000 salary（用户输入正确）

测试用例2：2008-12-20 #02342342 -4000 6000 buy a laptop

（用户输入正确）

测试用例3：2009-1-1 #S3755217 17.77 10517.8 interest（用户输入正确）

【步骤3】

测试用例1：2009-1-1 #C5392394 -50 -50 annual fee

测试用例2：2008-12-1 #C5392394 2016 0 repay the credit

测试用例3：2012-5-3 #C5392394 2046 0 repay the credit

【步骤4】

测试用例1：d 100 salary (用户正确存入存款)

测试用例2：w 10 buying items(用户正确取出存款)

测试用例3: c 32 (错误的超出了日期的范围)

【步骤5】

测试用例1：d 100 salary (用户正确存入存款)

测试用例2：w 10 buying items(用户正确取出存款)

测试用例3：a s #0100 0.5(正确创建账户)

【步骤6】

测试用例1：a s S3755217 0.015

a s 02342342 0.015

a c C5392394 10000 0.0005 50

c 5

d 0 5000 salary

c 15

w 2 2000 buy a cell

c 25

d 1 10000 sell stock 0323

n

d 2 2016 repay the credit

c 5

d 0 5500 salary

n

c 15

w 2 1500 buy a television（指令保存在文件中）

测试用例2：a s S3734217 0.235

a s 02342342 0.015

c 15

d 2 2016 repay the credit

c 5

d 0 5500 salary

n

c 15

w 2 1500 buy a television

测试用例3：a s S3734217 0.235

c a 02356418 0.224

a s 02342342 0.015

c 15

d 2 2016 repay the credit

c 5

d 0 5500 salary

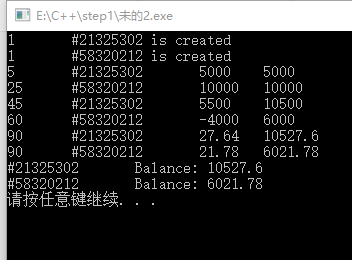
n

c 15

**实验数据处理：**每次都用一个记事本记下数据，然后与程序中的设定进行对比，验证程序是否正确运行，数据是否已经保存等。

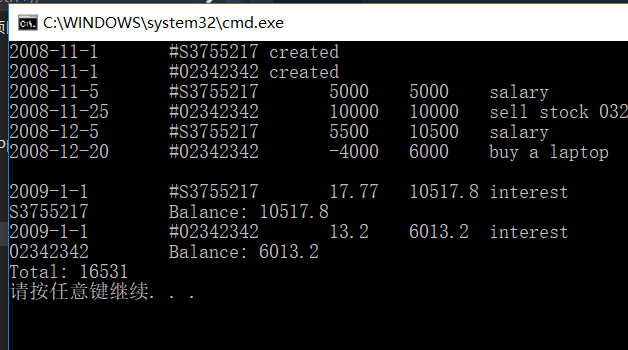
**实验结果与分析：**

【步骤1】



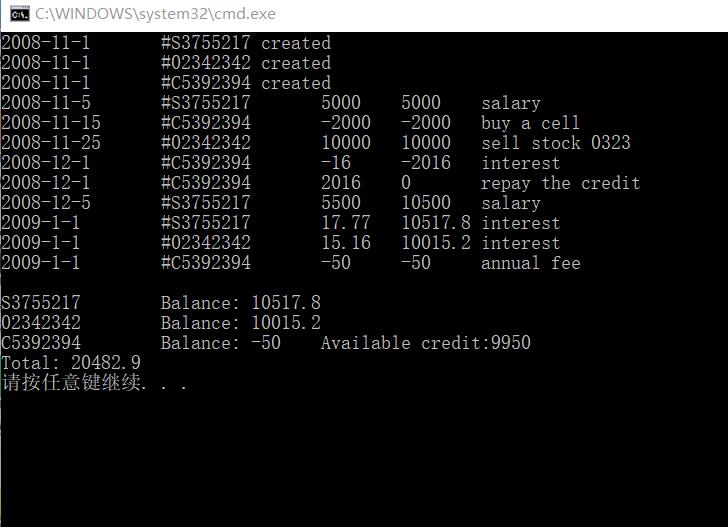
前两行内容说明账户对象创建成功，显示被创建账户ID以及创建成功信息；3到5行代表使用存款的函数deposit成功显示金额变化量以及余额量。6行表示取款函数withdraw使用成功。7、8行表示开户后的第90天到了计息日，结算所有账户年息，显示内容代表settle函数调用，成功显示金额变化量以及余额量。后两行是show函数的成功使用，反应用户id和对应的余额。

【步骤2】



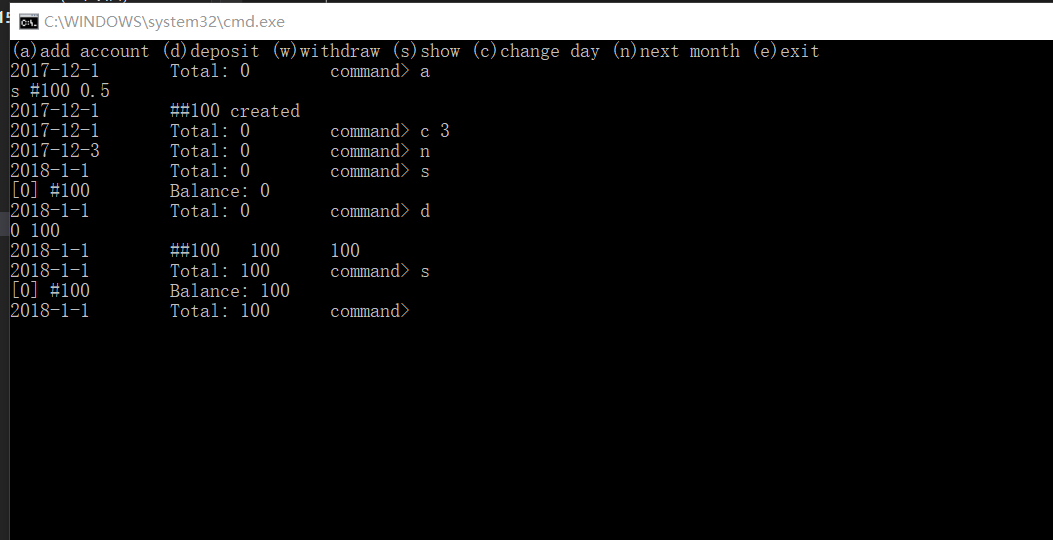
添加了静态数据成员total，使得对账户总金额有了掌握。日期从一个数字变成了对使用者较为友好的形式，并添加了详细的账户使用备注。其中第7.9行是利息计算。

【步骤3，4】



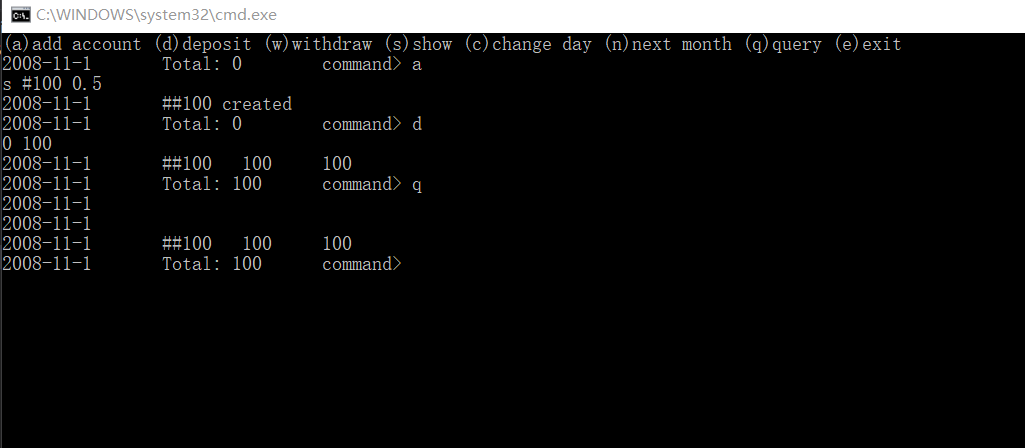
增加了信用账户的功能，使得账户的功能更加完善。并利用新的累加类来匹配派生类的计算。第3行表示创建成功一个信用账户。第8行表示对信用账户还款。倒数第二行表示信用账户的额度。

【步骤5】



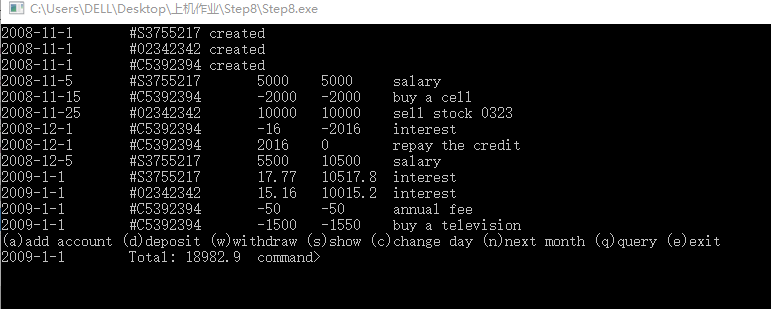
增加了动态添加账户的功能，将账户存于一个数组中。第3行表示创建了一个储蓄账户。第5行表示日期更改，从1号变成了3号。第6行表示进行到下一个月。命令s展示了账户的余额，是show函数的成功运行。

【步骤6，7】



增加了query查询功能。可以查询两个指定日期之间的账目。其中第8行到11行是query函数的成功应用，查询到了指定日期之间的账目信息。

【步骤8】



图十九

增加了用文件保存指令的功能，当使用exit退出程序时，能够将已经创建的账户信息保存在文件里。并且还含有异常处理的功能。

**实验结论与感想**

1、结论：通过个人账户的创建，了解到任何一个程序都是由起初的原始代码进行一步步优化得到最终的程序。

2、感想：

1）有关信息始终存放在内存中，程序结束后账户的信息也将随之消失，下次启动程序后还需要重新输入数据。如何将这些信息保存下来，使得下次启动程序时能够恢复上次的数据呢，还有待解决。

2）是否能够添加一个转账功能，联系各个账户之间的关系，能够互相

转账增加实用性。

3）能否增加一个密码设置，使每次登陆查询都需验证，提高账户安全性。