

实验名称：个人银行账户管理系统的改进

实验目的：

完善银行账户系统，并实现以下功能：

- 1.查询不合法日期异常处理机制
- 2.账户超额取款异常处理机制
- 3.未按指定输入字符进行操作异常处理

o o o

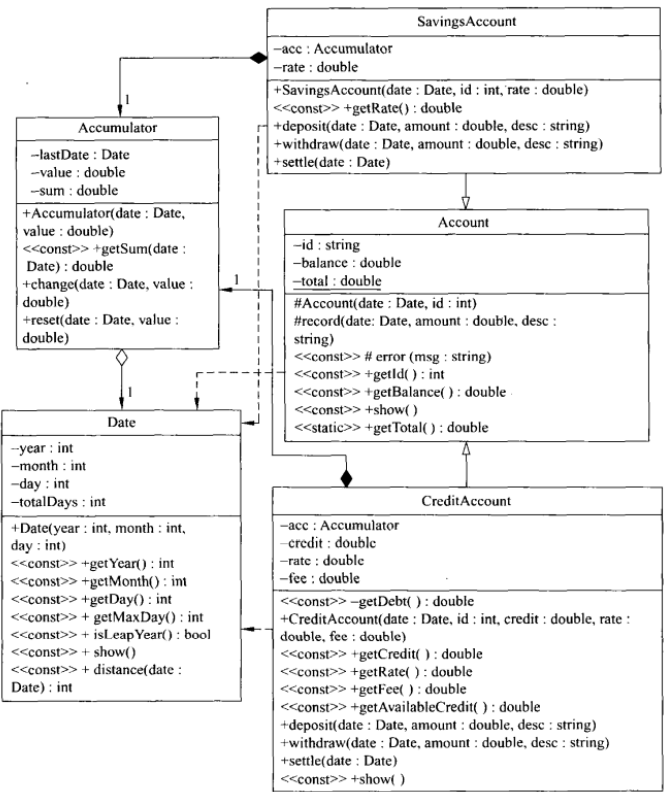
实验仪器：

操作系统：Windows 10

编译器：Clang

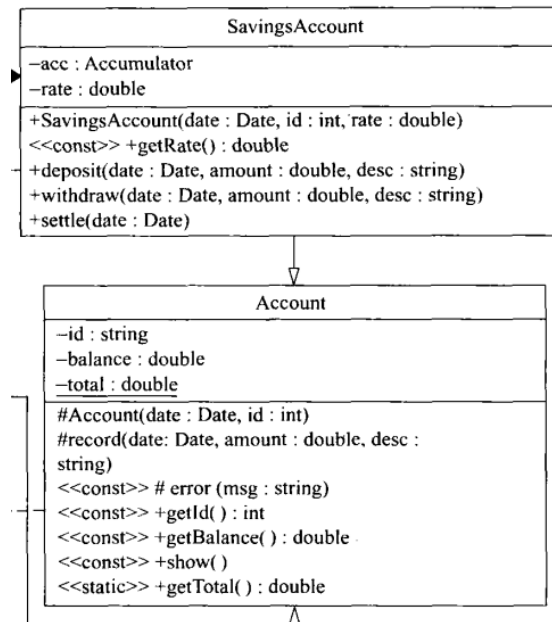
实验内容与步骤：

为了避免实验过程中编写代码造成功能冗杂，首先查看银行 1-6 所有需要的功能，规划一个总的流程图再分步骤完成每个功能所需。



【功能 1：建立简单的储蓄账户】

1.设计



为了实现储蓄账户类的设计，单独建立了一个类 `Account`，元素包含用户名 `id`、账户余额 `balance`，函数包含让该账户存钱 `deposit` 取钱 `withdraw` 利息获得 `ssttle` 和获取元素的 `get` 函数，分别在操作用户金额时调用，并建立类 `SavingsAccount` 公有继承于 `Account`，此为单独的储蓄账户类。

2.实现

```
class Account
{
private:
    string id; // 用户id
    double balance; // 余额
    static double total; // 总账户余额
    static RecordMap recordMap; // 多重映射

public:
    Account(const Date& date, const string& id); // 账户构造函数
    void record(const Date& date, double amount, const string &desc); // 账户账目记
    录
```

```

void error(const string& msg) const; // 报错函数

const string& getId() const { return id; } // 返回id
double getBalance() const { return balance; } // 返回余额
static double getTotal() { return total; } // 返回总账户余额

virtual void deposit(const Date& date, double amount, const string& desc) = 0;
// 存款虚函数
virtual void withdraw(const Date& date, double amount, const string& desc) = 0;
// 取款虚函数

virtual void show(); // 显示信息
virtual void show(ostream& out) const; // 显示信息

virtual void settle(const Date& date) = 0; // 计算利息

static void query(const Date& begin, const Date& end); // 查询信息
};

```

代码片段1 基类的关键代码

```

class SavingsAccount: public Account
{
private:
    Accumulator acc; // 计算器
    double rate; // 年利率

public:
    SavingsAccount(const Date& date, const string& id, double rate); // 储蓄账户构造
    double getRate() const { return rate; } // 获得利率
    void deposit(const Date& date, double amount, const string& desc); // 储蓄账户存款定义
    void withdraw(const Date& date, double amount, const string& desc); // 储蓄账户取款定义
    void settle(const Date& date);
};

Account::Account(const Date& date, const string& id) // 账户构建
: id(id), balance(0)
{
    cout << date << "\t#" << id << " created" << endl;
}

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

```

```

        :Account(date, id), rate(rate), acc(date, 0) {} // 储蓄账户构建

void SavingsAccount::deposit(const Date& date, double amount, const string& desc) //
储蓄账户存款
{
    record(date, amount, desc); // 账目记录
    acc.change(date, getBalance()); // 计算器计算
}

void SavingsAccount::withdraw(const Date &date, double amount, const string&desc) // 储
蓄账户取款
{
    if (amount > getBalance())
        error("not enough money"); // 超出余额报错
    else
        record(date, -amount, desc); acc.change(date, getBalance()); // 账目记录
}

```

代码片段2 储蓄账户类的关键代码

3.测试

(1) 测试用例 1: a s S1 0.05 (用户输入正确数据)

得到的运行截图如下:

```

D:\MHL\大二上\个人文件\程设实践\银行系统\1\Book-Bank\Debug\Book-Bank.exe
(a)add account (d)deposit (w)withdraw (s)show (c)change day (n)next month (q)query (e)exit
2008-11-1 Total: 0 command>a s S1 0.05
2008-11-1 #S1 created
2008-11-1 Total: 0 command>

```

分析可得该代码在输入正确情况下能正常完成储蓄账户建立。

(2) 测试用例: a s S2 0.07 (用户输入正常数据)

得到的运行截图如下:

```

D:\MHL\大二上\个人文件\程设实践\银行系统\1\Book-Bank\Debug\Book-Bank.exe
(a)add account (d)deposit (w)withdraw (s)show (c)change day (n)next month (q)query (e)exit
2008-11-1 Total: 0 command>a s S2 0.07
2008-11-1 #S2 created
2008-11-1 Total: 0 command>

```

分析可得该代码在输入正确情况下能正常完成储蓄账户建立。

(3) 测试用例: g (用户输入非正常字符)

得到的运行截图如下:

```

2008-11-1      #32 created
2008-11-1      Total: 0      command>g
Invalid command:g
2008-11-1      Total: 0      command>

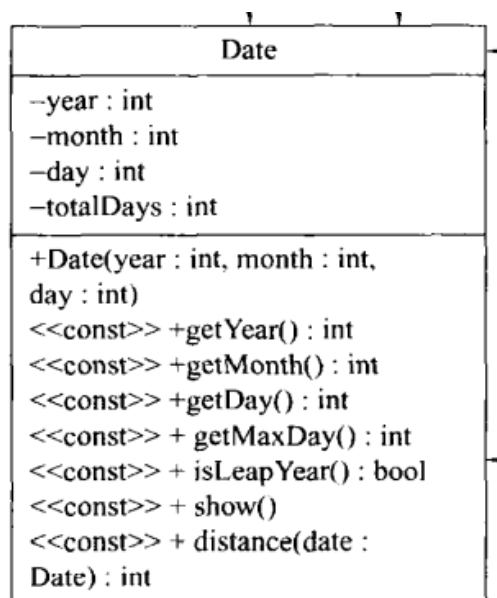
```

分析可得该代码在输入非正常操作字符时不能正常建立储蓄账户。

【功能 2：建立简单的储蓄账户】

1.设计

增加一个静态数据成员 `total` 用以记录所有账户总金额，并且及案例 `Date` 类来对日期进行友好处理，将账户 `id` 的类型修改为 `string`。



日期友好化处理后不能直接计算两日期相对差，故增加一个新参数 `totalDays` 用以计算与 1 年 1 月 1 日的相对天数，两日期减去其相对天数即为天数差。

计算相对天数时，需要考虑与 1 年 1 月 1 日相差多少个闰年 2 月，每相差 1 个时相对天数需要增加 1 天。

2.实现

```

class Account
{
private:
    string id; // 用户id
    double balance; // 余额
    static double total; // 总账户余额

    static RecordMap recordMap; // 多重映射
}

```

代码片段3 静态数据成员total的关键代码

```

class Date // 日期类
{
private:
    int year; // 年
    int month; // 月
    int day; // 日
    int totalDays; // 相对天数

public:
    Date()
        :year(0), month(0), day(0), totalDays(0) {} // 日期构造函数
    Date(int year, int month, int day); // 日期构造函数
    //static Date read();
    int getYear()const { return year; } // 返回年
    int getMonth()const { return month; } // 返回月
    int getDay()const { return day; } // 返回日
    int getMaxDay()const; // 获得当月最大天数
    bool isLeapYear()const // 判断是否闰年
    {
        return year % 4 == 0 && year % 100 != 0 || year % 400 == 0;
    }
    void show()const; // 显示信息

    int operator-(const Date& date)const
    {
        return totalDays - date.totalDays; // 重载-
    }
}

namespace // 每月距离一月天数

```

```

{
    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) // 日期构建
{
    if (day <= 0 || day > getMaxDay()) // 小于0或大于本月最大天数
        throw runtime_error("Invalid date");
    int years = year - 1;
    totalDays = years * 365 + years / 4 - years / 100 + years / 400 +
DAYS_BEFORE_MONTH[month - 1] + day; // 与1.1.1日相对日期
    if (isLeapYear() && month > 2) { totalDays++; } // 遇到闰年相对天数++
}

int Date::getMaxDay() const
{
    if (isLeapYear() && month == 2)
        return 29; // 二月闰年情况
    else
        return DAYS_BEFORE_MONTH[month] - DAYS_BEFORE_MONTH[month - 1];
}

void Date::show() const // 显示信息
{
    cout << getYear() << "-" << getMonth() << "-" << getDay();
}

```

代码片段4 日期友好化处理的关键代码

3.测试

(1) 测试用例 1: a s S1 0.5 (用户正常输入)

得到运行截图如下:

```

D:\MHL\大二上\个人文件\程设实践\银行系统\1\Book-Bank\Debug\Book-Bank.exe
(a)add account (d)deposit (w)withdraw (s)show (c)change day (n)next month (q)query (e)exit
2008-11-1      Total: 0      command>a s S1 0.05
2008-11-1      #S1 created
2008-11-1      Total: 0      command>

```

分析可得正常完成了时间友好化处理。

(2) 测试用例 2: a s S2 0.7 (用户正常输入)

得到运行截图如下:

```
2008-11-1      #S1 created
2008-11-1      Total: 0      command>a s S2 0.07
2008-11-1      #S2 created
2008-11-1      Total: 0      command>
```

分析可得正常完成了时间友好化处理。

(3) 测试用例 3: c 5 (修改日期)

得到运行截图如下:

```
a) (a)add account (d)deposit (w)withdraw (s)show
2008-11-1      Total: 0      command>c 5
2008-11-5      Total: 0      command>
```

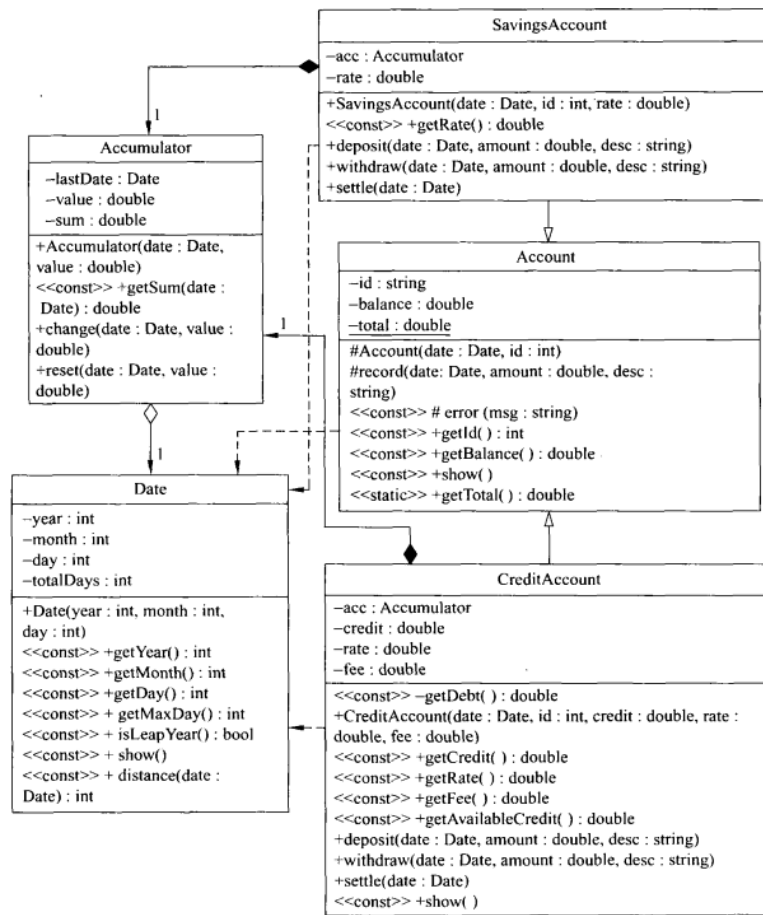
分析可得该时间类可以正常修改日期。

【功能 3: 增加信用卡账户】

1.设计

使用基类 `Account` 管理储蓄账户 `SavingsAccount` 和 `CreditAccount` 的共同元素,并由此派生出两类,两派生类再分别设置自己所需的元素,例如信用账户单独设置其需要的信用额度、年费等。

因为二者计息方式不同,储蓄账户是每年一次结息,信用账户是每月一次结息,所以添加类 `Accumulator` 用于描述两类账在计算利息时的共性和计息方法。



2. 实现

```

class CreditAccount : public Account
{
private:
    Accumulator acc; // 计算器
    double credit; // 信用值=可贷款额度
    double rate; // 年利率
    double fee; // 年费
    double getDebt() const // 获得债务
    {
        double balance = getBalance();
        return (balance < 0 ? balance : 0);
    }

public:
    CreditAccount(const Date& date, const string& id, double credit, double rate,
double fee); // 信用账户构造
    double getCredit() const { return credit; } // 获得额度
  
```

```

double getRate()const { return rate; } // 获得年利率
double getFee()const { return fee; } // 获得年费
double getAvailableCredit()const // 获得可用额度
{
    if (getBalance() < 0)
        return credit + getBalance();
    else
        return credit;
}

void deposit(const Date& date, double amount, const string &desc); // 信用账户存款
构造
void withdraw(const Date& date, double amount, const string &desc); // 信用账户取款
构造
void settle(const Date& date); // 信用账户利息计算

virtual void show(); // 显示信息
virtual void show(ostream &out)const; // 显示信息
};

CreditAccount::CreditAccount(const Date&date, const string&id, double credit, double
rate, double fee)
    :Account(date, id), credit(credit), rate(rate), fee(fee), acc(date, 0) {} // 信用账户构建

void CreditAccount::deposit(const Date& date, double amount, const string &desc) // 信
用账户存款
{
    record(date, amount, desc); // 存款记录
    acc.change(date, getDebt()); // 余额变动
}

void CreditAccount::withdraw(const Date& date, double amount, const string &desc) // 信
用账户取款
{
    if (amount - getBalance() > credit)
    {
        error("not enough credit"); // 取款大于余额与可用信用度之和
    }
    else
    {
        record(date, -amount, desc); // 记录取款
        acc.change(date, getDebt()); // 余额变动
    }
}

```

```

void CreditAccount::settle(const Date& date) // 信用账户计算年息
{
    double interest = acc.getSum(date) * rate;
    if (interest != 0)
        record(date, interest, "interest"); // 记录年息
    if (date.getMonth() == 1) // 月份=1
        record(date, -fee, "annual fee"); // 扣除年费
    acc.reset(date, getDebt()); // 变动余额总值重置
}

void CreditAccount::show() // 信息显示
{
    Account::show();
    cout << "\tAvailable credit:" << getAvailableCredit();
}

void CreditAccount::show(ostream& out) const // 信息显示
{
    Account::show(out);
    out << "\tAvailable credit:" << getAvailableCredit();
}

```

代码片段5 信用账户类的关键代码

```

class Accumulator // 计算器
{
private:
    Date lastDate; // 上次变动日期
    double value; // 中间值
    double sum; // 变动总值

public:
    Accumulator(const Date&date, double value)
        :lastDate(date), value(value), sum(0) {} // 计算器构造
    double getSum(const Date& date) const
    {
        return sum + value * (date-lastDate); // 获得变动总值
    }
    void change(const Date& date, double value)
    {
        sum = getSum(date); // 更改变动总值
        lastDate = date; this->value = value; // 修改上次修改日期和中间值
    }
    void reset(const Date& date, double value)

```

```

    {
        lastDate = date; // 修改变动日期
        this->value = value; // 修改中间值
        sum = 0; // 重置变动总值
    }
};

```

代码片段6 计数器的关键代码

3. 测试

(1) 测试用例 1: a c C1 20000 0.05 100 (用户正常输入)

得到的运行截图如下:

```

(a) add account (d) deposit (w) withdraw (s) show (e) change day (n) next
2008-11-5      Total: 0      command>a c C1 20000 0.05 100
2008-11-5      #C1 created
2008-11-5      Total: 0      command>_

```

分析可得信用账户建立有效, 可以正常建立信用账户。

(2) 测试用例 2: d 2 1000 (信用账户存款)

得到运行截图如下:

```

2008-11-5      #C1 created
2008-11-5      Total: 0      command>d 2 1000
2008-11-5      #C1      1000      1000
2008-11-5      Total: 1000

```

分析可得信用账户建立有效, 可以正常存款。

(3) 测试用例 3: n n (进入下年查询)

得到运行截图如下:

```

2008-11-5      Total: 1000      command>n
2008-12-1      Total: 1000      command>n
2009-1-1      #C1      -100      900      annual fee
2009-1-1      Total: 900      command>_

```

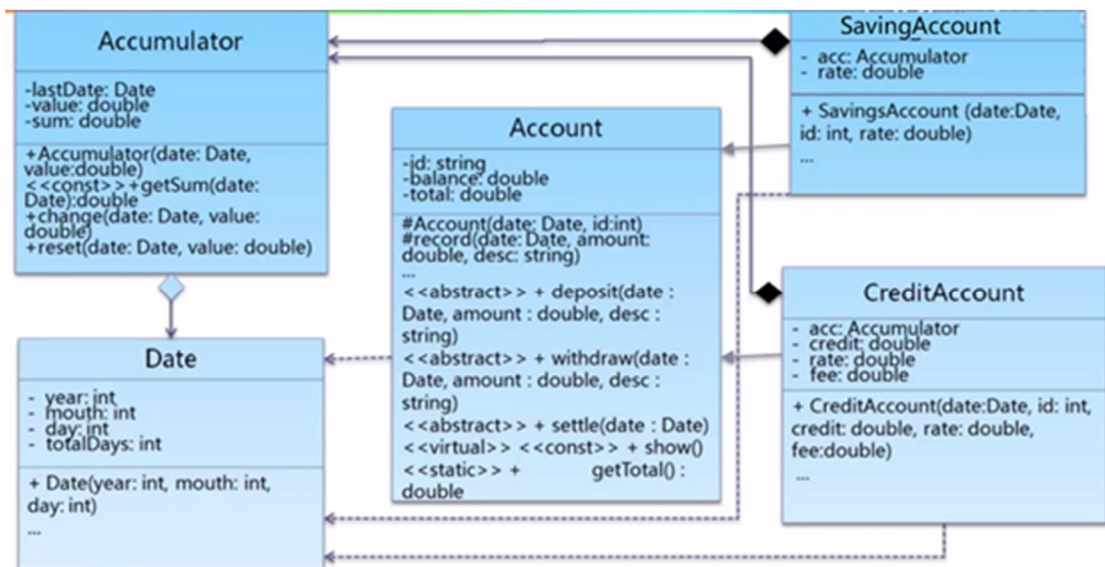
分析可得信用账户建立有效, 且能正常结算年费。

【功能 4：管理多个储蓄/信用账户】

1. 设计

改造 Account 类，将 show.deposit.withdraw.settle 四个函数声明为虚函数并在 SavingsAccount 类和 CreditAccount 类中重新定义，因为二者可以具有相同操作接口，但在不同类型的账户中其计算方式不相同，利用虚函数的多重性可以再简化代码的同时优化用户体验。

创建一个 Account 指针类型数组，其中各个元素指向各个对象，利用基类指针可以调用派生类函数的特性来统一管理账户。



2. 实现

```
class Controller // 主体控制器
{
private:
    Date date;
    vector<Account*>accounts;

    bool end;
}
```

代码片段7 Account数组的关键代码

```

class Account
{
    virtual void deposit(const Date& date, double amount, const string& desc) = 0; //
存款虚函数
    virtual void withdraw(const Date& date, double amount, const string& desc) = 0; //
取款虚函数

    virtual void show(); // 显示信息
    virtual void show(ostream& out) const; // 显示信息

    virtual void settle(const Date& date) = 0; // 计算利息
}

```

代码片段8 基类虚函数的关键代码

```

void SavingsAccount::deposit(const Date& date, double amount, const string& desc) // 储
蓄账户存款
{
    record(date, amount, desc); // 账目记录
    acc.change(date, getBalance()); // 计算器计算
}

void SavingsAccount::withdraw(const Date &date, double amount, const string& desc) // 储蓄
账户取款
{
    if (amount > getBalance())
        error("not enough money"); // 超出余额报错
    else
        record(date, -amount, desc); acc.change(date, getBalance()); // 账目记录
}

void SavingsAccount::settle(const Date& date) // 储蓄账户计算年息
{
    if (date.getMonth() == 1) // 月份=1开始计算年息
    {
        double interest = acc.getSum(date) * rate / (date - Date(date.getYear() - 1,
1, 1));
        if (interest != 0)
            record(date, interest, "interest"); // 记录年息
        acc.reset(date, getBalance()); // 重置本年变动总值
    }
}

```

```
}
```

代码片段9 储蓄账户类虚函数定义的关键代码

```
void CreditAccount::deposit(const Date& date, double amount, const string &desc) // 信用账户存款
{
    record(date, amount, desc); // 存款记录
    acc.change(date, getDebt()); // 余额变动
}

void CreditAccount::withdraw(const Date& date, double amount, const string &desc) // 信用账户取款
{
    if (amount - getBalance() > credit)
    {
        error("not enough credit"); // 取款大于余额与可用信用度之和
    }
    else
    {
        record(date, -amount, desc); // 记录取款
        acc.change(date, getDebt()); // 余额变动
    }
}

void CreditAccount::settle(const Date& date) // 信用账户计算年息
{
    double interest = acc.getSum(date) * rate;
    if (interest != 0)
        record(date, interest, "interest"); // 记录年息
    if (date.getMonth() == 1) // 月份=1
        record(date, -fee, "annual fee"); // 扣除年费
    acc.reset(date, getDebt()); // 变动余额总值重置
}

void CreditAccount::show() // 信息显示
{
    Account::show();
    cout << "\tAvailable credit:" << getAvailableCredit();
}

void CreditAccount::show(ostream& out) const // 信息显示
{
    Account::show(out);
```

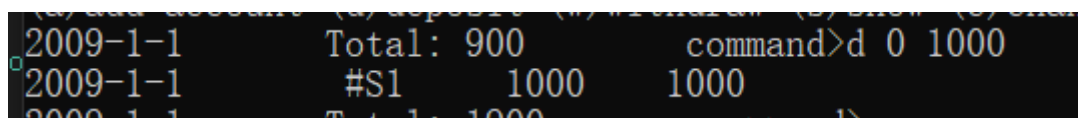
```
out << "\tAvailable credit:" << getAvailableCredit();  
}
```

代码片段10 信用账户类虚函数定义的关键代码

3. 测试

(1) 测试用例 1: d 0 1000 (储蓄账户存款)

得到运行截图如下:

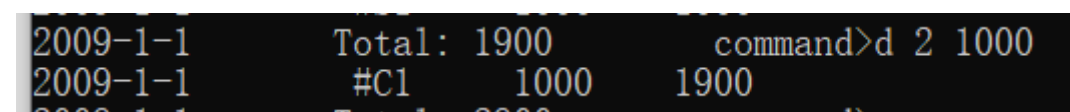


```
2009-1-1      Total: 900      command>d 0 1000  
2009-1-1      #S1      1000      1000  
2009-1-1      Total: 1000
```

分析可得基类指针可正常调用储蓄账户类存款函数进行存款。

(2) 测试用例 2: d 2 1000 (信用账户存款)

得到运行截图如下:

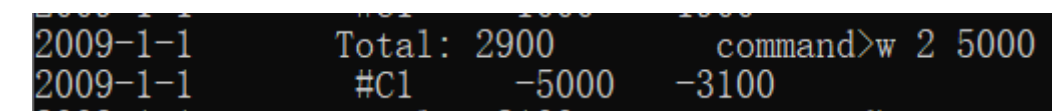


```
2009-1-1      Total: 1900      command>d 2 1000  
2009-1-1      #C1      1000      1900  
2009-1-1      Total: 1900
```

分析可得基类指针可正常调用信用账户类存款函数进行存款。

(3) 测试用例 3: d 2 5000 (信用账户取款)

得到运行截图如下:



```
2009-1-1      Total: 2900      command>w 2 5000  
2009-1-1      #C1      -5000      -3100  
2009-1-1      Total: -3100
```

分析可得基类指针可正常调用信用账户类取款函数进行取款。

【功能 5: 动态添加新账户】

1. 设计

利用 STL 中的容器类 Vector 来建立 Account 数组, 以此达到可以无限制增加用户功能。

利用 STL 的容器类 `multimap` 来建立账户查询系统，使得用户可以查询一段时间内的所有交易。

重载运算符，便于运算。

2. 实现

```
class Account;

class AccountRecord // 账户账目记录
{
private:
    Date date;
    const Account* account;
    double amount;
    double balance;
    string desc;
```

代码片段11 Vector建立Account数组的关键代码

```
public:
    AccountRecord(const Date &date, const Account* account, double amount, double
balance,const string& desc);
    void show()const;
};

AccountRecord::AccountRecord(const Date &date, const Account* account, double amount,
double balance,const string& desc)
    :date(date), account(account), amount(amount), balance(balance), desc(desc) {} // 账户记
录类构造

void AccountRecord::show()const // 账户展示
{
    cout << date << "\t#" << account->getId() << "\t" << amount << "\t" << balance <<
"\t" << desc << endl;
}

typedef multimap<Date, AccountRecord>RecordMap;

RecordMap Account::recordMap; // 多种映射构建

void Account::record(const Date& date, double amount, const string& desc) // 账户记录
```

```

{
    amount = floor(amount * 100 + 0.5) / 100; // 四舍五入取整数
    balance += amount; // 余额变动
    total += amount; // 变动总值变动
    date.show(); // 信息显示
    cout << "\t#" << id << "\t" << amount << "\t" << balance << "\t" << desc << endl;

    AccountRecord record(date, this, amount, balance, desc); // 账目记录
    recordMap.insert(make_pair(date, record)); // map记录账目
}

void Account::query(const Date& begin, const Date& end) // 查询账目
{
    if (begin <= end)
    {
        RecordMap::iterator iter1 = recordMap.lower_bound(begin); // 取这段时间最小值
        RecordMap::iterator iter2 = recordMap.upper_bound(end); // 取这段时间最大值
        for (RecordMap::iterator iter = iter1; iter!=iter2; ++iter) // 迭代器迭代
        {
            iter->second.show();
        }
    }
}

```

代码片段12 multimap的关键代码

```

istream& operator>>(istream& in, Date& date) // 重载>>
{
    int year, month, day;
    char c1, c2;
    in >> year >> c1 >> month >> c2 >> day;
    if (c1 != '-' || c2 != '-') // 用户输入日期不符合要求
        throw runtime_error("Bad time format");
    date = Date(year, month, day);
    return in;
}

ostream& operator<<(ostream& out, const Date& date) // 重载<<
{
    out << date.getYear() << "-" << date.getMonth() << "-" << date.getDay();
    return out;
}

```

```

class Date // 日期类
{
public:
    int operator-(const Date& date) const
    {
        return totalDays - date.totalDays; // 重载-
    }
    bool operator<(const Date& date) const
    {
        return totalDays < date.totalDays; // 重载<
    }
};

```

代码片段13 重载运算符的关键代码

3. 测试

(1) 测试用例 1: q 2008-11-1 2009-1-1 (查询指定时间段内交易)

得到的运行截图如下:

```

2009-1-1      #C1      1000      1000
2009-1-1      Total: -2100      command>q 2008-11-1 2009-1-1
2008-11-5      #C1      1000      1000
2009-1-1      #C1      -100      900      annual fee
2009-1-1      #S1      1000      1000
2009-1-1      #C1      1000      1900
2009-1-1      #C1      -5000      -3100
2009-1-1      Total: -2100      command>

```

分析可得代码可以正常运行多种映射进行交易信息查询。

(2) 测试用例 2: a s S3 0.1 (无限建立储蓄账户)

得到的运行截图如下:

```

2009-1-1      #C1      -5000      -3100
2009-1-1      Total: -2100      command>a s S3 0.1
2009-1-1      #S3 created
2009-1-1      Total: -2100      command>

```

分析可得代码可以在没有固定范围内多次创建储蓄账户。

(3) 测试用例 3: a c C2 10000 0.04 100 (无限建立信用账户)

得到的运行截图如下:

```

2009-1-1      Total: -2100      command>a c C2 10000 0.04 100
2009-1-1      #C2 created

```

分析可得代码可以在没有固定范围内多次创建信用账户。

【功能 6：使用文件记录账户信息】

1. 设计

利用输入流逐行读入指定文件内的历史输入信息，并利用输出流逐行输出交易信息到指定文件中，建立名为 conmmands.txt 的指定文件，用于存放历史交易信息。

重载输入输出流，以便逐行读入文档。

2. 实现

```
std::istream& operator>>(std::istream& in, Date& date);  
std::ostream& operator<<(std::ostream& out, const Date& date); // 输入输出流重载  
  
string cmdLine;  
const char* FILE_NAME = "conmmands.txt";  
ifstream fileIn(FILE_NAME);  
if (fileIn)  
{  
    while (getline(fileIn, cmdLine))  
        controller.runCommand(cmdLine);  
    fileIn.close();  
}  
ofstream fileOut(FILE_NAME, ios_base::app);  
  
while (!controller.isEnd())  
{  
    cout << controller.getDate() << "          Total: " << Account::getTotal() << "  
command>";  
  
    string cmdLine;  
  
    if (getline(cin, cmdLine))
```

```

    {
        if (controller.runCommand(cmdLine))
        {
            fileOut << cmdLine << endl; //写入文件
        }
    }
else
{
    break;
}
}

void Account::record(const Date& date, double amount, const string& desc) // 账户记录
{
    amount = floor(amount * 100 + 0.5) / 100; // 四舍五入取整数
    balance += amount; // 余额变动
    total += amount; // 变动总值变动
    date.show(); // 信息显示
    cout << "\t#" << id << "\t" << amount << "\t" << balance << "\t" << desc << endl;

    AccountRecord record(date, this, amount, balance, desc); // 账目记录
    recordMap.insert(make_pair(date, record)); // map记录账目
}

```

代码片段14 文件读入写入的关键代码

3. 测试

(1) 测试用例：将运行窗口关闭后再次打开

得到运行截图如下：

```

D:\MHL\大二上\个人文件\程设实践\银行系统\1\Book-Bank\Debug\Book-Bank.exe
2008-11-1      #S1 created
2008-11-1      #S2 created
2008-11-5      #C1 created
2008-11-5      #C1      1000      1000
2009-1-1      #C1      -100      900      annual fee
2009-1-1      #C1      1000      1900
2009-1-1      #C1      -5000      -3100
2009-1-1      #S3 created
2009-1-1      #C2 created
(a)add account (d)deposit (w)withdraw (s)show (c)change day (n)next month (q)query (e)exit
2009-1-1      Total: -3100      command>_

```

分析可得代码完成了正常的文档读入与操作写入，每次打开系统都可以保留之前的交易信息。

【功能 7：异常处理机制】

1. 设计

创建一个类 `AccountException` 从 `runtime_error` 派生，保存一个 `Account` 型常指针，指向发生错误的账户，用标准程序库中的 `runtime_error` 构造异常并抛出。

将上述结构加入到判断日期和判断账户是否超额取款时，即可进行异常错误处理。

2. 实现

```
class AccountException :public runtime_error // 账户报错
{
private:
    const Account* account;
public:
    AccountException(const Account *account,const string &msg)
        :runtime_error(msg),account(account) {} // 报错构造
    const Account* getAccount() { return account; } // 取得错误账户
};
```

代码片段15 账户报错类的关键代码

```
Date::Date(int year, int month, int day)
    :year(year), month(month), day(day) // 日期构建
{
    if (day <= 0 || day > getMaxDay()) // 小于0或大于本月最大天数
        throw runtime_error("Invalid date");
}

void Account::error(const string& msg)const // 报错
{
    throw AccountException(this, msg);
    //cout << "Error(#" << id << "):" << msg << endl;
}

void SavingsAccount::withdraw(const Date &date,double amount,const string&desc) // 储蓄
    账户取款
```

```

{
    if (amount > getBalance())
        error("not enough money"); // 超出余额报错
    else
        record(date, -amount, desc); acc.change(date, getBalance()); // 账目记录
}

void CreditAccount::withdraw(const Date& date, double amount, const string &desc) // 信用账户取款
{
    if (amount - getBalance() > credit)
    {
        error("not enough credit"); // 取款大于余额与可用信用度之和
    }
    else
    {
        record(date, -amount, desc); // 记录取款
        acc.change(date, getDebt()); // 余额变动
    }
}
}

```

代码片段16 异常处理抛出的关键代码

```

if (fileIn)
{
    while (getline(fileIn, cmdLine))
        try
        {
            controller.runCommand(cmdLine);
        }
        catch (const std::exception&e)
        {
            cout << "Bad line in " << FILE_NAME << ":" << cmdLine << endl;
            cout << "Error: " << e.what() << endl;
            return 1;
        }
    fileIn.close();
}

while (!controller.isEnd())
{
    cout << controller.getDate() << "          Total: " << Account::getTotal() << "
command>";
}

```

```

string cmdLine;
getline(cin, cmdLine);
try
{
    if (controller.runCommand(cmdLine))
        fileOut << cmdLine << endl; //写入文件

}
catch (AccountException&e)
{
    cout << "Error(# " << e.getAccount()->getId() << "):"
        << e.what() << endl;
}
catch (exception& e)
{
    cout << "Error: " << e.what() << endl;
}

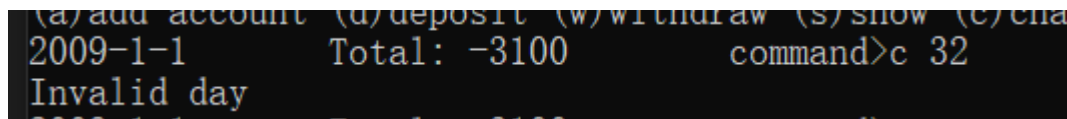
```

代码片段17 异常处理抛出的main中的关键代码

3. 测试

(1) 测试用例 1: c 32 (跳转不合理时间)

得到的运行截图如下:



```

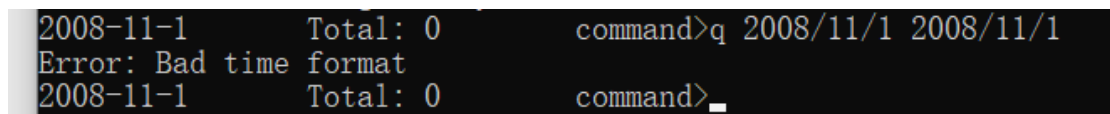
(a)/add account (d)/deposit (w)/withdraw (s)/show (c)/cha
2009-1-1      Total: -3100      command>c 32
Invalid day

```

分析可得代码将会自动判定日期是否合理, 若不合理则不操作。

(2) 测试用例 2: q 2008/11/1 2008/1/1 (不符合日期输入规定)

得到的运行截图如下:



```

2008-11-1      Total: 0      command>q 2008/11/1 2008/11/1
Error: Bad time format
2008-11-1      Total: 0      command>_

```

分析可得代码会自动判定日期输入格式, 若不合理则异常处理抛出。

(3) 测试用例 3: w 0 1000 (超额取款)

得到的运行截图如下:


```
[0] S1 Balance:0  
2008-11-1      Total: 0      command>w 0 1000  
Error(#S1):not enough money  
2008-11-1      Total: 0      command>
```

分析可得代码会自动判定是否超额取款，若超额则异常处理抛出。