



# 《高级语言程序设计》 课程设计

信息与计算机科学系  
2022-2023-2学期



# 个人银行账户管理程序的设计与实现（C++）


S1: 活期储蓄账户类SavingsAccount的设计

S2: 类SavingsAccount的定义和实现分离

S3: SavingsAccount类的完善

S4: Account基类、SavingsAccount类、CreditAccount类的设计

S5: 基于多态性的个人银行账户管理程序的完善



## 提交形式（以S2为例）

📁 学号-姓名

📁 S2

📄 学号-姓名《高级语言程序设计》课程设计报告-S2.docx

📄 account.o  
📄 main.o  
📄 Makefile.win  
📄 S2.dev  
📄 S2.exe  
📄 S2.layout  
📄 S2\_account.cpp  
📄 S2\_account.h  
📄 S2\_account.o  
📄 S2\_main.cpp  
📄 S2\_main.o

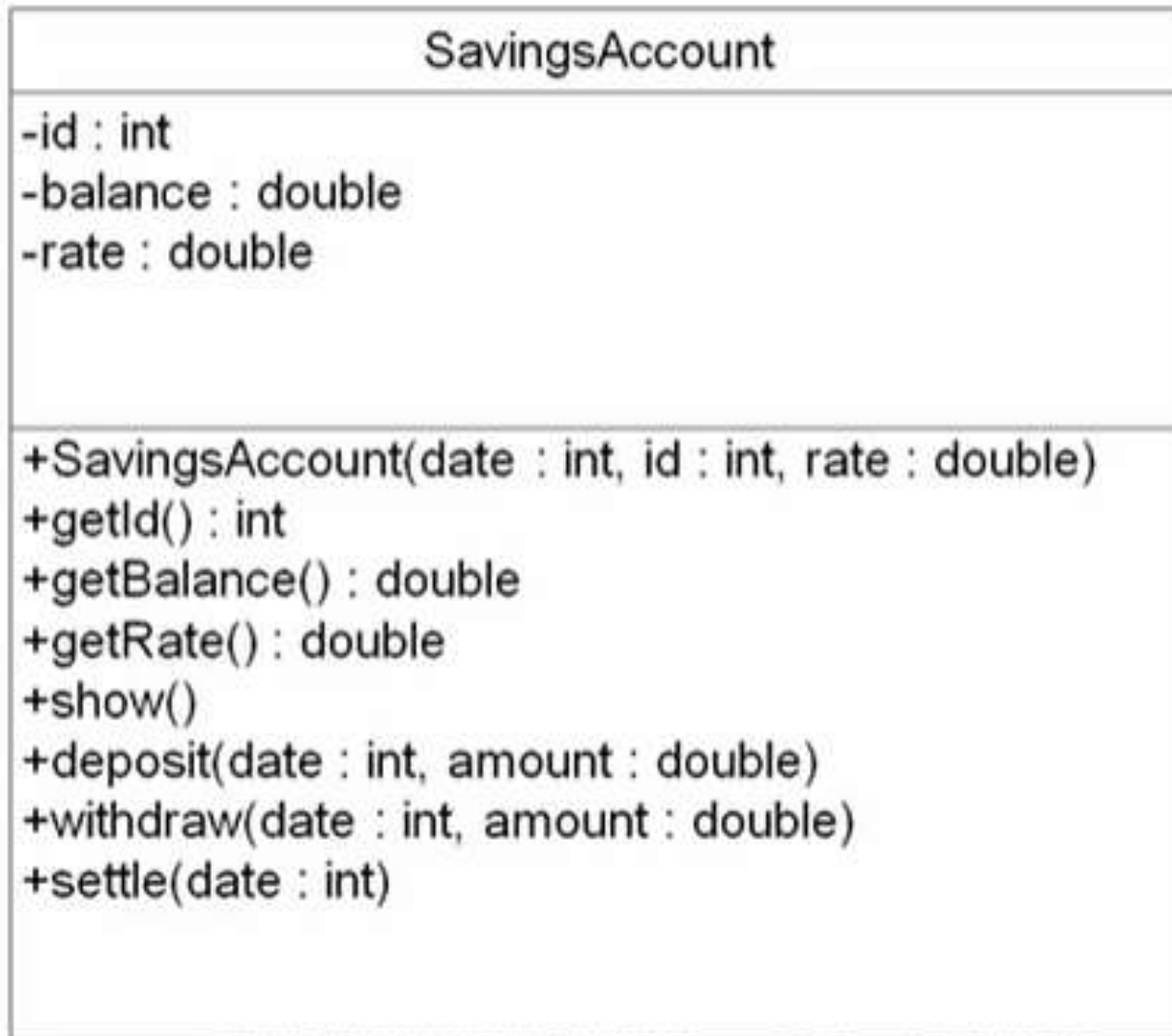
# S1: 活期储蓄账户类SavingsAccount的设计

一个人可以有多个活期储蓄账户，一个活期储蓄账户包括账号（id）、余额（balance）、年利率（rate）等信息，还包括显示账户信息（show）、存款（deposit）、取款（withdraw）、结算利息（settle）等操作。

请设计一个SavingsAccount类，将id, balance, rate均作为其成员数据，将show, deposit, withdraw, settle均作为其成员函数。无论存款、取款还是结算利息，都需要修改当前的余额并将余额的变动输出，这些公共操作由私有成员函数record来执行。



# UML类图



## SavingsAccount

-id : int  
-balance : double  
-rate : double

+SavingsAccount(date : int, id : int, rate : double)  
+getId() : int  
+getBalance() : double  
+getRate() : double  
+show()  
+deposit(date : int, amount : double)  
+withdraw(date : int, amount : double)  
+settle(date : int)  
-record(date: int, amount : double)

无论存款、取款还是  
结算利息，都需要修  
改当前的余额并将余  
额的变动输出



## SavingsAccount

-id : int  
-balance : double  
-rate : double

+SavingsAccount(date : int, id : int, rate : double)  
+getId() : int  
+getBalance() : double  
+getRate() : double  
+show()  
+deposit(date : int, amount : double)  
+withdraw(date : int, amount : double)  
+settle(date : int)  
-record(date : int, amount : double)

为简便起见，该类中的所有日期均用一个整数来表示

该整数是一个以日为单位的相对日期，例如如果以开户日为1，则开户日后的第3天就用4来表示

通过将两个日期相减就可以得到两个日期相差的天数，便于计算利息

# 利息的计算

由于账户的余额是不断变化的

余额\*年利率=年利? ❌

一年当中每天的余额累积起来/一年的总天数=日均余额

日均余额\*年利率=年利? ✓

例如:


如果年利率是1.5%，某账户第5天存入5000元，第45天存入5500元，第90天结算利息。

第5天到第45天之间的余额为5000元（持续40天）

第45天到第90天之间的余额为10500元（持续45天）

第90天结算利息：  $(40 \times 5000 + 45 \times 10500) / 365 \times 1.5\% = 27.64$  元





计算余额的按日累计值，SavingsAccount引入：

私有数据成员**lastDate**：存储上一次余额变动的日期

私有数据成员**accumulation**：存储上一次计算利息以后直到最近一次余额变动时**余额按日累加的值**

私有成员**accumulate**：计算截至指定日期的账户余额按日累积值

**当余额变动时，需要将变动前的余额与该余额所持续的天数相乘，累加到accumulation中，再修改lastDate**

$$\text{accumulation} + \text{balance} * (\text{date} - \text{lastDate})$$



## SavingsAccount

-id : int  
-balance : double  
-rate : double  
-lastDate : int  
-accumulation : double


-record(date: int, amount : double)  
-accumulate(date : int) : double  
+SavingsAccount(date : int, id : int, rate : double)  
+getId() : int  
+getBalance() : double  
+getRate() : double  
+show()  
+deposit(date : int, amount : double)  
+withdraw(date : int, amount : double)  
+settle(date : int)

# 源代码

```
#include <iostream>
#include <cmath>
using namespace std;
class SavingsAccount{ //储蓄账户类
private:
    long int id;//账号
    double balance;//余额
    double rate;//存款的年利率
    int lastDate;//上次变更余额的日期
    double accumulation;//余额按日累加之和

    void record(int date, double amount); //记录一笔帐, date为日期, amount为金额
    double accumulate(int date) const{
        return accumulation+balance*(date-lastDate);
    }

public:
    SavingsAccount(int date, long int id, double rate);//构造函数
    int getId(){return id;}
    double getBalance(){return balance;}
    double getRate(){return rate;}
    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
    void show();//显示账户信息
};
```



```
SavingsAccount::SavingsAccount(int date, long int id, double rate):
lastDate(date), balance(0), id(id), rate(rate), accumulation(0)
{cout<<date<<"\t#"<<id<<" is created"<<endl;
}
```

```
void SavingsAccount::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);}
```

```
void SavingsAccount::withdraw(int date, double amount)
{
    if(amount>getBalance())
        cout<<"Error: not enough money"<<endl;
    else
        record(date, -amount);
}
```

```
void SavingsAccount::settle(int date)
{double interest=accumulate(date)*rate/365; // 计算利息
if(interest!=0)
    record(date,interest);
accumulation=0;
}
```



```
void SavingsAccount::show()
{cout<<"#"<<id<<"\tBalance:"<<balance;
}

int main()
{//创建账户
SavingsAccount sa0(1,21325302,0.015);
SavingsAccount sa1(1,58320212,0.015);
//几笔账目
sa0.deposit(5,5000);
sa1.deposit(25,10000);
sa0.deposit(45,5500);
sa1.withdraw(60,4000);
//开户后第90天到了银行的计息日, 结算所有账户的年息
sa0.settle(90);
sa1.settle(90);
//输出各个账户信息
sa0.show();cout<<endl;
sa1.show();cout<<endl;
return 0;
}
```



E:\C++课程设计\S1.cpp - [Executing] - Dev-C++ 5.11

文件[F] 编辑[E] 搜索[S] 视图[V] 项目[P] 运行[R] 工具[T] AStyle 窗口[W] 帮助[H]

TDM-GCC 4.9.2 64-bit Release

(globals)

项目管理 查看类 调试

S1.cpp

```
1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4 class SavingsAccount{ //储蓄账户类
5     private:
```

E:\C++课程设计\S1.exe

```
1      #21325302 is created
1      #58320212 is created
5      21325302      5000      5000
25     58320212      10000     10000
45     21325302      5500      10500
60     58320212      -4000      6000
90     21325302      27.64     10527.6
90     58320212      21.78     6021.78
#21325302      Balance:10527.6
#58320212      Balance:6021.78

-----
Process exited after 0.0439 seconds with return value 0
请按任意键继续. . .
```



## S2: 类SavingsAccount的定义和实现分离

在S1的基础上进行改进:

- 1、为SavingsAccount类增加一个静态数据成员total，用来记录**各个账户的总金额**，并增加相应的静态成员函数getTotal来对其进行访问。
- 2、将SavingsAccount类中的getBalance，accumulate等不需要改变对象状态的成员函数声明为常成员函数。
- 3、将SavingsAccount类从主函数所在的源文件中分离出，建立两个新的文件account.h和account.cpp，分别存放SavingsAccount类的定义和实现。



SavingsAccount
-id : string -balance : double -rate : double -lastDate : Date -accumulation : double <u>-total : double</u>
-record(date: Date, amount : double )
<<const>> -accumulate(date : Date) : double +SavingsAccount(date : Date, id : int, rate : double) <<const>> +getId() : int <<const>> +getBalance() : double <<const>> +getRate() : double <<const>> +show() +deposit(date : Date, amount : double ) +withdraw(date : Date, amount : double ) +settle(date : Date) <<static>> +getTotal() : double

为SavingsAccount类增加一个静态数据成员**total**，便于全体账户对象共享，既节省存储空间、又不用担心数据一致性问题。

不需要改变对象状态的成员函数，可以将其声明为常成员函数，用**const**实现

增加静态成员函数**getTotal**，来访问增加的静态数据成员**total**



S2 - [S2.dev] - [Executing] - Dev-C++ 5.11

文件(F) 编辑(E) 搜索(S) 视图(V) 项目(P) 运行(R) 工具(T) AStyle 窗口(W) 帮助(H)

TDM-GCC 4.9.2 64-bit Release

(globals)

项目管理 查看类 调试

S2  
S2\_SavingsAccount.cpp  
S2\_SavingsAccount.h  
S2\_main.cpp

S2\_main.cpp S2\_SavingsAccount.cpp S2\_SavingsAccount.h

```
1 #include "E:\C++\S2_SavingsAccount.h"  
2 #include <iostream>  
3 using namespace std;  
4 int main()  
5 {  
    //创建账户
```

E:\C++\S2.exe

```
1 #21325302 is created  
1 #58320212 is created  
5 21325302 5000 5000  
25 58320212 10000 10000  
45 21325302 5500 10500  
60 58320212 -4000 6000  
90 21325302 27.64 10527.6  
90 58320212 21.78 6021.78  
#21325302 Balance:10527.6  
#58320212 Balance:6021.78  
Total:16549.4
```

Process exited after 0.04979 seconds with return value 0  
请按任意键继续. . .

编译器 资源 编译日志

编译结果

Shorten compiler paths

编译结果

- 错误:  
- 警告:  
- 输出文件名: E:\C++\S2.exe  
- 输出大小: 1.83819580078125 MiB  
- 编译时间: 0.91s



## S3: SavingsAccount类的完善

在S2的基础上进行改进：

### 1、将int date升级为Date类。

Date类中包括year, month, day和totalDays, 其中totalDays表示这一天的相对日期。成员函数除了构造函数和用来获得年、月、日的函数外, 还包括用来得到当前月的天数的getMaxDay函数、用来判断当前年是否为闰年的isLeapYear函数、用来将当前日期输出的show函数、用来判断当前日期与指定日期相差天数的distance函数, 这些函数都会被Date类的其他成员函数或者SavingsAccount类的函数调用。



在S2的基础上进行改进：

- 2、银行账号由整型改变为string类型，为deposit, withdraw和settle函数增加一个用来存储该笔账目说明信息的string类型的desc，并且增加一个专用于输出错误信息的error函数。
- 3、主函数中的账户放到一个数组中。
- 4、假定银行对活期储蓄账户的结算日期是每年的1月1日。



## 1、将int date升级为Date类

日期用一个整数表示：

计算两个日期相距天数非常方便，但这种表示方法不直观、对用户不友好


日期用一个类表示：

内含年、月、日三个数据成员，但计算两个日期相差天数比较复杂

计算日期间相差天数：

选取一个比较规整的基准日期，在创建日期对象时将该日期到这个基准日期的相对天数计算出来，记为“相对日期”

计算两个日期相差天数时，将两者的“相对日期”相减，即为相差天数



将公元元年1月1日作为公共的基准日期，将y年m月d日相距这一天的天数记为 $f(y/m/d, 1/1/1)$ ，将其分解为三部分：

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

$$f(y/1/1, 1/1/1) = 365(y-1) + \left\lfloor \frac{y-1}{4} \right\rfloor - \left\lfloor \frac{y-1}{100} \right\rfloor + \left\lfloor \frac{y-1}{400} \right\rfloor$$

$$f(y/m/d, y/m/1) = d-1$$

$f(y/m/1, y/1/1)$ ：由于每个月的天数不同，无法用统一的公式表示



**平年：**指定月份的1日与1月1日相差天数可以由月份 $m$ 唯一确定，因此可以把每月1日到1月1日的天数放在一个数组中，**计算时只要查询数组**，便可以得到  $f(y/m/1, y/1/1)$ 。

**闰年：**仍然可以通过数组查询，只需在 $m > 2$ 时将查得的值加1，该值只依赖于 $m$ 和 $y$ ，记为： **$g(m, y)$** 。



如果把公元元年1月1日的相对日期定为1，那么公元y年m月d日的相对日期为：

$$f(y/m/d, 1/1/1)+1 = 365(y-1) + \left\lfloor \frac{y-1}{4} \right\rfloor - \left\lfloor \frac{y-1}{100} \right\rfloor + \left\lfloor \frac{y-1}{400} \right\rfloor + g(m, y) + d$$



Date
- year : int - month : int - day : int - totalDays : int
+ Date(year : int, month : int, day : int) <<const>> + getYear() : int <<const>> + getMonth() : int <<const>> + getDay() : int <<const>> + getMaxDay() : int <<const>> + isLeapYear() : bool <<const>> + show() <<const>> + distance(date : Date) : int

totalDays表示这一天的相对日期

得到当前月的天数的getMaxDay函数

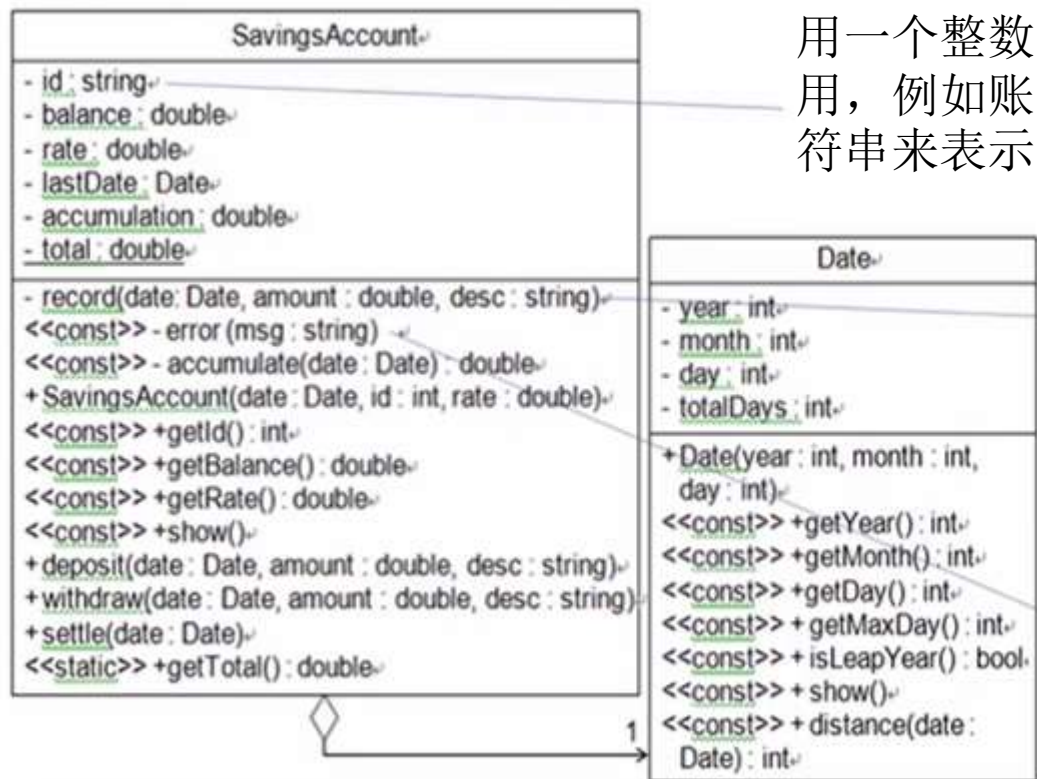
用来判断当前年是否为闰年的isLeapYear函数

用来将当前日期输出的show函数

用来判断当前日期与指定日期相差天数的distance函数




## 2、银行账号由整型改变为string类型，为deposit, withdraw和settle函数增加一个用来存储该笔账目说明信息的string类型的desc，并且增加一个专用于输出错误信息的error函数。



用一个整数来表示银行账号，不符合实际应用，例如账号中可能包含其他字符，改为字符串来表示银行账号更合理

账目列表，每笔账目都没有说明，使用字符串可以为各笔账目增加说明文字

增加了一个用来报告错误的函数，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即可，简化了错误信息的输出



### 3、主函数中的账户放到一个数组中。

由于主程序创建的两个账户为两个独立的变量：

- 只能用名字去引用它们，在主程序末尾分别对两个账户进行结算（**settle**）和显示（**show**）时需要将几乎相同的代码书写两遍
- 如果账户数量增加，将会非常麻烦

解决方案：将多个账户组织在一个数组中，可以把需要对各个账户做的事情放在循环中，避免了代码的冗余

S3 - [S3.dev] - [Executing] - Dev-C++ 5.11

文件(F) 编辑(E) 搜索(S) 视图(V) 项目(P) 运行(R) 工具(T) AStyle 窗口(W) 帮助(H)

IDE-GCC 4.9.2 64-bit Release

(global)

项目管理 查看类 调试

S3\_date.h S3\_date.cpp S3\_SavingsAccount.h S3\_SavingsAccount.cpp S3\_main.cpp

S3  
S3\_SavingsAccount.cpp  
S3\_SavingsAccount.h  
S3\_date.cpp  
S3\_date.h  
S3\_main.cpp

```
4 using namespace std;  
5 double SavingsAccount::total=0;
```

E:\C++\S3.exe

```
2008-11-1 #03755217 is created  
2008-11-1 #02342342 is created  
2008-11-5 #03755217 5000 5000 salary  
2008-11-25 #02342342 10000 10000 sell stock 0323  
2008-12-5 #03755217 5500 10500 salary  
2008-12-20 #02342342 -4000 6000 buy a laptop  
  
2009-1-1 #03755217 17.77 10517.8 interest  
#03755217 Balance:10517.8  
2009-1-1 #02342342 13.2 6013.2 interest  
#02342342 Balance:6013.2  
Total:16531
```

Process exited after 0.05892 seconds with return value 0  
请按任意键继续. . .

编译器 资源 编译日志

中止

编译结果

- 错误: 0
- 警告: 0
- 输出文件名: E:\C++\S3.exe
- 输出大小: 1.84522819519043 MiB
- 编译时间: 1.25s

☐ Shorten compiler paths

# S4: Account基类、SavingsAccount类、CreditAccount类的设计

在S3的基础上进行改进：

- 1、表示信用账户的类CreditAccount的设计。
- 2、基类Account的设计。
- 3、建立一个新类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。

## 信用卡账户业务的特点：

- **允许透支、有信用额度：**总的透支金额应在这个额度之内
- **利息：**信用账户中存钱不会有利息，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，为了简单起见，我们不考虑免息期。
- **每月结算：**与储蓄账户每年结算一次利息不同，信用账户每月进行一次结算，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，假定在**每年1月1日**结算的时候扣缴年费




设计一个基类**Account**用来描述所有账户的共性，再从中派生出：

**SavingsAccount**类

**CreditAccount**类

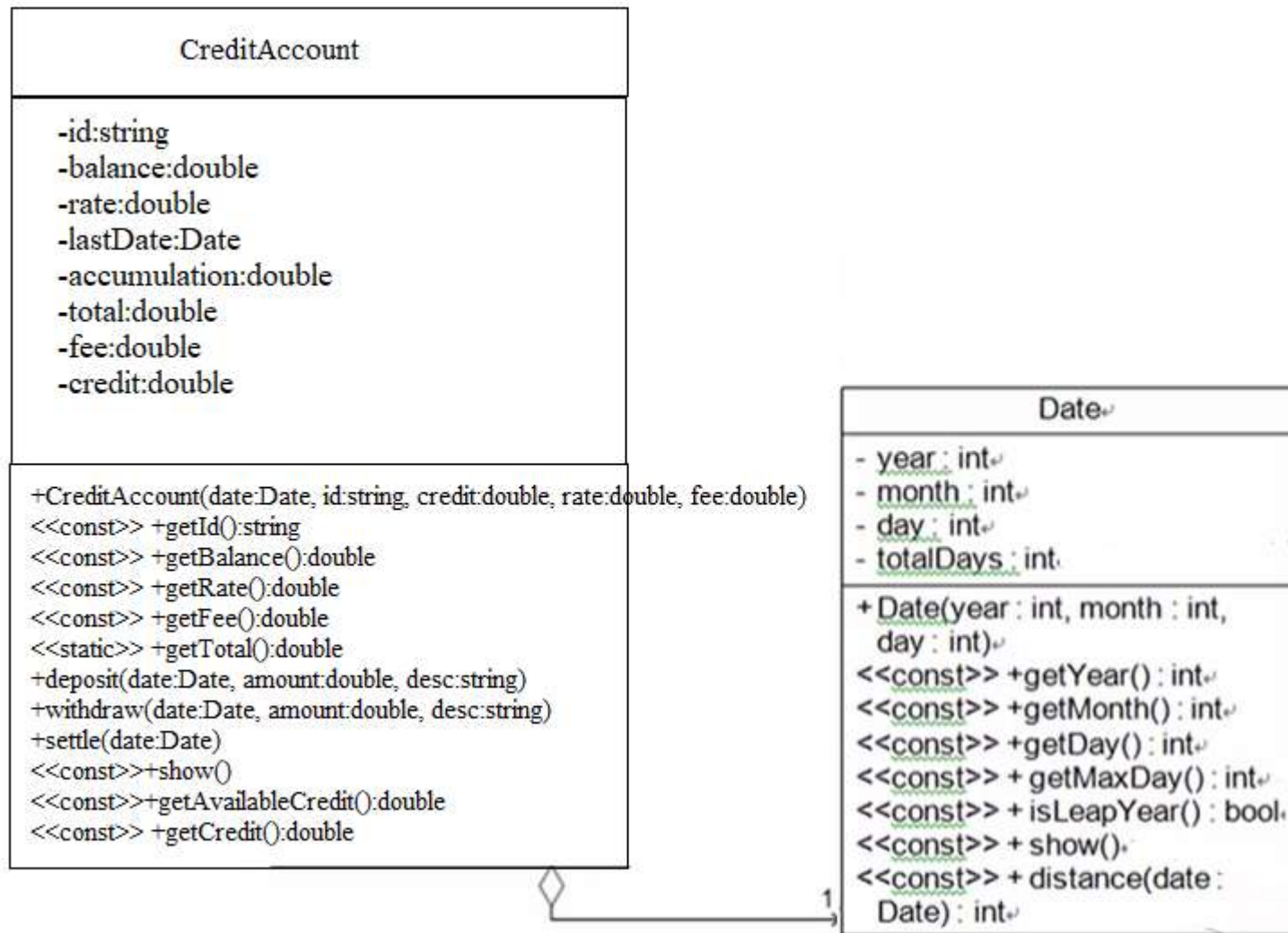
两类账户的利息计算具有很大差异：**计算对象、计算周期**  
因此，计息的任务不能由基类**Account**完成



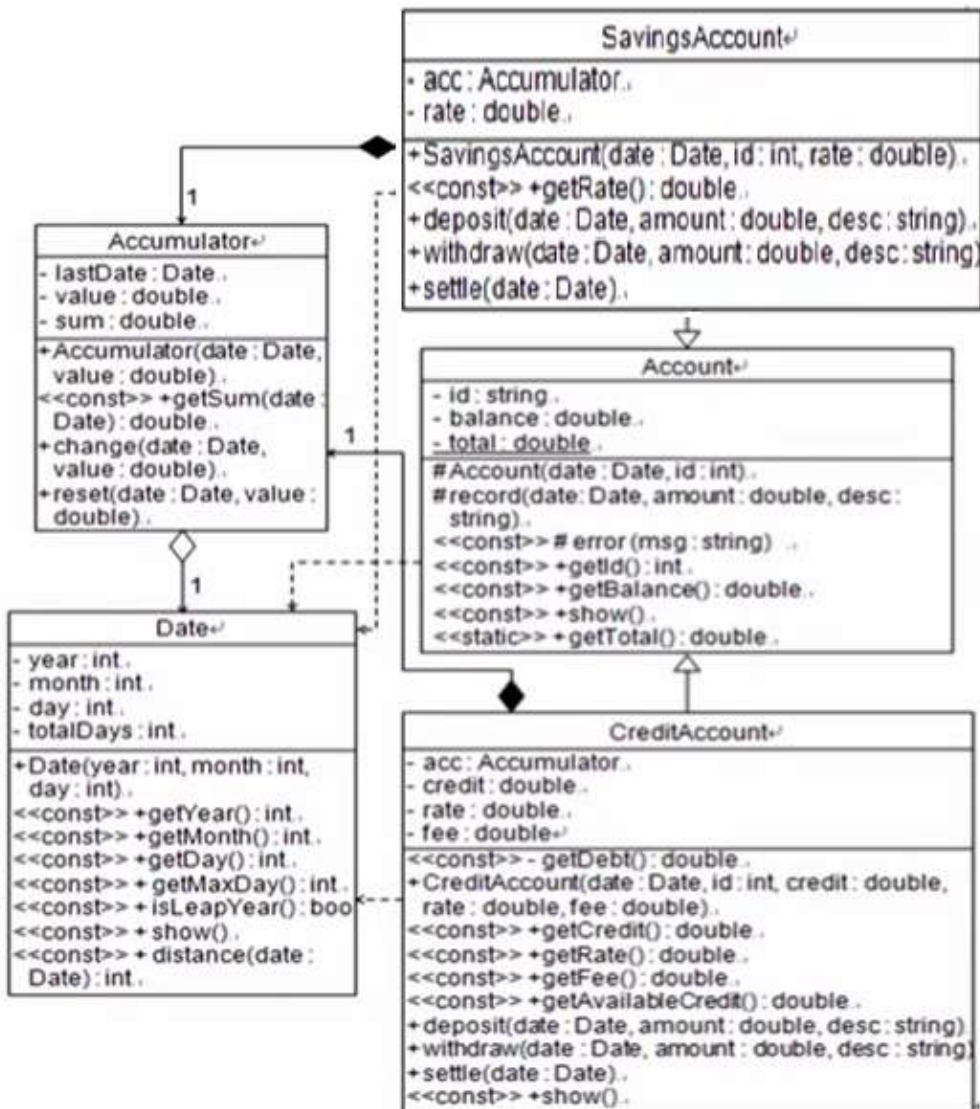
两类账户在计算利息时都需要将某个数值（余额或者欠款金额）按日累加，为了避免编写重复代码，有**两种解决方法**：

- （1）在基类Account中**设立几个保护的成员函数**来协助利息计算，然后在派生类中通过调用这些函数来计算利息
- （2）**建立一个新类**，由该类提供计算一项数值的按日累加之和所需要的接口，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

由于这项功能与其他账户管理功能相对独立，将这项功能从账户类中分类出来，从而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







S4 - [S4.dev] - [Executing] - Dev-C++ 5.11

文件(F) 编辑(E) 搜索(S) 视图(V) 项目(P) 运行(R) 工具(T) AStyle 窗口(W) 帮助(H)

编译选项: (globals)

项目: S4

源文件: S4\_account.cpp, S4\_account.h, S4\_accumulator.h, S4\_date.cpp, S4\_date.h, S4\_main.cpp

```
1 #include "E:\C++S4\S4_account.h"
2 #include <cmath>
3 #include <iostream>
```

编译输出:

```
2008-11-1 #S3755217 created
2008-11-1 #02342342 created
2008-11-1 #C5392394 created
2008-11-5 #S3755217 5000 5000 Salary
2008-11-15 #C5392394 -2000 -2000 buy a cell
2008-11-25 #02342342 10000 10000 sell stock 0323
2008-12-1 #C5392394 -16 -2016 interest
2008-12-1 #C5392394 2016 0 repay the credit
2008-12-5 #S3755217 5500 10500 Salary
2009-1-1 #S3755217 17.77 10517.8 interest
2009-1-1 #02342342 15.16 10015.2 interest
2009-1-1 #C5392394 -50 -50 annual fee

#S3755217 Balance:10517.8
#02342342 Balance:10015.2
#C5392394 Balance:-50 Available credit:9950
Total:20482.9

Process exited after 0.05753 seconds with return value 0
请按任意键继续. . .
```

编译选项: Shorten compiler paths

编译输出: 错误: 0, 警告: 0, 输出文件名: E:\C++S4\S4.exe, 输出大小: 1.8491678237915 MiB, 编译时间: 0.80s

# S5: 基于多态性的个人银行账户管理程序的完善

在S4的基础上进行改进：

- 1、将show函数声明为虚函数
- 2、将Account类定义为抽象类，在Account类中添加deposit，withdraw，settle这三个函数的声明，并将它们都声明为虚函数
- 3、使用运算符重载，对Date类进行修改，将原先用于计算两个日期相差天数的distance函数改为“-”减号运算符，使得计算两个日期相差天数的写法更直观、增加程序可读性。



两个派生类 **SavingsAccount** 和 **CreditAccount** 虽然具有相同的成员函数 **deposit**, **withdraw** 和 **settle**, 但由于其实现不同, 只能在派生类中给出它们的实现, 因而它们是彼此独立的函数。

需要明确知道一个对象的具体类型之后才能够调用它的 **deposit**, **withdraw** 和 **settle** 函数

因此, 不能将3个账户放在一个数组中进行操作。



## 将show函数声明为虚函数

通过指向**CreditAccount/SavingsAccount**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**CreditAccount/SavingsAccount**类定义的**show**函数

如果创建一个**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它们的**show**函数



在**Account**类中添加**deposit, withdraw, settle**这3个函数的声明，并将它们都声明为纯虚函数

通过基类的指针可以调用派生类的相应函数，而不需要给出它们在基类中的实现。

经过这一改动之后，**Account**类就变成了抽象类。



需要注意的是，虽然这几个函数在两个派生类中的原型相同，但是两个派生类的**settle**函数对外接口存在着隐式的差异：

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

储蓄账户一年结算一次，**SavingsAccount**类的**settle**函数需要对每年1月1日调用

信用账户一月结算一次，**CreditAccount**类的**settle**函数需要对每月1日调用



而使用基类**Account**的指针来调用**settle**函数时，事先并不知道该指针所指向对象的具体类型，无法决定采用何种方式调用**settle**函数。

每月1日？还是每年1月1日？

因此**只能将二者的调用要求统一为对每月1日的调用**

同时对活期储蓄账户**SavingsAccount**类的**settle**函数进行修改，使它在结算利息之前先判断是否为1月，只有参数所给的日期是1月时才进行结算。





经过以上修改后，对账户所做的各种操作都可以通过基类的指针来调用

可以把各账户对象的指针都放在一个数组中，只要给出数组索引，就能够对几个账户对象进行操作



使用运算符重载，对**Date**类进行修改：

将原先用来计算两日期相差天数的**distance**函数改为“-”运算符

使得计算两日期相差天数的写法更直观、增加程序的可读性

