WORKED EXAMPLE 9.1

Implementing a Bank Account Class

Problem Statement Write a class that simulates a bank account. Customers can deposit and withdraw funds. If sufficient funds are not available for withdrawal, a $10 overdraft penalty is charged. At the end of the month, interest is added to the account. The interest rate can vary every month.

Step 1 Get an informal list of the responsibilities of your objects.

The following responsibilities are mentioned in the problem statement:

Deposit funds.
Withdraw funds.
Add interest.

There is a hidden responsibility as well. We need to be able to find out how much money is in the account.

Get balance.

Step 2 Specify the public interface.

We need to specify parameter variables and determine which member functions are accessors. To deposit or withdraw money, one needs to know the amount of the deposit or withdrawal:

```
void deposit(double amount);
void withdraw(double amount);
```

To add interest, one needs to know the interest rate that is to be applied:

```
void add_interest(double rate);
```

Clearly, all these member functions are mutators because they change the balance.

Finally, we have

```
double get_balance() const;
```

This function is an accessor because inquiring about the balance does not change it.

Now we move on to constructors. A default constructor makes an account with a zero balance. It can also be useful to supply a constructor with an initial balance.

Here is the complete public interface:

```
class BankAccount
{
public:
   BankAccount();
   BankAccount(double initial_balance);

   void deposit(double amount);
   void withdraw(double amount);
   void add_interest(double rate);

   double get_balance() const;
private:
   . . .
};
```

Step 3 Document the public interface.

```
/**
   A bank account whose balance can be changed by deposits and withdrawals.
*/
class BankAccount
{
public:
   /**
      Constructs a bank account with zero balance.
   */
   BankAccount();

   /**
      Constructs a bank account with a given balance.
      @param initial_balance the initial balance
   */
   BankAccount(double initial_balance);

   /**
      Makes a deposit into this account.
      @param amount the amount of the deposit
   */
   void deposit(double amount);

   /**
      Makes a withdrawal from this account, or charges a penalty if
      sufficient funds are not available.
      @param amount the amount of the withdrawal
   */
   void withdraw(double amount);

   /**
      Adds interest to this account.
      @param rate the interest rate in percent
   */
   void add_interest(double rate);

   /**
      Gets the current balance of this bank account.
      @return the current balance
   */
   double get_balance() const;
private:
   . . .
};
```

Step 4 Determine data members.

Clearly we need to store the bank balance:

```
class BankAccount
{
   . . .
private:
   double balance;
};
```

Do we need to store the interest rate? No—it varies every month, and is supplied as an argument to `add_interest`. What about the withdrawal penalty? The problem description states that it is a fixed $10, so we need not store it. If the penalty could vary over time, as is the case with

most real bank accounts, we would need to store it somewhere (perhaps in a `Bank` object), but it is not our job to model every aspect of the real world.

Step 5 Implement constructors and member functions.

Let's start with a simple one:

```
double BankAccount::get_balance() const
{
   return balance;
}
```

The `deposit` member function is a bit more interesting:

```
void BankAccount::deposit(double amount)
{
   balance = balance + amount;
}
```

The `withdraw` member function needs to charge a penalty if sufficient funds are not available:

```
void BankAccount::withdraw(double amount)
{
   const double PENALTY = 10;
   if (amount > balance)
   {
      balance = balance - PENALTY;
   }
   else
   {
      balance = balance - amount;
   }
}
```

Finally, here is the `add_interest` member function. We compute the interest and then simply call the `deposit` member function to add the interest to the balance:

```
void BankAccount::add_interest(double rate)
{
   double amount = balance * rate / 100;
   deposit(amount);
}
```

The constructors are once again fairly simple:

```
BankAccount::BankAccount()
{
   balance = 0;
}

BankAccount::BankAccount(double initial_balance)
{
   balance = initial_balance;
}
```

This finishes the implementation (see `ch09/worked_example_1/account.cpp` in your companion code).

Step 6 Test your class.

Here is a simple test program that exercises all member functions:

```
int main()
{
   BankAccount harrys_account(1000);
   harrys_account.deposit(500); // Balance is now $1500
   harrys_account.withdraw(2000); // Balance is now $1490
```

```
   harrys_account.add_interest(1); // Balance is now $1490 + 14.90
   cout << fixed << setprecision(2)
      << harrys_account.get_balance() << endl;
   return 0;
}
```

Program Run

```
1504.90
```