Module IN3013/INM173 – Object-Oriented Programming in C++

Solutions to Exercise Sheet 2

1. Here is a simple bank account class:

```
class Account {
        string name;
        double balance;

public:
        Account(string n) : name(n), balance(0) {}
        Account(string n, double initial_balance) :
                name(n), balance(initial_balance) {}

        string get_name() const { return name; }
        double get_balance() const { return balance; }

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

        void withdraw(double amount) {
                balance -= amount;
        }
};
```

   Note we have two kinds of member functions (methods) here:

   - pure functions, like `get_name` and `get_balance`, that return data and do not change the object, as indicated by `const`.
   - procedures, like `deposit` and `withdraw`, whose purpose is to change the object, but do not return anything, as indicated by `void`.

   Separating the methods this way makes the class easier to understand, but many classes have methods that both change the state and return something.

2. This is achieved by the above class, which supplies constructors, but not a default one. The compiler will only generate a default constructor for a class if no constructors are supplied by the programmer.

3. Here is the class with all the active code removed:

```
class Account {
        string name;
        double balance;

public:
        Account(string n);
        Account(string n, double initial_balance);

        string get_name() const;
        double get_balance() const;

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

   Now the constructors will be defined outside the class, and so must be qualified with the class name:

```
Account::Account(string n) : name(n), balance(0) {}
Account::Account(string n, double initial_balance) :
        name(n), balance(initial_balance) {}
```

   Similarly the methods must be qualified when they are defined in this way:

```
string Account::get_name() const { return name; }
double Account::get_balance() const { return balance; }

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

void Account::withdraw(double amount) {
        balance -= amount;
}
```

   Note that the fields of the class are still accessible inside the constructors and methods, just as if they had been defined inside the class.

4. Here is a bank class with a collection of accounts:

```
class Bank {
        vector<Account> account;

public:
        void add_account(const Account &acct) {
                account.push_back(acct);
        }

        void print_accounts() const {
                for (int i = 0; i < account.size(); i++)
                        cout << account[i].get_name() << ": " <<
                                account[i].get_balance() << '\n';
        }
};
```

No constructor is specified, so the compiler will generate a default constructor, with initializes the vector field with its default constructor. This initializes `account` as an empty vector, which is what we want.

The `add_account` procedure take an `Account` object by reference, to avoid a copy, but declares it `const` to indicate that it will not chance the `Account` object argument. The `push_back` method enlarges the vector, initializing the new `Account` object in the vector as a copy of `acct`.

The `print_accounts` procedure loops through the vector in the usual way. Note that the body refers to `account[i]` twice. We could have written the method to declare an alias (reference) to `account[i]`, as follows:

```
void print_accounts() const {
        for (int i = 0; i < account.size(); i++) {
                const Account &acct = account[i];
                cout << acct.get_name() << ": " <<
                        acct.get_balance() << '\n';
        }
}
```

Note that the reference `acct` must also be declared `const`, indicating that the `Account` object will not be changed, and required by the `const` in the procedure declaration.

5. Here is a simple version of `deposit` and `withdraw`:

```
void deposit(string name, double amount) {
        for (int i = 0; i < account.size(); i++)
                if (account[i].get_name() == name)
```

```
                        account[i].deposit(amount);
}

void withdraw(string name, double amount) {
        for (int i = 0; i < account.size(); i++)
                if (account[i].get_name() == name)
                        account[i].withdraw(amount);
}
```

These implementations are flawed, because they keep going after finding a matching name. They also terminate successfully if the name was not found, which would be inappropriate in a real bank.

We could also have used references as in the previous part, but this time the references would not be `const`, as we're modifying the objects they refer to.

6. Again we loop through the accounts in the vector, doing the same thing to each:

```
void add_interest(double rate) {
        for (int i = 0; i < account.size(); i++)
                account[i].deposit(
                    account[i].get_balance() * rate / 100);
}
```