# Tutorial letter 202/2/2018

## Introduction to Programming II
## COS1512

## Semester 2

## School of Computing

**This tutorial letter contains the solution to Assignment 2**

**IMPORTANT INFORMATION:**

Please activate your *myUnisa* and *my*Life email addresses and ensure you have regular access to the *my*Unisa module site COS1512-2018-S2 as well as your e-tutor group site.

Due to regulatory requirements imposed by the Department of National Education the following apply:

To be considered for examination admission in COS1512, a student must meet the following requirement:
Submit assignment 1 or assignment 2 BEFORE 14 September 2018.

Note: This is a blended online module, and therefore your module is available on myUnisa. However, in order to support you in your learning process, you will also receive some study materials in printed format. Please visit the COS1512 course website on myUnisa at least twice a week.

# Content



# 1. Introduction

The purpose of this tutorial letter is to supply the solution for Assignment 2, and to indicate how you should interpret the results you obtained for this assignment. This assignment covers the work discussed in Chapters 10, 11 and 12 of the Study Guide (Tutorial Letter 102), as well as the relevant sections in Chapters 10, 11 and 12, and Appendices 7 and 8 of Savitch. Note that you should have included the input and output of all programs you were required to write.

The assessment and feedback given on your assignment, serve as an indication of your mastering of the study material.
If you received a good mark, you can be confident that you are on the right track. If you did not receive a good mark, it is an indication that you need to revise your study method for COS1512.

# 2. Tutorial matter distributed to date

| | |
|---|---|
| DISK 2018 | Prescribed software |
| COS1512/**101**/3/2018 | First tutorial letter: General information, study programme, exam admission and assignments |
| COS1512/**102**/3/2018 | Study Guide |
| COS1512/**103**/3/2018 | How to create an assignment as a PDF file |
| COS1512/**201**/2/2018 | Solution to Assignment 1 |
| COS1512/104/2/2018 | Exam Tutorial Letter |
| COS1512/**202**/2/2018 | This letter: Solution to Assignment 2 |

If you have not received all of the above-mentioned tutorial matter, please download it from myUnisa at https://my.unisa.ac.za.

# 3. Allocation of marks

When we mark assignments, we comment on your answers. Many students make the same mistakes and consequently we discuss general problems in the tutorial letters. It is, therefore, important to work through the tutorial letters and to make sure you understand our solutions and where you went wrong.

**The maximum number of marks you could obtain for Assignment 2 is 55**. This is converted to a percentage. If you for instance obtained 30 marks for Assignment 2, you received 55% for Assignment 2. This percentage in turn contributes a weight of 80% to the year mark, as can be seen in the summary of the weights allocated to the assignments for COS1512 below.

| Assignment number | Weight |
|---|---|
| 1 | 20 |
| 2 | 80 |
| 3 | 0 |

Questions 2, 4 and 6 have not been marked. However, 5 marks are awarded if you *attempted* questions 2, 4 and 6. **Please note that this is NOT the way exam answers will be marked.** In the exam you will receive marks for specific statements as indicated by the exam question. The purpose of the assignments is to provide you with an opportunity to practice the concepts you are studying. Your marks for the assignment is an indication of whether or not you could implement those concepts. We include complete solutions for *all* questions.

The marks you received for question 1 were determined on the following basis:

| | |
|---|---|
| Question not done | 0/5 |
| Question attempted, but the program does not work at all | 1/5 |
| A good attempt, but there are a few problems with your answer | 3/5 |
| The program works correctly and produces the correct output | 5/5 |

The marks you received for question 3 are indicated by ticks (√) in the solution.

The marks you received for questions 5 and 7 was determined on the following basis:

| | |
|---|---|
| Question not done | 0/15 |
| Question attempted, but the program does not work at all | 5/15 |
| A good attempt, but there are a few problems with your answer | 10/15 |
| The program works correctly and produces the correct output | 15/15 |

In other words, you can obtain a maximum of 5 marks for questions 1, 2, 3, 4 and 6; and a maximum of 15 marks for questions 5 and 7.

Note that not all mistakes are *corrected* – but we will provide informative comments.

If you did not include the program output for questions 1, it means there are “a few problems with your answer” and the maximum you can get is then 3/5. If you did not include the program output for questions 5 and 7, it means there are “a few problems with your answer” and the maximum you can get is then 10/15.

We did not award any marks to assignments submitted more than four weeks after the due date. However, we still provided informative comments.

# 4. Solution to Assignment

| Question 1 | max of 5 marks |
|---|---|

## Question 1 [max of 5 marks]

**Discussion:**

For this question, you had to convert the `struct Employee` into a class. There is essentially only a slight difference between a `struct` and a `class`. A `class` is the same as a `struct` (i.e. a `struct` may also contain both member variables and member functions just like a `class`, even though the examples in Savitch do not show that) except that, by default, all members are inaccessible to the general user of the class. This means that all members of a `struct` are by default `public`, and all members of a `class` are by default `private`. Therefore, we have to specify that the member functions in a class are **`public`**. (As an exercise, omit the `public` keyword from the class and recompile it.) While there are situations where a `struct` is more suitable, as a rule, you should use classes rather than structs. In general, it is preferable to use classes, as classes offer better protection.

An *object* encapsulates or combines data and operations on that data into a single unit. In C++, the mechanism that allows you to combine data and the operations on that data in a single unit is called a **`class`**.

A **`class`** can be seen as a user-defined data type. We use a class to declare or *instantiate* an object. When an object of a specific class is instantiated, the constructor for the class is called by default. The purpose of the constructor is to initialise the member variables of the object automatically. This means that a constructor should not include instructions to obtain values for the member variables (`cin` statements). It also means that the member variables should not be re-declared inside the constructor, since these variables will then be local to the function, and will prevent the constructor function from accessing the member variables to initialise them. Study the constructor for class `Employee` as shown below to see how these concepts are implemented:

```
Employee:: Employee ()           //constructor
//Note: no cin statements and no variable declarations
{
    firstName = " ";
    lastName = " ";
    salary = 0;
}
```

As `firstName`, `lastName`, and `salary` are **`private`** member variables (see page 549 of Savitch 6th edition/ page 581 of Savitch 7th edition/ pages 573-582 of Savitch, 8th edition/pages 589-595 of Savitch, 9th edition/ pages 593-599 of Savitch, 10th edition) of class `Employee`, they cannot be accessed directly in the `main()` function. As a result, `public` member functions `getFirstName()`, `getLastName()` and `getSalary()` are used to access these member variables in order to determine their values. These functions are known as *accessor* functions, while `setFirstName()`, `setLastName()` and `setSalary()` are *mutator* functions. (Study pages 553-554 of Savitch, 6th edition/ pages 585-586 of

Savitch 7th edition/ pages 581-582 of Savitch, 8th edition/ pages 597-598 of Savitch, 9th edition/ pages 601-602 of Savitch, 10th edition).

Note the prototypes for member functions:

```
string getFirstName()const;
string getLastName()const;
float getSalary()const;
```

These member function are accessors - hence the `const` keyword at the end of the function definition. Note that a member function that does not have the `const` keyword at the end of it could mutate (change or modify) the state of the object.

Note that the purpose of an **accessor** function (a function whose name starts with *get*, and sometimes called a get function) is not to obtain an input value for the member variable from a user, BUT to **return the current value of the member variable**. Accessor functions need to indicate a return type corresponding to the member variable for which it returns a value. An accessor function has no parameters. Since an accessor function is a member function of the class, it refers directly to the member variable which value it should return in the body of the function, i.e. there is no need to declare (and return) a local variable inside an accessor function. In fact, if you do declare a local variable inside an accessor function, and return the value of the local variable, it will NOT return the current value of the member variable!

Study the accessor function `getFirstName()` to see how these concepts are implemented:

```
//Accessors to retrieve (get) data from each of member variables
//Note: no parameters; return type of member function is the same as the
// type of the member variable whose value is returned; and no variable
// declarations
string Employee::getFirstName()const
{
    return firstName;
}
```

Mutator functions are used to change or modify member variables of an object. The parameter of the mutator function typically indicates the value according to which the member variable should be changed. For example, the mutator function `setFirstName()` below modifies member variable `firstName` to the string in variable `d`:

```
void Invoice::setFirstName(string d)
{
    description = d;
}
```

### Program listing:

```
#include <iostream>
#include <string>
#include <iomanip>
using namespace std;

//declare class type
class Employee
```

```
{
public:
   Employee();
   Employee(const string fName, const string lName, const float sal);
   void setFirstName(string fName);
   void setLastName(string lName);
   void setSalary(float sal);
   string getFirstName()const;
   string getLastName()const;
   float getSalary()const;
private:
    string firstName;
    string lastName;
    float salary;
};

//Implement class

Employee::Employee()
{
    firstName = " ";
    lastName = " ";
    salary = 0;
}

Employee::Employee(const string fName, const string lName,
          const float sal)
{
    firstName = fName;
    lastName = lName;
    salary = sal;
}

void Employee::setLastName(string lName)
{
    lastName = lName;
}

void Employee::setFirstName(string fName)
{
    firstName = fName;
}

void Employee::setSalary(float sal)
{
    salary = sal;
}

string Employee::getFirstName()const
{
    return firstName;
```

```
}

string Employee::getLastName()const
{
    return lastName;
}

float Employee::getSalary()const
{
    return salary;
}

int main()
{
    Employee anotherEmployee;
    string fName, lName;
    float sal;

//Instantiate an employee
    Employee anEmployee("Joe", "Soap", 1456.00);

//Obtain employee detail
    cout << "Please enter first name of employee: ";
    cin >> fName;
    anotherEmployee.setFirstName(fName);
    cout << "Enter last name of employee: ";
    cin >> lName;
    anotherEmployee.setLastName(lName);
    cout << "Enter salary for employee: ";
    cin >> sal;
    anotherEmployee.setSalary(sal);

//Display employee's salaries
    cout.setf(ios::showpoint);
    cout.setf(ios::fixed);

    cout << endl << setprecision(2) << anEmployee.getFirstName() << ' '
         << anEmployee.getLastName() << "'s salary is R"
         << anEmployee.getSalary() << endl;
    cout << anotherEmployee.getFirstName() << ' '
         << anotherEmployee.getLastName() << "'s salary is R"
         << anotherEmployee.getSalary() << endl << endl;

//Give employees a 10% raise
     float raise =  anEmployee.getSalary() * 1.1;
     anEmployee.setSalary(raise);
     raise =  anotherEmployee.getSalary() * 1.1;
     anotherEmployee.setSalary(raise);

//Display employee's salaries again
    cout << "Having received a 10% raise employees salaries are now:"
```

```
        << endl;
    cout << anEmployee.getFirstName() << ' ' << anEmployee.getLastName()
        << "'s salary is R"
        << anEmployee.getSalary() << endl;
    cout << anotherEmployee.getFirstName() << ' '
        << anotherEmployee.getLastName() << "'s salary is R"
        << anotherEmployee.getSalary() << endl;
    return 0;
}
```

Input and output:

```
Please enter first name of employee: Joanne
Enter last name of employee: Soape
Enter salary for employee: 15446.66

Joe Soap's salary is R1456.00
Joanne Soape's salary is R15446.66

Having received a 10% raise employees salaries are now:
Joe Soap's salary is R1601.60
Joanne Soape's salary is R16991.33
Press any key to continue . . .
```

## Question 2 This question was not marked, and *5 marks* were allocated for attempting it

(a) What is the purpose of the keywords **public** and **private** in the class declaration?
Member variables and member functions declared following a `private` label are accessible only to other member functions of the class. Data members and member functions following a `public` label are accessible to functions that are not members of the class (client functions).

(b) What is the difference between a **class** and an **object**?
A class can be seen as a user-defined data type. A class is a type whose variables are objects. We use a class to declare or instantiate an object.
An object is a variable that has functions associated with it. These functions are called member functions. The object's class determines which member functions the object has.

(c) What does it mean to '**instantiate'** an object?
To '**instantiate'** an object means to declare or create an object. This will allocate memory to the object where the values of the object's member variables can be stored. It will also initialize those member variable to the values specified in the object's constructor.

(d) What is the purpose of a **constructor**?
A constructor is automatically invoked when an object is instantiated (declared). The constructor can initialise some or all of the object's data members to appropriate values.

(e) What is the difference between the **default constructor** and the **overloaded constructor**?
The default constructor takes no arguments. The default constructor is automatically invoked when an object is instantiated. An overloaded constructor is a constructor with arguments so that the user can

specify the values with which the object should be initialized.

(f) What is the purpose of a **destructor**?
Like constructors, destructors are also functions and do not have a return type. However, a class may have only one destructor and the destructor has no parameters. The name of a destructor is the tilde character (~) followed by the name of the class. The destructor automatically executes when the class object goes out of scope and releases the memory used by the object.

(g) What is the purpose of an **accessor**?
An accessor is a member function that accesses the contents of an object in order to return the value of a specific member variable. An accessor for an object does not modify that object. This is why it is good practice to use the keyword const when accessors are defined.

(h) What is the purpose of a **mutator**?
A mutator is a member function that can modify an object. Every non-const member function of a class is potentially a mutator. In the simplest case, a mutator just assigns a new value to one of the data members of the class. In general, a mutator performs some computation and modifies any number of data members.

(i) What is the purpose of the **scope resolution operator**?
The **scope resolution operator ::** is used to specify the class name when giving the function definition for a member function.

(j) What is the difference between the **scope resolution operator** and the **dot** operator?
The scope resolution operator is used to indicate the class name to which an object belongs, whereas the dot operator is used to indicate to which object a member variable belongs.

(k) What is the difference between a **member function** and an **ordinary function**?
A member function is a function that is associated with an object. An ordinary function belongs to the main program.

(l) What is an **abstract data type (ADT)?**
An ADT is a data type that separates the logical properties from the implementation details, i.e. the interface and the implementation of the class are separated.

(m) How do we create an **ADT**?
We define an ADT as a class and place the definition of the class and implementation of its member functions in separate files. The ADT class is then compiled separately from any program that uses it.

(n) What are the advantages of using **ADT**s?
ADTs prevent programmers who use the ADT from having access to the details of how the values and operations are implemented. The user of the ADT only knows about the interface of the ADT and need not know anything about the implementation. This protects the ADT against inadvertent changes. The same ADT class can be used in any number of different programs.

(o) What is **separate compilation**?
**Separate compilation** means that a program is divided into parts that are kept in separate files, compiled separately and then linked together when the program is run.

(p) What are the **advantages** of **separate compilation**?
**Separate compilation** allows programmers to build up a library of classes so that many programs can use the same class.

(q) What is a **derived class?**
A **derived class** is a class that was obtained from another class by adding features through inheritance to

create a specialized class, which also includes the features of the base class.

(r) What is the purpose of inheritance?
**Inheritance** allows one to define a general class and then define more specialized classes by adding some new details to the existing general class.

## Question 3 max of 5 marks

For this question, you were given the following class declaration, and code fragment. You had to explain what is wrong in the code and write code to correct it:

```
class ExamType
{
      public:
          ExamType();
          ExamType(string m, string v, int t, string d);
      private:
          string module;
          string venue;
          int time;
          string date;
};

int main()
      {
          ExamType exams[12];
          for (int i = 0; i < 12; i++)
              if (exams.module[i] == "COS1512")
                  cout << "COS1512 will be written on "
                         << exams.date << " at " << exams.time;
          return 0;
      }
```

This code fragment contains three types of errors, some of which occur on more than one instance:
1. To refer to an object that is an element in an array, you need to place the index directly after the array name, before the name of the member variable in the object you want to access.
2. If you want to refer to a member variable of an object in an array, you need to indicate which element in the array it is.
3. Also, member variables `module`, `date` and `time` cannot be accessed directly. Therefore, you need to add accessors for these member variables to the class declaration.

Therefore **`exams[i].getModule()`** should be used instead of `exams.module[i].`
Also **`exams[i].getDate()`** should be used instead of `exams.date` and **`exams[i].getTime()`** should be used instead of `exams.time.`

**Solution**:

```
class ExamType
{
public:
    ExamType();
    ExamType(string m, string v, int t, string d);
```

```
    string getModule();     //one mark for all 3 additional member functions√
    string getVenue();
    int getTime();
    string getDate();
private:
    string module;
    string venue;
    int time;
    string date;
};

ExamType::ExamType(): module(""), venue(""), time(0), date("")
{
}

ExamType::ExamType(string m, string v, int t, string d): module(m), venue(v),
time(t), date(d)
{
}

string ExamType::getModule()     //one mark for member function √
{
    return module;
}

string ExamType::getVenue()
{
    return venue;
}

int ExamType::getTime()     //one mark for member function √
{
    return time;
}

string ExamType::getDate()  //one mark for member function √
{
    return date;
}

int main()
{
    ExamType exams[12];
    for (int i = 0; i < 12; i++)
        if (exams[i].getModule() == "COS1512")    //one mark for correct use of
                                                  //index for all three cases√
            cout << "COS1512 will be written on "
                 << exams[i].getDate() << " at "
                 << exams[i].getTime();
```

```
    return 0;
}
```

**Question 4 This question has not been marked. A max of 5 marks if you *attempted* this question**

a) Give implementations of both the default constructor and the second (overloaded) constructor.

Default constructor:

```
Chequebook::Chequebook()
{
            Balance = 0.0;
}
```

Overloaded constructor:

```
Chequebook::Chequebook(float AccountBalance)
{
            Balance = AccountBalance;
      }
```

b) Implement the `Deposit()` and `WithDraw()` member functions. The `Deposit()` member function should increment the data member `Balance` with the amount deposited, and the `WithDraw()` member function should decrement the data member `Balance` with the amount withdrawn.

```
void Chequebook::Deposit(float Amount)
{
            Balance = Balance + Amount;
}

void Chequebook::WithDraw(float Amount)
{
            Balance = Balance - Amount;
}
```

c) Implement the `CurrentBalance()` member function. It should return the current balance of the cheque book.

```
float Chequebook::CurrentBalance() const
{
    return Balance;
}
```

d) The member function `Adjust()` should increment the data member `Balance` by R100. Give an implementation for this member function.

```
      void Chequebook::Adjust()
{
            Balance = Balance + 100;
      }
```

e) Overload the stream insertion operator. It should write the balance of the account to the given output stream.

```
ostream& operator<<(ostream & out, Chequebook & cb)
{
          out << cb.CurrentBalance();
          return out;
}
```

f) The statement `++Chequebook1;` should increment the data member `Balance` of `Chequebook1` by R 100. Give three different implementations for the overloaded operator++ to accomplish this:

**METHOD 1:** Using the member function `Adjust()` in other words, defining `operator++` as a non-friend, non-member function:
We add on the prototype to the interface file, as shown in bold.

```
class Chequebook
{
public:
     friend ostream & operator << (ostream & out, const Chequebook &cb);
     Chequebook();
     Chequebook(float AccountBalance);
     void Deposit (float Amount);
     void WithDraw (float Amount);
     float CurrentBalance() const;
     void Adjust();
private:
     float Balance;
};
//as non-friend, non-member function:
Chequebook &operator++(Chequebook &cb);
```

The implementation:

```
Chequebook &operator++(Chequebook &cb)
{
     cb.Adjust();
     return cb;
}
```

NB: There is actually no need to make `operator++` a friend function, as `Adjust()` is a public member and does the work for it.

**METHOD 2**: Implementing overloaded `operator++` as a friend function
We insert the statement in bold into the class interface.

```
class Chequebook
{
public:
    friend Chequebook & operator++(Chequebook &cb);
    friend ostream & operator << (ostream & out, const Chequebook &cb);
    Chequebook();
    Chequebook(float AccountBalance);
    void Deposit (float Amount);
    void WithDraw (float Amount);
    float CurrentBalance() const;
    void Adjust();
private:
  float Balance;
};
```

The implementation:

```
Chequebook const &operator++(Chequebook &cb)
{
  cb.Balance = cb.Balance + 100;
  return cb;
}
```

We could have left the implementation the same as before. However for illustrative purposes - we show that we now have access to the private member variables of `Chequebook` and hence we can manipulate them directly without using the `Adjust()` member function.

**METHOD 3**: Implementing the overloaded `operator++` as a member function.

We insert the statement in bold into the interface:

```
class Chequebook
{
public:
  friend ostream & operator << (ostream & out, const Chequebook &cb);
  Chequebook & operator++();
  Chequebook();
  Chequebook(float AccountBalance);
  void Deposit (float Amount);
  void WithDraw (float Amount);
  float CurrentBalance() const;
  void Adjust();
private:
   float Balance;
};
```

The implementation:

```
Chequebook & Chequebook::operator++()
{
        Balance = Balance + 100;
        return *this;
}
```

**Discussion:**

(1) What's **`*this`**?
A member function of a class can directly access member variables of that class for a given object. Sometimes it is necessary for a function member to refer to the object as a whole, rather than the object's individual member variables. How do you refer to the object as a whole in the implementation of the member function, especially when the object is not passed as a parameter? [Note: it does not make sense to pass a parameter to this member function, because it acts on the calling object (see (2)]. Every object of a class maintains a hidden pointer to itself, and the name of this pointer is **`this`**. In C++, **`this`** is a reserved word.

We use the pointer **`this`** to return the object's value. Consider the statement: `++Chequebook1`; which increments the value of `Chequebook1` by 100 and use pointer **`this`** associated with `Chequebook1` to return the incremented value of `Chequebook1`.

(2) Why does member function **`operator++()`** have no parameters?
The simple answer is, because it is a member function - it has to be called in reference to an object (see page 542 of Savitch 6th ed/ page 574 of Savitch 7th ed/ page 570 of Savitch 8th ed/ page 586 of Savitch 9th ed/ page 590 of Savitch 10th ed). The following syntax is used when you write a call to a member function

of an object:
**`Calling_Object.`**`Member_Function_Name (Argument_List);`

Consider the statement, `++Chequebook1;` here `Chequebook1` would represent the calling object and ++ the member function. This function adds 100 to the calling object. In METHOD 2, however the implementation of `operator`++() has a parameter, as the operator is not a member function but rather a friend function. When the `operator++` is a friend function, it cannot refer to a calling object, thus the object must be passed as a parameter.

The implementation of member function operator++() has a type qualifier (see page 543, Savitch 6th ed/ page 575 of Savitch 7th ed/ page 571 of Savitch 8th ed / page 587 of Savitch 9th ed / page 591 of Savitch 10th ed for a definition). In other words we had to insert `Chequebook::` in the implementation. However the implementation of `operator++()` in METHOD 2 has no type qualifier, as the operator is not a member function but rather a friend function.

Friend functions of a class are actually non-member functions of the class that have access to the private members of that class (see pages 600 – 618 of Savitch, 6th ed/ pages 644 - 663 of Savitch, 7th ed/ pages 636 – 655 of Savitch, 8th ed/654 – 672 of Savitch, 9th ed/658 – 676 of Savitch, 10th ed). Some experts believe that friend functions should be generally avoided as they can manipulate the underlying data representation of an object. But there are valid reasons towards its use in terms of operator overloading. Operators may be overloaded as member functions, or as non-member non-friend functions or as friend functions. See appendix 8, on page 993 of Savitch, 6th ed/ page 1047 of Savitch, 7th ed/ page 1034 of Savitch, 8th ed / page 1062 of Savitch, 9th ed / page 1084 of Savitch, 10th ed for a complete discussion as to why defining operators as friend functions is preferable. As defining operators as friend functions are preferable we present our final version of the program accordingly (see (i)).

i) Test your program with the following input:
```
400
200
300
```
The output is shown below the program listing.

**Full Program Listing:**

```
#include <iostream>
#include <string>
using namespace std;

class Chequebook
{
public:
  friend const Chequebook & operator++(Chequebook &cb);
  friend ostream & operator << (ostream & out, const Chequebook &cb);
  Chequebook();
  Chequebook(float AccountBalance);
  void Deposit (float Amount);
  void WithDraw (float Amount);
  float CurrentBalance() const;
  void Adjust();
private:
  float Balance;
};


Chequebook::Chequebook(float InitBalance)
{
```

```
   Balance = InitBalance;
}

float Chequebook::CurrentBalance() const
{
   return Balance;
}

void Chequebook::Deposit (float Amount)
{
   Balance = Balance + Amount;
}

void Chequebook::WithDraw (float Amount)
{
   Balance = Balance - Amount;
}

void Chequebook::Adjust()
{
   Balance = Balance + 100;
}

ostream& operator<<(ostream &out, const Chequebook &cb)
{
   out<<cb.Balance;
   return out;
}

Chequebook const &operator++(Chequebook &cb)
{
    cb.Balance = cb.Balance + 100;
    return cb;
}

int main()
{
  cout << "Enter the Account balance:";
  float amount;
  cin >> amount;

  Chequebook Chequebook1(amount);

  cout << "Account balance: R" << Chequebook1 << endl;
  cout << "Enter amount to deposit:";
  cin >> amount;

  Chequebook1.Deposit(amount);

  cout << "Balance after deposit: R" << Chequebook1 << endl;
  cout << "Enter amount to withdraw:";
  cin >> amount;

  Chequebook1.WithDraw(amount);

  cout << "Balance after withdrawal: R" << Chequebook1 << endl;

  ++Chequebook1;
```

```
  cout << "Balance after adjusting: R" << Chequebook1;

   return 0;
}
```

**Output:**

```
Enter the Account balance:400
Account balance: R400
Enter amount to deposit:200
Balance after deposit: R600
Enter amount to withdraw:300
Balance after withdrawal: R300
Balance after adjusting: R400 Press any key to continue . . .
```

## Question 5 max of 15 marks

**Discussion:**
For this question, you had to define a class `Voter` as an Abstract Data Type (ADT), so that separate files are used for the interface and implementation. See section 10.3, page 573-582 of Savitch 6th edition / page 605-614 of Savitch 7th / page 601-610 of Savitch 8th edition / page 618-627 of Savitch 9th edition / page 622-631 of Savitch 10th edition for more on **ADT's** and section 12.1, page 683-693 of Savitch 6th edition / page 726-741 of Savitch 7th / page 716-730 of Savitch 8th edition / page 734-748 of Savitch 9th edition / page 738-752 of Savitch 10th edition for more on **separate compilation**.

C++ has facilities for dividing a program into parts that are kept in separate files, compiled separately, and then linked together when the program is run. The header (.h) files are effectively the interfaces of the different classes, and the .cpp files contain the implementations.

For this exercise, you should have created three files:
Voter.h
Voter.cpp
TestVoter.cpp

The ADT class `Voter` has member variables `ID`, `nr_times_voted` and `voted`. The class contains
- a default constructor,
- an overloaded constructor,
- a destructor,
- accessor functions for each of the member variables,
- one mutator, a `void` member function `set_voted();`
- and overloaded friend functions for the operator `++` as well as for the stream extraction operator `>>` and the stream insertion operator `<<`.

Consider the signature for operator `++` overloaded as friend function of the class `Voter`:

```
friend Voter operator++(Voter& V);
```

in comparison with the signature for operator `+` hypothetically overloaded as friend function of the class `Voter` (This was not asked in the question and would not make a lot of sense, we just want to illustrate a point.):

```
friend Voter operator+(const Voter & V1, const Voter & V2);
```

Should we overload the operator + as friend function of the class `Voter`, the **`const`** keyword is used to guarantee that the function will not modify the `Voter` objects. When we add two objects with +, we do not want to change or modify the objects we add. A **`friend`** function may manipulate the underlying structure of the class but the **`const`** keyword ensures that these manipulations do not modify the object in any possible way. We cannot place a **`const`** at the end of the prototype as the function is not a member function.

On the other hand, when we overload the operator ++ as friend function of the class `Voter`, we do want to change the object to increase the number of times voted, and therefore we do not use the `const` keyword. See the implementation for operator ++ below:

```
Voter operator++(Voter& V)       //update number of times voter has voted
{
    ++V.nr_times_voted;
    return V;
}
```

Note the difference between the overloaded friend functions for the stream extraction operator >> and the stream insertion operator <<:

```
    friend istream& operator >> (istream& ins, Voter& the_voter);
    friend ostream& operator << (ostream& outs, const Voter& the_voter);
```

The overloaded stream insertion operator << uses the `const` keyword to ensure that the object which is sent to the output stream will not be modified. In the case of the overloaded stream extraction operator >>, the object retrieved from the input stream, must change. The overloaded stream insertion operator << modifies the output stream and the overloaded stream extraction operator >> modifies the input stream, therefore both the `ostream` and the `istream` must be reference parameters.

Also, note that in the implementation of the overloaded operator >> we do not use `cout` statements to tell the user to type in input or in what format the input should be. The purpose of overloading the operator >> is to allow us to use the operator >> to read in (extract) an object in the same way we would use it to read in or extract an ordinary variable. Consider the implementation of the overloaded >>:

```
istream& operator >> (istream& ins, Voter& the_voter)
{
    ins >> the_voter.ID >> the_voter.nr_times_voted >> the_voter.voted;
    return ins;
}
```

and the way it is used in the application:

```
    Voter a_voter;
          :

    while (!(found) && (infile >> a_voter))
```

In the same way, we overload the stream insertion operator <<

```
ostream& operator << (ostream& outs,  const Voter& the_voter)
{
    outs << the_voter.ID << ' ' << the_voter.nr_times_voted << ' '
         << the_voter.voted;;
    return outs;
}
```

in order to be able to use it as follows:

```
        outfile << a_voter << endl;    //copy voter to voter's roll
```

Tips for separate compilation with multiple files:

- Only the `.cpp` files should be 'added' to the project.
- All files must be in the same directory or folder as the project file.
- The .cpp files must contain the preprocessor directive to include the `.h` files.
  e.g. `#include "Voter.h"`

Note that all the files (`Voter.h`, `Voter.cpp` and `TestVoter.cpp`) must be in the same folder.

The complete listing for the `Voter` ADT and the application program you had to use to test it, is shown below

**Program code:**

**Voter.h**

```
#ifndef VOTER_H
#define VOTER_H
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <iomanip>
using namespace std;

class Voter
{
    public:
    friend istream& operator >> (istream& ins, Voter& the_voter);
    friend ostream& operator << (ostream& outs, const Voter& the_voter);
    friend Voter operator++(Voter& V);
    Voter();                                  //default constructor
    Voter(string ID);      //overloaded constructor
    ~Voter();                                 //destructor
    string get_ID()const;
    int get_nr_times_voted()const;
    bool get_voted()const;
    void set_voted();        //mutator
    private:
        string ID;
        int nr_times_voted;
        bool voted;
    };

#endif
```

**Voter.cpp**

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <iomanip>
#include "Voter.h"
using namespace std;

Voter::Voter()                 //default constructor
{
    ID = "";
    nr_times_voted = 0;
```

```
    voted = false;
}

Voter::Voter(string new_ID)   //overloaded constructor
{
    ID = new_ID;
    nr_times_voted = 0;
    voted = false;
}

Voter::~Voter()         //destructor
{
    ;
}

string Voter::get_ID()const                   //accessor to retrieve ID
{
    return ID;
}

int Voter::get_nr_times_voted()const      //accessor to retrieve
                                            //nr_times_voted
{
    return nr_times_voted;
}

bool Voter::get_voted()const     //accessor to determine whether voter has
                                 //voted or not
{
    return voted;
}

void Voter::set_voted()          //mutator to indicate that voter has voted
{
    voted = true;
}

istream& operator >> (istream& ins,  Voter& the_voter)
{
    ins >> the_voter.ID >> the_voter.nr_times_voted >> the_voter.voted;
    return ins;
}

ostream& operator << (ostream& outs,  const Voter& the_voter)
{
    outs << the_voter.ID << ' ' << the_voter.nr_times_voted << ' '
         << the_voter.voted;;
    return outs;
}

Voter operator++(Voter& V)       //update number of times voter has voted
{
    ++V.nr_times_voted;
    return V;
}
```

**FindVoter.cpp:**

```
#include <iostream>
```

```
#include "Voter.h"
#include <fstream>
using namespace std;

int main()
{
    string voter_ID;
    cout << "Enter the ID for the voter: ";
    cin >> voter_ID;

    ifstream infile;
    infile.open ("VotersRoll.dat");
    if (infile.fail())
    {
       cout<<"Error opening file VotersRoll";
       exit(1); // for opening file"
    }
    ofstream outfile;
    outfile.open ("UpdatedVotersRoll.dat");
    if (outfile.fail())
    {
       cout<<"Error opening file UpdatedVotersRoll";
       exit(1); // for opening file"
    }
    Voter a_voter;
    bool found = false;

    while (!(found) && (infile >> a_voter))
    {
        if (a_voter.get_ID() == voter_ID)
           found = true;

        if (found)
        {

            if (!(a_voter.get_voted()))
            {
                cout << a_voter << " is allowed to vote" << endl;
                ++a_voter;
                a_voter.set_voted();
                cout << a_voter << " has now voted "
                     << a_voter.get_nr_times_voted()
                     << " times" << endl;
            }
            else cout << a_voter << " has already voted in this election"
                      << endl;
        }
        outfile << a_voter << endl;    //copy voter to voter's roll
    }

//copy rest of voter's roll to updated voters'roll
    while (infile >> a_voter)
        outfile << a_voter << endl;

    if (not found)  //message to indicate that voter is not on voters'roll
        cout << voter_ID << " is not on voters' roll" << endl;

    infile.close();
    outfile.close();
```

```
    return 0;
}
```

**Output:**
We show the output for two of the input values, once for a voter who has not yet voted (**19810102009**) indicated by the last 0 in the record in the data file which represents false, and also for a voter who has already voted (**19851010890**), indicated by the last 1 in the in the record in the data file which represents `true`. In each case the values for the voter in the `VotersRoll.dat` before the user voted and in the `UpdatedVotersRoll.dat` after the voter requested to vote is highlighted.

**First example:**
**VotersRoll.dat**

```
19810102009     1     0
19792003008     2     0
19851010890     3     1
19900909897     2     0
19561812567     6     0
19682703345     7     1
```

**Output:**

```
Enter the ID for the voter: 19810102009
19810102009 1 0 is allowed to vote
19810102009 2 1 has now voted 2 times
Press any key to continue . . .
```

**UpdatedVotersRoll.dat**

```
19810102009 2 1
19792003008 2 0
19851010890 3 1
19900909897 2 0
19561812567 6 0
19682703345 7 1
```

**Second example:**
**VotersRoll.dat**

```
19810102009     1     0
19792003008     2     0
19851010890     3     1
19900909897     2     0
19561812567     6     0
19682703345     7     1
```

**Output:**

```
Enter the ID for the voter: 19851010890
19851010890 3 1 has already voted in this election
Press any key to continue . . .
```

**UpdatedVotersRoll.dat**

```
19810102009 1 0
19792003008 2 0
19851010890 3 1
19900909897 2 0
19561812567 6 0
19682703345 7 1
```

**Tip:**
When coding a large abstract data type (ADT) it is best to write the member functions in a stepwise manner. The idea here is to "Code in small increments and then test." For instance, code the constructors and the `friend` function << initially. Then write a small test program to check if these member functions work properly. You could probably write the code for the `friend` function ++ next. Then include statements in your test program to test the `friend` function ++. In other words, you should not write the entire implementation (`Voter.cpp`) and then commence with testing. By coding stepwise, it is easier to isolate errors.

## Question 6 This question has not been marked. A max of 5 marks if you *attempted* this question

**Discussion:**
This question is based on the class `Employee` you created from the struct `Employee` in question 1. You had to define the class `Employee` as an Abstract Data Type (ADT), so that separate files are used for the interface and implementation. See section 10.3, page 573-582 of Savitch 6th edition / page 605-614 of Savitch 7th / page 601-610 of Savitch 8th edition / page 618-627 of Savitch 9th edition / page 622-631 of Savitch 10th edition for more on **ADT's** and section 12.1, page 683-693 of Savitch 6th edition / page 726-741 of Savitch 7th / page 716-730 of Savitch 8th edition / page 734-748 of Savitch 9th edition / page 738-752 of Savitch 10th edition for more on **separate compilation**.

C++ has facilities for dividing a program into parts that are kept in separate files, compiled separately, and then linked together when the program is run. The header (.h) files are effectively the interfaces of the different classes, and the .cpp files contain the implementations.

For this exercise, you should have created three files:
Employee.h
Employee.cpp
main.cpp

You had to overload the stream extraction operator >> and the stream insertion operator << for class `Employee`, and use the overloaded extraction operator >> to read records for employees from a file named `Employees.dat` into an array with a maximum of 20 elements. The program then had to use the array to calculate average salary as well as the highest and lowest salaries. Each employee' s salary also had to be increased by 10% and the overloaded stream insertion operator used to display both the updated salary on the screen as well as to write the updated `Employee` objects to a new data file.

Note the difference between the overloaded friend functions for the stream extraction operator >> and the stream insertion operator <<:

```
friend istream& operator >> (istream& ins, Employee & the_emp);
friend ostream& operator << (ostream& outs, const Employee & the_emp);
```

The overloaded stream insertion operator << uses the `const` keyword to ensure that the object which is sent to the output stream will not be modified. In the case of the overloaded stream extraction operator >>, the object retrieved from the input stream, must change. The overloaded stream insertion operator << modifies the output stream and the overloaded stream extraction operator >> modifies the input stream, therefore both the `ostream` and the `istream` must be reference parameters.

Also, note that in the implementation of the overloaded operator >> we do not use `cout` statements to tell the user to type in input or in what format the input should be. The purpose of overloading the operator >> is to allow us to use the operator >> to read in (extract) an object in the same way we would use it to read in or extract an ordinary variable. Consider the implementation of the overloaded >>:

```
istream& operator >> (istream& ins, Employee & the_emp)
{
    char R; //to handle the R sign before the salary
    ins >> the_emp.firstName >> the_emp.lastName >> R >> the_emp.salary;
    return ins;
}
```

and the way it is used in the application:

```
    Employee EmployeeArray[20];

    int nrEmployees = 0;

    while (infile >> EmployeeArray[nrEmployees])
    {
        nrEmployees++;
    }
```

and

```
    Employee E;
    while (copyOutfile >> E)
        cout << E;
```

Note that we use a character variable `R` to extract the 'R' in the input file before the salary.

We overload the stream insertion operator `<<` as follows:

```
ostream& operator << (ostream& outs, const Employee & the_emp)
{
    outs << the_emp.firstName << ' ' << the_emp.lastName << " R"
         << the_emp.salary << endl;
    return outs;
}
```

in order to be able to use it as follows:

```
    outfile << EmployeeArray[i];
```

and

```
    Employee E;
    while (copyOutfile >> E)
        cout << E;
```

Tips for separate compilation with multiple files:

- Only the `.cpp` files should be 'added' to the project.
- All files must be in the same directory or folder as the project file.
- The .cpp files must contain the preprocessor directive to include the `.h` files.
  e.g. `#include "Employee.h"`

Note that all the files (`Employee.h`, `Employee.cpp` and `main.cpp`) must be in the same folder.

The complete listing for the `Employee` ADT and the application program you had to use to test it, is shown below

**Program code:**

**//Employee.h**

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H
#include <iostream>
```

```
#include <fstream>
#include <cstdlib>
#include <iomanip>
using namespace std;
class Employee
{
public:
   friend istream& operator >> (istream& ins, Employee & the_emp);
   friend ostream& operator << (ostream& outs, const Employee & the_emp);
   Employee();
   Employee(const string fName, const string lName, const float sal);
   void setFirstName(string fName);
   void setLastName(string lName);
   void setSalary(float sal);
   string getFirstName()const;
   string getLastName()const;
   float getSalary()const;
private:
    string firstName;
    string lastName;
    float salary;
};
#endif
```

**// Employee.cpp**

```
//Implement class
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <iomanip>
#include "Employee.h"
using namespace std;

Employee::Employee()
{
    firstName = " ";
    lastName = " ";
    salary = 0;
}

Employee::Employee(const string fName, const string lName,
  const float sal)
{
    firstName = fName;
    lastName = lName;
    salary = sal;
}

void Employee::setLastName(string lName)
{
    lastName = lName;
}

void Employee::setFirstName(string fName)
{
    firstName = fName;
}
```

```
void Employee::setSalary(float sal)
{
    salary = sal;
}

string Employee::getFirstName()const
{
    return firstName;
}

string Employee::getLastName()const
{
    return lastName;
}

float Employee::getSalary()const
{
    return salary;
}

istream& operator >> (istream& ins, Employee & the_emp)
{
    char R; //to handle the R sign before the salary
    ins >> the_emp.firstName >> the_emp.lastName >> R >> the_emp.salary;
    return ins;
}

ostream& operator << (ostream& outs, const Employee & the_emp)
{
    outs << the_emp.firstName << ' ' << the_emp.lastName << " R"
         << the_emp.salary << endl;
    return outs;
}
```

**//main.cpp**

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <iomanip>
#include "Employee.h"

using namespace std;

int main()
{
    ifstream infile;
    infile.open ("Employees.dat");
    if (infile.fail())
    {
       cout<<"Error opening file Employees";
       exit(1); // for opening file"
    }
    ofstream outfile;
    outfile.open ("UpdatedEmployees.dat");
    if (outfile.fail())
    {
       cout<<"Error opening file UpdatedEmployees";
       exit(1); // for opening file"
```

```
    }

    Employee EmployeeArray[20];

    int nrEmployees = 0;

    while (infile >> EmployeeArray[nrEmployees])
        nrEmployees++;

    float highestSalary = EmployeeArray[0].getSalary();
    float lowestSalary = EmployeeArray[0].getSalary();
    float raise, average, sum = 0;

    for (int i = 0; i < nrEmployees; i++)
    {
        sum += EmployeeArray[i].getSalary();
        if (EmployeeArray[i].getSalary() > highestSalary)
            highestSalary = EmployeeArray[i].getSalary();
        if (EmployeeArray[i].getSalary() < lowestSalary)
            lowestSalary = EmployeeArray[i].getSalary();
        //Give employees a 10% raise
        raise =  EmployeeArray[i].getSalary() * 1.1;
        EmployeeArray[i].setSalary(raise);
        outfile << EmployeeArray[i];
    }
    average = sum/nrEmployees;

    cout.setf(ios::showpoint);
    cout.setf(ios::fixed);
    cout << setprecision(2);
    cout << "Salary statistics before the raise:"<< endl;
    cout << "The average salary of the employees is R" << average << endl;
    cout << "The highest salary for any employee is R" << highestSalary
         << endl;
    cout << "The lowest salary for any employee is R" << lowestSalary
         << endl;

    cout << endl << "The updated salaries are as follows:" << endl;
    infile.close();
    outfile.close();
    ifstream copyOutfile;
    copyOutfile.open ("UpdatedEmployees.dat");
    if (copyOutfile.fail())
    {
       cout<<"Error opening file UpdatedEmployees";
       exit(1); // for opening file"
    }
    Employee E;
    while (copyOutfile >> E)
        cout << E;
    copyOutfile.close();

    return 0;
}
```

**Input file (Employees.dat)**

```
Peter Pannier R134000
Lucas Radebe R75000
Albert Mokgaba R80000
```

```
Petrus Petersen R111120
Thabelo Tebogo R200453
Alexa Breda R231334
Jocasta Johnson R199181
Gian Grooteboom R60000
```

**Output file (UpdatedEmployees.dat)**

```
Peter Pannier R147400
Lucas Radebe R82500
Albert Mokgaba R88000
Petrus Petersen R122232
Thabelo Tebogo R220498
Alexa Breda R254467
Jocasta Johnson R219099
Gian Grooteboom R66000
```

**Output for main():**

```
Salary statistics before the raise:
The average salary of the employees is R136386.00
The highest salary for any employee is R231334.00
The lowest salary for any employee is R60000.00

The updated salaries are as follows:
Peter Pannier R147400.00
Lucas Radebe R82500.00
Albert Mokgaba R88000.00
Petrus Petersen R122232.00
Thabelo Tebogo R220498.00
Alexa Breda R254467.00
Jocasta Johnson R219099.00
Gian Grooteboom R66000.00

Process returned 0 (0x0)   execution time : 0.047 s
Press any key to continue.
```

**Tip**:
When coding a large abstract data type (ADT) it is best to write the member functions in a stepwise manner. The idea here is to "Code in small increments and then test." For instance, code the constructors and the `friend` function << initially. Then write a small test program to check if these member functions work properly. You could probably write the code for the `friend` function > next. Then include statements in your test program to test the `friend` function >. In other words, you should not write the entire implementation (`Employee.cpp`) and then commence with testing. By coding stepwise, it is easier to isolate errors.

## Question 7 — max of 15 marks

**7 (a)**
For this question you had to implement the class `Package`. You should have used separate compilation. The interface or header file `Package.h` and implementation `Package.cpp` of class `Package` are shown below.
The member function `Print()` overloads the stream insertion operator << as a **member function** of class `Package`, which is not the way we usually do it. The stream insertion operator << is usually overloaded as a friend function of the class. See for example the solution to question 6.

**Package.h**

```
#ifndef PACKAGE_H
#define PACKAGE_H
#include <iostream>

using namespace std;
class Package
{
public:
      Package(double the_cost, double the_weight,
                const string& the_sender, const string & the_recipient);
      double calculate_cost() const;//multiply weight by cost_per_kilogram
      string get_recipient() const;
      string get_sender() const;
protected:
      double cost_per_kilogram;
      double weight;
private:
      string sender;
      string recipient;
};
#endif
```

**Package.cpp**

```
#include "Package.h"
#include <iostream>
using namespace std;


//overloaded constructor
Package::Package(double the_cost, double the_weight,
                 const string& the_sender, const string & the_recipient)
{
   cost_per_kilogram = the_cost;
   weight = the_weight;
   sender = the_sender;
   recipient = the_recipient;
}

double Package::calculate_cost() const //multiply weight by
                                       //cost_per_kilogram
{
   return weight * cost_per_kilogram;
}

string Package::get_recipient() const
{
      return recipient;
}

      string Package::get_sender() const
{
      return sender;
}
```

**7 (b)**

For this question, you had to test your implementation of class `Package` in a driver program which use the overloaded constructor to instantiate an object of class `Package`, use the accessor functions

to display the names of the sender and recipient of the package and also to calculate the cost of sending the package with the member function `calculate_cost`.

**TestPackage.cpp:**

```
#include "Package.h"
#include <iostream>

using namespace std;

int main()
{
      Package aPackage(12.75, 1.25, "Charles Somerset", "Anne Barnard");
      cout << aPackage.get_sender() << " sent a package to "
           << aPackage.get_recipient() << endl;
      cout.setf(ios::showpoint);
      cout.setf(ios::fixed);
      cout << "This cost R" << aPackage.calculate_cost() << endl << endl;
      return 0;
}
```

**Output:**

```
Charles Somerset sent a package to Anne Barnard
This cost R15.937500

Press any key to continue
```

**7 (c)**

In this question you had to derive a class `TwoDayPackage` from class `Package` and implement it.

**TwodayPackage.h:**

```
# include <iostream>
#ifndef TwoDayPackage_h
#define TwoDayPackage_h
#include "Package.h"

using namespace std;

class TwoDayPackage: public Package
{
 public:
     TwoDayPackage(double the_cost, double the_weight,
                   const string& the_sender, const string & the_recipient,
                   double the_fixedfee);
     double calculate_cost() const;
     void Print() const;
 private:
     double fixedfee;
};
#endif
```

**TwodayPackage.ccp:**

```
#include "Package.h"
#include "TwoDayPackage.h"
#include <iostream>
#include <cstdlib>
#include <iomanip>
```

```
using namespace std;

TwoDayPackage::TwoDayPackage(double the_cost, double the_weight,
                             const string& the_sender,
                             const string & the_recipient,
                             double the_fixedfee):
      Package(the_cost,the_weight,the_sender,the_recipient),
      fixedfee(the_fixedfee)
{}


double TwoDayPackage::calculate_cost() const
{
    return (cost_per_kilogram *weight) + fixedfee;
}


void TwoDayPackage::Print() const
{
    cout.setf(ios::showpoint);
    cout.setf(ios::fixed);
    cout << "Details for the two day package" << endl
        << "Sender: " << get_sender() << endl
        << "Recipient: " << get_recipient() << endl
        << "Cost per kilogram: R" << setprecision(2) << cost_per_kilogram
        << endl
        << "Weight: " << weight << "kg" << endl
        << "Fixed fee: R" << fixedfee << endl
        << "Total cost of delivery: R" << calculate_cost() << endl << endl;
}
```

**7 (d)**

In this question you had to test the class `TwoDayPackage` derived from class `Package` in a driver program. Note that the implementation of the parent class (`Package.cpp`) must be included as part of the project files, and the header file `Package.h` must be included in the application file with the `#include "Package.h"` directive. Both `Package.h` and `Package.cpp` also need to be in the same folder as the other files for class `Package`.

**TestTwodayPackage.cpp:**

```
//Application
#include "Package.h"
#include "TwoDayPackage.h"
#include <iostream>
using namespace std;

int main()
{
      TwoDayPackage myTwoDayPackage (12.75, 1.25, "Charles Somerset", "Anne
      Barnard", 5.00);
      myTwoDayPackage.Print();
      return 0;
}
```

**Output:**

```
Details for the two day package
Sender: Charles Somerset
Recipient: Anne Barnard
```

```
Cost per kilogram: R12.75
Weight: 1.25kg
Fixed fee: R5.00
Total cost of delivery: R20.94

Press any key to continue . . .
```