Tutorial letter 202/1/2018

Introduction to Programming II
COS1512

Semester 1

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-S1 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 13 April 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 2017	Prescribed software
COS1512/**101**/3/2018	First tutorial letter: General information, study programme, exam admission and assignments
COS1512/**102**/1/2018	Study Guide
COS1512/**103**/1/2018	How to create an assignment as a PDF file
COS1512/**201**/1/2018	Solution to Assignment 1
COS1512/**202**/1/2018	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 (note that this is not the way in which questions will be marked in the examination!). 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

Discussion:
For this question, you had to convert the `struct Student` 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 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.`

As `name`, `quiz1, quiz2, midtermExam` and `finalExam` 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) of class `Student`, they cannot be accessed directly in the `main()` function. As a result, `public` member functions `getQuiz1()`, `getQuiz2(), getMidtermExam()`, `getFinalExam()` and `Term()` are used to access these member variables in order to determine their values. These functions are known as *accessor* functions, while `setName()`, `setQuiz1()`, `setQuiz2()`, `setMidtermExam()` and `setFinalExam()` 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).

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 `setQuiz1()` below modifies member variable `quiz1` to `q1`:

```
void Student::setQuiz1(int q1)
{
    quiz1 = q1;
}
```

Note the prototypes for member functions:

```
   string getName()const;
   int getQuiz1()const;
   int getQuiz2()const;
   int getMidtermExam()const;
   int getFinalExam()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. Although member function `calcAverage()` is not an accessor, it should not modify the object, and therefore the prototype for member function `calcAverage()` also has the `const` keyword at the end of the function definition:

```
   int calcAverage()const;
```

Program listing:
```
#include <iostream>
#include <string>
using namespace std;

//Class declare
class Student
{
public:
   Student();
   void setName(string n);
   void setQuiz1(int q1);
   void setQuiz2(int q2);
   void setMidtermExam(int m);
   void setFinalExam(int f);
   string getName()const;
   int getQuiz1()const;
   int getQuiz2()const;
   int getMidtermExam()const;
   int getFinalExam()const;
   int calcAverage()const;
private:
    string name;
    int quiz1;
    int quiz2;
    int midtermExam;
    int finalExam;
};

//Implementation of member functions for class Student
Student::Student()
{
    name = " ";
    quiz1 = 0;
    quiz2 = 0;
    midtermExam = 0;
    finalExam = 0;
}

void Student::setName(string n)
{
    name = n;
}

void Student::setQuiz1(int q1)
{
    quiz1 = q1;
}

void Student::setQuiz2(int q2)
{
    quiz2 = q2;
}

void Student::setMidtermExam(int m)
{
    midtermExam = m;
}
```

```
void Student::setFinalExam(int f)
{
    finalExam = f;
}
string Student::getName()const
{
    return name;
}

int Student::getQuiz1()const
{
    return quiz1;
}

int Student::getQuiz2()const
{
    return quiz2;
}

int Student::getMidtermExam()const
{
    return midtermExam;
}

int Student::getFinalExam()const
{
    return finalExam;
}

int Student::calcAverage()const
{
    return int((quiz1 * 10 + quiz2 * 10)/2 * 0.25 + midtermExam * 0.25
      + finalExam * 0.5);
}
//Main function of application
int main()
{
    Student aStudent;
    string n;
    int q1,q2,m,f;

// Obtain student detail
    cout << "Please enter student's name: ";
    getline(cin, n);
    aStudent.setName(n);
    cout << "Enter mark for quiz 1: ";
    cin >> q1;
    aStudent.setQuiz1(q1);
    cout << "Enter mark for quiz 2: ";
    cin >> q2;
    aStudent.setQuiz2(q2);
    cout << "Enter mark for midterm exam: ";
    cin >> m;
    aStudent.setMidtermExam(m);
    cout << "Enter mark for final exam: ";
    cin >> f;
    aStudent.setFinalExam(f);
//Display student detail
    cout << "\nStudent name: " << aStudent.getName() << endl;
    cout << "Mark for 1st quiz: " << aStudent.getQuiz1() << endl;
    cout << "Mark for 2nd quiz: " << aStudent.getQuiz2() << endl;
    cout << "Mark for midterm exam: " << aStudent.getMidtermExam() << endl;
    cout << "Mark for 1st final exam: " << aStudent.getFinalExam() << endl;
    cout << "Average numeric score for  course: " << aStudent.calcAverage()
         << endl;
    return 0;
}
```

Input and output:

```
Please enter student's name: Johnny Deppo
Enter mark for quiz 1: 7
Enter mark for quiz 2: 5
Enter mark for midterm exam: 65
Enter mark for final exam: 73

Student name: Johnny Deppo
Mark for 1st quiz: 7
Mark for 2nd quiz: 5
Mark for midterm exam: 65
Mark for 1st final exam: 73
Average numeric score for  course: 67
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 PersonType
{
     public:
          PersonType();
          PersonType(string n, int id, string bd);
     private:
          string name;
          int ID;
          string birthday;
};

int main()
{
    PersonType family[20],
                newBaby("Anny Dube", 20180912, "2 Sept");
//Assume family has been initialised
    for (int i = 0; i < 20; i++)
       if (family.birthday[5] == newBaby.birthday)
           cout << family.name[5] << " share a birthday with "
                 <<newBaby.name;
    return 0;
}
```

This code fragment contains two types of errors that occur on more than one instance:

1. From the class declaration, it can be seen that the member variables `birthday` and `name` is **`private`**. Therefore only member functions of class `PersonType` have direct access to member variables `birthday` and `name`. Since member variables `birthday` and `name` cannot be accessed directly, you need to add accessors for member variables `birthday` and `name` to the class definition as shown in bold:

```
class PersonType
{
public:
    PersonType();
    PersonType(string n, int id, string bd);
    string getName(); ½
    int getID();
    string getBirthday();  ½
private:
    string name;
    int ID;
    string birthday;
};

PersonType::PersonType(): name(""), ID(0), birthday(0)
```

```
{
}

PersonType::PersonType(string n, int id, string bd): name(n), ID(id),
birthday(bd)
{
}

string PersonType::getName()      √
{
    return name;
}

int PersonType::getID()
{
    return ID;
}

string PersonType::getBirthday() √
{
    return birthday;
}
```

Therefore **`newBaby.getBirthday()`** should be used instead of `newbaby.birthday` and **`newBaby.getName()`** should be used instead of `newBaby.name`.

2. 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. Therefore **`family[5].getBirthday()`** should be used instead of `family.birthday[5]`..
Also **`family[5].getName()`** should be used instead of `family.name[5]`.

.
```
int main()
{
    PersonType family[20],
                       newBaby("Anny Dube", 20180912, "2 Sept");
    for (int i = 0; i < 20; i++)
       if (family[5].getBirthday() == newBaby.getBirthday())       √
                  cout << family[5].getName() << " share a birthday with "
                       << newBaby.getName(); √
    return 0;
}
```

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

The full program listing below illustrates the use of both the default constructor and the overloaded constructor, the accessors `GetDay`, `GetMonth` and `GetYear`, the mutators `SetDay`, `SetMonth` and `SetYear`, and the private member function `MonthLength`. We present two versions of the final program, one where the operators ++, -- and < are overloaded as member functions and the second a version where operators ++, -- and < are overloaded as friend functions. The stream insertion operator << is overloaded as a friend function in both versions.

Consider the class declaration (interface) as specified in the question:

- The class has a default constructor

  ```
  Date()
  ```

  and an overloaded constructor:

  ```
  Date(int day, int month, int year)
  ```

  These two constructors are necessary in order to permit the Date class to be declared in the following ways:

  ```
  Date christmas2010(25,12,2010)//invokes the overloaded
                                //constructor
  Date today;           //invokes the default constructor
  ```

  The overloaded constructor allows us to initialize an object of class `Date` to a user-specified date. Note that when implementing an overloaded constructor, the parameters for the overloaded constructor specify the values which should be assigned to the member variables, e.g. the value of parameter day is assigned to member variable `theday`, as can be seen below:

  ```
  Date::Date(int day, int month, int year)  //overloaded constructor
  {
   theday = day;
   themonth = month;
   theyear = year;
  }
  ```

  Refer to pages 562 – 572 of Savitch 6th edition / pages 594 -604 of Savitch 7th edition/ pages 592 - 593 of Savitch 8th edition/ pages 606 -613 of Savitch 9th edition for more on constructors.

- Consider operators `++`, `--` and `<` overloaded as **member functions** of the class `Date`:

  ```
  Date operator++();
  Date operator--();
  bool operator<(const Date &d)
  ```

  An operator is overloaded by writing a function definition for the operator. The function name must be the keyword `operator` followed by the symbol for the operator being overloaded.

  From the prototypes given in the class declaration for the operators to calculate the previous and next day (operator `++` and operator `--`)

  ```
  Date operator++();
  Date operator--();
  ```

  we can see that these operators receive no parameters and should return an object of class `Date` also (the new date). Both operator `++` and operator `--` will modify the date and return the current object `*this`. See chapter 11 in the Study Guide (Tutorial Letter 102).

  The prototype for operator `<` shows that this operator should accept one object of class `Date`, and return a boolean value:

  ```
  bool operator<(const Date &d)
  ```

  Parameter `d` should not be modified and is therefore declared as a `const` parameter. The parameter `d` is compared to the calling object when the operator < is used.

- Consider operators `++`, `--` and `<` overloaded as **friend functions** of the class `Date` and compare the function prototypes of the member functions and the `friend` functions:

```
//operators to calculate next and previous days
friend Date operator++(Date d);
friend Date operator--(Date d);
friend bool operator<(const Date &d1, const Date &d2);
```

An operator overloaded as a `friend` function is also overloaded by writing a function definition for the operator. The function header is preceded by the word friend and the function name must be the keyword `operator` followed by the symbol for the operator being overloaded. Note the return type appears between the `friend` modifier and the `operator` keyword.

From the prototypes given in the class declaration for the operators to calculate the previous and next day (operator ++ and operator `--`)

```
friend Date operator++(Date d);
friend Date operator--(Date d);
```

we can see that these operators receive as parameter the object that the operator will modify and should return an object of class `Date` also (the new date). Both operator `++` and operator `--` will modify the date (the parameter) and return the parameter. The parameter should therefore ***not*** be a **`const`** object. See chapter 11 in the Study Guide (Tutorial Letter 102).

The prototype for operator < shows that this operator should accept **two** objects of class `Date`, and return a boolean value:

```
friend bool operator<(const Date &d1, const Date &d2);
```

Neither of the parameters `d1` and `d2` should be modified and they are therefore declared as `const` parameters.

- The << operator is overloaded as a `friend` function. It should write the date in the format *dd monthname yyyy* to the given output stream. The overloaded stream insertion operator returns a reference to `ostream`, and accepts two arguments: a reference to `ostream` and a copy of the object to output. Note that the object which should be written to the output stream (`d` in this case), is a parameter of the operator function and therefore the name of the object must be specified when referring to a member variable or member function of this object, e.g. `d.GetDay()`.

```
//Overload the insertion operator to output a given date
ostream &operator<<(ostream &sout, const Date &d)
{
 sout << d.GetDay() << ' ' << monthName(d.GetMonth()) << ' '
 << d.GetYear();
 return sout;
};
```

- A member function of a class such as the private member function `MonthLength` in class `Date`, can directly access all the member variables of that class. The member variables will therefore *not* be parameters of such a member function. Verify this below:

```
int Date::MonthLength()
{
 if ((themonth == 2) && (((theyear % 4 == 0)
         && (theyear % 100 != 0)) || (theyear % 400 == 0)))
             return 29;
 else
             return daysInMonth[themonth-1];
 }
```

- A friend function of a class can also directly access all the member variables of that class:

```
Date operator++(Date d)
{
 if (d.theday < d.MonthLength())
      d.theday++;
 else
      {
      d.theday = 1;
      if (d.themonth < 12)
           d.themonth++;
      else
      {
           d.themonth = 1;
           d.theyear++;
      };
 };
 return d;
}
```

  Note that the parameter is returned to implement the change in value as a result of applying the operator (**`return d;`**).

- See also the Notes on Overloading Operators as Friends document under Additional Resources on the COS1512 website on myUnisa.

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 overloaded operator ++ next. Then include statements in your test program to test the overloaded operator ++. In other words, you should not write the entire implementation (Date.cpp) and then commence with testing. By coding stepwise, it is easier to isolate errors.

Full program listing for overloading operators as member functions:

```
//Date.cpp: Interface and implementation file for the Date ADT

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

// Days in each month if not a leap year
const int DaysInMonth[12] = {31,28,31,30,31,30,31,31,30,31,30,31};

//Class declaration (interface)
class Date
{
public:
Date();
Date(int day, int month, int year);
int GetDay() const;
int GetMonth() const;
int GetYear() const;
void SetDay(int day);
void SetMonth(int month);
void SetYear(int year);
//operators to calculate next and previous days
 Date operator++();
 Date operator--();
 bool operator<(const Date &d);
private:
int MonthLength();//return the length of current month, taking
                            // into account leap years
int theday;          //the current day
int themonth;     //the current month
int theyear;         //the current year
};


//Implementation

//This function assists in the ostream operator function to print the name
//of a month, although it is not a member function of the class
//return the name of the month as a string
string monthName(int month)
{
      switch(month) {
            case 1:
                  return "January";
                  break;
            case 2:
                  return "February";
                  break;
            case 3:
                  return "March";
                  break;
            case 4:
                  return "April";
                  break;
            case 5:
                  return "May";
                  break;
            case 6:
                  return "June";
                  break;
            case 7:
                  return "July";
                  break;
            case 8:
                  return "August";
                  break;
            case 9:
                  return "September";
                  break;
            case 10:
                  return "October";
                  break;
            case 11:
                  return "November";
                  break;
            case 12:
                  return "December";
                  break;
      };
};
```

```
//Overload the insertion operator to output a given date
ostream &operator<<(ostream &sout, const Date &d)
{
      sout << d.GetDay() << ' ' << monthName(d.GetMonth()) << ' '
             << d.GetYear();
      return sout;
};

Date::Date()     // default constructor
{
      theday = 14;
      themonth = 9;
      theyear = 1752;
}

Date::Date(int day, int month, int year)    //overloaded constructor
{
      theday = day;
      themonth = month;
      theyear = year;
}

//accessors
int Date::GetDay()const
{
      return theday;
}

int Date::GetMonth()const
{
      return themonth;
}

int Date::GetYear()const
{
      return theyear;
}

//mutators
void Date::SetDay(int day)
{
      theday = day;
}

void Date::SetMonth(int month)
{
      themonth = month;
}

void Date::SetYear(int year)
{
      theyear = year;
}

int Date::MonthLength()
{
      if ((themonth == 2) && (((theyear % 4 == 0)
            && (theyear % 100 != 0)) || (theyear % 400 == 0)))
```

```
                return 29;
        else
                return DaysInMonth[themonth-1];
}

//operators to calculate next and previous days
Date Date:: operator++()
{
        if (theday < MonthLength())
                theday++;
        else
                {
                theday = 1;
                if (themonth < 12)
                        themonth++;
                else
                {
                        themonth = 1;
                        theyear++;
                };
        };
        return *this;
}

Date Date:: operator--()
{
        if (theday >1)
                theday--;
        else
        {
                if (themonth > 1)
                        themonth --;
                else
                {
                        themonth = 12;
                        theyear--;
                };
                theday = MonthLength();
        };
        return *this;
};

bool Date::operator<(const Date &d)
{

        if (theyear < d.theyear)
                return true;
        else
        {
                if (theyear > d.theyear)
                        return false;
                else
                {
                        //the years are equal
                        if (themonth < d.themonth)
                                return true;
                        else
                        {
                                if (themonth > d.themonth)
```

```
                           return false;
                      else
                      {
                           //the months are equal
                           if (theday < d.theday)
                                 return true;
                           else
                           {
                                 return false;
                           }
                      }
                 }
           }
      }
};
int main()
{
   Date d1;
   // display the day, month and year of d1
   cout << "Date set by constructor: " << d1 << endl;
   cout << d1.GetDay() << endl;
   cout << d1.GetMonth() << endl;
   cout << d1.GetYear() << endl << endl;

   // change the date to 28 Feb 2000
   d1.SetDay(28);
   d1.SetMonth(2);
   d1.SetYear(2000);

   //advance this date by one and display it
   cout << "New date:" << ++d1 << endl;

   // change the date to 1 Jan 2002
   d1.SetDay(1);
   d1.SetMonth(1);
   d1.SetYear(2002);
   cout << "New date:" << --d1<< endl;

   // change the date to 31 Des 2002
   d1.SetDay(31);
   d1.SetMonth(12);
   d1.SetYear(2002);
   cout << "New date:" << ++d1 << endl;
   Date d2(1,1,2003);
   //determine if date d1 is earlier than d2 and write the result
   //on the screen
   if (d2 < d1)
     cout << d2 << " is earlier than " << d1;
   else
     cout << d2 << " is not earlier than " << d1;
   return 0;
}
```

Output:

```
Date set by constructor: 14 September 1752
14
9
1752
```

```
New date:29 February 2000
New date:31 December 2001
New date:1 January 2003
1 January 2003 is not earlier than 1 January 2003
Process returned 0 (0x0)   execution time : 0.010 s
Press any key to continue.
```

Full program listing for overloading operators as friend functions:

```
//Date.cpp: Interface and implementation file for the Date ADT

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

// Days in each month if not a leap year
const int DaysInMonth[12] = {31,28,31,30,31,30,31,31,30,31,30,31};

//Class declaration (interface)
class Date
{
public:
Date();
Date(int day, int month, int year);
int GetDay() const;
int GetMonth() const;
int GetYear() const;
void SetDay(int day);
void SetMonth(int month);
void SetYear(int year);
//operators to calculate next and previous days
friend Date operator++(Date d);
friend Date operator--(Date d);
friend bool operator<(const Date &d1, const Date &d2);
private:
int MonthLength();//return the length of current month, taking
                            // into account leap years
int theday;          //the current day
int themonth;     //the current month
int theyear;         //the current year
};


//Implementation

//This function assists in the ostream operator function to print the name
//of a month, although it is not a member function of the class
//return the name of the month as a string
string monthName(int month)
{
      switch(month) {
            case 1:
                  return "January";
                  break;
            case 2:
                  return "February";
                  break;
```

```
          case 3:
               return "March";
               break;
          case 4:
               return "April";
               break;
          case 5:
               return "May";
               break;
          case 6:
               return "June";
               break;
          case 7:
               return "July";
               break;
          case 8:
               return "August";
               break;
          case 9:
               return "September";
               break;
          case 10:
               return "October";
               break;
          case 11:
               return "November";
               break;
          case 12:
               return "December";
               break;
     };
};

//Overload the insertion operator to output a given date
ostream &operator<<(ostream &sout, const Date &d)
{
     sout << d.GetDay() << ' ' << monthName(d.GetMonth()) << ' '
            << d.GetYear();
     return sout;
};

Date::Date()     // default constructor
{
     theday = 14;
     themonth = 9;
     theyear = 1752;
}

Date::Date(int day, int month, int year)    //overloaded constructor
{
     theday = day;
     themonth = month;
     theyear = year;
}

//accessors
int Date::GetDay()const
{
     return theday;
```

```
}

int Date::GetMonth()const
{
      return themonth;
}

int Date::GetYear()const
{
      return theyear;
}

//mutators
void Date::SetDay(int day)
{
      theday = day;
}

void Date::SetMonth(int month)
{
      themonth = month;
}

void Date::SetYear(int year)
{
      theyear = year;
}

int Date::MonthLength()
{
      if ((themonth == 2) && (((theyear % 4 == 0)
             && (theyear % 100 != 0)) || (theyear % 400 == 0)))
                  return 29;
      else
                  return DaysInMonth[themonth-1];
}

//operators to calculate next and previous days
Date  operator++(Date d)
{
      if (d.theday < d.MonthLength())
            d.theday++;
      else
            {
            d.theday = 1;
            if (d.themonth < 12)
                  d.themonth++;
            else
            {
                  d.themonth = 1;
                  d.theyear++;
            };
      };
      return d;
}

Date  operator--(Date d)
```

```
{
      if (d.theday >1)
            d.theday--;
      else
      {
            if (d.themonth > 1)
                  d.themonth --;
            else
            {
                  d.themonth = 12;
                  d.theyear--;
            };
            d.theday = d.MonthLength();
      };
      return d;
};

bool operator<(const Date &d1, const Date &d2)
{
      if (d1.GetYear() < d2.GetYear())
            return true;
      else
      {
            if (d1.GetYear() > d2.GetYear())
                  return false;
            else
            {
                  //the years are equal
                  if (d1.GetMonth() < d2.GetMonth())
                        return true;
                  else
                  {
                        if (d1.GetMonth()  > d2.GetMonth())
                              return false;
                        else
                        {
                              //the months are equal
                              if(d1.GetDay() < d2.GetDay())
                                    return true;
                              else
                              {
                                    return false;
                              }
                        }
                  }
            }
      }
};

int main()
{
   Date d1;
   // display the day, month and year of d1
   cout << "Date set by constructor: " << d1 << endl;
   cout << d1.GetDay() << endl;
   cout << d1.GetMonth() << endl;
   cout << d1.GetYear() << endl << endl;

   // change the date to 28 Feb 2000
   d1.SetDay(28);
   d1.SetMonth(2);
   d1.SetYear(2000);

   //advance this date by one and display it
   cout << "New date:" << ++d1 << endl;

   // change the date to 1 Jan 2002
   d1.SetDay(1);
   d1.SetMonth(1);
   d1.SetYear(2002);
   cout << "New date:" << --d1<< endl;

   // change the date to 31 Des 2002
   d1.SetDay(31);
   d1.SetMonth(12);
   d1.SetYear(2002);
   cout << "New date:" << ++d1 << endl;
   Date d2(1,1,2003);
   //determine if date d1 is earlier than d2 and write the result
   //on the screen
   if (d2 < d1)
     cout << d2 << " is earlier than " << d1;
   else
     cout << d2 << " is not earlier than " << d1;
   return 0;
}
```

Output:

```
Date set by constructor: 14 September 1752
14
9
1752

New date:29 February 2000
New date:31 December 2001
New date:1 January 2003
1 January 2003 is not earlier than 31 December 2002
Process returned 0 (0x0)   execution time : 0.020 s
Press any key to continue.
```

Question 5 max of 15 marks

Discussion:
For this question, you had to define a class `PhoneCall` 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 for more on ADT's and section 12.1, page 683-693 of Savitch 6th edition / page 726-741 of Savitch 7th / page 717-727 of Savitch 8th edition / page 734-748 of Savitch 9th 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:

```
PhoneCall.h
PhoneCall.cpp
TestPhoneCall.cpp
```

The ADT class `PhoneCall` has member variables `number`, `length` and `rate`. The class contains

- a default constructor,
- an overloaded constructor,
- a destructor,
- accessor functions for each of the member variables,
- one `void` member function `calcRate();`
- and overloaded friend functions for the operator == as well as for the stream extraction operator >> and the stream insertion operator <<.

Consider operator == overloaded as friend function of the class `PhoneCall`:

```
friend bool operator==(const PhoneCall & call1,
                       const PhoneCall & call2);
```

Note, the **`const`** keyword is used to guarantee that this function will not modify the `PhoneCall` objects. When we compare two `PhoneCall` objects with ==, we do not want to change or modify the objects we compare. 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.

The relational `friend` function == returns a boolean – it returns `true` when the `number` member variables of the two phone calls are the same.

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

```
friend istream& operator >> (istream& ins, PhoneCall& the_call);
friend ostream& operator << (ostream& outs, const PhoneCall& the_call);
```

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, PhoneCall & the_call)
{
    ins >> the_call.number >> the_call.length >> the_call.rate;
    return ins;
}
```

and the way it is used in the application:

```
    PhoneCall aCall;
          :
          :
     while (infile >> aCall)
```

In the same way, we overload the stream insertion operator << in order to be able to use it as follows:

```
  cout << "Detail for a phone call: " << aCall << endl;
```

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 "PhoneCall.h"`

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

The complete listing for the `PhoneCall` ADT and the application program you had to write to test it, is shown below.

Program listing:

//PhoneCall.h

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

class PhoneCall
{
    public:
    friend istream& operator >> (istream& ins, PhoneCall& the_call);
    friend ostream& operator << (ostream& outs,
                                 const PhoneCall& the_call);
    friend bool operator==(const PhoneCall & call1,
                           const PhoneCall & call2);
    PhoneCall();                          //default constructor
    PhoneCall(string new_number);         //overloaded constructor
    ~PhoneCall();                         //destructor
    string get_number()const;             //accessor
    int get_length()const;                //accessor
    float get_rate()const;           //accessor
    float calcCharge();          //calculate amount charged for this call
    private:
        string number;                    // member variables
        int length;
        float rate;
    };

#endif //PHONECALL_H
```

//PhoneCall.cpp

```
#include <iostream>
```

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

PhoneCall::PhoneCall()      //default constructor
{
    number = " ";
    length = 0;
    rate = 0;
}

PhoneCall::PhoneCall(string new_number)   //overloaded constructor
{
    number = new_number;
    length = 0;
    rate = 0;
}

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

string PhoneCall::get_number()const
{
    return number;
}

int PhoneCall::get_length()const
{
    return length;
}

float PhoneCall::get_rate()const
{
    return rate;
}

float PhoneCall::calcCharge()               //determine cost of call
{
    return length * rate;
}

bool operator==(const PhoneCall & call1, const PhoneCall & call2)
{
    return (call1.number == call2.number);
}


istream& operator >> (istream& ins, PhoneCall & the_call)
{
    ins >> the_call.number >> the_call.length >> the_call.rate;
    return ins;
}

ostream& operator << (ostream& outs, const PhoneCall & the_call)
{
    outs << the_call.number << endl << the_call.length << endl
         << the_call.rate;
    return outs;
}
```

//TestPhoneCall.cpp

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

int main()
{
    string theNumber;
    cout << "Enter the phone number for which the total amount "
         << endl << " spend on calls to it, should be determined: ";
    cin >> theNumber;
    PhoneCall theCall(theNumber);
    ifstream infile;
    infile.open ("MyCalls.dat");
    if (infile.fail())
    {
       cout<<"Error opening file";
       exit(1); // for opening file"
    }
    PhoneCall aCall;
    float total_amount = 0;
    int nr_of_calls = 0;
    int longestCall = 0;
     while (infile >> aCall)
     {
        if (aCall == theCall)
        {
             total_amount = total_amount + aCall.calcCharge();
             ++nr_of_calls;
             if (aCall.get_length() > longestCall)
                 longestCall = aCall.get_length();
        }
     }
     cout.setf(ios::fixed);
     cout.precision(2);
     cout << nr_of_calls << " calls were made to " << theNumber
          << endl
          << "The total amount spend on these calls is R"
          << total_amount <<endl
          << "The duration of the longest call was "
          << longestCall <<endl;

     infile.close();
     return 0;
  }
```

Output for TestPhoneCall:

```
Enter the phone number for which the total amount
 spend on calls to it, should be determined: 0337698210
4 calls were made to 0337698210
The total amount spend on these calls is R97.65
The duration of the longest call was 15
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 (`PhoneCall.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 `Student` you created from the struct `Student` in question 1.You had to define the class `Student` 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 for more on ADT’s and section 12.1, page 683-693 of Savitch 6th edition / page 726-741 of Savitch 7th / page 717-727 of Savitch 8th edition/ page 734-744 of Savitch 9th 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:
Student.h
Student.cpp
main.cpp

You had to overload the stream extraction operator >> and the stream insertion operator << for class `Student`, and use the overloaded extraction operator >> to read records for students from a file named Student.dat into an array with a maximum of 20 elements. The program then had to calculate the weighted average for each student (similar to what was done in question 1) and display each student’s marks and weighted average on the screen. The program also had to calculate the class average.

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

```
   friend istream& operator >> (istream& ins, Student & the_student);
   friend ostream& operator << (ostream& outs, const Student &
the_student);
```

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, Student & the_student)
{
    getline(ins,the_student.name);
    ins >> the_student.quiz1 >> the_student.quiz2
        >> the_student.midtermExam >> the_student.finalExam;
    ins.get();
    return ins;
}
```

and the way it is used in the application:

```
    Student COS1512[20]; //20 students in COS1512

    int nrStudents = 0;
    while (infile >> COS1512[nrStudents])
        nrStudents++;
```

Note that we modified the input file so that the student's name appears on one line and his marks on the following line (see Students.dat below) to allow us to use `getline` to input the name and surname. You could have used two strings to input the name and surname and then combined it into one string and used the `setName()` mutator to assign the name to the member variable `name`.

We overload the stream insertion operator << as follows:

```
ostream& operator << (ostream& outs, const Student & the_student)
{
    outs << the_student.name << "\t\t" << the_student.quiz1 << "\t"
         << the_student.quiz2 << "\t" << the_student.midtermExam << "\t"
         <<the_student.finalExam << "\t\t" << the_student.calcAverage()
<<endl;
    return outs;
}
```

in order to be able to use it as follows:

```
        cout << COS1512[i];
```

Note the tabs "\t" we used to produce a neat table in the output.

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 "Student.h"`

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

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

Program code:

//Student.h

```
#include <iostream>
#include <string>
#ifndef STUDENT_H
#define STUDENT_H
using namespace std;
```

```
class Student
{
public:
   Student();
   void  (string n);
   void setQuiz1(int q1);
   void setQuiz2(int q2);
   void setMidtermExam(int m);
   void setFinalExam(int f);
   string getName()const;
   int getQuiz1()const;
   int getQuiz2()const;
   int getMidtermExam()const;
   int getFinalExam()const;
   int calcAverage()const;
   friend istream& operator >> (istream& ins, Student & the_student);
   friend ostream& operator << (ostream& outs, const Student &
the_student);

private:
    string name;
    int quiz1;
    int quiz2;
    int midtermExam;
    int finalExam;
};
#endif
```

//Student.cpp

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

//Implementation of member functions for class Student
Student::Student()
{
    name = " ";
    quiz1 = 0;
    quiz2 = 0;
    midtermExam = 0;
    finalExam = 0;
}

void Student::setName(string n)
{
    name = n;
}

void Student::setQuiz1(int q1)
{
    quiz1 = q1;
}

void Student::setQuiz2(int q2)
{
    quiz2 = q2;
}

void Student::setMidtermExam(int m)
{
    midtermExam = m;
}

void Student::setFinalExam(int f)
{
    finalExam = f;
}
string Student::getName()const
{
    return name;
}

int Student::getQuiz1()const
{
    return quiz1;
}

int Student::getQuiz2()const
{
    return quiz2;
}

int Student::getMidtermExam()const
{
    return midtermExam;
}

int Student::getFinalExam()const
{
    return finalExam;
}

int Student::calcAverage()const
{
    return int((quiz1 * 10 + quiz2 * 10)/2 * 0.25 + midtermExam * 0.25
    + finalExam * 0.5);
}

istream& operator >> (istream& ins, Student & the_student)
{
    getline(ins,the_student.name);
    ins >> the_student.quiz1 >> the_student.quiz2
        >> the_student.midtermExam >> the_student.finalExam;
    ins.get();
    return ins;
}

ostream& operator << (ostream& outs, const Student & the_student)
{
    outs << the_student.name << "\t\t" << the_student.quiz1 << "\t"
         << the_student.quiz2 << "\t" << the_student.midtermExam << "\t"
         <<the_student.finalExam << "\t\t" << the_student.calcAverage()
<<endl;
    return outs;
}
```

//main.cpp
```
#include <iostream>
#include <string>
#include <fstream> // for file
#include <cstdlib> // for exit using namespace std;
#include <cstring>

#include "Student.h"
using namespace std;

int main()
{
    Student COS1512[20]; //20 students in COS1512
    ifstream infile;
    string inName;

    cout << endl << "Enter the input file name: " << endl;
    cin >> inName;
    infile.open(inName.c_str());
    if (infile.fail())
    {
        cout << "Cannot open file " << inName << " Aborting!" << endl;
        exit(1);
    }

    int nrStudents = 0;
    while (infile >> COS1512[nrStudents])
        nrStudents++;

    cout << endl << "Nr students = " << nrStudents << endl <<endl;

    cout << "COS1512 Class Results" <<endl;
    cout << "Student name"  << "\t" << "\tQuiz1" << "\tQuiz2"
         << "\tMidterm"  << "\tFinal Exam" << "\tWeighted Average" << endl;
    int total = 0;
    for(int i = 0; i <nrStudents; i++)
    {
        total += COS1512[i].calcAverage();
        cout << COS1512[i];
    }

    int classAverage = total/nrStudents;

    cout << endl << "Class average for COS1512: " << classAverage << endl;

    infile.close();
    return 0;
}
```

Input file (Students.dat)
```
Peter Pan
 5 3 45 51
Wendy Hill
 7 5 63 58
Alice Mokgaba
 8 6 51 67
Precious Peters
```

```
 5 7 49 46
Thumi Tebogo
 4 7 69 65
```

Output for main():

```
Enter the input file name:
Students.dat

Nr students = 5

COS1512 Class Results
Student name             Quiz1    Quiz2    Midterm  Final  Exam       Weighted
Average

Peter Pan               5        3        45       51                46
Wendy Hill              7        5        63       58                59
Alice Mokgaba           8        6        51       67                63
Precious Peters         5        7        49       46                50
Thumi Tebogo            4        7        69       65                63

Class average for COS1512: 56

Process returned 0 (0x0)   execution time : 6.160 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 (`Runner.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 `Student`. You should have used separate compilation. The interface or header file `Student.h` and implementation `Student.cpp` of class `Student` are shown below.
The member function `display_info()` overloads the stream insertion operator << as a **member function** of class `Student`, 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.

Student.h
```
#ifndef STUDENT_H
#define STUDENT _H
#include <iostream>

using namespace std;

class Student
{
```

```
public:
   Student();
   Student(string sName, string sNr, string sAddress, string sDegree);
   ~Student();
   void display_info(ostream & out)const;
   string get_name()const;
   string get_stdtNr()const;
   string get_address()const;
   string get_degree()const;
   float calcFee();
private:
   string name;
   string stdtNr;
   string address;
   string degree;
};
#endif
```

Student.cpp

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

Student::Student()
{
    name = "";
    stdtNr = "";
    address = "";
}

Student::Student(string sName, string sNr, string sAddress, string
                       sDegree)
{
    name = sName;
    stdtNr = sNr;
    address = sAddress;
    degree = sDegree;
}

Student::~Student()
{
    //destructor - do nothing
}

void Student::display_info(ostream & out)const
{
    out << "Name :" << name << endl;
    out << "StdtNr :" << stdtNr << endl;
    out << "Address :" << address << endl;
    out << "Degree :" << degree << endl;
}

string Student::get_name()const
{
    return name;
}

string Student::get_stdtNr ()const
```

```
{
    return stdtNr;
}

string Student::get_address()const
{
    return address;
}

string Student::get_degree()const
{
    return degree;
}

float Student::calcFee()
{
    if (degree[0] == 'B')
       return 500;
       else return 600;
}
```

7 (b)
For this question, you had to test your implementation of class `Student` in a driver program which use the overloaded constructor to instantiate an object of class `Student`, use the accessor functions to display the values of the member variables of the instantiated object and also display the specifications of the instantiated object with the member function `display_info()`.

TestStudent:

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

using namespace std;

int main()
{
    Student me("Mary Mbeli","12345678","Po Box 16, Pretoria, 0818","BSc" );
    cout << "Name: " << me.get_name() << endl << "Student number: "
         << me.get_stdtNr()
         <<endl << "Address: "<< me.get_address() << endl
         << "Degree: "<< me.get_degree() << endl << endl;
    cout << "Using member function display_info to display "
         << "student info:"
         << endl;
    me.display_info(cout);
    cout << "Fee due: R"  << me.calcFee() << endl;
    return 0;
}
```

Output:

```
Name: Mary Mbeli
Student number: 12345678
Address: Po Box 16, Pretoria, 0818
Degree: BSc

Using member function display_info to display student info:
Name :Mary Mbeli
StdtNr :12345678
Address :Po Box 16, Pretoria, 0818
```

```
Degree :BSc
Fee due: R500

Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```

7 (c)
In this question you had to derive a class `PostgradStd` from class `Student` and implement it.

PostgradStd.h:
```
#ifndef POSTGRADSTD_H
#define POSTGRADSTD_H
#include "Student.h"
#include <iostream>
using namespace std;


class PostgradStd  :public Student
{
    public:
        PostgradStd  ();
        ~PostgradStd  ();
        PostgradStd  (string sName, string sNr, string sAddress,
                      string sDegree, string sDissertation);
        void display_info(ostream & out)const;
        string get_dissertation()const;
        float calcFee();
    private:
        string dissertation;
};

#endif // POSTGRADSTD  _H
```

PostgradStd .cpp:
```
#include "PostgradStd.h"
//#include "Student.h"
#include <iostream>
using namespace std;

PostgradStd ::PostgradStd  ():Student()
{
    dissertation = "";

}

PostgradStd ::~PostgradStd  ()
{
}

PostgradStd ::PostgradStd  (string sName, string sNr, string
              sAddress, string sDegree, string sDissertation):
Student(sName,sNr,sAddress,sDegree), dissertation(sDissertation)
{
};

void PostgradStd  ::display_info(ostream & out)const
{
    out << "Name :" << get_name() << endl;
    out << "StdtNr :" << get_stdtNr() << endl;
    out << "Address :" << get_address() << endl;
    out << "Degree: " << get_degree() << endl;
    out << "Dissertation: " << dissertation << endl;
}

string PostgradStd::get_dissertation()const
{
    return dissertation;
}

float PostgradStd  ::calcFee()
{
    return (12000);
}
```

7 (d)
In this question you had to test the class `PostgradStd` derived from class Student in a driver program. Note that the implementation of the parent class (`Student.cpp`) must be included as part of the project files, and the header file `Student.h` must be included in the `.h` and `.cpp` files for class `PostgradStd` with the `#include "Student.h"` directive. However, the `#include "Student.h"` directive should not be included in the implementation file `TestPostgradStd.cpp`.
Both `Student.h` and `Student.cpp` also need to be in the same folder as the other files for class `PostgradStd`.

TestPostgradStd.cpp:
```
//#include "Student.h"
#include "PostgradStd.h"
#include <iostream>

using namespace std;

int main()
{
    PostgradStd me("Mary Mbeli","12345678",
    "Po Box 16, Pretoria, 0818","PhD","How to get a PhD");
    cout << "Name: " << me.get_name() << endl << "Student number: "
         << me.get_stdtNr() <<endl << "Address: "
         << me.get_address() << endl << "Degree: "<< me.get_degree()
         << endl << "Dissertation: " << me.get_dissertation()
         << endl << endl;
    cout << "Using member function display_info to display "
         << "student info:"
         << endl;
    me.display_info(cout);
    cout << "Fee due: R"  << me.calcFee() << endl;
    return 0;

    return 0;
}
```

Output:
```
Name: Mary Mbeli
Student number: 12345678
Address: Po Box 16, Pretoria, 0818
Degree: PhD
```

```
Dissertation: How to get a PhD

Using member function display_info to display student info:
Name :Mary Mbeli
StdtNr :12345678
Address :Po Box 16, Pretoria, 0818
Degree: PhD
Dissertation: How to get a PhD
Fee due: R12000

Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```