Tutorial letter 202/1/2015

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 COS1512-2015-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 2 April 2015.

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 2015	Prescribed software
COS1512/**101**/3/2015	First tutorial letter: General information, study programme, exam admission and assignments
COS1512/**102**/2/2015	Study Guide
COS1512/**103**/2/2015	How to create an assignment as a PDF file
COS1512/**201**/2/2015	Solution to Assignment 1
COS1512/**202**/2/2015	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 75. This is converted to a percentage. If you for instance obtained 40 marks for Assignment 2, you received 53% 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 3, 4 and 6 have not been marked. However, 5 marks are awarded if you *attempted* questions 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	2/5
A good attempt, but there are a few problems with your answer	4/5
The program works correctly and produces the correct output	5/5

The marks you received for question 2 were determined on the following basis:

Question not done	0/30
Each correct response to questions (a,c,d,i,j,k,l,m,n and q) = 2 marks	20/30
Each correct response to questions (b,e,f,g,o,p and r) = 1 mark	7/30
Question (h) = 3 marks	3/30

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

Question not done	0/15
Question attempted, but the program does not work at all	6/15
A good attempt, but there are a few problems with your answer	12/15
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, 4 and 6; a maximum of 30 marks for question 2; and a maximum of 15 marks for questions 5 and 7; while no marks are allocated for question 3.

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 4/5. 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

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 Address` 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 `struct`s. 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 `Address` as shown below to see how these concepts are implemented:

```
//implementation
Address::Address()    //constructor
{
    streetName = " ";
    streetNr = 0;
    city = " ";
    postalCode = "0000";
}
```

As `streetName`, `streetNr`, `city`, and `postalCode` are **`private`** member variables (see page 549 of Savitch 6th edition/ page 581 of Savitch 7th edition/ page 579 of Savitch 8th edition) of class `Address`, they cannot be accessed directly in the `main`() function. As a result, `public` member functions `getStName()`,`getStNr()`,`getCity()` and `getPCode()` are used to access these member variables in order to determine their values. These functions are known as *accessor* functions, while `setStName()`, `setStNr()`,`setCity()` and `setPCode()` 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).

Note the prototypes for member functions:

```
      string getStName() const;
      int getStNr()const;
      string getCity()const;
      string getPCode()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 may possibly *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 `getStName()` to see how these concepts are implemented:

```
//Accessors to retrieve (get) data from each of member variables
string Address::getStName() const
{
    return streetName;
}
```

Mutator functions (functions whose names start with *set*, and sometimes called 'set 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 `setStName()` below modifies member variable `streetName` to `s`:

```
void Address::setStName(string s)
{
    streetName = s;
}
```

Program listing:

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

class Address
{
public:
    Address();
    void setStName(string s);
    void setStNr(int n);
    void setCity(string c);
    void setPCode(string p);
    string getStName()const;
    int getStNr()const;
    string getCity()const;
    string getPCode()const;
private:
    string streetName;
    int streetNr;
    string city;
    string postalCode;
};

//implementation
Address::Address()          //constructor
{
    streetName = " ";
    streetNr = 0;
    city = " ";
    postalCode = "0000";
}

//Mutators to change (set) the value of each member variable

void Address::setStName(string s)
{
    streetName = s;
}

void Address::setStNr(int n)
{
    streetNr = n;
}

void Address::setCity(string c)
{
    city = c;
}

void Address::setPCode(string p)
{
    postalCode = p;
}

//Accessors to retrieve (get) data from each of member variables

string Address::getStName() const
{
    return streetName;
}

int Address::getStNr() const
{
    return streetNr;
}

string Address::getCity() const
{
    return city;
}

string Address::getPCode() const
{
    return postalCode;
}

//main application
int main()
{
    Address myAddress;
    string sName;
    int sNr;
    string cCity;
    string pCode;

    cout << "Please enter your address:" << endl;
    cout << "Street name: ";
    getline(cin, sName, '\n');
    myAddress.setStName(sName);
    cout << "Street number: ";
    cin >> sNr;
    myAddress.setStNr(sNr);
    cout << "City: ";
    cin >> cCity;
    myAddress.setCity(cCity);
    cout << "Postal code: ";
    cin >> pCode;
    myAddress.setPCode(pCode);
    cout << endl << "My address is: " << endl;
    cout << myAddress.getStNr() << " " << myAddress.getStName() << endl;
    cout << myAddress.getCity() << endl;
    cout << myAddress.getPCode() << endl;
    return 0;
}
```

Input and Output:

```
Please enter your address:
Street name: Hector Peterson St
Street number: 543
City: Soweto
Postal code: 0192

My address is:
543 Hector Peterson St
Soweto
0192
Press any key to continue . . .
```

Question 2 [max of 30 marks]

(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.

(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 and the same ADT class can be used in any number of different programs.

(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.

(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 [This question was not marked, and *no* marks were allocated for attempting it]

3 (a) For this question, you were given the following class declaration:

```
class Employee
{
public:
   Employee ();
   string getName();
private:
   bool citizen();
   string lastName;
   string firstName;
   string employeeNumber;
};
```

and had to explain what is wrong with the following code fragment:

```
int main()
{
      Employee e;
      ……
      string eNumber = e.employeeNumber;
      return 0;
}
```

From the class declaration, it can be seen that the member variable `employeeNumber` is private. Therefore only member functions of class `Employee` have direct access to member variable `employeeNumber` of object `e`. To allow access from outside to the private member variables of object `e`, we need to define an accessor function for `employeeNumber`, i.e. change the class definition as shown in bold:

```
class Employee
{
public:
   Employee ();
   string getName();
   string getEmployeeNumber();
private:
   bool citizen();
   string lastName;
   string firstName;
   string employeeNumber;
};
```

and then use it as shown below:

```
int main()
{
      Employee e;
      ……
      string eNumber = e.getEmployeeNumber();
```

```
            return 0;
      }
```

3 (b) For this question, you were given the following class declaration:

```
      class Employee
      {
         Employee ();
         string getName();
      private:
         bool citizen();
         string lastName;
         string firstName;
         string employeeNumber;
      };
```

and you had to indicate whether the access to the accessor `getName()` is public or private.

In this instance access to accessor `getName()` is private since the access is not specified. If access is not specified, it is private by default.

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

There are many acceptable solutions to this program. We designed our `Module` class to contain information on the module code, the module name, the marks for two assignments, the year mark and the final mark.

Our class should be able to do the following:

- Retrieve the module code
- Retrieve the module name
- Retrieve the marks for two assignments
- Retrieve the year mark
- Retrieve the final mark
- Set the module code
- Set the module name
- Set the marks for two assignments
- Set the year mark
- Set the final mark
- Determine the year mark

Program listing:

```
#include <iostream>
#include <cstdlib>

using namespace std;

//Define the class
class Module
{
public:
    Module();
    Module(string pCode, string pName, int pMarkAsg1, int pMarkAsg2,
           int pYearMark,
           int pFinalMark);
    ~Module();
    string getCode() const;
    string getName() const;
    int getMarkAsg1() const;
    int getMarkAsg2() const;
    int getYearMark() const;
    int getFinalMark() const;
    void setCode(string pCode);
    void setName(string pName);
    void setMarkAsg1(int pMarkAsg1);
    void setMarkAsg2(int pMarkAsg2);
    void setYearMark(int pYearMark);
    void setFinalMark(int pFinalMark);
    int calcYearMark();
private:
    string code;
    string name;
    int markAsg1;
    int markAsg2;
    int yearMark;
    int finalMark;
};

//The implementation

Module::Module( ) //default constructor
{
    code = " ";
    name= " ";
    markAsg1 = 0;
    markAsg2 = 0;
    yearMark = 0;
    finalMark = 0;
}

Module::Module(string pCode, string pName, int pMarkAsg1, int pMarkAsg2,
               int pYearMark, int pFinalMark) //overloaded constructor
{
    code = pCode;
    name= pName;
    markAsg1 = pMarkAsg1;
    markAsg2 = pMarkAsg2;
    yearMark = pYearMark;
    finalMark = pFinalMark;
    }

Module::~Module()    //destructor
{ }

string Module::getCode() const
{
    return code;
}

string Module::getName() const  //accessor
{
    return name;
}
```

```
int Module::getMarkAsg1() const //accessor
{
    return markAsg1;
}

int Module::getMarkAsg2() const //accessor
{
    return markAsg2;
}

int Module::getYearMark() const //accessor
{
    return yearMark;
}

int Module::getFinalMark() const    //accessor
{
    return finalMark;
}

void Module::setCode(string pCode)  //mutator
{
    code = pCode;
}

void Module::setName(string pName)  //mutator
{
   name= pName;
}

void Module::setMarkAsg1(int pMarkAsg1) //mutator
{
   markAsg1 = pMarkAsg1;
}

void Module::setMarkAsg2(int pMarkAsg2) //mutator
{
   markAsg2 = pMarkAsg2;
}

void Module::setYearMark(int pYearMark) //mutator
{
   yearMark = pYearMark;
}

void Module::setFinalMark(int pFinalMark)   //mutator
{
   finalMark = pFinalMark;
}

int Module::calcYearMark()
{
   return (int(markAsg1 * 0.3 + markAsg2 * 0.7));
}

//Application program

const int nrOfModules = 2;
```

```
int main()
{
  Module myModules[nrOfModules];
  string c;
  string n;
  int mAsg1;
  int mAsg2;
  int yrMark;
  int fnlMark;

  for (int i = 0; i < nrOfModules; i++)
  {
      cout << "Enter module code: ";
      cin >> c;
      cout << "Enter module name: ";
      cin >> n;
      cout << "Enter assignment 1 mark: ";
      cin >> mAsg1;
      cout << "Enter assignment 2 mark: ";
      cin >> mAsg2;
      myModules[i] = Module(c, n, mAsg1, mAsg2, 0, 0);
      myModules[i].setYearMark(myModules[i].calcYearMark());
      cout << "Year mark for " << myModules[i].getName() << " is "
           << myModules[i].getYearMark() << endl << endl;
   }
//We assume details for assignments for COS1512 is stored in myModules[1]
   myModules[1].setMarkAsg2( int(myModules[1].getMarkAsg2() *1.05));
   myModules[1].setYearMark(myModules[1].calcYearMark());
   cout << "After updating with 5%, year mark for "
        << myModules[1].getName() << " is "
        << myModules[1].getYearMark() << endl << endl;

   //Display year marks for all modules
   cout << "All year marks:" << endl;
   for (int i = 0; i < nrOfModules; i++)
   {
      cout << "Year mark for " << myModules[i].getName() << " is "
           << myModules[i].getYearMark() << endl;
   }

   return 0;
}
```

Output:

```
Enter module code: COS1511
Enter module name: ProgrammingI
Enter assignment 1 mark: 60
Enter assignment 2 mark: 80
Year mark for ProgrammingI is 73

Enter module code: COS1512
Enter module name: ProgrammingII
Enter assignment 1 mark: 75
Enter assignment 2 mark: 80
Year mark for ProgrammingII is 78

After updating with 5%, year mark for ProgrammingII is 81
```

```
All year marks:
Year mark for ProgrammingI is 73
Year mark for ProgrammingII is 81
Press any key to continue . . .
```

Discussion:
There are various solutions to this question. The concepts of declaring a class of objects may be new to you, and you may have struggled to implement it. We hope that after studying our solution you will understand it, and be able to apply it.

- If a class has constructors and you declare an array of class objects, the class must have the default constructor. The default constructor is used to initialise each (`array`) class object.
- The declaration statement
  `Module myModules [nrOfModules]`
  declares an array named `myModules` that contains `nrOfModules` elements that each is of type `Module`.
- We refer to a specific member function of a specific element in the array by using the dot operator "`.`".

Question 5 [max of 15 marks]

Discussion:
For this question, you had to define a class `Team` as an ADT with member variables `country`, `round`, `points`, `goalsFor` and `goalsAgainst`. The class contains

- a default constructor,
- an overloaded constructor,
- a destructor,
- accessor and mutator functions for each of the member variables,
- a void member function `reset();`
- two member functions `calcGoalDifference()` and `update()`
- and overloaded friend functions for the operators ==, >, ++ as well as for the stream extraction operator >> and the stream insertion operator <<.

Consider operators ==, > and ++ overloaded as friend functions of the class `Team`:

```
        friend bool operator>(const Team &Team1, const Team &Team2);
        friend bool operator==(const Team &Team1, const Team &Team2);
```

Note, the **`const`** keyword is used to guarantee that these functions will not modify the `Team` objects. When we compare two `Team` objects with == or >, we do not want to change or modify the objects we compare. The same situation holds if we would define + and - operators for class `Team`. 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 each prototype as the function is not a member function.

The relational friend functions == and > return boolean values. `Operator==` returns `true` when the `points` member variables and the goal difference of the two teams are the same. `Operator>` returns `true` when the `points` member variable of `Team1` is bigger than that of `Team2`; or if the `points` member variable of `Team1` is equal to that of `Team2` and the goal difference of `Team1` is bigger than that of `Team2`.

The prefix `operator++` is a unary operator, which means that it takes only one operand, in contrast to binary operators such as + or – which use two operands. Therefore, when overloading the prefix `operator++` as a friend function, only one parameter is specified, which represents the object that we want to increment:

```
        friend Team operator++(Team &T);
```

Since the object will be modified in the `friend` function, it must be a reference parameter. In this implementation of the prefix operator++, we increment the `round` member variable of the object:

```
      Team operator++(Team &T)
      {
          ++T.round;
          return T;
      }
```

See also the discussion on Question 2(h) in this tutorial letter.

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

```
      friend ostream & operator<<(ostream & out, const Team & T);
      friend istream & operator>>(istream & ins,  Team & T);
```

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,  Team & T)
{
      ins >> T.country;
      ins >> T.round;
      ins >> T.points;
      ins >> T.goalsFor;
      ins >> T.goalsAgainst;
      return ins;
}
```

and the way it is used in the application:

```
    Team opponent, newOpponent;
    cout << "Enter data for new opponent (country, round, points, goals "
         << "for and goals against): ";
    cin >> newOpponent;
```

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

```
    Team opponent, newOpponent;
    Team home("South-Africa", 1, 4, 6, 4);
    opponent.reset("Germany", 1, 4, 6, 4);

    cout << "The home team is: " << endl << home << endl;
    cout << "The opponent team is " << endl << opponent << endl;
```

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

Program listing:
Team.h class definition / interface for class Team:

```
#ifndef TEAM_H
#define TEAM_H
#include <iostream>
#include <string>
using namespace std;
class Team
{
    public:
        Team();
        Team(string c, int r, int p, int gFor, int gAgainst);
        ~Team();
        string get_country() const;
        int get_round() const;
        int get_points()const;
        int get_goals_for()const;
        int get_goals_against()const;
        void set_country(string c);
        void set_round(int r);
        void set_points(int p);
        void set_goals_for(int gFor);
        void set_goals_against(int gAgainst);
        void reset(string c, int r, int p, int gFor, int gAgainst);
        int goalDifference()const;
        void update(int p, int gFor, int gAgainst);
        friend Team operator++(Team &T);
        friend bool operator>(const Team &Team1, const Team &Team2);
        friend bool operator==(const Team &Team1, const Team &Team2);
        friend istream & operator>>(istream & ins, Team & T);
        friend ostream & operator<<(ostream & out, const Team & T);

    private:
        string country;// the name of the country for which this team plays
        int round;         // the round in which the team currently plays
        int points;   // the points the team has accumulated
        int goalsFor; // the goals the team has scored
        int goalsAgainst;  // the goals scored against the team

};
#endif
```

Implementation file: Team.cpp

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

Team::Team()
{
    country = "Country 0";
    round = 0;
    points = 0;
    goalsFor = 0;
    goalsAgainst = 0;
}

Team::Team(string c, int r, int p, int gFor, int gAgainst)
```

```
{
    country = c;
    round = r;
    points = p;
    goalsFor = gFor;
    goalsAgainst = gAgainst;
}
Team::~Team()
{
    cout<<"Game Over!" << endl;
}

string Team::get_country() const
{
    return country;
}

int Team::get_round() const
{
    return round;
}

int Team::get_points()const
{
    return points;
}

int Team::get_goals_for()const
{
    return goalsFor;
}

int Team::get_goals_against()const
{
    return goalsAgainst;
}
void Team::set_country(string c)
{
    country = c;
}

void Team::set_round(int r)
{
    round = r;
}

void Team::set_points(int p)
{
    points = p;
}

void Team::set_goals_for(int gFor)
{
    goalsFor = gFor;
}

void Team::set_goals_against(int gAgainst)
{
    goalsAgainst = gAgainst;
```

```
}
void Team::reset(string c, int r, int p, int gFor, int gAgainst)
{
    country = c;
    round = r;
    points = p;
    goalsFor = gFor;
    goalsAgainst = gAgainst;
}

int Team::goalDifference()const
{
    return (goalsFor - goalsAgainst);
}

void Team::update(int p, int gFor, int gAgainst)
{
    points = points + p;
    goalsFor = goalsFor + gFor;
    goalsAgainst = goalsAgainst + gAgainst;
}

Team operator++(Team &T)
{
    ++T.round;
    return T;
}

bool operator>(const Team &Team1, const Team &Team2)
{
    if ((Team1.points > Team2.points) ||
        ((Team1.points == Team2.points) &&
         (Team1.goalDifference() > Team2.goalDifference())))
        return true;
    return false;
}

bool operator==(const Team &Team1, const Team &Team2)
{
    if ((Team1.points == Team2.points) &&
        (Team1.goalDifference() == Team2.goalDifference()))
        return true;
    return false;
}

istream & operator>>(istream & ins,  Team & T)
{
      ins >> T.country;
      ins >> T.round;
      ins >> T.points;
      ins >> T.goalsFor;
      ins >> T.goalsAgainst;
      return ins;
}

ostream & operator<<(ostream & out, const Team & T)
{
      cout << "Country : " << T.country << endl;
      cout << "round : " << T.round << endl;
      cout << "points : " << T.points << endl;
      cout << "goals for: " << T.goalsFor << endl;
      cout << "goals against: " << T.goalsAgainst << endl;
      return out;
}
```

Application file - TestTeam.cpp:

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

int main()
{
    Team opponent, newOpponent;
    Team home("South-Africa", 1, 4, 6, 4);
    opponent.reset("Germany", 1, 4, 6, 4);

    cout << "The home team is: " << endl << home << endl;
    cout << "The opponent team is " << endl << opponent << endl;

    if (home == opponent)
        cout << "This is a tie!" << endl;
    else if (home > opponent)
            ++home;
         else ++opponent;

    cout << home.get_country() << " has " << home.get_points()
         << " points and " << opponent.get_country() << " has "
         << opponent.get_points() << " points" << endl;
    cout << home.get_country() << " is in round " << home.get_round()
         << " and " << opponent.get_country() << " is in round "
         << opponent.get_round() << endl << endl;

    cout << "Enter data for new opponent (country, round, points, goals "
         << ”for and goals against): ";
    cin >> newOpponent;
    cout << endl;

    int SApoints, Spoints, goalsForSA, goalsForS,goalsAgainstSA;
    int goalsAgainstS;
    cout << "Enter points earned for this match by South-Africa: ";
    cin >> SApoints;
    cout << "Enter goals for SA: ";
    cin >> goalsForSA;
    cout << "Enter goals against SA: ";
    cin >> goalsAgainstSA;
    cout << "Enter points earned for this match by Spain: ";
    cin >> Spoints;
    cout << "Enter goals for Spain: ";
    cin >> goalsForS;
    cout << "Enter goals against Spain: ";
    cin >> goalsAgainstS;
    home.update(SApoints,goalsForSA,goalsAgainstSA);
    opponent.update(Spoints,goalsForS,goalsAgainstS);

    if (home == newOpponent)
```

```
        cout << "This is a tie!" << endl;
    else if (home > newOpponent)
            ++home;
         else ++newOpponent;

    cout << home.get_country() << " has " << home.get_points()
         << " points and " << newOpponent.get_country() << " has "
         << newOpponent.get_points() << " points" << endl;
    cout << home.get_country() << " is in round " << home.get_round()
         << " and " << newOpponent.get_country() << " is in round "
         << newOpponent.get_round() << endl << endl;

  return 0;
}
```

Output:

```
The home team is:
Country : South-Africa
round : 1
points : 4
goals for: 6
goals against: 4

The opponent team is
Country : Germany
round : 1
points : 4
goals for: 6
goals against: 4

This is a tie!
South-Africa has 4 points and Germany has 4 points
South-Africa is in round 1 and Germany is in round 1

Enter data for new opponent (country, round, points, goals for and goals
aga
): Spain 1 7 8 2

Enter points earned for this match by South-Africa: 3
Enter goals for SA: 2
Enter goals against SA: 0
Enter points earned for this match by Spain: 0
Enter goals for Spain: 0
Enter goals against Spain: 2
Game Over!
South-Africa has 7 points and Spain has 7 points
South-Africa is in round 1 and Spain is in round 2

Game Over!
Game Over!
Game Over!
Press any key to continue . . .
```

Enrichment exercise

For this question, you had to overload the ++, > and == operators of the class **`Team`**, as member functions. The same application file as in Question 4 should be used, and it should produce the same output.

Consider the new prototypes:

```
        Team operator++();
        bool operator>(const Team &Team2)const;
        bool operator==(const Team &Team2)const;
```

Note that the **operator++()** member function has no parameters and that the **operator>()** and **operator==()** member functions each has only one parameter passed to it, instead of two parameters as for a friend function. For instance `operator==` determines whether the parameter passed and the calling object have the same `points` member variable, and the same value for the goal difference. For example, consider the statement: `if (x == y) cout << "Equal";` where `x` would represent the calling object and `y` the argument being passed to member function **operator==()** With the definition of `operator==` as a member function, we have to choose one of the objects to be the calling object. In contrast, with `operator==` as a non-member `friend` function (as in question 4), we do not need to do so, with the result that both parameters are treated equally.

Note also that for member functions **operator>()** and **operator==()** the parameters are declared as constant references to guarantee that these parameters are not modified by the calling member function (see pages 616 – 617 of Savitch 6th edition / pages 660 -661 of Savitch 7th edition/ page421 of Savitch 8th edition) and the `const` at end of the prototype is a guarantee that the calling object itself will not be modified. For example, the statement below should not modify either x, or y in any possible way.

```
                        if (x == y) cout << "Equal";
```

The implementation of the member function **operator==()** has a type qualifier (see page 543, Savitch 6th edition / page 575 7th edition for a definition/ page 571 8th edition for a definition). In other words we had to insert `Team::`in the implementation. However the implementation of `operator==()` in question 4 has no type qualifier, since the operator is not a member function but rather a `friend` function.

```
bool Team::operator==(const Team &Team2)const
{
    if ((this->points == Team2.points) &&
         (this->goalDifference() == Team2.goalDifference()))
        return true;
    return false;
}
```

Here `this->points` represent the member variable of the calling object, while `Team2.points` refer to the member variable of the object being passed as a parameter.

We use the `this` pointer to refer to the calling object (`*this`). See Appendix 7 page 990 Savich 6th edition / page 1044 Savitch 7th edition / page 1031 Savitch 8th edition and chapter 11 of Tutorial Letter 201 (Study Guide), section 11.5 for more on the `*this`.

Many students find this version of **operator==()** rather confusing because it has only one parameter. For the purposes of this module, we prefer defining operators as `friend` functions.

Reference for member functions: Appendix 8 page 993 Savich 6th edition / page 1047 Savitch 7th edition/ page 1034 Savitch 8th edition and chapter 11 of Tutorial Letter 201 (Study Guide), section 11.4.

Changes to the interface and implementation of the class `Team` to overload the ++, > and == operators as member functions, are shown in bold.

Changes to class interface Team.h shown in bold:

```
#ifndef TEAM_H
#define TEAM_H
```

```
#include <iostream>
#include <string>
using namespace std;
class Team
{
    public:
        Team();
        Team(string c, int r, int p, int gFor, int gAgainst);
        ~Team();
        string get_country() const;
        int get_round() const;
        int get_points()const;
        int get_goals_for()const;
        int get_goals_against()const;
        void set_country(string c);
        void set_round(int r);
        void set_points(int p);
        void set_goals_for(int gFor);
        void set_goals_against(int gAgainst);
        void reset(string c, int r, int p, int gFor, int gAgainst);
        int goalDifference()const;
        void update(int p, int gFor, int gAgainst);
        Team operator++();
        bool operator>(const Team &Team2)const;
        bool operator==(const Team &Team2)const;
        friend istream & operator>>(istream & ins, Team & T);
        friend ostream & operator<<(ostream & out, const Team & T);

    private:
        string country;    // the name of the country for which this
                           //team plays
        int round;         // the round in which the team currently
                           //plays
        int points;        // the points the team has accumulated
        int goalsFor;      // the goals the team has scored
        int goalsAgainst;  // the goals scored against the team

};
#endif
```

Changes to implementation file Team.cpp:

```
Team Team:: operator++()
{
    ++this->round;
    return *this;
}

bool Team::operator>(const Team &Team2)const
{
    if ((this->points > Team2.points) ||
        ((this->points == Team2.points) &&
          (this->goalDifference() > Team2.goalDifference())))
        return true;
    return false;
}

bool Team::operator==(const Team &Team2)const
```

```
{
    if ((this->points == Team2.points) &&
        (this->goalDifference() == Team2.goalDifference()))
        return true;
    return false;
}
```

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

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 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 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 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.

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.
```