# Acknowledgement

FPC has been an extremely challenging, yet rewarding, module that I have followed this semester. The project given to us was both a test in newfound programming skills while also encouraging us to delve into Object Oriented Programming. It is with great pleasure and immense relief that I wish to thank the individuals who helped me throughout the project especially in understanding OOP.

I would first like to thank my lecturer, Ms. Oshini Jayatunge whose assistance has always been there when I asked for it. I thank her for not only giving up so much time for my queries but also for the effort she put into making this module understandable to all of us.

I would also like to thank Mr. Gamindu, Mr. Dilhan and Ms. Zeenath for their valuable input on Object Oriented Programming without which I may never have fully understood the basics of OOP.

I wish to thank my parents for having given me everything I could ask for to complete my education. I thank APIIT Lanka as well for providing us with the opportunity to gain this knowledge.

Finally I wish to thank God for all the blessings given without which this project would not have been possible.

# 

# List of figures

[Figure 1 5](#_Toc241243915)

[Figure 2 5](#_Toc241243916)

[Figure 3 6](#_Toc241243917)

[Figure 4 6](#_Toc241243918)

[Figure 5 6](#_Toc241243919)

# Introduction

The project given to us is to deliver a fully functional students’ grade tracking system using C++ to develop a win32 console application. The application is divided into two main parts. The first being the login area and the second being the manipulation of records. As specified the application should allow a user to log in and based on that login should identify the user’s privileges and permissions. The functions are divided into four main categories, Administrator, Administrative staff, Lecturers and Students.

The system requires that data be stored so that it may be retrieved at each run. The given scenario has specifications as to which functions are allowed for which user.

In order to complete this project it is a requirement that both OOP and C++ knowledge must be utilized. Therefore basic UML Class diagrams are required before actual development begins.

The following report is a summary of all my findings which include justifications for the final program.

# 2.0 Description and justification of the design of implementation codes

## 2.1 Implementation of Controller Classes

This concept is what separates the equivalent of a GUI in this project into several completely independent classes that handle all data collection and other tasks such as the output of menus. There are five different classes that fall under this category which have been divided according to their functionality. They have been made to handle the adding, modifying, deleting, viewing and searching of records. Each class has several methods which are accessed according to the type of user. Initially I planned to divide them based on what type of object they would be handling (lecturer, student, etc). However I opted against this since several of the functions would be accessed by different types of users. Functions such as those in the search module are accessed by lecturers, administrative staff and administrators alike. Therefore it made more sense to have them grouped by functionality rather than usage since there was no common pattern that could be identified. In the diagram however I have not represented the full associations as this would complicate it thus making it lose its effectiveness. Ideally the student, lecturer, module and administrative staff classes should be connected to all the controller classes since instances of each object exist within them. The class diagram is given below.

## 2.2 Implementing user class as abstract

The purpose of creating a single class as an abstract class was so that users could inherit the common properties from the parent class. The classes that inherit are the lecturer, administrative staff and the student class. The methods in user class are also common to the above mentioned classes. By inheriting a great deal of time is saved both in design as well as in programming. When adding similar classes this would prove to be invaluable since the real lines of code written are all stored inside one class.

## 2.3 Implementation of student reports

At first although I had kept in mind the possibility of having an association class that would combine the module with the student I felt that this would be unnecessary since having the student report as part of the student details felt appropriate. Therefore I implemented the student reports by associating the module class with it and simply adding an extra attribute called marks in the student class. Since all the reports class would have held would have been student id, module id and mark I felt that this alternative was sufficient to complete the same task.

## 2.4 Non implementation of Admin class

During development of the system I revised the design to not include an administrator class. This was because I did not feel it necessary to keep an administrator’s details within the system since their role is only within the scope of the implemented system and not within the actually problem scope. Therefore administrators are recognized only by their logins. Furthermore I had initially planned to implement all methods relating to the system within the Admin class. However I realized that this was neither practical nor correct from an OOP point of view (since methods of a class from which objects will be instantiated should ideally be written only to access that class’s private variables). This was the reason for switching all functions to one main control class which eventually became categorized into several sub sections.

# 3.0 Description and justification of the implementation codes in terms of object oriented programming concepts

In the following section I have described the structure that I have used to implement as well as reasons for implementing code in the manner I have chosen.

## 3.1 Implementation of User class

The code implemented here is such that all variables in the User class, which is an abstract class, are inherited into the classes derived from it. The attributes have been specified as protected while the inheritance for each class has been done so publicly. The reason for not specifying these attributes as private is due to the fact that C++ does not inherit private members. Therefore the abstract class would in this case, serve no purpose. However instead of making them public I have given them the protected property since making them public creates issues pertaining to security. However protected suits the purpose perfectly since it is inherited while retaining the general properties of a private variable.

class User{

protected:

long id;

string name;

Date dob;

char gender;

string NIC;

Figure 3.1

#include "User.h"

class Admin:public User{  
  
#include "User.h"

class Lecturer:public User{

Figure 3.2

In order to make the User class abstract it is required that at least one method be pure virtual. Since it posed no logical errors and maintained consistency, all methods of the User class (excluding the constructors) were made pure virtual. These methods were implemented in the User class and called on in the derived classes

virtual void setID(long)=0;

virtual void setName(string)=0;

virtual void setDate(Date)=0;

Figure 3.3

void Lecturer::setID(long ID){

User::setID(ID);

}

void Lecturer::setName(string n){

User::setName(n);

}

void Lecturer::setDate(Date d){

User::setDate(d);

}

Figure 3.4

## 3.2 Implementation of Student Class

The student class requires a brief explanation as its implementation differs slightly from the other classes. This class, unlike the others, required several attributes and methods apart from its parent class. In order to implement the constructor the already overloaded User constructor was overridden to give the necessary result.

protected:

string Batch;

int marks[6];

Date dateJ;

string Nationality;

Module mod[3];

//constructor  
Student::Student(long ID, string n, Date d, char g, string nic,string bc,Date dj,string na):User(ID,n,d,g,nic),Batch(bc),dateJ(dj),Nationality(na) {

}

Figure 3.5

## 3.3 Implementation of File Controller class

This class was implemented as a template class. The reason for this being that it would be much more efficient to enable the file handling commands at a very common level. This means that regardless of the object or file we are handling we would still be able to use the same functions for a similar action. Given below is the declaration of the class.

template <typename objType>

class FileControl{

Figure 3.6

Following is an example of the function written to add a record to the file.

template <typename objType>

void FileControl<objType>::add(const objType &addObj,string fileName){

fstream addFile(fileName.c\_str(),ios::in|ios::out|ios::ate);

addFile.write((char\*)&addObj,sizeof(objType));

addFile.close();

}

Figure 3.7

This function takes in two parameters. The first being the object of any type and the second being the file name. These two parameters are in fact the only variables in file handling at this level. Therefore we can compare two calls to this function,

FileControl<Login> newFC;

newFC.add(temp,fName);

FileControl<Student> newFC;

newFC.add(temp,fName);

Figure 3.8

Thus the only change made is to declare which object is being used while also initializing fName to the appropriate file.

# 4.0 Description and justification of the validation codes applied into the implementation codes

## 4.1 Validation codes for integer input

The code for validation of integers is a standard piece of code which uses the limits class to help clear any failed input streams. The code given below has been taken from the validation used for the login control. The user must enter the user level which is of an integer type. At this point validation must be performed to prevent strings, characters or any other undesired input from being given.

do{

check=0;

cout<<"User Access Level: ";

cin>>level;

if (cin.fail()){

check=1;

cout<<"Wrong input!!! Please enter a number"<<endl<<endl;

cin.clear();

cin.ignore (std::numeric\_limits<int>::max(),'\n');

}

}while(check==1);

Figure 4.1

This prompts the user for the input while “check” which is a flag to check for errors (1 if there is an error, 0 if the user has given a correct input). cin.fail()is a flag that is set by C++ during runtime based on whether or not the input stream matches the location it is to be written to. cin.clear() resets this status after it has been identified that there is an error. cin.ignore (std::numeric\_limits<int>::max(),'\n') is the code that handles the clearing of the stream. It ignores the bits specified in the first parameter until it comes to the termination character specified in the second parameter. This is very much the equivalent of the function fflush(stdin) used in C.

## 4.2 Validation code used for character input

This code is very similar to that used above. Should the user give an incorrect input it is filtered and the “bad” bits are ignored until a new line character is given. However the issue here would be that ignoring an input of an integer would not be enough since integers could also be recognized as characters. Therefore there was an additional check to identify incorrect characters even if the bitstream did not contain any visible errors.

do{

check=0;

cout<<"Student gender(m/f): ";

cin.ignore(std::numeric\_limits<streamsize>::max(),'\n');

cin>>g;

if((g!='m')&&(g!='M')&&(g!='f')&&(g!='F')){

check=1;

cout<<"Please enter either m or M or f or F for gender (m=Male,f=Female)"<<endl;

cin.clear();

}

if(cin.fail()){

check=1;

cout<<"Please enter either m or M or f or F for gender (m=Male,f=Female)"<<endl;

cin.clear();

}

}while(check==1);

Figure 4.2

This code however is not entirely perfect and I have not yet identified a solution to the error. Should the user give the input “mx” where x can be any character, it fails to catch the error.

## 4.3 Validation code for NIC input

The validation code written for the NIC is a very simple piece of code. The logic behind it is that it takes in the user input and then checks its length to see if it matches the required 10 characters of the NIC. If it does not it simply loops in order to repeat the question and take in an input until a valid input is given. The code for this is given below.

do{

cout<<"Student NIC no: ";

cin>>nic;

if(nic.length()!=10)

cout<<"NIC number can not be lesser or greater than 10 characters!!"<<endl;

}while(nic.length()!=10);

Figure 4.3

The key words here are “while(nic.length()!=10);”.

# 5.0 Description and discussions of the testing plan and result

## 5.1 Test Plan for Login

| **Test Case** | **Test Description** | **Expected Result** | **Actual Output** |
| --- | --- | --- | --- |
| 1(a) | Login checks for valid input | Print login successful if valid. Print error message if not valid. Shut down system after 5 attempts. | Worked as expected |
| 1(b) | Login validates input on user level | If a character or string or any other type is entered instead of an integer the program throws an error and requests the user for a valid input | Worked as expected |
| 1(c) | Login masks user input on password | Characters are masked with “\*” symbol. | Worked as expected |
| 1(d) | Login redirects user based on level | Administators (level 1) are directed to administrator page. Similarly lecturers (level 2) are directed to the lecturer page | Worked as expected |

## 5.2 Test plan for Administrator functions

| **Test Case** | **Test Description** | **Expected Result** | **Actual Output** |
| --- | --- | --- | --- |
| 2(a) | Administrator menu redirection | Administrator main menu redirects to each submenu as specified | Worked as expected |
| 2(b) | Administrator add functions | Administrator can add records to each file and these records are appended to the file | Worked as expected |
| 2(c) | Administrator modify functions | User may specify which record is to be modified and modify all records | Worked as expected |
| 2(d) | Administrator delete function | User specifies which record is to be deleted and confirms deletion | Worked as expected |
| 2(e) | Administrator view functions | All records corresponding to menu are displayed upon choice selection | Worked as expected |
| 2(f) | Administrator search functions | Except for ID (where a precise match is looked for) all other fields are searched through for strings that match the user input. User may reject the choices offered until end of file | Did not function as expected. Case matching was an issue. |

### 5.2.1 Further comments

While testing the latter section of the search function it was found that case sensitivity was an issue. Therefore typing “student name” or “Student Name” yielded unexpected results.

As all other functions were derivations of the administrator’s functions further tests were done quickly and in the same manner, thus giving similar results to that of the administrator class. However further testing was required for the lecturer and student functions. Details of testing are given below.

## 5.3 Test plans for Lecturer functions

| **Test Case** | **Test Description** | **Expected Result** | **Actual Output** |
| --- | --- | --- | --- |
| 3(a) | Lecturer is recognized based on login | Depending on the lecturer’s login details, their personal details are automatically identified | Worked as expected |
| 3(b | Lecturer modifying and deleting student marks | Only the modules that the lecturer takes can be modified in anyway | Worked as expected |

## 5.4 Test plans for Student functions

| **Test Case** | **Test Description** | **Expected Result** | **Actual Output** |
| --- | --- | --- | --- |
| 4(a) | Student views result based on login details | Student ID is automatically recognized based on his/her login | Worked as expected |

## 5.5 Test plans for validation

| **Test Case** | **Test Description** | **Expected Result** | **Actual Output** |
| --- | --- | --- | --- |
| 5(a) | Entering a character in place of an integer | The program should prompt user for a correct input | Worked as expected |
| 5(b) | Enter a string or integer instead of a character | User should be prompted for the correct input | This works in all situations except that which the user has entered one of the required characters as the first character while adding some more characters to the list. |
| 5(c) | NIC number is checked for length | User is prompted to enter 10 characters for the NIC | Worked as expected |
| 5(d) | Date is checked for day month and year | User should be prompted if he/she enters an impossible value | Not handled due to months with different number of days. However validation for a number value is performed. |

# 6.0 Conclusion

I have to the best of my knowledge implemented the basic functions of this system to work without errors should the system be used correctly. However in the case of incorrect inputs and validation requirements I feel that my system falls slightly short of what I would have ideally wanted. Towards the end of the project I realized that I should have kept a class for validation control only. Unfortunately due to time constraints this was not possible. Despite this I have provided validation for each possible case in at least one place within the program in order to demonstrate that I have researched sufficiently on the programming language in order to handle any errors.

In terms of automation I spent time on automating the lecturers’ and students’ sections of the application so that upon login, no further input would be required from the user in order to identify his/her identity. By this I gave the ability to the student to simply choose view report card upon logging in which would instantly display the record corresponding to their profile. This was just as useful for implementing the lecturer marking section where the lecturer had to be given rights only to modules they were in charge of.

However there are limitations in this respect which given more time I would have been able to tackle. If a lecturer is deleted from the system that lecturer’s modules would still contain their name. I have not given the option to delete this, only to replace as it made sense from a business logic point of view. However since the project did not require us to justify our assumptions in business logic I have stated this fact only here.

Given more time, the one major automation that I would have wished to introduce would be record duplication checks through NIC numbers as well as record input automation through the same data. Using the NIC number I would have wanted to automate the date of birth input and gender input.

In the aspects of the looks of the program I have not paid very much attention as it is not a requirement of the project. In all other aspects of the program I have given ease of use priority, especially in search where the user will search based on several criteria and a preview of each matching piece of data will be given for the user to either view or ignore.

# APPENDICES

# Appendix A – References

Deitel, H.M. 2005*, C++ How to Program*, Prentice Hall

Fowler, M. *UML Distilled*, Addison Wesley

*C++ Reference Website*, [Online], Available: <http://www.cppreference.com>

# Appendix B – Class Diagrams