

DCU School of Electronic Engineering Assignment Submission

Student Name(s): Michael Lenehan
Student Number(s): 15410402
Programme: B.Eng in Electronic and Computer Engineering
Module Code: EE402
Lecturer: D. Molloy
Project Due Date: 5/11/2018

Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying is a grave and serious offence in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion, or copying. I have read and understood the Assignment Regulations set out in the module documentation. I have identified and included the source of all facts, ideas, opinions, viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references.

I have not copied or paraphrased an extract of any length from any source without identifying the source and using quotation marks as appropriate. Any images, audio recordings, video or other materials have likewise been originated and produced by me or are fully acknowledged and identified.

This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study. I have read and understood the referencing guidelines found at <http://www.library.dcu.ie/citing&refguide08.pdf> and/or recommended in the assignment guidelines.

I understand that I may be required to discuss with the module lecturer/s the contents of this submission.

I/me/my incorporates we/us/our in the case of group work, which is signed by all of us.

Signed: Michael Lenehan

Assignment 1

Michael Lenehan

Introduction

The purpose of this assignment is to demonstrate a knowledge of the workings of C++. The application offers a number of classes, used to create “Instrument” based objects. Each of these classes adds further information to its base class. The following table lists the each of these classes, and their associated parent class.

Table 1: Description of Classes and Variable Information

| Class Name | Parent Class |
|-----------------------|----------------|
| Instrument | NA |
| Guitar | Instrument |
| AcousticGuitar | Instrument |
| ElectroAcousticGuitar | ElectricGuitar |
| | AcousticGuitar |
| Piano | Instrument |
| AcousticPiano | Piano |

Implementation

The following sections describe the implementation of the twenty points of the assignment.

2.1 Correct Use of all Access Specifiers

The use of the three C++ access specifiers - private, protected, and public - can be seen within the “Guitar.h” class declaration. This declaration is structured as follows:

```
class Guitar: public Instrument{  
private:
```

```
    static int nextGuitarSerialNumber;
    int serialNumber;
    void construct();
protected:
    string bodyShape;
    int noOfStrings;
public:
    Guitar(int, string, float, string, int);
    Guitar(string, float, string, int);
    virtual void display() =0;
    virtual ~Guitar();
};
```

Within this class header file, the integer and static integer variables *serialNumber*, and *nextSerialNumber*, and the void method *construct()* are declared as private. These pieces of information are therefore not accessible outside this class, as required.

The string variable *bodyShape* and integer variable *noOfStrings* are declared as protected, as they are required to be modifiable by any child of the abstract Guitar class.

The two Guitar object constructors, virtual void *display()* method, and destructor are declared as public, as these must be available to be called at any point in the codes main function.

2.2 Over-Loading using either Methods or Constructors

Overloading of constructors is utilised in all class declarations of this project. For example in the “ElectricGuitar.h” class declaration:

```
public:
    ElectricGuitar(int, string, float, string, int, int, string);
    ElectricGuitar(string, float, string, int, int, string);
    void display();
    virtual ~ElectricGuitar();
```

In this context, the overloading of constructors allow for the user to input a *serialNumber* for their ElectricGuitar object, or, if no value is entered, the Guitar class construct method will set the value using a static integer variable.

2.3 Abstract Classes

The Guitar class is abstract within this project, containing a display method which must be over-ridden within its child classes. The class is made abstract using the following syntax on a method of the class:

```
virtual <classMethod> = 0;
```

The following shows how this is implemented for both the Guitar class, making its *display()* methods virtual.

```
virtual void display() =0;
```

2.4 Over-Riding of a Method

Over-riding is used within all non-abstract child classes of the base “Instrument” class. Over-riding is used in each case to implement the *display()* method of each class. The following example demonstrates how the *display()* method is over- ridden in the “AcousticGuitar” class.

```
//Header File
```

```
void display();
```

```
//C++ File
```

```
void AcousticGuitar::display() {  
    this->Guitar::display();  
    cout << "Number of Soundholes: " << this->noOfSoundholes << "." << endl;  
}
```

2.5 Multiple Inheritance

Multiple inheritance is used to inherit properties from multiple base classes into a single derived class. This is used in this assignment for the “ElectroAcousticGuitar” class, as it requires properties of both the “ElectricGuitar” class, and the “AcousticGuitar” class. The constructor for this class must make use of both the “ElectricGuitar”, and the “AcousticGuitar” constructors. As the two base classes share a common base class

in the “Guitar” class, the base classes must declare the shared base class as virtual, as follows:

```
class ElectricGuitar: public virtual Guitar {
```

```
class AcousticGuitar: public virtual Guitar {
```

This modification allows for the “ElectroAcousticGuitar” class to be defined.

```
//Header
```

```
class ElectroAcousticGuitar: public AcousticGuitar, public ElectricGuitar {
public:
    ElectroAcousticGuitar(int, string, float, string, int, int, int, string);
    ElectroAcousticGuitar(string, float, string, int, int, int, string);
    void display();
    virtual ~ElectroAcousticGuitar();
};
```

```
//Constructors
```

```
ElectroAcousticGuitar::ElectroAcousticGuitar(int aSerialNumber,
    string aManufacturer, float aPrice, string aBodyShape,
    int numStrings, int numSoundholes, int numPickups,
    string typePickup) :
    AcousticGuitar(aSerialNumber, aManufacturer,
        aPrice, aBodyShape, numStrings, numSoundholes),
    ElectricGuitar(aSerialNumber, aManufacturer,
        aPrice, aBodyShape, numStrings, numPickups,
        typePickup),
    Guitar(aSerialNumber, aManufacturer, aPrice,
        aBodyShape, numStrings){
}

ElectroAcousticGuitar::ElectroAcousticGuitar(string aManufacturer,
    float aPrice, string aBodyShape, int numStrings,
    int numSoundholes, int numPickups, string typePickup) :
    AcousticGuitar(aManufacturer, aPrice, aBodyShape,
        numStrings, numSoundholes),
    ElectricGuitar(aManufacturer, aPrice, aBodyShape,
        numStrings, numPickups, typePickup),
    Guitar(aManufacturer, aPrice, aBodyShape, numStrings){
}
```

2.6 Separate Compilation with all Classes

Separate compilation is used with all classes of the project. Each class is defined within a header file, with their method implementations contained in the C++ file. The header files are then included within the main project file, “TestApp.cpp”, using `*#include*` statements.

```
#include "Instrument.h"
#include "Guitar.h"
#include "Piano.h"
#include "ElectricGuitar.h"
#include "AcousticGuitar.h"
#include "AcousticPiano.h"
#include "ElectroAcousticGuitar.h"
#include "ObjectStorage.h"
```

2.7 Friend Functions

A friend function is implemented within the “Instrument” class. This function allows access to the *protected*: `float Price` variable, from outside of the class. In this application, it is used to modify the `price` variable, allowing a discount to be given to the instrument. The function is declared a friend within the “Instrument” class, and declared within the “TestApp.cpp” file. The `salePrice()` method takes a float, and an Instrument object, and subtracts the float value from the Instruments `price` value.

```
void salePrice(Instrument &saleInstrument, float discount){
    saleInstrument.price = saleInstrument.price-discount;
    cout << "Rounded Guitar Sale Price: "
    << static_cast<int>(saleInstrument.price) << "." << endl;
}
```

2.8 Modified Copy Constructor

The copy constructor is called whenever an object is passed by value to a function. It creates an object with the same properties as the original object, and is destroyed once the function is exited. The modified copy constructor is implemented within the “AcousticPiano” class. It creates an object with *manufacturer* value of “Copy”, a *price* value of 0, a *numKeys* value of 60, and a *pianoShape* value of “Upright”.

```
AcousticPiano::AcousticPiano(const AcousticPiano &sourcePiano) :
    AcousticPiano("Copy", 0, 60, "Upright"){
```

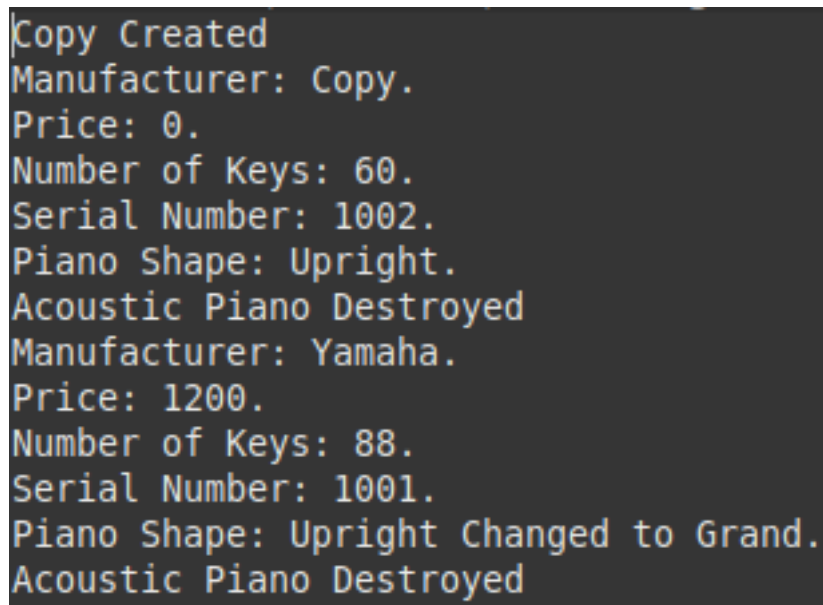
```
    cout << "Copy Created" << endl;  
}
```

2.8.1 Pass-by-Value

In order to test the affect of the modified copy constructor on pass-by-value, the *changePianoShapePBV()* method is used. This method is passed an object, and a *newShape* string value. The *pianoShape* of the object is then modified, and returned from the function.

```
string changePianoShapePBV(AcousticPiano source, string newShape){  
    source.display();  
    return source.pianoShape + " Changed to " + newShape;  
}
```

Calling this function, followed by a display function, returns the output seen in Figure 1, and demonstrates that the copy constructor is used when pass-by-value is used.



```
Copy Created  
Manufacturer: Copy.  
Price: 0.  
Number of Keys: 60.  
Serial Number: 1002.  
Piano Shape: Upright.  
Acoustic Piano Destroyed  
Manufacturer: Yamaha.  
Price: 1200.  
Number of Keys: 88.  
Serial Number: 1001.  
Piano Shape: Upright Changed to Grand.  
Acoustic Piano Destroyed
```

Figure 1: Copy Constructor Pass-By-Value Output

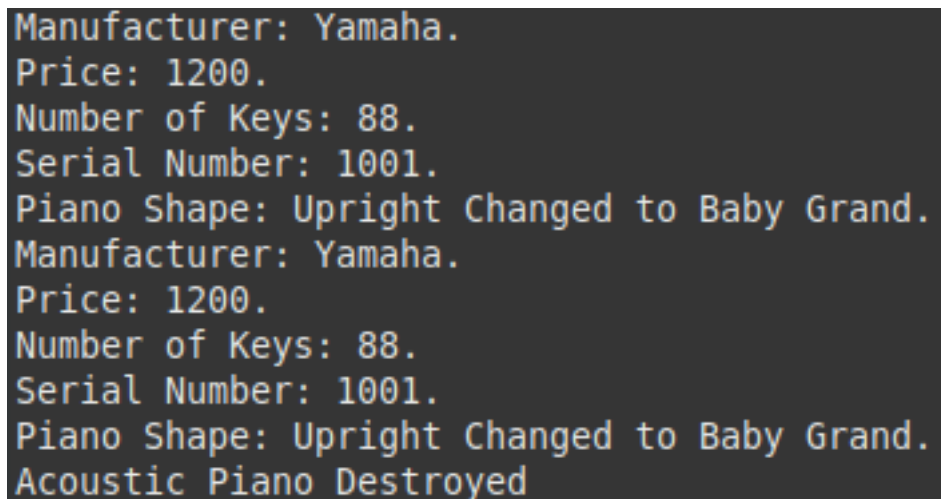
2.8.2 Pass-by-Reference

In order to test the affect of the modified copy constructor on pass-by-reference, the *changePianoShapePBR()* method is used. This method works in the same way as the

`changePianoShapePBV()` method, however, the “AcousticPiano” object is passed by reference into the function, and the method is defined as *void*.

```
void changePianoShapePBR(AcousticPiano &source, string newShape){  
    source.pianoShape = source.pianoShape + " Changed to " + newShape;  
    source.display();  
}
```

Calling this function, followed by a display function gives the output seen in Figure 2, and demonstrates that the copy constructor is not used when pass-by-reference is used.



```
Manufacturer: Yamaha.  
Price: 1200.  
Number of Keys: 88.  
Serial Number: 1001.  
Piano Shape: Upright Changed to Baby Grand.  
Manufacturer: Yamaha.  
Price: 1200.  
Number of Keys: 88.  
Serial Number: 1001.  
Piano Shape: Upright Changed to Baby Grand.  
Acoustic Piano Destroyed
```

Figure 2: Copy Constructor Pass-By-Reference Output

2.9 Destructor with Basic Functionality

A destructor is implemented in each of the derived classes of the project. The destructor is called at the end of the main function, or can otherwise be called explicitly. In each case, the destructor informs the user that the object has been destroyed.

```
ElectricGuitar::~ElectricGuitar() {  
    cout << "Electric Guitar Destroyed" << endl;  
}
```

2.10 Over-Loaded Operators

Over-loading operators allows for specific and tailored functionality for objects of a class. It allows the programmer, for example, to define what it means for one object to be less

than another, or to define what equality between objects means. Within this project, the `+`, `==`, and `<` operators of the “Piano” class are over-loaded.

2.10.1 `+`

The following code snippet displays how the `+` operator is over-loaded within the “Piano” class.

```
Piano Piano::operator +(Piano inPiano) {  
    return Piano(inPiano.manufacturer+manufacturer, inPiano.price+price, noOfKeys+inPiano.noOfKeys);  
}
```

2.10.2 `==`

The following code snippet displays how the `==` operator is over-loaded within the “Piano” class.

```
bool Piano::operator ==(Piano inPiano){  
    if((inPiano.serialNumber == serialNumber) && (inPiano.manufacturer == manufacturer)  
        && (inPiano.price == price) && (inPiano.noOfKeys == noOfKeys)){  
        return true;  
    }  
    else return false;  
}
```

2.10.3 `<`

The following code snippet displays how the `<` operator is over-loaded within the “Piano” class.

```
bool Piano::operator <(Piano inPiano){  
    return (serialNumber<inPiano.serialNumber? true:false);  
}
```

The over-loaded operators were tested in by first “adding” two Piano objects, and then, by equating two objects.

```
Piano addPiano1 = arr[0]+arr[1];  
Piano addPiano2 = addPiano1;  
if(arr[0]==arr[1]){
```

```
        cout << "Matching Pianos." << endl;
    }
    else cout << "Non-matching Pianos" << endl;
    if(addPiano1==addPiano2){
        cout << "Matching Pianos." << endl;
    }
    else cout << "Non-matching Pianos" << endl;
```

2.11 Pointers to Arrays of Objects

Pointers are used in the main function to demonstrate the use of operator over-loading of the “Piano” class. In this code snippet, a pointer to an array of Piano objects is initialized, and two objects are created using the Piano constructor. The operations performed are described in the “Operator Over-Loading” section above.

```
Piano *arr = new Piano[2];
arr[0] = Piano("Yamaha", 1200, 88);
arr[1] = Piano("Thomann", 600, 88);
```

2.12 C++ Explicit Style Casts

Casts are used to change a variable or object from one type to another. While this may cause issues, it can be used to the benefit of the application.

2.12.1 Static Cast

The static cast is used within this project as a “narrowing conversion”, changing a *float* variable to an *int* variable. It is implemented within the *salePrice()* function, converting the *price* float variable to an integer.

```
void salePrice(Instrument &saleInstrument, float discount){
    saleInstrument.price = saleInstrument.price-discount;
    cout << "Rounded Guitar Sale Price: "
    << static_cast<int>(saleInstrument.price) << "." << endl;
}
```

2.12.2 Dynamic Cast

Dynamic casting is used to convert from a base class to a derived class. Within this project this is demonstrated by first upcasting an object of the “ElectroAcousticGuitar” class, and then, using dynamic casting, converting it back to an “ElectroAcousticGuitar” object.

```
Instrument& upCastInstrument = testElectroAcoustic1;  
ElectricGuitar& downCastElectric = dynamic_cast<ElectricGuitar&>(upCastInstrument);
```

2.12.3 Constant Cast

Constant casts are used to convert from a constant variable type to another type. Within this project this is demonstrated by passing a *const float* value to the *salePrice()* method.

```
const float discount = 400.55;  
salePrice(testElectroAcoustic1, *const_cast<float*>(&discount));
```

2.12.4 Reinterpret Cast

Reinterpret casts can be used to convert from any type to any other type. This can cause numerous problems within a project, and is typically used for low level programming purposes only. Within this project it is used to convert an “AcousticGuitar” object to a variable of type long, and then back to an “AcousticGuitar” object.

```
AcousticGuitar *reintAcoustic = new AcousticGuitar("Martin", 900, "Jumbo", 6, 1);  
long longAcoustic = reinterpret_cast<long>(reintAcoustic);  
cout << longAcoustic << endl;  
AcousticGuitar *reintCastAcoustic =  
    reinterpret_cast<AcousticGuitar*>(longAcoustic);
```

2.13 Dynamic Binding

Dynamic binding is used to allow a function determine at compilation what the class of the object passed in is, i.e. is it an object of the base class, or an object of the derived class. By declaring a method of a class as virtual, dynamic binding may be used with this method. This is demonstrated with the *display()* method of the “Guitar” class, from which the “ElectricGuitar”, “AcousticGuitar”, and “ElectroAcousticGuitar” classes are derived.

The `displayGuitarInfo()` method takes an object of class “Guitar”, or any of its derived classes, and executes its `display()` method. This is demonstrated within the project by passing in an object of type “AcousticGuitar”.

```
void displayGuitarInfo(Guitar &A){
    A.display();
}

displayGuitarInfo(testAcoustic1);
```

2.14 Correct use of “new” and “delete”

The `new` and `delete` keywords are used in C++ for the manual allocation and deallocation of memory, often when using pointers. Within this project, they can be seen in the “Pointers to Arrays of Objects” section, as a pointer to an array is used, and objects of this array must be created using the `new` keyword. Once the operations on these pointers are completed their memory is deallocated using the `delete` keyword.

```
Piano *arr = new Piano[2];
arr[0] = Piano("Yamaha", 1200, 88);
arr[1] = Piano("Thomann", 600, 88);

:

delete arr;
```

2.15 Static States of a Class

Static variables are variables which are shared with all instances of a class. As such they have been utilized in this project in order to allow each derived “Guitar” or “Piano” object to have unique *serialNumber* values, and therefore not allowing duplicates (although, using the over-loaded constructor, a *serialNumber* may be defined by the user).

```
class Guitar: public Instrument{
private:
    static int nextGuitarSerialNumber;
    int serialNumber;
    void construct();
```

Within this example, the `construct()` method is used to set the *serialNumber* value, and increment the *nextGuitarSerialNumber* value.

```
void Guitar::construct(){
    serialNumber = nextGuitarSerialNumber++;
}
```

2.16 C++ Class vs. C++ Struct

C++ Structures are an extended form of C Structures, which allow for similar functionality as classes. Within C++ the main difference between a Class and Structure is that, within a structure, the default access specifier is *public*, whereas in Classes, the default is *private*. This has been demonstrated within the “TestApp.cpp” file, wherein an “InstrumentStruct” structure is implemented, with all properties using the default access specifier. Within the main function, an object of the structure is created, its variables initialized, and its *display()* method called. None of this would be possible using a Classes default access specifier, as the variables and methods would not have been declared within the scope of the main function.

```
//Structure Definition
struct InstrumentStruct{
    int serialNumber;
    string manufacturer;
    float price;
    void changeSerialNumber();
    void display();
};

//Structure Implementation
InstrumentStruct testStructObject;
testStructObject.serialNumber = 12345;
testStructObject.manufacturer = "TestManufacturer";
testStructObject.price = 12500.50;
testStructObject.display();
testStructObject.changeSerialNumber();
testStructObject.display();
```

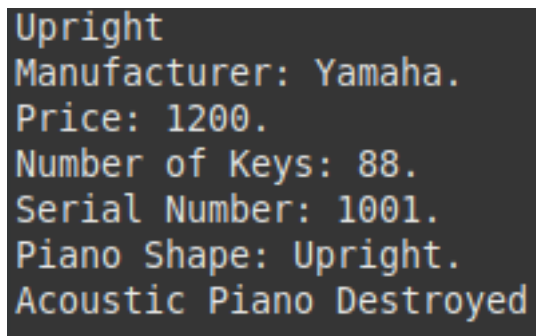
2.17 Passing by Constant Reference

Passing by constant reference combines functionality of both pass-by-value and pass-by-reference. It allows for an object to be passed to a function without creating a copy of that object, but does not allow for any alterations to be made to any properties of that object. The *showPianoShapeCPBR()* method within the project demonstrates this. An “AcousticPiano” object is passed by constant reference, and its shape output to the

console. The commented out assignment of the *pianoShape* property of the object would cause compilation errors if left uncommented.

```
void showPianoShapeCPBR(const AcousticPiano &source){  
    cout << source.pianoShape << endl;  
    //source.pianoShape = "Cannot Make Changes Here";  
}
```

Calling this function, followed by a display function gives the output seen in Figure 3. It can also be seen that the copy constructor is not used when constant pass-by-reference is used.



```
Upright  
Manufacturer: Yamaha.  
Price: 1200.  
Number of Keys: 88.  
Serial Number: 1001.  
Piano Shape: Upright.  
Acoustic Piano Destroyed
```

Figure 3: Constant Pass-By-Reference Output

2.18 Class Template

Class templates give a generic functionality, which may be applied to objects of different types. Objects of any type may be passed to a template class, which allows them to be used for storage purposes. Within this project, templates are demonstrated in two areas, however, only one is implemented as a class for storage purposes. The other is a functor, used in conjunction with vectors, and shall be explained in the “Vector” section.

The “ObjectStorage” template class takes an object of any type, and an array size integer, and creates an array of these objects. There is a method defined, *getMaxOfTwo()* which returns which of the two specified objects is greater. This method requires operator over-loading to define what is meant by one object being “greater than” another.

```
template<class T, int storeSize>  
class ObjectStorage {  
    T array[storeSize];  
public:  
    T getMaxOfTwo(int i, int j){  
        T outVal = array[i]>array[j]? array[i]:array[j];
```

```
        return outVal;}  
};
```

2.19 Vectors

A vector is a form of container, which has advanced functions. It is used to store an array of elements, and allows for the use of a number of sorting and iterating functions. In order to demonstrate this within the assignment, it is used with objects of the “Piano” class. Using the *pushBack()* method, objects are added to the vector.

```
// Vector Implementation  
vector<Piano> pianoVector;  
pianoVector.push_back(Piano(12456, "Test", 1200, 88));  
pianoVector.push_back(Piano(87456, "Test", 1200, 88));  
pianoVector.push_back(Piano(12556, "Test", 1200, 88));  
pianoVector.push_back(Piano(99999, "Test", 1200, 88));  
pianoVector.push_back(Piano(50820, "Test", 1200, 88));
```

2.20 Sort Algorithm

The vector described above is sorted using the *sort()* method, which, utilising the “Piano” classes overloaded *<* operator, sorts the objects in order of increasing *serialNumber* values. The *for_each()* method is then used, alongside the *dispFunc<>()* functor, to iterate through the vector, and display the object information. The defined functor calls the *display()* method of the object which is passed into it, in this case, the “Piano” objects of the vector.

```
//Functor  
  
template<class T>class dispFunc{  
public:  
    void operator () (T x){  
        x.display();  
    }  
};  
  
//Sort Implementation  
sort(pianoVector.begin(), pianoVector.end());  
for_each(pianoVector.begin(), pianoVector.end(), dispFunc<Piano>());
```

Appendix

The following is the C++ code used for the completion of this assignment.

3.1 Instrument Header and C++ File

3.1.1 Header

```
#ifndef INSTRUMENT_H_
#define INSTRUMENT_H_

#include<iostream>

using std::string;

class Instrument {
private:
    friend void salePrice(Instrument &, float);
protected:
    string manufacturer;
    float price;
    Instrument();
    Instrument(string, float);
    virtual void display() = 0;
    virtual ~Instrument();
};

#endif /* INSTRUMENT_H_ */
```

3.1.2 C++

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

Instrument::Instrument(){
}

Instrument::Instrument(string aManufacturer, float aPrice) :
    manufacturer(aManufacturer), price(aPrice) {
```



```
}

Instrument::~Instrument() {
}

void Instrument::display(){
    cout << "Manufacturer: " << this->manufacturer << "." << endl;
    cout << "Price: " << this->price << "." << endl;
}
```

3.2 Guitar Header and C++ File

3.2.1 Header

```
#ifndef GUITAR_H_
#define GUITAR_H_

#include <iostream>
#include "Instrument.h"

using std::string;

//Report Section 2.1 Access Specifiers
//Report Section 2.15 Static States of a Class
class Guitar: public Instrument{
private:
    static int nextGuitarSerialNumber;
    int serialNumber;
    void construct();
protected:
    string bodyShape;
    int noOfStrings;
public:
    Guitar(int, string, float, string, int);
    Guitar(string, float, string, int);
    //Report Section 2.3 Abstract classes
    virtual void display() =0;
    virtual ~Guitar();
};

#endif /* GUITAR_H_ */
```

3.2.2 C++

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

using namespace std;

Guitar::Guitar(int aSerialNumber, string aManufacturer, float aPrice,
               string aBodyShape, int numStrings) :
    Instrument(aManufacturer, aPrice), serialNumber(aSerialNumber),
    bodyShape(aBodyShape), noOfStrings(numStrings) {
}

Guitar::Guitar(string aManufacturer, float aPrice, string aBodyShape,
               int numStrings) :
    Instrument(aManufacturer, aPrice), bodyShape(aBodyShape),
    noOfStrings(numStrings) {
    construct();
}

Guitar::~Guitar() {
}

int Guitar::nextGuitarSerialNumber = 0001;

void Guitar::display() {
    this->Instrument::display();
    cout << "Material: " << this->bodyShape << "." << endl;
    cout << "Number of Strings: " << this->noOfStrings << "." << endl;
    cout << "Serial Number: " << this->serialNumber << "." << endl;
}

//Report Section 2.15 Static States of a Class
void Guitar::construct(){
    serialNumber = nextGuitarSerialNumber++;
}
```

3.3 Piano Header and C++ File

3.3.1 Header

```
#ifndef PIANO_H_
#define PIANO_H_

#include <iostream>
#include "Instrument.h"

using std::string;

class Piano: public Instrument {
private:
    static int nextPianoSerialNumber;
    int serialNumber;
    void construct();
protected:
    int noOfKeys;
public:
    Piano();
    Piano(int, string, float, int);
    Piano(string, float, int);
    bool operator < (Piano);
    bool operator == (Piano);
    Piano operator + (Piano);
    virtual void display();
    virtual ~Piano();
};

#endif /* PIANO_H_ */
```

3.3.2 C++

```
#include <iostream>
#include <string>
#include "Piano.h"

using namespace std;

Piano::Piano(){
}
```

```
Piano::Piano(int aSerialNumber, string aManufacturer, float aPrice, int numKeys) :
    Instrument(aManufacturer, aPrice), serialNumber(aSerialNumber),
    noOfKeys(numKeys){
}

Piano::Piano(string aManufacturer, float aPrice, int numKeys) :
    Instrument(aManufacturer, aPrice), noOfKeys(numKeys){
    construct();
}

Piano::~~Piano() {
}

int Piano::nextPianoSerialNumber = 1001;

void Piano::display(){
    this->Instrument::display();
    cout << "Number of Keys: " << this->noOfKeys << "." << endl;
    cout << "Serial Number: " << this->serialNumber << "." << endl;
}

void Piano::construct(){
    serialNumber = nextPianoSerialNumber++;
}

//Report Section 2.10.1 Operator Overloading "+"
Piano Piano::operator +(Piano inPiano) {
    return Piano(inPiano.manufacturer+manufacturer, inPiano.price+price,
    noOfKeys+inPiano.noOfKeys);
}

//Report Section 2.10.3 Operator Overloading "<"
bool Piano::operator <(Piano inPiano){
    return (serialNumber<inPiano.serialNumber? true:false);
}

//Report Section 2.10.2 Operator Overloading "=="
bool Piano::operator ==(Piano inPiano){
    if((inPiano.serialNumber == serialNumber)
        && (inPiano.manufacturer == manufacturer)
        && (inPiano.price == price)
        && (inPiano.noOfKeys == noOfKeys)){
        return true;
    }
}
```

```
    }  
    else return false;  
}
```

3.4 ElectricGuitar Header and C++ File

3.4.1 Header

```
#ifndef ELECTRICGUITAR_H_  
#define ELECTRICGUITAR_H_  
  
#include <iostream>  
#include "Guitar.h"  
  
using std::string;  
  
//Report Section 2.2 Over-Loading using Constructors  
//Report Section 2.5 Multiple Inheritance  
class ElectricGuitar: public virtual Guitar {  
private:  
protected:  
    int noOfPickups;  
    string pickupType;  
public:  
    ElectricGuitar(int, string, float, string, int, int, string);  
    ElectricGuitar(string, float, string, int, int, string);  
    void display();  
    virtual ~ElectricGuitar();  
};  
  
#endif /* ELECTRICGUITAR_H_ */
```

3.4.2 C++

```
#include "ElectricGuitar.h"  
  
using namespace std;  
  
ElectricGuitar::ElectricGuitar(int aSerialNumber, string aManufacturer,  
                                float aPrice, string aBodyShape, int numStrings, int numPickups,
```

```

        string typePickup) :
        Guitar(aSerialNumber, aManufacturer, aPrice, aBodyShape, numStrings),
        noOfPickups(numPickups), pickupType(typePickup) {
    }

    ElectricGuitar::ElectricGuitar(string aManufacturer, float aPrice,
        string aBodyShape, int numStrings, int numPickups,
        string typePickup) :
        Guitar(aManufacturer, aPrice, aBodyShape, numStrings),
        noOfPickups(numPickups), pickupType(typePickup){
    }

    //Report Section 2.9 Destructor with Basic Functionality
    ElectricGuitar::~ElectricGuitar() {
        cout << "Electric Guitar Destroyed" << endl;
    }

    void ElectricGuitar::display() {
        this->Guitar::display();
        cout << "Number of Pickups: " << this->noOfPickups << "." << endl;
        cout << "Pickup Type: " << this->pickupType << "." << endl;
    }

```

3.5 AcousticGuitar Header and C++ File

3.5.1 Header

```

#ifndef ACOUSTICGUITAR_H_
#define ACOUSTICGUITAR_H_

#include <iostream>
#include "Guitar.h"

using std::string;

//Report Section 2.5 Multiple Inheritance
class AcousticGuitar: public virtual Guitar {
private:
protected:
public:
    int noOfSoundholes;
    AcousticGuitar(int, string, float, string, int, int);

```

```

    AcousticGuitar(string, float, string, int, int);
    //Report Section 2.4 Over-Riding of a Method
    void display();
    virtual ~AcousticGuitar();
};

#endif /* ACOUSTICGUITAR_H_ */

```

3.5.2 C++

```

#include "AcousticGuitar.h"

using namespace std;

AcousticGuitar::AcousticGuitar(int aSerialNumber, string aManufacturer,
    float aPrice, string aBodyShape, int numStrings, int numSoundholes):
    Guitar(aSerialNumber, aManufacturer, aPrice, aBodyShape, numStrings),
    noOfSoundholes(numSoundholes){
}

AcousticGuitar::AcousticGuitar(string aManufacturer, float aPrice,
    string aBodyShape, int numStrings,
    int numSoundholes) :
    Guitar(aManufacturer, aPrice, aBodyShape, numStrings),
    noOfSoundholes(numSoundholes){
}

AcousticGuitar::~AcousticGuitar() {
    cout << "Acoustic Guitar Destroyed" << endl;
}

//Report Section 2.4 Over-Riding of a Method
void AcousticGuitar::display() {
    this->Guitar::display();
    cout << "Number of Soundholes: " << this->noOfSoundholes << "." << endl;
}

```

3.6 AcousticPiano Header and C++ File

3.6.1 Header

```
#ifndef ACOUSTICPIANO_H_
#define ACOUSTICPIANO_H_

#include <iostream>
#include "Piano.h"

using std::string;

class AcousticPiano: public Piano {
private:
public:
    string pianoShape;
    AcousticPiano(int, string, float, int, string);
    AcousticPiano(string, float, int, string);
    AcousticPiano(const AcousticPiano&);
    void display();
    virtual ~AcousticPiano();
};

#endif /* ACOUSTICPIANO_H_ */
```

3.6.2 C++

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

using namespace std;

AcousticPiano::AcousticPiano(int aSerialNumber, string aManufacturer, float aPrice,
    int numKeys, string aPianoShape) :
    Piano(aSerialNumber, aManufacturer, aPrice, numKeys), pianoShape(aPianoShape){
}

AcousticPiano::AcousticPiano(string aManufacturer, float aPrice, int numKeys,
    string aPianoShape) :
    Piano(aManufacturer, aPrice, numKeys), pianoShape(aPianoShape){
}
```



```

//Report Section 2.8 Modified Copy Constructor
AcousticPiano::AcousticPiano(const AcousticPiano &sourcePiano) :
    AcousticPiano("Copy", 0, 60, "Upright"){
    cout << "Copy Created" << endl;
}

AcousticPiano::~AcousticPiano() {
    cout << "Acoustic Piano Destroyed" << endl;
}

void AcousticPiano::display(){
    this->Piano::display();
    cout << "Piano Shape: " << this->pianoShape << "." << endl;
}

```

3.7 ElectroAcousticGuitar Header and C++ File

3.7.1 Header

```

#ifndef ELECTROACOUSTICGUITAR_H_
#define ELECTROACOUSTICGUITAR_H_

#include <iostream>
#include "AcousticGuitar.h"
#include "ElectricGuitar.h"

using std::string;

class ElectroAcousticGuitar: public AcousticGuitar, public ElectricGuitar {
public:
    //Report Section 2.5 Multiple Inheritance
    ElectroAcousticGuitar(int, string, float, string, int, int, int, string);
    ElectroAcousticGuitar(string, float, string, int, int, int, string);
    void display();
    virtual ~ElectroAcousticGuitar();
};

#endif /* ELECTROACOUSTICGUITAR_H_ */

```

3.7.2 C++

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

using namespace std;

//Report Section 2.5 Multiple Inheritance
ElectroAcousticGuitar::ElectroAcousticGuitar(int aSerialNumber,
    string aManufacturer, float aPrice, string aBodyShape,
    int numStrings, int numSoundholes, int numPickups, string typePickup) :
    AcousticGuitar(aSerialNumber, aManufacturer, aPrice, aBodyShape,
        numStrings, numSoundholes),
    ElectricGuitar(aSerialNumber, aManufacturer, aPrice, aBodyShape,
        numStrings, numPickups, typePickup),
    Guitar(aSerialNumber, aManufacturer, aPrice, aBodyShape,
        numStrings){
}

ElectroAcousticGuitar::ElectroAcousticGuitar(string aManufacturer, float aPrice,
    string aBodyShape, int numStrings, int numSoundholes,
    int numPickups, string typePickup) :
    AcousticGuitar(aManufacturer, aPrice, aBodyShape, numStrings,
        numSoundholes),
    ElectricGuitar(aManufacturer, aPrice, aBodyShape, numStrings,
        numPickups, typePickup),
    Guitar(aManufacturer, aPrice, aBodyShape, numStrings){
}

ElectroAcousticGuitar::~ElectroAcousticGuitar() {
    cout << "Destructor Output: " << endl;
    cout << "ElectroAcoustic Destroyed" << endl;
    cout << "Guitar had the following details:" << endl;
    this->display();
}

void ElectroAcousticGuitar::display(){
    cout << "Display Function Output: " << endl;
    this->AcousticGuitar::display();
    cout << "Number of Pickups: " <<
    this->ElectricGuitar::noOfPickups << "." << endl;
    cout << "Pickup Type: " << this->ElectricGuitar::pickupType << "." << endl;
}
```

3.8 ObjectStorage Template Header File

3.8.1 Header File

```
#ifndef OBJECTSTORAGE_H_
#define OBJECTSTORAGE_H_

#include <iostream>

using std::string;

//Report Section 2.18 Class Template
template<class T, int storeSize>
class ObjectStorage {
    T array[storeSize];
public:
    T getMaxOfTwo(int i, int j){
        T outVal = array[i]>array[j]? array[i]:array[j];
        return outVal;}
};

#endif /* OBJECTSTORAGE_H_ */
```

3.9 TestApp.cpp File

```
//Report Section 2.6 Separate Compilation with all Classes
#include "Instrument.h"
#include "Guitar.h"
#include "Piano.h"
#include "ElectricGuitar.h"
#include "AcousticGuitar.h"
#include "AcousticPiano.h"
#include "ElectroAcousticGuitar.h"
#include "ObjectStorage.h"
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

//Report Section 2.20 Sort Algorithm
```

```
template<class T>class dispFunc{
public:
    void operator () (T x){
        x.display();
    }
};

//Report Section 2.16 C++ Class vs. C++ Struct
struct InstrumentStruct{
    int serialNumber;
    string manufacturer;
    float price;
    void changeSerialNumber();
    void display();
};

void InstrumentStruct::display(){
    cout << this->serialNumber << endl;
    cout << this->manufacturer << endl;
    cout << this->price << endl;
}

void InstrumentStruct::changeSerialNumber(){
    serialNumber++;
}

//Friend Function
//Report Section 2.7 Friend Functions
void salePrice(Instrument &saleInstrument, float discount){
    saleInstrument.price = saleInstrument.price-discount;
    //Report Section 2.12.1 Static Cast
    cout << "Rounded Guitar Sale Price: " <<
        static_cast<int>(saleInstrument.price) << "." << endl;
}

//Report Section 2.8.1 Pass-By-Value
string changePianoShapePBV(AcousticPiano source, string newShape){
    source.display();
    return source.pianoShape + " Changed to " + newShape;
}

//Report Section 2.8.2 Pass-By-Reference
void changePianoShapePBR(AcousticPiano &source, string newShape){
    source.pianoShape = source.pianoShape + " Changed to " + newShape;
```

```
        source.display();
    }

    //Report Section 2.17 Passing by Constant Reference
    void showPianoShapeCPBR(const AcousticPiano &source){
        cout << source.pianoShape << endl;
    }

    //Dynamic Binding
    //Report Section 2.13 Dynamic Binding
    void displayGuitarInfo(Guitar &A){
        A.display();
    }

    int main(){

        //Report Section 2.12.3 Constant Cast
        const float discount = 400.55;

        //Constructor Overloading
        ElectricGuitar testElectric1 = ElectricGuitar("Gibson", 2400,
            "LP", 6, 2, "Humbucker");
        AcousticGuitar testAcoustic1 = AcousticGuitar("Taylor", 800,
            "Dreadnought", 12, 1);
        ElectricGuitar testElectric2 = ElectricGuitar(12345, "Gibson", 2400,
            "LP", 6, 2, "Humbucker");
        AcousticGuitar testAcoustic2 = AcousticGuitar(12611, "Taylor", 800,
            "Dreadnought", 12, 1);
        AcousticPiano testAcousticPiano = AcousticPiano("Yamaha", 1200, 88,
            "Upright");

        //Copy Constructor Pass-By-Value Pass-By-Reference Constant-Pass-By-Reference
        testAcousticPiano.pianoShape = changePianoShapePBV(testAcousticPiano, "Grand");
        testAcousticPiano.display();
        changePianoShapePBR(testAcousticPiano, "Baby Grand");
        testAcousticPiano.display();
        showPianoShapeCPBR(testAcousticPiano);
        testAcousticPiano.display();

        //Pointers to Object Arrays
        //Report Section 2.11 Pointers to Arrays of Objects
        //Report Section 2.14 Correct use of "new" and "delete"
        Piano *arr = new Piano[2];
        arr[0] = Piano("Yamaha", 1200, 88);
```

```

arr[1] = Piano("Thomann", 600, 88);

//Operator Overloading
//Report Section 2.10.3 Operator Overloading "<"
Piano addPiano1 = arr[0]+arr[1];
Piano addPiano2 = addPiano1;
if(arr[0]==arr[1]){
    cout << "Matching Pianos." << endl;
}
else cout << "Non-matching Pianos" << endl;
if(addPiano1==addPiano2){
    cout << "Matching Pianos." << endl;
}
else cout << "Non-matching Pianos" << endl;
delete arr;

//Multiple Inheritance
ElectroAcousticGuitar testElectroAcoustic1 = ElectroAcousticGuitar("Gretsch",
5000, "G2622T", 6, 2, 2, "Humbucker");
ElectroAcousticGuitar testElectroAcoustic2 = ElectroAcousticGuitar(72247,
"Gretsch", 5000, "G2622T", 6, 2, 2, "Humbucker");

//Dynamic Casting & Reinterpret Casting
//Report Section 2.12.2 Dynamic Cast
Instrument& upCastInstrument = testElectroAcoustic1;
ElectricGuitar& downCastElectric =
dynamic_cast<ElectricGuitar&>(upCastInstrument);
downCastElectric.display();
//Report Section 2.12.4 Reinterpret Cast
AcousticGuitar *reintAcoustic =
new AcousticGuitar("Martin", 900,
"Jumbo", 6, 1);
reintAcoustic->noOfSoundholes = 0;
long longAcoustic = reinterpret_cast<long>(reintAcoustic);
cout << longAcoustic << endl;
AcousticGuitar *reintCastAcoustic =
reinterpret_cast<AcousticGuitar*>(longAcoustic);
delete reintAcoustic;
cout << reintCastAcoustic->noOfSoundholes << endl;
testElectric1.display();
testAcoustic1.display();
testElectroAcoustic1.display();

// Friend Function & Constant Cast

```

```
//Report Section 2.12.3 Constant Cast
salePrice(testElectroAcoustic1, *const_cast<float*>(&discount));
testElectroAcoustic1.display();

//Dynamic Binding
//Report Section 2.13 Dynamic Binding
displayGuitarInfo(testAcoustic1);

//Structs
//Report Section 2.16 C++ Class vs. C++ Struct
InstrumentStruct testStructObject;
testStructObject.serialNumber = 12345;
testStructObject.manufacturer = "TestManufacturer";
testStructObject.price = 12500.50;
testStructObject.display();
//Unspecified Access Specifiers
//In Class would be private, in Struct are public
testStructObject.changeSerialNumber();
testStructObject.display();

//Vector Container and Sort
//Report Section 2.19 Vectors
vector<Piano> pianoVector;
pianoVector.push_back(Piano(12456, "Test", 1200, 88));
pianoVector.push_back(Piano(87456, "Test", 1200, 88));
pianoVector.push_back(Piano(12556, "Test", 1200, 88));
pianoVector.push_back(Piano(99999, "Test", 1200, 88));
pianoVector.push_back(Piano(50820, "Test", 1200, 88));
//Report Section 2.20 Sort Algorithm
sort(pianoVector.begin(), pianoVector.end());
for_each(pianoVector.begin(), pianoVector.end(), dispFunc<Piano>());
}
```