Instructions:

- Print your name in the space provided below.
- Answer each question in the space provided. If you need to continue an answer onto the back of a page, clearly indicate that you have done so; and label the continuation with the question number.
- If you want partial credit, justify your answers briefly and concisely, even when justification is not explicitly required.
- There are 11 questions, priced as marked. The maximum score is 100.
- When you have completed the test, sign the pledge at the bottom of this page and turn in the test.
- This is a closed-book, closed-notes examination. No calculators or other electronic devices may be used during this examination. You may not discuss (in any form: written, verbal or electronic) the content of this examination with any student who has not taken it. You must return this test form when you complete the examination. Failure to adhere to any of these restrictions is an Honor Code violation.

Do not start the test until instructed to do so!

## KEY

Name ____________________________ printed

PID ______________________________ printed

Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

_________________________ signed
Assume that the CD class referred to in the following code is a simple data class, containing only primitive variables or string objects and the insertion stream operator << is overloaded in the CD class. For the next three questions, consider the partial class interface code that follows:

```cpp
// "Music.h"
#include "CD.h"
const int MAXCDs = 100;

class Music {
    private:
        CD* discs[MAXCDs];
        int usage;
    public:
        Music();
        Music(CD* cds[], int num = 10);
        //possible missing Fn prototypes?
        ~Music();
        friend ostream& operator<<(ostream& out, const Music& m);
};

// "Music.cpp"
Music::Music(): usage(0) {
    for (int i=0; i < MAXCDs; i++)
        discs[i] = NULL;
}
Music::~Music() { }
Music::Music(CD* cds[], int num) {
    if (num >= MAXCDs) usage = MAXCDs;
    else usage = num;
    for (int i=0; i < usage; i++)
        discs[i] = cds[i];
    for (int i=usage; i < MAXCDs; i++)
        discs[i] = NULL;
}
ostream& operator<<(ostream& out, const Music& m) {
    for (int i=0; i<m.usage; i++)
        out << *(m.discs[i]) << endl;
    return out;
}
```

1. [6 points] The above code is an example of an association relationship between the CD and Music classes. Identify whether the above association relationship is dynamic or static association and very briefly explain/justify your response?

**Static, no mutator method exists to change the association.**

*Static/dynamic associations are NOT determined by static or dynamic memory allocation. A static association occurs when an association is created during object instantiation by a constructor and there is no way to alter the association until the object goes out of scope. A dynamic association occurs when an association between objects can be changed, (usually through a mutator method) at any time during the lifetime of the objects.*
2. [10 points] For whichever form of association you selected for problem one, explain what change(s) or addition(s) would need to be made to transform the association relationship from dynamic to static or vice versa. If extra functions are necessary for the transformation, give the prototype and implementation of each.

```cpp
bool addCD( CD* cd);

bool Music::addCD( CD* cd){
    if ( usage < MAXCDs ) {
        discs[usage] = cd;
        usage++;
        return true;
    } else
        return false;
}
```

*A removeCD method would also serve as a mutator method to affect a dynamic change in the association.*

3. [10 points] Assume that a *Music* object named *mus* exists and that the diagram below graphically depicts its current state in memory. Given the client code function prototype: `void manageCollection( Music myMusic);` If the function invocation: `manageCollection( mus );` occurs, then draw the resulting graphical diagram representing the state in memory of the formal parameter *myMusic*:

```
usage: 3
discs:
  [0] [1] [2] • • • [98] [99]
  CD1
  Title: Revolver
  Artist: Beatles
  CD2
  Title: Pilgrim
  Artist: E. Clapton
  CD3
  Title: Blues Summit
  Artist: BB King

usage: 3
discs:
  [0] [1] [2] • • • [98] [99]
  myMusic
```
4. [10 points] You have been charged with developing a taxi simulation. Another programmer in the project is developing the Taxi class and you are developing the Fare class. Consider the partial Taxi class interface:

```cpp
class Taxi {
private:
    Fare** fares; // array of Fare ptrs
    unsigned int riders;
public:
    // omitted constructors
    // omitted function prototypes
    bool pickupFare(Fare* f) {
        fares[++riders] = f;
        return true;
    }
}
```

Give the code to implement a `boardTaxi( ? )` method of the Fare class by invoking the Taxi class `pickupFare` service.

```cpp
bool Fare::boardTaxi( Taxi* t) {
    taxi = t; // optional - store association, bi-directional
    if ( t != NULL ) {
        t -> pickupFare( this );
        return true;
    } else return false;
}
```

5. [9 points] List the three C++ mechanisms necessary to achieve polymorphism:

1. __inheritance________________________________________

2. __virtual (over-ridden) functions or late binding or v-table________________________________________

3. __base pointers________________________________________
For the next two questions, consider the following simple class hierarchy:

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

class Counter {
private:
    unsigned int Sec;
public:
    Counter(unsigned int Seconds = 0);
    unsigned int Increment();
};

Counter::Counter(unsigned int Seconds) {
    Sec = Seconds;
}

unsigned int Counter::Increment() {
    Sec = (Sec + 1) % 60;
    if (Sec == 0)
        return 1;
    else
        return 0;
}

class Timer : public Counter {
private:
    unsigned int Min;
public:
    Timer(unsigned int Seconds = 0);
    unsigned int Increment();
};

Timer::Timer(unsigned int Seconds) {
    Min = Seconds / 60;
    Sec = Seconds % 60;
}

unsigned int Timer::Increment() {
    unsigned int Carry = // invoke base Increment();
    Min = Min + Carry;
    return 0;
}
```
6. [10 points] One of the lines in the above class code contains a common error dealing with inheritance. Quote the statement with the error, briefly explain the error and then list two simple fixes for the problem.

Sec = Seconds % 60;

The public derived class Timer cannot access the private member of the base Counter class.

1. Invoke the Counter constructor in the initializer list of the Timer constructor to initialize the base slice: Timer::Timer(unsigned int Seconds): Counter(Seconds%60) {

2. Change the private member in the Counter class to protected, (changing it to public will also work but it is an abomination).

The answers below are acceptable, but are kluges at best.

3. Invoke the Counter::Increment() method in a loop (Seconds % 60) times.

4. Add a protected mutator method to Counter to change Sec.

5. Make Timer a friend class of Counter.

7. [9 points] Is it possible to invoke the over-ridden Increment base class function from within the derived Timer class over-riding function? If your response is yes, then give the syntax for doing so. If your response is no, then briefly explain why not.

Yes,

Unsigned int Carry = Counter::Increment();

To invoke base class over-ridden functions they must qualified by the base class name and the scope resolution operator to avoid a recursive call.
For the next three questions, consider the following poorly designed class hierarchy:

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

class Red {
private:
  int r;
public:
  Red(int i = 11): r(i) {} //{r = i;}
  int getR() const {return r;}
  virtual int sum() const = 0;
};
class Green : public Red {
private:
  int g;
public:
  Green(int i = 22): g(i) {} //{g = i;}
  int getG() const {return g;}
  virtual int sum() const {return getR() + g;}
  int avg() const {return sum()/2;}
};
class Blue : public Green {
private:
  int b;
public:
  Blue(int i = 33): b(i) {} //{b = i;}
  int getB() const {return b;}
  virtual int RGB() const {return getR()*10000 + getG()*100 + b;}
};
```

For the next two questions, consider the following declaration:

```cpp
Green  g(44);
Green* gr = &g;
Blue*  bl = new Blue(66);
Green* gb = bl;
Red*   rd = gr;
```

8. [7 points] Can one make the following member invocation, (very briefly explain)? (If yes, give the output.)

```cpp
int i9 = gb->RGB();
cout << "i9 = " << i9 << endl;
```

No,

There is no RGB() member in Green or its base class Red.

The compiler will not even implement polymorphism/late binding during compilation if the class of the pointer type (or its base class) does not contain a virtual function prototype declaration.
9. [7 points] Can one make the following member invocation, (very briefly explain)? (If yes, give the output.)

```cpp
int i10 = bl->avg();
cout << “i10 = ” << i10 << endl;
```

Yes, Blue inherits `avg()`

\[ i10 = 49 \]

*The call to `avg()` is statically (early-binding) bound to `Green::avg()`, but within the function the call to the virtual `sum()` function is dynamically (late-binding) bound to the type of object being pointed to which is a blue object, thus the binding is made during execution to `Blue::sum()`.*

10. [7 points] Can one make the following member invocation, (very briefly explain)? (If yes, give the output.)

```cpp
int i11 = rd->sum();
cout << “i11 = ” << i11 << endl;
```

Yes, due to the over-riding

\[ i11 = 55 \]

*The call the virtual `sum()` function is dynamically (late-binding) bound to the type of object being pointed to which is a green object, thus the binding is made during execution to `Blue::sum()`.*
With the growth in specialized footwear in recent years software is needed for categorization. You have been selected to develop an inheritance hierarchy for the various types of footwear in order to minimize the code size for the system. The following various types of footwear and data must be modeled:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sandal</td>
<td>Slip-on shoe with a flat sole, open at toe and heel, held on with straps at toe and heel</td>
</tr>
<tr>
<td>Sneaker</td>
<td>Lace-up shoe with low synthetic upper and flat semi-rigid sole</td>
</tr>
<tr>
<td>Wingtip</td>
<td>Lace-up shoe with flat semi-rigid sole and decorative, pointed cap over toe</td>
</tr>
<tr>
<td>Duck boot</td>
<td>Waterproof lace-up shoe with high leather upper and flat lug sole</td>
</tr>
<tr>
<td>High heel</td>
<td>Slip-on shoe with open, elevated heel</td>
</tr>
<tr>
<td>Flip-flop</td>
<td>A flat sole that is held to the foot by straps over the toes</td>
</tr>
<tr>
<td>Hiker</td>
<td>Lace-up shoe with high leather upper and flat lug sole</td>
</tr>
<tr>
<td>Mule</td>
<td>Slip-on shoe with a closed toe and open heel, flat sole</td>
</tr>
</tbody>
</table>

11. [15 points] Considering only the possible types listed above, draw a sensible class relationship / inheritance hierarchy. You do not have to show any class members. (Hint: consider the information above and much less about any program being developed from the hierarchy. Be aware that base/intermediate types not explicitly listed above may need to be modeled for organizational purposes.)