2143 - OOP
Spring 2021
Take Home Exam
April 25, 2021

Name: Caleb Sneath

READ THESE INSTRUCTIONS
D Create a digital document (PDF) that has zero handwriting on it. Print and bring to the final exam on *Tuesday April 27th from 8:00 am - 10:00am*.
D Your presentation and thoroughness of answers is a large part of your grade. Presentation means: use examples when you can, graphics or images, and organize your answers!
D I make every effort to create clear and understandable questions. You should do the same with your answers.
D Questions should be answered in order and clearly marked.
D Your name should be on each page, in the heading if possible.
D Place your PDF on GitHub (after you take the actual final) and in your assignments folder.
D Create a folder called **TakeHomeExam** and place your document in there. Name the actual document: **exam.pdf** within the folder.
Failure to comply with any of these rules will result in a NO grade. This is a courtesy exam to help you solidify your grade.

Grade Table (don't write on it)

Question	Points	Score
1	70	
2	15	
3	40	
4	10	
5	15	
6	10	
7	10	
8	20	
9	15	
10	20	
11	35	
12	10	
13	10	
Total:	280	

Warning: Support each and every answer with details. I do not care how mundane the question is ... justify your answer. Even for a question as innocuous or simple as "What is your name?", you should be very thorough when answering:

What is your name?: My name is Attila. This comes from the ancient figure "Attila the Hun". He was the leader of a tribal empire consisting of Huns, Ostrogoths, Alans and Bulgars, amongst others, in Central and Eastern Europe. My namesake almost conquered western Europe, but his brother died and he decided to go home. Lucky for us! We would all be speaking a mix of Asiatic dialects :)

Single word answers, and in fact single sentence answers will be scored with a zero. This is a take-home exam to help study for the final and boost your grade. Work on it accordingly.

1. This VS That. Not so simple answers Ç:

(a) (7 points) Explain the difference between a *struct* and a *class*. Can one do whatever the other does?

Structs and classes are both programming tools that allow users to create and implement their own custom abstract data and encapsulate related methods. This does not mean they completely the same however. The main difference between the two is what access rights they assume. Classes assume their members are private access unless specified. This means that only objects of the same class have access to those members. Structs however are private. Anything can access all their data by default. Technically, both structs and classes have support for private and public members if they are specified. Aside from that, the original C language had support for simple structs. Some clever programmers have used this fact to create source code that could be compiled in both C++ as well as C, although doing so requires limiting the kind of code used, not adding anything unsupported by C within the struct itself, and formatting the rest of the code correctly.

So if they are so similar, does that mean they should just be used more or less at random according to a designer's whims, sticking private members in structs with no hesitation? The answer is no. If I told you I had this series of characters on my program… "myColor", there is a good chance you'd automatically assume it is a variable name, simply because of capitalization conventions. The compiler would give no trouble to naming a class myColor, but simply from the convention it conveys information about the code. The same is true for the use of classes and structs. Structs convey an intent by the designer that a data type is rather simple and plain. Classes on the other hand let it be known that a piece of code is likely more encapsulated and functional.

(b) (7 points) What is the difference between a *class* and an *object*?

Classes are merely like the blueprints that describe how objects should work and what members they will have. However, they have not actually been called and instantiated. Objects are actually instantiated and only based off of a class. They have memory allocated to them and change during the life of the program. Here is an example.

"

```
//This is the class
class Table
{
 public:
        Table()
        {
                int legs = 4;
        }
        int getLegs()
        {
                Return legs;
        }
 private:
        int legs;
}
“
void main()
{
 //This is the object
 Table myTabe();
 //This is the object doing something
 cout << myTable.getLegs();
}
```

(c) (7 points) What is the difference between *inheritance* and *composition*? Which one should you lean towards when designing your solution to a problem?

Inheritance and composition are relationships classes can have. Inheritance can be thought of as something “is a” something. Composition however is like when something “has a” something. For example, assume I made a parent class called RoundShape that handled a great deal of code associated with the handling of shapes that are round and was thinking about making a new class called Circle. A Circle is a RoundShape and could reuse a lot of its code. I may make Circle inherit from RoundShape and only need to add code for the ways Circles are distinct from all other round shapes. In contrast, a graph has a collection of vertices and edges. If I were to have or make code to handle vertices and edges as two different types of classes, I could and in fact have made a graph class contain objects that were of the type vertex and edge. This graph can be said to be “composed” of these other objects.

It’s generally a better idea to prefer implementing solutions that utilize composition instead of inheritance. Using inheritance, it is easy to get into the trap of overutilizing the protected status. This can result in a very high degree of coupling between a derived class and the parent class. Inheritance on the other end helps force the good habit of developing code to an interface rather than a solution. Additionally, in some cases, inheritance’s “is a” model can break down. This occurs whenever something is a particular class, but for some reason is a unique exception and does not possess an attribute all other members of the class does. Inheritance in this case requires the programmer to go through roundabout

ways to remove functionality from a derived class, while composition instead would have allowed the designer to just… not add that thing it doesn't have. For example, imagine a class was made to describe mammals, with derived classes for specific species. It may seem reasonable to add code that provides support for information about the gestation period of mammals to the parent class. A platypus however, is unique in that out of the many mammal types that exist, it is the only mammal to lay eggs instead of give live birth. The composition approach could simply have animal subtypes attach an object of the class type GestationInformation, and add it to all mammals but the platypus, rather than trying to override the parent attributes and create explicit exceptions in code for just the platypus.

(d) (7 points) What is the difference between a *deep* vs a *shallow* copy? What can you do to make one or the other happen?

Shallow copies are exact copies. This notably also includes copying stored memory addresses, such as pointers. A deep copy however will go ahead and check the values stored in those memory locations and copy them to a new memory address held by the copy. This is important because a simple copy can, intentionally or unintentionally, end up changing a stored value for the original as well as the copy if only one of the two changes it if this value is accessed by pointer. It's not usually necessary to create a deep copy. The compiler will usually automatically create a fairly good shallow copy without having to tell it how to do so if it needs to. The exception to this is when dynamic memory allocation and pointers are involved. In this case, you must specify how the program will create the deep copy, such as by overloading the "=" operator.

(e) (7 points) What is the difference between a *constructor* and a *destructor*? Are they both mandatory or even necessary?

At first glance, constructors and destructors both look somewhat similar syntactically, although they still have minor syntax differences. They both are named exactly after the class they are part of, although a destructor's name begins with "~" to specify it is the destructor. Neither method has an attached type or return statement, and additionally the destructor does not support any arguments. As such, while a class can support multiple constructors if they accept different parameters, a class will only have one destructor.

Although their syntax looks somewhat similar to other methods in a class aside from these minor differences, neither the constructor or destructor can be explicitly called. Instead, the constructor is called automatically at the creation of an object according to which parameters are passed in, while the destructor in contrast is called any times an object is deleted or goes out of scope.

Strictly speaking, specifying a constructor or destructor is not required as the compiler will automatically provide them for you if one is missing, however it may be a good idea to do so. Destructors rarely need to be explicitly defined, with the only general exception that they should be defined if an object is to make heavy use of pointers and dynamic memory

so that the destructor can define how to return memory to the system and avoid a memory leak. Constructors on the other hand are pretty useful for specifying the initial state of objects on their creation. Making any nondefault constructors for this purpose however requires a default constructor to be specified as well, even if the programmer doesn't end up using it in the program.

(f) (7 points) What is *static* vs *dynamic* typing? Which does C++ employ and which does Python employ?

Static vs dynamic refers to a mutually exclusive property of different programming languages. Static typing requires the programmer to specify the type of a variable explicitly. In contrast, with dynamic typing, the type of a variable does not need to be specified and is instead done automatically according to the context of the value it is created with. C++ and Java are examples of languages which employ static typing, while Python is a good example of a language which uses dynamic casting. Here is a simple example of how an integer is declared in C++ vs Python to illustrate.

C++:

```
int myNumber = 10;
```

Python:

```
myNymber = 10
```

(g) (7 points) What is *encapsulation* vs *abstraction*? Please give some examples!

I don't know much about the specifics of how electronic circuits work. This may seem almost impossible given that I study computers, devices which rely upon countless electronic circuits, however the reason this is possible is a concept called abstraction. Abstraction involves ignoring or hiding the irrelevant specifics in order focus only on the relevant details. In software design, abstraction is used as a design principle to roughly plan how a program will be created. Abstraction is often confused with encapsulation. While the two may sometimes overlap, they nonetheless have clear differences.

Encapsulation is an implementation concept of taking and bundling together a set of code related to some purpose, and then protecting it by making it so that outside pieces of code will only utilize the internals of the code through a set of predefined interfaces to protect and sometimes hide the internal state of the code. Here is a good example. Lets say I make a class to help model the inventory in a warehouse, and that inventory has a name, as well as a unique identifier number which is generated based upon that name. I could start by adding all the methods I believe would be useful when adding inventory. Then, it would be important to ensure that the name is not changed without also changing the identifier number, since that is dependent upon the name. In order to do this, both the name and identifier are set to private, and a method is written to properly handle changing a name so that the identifier also changes called setName(string newName). In this case, it

is pretty clear. As the designer of this class, I have not hidden concept from myself about it, since I designed it and know how it works. I have however, encapsulated the code into a single entity capable of handling it.

(h) (7 points) What is the difference between an *abstract class* and an *interface*?

In OOP, interfaces describe what objects can do. This idea can be extended to create an interface class. Interface classes are similar to abstract classes, or even a type of abstract class. The distinction is that while an abstract class has at least one pure virtual method and may have one or more normal methods as well, an interface will only have pure virtual methods. No method within an interface class will have any implementation, and much like abstract classes, interfaces can not directly be used to create objects.

The underlying concept behind both is polymorphism. It may be obvious what behavior and methods objects derived from a parent class should be capable of, however many derived classes of the parent may need vastly different code to achieve such a result. For example, a parent class called Shape may make sense to add a function called getArea() that returns the area of the shape. In fact, now that I'm proofreading my answers, some code on a later question did essentially that. Of course, every shape will require a different formula in order to calculate its area even if they all can return their area somehow, and can fill out the implementation in its own derived class. In turn, code outside doesn't need to bother knowing what type of derived shape it will be getting the area for in advance to know it can call the getArea() function. On the other hand, it is important to be able to reuse code when possible, so defining implementations for a parent class can be helpful. This however, creates another issue, the possibility of inheritance issues. As such, a class which will be involved in a great deal of inheritance without much potential to reuse code is best implemented as an interface class. In contrast, when inheritance issues are less likely and it is desirable to reuse code by defining implementations in the parent class, abstract classes with one or more implementations for a method is preferable.

(i) (7 points) What is the difference between a *virtual function* and a *pure virtual function*?

Virtual functions are an important way to implement run-time polymorphism. Virtual functions are when a method of a parent class can be overridden by its derived classes. This does not mean that this must be used, as a virtual function still contains an implementation, allowing the parent class it is part of to still be called to create a fully functional object if no pure virtual functions are used as well. In contrast, a pure virtual function is used to create abstract or interface classes. They have no implementation at all, and must somewhere down the line be overridden by a derived class in order to be able to create an object.

Example:

```
class Shape{
public:
```

```
~Shape();
int height, area;

//This is a virtual function
virtual int getHeight( )
{
  return height;
}

  //This is a pure virtual function
  virtual int getArea( ) = 0;
  };
```

(j) (7 points) What is the difference between *Function Overloading* and *Function Overriding*?

Function overloading and function overriding are both ways of achieving polymorphism in a program, however they differ in type. Function overloading is a way of achieving compile time polymorphism. It is achieved by defining and calling a function with the same name, but a different list of parameters. The compiler simply links the line of code to the corresponding function according to the parameters passed in where it gets called. In contrast, function overriding achieves runtime polymorphism. A derived class uses its own implementation for a function instead of its parents. Look at the above example in part (i). If a derived class called Ellipse is created with Shape as its parent and implements its own method for getHeight(), such as if it also stored a rotation for the ellipse and needed to return the height for a given rotation, it would be overriding its parent class when this function is called.

---

2. Define the following and give examples of each :

(a) (5 points) Polymorphism

Polymorphism is when the same form can give different behavior in certain circumstances. In OOP, it is used to create code that appears to behave more naturally and intuitively for given inputs. For example, a function can be overloaded creating compile time polymorphism in order appropriately handle inputs, such as treating a floating point or integer parameter different, or a function overridden so that a derived class can perform a function more appropriate for it creating runtime polymorphism.

(b) (5 points) Encapsulation

Encapsulation is the bundling together of related piece of code that serves a single purpose. Proper class design in C++ is a good example. A class exhibits a good degree of encapsulation if it has a clear purpose, properly protects with access restrictions its attributes in order to preserve its integrity, and has public functions that clearly establish how it is supposed to communicate with outside code in order to serve its purpose. Encapsulation is primarily associated with the implementation of code.

(c) (5 points) Abstraction

Abstraction is simply filtering out unnecessary details to efficiently think about a subject. For example, a project lead does not always need to know the specifics about how computers work, or how the team members he manages will eventually decide to specifically implement his plan in order to design the general blueprint for how a program will work. It is already enough that he knows what purpose the program as a whole must serve, what general parts to it must exist, and how he can divide those certain parts of the program between team members to iron out the details.

---

3. (a) (5 points) What is a default constructor?

Default constructors run when no parameter is passed when creating a class. It is like the established default protocol for how an object of a certain class should be created. If no default constructor exists and the user has not created nondefault constructors, the compiler will attempt to automatically generate one for you.

(b) (5 points) What is an overloaded constructor? And is there a limit to the number of overloaded constructors you can have?

An overloaded constructor is a nondefault constructor which accepts a set of parameters. This is useful because it allows you to easily support selecting the initial state of your objects, like the initial dimensions in a constructor for a shape class. The only major limit to the number of overloaded constructors is the fact that they all must have separate parameter lists. Otherwise, the programmer can include however many they desire.

(c) (5 points) What is a copy constructor? Do you need to create a copy constructor for every class you define?

Copy constructors define how to initialize an object using another object of the same class. It is not always needed as the compiler will attempt to generate a copy constructor for you which creates a shallow copy of the object. However, if the programmer wishes to obtain a deep copy, such as when pointers or dynamic memory allocation is used, they must explicitly design one. Copy constructors have this general syntax:

```
class exampleClass
{
 public:
        //This is the copy constructor
        exampleClass(const point &rhs)
       {//Internal copy code goes here
        }

}
```

(d) (5 points) What is a deep copy, and when do you need to worry about it?

Deep copies are copies which properly handle dynamic memory allocation and pointers. In a simple shallow copy, issues can arrise because everything is copied perfectly, including memory addresses. This leads to issues where changing the values stored at that address by the original or the copy will effect the value for both. Deep copies get around this by copying the values stored at these locations to a new address. If dynamic memory and pointers aren't involved, you are free to simply use a shallow copy, which the compiler will generally provide.

(e) (5 points) Is there a relationship between copy constructors and deep copying?

Yes, compiler generated copy constructors can cause issues when deep copying is needed. The compiler automatically creates a shallow copy constructor if none is explicitly created by the programmer. As such, whenever dynamic memory allocation and pointers are involved and a programmer wants the values of the copy and original object to stay separate, a programmer needs to design a copy constructor.

(f) (5 points) Is a copy constructor the same as overloading the assignment operator?

Not quite. When written to work on an object, the assignment operator will replace the values of an already existing object. In contrast, a copy contstructor is used to create an entirely new objects. Additionally, the methods to go about setting up support for the two are not the same. A copy constructor must be set up like in the answer to 3(c), whereas the assignment overload looks like this:

```
exampleClass& operator = (const exampleClass& rhs)
{
//Self assignment check
//Internal code
Return *this;
}
```

(g) (10 points) Give one or more reason(s) why a class would need a destructor.

Destructors are necessary when dealing with pointers and dynamic memory allocation. If memory gets dynamically allocated without a destructor being defined and designed properly by the programmer, a memory leak will occur when the object gets deleted or leaves scope. As such, destructors should be made and filled with the corresponding code to delete dynamically allocated variables whever they are used.

---

4. (10 points) What is the difference between an abstract class and an interface?

Hint:
You should include in your discussion:

- Virtual Functions
- Pure Virtual Functions

In OOP, interfaces describe what objects are capable of doing. This idea can be extended to create an interface class. Interface classes are similar to abstract classes, and even a type of abstract class. The distinction is that while an abstract class has at least one pure virtual method and may have one or more normal methods as well, an interface will only have pure virtual methods. No method within an interface class will have any implementation, and much like abstract classes, interfaces can not directly be used to create objects because they are exclusively pure virtual functions.

The underlying concept behind both is polymorphism. It may be obvious what behavior and methods objects of a parent class should be capable of, however many derived classes of the parent may need vastly different code to achieve such a result. For example, a parent class called Shape may make sense to add a function called getArea() that returns the area of the shape. Of course, every shape will require a different formula in order to calculate its area even if they all can return their area somehow, and can fill out the implementation in its own derived class. In turn, code outside doesn't need to bother knowing what type of derived shape it will be getting the area for in advance to know it can call the getArea() function. On the other hand, it is important to be able to reuse code when possible, so defining implementations for a parent class can be helpful. This however, creates another issue, the possibility of inheritance issues. As such, a class which will be involved in a great deal of inheritance without much potential to reuse code is best implemented as an interface class. In contrast, when inheritance issues are less likely and it is desirable to reuse code by defining implementations in the parent class, abstract classes with one or more implementations for a method is preferable, including the use of virtual methods, which can be functional methods but later overridden by a derived class if necessary.

---

5. Describe the following (make sure you compare and contrast as well):

(a) (5 points) Public

Public members are those that will be accessible to everything even outside the object as if it is a normal variable or function. Member with public access rights is similar to the default behavior of structs. It has no access protection and is declared in a class or struct code block with the private: keyword, as well as during inheritance, although not to escalate access rights. For example, “setter” and “getter” methods are customary public methods to allow outside code to interact with other more private methods in a controlled manner. These methods are called outside the object, and the object responds accordingly by changing or returning the corresponding attributes.

(b) (5 points) Private

Private members are those that are restricted to only be usable by the specific object itself. Code outside the class is not able to access or change any private code within an object of a class. Not even different objects of a derived class can interact with it. There is one exception to this, a method can be specified as a friend to be allowed access to a private method. Private access can be a key aspect in encapsulation, as it helps protect and preserve the internal state of an object’s variables from being improperly modified. By default, all members of a class are private unless specified. It can be declared by writing “private: “, as well as during inheritance to hide members.

(c) (5 points) Protected

Protected access means that only members of the base class as well as any derived classes have access to an object of the class’s members. Additionally, access can be granted to other functions by declaring them as a “friend” function. No other code can interact with the protected code. It can be useful when heavily using inheritance with the intent that subclasses be able to function similar to the base class. It can be declared by writing “private: “, as well as during inheritance to hide members, although not to escalate access rights. Here is an example.

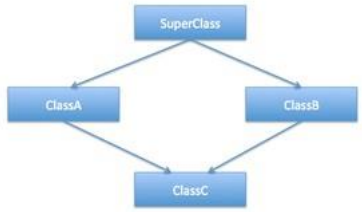
```
class ExampleClass2 : protected ExampleClass
{
//Internals
}
…
ExampleClass2 example2Object;
```

example2Object will have access to all the public and protected members of ExampleClass.

Hint:

- Make sure you define each item individually as well.
- Use examples.
- If your not sure, use examples to make your point.
- Ummm, example code is always welcome.

6. (10 points) What is the diamond problem?



Hint:

- This is a question about multiple inheritance and its potential problems.
- Use examples when possible, but explain thoroughly.

The diamond problem is an issue that can arise when dealing with multiple inheritance. It occurs when a class, lets call it class Square, inherits from two classes, which we will call Rectangle and Trapezoid, which in turn inherited from a common parent class, which we will call Quadrilateral and imagine has the method getSides() which just returns the integer four. It is named as it is because the corresponding inheritance tree looks like a diamond. The issue is that Square will try to inherit Quadrilateral's members such as getSides() from both Rectangle and Trapezoid, but won't know which to inherit from.

This can be solved in C++ by using virtual inheritance like this:

```
class Rectangle: virtual Quadrilateral{… };
class Trapezoid: virtual Quadrilateral {… };
class Square: Rectangle, Trapezoid {… };
```

This lets the compiler know to treat Quadrilateral directly as the parent class that Square gets its getSides() function from, instead of trying to hand it two copies of getSides() from both Trapezoid and Rectangle.

---

7. (10 points) Discuss Early and Late binding.

Hint:

- These keywords should be in your answer: **static, dynamic, virtual, abstract, interface**.
- If you haven't figured it out .... use examples.

Early and late binding are concepts related to polymorphism. Early binding, also known as static binding, is when the compiler pairs a function call and its definition during compilation. This can be used to achieve compile time polymorphism by overloading functions, as the compiler will simply pair a function with the corresponding definition for the passed parameters.

Late binding, also known as dynamic binding, is when the program only matches the correct definition for a function call while the program is running. This can be used to achieve runtime polymorphism. The method to achieve this is with the use of virtual functions. Labeling something as a virtual function lets the compiler know to perform late binding and override it with the correct function at runtime. Abstract classes and interfaces are good examples of this. An abstract class has at least one pure virtual function that doesn't even have an

implementation and as such can't even produce an object outside of inheritance. Interface classes take this to an extreme however, being composed entirely of pure virtual functions to be overridden later.

---

8. (20 points) Using a **single** variable, execute the show method in *Base* and in *Derived*. Of course you can use other statements as well, but only one variable.

```
class Base{
        public:
        virtual void show() { cout<<" In Base n"; }
};

class Derived: public Base{
        public:
        void show() { cout<<"In Derived n"; }
};
```

Hint: This is implying that dynamic binding should be used. A pointer to the base class can be used to point to the derived as well.

```
void main()
{
Base *singleVar = new Derived;
singleVar->show();
singleVar = new Base;
singleVar->show();
}
```

---

9. (15 points) Given the two class definitions below:

```
class Engine {} // The Engine class. class Automobile {} // Automobile class
which is parent to Car class.
```

You need to write a definition for a Car class using the above two classes. You need to extend one, and use the other as a data member. This question boils down to composition vs inheritance. Explain your reasoning after your write you Car definition (bare bones definition).

```
class Car: public Automobile
{
  private:
        Engine thisEngine;
        //Other private members can go here.
  public:
        //Public members can go here.
};
```

I chose to make Car a derived class of Automobile and composed of an engine. My reasoning for this is that it fit pretty neatly into the "is a" vs "has a" questioning to do it like this. A car is an automobile that has an engine. As such, its general behavior for whatever program this would be a part of would probably best be modeled like how Automobile is modeled, with minor tweaks as necessary.

---

10. (20 points) Write a class that contains two class data members *numBorn* and *numLiving*. The value of *numBorn* should be equal to the number of objects of the class that have been instanced. The value of *numLiving* should be equal to the total number of objects in existence currently (i.e., the objects that have been constructed but not yet destructed.)

```
class Num
{
static int numBorn;
static int numLiving;
public:
   Num()
   {
            numLiving++;
            numBorn++;
   }
   ~Num()
   {
            numLiving--;
   }
}
int Num::numBorn = 0;
int Num:: numLiving = 0;
```

---

11. (a) (10 points) Write a program that has an abstract base class named *Quad*. This class should have four member data variables representing side lengths and a *pure virtual function* called *Area*. It should also have methods for setting the data variables.

(b) (15 points) Derive a class *Rectangle* from *Quad* and override the *Area* method so that it returns the area of the Rectangle. Write a main function that creates a Rectangle and sets the side lengths.

(c) (10 points) Write a top-level function that will take a parameter of type *Quad* and return the value of the appropriate Area function.

Note: A **top-level function** is a function that is basically stand-alone. This means that they are functions you call directly, without the need to create any object or call any class.

```
#include <iostream>
```

```
using namespace std;

class Quad
{
public:
	void setSideA(int input)
	{
		sideA = input;
	}
	void setSideB(int input)
	{
		sideB = input;
	}	void setSideC(int input)
	{
		sideC = input;
	}	void setSideD(int input)
	{
		sideD = input;
	}
	virtual int Area() = 0;
protected:
	int sideA, sideB, sideC, sideD;
	friend int areaQuad(Quad&);
};

class Rectangle : public Quad
{
protected:
	friend int areaQuad(Quad&);
	int Area()
	{
		return sideA * sideB;
	}
public:
	void createRectangle(const int length, const int width)
	{
		sideA = sideC = length;
		sideB = sideC = width;
	}
};

int areaQuad(Quad& inQuad)
{
	return inQuad.Area();
}


int main()
```

```
{
        Rectangle rec1 = Rectangle();
        rec1.createRectangle(5, 5);
        cout << areaQuad(rec1) << endl;


        return 0;
}
```

---

12. (10 points) What is the rule of three? You will have answered this question (in pieces) already, but in the OOP world, what does it mean?

The rule of three in regards to object oriented programming makes a prescription based upon three possible parts of a class. These three parts are the destructor, the copy constructor, and the copy assignment operator. The rule states that if a class has one of these explicitly designed, it should probably have all three. The reason for this is that the most common reason for creating any one of these is the same reason they all are created.

But first, why are any one of these generally created? The compiler will generally automatically create simple versions of all of these for you. For example, when trying to make use of the copy constructor or copy assignment operator, the compiler will try to automatically make you a shallow copy. In general the compiler does a pretty good job of this, with one exception. When pointers and dynamically allocated memory are involved, the compiler generated versions are usually insufficient for all three of these things. It will fail to make a deep copy for the copy constructor and copy assignment operator, and it will allow a memory leak unless the programmer defines a proper destructor for it. It is for this reason that these three parts are a good deal to implement together if any are to be used.

---

13. (10 points) What are the limitations of OOP?

Object oriented programming is a broad programming paradigm with three main focuses: encapsulation, inheritance, and polymorphism. The general idea is to bundle together and protect the state of related pieces of code focused on a singular purpose into an abstract data type which communicates with other code based upon signals it receives. The first main criticism of OOP is that this may not be necessary in smaller programs. For a very small program that doesn't need the use of any abstract data types, this can be too much, and waste massive amounts of time overengineering a simple problem.

The second main limitation with OOP, is that it is a broad programming paradigm with those three main focuses. Each of those three focuses bring numerous benefits, but they also require sacrifices to obtain. Encapsulation can be a wonderful idea that creates a powerful piece of code capable of accurately achieving its purpose in an organic way, without the need for others to totally understand the mechanics of said code, only how interact with it. But it can also obfuscate and hide critical underlying issues when said code goes wrong. Inheritance can lead to less code repetition and better polymorphism, but it can also lead to

ambiguity, complex code, and issues like the diamond problem. Polymorphism can make code respond dynamically to different possible states, however this by definition makes it at odds with the concept of creating deterministic and immutable programs. As such, the complex relationships that can emerge within OOP can lead to unobvious causes of failure as well as missed potentials for optimization. As such, some programs even outside of small ones may benefit more from a diametrically opposed coding paradigm to OOP such as functional programming.

All of this assumes that everyone will receive all of these benefits from the use of OOP. In reality OOP is just as much a theory as it is a skill that takes a great deal of study and practice to develop and use properly. While an experienced OOP designer has a major advantage creating massive and modular projects, an amateur is quite liable to instead create massive, unmanageable, and highly coupled code by focusing too heavily on classes.