2143 - OOP
Spring 2021
Take Home Exam
April 22, 2021

Name: Loic Konan

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 **Take Home Exam** 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 (do not 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 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:

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

- A **structure** is a value type, so it is stored on the stack, but a **class** is a reference type and is stored on the heap.
- By default, all the **struct** members are ***public,*** but **class** members are by default ***private*** in nature.
- A **class** rises the level of **abstraction** much more than a **struct**.
- The implementation of a **structure** is not secure and does not hide the ***complexity*** of the implementation from the user.
- In a **struct** there is no relation between its ***members***.

  - In conclusion, you can do anything in **struct** that you would do in a **class**.
    The only technical difference is that if you do not say anything about scoping in a **struct**, it is going to be ***public*** whereas what is in a **class** is going to be ***private***.
    Same goes for ***inheritance***: if a **struct** inherits from something else and you do not specify if it is a ***public*** or ***private*** inheritance the ***inheritance*** is ***public***. And for a **class** it would be ***private***.

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

- A **class** is a template for ***creating*** an **object** within a program while an **object** is an ***instance*** of a **class**.
- A **class** is a ***logical*** entity while **Object** is a ***physical*** entity.
- A **class** does not get any ***memory*** when it is created whereas **objects** get ***memory*** when they are created.
- A **class** is declared once while multiple **objects** are created using that **class**.

(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** expresses *a **is-a relationship***, while **composition** expresses a **has-a relationship** between the two classes.
- **Composition** is usually used for wrapping classes and to express relationships between classes that contain one another.
- **Inheritance** is used for ***polymorphism***, where you have a base class, and you want to extend or change its functionality.
- **Inheritance** means inheriting something from a parent. For example, you may inherit your mother's skin tone or inherit your father's teeth.
- **Inheritance** means deriving properties, characteristics, and behaviors from a parent class.

  - Example: Car **has** Engine (Composition), and Car **is a** (Inheritance) Automobile

  - In conclusion, if you are designing a solution to a problem, you should lean toward **composition**. Composition put things together and prevent complexity.
    But **Inheritance** creates dependency between child and parent, when a class **inherit** another class, we include all methods and attributes from parent class, and expose to the child class, therefore we break the encapsulation, the child object can access all the methods in parent object and overwrite them.

(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 Copy:** Copy of the reference to A into B. Think about it as a copy of A's Address. So, the addresses of A and B will be the same i.e., they will be pointing to the same memory location i.e., data contents.
- **Deep copy:** Copy of all the members of A, allocates memory in a different location for B and then assigns the copied members to B to achieve deep copy. In this way if A becomes non-existent B is still valid in the memory.

- **Deep Copy** you will need to utilize pointers and dynamic allocated memory.
- **Shallow Copy** you will just copy all member field values and not mess with pointers or Dynamic memories.

  - In conclusion, to make one the other you can just use pointers and dynamic memory for **Deep Copy** or not use memory for **Shallow Copy**.

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

- **Constructor** helps to initialize the object of a class.
- **Destructor** is used to destroy the instances.
- In a **class**, there can be multiple **constructors**.
- **Constructor** can be overloaded.
- **Destructor** it cannot be overloaded.
- The **constructor's** name is same as the class name.
- **Destructor** name is also same as the class name preceded by tiled **(~)** operator.
- In a class, there can be multiple **constructors**.
- In a class, there is always a single **destructor**.

  - In conclusion, no they are not both mandatory:

- if you ***dynamically*** allocate memory, and you want this memory to be ***deallocated*** only when the object itself is "terminated", then you need to have a **destructor**.
- if you want to ensure that the ***instantiated object*** is initialized with certain data then you should have a **default Constructor**.
- A default **constructor** will only be automatically generated by the compiler if no other **constructors** are defined. Regardless of any inheritance.
- The default **constructor** is generated only if you have not defined any other **constructors**.

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

- **Dynamically typed** languages perform type checking at runtime, while **statically typed** languages perform type checking at compile time.
- **Static typing** has to do with the explicit declaration (or initialization) of variables before they are employed.
- **Static-typing** languages require you to declare the data types of your variables before you use them, while **dynamically** typed languages do not.
- **Dynamic typed** languages do not require the explicit declaration of the variables before they are used.
    - Python is an example of a dynamic typed programming language.

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

- **Abstraction** handle complexity by hiding unnecessary details from the user. That enables the user to implement more complex logic on top of the provided abstraction without understanding or even thinking about all the hidden complexity.
- Data **Abstraction** is the property by virtue of which only essential details are displayed to the user. The trivial or the non-essentials units are not displayed to the user.
    - Example: A car is viewed as a car rather than its individual components.

- **Encapsulation** describes the idea of bundling data and methods that work on that data within one unit. This concept is also often used to hide the internal representation, or state, of an object from the outside. This is called information hiding.

❖ Example: **Encapsulation** could be like a capsule. Basically, capsule encapsulates several combinations of medicine. If combinations of medicine are variables and methods, then the capsule will act as a class and the whole process is called **Encapsulation**.

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

- An **abstract** class is a class that cannot be instantiated and is usually implemented as a class that has one or more **pure virtual functions**.
- An **abstract** class is used to define an implementation and is intended to be inherited from by concrete classes.
- An **interface** class has no implementation.
- An **interface** class contains only a virtual destructor and pure virtual functions.
- An **interface** class is a class that specifies the polymorphic interface i.e., pure virtual function declarations into a base class.

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

- A **virtual function** is a member function in the base class that we expect to redefine in derived classes.
- A **virtual function** is used in the base class to ensure that the function is overridden.
- A **pure virtual function** is a virtual function which has no definition.
- **Pure virtual functions** are also called abstract functions.
- To create a **pure virtual function**, we assign a value 0 to the function.

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

- **Overriding** of functions occurs when one class is inherited from another class.
- **Overloading** can occur without inheritance.

- **Overloaded** functions must differ in function signature i.e., either number of parameters or type of parameters should differ.
- In **overriding**, function signatures must be same.
- **Overridden** functions are in different scopes, whereas overloaded functions are in same scope.
- **Overriding** is needed when derived class function must do some added or different job than the base class function.
- **Overloading** is used to have same name functions which behave differently depending upon parameters passed to them.

2. Define the following and give examples of each:

(a) (5 points) Polymorphism

- Polymorphism means having many forms.
- The same entity (function or operator) behaves differently in different situations.

- Example: A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee This is called polymorphism.

- Example: The + operator in C++ is used to perform two specific functions. When it is used with numbers it performs addition, but when used with string it performs concatenation.

When it is used with numbers it performs addition	**when used with string it performs concatenation**

int a = 4;
int b = 3;
 	
int sum = a + b; // sum = 7
	string firstName = "LOIC ";
	string LastName = "Konan";
	
	string name = firstName + LastName; // LOIC Konan

(b) (5 points) Encapsulation

- **Encapsulation** describes the idea of bundling data and methods that work on that data within one unit. This concept is also often used to hide the internal representation, or state, of an object from the outside. This is called information hiding.

- In C++, **encapsulation** helps us keep related data and functions together, which makes our code cleaner and easy to read

  - Example: **Encapsulation** could be like a capsule. Basically, capsule encapsulates several combinations of medicine. If combinations of medicine are variables and methods, then the capsule will act as a class and the whole process is called **Encapsulation**.

  - Snippet code for Encapsulation.

```
class Rectangle
{
  private:
    int age;

  public:
    void setLength(int len)
{
      if (len >= 0)
        length = len;
    }
};
```

(c) (5 points) Abstraction

- Data **Abstraction** is the property by virtue of which only essential details are displayed to the user. The trivial or the non-essentials units are not displayed to the user.
- Can change internal implementation of class independently without affecting the user.
- Helps to increase security of an application or program as only important details are provided to the user.

- Example: A car is viewed as a car rather than its individual components.

- Snippet code for abstraction.

```
#include <iostream>
Using namespace std:
class implementAbstraction
{
  private:
    int a, b;

  public:

    void set(int x, int y)                    // Method to set values of private members
    {
      a = x;
      b = y;
    }
    void display()
    {
      cout << “a = “ << a << endl;
      cout << “b = “ << b << endl;
    }
};
```

3.

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

- A **default constructor** is a constructor that either has no parameters, or if it has parameters, it must have default values in the parentheses.
- If a **default constructor** is not provided by the programmer explicitly, then the compiler provides an implicit default constructor. In that case, the default values of the variables are 0.

```
class X
{
 public:
   X();                    // Default constructor with no arguments
   X(int = 0);             // Default constructor with one default argument
 };
```

(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** essentially have the same name of the class and different number of arguments.
- In C++, we can have more than one **constructor** in a class with same name, as long as each has a different list of arguments, which will be call an **Overloaded Constructor**.

  - There is no limit to the number of **overloaded constructors** that you can have.

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

- A copy constructor is a member function that initializes an object using another object of the same class which has been created previously.
- **Copy Constructor** is used to :
  - Initialize one object from another of the same type.
  - Copy an object to pass it as an argument to a function.
  - Copy an object to return it from a function.

- No, you do not need to create a copy constructor for every class you define, you should only define a **copy constructor** only if an object has pointers or any runtime allocation

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

- **Deep copy,** copy of all the members of A, allocates memory in a different location for B and then assigns the copied members to B to achieve deep copy. In this way if A becomes non-existent B is still valid in the memory.
  - **You** will need to worry about utilizing pointers and dynamic allocated memory.

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

- To make a **deep copy**, you must write a **copy constructor** and overload the assignment operator, otherwise the copy will point to the original, with disastrous consequences.
  - So, copy constructors and deep copying have a relationship, because without making a copy constructor your deep copy will not work..

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

- **Copy constructor** creates a new object which has the copy of the original object .
- On the other hand, an **assignment operator** does not create any new object. It instead, deals with existing objects.
- For the **assignment operator** to be used to create deep copy and store separate pointer, we have to check the *self-reference* otherwise the assignment operator may change the value of current object.

    - So, they are not the same.

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

- A class would need a **Destructor** to *avoid memory leak*; because *memory leaked* occurs if no-longer-needed dynamic memory is not freed. The memory is unavailable for reuse within the program.
- **Destructor** is used to ***deallocated memory*** as well.

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

    - An **abstract** class is a class that cannot be instantiated and is usually implemented as a class that has one or more **pure virtual functions**.
    - An **abstract** class is used to define an implementation and is intended to be inherited from by concrete classes.
    - An **interface** class has no implementation.
    - An **interface** class contains only a virtual destructor and pure virtual functions.

- An **interface** class is a class that specifies the polymorphic interface i.e., pure virtual function declarations into a base class.

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

(a) (5 points) Public

- Access Modifiers.
- The **public** keyword is used to create public members (data and functions).
- All the class members declared under the public specifier will be available to everyone.
- The data members and member functions declared as public can be accessed by other classes and functions too unlike the Private access Modifiers
- The **public** members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

❖ Snippet code for **Public** access modifier example.

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

// define a class
class Sample {

    // public elements
   public:
    int age;

    void displayAge() {
        cout << "Age = " << age << endl;
    }
};

int main() {

    // declare a class object
    Sample obj1;

    cout << "Enter your age: ";

    // store input in age of the obj1 object
    cin >> obj1.age;

    // call class function
    obj1.displayAge();

    return 0;
}
```

- Notice that the public elements are accessible from main(). This is because public elements are accessible from all parts of the program.

(b) (5 points) Private

- Private Access Modifier.
- The private keyword is used to create private members (data and functions).
- The private members can only be accessed from within the class.
- However, friend classes and friend functions can access private members.

- Snippet code for **Private** access modifier example.

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

// define a class
class Sample {

    // private elements
   private:
    int age;

    // public elements
   public:
    void displayAge(int a) {
        age = a;
        cout << "Age = " << age << endl;
    }
};

int main() {

    int ageInput;

    // declare an object
    Sample obj1;

    cout << "Enter your age: ";
    cin >> ageInput;

    // call function and pass ageInput as argument
    obj1.displayAge(ageInput);

    return 0;
}
```

- In main(), the object obj1 cannot directly access the class variable age.

(c) (5 points) Protected -

- Protected Access Modifier.
- The protected keyword is used to create protect4ed members(data and function).
- The protected members can be accessed within the class and from the derived class.
- Protected elements are just like the private, except they can be accessed by derived classes.
  - Snippet code for **Protected** access modifier example.

main.cpp

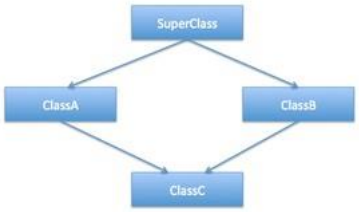
```cpp
#include <iostream>
using namespace std;

// declare parent class
class Sample {
    // protected elements
  protected:
    int age;
};

// declare child class
class SampleChild : public Sample {

  public:
    void displayAge(int a) {
        age = a;
        cout << "Age = " << age << endl;
    }

};

int main() {
    int ageInput;

    // declare object of child class
    SampleChild child;

    cout << "Enter your age: ";
    cin >> ageInput;

    // call child class function
    // pass ageInput as argument
    child.displayAge(ageInput);

    return 0;
}
```

- Here, SampleChild is an inherited class that is derived from Sample. The variable age is declared in Sample with the protected keyword.

- This means that SampleChild can access age since Sample is its parent class.

- ❖ We see this as we have assigned the value of age in SampleChild even though age is declared in the sample class.

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



- ➢ The **diamond problem** is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If there is a method in A that B and C have overridden, and D does not override it, then which version of the method does D inherit: that of B, or that of C?

- ➢ This will cause an error due to the "**Diamond Problem**" of multiple inheritance.

- ❖ This conundrum is solved by using **a Virtual inheritance**, it *solves* the classic "**Diamond Problem**". It ensures that the child class gets only a single instance of the common base class

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

Hint:

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

- ➢ **Binding** means matching the function call with the correct function definition by the compiler. It takes place either at **compile time or runtime**.

- **Early and Late binding** deals mainly with **Compile-time polymorphism** (**Early Binding**) and Run-time polymorphism (**Late Binding**).

- In **Early Binding,** the compiler matches the function call with the correct function definition at **compile time.** This is also known at **Static Binding** or Compile time Binding.

- **Late Binding** primarily deals with **Virtual Functions**. This is because since **Late Binding**, also sometimes known as **Dynamic Binding** or **Run-Time polymorphism**.

- In **late binding**, the compiler identifies the type of object at runtime and then matches the function call with the correct function definition.

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.

main.cpp

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

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

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

int main() {
Base *A;
Derived D;
A = &D;
A -> show();

    return 0;
}
```

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 you write your Car definition (bare bones definition).

- A car "is an" automobile so it can inherit from automobile (Is a relationship).
- A car "HAS-A" engine and therefore engine should be a class data member of my car class.

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

class Engine{
  int cylinders, pistons, fuelType;
};

class Automobile{
protected:
int tires, doors, window;

Automobile(){
  int tires, door = 4, window = 4;
}
};

class Car: public Automobile{
  protected:
  Engine Bugatti;
  string Brand, Model;
  double price;
  public:
  Car();
  ~Car();
};

int main() {
    return 0;
}
```

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.)

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

class Something {
    public:
        static int numBorn;
        int numLiving = 0;

        Something() {      // Constructor definition
            numBorn++;
        }
};
int Something::numBorn = 0;

int main(void) {

  Something obj, obj2,obj3;
  obj.numLiving++;


  cout << "Total numBorn: " << Something::numBorn << endl;
  cout << "Total numLiving: " << obj.numLiving << endl;
  return 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{
protected:
        double height, width;
public:
    virtual double Area() = 0;
   void setSides(double h, double w){
      height = h;
      width = w;
    }
};

class Rectangle :public Quad {
public:
    double Area(){
     return height * width;
    }
};

double get_Area(Quad *quad){
  return quad->Area();
}
int main(){
  Rectangle rect;
  rect.setSides(4, 5);
  cout << "Area of Rectangle: " <<
  get_Area(&rect) << 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?

Rule Of Three in C++, basically states that if a class defines one (or more) of the following, it should probably explicitly define all three, which are:

- **Destructor.**
- **Copy Construtor.**
- **Copy Assignment Operator.**

- The **default constructors** and **assignment operators** do shallow copy, and we create our own constructor and assignment operators when we need deep copy (For example when class contains pointers pointing to dynamically allocated Memory)

- A **destructor** contains code that is supposed to run whenever an object is destroyed. To only affect the contents of the object would be useless. An object in the process of being destroyed cannot have any changes made to it. Therefore, the destructor affects the program's state as a whole.

- Now, supposing our class does not have a **copy constructor**. Then copying an object will copy all of its data members to the target object. What happens when these objects are destroyed? The **destructor** runs twice.

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

I would not necessarily say it has limitations, but I believe it has some disadvantages:

- **Size:** Object Oriented Programs are much larger than other programs.
- **Effort:** Object Oriented Programs require a lot of work to create.
- **Speed:** Object Oriented Programs are slower than other programs, because of their size.

- **Steep learning curve:** The thought process involved in OO programming may not be natural for some people, and it will take the time to get used to it.

- **The complexity of creating programs:** it is overly complex to create programs based on the interaction of objects. Some of the key programming techniques, such as inheritance and polymorphism, can be a big challenging to comprehend initially.