Segundo Certamen

Primera Parte (Sin Apuntes, 30 minutos, 1/3 puntaje): (5 pts cada una, el total se multiplica por 34/40)

1. ¿Cuál es la relación entre clases protegidas, privadas y el calificador Friend? Dé un ejemplo simple.
   La relación es que una clase o función global pueden acceder a atributos o métodos privados o protegidos de una clase cuando ésta ha declarado como friend a esa clase o función.
   Ejemplo:
   ```
   class A {
   private:
     int i_privado;
   public:
     friend void foo(A &a);
   …...
   };
   ```
   Luego en foo podemos acceder a ese atributo privado.
   ```
   void foo(A &a) {
    a.i_privado++;
   …..
   }
   ```
2. Explique y dé ejemplos simples del uso de Static en Datos y Funciones.
   Un atributo estático es aquel compartido por todos los miembros de la clase. Puede ser entendido como un atributo de la clase (o categoría de objetos) y no de un objeto o instancia particular.
   Una función o método estático son aquellas comunes a todos las instancias de una clase y no dependen de atributos específicos de una instancia. Pueden manipular sólo atributos estáticos. Tanto los datos (atributos) como las funciones (métodos) estáticas pueden ser accedidas con el nombre de la clase o un el de un objeto.
   Nota: Para obtener máximo puntaje basta con mencionar algunos de los elementos previos.
   Ejemplo: caso de atributo estático y función estática.
   ```
   class A {
   private:
     static int count;
   public:
     static int getActiveInstances();
   ….
   };
   ```
   Luego en la implementación:
   ```
   int A::count=0;
   int A::getActiveInstances() {
     return count;
   }
   ```
3. Muestre una implementación jerárquica de clases en C++, use una clase Animales como base y genere clases derivadas a partir de Animales.
   Hay múltiples soluciones, no se especifica un mínimo. La correcta creación de la clase Animales y

dos derivadas de ella basta. Una solución posible (más completa) es la siguiente.

```
class Animales {
private:
    double peso;
public:
    Animales (double pesoNacer);
    virtual void Display() const;
};
Animales::Animales(double pn) {
  peso = pn;
}
Animales::Display() const {
  cout <<  “Peso:” << peso << endl;
}
class Gato:public Animales {
private:
    string nombre;
public:
   Gato(double p, string n);
   virtual void Display() const;
};
Gato::Gato(double p, string n):Animales(p){
  nombre=n;
}
void Gato::Display() const {
   cout << nombre << “es un Gato”<< endl;
   Animales::Display();
}
class Perro:public Animales {
private:
   string: raza;
public:
  Perro(double p, string r);
  virtual void Display() const;
};
Perro::Perro(double p, string r):Animales(p) {
  raza = r;
}
Perro::Display() const {
   cout << “Perro de raza:”<<raza<<endl;
   Animales::Display();
}
```

4. Describa los calificadores de acceso usados en las clases derivadas. Justifique por qué se usaron en su ejemplo de la pregunta anterior.
   Si entendemos esta pregunta como los calificadores de acceso usados en los métodos y atributos de las clase derivadas, éstos son los mismos que pueden usarse en clases bases.
   En las clases previas he usado los calificadores public y private. Miembros públicos de una clase pueden ser accedidos desde cualquier parte del código. Miembros privados sólo son accesibles desde implementaciones de la misma clase.

   Si entendemos la pregunta como los calificadores de acceso usados al crear una clase derivada, éste es sólo public en este caso. Esto significa que la visibilidad de los métodos de la clase Animales se mantiene para instancias de Gato o Perro.
   Nota: una de las dos interpretaciones es suficiente para acreditar puntaje completo.

5. Muestre el uso de un constructor con parámetros y explique la diferencia con un constructor por omisión. Haga que su constructor inicialice algún atributo en la clase base Animales.
   Muestra y ejemplo de constructor con parámetros:

```
class Animales {
private:
   double peso;
public:
   Animales (double pesoNacer);
   virtual void Display() const;
};
Animales::Animales(double pn) {
  peso = pn;
}
```

   Este constructor difiere del por omisión porque este último no tiene parámetros. Se le llama por omisión pues ponemos no ponerlo. En este caso, el compilador crea uno y nos permite crear objetos.

6. Utilice y describa el uso de mutadores y accesores al hacer una instanciación de jerarquía de animales.
   Un mutador es un método que modifica el estado de un objeto (cambia alguno de sus atributos). Un accesor es un método que sólo accede a valores sin modificar el estado del objeto.
   En el ejemplo de Animales Displaye es un método accesor.

7. ¿Cuál es la diferencia entre funciones out of line e in line? ¿Por qué la diferencia?
   La diferencia sintáctica está en su implementación. Si se hace directamente en la declaración de la clase se dice in-line. Si se hace en un fuera de la declaración de la clase se dice out-of-line.
   Las funciones in-line permiten optimizar el código generado cuando las implementaciones son de pocas líneas.
   Nota: en caso de código in-line, el compilador no genera salto a la implementación de la función, sino inserta su implementación en el lugar donde el método es llamado.

8. ¿Qué es sobre carga de operadores? Dé un ejemplo de uso en su jerarquía de Animales.
   La sobrecarga de operadores es la definición que hacemos en una clase para usar instancias de la clase como operandos de algunos operadores.
   Ejemplo: Si deseamos comparar Animales según su peso, podemos dar sentido al operador < agregando como método de la clase:
   bool operator<(const Animales&a) const;
   Y podemos implementarlo comparando su peso:
   bool Animales::operator<(const Animales &a) const {
   return peso < a.peso;
   }

Segunda Parte (Con Apuntes, 60 minutos, 2/3 puntaje):

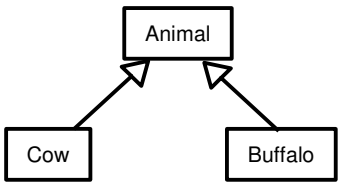
9. Declare una clase base Animal con atributos privados (e.g. int color, double weight)
Declare una clase Cow derivada de Animal que tenga métodos para eat() y milk(). Estos métodos tienen que impactar el peso (de instancias) de la clase Cow de manera de que eat() cause que el peso se incremente y milk() cause que el peso baje. La clase Cow adicionalmente tiene que tener una variable que indique la cantidad de leche que tiene inicialmente, ésta se configura en el constructor. También se tiene que tener un método Display() que imprima todos sus contenidos.
Declare una clase Buffalo derivada de Animal que implemente eat() pero no implemente milk(). Aparte de los atributos de la clase tiene que tener int horn_length con un valor configurado inicialmente por el constructor. También tiene que tener un método Display() que imprima todos sus contenidos.
Implemente un main que utilizando las clases anteriormente descritas cree varios cows y buffaloes. Haga que éstos coman y se les saque leche utilizando milk() en el caso de las cows.
Muestre un diagrama UML de su jerarquía de clases.

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

/* declaración de clase 5 pts. */
class Animal {
private:
  int color;      // debe ir, se pide
  double weight;  // debe ir, se pide
protected:
  void addWeight(double w) { weight+=w;}  // o public, se debe agregar
  void reduceWeight(double w);          // para cambiar weight (privado).
  virtual void Display();  // para imprimir atributos privados, hay otras opciones.
public:
  Animal(int c, double w);  // Uso parámetros para dar valor a atributos.
  virtual void eat()=0;   // Por simplicidad, aumentará el peso en uno.
                    // virtual es importante.
};
/**  implementaciones Animal 5 pts **/
Animal::Animal(int c, double w){
  color=c;
  weight=w;
}
```

```
void Animal::reduceWeight(double w){
  weight-=w;
  if (weight<0) weight=0;
}
void Animal::Display(){
  cout << "Color: " << color << endl;
  cout << "Weight: " << weight << endl;
}

/* definición de clase Cow  5 pts */
class Cow:public Animal {
public:
 Cow(int c, double w, double iMilk); // milk debe ir.
  virtual void eat(); // otras declaraciones son posibles
  void milk();   // otras declaraciones son posibles
  virtual void Display();
private:
  double milk_storage;
};

/* Implementaciones de Cow 5 pts */
Cow::Cow(int c, double w, double initM): Animal(c,w){
 milk_storage = initM;
}
void Cow::eat(){
  addWeight(1);
}
void Cow::milk(){
  reduceWeight(1);
  milk_storage--;
  if (milk_storage<0) milk_storage=0;
}
void Cow::Display(){
  cout << "Soy una Cow" << endl;
  Animal::Display();
  cout << "Disponibilidad de Leche: " << milk_storage<< endl;
}

/*Declaración clase Buffalo 5 pts */
class Buffalo:public Animal {
public:
  Buffalo(int c, double w, int hl);
  virtual void eat();
  virtual void Display();
private:
  int horn_length;
};

/* Implementaciones de Buffalo 5 pts */
Buffalo::Buffalo(int c, double w, int hl): Animal(c,w){
  horn_length=hl;
}
void Buffalo::eat(){
  addWeight(1);
}
void Buffalo::Display(){
```

```
  cout << "Soy un buffalo" << endl;
  Animal::Display();
  cout << "Largo del cacho : " << horn_length<< endl;
}
/* Main 5 pts */
int main(){  // la pregunta no es clara aquí sobre cómo y cuántos crear.
  Cow cow(1,6,5);  // Aquí se crea uno de cada uno, podrían ser más.
  cow.Display();
  cow.eat();
  cow.Display();
  cow.milk();
  cow.Display();
  Buffalo buf(2,7,3);
  buf.Display();
}
```

UML de clases involucradas en este problema: **10 pts**.



10. Muestre el código para hacer una lista con cows para ordenar esta lista. ¿Es posible hacer una lista que contenga cows y buffaloes? Si es posible, ¿cómo se podría hacer?

Cows se pueden insertar en una lista; sin embargo, para ordenarlas es necesario implementar el operador < en cows. Para esto incluir como método público en Cow:

```
class Cow:public Animal {
public:
  Cow(int c, double w, double iMilk);
  virtual void eat();
  void milk();
  void Display();
   bool operator <(const Cow & c) const;  // 4 pts.
private:
  double milk_storage;
};
```

Su implementación puede ser (ordenando por peso, ordenar por cantidad de leche es OK):

```
bool Cow::operator<(const Cow &c) const {  // 4 pts.
   return getWeight()<c.getWeight();
}
```

Luego podemos hacer un main que ordene cows: // 5 pts.

Aquí se pide insertar cows en la lista e invocar a sort() correctamente. Listas antes y después podría omitirse.

```
int main(){
  list<Cow> l;
  Cow c1(1,7,3);
  Cow c2(1,5,3);
  Cow c3 (1,6,3);
```

```
  l.push_back(c1);
  l.push_back(c2);
  l.push_back(c3);
  list<Cow>::iterator pos;
  pos = l.begin();
  while( pos != l.end()){
   (*pos).Display();
   pos++;
  }
  l.sort();
  cout << "ORDENADOS" << endl;
  pos = l.begin();
  while( pos != l.end()){
   (*pos).Display();
   pos++;
  }
}
```

¿Es posible hacer una lista que contenga cows y buffaloes?

Sí. **4 pts.**

¿Cómo? Haciendo una lista de punteros a Animal (como en la tarea 4). **4 pts.**

Si no pone código ejemplo, -1 pts.

```
int main(){
  Cow c1(1,7,3);
  Cow c2(1,5,3);
  Cow c3 (1,6,3);

  list<Animal* > la;
  Buffalo b1(1,4,2);
  la.push_back(&c1);
  la.push_back(&b1);
  la.push_back(&c2);
  la.push_back(&c3);
  list<Animal*>::iterator posa;
  posa = la.begin();
  while( posa != la.end()){
   (*posa)->Display();
   posa++;
  }
}
```