Instituto Superior de Formación Técnica Nº 151
Carrera: Analista en Sistemas
Asignatura: Algoritmos y Estructuras de Datos I
Docente: Lic. José Oemig

Simulación Final A	
Modalidad: Semipresencial	**Estrategia Didáctica:** Individual
Metodología de Desarrollo: Det. docente	**Metodología de Corrección:** Det. docente
Carácter del Trabajo: Obligatorio – Con Nota	**Fecha de Entrega:** Det. docente
Alumno/a: Bravo, David Hernan	

Marco Teórico:

1. Desarrollar y Relacionar los conceptos de Problema, Algoritmo y Programa.

Problema:

- **Definición:** Un problema es una situación que requiere una solución. En programación, un problema es una tarea específica que queremos que la computadora realice.
- **Ejemplo:** Calcular el promedio de un conjunto de números, ordenar una lista de nombres alfabéticamente, o simular el movimiento de un objeto en una pantalla.

Algoritmo:

- **Definición:** Un algoritmo es una secuencia finita de pasos bien definidos que conducen a la solución de un problema. Es como una receta, pero para resolver problemas computacionales.
- **Características:** Debe ser claro, preciso, finito y efectivo.
- **Ejemplo:** Para calcular el promedio, sumaríamos todos los números y dividiríamos por la cantidad de números.

Programa:

- **Definición:** Un programa es la implementación de un algoritmo en un lenguaje de programación específico. Es el código fuente que la computadora puede ejecutar.
- **Componentes:** Un programa en C++ típicamente incluye variables, tipos de datos, operadores, estructuras de control (condicionales, bucles), funciones y entrada/salida.

Relación entre los tres conceptos:

- **Problema → Algoritmo → Programa:**
  - Identificamos un problema a resolver.
  - Diseñamos un algoritmo que describa los pasos para resolverlo.
  - Codificamos el algoritmo en C++ para crear un programa ejecutable.

2. Relacionar Problema, Abstracción y Algoritmia.

Problema: Es la situación o pregunta que buscamos resolver. Puede ser desde una simple operación matemática hasta un complejo sistema de inteligencia artificial. Es el punto de partida de todo proceso de programación.

Abstracción: Consiste en simplificar la realidad, enfocándonos en los aspectos relevantes del problema y dejando de lado los detalles innecesarios. Es como tomar una fotografía aérea de un paisaje: vemos la forma general, pero no los detalles de cada hoja de un árbol. En programación, la abstracción nos permite crear modelos más manejables y comprensibles del problema.

Algoritmia: Es el arte de diseñar secuencias de pasos lógicos y precisos para resolver un problema. Un algoritmo es como una receta: una serie de instrucciones detalladas que, si se siguen correctamente, conducen a un resultado específico.

¿Cómo se relacionan estos conceptos?

- **Del problema a la abstracción:** Cuando enfrentamos un problema, el primer paso es comprenderlo a fondo. La abstracción nos ayuda a identificar los elementos esenciales del problema y a descartar aquellos que no son relevantes para la solución.
- **De la abstracción al algoritmo**: Una vez que tenemos una visión abstracta del problema, podemos comenzar a diseñar el algoritmo. La abstracción nos proporciona una base sólida para definir los pasos necesarios para resolver el problema.
- **Del algoritmo al programa:** El algoritmo es la representación lógica de la solución. Al implementarlo en un lenguaje de programación (como C++), obtenemos un programa que la computadora puede ejecutar.

Ejemplo:

Imaginemos que queremos crear un programa que ordene una lista de números de menor a mayor.

- **Problema:** Ordenar una lista de números.
- **Abstracción:** Nos enfocamos en la acción de comparar números y en la idea de intercambiarlos de posición si están en el orden incorrecto.
- **Algoritmo:** Podríamos utilizar el algoritmo de burbuja, que consiste en comparar pares de números adyacentes y cambiarlos de posición si están en el orden equivocado. Repetimos este proceso hasta que la lista esté completamente ordenada.

3. ¿Qué entiende por Arreglo y Matriz? Dar un ejemplo aplicado de la Realidad.

   Arreglos y **matrices** son estructuras de datos que nos permiten almacenar múltiples valores del mismo tipo en una sola variable. Un arreglo es como una lista ordenada de elementos, mientras que una matriz es como una tabla donde los elementos se organizan en filas y columnas.

   Ejemplo: Imagina una lista de compras. Un arreglo podría almacenar los nombres de los productos, y una matriz podría almacenar información más detallada como el nombre del producto, la cantidad y el precio. Esto nos permitiría realizar operaciones como calcular el total de la compra o buscar un producto específico.

4. ¿Qué características Tiene un TAD? ¿Qué representa?

   Un **Tipo Abstracto de Datos (TAD)** es una herramienta fundamental en programación que nos permite modelar conceptos del mundo real de una manera más abstracta y manejable. En lugar de preocuparnos por la implementación interna de una estructura de datos, nos enfocamos en su comportamiento y en las operaciones que se pueden realizar sobre ella.

   Características principales de un TAD:

   - **Abstracción:** Oculta la complejidad interna de la implementación, presentando al usuario una interfaz simple y fácil de usar.
   - **Encapsulación:** Agrupa los datos y las operaciones que se pueden realizar sobre esos datos en una única unidad. Esto protege los datos de modificaciones no intencionales y facilita la reutilización del código.
   - **Modularidad:** Los TADs son independientes y pueden ser utilizados en diferentes partes de un programa, promoviendo la modularidad y la organización del código.
   - **Reutilización:** Al definir un TAD, se crea una plantilla que puede ser utilizada en múltiples proyectos, ahorrando tiempo y esfuerzo.

   ¿Qué representa un TAD?

   Un TAD representa un concepto o entidad del mundo real, como un número, una lista, una pila, una cola, un árbol, etc. Por ejemplo, un TAD "Pila" representaría una estructura de datos donde los elementos se añaden y se eliminan siguiendo el principio LIFO (Last In, First Out).

Ejemplo:
Imagina un TAD "CuentaBancaria". Este TAD podría tener atributos como "saldo" y "titular", y operaciones como "depositar", "retirar" y "consultarSaldo". Al utilizar este TAD, no necesitamos preocuparnos por cómo se almacena internamente el saldo o cómo se realiza un depósito, simplemente utilizamos las operaciones proporcionadas por el TAD.

5. ¿Qué diferencia hay en C++ entre una Estructura (Struct) y una Clase (Class)?

Tanto las **estructuras** (structs) como las **clases** (classes) son herramientas fundamentales para agrupar datos y funciones relacionadas. Aunque comparten muchas similitudes, existen algunas diferencias clave en su comportamiento por defecto:

Diferencias Principales:

Visibilidad por defecto:

- **Structs**: Los miembros de una estructura son públicos por defecto. Esto significa que se puede acceder a ellos directamente desde cualquier parte del código donde la estructura sea visible.
- **Classes:** Los miembros de una clase son privados por defecto. Esto encapsula los datos y los métodos, permitiendo un mayor control sobre el acceso y la modificación de los mismos.

Concepción:

- **Structs:** Históricamente, se utilizaban principalmente para agrupar datos simples.
- **Classes:** Se concibieron para modelar objetos del mundo real con mayor complejidad, incluyendo datos y comportamientos (métodos).

Similitudes:

- **Miembros:** Ambas pueden contener miembros de datos (variables) y miembros de función (métodos).
- **Herencia:** Tanto structs como classes pueden heredar de otras clases o structs, permitiendo la creación de jerarquías de clases.
- **Constructores y Destructores:** Ambas pueden tener constructores y destructores para inicializar y liberar recursos asociados a un objeto.
- **Sobrecarga de Operadores:** Se pueden sobrecargar operadores para definir un comportamiento personalizado para objetos de structs y classes.

¿Cuándo usar una u otra?

Structs:

- Para agrupar datos simples sin una gran cantidad de comportamiento asociado.

- Cuando se necesita un acceso público directo a los miembros.

Classes:

- Para modelar objetos del mundo real con un comportamiento más complejo.
- Cuando se requiere encapsulación y control sobre el acceso a los datos.
- Cuando se necesita herencia y polimorfismo.

6. Clasifique y Explique los distintos tipos de Memoria Vistos (que usa un programa de C++).

Los programas utilizan diferentes áreas de memoria para almacenar distintos tipos de datos durante su ejecución. Cada área tiene características y propósitos específicos. A continuación, se detallan los principales tipos de memoria:

1. **Memoria de Código:**

- **Contenido:** Contiene el código ejecutable del programa, es decir, las instrucciones que la CPU ejecutará.
- **Características:** Es de solo lectura y su tamaño es fijo desde que se carga el programa en memoria.

2. **Memoria de Datos:**

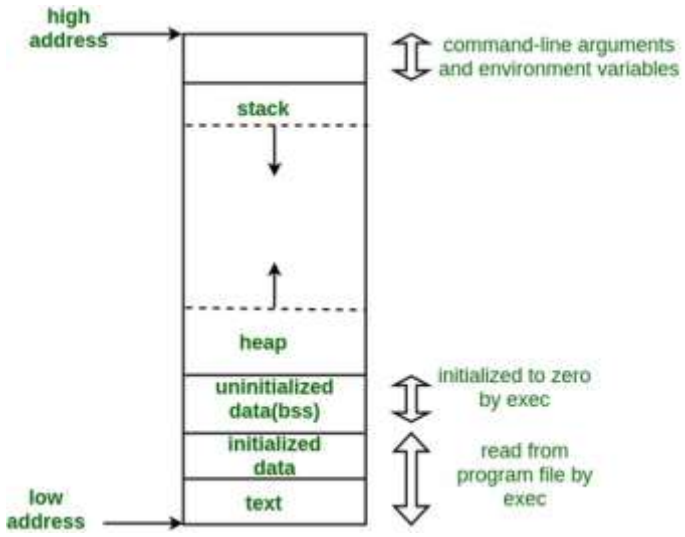
- **Memoria Estática:**
  - **Variables globales:** Declaradas fuera de cualquier función, su vida útil es todo el programa.
  - **Variables estáticas:** Declaradas dentro de una función, pero con la palabra clave static, conservan su valor entre llamadas a la función.
  - **Características:** Se inicializan al inicio del programa y permanecen en memoria hasta que este finaliza.

- **Pila (Stack):**
  - **Variables locales:** Declaradas dentro de una función.
  - **Parámetros de funciones:** Valores pasados a las funciones.
  - **Valores de retorno de funciones.**
  - **Características:** Se asigna y libera memoria de forma automática cuando se entra y sale de un bloque de código. Su tamaño es limitado y crece y decrece de manera LIFO (Last In, First Out).

- **Montículo (Heap):**
  - **Memoria dinámica:** Asignada y liberada explícitamente por el programador utilizando operadores como new y delete.
  - **Características:** Es más flexible que la pila, pero requiere una gestión más cuidadosa por parte del programador para evitar fugas de memoria.

Resumen de las Características de Cada Tipo de Memoria:

Tipo de Memoria	Alcance	Vida útil	Gestión de memoria
Código	Global	Todo el programa	Automática
Datos Estáticos	Global o local (con static)	Todo el programa	Automática
Pila (Stack)	Local	Dentro del bloque de código	Automática
Montículo (Heap)	Global	Hasta que se libera explícitamente	Manual (programador)



¿Cuándo utilizar cada tipo?

- **Memoria de código:** Para almacenar las instrucciones del programa.
- **Memoria estática:** Para variables que deben persistir durante todo el programa o para variables locales que necesitan conservar su valor entre llamadas a una función.
- **Pila:** Para variables locales y parámetros de funciones, ya que su vida útil está estrechamente relacionada con el flujo de control del programa.
- **Montículo:** Para estructuras de datos de tamaño variable o para cuando la cantidad de memoria necesaria no se conoce de antemano.

Consideraciones importantes:

- **Desbordamiento de pila:** Ocurre cuando se asigna demasiada memoria en la pila, lo que puede causar un error de segmentación.
- **Fugas de memoria:** Se producen cuando se asigna memoria en el montón y no se libera correctamente, lo que puede llevar a un agotamiento de la memoria.

- **Eficiencia:** La gestión de la memoria puede tener un impacto significativo en el rendimiento de un programa.

7. ¿Qué diferencia tiene la Memoria Stack (Pila) y el Heap (montón)? ¿Para que se utilizan? Dar ejemplo de un programa que lo consuma.

   La pila y el montón son dos regiones de memoria fundamentales en C++ que se utilizan para almacenar datos durante la ejecución de un programa. Cada una tiene características y propósitos distintos:

   Pila (Stack)

   Características:

   - **Tamaño fijo:** Su tamaño está predefinido y puede variar ligeramente durante la ejecución, pero no de forma significativa.
   - **Asignación automática:** El compilador gestiona la asignación y liberación de memoria de forma automática.
   - **LIFO (Last In, First Out):** El último elemento en ser agregado es el primero en ser eliminado.
   - **Almacenamiento de datos locales:** Variables locales, parámetros de funciones y direcciones de retorno se almacenan en la pila.

   - **Uso:**

     - **Almacenamiento de datos temporales** que son necesarios durante la ejecución de una función o bloque de código.
     - **Gestión de llamadas a funciones:** Al llamar a una función, se crea un nuevo marco de pila para almacenar los parámetros, variables locales y la dirección de retorno.

   Montón (Heap)

   Características:

   - **Tamaño dinámico:** Su tamaño puede crecer y decrecer durante la ejecución del programa.
   - **Asignación manual:** El programador debe gestionar la asignación y liberación de memoria utilizando los operadores new y delete.
   - **No hay orden específico:** Los datos se almacenan en cualquier lugar disponible.

- Almacenamiento de datos de larga duración: Objetos que necesitan persistir más allá de la vida de una función o que tienen un tamaño variable.

- **Uso:**

    - **Creación de objetos dinámicos:** Cuando se necesita crear objetos en tiempo de ejecución sin conocer su tamaño de antemano.
    - **Gestión de estructuras de datos complejas:** Listas enlazadas, árboles, etc.
    - **Pasaje de grandes objetos por valor a funciones.**

Ejemplo:

```
#include <iostream>

using namespace std;

int main()
  {
    // Variables en la pila.
    int x = 10;
    double y = 3.14;

    // Arreglo dinámico en el montón.
    int *array = new int[5]; // Asignación de memoria en el montón.

    for (int i = 0; i < 5; ++i)
      {
        array[i] = i * 2;
      }

    // Uso del arreglo.
    cout << "El segundo elemento del arreglo es: " << array[1] << endl;

    // Liberación de la memoria del montón.
    delete[] array;

    return 0;
  }
```

Explicación:

- x e y son variables locales almacenadas en la pila.
- array es un puntero a un arreglo de enteros. El arreglo en sí se crea en el montón utilizando new.
- Al final del programa, se libera la memoria del montón utilizando delete[].

Resumen de las Diferencias

Características	Pila	Montón
Tamaño	Fijo	Dinámico
Asignación	Automático	Manual
Orden	LIFO	No hay orden especifico
Uso típico	Variables locales, parámetros de funciones	Objetos dinámicos, estructuras de datos complejas.

¿Cuándo utilizar cada uno?

- **Pila:** Para datos de vida corta y tamaño conocido, como variables locales y parámetros de funciones. Es más eficiente y segura, ya que la gestión de memoria es automática.
- **Montón:** Para datos de vida larga, tamaño variable o cuando se necesita flexibilidad en la gestión de la memoria. Sin embargo, requiere una gestión cuidadosa para evitar fugas de memoria.

8. ¿Qué es un Puntero? ¿Qué utilidad tiene?

Un puntero es una variable que almacena la dirección de memoria de otro valor. En vez de contener el valor directamente, un puntero "apunta" a la ubicación donde se encuentra ese valor en la memoria. Esto permite un acceso más directo y flexible a los datos, así como la creación de estructuras de datos más complejas y eficientes. Los punteros son fundamentales para tareas como la gestión de memoria dinámica, la creación de estructuras de datos enlazadas y la implementación de algoritmos eficientes.

9. ¿Por qué debemos Liberar recursos (Delete) en la memoria Dinámica?

Liberar recursos (utilizando delete) en la memoria dinámica es esencial para evitar fugas de memoria. Cuando se asigna memoria dinámicamente (con new), esta permanece ocupada hasta que se libera explícitamente. Si no se libera, el programa consume cada vez más memoria, lo que puede llevar a un rendimiento lento, inestabilidad y, en casos extremos, al bloqueo del sistema. Liberar la memoria cuando ya no es necesaria garantiza un uso eficiente de los recursos y evita problemas de gestión de memoria.

10. Dar ejemplos de Punteros a Arrays, Estructuras y Objetos.

Punteros a Arreglos

- **¿Por qué utilizarlos?**
    - Pasar arreglos a funciones como argumentos.
    - Acceder a elementos de un arreglo de forma más eficiente.
    - Manipular arreglos dinámicos.
- **Ejemplo:**

```
include <iostream>

using namespace std;

int main()
   {
      int numeros[5] = {1, 2, 3, 4, 5};
      int *puntero = numeros; // Puntero apunta al primer elemento de numeros.

      // Accediendo al segundo elemento.
      cout << *(puntero + 1) << endl; // Imprime 2.

      // Modificando el primer elemento.
      *puntero = 10;
      cout << numeros[0] << endl; // Imprime 10.

      return 0;
   }
```

Punteros a Estructuras

- **¿Por qué utilizarlos?**
    - Pasar estructuras a funciones como argumentos.
    - Retornar estructuras de funciones.
    - Crear listas enlazadas y otras estructuras de datos.
- **Ejemplo:**

```
#include <iostream>

using namespace std;

struct Persona
   {
      string nombre;
      int edad;
   };

int main()
```

```
{
    Persona persona1 = {"Juan", 30};
    Persona *punteroPersona = &persona1;

    // Accediendo a los miembros de la estructura.
    cout << punteroPersona->nombre << endl; // Imprime "Juan".
        return 0;
}
```

Punteros a Objetos

- **¿Por qué utilizarlos?**
    - Pasar objetos a funciones como argumentos.
    - Retornar objetos de funciones.
    - Implementar polimorfismo.
- **Ejemplo:**

```
#include <iostream>

using namespace std;

class Coche
  {
    public:
      string marca;
      int modelo;
  };

int main()
  {
    Coche coche1;
    Coche *punteroCoche = &coche1;

    // Accediendo a los miembros del objeto.
    punteroCoche->marca = "Volkswagen";

    return 0;
  }
```

Consideraciones importantes:

- **Operador de indirección (*):** Se utiliza para acceder al valor al que apunta un puntero.
- **Operador de dirección (&):** Se utiliza para obtener la dirección de memoria de una variable.
- **Aritmética de punteros:** Se pueden sumar o restar enteros a un puntero para moverlo a diferentes posiciones en la memoria.

- **Punteros nulos:** Un puntero nulo no apunta a ninguna dirección de memoria válida. Es común inicializar punteros a nulo antes de asignarles una dirección.
- **Punteros colgantes:** Un puntero colgante apunta a una ubicación de memoria que ya ha sido liberada. Esto puede causar errores de segmentación.

11. ¿Qué entiende por Modularización? Dar ejemplos.

La modularización es la práctica de dividir un programa complejo en partes más pequeñas y manejables, llamadas módulos o funciones. Cada módulo se encarga de una tarea específica y bien definida, lo que facilita la escritura, lectura y mantenimiento del código. Por ejemplo, en lugar de tener un gran bloque de código que realiza todas las operaciones de un programa, podemos crear funciones separadas para calcular el área de un círculo, ordenar una lista de números o realizar una búsqueda en una base de datos. Esto hace que el código sea más organizado, reutilizable y fácil de depurar. Imagine un rompecabezas: cada pieza (módulo) contribuye a la imagen completa, pero puede ser entendida y manipulada por separado.

Ejemplos de modularización:

- **Funciones:** Cada función encapsula una tarea específica y puede ser llamada desde diferentes partes del programa.
- **Clases:** Las clases agrupan datos (atributos) y funciones (métodos) relacionadas, creando módulos más complejos y reutilizables.
- **Namespaces:** Los namespaces organizan el código en diferentes espacios de nombres para evitar conflictos de nombres y mejorar la legibilidad.
- **Archivos de cabecera (.hpp) y archivos de implementación (.cpp):** Separar las declaraciones de las definiciones de funciones y clases en archivos diferentes promueve la modularidad y la organización del código.

12. ¿Qué entiende por Clases y Objetos?

Definición de Clase

Una **clase** es como un plano o plantilla que define las características y comportamientos de un conjunto de objetos. En otras palabras, es una descripción abstracta de un tipo de entidad. Una clase especifica:

Atributos: Las propiedades o características que identifican a los objetos de esa clase. Por ejemplo, en una clase Persona, los atributos podrían ser nombre, edad y altura.

Métodos: Las acciones u operaciones que los objetos de esa clase pueden realizar. Por ejemplo, en la clase Persona, los métodos podrían ser caminar(), hablar() o comer().

Definición de Objeto

Un **objeto** es una instancia concreta de una clase. Es como crear una copia de ese plano o plantilla. Cada objeto tiene sus propios valores para los atributos definidos en la clase. Por ejemplo, si Persona es una clase, Juan y María podrían ser objetos de esa clase, cada uno con sus propios nombres, edades y alturas.

Analogía:

Imaginemos que una clase es una receta para hacer galletas. La receta especifica los ingredientes (atributos) y los pasos a seguir (métodos). Cada galleta que horneamos a partir de esa receta es un objeto. Todas las galletas tendrán las mismas características básicas (definidas por la receta), pero cada una será ligeramente diferente (por ejemplo, en tamaño o forma).

Ejemplo:

```
#include <iostream>

using namespace std;

class Persona
  {
    public:
      string nombre;
      int edad;

      void hablar()
        {
          cout << "Hola, me llamo " << nombre << endl;
        }
  };

int main()
  {
    Persona persona1;
    persona1.nombre = "David";
    persona1.edad = 49;
    persona1.hablar(); // Imprime: Hola, me llamo David.
  }
```

En este ejemplo:

- Persona es la clase.
- nombre y edad son los atributos.

- hablar() es un método.
- persona1 es un objeto de la clase Persona.

13. Explicar los conceptos de Abstracción, Encapsulamiento, Herencia y Polimorfismo.

Abstracción:

La abstracción consiste en centrarse en las características esenciales de un objeto o sistema, ignorando los detalles de implementación. Es como crear un modelo simplificado de un objeto del mundo real. En programación, la abstracción se logra definiendo clases que capturan los atributos y comportamientos más relevantes de un objeto, sin revelar los detalles internos de cómo se implementan esos comportamientos. Por ejemplo, al definir una clase Coche, podemos abstraer las características como color, marca y modelo, sin preocuparnos por los detalles mecánicos del motor.

Encapsulamiento:

El encapsulamiento es el mecanismo que oculta los detalles internos de un objeto, protegiendo sus datos y métodos de accesos no autorizados. Los datos de un objeto se almacenan en variables miembro, que pueden ser declaradas como privadas, protegidas o públicas, controlando así el nivel de acceso. Los métodos proporcionan una interfaz para interactuar con el objeto, permitiendo modificar o acceder a sus datos de forma controlada. El encapsulamiento promueve la modularidad y la seguridad del código.

Herencia:

La herencia es un mecanismo que permite crear nuevas clases (subclases o clases derivadas) a partir de clases existentes (superclases o clases base). La subclase hereda todos los atributos y métodos de la superclase, además de poder agregar nuevos atributos y métodos o redefinir los heredados. La herencia establece una jerarquía de clases, donde las subclases son especializaciones de las superclases. Esto fomenta la reutilización de código y la creación de jerarquías de clases que reflejan relaciones "es un" (por ejemplo, un Perro es un Animal).

Polimorfismo:

El polimorfismo es la capacidad de que objetos de diferentes tipos puedan ser tratados como si fueran del mismo tipo. Esto se logra mediante la sobrecarga de métodos y la

sobreescritura de métodos. La sobrecarga de métodos permite definir múltiples métodos con el mismo nombre, pero con diferentes parámetros, mientras que la sobreescritura permite que una subclase redefina un método heredado de la superclase. El polimorfismo permite escribir código más flexible y adaptable, ya que permite tratar a objetos de diferentes clases de forma genérica.

14. Explicar los Factores de Calidad.

Los factores de calidad en relación a las clases en programación orientada a objetos se refieren a las características que hacen que una clase sea bien diseñada, eficiente y fácil de mantener. Estos factores incluyen la **cohesión** (que una clase se enfoque en una única responsabilidad), el **acoplamiento** (la dependencia de una clase con otras), la **abstracción** (la capacidad de ocultar detalles internos), el **encapsulamiento** (la protección de los datos internos), la **herencia** (la capacidad de crear nuevas clases a partir de clases existentes), el **polimorfismo** (la capacidad de que objetos de diferentes clases se traten como si fueran del mismo tipo), y la **reusabilidad** (la capacidad de utilizar una clase en diferentes contextos). Una clase bien diseñada es aquella que cumple con estos principios, lo que resulta en un código más limpio, organizado y mantenible.

15. ¿Qué es una Clase Abstracta y para qué sirve?

Una clase abstracta es una clase que no puede ser instanciada directamente, es decir, no puedes crear objetos de esa clase. Su propósito principal es servir como una plantilla o base para otras clases.

¿Por qué usar clases abstractas?

a. **Definición de interfaces**: Una clase abstracta define un contrato o interfaz que las clases derivadas deben cumplir. Es decir, establece los métodos que todas las clases hijas deben implementar.
b. **Polimorfismo:** Las clases abstractas son fundamentales para lograr el polimorfismo, permitiendo que objetos de diferentes clases derivadas sean tratados como si fueran de la clase base abstracta.
c. **Abstracción:** Al definir una clase abstracta, te concentras en las características esenciales de un concepto, sin entrar en los detalles de implementación específicos.

Características clave de las clases abstractas:

a. **Métodos puros virtuales:** Una clase abstracta debe contener al menos un método puro virtual. Un método puro virtual se declara con la sintaxis `virtual tipo_de_retorno nombre_metodo() = 0;`. Este método no tiene una implementación en la clase base y debe ser implementado por las clases derivadas.
b. **No se pueden instanciar:** No puedes crear objetos directamente de una clase abstracta.
c. **Sirven como base para otras clases:** Las clases abstractas son utilizadas como base para crear clases más específicas, que heredan los métodos y atributos de la clase abstracta.

Ejemplo:

```
#include <iostream>

using namespace std;

class Animal
   {
      public:
         virtual void hacerSonido() = 0; // Método puro virtual.
   };

class Perro : public Animal
   {
      public:
         void hacerSonido() override
            {
               cout << "Guau!" << endl;
            }
   };

class Gato : public Animal
   {
   public:
      void hacerSonido() override
         {
            cout << "Miau!" << endl;
         }
   };

int main()
   {
      Perro* miPerro = new Perro();
      Gato* miGato = new Gato();

      cout << "Mi Perro hace: "; //Imprime Mi Perro hace: Guau!.
      miPerro->hacerSonido();

      cout << "Mi Gato hace: "; //Imprime Mi Gato hace: Miau!.
      miGato->hacerSonido();

      return 0;
   }
```

En este ejemplo:

Animal es una clase abstracta porque tiene un método puro virtual hacerSonido().
Perro y Gato son clases derivadas de Animal y deben implementar el método hacerSonido().

¿Cuándo usar clases abstractas?

- Cuando quieras definir una interfaz común para un conjunto de clases relacionadas.
- Cuando quieras evitar que se creen instancias directas de una clase base.
- Cuando quieras modelar conceptos abstractos que no pueden ser instanciados directamente.

16. ¿Qué diferencia hay entre Interfaz e Implementación?

La interfaz define el "qué" y la implementación define el "cómo". Una interfaz es un contrato que especifica un conjunto de métodos que una clase debe tener, pero no detalla cómo se implementarán esos métodos. Es como un contrato que establece las responsabilidades de una clase. Por otro lado, la implementación es la realización concreta de esos métodos, es decir, el código que define cómo se lleva a cabo cada acción. Una interfaz es como un plano, mientras que la implementación es la construcción real. Las interfaces promueven la abstracción y el polimorfismo, permitiendo que diferentes clases implementen la misma interfaz de formas distintas, mientras que la implementación se encarga de la lógica específica de cada clase.

17. ¿Para qué sirve el Método virtual? ¿Qué relación tiene con Polimorfismo?

Método Virtual
Un método virtual es una función miembro de una clase base que se declara con la palabra clave virtual. Esto indica al compilador que esta función puede ser redefinida (o sobrescrita) en las clases derivadas.

¿Cuál es su relación con el Polimorfismo?
El polimorfismo es la capacidad de que un objeto pueda tomar muchas formas. En el contexto de los métodos virtuales, esto significa que cuando se llama a un método

virtual a través de un puntero o referencia a la clase base, se ejecutará la versión del método correspondiente a la clase del objeto real, no de la clase del puntero o referencia.

18. Explicar Herencia (*es un*, *es como un*), Agregación y Composición (*forma parte de*, *tiene un*). Dar ejemplos de estas asociaciones de Objetos.

Herencia (Es un)

La herencia establece una relación *"es un"* entre clases. Una clase hija (subclase) hereda todas las propiedades y métodos de una clase padre (superclase). Esto significa que un objeto de la clase hija también es un objeto de la clase padre.

Ejemplo:

- **Clase Padre:** Animal
- **Clases Hijas:** Perro, Gato, Ave Un perro es un animal, un gato es un animal, y un ave es un animal.

```
#include <iostream>

using namespace std;

class Animal
{
   public:
      void comer()
         {
            cout << "El animal esta comiendo." << endl;
         }
};

class Perro : public Animal
{
   public:
      void ladrar()
         {
            cout << "Guau!" << endl;
         }
};

int main()
   {
      Perro* miPerro = new Perro();

      miPerro->comer(); // Imprime El animal esta comiendo. (método heredado de Clase Padre).
      miPerro->ladrar(); // Imprime Guau! (método propio de la Clase Perro).

      return 0;
}
```

Agregación (Tiene un)

La agregación representa una relación "tiene un" entre clases. Una clase contiene una instancia de otra clase, pero esta instancia puede existir independientemente. Es una relación más débil que la composición.

Ejemplo:

Clase: Automóvil
Clase: Motor Un automóvil tiene un motor. El motor puede existir independientemente del automóvil.

```
#include <iostream>

using namespace std;

class Motor
   {
      public:
         void encender()
         {
            cout << "El motor esta encendido." << endl;
         }
   };

class Automovil
{
   private:
      Motor motor;

   public:
      void arrancar()
         {
            motor.encender();
         }
};

int main()
   {
      Automovil* miAuto = new Automovil();

      miAuto->arrancar(); // Imprime El motor esta encendido. (Agregación de Clase Motor).

      return 0;
   }
```

Composición (Forma parte de)
La composición es un tipo especial de agregación que representa una relación más fuerte. Una clase está compuesta por otras clases, y la vida de las partes está ligada a la vida del todo. Si se destruye el objeto compuesto, también se destruyen sus componentes.

Ejemplo:

Clase: Casa

Clases: Pared, Techo, Puerta Una casa está formada por paredes, techo y puertas. Si la casa es demolida, también se destruyen sus componentes.

```
#include <iostream>

using namespace std;

class Pared
  {
    public:
      Pared()
        {
          cout << "Construyendo una pared..." << endl;
        }
  };

class Techo
  {
    public:
      Techo()
        {
          cout << "Construyendo un techo..." << endl;
        }
  };

class Puerta
  {
    public:
      Puerta()
        {
          cout << "Construyendo una puerta..." << endl;
        }
  };

class Casa
  {
    private:
      Pared pared1, pared2, pared3, pared4;
      Techo techo;
      Puerta puertaPrincipal;

    public:
      Casa()
        {
          cout << endl << "CONSTRUYENDO UNA CASA..." << endl;
        }
  };

int main()
  {
    Casa miCasa;
    return 0;
  }

/* EL MENSAJE DE SALIDA SERÍA:
```

```
Construyendo una pared...
Construyendo una pared...
Construyendo una pared...
Construyendo una pared...
Construyendo un techo...
Construyendo una puerta...

CONSTRUYENDO UNA CASA...
*/
```

Explicación:

a. **Clases:** Hemos definido cuatro clases: Pared, Techo, Puerta y Casa.
b. **Composición:** La clase Casa tiene como miembros a objetos de las clases Pared, Techo y Puerta. Esto significa que una casa está compuesta por paredes, un techo y una puerta.
c. **Constructor de Casa:** El constructor de la clase Casa se encarga de crear las instancias de las partes que componen la casa. Al crear un objeto de tipo Casa, se invocan automáticamente los constructores de Pared, Techo y Puerta, lo que simula el proceso de construcción de una casa en el mundo real.
d. **Main:** En la función main, al crear un objeto miCasa, se ejecuta el constructor de la clase Casa y, en consecuencia, se construyen todas las partes de la casa.

Puntos clave a destacar:

- **Vida útil:** La vida útil de las partes (paredes, techo, puerta) está ligada a la vida útil de la casa. Si se destruye el objeto miCasa, también se destruirán automáticamente las partes que la componen.
- **Encapsulación:** Los detalles de cómo se construyen las paredes, el techo y la puerta están encapsulados dentro de sus respectivas clases. La clase Casa solo se preocupa por la composición de estas partes.
- **Reutilización:** Las clases Pared, Techo y Puerta pueden ser reutilizadas en otros contextos, como, por ejemplo, para construir otros tipos de edificios.

19. ¿Por qué suponemos que el POO supera al Estructurado?

La programación orientada a objetos (POO) se ha convertido en el paradigma dominante en el desarrollo de software debido a una serie de ventajas significativas que ofrece en comparación con la programación estructurada. A continuación, presento las principales razones por las cuales se considera que la POO supera al enfoque estructurado:

1. **Modularidad y Reutilización:**

- **Encapsulación:** La POO permite agrupar datos y funciones relacionadas dentro de objetos, lo que facilita la creación de componentes reutilizables. Esto reduce la duplicación de código y mejora la mantenibilidad.
- **Herencia:** Al permitir que las clases hereden atributos y métodos de clases base, se promueve la reutilización de código y la creación de jerarquías de clases.
- **Polimorfismo:** La capacidad de tratar objetos de diferentes tipos como si fueran del mismo tipo permite una mayor flexibilidad y extensibilidad en el código.

2. **Abstracción:**
   - **Modelado del mundo real:** La POO permite modelar entidades del mundo real de manera más natural, facilitando la comprensión y el diseño de sistemas complejos.
   - **Ocultamiento de información:** Los objetos encapsulan sus datos y métodos internos, lo que permite a los desarrolladores centrarse en la interfaz del objeto sin preocuparse por los detalles de implementación.

3. **Mantenibilidad:**
   - **Localización de errores:** Los errores tienden a estar más localizados dentro de los objetos, lo que facilita su identificación y corrección.
   - **Modificaciones:** Al realizar cambios en un sistema orientado a objetos, es menos probable que se afecten otras partes del sistema, ya que los cambios suelen estar confinados a clases específicas.

4. **Escalabilidad:**
   - **Crecimiento orgánico:** Los sistemas orientados a objetos pueden crecer y evolucionar de manera más natural a medida que cambian los requisitos del sistema.
   - **Complejidad manejable:** La POO proporciona mecanismos para manejar la complejidad de los sistemas grandes al dividirlos en componentes más pequeños y manejables.

5. **Colaboración:**
   - **Trabajo en equipo:** La POO facilita el trabajo en equipo al permitir que diferentes desarrolladores trabajen en diferentes partes del sistema de forma independiente.
   - **Diseño modular:** Los objetos pueden ser diseñados y desarrollados por diferentes equipos, lo que acelera el desarrollo de grandes proyectos.

20. ¿Qué entiende por *"Principio de sustitución"*, es decir que una Clase Padre Reemplace a la clase hijo? Dar ejemplos en Código de C++.

El principio de sustitución de Liskov (LSP) dicta **que cualquier objeto de una subclase debe poder ser utilizado en lugar de un objeto de su clase base sin que se altere el correcto funcionamiento del programa.** Es decir, una subclase debe ser una versión más específica o especializada de su clase base, pero no debe cambiar el comportamiento fundamental de esta.

En términos más simples, si tenemos una función que espera un objeto de una clase base, debería poder funcionar sin problemas si le pasamos un objeto de una subclase de esa clase base. Esto garantiza la coherencia y la extensibilidad del código, ya que permite crear jerarquías de clases más robustas y fáciles de mantener.

Ejemplo:
Imaginemos una jerarquía de clases para representar diferentes tipos de vehículos:

```
class Vehiculo
   {
      public:
         virtual void acelerar() = 0;
         virtual void frenar() = 0;
   };

class Automovil : public Vehiculo
   {
      public:
         void acelerar() override
            {
               // Implementación específica para acelerar un automóvil.
            }
         void frenar() override
            {
               // Implementación específica para frenar un automóvil.
            }
   };

class AutoElectrico : public Automovil
   {
      public:
         void acelerar() override
            {
               // Implementación específica para acelerar un automóvil. eléctrico.
            }

         // ... otros métodos específicos de un auto eléctrico.
   };
```

En este ejemplo:

Vehiculo: Es la clase base que define el comportamiento común de todos los vehículos (acelerar y frenar).

Automovil: Es una subclase de Vehiculo que representa un automóvil específico.
AutoElectrico: Es una subclase de Automovil que representa un automóvil eléctrico.

¿Por qué esto cumple con el LSP?

Un objeto de tipo AutoElectrico puede ser utilizado en cualquier lugar donde se espera un objeto de tipo Vehiculo o Automovil sin causar problemas. Un AutoElectrico es un tipo especial de automóvil, por lo que puede hacer todo lo que puede hacer un automóvil convencional y más. Al llamar a los métodos acelerar() o frenar() sobre un objeto AutoElectrico, se ejecutará la implementación específica para un automóvil eléctrico, pero el programa seguirá funcionando correctamente.

Violación del LSP:

Imaginemos que agregamos un método recargarBateria() a la clase AutoElectrico. Este método no tendría sentido para un Automovil convencional, por lo que estaríamos violando el LSP. Si intentáramos llamar a este método sobre un objeto de tipo Automovil, obtendríamos un error en tiempo de compilación o en tiempo de ejecución.

Ejemplo:

```
class Vehiculo
   {
      public:
         virtual void acelerar() = 0;
         virtual void frenar() = 0;
   };

class Automovil : public Vehiculo
   {
      public:
         void acelerar() override
            {
               // Implementación específica para acelerar un automóvil.
            }

         void frenar() override
            {
               // Implementación específica para frenar un automóvil.
            }
   };

class AutoElectrico : public Automovil
   {
      public:
         void acelerar() override
            {
               // Implementación específica para acelerar un automóvil. eléctrico
```

```
        }

        void recargarBateria()
        {
            // Implementación para recargar la batería.
        }
};

int main()
{
    Automovil *miAuto = new Automovil();

    miAuto->acelerar();
    miAuto->frenar();

    // Esto causará un error en tiempo de compilación o ejecución.
    miAuto->recargarBateria(); // El método recargarBateria() no existe en Automovil.

    return 0;
}
```

Marco Práctico:

Se pide desarrollar una solución que Implemente el diagrama de clases adjunto, Implementar tanto el .hpp como el .cpp de las clases, realizar la Herencia, notar que taller tiene un atributo vehículo que es un arreglo de Vehículos, Implementar La función meter (coche o moto) implica introducir el coche en el taller y la función arreglarCoches o arreglarMotos implica recorrer todos los coches y motos que haya y arrancarlos.

Instituto Superior de Formación Técnica Nº 151
Carrera: Analista en Sistemas
Asignatura: Algoritmos y Estructuras de Datos I
Docente: Lic. José Oemig

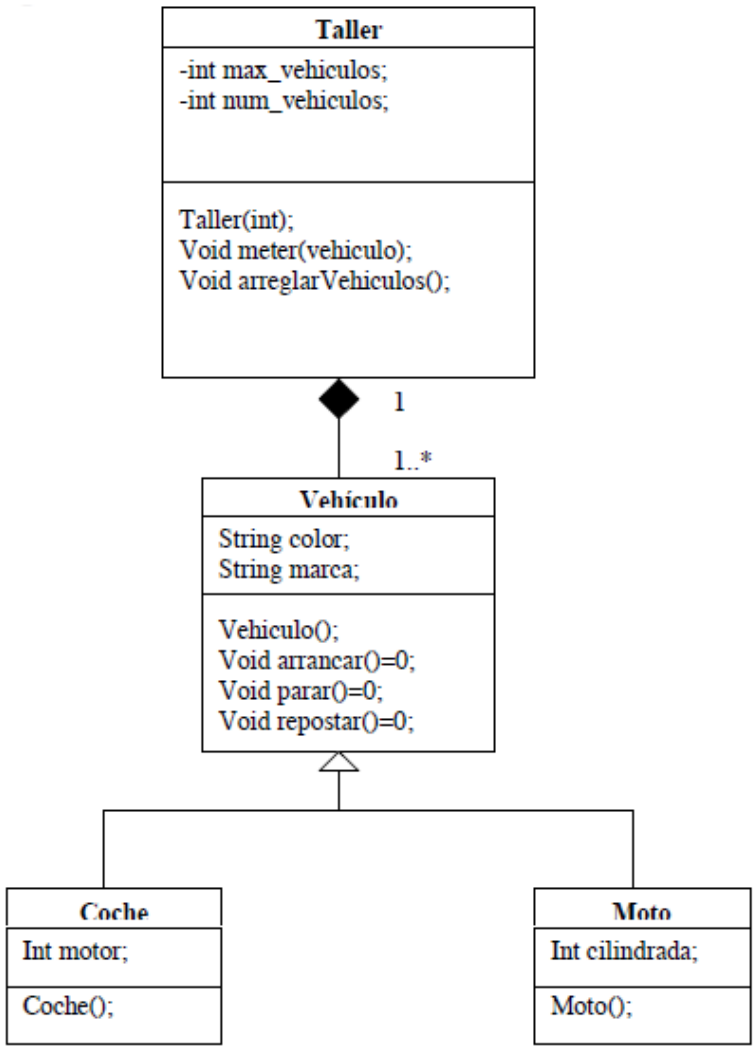
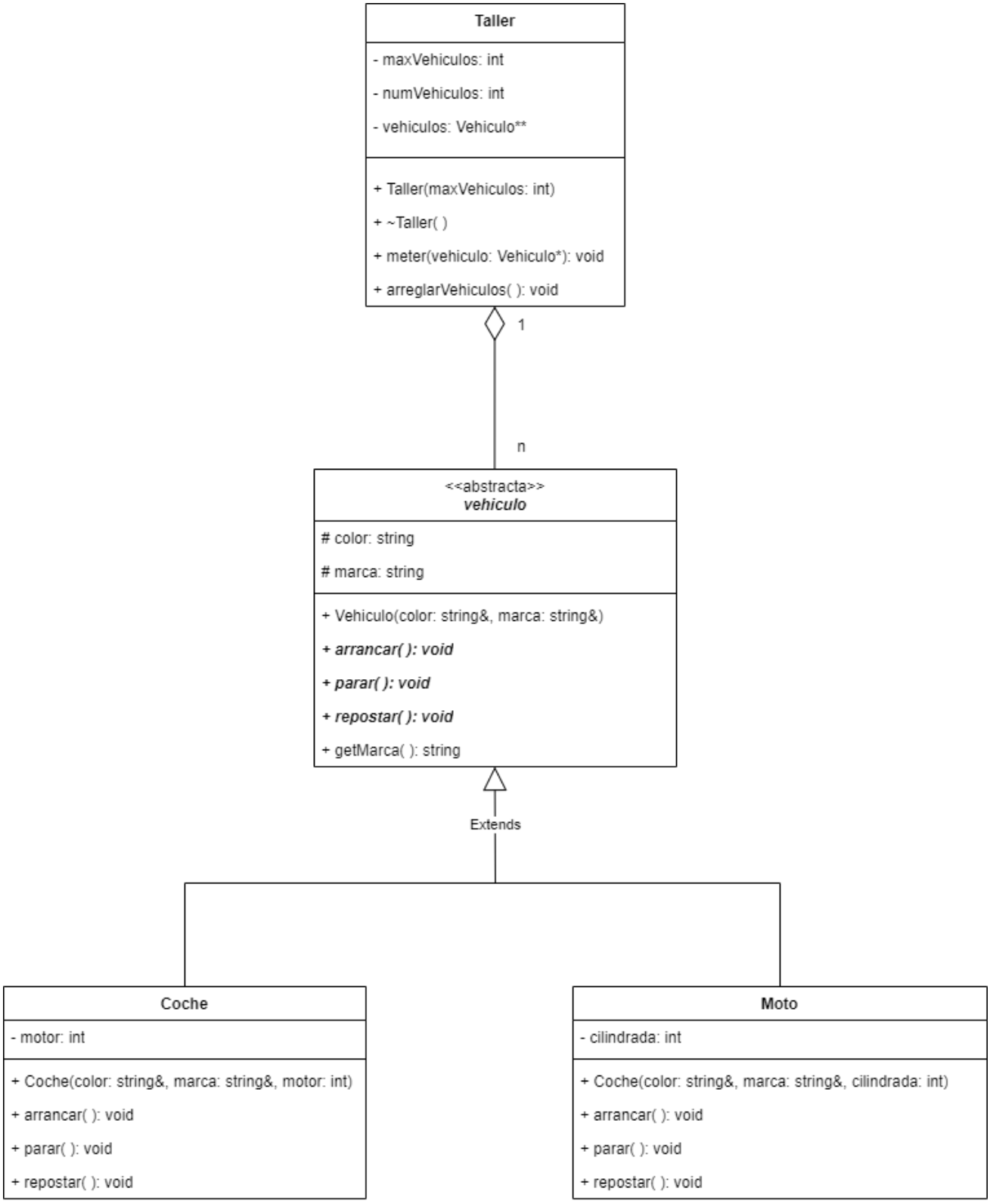


DIAGRAMA UML

Instituto Superior de Formación Técnica Nº 151
Carrera: Analista en Sistemas
Asignatura: Algoritmos y Estructuras de Datos I
Docente: Lic. José Oemig



EXPLICACIÓN DE LAS CONEXIONES

1. Vehiculo (Clase Base Abstracta):

- Define métodos abstractos para todos los vehículos (arrancar, parar, repostar).
- Proporciona atributos comunes como color y marca.

2. Coche y Moto (Clases Derivadas):

- Heredan de Vehiculo y sobrescriben los métodos abstractos.
- Introducen atributos específicos como motor (Coche) y cilindrada (Moto).

3. Taller:

- Contiene un arreglo de punteros a objetos Vehiculo.
- Gestiona los vehículos mediante los métodos meter y arreglarVehiculos.
- No conoce los detalles de Coche o Moto, solo interactúa con ellos a través de la interfaz Vehiculo.

MAIN.CPP

```
#include "include/taller.hpp"
#include "include/coche.hpp"
#include "include/moto.hpp"

int main()
  {
    Taller* miTaller = new Taller(3);

    Vehiculo* coche1 = new Coche("Rojo", "Fiat Palio", 1100);
    Vehiculo* coche2 = new Coche("Gris Tormenta", "Golf GTI", 2000);
    Vehiculo* moto1 = new Moto("Negro", "Yamaha", 600);

    cout << "TALLER EL PATA SUCIA" << endl;
    cout << "********************" << endl << endl;

    miTaller->meter(coche1);
    miTaller->meter(coche2);
    miTaller->meter(moto1);

    cout << endl;

    miTaller->arreglarVehiculos();

    cout << endl << "Probando los vehiculos reparados..." << endl << endl;

    coche1->parar();
    coche1->repostar();

    cout << endl;

    coche2->parar();
    coche2->repostar();

    cout << endl;

    moto1->parar();
    moto1->repostar();

    cout << endl << "Llego la Hora Feliz del Loko..." << endl;
    cout << "Si te queres llevar el tutu... Poniendo estaba la Gansa..." << endl;

    delete miTaller;
```

```
        return 0;
}
```

INCLUDE/TALLER.HPP

```
#ifndef TALLER_HPP
#define TALLER_HPP

#include "vehiculo.hpp"

class Taller
{
    private:
        int maxVehiculos;
        int numVehiculos;
        Vehiculo** vehiculos;

    public:
        Taller(int maxVehiculos);
        ~Taller();

        void meter(Vehiculo* vehiculo);
        void arreglarVehiculos();

};

#endif
```

SRC/TALLER.CPP

```
#include "../include/taller.hpp"

Taller::Taller(int maxVehiculos)
    : maxVehiculos(maxVehiculos), numVehiculos(0)
        {
        vehiculos = new Vehiculo*[maxVehiculos];
        }

Taller::~Taller()
    {
        for (int i = 0; i < numVehiculos; i++)
            {
                delete vehiculos[i];
            }

        delete[] vehiculos;
    }

void Taller::meter(Vehiculo* vehiculo)
    {
        if (numVehiculos < maxVehiculos)
            {
                vehiculos[numVehiculos++] = vehiculo;

                cout << "Vehiculo " << vehiculo->getMarca() << " metido al taller." << endl;
```

```
                } else
                    {
                        cout << endl <<"El taller esta hasta las manos. No se pueden meter mas vehiculos." << endl;
                    }
            }

        void Taller::arreglarVehiculos()
            {
                cout << "Arreglando vehiculos.." << endl;
                cout << "Mecanico Loko Trabajando..." << endl;
                cout << "(Bah, toma mate mientras el peon arregla)..." << endl << endl;

                for (int i = 0; i < numVehiculos; i++)
                    {
                        vehiculos[i]->arrancar();
                    }
    }
```

INCLUDE/VEHICULO.HPP

```
        #ifndef VEHICULO_HPP
        #define VEHICULO_HPP

        #include <string>
        #include <iostream>

        using namespace std;

        class Vehiculo
            {
                protected:
                    string color;
                    string marca;

                public:
                    Vehiculo(const string& color, const string& marca);
                    virtual ~Vehiculo() = default;

                    virtual void arrancar() const = 0;
                    virtual void parar() const = 0;
                    virtual void repostar() const = 0;

                    string getMarca() const;
            };

#endif
```

SRC/VEHICULO.CPP

```
        #include "../include/vehiculo.hpp"

        Vehiculo::Vehiculo(const string& color, const string& marca)
            : color(color), marca(marca) {}

        string Vehiculo::getMarca() const
            {
                return marca;
```

```
}
```

INCLUDE/COCHE.HPP

```
#ifndef COCHE_HPP
#define COCHE_HPP

#include "vehiculo.hpp"

class Coche : public Vehiculo
   {
      private:
         int motor;

      public:
         Coche(const string& color, const string& marca, int motor);

         void arrancar() const override;
         void parar() const override;
         void repostar() const override;

   };

#endif
```

SRC/COCHE.CPP

```
#include "../include/coche.hpp"

Coche::Coche(const string& color, const string& marca, int motor)
   : Vehiculo(color, marca), motor(motor) {}

void Coche::arrancar() const
   {
      cout << "El coche " << marca << " esta arrancando." << endl;
   }

void Coche::parar() const
   {
      cout << "El coche " << marca << " se ha parado." << endl;
   }

void Coche::repostar() const
   {
      cout << "El coche " << marca << " esta repostando." << endl;
}
```

INCLUDE/MOTO.HPP

```
#ifndef MOTO_HPP
#define MOTO_HPP

#include "vehiculo.hpp"

class Moto : public Vehiculo
   {
```

```
        private:
            int cilindrada;

        public:
            Moto(const string& color, const string& marca, int cilindrada);

            void arrancar() const override;
            void parar() const override;
            void repostar() const override;
    };

#endif
```

SRC/MOTO.CPP

```
    #include "../include/moto.hpp"

    Moto::Moto(const string& color, const string& marca, int cilindrada)
            : Vehiculo(color, marca), cilindrada(cilindrada) {}

    void Moto::arrancar() const
      {
        cout << "La moto " << marca << " esta arrancando." << endl;
      }

    void Moto::parar() const
      {
        cout << "La moto " << marca << " se ha parado." << endl;
      }

    void Moto::repostar() const
      {
        cout << "La moto " << marca << " esta repostando." << endl;
}
```

```
TALLER EL PATA SUCIA
********************

Vehiculo Fiat Palio metido al taller.
Vehiculo Golf GTI metido al taller.
Vehiculo Yamaha metido al taller.

Arreglando vehiculos..
Mecanico Loko Trabajando...
(Bah, toma mate mientras el peon arregla)...

El coche Fiat Palio esta arrancando.
El coche Golf GTI esta arrancando.
La moto Yamaha esta arrancando.

Probando los vehiculos reparados...

El coche Fiat Palio se ha parado.
El coche Fiat Palio esta repostando.

El coche Golf GTI se ha parado.
El coche Golf GTI esta repostando.

La moto Yamaha se ha parado.
La moto Yamaha esta repostando.

Llego la Hora Feliz del Loko...
Si te queres llevar el tutu...  Poniendo estaba la Gansa...
Presione una tecla para continuar . . .
```