Ingeniería en Sistemas de Información
y Ciencias de la Computación
Fisica

Tarea Final - Programación 1

Daniel Alejandro González Juárez 1290-22-4469

Juan Esteban Can López 1290-22-10376

Investigar los siguientes temas:

Unidad 1

- **Lenguaje C++**
  - **Conceptos Básicos:**

    Lenguaje de programación de propósito general y se usa ampliamente en la actualidad para la programación competitiva. Tiene características de programación imperativa, orientada a objetos y genérica. En esta sección se explican conceptos necesarios para aprender C + +.

- **Herramientas de desarrollo para C++**
  - Algunas de las herramientas principales serían MSVC, Clang, CMake o MSBuild, DevC++ o CodeBlocks. Otro también conocido es Visual Studio que nos permite agregar elementos necesarios para desarrollar aplicaciones en C++.

- Tipos de datos nativos en C++
  - int , double, char

    Un bloque de memoria asociado a un tipo fundamental en C++ se caracteriza por:

    1. Número de celdas (bytes) que lo componen es fijo
    2. Las celdas son contiguas
    3. Hay un único valor representado, no hay información accesoria ni metadatos

A este tipo de datos se los conoce como POD (Plain Old Data), tipos simples, lisos, cuya representación interna binaria se limita a codificar el valor .

C++ permite crear por parte del usuario nuevos tipos POD como agregación de otros tipos POD.

- Bibliotecas en C++
  - **fstream:** Flujos hacia/desde ficheros. Permite la manipulación de archivos desde el programar, tanto leer como escribir en ellos.
  - **iosfwd:** Contiene declaraciones adelantadas de todas las plantillas de flujos y sus typedefs estándar. Por ejemplo ostream.
  - **iostream:** Parte del a *STL* que contiene los algoritmos estándar, es quizá la más usada e importante (aunque no indispensable).
  - **La biblioteca list:** Parte de la *STL* relativa a contenedores tipo list; listas doblemente enlazadas
  - **math:** Contiene los prototipos de las funciones y otras definiciones para el uso y manipulación de funciones matemáticas.
  - **memory:** Utilidades relativas a la gestión de memoria, incluyendo asignadores y punteros inteligentes (*auto_ptr*).
    "auto_ptr" es una clase que conforma la librería

memory y permite un fácil manejo de punteros y su destrucción automaticamente.

- **Biblioteca new:** Manejo de memoria dinámica
- **numeric:** Parte de la librería numérica de la *STL* relativa a operaciones numéricas.
- **ostream:** Algoritmos estándar para los flujos de salida.
- **queue:** Parte de la *STL* relativa a contenedores tipo queue (colas de objetos).
- **Librería stdio:** Contiene los prototipos de las funciones, macros, y tipos para manipular datos de entrada y salida.
- **Librería stdlib:** Contiene los prototipos de las funciones, macros, y tipos para utilidades de uso general.
- **string:** Parte de la *STL* relativa a contenedores tipo string; una generalización de las cadenas alfanuméricas para albergar cadenas de objetos. Muy útil para el fácil uso de las cadenas de caracteres, pues elimina muchas de las dificultades que generan los char
- **typeinfo:** Mecanismo de identificación de tipos en tiempo de ejecución
- **vector:** Parte de la *STL* relativa a los contenedores tipo vector; una generalización de las matrices unidimensionales C/C++
- **forward_list:** Esta librería es útil para implementar con gran facilidad listas enlazadas simples.

- **list:** Permite implementar listas doblemente enlazadas (listas enlazadas dobles) fácilmente.
- **iterator:** Proporciona un conjunto de clases para iterar elementos.
- **regex:** Proporciona fácil acceso al uso de expresiones regulares para la comparación de patrones.
- **thread:** Útil para trabajar programación multihilos y crear múltiples hilos en nuestra aplicación.
- **time:** Útil para obtener marcas de tiempo durante la ejecución. Se usa con frecuencia para conocer el tiempo exacto durante un programa.

- Entrada y salida de flujos
  - E/S estándar

    Nos referimos a los flujos estándar como canales de comunicación de entrada y salida interconectados entre un programa de la computadora y su entorno cuando comienza su ejecución. Las tres conexiones de entrada /salida (E/S) se denominan entrada estándar (stdin), salida (stdout) y error estándar (stderr).

- **Programación estructurada**
  - Se caracteriza porque los programadores pueden fragmentar el código fuente de sus programas en bloque lógicamente estructurados, que consisten en bucles, bloques lógicos y comandos

condicionales. El objetivo de la programación estructurada es realizar programas fáciles de escribir, depurar y modificar. Los programas conseguidos son claros, ordenados, comprensibles, sin saltos

- **Programación orientada a objetos**
    - Es un paradigma de programación, lo que se define como un modelo de diseño de sistemas de software.
    - Centrado en clases y objetos. Esto quiere decir que separa y estructura el código en fragmentos más simples y reutilizables como lo que son las clases y nos permite instanciar.
    - **Los 4 pilares de la OOP**
    - 💡 El primer y más importante concepto que tenemos que tener en cuenta a la hora de hablar de POO es la distinción entre clase y objeto.

    - Una clase es una plantilla. Define de manera genérica cómo van a ser los objetos de un determinado tipo y un objeto es la implementación de esa plantilla.

    - Veámoslo con un ejemplo no tan abstracto. Supongamos que estamos en un juego y tenemos que crear nuestro propio player.

    - El juego nos da un personaje por default al que vamos a cambiarle el pelo, la ropa, las herramientas que tenga para personalizarlo y ser únicos en el juego. Este player por default es la clase Player. Y nuestra personalización es un

objeto. Cada jugador tendrá su personaje personalizado que es su propia instancia de la clase Player.

- 💡 El segundo concepto a entender es la diferencia entre atributo y método.

- Los atributos son propiedades del objeto que lo enmarcan y le dan forma. Estos pueden ser por ejemplo un id, nombre, etc. También pueden ser acciones que puede realizar y los mismos pueden ser implementados como métodos. Los métodos de la clase son funciones que el objeto puede invocar (por ejemplo, hablar, caminar, respirar en el objeto de una persona).

- Siguiendo el ejemplo anterior, todos los Players tendrán un atributo que es una herramienta que los ayudará en el juego. Algunos elegirán un hacha, otros una maza y así. Sin embargo, todos cuentan con su atributo Tool. Cuando entren al juego, lo primero que van a hacer es querer moverse por el plano que nos ofrece el mismo. Para eso, nuestra clase Player, necesita un método común a todas sus instancias que va a ejecutar la acción de moverse por el plano. Por eso, la clase tendrá su método Move.
- **1. Abstracción**
- La abstracción tiene dos puntos. Lo primero es que nos permite ocultar del usuario el funcionamiento interno de nuestro sistema. Por

ejemplo exponer en una GUI (Interfaz Gráfica) un botón de comprar con lo que entiende que puede comprar su producto con un click en ese botón sin saber qué sucede de fondo en nuestro sistema. También nos permite abstraer de los desarrolladores las funcionalidades internas de la clase. Permitiendo que los desarrolladores utilicen y manipulen la clase sin necesidad de entender qué sucede internamente en la misma.

- Por otro lado, nos permite abstraer del código estas entidades dándoles una forma y una representación que nos permita entender de otra forma de qué se trata el fragmento de código que la conforma. Un ejemplo de esto es cuando definimos una clase como Usuario.

- **2. Herencia**
- La herencia es la cualidad que nos permite reutilizar nuestro código y modelos plasmados en forma de clase implementando subclases. Las nuevas clases heredarán (como en la herencia genética) atributos y métodos de las clases padres o superclases. Este tipo de implementación de clases y superclases nos ayuda a crear sistemas escalables y con multiplicidad de opciones. Podemos ver esta característica en sistemas de ecommerce en los que tenemos diferentes tipos de usuarios(superclase) en los que unos se definen como compradores y otros como vendedores; también así en los productos y sus categorías.

- **3. Polimorfismo**
- El polimorfismo nos permite modificar ligeramente los métodos y atributos de las subclases previamente definidas en la superclase.

- Para entender mejor el polimorfismo podemos entenderlo bajo el aspecto de un sistema de usuarios en una plataforma de streaming de películas. Podemos tener usuarios con diferentes características pero un modelo similar, como es el caso de usuarios menores de edad a los que se les asignan cierto contenido que pueden ver dependiendo de su edad. Sin embargo el resto de los métodos y atributos de la clase Usuario las comparte con otros usuarios.

- **4. Encapsulación**
- El proceso de encapsulación nos permite proteger los datos y la integridad de nuestro sistema mediante la privacidad de los mismos.

- Existen diferentes tipos de encapsulación dependiendo del lenguaje. En casos como Typescript contamos con la posibilidad de encapsular los datos en el scope de la clase (private) y las subclases (protected). Como también existe la posibilidad de encapsular la modificación de datos permitiendo que la instancia de la misma solo pueda leer el dato (readonly) pero no modificarlo.

- Con estas posibilidades podemos proteger los procesos de nuestro sistema tanto de los usuarios como de otros desarrolladores definiendo las posibilidades de acceso de cada atributo y método.

- **Clase Y Super Clase**
  - Una clase es una plantilla para crear objetos. Una clase define un conjunto de atributos y métodos que se utilizan para crear objetos. Los objetos se crean a partir de una clase y se utilizan para acceder a los atributos y métodos de la clase. Una **superclase** es una clase que se utiliza como base para crear otras clases.

- **Sobrecarga de operadores**
  - Nos permite llamar a varios métodos con idéntica signatura, pero con diferente variedad de parámetros. Bajo un mismo método, podemos realizar implementaciones diferentes.

- **Propiedades y métodos de una clase**
  - Las propiedades de una clase se denominan miembros de datos y los métodos se llaman miembros de funciones

- **Creación de objetos**

- Se crean a partir de la sintaxis del operador “new” o simplemente declarándose como variables locales.

- **Constructores**
  - Son métodos especiales dentro de una clase que se utilizan para inicializar los objetos de esa clase. Los constructores tienen el mismo nombre que la clase y no tienen un tipo de retorno explícito. Hay varios tipos de constructores en C++:
  - Constructor por parámetros
  - Constructor por copia

Unidad 2

- **Operadores Lógicos**
  - **AND (&&)** devuelve un valor true, si ambas expresiones son verdaderas y false en el caso contrario
  - **OR (||)** Devuelve true si al menos una de las operaciones es verdadera y false si ambas son falsas
  - **NOT (!)** Invierte el valor de una expresión booleana, devolviendo true si la expresión original es falsa y false si la expresión original es verdadera.

- **Estructuras de control**

- **if:** Se utiliza para tomar decisiones basadas en una condición. Si la condición es verdadera, se ejecuta el bloque de código dentro del if. Si la condición es falta, se puede optar por ejecutar un bloque de código alternativo utilizando "else".
- **IF anidados:** Es posible anidar múltiples estructuras "if" dentro de otras estructuras if para realizar comprobaciones adicionales. Esto permite tomar decisiones más complejas basadas en múltiples condiciones.
- **for:** Estructura de control "for" se utiliza para ejecutar un bloque de código en un número específico de veces. Se compone de tres partes: inicialización, condición, y la expresión de incremento o decremento.
- **While:** Estructura de control "while" se utiliza para ejecutar un bloque de código repetidamente mientras se cumpla una condición.
- **Switch:** Estructura de control "switch" se utiliza para seleccionar una opción específica entre varias posibles, basada en valor de una expresión. Cada opción se representa con un "case" y se puede utilizar "break" para salir del "switch" una vez se haya encontrado una coincidencia. Además, se puede utilizar "continue" para omitir la ejecución de un bloque de código y pasar a la siguiente iteración del bucle más cercano

- **Diferencias entre operadores de igualdad y de asignación**
    - Los operadores de igualdad y asignación son dos operadores diferentes que tiene diferentes usos en C++
        - El operador de igualdad (==) se utiliza para comparar dos valores y devolver un valor booleano que indica si los valores son iguales o no
        - El operador de asignación (=) se utiliza para asignar un valor a una variable.

- **Recursividad**
    - Hablamos de que un programa se llama a sí mismo múltiples veces. Esto quiere decir que es un proceso que se repite. Su utilidad radica en mejorar la eficiencia de tu código, ya que puede ayudarte a solucionar problemas con rapidez.

- **Manejo de Excepciones**
    - El manejo de excepciones en C++ es un mecanismo que permite tratar errores en el tiempo de ejecución. Las excepciones se pueden lanzar desde cualquier punto del código y se pueden capturar en otro punto.

- **Procesamiento de Archivos y flujos en C++**
    - Cualquier archivo se abre, se asocia un flujo con el archivo. Al empezar la ejecución de un programa automáticamente se abren tres archivos y sus flujos asociados, la entrada estándar, la salida estándar y el error estándar. Los flujos proporcionan canales de comunicación entre archivos.

Unidad 3

- **Arreglos y Vectores**
  - Un vector o array en algunas traducciones es una secuencia de objetos del mismo tipo almacenados consecutivamente en memoria. El tipo de objeto almacenado en el array puede ser cualquier tipo definido en C/C + +. Los vectores y matrices son pasados siempre por referencia como argumentos de una función.

- **Declaración y creación de arreglos y ejemplos de uso de arreglos**
  - Para declarar un arreglo o vector, se define primero que tipo de dato va a ser, luego el nombre del vector y, entre corchetes, el tamaño que tendrá el vector.
    - Este debe inicializarse;
      - `int numero [] = {1,2,3,4,5,6};`
      - numero[0] ---> 1
      - numero[1] ---> 2
      - numero[2] ---> 3
      - numero[3] ---> 4
      - numero[4] ---> 5
      - numero[5] ---> 6

- **Arreglos a funciones**
  - Este proceso se realiza por medio de punteros

- **Búsqueda de datos en arrays en C++**

- Para obtener el índice o posición de un elemento en arreglo de PP; Para ellos usaremos simplemente el ciclo for y comparaciones, recorriendo el arreglo y en cada paso del ciclo, comparar la búsqueda con el valor ingresado.

- **Ordenamiento de arreglos**
  - El ordenamiento de arreglos en C++ se puede hacer de varias maneras. Una de ellas es el método de ordenamiento con estructuras por burbuja. Este método consiste en recorrer el arreglo varias veces y comparar si están en el orden equivocado. El proceso se repite hasta que el arreglo esté completamente ordenado.

- **Arreglos multidimensionales**
  - En c++ los arreglos multidimensionales son matrices que contienen elementos distribuidos en más de una dimensión. Puedes declarar y acceder a arreglos multidimensionales utilizando la sintaxis de corchetes.
- **Algoritmos de búsqueda y de ordenamiento**
  - Estos algoritmos funcionan a través de librerías, incluidas en la cabecera.
  - find
  - binary_search

    _ Ordenamiento _
  - sort
  - reverse

- **Motores de bases de datos**
  - **MYSQL**
    - Sistema de gestión de bases de datos relacionado muy popular. Se utiliza ampliamente para almacenar y administrar datos en una amplia variedad de aplicaciones y sitios web.

  - **MYSQL Server**
    - Se compone de un servidor de base de datos que gestiona y almacena los datos y una variedad de herramientas y clientes que permiten interactuar con el servidor.
    - Es una implementación del modelo de base de datos relación y que sigue el estándar SQL.
  - **PostgreSQL**
    - Sistema de gestión de bases de datos relacionales de código abierto y altamente escalable, es compatible con SQL y es conocido por su robustez, flexibilidad y capacidad para manejar grandes volúmenes.

Unidad 4

- **DML**
    - Se refiere a las instrucciones utilizadas para manipular y gestionar datos en una base de datos relacional. Estas instrucciones permiten realizar operaciones de inserción, actualización eliminación y consulta de datos en las tablas de base de datos. (Select, insert, update, delete)
- **DLL**

DLL significa Biblioteca de Vínculos Dinámicos. Es una extensión de archivo de Microsoft Windows que contiene funciones y datos que pueden ser utilizados por varios programas al mismo tiempo. SQL significa Lenguaje de Consulta Estructurado y se utiliza para administrar bases de datos relacionales. Las DLL se pueden utilizar en SQL Server para ampliar su funcionalidad proporcionando características y funcionalidades adicionales que no están disponibles en SQL Server en sí.

Aquí hay algunos recursos adicionales en español que pueden ayudarte a aprender más sobre SQL y DLL:

- [DDL es un subconjunto o parte del lenguaje SQL que trata con la definición y actualización de las estructuras de una base de datos.]
- Los Comandos DDL (Lenguaje de definición) de sql, son una serie de comandos que nos permiten definir nuestros objetos en la base de datos. Dentro de los comandos DDL se encuentran el CREATE, ALTER y TRUNCATE.]
- Dentro del lenguaje SQL existen varios tipos de sentencias (SELECT, INSERT, UPDATE, CREATE,…) las cuales podemos dividir en 2 grupos: Las sentencias DDL y DML. Las sentencias DDL (Data Definition Language)«Lenguaje de Definición de Datos» Son sentencias que nos permiten definir, alterar, modificar objetos dentro de mi base de datos.]

Unidad 5: Memoria dinámica y estructuras de datos Esta unidad aborda el uso de la memoria dinámica en C++ y la implementación de estructuras de datos básicas. Los temas incluyen: 1. Memoria dinámica ○ La memoria dinámica en C++ permite asignar y liberar memoria durante la ejecución del programa. A diferencia de la memoria estática, que se asigna en tiempo de compilación, la memoria dinámica se reserva en tiempo de ejecución y es especialmente útil cuando se requiere una cantidad de memoria variable o cuando se necesitan estructuras de datos como arreglos o listas enlazadas. ○ En C++, la gestión de la memoria dinámica se realiza a través de los operadores new y delete, que permiten asignar y liberar memoria, respectivamente. ○ Es importante destacar que la liberación de memoria dinámica debe realizarse adecuadamente para evitar fugas de memoria. Si no se libera la memoria asignada, se producirá una pérdida gradual de memoria a medida que el programa se ejecute. ○ Además, es posible que se produzcan errores si se intenta acceder a memoria liberada o se libera la misma memoria más de una vez. Por lo tanto, se recomienda seguir buenas prácticas al utilizar la memoria dinámica, como liberarla cuando ya no se necesite y evitar fugas de memoria. ○ Cabe mencionar que, en C++, también existen constructores y destructores especiales para gestionar la memoria dinámica en objetos de clases. Estos se utilizan junto con los operadores new y delete para asegurar la correcta inicialización y liberación de memoria en objetos dinámicos. 2. Relación entre apuntadores y arreglos ○ En C++, los punteros y los arreglos están estrechamente relacionados. De hecho, un arreglo se puede considerar

como un tipo especial de puntero. ○ Cuando se declara un arreglo en C++, el nombre del arreglo se comporta como un puntero constante que apunta a la primera posición de memoria del arreglo. Por lo tanto, se puede acceder a los elementos del arreglo utilizando tanto la notación de corchetes [] como la aritmética de punteros.

○ A continuación, se muestra un ejemplo que ilustra la relación entre los punteros y los arreglos:
○ cpp i. int arreglo[5] = {1, 2, 3, 4, 5}; ii. iii.

Acceso a elementos del arreglo utilizando la notación de corchetes [] iv. cout << arreglo[0]; // Imprime el primer elemento del arreglo (1) v. cout << arreglo[2]; // Imprime el tercer elemento del arreglo (3) vi. vii. // Acceso a elementos del arreglo utilizando la aritmética de punteros viii. cout << *(arreglo + 1); // Imprime el segundo elemento del arreglo (2) ix. cout << *(arreglo + 3); // Imprime el cuarto elemento del arreglo (4) ○ ○ En el ejemplo anterior, se declara un arreglo de enteros llamado arreglo con 5 elementos. Se puede acceder a los elementos del arreglo utilizando la notación de corchetes o utilizando la aritmética de punteros. Ambos enfoques proporcionan el mismo resultado. ○ La relación entre los punteros y los arreglos también se evidencia en la asignación de punteros a arreglos. Al asignar un puntero a un arreglo, el puntero apuntará a la primera posición de memoria del arreglo. ○ cpp i. int arreglo[3] = {1, 2, 3}; ii. int* ptr = arreglo; // Asignación del puntero al arreglo iii. iv. cout << *ptr; // Imprime el primer elemento del arreglo (1) v. cout << *(ptr + 1); // Imprime

el segundo elemento del arreglo (2) ○ ○ En este ejemplo, el puntero ptr se asigna al arreglo arreglo.

Ahora, el puntero ptr apunta al primer elemento del arreglo y se puede acceder a los elementos del arreglo utilizando la aritmética de punteros. ○ Es importante tener en cuenta que los punteros y los arreglos no son idénticos. Aunque el nombre del arreglo se comporta como un puntero constante, no se puede asignar un nuevo valor al nombre del arreglo ni realizar operaciones aritméticas directamente sobre él. Sin embargo, al asignar un puntero a un arreglo, el puntero puede modificarse y utilizarse para acceder y manipular los elementos del arreglo. 3. Apuntadores a funciones ○ En C++, los punteros a funciones son punteros que apuntan a la dirección de memoria de una función. Esto permite tratar a las funciones como objetos y utilizar los punteros para invocar dinámicamente diferentes funciones en tiempo de ejecución. ○ La sintaxis para declarar un puntero a una función es la siguiente: ○ cpp i. tipo_retorno
(*nombre_puntero_funcion)(tipo_parametros); ○
○ Donde tipo_retorno es el tipo de dato devuelto por la función, nombre_puntero_funcion es el nombre del puntero a la función y tipo_parametros son los tipos de datos de los parámetros que recibe la función.

○ A continuación, se muestra un ejemplo de declaración de un puntero a una función que recibe un entero y devuelve un entero: ○ cpp i. int (*puntero_funcion)(int); ○ ○ Una vez que se ha declarado un puntero a una función, se puede asignar a él la dirección de una función existente

utilizando el nombre de la función: ○ cpp i. int suma(int a, int b) { ii. return a + b; iii. } iv. v. int (*puntero_funcion)(int) = suma; ○ ○ En este ejemplo, se asigna la dirección de la función suma al puntero puntero puntero_funcion.

Ahora, el puntero puede utilizarse para invocar la función utilizando la sintaxis de puntero a función: ○ cpp ○ int resultado = puntero función(3); ○ ○ En este caso, se invoca la función apuntada por el puntero puntero puntero función con un argumento de 3, y se almacena el resultado en la variable resultado. ○ 4. Asignación de memoria dinámica ○ La asignación de memoria dinámica en C++ se realiza utilizando el operador new seguido del tipo de dato que se desea asignar y, opcionalmente, la cantidad de elementos si se trata de un arreglo. La asignación de memoria dinámica devuelve un puntero al espacio de memoria reservado. ○ La sintaxis básica para asignar memoria dinámica es la siguiente: ○ cpp i. tipo* puntero = new tipo; ○ Donde tipo representa el tipo de dato que se desea asignar y puntero es un puntero que apunta al espacio de memoria asignado. ○ A continuación, se muestra un ejemplo de asignación dinámica de un entero: ○ cpp i. int* punteroEntero = new int; ○ En este caso, se asigna memoria dinámica para un entero y se guarda la dirección de memoria asignada en el puntero punteroEntero. ○ También es posible asignar memoria dinámica para arreglos utilizando el operador new[] seguido del tipo de dato y la cantidad de elementos: ○ cpp i. tipo* puntero = new tipo[tamaño]; ○ A continuación, se muestra un ejemplo de asignación dinámica de un arreglo de enteros: ○ cpp

i. int tamaño = 5;
ii. int* punteroArreglo = new int[tamaño];
○ ○ En este caso, se reserva memoria dinámica para un arreglo de enteros con tamaño 5 y se guarda la dirección de memoria asignada en el puntero punteroArreglo. ○ Es importante tener en cuenta que, después de utilizar la memoria asignada dinámicamente, se debe liberar utilizando el operador delete para evitar fugas de memoria. La liberación de memoria se realiza de la siguiente manera: ○ cpp ○ delete puntero; ○ Para liberar la memoria de un arreglo dinámico, se utiliza el operador delete[]: ○ cpp i. delete[] puntero; ○ Es fundamental liberar la memoria asignada dinámicamente cuando ya no se necesite para evitar fugas de memoria y asegurar una gestión adecuada de los recursos. 5. Introducción a listas enlazadas (Conceptos) ○ Una lista enlazada, también conocida como lista enlazada o linked list en inglés, es una estructura de datos dinámica utilizada para almacenar una colección de elementos. A diferencia de los arreglos estáticos, que tienen un tamaño fijo, las listas enlazadas permiten una asignación y liberación flexible de memoria a medida que se agregan o eliminan elementos de la lista. ○ En una lista enlazada, cada elemento, llamado nodo, consta de dos partes principales: el dato que se desea almacenar y un puntero que apunta al siguiente nodo en la lista. El último nodo de la lista tiene un puntero nulo o vacío para indicar el final de la lista. ○ La estructura básica de un nodo en una lista enlazada se define de la siguiente manera: ○ cpp i. struct Nodo { ii. tipo_dato dato; iii. Nodo* siguiente; iv. }; ○ ○ Donde tipo_dato representa el tipo de dato que se desea almacenar en la lista y

siguiente es un puntero que apunta al siguiente nodo. ○ La lista enlazada se forma mediante la conexión secuencial de los nodos, donde el puntero siguiente de un nodo apunta al siguiente nodo de la lista. El primer nodo de la lista se llama nodo cabeza o nodo raíz, y sirve como punto de entrada para acceder a los demás nodos. ○ La siguiente ilustración muestra un ejemplo de una lista enlazada que contiene tres nodos: ○ lua
i. +-------+ +-------+ +-------+ ii. | dato1 | ---> | dato2 | ---> | dato3 | iii. +-------+ +-------+ +-------+ ○ En este ejemplo, cada nodo almacena un dato y un puntero que apunta al siguiente nodo. El último nodo tiene un puntero nulo para indicar el final de la lista. ○ Las listas enlazadas ofrecen varias ventajas y características, como la flexibilidad en la asignación de memoria, la capacidad de agregar y eliminar elementos de forma eficiente y la capacidad de reorganizar los elementos de la lista sin la necesidad de mover grandes bloques de memoria. Sin embargo, también presentan algunas desventajas, como el mayor consumo de memoria debido a la necesidad de almacenar los punteros y la falta de acceso directo a los elementos de la lista, ya que se debe recorrer secuencialmente desde el nodo raíz. ○ Las listas enlazadas se utilizan en diversos escenarios y se pueden implementar de diferentes maneras, como listas enlazadas simples, listas enlazadas dobles y listas enlazadas circulares, cada una con sus propias características y aplicaciones específicas. Unidad 6: Estructura de datos avanzadas Esta unidad se enfoca en estructuras de datos más avanzadas, como listas enlazadas, pilas y colas. Los temas tratados son: 1. Introducción a la estructura de

datos en C++ ○ La estructura de datos en C++ se refiere a la organización y manipulación de datos en un programa utilizando diversas estructuras y algoritmos. Las estructuras de datos permiten almacenar, organizar y acceder a los datos de manera eficiente, lo que es fundamental para el diseño y la implementación de programas eficaces y escalables. ○ C++ proporciona una variedad de estructuras de datos incorporadas y también permite la implementación de estructuras de datos personalizadas.

Algunas de las estructuras de datos más comunes en C++ son: ○ Arreglos: Son colecciones contiguas de elementos del mismo tipo. Los arreglos tienen un tamaño fijo y permiten el acceso directo a los elementos mediante índices. ○ Vectores: Son contenedores dinámicos que pueden aumentar o disminuir de tamaño automáticamente según sea necesario. Los vectores son similares a los arreglos, pero ofrecen mayor flexibilidad en la gestión de memoria. ○ Listas enlazadas: Como se mencionó anteriormente, las listas enlazadas son estructuras de datos dinámicas que constan de nodos enlazados mediante punteros. Permiten la inserción y eliminación eficiente de elementos en cualquier posición de la lista.

○ **Pilas:** Son estructuras de datos de tipo LIFO (Last In, First Out), lo que significa que el último elemento insertado es el primero en ser eliminado. Las pilas se utilizan en situaciones en las que se requiere un acceso rápido a los elementos más recientes. ○ Colas: Son estructuras de datos de tipo **FIFO** (First In, First Out),

lo que significa que el primer elemento insertado es el primero en ser eliminado. Las colas se utilizan en situaciones en las que se requiere un procesamiento ordenado de elementos.

○ **Árboles**: Son estructuras de datos jerárquicas compuestas por nodos conectados mediante enlaces. Los árboles se utilizan en una amplia gama de aplicaciones, como la representación de estructuras jerárquicas de datos y la implementación de algoritmos de búsqueda y ordenamiento. ○ Grafos: Son estructuras de datos compuestas por un conjunto de nodos conectados mediante aristas. Los grafos se utilizan para representar relaciones entre entidades y se aplican en algoritmos de búsqueda, rutas, redes y muchas otras áreas.

○ Estas son solo algunas de las estructuras de datos básicas en C++, y hay muchas otras disponibles. La elección de la estructura de datos adecuada depende de los requisitos y las características específicas del problema a resolver. C++ proporciona bibliotecas estándar, como , , , , , , , que ofrecen implementaciones optimizadas de estas estructuras de datos y algoritmos asociados. ○ 2. Clases auto referenciadas ○ Las clases auto referenciadas son aquellas en las que un miembro de la clase es un puntero o una referencia a otra instancia de la misma clase. Esto permite crear estructuras de datos recursivas o relacionadas entre sí.

○ En C++, las clases auto referenciadas se utilizan comúnmente para implementar estructuras de datos como

listas enlazadas, árboles, grafos y otras estructuras dinámicas y recursivas. ○ Veamos un ejemplo de cómo se puede utilizar una clase auto referenciada para implementar una lista enlazada en C++: ○ cpp

```
i. class Nodo {
ii. public: iii. int dato;
iv. Nodo* siguiente;
v.
vi. Nodo(int dato) {
vii. this->dato = dato;
viii. siguiente = nullptr;
ix. }
x. };
xi.
xii. class ListaEnlazada {
xiii. private: xiv. Nodo* cabeza;
xv. public:
xvi. ListaEnlazada() {
xvii. cabeza = nullptr;
xviii. }
xix.
xx. void insertar(int dato) {
xxi. Nodo* nuevoNodo = new Nodo(dato);
xxii. if (cabeza == nullptr) {
xxiii. cabeza = nuevoNodo;
xxiv. } else {
xxv. Nodo* temp = cabeza;
xxvi. while (temp->siguiente != nullptr) {
xxvii. temp = temp->siguiente;
xxviii. }
xxix. temp->siguiente = nuevoNodo;
```

```
xxx. }
xxxi. }
xxxii.
xxxiii. void imprimir() {
xxxiv. Nodo* temp = cabeza;
xxxv. while (temp != nullptr) {
xxxvi. cout << temp->dato << " ";
xxxvii. temp = temp->siguiente;
xxxviii. }
xxxix. cout << endl;
xl. }
xli. }; ○
```

○ En este ejemplo, la clase Nodo representa un nodo de la lista enlazada. Tiene un miembro dato para almacenar el valor del nodo y un puntero siguiente que apunta al siguiente nodo en la lista. La clase ListaEnlazada tiene un puntero cabeza que indica el primer nodo de la lista. ○ En el método insertar(), se crea un nuevo nodo y se enlaza al final de la lista. Si la lista está vacía, el nuevo nodo se convierte en la cabeza de la lista. Si la lista no está vacía, se recorre la lista hasta llegar al último nodo y se enlaza el nuevo nodo. ○ El método imprimir() muestra los valores de todos los nodos en la lista. ○ Este es solo un ejemplo básico de cómo se puede implementar una lista enlazada utilizando una clase auto referenciada. Se pueden agregar más operaciones como eliminar nodos, buscar elementos, etc.

○ **Las clases auto referenciadas** son útiles para crear estructuras de datos flexibles y dinámicas en las que los

nodos o elementos están interconectados mediante punteros o referencias. Esto permite la creación de estructuras de datos complejas y eficientes para resolver diversos problemas de programación. 3. Listas enlazadas

○ **Una lista enlazada** es una estructura de datos lineal compuesta por una secuencia de nodos, donde cada nodo contiene un valor y un enlace (puntero o referencia) al siguiente nodo en la lista. Cada nodo se considera un objeto independiente que contiene el dato y una referencia al siguiente nodo, lo que permite la creación de una secuencia enlazada. ○ En C++, una lista enlazada se puede implementar utilizando clases auto referenciadas, como se mencionó anteriormente. Cada nodo de la lista enlazada se representa mediante una clase que contiene el dato y un puntero al siguiente nodo. ○ A continuación se muestra un ejemplo de implementación básica de una lista enlazada en C++: ○ cpp

```
i. class Nodo {
ii. public:
iii. int dato;
iv. Nodo* siguiente;
v.
vi. Nodo(int dato) {
vii. this->dato = dato;
viii. siguiente = nullptr;
ix. }
x. };
xi.
xii. class ListaEnlazada {
xiii. private:
```

```
xiv. Nodo* cabeza;
xv. public:
xvi. ListaEnlazada() {
xvii. cabeza = nullptr;
xviii. }
xix.
xx. void insertar(int dato) {
xxi. Nodo* nuevoNodo = new Nodo(dato);
xxii. if (cabeza == nullptr) {
xxiii. cabeza = nuevoNodo;
xxiv. } else { xxv. Nodo* temp = cabeza;
xxvi. while (temp->siguiente != nullptr) {
xxvii. temp = temp->siguiente;
xxviii. }
xxix. temp->siguiente = nuevoNodo;
xxx. }
xxxi. }
xxxii.
xxxiii. void imprimir() {
xxxiv. Nodo* temp = cabeza;
xxxv. while (temp != nullptr) {
xxxvi. cout << temp->dato << " ";
xxxvii. temp = temp->siguiente;
xxxviii. } xxxix. cout << endl;
xl. }
xli. };
```

○ En este ejemplo, la clase Nodo representa un nodo de la lista enlazada. Tiene un miembro dato para almacenar el valor del nodo y un puntero siguiente que apunta al siguiente nodo en la lista.

○ La **clase Lista Enlazada** contiene un puntero cabeza que indica el primer nodo de la lista. Se proporcionan métodos para insertar elementos al final de la lista y para imprimir los valores de todos los nodos en la lista. ○ La inserción se realiza recorriendo la lista hasta llegar al último nodo y enlazando el nuevo nodo al final. ○ La impresión se realiza recorriendo la lista desde el nodo de la cabeza hasta el último nodo, mostrando el valor de cada nodo. ○ Esta es solo una implementación básica de una lista enlazada en C++. Se pueden agregar más operaciones según las necesidades, como eliminar nodos, buscar elementos, etc. ○ Las listas enlazadas son útiles cuando se necesita una estructura de datos dinámica que permita inserciones y eliminaciones eficientes en cualquier posición de la lista. Además, tienen la ventaja de utilizar memoria de manera más eficiente que las estructuras de datos de tamaño fijo, como los arreglos. Sin embargo, también tienen la desventaja de requerir un mayor consumo de memoria debido a los punteros adicionales utilizados para enlazar los nodos. ○

4. **Asignación dinámica** de memoria y estructura de datos ○ La asignación dinámica de memoria es un concepto importante en C++ que permite reservar memoria durante el tiempo de ejecución para almacenar datos. Esta capacidad es especialmente útil cuando se trabaja con estructuras de datos cuyo tamaño no se conoce de antemano o puede cambiar durante la ejecución del programa. ○ La asignación dinámica de memoria se realiza utilizando el operador new en C++. El operador new solicita un bloque de memoria del

tamaño especificado y devuelve un puntero al inicio de ese bloque de memoria.

A continuación, se puede utilizar ese puntero para acceder y manipular los datos en la memoria reservada. ○ Cuando se utiliza la asignación dinámica de memoria en conjunción con estructuras de datos, se pueden crear estructuras de datos flexibles y dinámicas que se ajusten a las necesidades del programa en tiempo de ejecución. Por ejemplo, se pueden crear listas enlazadas, árboles binarios, grafos y otras estructuras de datos complejas cuyo tamaño y forma se determinan dinámicamente. ○ Veamos un ejemplo de cómo se puede utilizar la asignación dinámica de memoria para crear una lista enlazada: ○ cpp

```
i. class Nodo {
ii. public:
iii. int dato;
iv. Nodo* siguiente;
v.
vi. Nodo(int dato) {
vii. this->dato = dato;
viii. siguiente = nullptr;
ix. }
x. };
xi.
xii. class ListaEnlazada {
xiii. private:
xiv. Nodo* cabeza;
xv. public:
xvi. ListaEnlazada() {
xvii. cabeza = nullptr; xviii. }
```

```
xix.
xx. void insertar(int dato) {
xxi. Nodo* nuevoNodo = new Nodo(dato);
xxii. if (cabeza == nullptr) {
xxiii. cabeza = nuevoNodo;
xxiv. } else {
xxv. Nodo* temp = cabeza;
xxvi. while (temp->siguiente != nullptr) {
xxvii. temp = temp->siguiente;
xxviii. }
xxix. temp->siguiente = nuevoNodo;
xxx. }
xxxi. }
xxxii.
xxxiii. void imprimir() {
xxxiv. Nodo* temp = cabeza;
xxxv. while (temp != nullptr) {
xxxvi. cout << temp->dato << " ";
xxxvii. temp = temp->siguiente;
xxxviii. }
xxxix. cout << endl; xl. }
xli. }; ○
```

En este ejemplo, la clase Nodo y la clase Lista Enlazada representan una lista enlazada implementada con asignación dinámica de memoria. Cada vez que se inserta un nuevo nodo en la lista, se utiliza new para asignar memoria para el nuevo nodo en el heap. Luego, se enlaza ese nodo en la lista. ○ Es importante tener en cuenta que, al utilizar la asignación dinámica de memoria, también es necesario liberar la memoria cuando ya no se necesita. Esto se realiza utilizando el operador delete. En el caso de la

lista enlazada, se puede agregar un destructor en la clase Lista Enlazada para liberar la memoria ocupada por todos los nodos de la lista. ○ La asignación dinámica de memoria es una herramienta poderosa en C++ que permite la creación de estructuras de datos flexibles y eficientes. Sin embargo, también requiere una gestión cuidadosa de la memoria para evitar fugas de memoria o el acceso a memoria no válida. Es importante liberar adecuadamente la memoria reservada utilizando delete cuando ya no se necesite. ○ 5. Pilas y colas ○ Las pilas y colas son estructuras de datos lineales que se utilizan para organizar y manipular elementos de manera específica. Ambas estructuras siguen el principio de "último en entrar, primero en salir" (LIFO) para las pilas y "primero en entrar, primero en salir" (FIFO) para las colas. ○ Una pila es una estructura de datos en la que los elementos se insertan y eliminan solo desde uno de los extremos, llamado "tope" o "cima" de la pila. La operación de inserción se conoce como "push" y coloca un elemento en la cima de la pila, mientras que la operación de eliminación se conoce como "pop" y retira el elemento superior de la pila. ○ Por otro lado, una cola es una estructura de datos en la que los elementos se insertan al final de la cola y se eliminan desde el frente de la cola. La operación de inserción se conoce como "enqueue" y agrega un elemento al final de la cola, mientras que la operación de eliminación se conoce como "dequeue" y retira el elemento frontal de la cola.

○ En C++, se pueden implementar pilas y colas utilizando listas enlazadas o arreglos, pero aquí se presentará una implementación utilizando listas enlazadas: ○ cpp

```
 i. class Nodo {
ii. public: iii. int dato;
iv. Nodo* siguiente;
v.
vi. Nodo(int dato) {
vii. this->dato = dato;
viii. siguiente = nullptr;
ix. }
x. };
xi.
xii. class Pila {
xiii. private:
xiv. Nodo* cima;
xv. public:
xvi. Pila() { xvii. cima = nullptr;
xviii. }
xix.
xx. void push(int dato) {
xxi. Nodo* nuevoNodo = new Nodo(dato);
xxii. nuevoNodo->siguiente = cima;
xxiii. cima = nuevoNodo;
xxiv. }
xxv.
xxvi. void pop() {
xxvii. if (cima == nullptr) {
xxviii. cout << "La pila está vacía" << endl;
xxix. } else {
xxx. Nodo* temp = cima;
xxxi. cima = cima->siguiente;
xxxii. delete temp;
xxxiii. } xxxiv. }
```

```
xxxv. };
xxxvi.
xxxvii. class Cola {
xxxviii. private:
xxxix. Nodo* frente;
xl. Nodo* fin;
xli. public:
xlii. Cola() {
xliii. frente = nullptr;
xliv. fin = nullptr;
xlv. }
xlvi.
xlvii. void enqueue(int dato) {
xlviii. Nodo* nuevoNodo = new Nodo(dato);
xlix. if (frente == nullptr) {
l. frente = nuevoNodo;
li. fin = nuevoNodo;
lii. } else {
liii. fin->siguiente = nuevoNodo;
liv. fin = nuevoNodo;
lv. } lvi. } lvii.
lviii. void dequeue() {
lix. if (frente == nullptr) {
lx. cout << "La cola está vacía" << endl;
lxi. } else {
lxii. Nodo* temp = frente;
lxiii. frente = frente->siguiente;
lxiv. delete temp;
lxv. }
lxvi. }
lxvii. };
```

○ En este ejemplo, se implementan las clases Pila y Cola utilizando listas enlazadas. Ambas clases tienen operaciones básicas como push/enqueue para agregar elementos y pop/dequeue para eliminar elementos. ○ En la clase Pila, el método push agrega un nuevo nodo en la cima de la pila, y el método pop elimina el nodo superior de la pila. ○ En la clase Cola, el método enqueque agrega un nuevo nodo al final de la cola, y el método dequeue elimina el nodo frontal de la cola. ○ Las pilas y colas son estructuras de datos comunes utilizadas en una amplia variedad de aplicaciones, como la gestión de tareas, la implementación de algoritmos de búsqueda y recorrido, el manejo de eventos, entre otros. Son especialmente útiles en situaciones en las que el orden de los elementos y su procesamiento son importantes