UNIVERSIDAD MARIANO GÁLVEZ DE GUATEMALA

INGENIERÍA EN SISTEMAS DE INFORMACIÓN Y CIENCIAS DE LA COMPUTACIÓN

CURSO: PROGRAMACIÓN I

CATEDRÁTICO: ING. ARMANDO CARDONA PAIZ

TAREA FINAL

PABLO RAFAEL ALVAREZ FAREL 1290-22- 3226

MAURO FRANCISCO MARROQUÍN AZURDIA 1290-22-3436

LA ANTIGUA GUATEMALA

28 DE MAYO DE 2022

Unidad 1: Conceptos básicos

El primer tema se centra en los conceptos básicos de C + +, sentando las bases para el aprendizaje posterior. Aquí se exploran los siguientes temas:

1. **Herramientas de desarrollo para C++**
    - Los editores más comunes utilizados para programar en C++, como Visual Studio Code, Eclipse, Xcode, etc.
    - Configuración y uso de un entorno de desarrollo integrado (IDE) para programar en C++.
    - Entornos de desarrollo integrados (IDE):
        i. Visual Studio: Es un IDE desarrollado por Microsoft que ofrece un conjunto completo de herramientas para el desarrollo en C++ y otras plataformas. Proporciona un editor de código, compilador, depurador y otras características avanzadas para facilitar el desarrollo de aplicaciones en C++.
        ii. Eclipse CDT: Es una versión de Eclipse especialmente diseñada para el desarrollo en C++ y otras variantes de C. Ofrece un entorno de desarrollo completo con características como resaltado de sintaxis, autocompletado de código, depuración y gestión de proyectos.
        iii. Xcode: Es el IDE oficial de Apple para el desarrollo de aplicaciones en macOS, iOS, watchOS y tvOS. Incluye herramientas para programar en C++ y otros lenguajes, así como para el desarrollo de aplicaciones gráficas y móviles.
        iv. Code::Blocks: Es un IDE de código abierto y multiplataforma que admite múltiples compiladores, incluido el compilador GNU GCC para C++. Proporciona un entorno de desarrollo simple pero efectivo con características esenciales para la escritura y compilación de código en C++.
    - Editores de texto:
        i. Visual Studio Code: Aunque es un editor de texto, Visual Studio Code ofrece una amplia gama de extensiones y complementos para habilitar el desarrollo en C++. Proporciona funciones como resaltado de sintaxis, autocompletado de código, depuración y gestión de proyectos.

ii. Sublime Text: Es un editor de texto ligero y altamente personalizable que admite el desarrollo en C++. Proporciona funciones avanzadas como resaltado de sintaxis, búsqueda y reemplazo avanzado, y soporte para complementos y paquetes adicionales.
iii. Atom: Es otro editor de texto de código abierto y altamente personalizable que se utiliza ampliamente para programar en C++. Ofrece características similares a Sublime Text, como resaltado de sintaxis, autocompletado y una amplia gama de complementos.

- Compiladores:
  i. GNU GCC: Es un conjunto de compiladores de código abierto ampliamente utilizados que incluye el compilador de C++ (g++). Es compatible con múltiples plataformas y ofrece un alto nivel de optimización y rendimiento.
  ii. Clang: Es otro conjunto de compiladores de código abierto que se ha vuelto popular en los últimos años. Proporciona un compilador de C++ (clang++) con características avanzadas y un enfoque en la detección de errores y el rendimiento.
-

2. Tipos de datos nativos en C++

- Los tipos de datos básicos que se encuentran en C++, como enteros, flotantes, caracteres, booleanos, etc.
- Enteros (Integer):
  i. int: Representa números enteros con signo. Suele ocupar 4 bytes de memoria.
  ii. short: Representa números enteros con signo más pequeños que int. Suele ocupar 2 bytes de memoria.
  iii. long: Representa números enteros con signo más grandes que int. Suele ocupar al menos 4 bytes de memoria.
  iv. long long: Representa números enteros con signo aún más grandes que long. Suele ocupar al menos 8 bytes de memoria.
  v. unsigned int, unsigned short, unsigned long, unsigned long long: Representan números enteros sin signo. Utilizan la misma cantidad de bytes que sus equivalentes con signo, pero solo almacenan valores no

negativos.

- Punto flotante (Floating-point):
    i. float: Representa números en coma flotante de precisión simple. Suele ocupar 4 bytes de memoria y tiene una precisión limitada.
    ii. double: Representa números en coma flotante de precisión doble. Suele ocupar 8 bytes de memoria y ofrece mayor precisión que float.
    iii. long double: Representa números en coma flotante de mayor precisión que double. Suele ocupar al menos 8 bytes de memoria.
- Caracteres y cadenas:
    i. char: Representa un único carácter. Suele ocupar 1 byte de memoria.
    ii. char16_t, char32_t: Representan caracteres Unicode de 16 y 32 bits, respectivamente.
    iii. wchar_t: Representa caracteres de ancho amplio (wide characters) que pueden ocupar más de 1 byte.
    iv. string: Representa una secuencia de caracteres. Es un tipo de dato compuesto que proporciona funcionalidades adicionales para trabajar con cadenas de texto.
- Booleano:
    i. bool: Representa un valor booleano que puede ser verdadero (true) o falso (false). Suele ocupar 1 byte de memoria.

3. Bibliotecas en C++

- Biblioteca estándar de C++ (STL):
    i. iostream: Proporciona las funciones básicas de entrada y salida, como la lectura y escritura en la consola.
    ii. vector: Ofrece una implementación de arreglos dinámicos que permite la gestión de secuencias de elementos.
    iii. string: Proporciona funcionalidades para trabajar con cadenas de texto.
    iv. algorithm: Contiene algoritmos genéricos para realizar operaciones comunes en contenedores, como la búsqueda, ordenación y manipulación de elementos.
    v. container classes: Incluye clases de contenedores como list, deque, stack, queue, etc., que ofrecen diferentes estructuras de datos para almacenar elementos.

vi. entre otras.

- Biblioteca de entrada y salida estándar (stdio.h):
    i. printf, scanf: Permiten realizar operaciones de entrada y salida formateada en la consola.
    ii. fopen, fclose, fprintf, fscanf: Proporcionan funcionalidades para trabajar con archivos de texto.
    iii. fread, fwrite: Permiten leer y escribir datos binarios en archivos.
- Biblioteca de funciones matemáticas (cmath):
    i. sin, cos, tan: Funciones trigonométricas.
    ii. sqrt: Calcula la raíz cuadrada de un número.
    iii. pow: Calcula la potencia de un número.
    iv. abs: Calcula el valor absoluto de un número.
    v. entre otras.
- Biblioteca de manipulación de cadenas (cstring):
    i. strcpy, strncpy: Copian cadenas de texto.
    ii. strcat, strncat: Concatenan cadenas de texto.
    iii. strcmp, strncmp: Comparan cadenas de texto.
    iv. strlen: Calcula la longitud de una cadena.
    v. entre otras.
- Biblioteca de manipulación de archivos (fstream):
    i. ifstream: Permite la lectura de archivos de texto.
    ii. ofstream: Permite la escritura en archivos de texto.
    iii. fstream: Proporciona funcionalidades para leer y escribir archivos de texto.
    iv. entre otras.

4. Entrada y salida de flujos

- Flujos de entrada (input streams):
    i. cin: Es el flujo de entrada estándar utilizado para leer datos desde la consola. Permite la lectura de diferentes tipos de datos, como enteros, números en coma flotante, caracteres y cadenas, utilizando el operador de extracción (>>).
- Flujos de salida (output streams):

i. cout: Es el flujo de salida estándar utilizado para imprimir datos en la consola. Permite la escritura de diferentes tipos de datos utilizando el operador de inserción (<<). Se utiliza comúnmente para mostrar resultados y mensajes en la consola.
ii. cerr: Es el flujo de salida estándar para mensajes de error. Al igual que cout, se utiliza para imprimir mensajes en la consola, pero se recomienda su uso específicamente para mensajes de error. No se realiza ninguna operación de formato automático en cerr.
iii. clog: Es el flujo de salida estándar para mensajes de registro (logging). Se utiliza para imprimir mensajes de registro o seguimiento en la consola. Similar a cerr, no se realiza ninguna operación de formato automático en clog.

- Flujos de archivos (file streams):
  i. ifstream: Es un flujo de entrada utilizado para leer datos desde un archivo de texto. Permite la lectura de datos en diferentes formatos desde un archivo.
  ii. ofstream: Es un flujo de salida utilizado para escribir datos en un archivo de texto. Permite la escritura de datos en diferentes formatos en un archivo.
  iii. fstream: Es un flujo de archivos que combina las funcionalidades de ifstream y ofstream, lo que permite tanto la lectura como la escritura de datos en un archivo.

5. Programación estructurada

- Los principios básicos de la programación estructurada incluyen:
  i. Secuencia: El programa se organiza en una secuencia de instrucciones que se ejecutan en orden. Cada instrucción se ejecuta después de la anterior, controlando el flujo del programa.
  ii. Selección: Se utilizan estructuras de control condicionales, como la instrucción if-else, para tomar decisiones en función de condiciones lógicas. Esto permite que el programa realice diferentes acciones según los resultados de las evaluaciones.
  iii. Repetición: Se utilizan estructuras de control iterativas, como los bucles (for, while), para repetir un conjunto de instrucciones varias

veces. Esto permite ejecutar una secuencia de instrucciones de forma repetitiva hasta que se cumpla una condición específica.

- Ventajas de la programación estructurada.
    i. Legibilidad: Los programas estructurados son más fáciles de leer y entender debido a la organización lógica de las instrucciones y el uso de estructuras de control claras.
    ii. Mantenibilidad: La división del programa en módulos más pequeños y la claridad del flujo de ejecución facilitan el mantenimiento y la modificación del código.
    iii. Depuración: La estructura bien definida del programa facilita la identificación y solución de errores (bugs) durante el proceso de depuración.
    iv. Reutilización: Los módulos lógicos pueden ser reutilizados en diferentes partes del programa o en otros proyectos, lo que ahorra tiempo y esfuerzo en el desarrollo de software.
    v. Eficiencia: La programación estructurada puede generar código eficiente y optimizado, ya que se evitan redundancias y se aprovechan las estructuras de control adecuadas.

6. Programación orientada a objetos

- Clases: Las clases son estructuras que definen las propiedades y comportamientos de los objetos. Una clase puede considerarse como un plano o molde a partir del cual se crean los objetos. Define los atributos (datos) y métodos (funciones) que los objetos pueden tener.
- Objetos: Los objetos son instancias concretas de una clase. Cada objeto tiene su propio conjunto de datos y puede realizar operaciones utilizando los métodos definidos en su clase. Los objetos pueden interactuar entre sí a través del intercambio de mensajes.
- Encapsulamiento: El encapsulamiento es el proceso de ocultar los detalles internos de un objeto y exponer solo una interfaz pública que permite interactuar con el objeto. Esto se logra mediante el uso de modificadores de acceso, como public, private y protected, que definen el nivel de visibilidad de los miembros de la clase.
- Abstracción: La abstracción consiste en identificar las características

esenciales y relevantes de un objeto y omitir los detalles irrelevantes. Permite representar conceptos del mundo real en forma de clases y objetos, simplificando la complejidad y facilitando el diseño y desarrollo de software.

- Herencia: La herencia es un mecanismo que permite que una clase herede las propiedades y comportamientos de otra clase. La clase que se hereda se conoce como superclase o clase base, y la clase que hereda se conoce como subclase o clase derivada. La herencia permite la reutilización de código y facilita la organización jerárquica de las clases.
- Polimorfismo: El polimorfismo se refiere a la capacidad de objetos de diferentes clases de responder a un mismo mensaje de diferentes maneras. Permite tratar diferentes tipos de objetos de manera uniforme a través del uso de interfaces comunes y la capacidad de redefinir métodos en las subclases..

7. Propiedades y métodos de una clase

- Propiedades (atributos o variables miembro): Las propiedades son variables que representan el estado o las características de un objeto. Cada objeto creado a partir de una clase tiene su propia copia de estas propiedades. Las propiedades pueden tener diferentes tipos de datos y se definen dentro de la clase. Algunos ejemplos comunes de propiedades son:
  i. Variables primitivas: como enteros, números en coma flotante, caracteres, booleanos, etc.
  ii. Objetos: como instancias de otras clases.
  iii. Arreglos: colecciones de elementos del mismo tipo.
- Las propiedades se declaran en la sección de variables miembro de la clase y pueden tener diferentes niveles de acceso, como public, private o protected, para controlar su visibilidad y manipulación desde fuera de la clase.
- Métodos (funciones miembro): Los métodos son funciones que definen el comportamiento de un objeto. Representan las operaciones que un objeto puede realizar o las acciones que se pueden realizar sobre él. Los métodos se definen dentro de la clase y pueden acceder y manipular las propiedades del objeto.

  Los métodos pueden recibir argumentos (parámetros) y pueden devolver un resultado. Pueden tener diferentes niveles de acceso y se definen en la sección de funciones miembro de la clase. Algunos ejemplos comunes de métodos

son:

i. Métodos de acceso (getters y setters): utilizados para acceder y modificar las propiedades privadas de un objeto.
ii. Métodos de cálculo: realizan operaciones o cálculos basados en las propiedades del objeto.
iii. Métodos de manipulación: realizan acciones que afectan el estado del objeto.
iv. Métodos de comparación: comparan el objeto actual con otro objeto.
v. Métodos de impresión o visualización: muestran información sobre el objeto en la consola o en algún otro medio.

- Los métodos se invocan en un objeto específico utilizando la sintaxis del punto (objeto.metodo()).

8. Creación de objetos

- La creación de objetos en la programación orientada a objetos se realiza mediante la instanciación de una clase. Un objeto es una instancia concreta de una clase, es decir, es una entidad real que se crea basándose en la plantilla definida por la clase. Para crear un objeto, se siguen los siguientes pasos:
    i. Definir una clase: Primero, se debe tener una clase definida que describa las propiedades y métodos que tendrá el objeto. La clase es como un plano o molde que define la estructura y comportamiento del objeto.
    ii. Declarar una variable del tipo de la clase: Se declara una variable que será el objeto en sí. La variable se declara utilizando el nombre de la clase como tipo de dato y se puede asignar a un identificador específico.
    iii. Instanciar la clase mediante el operador new: Para crear el objeto, se utiliza el operador new seguido del nombre de la clase y paréntesis (). Esto asigna memoria para el objeto y llama al constructor de la clase, que es un método especial utilizado para inicializar el objeto.
    iv. Asignar el objeto a la variable declarada: El resultado de la instanciación (operador new) se asigna a la variable declarada, lo que hace que la variable sea una referencia al objeto recién creado.

Unidad 2: Operadores lógicos y estructuras de control

En esta unidad, se exploran los operadores lógicos y las estructuras de control disponibles en C++. Los temas abordados son:

- **Operadores lógicos**
    - DLos operadores lógicos son utilizados en programación para evaluar expresiones lógicas y realizar operaciones de comparación entre valores booleanos. Los operadores lógicos más comunes son:
        i. Operador AND (&&): Este operador devuelve true si ambas expresiones evaluadas son verdaderas, y false en cualquier otro caso. Su sintaxis es:

        cpp

        ```
        expresion1 && expresion2
        ```

        ii. Ejemplo:

        ```
        cpp
        bool a = true;
        bool b = false;
        bool resultado = a && b;  // resultado es false

        ```

        6. Operador OR (||): Este operador devuelve true si al menos una de las expresiones evaluadas es verdadera, y false solo si ambas expresiones son falsas. Su sintaxis es:

        cpp

        ```
        expresion1 || expresion2
        ```

        iii. Ejemplo:

        ```
        cpp
        bool a = true;
        bool b = false;
        bool resultado = a || b;  // resultado es true
        ```

        5. Operador NOT (!): Este operador invierte el valor de una expresión booleana. Si la expresión es verdadera, el operador

NOT la convierte en falsa, y si la expresión es falsa, el operador NOT la convierte en verdadera. Su sintaxis es:

cpp

```
!expresion
```

iv. Ejemplo:

```
cpp
bool a = true;
bool resultado = !a;  // resultado es false
```

v. Estos operadores lógicos se utilizan frecuentemente en estructuras condicionales (como if y while) para evaluar condiciones y tomar decisiones en función de los valores booleanos resultantes. También se pueden combinar y anidar para construir expresiones lógicas más complejas.

vi. Es importante tener en cuenta que los operadores lógicos siguen reglas de precedencia y asociatividad, al igual que otros operadores en C++. Si es necesario, se pueden utilizar paréntesis para agrupar expresiones y establecer el orden de evaluación deseado.

- **Estructuras de control**
  - Las estructuras de control en C++ son utilizadas para controlar el flujo de ejecución de un programa, permitiendo la toma de decisiones y la repetición de bloques de código. Las estructuras de control más comunes son:
    i. Estructura de control IF: Permite ejecutar un bloque de código si se cumple una determinada condición. Su sintaxis es:

    cpp

```
if (condicion) {
    // Código a ejecutar si la condición es verdadera
} else {
    // Código a ejecutar si la condición es falsa
}

```

    ii. Ejemplo:

```
cpp
int edad = 18;
```

```
3.
4. if (edad >= 18) {
5.     cout << "Eres mayor de edad." << endl;
6. } else {
7.     cout << "Eres menor de edad." << endl;
8. }
9.
```

10. Estructura de control IF anidado: Permite anidar múltiples bloques IF dentro de otro bloque IF. Esto permite evaluar condiciones adicionales en caso de que la condición anterior sea verdadera. Su sintaxis es:

    cpp

```
11. if (condicion1) {
12.     // Código a ejecutar si la condicion1 es verdadera
13.
14.     if (condicion2) {
15.         // Código a ejecutar si tanto la condicion1 como la condicion2 son verdaderas
16.     }
17. }
18.
```

iii. Ejemplo:

```
1. cpp
2. int edad = 18;
3. bool tieneLicencia = true;
4.
5. if (edad >= 18) {
6.     if (tieneLicencia) {
7.         cout << "Puede conducir." << endl;
8.     } else {
9.         cout << "No puede conducir sin licencia." << endl;
10.    }
```

```
}
```

13. Estructura de control FOR: Permite ejecutar un bloque de código un número determinado de veces. Se compone de tres partes: la inicialización, la condición de terminación y la actualización. Su sintaxis es:

    cpp

```
for (inicializacion; condicion; actualizacion) {
    // Código a ejecutar en cada iteración
}
```

iv. Ejemplo:

```
cpp
for (int i = 1; i <= 5; i++) {
    cout << "El valor de i es: " << i << endl;
}
```

6. Estructura de control WHILE: Permite ejecutar un bloque de código mientras se cumpla una determinada condición. Su sintaxis es:

   cpp

```
while (condicion) {
    // Código a ejecutar mientras la condición sea verdadera
}
```

v. Ejemplo:

```
cpp
int contador = 0;

while (contador < 5) {
    cout << "El valor del contador es: " << contador << endl;
    contador++;
}
```

8.
9. Estructura de control SWITCH: Permite evaluar diferentes casos y ejecutar un bloque de código dependiendo del valor de una expresión. Su sintaxis es:

   cpp

```
switch (expresion) {
    case valor1:
        // Código a ejecutar si la expresion es igual a valor1
        break;
    case valor2:
        // Código a ejecutar si la expresion es igual a valor2
        break;
    default:
        // Código a ejecutar si la expresion no coincide con ninguno de los valores anteriores
        break;
}
```

21.

vi. Ejemplo:

```
cpp
int opcion = 2;

switch (opcion) {
    case 1:
        cout << "Seleccionaste la opción 1." << endl;
        break;
    case 2:
        cout << "Seleccionaste la opción 2." << endl;
        break;
    default:
        cout << "Opción inválida." << endl;
        break;
```

```
14. }
```

- Estas estructuras de control son fundamentales para controlar el flujo de ejecución de un programa y permitir la toma de decisiones y la repetición de bloques de código según sea necesario.

- Diferencia entre operadores de igualdad y de asignación
    i. EEn C++, los operadores de igualdad (==) y de asignación (=) son utilizados para propósitos diferentes:
- Operador de igualdad (==):
    i. El operador de igualdad se utiliza para comparar si dos valores son iguales.
    ii. Retorna true si los valores son iguales y false en caso contrario.
    iii. Se utiliza en expresiones de comparación dentro de estructuras condicionales (como if, while, etc.) y en expresiones lógicas.
    iv. Ejemplo:

    cpp

```
v.  int a = 5;
vi. int b = 7;
vii.
viii.       if (a == b) {
ix.     cout << "a y b son iguales." << endl;
x.  } else {
xi.     cout << "a y b son diferentes." << endl;
xii. }
```

-
    i. Operador de asignación (=):
- El operador de asignación se utiliza para asignar un valor a una variable.
- Toma el valor de la expresión a la derecha y lo asigna a la variable a la izquierda.
- Es importante tener en cuenta que la asignación no verifica igualdad, simplemente copia el valor.
- Ejemplo:

  cpp

    i. int a = 5;

ii. int b;
iii.
iv. b = a; // Se asigna el valor de a a la variable b
v.
vi. cout << "El valor de b es: " << b << endl;
vii. En este caso, se asigna el valor de a a la variable b utilizando el operador de asignación (=).

- En resumen, la diferencia fundamental entre el operador de igualdad (==) y el operador de asignación (=) radica en su funcionalidad. El operador de igualdad compara dos valores, mientras que el operador de asignación asigna un valor a una variable.

- **Recursividad**
  - La recursividad es un concepto en programación que se refiere a la capacidad de una función de llamarse a sí misma durante su propia ejecución. En otras palabras, una función recursiva es aquella que se llama a sí misma para resolver un problema de manera iterativa.
    i. La recursividad se basa en dos aspectos clave:
    ii. Caso base: Es el caso más simple o básico que no requiere de llamadas recursivas. Define el punto de término de la recursividad y evita que la función se llame infinitamente.
    iii. Caso recursivo: Es la parte en la que se llama a la función nuevamente, pero con un problema más pequeño o una instancia simplificada del problema original. Se espera que esta llamada recursiva se acerque al caso base en cada iteración, permitiendo así la resolución del problema completo.
    iv. Es importante asegurarse de que la llamada recursiva se realice correctamente y se acerque al caso base en cada iteración. De lo contrario, se podría producir una recursión infinita, lo que llevaría al desbordamiento de la pila de llamadas y al término abrupto del programa.
    v. La recursividad se utiliza en situaciones donde un problema puede ser dividido en subproblemas más pequeños y se puede aplicar la misma lógica para resolver cada subproblema de forma individual. Algunos

ejemplos comunes de uso de la recursividad son:

vi. Cálculo de factoriales.

vii. Implementación de algoritmos de búsqueda y ordenamiento (como la búsqueda binaria o el ordenamiento quicksort).

viii. Recorrido de estructuras de datos jerárquicas (como árboles y grafos) mediante llamadas recursivas a los nodos hijos.

ix. Resolución de problemas matemáticos complejos (como la sucesión de Fibonacci).

- **Manejo de excepciones**
  - El manejo de excepciones en C++ es una técnica que permite detectar y controlar situaciones excepcionales o errores que ocurren durante la ejecución de un programa. Estas excepciones pueden ser errores en tiempo de ejecución, condiciones inesperadas o situaciones que impiden que el programa continúe su ejecución normal.
  - El manejo de excepciones se basa en tres componentes principales:
  - Lanzamiento de excepciones: Un programa puede lanzar una excepción utilizando la palabra clave throw seguida de un objeto que representa la excepción. Esto indica que se ha producido una situación excepcional y que el programa no puede continuar su ejecución normal.

    Ejemplo:

    cpp

```
i.   void dividir(int numerador, int denominador) {
ii.      if (denominador == 0) {
iii.         throw std::runtime_error("División por cero");
iv.      }
v.       int resultado = numerador / denominador;
vi.      std::cout << "El resultado de la división es: " << resultado <<
     std::endl;
vii. }
```

    viii. En este ejemplo, la función dividir() lanza una excepción utilizando throw si el denominador es igual a cero.

ix. Captura de excepciones: Para manejar las excepciones lanzadas, se utiliza un bloque try-catch. El bloque try contiene el código donde se espera que se produzcan excepciones, y el bloque catch captura y maneja esas excepciones.

Ejemplo:

cpp

```
x.  int main() {
xi.     try {
xii.        int numerador = 10;
xiii.               int denominador = 0;
xiv.                dividir(numerador, denominador);
xv.     } catch (const std::runtime_error& e) {
xvi.                std::cout << "Error: " << e.what() << std::endl;
xvii.           }
xviii.          return 0;
xix.        }
```

xx. En este caso, el bloque try contiene la llamada a la función dividir() que puede lanzar una excepción. El bloque catch captura la excepción del tipo std::runtime_error (u otro tipo compatible) y maneja la excepción imprimiendo un mensaje de error.

xxi. Bloques catch múltiples: Es posible tener varios bloques catch para manejar diferentes tipos de excepciones. Los bloques catch se evalúan en orden y el primero que coincide con el tipo de excepción lanzada se ejecuta.

Ejemplo:

cpp

```
xxii.       int main() {
xxiii.          try {
xxiv.               // Código que puede lanzar excepciones
xxv.            } catch (const std::runtime_error& e) {
xxvi.               // Manejo de excepción std::runtime_error
xxvii.          } catch (const std::logic_error& e) {
xxviii.             // Manejo de excepción std::logic_error
```

```
xxix.           } catch (...) {
xxx.                // Manejo de cualquier otra excepción
xxxi.           }
xxxii.          return 0;
xxxiii.     }
```

- En este ejemplo, se definen tres bloques catch para manejar diferentes tipos de excepciones. El último bloque catch (...) se utiliza para capturar cualquier otra excepción que no haya sido capturada por los bloques anteriores.
- El manejo de excepciones permite separar la lógica de detección y manejo de errores del flujo principal del programa, lo que facilita el mantenimiento y mejora la legibilidad del código. Además, proporciona una forma estructurada de manejar situaciones excepcionales y tomar acciones adecuadas en consecuencia.
- Es importante tener en cuenta que el manejo de excepciones debe utilizarse de manera adecuada y selectiva, enfocándose en situaciones excepcionales genuinas y evitando abusos que puedan afectar negativamente el rendimiento del programa.

- **Procesamiento de archivos**
  - El procesamiento de archivos en C++ permite leer y escribir información en archivos de datos externos. Esto es útil para almacenar y recuperar datos persistentes, interactuar con archivos de texto, realizar operaciones de lectura y escritura en binario, entre otras tareas relacionadas con el manejo de archivos.
  - En C++, existen varias clases y funciones que proporcionan funcionalidades para el procesamiento de archivos. Algunas de las más comunes son:
    i. std::ofstream (output file stream): Esta clase se utiliza para escribir datos en un archivo. Proporciona métodos para abrir, cerrar y escribir en un archivo.
    ii. std::ifstream (input file stream): Esta clase se utiliza para leer datos de un archivo. Proporciona métodos para abrir, cerrar y leer desde un archivo.
    iii. std::fstream (file stream): Esta clase combina las funcionalidades de std::ofstream y std::ifstream, permitiendo realizar operaciones de

lectura y escritura en un archivo.

Unidad 3: Arreglos, búsqueda y ordenamiento

En esta unidad, se estudian los arreglos, su declaración y manipulación, así como los algoritmos de búsqueda y ordenamiento. Los temas incluyen:

- **Arreglos y vectores**
    - Los arreglos y los vectores son estructuras de datos utilizadas en C++ para almacenar y manipular conjuntos de elementos del mismo tipo. Ambos permiten acceder a los elementos mediante un índice y ofrecen capacidades para gestionar conjuntos de datos de manera eficiente. Sin embargo, existen algunas diferencias importantes entre ellos.

  Arreglos:

    - Un arreglo en C++ es una colección contigua de elementos del mismo tipo, donde cada elemento se accede mediante un índice entero.
    - La longitud de un arreglo se determina en tiempo de compilación y no puede cambiar durante la ejecución del programa.
    - La declaración de un arreglo requiere especificar su tipo y tamaño.
    - Uso de vectores de la biblioteca estándar para trabajar con arreglos dinámicos.

  Vectores:

    - Un vector es una clase de la biblioteca estándar de C++ (std::vector) que proporciona una implementación dinámica de un arreglo.
    - Los vectores son contenedores flexibles que pueden cambiar de tamaño durante la ejecución del programa.
    - Pueden aumentar o reducir su capacidad automáticamente según sea necesario.
    - Los vectores ofrecen métodos y funciones integradas para facilitar la manipulación y gestión de los elementos.
- **Ejemplos del uso de arreglos**
    - Suma de elementos de un arreglo:
        - cpp
        - #include <iostream>

```
int main() {
    int arreglo[] = {5, 10, 15, 20, 25};
    int suma = 0;

    for (int i = 0; i < 5; i++) {
        suma += arreglo[i];
    }

    std::cout << "La suma de los elementos del arreglo es: " << suma << std::endl;

    return 0;
}
```

- Búsqueda de un elemento en un arreglo:

```
cpp
#include <iostream>

int main() {
    int arreglo[] = {5, 10, 15, 20, 25};
    int elementoBuscado = 15;
    bool encontrado = false;

    for (int i = 0; i < 5; i++) {
        if (arreglo[i] == elementoBuscado) {
            encontrado = true;
            break;
        }
    }

    if (encontrado) {
```

```
        std::cout << "El elemento " << elementoBuscado << " se encuentra en el arreglo." << std::endl;
    } else {
        std::cout << "El elemento " << elementoBuscado << " no se encuentra en el arreglo." << std::endl;
    }

    return 0;
}
```

- Copia de elementos de un arreglo a otro:

```
cpp
#include <iostream>

int main() {
    int arreglo1[] = {1, 2, 3, 4, 5};
    int arreglo2[5];

    for (int i = 0; i < 5; i++) {
        arreglo2[i] = arreglo1[i];
    }

    std::cout << "Elementos del arreglo2: ";
    for (int i = 0; i < 5; i++) {
        std::cout << arreglo2[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

- **Arreglos a funciones**
  - En C++, los arreglos pueden ser pasados a funciones de varias formas, ya sea

como un arreglo estático o como un puntero. Aquí tienes algunos ejemplos de cómo pasar arreglos a funciones:

- Pasar un arreglo como un puntero:
  - cpp
  - ```
    #include <iostream>

    // Función que toma un arreglo y su tamaño como parámetros
    void imprimirArreglo(int* arreglo, int tamano) {
        for (int i = 0; i < tamano; i++) {
            std::cout << arreglo[i] << " ";
        }
        std::cout << std::endl;
    }

    int main() {
        int arreglo[] = {1, 2, 3, 4, 5};

        imprimirArreglo(arreglo, 5);

        return 0;
    }
    ```
- Pasar un arreglo como un arreglo estático:
  - cpp
  - ```
    #include <iostream>

    // Función que toma un arreglo y su tamaño como parámetros
    void imprimirArreglo(int arreglo[], int tamano) {
        for (int i = 0; i < tamano; i++) {
            std::cout << arreglo[i] << " ";
        }
        std::cout << std::endl;
    ```

```
}

int main() {
    int arreglo[] = {1, 2, 3, 4, 5};

    imprimirArreglo(arreglo, 5);

    return 0;
}
```

- Usar un parámetro de plantilla para pasar un arreglo:

```
cpp
#include <iostream>

// Función de plantilla que toma un arreglo y su tamaño como parámetros
template <std::size_t N>
void imprimirArreglo(int (&arreglo)[N]) {
    for (int i = 0; i < N; i++) {
        std::cout << arreglo[i] << " ";
    }
    std::cout << std::endl;
}

int main() {
    int arreglo[] = {1, 2, 3, 4, 5};

    imprimirArreglo(arreglo);

    return 0;
}
```

- **Búsqueda de datos en arreglos**

- En C++, puedes realizar la búsqueda de datos en arreglos utilizando diferentes enfoques, como una búsqueda lineal o una búsqueda binaria si el arreglo está ordenado. Aquí tienes ejemplos de ambas:
- Búsqueda lineal:

  cpp

  ```
  #include <iostream>

  // Función que realiza una búsqueda lineal en un arreglo
  // Devuelve el índice del elemento encontrado o -1 si no se encuentra
  int busquedaLineal(int arreglo[], int tamano, int elemento) {
      for (int i = 0; i < tamano; i++) {
          if (arreglo[i] == elemento) {
              return i;  // Elemento encontrado, se devuelve su índice
          }
      }

      return -1;  // Elemento no encontrado
  }

  int main() {
      int arreglo[] = {5, 10, 15, 20, 25};
      int tamano = 5;
      int elementoBuscado = 15;

      int indice = busquedaLineal(arreglo, tamano, elementoBuscado);

      if (indice != -1) {
          std::cout << "El elemento " << elementoBuscado << " se
  encuentra en el índice " << indice << std::endl;
      } else {
          std::cout << "El elemento " << elementoBuscado << " no se
  ```

```
    encuentra en el arreglo." << std::endl;
    }

    return 0;
}
```

- Búsqueda binaria (en un arreglo ordenado):

```
cpp

#include <iostream>

// Función que realiza una búsqueda binaria en un arreglo ordenado
// Devuelve el índice del elemento encontrado o -1 si no se encuentra
int busquedaBinaria(int arreglo[], int tamano, int elemento) {
    int inicio = 0;
    int fin = tamano - 1;

    while (inicio <= fin) {
        int medio = inicio + (fin - inicio) / 2;

        if (arreglo[medio] == elemento) {
            return medio;  // Elemento encontrado, se devuelve su índice
        } else if (arreglo[medio] < elemento) {
            inicio = medio + 1;  // El elemento está en la mitad derecha
        } else {
            fin = medio - 1;  // El elemento está en la mitad izquierda
        }
    }

    return -1;  // Elemento no encontrado
}

int main() {
```

```
    int arreglo[] = {10, 20, 30, 40, 50};
    int tamano = 5;
    int elementoBuscado = 30;

    int indice = busquedaBinaria(arreglo, tamano, elementoBuscado);

    if (indice != -1) {
        std::cout << "El elemento " << elementoBuscado << " se encuentra en el índice " << indice << std::endl;
    } else {
        std::cout << "El elemento " << elementoBuscado << " no se encuentra en el arreglo." << std::endl;
    }

    return 0;

}
```

- **Ordenamiento de arreglos**
  - En C++, hay varios algoritmos populares para ordenar arreglos, como el ordenamiento burbuja, el ordenamiento por inserción, el ordenamiento por selección, el ordenamiento rápido (quicksort) y el ordenamiento por mezcla (mergesort). A continuación, te mostraré ejemplos de implementaciones para algunos de estos algoritmos:
    - Ordenamiento burbuja:

```
cpp

#include <iostream>

// Función para intercambiar dos elementos
void intercambiar(int& a, int& b) {
    int temp = a;
    a = b;
```

```
    b = temp;
}

// Función para ordenar un arreglo utilizando el ordenamiento burbuja
void ordenamientoBurbuja(int arreglo[], int tamano) {
    for (int i = 0; i < tamano - 1; i++) {
        for (int j = 0; j < tamano - i - 1; j++) {
            if (arreglo[j] > arreglo[j + 1]) {
                intercambiar(arreglo[j], arreglo[j + 1]);
            }
        }
    }
}

int main() {
    int arreglo[] = {5, 3, 8, 2, 1};
    int tamano = sizeof(arreglo) / sizeof(arreglo[0]);

    ordenamientoBurbuja(arreglo, tamano);

    std::cout << "Arreglo ordenado: ";
    for (int i = 0; i < tamano; i++) {
        std::cout << arreglo[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

- Ordenamiento por inserción:

  cpp

```
#include <iostream>

// Función para ordenar un arreglo utilizando el ordenamiento por inserción
void ordenamientoInsercion(int arreglo[], int tamano) {
    for (int i = 1; i < tamano; i++) {
        int valorActual = arreglo[i];
        int j = i - 1;

        while (j >= 0 && arreglo[j] > valorActual) {
            arreglo[j + 1] = arreglo[j];
            j--;
        }

        arreglo[j + 1] = valorActual;
    }
}

int main() {
    int arreglo[] = {5, 3, 8, 2, 1};
    int tamano = sizeof(arreglo) / sizeof(arreglo[0]);

    ordenamientoInsercion(arreglo, tamano);

    std::cout << "Arreglo ordenado: ";
    for (int i = 0; i < tamano; i++) {
        std::cout << arreglo[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

- Ordenamiento rápido (quicksort):

  cpp

  ```
  #include <iostream>

  // Función para intercambiar dos elementos
  void intercambiar(int& a, int& b) {
      int temp = a;
      a = b;
      b = temp;
  }

  // Función para dividir el arreglo y colocar el pivote en su posición correcta
  int particion(int arreglo[], int inicio, int fin) {
      int pivote = arreglo[fin];
      int i = inicio - 1;

      for (int j = inicio; j < fin; j++) {
          if (arreglo[j] <= pivote) {
              i++;
              intercambiar(arreglo[i], arreglo[j]);
          }
      }

      intercambiar(arreglo[i + 1], arreglo[fin]);
      return i + 1;
  }

  // Función para ordenar un arreglo utilizando el ordenamiento rápido (quicksort)
  void ordenamientoQuicksort(int arreglo[], int inicio, int fin) {
  ```

```
    if (inicio < fin) {
        int indicePivote = particion(arreglo, inicio, fin);

        ordenamientoQuicksort(arreglo, inicio, indicePivote - 1);
        ordenamientoQuicksort(arreglo, indicePivote + 1, fin);
    }
}

int main() {
    int arreglo[] = {5, 3, 8, 2, 1};
    int tamano = sizeof(arreglo) / sizeof(arreglo[0]);

    ordenamientoQuicksort(arreglo, 0, tamano - 1);

    std::cout << "Arreglo ordenado: ";
    for (int i = 0; i < tamano; i++) {
        std::cout << arreglo[i] << " ";
    }
    std::cout << std::endl;

    return 0;

}
```

- **Arreglos multidimensionales**
    - En C++, los arreglos multidimensionales permiten almacenar datos en una estructura de matriz de múltiples dimensiones. Puedes crear arreglos bidimensionales (matrices), tridimensionales y de dimensiones superiores. A continuación, te muestro ejemplos de cómo declarar, acceder y trabajar con arreglos multidimensionales:
- Arreglo bidimensional (matriz):

cpp

```cpp
#include <iostream>

const int FILAS = 3;
const int COLUMNAS = 4;

int main() {
    // Declaración y asignación de una matriz de 3x4
    int matriz[FILAS][COLUMNAS] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };

    // Acceso y modificación de elementos de la matriz
    matriz[0][2] = 30;
    int elemento = matriz[1][3];

    // Recorrido e impresión de la matriz
    for (int i = 0; i < FILAS; i++) {
        for (int j = 0; j < COLUMNAS; j++) {
            std::cout << matriz[i][j] << " ";
        }
        std::cout << std::endl;
    }

    return 0;
}
```

- Arreglo tridimensional:

cpp

```cpp
#include <iostream>

```

```
const int DIMENSION1 = 2;
const int DIMENSION2 = 3;
const int DIMENSION3 = 4;

int main() {
    // Declaración y asignación de un arreglo tridimensional de 2x3x4
    int arreglo[DIMENSION1][DIMENSION2][DIMENSION3] = {
        {
            {1, 2, 3, 4},
            {5, 6, 7, 8},
            {9, 10, 11, 12}
        },
        {
            {13, 14, 15, 16},
            {17, 18, 19, 20},
            {21, 22, 23, 24}
        }
    };

    // Acceso y modificación de elementos del arreglo
    arreglo[1][0][2] = 30;
    int elemento = arreglo[0][1][3];

    // Recorrido e impresión del arreglo
    for (int i = 0; i < DIMENSION1; i++) {
        for (int j = 0; j < DIMENSION2; j++) {
            for (int k = 0; k < DIMENSION3; k++) {
                std::cout << arreglo[i][j][k] << " ";
            }
            std::cout << std::endl;
        }
        std::cout << std::endl;
```

        - }
        -
        - return 0;
        - }
    - Puedes extender estos ejemplos para trabajar con arreglos de dimensiones superiores, simplemente agregando más índices en la declaración y el acceso a los elementos.

- **Búsqueda y ordenamiento**

Para realizar la búsqueda y el ordenamiento en arreglos, existen varios algoritmos populares que puedes utilizar en C++. A continuación, te mostraré ejemplos de algoritmos de búsqueda y ordenamiento:

- Búsqueda en un arreglo:
- a) Búsqueda lineal:
    - cpp
    - #include <iostream>
    -
    - // Función para realizar una búsqueda lineal en un arreglo
    - int busquedaLineal(int arreglo[], int tamano, int elemento) {
    - for (int i = 0; i < tamano; i++) {
    - if (arreglo[i] == elemento) {
    - return i; // Retorna el índice donde se encuentra el elemento
    - }
    - }
    - return -1; // Retorna -1 si el elemento no se encuentra en el arreglo
    - }
    -
    - int main() {
    - int arreglo[] = {5, 3, 8, 2, 1};
    - int tamano = sizeof(arreglo) / sizeof(arreglo[0]);
    - int elemento = 8;
    -

```
    int indice = busquedaLineal(arreglo, tamano, elemento);

    if (indice != -1) {
        std::cout << "El elemento " << elemento << " se encuentra en el índice " << indice << std::endl;
    } else {
        std::cout << "El elemento " << elemento << " no se encuentra en el arreglo" << std::endl;
    }

    return 0;
}
```

- b) Búsqueda binaria (requiere que el arreglo esté ordenado):

```
cpp
#include <iostream>

// Función para realizar una búsqueda binaria en un arreglo ordenado
int busquedaBinaria(int arreglo[], int inicio, int fin, int elemento) {
    if (inicio <= fin) {
        int medio = inicio + (fin - inicio) / 2;

        if (arreglo[medio] == elemento) {
            return medio; // Retorna el índice donde se encuentra el elemento
        }

        if (arreglo[medio] < elemento) {
            return busquedaBinaria(arreglo, medio + 1, fin, elemento);
        }

        return busquedaBinaria(arreglo, inicio, medio - 1, elemento);
    }
```

```
    return -1; // Retorna -1 si el elemento no se encuentra en el arreglo
}

int main() {
    int arreglo[] = {1, 2, 3, 5, 8};
    int tamano = sizeof(arreglo) / sizeof(arreglo[0]);
    int elemento = 5;

    int indice = busquedaBinaria(arreglo, 0, tamano - 1, elemento);

    if (indice != -1) {
        std::cout << "El elemento " << elemento << " se encuentra en el índice " << indice << std::endl;
    } else {
        std::cout << "El elemento " << elemento << " no se encuentra en el arreglo" << std::endl;
    }

    return 0;
}
```

- Ordenamiento de un arreglo:
- a) Ordenamiento burbuja:
  - cpp

```
#include <iostream>

// Función para intercambiar dos elementos
void intercambiar(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
```

```
}

// Función para ordenar un arreglo utilizando el ordenamiento burbuja
void ordenamientoBurbuja(int arreglo[], int tamano) {
    for (int i = 0; i < tamano - 1; i++) {
        for (int j = 0; j < tamano - i - 1; j++) {
            if (arreglo[j] > arreglo[j + 1]) {
                intercambiar(arreglo[j], arreglo[j + 1]);
            }
        }
    }
}

int main() {
    int arreglo[] = {5, 3, 8, 2, 1};
    int tamano = sizeof(arreglo) / sizeof(arreglo[0]);

    ordenamientoBurbuja(arreglo, tamano);

    std::cout << "Arreglo ordenado: ";
    for (int i = 0; i < tamano; i++) {
        std::cout << arreglo[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

- b) Ordenamiento rápido (quicksort):

```
cpp
#include <iostream>
```

```
// Función para intercambiar dos elementos
void intercambiar(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

// Función para particionar el arreglo en torno a un pivote
int particion(int arreglo[], int inicio, int fin) {
    int pivote = arreglo[fin];
    int i = inicio - 1;

    for (int j = inicio; j <= fin - 1; j++) {
        if (arreglo[j] < pivote) {
            i++;
            intercambiar(arreglo[i], arreglo[j]);
        }
    }

    intercambiar(arreglo[i + 1], arreglo[fin]);

    return i + 1;
}

// Función para ordenar un arreglo utilizando el ordenamiento rápido (quicksort)
void ordenamientoQuicksort(int arreglo[], int inicio, int fin) {
    if (inicio < fin) {
        int indicePivote = particion(arreglo, inicio, fin);

        ordenamientoQuicksort(arreglo, inicio, indicePivote - 1);
        ordenamientoQuicksort(arreglo, indicePivote + 1, fin);
```

```
    }
}

int main() {
    int arreglo[] = {5, 3, 8, 2, 1};
    int tamano = sizeof(arreglo) / sizeof(arreglo[0]);

    ordenamientoQuicksort(arreglo, 0, tamano - 1);

    std::cout << "Arreglo ordenado: ";
    for (int i = 0; i < tamano; i++) {
        std::cout << arreglo[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Unidad 4: SQL (DML y DDL)

Esta unidad se centra en el lenguaje SQL y sus comandos para manipular bases de datos. Los temas tratados son:

1. **SQL (DML y DDL)**
    - SQL, que significa "Structured Query Language" (Lenguaje de Consulta Estructurado), es un lenguaje de programación utilizado para gestionar y manipular bases de datos relacionales. SQL se divide en dos categorías principales de comandos: DML (Data Manipulation Language, Lenguaje de Manipulación de Datos) y DDL (Data Definition Language, Lenguaje de Definición de Datos).
        i. DML (Data Manipulation Language): El lenguaje DML se utiliza para manipular y gestionar los datos almacenados en una base de datos. Algunos de los comandos DML más comunes son:
        ii. SELECT: Se utiliza para recuperar datos de una o varias tablas.

iii. INSERT: Se utiliza para insertar nuevos registros en una tabla.
iv. UPDATE: Se utiliza para actualizar los valores de uno o varios registros en una tabla.
v. DELETE: Se utiliza para eliminar uno o varios registros de una tabla.
vi. MERGE: Se utiliza para combinar (actualizar o insertar) datos de una tabla origen en una tabla destino.

2. DDL (Data Definition Language): El lenguaje DDL se utiliza para definir la estructura y características de los objetos de la base de datos, como tablas, índices, vistas, etc. Algunos de los comandos DDL más comunes son:

- CREATE: Se utiliza para crear objetos de la base de datos, como tablas, vistas, índices, etc.
- ALTER: Se utiliza para modificar la estructura de los objetos existentes en la base de datos.
- DROP: Se utiliza para eliminar objetos de la base de datos.
- TRUNCATE: Se utiliza para eliminar todos los datos de una tabla sin eliminar la estructura de la tabla en sí.

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

```
int arreglo[5] = {1, 2, 3, 4, 5};

// Acceso a elementos del arreglo utilizando la notación de corchetes []
cout << arreglo[0];  // Imprime el primer elemento del arreglo (1)
cout << arreglo[2];  // Imprime el tercer elemento del arreglo (3)

// Acceso a elementos del arreglo utilizando la aritmética de punteros
cout << *(arreglo + 1);  // Imprime el segundo elemento del arreglo (2)
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
    1. `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
    1. `int suma(int a, int b) {`
    2. `    return a + b;`
    3. `}`
    4.
    5. `int (*puntero_funcion)(int) = suma;`
-
- En este ejemplo, se asigna la dirección de la función `suma` al puntero `puntero_funcion`. Ahora, el puntero puede utilizarse para invocar la función utilizando la sintaxis de puntero a función:
- cpp
- `int resultado = puntero_funcion(3);`
-
- En este caso, se invoca la función apuntada por el puntero `puntero_funcion` 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
        1. `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

```
struct Nodo {
    tipo_dato dato;
    Nodo* siguiente;
};
```

-
- 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

```
+-------+     +-------+     +-------+
| dato1 | ---> | dato2 | ---> | dato3 |
+-------+     +-------+     +-------+
```

- 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 <array>, <vector>, <list>, <stack>, <queue>, <set>, <map>, 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

```
class Nodo {
public:
    int dato;
    Nodo* siguiente;

    Nodo(int dato) {
        this->dato = dato;
        siguiente = nullptr;
    }
};

class ListaEnlazada {
    private:
        Nodo* cabeza;
public:
        ListaEnlazada() {
            cabeza = nullptr;
        }

    void insertar(int dato) {
            Nodo* nuevoNodo = new Nodo(dato);
            if (cabeza == nullptr) {
                cabeza = nuevoNodo;
            } else {
                Nodo* temp = cabeza;
                while (temp->siguiente != nullptr) {
                    temp = temp->siguiente;
                }
                temp->siguiente = nuevoNodo;
            }
        }

```

```
    void imprimir() {
        Nodo* temp = cabeza;
        while (temp != nullptr) {
            cout << temp->dato << " ";
            temp = temp->siguiente;
        }
        cout << endl;
    }
};
```

-
- 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

```
class Nodo {
public:
    int dato;
    Nodo* siguiente;

    Nodo(int dato) {
        this->dato = dato;
        siguiente = nullptr;
    }
};

class ListaEnlazada {
    private:
        Nodo* cabeza;
public:
        ListaEnlazada() {
            cabeza = nullptr;
        }

    void insertar(int dato) {
        Nodo* nuevoNodo = new Nodo(dato);
        if (cabeza == nullptr) {
            cabeza = nuevoNodo;
        } else {
            Nodo* temp = cabeza;
```

```
            while (temp->siguiente != nullptr) {
                temp = temp->siguiente;
            }
            temp->siguiente = nuevoNodo;
        }
    }

    void imprimir() {
        Nodo* temp = cabeza;
        while (temp != nullptr) {
            cout << temp->dato << " ";
            temp = temp->siguiente;
        }
        cout << endl;
    }
};
```

- 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 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

```
class Nodo {
public:
    int dato;
    Nodo* siguiente;

    Nodo(int dato) {
        this->dato = dato;
                siguiente = nullptr;
    }
};
```

```
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 ListaEnlazada 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 ListaEnlazada 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;
```

```
                cima = cima->siguiente;
                delete temp;
            }
        }
    };

    class Cola {
    private:
        Nodo* frente;
        Nodo* fin;
    public:
        Cola() {
            frente = nullptr;
            fin = nullptr;
        }

        void enqueue(int dato) {
            Nodo* nuevoNodo = new Nodo(dato);
            if (frente == nullptr) {
                frente = nuevoNodo;
                fin = nuevoNodo;
            } else {
                fin->siguiente = nuevoNodo;
                fin = nuevoNodo;
            }
        }

        void dequeue() {
            if (frente == nullptr) {
                cout << "La cola está vacía" << endl;
            } else {
                Nodo* temp = frente;
```

```
            frente = frente->siguiente;
            delete temp;
        }
    }
};
```

- 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 enqueue 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.