

**UNIVERSIDAD DE LAS FUERZAS ARMADAS ESPE**

**DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN**

**ESTRUCTURA DE DATOS**

**MATERIA**

**NOMBRE:** QUISHPE CHAVEZ DENISSE PAULINA

**NRC:** 29852

**DOCENTE:** SOLIS FERNANDO

Tabla

Descripción generada automáticamente

Todo programar con objetos

1. **TIPOS DE DATOS ABSTRACTOS (TDA)**
2. **USO DE TDA**

Un Tipo de Dato Abstracto (TDA) es un modelo que define un conjunto de datos y las operaciones permitidas sobre ellos, sin especificar los detalles de cómo se implementan.

En otras palabras, un TDA se centra en qué hace y no en cómo lo hace.

En la programación con C++, los TDA se implementan principalmente mediante clases, aprovechando el principio de encapsulamiento propio de la programación orientada a objetos.

Esto permite que el usuario trabaje con los datos de forma segura, sin preocuparse por la estructura interna.

**Características principales de un TDA**

* Encapsulamiento: Los datos se ocultan dentro del TDA, protegiendo su estructura interna.
* Interfaz definida: El programador solo puede acceder a los datos mediante las funciones públicas del TDA.
* Reutilización: Un mismo TDA puede ser usado en diferentes programas.
* Modularidad: Facilita la organización del código en partes más comprensibles.
* Independencia: El usuario no necesita conocer la implementación para poder usarlo.

**Tipos de Datos Abstractos más comunes en C++**

|  |  |  |  |
| --- | --- | --- | --- |
| TDA | DESCRIPCION | OPERACIONES BASICAS | EJEMPLO DE USO |
| Lista (List) | Colección ordenada de elementos | Insetar, eliminar, bucar, recorrer. | Listado de estudiantes o productos. |
| Pila (Stack) | Estructura LIFO (último en entrar, primero en salir). | push(),  pop(),  top(),  empty(). | Deshacer acciones, control de llamadas de funciones. |
| Cola (Queue) | |  | | --- | |  |  |  | | --- | | Estructura FIFO (primero en entrar, primero en salir). | | enqueue(), dequeue(), front(), empty(). | Cola de impresión, gestión de turnos. |
| Árbol (Tree) | Estructura jerárquica con nodos. | Insertar, buscar, recorrer (pre, in, post). | Organización de archivos, árboles binarios. |
| Grafo (Graph) | Conjunto de nodos conectados por aristas. | Agregar vértice, agregar arista, buscar camino. | Mapas de rutas, redes sociales. |
| Conjunto (Set) | Colección sin elementos repetidos. | Insertar, eliminar, unión, intersección. | Almacenamiento de claves únicas. |
| Diccionario o Mapa (Map) | Pares clave–valor. | Insertar, buscar, eliminar. | Registro de datos de usuarios. |

**Uso de los Tipos de Datos Abstractos (TDA)**

El uso de los TDA es fundamental en la programación estructurada y orientada a objetos, ya que permite trabajar con datos de manera organizada y eficiente.

**Principales usos:**

* Simplificación del código: Permiten resolver problemas complejos de forma ordenada.
* Optimización del almacenamiento: Administran mejor la memoria y los recursos.
* Modularidad: Facilitan dividir el programa en partes más pequeñas y entendibles.
* Mantenimiento: Si se cambia la implementación interna, el resto del programa no se ve afectado.
* Reutilización: Un mismo TDA puede aplicarse a distintos proyectos o contextos.

**Usos del TDA en la programación con C++**

Los TDA se aplican ampliamente en el desarrollo de programas y sistemas donde se requiere organización de datos complejos.

**Algunos ejemplos de uso práctico son:**

* Gestión de memoria en compiladores y sistemas operativos (uso de pilas).
* Colas de impresión o tareas en sistemas multitarea.
* Bases de datos y sistemas de búsqueda, donde se utilizan árboles y listas enlazadas.
* Juegos y simulaciones, donde los grafos y árboles ayudan a representar caminos o decisiones.
* Software financiero, donde se manejan registros y estructuras dinámicas con listas o mapas.

**Conclusión**

En la programación con C++, los Tipos de Datos Abstractos (TDA) son la base de la Estructura de Datos.

Permiten construir programas más organizados, seguros y fáciles de mantener, gracias al encapsulamiento y la independencia entre interfaz e implementación.

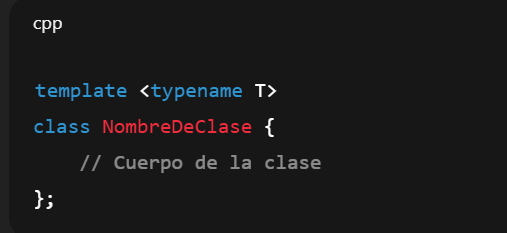
Su correcta aplicación ayuda a resolver problemas complejos mediante estructuras como pilas, colas, listas, árboles y grafos, fundamentales en el desarrollo de software profesional.

1. **MANEJO DE PLANTILLA (CLASES GENERICAS)**

En C++, el manejo de plantillas (también conocido como clases genéricas) es una de las características más poderosas del lenguaje, ya que permite crear funciones y clases que pueden trabajar con cualquier tipo de dato sin necesidad de escribir el mismo código varias veces. El uso de plantillas es fundamental en la programación genérica, ya que mejora la eficiencia, reutilización y flexibilidad del código. En la materia de Estructura de Datos, las plantillas se aplican para construir estructuras como pilas, colas, listas y árboles, que deben funcionar con distintos tipos de datos.

**¿Qué es una plantilla en C++?**

Una **plantilla (template)** es un modelo o patrón que permite definir **clases o funciones genéricas**.  
Cuando se usa una plantilla, se define el tipo de dato mediante un **parámetro genérico**, que se reemplaza por el tipo real (por ejemplo, int, float, char, string) cuando se crea un objeto.



Donde:

* template indica que se va a crear una plantilla.
* typename T declara un tipo genérico (T puede reemplazarse por cualquier tipo de dato).
* NombreDeClase es el nombre de la clase genérica.

**Ventajas del uso de plantillas**

* Reutilización del código: Una sola clase puede trabajar con diferentes tipos de datos.
* Eficiencia: No es necesario crear múltiples versiones de la misma clase.
* Seguridad de tipos: El compilador detecta errores de tipo en tiempo de compilación.
* Flexibilidad: Permite combinar estructuras con distintos tipos de datos sin modificar el código.
* Facilita la creación de estructuras de datos genéricas, como las que usa la STL (Standard Template Library).

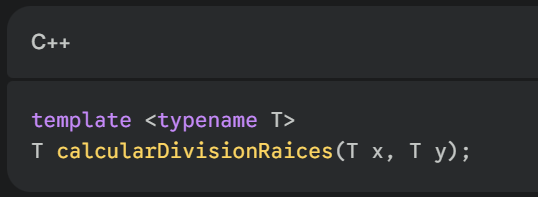
**Aplicaciones de las clases genéricas**

El manejo de plantillas es ampliamente utilizado en la Standard Template Library (STL) de C++, la cual incluye estructuras y algoritmos genéricos, como:

* vector<T>: lista dinámica de elementos.
* stack<T>: implementación genérica de pila.
* queue<T>: implementación genérica de cola.
* map<K, V>: estructura tipo diccionario (clave-valor).
* set<T>: colección sin elementos repetidos.

Estas herramientas son posibles gracias al uso de clases genéricas (templates), lo que permite trabajar con cualquier tipo de dato sin reescribir código.

El **manejo de plantillas en C++** es una herramienta esencial para crear **clases y funciones genéricas** que trabajan con distintos tipos de datos de manera flexible.  
Gracias a ellas, se pueden implementar **estructuras de datos abstractas (TDA)** reutilizables, como pilas, colas y listas, sin depender de un tipo de dato específico.  
El uso de plantillas mejora la **eficiencia, la legibilidad y el modularidad del código**, siendo un pilar fundamental en la programación moderna con C++.



1. **SOBRECARGA DE OPERADORES Y FUNCIONES**

En el lenguaje C++, la sobrecarga de operadores y funciones es una característica fundamental de la programación orientada a objetos (POO).

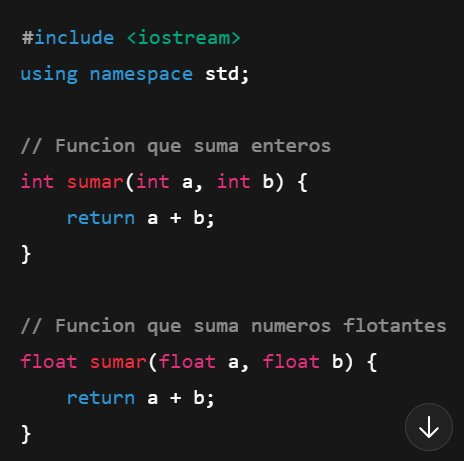
Permite definir nuevos comportamientos para los operadores y funciones ya existentes, de manera que puedan trabajar correctamente con objetos o tipos de datos definidos por el programador.

Esto mejora la legibilidad, reutilización y naturalidad del código, haciendo posible que los objetos se manipulen de forma similar a los tipos primitivos.

**Sobrecarga de funciones**

La sobrecarga de funciones consiste en declarar varias funciones con el mismo nombre, pero con diferente número o tipo de parámetros.

El compilador selecciona automáticamente la versión correcta según los argumentos usados en la llamada.

Texto

Descripción generada automáticamente

Explicación:

* Las tres funciones se llaman sumar, pero cada una tiene parámetros distintos.
* El compilador elige automáticamente cuál ejecutar según los tipos de datos pasados.
* Este mecanismo facilita la creación de funciones más versátiles y reutilizables.

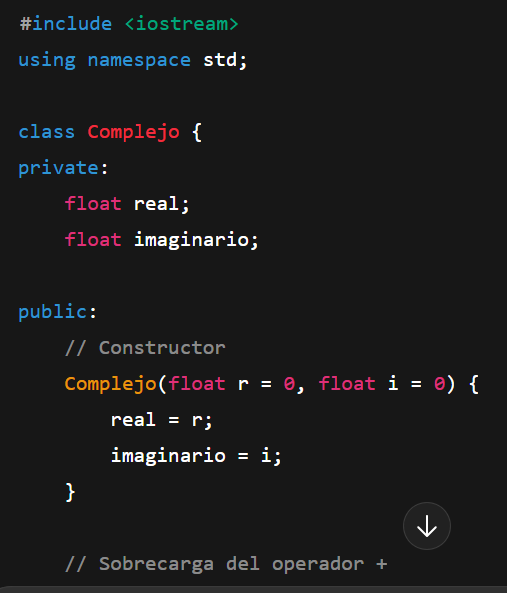
**Sobrecarga de operadores**

La sobrecarga de operadores permite redefinir el funcionamiento de los operadores aritméticos, relacionales o lógicos (+, -, \*, /, ==, <, etc.) para que puedan aplicarse a objetos creados por el usuario.

Esto se logra mediante funciones especiales llamadas operator functions, que inician con la palabra reservada operator seguida del símbolo del operador que se quiere redefinir.

Ejemplo: Sobrecarga del operador “+”

En este ejemplo, se sobrecarga el operador + para sumar dos objetos de tipo Complejo.

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Explicación:

* La función operator+ permite usar el símbolo + entre dos objetos de tipo Complejo.
* Cuando se ejecuta c1 + c2, el compilador llama automáticamente a la función sobrecargada.
* Esto hace que el código sea más natural y fácil de leer.

Una captura de pantalla de un celular

Descripción generada automáticamente con confianza media

**Ejemplo: Sobrecarga del operador “<<” para mostrar objetos**

Este operador permite imprimir objetos con **cout** de forma directa.

Texto

Descripción generada automáticamente Texto

Descripción generada automáticamente

Explicación:

* Se usa una función amiga (friend) para acceder a los atributos privados.
* El operador << se redefine para imprimir directamente el contenido del objeto.
* Ahora cout << p1; es posible, gracias a la sobrecarga.

Beneficios de la sobrecarga

* Permite trabajar con objetos de forma natural, como con tipos básicos.
* Aumenta la claridad y legibilidad del código.
* Facilita el reuso y la flexibilidad del programa.
* Permite que los TDA (Tipos de Datos Abstractos) sean más intuitivos.
* Se integra perfectamente con el paradigma orientado a objetos de C++.

La **sobrecarga de funciones y operadores** es una de las herramientas más útiles de C++, ya que permite **personalizar el comportamiento de funciones y operadores** para trabajar con objetos.  
Gracias a esta característica, los **Tipos de Datos Abstractos (TDA)** pueden comportarse como tipos primitivos, logrando programas **más naturales, organizados y eficientes**.  
En la **Estructura de Datos**, esto se aplica en la implementación de clases genéricas, estructuras matemáticas, listas, pilas o vectores personalizados.

**Sobrecarga de Operadores en C++**

La sobrecarga de operadores en C++ permite redefinir el comportamiento de los operadores tradicionales (como +, -, \*, ==, etc.) para que puedan trabajar con objetos definidos por el programador, no solo con tipos de datos primitivos.

Esto significa que el programador puede decidir cómo deben comportarse los operadores cuando se aplican a clases o tipos de datos abstractos (TDA).

Por ejemplo, si tenemos una clase Fraccion, podemos hacer que el operador + sume dos fracciones correctamente, igual que en matemáticas.

2. Características principales

* El operador sobrecargado mantiene su prioridad y asociatividad original.
* Se puede sobrecargar casi cualquier operador, excepto algunos como ::, .\*, sizeof, y ?:.
* La sobrecarga no crea nuevos operadores, sino que extiende su funcionalidad.
* Puede hacerse mediante funciones miembro o funciones externas (amigas) de la clase.

3. Sintaxis general

Para sobrecargar un operador dentro de una clase se usa la palabra clave operator seguida del símbolo del operador.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

**4. Ejemplo práctico: operador +**

En este ejemplo, se sobrecarga el operador + para que sume dos objetos de la clase Fraccion.

Texto

Descripción generada automáticamenteTexto

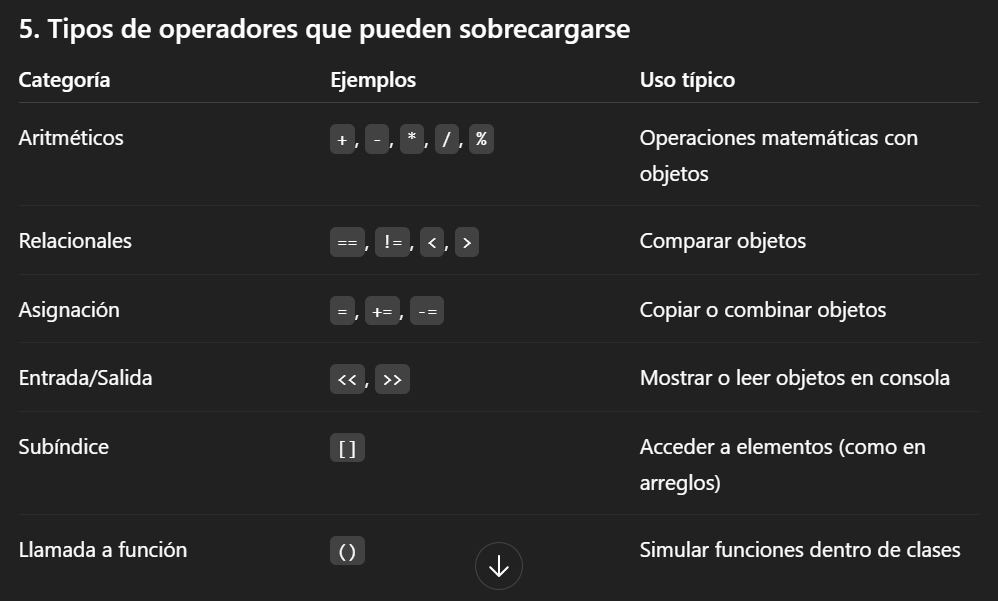
Descripción generada automáticamente

Texto

Descripción generada automáticamente

Explicación:

* La función operator+ define cómo sumar dos fracciones.
* Al ejecutar f1 + f2, el compilador llama automáticamente a esa función.
* El resultado se devuelve como un nuevo objeto Fraccion.



Ventajas de la sobrecarga de operadores

* Permite que los objetos se manipulen como los tipos primitivos.
* Hace que el código sea más legible y natural.
* Facilita el uso de tipos de datos abstractos (TDA) en operaciones complejas.
* Mejora la reutilización y el mantenimiento del código.

La sobrecarga de operadores en C++ es una herramienta poderosa que mejora la interacción con los objetos y estructuras de datos personalizadas.

Gracias a ella, los programadores pueden hacer que los operadores clásicos se comporten de manera lógica con las clases creadas, logrando un código más expresivo, limpio y cercano al lenguaje matemático o natural.

**Sobrecarga de Funciones en C++**

La sobrecarga de funciones en C++ consiste en declarar varias funciones con el mismo nombre, pero que se diferencian por el número o tipo de parámetros que reciben.

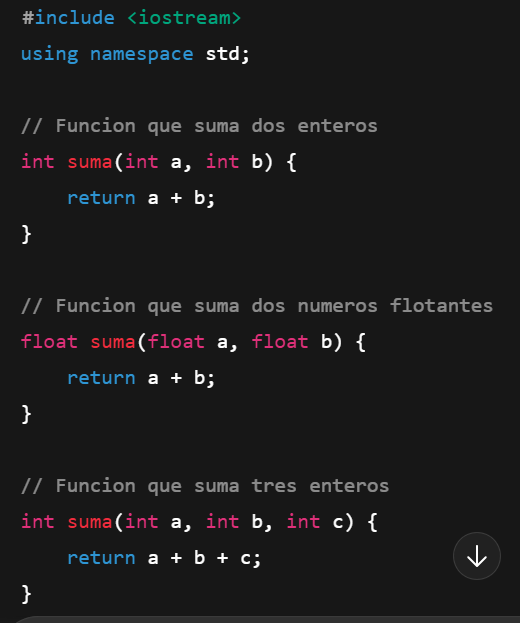
El compilador determina cuál función ejecutar según los argumentos usados en la llamada.

Esta característica permite que un mismo nombre describa acciones similares pero con distintos tipos de datos, haciendo el código más claro y fácil de mantener.

Características principales

* Todas las funciones sobrecargadas deben tener el mismo nombre.
* Deben diferir en la lista de parámetros (tipo, número o ambos).
* El tipo de retorno no influye en la sobrecarga.
* El compilador selecciona automáticamente la función adecuada durante la ejecución.

Ejemplo básico de sobrecarga

Texto

Descripción generada automáticamente

Explicación:

* Aunque las tres funciones se llaman suma, el compilador elige la que corresponde dependiendo de los tipos y cantidad de argumentos.
* Esta técnica hace que el código sea más flexible y reutilizable.

Importancia en la programación

La sobrecarga de funciones es muy útil cuando un mismo proceso debe aplicarse a diferentes tipos de datos o cantidades de argumentos.

Por ejemplo, en un TDA (Tipo de Dato Abstracto), se puede sobrecargar una función insertar() para que acepte distintos tipos de valores según la estructura de datos.

Beneficios

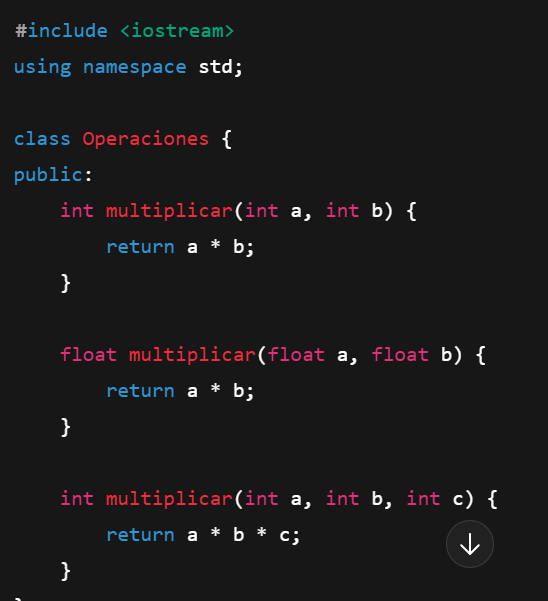
* Reutilización del código: se pueden usar nombres comunes para tareas relacionadas.
* Facilita la comprensión: los nombres de las funciones se asocian fácilmente a su propósito.
* Evita duplicación innecesaria de nombres: no se necesita crear funciones como sumaInt, sumaFloat, etc.
* Mejor organización: agrupa operaciones similares bajo un mismo identificador.

Reglas de la sobrecarga

* Deben tener el mismo nombre pero diferente lista de parámetros.
* No se puede sobrecargar una función solo cambiando el tipo de retorno.
* Pueden combinarse con valores por defecto.
* El compilador selecciona la versión correcta durante la compilación (no en ejecución).

Ejemplo aplicado a clases

También es posible sobrecargar funciones dentro de una clase.

Texto

Descripción generada automáticamente

Explicación:

* La clase Operaciones tiene tres métodos con el mismo nombre multiplicar, pero con diferentes parámetros.
* El compilador elige automáticamente el método adecuado según los valores usados.

Conclusión

La sobrecarga de funciones en C++ es una herramienta fundamental para escribir programas claros, reutilizables y eficientes.

Permite definir múltiples versiones de una misma función que realizan tareas similares con distintos tipos de datos, facilitando el trabajo con estructuras de datos y clases.

En el contexto de Estructura de Datos, esta técnica mejora la organización del código y hace que los TDA (Tipos de Datos Abstractos) sean más versátiles y fáciles de manejar.

**Clases y Objetos**

Una **clase** es una **plantilla o modelo** que define cómo se comportará un tipo de dato personalizado (un TDA).  
Un **objeto** es una instancia (un ejemplar) concreta de esa clase.

**Constructores**

Un **constructor** es una función especial dentro de una clase que se ejecuta automáticamente **cuando se crea un objeto**.  
Sirve para **inicializar atributos**.

**Métodos Get y Set**

Los métodos **get** y **set** se usan para **acceder y modificar atributos privados** de una clase.  
Esto permite mantener la **encapsulación**, es decir, proteger los datos internos del objeto.

**Propiedades o Atributos**

Los **atributos** son las **variables internas** de una clase que representan las características del objeto.  
Pueden ser de tipo primitivo (int, float, char) o incluso otros objetos.

**Punteros en C++**

Un **puntero** es una variable que **almacena la dirección de memoria** de otra variable.  
Se usan mucho en **estructuras de datos dinámicas** (listas enlazadas, pilas, colas, árboles) porque permiten **crear y enlazar nodos en tiempo de ejecución**.

**Encapsulación y Abstracción**

* **Encapsulación:** protege los datos internos de una clase, permitiendo el acceso solo a través de funciones.
* **Abstracción:** permite al usuario usar la clase sin preocuparse por cómo funciona internamente (por ejemplo, al usar un TDA Pila o Cola).

1. **GESTION DE MEMORIA ESTATICA CON TDA**

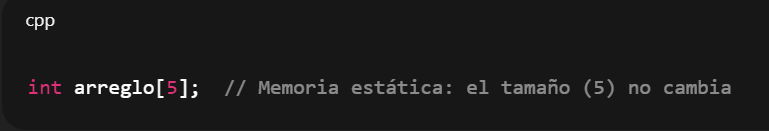
Gestión de Memoria Estática con TDA en C++

¿Qué es la memoria estática? La memoria estática es aquella que se reserva antes de ejecutar el programa, normalmente durante la compilación.

El tamaño de los datos no cambia mientras el programa se ejecuta.

En C++, las variables locales estáticas, arreglos o estructuras definidas con tamaño fijo utilizan este tipo de memoria.

Ejemplo simple:



¿Qué es un TDA (Tipo de Dato Abstracto)?

Un TDA (Tipo de Dato Abstracto) es una estructura lógica que permite organizar datos y operaciones sin importar cómo están implementados internamente.

Ejemplos de TDA comunes:

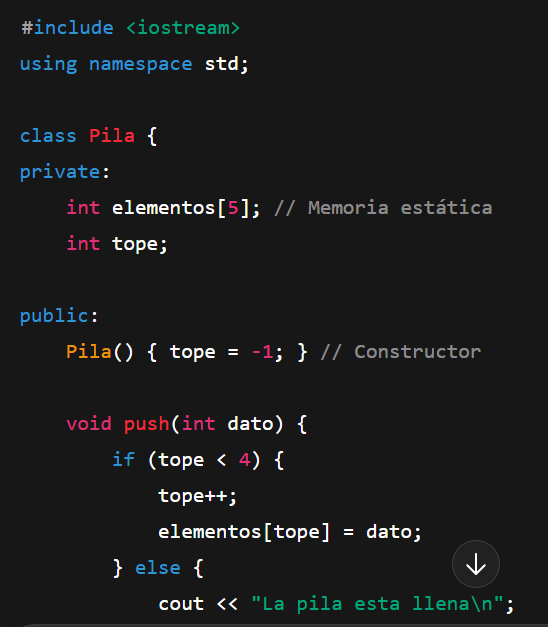
* Pila (Stack)
* Cola (Queue)
* Lista (List)
* Arreglo (Array)
* Registro o estructura (struct)

**Gestión de memoria estática con TDA**

Cuando un TDA se implementa usando memoria estática, el tamaño del almacenamiento se define de forma fija dentro de la clase o estructura, lo que significa que no se usa memoria dinámica (new/delete).

Esto es común cuando se sabe de antemano cuántos elementos tendrá el TDA.

Ejemplo: TDA Pila con memoria estática

Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamente

Características clave

* El tamaño del arreglo es fijo.
* No se usa new ni delete (por eso es estática).
* La memoria se libera automáticamente al finalizar el programa.
* No permite crecimiento o reducción del tamaño durante la ejecución.

Ventajas

* Más eficiente y rápida, porque no hay gestión de memoria dinámica.
* No hay riesgo de fugas de memoria.
* Fácil de implementar y depurar.
* Simplicidad: fácil de implementar y depurar.
* Eficiencia: no requiere operaciones de asignación o liberación en tiempo de ejecución.
* Seguridad: no hay fugas de memoria ni errores de punteros.
* Rapidez: acceso directo a posiciones de memoria contiguas.

Desventajas

* Tamaño fijo, no se puede cambiar.
* Desperdicio de memoria si el arreglo es muy grande y no se usa todo.
* No se adapta bien a estructuras que deben crecer dinámicamente.
* Limitada flexibilidad: no sirve para estructuras de datos que deben crecer o reducirse dinámicamente.
* Depende del tamaño máximo definido: si el límite es bajo, puede haber desbordamientos (overflow).

Aplicaciones típicas

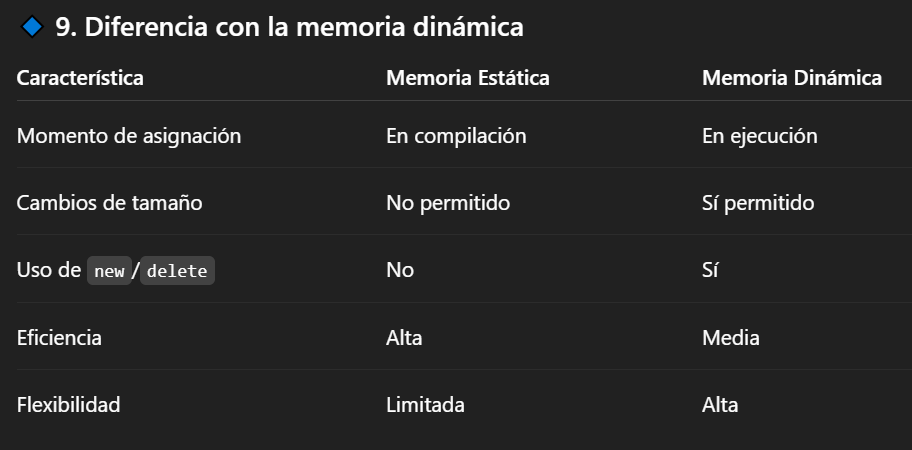
* Pilas o colas de tamaño conocido (por ejemplo, buffer de teclado).
* Tablas fijas o arreglos donde el número de datos no cambia.
* Implementaciones de TDA para fines educativos o de rendimiento fijo.

Cómo se gestiona la memoria estática

* En un TDA con memoria estática:
* La reserva de memoria se hace automáticamente cuando el programa empieza.
* No se usa new ni delete.
* Los objetos estáticos viven durante toda la ejecución del programa o mientras su función esté activa.
* El sistema operativo libera la memoria al finalizar el programa.

Aplicaciones prácticas

* El uso de memoria estática en TDAs se aplica en:
* Sistemas embebidos o de tiempo real (donde la memoria disponible es limitada).
* Estructuras de tamaño fijo como tablas, buffers o matrices.
* Programas educativos para enseñar estructuras básicas sin complicaciones dinámicas.



La gestión de memoria estática con TDA es un enfoque muy usado cuando se requiere simplicidad, rapidez y control del tamaño de las estructuras de datos.

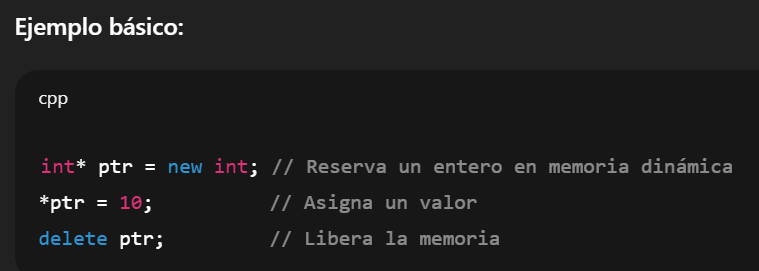
Aunque no permite crecer en tiempo de ejecución, es ideal para casos donde el tamaño máximo de los datos se conoce previamente.

En C++, esta técnica se implementa fácilmente con arreglos de tamaño fijo dentro de clases o estructuras, garantizando un manejo seguro y eficiente de la memoria.

1. **GESTION DE MEMORIA DINAMICA CON TDA**

La memoria dinámica es aquella que se asigna durante la ejecución del programa, es decir, en tiempo de ejecución, usando punteros y las funciones new y delete en C++.

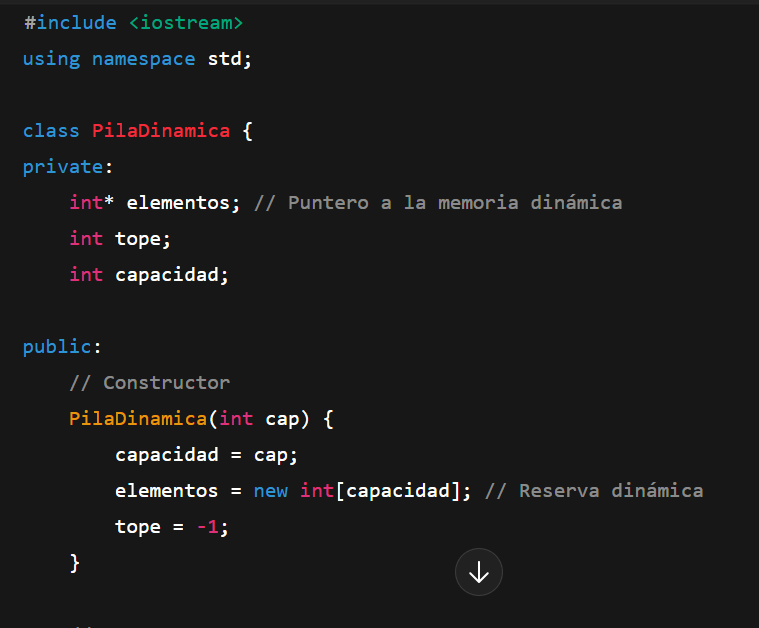
A diferencia de la memoria estática, su tamaño puede cambiar mientras el programa corre, lo que permite manejar estructuras de datos que crecen o se reducen según las necesidades.



Gestión dinámica dentro de un TDA

* En un TDA con memoria dinámica:
* Se usan punteros para almacenar la referencia a la memoria asignada.
* Se puede usar new para asignar espacio y delete para liberarlo.
* La memoria no se libera automáticamente; el programador debe gestionarla para evitar fugas de memoria (memory leaks).

Ejemplo: TDA Pila dinámica

 Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamenteTexto

Descripción generada automáticamente

Explicación:

* new int[capacidad] asigna memoria dinámica para la pila.
* delete[] elementos libera la memoria al finalizar el objeto (Destructor).
* La pila puede crecer según capacidad definida al crear el objeto, permitiendo flexibilidad.

Gestión de Memoria Dinámica: malloc, calloc y new

En C++, la memoria dinámica permite crear variables, arreglos u objetos en tiempo de ejecución, es decir, mientras el programa está corriendo.

Esto se usa cuando no sabemos cuántos datos necesitaremos o cuando queremos estructuras flexibles (listas, pilas, colas, árboles, etc.).

**malloc (memory allocation)**

Definición: malloc() (de memory allocation) viene del lenguaje C, pero también puede usarse en C++.

Sirve para reservar un bloque de memoria continua de un tamaño específico en bytes.

No puede ser bool por que solo devuelve true o false

Id\*

Int \*p = (int\*) malloc (4\* sizeof (\*int));

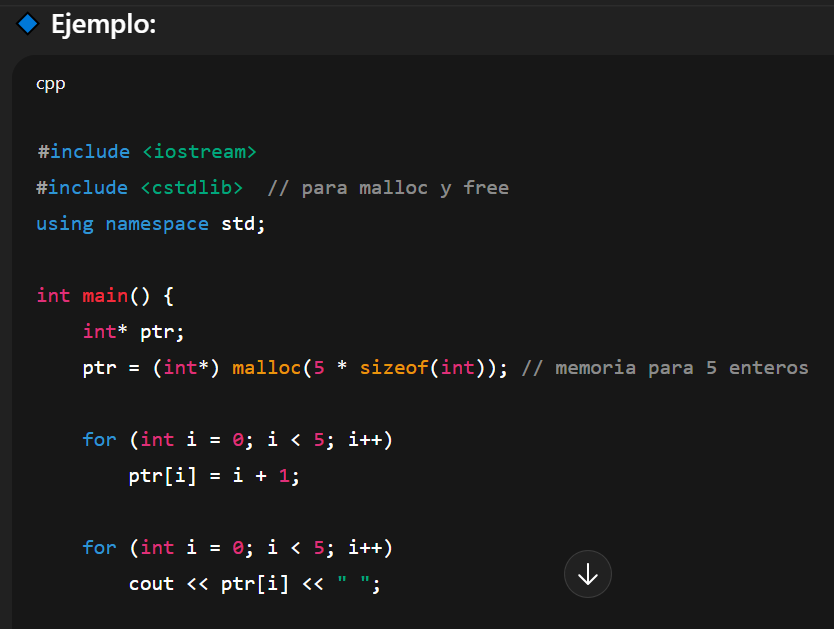
Sizeof (\*int) seria 4 y con el 4 que tengo a fuera seria 16 ( revisar: y con el int\* del primer paréntesis seria 20 )

**El operador punto ( . )**

Se usa cuando **tienes un objeto normal**, es decir, una **variable del tipo de la clase**, no un puntero.

**El operador flecha ( -> )**

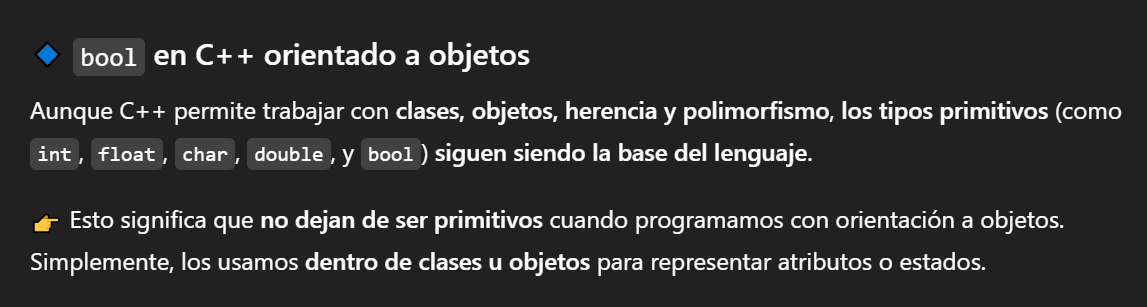
Se usa cuando **trabajas con punteros a objetos**, es decir, **cuando reservas memoria dinámica** con new.



Texto

Descripción generada automáticamente





CASTING: **Casting** (también llamado **conversión de tipo**) es el **proceso de transformar una variable de un tipo de dato a otro**.  
Sirve para que C++ interprete un valor **como si fuera de otro tipo** cuando se necesita, por ejemplo, al realizar operaciones entre distintos tipos o al trabajar con memoria.

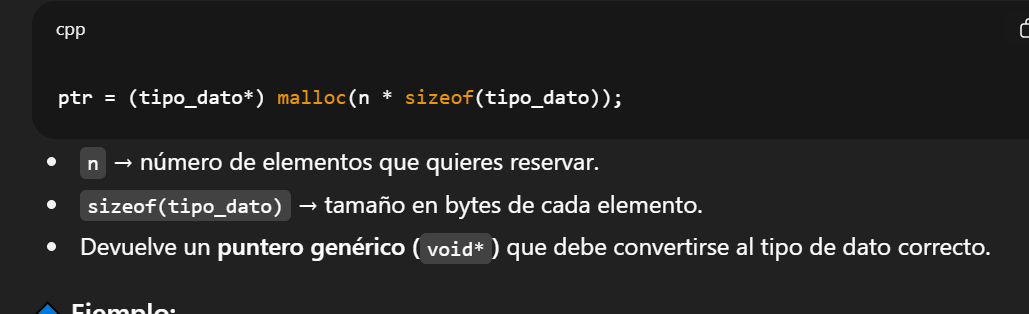
**Tipos de *casting***

**1. Casting implícito (automático)**

C++ lo hace **de forma automática** cuando combinas tipos diferentes en una operación.

**Casting explícito (manual)**

El programador **indica manualmente** que un valor debe tratarse como otro tipo.  
Se hace usando paréntesis o con una sintaxis más moderna.



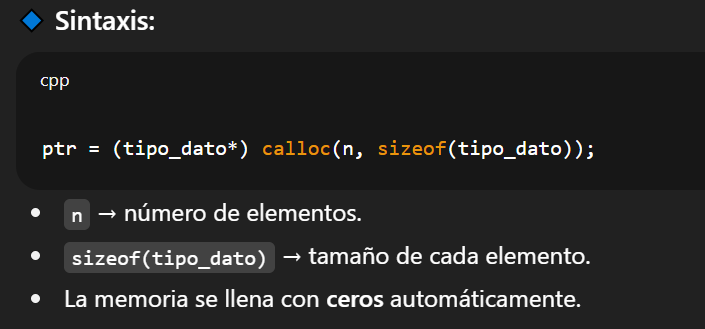
**calloc (contiguous allocation)**

Definición:

calloc() también viene de C y sirve para reservar e inicializar bloques de memoria.

La diferencia con malloc() es que inicializa toda la memoria en cero.

Sintaxis:



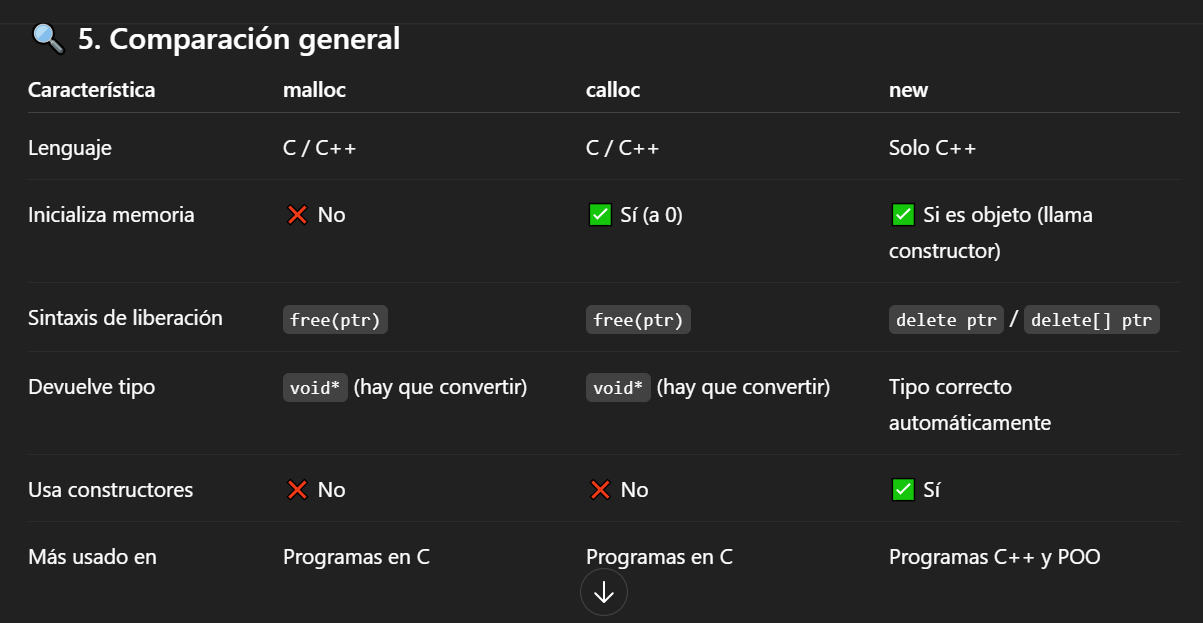
**new (operador de C++)**

Definición:

* new es el operador propio de C++ para reservar memoria dinámica.
* A diferencia de malloc() y calloc(), llama automáticamente al constructor si estás creando objetos.

Texto

Descripción generada automáticamente



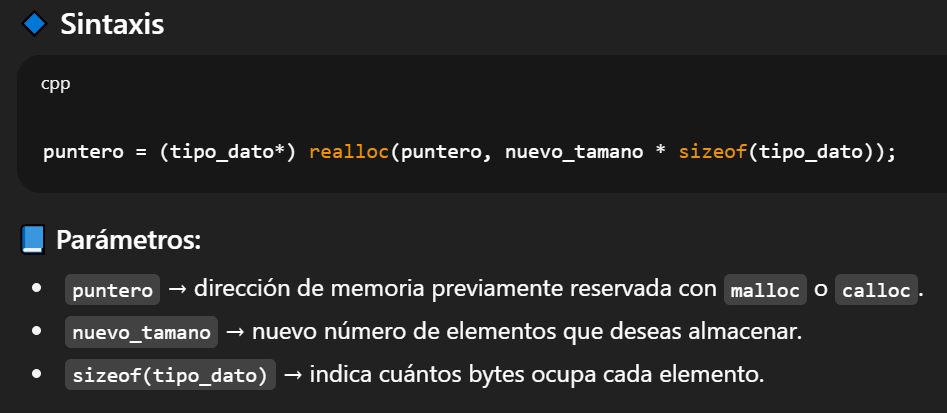
Una captura de pantalla de un celular

Descripción generada automáticamente

**Qué es realloc?**

La función **realloc** (abreviatura de *reallocation*, o **reasignación de memoria**) se utiliza para **cambiar el tamaño de un bloque de memoria** que fue previamente asignado con malloc() o calloc().  
**realloc sirve para ampliar o reducir el espacio de memoria dinámica sin perder los datos ya almacenados.**

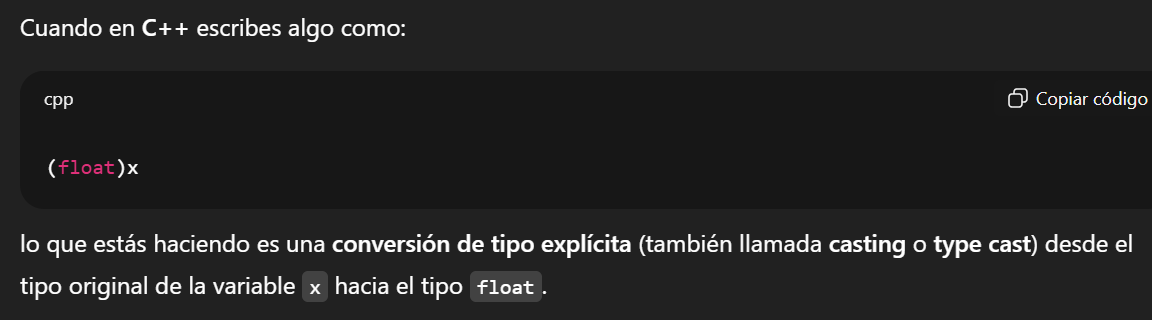
**Sintaxis**



¿Qué hace internamente?

1. Toma el bloque de memoria apuntado por puntero.
2. Intenta redimensionarlo:
   * Si hay espacio disponible, amplía el mismo bloque.
   * Si no hay espacio contiguo, crea un nuevo bloque más grande y copia los datos anteriores.
3. Devuelve la nueva dirección de memoria del bloque (puede cambiar).
4. Si no puede asignar más memoria, devuelve NULL.

**Ejemplo**



Int y = (float)x

No quedo bien con el entero (int) y se cambia a float si funciona el casting (conversión de tipo)

Float y = (float) x

**DIFERENCIAS**

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Sitio web

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Captura de pantalla de computadora

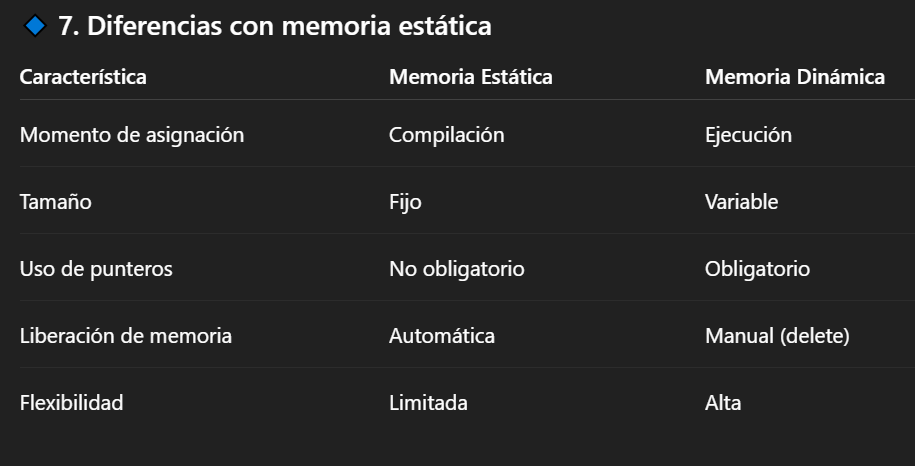
Descripción generada automáticamente con confianza media

Ventajas de la memoria dinámica

* Tamaño flexible: se puede definir en tiempo de ejecución.
* Ahorro de memoria: se asigna solo lo que se necesita.
* Permite estructuras complejas: listas, árboles, colas y grafos dinámicos.
* Reutilización de memoria: se puede liberar y reasignar durante la ejecución.

Desventajas

* Mayor complejidad: se deben gestionar punteros y memoria explícitamente.
* Posibles fugas de memoria: si no se libera la memoria correctamente (delete).
* Más lento que memoria estática: debido a la gestión en tiempo de ejecución.
* Errores por punteros: pueden generar fallos graves si se usan punteros inválidos o no inicializados.



Aplicaciones típicas de memoria dinámica en TDAs

* Pilas y colas de tamaño variable.
* Listas enlazadas (simple, doble, circular).
* Árboles binarios, grafos y estructuras complejas.
* Sistemas donde no se conoce la cantidad de elementos de antemano.

Buenas prácticas

* Siempre inicializar los punteros antes de usarlos.
* Usar delete o delete[] para liberar memoria.
* Evitar asignaciones múltiples sin liberar memoria previa.
* Combinar con constructores y destructores para un manejo seguro.

Conclusión

La gestión de memoria dinámica con TDA en C++ permite crear estructuras de datos flexibles y eficientes, adaptadas a las necesidades del programa.

A diferencia de la memoria estática, su tamaño no está limitado a la compilación, pero requiere responsabilidad en la liberación de memoria y buen uso de punteros para evitar errores y fugas.

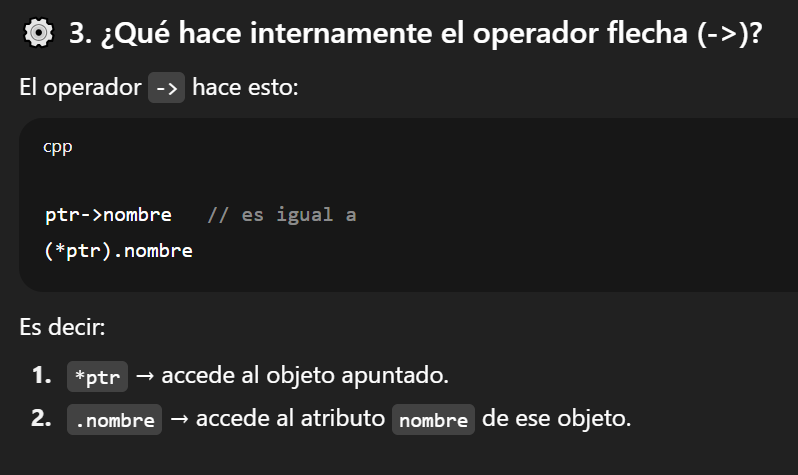
el **punto (.)** y la **flecha (->)** se usan para **acceder a miembros (atributos o métodos)**, pero **dependiendo de si estás usando un objeto normal o un puntero**.

**El operador punto ( . )**

Se usa cuando **tienes un objeto normal**, es decir, una **variable del tipo de la clase**, no un puntero.

**El operador flecha ( -> )**

Se usa cuando **trabajas con punteros a objetos**, es decir, **cuando reservas memoria dinámica** con new.



Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

un miembro de una clase es un componente que define su estructura y comportamiento, y puede ser un atributo (variable) o una función (método). Los atributos representan las propiedades de un objeto, mientras que las funciones miembro definen las acciones que puede realizar.

Tipos de miembros

Atributos o variables miembro:

Son variables que almacenan el estado o las propiedades de un objeto.

Cada objeto creado a partir de la clase tiene su propia copia de estas variables.

Funciones miembro o métodos:

Son funciones declaradas dentro de una clase que operan sobre los datos de esa clase.

Controlan el comportamiento de los objetos y pueden acceder a los atributos privados de la clase.

Tienen acceso directo a los datos del objeto que las invoca (como si tuvieran un parámetro this implícito).

EJEMPLO:

\*

=

Memoria estática:

A[F][C]

(\*(\*(A)+i)+j) es lo mismo

Explicación:

For (i=0; i<3; i++)

For (j=0; j<3; j++)

\*(\*(A+i)+j) = 0 ;

Hacer la prueba en c++ en el escritorio hacer la prueba de escritorio.

#include <iostream>

using namespace std;

class Matriz {

private:

int filas, columnas;

int datos[3][3]; // Memoria estática para 3x3

public:

Matriz() {

filas = columnas = 3;

// Inicializamos en 0

for (int i = 0; i < filas; i++)

for (int j = 0; j < columnas; j++)

\*(\*(datos + i) + j) = 0;

}

void leer() {

cout << "Ingrese los elementos de la matriz (" << filas << "x" << columnas << "):\n";

for (int i = 0; i < filas; i++) {

for (int j = 0; j < columnas; j++) {

cin >> \*(\*(datos + i) + j);

}

}

}

void mostrar() {

for (int i = 0; i < filas; i++) {

for (int j = 0; j < columnas; j++) {

cout << \*(\*(datos + i) + j) << " ";

}

cout << endl;

}

}

Matriz multiplicar(Matriz &B) {

Matriz resultado;

for (int i = 0; i < filas; i++) {

for (int j = 0; j < columnas; j++) {

for (int k = 0; k < columnas; k++) {

\*(\*(resultado.datos + i) + j) += \*(\*(datos + i) + k) \* \*(\*(B.datos + k) + j);

}

}

}

return resultado;

}

};

int main() {

Matriz A, B, C;

cout << "Ingrese la primera matriz:\n";

A.leer();

cout << "Ingrese la segunda matriz:\n";

B.leer();

C = A.multiplicar(B);

cout << "Resultado de la multiplicación:\n";

C.mostrar();

return 0;

}

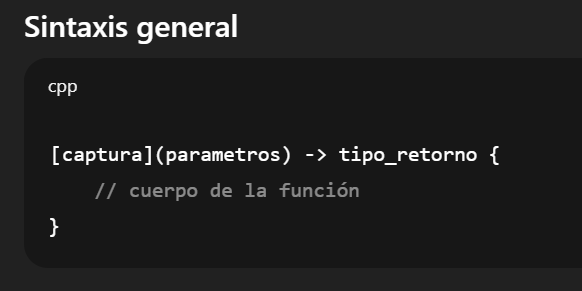
crea un progrma que maneje espacio de memoria dinamica no estatico, una funcion para encerar, una funcion para procesar,e imprimir, en proceso se envia 3 matrices en donde dos se crean y la otra donde se devuelva el resultado

* Crear una función enserar mando la matriz debes segmentar y cuando ya esta segmentada la matriz 1, 2,3 toca tener una matriz llamada enserar no hay necesidad de retornar hacerlo con punteros de tipo flotante debe ser la matriz . Y lo convierte en unidimensional cuando utilizo memoria dinámica se debe liberar Enserar, liberar imprimir, procesar y multiplicar y darme el resultado
* Deber multiplicación de matrices con clases modelo utilizar template en power designer. Tipo de dato abstracta plantillas utilizar un operador sobrecargado, Modelo (set get) vista modelo controlador
* Y refuerzo por cada operador que se puede sobrecargar un ejemplo, hacer complejos.

**¿Qué es una función lambda? Extras que toca presentar**

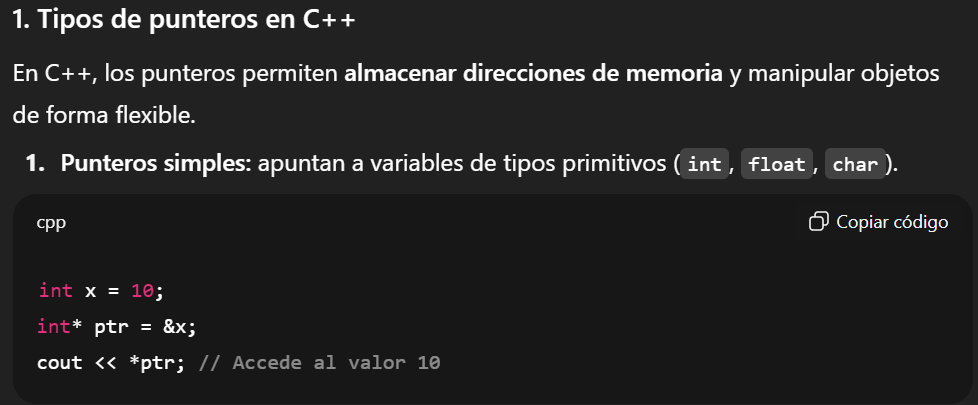
Una **función lambda** (también llamada *expresión lambda*) es una **función anónima**, es decir, **una función sin nombre** que puede definirse directamente dentro del código, justo donde se necesita.

Se introdujeron a partir de **C++11** para escribir funciones **cortas y rápidas**, especialmente cuando se trabaja con **algoritmos**, **colecciones** o **estructuras de datos**.



Texto

Descripción generada automáticamente



Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Captura de pantalla de un celular

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

1. **RECURSIVIDAD**
2. **CONCEPTOS DE RECURSIVIDAD**

**Para poder ejecutar una función debe ser llamada primero, en caso contrario estaría en caso pasivo**

**parámetro, caso base y caso recursivo.**

**Ejemplo si se parametriza una variable x debe estar declarada esa función recursiva no va poder ser utilizada.**

La recursividad es una técnica de programación donde una función se llama a sí misma para resolver un problema.

Cada llamada recursiva se ejecuta con un caso más pequeño o más simple del problema original, hasta llegar a una condición base que detiene la recursión.

“Un problema grande se divide en subproblemas más pequeños del mismo tipo, hasta llegar a uno tan simple que pueda resolverse directamente.”

¿Que es la programacion exhaustiva?

La programacion exhaustiva es un enfoque que prueba todas las soluciones posibles de un problema hasta encontrar la(s) correcta(s). Tambien se conoce como busqueda por fuerza bruta. Es simple de entender e implementar y garantiza encontrar la solucion si existe, pero suele ser costosa en tiempo y memoria.

Principios y caracteristicas

* Enumeracion completa: Genera todo el espacio de soluciones posibles.
* Verificacion: Para cada candidato se verifica si cumple la condicion requerida.
* Simplicidad: Algoritmos faciles de escribir y razonar.
* Costo: Exponencial o factorial en muchos casos; no escalable sin optimizaciones.
* Uso: Bueno para instancias pequenas, para validar soluciones optimas, o cuando no hay heuristicas claras.

Tecnicas relacionadas

* Fuerza bruta directa: probar todas las combinaciones posibles.
* Generacion de permutaciones/combinaciones/subconjuntos: util para problemas de combinatoria.
* Backtracking (vuelta atras): generar soluciones parcialmente y podar ramas que no pueden dar soluciones validas. Muy comun en sudoku, n-queens, coloracion de grafos.
* Divide and conquer + exhaustiva: dividir espacio en partes y aplicar exhaustiva en cada parte.
* Busqueda con poda: añadir condiciones para cortar ramas (pruning).
* Programacion dinamica / memoizacion: cuando se repiten subproblemas, evita recalculos (no exactamente exhaustiva pura, pero reduce coste).

Cuando usar programacion exhaustiva

* Cuando el espacio de soluciones es relativamente pequeno (n <= 20 en subconjuntos, por ejemplo).
* Para probar correctitud de algoritmos heurísticos.
* En competiciones, cuando no hay otra solucion clara y constraints permiten fuerza bruta.
* Para problemas NP-completos con entradas pequenas.

Ventajas / Desventajas

Ventajas: simple, completa (encuentra solucion si existe), no necesita teoria compleja.

Desventajas: coste muy alto, no practica para instancias grandes, puede consumir mucha memoria.

Optimaciones comunes

* Poda temprana (pruning): cortar una rama cuando sabemos que no dara solucion.
* Ordenar opciones: intentar primero las opciones mas prometedoras.
* Memoizacion: almacenar resultados de subproblemas repetidos.
* Recubrimiento parcial: generar solo hasta cierto tamano/umbral.
* Uso de bitmask para representar subconjuntos de forma eficiente.

Ejemplos practicos y pseudocodigo

Estructura básica de una función recursiva

Una función recursiva debe tener dos partes fundamentales:

**Caso base: Es la condición que detiene la recursión.**

Sin un caso base, la función se llamaría infinitamente y el programa se colapsaría (error de desbordamiento de pila o stack overflow).

Caso recursivo: Es la parte donde la función se llama a sí misma, pero con un subproblema más simple que el original.

Tipos de recursividad

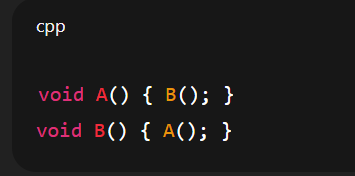
Recursividad directa:

La función se llama directamente a sí misma.

Ejemplo: factorial(n) que se llama con factorial(n-1).

Recursividad indirecta:

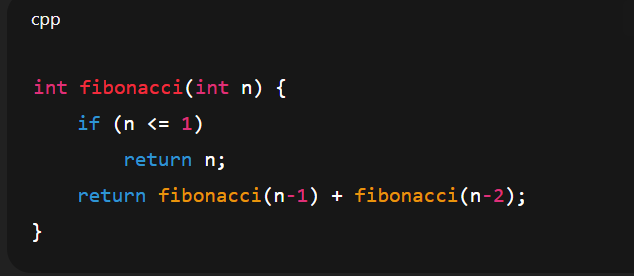
Una función llama a otra, y esa otra vuelve a llamar a la primera.



Recursividad múltiple:

Una función se llama a sí misma más de una vez en cada ejecución.

Ejemplo clásico: Fibonacci



Ventajas de la recursividad

✅ Permite resolver problemas complejos de forma más simple y elegante.

✅ Reduce el código al dividir el problema en subproblemas.

✅ Es útil para problemas naturales de tipo recursivo (como factorial, Fibonacci, torres de Hanoi, búsqueda binaria, etc.).

⚠️ Desventajas

❌ Puede consumir mucha memoria si las llamadas son muy profundas.

❌ Es más lenta que la iteración (debido al uso de pila de llamadas).

❌ Si no se define un caso base correcto, causa un bucle infinito o error de stack overflow.

Ejemplos típicos de recursividad

* Calcular el factorial de un número.
* Calcular la serie de Fibonacci.
* Resolver las Torres de Hanoi.
* Búsqueda binaria o ordenamiento rápido (QuickSort).
* Recorrer árboles o estructuras jerárquicas.

1. **PRINCIPIOS DE LOS ALGORITMOS RECURSIVOS**

Un algoritmo recursivo se basa en el principio de descomposición: resolver un problema grande dividiéndolo en subproblemas más pequeños del mismo tipo, hasta que estos sean lo suficientemente simples para resolverse directamente.

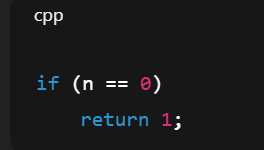
Para que un algoritmo recursivo funcione correctamente, debe cumplir ciertos principios fundamentales.

1. Caso base (condición de parada)

* Es la condición que detiene la recursión.
* Define el caso más sencillo del problema, aquel que puede resolverse sin llamadas adicionales.
* Si no existe un caso base, la función se llamará infinitamente, provocando un error (stack overflow).

Ejemplo:

En el factorial, el caso base es cuando n == 0, ya que 0! = 1 y no es necesario seguir llamando a la función.



2. Llamada recursiva (autorreferencia)

* Es el corazón del algoritmo recursivo.
* En esta parte, la función se llama a sí misma con un problema más pequeño.
* Cada llamada debe acercarse más al caso base.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

3. Reducción del problema

* En cada llamada recursiva, el problema original se simplifica.
* La idea es que el algoritmo avance paso a paso hacia el caso base, evitando bucles infinitos.
* Esto garantiza que cada llamada trabaje con un subconjunto o versión más pequeña del problema.

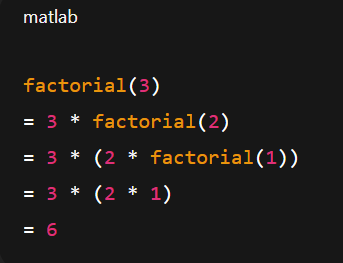
Ejemplo:

En el factorial, se pasa de n a n-1 en cada paso → hasta llegar a 0.

4. Combinación de resultados

* Una vez que se llega al caso base, la función comienza a retornar resultados hacia atrás, combinando los valores obtenidos en cada nivel de recursión.
* Este proceso se conoce como retorno o desenrollamiento de la recursión.

Ejemplo:



5. Correctitud y terminación

Un algoritmo recursivo debe garantizar:

* Que siempre alcanza el caso base.
* Que cada llamada recursiva trabaja con datos más simples.
* Que el resultado final es correcto cuando todas las llamadas se combinan.

Esto se logra analizando bien las condiciones del caso base y cómo se reduce el problema.

6. Uso de la pila de llamadas

* Cada vez que una función se llama a sí misma, el programa almacena la información de esa llamada en la pila de ejecución (stack).
* Cuando la función termina, el programa retoma la ejecución donde se dejó en la llamada anterior.
* Por eso, un mal manejo de la recursión puede causar exceso de memoria o un error de “desbordamiento de pila”.

7. Equivalencia con la iteración

Aunque muchos problemas recursivos pueden resolverse con bucles iterativos, la recursividad ofrece una forma más natural y clara de expresar ciertos algoritmos, especialmente aquellos que implican estructuras jerárquicas o repetición en niveles.

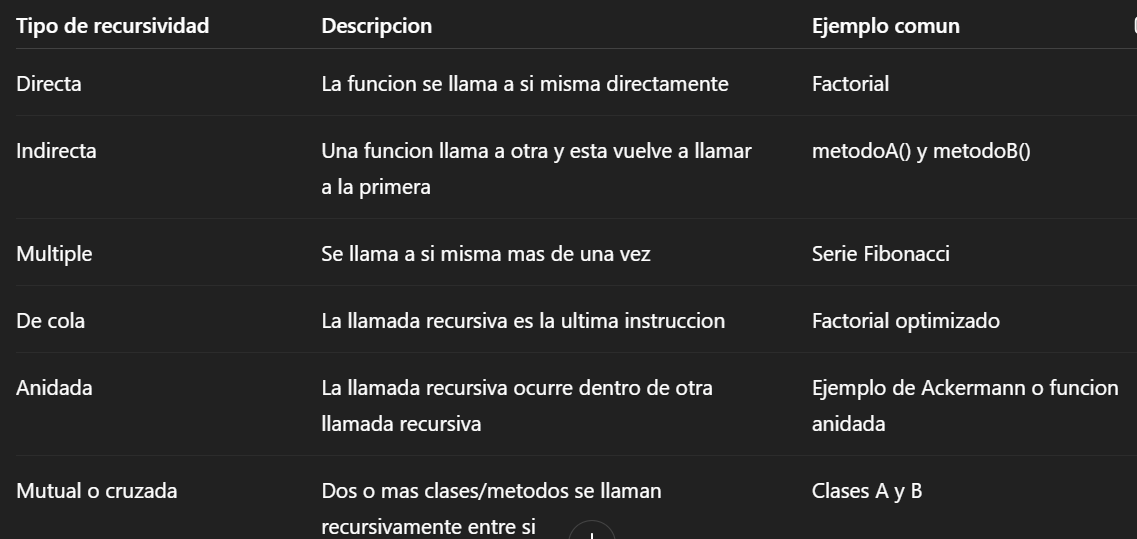
Ejemplo:

Factorial → puede hacerse con for (iterativo) o con recursión.

Árbol binario → mucho más natural usar recursión.



1. **TIPOS DE RECURSIVIDAD**



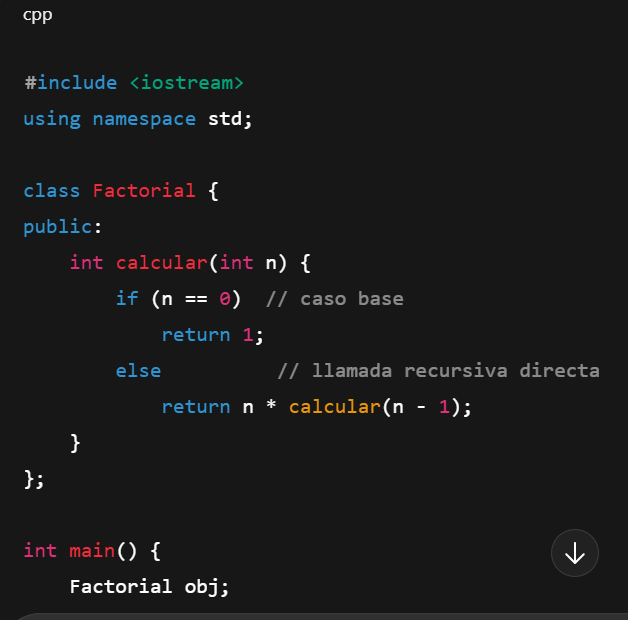
La recursividad puede clasificarse de distintas formas segun como y donde se llame la funcion o metodo.

En programacion orientada a objetos (POO), la recursividad tambien se puede aplicar dentro de metodos de una clase, haciendo que un objeto llame a su propio metodo recursivamente.

1. Recursividad directa

Es cuando una funcion o metodo se llama a si mismo directamente dentro de su cuerpo.

Ejemplo: metodo recursivo directo en una clase



Texto

Descripción generada automáticamente

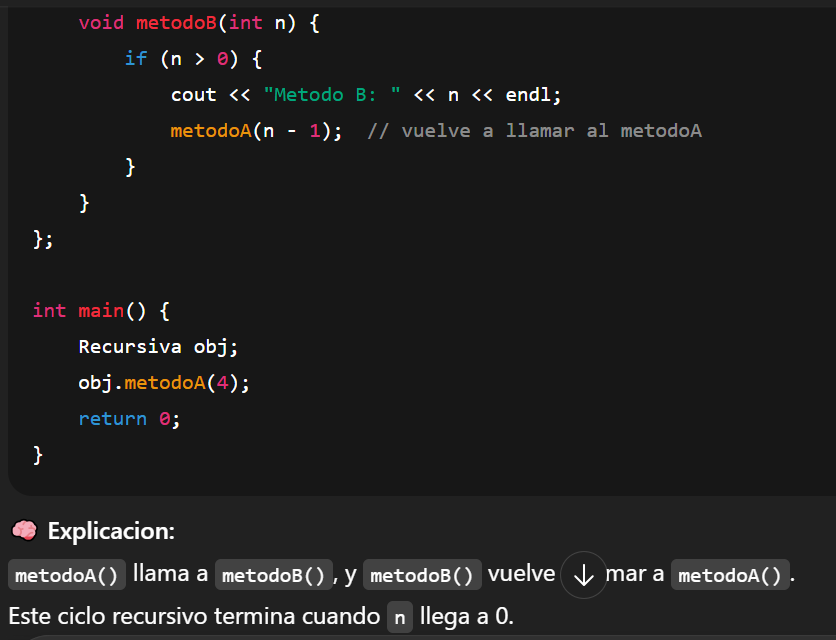
2. Recursividad indirecta

Sucede cuando un metodo llama a otro metodo, y ese segundo vuelve a llamar al primero, formando un ciclo.

Ejemplo: recursividad indirecta entre métodos

Texto

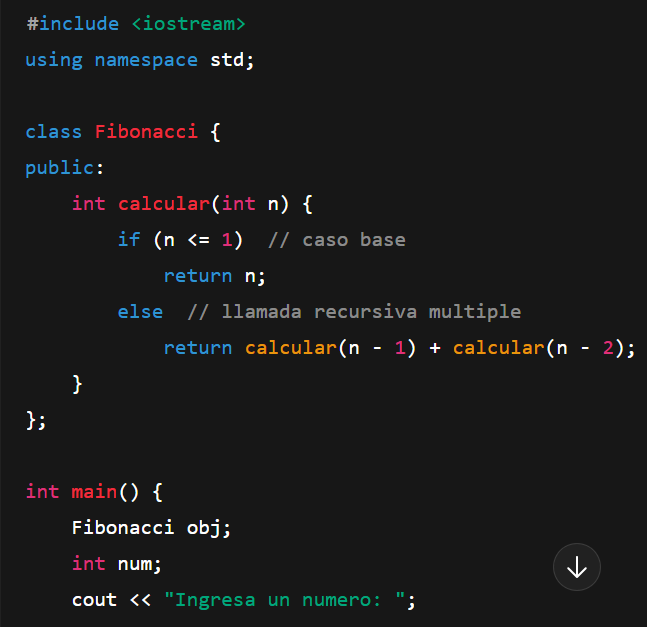
Descripción generada automáticamente

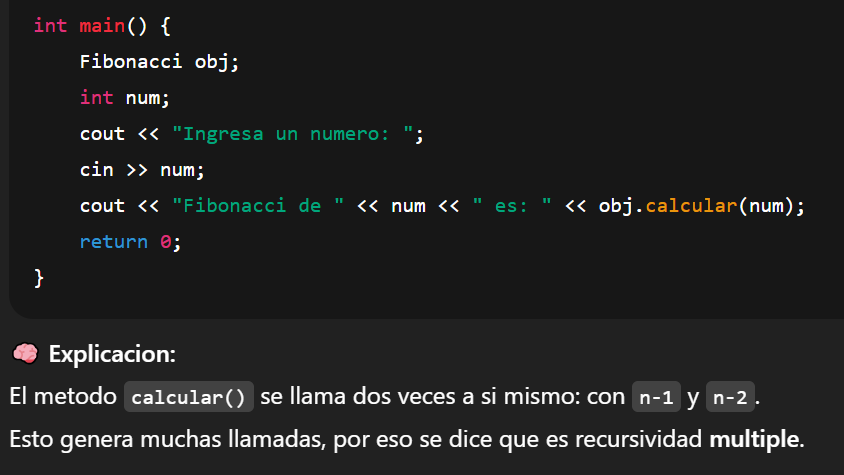


3. Recursividad multiple

Una funcion o metodo se llama a si mismo mas de una vez dentro de su cuerpo.

Ejemplo: serie Fibonacci (recursividad multiple)





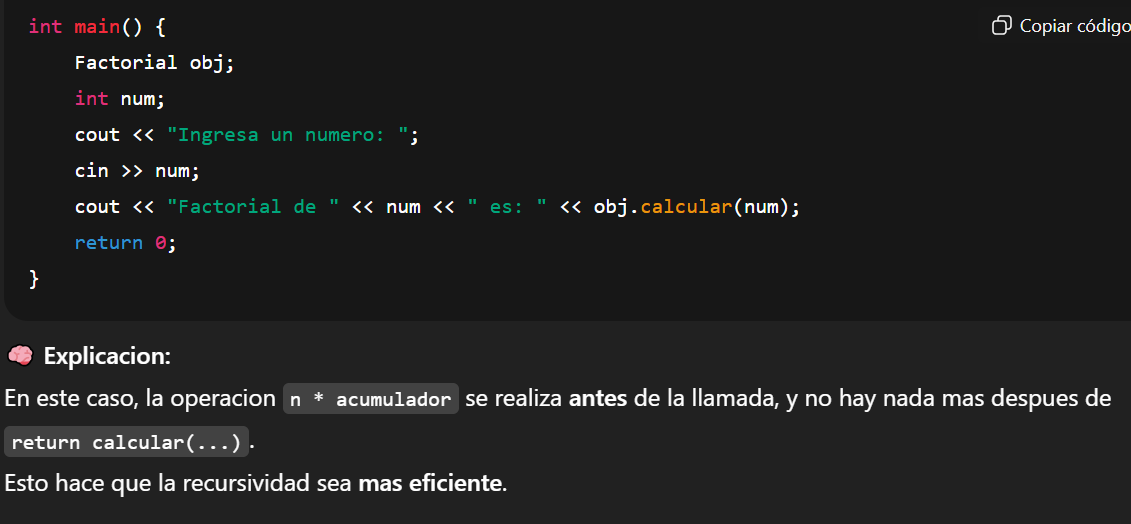
**4. Recursividad de cola (tail recursion)**

La **llamada recursiva es la ultima instruccion** que se ejecuta dentro del metodo.  
No hay operaciones pendientes despues de la llamada.

**Ejemplo: factorial con recursividad de cola**

Captura de pantalla de computadora

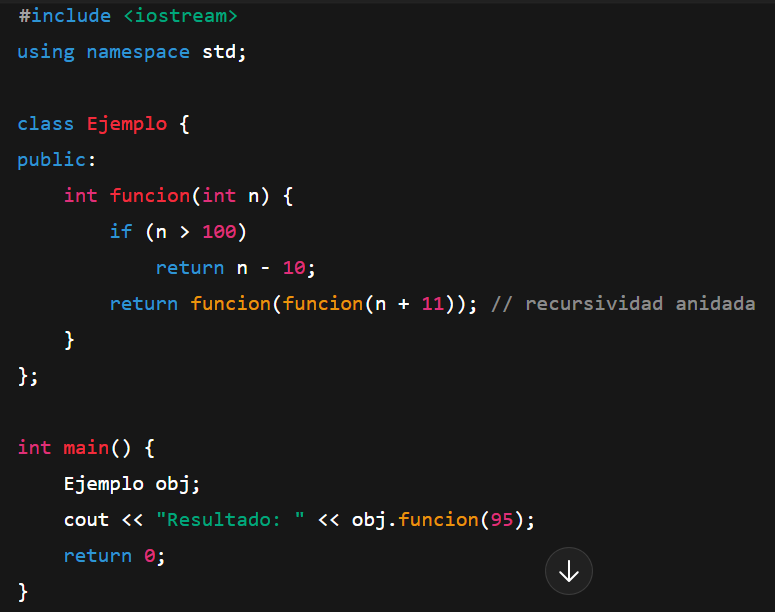
Descripción generada automáticamente



**5. Recursividad anidada**

La llamada recursiva **esta dentro de otra llamada recursiva**.

**Ejemplo:**



Explicacion:

Aqui, el metodo funcion() se llama a si mismo dentro de otra llamada del mismo metodo.

Primero se evalua funcion(n + 11) y luego el resultado se usa en otra llamada de funcion().

6. Recursividad mutual o cruzada

Similar a la indirecta, pero involucra mas de dos metodos u objetos, donde el llamado se “cruza” entre ellos.

Ejemplo basico:

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

1. **PROGRAMAS RECURSIVOS**

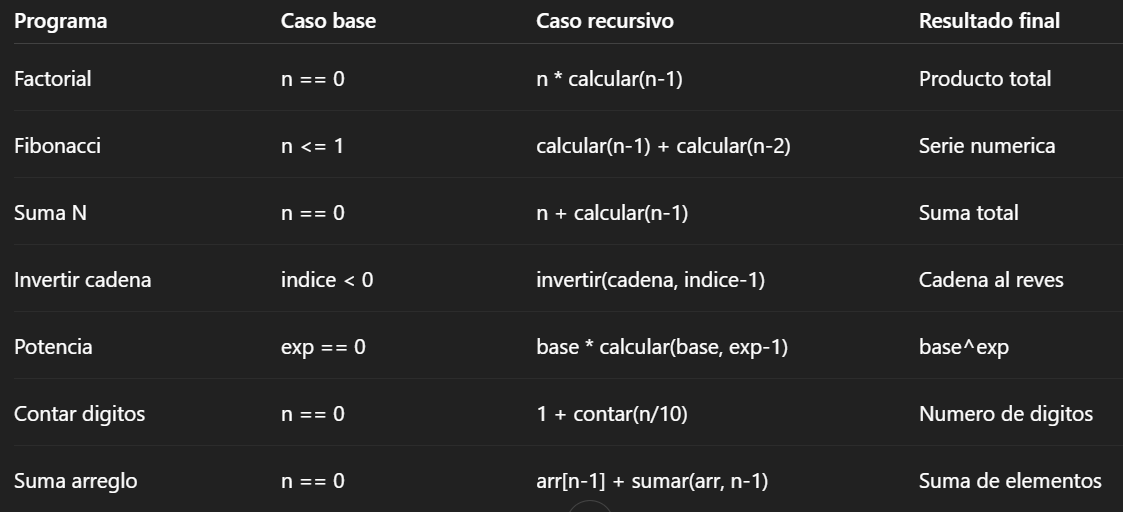
Un programa recursivo es aquel que contiene al menos una funcion o metodo que se llama a si mismo (directa o indirectamente) para resolver un problema.

La recursividad se usa cuando un problema puede dividirse en subproblemas del mismo tipo, lo que permite obtener soluciones mas simples y elegantes.

**Estructura general de un programa recursivo**

Todo programa recursivo sigue esta estructura basica:

* Entrada de datos (valores o parametros iniciales).
* Definicion de la funcion o metodo recursivo, que contiene:
* Caso base: condicion que detiene la recursividad.
* Caso recursivo: llamada a la misma funcion con un problema mas pequeño.
* Salida o resultado final (cuando se llega al caso base y se combinan los resultados).



EJEMPLO DE RECURSIVIDAD: hacer una suma de matrices tridimensional utilizando interfaz enserar ingresar imprimir segmentar (debe imprimir los 3 resultados de la matrices) reservar memoria y retorne un int doble puntero (se debe parametrizar por dirección)

DIAGRAMA UML

Enserar: creado en una interfaz

Void encerar (int \*\*&, int&, int&, int&);

Ingresar:

Void ingresar (int \*\*&, int&, int&, int&);

Procesando:

Imprimir: muestra el resultado de la suma

Void imprimir (int \*\*&, int&, int&, int&);

Reservar memoria:

Int \*\*reservar (int&p, int&f, int&c){

Int \*\*mat = (int\*\*) malloc (…);

Observación: // Reservar memoria para capas

int \*\*\*matriz = (int\*\*\*) malloc(x \* sizeof(int\*\*)); no copiar

for (int i=0; i<p; i++){

\*(mat+i) = (int \*\*) malloc (p \* sizeof (int\*)); //casting

For ( int i==; i<f; j++){

\*(mat+i) = (int \*\*) malloc (f \* sizeof (int\*));

For ( int i=0 ; i<f; j++){

\*(mat+j) = (int \*\*) malloc (f \* sizeof (int\*));

For ( int k=0 ; k<f; k++){

\*(mat+k) = (int \*\*) malloc (k\* sizeof (int\*));

\*(\*mat +i)+k = (int\*) malloc(p\*size (int\*))

}

}

}

Segmentar:

Int \*\*mat;

Retornar: int\*\* reservarMemoria ( int&, int&, int&);

[][]

Liberar memoria: matriz, profundidad, fila, columna

For (int i=0; i<p; i++){

For(int k=0; k<c; k++)

Free (\*(\*matriz+i, \*matriz+k));

}

= Free \*( matriz);

Recursividad utilizando funciones landa implementar la siguiente clase hacemos con el ing

Implementar arreglar el modelo template punteros herramienta manejar sumar de matrices tridimensional k=2 tendre que ingresar 4 matrices

Subir al git todo

|  |
| --- |
| Matriz |
| -Matriz[][]: int |
| +set  +get |

1. **LISTAS**
2. **TIPOS DE LISTAS**
3. **OPERACIONES BASICAS CON LISTAS**
4. **APLICACIONES**
5. **PILAS**
6. **REPRESENTACION EN MEMORIA ESTATICA Y DINAMICA**
7. **OPERACIONES BASICAS CON PILAS**
8. **NOTACION INFIJA, PREFIJA Y POSTFIJA**
9. **RECURSIVIDAD CON AYUDA DE PILAS**
10. **APLICACIONES**
11. **COLAS**
12. **REPRESENTACION EN MEMORIA ESTATICA Y DINAMICA**
13. **TIPOS DE COLAS**
14. **OPERACIONES CON COLAS**
15. **APLICACIONES: COLAS DE PRIORIDAD**