**Tipos de recursividad**

### Tipos de Recursividad

### La clasificación de la recursividad se basa en cómo se realizan las llamadas y las interacciones entre las funciones recursivas. Aquí se describen algunos tipos comunes:

**1.1 Recursividad Directa:** En este tipo, una función se llama a sí misma de manera directa. La función realiza una tarea y resuelve un problema más pequeño de manera directa mediante la llamada recursiva.

**Ejemplo:**

#include <iostream>

**using** **namespace** std;

**class** **Potencia** {

**public:**

**int** calcularPotencia(**int** base, **int** exponente) {

**if** (exponente == **0**) {

**return** **1**;

} **else** {

**return** base \* calcularPotencia(base, exponente - **1**);

}

}

};

**int** **main**() {

Potencia potencia;

**int** base, exponente;

cout << "Ingrese la base: ";

cin >> base;

cout << "Ingrese el exponente: ";

cin >> exponente;

cout << "Potencia: " << potencia.calcularPotencia(base, exponente) << endl;

**return** **0**;

}

La función calcularPotencia utiliza la recursividad para calcular la potencia de un número. Puede ser más práctico en situaciones donde se necesitan operaciones repetitivas de multiplicación.

**1.2 Recursividad Indirecta:** En este caso, dos o más funciones se llaman entre sí de manera cíclica para resolver un problema. Cada función realiza una parte del trabajo y delega el resto a las demás funciones.

#include <iostream>

**using** **namespace** std;

**class** **SecuenciaParImpar** {

**public:**

**void** generarSecuencia(**int** n) {

**if** (n > **0**) {

cout << n << " ";

**if** (n % **2** == **0**) {

SecuenciaParImpar spi;

spi.generarSecuencia(n / **2**);

// Llamada recursiva para números pares

} **else** {

SecuenciaParImpar spi;

spi.generarSecuencia(**3** \* n + **1**);

// Llamada recursiva para números impares

}

}

}

};

**int** **main**() {

SecuenciaParImpar spi;

cout << "Secuencia para 7: " << endl;

spi.generarSecuencia(**7**);

cout << "**\n**";

**return** **0**;

}

En este ejemplo, la función generarSecuencia utiliza la recursividad indirecta para generar una secuencia específica. La llamada recursiva depende de si el número es par o impar.

**1.3 Recursividad de Cola:** En esta variante, la llamada recursiva es la última operación realizada dentro de la función. Especialmente importante en algunos lenguajes de programación que pueden optimizar este tipo de recursividad convirtiéndola en un bucle iterativo.

#include <iostream>

**using** **namespace** std;

**class** **Factorial** {

**public:**

**int** calcularFactorial(**int** n, **int** resultado = **1**) {

**if** (n == **0**) {

**return** resultado;

} **else** {

**return** calcularFactorial(n - **1**, n \* resultado);

}

}

};

**int** **main**() {

Factorial factorial;

**int** numero;

cout << "Ingrese un numero para calcular su factorial: ";

cin >> numero;

cout << "Factorial de " << numero << ": " << factorial.calcularFactorial(numero) << endl;

**return** **0**;

}

En este ejemplo, la función calcularFactorial utiliza la recursividad de cola para calcular el factorial de un número. La multiplicación se realiza acumulando el resultado en un parámetro adicional.

Estos tipos de recursividad ofrecen diferentes enfoques para abordar problemas y es crucial elegir el tipo que mejor se adapte al contexto del problema y a las características del lenguaje de programación utilizado. La recursividad, cuando se implementa de manera adecuada, puede simplificar la estructura del código y facilitar la comprensión del problema, pero también requiere precaución para evitar problemas como el desbordamiento de la pila de llamadas.

1. **Conclusiones:**

La recursividad en C++ ofrece una forma poderosa y elegante de abordar problemas al dividirlos en casos más simples y resolverlos de manera iterativa. Sin embargo, su uso requiere precaución debido al riesgo de desbordamiento, ya que cada llamada recursiva agrega una nueva entrada de llamadas, lo que puede consumir recursos de memoria significativos. Además, aunque la recursividad puede ser intuitiva y fácil de entender para ciertos problemas, en otros casos puede ser menos eficiente que enfoques iterativos equivalentes, lo que puede afectar el rendimiento del programa en situaciones críticas. Por lo tanto, al usar recursividad en C++, es importante evaluar cuidadosamente la complejidad del problema y considerar las implicaciones de rendimiento y memoria antes de elegir entre enfoques recursivos o iterativos.

1. **Referencias**

* Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to Algorithms (4th ed.). MIT Press.

<https://dl.ebooksworld.ir/books/Introduction.to.Algorithms.4th.Leiserson.Stein.Rivest.Cormen.MIT.Press.9780262046305.EBooksWorld.ir.pdf>

* Drozdek, Adam. "Data Structures and Algorithms in C++." 5th Edition, Cengage Learning, 2019.

<https://itlectures.ro/wpcontent/uploads/2016/04/AdamDrozdek__DataStructures_and_Algorithms_in_C_4Ed.pdf>