Recursion

This C++ file is a focused and in-depth exploration of **recursion**, a fundamental concept in computer science. It is meticulously organized to not only demonstrate what recursion is but also to explore its various forms, applications, and performance implications.

The file starts by differentiating between the basic patterns of **Tail Recursion** and **Head Recursion**, showing how a simple change in the order of operations can completely alter the program's output. It then introduces more complex patterns like **Tree Recursion** (where a function calls itself multiple times), **Indirect Recursion** (a cycle of function calls), and the mind-bending **Nested Recursion**. This systematic approach helps in building an intuition for how recursive calls unfold.

The code is rich with practical examples of classic algorithms solved recursively. These include mathematical problems such as calculating the **sum of natural numbers**, **factorials**, and **exponents** (using an efficient method). It also tackles more advanced topics like the **Taylor series expansion**, providing multiple recursive implementations, including an optimized one using Horner's Rule.

One of the most valuable sections directly compares different solutions for the **Fibonacci sequence**. It presents the naive (but elegant) recursive solution, a highly efficient **iterative** (loop-based) solution, and an optimized recursive solution using **memoization**. This comparison is crucial as it highlights the performance pitfalls of recursion and introduces a powerful dynamic programming technique to overcome them. The file concludes with well-known recursive puzzles and formulas like the **Tower of Hanoi** and **nCr combinations**, solidifying the understanding of how to apply recursive thinking to complex problems. This file is an excellent study guide for anyone looking to master recursion.

**Tail Recursion**

This code demonstrates **Tail Recursion**. This is a type of recursion where the recursive call is the very last operation in the function. There is no pending operation to be performed on the return of the recursive call. Modern compilers can optimize tail recursion to be as efficient as a loop.

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

This is the base case check. The recursion continues as long as n is greater than 0 and stops when n becomes 0, preventing an infinite loop.

* printf("%d ", n); fun(n-1);

The function first performs its main operation (printing the number) and then makes the recursive call. Because nothing happens after fun(n-1) returns, this is a classic example of tail recursion.

**Head Recursion**

This code demonstrates **Head Recursion**. In this type of recursion, the recursive call is the very first operation performed inside the function. All other operations are executed after the recursive call returns, during the "returning phase" of the recursion.

* fun(n-1); printf("%d ", n);

The function first calls itself with a smaller value. The printf statement is only executed after the entire chain of recursive calls (fun(2), fun(1), fun(0)) has completed and starts returning. This results in the numbers being printed in ascending order.

**Returning function and adding constant**

This code shows how a recursive function can build up a value as it returns. Each function call waits for the result of the call it made, adds its own value (n) to that result, and then returns the new sum up the call stack.

* if(n>0){ ... }; return 0;

The return 0; line defines the base case. When n reaches 0, the recursion stops and returns a starting value of 0.

* return fun(n-1) + n;

This is the core of the recursion. The function calls itself with n-1, and when that call eventually returns a value, the current function adds its own n to it. For n=5, this calculates (fun(4) + 5), which is ((fun(3)+4) + 5), and so on.

**Returning function and adding static variable**

This example demonstrates how a static variable behaves within a recursive function. A static variable is initialized only once and retains its value across all subsequent calls to the function, acting like a global variable but with its scope limited to the function.

* static int x = 0;

This line declares a static variable x. It is set to 0 only the very first time fun is called. In all subsequent recursive calls, this line is effectively skipped, and x retains its most recent value.

* x++; return fun(n-1) + x;

In each recursive call, x is incremented. So, for n=5, x will become 1, 2, 3, 4, 5 during the calling phase. The sum is then calculated during the returning phase using these incremented values of x.

**Tree Recursion**

This code is an example of **Tree Recursion**, where a function calls itself more than once within its own code. This creates a tree-like structure of function calls.

* printf("%d ", n);

The value of n is printed when the function is first called.

* fun(n-1); fun(n-1);

The function makes two recursive calls to itself with the same parameter. For an input of 3, fun(3) will call fun(2) twice. Each of those fun(2) calls will call fun(1) twice, and so on, branching out like a tree.

**Indirect Recursion**

This code demonstrates **Indirect Recursion**. This occurs when two or more functions call each other in a circular chain. Here, funA calls funB, and funB in turn calls funA, creating a recursive cycle.

* void funB(int n);

This is a forward declaration. It tells the compiler about the existence of funB before it is actually defined, which is necessary because funA calls funB before the compiler has seen funB's implementation.

* funB(n-1);

Inside funA, a call is made to funB.

* funA(n/2);

Inside funB, a call is made back to funA, completing the recursive loop. The recursion stops when the conditions in the if statements are no longer met.

**Nested recursion**

This code shows **Nested Recursion**, where a recursive call appears as a parameter to another recursive call. This can be very complex to trace and can grow in value very quickly.

* if(n>100) return n -10;

This is the base case. The recursion will stop unwinding only when a value of n greater than 100 is passed into the function. It then returns n-10.

* return fun(fun(n+11));

This is the nested recursive call. To evaluate fun(95), the program must first evaluate the inner call, fun(95+11) which is fun(106). Since 106 > 100, fun(106) returns 106-10 = 96. This result is then passed to the outer call, becoming fun(96). The process repeats until the final result is calculated.

**Sum of n natural numbers using recursion**

This code provides a classic recursive solution to find the sum of the first n natural numbers (1 + 2 + 3 + ... + n).

* if(n==0){ return 0; }

This is the base case. The sum of 0 numbers is 0. This stops the recursion.

* return sum(n-1)+n;

This is the recursive step. It assumes that the function can correctly calculate the sum of the first n-1 numbers. It then adds the current number n to that result to get the final answer.

**Factorial of n numbers**

This code calculates the factorial of a number (n!), which is the product of all positive integers up to n. It uses a simple recursive approach.

* if(n==0 || n ==1){ return 1; }

This is the base case. The factorial of 0 and 1 is defined as 1. This condition terminates the recursion.

* return fact(n-1)\*n;

This is the recursive definition of factorial. It states that the factorial of n is n multiplied by the factorial of (n-1). The function relies on itself to solve the smaller sub-problem.

**Exponents using recursion**

This code calculates m raised to the power of n (m^n) using an efficient recursive method. It reduces the number of multiplications by halving the exponent in each step.

* if(n==0){ return 1; }

The base case: any number raised to the power of 0 is 1.

* if(n%2==0){ return exp(m\*m, n/2); }

If the power n is even, we can rewrite m^n as (m^2)^(n/2). This single step halves the exponent, making the calculation much faster.

* else{ return m\*exp(m\*m, (n-1)/2); }

If the power n is odd, we can rewrite m^n as m \* m^(n-1). Since (n-1) is now even, we can apply the logic from the previous step, resulting in m \* (m^2)^((n-1)/2).

**Taylor series using recursion**

This code calculates the Taylor series expansion for $e^x$ up to n terms using a straightforward recursion. It uses static variables to keep track of the numerator (p) and denominator (f) of each term across recursive calls.

* static double p = 1, f = 1;

These static variables store the values of power (p) and factorial (f) for the current term. They are initialized once and retain their values across the recursive calls.

* r = e(x, n-1);

The function first recursively calls itself to calculate the sum of the series up to the (n-1)th term.

* p = p\*x; f = f\*n; return r+p/f;

After the recursive call returns, the power (p) and factorial (f) are updated for the current term n, and the new term (p/f) is added to the previously calculated sum (r).

**Taylor series(with horner's rule) using recursion**

This is a more efficient recursive method for calculating the Taylor series using **Horner's Rule**. This method reduces the number of multiplications and improves performance by factoring out common terms. The calculation starts from the nth term and works its way inwards.

* static double s;

The static variable s holds the result of the calculation as it's built up during the recursive calls.

* s=1+x\*s/n;

This is the core of Horner's rule. Instead of calculating powers and factorials separately, it reuses the result of the previous term (s) to compute the current one in a single line.

* return e(x, n-1);

The function calls itself with n-1, and the updated value of s is carried into the next call. The final result is returned when n reaches the base case of 0.

**Taylor series using normal coding**

This code calculates the Taylor series expansion for $e^x$ using a standard **iterative approach** with a for loop. It is often more straightforward to understand and can be more efficient than recursion due to the absence of function call overhead.

* double s = 1; double num=1; double den = 1;

These variables are initialized to hold the total sum s, the numerator num (xi), and the denominator den (i!) for each term in the series. s starts at 1 for the first term of the series.

* for(i = 1; i<=n; i++){ ... }

This loop iterates from 1 to n to calculate each term of the series.

* num\*=x; den\*=i; s+=num/den;

Inside the loop, for each iteration i, the numerator is updated by multiplying by x, the denominator is updated by multiplying by i, and the new term (num/den) is added to the running total s.

**Fibonacci Series using iteration, recursion, memoization**

This section compares three different methods to calculate the nth Fibonacci number, showcasing the trade-offs between them in terms of simplicity and performance.

* for(i=2; i<=n; i++){ s = f0+f1; ... }

This is the iterative approach. It is the most efficient in terms of both time (O(n)) and space. It simply uses a loop to calculate each number based on the previous two, starting from 0 and 1.

* return rfib(n-2) + rfib(n-1);

This is the pure recursive approach. It is elegant and directly follows the mathematical definition of the Fibonacci sequence. However, it is extremely inefficient (time complexity of O(2n)) because it recalculates the same values many times.

* if(f[n-2] == -1){ f[n-2] = mfib(n-2); }

This is the memoization approach. It's a top-down dynamic programming technique that enhances the recursive solution. It uses a global array f to store results that have already been calculated. Before computing mfib(n-2), it first checks if the result is already in the array. This avoids redundant calculations and brings the time complexity down to O(n).

**nCr using recursion**

This code shows two ways to calculate the combination formula (nCr), which represents the number of ways to choose r items from a set of n items.

* int nCr(int n, int r){ ... }

This function calculates nCr using the standard formula nCr=n!/(r!∗(n−r)!). It relies on a separate fact() function to compute the factorials. This approach can be slow and prone to overflow for large numbers.

* int NCR(int n, int r){ ... }

This function uses a recursive approach based on the properties of Pascal's Triangle, where nCr=(n−1)C(r−1)+(n−1)Cr.

* if(r==n || r==0) return 1;

This is the base case for the recursive NCR function. The number of ways to choose all items (r==n) or no items (r==0) is always 1.

**Tower of hanoi using recursion**

This code provides an elegant recursive solution to the classic **Tower of Hanoi** puzzle. The goal is to move n disks from a source tower (A) to a destination tower (C) using an auxiliary tower (B), without placing a larger disk on top of a smaller one.

* void TOH(int n, int A, int B, int C){ ... }

The function takes the number of disks n and the numbers representing the three towers (A, B, C) as input.

* TOH(n-1, A, C, B);

Step 1: Recursively move the top n-1 disks from the source tower A to the auxiliary tower B, using C as the destination for this sub-problem.

* printf("(%d,%d) \n", A, C);

Step 2: Move the single remaining (largest) disk from the source tower A to the destination tower C.

* TOH(n-1, B, A, C);

Step 3: Recursively move the n-1 disks from the auxiliary tower B to the final destination tower C, using the original source tower A as the auxiliary for this sub-problem.