Bài 12: Đệ Quy – Recursion

I. Vấn Đề

- Giả sử bạn phải sơn một vài quả bóng. Nếu bạn làm một mình, sẽ mất rất nhiều thời gian. Một điều bạn có thể làm là nhờ bạn của mình giúp đỡ. Giả sử rằng bạn có cùng tốc độ làm việc, nhiệm vụ sẽ được hoàn thành trong một nửa thời gian. Bây giờ, thay vì chỉ nhờ một người bạn giúp đỡ, bạn nhờ nhiều người bạn giúp đỡ sao cho mỗi người bạn chỉ phải sơn một quả bóng. Nhiệm vụ sẽ được hoàn thành nhanh hơn nhiều so với khi bạn tự làm. Đệ quy là một kỹ thuật giải quyết vấn đề hoạt động theo cách tương tự.

- Đệ quy trong C++ là một kỹ thuật trong đó một hàm gọi chính nó nhiều lần cho đến khi một điều kiện nhất định được thỏa mãn. Nói cách khác, đệ quy là quá trình giải quyết một vấn đề bằng cách chia nhỏ nó thành các vấn đề con nhỏ hơn, đơn giản hơn.

II. Khái niệm

- Một hàm gọi chính nó được gọi là **hàm đệ quy** . Khi một hàm đệ quy được gọi, nó thực thi một tập lệnh và sau đó gọi chính nó để thực thi cùng một tập lệnh với đầu vào nhỏ hơn. Quá trình này tiếp tục cho đến khi đạt đến trường hợp cơ sở, đây là điều kiện dừng đệ quy và trả về một giá trị.
- Cấu trúc cú pháp của đệ quy

```
return_type recursive_func {
   .... // Điều kiện cơ sở
        ....  // Trường hợp đệ quy
}
```

- Điều kiện cơ sở là điều kiện được sử dụng để chấm dứt đệ quy. Hàm đệ quy sẽ tiếp tục gọi chính nó cho đến khi điều kiện cơ sở được thỏa mãn.

- Trường hợp đệ quy là cách mà lệnh gọi đệ quy có mặt trong hàm. Trường hợp đệ quy có thể chứa nhiều lệnh gọi đệ quy hoặc các tham số khác nhau sao cho khi kết thúc, điều kiện cơ sở được thỏa mãn và đệ quy kết thúc.

III. Ví dụ về đệ quy

- Ví dụ:

```cpp
// C++ Program to calculate the sum of first N natural
// numbers using recursion
#include <iostream>
using namespace std;

int nSum(int n)
{
    // base condition to terminate the recursion when N = 0
    if (n == 0) {
        return 0;
    }

    // recursive case / recursive call
    int res = n + nSum(n - 1);

    return res;
}

int main()
{
    int n = 5;

    // calling the function
    int sum = nSum(n);

    cout << "Sum = " << sum;
    return 0;
}
```

- **Hàm đệ quy:** nSum() là hàm đệ quy
- **Trường hợp đệ quy:** Biểu thức **int res = n + nSum(n – 1)** là trường hợp đệ quy.
- **Điều kiện cơ sở:** Điều kiện cơ sở là **if (n == 0) { return 0;}**

IV. Hoạt động của đệ quy

Để hiểu cách đệ quy C hoạt động, chúng ta sẽ tham khảo lại ví dụ trên và theo dõi luồng của chương trình.

1. Trong hàm nSum(), **Trường hợp đệ quy** là

```
int res = n + nTổng(n - 1);
```

2. Trong ví dụ, n = 5, vì vậy trong trường hợp đệ quy **của nSum(5)** , chúng ta có

```
int res = 5 + nSum(4);
```

3. Trong **nSum(4)** , trường hợp đệ quy và mọi thứ khác sẽ giống nhau, nhưng n = 4. Hãy đánh giá trường hợp đệ quy cho n = 4,

```
int res = 4 + nSum(3);
```

4. Tương tự với **nSum(3), nSum(2) và nSum(1)**

```
int res = 3 + nSum(2); // nSum(3)
int res = 2 + nSum(1); // nSum(2)
int res = 1 + nSum(0); // nSum(1)
```

5. Bây giờ hãy nhớ lại rằng **giá trị trả về** của hàm nSum() trong cùng số nguyên này có tên là **res** . Vì vậy, thay vì hàm, chúng ta có thể đặt giá trị trả về bởi các hàm này. Như vậy, đối với nSum(5), chúng ta có

```
int res = 5 + 4 + nTổng(3);
```

6. Tương tự, đặt các giá trị trả về của nSum() cho mỗi n, chúng ta có

```
int res = 5 + 4 + 3 + 2 + 1 + nSum(0);
```

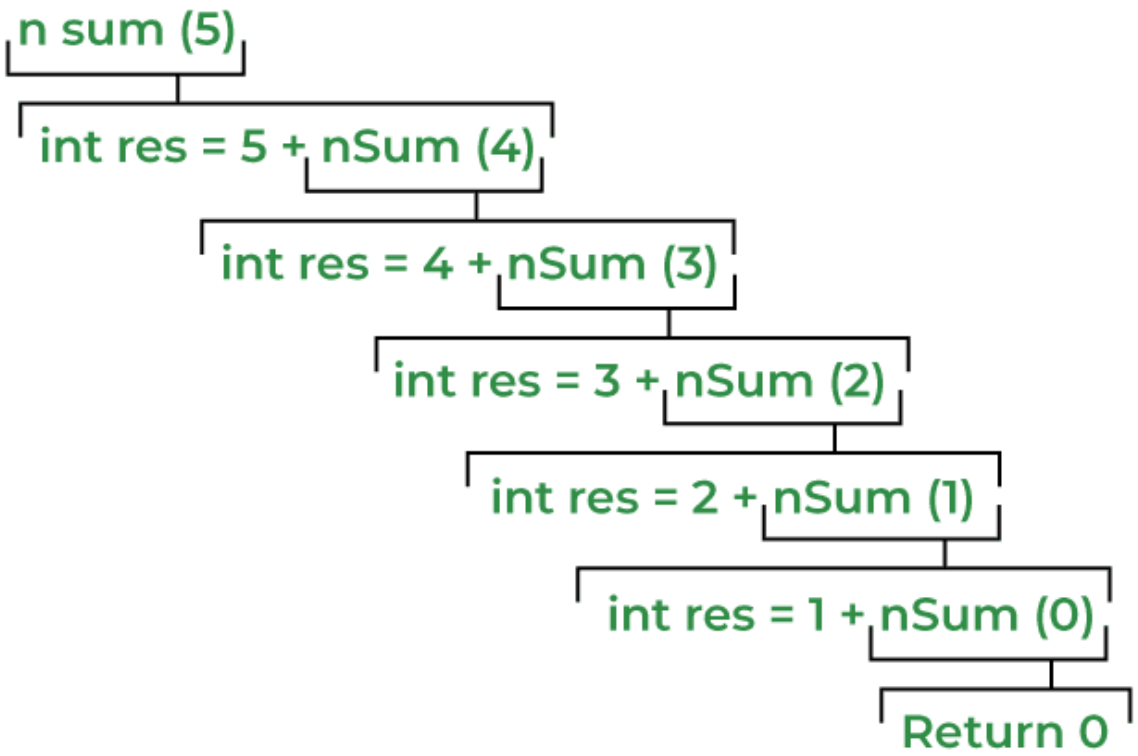
7. Trong hàm nSum(), **điều kiện cơ sở** là

```
nếu (n == 0) {
   trả về 0;
}
```

điều đó có nghĩa là khi nSum(0) sẽ trả về 0. Đặt giá trị này vào trường hợp đệ quy của nSum(5), chúng ta nhận được

```
số nguyên = 5 + 4 + 3 + 2 + 1 + 0;
      = 15
```

8. Tại thời điểm này, chúng ta có thể thấy rằng không còn lệnh gọi hàm nào trong trường hợp đệ quy. Vì vậy, đệ quy sẽ dừng tại đây và giá trị cuối cùng mà hàm trả về sẽ là **15** , là tổng của 5 số tự nhiên đầu tiên.



V. Quản lí bộ nhớ

Giống như tất cả các hàm khác, dữ liệu của hàm đệ quy được lưu trữ trong bộ nhớ ngăn xếp dưới dạng khung ngăn xếp. Khung ngăn xếp này bị xóa sau khi hàm trả về một giá trị nào đó. Trong đệ quy,

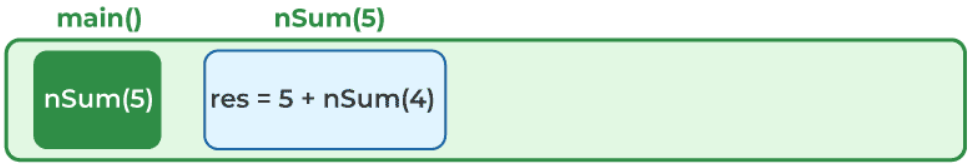
Lệnh gọi hàm được thực hiện trước khi trả về giá trị, do đó khung ngăn xếp cho các lệnh gọi đệ quy tiến bộ được lưu trữ trên các khung ngăn xếp hiện có trong bộ nhớ ngăn xếp.

Khi bản sao hàm trên cùng trả về một giá trị nào đó, khung ngăn xếp của nó sẽ bị hủy và quyền điều khiển sẽ chuyển đến hàm ngay trước bản sao cụ thể đó sau điểm mà lệnh gọi đệ quy được thực hiện cho bản sao trên cùng.

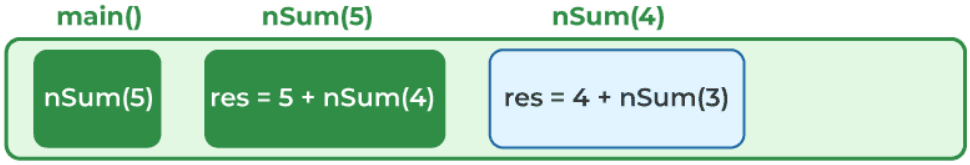
Trình biên dịch duy trì một con trỏ lệnh để theo dõi nơi trả về sau khi thực thi hàm.

Hãy cùng xem xét ví dụ trên và tìm hiểu cách quản lý bộ nhớ của hàm nSum(5).

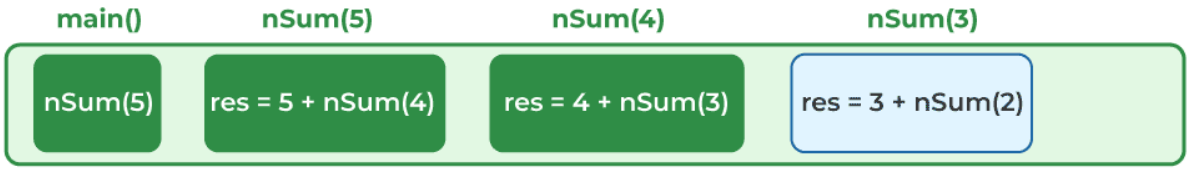
Bước 1: Khi nSum() được gọi từ hàm main() với đối số là 5, **một khung ngăn xếp cho nSum(5)** sẽ được tạo.



Bước 2: Trong khi thực thi nSum(5), một **lệnh gọi đệ quy** được phát hiện là **nSum(4)** . Trình biên dịch sẽ tạo một khung ngăn xếp mới trên khung ngăn xếp của nSum(5) và duy trì một con trỏ lệnh tại câu lệnh mà **nSum(4)** được phát hiện.

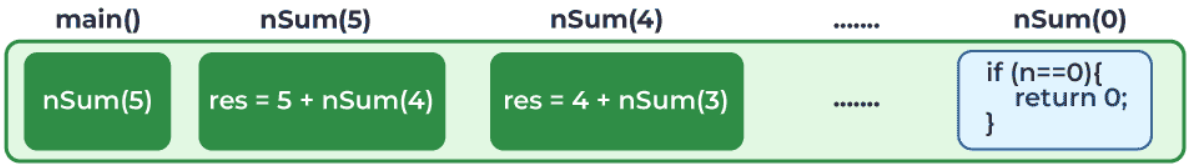


Bước 3: Việc thực thi nSum(4) sẽ bắt đầu, nhưng giống như hàm trước, chúng ta gặp một lệnh gọi đệ quy khác là nSum(3). Trình biên dịch sẽ lại thực hiện các bước tương tự và duy trì một con trỏ lệnh và khung ngăn xếp khác cho nSum(3).

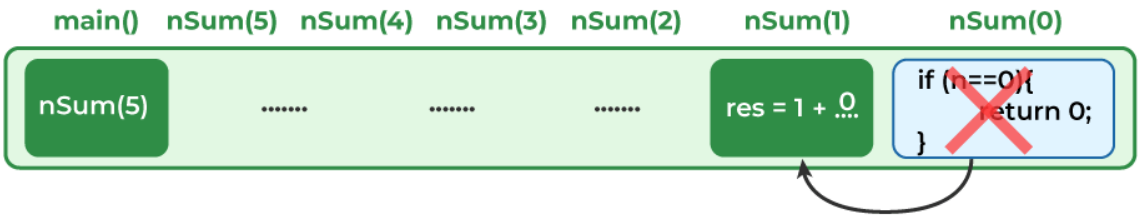


Bước 4: Điều tương tự sẽ xảy ra với quá trình thực thi nSum(3), nSum(2) và nSum(1).

Bước 5: Nhưng khi điều khiển đến nSum(0), điều kiện (n == 0) trở thành đúng và câu lệnh **return 0** được thực thi.

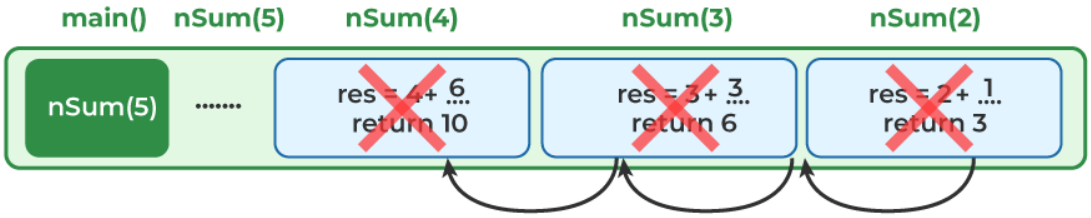


Bước 6: Khi giá trị được trả về bởi nSum(0), khung ngăn xếp cho nSum(0) sẽ bị hủy và bằng cách sử dụng con trỏ lệnh, điều khiển chương trình sẽ trở về hàm nSum(1) và lệnh gọi nSum(0) sẽ được thay thế bằng giá trị 0.

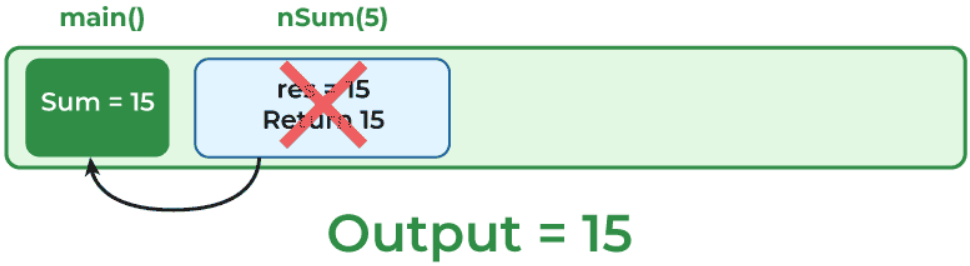


Bước 7: Bây giờ, trong nSum(1), biểu thức **int res = 1 + 0** sẽ được đánh giá và câu lệnh **return res** sẽ được thực thi. Điều khiển chương trình sẽ di chuyển đến nSum(2) bằng con trỏ lệnh của nó.

Bước 8: Trong nSum(2), lệnh gọi nSum(1) sẽ được thay thế bằng giá trị mà nó trả về, là 1. Vì vậy, sau khi đánh giá **int res = 2 + 1, 3** sẽ được trả về nSum(3). Điều tương tự sẽ tiếp tục xảy ra cho đến khi điều khiển đến nSum(5) một lần nữa.



Bước 9: Khi điều khiển đạt đến nSum(5), biểu thức **int res = 5 + nSum(4)** sẽ trông giống như **int res = 5 + 10.** Cuối cùng, giá trị này sẽ được trả về hàm main() và việc thực thi hàm nSum() sẽ dừng lại.



Stack Overflow là gì?

- Tràn ngăn xếp là một trong những lỗi phổ biến nhất liên quan đến đệ quy xảy ra khi một hàm gọi chính nó quá nhiều lần. Như chúng ta đã biết, mỗi lệnh gọi đệ quy yêu cầu không gian riêng trong bộ nhớ ngăn xếp hạn chế. Khi có một số lượng lớn các lệnh gọi đệ quy hoặc đệ quy diễn ra vô hạn lần, bộ nhớ ngăn xếp này có thể bị cạn kiệt và không thể lưu trữ thêm dữ liệu dẫn đến việc chương trình bị chấm dứt.

VI. Ứng dụng của đệ quy

- Đệ quy có nhiều ứng dụng trong khoa học máy tính và lập trình. Sau đây là một số ứng dụng phổ biến nhất của đệ quy:

- **Giải quyết:** Dãy số Fibonacci, Hàm giai thừa, Đảo ngược mảng, Tháp Hà Nội.
- **Quay lui:** Đây là một kỹ thuật giải quyết vấn đề bằng cách thử các giải pháp khác nhau và hoàn tác nếu chúng không hiệu quả. Thuật toán đệ quy thường được sử dụng trong quay lui.
- **Thuật toán tìm kiếm và sắp xếp:** Nhiều thuật toán tìm kiếm và sắp xếp, chẳng hạn như tìm kiếm nhị phân và sắp xếp nhanh, sử dụng đệ quy để chia vấn đề thành các vấn đề nhỏ hơn.
- **Duyệt cây và đồ thị:** Các thuật toán đệ quy thường được sử dụng để duyệt cây và đồ thị, chẳng hạn như tìm kiếm theo chiều sâu và tìm kiếm theo chiều rộng.
- **Tính toán toán học:** Đệ quy cũng được sử dụng trong nhiều phép tính toán học, chẳng hạn như hàm giai thừa và dãy số Fibonacci.
- **Lập trình động:** Đây là một kỹ thuật giải quyết các vấn đề tối ưu hóa bằng cách chia nhỏ chúng thành các vấn đề con nhỏ hơn. Thuật toán đệ quy thường được sử dụng trong lập trình động.

- Nhìn chung, đệ quy là một kỹ thuật mạnh mẽ và linh hoạt có thể được sử dụng để giải quyết nhiều vấn đề trong lập trình và khoa học máy tính.

VII. Nhược điểm của đệ quy

- **Hiệu suất:** Trong một số trường hợp, thuật toán đệ quy có thể kém hiệu quả hơn thuật toán lặp, đặc biệt nếu cấu trúc dữ liệu lớn hoặc nếu đệ quy đi quá sâu.
- **Sử dụng bộ nhớ:** Thuật toán đệ quy có thể sử dụng rất nhiều bộ nhớ, đặc biệt là nếu đệ quy đi quá sâu hoặc nếu cấu trúc dữ liệu lớn. Mỗi lệnh gọi đệ quy tạo một khung ngăn xếp mới trên ngăn xếp lệnh gọi, có thể nhanh chóng tăng lên một lượng bộ nhớ đáng kể.
- **Độ phức tạp của mã:** Thuật toán đệ quy có thể phức tạp hơn thuật toán lặp.
- **Gỡ lỗi:** Thuật toán đệ quy có thể khó gỡ lỗi hơn thuật toán lặp, đặc biệt nếu đệ quy đi quá sâu hoặc nếu chương trình sử dụng nhiều lệnh gọi đệ quy.
- **Tràn ngăn xếp:** Nếu đệ quy đi quá sâu, nó có thể gây ra lỗi tràn ngăn xếp, có thể làm chương trình bị sập.

VIII. Luyện tập

1. Tính Giai Thừa

Bài Tập : Viết một hàm đệ quy để tính giai thừa của một số nguyên không âm nnn.

- **Đầu vào**: n=5
- **Đầu ra**: 120 (vì 5!=120)

2. Tìm Kiếm Nhị Phân

Bài Tập: Viết một hàm đệ quy để thực hiện tìm kiếm nhị phân trong một mảng đã được sắp xếp.

- **Đầu vào**: Mảng: [1, 2, 3, 4, 5], Tìm kiếm: 3
- **Đầu ra**: 222 (chỉ số của số 3 trong mảng)

Test Cases:

- Mảng: [1, 2, 3, 4, 5], Tìm kiếm: $1 \rightarrow 0$
- Mảng: [1, 2, 3, 4, 5], Tìm kiếm: $6 \rightarrow -1$ (không tìm thấy)