Đệ Quy (Recursion)

Hôm nay chúng ta sẽ quay lại với ĐỆ QUY. Thực chất đệ quy không phức tạp như mọi người nghĩ, đệ quy cũng chỉ là một hàm nhưng hàm này đặc biệt hơn những hàm khác. Hàm đệ quy tự gọi chính nó.

Do cách thức đặc biệt này của đệ quy nên xảy ra rất nhiều vấn đề xung quanh đệ quy. Vấn đề đầu tiên mà mọi người nghĩ tới có lẽ sẽ là làm sao để hàm đệ quy này không gọi lại nó nữa. Chúng ta gọi việc này là điều kiện chấm dứt đệ quy.

Như chúng ta đã bàn bạc bên trên, đệ quy có tính chất cũng khá giống vòng lặp, và có thể sử dụng được ở mọi nơi mà vòng lặp có thể sử dụng. Đôi khi cách sử dụng vòng lặp rõ ràng và ngắn gọn, nhưng cách dùng đệ quy còn rõ ràng, ngắn gọn hơn rất nhiều.

Recursion Revealed – Tính chất "Đảo" của đệ quy

Bây giờ chúng ta hãy đi sâu và xem cách thức mà một hàm đệ quy thực thi và kết quả của hàm đệ quy.

Trong đoạn code bên dưới, chúng ta thấy được hàm main() sẽ gọi một hàm mang tên up_and_down. Trong bài này mình sẽ gọi đây là "lần đầu tiên gọi hàm". Tiếp theo đó hàm up_and_down() sẽ gọi lại chính nó với tham số truyền vào là n+1, mình sẽ gọi đây là "lần thứ hai gọi hàm", cứ thế tiếp tục mọi ngươi theo cách đó nhé.

Ở trong ví dụ bên dưới, hàm up_and_down() sẽ tự gọi nó 4 lần, mình gọi mỗi lần là 1 level, cứ thế mà tăng lên. Ở trong ví dụ mình có sử dụng operation &, để hiểu rõ thêm chức năng của operation này, bạn có thể theo dõi ở các bài viết sau. Ở bài viết này mình chỉ nói sơ qua Operation & lấy địa chỉ của biến lưu trữ trong bộ nhớ máy tính. Để sử dụng operation &, trong hàm printf mình phải xuất dạng %p, nếu các bạn không thể sử dụng %p thì có thể dùng %lu hoặc %u để thay thế.

Chương trình recur.c

```c
/* recur.c -- */
#include <stdio.h>
void up_and_down(int);
int main(void)
{
    up_and_down(1);
    return 0;
}
void up_and_down(int n)
{
    printf("Level %d: n location %p\n", n, &n); // 1
    if (n < 4)
        up_and_down(n+1);
```

```
    printf("LEVEL %d: n location %p\n", n, &n); // 2
}
```

Trong ví dụ trên Output sẽ như thế này:

Level 1: n location 0x0012ff48
Level 2: n location 0x0012ff3c
Level 3: n location 0x0012ff30
Level 4: n location 0x0012ff24
LEVEL 4: n location 0x0012ff24
LEVEL 3: n location 0x0012ff30
LEVEL 2: n location 0x0012ff3c
LEVEL 1: n location 0x0012ff48

Bây giờ chúng ta hãy bàn về cách mà đệ quy hoạt động.

Ban đầu, chúng ta xem trong hàm main() có lời gọi hàm up_and_down() với tham số truyền vào là 1 (up_and_down(1)). Vậy nghĩa là n trong hàm up_and_down() sẽ bằng 1.
Như đã nói bên trên, mình sẽ gọi đây là Level 1, để xác nhận gọi hàm thành công, mình sẽ in ra dòng Level 1: n location 0x0012ff48 nhờ câu lệnh: `printf("Level %d: n location %p\n", n, &n);`. Tiếp tới máy sẽ tiếp tục chạy câu lệnh tiếp theo là câu lệnh `if (n < 4);` Tại câu lệnh này, máy sẽ xét điều kiện n có nhỏ hơn 4 không, nếu đúng thì sẽ tiếp tục gọi hàm `up_and_down(n+1);`. Tại đây như các bạn đã thấy,
hàm `up_and_down(n);` gọi `up_and_down(n+1);`. hay có thể gọi `up_and_down(2);` do `n = 1, n + 1 =2`. Lần này mình sẽ gọi đây là lần gọi hàm thứ hai (hay Level 2).

Ở Level 2, do cùng cách thức hoạt động nên hàm này sẽ thực thi giống như trên mình đã nói, và sẽ gọi tiếp hàm thứ 3, hàm thứ 4…(gọi là Level 3, Level 4). Trong đoạn code này mình chỉ giới hạn đệ quy gọi hàm 4 lần, vậy tại Level 4 sẽ kết thúc gọi hàm (điều kiện if sai nên không còn lời gọi hàm nữa
)

Vậy là tới đây chúng ta đã giải quyết 1/2 chặn đường của Output rồi. Khi code chạy tới đây thì Output sẽ giống thế này:

Level 1: n location 0x0012ff48
Level 2: n location 0x0012ff3c
Level 3: n location 0x0012ff30
Level 4: n location 0x0012ff24
------ket qua chay toi day-------
LEVEL 4: n location 0x0012ff24
LEVEL 3: n location 0x0012ff30
LEVEL 2: n location 0x0012ff3c
LEVEL 1: n location 0x0012ff48

Vậy tại sao trong OUTPUT của chươn trình lại còn có thêm kết quả bên dưới. Đây là vấn đề của đệ quy, khi chạy tới bên trên thì đệ quy up_and_down() vẫn chưa hết nhiệm vụ của nó. Hãy xem lại đệ quy còn 1 đoạn chương trình bên dưới nữa:

```
if (n < 4)
up_and_down(n+1);
printf("LEVEL %d: n location %p\n", n, &n); // 2
```

Chúng ta quay lại giá trị n tại Level 4. Lúc này n đang giữ giá trị là 4, vậy nghĩa là điều kiện if trong hàm up_and_down() không còn đúng nữa, vì thế chúng ta không còn lời gọi hàm nào nữa ở đây. Chúng ta đều biết chương trình là một tập các lệnh dành cho máy tính, máy tính thực thi các lệnh từ trên xuống dưới và không bỏ xót lệnh nào. Vì thế khi chạy tới Level 4 thì máy tính cũng mới chỉ chạy xong lệnh `up_and_down(n+1)`, và lúc này vẫn còn `printf("LEVEL %d: n location %p\n", n, &n);`.

Khi Level 4 không còn lời gọi hàm, thì nó còn 1 lệnh in, và máy tính sẽ thực hiện lệnh in này. Đó là nguyên nhân tại sao Level 4 lại được in 2 lần, vì trước tiên máy sẽ in `//1` và sau đó máy sẽ in `//2`. Thế là kết thúc Level 4.

Nhưng hãy tưởng tượng, khi bạn sử dụng một hàm bình thường, cái bạn cần là "trả trị", đệ quy cũng thế. Level 1 gọi Level 2, chờ level 2 trả trị về, level 2 gọi level 3 chờ level 3 trả trị về, level 3 gọi level 4 chờ level 4 trả trị về… Cứ tiếp thế. Vậy nghĩa là hàm a gọi hàm b thì phải chờ hàm b thực thi xong và "trả trị" về và hàm a mới có thể tiếp tục thực hiện các câu lệnh tiếp theo trong hàm.

Trở về quá trình hoạt động của hàm up_and_down(), chúng ta nhớ ra rằng các level của đệ quy trước đó chỉ mới thực hiện tới `up_and_down(n+1)` và nó còn 1 dòng lệnh cuối cùng nữa `printf("LEVEL %d: n location %p\n", n, &n);`. Vì thế nên khi đệ quy Level 4 thực thi xong, "quyền kiểm soát" sẽ được giao cho Level 3, level 3 lại thực hiện lệnh in cuối cùng và trả quyền này về level 2, level 2 trả về level 1 và level 1 trả về main() rồi kết thúc chương trình.

Lưu ý rằng mỗi level của đệ quy đều sử dụng biến private n. Thông qua địa chỉ của biến n thì ta có thể gọi n (phần này liên quan đến con trỏ pointer nên mình không đề cập ở đây). Ở mỗi hệ thống thì địa chỉ này có thể khác nhau.

Ohm, Nếu cái lý giải bên trên khó hiểu với bạn, thì hãy thử tưởng tượng; Bạn cài đặt các hàm như ham1(), ham2(), ham3(), ham4() với nội dung sau đây:

```
int ham1(){ ham2(); printf("hello, im in level 1!!!!\n"); return 0;}
int ham2(){ ham3(); printf("hello, im in level 2!!!!\n"); return 0;}
int ham3(){ ham4(); printf("hello, im in level 3!!!!\n"); return 0;}
int ham4(){ printf("hello, im in level 4!!!!\n");return 0;}

int main(){ ham1(); return 0;}
```

Kết quả nhận được:

```
hello, im in level 4!!!
hello, im in level 3!!!
hello, im in level 2!!!
hello, im in level 1!!!
```

Như chúng ta thấy cách thức của các hàm ham1-4 rất giống cách thức hoạt động của đệ quy. Đệ quy thực sự chỉ là cách viết gọn của 4 ham có cùng cách thức hoạt động thôi.

Nguyên tắc cơ bản của đệ quy

Chúng ta đã biết cách thức hoạt động của đê quy, nếu bạn là người mới biết sử dụng đệ quy, bạn sẽ thấy đệ quy khá là khó để sử dụng và cũng khá rắc rối để hiểu. Bây giờ mình sẽ đưa ra một vấn đề chủ chốt của đệ quy.

Đầu tiên, như đã nói bên trên, Đệ quy là hàm tự gọi nó, và nó sẽ cứ thế cho tới khi một điều kiện nào đó thỏa. Như ở ví dụ trên, khi n >= 4 thì sẽ không có lời gọi hàm đệ quy nào nữa. Nhưng tại sao n lại bằng 4? Tất nhiên là do trong mỗi lần đệ quy, chúng ta tăng n lên 1 và cứ thế n sẽ là bằng 4. Như cách mình đã nói bên trên, mỗi hàm "sẽ chờ" hàm được gọi thực thi xong thì hàm đó mới thực thi tiếp. Vậy nghĩa là phải có "một chỗ nào đó" chứa biến số n. Đó là nguyên nhân tại sao biến số n lại có 4 địa chỉ khác nhau trong OUTPUT.

Bây gờ chúng ta hãy xem 4 biến số đó như thế nào.

```
        variables:                |    n     n     n     n
after level 1 call                |    1
after level 2 call                |    1    2
after level 3 call                |    1    2    3
after level 4 call                |    1    2    3    4
 after return from level 4        |    1    2    3
 after return from level 3        |    1    2
 after return from level 2        |    1
 after return from level 1        |
```

Cũng khá dễ tưởng tượng đúng không nào? Vậy tại sao 1 biến số n mà có tới 4 giá trị, máy tính lưu trữ ra sao. Mình sẽ bàn tiếp trong bài tới. Bài này chỉ để đệ quy thôi nhé.

Thứ hai, trong mỗi lần gọi hàm thì hàm được gọi sẽ trả trị (chúng ta hay gọi là return). Cứ thế thì level 4 sẽ trả trị về level 3, 3 trả về 2, 2 trả về 1, 1 trả về main(). Chúng ta không thể trả vượt cấp ngay về main() được.

Thứ ba, hàm đệ quy thực hiện theo điều kiện, điều kiện nào đến trước sẽ được thực hiện trước, điều kiện nào đến sau sẽ thực hiện sau, mình đã giải thích bên trên.

Thứ tư, điều kiện được gọi sau khi gọi hàm đệ quy sẽ được thực hiện khi hàm đệ quy đó nhận được trị trả về của hàm đệ quy nó gọi. Ví dụ, điều kiện in ở #2 được thực hiện sau khi quá trình gọi hàm đệ quy kết thúc và được thực thi thông qua các lệnh: Level 4, Level 3, Level2, Level1. Chức năng này của đệ quy thực sự hữu dụng trong các chương trình phải xử lý các tiến trình liên quan đến revelsals.

Thứ năm, mặc dầu mỗi Level của đệ quy có biến của riêng nó, nhưng code thì lại được dùng chung. Code là một chuỗi các hướng dẫn cho máy tính, và lời gọi hàm đơn giản di chuyển đến điểm bắt đầu của chuỗi các hướng dẫn đó. Ngoài việc tạo ra biến riếng cho

mỗi lần gọi hàm, đệ quy giống như vòng lặp. Thực tế, đôi khi đệ quy có thể được dùng thay cho vòng lặp và ngược lại.

Cuối cùng, hàm đệ quy phải có một điều kiện để ngăn chặn tiến trình gọi hàm. Thông thường các lập trình viên sử dụng if else. Để làm việc này, mỗi lần gọi đệ quy ta phải đưa vào một tham số khác với tham số trước đó (như ở trên là n và n + 1). Trong ví dụ trên các tham số được đưa vào mỗi lần gọi hàm là 1, 2, 3, 4; Bạn có thể xem lại bảng variable bên trên để hiểu rõ thêm.

Đệ quy Đuôi—Tail Recursion

Cách đơn giản để sử dụng một hàm đệ quy là sử dụng nó ngay return statement. Đây được gọi là đệ quy đuôi, có thể gọi là tail recursion hay end recursion. Đệ quy đuôi là một cách đơn giản và nó hoạt động khá giống vòng lặp.

Ở ví dụ dưới đây, mình sẽ làm một bài toán tính giá trị giai thừa của một số; Ví dụ 3! = 1 * 2 * 3

```c
// factor.c
#include <stdio.h>
long fact(int n);
long rfact(int n);
int main(void)
{
    int num;
    printf("This program calculates factorials.\n");
    printf("Enter a value in the range 0-12 (q to quit):\n");
    while (scanf("%d", &num) == 1)
    {
        if (num < 0)
            printf("No negative numbers, please.\n");
        else if (num > 12)
            printf("Keep input under 13.\n");
        else
        {
            printf("loop: %d factorial = %ld\n",
                    num, fact(num));
            printf("recursion: %d factorial = %ld\n",
                    num, rfact(num));
        }
        printf("Enter a value in the range 0-12 (q to quit):\n");
    }
    printf("Bye.\n");
    return 0;
}
long fact(int n)
{
```

```
    long ans;
    // loop-based function
    for (ans = 1; n > 1; n--)
        ans *= n;
    return ans;
}
long rfact(int n) // recursive version
{
    long ans;
    if (n > 0)
        ans= n * rfact(n-1);
    else
        ans = 1;
    return ans;
}
```

Chương trình này của mình chỉ giới hạn nhập số interger có giá trị giao động từ 0 tới 12. Vì giá trị của 12! lên tới nữa tỉ, giá trị này thực sự lớn hơn kiểu long của máy tính, và nếu như bạn muốn sử dụng số lớn hơn 12! thì bạn chắc phải dùng kiểu double hoặc long long.

Dưới đây là kết quả sau khi kết thúc chương trình trên:

```
This program calculates factorials.
Enter a value in the range 0-12 (q to quit):
5
loop: 5 factorial = 120
recursion: 5 factorial = 120
Enter a value in the range 0-12 (q to quit):
10
loop: 10 factorial = 3628800
recursion: 10 factorial = 3628800
Enter a value in the range 0-12 (q to quit):
q
Bye.
```

Ở hàm mình sử dụng vòng lặp, máu tính sẽ khởi tạo một biến tên ans = 1, sau đó nhân với các số nguyên từ 2 tới n (hoặc từ n về 2 - nói chính xác là thế). Về mặt kỹ thuật thì phải từ n về 1, nhưng số nào nhân với 1 cũng sẽ bằng chính nó, việc này trở lên vô nghĩa trong lập trình.

Bây giờ hãy xem cách mà đệ quy thực hiện. Hướng làm khi tính giai thừa của một số n! là n! = n * (n -1)!.

Hãy xem ví dụ sau:

```
3!  = 3 * (3-1)!
    = 3 * 2!
```

Ta xem

```
2!  = 2 * (2 - 1)!
    = 2 * 1!
```

Ta có `1! = 1`

Vậy có đúng là `3! = 3 * 2 * 1.`

Quay trở lại vấn đề, ta thấy được công thức này là cách để chúng ta tiếp cận và sử dụng đệ quy trong bài toán tìm số giai thừa. Bên trên mình đã khai báo một hàm `rfact()`, chúng ta chỉ cần truyền vào 1 tham số n, vậy có nghĩa là chúng ta sẽ sử dụng hàm `rfact` thế này: `n * rfact(n - 1)`

Như các bạn thấy đấy, cả 2 cách dùng vòng lặp hoặc đệ quy đều cho ra kết quả như nhau. Nhưng nhớ rằng `rfact()` không phải là statement cuối cùng trong hàm đệ quy, nó chỉ là statement cuối cùng trong hàm if (n > 0), và nó là tail recursion.

Vậy có một câu hỏi vui đặt ra là thế này: “Nếu sử dụng được cả hai cách bạn sẽ sử dụng cách nào?”. Thông thường, sử dụng vòng lặp là cách được khuyến khích hơn. Vậy tại sao cách này lại được khuyến khích hơn; Chúng ta cùng phân tích bên dưới.

Trở về bảng Variable bên trên, chỉ qua 4 lần gọi hàm, chúng ta cần tốn 4 ô nhớ để chứa 4 biến với 4 giá trị khác nhau. Đệ quy lưu trữ giá trị trong stack, và bộ nhớ stack là có giới hạn. Nếu 4 ô nhớ này chưa đủ thuyết phục bạn, hãy xem tiếp ví dụ mình sẽ nêu bên dưới.

Thứ hai, đệ quy có tốc độ chậm hơn, việc gọi và xử lý, lưu trữ giá trị… của hàm làm tiêu tốn rất nhiều thời gian.

Và bây giờ, bạn đang tự đặt ra câu hỏi là: “Tại sao lại đưa ra ví dụ trên?”. Tất nhiên là ví dụ trên là cách đơn giản nhất để giảng giải như thế nào là “tail recursion”.

Đệ quy và Reversal

Reversal, hay còn gọi là tính đảo ngược. Đây là một vấn đề khá thú vị trong đệ quy. Chúng ta hãy cùng thảo luận.

Mình đưa ra một yêu cầu thế này: “Hãy viết hàm chuyển số thập phân về nhị phân”. Đây là một vấn đề mà ai là lập trình viên đều biết, thực sự rất đơn giản, nhưng qua yêu cầu này chúnng ta có thể thấy được tính đảo ngược này có ích như thế nào.

```c
/* binary.c -- prints integer in binary form */
#include <stdio.h>
void to_binary(unsigned long n);
int main(void)
{
    unsigned long number;
```

```
    printf("Enter an integer (q to quit):\n");
    while (scanf("%lu", &number) == 1)
    {
        printf("Binary equivalent: ");
        to_binary(number);
        putchar('\n');
        printf("Enter an integer (q to quit):\n");
    }
    printf("Done.\n");
    return 0;
}
void to_binary(unsigned long n)
{
    int r; /* recursive function */
    r = n % 2;
    if (n >= 2)
        to_binary(n / 2);
    putchar(r == 0 ? '0' : '1');
    return;
}
```

Ở đây mình đã xây dựng một hàm to_binary(), ở hàm này sẽ xét r = n % 2, hàm sẽ hiển thị một ký tự ‘0’ nếu trị r là 0 và hiển thị ‘1’ nếu giá trị r là 1. Biểu thức r == 0 ? ‘0’ : ‘1’ cung cấp phương thức này, giải nghĩa ra ta có thể hiểu rằng: xét r có bằng 0 không, nếu r = 0 thì in ra 0, nếu r !=0 thì in ra 1. Các phương thức tính toán binary thì mình không bàn tới, vì đây là cơ bản.

Bên dưới đây là kết quả sau khi chạy chương trình:

```
Enter an integer (q to quit):
9
Binary equivalent: 1001
Enter an integer (q to quit):
255
Binary equivalent: 11111111
Enter an integer (q to quit):
1024
Binary equivalent: 10000000000
Enter an integer (q to quit):
q
done.
```

Liệu bạn có thể sử dụng vòng lặp để tính binary? Tất nhiên là được, nhưng bạn cần phải lưu trữ từng giá trị n % 2 trong một nơi nào đó, một mảng chẳng hạn.

Tất nhiên trong hàm đệ quy có nhiều vấn đề rắc rối mình đã đề cập bên trên. Việc không quản lý được số lần gọi đệ quy sẽ làm hao tốn rất nhiều bộ nhớ và tài nguyên máy tính. Trong ví dụ này, nếu con số muốn chuyển qua binary lên tới nữa tỉ như 12! bên trên, thì gọi

đệ quy thực sự là nhức đầu khi mà máy tính phải lưu trữ rất nhiều nhiều giá trị r… Và tất nhiên cách sử dụng mảng để lưu trữ cùng vòng lặp cũng nhức đầu không kém. Quan trọng bạn phải biết cách sử dụng cái nào tốt hơn.

Recursion Pros and Cons

Có lẽ đệ quy không được đánh giá cao chỉ do một lý do duy nhất: Sử dụng quá nhiều tài nguyên máy tính. Tài nguyên của máy tính không phải là vô hạn, vậy nên ta phải biết tiết kiệm. Ngoài ra các thuật toán của đệ quy cũng khó giải thích và bảo trì theo thời gian.

Về vấn đề tài nguyên của máy tính mình đã nhấn mạnh ở hai ví dụ bên trên. Nếu hai ví dụ đó chưa đủ thuyết phục bạn về đệ quy sử dụng nhiều tài nguyên, thì hãy xem tiếp tục ví dụ bên dưới:

Chúng ta bắt đầu đi tính các số Fibonacci, vốn dĩ số này rất nổi tiếng vì tính thực tế của nó; Hiện trên thế giới có rất nhiều nhà thiết kế sử dụng dãy số này cho thiết kế của mình.

Một dãy Fibo bắt đầu bằng 2 số 1, các số tiếp theo sẽ là tổng của 2 số đứng trước, một đoạn ngắn đầu của fibo giống thế này: 1, 1, 2, 3, 5, 8, 13, … Số Fibo không có giới hạn.

Ohm, mình nghĩ bạn cũng đã có lời giải đệ quy rồi. Thế mạnh của đệ quy: đệ quy hỗ trợ định nghĩa đơn giản. Nếu chúng ta đặt tên hàm là Fibonacci, Fibonacci(n) thì trả về (return) 1 nếu n = 1 hoặc n = 2, nếu không hàm sẽ trả về giá trị của tổng Fibbonacci(n-1) + Fibbonacci(n-2).

```
unsigned long Fibonacci(unsigned n)
{
    if (n > 2)
        return Fibonacci(n-1) + Fibonacci(n-2);
    else
        return 1;
}
```

Như bạn thấy đây, Trong đệ quy này tự gọi chính nó tới 2 hàm, việc này là điểm yếu của đệ quy. Ví dụ như bạn cần tìm n = 40, đây là level 1 của đệ quy. tại Level 2 hàm này tự gọi mình 2 lần, nghĩa là cần thêm 2 vùng nhớ nữa để chứa giá trị lưu trữ; Mỗi hàm Fibbonacci(n - 1) và Fibbonacci(n - 1) lại tự gọi lại hàm nữa… Nếu là mình, mình đã thấy choáng váng ở đây vì không thể nào tính được số lần gọi hàm. Số lượng vùng nhớ này là cấp số nhân, sẽ tiếp tục tăng và tăng, tăng tới lúc vùng nhớ không còn đủ để cung cấp cho đệ quy hoạt động, và nó sẽ crash.

Đây là một ví dụ đơn giản, nhưng là ví dụ nhắc nhớ cho lập trình viên cần phải cẩn trọng trong cách sử dụng đệ quy, nếu không đúng cách thì hậu quả thực sự nghiêm trọng.

Tất nhiên đệ quy tự gọi mình 1 lần, đệ quy kép vẫn chưa đủ thuyết phục, bạn có thể tìm tới tam đệ quy hay tứ đệ quy.