ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ
Khoa Công nghệ Thông tin
PHẠM HỒNG THÁI

Bài giảng
NGÔN NGỮ LẬP TRÌNH C/C++

(tiếp theo)

III. ĐỆ QUI

1. Khái niệm đệ qui

Một hàm gọi đến hàm khác là bình thường, nhưng nếu hàm lại gọi đến chính nó thì ta gọi hàm là đệ qui. Khi thực hiện một hàm đệ qui, hàm sẽ phải chạy rất nhiều lần, trong mỗi lần chạy chương trình sẽ tạo nên một tập biến cục bộ mới trên ngăn xếp (các đối, các biến riêng khai báo trong hàm) độc lập với lần chạy trước đó, từ đó dễ gây tràn ngăn xếp. Vì vậy đối với những bài toán có thể giải được bằng phương pháp lặp thì không nên dùng đệ qui. Để minh hoạ ta hãy xét hàm tính n giai thừa. Để tính n! ta có thể dùng phương pháp lặp như sau:

```
main()
{
int n; doule kq = 1;
cout << "n = " ; cin >> n;
for (int i=1; i<=n; i++) kq *= i;
cout << n << "! = " << kq;
}
```

Mặt khác, n! giai thừa cũng được tính thông qua (n-1)! bởi công thức truy hồi
n! = 1 nếu n = 0
n! = (n-1)!n nếu n > 0
do đó ta có thể xây dựng hàm đệ qui tính n! như sau:

```
double gt(int n)
{
```

```
        if (n==0) return 1;
        else return gt(n-1)*n;
}

main()
{
        int n;
        cout << "n = " ; cin >> n;
        cout << gt(n);
}
```

Trong hàm main() giả sử ta nhập 3 cho n, khi đó để thực hiện câu lệnh cout << gt(3) để in 3! đầu tiên chương trình sẽ gọi chạy hàm gt(3). Do 3 ≠ 0 nên hàm gt(3) sẽ trả lại giá trị gt(2)*3, tức lại gọi hàm gt với tham đối thực sự ở bước này là n = 2. Tương tự gt(2) = gt(1)*2 và gt(1) = gt(0)*1. Khi thực hiện gt(0) ta có đối n = 0 nên hàm trả lại giá trị 1, từ đó gt(1) = 1*1 = 1 và suy ngược trở lại ta có gt(2) = gt(1)*2 = 1*2 = 2, gt(3) = gt(2)*3 = 2*3 = 6, chương trình in ra kết quả 6.

Từ ví dụ trên ta thấy hàm đệ qui có đặc điểm:

- Chương trình viết rất gọn,
- Việc thực hiện gọi đi gọi lại hàm rất nhiều lần phụ thuộc vào độ lớn của đầu vào. Chẳng hạn trong ví dụ trên hàm được gọi n lần, mỗi lần như vậy chương trình sẽ mất thời gian để lưu giữ các thông tin của hàm gọi trước khi chuyển điều khiển đến thực hiện hàm được gọi. Mặt khác các thông tin này được lưu trữ nhiều lần trong ngăn xếp sẽ dẫn đến tràn ngăn xếp nếu n lớn.

Tuy nhiên, đệ qui là cách viết rất gọn, dễ viết và đọc chương trình, mặt khác có nhiều bài toán hầu như tìm một thuật toán lặp cho nó là rất khó trong khi viết theo thuật toán đệ qui thì lại rất dễ dàng.

2. Lớp các bài toán giải được bằng đệ qui

Phương pháp đệ qui thường được dùng để giải các bài toán có đặc điểm:

- Giải quyết được dễ dàng trong các trường hợp riêng gọi là trường hợp *suy biến* hay *cơ sở*, trong trường hợp này hàm được tính bình thường mà không cần gọi lại chính nó,
- Đối với trường hợp *tổng quát*, bài toán có thể giải được bằng bài toán cùng dạng nhưng với tham đối khác có kích thước nhỏ hơn tham đối ban đầu. Và sau một số bước hữu hạn biến đổi cùng dạng, bài toán đưa được về trường hợp

suy biến.

Như vậy trong trường hợp tính n! nếu n = 0 hàm cho ngay giá trị 1 mà không cần phải gọi lại chính nó, đây chính là trường hợp suy biến. Trường hợp n > 0 hàm sẽ gọi lại chính nó nhưng với n giảm 1 đơn vị. Việc gọi này được lặp lại cho đến khi n = 0.

Một lớp rất rộng của bài toán dạng này là các bài toán có thể định nghĩa được dưới dạng đệ qui như các bài toán lặp với số bước hữu hạn biết trước, các bài toán UCLN, tháp Hà Nội, ...

3. Cấu trúc chung của hàm đệ qui

Dạng thức chung của một chương trình đệ qui thường như sau:

```
if (trường hợp suy biến)
{
    trình bày cách giải          // giả định đã có cách giải
}
else                             // trường hợp tổng quát
{
    gọi lại hàm với tham đối "bé" hơn
}
```

4. Các ví dụ

Ví dụ 1 : Tìm UCLN của 2 số a, b. Bài toán có thể được định nghĩa dưới dạng đệ qui như sau:

- nếu a = b thì UCLN = a
- nếu a > b thì UCLN(a, b) = UCLN(a-b, b)
- nếu a < b thì UCLN(a, b) = UCLN(a, b-a)

Từ đó ta có chương trình đệ qui để tính UCLN của a và b như sau.

```
int UCLN(int a, int b)                    // qui uoc a, b > 0
{
    if (a < b) UCLN(a, b-a);
    if (a == b) return a;
    if (a > b) UCLN(a-b, b);
}
```

Ví dụ 2 : Tính số hạng thứ n của dãy Fibonaci là dãy f(n) được định nghĩa:

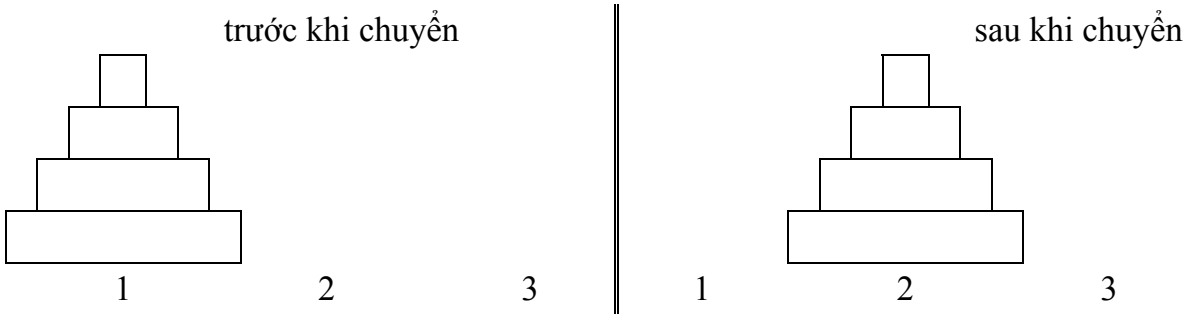
- f(0) = f(1) = 1
- f(n) = f(n-1) + f(n-2) với $\forall n \geq 2$.

```
long Fib(int n)
{
  long kq;
  if (n==0 || n==1) kq = 1; else kq = Fib(n-1) + Fib(n-2);
  return kq;
}
```

Ví dụ 3 : Chuyển tháp là bài toán cổ nổi tiếng, nội dung như sau: Cho một tháp n tầng, đang xếp tại vị trí 1. Yêu cầu bài toán là hãy chuyển toàn bộ tháp sang vị trí 2 (cho phép sử dụng vị trí trung gian 3) theo các điều kiện sau đây

- mỗi lần chỉ được chuyển một tầng trên cùng của tháp,
- tại bất kỳ thời điểm tại cả 3 vị trí các tầng tháp lớn hơn phải nằm dưới các tầng tháp nhỏ hơn.

Bài toán chuyển tháp được minh hoạ bởi hình vẽ dưới đây.



Bài toán có thể được đặt ra tổng quát hơn như sau: chuyển tháp từ vị trí di đến vị trí den, trong đó di, den là các tham số có thể lấy giá trị là 1, 2, 3 thể hiện cho 3 vị trí. Đối với 2 vị trí di và den, dễ thấy vị trí trung gian (vị trí còn lại) sẽ là vị trí 6-di-den (vì di+den+tg = 1+2+3 = 6). Từ đó để chuyển tháp từ vị trí di đến vị trí den, ta có thể xây dựng một cách chuyển đệ qui như sau:

- chuyển 1 tầng từ di sang tg,
- chuyển n-1 tầng còn lại từ di sang den,
- chuyển trả tầng tại vị trí tg về lại vị trí den

hiển nhiên nếu số tầng là 1 thì ta chỉ phải thực hiện một phép chuyển từ di sang den.

Mỗi lần chuyển 1 tầng từ vị trí i đến j ta kí hiệu i → j. Chương trình sẽ nhập vào input là số tầng và in ra các bước chuyển theo kí hiệu trên.

Từ đó ta có thể xây dựng hàm đệ qui sau đây ;

```
void chuyen(int n, int di, int den)              // n: số tầng, di, den: vị trí đi, đến
{
    if (n==1) cout << di << " → " << den << endl;
    else {
        cout << di << "→" << 6-di-den << endl; // 1 tầng từ di qua trung gian
        chuyen(n-1, di, den) ;                    // n-1 tầng từ di qua den
        cout << 6-di-den << "→" den << endl;      // 1 tầng từ tg về lại den
    }
}

main()
{
    int sotang ;
    cout << "Số tầng = " ; cin >> sotang;
    chuyen(sotang, 1, 2);
}
```

Ví dụ nếu số tầng bằng 3 thì chương trình in ra kết quả là dãy các phép chuyển sau đây:

$$1 \to 2\ ,\ 1 \to 3\ ,\ 2 \to 3\ ,\ 1 \to 2\ ,\ 3 \to 1\ ,\ 3 \to 2\ ,\ 1 \to 2.$$

có thể tính được số lần chuyển là $2^n - 1$ với n là số tầng.

IV. TỔ CHỨC CHƯƠNG TRÌNH

1. Các loại biến và phạm vi

a. Biến cục bộ

Là các biến được khai báo trong thân của hàm và chỉ có tác dụng trong hàm này, kể cả các biến khai báo trong hàm main() cũng chỉ có tác dụng riêng trong hàm main(). Từ đó, tên biến trong các hàm là được phép trùng nhau. Các biến của hàm nào sẽ chỉ

tồn tại trong thời gian hàm đó hoạt động. Khi bắt đầu hoạt động các biến này được tự động sinh ra và đến khi hàm kết thúc các biến này sẽ mất đi. Tóm lại, một hàm được xem như một đơn vị độc lập, khép kín.

Tham đối của các hàm cũng được xem như biến cục bộ.

Ví dụ 1 : Dưới đây ta nhắc lại một chương trình nhỏ gồm 3 hàm: luỹ thừa, xoá màn hình và main(). Mục đích để minh hoạ biến cục bộ.

```
float luythua(float x, int n)                    // hàm trả giá trị x^n
{
      int i ;
      float kq = 1;
      for (i=1; i<=n; i++) kq *= x;
      return kq;
}

void xmh(int n)                                  // xoá màn hình n lần
{
      int i;
      for (i=1; i<=n; i++) clrscr();
}

main()
{
      float x; int n;
      cout << "Nhập x và n: "; cin >> x >> n;
      xmh(5);                                    // xoá màn hình 5 lần
      cout << luythua(x, n);                     // in x^n
}
```

Qua ví dụ trên ta thấy các biến i, đối n được khai báo trong hai hàm: luythua() và xmh(). kq được khai báo trong luythua và main(), ngoài ra các biến x và n trùng với đối của hàm luythua(). Tuy nhiên, tất cả khai báo trên đều hợp lệ và đều được xem như khác nhau. Có thể giải thích như sau:

– Tất cả các biến trên đều cục bộ trong hàm nó được khai báo.