BÀI 3
KỸ THUẬT LẬP TRÌNH VỚI HÀM

I. Đơn thể và lập trình đơn thể

I.1. Khái niệm và phân loại đơn thể

Khi viết một chương trình, chúng ta có thể triển khai theo hai cách:

Cách 1: Toàn bộ các lệnh của chương trình được viết trong hàm main. Các lệnh được viết theo trình tự để giải quyết bài toán đặt ra.

Cách 2: Chương trình được chia nhỏ thành các đơn vị chương trình tương đối độc lập gọi chung là hàm. Các hàm thực hiện những nhiệm vụ nhất định và được sử dụng trong chương trình thông qua những lời gọi hàm trong hàm main.

Ưu nhược điểm:

- Với cách 1: sẽ thích hợp khi viết những chương trình có kích thước nhỏ. Toàn bộ bài toán, thuật toán được thể hiện trong một đoạn mã tuần tự từ trên xuống. Tuy nhiên, cách này không phù hợp với các chương trình lớn do:

  + Kích thước chương trình cồng kềnh, khó kiểm soát, chỉnh sửa.

  + Các đoạn mã có thể lặp đi lặp lại, không sử dụng lại mã lệnh.

  + Khó khăn trong việc tổ chức làm việc nhóm…

- Với cách 2: Chương trình được chia nhỏ thành các đơn vị (hàm) khắc phục được hai nhược điểm cơ bản trên. Đặc biệt phù hợp với các chương trình có kích thước lớn và phức tạp.

Các hàm: Chương trình được cấu tạo từ các hàm. Mỗi hàm thực thi một nhiệm vụ tương đối độc lập, trong đó có một hàm main đóng vai trò như chương trình chính để sử dụng các hàm khác.

Các hàm (vì được biên dịch độc lập) nên lại có thể tổ chức trong nhiều file khác nhau. Mỗi file chứa 1 hay nhiều hàm thường được gọi là một module (trong C++ ta hay gọi là một thư viện). Như vậy, bằng cách chia nhỏ chương trình thành các hàm, ta có thể lập trình theo module.

Các đặc trưng của hàm:

- **Tên hàm:** do người lập trình tự đặt và có những đặc điểm sau:

+ Tên hàm thường mang tính đại diện cho công việc mà hàm sẽ đảm nhiệm.

+ Tên hàm được đặt theo quy ước đặt tên trong C++ (xem quy ước đặt tên biến).

- **Kiểu giá trị trả về của hàm:** Nếu hàm trả về một giá trị thì giá trị đó được gán vào tên hàm (chính xác hơn là lời gọi hàm sẽ chính là giá trị trả về của hàm) và nó phải thuộc một kiểu dữ liệu nào đó mà ta gọi là kiểu giá trị trả về của hàm. Kiểu giá trị trả về của hàm có thể là các kiểu dữ liệu chuẩn hoặc một kiểu do người lập trình tự định nghĩa.

- **Kiểu và tên các tham số (parameter) của hàm:** Nếu hàm sử dụng các tham số (để chuyển thông tin vào/ ra hàm) thì các tham số phải thuộc một kiểu dữ liệu nào đó. Khi thiết lập một hàm, ta cần chỉ ra danh sách các tham số của hàm (một cách hình thức) và kiểu dữ liệu của mỗi tham số.

- **Thân hàm:** là nội dung chính của hàm, chứa toàn bộ các lệnh của hàm.

Phân loại hàm:

Trong C++, chúng ta có duy nhất một loại "chương trình con", đó là hàm.

Một chương trình trong C++ được cấu tạo từ các hàm, trong đó hàm main là hàm bắt buộc phải có, đóng vai trò như chương trình chính.

Trong C++, tùy theo giá trị trả về (có hay không) mà các hàm được chia làm hai loại:

- **Hàm không có giá trị trả về:** Là hàm chỉ có chức năng thực hiện một công việc nào đó mà ta không quan tâm tới giá trị trả về của hàm.

- **Hàm có giá trị trả về:** Ngoài việc thực hiện một công việc nào đó, ta còn quan tâm tới giá trị thu được sau khi hàm thực thi để dùng trong những đoạn trình tiếp theo.

Tùy theo nguồn gốc của hàm người ta phân ra:

- **Các hàm có sẵn:** Là các hàm chứa trong các thư viện của C++. Chúng thường là các file có phần mở rộng là .h, đã được định nghĩa từ trước. Người lập trình chỉ việc sử dụng thông qua các chỉ thị: #include <Tên thư viện chứa hàm>.

- **Các hàm tự định nghĩa:** Là các hàm do người lập trình tự định nghĩa. Các hàm này cũng có thể được tập hợp lại trong một file .h để dùng như một thư viện có sẵn.

I.2. Định nghĩa và sử dụng hàm

- **Định nghĩa hàm**

Một hàm thường có cấu trúc như sau:

```
<Kiểu trả về>  <Tên hàm> ([kiểu tham số] [tên tham số])
{
        Các lệnh trong thân hàm;
}
```

Trong đó:

- **<Kiểu trả về>:** là kiểu của giá trị trả về của hàm. Nếu hàm không có giá trị trả về, ta dùng kiểu trả về là ***void***. Ngược lại, ta thường sử dụng các kiểu chuẩn như int, float, double, char…

- **<Tên hàm>:** do người dùng tự định nghĩa theo quy ước đặt tên biến.

- **[kiểu tham số] [tên tham số]**: liệt kê danh sách các tham số (parameter) của hàm và kiểu dữ liệu của tham số (nếu có tham số). Nếu hàm có nhiều tham số thì các tham số cách nhau bởi dấu phảy. Một nguyên tắc trong C++ là mỗi tham số đều phải có một kiểu đi kèm trước tên của nó.

Ví dụ 1: Hàm tính n! đơn giản được viết như sau, với một tham số n đồng vai trò đầu vào của hàm:

```
long GT(int n)
{
        long kq=1;
        for (int i=1; i<=n; i++)
                kq *=i;
        return kq;
}
```

- Nếu hàm có giá trị trả về thì cần có câu lệnh return <Giá trị trả về>; để gán giá trị này vào tên hàm. Tuyệt đối không được gán <Tên hàm> = <Giá trị trả về>;. <Giá trị trả về> có thể là một biểu thức, một biến hoặc một hằng.

- Như vậy, riêng hàm **void** (kiểu trả về là void) sẽ không có lệnh return.

Ví dụ 2. Viết hàm giải phương trình bậc nhất với tham số đầu vào là hai hệ số a, b.

```
void PTBN(float a, float b)
{
        if (a==0 && b==0)
                cout<<”Phuong trinh vo so nghiem”;
```

```
        else
            if (a==0 && b!=0)
                cout<<"phuong trinh vo nghiem";
            else
                cout<<"Phuong trinh co nghiem "<<-b/a;
    }
```

- **Sử dụng hàm**

Hàm được sử dụng thông qua lời gọi của nó. Thông thường, chúng được sử dụng trong hàm main để giải quyết bài toán đặt ra. Tuy nhiên, về nguyên tắc một hàm bất kỳ đều có thể gọi tới các hàm khác, miễn là các hàm đó đã được định nghĩa hoặc khai báo trước.

Khi gọi hàm, ta gọi tới tên hàm. Nếu hàm có tham số, ta phải truyền các đối số (argument) phù hợp về kiểu vào vị trí các tham số này. Số lượng đối số truyền vào khi gọi hàm phải bằng số lượng các tham số và theo đúng thứ tự khi ta định nghĩa hàm.

Cách viết một lời gọi hàm như sau:

<Tên hàm> <([danh sách các đối số])>

Như vậy:

- Các đối số phải có kiểu trùng với kiểu của tham số tương ứng.

- Nếu hàm không có tham số thì lời gọi hàm vẫn phải sử dụng dấu () kèm tên hàm: <Tên hàm> ().

Tuy nhiên, vì hàm có 2 loại: có và không có giá trị trả về nên cách sử dụng hai loại hàm này cũng khác nhau.

- Nếu hàm có giá trị trả về thì tên hàm được sử dụng như một biến, tức là ta không thể sử dụng hàm một cách độc lập mà lời gọi hàm có thể được đặt ở vế phải của phép gán, trong biểu thức hoặc kèm với một lệnh khác. Nói chung, ta coi lời gọi hàm như một giá trị (là giá trị trả về của hàm).

- Ngược lại, nếu hàm không có giá trị trả về, tên hàm được sử dụng như một lệnh, tức là lời gọi hàm được viết độc lập, không viết trong phép gán, trong biểu thức hay kèm với một câu lệnh khác.

Ví dụ: Hàm tính n! được viết ở 2 dạng: có và không có giá trị trả về:

```
long GT(int n)
{
        long kq=1;
        for (int i=1; i<=n; i++)
                kq *=i;
        return kq;
}
```

```
void GT(int n)
{
        Long kq=1;
        for (int i=1; i<=n; i++)
                kq *=i;
        cout<< "Ket qua:"<<kq;
}
```

Có thể nhận thấy 2 điểm khác biệt của hai cách viết cho hai hàm cùng chức năng. Tuy nhiên, ta quan tâm tới sự khác nhau trong cách gọi (sử dụng) hai hàm trên.

Ở hàm thứ nhất (bên trái), do là hàm có giá trị trả về nên nó được sử dụng như một biến. Giả sử ta cần tính 5!, vậy ta có thể gọi hàm này theo các cách như bảng sau:

Cách gọi sai	Cách gọi đúng	ý nghĩa
GT(5);	**b = GT(5);**	Tại vế phải của phép gán
	cout<< GT(5);	Dùng kèm với lệnh cout
	b = GT(5) + 1;	Dùng trong biểu thức

Tuy nhiên, ở hàm thứ 2 thì cách sử dụng ngược lại

Cách gọi sai	Cách gọi đúng
b = GT(5);	**GT(5);**
cout<< GT(5);	
b = GT(5) + 1;	

I.3. Tổ chức các hàm

Khi một chương trình có nhiều hàm, ta quan tâm tới việc tổ chức chúng như thế nào cho khoa học. Thông thường có 2 cách tổ chức các hàm:

Cách 1: Các hàm đặt trong cùng một tệp với chương trình chính.

Chương trình ngoài hàm main còn có các hàm khác thì các hàm có thể đặt trước hoặc sau hàm main đều được:

Các hàm đặt trước hàm main:

```
#include…
…
<Hàm 1>
<Hàm 2>
…
int main()
{
        Thân hàm main;
}
```

Các hàm đặt sau hàm main:

```
#include…
…
<Nguyên mẫu của hàm 1>;
<Nguyên mẫu của hàm 2>;
…
void main()
{
Thân hàm main;
}
<Hàm 1>
<Hàm 2>
…
```

Trong đó, <Nguyên mẫu của hàm> chính là dòng đầu tiên của hàm có kèm theo dấu chấm phảy ';'. Nguyên mẫu của hàm có dạng:

<Kiểu trả về> <Tên hàm> ([Kiểu tham số] [Tên tham số]);

Như vậy, nếu hàm được đặt sau hàm main thì cần khai báo nguyên mẫu của hàm trước hàm main để chương trình dịch có thể biết trước sự tồn tại của chúng khi dịch hàm main.

Các hàm luôn đặt rời nhau. Một hàm không bao giờ được phép đặt trong một hàm khác.

Ví dụ 1. Viết chương trình kiểm tra một số nguyên n có phải là số nguyên tố không, nếu n là số nguyên tố, hãy tính n!.

Chương trình được chia làm hai hàm: hàm kiểm tra xem n có phải số nguyên tố không và hàm tính n!. Một hàm main sử dụng hai hàm trên để giải quyết bài toán.

Hai hàm đặt trước hàm main:

```
bool NT(int n)
{
	if (n ==1 | | n ==2)
		return true;
	else
	{
		for (int i=2; i<n; i++)
			if (n%i == 0) return false;
		return true;
	}
}
//================
long GT(int n)
{
	long kq=1;
	for (int i=1; i<=n; i++)
		kq *=i;
	return kq;
}
int main()
{
	int a;
	cout<<"Nhập a"; cin>>a;
	if (!NT(a))
		cout<<"So "<<a<<" Khong phai nguyen to";
	else
	{
		cout<<"So "<<a<<" la so nguyen to";
		cout<<"Giai thua cua "<<a<<" la "<<GT(a);
	}
}
```

Hai hàm đặt sau hàm main:

```
//Khai báo nguyên mẫu của hàm:
bool NT(int n);
long GT(int n);
//hàm main----------------------------
int main()
{
	int a;
```

```
        cout<<"Nhap a";     cin>>a;
        if (!NT(a))
                cout<<"So "<<a<<" Khong phai nguyen to";
        else
        {
                cout<<"So "<<a<<" la so nguyen to";
                cout<<"Giai thua cua "<<a<<" la "<<GT(a);
        }
}

bool NT(int n)
{
        if (n ==1 | | n ==2)
                return true;
        else
        {
                for (int i=2; i<n; i++)
                        if (n%i==0) return false;
                return true;
        }
}
//=================
long GT(int n)
{
        long kq=1;
        for (int i=1; i<=n; i++)
                kq *=i;
        return kq;
}
```

Cách 2: Các hàm đặt trong tệp thư viện (module):

B1: Viết các hàm (trừ hàm main()) trong một file sau đó lưu dưới định dạng .h. File này thường được gọi là file thư viện (hoặc header file). Để thuận tiện cho việc soát lỗi, tốt nhất trước tiên nên tổ chức các hàm như cách 1, sau khi đảm bảo các hàm chạy tốt, di chuyển toàn bộ các hàm (*trừ hàm main()*) sang một file mới và lưu lại với đuôi .h.

B2: Viết hàm main() trong một tệp riêng. Để hàm main() có thể sử dụng các hàm viết trong file thư viện đã tạo trong B1, cần thêm chỉ thị:

#include <[đường dẫn] "Tên thư viện.h">

Chú ý: Để tránh việc đặt đường dẫn tới thư viện, ta nên tạo thư viện ngay trong project. Muốn vậy, kích chuột lên tên project, chọn File/ New/ File..., sau đó chọn C/C++ header/ Go/ Next và đặt tên file kèm đường dẫn đầy đủ trong ô File name with full path. Hãy kích chuột vào dấu '...' bên cạnh của ô và đặt tên file (có thể chọn đường dẫn) thay vì gõ tên file trực tiếp vào ô.

Ví dụ: Tạo file .h với nội dụng sau, (ví dụ file "TV.h"):

```
bool NT(int n)
{
        if (n ==1 | | n ==2)
                return true;
        else
        {
                for (int i=2; i<n; i++)
                        if (n%i==0) return false;

                return true;
        }
}
//=================

long GT(int n)
{
    long kq=1;
    for (int i=1; i<=n; i++)
                kq *=i;
    return kq;
}
```

Giả sử file TV.h này được đặt trong thư mục của project và được thêm vào theo đúng quy cách (xem chú ý ở trên). Ta mở một file mới và viết hàm main() như sau:

```
#include <conio.h>
#include <stdio.h>
#include <iostream>
using namespace std;
#include "TV.h"
int main()
{
```

```
        int a;
        cout<<"Nhap a";     cin>>a;
        if (!NT(a))
                cout<<"So "<<a<<" Khong phai nguyen to";
        else
        {
                cout<<"So "<<a<<" la so nguyen to";
                cout<<"Giai thua cua "<<a<<" la "<<GT(a);
        }
}
```

- Các file thư viện .h không cần phải có các chỉ thị tiền xử lý #include …

- Không thể soát lỗi bằng cách bấm F9 trong file thư viện .h.

Chú ý: Đặt đường dẫn tới thư viện trong CodeBlock

Đôi khi ta cần đặt đường dẫn tới một thư mục chứa các thư viện trong môi trường CodeBlock (thay vì phải viết trong chỉ thị tiền xử lý #include). Muốn vậy hãy làm như sau:

Chọn menu Settings/ Compiler.... Chọn Search Directories, chọn Add và sau đó chọn đường dẫn (là thư mục có chứa file thư viện).

I. 4. Phạm vi hoạt động của biến

Theo phạm vi hoạt động của biến, ta chia ra:

- **Biến toàn cục:** (global) là các biến có phạm vi hoạt động trong toàn bộ chương trình, kể từ vị trí khai báo biến.

Biến toàn cục có vị trí khai báo nằm ngoài các hàm (kể cả hàm main). Thông thường nó được khai báo ngay từ những dòng đầu tiên của chương trình (sau các chỉ thị tiền xử lý).

Nếu chương trình được viết trên nhiều tệp, để phạm vi hoạt động của biến bao gồm cả các tệp khác, ta cần thêm chỉ danh extern vào trước khai báo biến toàn cục. Trong trường hợp này, từ khoá extern còn được đặt trước các nguyên mẫu của hàm với ý nghĩa tương tự.

- **Biến cục bộ:** (local) là các biến được khai báo trong thân một hàm, thậm trí trong một khối lệnh nào đó của thân hàm.

Phạm vi hoạt động: Nếu biến được khai báo trong thân một khối nào đó sẽ có phạm vi hoạt động chỉ trong khối, kể cả các khối con nằm bên trong khối đó, kể từ vị trí khai báo. Kết thúc khối, biến cục bộ sẽ được giải phóng.

Muốn biến này tồn tại trong suốt thời gian chương trình làm việc, ta cần thêm từ khóa static trước khai báo biến để khai báo biến dưới dạng biến tĩnh.

Ví dụ: Chương trình sau sử dụng ba biến x, trong đó có 1 biến toàn cục và hai biến cục bộ (một biến x cục bộ được khai báo trong hàm main, mang giá trị là 3, và một biến x cục bộ được khai báo trong hàm Ham, mang giá trị 7).

```
int x;
void Ham(int a)
{
	cout<<"Bien x gloal: "<<x<<endl;
	if (a%2==0)
	{
		int x=5;
		x+=a;
		cout<<"Bien x local trong Ham: "<<x<<endl;
	}
}
int main()
{
	x=1;
	int a = 2;
	Ham(a);
	cout<< "Bien x global: "<<x<<endl;
	int x = 3;
	cout<<"Bien x local trong main: "<<x<<endl;
}
```

- Biến x dưới dạng toàn cục có phạm vi hoạt động trong toàn bộ chương trình, kể từ khi khai báo.

- Nếu trong một khối có khai báo biến cục bộ trùng tên với biến toàn cục thì kể từ khi khai báo, khối đó sẽ sử dụng biến cục bộ mà không sử dụng biến toàn cục.

II. Kỹ thuật đệ quy

II.1. Khái niệm về đệ quy

Trong C++, một hàm có thể gọi đến chính nó, tính chất này của hàm gọi là tính đệ quy. Khi một hàm thể hiện tính đệ quy, ta gọi hàm đó là hàm đệ quy.

Ví dụ: Xét hàm tính n!. Ngoài cách viết sử dụng vòng lặp như trên, ta có thể có cách tiếp cận khác để giải quyết bài toán:

Ta định nghĩa: **n! = (n-1)! * n.** Với định nghĩa này, giả sử n=5 thì n! được tính như sau:

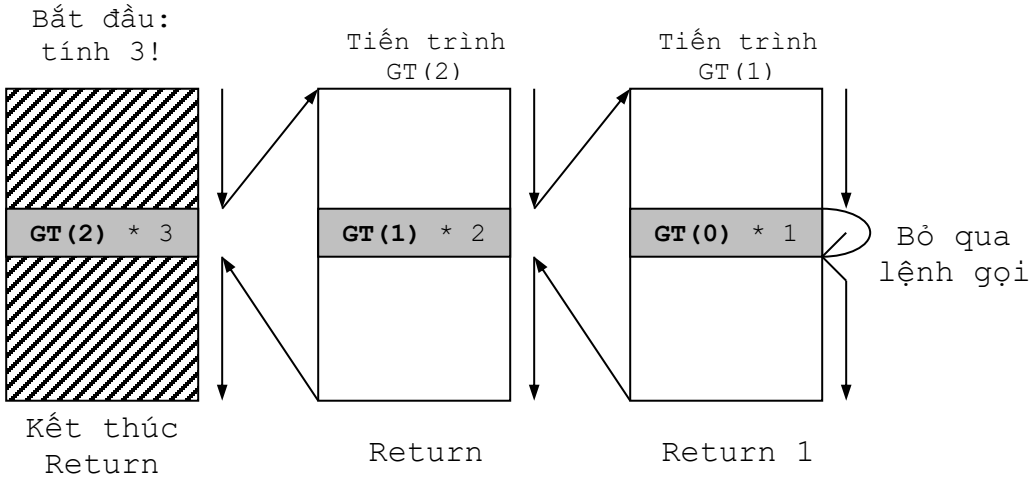
5! = 4! * 5
= 3! * 4 * 5.
= 2! * 3 * 4 * 5.
= 1! * 2 * 3 * 4 * 5.

Với 1! = 1, ta hoàn toàn có thể tính được 5! bằng cách tính các giá trị 2!, 3!, 4! để rồi thay vào công thức 5!= 4! *5.

Một cách tổng quát, nếu ta có hàm GT(n) để tính n! thì GT(n) = GT(n-1) * n. Đây chính là công thức thể hiện tính đệ quy. Từ công thức này, hàm đệ quy tính n! có thể viết như sau:

```
long GT(int n)
{
        if (n==1)
                return 1;
        else
                return GT(n-1) * n;
}
```

Để hiểu bản chất của đệ quy, ta xét quá trình tính 3!. Quá trình này được thể hiện qua sơ đồ sau:



Khi gặp lời gọi GT(3) để tính 3!, máy tính tạo ra một tiến trình (process) với đầu vào là 3. Tuy nhiên, khi thực hiện, để tính 3!, tiến trình này gặp phải lời gọi đệ quy 3*GT(2).

Khi đó, toàn bộ tiến trình này phải dừng lại và chờ để máy tính tạo ra một tiến trình mới (quá trình thực thi hàm mới) với đối vào là 2. Khi tiến trình mới này thực hiện xong

(tức là tính xong 2!) nó sẽ quay về tiến trình ban đầu với kết quả 2! tính được và tiếp tục thực thi tiến trình ban đầu với kết quả là 2!*3.

Tuy nhiên, tiến trình tính 2! lại gặp lời gọi đệ quy tính 1! Nên nó cũng phải tạm dừng và chờ 1 tiến trình thứ 3 được tạo ra để tính 1!. Rất may là trong tiến trình tính 1! không có lời gọi đệ quy (vì if(n==1) return 1;) nên quá trình dừng-chờ-tạo tiến trình mới không xảy ra. Do vậy các tiến trình đang chờ trước đó lần lượt được khôi phục và trả về kết quả mong muốn.

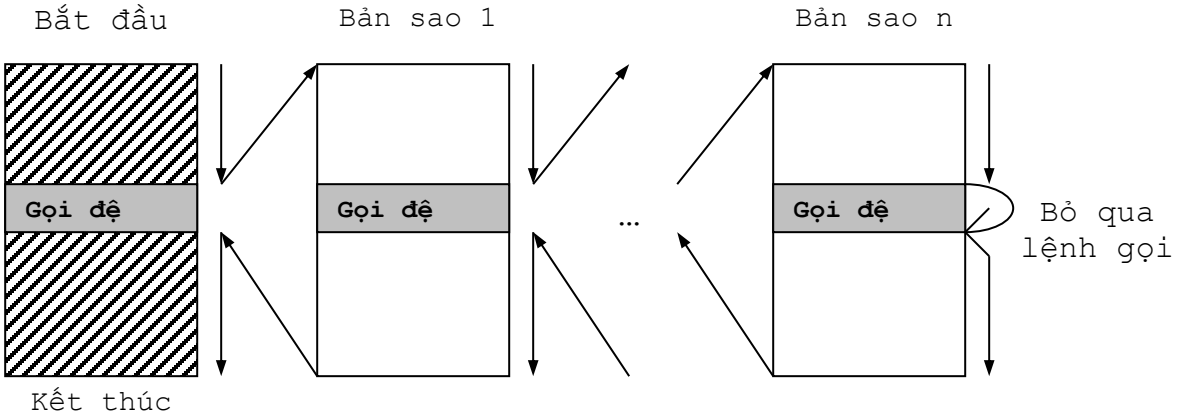
Như vậy, khi gặp một lời gọi đệ quy, máy tính sẽ:

- Tạm dừng tiến trình hiện tại, lưu địa chỉ của dòng lệnh gọi đệ quy vào ngăn xếp.

- Tạo một tiến trình hoàn toàn mới, cấp phát các vùng nhớ mới cho các biến cục bộ, thực hiện tiến trình mới này.

- Khi việc thực thi tiến trình mới hoàn thành, chương trình quay về ngăn xếp, lấy địa chỉ của dòng lệnh gọi đệ quy và quay về tiến trình ban đầu.

Một cách tổng quát ta có sơ đồ quá trình thực thi hàm đệ quy như sau:



Nói chung, một hàm được viết theo kiểu đệ quy sẽ tốn bộ nhớ hơn, thực thi phức tạp hơn và do vậy người ta thường tìm cách khử đệ quy (tức viết chương trình không theo kiểu đệ quy).

Tuy nhiên, cách tiếp cận đệ quy lại tỏ ra rất có hiệu quả với các bài toán liên quan tới duyệt cây, đồ thị, danh sách tuyến tính .v.v…

II.2. Thiết kế hàm đệ quy.

Các bài toán áp dụng giải thuật đệ quy thường có đặc điểm sau:

- Bài toán dễ dàng giải quyết trong một số trường hợp riêng ứng với các giá trị đặc biệt của tham số. Trong trường hợp này, ta có thể giải quyết bài toán mà không cần gọi đệ quy. Ta gọi trường hợp này là trường hợp **suy biến**.

- Trong trường hợp **tổng quát**, bài toán có thể quy về bài toán cùng dạng nhưng giá trị của tham số thay đổi. Và sau một số hữu hạn bước biến đổi đệ quy, sẽ dẫn tới trường hợp suy biến.

Trường hợp suy biến rất quan trọng trong bài toán đệ quy. Nếu không có trường hợp này, quá trình tạo tiến trình mới sẽ không thể dừng lại và ta gặp phải trường hợp đệ quy vô hạn. Nếu trường hợp tổng quát mà sau một số hữu hạn lần gọi đệ quy không thể quy về trường hợp suy biến thì cũng không thể thoát khỏi quá trình gọi đệ quy vô hạn này.

Giả sử bài toán tính n!, dễ dàng thấy:

- Với n = 0 hoặc n = 1 thì n! = 1. Khi đó ta không cần gọi đệ quy vẫn có thể tính được n!. Đây là trường hợp suy biến.

- Trường hợp tổng quát, n! = n* (n-1)!. Tức là để tính n!, ta có thể quy về bài toán tính (n-1)!. Sau một số hữu hạn bước biến đổi, ta có thể quy về bài toán tính 1!.

Như vậy:

- Trường hợp suy biến: n=0 hoặc n=1. Công thức trong trường hợp này : n! = 1;
- Trường hợp tổng quát: là các trường hợp còn lại (n >1) khi đó: n! = n* (n-1)!.

Từ phát hiện trên, ta có thể tạm chấp nhận phương pháp thiết kế hàm đệ quy theo 3 bước như sau:

Bước 1:

- Xác định trường hợp suy biến, giá trị của tham số, công thức để tính toán trong trường hợp này.
- Xác định trường hợp tổng quát, giá trị của tham số, công thức để tính toán trong trường hợp này.

Bước 2: Viết nội dung đệ quy dạng:

```
if  (suy biến)
        return <công thức suy biến>;
else
        return <công thức tổng quát>;
```

Bước 3: Hoàn thiện hàm đệ quy.

Ví dụ 1: thiết kế hàm đệ quy tính n! với n nguyên dương.

Bước 1:

- Suy biến : n = 1; Công thức trong trường hợp suy biên: n! = 1;
- Tổng quát: n > 1; Công thức n! = n* (n-1)!.

Bước 2:

```
if  (n= = 1)
    return 1;
else
    return n * GT(n-1);
```

Bước 3: Hoàn thiện để thu được hàm đệ quy tính n!.

Ví dụ 2: Dãy số Catalan được phát biểu đệ quy như sau:

$C_1 = C_2 = 1;$

$C_n = \sum_{i=1}^{n-1} C_i C_{n-i}$ với mọi n > 2.

Hãy xây dựng hàm đệ quy tìm số CataLan thứ n.

Hàm đệ quy được thiết kế như sau:

Bước 1:

- Suy biến: n = 1 hoặc n = 2; Công thức suy biến: $C_n = 1$;
- Không suy biến: n > 2; công thức tổng quát: $C_n = \sum_{i=1}^{n-1} C_i C_{n-i}$

Bước 2:

```
if (n = = 1 || n == 2)
        return 1;
else
        {       int C =0;
                for (int i=1; i<n; i++)
```

```
                C+= CataLan(i) * CataLan(n-i);
            return C;
        }
```

Với phần thiết kế trên, hàm đệ quy tính số CataLan thứ n (n nguyên dương) được viết như sau (bước 3):

```
int CataLan(int n)
{
    if (n==1 || n == 2)
        return 1;
    else
    {
        int C=0;
        for (int i=1; i<n; i++)
            C+=CataLan(i)*CataLan(n-i);
        return C;
    }
}
```

II.3. Đệ quy và các dãy truy hồi

Khái niệm: Một dãy truy hồi là dãy mà các số hạng đứng sau được định nghĩa dựa trên các số hạng đứng trước của dãy.

Ví dụ: cho dãy sau:

a[1] = 1;

a[n] = a[n-1] * n;

Dễ dàng thấy đây chính là dãy các giai thừa của các số tự nhiên: 1!, 2!, 3!, 4!…Với số hạng thứ n được định nghĩa từ số hạng thứ n-1.

Hoặc dãy các số Fibonacci:

F[1] = 1; F[2] = 1;

F[n] = F[n-1] + F[n-2] (với n>2).

Đặc điểm của các dãy truy hồi:

- Luôn tồn tại một hoặc một số số hạng đứng đầu dãy mà các số hạng này dễ dàng được xác định hoặc được cho trước. Đây chính là trường hợp suy biến cho bài toán đệ quy tìm số hạng thứ n.

- Luôn tồn tại một công thức truy hồi mà số hạng sau được xác định dựa vào số hạng đứng trước. Vậy theo công thức truy hồi ta luôn xác định được số hạng thứ n từ các

số hạng đầu dãy (sau một số hữu hạn lần truy hồi). Vậy công thức truy hồi chính là công thức trong trường hợp tổng quát cho bài toán đệ quy tìm số hạng thứ n.

Do vậy, với các dãy truy hồi, người ta thường sử dụng các giải thuật đệ quy để xác định chúng.

II.4. Một số ví dụ về đệ quy

Ví dụ 1: USCLN của hai số nguyên a, b được định nghĩa như sau:

USCLN(a, b) = a nếu b =0

= USCLN(b, a%b) nếu b khác 0.

Viết hàm lặp và đệ quy để tính USCLN của hai số nguyên a, b.

Hàm lặp:

```
int USCLN_Lap(int a, int b)
{
        int Sodu;
        while (y !=0)
                {
                        Sodu = a%b;
                        a = b;
                        b = Sodu;
                }
        return a;
}
```

Hàm đệ quy:

Bước 1:

- Suy biến : b=0; công thức suy biến: USCLN(a, b) = b;
- Không suy biến: b khác 0; công thức tổng quát:

USCLN(a, b) = USCLN(b, a%b);

Bước 2:

```
if(b==0)
return a;
else
return USCLN(b, a%b)
```

Bước 3:

```
int USCLN_DQ(int a, int b)
{
if (b==0)
```

```
        return a;
    else
        return USCLN_DQ(b, a%b);
    }
```

Ví dụ 2: Các số Fibonacci được định nghĩa đệ quy như sau:

F[0] = F[1] = 1;

F[i] = F[i-1] + F[i-2];

VD: 1, 1, 2, 3, 5, 8, 13… Viết hàm đệ quy tìm số Fibonacci thứ n.

Bước 1:

Suy biến: n <=1; công thức suy biến: Fibo(n) = 1;

Không suy biến: n >1; công thức tổng quát: Fibo(n) = Fibo(n-1) + Fibo(n-2).

Bước 2:

```
if(n<=1)
    return 1;
else
    return Fibo(n-1) + Fibo(n-2);
```

Bước 3:

```
int Fibo(int n)
{
    if(n<=1)
        return 1;
    else
        return Fibo(n-1) + Fibo(n-2);
}
```

III. Kỹ thuật truyền tham số

III.1. Khái niệm và phân loại tham số

Khi định nghĩa hàm, để truyền thông tin từ bên ngoài vào hàm hoặc ngược lại, ta sử dụng các tham số (**parameter**). Tham số được định nghĩa một cách hình thức.

Nếu hàm có tham số, khi gọi hàm ta phải truyền các giá trị thực sự cho tham số này (nếu tham số là đầu vào của hàm) hoặc truyền một biến thực sự (nếu tham số là đầu ra). Các giá trị được truyền vào hàm, lấy ra khỏi hàm thông qua tham số được gọi là các đối số (**argument**). Các đối số là các hằng hoặc biến cụ thể và tương ứng về kiểu với các tham số của hàm.

Tham số được chia làm hai loại:

[1]. Biến tham trị: Khi định nghĩa hàm, nếu tham số là các biến thông thường (ví dụ int a, float b, ...) thì chúng được gọi là các tham trị (hay biến). Nếu hàm có tham trị, quá trình truyền tham số gọi là truyền tham trị.

[2]. Biến tham chiếu: Nếu tham số được định nghĩa là các biến tham chiếu (ví dụ int & a, float & b, ...) thì khi sử dụng hàm ta truyền tham số dưới dạng tham chiếu.

Tham số là biến tham chiếu được khai báo như sau:

<kiểu> & <Tên>

Biến tham chiếu thực chất là tên gọi khác của biến được gán cho nó. Ví dụ ta có biến thực b mang giá trị 5. Biến a là một biến tham chiếu và ta gán b vào a. Khi đó, a và b là hai tên gọi khác nhau của cùng một ô nhớ (là ô nhớ a ban đầu):

```
float b = 5;
float &a = b;
cout<<b<<" va "<<a;
```

Kết quả sẽ in ra màn hình là: “5 va 5”

Như vậy, việc truyền tham số cho hàm dưới dạng tham chiếu hay tham trị là đã được xác định từ lúc định nghĩa hàm. Vậy hai cách truyền tham số này có gì khác nhau:

Khi truyền tham số dưới dạng tham trị, đối số truyền vào sẽ không được hàm truy cập trực tiếp. Hàm sẽ cấp phát một vùng nhớ mới và sao chép giá trị của đối số vào đó. Các lệnh trong thân hàm sẽ thao tác trên vùng nhớ mới này. Như vậy, một đối số khi truyền vào một hàm sẽ không bị thay đổi giá trị bởi các lệnh trong thân hàm.

Ngược lại, khi truyền tham số dưới dạng tham chiếu, vì đối số được gán cho biến tham chiếu và lúc này, biến tham chiếu chỉ là tên gọi khác của đối số nên hàm sẽ tham chiếu tới ô nhớ của đối số. Như vậy, các một đối số khi truyền vào một hàm có thể bị biến đổi giá trị của nó bởi các lệnh trong thân hàm.

Ví dụ 1. Xét hàm sau:

```
int tang(int a)
{
        a++;
}
int main()
{
```

```
        int n=1;
        cout<<"Giá trị trước khi gọi hàm "<<n;
        tang(n);
        cout<<"Giá trị sau khi gọi hàm "<<n;
}
```

Hàm tang(int a) có một tham số dưới dạng biến tham trị a. Do vậy khi sử dụng hàm, đối số truyền vào hàm luôn dưới dạng tham trị.

Biến n là một biến thông thường và đang mang một giá trị cụ thể (n = 1). Khi nó được truyền vào hàm **tang**(int a), nó trở thành một sối số. Tham số a nhận giá trị của đối số n. Hàm tang() sau đó thao tác trên a mà không hề thao tác trên n. Do vậy, khi kết thúc hàm, giá trị của đối số n không hề thay đổi (vẫn là 1) mặc dù trong thân hàm int tang(int a) thì giá trị của tham số bị thay đổi (a++).

Ví dụ 2. Xét hàm sau

```
int Ham(int a, int b)
{
        a+=1;
        b+=a;
        cout<< "Gia tri a trong than ham: "<<a<<endl;
        cout<< "Gia tri b trong than ham: "<<b<<endl;
}
int main()
{
        int a = 1, b = 2;
        cout<< "Gia tri a truoc khi goi ham: "<<a<<endl;
        cout<< "Gia tri b truoc khi goi ham: "<<b<<endl;
        Ham(a, b);
        cout<< "Gia tri a sau khi goi ham: "<<a<<endl;
        cout<< "Gia tri b sau khi goi ham: "<<b<<endl;
}
```

Cần nhớ hai biến a, b của hàm **Ham(int a, int b)** là các tham số, trong khi các biến a, b trong hàm main là các đối số. Chúng chỉ vô tình trùng tên nhau.

Biến a trong hàm **Ham()** và biến a trong hàm **main** sẽ là hai ô nhớ khác nhau. Tương tự cho biến b. Do vậy, mặc dù trong thân hàm **Ham()** ta thay đổi giá trị của a, b nhưng hai biến a, b trong hàm **main** không bị ảnh hưởng gì.

Nhưng nếu khi định nghĩa hàm Ham(), ta định nghĩa a, b là hai biến tham chiếu như sau:

```
int Ham(int &a, int &b)
{
        a+=1;
        b+=a;
        cout<< "Gia tri a trong than ham: "<<a<<endl;
        cout<< "Gia tri b trong than ham: "<<b<<endl;
}
int main()
{
        int a = 1, b = 2;
        cout<< "Gia tri a truoc khi goi ham: "<<a<<endl;
        cout<< "Gia tri b truoc khi goi ham: "<<b<<endl;
        Ham(a, b);
        cout<< "Gia tri a sau khi goi ham: "<<a<<endl;
        cout<< "Gia tri b sau khi goi ham: "<<b<<endl;
}
```

thì tham số a trong hàm **Ham()** chỉ là tên gọi khác của biến a trong hàm **main**() mà thôi (tương tự với b). Chúng đều tham chiếu tới cùng một ô nhớ, là ô nhớ của biến a trong hàm **main()**. Do vậy, các lệnh trong thân hàm **Ham()** sẽ thao tác trực tiếp trên biến a, b trong hàm main khi mà nó được truyền vào hàm Ham(). Kết quả là các biến a, b trong hàm main đã bị thay đổi giá trị bởi các lệnh trong thân hàm **Ham().**