# Bài 12 - HÀM VÀ CẤU TRÚC CHƯƠNG TRÌNH

## Nội dung bài học



## *I. Tổ chức chương trình*

Mỗi chương trình như đã nêu ra ở các ví dụ trong các chương trước đây thường khá ngắn; do đó:

- Thường không khó để hiểu;
- Dễ nhớ toàn bộ nội dung chương trình cũng như
- Hiểu trình tự logic các bước của công việc.

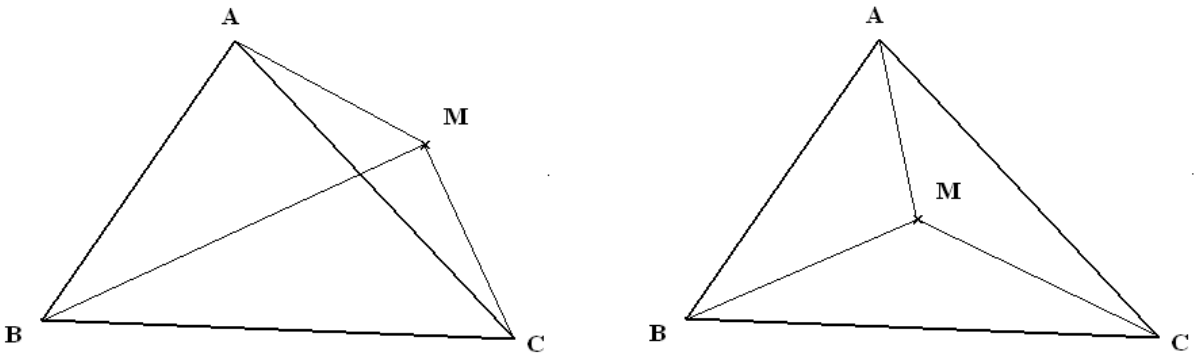
Tuy nhiên khi giải quyết các bài toán thực tế thì văn bản chương trình thường dài hơn rất nhiều, khi đó:

- Việc quản lý trình tự logic các công việc là tương đối khó khăn.
- Thêm nữa, khi viết chương trình chúng ta thường gặp những đoạn chương trình lặp đi lặp lại nhiều lần ở những chỗ khác nhau với sự khác biệt rất nhỏ hoặc thậm chí giống nhau hoàn toàn.

Để giải quyết vấn đề này, tất cả các ngôn ngữ lập trình đều cho phép người sử dụng tổ chức chương trình thành chương trình chính và các chương trình con dạng thủ tục và hàm.

### 1. Ví dụ

Ví dụ, xét bài toán kiểm tra vị trí tương đối của điểm M trên mặt phẳng so với tam giác ABC là ở trong, nằm trên cạnh hay ngoài tam giác.




Bài toán này có thể giải bằng cách:

- Nếu diện tích tam giác ABC bằng tổng diện tích các tam giác MAB, MBC và MAC thì kết luận là M nằm trong tam giác ABC.
- Ngược lại, khi diện tích tam giác ABC nhỏ hơn tổng diện tích các tam giác MAB, MBC và MAC thì kết luận là M nằm ngoài tam giác ABC.

Nếu theo biện pháp này thì rõ ràng là trong chương trình phải cần ít nhất là bốn lần tính diện tích tam giác. Nếu ta viết một chương trình con tính diện tích tam giác khi biết ba đỉnh U, V, E như DT (U,V,E) chẳng hạn, thì chương trình của chúng ta dường như chỉ còn là một dòng lệnh đơn giản:

```
If  (DT (A,B,C) < DT (M,B,C)+DT(M,C,A)+DT(M,A,B))
    printf("M nam ngoai ABC");
else
    printf("M nam trong ABC");
```

Với ví dụ vừa rồi chúng ta thấy rất rõ một lợi ích của việc sử dụng chương trình con là:

- Làm gọn nhẹ chương trình, thay vì phải viết bốn lần cùng một đoạn chương trình rất giống nhau một cách nhàm chán thì giờ đây ta chỉ cần viết có một lần.
- Ngoài ra nó cho phép người lập trình có thể kiểm soát chương trình của mình một cách dễ dàng và thuận tiện hơn.
- Hiển nhiên là việc phải kiểm tra, tìm lỗi lôgic trong một chương trình có bốn đoạn tính diện tích tam giác so với việc kiểm tra kỹ một đoạn chương trình tính diện tích tam giác cùng với một dòng lệnh rõ ràng và dễ hiểu như trên là rất khác nhau về sự phức tạp.

## 2. Cấu trúc chương trình

Một chương trình hoàn chỉnh trong C/C++ có 6 phần chính (nhưng không bắt buộc) theo thứ tự như sau:

1. Chỉ thị tiền xử ký;
2. Định nghĩa kiểu dữ liệu;
3. Khái báo prototype;
4. Khai báo biến ngoài;
5. Chương trình chính và
6. Cài đặt hàm.

Nội dung cơ bản các phần này được mô tả chi trong các phần sau đây.

### *1. Các chỉ thị tiền xử lý*

Như đã biết trước khi chạy chương trình (bắt đầu từ văn bản chương trình tức chương trình nguồn) C/C++ sẽ dịch chương trình ra tệp mã máy còn gọi là chương trình đích. Thao tác dịch chương trình nói chung gồm có 2 phần:

- Xử lý sơ bộ chương trình, hay còn gọi là ***tiền xử lý*** và
- Dịch.

Phần xử lý sơ bộ được gọi là tiền xử lý, trong đó có các công việc liên quan đến các chỉ thị được đặt ở đầu tệp chương trình nguồn như #include, #define …

**Chỉ thị bao hàm tệp #include**

Cho phép ghép nội dung các tệp đã có khác vào chương trình trước khi dịch. Các tệp cần ghép thêm vào chương trình thường là các tệp chứa khai báo nguyên mẫu của các hằng, biến, hàm … có sẵn trong C hoặc các hàm do lập trình viên tự viết. Có hai dạng viết chỉ thị này:

1. #include <tệp>
2. #include "đường dẫn\tệp"

- Dạng khai báo 1 cho phép trình biên dịch tìm tệp cần ghép tại thư mục định sẵn của công cụ lập trình. Thường thì mọi công cụ lập trình dạng C đều xây dựng sẵn các hàm trong các tệp nguyên mẫu, các tệp này được lưu trong thư mục INCLUDES, và thiết lập thư mục mặc định đến thư mục INCLUDES này.
- Dạng khai báo 2 cho phép tìm tệp theo đường dẫn, nếu không có đường dẫn sẽ tìm trong thư mục hiện tại. Tệp thường là các tệp (thư viện) được tạo bởi lập trình viên và được đặt trong cùng thư mục chứa chương trình. Cú pháp này cho phép lập trình viên chia một chương trình thành nhiều môđun đặt trên một số tệp khác nhau để dễ quản lý.

**Chỉ thị macro #define**

#define tên_macro xaukitu

- Trước khi dịch bộ tiền xử lý sẽ tìm trong chương trình và thay thế bất kỳ vị trí xuất hiện nào của tên_macro bởi xâu kí tự. Ta thường sử dụng macro để định nghĩa các hằng hoặc thay cụm từ này bằng cụm từ khác dễ nhớ hơn.

  Ví dụ:

```
#define then            // thay then bằng dấu cách
#define begin   {       // thay begin bằng dấu          {
#define end     }       // thay end bằng dấu            }
#define MAX 100                 // thay MAX bằng 100
#define TRUE 1                  // thay TRUE bằng 1
```

  Từ đó trong chương trình ta có thể viết những đoạn lệnh như:

```
if (i < MAX) then
begin
    ok = TRUE;
    printf("%d",i) ;
end
```

  Và trước khi dịch bộ tiền xử lý sẽ chuyển đoạn chương trình trên thành:

```
if (i < 100)
{
    ok = 1;
    printf("%d",i);
```

```
}
```

theo đúng cú pháp của C/C++ và rồi mới tiến hành dịch.

- Ngoài việc chỉ thị #define cho phép thay tên_macro bởi một xâu kí tự bất kỳ, nó còn cũng được phép viết dưới dạng có đối.

  Ví dụ, để tìm số lớn nhất của 2 số, thay vì ta phải viết nhiều hàm max (mỗi hàm ứng với một kiểu số khác nhau), bây giờ ta chỉ cần thay chúng bởi một macro có đối đơn giản như sau:

  ```
  #define max(A,B) ((A) > (B) ? (A): (B))
  ```

  Khi đó trong chương trình nếu có dòng x = max(a, b) thì nó sẽ được thay bởi: x = ((a) > (b) ? (a): (b))

- Chú ý:
  - Tên macro phải được viết liền với dấu ngoặc của danh sách đối. Ví dụ không viết max (A,B).
  - #define bp(x) (x*x) viết sai vì bp(5) đúng nhưng bp(a+b) sẽ thành (a+b*a+b) (tức a+b+ab).
  - Tương tự như trên, viết #define max(A,B) (A > B ? A: B) là sai (?) vì vậy luôn luôn bao các đối bởi dấu ngoặc đơn ().
  - #define bp(x) ((x)*(x)) viết đúng nhưng nếu giả sử lập trình viên muốn tính bình phương của 2 bằng đoạn lệnh sau:

    ```
    int i = 1;
    printf("Ket qua: %d",bp(++i));
    ```

    thì kết quả in ra sẽ là 6 (trên Dev-C++ kết quả này là 9 ?) thay vì kết quả mong muốn là 4. Lí do là ở đây là chương trình dịch sẽ thay bp(++i) bởi ((++i)*(++i)), và với i = 1 chương trình sẽ thực hiện như 2*3 = 6. Do vậy cần cẩn thận khi sử dụng các phép toán tự tăng giảm trong các macro có đối. Nói chung, nên hạn chế việc sử dụng các macro phức tạp, vì nó có thể gây nên những hiệu ứng phụ khó kiểm soát.

**Các chỉ thị biên dịch có điều kiện #if, #ifdef, #ifndef**

1. #if dãy lệnh … #endif
2. #if dãy lệnh … #else dãy lệnh … #endif,
3. #ifdef và #ifndef

### *2. Định nghĩa kiểu dữ liệu*

Dùng để đặt tên lại cho một kiểu dữ liệu nào đó để gợi nhớ hay đặt 1 kiểu dữ liệu cho riêng mình dựa trên các kiểu dữ liệu đã có; phần này không bắt buộc phải có. Trong C/C++ cho phép định nghĩa kiểu dữ liệu với từ khóa struc và typedef; nội dung chi tiết về vấn đề này được trình bày trong phần chương 6.

### *3. Khai báo các prototype*

Khai báo mô tả các hàm sẽ dùng trong chương trình chính bao gồm tên hàm, các tham số hình thức, kiểu dữ liệu trả về của hàm. Phần này là không bắt buộc, tuy nhiên những hàm được dùng trong chương trình chính hoặc chương trình con khác bắt buộc phải được khai báo prototype trước khi sử dụng. Tại vị trí này có thể khai báo đầy đủ một hàm bao gồm cả phần mô tả và thân

hàm, tuy nhiên cách viết đó làm cho chương trình chính bị đẩy sâu xuống cuối chương trình làm cho chương trình khó theo dõi. Vì vậy việc viết các hàm, chương trình con thường được tổ chức thành 2 phần là phần khai báo prototype trước hàm main và phần cài đặt nội dung hàm sau hàm main. Nội dung chi tiết về công việc này sẽ được đề cập cụ thể trong phần sau của chương này.

***4. Khai báo các biến ngoài (các biến toàn cục)***

Cho phép khai báo các biến, hằng có phạm vi tác động trong toàn bộ chương trình. Phần này là tùy vào nhu cầu xử dụng trong mỗi chương trình mà có thể có hoặc không.

***5. Chương trình chính***

Phần này bắt buộc phải có. Trong C/C++ chương trình chính được qui định có tên là main. Phần này đã được mô tả trong các phần sau.

***6. Cài đặt các hàm***

Đây là phần mã nguồn đầy đủ của các hàm. Nội dung này sẽ được nghiên cứu trong các phần tiếp theo của chương này.

## 3. Hàm xây dựng sẵn

7. **stdio.h**: Thư viện chứa các hàm vào/ ra chuẩn (standard input/output). Gồm các hàm printf(), scanf(), getc(), putc(), gets(), puts(), fflush(), fopen(), fclose(), fread(), fwrite(), getchar(), putchar(), getw(), putw()…
8. **conio.h**: Thư viện chứa các hàm vào ra trong chế độ DOS (DOS console). Gồm các hàm clrscr(), getch(), getche(), getpass(), cgets(), cputs(), putch(), clreol(),…
9. **math.h**: Thư viện chứa các hàm tính toán gồm các hàm abs(), sqrt(), log(). log10(), sin(), cos(), tan(), acos(), asin(), atan(), pow(), exp(),…
10. **alloc.h**: Thư viện chứa các hàm liên quan đến việc quản lý bộ nhơ. Gồm các hàm calloc(), realloc(), malloc(), free(), farmalloc(), farcalloc(), farfree(), …
11. **io.h**: Thư viện chứa các hàm vào ra cấp thấp. Gồm các hàm open(), _open(), read(), _read(), close(), _close(), creat(), _creat(), creatnew(), eof(), filelength(), lock(),…
12. **graphics.h**: Thư viện chứa các hàm liên quan đến đồ họa. Gồm initgraph(), line(), circle(), putpixel(), getpixel(), setcolor(), …

# *II. Hàm do người dùng định nghĩa*

Hàm nhận (hoặc không) các đối số và trả lại (hoặc không) một giá trị cho chương trình gọi nó. Trong trường hợp không trả lại giá trị, hàm hoạt động như một thủ tục trong các ngôn ngữ lập trình khác. Một chương trình là tập các hàm, trong đó có một hàm chính với tên gọi main(), khi chạy chương trình, hàm main() sẽ được chạy đầu tiên và gọi đến hàm khác. Kết thúc hàm main() cũng là kết thúc chương trình.

Hàm giúp cho việc phân đoạn chương trình thành những môđun riêng rẽ, hoạt động độc lập với ngữ nghĩa của chương trình lớn, có nghĩa một hàm có thể được sử dụng trong chương trình này mà cũng có thể được sử dụng trong chương trình khác, dễ cho việc kiểm tra và bảo trì chương trình. Hàm có một số đặc trưng:

- Nằm trong hoặc ngoài văn bản có chương trình gọi đến hàm. Trong một văn bản có thể chứa nhiều hàm;

- Được gọi từ chương trình chính (main), từ hàm khác hoặc từ chính nó (đệ quy);
- Không lồng nhau;
- Có 3 cách truyền giá trị: Truyền theo tham trị, tham biến và tham trỏ.

## 1. Khai báo và định nghĩa Hàm

### *1. Khai báo*

Một hàm thường làm chức năng:

- Tính toán trên các tham đối và cho lại giá trị kết quả, hoặc;
- Chỉ đơn thuần thực hiện một chức năng nào đó, không trả lại kết quả tính toán.

Thông thường kiểu của giá trị trả lại được gọi là kiểu của hàm. Các hàm thường được khai báo ở đầu chương trình. Các hàm viết sẵn được khai báo trong các file nguyên mẫu *.h. Do đó, để sử dụng được các hàm này, cần có chỉ thị #include <*.h> ở ngay đầu chương trình, trong đó *.h là tên file cụ thể có chứa khai báo của các hàm được sử dụng (ví dụ để sử dụng các hàm toán học ta cần khai báo file nguyên mẫu math.h).

**Khai báo một hàm như sau:**

```
<kiểu giá trị trả lại> <tên hàm>(d/s kiểu đối) ;
```

- Trong đó, kiểu giá trị trả lại còn gọi là kiểu hàm và có thể nhận kiểu bất kỳ chuẩn của C++ và cả kiểu của NSD tự tạo. Đặc biệt nếu hàm không trả lại giá trị thì kiểu của giá trị trả lại được khai báo là void. Nếu kiểu giá trị trả lại được bỏ qua thì chương trình ngầm định hàm có kiểu là int (phân biệt với void !).
- Ví dụ:

```
int bp(int);              // Khai báo hàm bp, có đối kiểu int và kiểu hàm là int
int rand100();            // Không đối, kiểu hàm (giá trị trả lại) là int
void alltrim(char[ ]) ;   // đối là xâu kí tự, hàm không trả lại giá trị (không kiểu).
cong(int, int);           // Hai đối kiểu int, kiểu hàm là int (ngầm định).
```

Thông thường để chương trình được rõ ràng chúng ta nên tránh lạm dụng các ngầm định. Ví dụ trong khai báo cong(int, int); nên khai báo rõ cả kiểu hàm (trong trường hợp này kiểu hàm ngầm định là int) như sau : int cong(int, int);.

### *2. Định nghĩa hàm*

Cấu trúc một hàm bất kỳ được bố trí cũng giống như hàm main() trong các phần trước.

**Hàm có trả về giá trị**

```
<kiểu hàm> <tên hàm>(danh sách tham đối hình thức)
{
    khai báo cục bộ của hàm ; // chỉ dùng riêng cho hàm này
    dãy lệnh của hàm ;
    return (biểu thức trả về);    // có thể nằm đâu đó trong dãy lệnh.
}
```

- Trong đó:
  - Danh sách tham đối hình thức còn được gọi ngắn gọn là danh sách đối gồm dãy

các đối cách nhau bởi dấu phẩy, đối có thể là một biến thường, biến tham chiếu hoặc biến con trỏ, hai loại biến sau ta sẽ trình bày trong các phần tới. Mỗi đối được khai báo giống như khai báo biến, tức là cặp gồm <kiểu đối> <tên đối>.

- Với hàm có trả lại giá trị cần có câu lệnh return kèm theo sau là một biểu thức. Kiểu của giá trị biểu thức này chính là kiểu của hàm đã được khai báo ở phần tên hàm. Câu lênh return có thể nằm ở vị trí bất kỳ trong phần câu lệnh, tuỳ thuộc mục đích của hàm. Khi gặp câu lệnh return chương trình tức khắc thoát khỏi hàm và trả lại giá trị của biểu thức sau return như giá trị của hàm.

- Ví dụ : Ví dụ sau định nghĩa hàm tính luỹ thừa n (với n nguyên) của một số thực bất kỳ. Hàm này có hai đầu vào (đối thực x và số mũ nguyên n) và đầu ra (giá trị trả lại) kiểu thực với độ chính xác gấp đôi là $x^n$.

```
double luythua(float x, int n)
{
    int i ;                          // biến chỉ số
    double kq = 1 ;                  // để lưu kết quả
    for (i=1; i<=n; i++) kq *= x ;
    return kq;
}
```

**Hàm không trả về giá trị**

```
void   <tên hàm>(danh sách tham đối hình thức)
{
    khai báo cục bộ của hàm ; // chỉ dùng riêng cho hàm này
    dãy lệnh của hàm ;
    return;                          // có thể nằm đâu đó trong dãy lệnh.
}
```

- Nếu hàm không trả lại giá trị (tức kiểu hàm là void), khi đó có thể có hoặc không có câu lệnh return, nếu có thì đằng sau return sẽ không có biểu thức giá trị trả lại.

- Ví dụ: Hàm xoá màn hình 100 lần, hàm chỉ làm công việc cẩn thận xoá màn hình nhiều lần để màn hình thật sạch, nên không có giá trị gì để trả lại.

```
void xmh()
{
    int i;
    for (i=1; i<=100; i++)
    clrscr();
    return ;
}
```

- Hàm main() thông thường có hoặc không có giá trị trả về cho hệ điều hành khi chương trình chạy xong, vì vậy ta thường khai báo kiểu hàm là int main() hoặc void main() và câu lệnh cuối cùng trong hàm thường là return 1 hoặc return. Trường hợp bỏ qua từ khoá void nhưng trong thân hàm không có câu lệnh return (giống phần lớn ví dụ trong giáo trình này) chương trình sẽ ngầm hiểu hàm main() trả lại một giá trị nguyên nhưng vì không có nên khi dịch chương trình ta sẽ gặp lời cảnh báo "Cần có giá trị trả lại cho hàm" (một lời cảnh báo không phải là lỗi, chương trình vẫn chạy bình thường). Để tránh bị quấy rầy về những lời cảnh báo "không mời" này chúng ta có thể đặt thêm câu lệnh return 0; (nếu không khai báo void main()) hoặc khai báo kiểu hàm là void main() và đặt câu lệnh return vào cuối hàm.

**Chú ý:**

- Danh sách đối trong khai báo hàm có thể chứa hoặc không chứa tên đối, thông thường ta chỉ khai báo kiểu đối chứ không cần khai báo tên đối, trong khi ở dòng đầu tiên của định nghĩa hàm phải có tên đối đầy đủ;
- Cuối khai báo hàm phải có dấu chấm phẩy (;), trong khi cuối dòng đầu tiên của định nghĩa hàm không có dấu chấm phẩy;
- Hàm có thể không có đối (danh sách đối rỗng), tuy nhiên cặp dấu ngoặc sau tên hàm vẫn phải được viết. Ví dụ clrscr(), lamtho(), vietgiaotrinh(), … ;
- Một hàm có thể không cần phải khai báo nếu nó được định nghĩa trước khi có hàm nào đó gọi đến nó. Ví dụ có thể viết hàm main() trước (trong văn bản chương trình), rồi sau đó mới viết đến các hàm "con". Do trong hàm main() chắc chắn sẽ gọi đến hàm con này nên danh sách của chúng phải được khai báo trước hàm main(). Trường hợp ngược lại nếu các hàm con được viết (định nghĩa) trước thì không cần phải khai báo chúng nữa (vì trong định nghĩa đã hàm ý khai báo). Nguyên tắc này áp dụng cho hai hàm A, B bất kỳ chứ không riêng cho hàm main(), nghĩa là nếu B gọi đến A thì trước đó A phải được định nghĩa hoặc ít nhất cũng có dòng khai báo về A.

## 2. Lời gọi Hàm

Lời gọi hàm được phép xuất hiện trong bất kỳ biểu thức, câu lệnh của hàm khác … Nếu lời gọi hàm lại nằm trong chính bản thân hàm đó thì ta gọi là đệ quy. Để gọi hàm ta chỉ cần viết tên hàm và danh sách các giá trị cụ thể truyền cho các đối đặt trong cặp dấu ngoặc tròn ().

**Cú pháp**

Tên_hàm(danh sách tham đối thực sự) ;

- Trong đó:
    - Danh sách tham đối thực sự còn gọi là danh sách giá trị gồm các giá trị cụ thể để gán lần lượt cho các đối hình thức của hàm. Khi hàm được gọi thực hiện thì tất cả những vị trí xuất hiện của đối hình thức sẽ được gán cho giá trị cụ thể của đối thực sự tương ứng trong danh sách, sau đó hàm tiến hành thực hiện các câu lệnh của hàm (để tính kết quả);
    - Danh sách tham đối thực sự truyền cho tham đối hình thức có số lượng bằng với số lượng đối trong hàm và được truyền cho đối theo thứ tự tương ứng. Các tham đối thực sự có thể là các hằng, các biến hoặc biểu thức. Biến trong giá trị có thể trùng với tên đối. Ví dụ ta có hàm in n lần kí tự c với tên hàm inkitu(int n, char c); và lời gọi hàm inkitu(12, 'A'); thì n và c là các đối hình thức, 12 và 'A' là các đối thực sự hoặc giá trị. Các đối hình thức n và c sẽ lần lượt được gán bằng các giá trị tương ứng là 12 và 'A' trước khi tiến hành các câu lệnh trong phần thân hàm. Giả sử hàm in kí tự được khai báo lại thành inkitu(char c, int n); thì lời gọi hàm cũng phải được thay lại thành inkitu('A', 12);
    - Các giá trị tương ứng được truyền cho đối phải có kiểu cùng với kiểu đối (hoặc C++ có thể tự động chuyển kiểu được về kiểu của đối);
    - Khi một hàm được gọi, nơi gọi tạm thời chuyển điều khiển đến thực hiện dòng lệnh đầu tiên trong hàm được gọi. Sau khi kết thúc thực hiện hàm, điều khiển lại được trả về thực hiện tiếp câu lệnh sau lệnh gọi hàm của nơi gọi.

- Ví dụ: Giả sử ta cần tính giá trị của biểu thức $2x^3 - 5x^2 - 4x + 1$, thay cho việc tính trực tiếp $x^3$ và $x^2$, ta có thể gọi hàm luythua() trong ví dụ trên để tính các giá trị này bằng cách gọi nó trong hàm main() như sau:

```
#include <iostream.h>
#include <iomanip.h>
void xmh(int);
double luythua(float x, int n);
main()                                  // tính giá trị 2x^3 - 5x^2 - 4x + 1
{
      float x ;                         // tên biến có thể trùng với đối của hàm
      double f ;                        // để lưu kết quả
      printf("x = ");
      scanf("%f" ,&x ) ;
      f = 2*luythua(x,3) - 5*luythua(x,2) - 4*x + 1;
      xmh(100);                         // xoá thật sạch màn hình 100 lần
      printf("%d",f) ;
      getch();
      return 0;
}
double luythua(float x, int n) // trả lại giá trị x^n
{
      int i ;                           // biến chỉ số
      double kq = 1 ;                   // để lưu kết quả
      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();
      return ;
}
```

Qua ví dụ này ta thấy lợi ích của lập trình cấu trúc, chương trình trở nên gọn hơn, chẳng hạn hàm luythua() chỉ được viết một lần nhưng có thể sử dụng nó nhiều lần (2 lần trong ví dụ này) chỉ bằng một câu lệnh gọi đơn giản cho mỗi lần sử dụng thay vì phải viết lại nhiều lần đoạn lệnh tính luỹ thừa.

## 3. Hàm với đối mặc định

- Mục này và mục sau chúng ta bàn đến một vài mở rộng thiết thực của C/C++ đối với C có liên quan đến hàm, đó là hàm với đối mặc định và cách tạo, sử dụng các hàm có chung tên gọi. Một mở rộng quan trọng khác là cách truyền đối theo tham chiếu sẽ được bàn chung trong mục truyền tham đối thực sự cho hàm.
- Trong phần trước chúng ta đã khẳng định số lượng tham đối thực sự phải bằng số lượng tham đối hình thức khi gọi hàm. Tuy nhiên, trong thực tế rất nhiều lần hàm được gọi với các giá trị của một số tham đối hình thức được lặp đi lặp lại. Trong trường hợp như vậy lúc nào cũng phải viết một danh sách dài các tham đối thực sự giống nhau cho mỗi lần gọi là một công việc không mấy thú vị. Từ thực tế đó C++ đưa ra một cú pháp mới về hàm sao cho một danh sách tham đối thực sự trong lời gọi không nhất thiết phải viết đầy

đủ nếu một số trong chúng đã có sẵn những giá trị định trước.

**Cú pháp**

<kiểu hàm> <tên hàm>(đ1, ..., đn, đmđ1 = gt1, ..., đmđm = gtm

- Trong đó:
- Các đối đ1, …, đn và đối mặc định đmđ1, …, đmđm đều được khai báo như cũ nghĩa là gồm có kiểu đối và tên đối;
- Riêng các đối mặc định đmđ1, …, đmđm có gán thêm các giá trị mặc định gt1, …, gtm. Một lời gọi bất kỳ khi gọi đến hàm này đều phải có đầy đủ các tham đối thực sự ứng với các đ1, …, đm nhưng có thể có hoặc không các tham đối thực sự ứng với các đối mặc định đmđ1, …, đmđm. Nếu tham đối nào không có tham đối thực sự thì nó sẽ được tự động gán giá trị mặc định đã khai báo.
- Ví dụ 4.5: Xét hàm xmh(int n = 100), trong đó n mặc định là 100, nghĩa là nếu gọi xmh(99) thì màn hình được xoá 99 lần, còn nếu gọi xmh(100) hoặc gọn hơn xmh() thì chương trình sẽ xoá màn hình 100 lần. Tương tự, xét hàm int luythua(float x, int n = 2); Hàm này có một tham đối mặc định là số mũ n, nếu lời gọi hàm bỏ qua số mũ này thì chương trình hiểu là tính bình phương của x (n = 2). Ví dụ lời gọi luythua(4, 3) được hiểu là $4^3$, còn luythua(2) được hiểu là $2^2$.
- Hàm tính tổng 4 số nguyên: int tong(int m, int n, int i = 0; int j = 0); khi đó có thể tính tổng của 5, 2, 3, 7 bằng lời gọi hàm tong(5,2,3,7) hoặc có thể chỉ tính tổng 3 số 4, 2, 1 bằng lời gọi tong(4,2,1) hoặc cũng có thể gọi tong(6,4) chỉ để tính tổng của 2 số 6 và 4.
- Chú ý: Các đối ngầm định phải được khai báo liên tục và xuất hiện cuối cùng trong danh sách đối. Ví dụ:

```
int tong(int x, int y=2, int z, int t=1);      // sai vì các đối mặc định không liên tục
void xoa(int x=0, int y)                       // sai vì đối mặc định không ở cuối
```

## 4. Khai báo hàm trùng tên

- Hàm trùng tên hay còn gọi là hàm chồng (đè). Đây là một kỹ thuật cho phép sử dụng cùng một tên gọi cho các hàm "giống nhau" (cùng mục đích) nhưng xử lý trên các kiểu dữ liệu khác nhau hoặc trên số lượng dữ liệu khác nhau. Ví dụ hàm sau tìm số lớn nhất trong 2 số nguyên:

```
int max(int a, int b) { return (a > b) ? a: b ; }
```

  - Nếu đặt c = max(3, 5) ta sẽ có c = 5.
  - Tuy nhiên cũng tương tự như vậy nếu đặt c = max(3.0, 5.0) chương trình sẽ bị lỗi vì các giá trị (float) không phù hợp về kiểu (int) của đối trong hàm max.
- Trong trường hợp như vậy chúng ta phải viết hàm mới để tính max của 2 số thực. Mục đích, cách làm việc của hàm này hoàn toàn giống hàm trước, tuy nhiên trong C và các ngôn ngữ lập trình khác chúng ta buộc phải sử dụng một tên mới cho hàm "mới" này. Ví dụ:

```
float fmax(float a, float b) { return (a > b) ? a: b ; }
```

Tương tự để tuận tiện ta sẽ viết thêm các hàm:

```
char cmax(char a, char b) { return (a > b) ? a: b ; }
```

```
long lmax(long a, long b) { return (a > b) ? a: b ; }
double dmax(double a, double b) { return (a > b) ? a: b ; }
```

Tóm lại ta sẽ có 5 hàm: max, cmax, fmax, lmax, dmax, việc sử dụng tên như vậy sẽ gây bất lợi khi cần gọi hàm.

- C++ cho phép ta có thể khai báo và định nghĩa cả 5 hàm trên với cùng 1 tên gọi ví dụ là max chẳng hạn. Khi đó ta có 5 hàm:

```
int max(int a, int b)                 { return (a > b) ? a: b ; }
float    max(float a, float b)        { return (a > b) ? a: b ; }
char     max(char a, char b)          { return (a > b) ? a: b ; }
long     max(long a, long b)          { return (a > b) ? a: b ; }
double   max(double a, double b)      { return (a > b) ? a: b ; }
```

Và lời gọi hàm bất kỳ dạng nào như max(3,5), max(3.0,5), max('O', 'K') đều được đáp ứng. Chúng ta có thể đặt ra vấn đề: với cả 5 hàm cùng tên như vậy, chương trình gọi đến hàm nào. Vấn đề được giải quyết dễ dàng vì chương trình sẽ dựa vào kiểu của các đối khi gọi để quyết định chạy hàm nào. Ví dụ lời gọi max(3,5) có 2 đối đều là kiểu nguyên nên chương trình sẽ gọi hàm 1, lời gọi max(3.0,5) hướng đến hàm số 2 và tương tự chương trình sẽ chạy hàm số 3 khi gặp lời gọi max('O','K').

- Một đặc điểm của các hàm trùng tên đó là trong danh sách đối của chúng phải có ít nhất một cặp đối nào đó khác kiểu nhau. Một đặc trưng khác để phân biệt thông qua các đối đó là số lượng đối trong các hàm phải khác nhau (nếu kiểu của chúng là giống nhau). Ví dụ việc vẽ các hình: thẳng, tam giác, vuông, chữ nhật trên màn hình là giống nhau, chúng chỉ phụ thuộc vào số lượng các điểm nối và toạ độ của chúng. Do vậy ta có thể khai báo và định nghĩa 4 hàm vẽ nói trên với cùng chung tên gọi. Chẳng hạn:

```
void ve(Diem A, Diem B) ;                    // vẽ đường thẳng AB
void ve(Diem A, Diem B, Diem C) ;            // vẽ tam giác ABC
void ve(Diem A, Diem B, Diem C, Diem D) ;    // vẽ tứ giác ABCD
```

Trong ví dụ trên ta giả thiết Diem là một kiểu dữ liệu lưu toạ độ của các điểm trên màn hình. Hàm ve(Diem A, Diem B, Diem C, Diem D) sẽ vẽ hình vuông, chữ nhật, thoi, bình hành hay hình thang phụ thuộc vào toạ độ của 4 điểm ABCD, nói chung nó được sử dụng để vẽ một tứ giác bất kỳ.

- Tóm lại nhiều hàm có thể được định nghĩa chồng (với cùng tên gọi giống nhau) nếu chúng thoả các điều kiện sau:
  - Số lượng các tham đối trong hàm là khác nhau, hoặc
  - Kiểu của tham đối trong hàm là khác nhau.

## 5. Truyền tham số

Có 3 cách truyền tham đối thực sự cho các tham đối hình thức trong lời gọi hàm. Trong đó cách ta đã dùng cho đến thời điểm hiện nay được gọi là truyền theo tham trị, tức các đối hình thức sẽ nhận các giá trị cụ thể từ lời gọi hàm và tiến hành tính toán rồi trả lại giá trị. Để dễ hiểu các cách truyền đối chúng ta sẽ xem qua cách thức chương trình thực hiện với các đối khi thực hiện hàm.

### 1. *Truyền theo tham trị*

- Ta xét lại ví dụ hàm luythua(float x, int n) tính $x^n$. Giả sử trong chương trình chính ta có các biến a, b, f đang chứa các giá trị a = 2, b = 3, và f chưa có giá trị. Để tính $a^b$ và gán giá trị tính được cho f, ta có thể gọi f = luythua(a,b). Khi gặp lời gọi này, chương trình sẽ tổ chức như sau:
  - Tạo 2 biến mới (tức 2 ô nhớ trong bộ nhớ) có tên x và n. Gán nội dung các ô nhớ này bằng các giá trị trong lời gọi, tức gán 2 (a) cho x và 3 (b) cho n;
  - Tới phần khai báo (của hàm), chương trình tạo thêm các ô nhớ mang tên kq và i;
  - Tiến hành tính toán (gán lại kết quả cho kq);
  - Cuối cùng lấy kết quả trong kq gán cho ô nhớ f (là ô nhớ có sẵn đã được khai báo trước, nằm bên ngoài hàm);
  - Kết thúc hàm quay về chương trình gọi. Do hàm luythua đã hoàn thành xong việc tính toán nên các ô nhớ được tạo ra trong khi thực hiện hàm (x, n, kq, i) sẽ được xoá khỏi bộ nhớ. Kết quả tính toán được lưu giữ trong ô nhớ f (không bị xoá vì không liên quan gì đến hàm)
- Trên đây là truyền đối theo cách thông thường. Vấn đề đặt ra là giả sử ngoài việc tính f, ta còn muốn thay đổi các giá trị của các ô nhớ a, b (khi truyền nó cho hàm) thì có thể thực hiện được không ? Để giải quyết bài toán này ta cần theo một kỹ thuật khác, nhờ vào vai trò của biến con trỏ và tham chiếu.

### 2. *Truyền theo dẫn trỏ*

- Xét ví dụ tráo đổi giá trị của 2 biến. Đây là một yêu cầu nhỏ nhưng được gặp nhiều lần trong chương trình, ví dụ để sắp xếp một danh sách. Do vậy cần viết một hàm để thực hiện yêu cầu trên. Hàm không trả kết quả. Do các biến cần trao đổi là chưa được biết trước tại thời điểm viết hàm, nên ta phải đưa chúng vào hàm như các tham đối, tức hàm có hai tham đối x, y đại diện cho các biến sẽ thay đổi giá trị sau này.
- Từ một vài nhận xét trên, theo thông thường hàm tráo đổi sẽ được viết như sau:

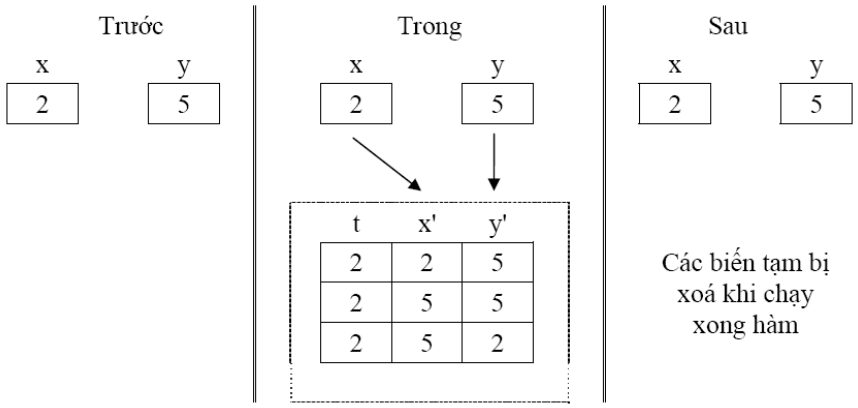
```
void doi_cho(int x, int y)
{
    int t ;
    t = x ;
    x = y ;
    y = t ;
}
```

Giả sử trong chương trình chính ta có 2 biến x, y chứa các giá trị lần lượt là 2, 5. Ta cần đổi nội dung 2 biến này sao cho x = 5 còn y = 2 bằng cách gọi đến hàm doi_cho(x, y).

```
int main()
{
     int x = 2;
     int y = 5;
     doi_cho(x, y) ;
     printf("%d, %d",x, y);          //      2, 5 (x, y vẫn không đổi)
     getch();
     return 0;
}
```

Thực sự sau khi chạy xong chương trình ta thấy giá trị của x và y vẫn không thay đổi !?.

- Như đã giải thích trong mục trên (gọi hàm luythua), việc đầu tiên khi chương trình thực hiện một hàm là tạo ra các biến mới (các ô nhớ mới, độc lập với các ô nhớ x, y đã có sẵn) tương ứng với các tham đối, trong trường hợp này cũng có tên là x, y và gán nội dung của x, y (ngoài hàm) cho x, y (mới). Và việc cuối cùng của chương trình sau khi thực hiện xong hàm là xoá các biến mới này. Do vậy nội dung của các biến mới thực sự là có thay đổi, nhưng không ảnh hưởng gì đến các biến x, y cũ. Hình vẽ dưới đây minh hoạ cách làm việc của hàm swap, trước, trong và sau khi gọi hàm.




*Hình 4.1* - Sự thay đổi giá trị của tham số thực

- Như vậy hàm doi_cho cần được viết lại sao cho việc thay đổi giá trị không thực hiện trên các biến tạm mà phải thực sự thực hiện trên các biến ngoài. Muốn vậy thay vì truyền giá trị của các biến ngoài cho đối, bây giờ ta sẽ truyền địa chỉ của nó cho đối, và các thay đổi sẽ phải thực hiện trên nội dung của các địa chỉ này. Đó chính là lý do ta phải sử dụng con trỏ để làm tham đối thay cho biến thường. Cụ thể hàm swap được viết lại như sau:

```
void doi_cho(int *p, int *q)
{
    int t;              // khai báo biến tạm t
    t = *p ;            // đặt giá trị của t bằng nội dung nơi p trỏ tới
    *p = *q ;           // thay nội dung nơi p trỏ bằng nội dung nơi q trỏ
    *q = t ;            // thay nội dung nơi q trỏ tới bằng nội dung của t
}
```

Với cách tổ chức hàm như vậy rõ ràng nếu ta cho p trỏ tới biến x và q trỏ tới biến y thì hàm doi_cho sẽ thực sự làm thay đổi nội dung của x, y chứ không phải của p, q. Từ đó lời gọi hàm sẽ là doi_cho(&x, &y) (tức truyền địa chỉ của x cho p, p trỏ tới x và tương tự q trỏ tới y).

- Như vậy có thể tóm tắt 3 đặc trưng để viết một hàm làm thay đổi giá trị biến ngoài như sau:
  - Đối của hàm phải là con trỏ (ví dụ int *p);
  - Các thao tác liên quan đến đối này (trong thân hàm) phải thực hiện tại nơi nó trỏ đến (ví dụ *p = …);
  - Lời gọi hàm phải chuyển địa chỉ cho p (ví dụ &x).
- Ngoài hàm doi_cho đã trình bày, ở đây ta đưa thêm ví dụ để thấy sự cần thiết phải có hàm

cho phép thay đổi biến ngoài. Ví dụ hàm giải phương trình bậc 2 với yêu cầu:

- Cho trước 3 số a, b, c như 3 hệ số của phương trình, cần tìm 2 nghiệm x1, x2 của nó.
- Không thể lấy giá trị trả lại của hàm để làm nghiệm vì giá trị trả lại chỉ có 1 trong khi ta cần đến 2 nghiệm.
- Do vậy ta cần khai báo 2 biến "ngoài" trong chương trình để chứa các nghiệm, và hàm phải làm thay đổi 2 biến này (tức chứa giá trị nghiệm giải được).
- Như vậy hàm được viết cần phải có 5 đối, trong đó 3 đối a, b, c đại diện cho các hệ số, không thay đổi và 2 biến x1, x2 đại diện cho nghiệm, 2 đối này phải được khai báo dạng con trỏ.
- Ngoài ra, phương trình có thể vô nghiệm, 1 nghiệm hoặc 2 nghiệm do vậy hàm sẽ trả lại giá trị là số nghiệm của phương trình, trong trường hợp 1 nghiệm (nghiệm kép), giá trị nghiệm sẽ được cho vào x1.

```
int gptb2(float a, float b, float c, float *p, float *q)
{
    float d ;                   // để chứa Δ
    d = (b*b) - 4*a*c ;
    if (d < 0) return 0 ;
    else
        if (d == 0)
            { *p = -b/(2*a) ; return 1 ; }
        else
        {
            *p = (-b + sqrt(d))/(2*a) ;
            *q = (-b - sqrt(d))/(2*a) ;
            return 2 ;
        }
}
```

- Một ví dụ của lời gọi hàm trong chương trình chính như sau:

```
main()
{
    float a, b, c ;             // các hệ số
    float x1, x2 ;              // các nghiệm
    printf("Nhập hệ số: ");
    scanf("%f %f %f",&a, &b, &c);
    switch (gptb2(a, b, c, &x1, &x2))
    {
        case 0:
            printf("Phương trình vô nghiệm");
            break;
        case 1:
            printf("Phương trình có nghiệm kép x = %f", x1);
            break;
        case 2:
            printf("Phương trình có 2 nghiệm phân biệt:\n");
            printf("x1 = %f , và x2 = %f", x1, x2);
            break;
    }
```

```
}
```

- Trên đây chúng ta đã trình bày cách xây dựng các hàm cho phép thay đổi giá trị của biến ngoài. Một đặc trưng dễ nhận thấy là cách viết hàm tương đối phức tạp. Do vậy C++ đã phát triển một cách viết khác dựa trên đối tham chiếu và việc truyền đối cho hàm được gọi là truyền theo tham chiếu.

### *3. Truyền theo tham chiếu*

- Một biến có thể được gán cho một bí danh mới, và khi đó chỗ nào xuất hiện biến thì cũng tương đương như dùng bí danh và ngược lại. Một bí danh như vậy được gọi là một biến tham chiếu, ý nghĩa thực tế của nó là cho phép "tham chiếu" tới một biến khác cùng kiểu của nó, tức sử dụng biến khác nhưng bằng tên của biến tham chiếu.
- Giống khai báo biến bình thường, tuy nhiên trước tên biến ta thêm dấu và (&). Có thể tạm phân biến thành 3 loại: biến thường với tên thường, biến con trỏ với dấu * trước tên và biến tham chiếu với dấu &.

```
<kiểu biến> &<tên biến tham chiếu> = <tên biến được tham chiếu>;
```

- Cú pháp khai báo này cho phép ta tạo ra một biến tham chiếu mới và cho nó tham chiếu đến biến được tham chiếu (cùng kiểu và phải được khai báo từ trước). Khi đó biến tham chiếu còn được gọi là bí danh của biến được tham chiếu. Chú ý không có cú pháp khai báo chỉ tên biến tham chiếu mà không kèm theo khởi tạo. Ví dụ:

```
int hung, dung ;            // khai báo các biến nguyên hung, dung
int &ti = hung;             // khai báo biến tham chiếu ti, teo tham chieu đến
int &teo = dung;            // hung dung. ti, teo là bí danh của hung, dung
```

Từ vị trí này trở đi việc sử dụng các tên hung, ti hoặc dung, teo là như nhau. Ví dụ:

```
hung = 2 ;
ti ++;                      // tương đương hung ++;
printf("%d, %d",hung, ti);  // 3 3
teo = ti + hung ;           // tương đương dung = hung + hung
dung ++ ;                   // tương đương teo ++
printf("%d, %d",dung,teo);  // 7 7
```

- Vậy sử dụng thêm biến tham chiếu để làm gì ?
  - Cách tổ chức bên trong của một biến tham chiếu khác với biến thường ở chỗ nội dung của nó là địa chỉ của biến mà nó đại diện (giống biến con trỏ), ví dụ câu lệnh

    ```
    printf("%d", teo) ;          // 7
    ```

    in ra giá trị 7 nhưng thực chất đây không phải là nội dung của biến teo, nội dung của teo là địa chỉ của dung, khi cần in teo, chương trình sẽ tham chiếu đến dung và in ra nội dung của dung (7). Các hoạt động khác trên teo cũng vậy (ví dụ teo++), thực chất là tăng một đơn vị nội dung của dung (chứ không phải của teo). Từ cách tổ chức của biến tham chiếu ta thấy chúng giống con trỏ nhưng thuận lợi hơn ở chỗ khi truy cập đến giá trị của biến được tham chiếu (dung) ta chỉ cần ghi tên biến tham chiếu (teo) chứ không cần thêm toán tử (*) ở trước như trường hợp dùng con trỏ. Điểm khác biệt này có ích khi được sử dụng để truyền đối cho các

hàm với mục đích làm thay đổi nội dung của biến ngoài.

- Chú ý:
  - Biến tham chiếu phải được khởi tạo khi khai báo;
  - Tuy giống con trỏ nhưng không dùng được các phép toán con trỏ cho biến tham chiếu. Nói chung chỉ nên dùng trong truyền đối cho hàm.
- Một hàm viết dưới dạng đối tham chiếu sẽ đơn giản hơn rất nhiều so với đối con trỏ và giống với cách viết bình thường (truyền theo tham trị), trong đó chỉ có một khác biệt đó là các đối khai báo dưới dạng tham chiếu.
- Để so sánh 2 cách sử dụng ta nhắc lại các điểm khi viết hàm theo con trỏ phải chú ý đến, đó là:
  - Đối của hàm phải là con trỏ (ví dụ int *p);
  - Các thao tác liên quan đến đối này trong thân hàm phải thực hiện tại nơi nó trỏ đến (ví dụ *p = …);
  - Lời gọi hàm phải chuyển địa chỉ cho p (ví dụ &x).
- Hãy so sánh với đối tham chiếu, cụ thể:
  - Đối của hàm phải là tham chiếu (ví dụ int &p);
  - Các thao tác liên quan đến đối này phải thực hiện tại nơi nó trỏ đến, tức địa chỉ cần thao tác. Vì một thao tác trên biến tham chiếu thực chất là thao tác trên biến được nó tham chiếu nên trong hàm chỉ cần viết p trong mọi thao tác (thay vì *p như trong con trỏ);
  - Lời gọi hàm phải chuyển địa chỉ cho p. Vì bản thân p khi tham chiếu đến biến nào thì sẽ chứa địa chỉ của biến đó, do đó lời gọi hàm chỉ cần ghi tên biến, ví dụ x (thay vì &x như đối với dẫn trỏ).
- Tóm lại, đối với hàm viết theo tham chiếu chỉ thay đổi ở đối (là các tham chiếu) còn lại mọi nơi khác đều viết đơn giản như cách viết truyền theo tham trị.
- Ví dụ: Đổi giá trị 2 biến

```
void doi_cho(int &x, int &y)
{
    int t ;
    t = x;
    x = y;
    y = t;
}
```

và lời gọi hàm cũng đơn giản như trong truyền đối theo tham trị. Ví dụ:

```
int a = 5, b = 3;
doi_cho(a, b);
printf("%d, %d",a,b);
```

**Bảng dưới đây minh hoạ tóm tắt 3 cách viết hàm thông qua ví dụ đổi biến ở trên**

| | Tham trị | Tham chiếu | Dẫn trỏ |
|---|---|---|---|
| Khai báo đối | `void doi_cho(int x, int y)` | `void doi_cho(int &x,int &y)` | `void doi_cho(int *x, int *y)` |
| Câu lệnh | `t=x; x=y; y=t` | `t=x; x=y; y=t` | `t=*x; *x=*y; *y=t` |
| Lời gọi | `doi_cho(a,b)` | `doi_cho(a,b)` | `doi_cho(&a,&b)` |
| Tác dụng | `a,b không thay đổi` | `a,b có thay đổi` | `a,b có thay đổi` |

*Bảng 4.1* - So sánh các cách truyền tham số

## 6. Hàm và mảng

### *1. Truyền mảng 1 chiều cho hàm*

- Thông thường chúng ta hay xây dựng các hàm làm việc trên mảng như vectơ hay ma trận các phần tử. Khi đó tham đối thực sự của hàm sẽ là các mảng dữ liệu này. Trong trường hợp này ta có 2 cách khai báo đối.
  - Cách thứ nhất đối được khai báo bình thường như khai báo biến mảng nhưng không cần có số phần tử kèm theo, ví dụ:

    ```
    int x[ ];
    float x[ ];
    ```

  - Cách thứ hai khai báo đối như một con trỏ kiểu phần tử mảng, ví dụ:

    ```
    int *p;
    float *p
    ```

- Trong lời gọi hàm tên mảng a (tham số thực) sẽ được viết vào danh sách tham đối thực sự, vì a là địa chỉ của phần tử đầu tiên của mảng a, nên khi hàm được gọi địa chỉ này sẽ gán cho con trỏ p. Vì vậy giá trị của phần tử thứ i của a có thể được truy cập bởi x[i] (theo khai báo 1) hoặc *(p+i) (theo khai báo 2) và nó cũng có thể được thay đổi thực sự (do đây cũng là cách truyền theo dẫn trỏ). Sau đây là ví dụ đơn giản, nhập và in vectơ, minh hoạ cho cả 2 kiểu khai báo đối.
- Ví dụ: Hàm nhập và in giá trị 1 vectơ

```
#include <stdio.h>
#include <conio.h>
void nhap(int x[ ], int n)          // n: số phần tử
{
    int i;
    for (i=0; i<n; i++)
        scanf("%d",&x[i]);
}
void in(int *p, int n)
{
    int i;
    for (i=0; i<n; i++)
```

```
            printf("%d", *(p+i));
}
int main()
{
    int a[10] ;                 // mảng a chứa tối đa 10 phần tử
    nhap(a,7);                  // vào 7 phần tử đầu tiên cho a
    in(a,7);                    // ra 3 phần tử đầu tiên của a
    getch();
    return 0;
}
```

### *2. Truyền mảng 2 chiều cho hàm*

- Đối với mảng 2 chiều khai báo đối cũng như lời gọi là phức tạp hơn nhiều so với mảng 1 chiều. Ta có hai cách khai báo đối như sau:

**a. Khai báo theo đúng bản chất của mảng 2 chiều float x[m][n]**

- Với khai báo x[m][n] có thể hiểu x là mảng 1 chiều m phần tử, mỗi phần tử của nó có kiểu float[n]. Từ đó, đối được khai báo như một mảng hình thức 1 chiều (không cần số phần tử - ở đây là số dòng) của kiểu float[n]. Tức có thể khai báo như sau:

```
float x[ ][n] ;     // mảng với số phần tử không định trước, mỗi phần tử là n số
float (*x)[n] ;     // một con trỏ, có kiểu là mảng n số (float[n])
```

Để truy nhập đến đến phần tử thứ i, j ta vẫn sử dụng cú pháp x[i][j]. Tên của mảng a (tham số thực) được viết bình thường trong lời gọi hàm. Nói chung theo cách khai báo này việc truy nhập là đơn giản nhưng phương pháp cũng có hạn chế đó là số cột của mảng truyền cho hàm phải cố định bằng n.

- Ví dụ: Tính tổng các số hạng trong ma trận

```
#include <stdio.h>
#include <conio.h>
float tong(float x[][10], int m, int n)
{
    float t = 0;
    int i, j ;
    for (i=0; i<m; i++)
    for (j=0; j<n; j++) t += x[i][j] ;
    return t;
}
int main()
{
    float a[8][10], b[8][10] ;
    int i, j, ma, na, mb, nb;
    printf("nhap so dong, so cot ma tran a: ");
    scanf("%d%d",&ma,&na);
    for (i=0; i<ma; i++)
        for (j=0; j<na; j++)
            {
            printf("a[%d,%d]= ",i,j);
            scanf("%f",&a[i][j]);
            }
    printf("nhap so dong, so cot ma tran b: ");
    scanf("%d%d",&mb,&nb);
```

```
    for (i=0; i<mb; i++)
        for (j=0; j<nb; j++)
        {
            printf("a[%d,%d]= ",i,j);
            scanf("%f",&b[i][j]);
        }
    printf("%f\n",tong(a, ma, na));
    printf("%f\n",tong(b, mb, nb));
    getch();
    return 0;
}
```

**b. Xem mảng float x[m][n] thực sự là mảng một chiều float x[m*n]**

- Khi đó sử dụng cách khai báo như trong mảng một chiều, đó là sử dụng con trỏ float *p để truy cập được đến từng phần tử của mảng. Cách này có hạn chế trong lời gọi: địa chỉ truyền cho hàm không phải là mảng a mà cần phải ép kiểu về (float*) (để phù hợp với p). Với cách này gọi k là thứ tự của phần tử a[i][j] trong mảng một chiều (m*n), ta có quan hệ giữa k, i, j như sau: k = *(p + i*n + j) , i = k/n và j = k%n trong đó n là số cột của mảng truyền cho hàm. Điều này có nghĩa để truy cập đến a[i][j] ta có thể viết *(p+i*n+j), ngược lại biết chỉ số k có thể tính được dòng i, cột j của phần tử này. Ưu điểm của cách khai báo này là ta có thể truyền mảng với kích thước bất kỳ (số cột không cần định trước) cho hàm.
- Ví dụ: Tìm phần tử bé nhất của ma trận

```
#include <stdio.h>
#include <conio.h>
void minmt(float *x, int m, int n)
{
    float min = *x;
    int k, kmin;
    for (k=1; k<m*n; k++)
    if (min > x[k])
    {
        min = x[k];
        kmin = k;
    }
    printf("Gia tri min la: %f , tai dong %d, cot %d\n",min,kmin/n,kmin%n);
}
int main()
{
    float a[3][3];
    int i, j ;
    for (i=0; i<3; i++)
    for (j=0; j<3; j++)
    {
    printf("a[%d,%d]",i,j);
    scanf("%f",&a[i][j]);
    }
    minmt((float*)(a), 3, 3) ;
    getch();
    return 0;
}
```

### *3. Giá trị trả lại của hàm là mảng*

- Không có cách nào để giá trị trả lại của một hàm là mảng. Tuy nhiên thực sự mỗi mảng cũng chính là một con trỏ, vì vậy việc hàm trả lại một con trỏ trỏ đến dãy dữ liệu kết quả là tương đương với việc trả lại mảng. Ngoài ra còn một cách dễ dùng hơn đối với mảng 2 chiều là mảng kết quả được trả lại vào trong tham đối của hàm (giống như nghiệm của phương trình bậc 2 được trả lại vào trong các tham đối). Ở đây chúng ta sẽ lần lượt xét 2 cách làm việc này.

**Giá trị trả lại là con trỏ trỏ đến mảng kết quả**

- Trước hết chúng ta xét ví dụ sau đây:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int* day1()
{
      int kq[3] = { 7, 5, 3 };
      return kq;
}
int* day2()
{
      int *kq;
      kq=(int*)malloc(3*sizeof(int));
      *kq = *(kq+1) = *(kq+2) = 0 ;
      return kq ;
}
int main()
{
      int *a, i;
      a = day1();
      for (i=0; i<3; i++)
      printf("%d\n",*(a+i));
      a = day2();
      for (i=0; i<3; i++)
      printf("%d\n",*(a+i));
      getch();
      return 0;
}
```

- Từ ví dụ trên ta thấy hai hàm trả giá trị đều tạo bên trong nó một mảng 3 số nguyên và trả lại địa chỉ mảng này cho con trỏ kết quả hàm. Tuy nhiên, chỉ có day2() là cho lại kết quả đúng. Tại sao ?
    - Xét mảng kq được khai báo và khởi tạo trong day1(), đây là một mảng cục bộ (được tạo bên trong hàm) như sau này chúng ta sẽ thấy, các loại biến "tạm thời" này (và cả các tham đối) chỉ tồn tại trong quá trình hàm hoạt động. Khi hàm kết thúc các biến này sẽ mất đi. Do vậy tuy hàm đã trả lại địa chỉ của kq trước khi nó kết thúc, thế nhưng sau khi hàm thực hiện xong, toàn bộ kq sẽ được xoá khỏi bộ nhớ và vì vậy con trỏ kết quả hàm đã trỏ đến vùng nhớ không còn các giá trị như kq đã có. Từ điều này việc sử dụng hàm trả lại con trỏ là phải hết sức cẩn thận. Muốn trả lại con trỏ cho hàm thì con trỏ này phải trỏ đến dãy dữ liệu nào sao cho nó không mất đi sau khi hàm kết thúc, hay nói khác hơn đó phải là những dãy dữ