Ricky nguyen

ĐỀ CƯƠNG CHI TIẾT

MÔN HỌC

KỸ THUẬT LẬP TRÌNH

(Tài liệu giảng dạy - lưu hành nội bộ)

HÀ NỘI – 4/2007

Tài liệu tham khảo

1. **Kỹ thuật lập trình C –** GS.TS. Phạm Văn ất
2. **Ngôn ngữ lập trình C++** - GS. TS. Phạm Văn Ất.
3. **Kỹ thuật lập trình -** Nguyễn Tiến Huy – Trần Hạnh Nhi.
4. **Ngôn ngữ lập trình C++** - Ngô Trung Việt.

….

Nội dung

Chương I. Tổng quan về C++

Chương II. Các cấu trúc điều khiển

Chương III. Kỹ thuật lập trình đơn thể

Chương IV. Kỹ thuật lập trình dùng mảng

Chương V. Kỹ thuật lập trình dùng con trỏ

Chương VI. Kỹ thuật lập trình với tệp

Chương VII. Dữ liệu kiểu cấu trúc

Phân bổ thời gian:

30 tiết lý thuyết + 30h thực hành

CHƯƠNG I. TỔNG QUAN VỀ C++

I. QUY TRÌNH LÀM VIỆC TRONG C++

I.1. Các bước để lập chương trình bằng C++

Để thực hiện việc viết và thực thi một chương trình đơn giản trong C++, người ta thường làm theo các bước sau:

- Vào môi trường soạn thảo mã lệnh của C++: để làm được việc này, trên máy tính phải được cài đặt phần mềm Turbo C 3.0 (hoặc cao hơn, hoặc Booland C…). Tìm file TC.exe trong thư mục TC\BIN (hoặc TC30\BIN) và thực thi file này.

- Soạn thảo mã lệnh của chương trình: Môi trường soạn thảo của TC là một cửa sổ soạn thảo và hệ thống menu trợ giúp quá trình soạn thảo cũng như dịch và thực thi chương trình. Ta tiến hành soạn thảo mã lệnh của chương trình trong cửa sổ này theo đúng cú pháp của C++.

- Soát lỗi, dịch chương trình: Sau khi soạn thảo mã lệnh bằng ngôn ngữ C++, ta tiến hành dịch chương trình thành ngôn ngữ máy. Quá trình dịch chỉ thành công khi toàn bộ mã lệnh ta soạn thảo không có lỗi cú pháp. Vì vậy, trong quá trình dịch, TC sẽ tiến hành soát lỗi. Quá trình soát lỗi được tiến hành lần lượt qua các dòng lệnh từ trên xuống. Khi gặp lỗi, chương trình dịch sẽ báo lỗi tại vị trí gần nơi xảy ra lỗi. Để làm các công việc dịch – soát lỗi, ta bấm phím F9, nếu chương trình báo lỗi, hãy tiến hành sửa lỗi.

Nếu muốn quá trình dịch cho ta một file thực thi được của chương trình trên đĩa (file .exe) ta cần đảm bảo trên menu: Options\ Linker\ Settings\ Output\ Standard exe đang được chọn.

- Thực thi chương trình: Khi chương trình đã hết lỗi ta có thể thực thi chương trình bằng cách bấm tổ hợp phím Ctrl - F9. Kết thúc quá trình thực thi sẽ quay về môi trường soạn thảo mã lệnh ban đầu.

Các thao tác khi soạn thảo:

Mở file mới: Chọn File\ New hoặc bấm phím chức năng F3, gõ tên file mới vào và bấm Enter.

Mở file có sẵn: Chọn File\ Open hoặc bấm phím chức năng F2 rồi chọn file cần mở và bấm Enter.

Lưu file: Chọn File\ Save hoặc bấm phím chức năng F2 rồi đặt tên file và bấm Enter. Chú ý trong C++, phần mở rộng của file mã nguồn là .CPP.

Đóng file: Bấm tổ hợp phím Alt F3. Nếu file chưa lưu, C++ sẽ yêu cầu lưu file hoặc bỏ qua việc lưu file, hãy bấm Y hoặc N nếu muốn lưu file hoặc không lưu file.

Phóng to, thu nhỏ của sổ soạn thảo: Bấm phím chức năng F5.

Chuyển đến cửa sổ soạn thảo bị ẩn đằng sau cửa sổ hiện tại: bấm phím chức năng F6.

Thoát khỏi môi trường C++: Bấm tổ hợp phím Alt X.

Bôi đen vùng mã lệnh: kích, giữ và rê chuột lên vùng cần bôi đen hoặc chuyển con trỏ về đầu vùng cần bôi đen rồi giữ phím Shift trong lúc di chuyển con trỏ qua vùng cần bôi đen.

Sao chép vùng bôi đen: Chọn Edit\ Copy hoặc bấm tổ hợp phím Ctrl + Insert.

Dán vùng mã lệnh đã sao chép: Chọn Edit\ Paste hoặc bấm tổ hợp phím Shift + Insert.

Sao chép nhanh vùng bôi đen: Di chuyển đến vị trí mới và bấm phím K rồi C trong lúc giữ phím Ctrl.

Di chuyển vùng bôi đen: Chuyển đến vị trí mới và bấm phím K rồi V trong lúc giữ phím Ctrl.

Xoá vùng mã lệnh đã bôi đen: Bấm phím K rồi Y trong lúc giữ phím Ctrl.

Bỏ bôi đen: Bấm phím K rồi K trong lúc giữ phím Ctrl hoặc có thể bấm phím K rồi H trong lúc đang giữ phím Ctrl.

Chuyển con trỏ về đầu dòng: Bấm phím Home.

Chuyển con trỏ về cuối dòng: Bấm phím End.

I.2. Cấu trúc một chương trình đơn giản trong C++

Một chương trình đơn giản trong C++ thường được viết trong một cửa sổ soạn thảo (điều này không hoàn toàn đúng trong các chương trình lớn).

Một chương trình bất kỳ trong C++ đều được tạo nên từ rất nhiều hàm (tạm gọi là những đơn vị chương trình), trong đó có một hàm đặc biệt được xem như "chương trình chính" và được gọi là hàm main. Thông thường hàm main được viết ra để gọi (sử dụng) các hàm khác nhằm giải quyết bài toán.

Tuy nhiên, ở đây ta chỉ xét một chương trình đơn giản bao gồm duy nhất một hàm main, ta viết theo cấu trúc sau:

```
1  #include <tên thư viện>
2  void main()
3  {
4  Các lệnh trong thân hàm main
5  }
```

Dòng 1: được gọi là các chỉ thị tiền xử lý. Dòng này có nhiệm vụ khai báo các thư viện sẽ sử dụng trong chương trình.

Trong C++, các lệnh (hàm) thường được đặt trong các thư viện, là các file .h (có thể tìm thấy chúng trong thư mục Include của bộ TC). Nếu trong chương trình muốn sử dụng các lệnh này thì cần phải khai báo sẽ sử dụng các thư viện tương ứng tại đây.

Có rất nhiều thư viện có thể sử dụng nhưng những thư viện hay dùng là: conio.h, stdio.h, iostream.h, math.h, iomanip.h, string.h, ofstream.h, ifstream.h, fstream.h,…

Dòng 2: là từ khoá void main() khai báo bắt đầu hàm main.

Dòng 3 và 5: các dấu '{' và '}' báo hiệu bắt đầu và kết thúc thân hàm main hoặc bắt đầu và kết thúc một khối lệnh.

Dòng 4: là nơi ta đặt các lệnh của chương trình chính.

Ví dụ: chương trình in ra màn hình dòng chữ "Hello wold !"

```
#include "iostream.h"
void main()
{
cout<< "Hello wold !"; getch();
}
```

Để xem trong một thư viện có chứa những hàm nào, trên menu ta chọn Help\ Index rồi gõ tên thư viện và bấm Enter.

Để xem một hàm ta đang sử dụng thuộc vào thư viện nào, ta cũng chọn help\ Index rồi gõ tên hàm và bấm Enter.

Trong C++ có phân biệt chữ hoa và chữ thường. Sau mỗi lệnh đều có dấu ";" để báo kết thúc lệnh.

Đặt lại đường dẫn tới các thư viện: Trong một chương trình ta thường sử dụng các hàm trong các thư viện khác nhau được khai báo tại dòng 1 (như trên). Thông thường, các thư viện đặt trong các thư mục TC\INCLUDE. Môi trường lập trình C++ tự thiết đặt đường dẫn tới các thư viện này. Tuy nhiên, trong trường hợp đường dẫn bị thay đổi, chương trình dịch sẽ không tìm thấy chúng và báo lỗi, khi đó ta cần phải thiết đặt lại:

B1: Trong Menu chính, chọn Option\ Directories.

B2: Trong Include, đặt đường dẫn tới các thư viện có đuôi .h(ví dụ: C:\TC\INCLUDE). Trong Libraries, đặt đường dẫn tới các thư viện đuôi .lib (ví dụ: C:\TC\LIB).

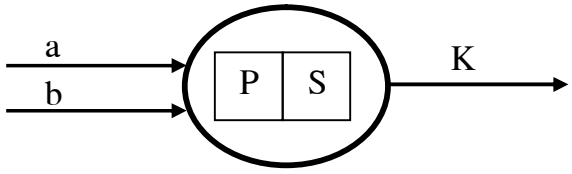
II. BIẾN, BIỂU THỨC, CÁC LỆNH NHẬP XUẤT

II.1. Biến

Để hiểu khái niệm và bản chất biến, ta xét chương trình đơn giản sau:

Nhập vào các số nguyên a, b. Gọi P là tổng của a và b, S là tích của a và b. Tính K=S*P/(S+P).

Ta dễ dàng biểu diễn bài toán bằng mô hình sau:



Có thể coi đầu vào của bài toán là a, b và đầu ra là K. Các giá trị P và S là các giá trị trung gian.

Khi người dùng nhập vào các giá trị a và b, chúng cần được lưu trữ trong bộ nhớ. Tương tự như vậy các giá trị P, S, K nếu cần cũng sẽ được lưu trữ trong bộ nhớ. Vậy tối đa ta cần sử dụng 5 ô nhớ để lưu trữ chúng. Các ô nhớ này gọi là các biến.

Biến là một vùng nhớ được đặt tên.

Mỗi ô nhớ (biến) đều được đặt 1 tên tuân theo quy ước đặt tên như sau:

- Tên biến bao gồm chữ cái, chữ số hoặc dấu gạch nối '_'
- Ký tự đầu tiên của tên biến không được là một chữ số.
- Không được trùng với từ khoá.
- Trong cùng một phạm vi không có 2 biến trùng tên.

Trong chương trình, ta sử dụng tên biến để truy cập vào ô nhớ của biến. Khi đó tên biến chính là giá trị đang chứa trong ô nhớ của nó.

Tất cả các biến khi sử dụng đều phải khai báo. Cú pháp như sau:

<kiểu biến> <tên biến>;

Trong đó:

<kiểu biến>: mỗi biến dùng để chứa một loại giá trị khác nhau như: số nguyên, số thực, ký tự .v.v.., do vậy chúng phải có kiểu tương ứng. Một số kiểu cơ bản như sau:

Kiểu số: bao gồm

+ *Số nguyên int/ short int:* là kiểu dữ liệu có độ dài 2 byte, dùng để khai báo các biến nguyên có giá trị trong khoảng –32768 -> 32767

+ Số nguyên không dấu: unsigned int: độ dài 2 byte, khai báo các biến nguyên có giá trị từ 0 tới 65535.

+ *Số nguyên dài long:* là kiểu dữ liệu có độ dài 4 byte, dùng khai báo các biến nguyên có giá trị trong khoảng – 2.147.483.648 -> 2.147.483.647.

+ *Số nguyên dài không dấu: unsigned long:* độ dài 4 byte, khai báo các biến có giá trị từ 0 tới 4.294.967.295.

+ *Số thực (dấu phảy động) float:* kích thước 4 byte khai báo các biến thực từ $3.4*10^{-38}$ -> $3.4*10^{38}$.

+ *Số thực* (dấu phảy động, độ chính xác kép) *double:* kích thước 8 byte, có phạm vi từ $1.7*10^{-308}$ - > $1.7*10^{308}$

+ *Số thực* (dấu phảy động, độ chính xác kép) *dài long double*: kích thước 10 byte, khai báo các biến từ $3.4 * 10^{-4932}$ tới $1.1 * 10^{4932}$.

Kiểu ký tự: bao gồm

+ *Kiểu ký tự char:* khai báo biến chứa một ký tự.

+ *Kiểu con trỏ ký tự char *:* tương đương với chuỗi ký tự.

<tên biến>: được đặt tuỳ ý tuân theo các quy ước đặt tên biến ở trên.

Ví dụ: `int a, b;  float c;`

Ở đây, ta khai báo 2 biến a và b có cùng kiểu số nguyên và biến c có kiểu số thực. Khi đó chương trình sẽ cấp phát 2 ô nhớ có kích thước 2 byte mỗi ô và đặt tên là a, b; một ô nhớ kích thước 4 byte được đặt tên c.

Vị trí khai báo: Có thể khai báo biến tại bất kỳ đâu trong thân chương trình trước khi sử dụng.

II.2. Biểu thức

Một biểu thức bao gồm 2 thành phần: các toán tử (phép toán) và các toán hạng (số hạng).

Các toán tử:

Trong C++, các toán tử được tạm phân chia làm 4 loại theo chức năng của chúng. Sau đây là một số toán tử hay dùng:

- Các toán tử số học:

Stt	Toán tử	Cách viết
1	Cộng	+
2	Trừ	-
3	Nhân	*
4	Chia	/
5	Đồng dư	%
6	Tăng 1 đơn vị	++
7	Giảm 1 đơn vị	- -

- Các toán tử Logic:

Stt	Toán tử	Cách viết
1	Và	&&
2	Hoặc	||
3	Phủ định	!

- Các toán tử so sánh:

Stt	Toán tử	Cách viết
1	Lớn hơn	>
2	Nhỏ hơn	<
3	Lớn hơn hoặc bằng	>=
4	Nhỏ hơn hoặc bằng	<=
5	Bằng	==
6	Không bằng	!=

- Toán tử gán:

Stt	Toán tử	Cách viết
1	Gán	=

Một bảng tương đối đầy đủ các toán tử trong C++ như sau:

[ ]	( )	.	->	++	--	&
*	+	-	~	!	sizeof	/
%	<<	>>	<	>	<=	>=
==	!=	^	|	&&	||	?:
=	*=	/=	%=	+=	-=	<<=
>>=	&=	^=	|=	,	#	##

Các toán hạng:

Có thể chia các toán hạng làm 3 loại gồm: hằng, biến và hàm.

Hằng: gồm hằng số, hằng xâu ký tự và hằng ký tự. Hằng xâu ký tự khi viết cần được đặt giữa hai dấu nháy kép “”; hằng ký tự được đặt giữa hai dấu nháy đơn ‘’ còn hằng số thì không.

Hàm: gồm những hàm trả về một giá trị nào đó và giá trị này được sử dụng trong biểu thức. Có rất nhiều hàm có sẵn trong các thư viện mà ta có thể sử dụng cho biểu thức. Sau đây là một số hàm toán học (thư viện math.h) thường dùng:

STT	Tên hàm	Cách viết
1.	Sin(x)	sin(x)
2.	Cos(x)	cos(x)
3.	$\sqrt{x}$	sqrt(x)
4.	e^x	exp(x)
5.	Ln(x)	log(x)
6.	$Log_{10}(x)$	log10(x)
7.	|x| (x nguyên)	abs(x)
8.	|x| (x thực)	fabs(x)

Ví dụ: xét biểu thức toán học sau

$$((2e^x + |x|.Ln(x)) > \sqrt{x} / \sin(x)) \wedge (x < 5)$$

Biểu thức được viết dưới dạng ngôn ngữ C++ như sau:

(2*exp(x) + fabs(x)*log(x) > sqrt(x)/ sin(x)) && (x < 5)

Trong biểu thức này, các toán tử số học gồm: +, *; toán tử logic gồm: &&; các toán tử so sánh gồm: >, <; các toán hạng là hằng số gồm: 2, 5; toán hạng là biến gồm: x; các toán hạng là hàm gồm: exp(x), fabs(x), log(x), sqrt(x), sin(x).

II.3. Các lệnh nhập-xuất

a. Các lệnh nhập xuất trong IOStream.h

- Lệnh xuất: **cout<< <Nội dung cần xuất>;**

Trong đó:

<<: được gọi là toán tử xuất.

<Nội dung cần xuất>: có thể là Hằng ký tự, Hằng xâu ký tự, Biến, Hàm hoặc phương thức định dạng.

Ví dụ: cout<<“Sin(x) =”; cout<<sin(x);

Có thể sử dụng liên tiếp nhiều toán tử xuất trên một dòng cout, chẳng hạn: cout<<“Sin(x) =”<<sin(x);

Nếu muốn xuất dữ liệu trên nhiều dòng ta có thể sử dụng toán tử endl để xuống dòng. Ví dụ: cout<<"Sin(x) ="<<endl<<sin(x); sẽ xuất dữ liệu trên 2 dòng.

- **Định dạng dữ liệu trước khi xuất:**

Ta có thể sử dụng một trong 2 cách sau:

Cách 1: sử dụng toán tử định dạng:

cout.width(int n): chỉ định tối thiểu n vị trí dành cho việc xuất dữ liệu.

Nếu giá trị xuất chiếm ít hơn n vị trí thì mặc định là các ký tự trống sẽ chèn vào phía trước. Nếu giá trị xuất chiếm nhiều hơn n vị trí, số vị trí dành cho giá trị xuất đó sẽ được tăng lên sao vừa đủ thể hiện giá trị xuất.

cout.fill(char ch): Chỉ định ký tự ch sẽ được điền vào những vị trí trống (nếu có).

cout.precision(int n): chỉ định độ chính xác của giá trị số khi xuất là n ký tự phần thập phân.

Ví dụ: giả sử ta có biến thực a: float a = 123.4523; Nếu muốn xuất a ra màn hình dưới dạng: 000123.45 ta có thể định dạng như sau:

```
cout.width(9);
cout.fill('0');
cout.precision(2);
cout<<a;
```

Cách 2: sử dụng các hàm định dạng:

Tương tự như các phương thức định dạng, các hàm định dạng tương ứng là:

setw(int n) – tương tự như cout.width(int n).

setfill(char ch) – tương tự như cout.fill(char ch).

setprecision(int n) – tương tự như cout.precision(int n).

Cách dùng: sử dụng các hàm định dạng ngay trên các dòng cout, trước khi đưa ra giá trị xuất. Với ví dụ trên, ta có thể viết:

```
cout<<setw(9)<<setfill('0')<<setprecision(2)<<a;
```

- Lệnh nhập: **cin >> <Biến>;**

Trong đó:

>>: được gọi là toán tử nhập.

Dòng cin dùng để nhập các giá trị (thông thường là) từ bàn phím vào các biến.

Ví dụ: để nhập giá trị cho biến a, ta viết:

cout<< "a= "; cin>>a;

Có thể dùng liên tiếp nhiều toán tử nhập trên một dòng cin để nhập giá trị cho nhiều biến, chẳng hạn: **cin>>a>>b>>c;**

Lệnh cin chỉ thích hợp cho việc nhập các biến kiểu số. Với các biến kiểu xâu ký tự thì xâu nhập vào phải không chứa dấu cách vì lệnh cin sẽ kết thúc khi ta nhập vào dấu cách hoặc phím Enter.

Ví dụ: Viết chương trình nhập vào một số thực x, in ra màn hình giá trị của $F(x) = \sin^2(x) + |x| + e^{\ln(x)}$ với độ chính xác 2 chữ số sau dấu phảy.

```
#include <conio.h>
#include <math.h>
#include <iostream.h>
void main()
{
clrscr();
float x, F;
        cout<<"nhập số thực x ";  cin>>x;
        cout.precision(2);
        cout<<"Giá trị F("<<x<<") =";
        cout<<sin(x)*sin(x) + fabs(x) + exp(log(x));
getch();
}
```

b. Các lệnh nhập xuất trong Stdio.h

- Lệnh xuất: **printf("chuỗi cần xuất" , <Biến 1>, <Biến 2>…);**

Trong đó:

- **"chuỗi cần xuất"** có thể gồm:
- Hằng ký tự, hằng xâu ký tự: Là các ký tự cần in lên màn hình.
- Các đặc tả hay ký tự đại diện, bao gồm:

  %d: đại diện cho biến nguyên.

  %f: đại diện cho biến thực.

  %c: đại diện cho biến kiểu ký tự (mặc định).

  …

- Mỗi biến cần đưa ra màn hình cần có một đặc tả tương ứng tại vị trí muốn đưa ra, như vậy số lượng biến bằng số lượng các đặc tả.

Ví dụ: Cần đưa ra các giá trị của các biến a, b, c kiểu nguyên, ta viết:

printf ("Giá trị của a b c la %d %d %d", a, b, c);

- Lệnh nhập: **scanf("chuỗi các đặc tả", &<Biến 1>, &<Biến 2>…);**

Trong đó, mỗi biến cần phải có một đặc tả tương ứng. Lệnh scanf nhập giá trị vào các biến thông qua địa chỉ của biến. Toán tử & được đặt trước tên biến để lấy địa chỉ của biến.

c. Các lệnh nhập xuất trong Conio.h

- Lệnh xuất: **puts(p);**

Trong đó p là một con trỏ xâu ký tự, tức p có kiểu char* hoặc là một mảng kiểu char. Lệnh puts(p) sẽ đưa các ký tự được con trỏ p trỏ tới lên màn hình.

- Lệnh nhập: **gets(p);**

Trong đó p là một con trỏ xâu ký tự, tức p có kiểu char* hoặc là một mảng kiểu char. Lệnh gets(p) có chức năng cho phép người sử dụng nhập vào một xâu ký tự và chứa xâu vừa nhập vào biến p.

Nếu sử dụng liên tiếp nhiều lệnh gets thì xen giữa các lệnh gets ta cần làm sạch bộ đệm bàn phím bằng lệnh**: fflush(stdin);**

Các lệnh gets, puts thích hợp cho việc nhập xuất các biến kiểu xâu ký tự.

Ví dụ sau đây minh hoạ các sử dụng các lệnh nhập xuất một cách phù hợp cho từng loại biến:

Nhập vào Họ tên, quê quán, số ngày công của một công nhân. In các thông tin vừa nhập lên màn hình kèm theo tiền công, biết rằng mỗi ngày công được trả 50.000Đ.

```
#include "iostream.h"
#include "stdio.h"
#include "conio.h"
void main()
{
char HoTen[30]; char Que[50];
int NgayCong;
        cout<<"Nhap ho ten: "; gets(HoTen);
        fflush(stdin);
        cout<<"Nhap que quan: "; gets(Que);
        cout<<"Nhap ngay cong: "; cin>>NgayCong;
long Luong = (long) NgayCong*50000;
        cout<<"Thong tin vua nhap la: "<<endl;
        cout<<"Ho ten:"<<HoTen<<endl<<"Que:"<<Que<<endl;
        cout<<"Ngay cong:"<<NgayCong<<endl<<"Luong:"<<Luong;
getch();
}
```

Vì NgayCong là biến kiểu int nên khi ta nhân với 50000 sẽ được một con số thuộc kiểu int. Tuy nhiên con số này quá lớn so với dữ liệu kiểu int nên khi gán sang biến `long Luong` ta cần chuyển nó về kiểu long bằng cách viết: `(long) NgayCong*50000;` Cách viết này gọi là ép kiểu.

Để ép kiểu một biểu thức ta viết: **(<kiểu>) <Biểu thức>;**

Đặc biệt trong C++ luôn quy định phép chia một số nguyên cho một số nguyên sẽ thu được thương cũng là một số nguyên. Vì vậy muốn thu được thương là số thực ta cần ép kiểu thương số này.

Ví dụ: nới n nguyên dương, phép chia 1/n sẽ cho ta kết quả là 1 số nguyên (lấy phần nguyên của thương). Để lấy kết quả là số thực ta viết: (float) 1/n.

CHƯƠNG II. CÁC CẤU TRÚC ĐIỀU KHIỂN TRONG C++

I. CẤU TRÚC RẼ NHÁNH VÀ CẤU TRÚC CHỌN

I.1. Cấu trúc rẽ nhánh

Trong thực tế, khi giải quyết một công việc thường ta phải lựa chọn nhiều phương án giải quyết khác nhau. Người ta thường biểu diễn vấn đề này bằng 2 mệnh đề logic sau:

- [1]. **Nếu … thì …;**
- [2]. **Nếu … thì … ngược lại thì…**

Để mô phỏng hai mệnh đề đó, trong ngôn ngữ lập trình C++ đưa ra cấu trúc rẽ nhánh.

Cú pháp:

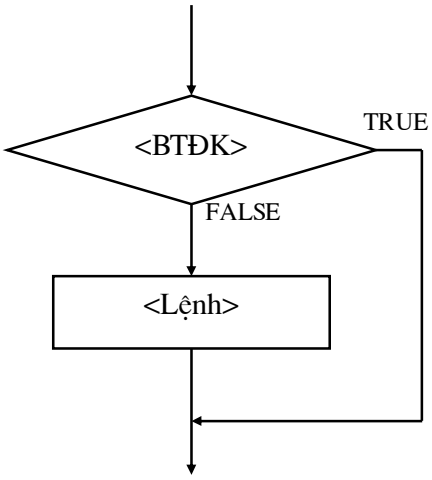
```
if (<biểu thức điều kiện>)  <Lệnh 1>;
[else  <Lệnh 2>;]
```

Ý nghĩa:

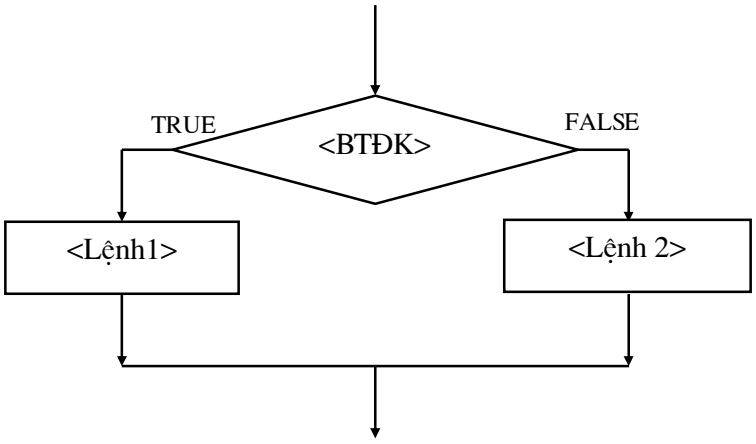
Nếu <biểu thức điều kiện> nhận giá trị đúng (TRUE), sẽ thực hiện <Lệnh1>, ngược lại, nếu <biểu thức điều kiện> nhận giá trị sai (FALSE) sẽ thực hiện <Lệnh2>; (nếu có thành phần **[else <Lệnh 2>;]**).

- <Lệnh 1> và <Lệnh 2> có thể là một lệnh, một khối lệnh hoặc một, một khối các cấu trúc điều khiển. Các khối lệnh hoặc khối cấu trúc điều khiển được đặt trong hai dấu { }.

Cấu trúc rẽ nhánh có hai dạng (tuỳ thuộc vào sự có hay không có thành phần **[else <Lệnh 2>;])** như trong sơ đồ khối dưới đây.



a). Mô tả mệnh đề [1]



b) Mô tả mệnh đề [2]

Ví dụ: Lập chương trình nhập vào một số nguyên. Kiểm tra tính chẵn lẻ của số đó và thông báo ra màn hình.

```
#include <conio.h>
#include <stdio.h>
#include <iostream.h>
void main()
{
clrscr(); int a;
        cout<< “nhập số nguyên a ”;
        cin>>a;
if (a%2 = = 0)
                cout<<”số “<<a<<” chẵn”;
else
                cout<<”số “<<a<<” lẻ”;
getch();
}
```

Các lệnh if có thể lồng nhau theo nghĩa: Các câu lệnh bên trong một mệnh đề if lại có thể là các mệnh đề if.

Mỗi lệnh *if* đầy đủ sẽ cho phép lựa chọn 2 khả năng để thực hiện. Trong trường hợp có n khả năng lựa chọn và các khả năng loại trừ nhau, ta có thể sử dụng n-1 lệnh *if* đầy đủ lồng nhau.

Ví dụ: Viết chương trình thực hiện việc nhập vào số tiền phải trả của khách hàng. Nếu số tiền nhập vào từ 300 tới 400, khuyến mại 20% số tiền phải trả. Nếu số tiền từ 400 trở lên, khuyến mại 30%. Các trường hợp khác không được khuyến mại. Tính và in số tiền khuyến mại của khách lên màn hình.

```
#include <conio.h>
#include <stdio.h>
#include <iostream.h>
void main()
{
clrscr();
int T, km;
        cout<<”Nhập số tiền “; cin>>T;
if (T>=300 && T <=400)     km = T*0.2;
else
                if (T>400) km = T*0.3;
else    km = 0;

cout<<”Số tiền khuyến mại “<<km;
getch();
}
```

Nếu n khả năng là loại trừ nhau thì khi đó có thể sử dụng n-1 lệnh if lồng nhau hoặc có thể sử dụng n lệnh if rời nhau cho n khả năng lựa chọn. Trường hợp ngược lại, ta nên sử dụng các lệnh if rời nhau.

Ví dụ: Viết chương trình nhập vào điểm tổng kết và xếp loại đạo đức của một sinh viên. Sau đó tính số tiền học bổng cho sinh viên đó như sau:

Nếu tổng kết >=7.00 thì được 300000.

Nếu điểm tổng kết >=9.00 và đạo đức = "T" thì được cộng thêm 100000.

Xét đoạn trình sau:

```
#include <conio.h>
#include <stdio.h>
#include <iostream.h>
void main()
{
clrscr();  float tk; char  hk; long T;
        cout<<"Nhap điểm tong ket";    cin>>tk;
        cout<<"Nhap hanh kiem";        cin>>hk;
T=0;
if (tk >= 7)     T = 300000;
else  if (tk>=9 && hk = = 'T')     T += 100000;
        cout<<"Học bổng " <<T;
getch();
}
```

Đoạn trình trên sẽ cho kết quả sai trong trường hợp sinh viên tổng kết >=9.0 và đạo đức tốt. Lý do là sử dụng hai lệnh *if* lồng nhau khi các khả năng không loại trừ nhau.

Đoạn trình trên có thể được viết lại như sau:

```
#include <conio.h>
#include <stdio.h>
#include <iostream.h>
void main()
{
clrscr();  float tk; char  hk; long T;
        cout<<"Nhap điểm tong ket";    cin>>tk;
        cout<<"Nhap hanh kiem";        cin>>hk;
T=0;
if (tk >= 7)     T = 300000;
if (tk>=9 && hk = = 'T')     T += 100000;
        cout<<"Học bổng " <<T;
getch();
}
```

I.2. Cấu trúc chọn

Trong trường hợp có quá nhiều khả năng lựa chọn và các khả năng loại trừ nhau, nếu sử dụng nhiều lệnh *if* lồng nhau sẽ làm cho chương trình phức tạp, khó kiểm soát. Vì vậy C++ cung cấp một cấu trúc điều khiển khác sử dụng trong trường hợp này, đó là cấu trúc chọn.

Cú pháp:

```
switch (<Biến nguyên>)
{
case <GT 1>:  <Lệnh 1;>  break;
case <GT 2>:  <Lệnh 2;>  break;
…
case <GT n>:  <Lệnh n;> break;
[default:        <Lệnh mặc định>;]
}
```

Ý nghĩa: Kiểm tra giá trị của <Biến nguyên>:

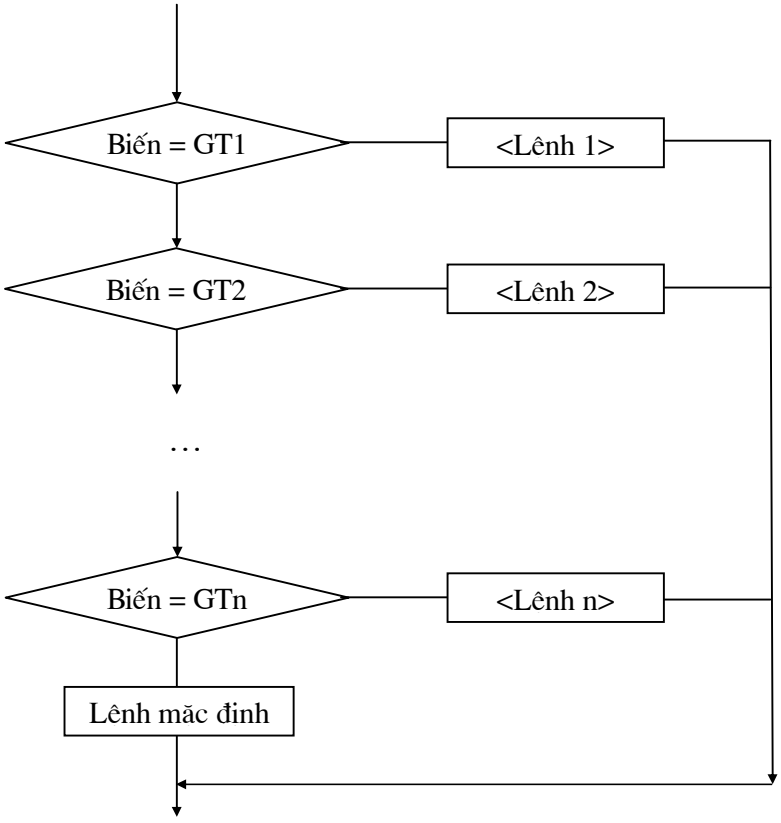
Nếu <Biến nguyên> nhận giá trị <GT1>, thực hiện <Lệnh 1>

Nếu <Biến nguyên> nhận giá trị <GT2>, thực hiện <Lệnh 2>…

Nếu <Biến nguyên> nhận giá trị <GTn>, thực hiện <Lệnh n>

Nếu có thành phần [default: …], thực hiện <lệnh mặc định> khi biến nguyên không nhận giá trị nào trong n giá trị ở trên.

Sơ đồ khối:



- Lệnh ***switch*** chỉ thực hiện trên biến nguyên. Các câu lệnh <Lệnh 1>, <Lệnh 2>… có thể là một khối lệnh hoặc khối các cấu trúc điều khiển (tức nhiều lệnh, nhiều cấu trúc điều khiển đặt giữa hai ký tự { và }). Sau đó phải có lệnh break; nếu không lệnh switch sẽ chạy sai ý nghĩa của nó.

- Thành phần [default: …] là không bắt buộc. Nếu có thành phần này, <Lệnh mặc định> sẽ được được thực hiện sau khi tất cả các trường hợp *case* đều không thỏa mãn.

Ví dụ 1: Viết chương trình nhập vào mã học vị (là một số nguyên) của một nhân viên. In ra học vị tương ứng với quy định:

Mã =1: Cử nhân.
Mã =2: Kỹ sư.
Mã =3: Thạc sỹ.
Mã =4: Tiến sỹ.
Các mã khác: Không xếp loại học vị.

```
void main()
{
clrscr(); int M;
        cout<<"Nhập mã học vị"; cin>>M;
switch (M)
{
case 1:     cout<<"Cử nhân";          break;
case 2:     cout<<"Kỹ sư";            break;
case 3:     cout<<"Thạc sỹ";          break;
case 4:     cout<<"Tiến sỹ";          break;
default:    cout<<"Không xếp loại học vị";
}
getch();
}
```

Ví dụ 2: Viết chương trình nhập vào một tháng của một năm nào đó. In số ngày của tháng đó ra màn hình.

```
#include <conio.h>
#include <stdio.h>
#include <iostream.h>
void main(void)
{
clrscr(); int T, N;
        cout<<"Nhập tháng"; cin>>T;
        cout<<"Nhập năm ";  cin>>N;
switch (T)
{
case 1:
case 3:
case 5:
case 7:
```

```
case 8:
case 10:
case 12:    cout<<"Tháng có 31 ngày";          break;
case 4:
case 6:
case 9:
case 11:    cout<<"Tháng có 30 ngày";   break;
case 2:     if (N%4 = = 0)  cout<<"Tháng có 28 ngày";
            else                cout<<"Tháng có 29 ngày";
            break;
}
getch();
}
```

Chuyển đổi giữa cấu trúc chọn và rẽ nhánh:

Với cấu trúc rẽ nhánh, các biến trong biểu thức điều kiện có thể có kiểu bất kỳ. Ngược lại, với cấu trúc chọn, chỉ lựa chọn các trường hợp của biến nguyên. Do vậy, việc chuyển đổi từ cấu trúc chọn sang cấu trúc rẽ nhánh bao giờ cũng thực hiện được một cách dễ dàng, điều ngược lại không đúng.

Để chuyển đổi một cấu trúc rẽ nhánh mà biểu thức điều kiện có các biến không phải kiểu nguyên sang cấu trúc chọn cần sử dụng thêm một biến nguyên để mã hoá các trường hợp của nó, sau đó ta áp dụng cấu trúc chọn trên biến nguyên này.

II. CẤU TRÚC LẶP

II.1. Vòng lặp với số lần lặp xác định

Giả sử cần thực hiện một vòng lặp với n lần lặp. Trong Pascal ta thường viết: For i:=1 to n do <Lệnh lặp>;

Khi thực hiện lệnh lặp này, máy tính phải thực hiện tuần tự các công việc sau: (1) Gán i:=1; (2) Kiểm tra xem i đã vượt quá n chưa, tức kiểm tra giá trị của biểu thức: i<=n; và (3) Tăng giá trị của i lên 1 đơn vị sau mỗi lần lặp: i:=i+1;

Như vậy, dễ thấy máy tính cần thực hiện 3 biểu thức của vòng lặp: i:=1; i<=n; i:=i+1;. Mỗi vòng lặp, bao giờ cũng phải xác định cho được 3 biểu thức này. Ta tạm gọi chúng là các biểu thức <BT1>, <BT2>, <BT3>.

Trong C++, vòng lặp xác định cũng được xác định bằng 3 biểu thức dạng như trên. Cú pháp như sau:

for (<BT1>; <BT2>; <BT3)

<Lệnh lặp>;

Trong đó, <BT1> thường nhận nhiệm vụ khởi gán giá trị ban đầu cho biến chạy; <BT2> là một biểu thức logic được dùng làm điều kiện dừng cho vòng lặp. Vòng lặp sẽ dừng khi <BT2> nhận giá trị sai (FALSE); <BT3> được dùng để thay đổi giá trị của biến chạy sau mỗi lần lặp.

<Lệnh lặp> có thể là một lệnh, một khối lệnh hoặc một, một khối các cấu trúc điều khiển.

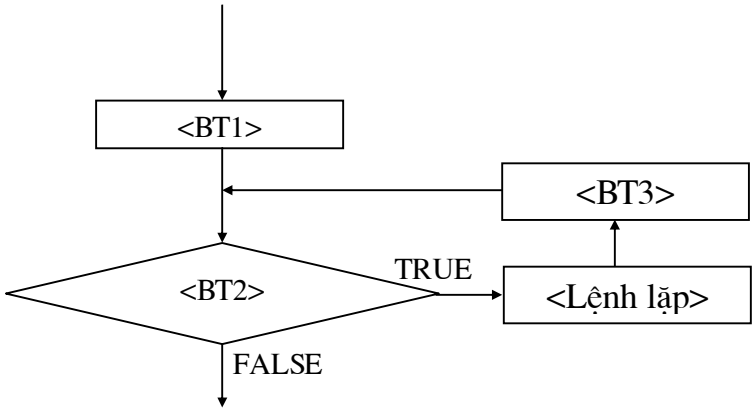
Ý nghĩa:

B1: Thực hiện <BT1>

B2: Kiểm tra <BT2>. Nếu sai, thoát khỏi vòng for. Ngược lại, sang B3.

B3: Thực hiện <Lệnh lặp>, thực hiện <BT3>, quay lại B2.

Sơ đồ khối:



Các trường hợp đặc biệt của vòng lặp for

- **Trường hợp 1:** Các biểu thức <BT1>, <BT2>, <BT3> có thể khuyết nhưng các dấu ";" phải được giữ nguyên.

Ví dụ: Viết chương trình tính n! (n nguyên)

Cách 1: Sử dụng vòng lặp for thông thường

```
void main()
{
clrscr();
int n; long GT=1;
        cout<<"Nhap n "; cin>>n;
for (int i=2;i<=n; i++)GT*=i;
cout<<n<<" Giai thua : "<<GT;
getch();
}
```

Cách 2: Vòng for khuyết <BT1>.

```
void main()
{
clrscr();
int n; long GT=1;
        cout<<"Nhap n "; cin>>n;
        int i=2;
for (;i<=n; i++)        GT*=i;
cout<<n<<" Giai thua : "<<GT;
getch();
}
```

Cách 3: Vòng for khuyết <BT1> và <BT3>

```
void main()
{
clrscr();
int n; long GT=1;
        cout<<"Nhap n "; cin>>n;
        int i=2;
for (;i<=n; )
{
        GT*=i;   i++;
}
cout<<n<<" Giai thua : "<<GT;
getch();
}
```

Cách 4: Vòng for khuyết cả 3 biểu thức. Trong trường hợp này, vòng for không thể dừng một cách tự nhiên được (do thiếu điều kiện dừng là <BT2>). Khi đó ta cần thoát khỏi vòng lặp một cách có chủ định.

Các cách thoát khỏi vòng lặp for khi thiếu <BT2>

+ Cách 1: sử dụng lệnh break;

Khi gặp lệnh break; trong thân vòng for, chương trình sẽ lập tức thoát khỏi vòng lặp for và chuyển tới lệnh tiếp theo bất kể <Biểu thức 2> vẫn nhận giá trị đúng hoặc khuyết <BT2>.

+ Cách 2: sử dụng lệnh goto;

Lệnh goto có dạng: goto <Nhãn>; . Trong đó,<Nhãn> có dạng:

<Tên nhãn> :

<Tên nhãn> tuỳ ý đặt theo quy ước đặt tên trong C.

Khi gặp lệnh goto <Nhãn>;, chương trình sẽ nhảy tới vị trí đặt nhãn. Nếu nhãn đặt ngoài vòng for, chương trình sẽ thoát khỏi vòng for.

Cần chú ý trong trường hợp 2 lệnh for lồng nhau, khi đó lệnh break chỉ làm cho chương trình thoát khỏi vòng for gần nhất chứa lệnh break. Do vậy, để thoát khỏi cả 2 vòng for lồng nhau, ta có thể sử dụng lệnh goto.

Thoát khỏi for sử dụng break:

```
void main()
{
clrscr();
int n; long GT=1;
        cout<<"Nhap n "; cin>>n;
        int i=2;
for (;;)
{
        if (i==n) break;
        GT*=i; i++;
}
cout<<n<<" Giai thua : "<<GT;
getch();
}
```

Thoát khỏi for, sử dụng goto:

```
void main()
{
clrscr();
int n; long GT=1;
        cout<<"Nhap n "; cin>>n;
        int i=2;
for (;;)
{
        if (i==n) goto Ex; // nhãn là Ex, tên tự đặt.
        GT*=i; i++;
}
Ex:
        cout<<n<<" Giai thua : "<<GT;
getch();
}
```

Vì độ "rắc rối" của chương trình (đối với chương trình dịch) tỷ lệ thuận với số lệnh goto sử dụng trong chương trình, vì vậy nên hạn chế sử dụng goto.

- **Trường hợp 2:** Các <BT1>, <BT3> có thể là các biểu thức phức hợp (tức là gồm nhiều biểu thức con). Khi đó, các biểu thức con được đặt cách nhau bởi dấu phẩy.

Ví dụ 2: Cho dãy số nguyên x[] = { 1, 4, 5, 7, 3, 2}. Viết chương trình đảo ngược dãy số trên và in kết quả lên màn hình.

Để giải quyết bài toán trên, có thể có nhiều cách. Cách giải sau minh hoạ cách viết khác của vòng for với <BT1> và <BT3> là các biểu thức phức hợp.

```
#include "conio.h"
#include "iostream.h"
#include "stdio.h"
void main()
{
clrscr();
int x[ ] = {1, 4, 5, 7, 3, 2}, n;
        n=sizeof(x)/ sizeof(int); //n là số phần tử của x (6)
for (int i=0, j=n-1; i<n/2; i++, j--)
{
        int tg = x[i];
        x[i]=x[j];
        x[j]=tg;
}
for (i=0; i<n; i ++)
        cout<<x[i]<<"   ";
getch();
}
```

Thậm chí, có thể viết lại lời giải trên bằng cách sử dụng vòng for với các lệnh thân vòng for được đưa vào<BT3> như sau:

```
void main()
{
clrscr();
int x[ ] = {1, 4, 5, 7, 3, 2}, n;
        n=sizeof(x)/ sizeof(int);
        int tg;
for (int i=0, j=n-1; i<n/2;  tg=x[i], x[i]=x[j], x[j]=tg, i++, j--);

for (i=0; i<n; i ++)
cout<<x[i]<<"   ";
getch();
}
```

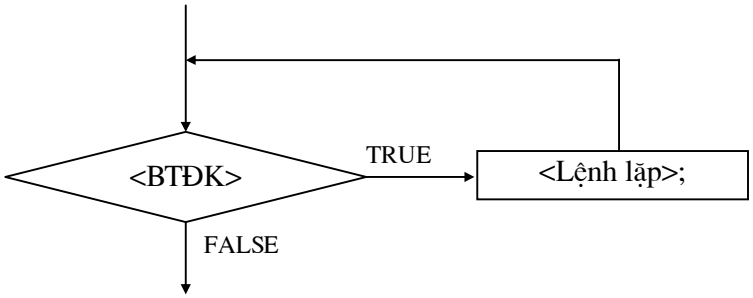
II.2. Vòng lặp với số lần lặp không xác định

Trong C++, ta chia vòng lặp với số lần lặp không xác định ra làm hai loại:

a. Lặp kiểm tra điều kiện trước:

Trước tiên, kiểm tra biểu thức điều kiện. Nếu biểu thức điều kiện còn đúng, sẽ thực hiện lệnh lặp. Nếu biểu thức điều kiện sai, ra khỏi vòng lặp.

Sơ đồ khối:



Như vậy: Lệnh lặp có thể không được thực hiện lần nào nếu <BTĐK> sai ngay từ đầu hoặc cũng có thể lặp vô hạn nếu <BTĐK> luôn đúng.

Cú pháp:

```
while (<BTĐK>)
<Lệnh lặp>;
```

- <Lệnh lặp> có thể là một lệnh, một khối lệnh hoặc một, một khối cấu trúc điều khiển.

Ý nghĩa:

B1: Kiểm tra biểu thức điều kiện <BTĐK>. Nếu biểu thức điều kiện sai, thoát ra khỏi vòng lặp. Nếu biểu thức điều kiện đúng, chuyển qua bước 2.

B2: Thực hiện <Lệnh lặp>. Quay lại B1.

Ví dụ 1. Viết chương trình tìm số lũy thừa 2 đầu tiên lớn hơn 1000.

```
#include <conio.h>
#include <stdio.h>
#include <iostream.h>
void main()
{
clrscr(); int So=2;
        while (So <=1000)  So *=2;
cout<<"Số cần tìm là "<<So;
getch();
}
```

Ví dụ 2. Viết chương trình nhập vào điểm đạo đức của một học sinh (thang điểm 100, nguyên) và số ngày đi học muộn của học sinh đó. Nếu học sinh đi học muộn 1 ngày, trừ 1 điểm. Tính và in số điểm còn lại của học sinh.

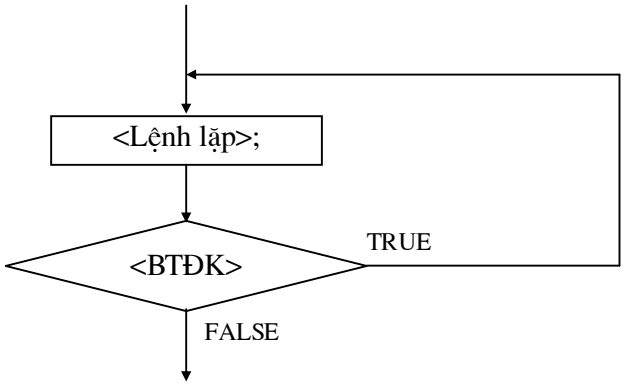
```
#include <conio.h>
#include <stdio.h>
#include <iostream.h>
void main()
{
```

```
clrscr();
int D, HM;
        cout<<"Nhập điểm đạo đức";              cin>>D;
        cout<<"Nhập số ngày đi học muộn";       cin>>HM;
while (HM>0)
{
        D - -;
        HM - -;
}
cout<<"Điểm còn lại "<<D;
getch();
}
```

b. Lặp kiểm tra điều kiện sau:

Tương tự như vòng lặp kiểm tra điều kiện trước, chỉ khác ở chỗ biểu thức điều kiện được kiểm tra mỗi khi đã thực hiện lệnh lặp. Như vậy, lệnh lặp luôn được thực hiện ít nhất một lần.

Sơ đồ khối:



Cú pháp:

```
do
    <Lệnh lặp>;
while (<BTĐK>);
```

- <Lệnh lặp> có thể là một lệnh, một khối lệnh hoặc một, một khối cấu trúc điều khiển.

Ý nghĩa:

B1: Thực hiện <Lệnh lặp>;

B2: Kiểm tra biểu thức điều kiện <BTĐK>. Nếu biểu thức điều kiện sai, thoát ra khỏi vòng lặp. Nếu biểu thức điều kiện đúng, chuyển qua bước 1.

Ví dụ 1: Viết chương trình tìm số nguyên x đầu tiên lớn hơn 5 mà thỏa mãn sin(x) = 1

```
void main()
{
clrscr(); int x=0;
do
        x +=1;
while (x <=5 | | sin(x) !=1);
cout<<"Số cần tìm là "<<x;
getch();
}
```

Ví dụ 2: Viết chương trình nhập vào một số nguyên x. Kiểm tra xem số đó đã lớn hơn 10 hay chưa. Nếu chưa, yêu cầu nhập lại cho tới khi số nhập vào lớn hơn 10. Kiểm tra xem số đó có phải là số nguyên tố không, in kết luận lên màn hình.

```
void main()
{
clrscr(); int So;
do
{
      cout<<"Nhập một số nguyên ";  cin>>So;
      if (So <=10)  cout<<"Số không thỏa mãn. Nhập lại : ";
}
while (So <=10);
//Kiểm tra xem So có phải là số nguyên tố không
int kt=0; int d =2;
do
{
      if (So % d ==0) kt = 1;
      d++;
}
while (d<So);
if (kt==0)    cout<<"Số "<<So<<" Là số nguyên tố";
else          cout<<"Số "<<So<<" Không là số nguyên tố";
getch();
}
```

Để thoát khỏi vòng lặp không xác định, ta cũng có thể sử dụng break hoặc goto. Tuy nhiên, các trường hợp này ít xảy ra do đặc thù của vòng lặp.

Chuyển đổi giữa các cấu trúc lặp:

Từ vòng lặp xác định (for), ta có thể chuyển sang vòng lặp không xác định (while hoặc do/ while) bằng cách sử dụng thêm một biến đếm kiểu nguyên trong thân vòng lặp không xác định. Trước khi lặp ta khởi gán giá trị của biến đếm này bằng 0. Mỗi khi thực hiện một lần lặp, ta tăng giá trị của biến đếm này lên 1. Điều kiện dừng là khi số lần lặp đã đủ.

Để chuyển đổi ngược lại (từ cấu trúc lặp không xác định sang cấu trúc lặp xác định) thì nói chung, ta phải sử dụng các vòng for khuyết biểu thức 2, dưới dạng:

for (<Biểu thức 1> ; ; <biểu thức 3>)

bởi vì ta không biết chắc số lần lặp của cấu trúc. Khi đó ta phải sử dụng lệnh break hoặc goto trong thân vòng for để thoát cưỡng bức.

II.3. Các ví dụ minh hoạ sử dụng vòng lặp

Ví dụ 1. Viết chương trình nhập vào một số nguyên n, sau đó tính tổng các số nguyên tố thuộc đoạn [1..n]. Cho biết có bao nhiêu số nguyên tố thuộc đoạn trên?

```
void main()
{
clrscr(); int n, Tong, Dem;
        cout<<"Nhập số nguyên"; cin>>n;
Tong =Dem=0;
for (int i=1; i<=n; i++)
{
        int Check = 0;
        for (int j=2; j< i; j++)
        if (i%j==0) Check = 1;

        if (Check ==0)
        {
                Tong +=i; Dem++;
        }
}
cout<<"Tổng các số nguyên tố "<<Tong;
cout<<" Có "<<Dem<<số nguyên tố trong đoạn 1..n";
getch();
}
```

Ví dụ 2. Viết chương trình nhập vào một số nguyên n và một số thực x, sau đó tính giá trị biểu thức:

$$F = \begin{cases} x + \dfrac{x^2}{3} + \dfrac{x^3}{3^2} + .. + \dfrac{x^n}{3^{n-1}} & \text{Nếu n chẵn} \\ 0 & \text{Nếu n lẻ} \end{cases}$$

```
void main()
{
clrscr();
int n; float x, F, ts, ms;
        cout<<"Nhập x"; cin>>x;
        cout<<"Nhập n"; cin>>n;
F=0;
if (n%2==0)
{       F=x; ts=x; ms=1;
        for(int i=1; i<=n; i++)
                {
                        ts*=x;
                        ms*=3;
                        F+=ts/ms;
                }
}
cout<<"Giá trị của biểu thức "<<F;
getch();
}
```

CHƯƠNG III. KỸ THUẬT LẬP TRÌNH ĐƠN THỂ

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 là đơn thể. Các đơn thể thực hiện những nhiệm vụ nhất định và được sử dụng trong chương trình thông qua những lời gọi đơn thể 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 thể khắc phục được hai nhược điểm cơ bản trên. Đặc biệt phù hợp với các chương trình có kích thước lớn và phức tạp.

Phân loại đơn thể: Trong C++, đơn thể có thể là:

[1]. Các lớp đối tượng: Chương trình bao gồm một số đoạn mã mô tả các lớp các đối tượng nào đó sẽ sử dụng trong chương trình chính. Loại đơn thể này được nghiên cứu trong nội dung môn học "Lập trình hướng đối tượng".

[2]. 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.

Trong phạm vi môn học, ta chỉ xem xét các đơn thể dưới dạng các hàm.

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 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 đối của hàm:** Nếu hàm sử dụng các đối (các giá trị đầu vào) thì các đối phải thuộc một kiểu dữ liệu nào đó. Khi thiết lập một hàm, ta cần chỉ ra danh sách các đối của hàm và kiểu dữ liệu của mỗi đối.

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

Phân loại hàm

Trong pascal, ta có hai loại chương trình con: thủ tục (procedure) và hàm (function). Trong C++, chúng ta có duy nhất một loại "chương trình con" (mà ta gọi là đơn thể), 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.

Nếu trong Pascal, tất cả các hàm đều trả về một giá trị nào đó thì trong C++ 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. Nó tương đương với thủ tục trong Pascal.

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

Ví dụ 1: Hàm tính n! đơn giản được viết như sau:

```
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. Nếu không có lệnh return này, chương trình sẽ báo lỗi.

- 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 đối vào là hai hệ số a, b.

```
void PTBN(float a, float b)
{
if (a==0 && b==0)
        cout<<"Phương trình vô số nghiệm";
else
        if (a==0 && b!=0)
                cout<<"phương trình vô nghiệm";
        else
                cout<<"Phương trình có nghiệm "<<-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 trước.

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

Cách viết một lời gọi hàm như sau:

<Tên hàm> <([danh sách các tham số thực sự])>

Như vậy:

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

- Nếu hàm không có đối 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.

- 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<< "Kết quả:"<<kq;
}
```

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

Ở hàm thứ nhất, 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>
…
void 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 đối] [Tên đối]);

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:

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

```
{
int a; cout<<"Nhập a"; cin>>a;
if (NT(a) == 0)
cout<<"Số "<<a<<" Không phải nguyên tố";
else
        {
                cout<<"Số "<<a<<" là số nguyên tố";
                cout<<" Giai thừa của "<<a<<" là "<<GT(a);
        }
getch();
}
```

Hai hàm đặt sau hàm main:

```
//Khai báo nguyên mẫu của hàm:
int NT(int n);
long GT(int n);
//hàm main----------------------------
void main()
{
int a; cout<<"Nhập a"; cin>>a;
if (NT(a) == 0)
cout<<"Số "<<a<<" Không phải nguyên tố";
else
        {
                cout<<"Số "<<a<<" là số nguyên tố";
                cout<<" Giai thừa của "<<a<<" là "<<GT(a);
        }
getch();
}

int NT(int n)
{
if (n ==1 | | n ==2)
        return =1;
else
        {Check =0;
                for (int i=2; i<n; i++)
                        if (n%i==0) Check =1;
        if (Check == 0)     return 1;
        else                return 0;
        }
}
//=================
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:

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">

Nếu đặt thư viện trên trong thư mục TC\ Include thì trong chỉ thị #include không cần thêm đường dẫn. Ngược lại, cần thêm đầy đủ đường dẫn tới file thư viện nói trên.

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

```
int NT(int n)
{
if (n ==1 | | n ==2)
        return =1;
else
        {Check =0;
                for (int i=2; i<n; i++)
                        if (n%i==0)
                                Check =1;
        if (Check == 0)
                return 1;
        else
                return 0;
        }
}
//=================

long GT(int n)
{
long kq=1;
if (n==0 | | n==1)
kq=1;
else
for (int i=1; i<=n; i++)
kq *=i;
return kq;
}
```

Giả sử file TV.h này được đặt trong thư mục C:\TC\BIN. Ta mở một file mới và viết hàm main() như sau:

```
#include <conio.h>
#include <stdio.h>
#include <iostream.h>
#include "C:\TC\BIN\ TV.h"
void main()
{
        int a; cout<<"Nhập a"; cin>>a;
if (NT(a) == 0)
        cout<<"Số "<<a<<" Không phải nguyên tố";
else
        {
                cout<<"Số "<<a<<" là số nguyên tố";
                cout<<" Giai thừa của "<<a<<" là "<<GT(a);
        }
getch();
}
```

- 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.

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:** (private) 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. 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:

```
int x;
void Ham(int a)
```

```
{
        cout<<"Biến x trong hàm "<<x;
        if (a%2==0) {
                int x=5;
                x+=a;
                cout<<"Biến x trong hàm "<< x;}
}
void main()
{
        x=1;
        int a = 2;
Ham(a);
        cout<< "Biến x trong hàm main "<<x;
int x = 3;
        cout<<"Biến x trong hàm main "<<x;
getch();
}
```

- 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:

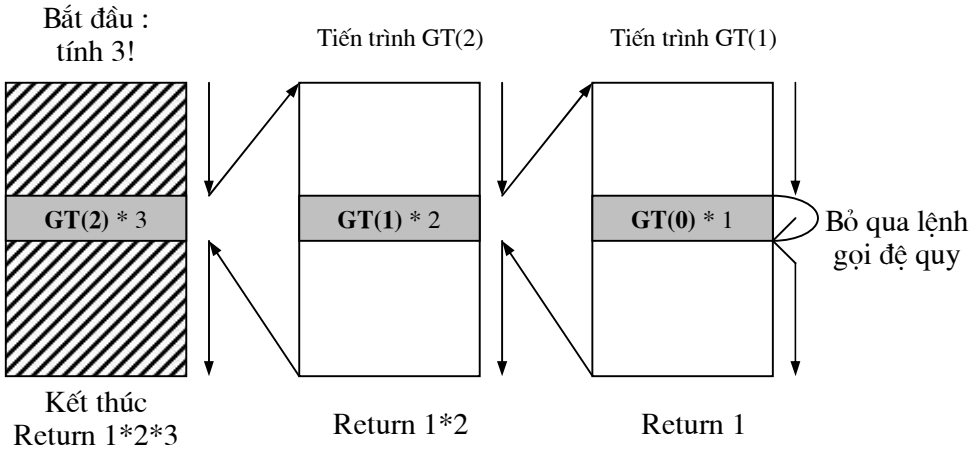
5! = 4! * 5
= 3! * 4 * 5.
= 2! * 3 * 4 * 5.
= 1! * 2 * 3 * 4 * 5.

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.

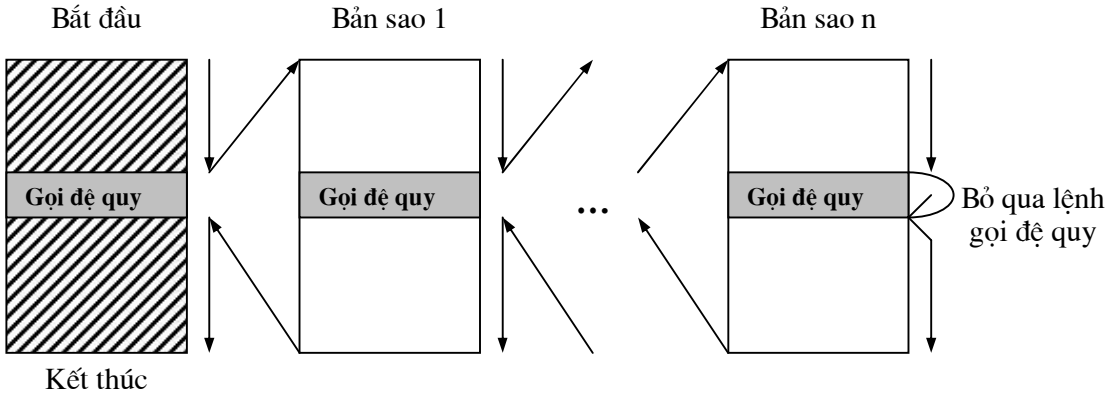
Như vậy, khi gặp một lời gọi đệ quy, máy tính sẽ:

- 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!.

Bước 1:
- Suy biến : n=0 hoặc n = 1; Công thức n! = 1;
- Tổng quát: n khác 0 và n khác 1; Công thức n! = n* (n-1)!.

Bước 2:

```
if  (n= = 0 || 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:

$C_1 = 1;$

$C_n = \sum_{i=1}^{n-1} C_i C_{n-i}$

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

Bước 2:

```
if (n = = 1)
        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 được viết như sau (bước 3):

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

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. 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, thông thường các giá trị đầu vào được định nghĩa một cách hình thức (giả định) và chúng được gọi là các đối số (hay tham số hình thức).

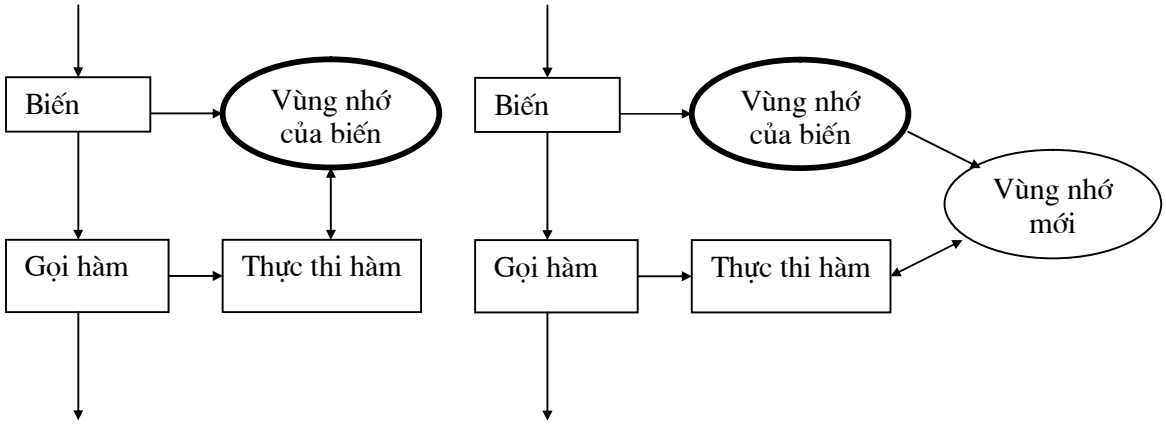
Khi sử dụng hàm, nếu hàm có đối số (tham số hình thức), khi gọi hàm ta phải truyền các tham số (đối số thực sự) tương ứng cho hàm. Các tham số là các giá trị cụ thể và tương ứng về kiểu với các đối số của hàm, chúng có thể là các biến hoặc các hằng giá trị. Ở đây ta chỉ xét các tham số là biến.

Tham số là biến được chia làm 2 loại:

[1]. Tham trị: Là các biến thông thường được truyền vào hàm. Khi truyền tham số dưới dạng tham trị, tham số sẽ không được truy cập trực tiếp.

Hàm sẽ cấp phát một vùng nhớ mới và sao chép giá trị của tham số vào đó. Các lệnh trong thân hàm sẽ thao tác trên vùng nhớ mới này. Như vậy, một tham số khi truyền vào một hàm sẽ không bị thay đổi giá trị của nó khi ra khỏi hàm.

[2]. Tham chiếu: Là *địa chỉ của các biến thông thường* hoặc *các biến con trỏ* (vì bản thân con trỏ đang chứa địa chỉ của các biến thường). Khi truyền tham số dưới dạng tham chiếu, tham số là các biến và tham số sẽ được truy cập trực tiếp. Như vậy, các một tham số khi truyền vào một hàm có thể bị biến đổi giá trị của nó.



a) truyền tham chiếu *b) Truyền tham trị*

III.2. Truyền tham số

Khi ta truyền một biến thông thường vào hàm tức là ta đã truyền dưới dạng tham trị. Hàm sẽ cấp phát vùng nhớ mới và sao chép giá trị của biến vào ô nhớ này để sử dụng. Như vậy, ra khỏi thân hàm, ô nhớ mới được cấp phát bị xóa ngay và giá trị của biến không hề thay đổi.

Ví dụ 1. Xét hàm sau:

```
int tang(int a)
{
        a++;
}
void main()
{
        int n=1;
        cout<<"Giá trị trước khi gọi hàm "<<n;
tang(n);
        cout<<"Giá trị sau khi gọi hàm "<<n;
getch();
}
```

Biến n là một biến thông thường và đang mang một giá trị cụ thể (n=1;), được truyền vào hàm dưới dạng tham trị nên sau khi ra khỏi hàm, giá trị của nó không hề thay đổi (vẫn là 1) mặc dù trong thân hàm int tang(int a) thì giá trị của đối số bị thay đổi (a++).

Ví dụ 2. Xét hàm sau

```
int Ham(int a, int b)
{
        a+=1;
        b+=a;
cout<<"Giá trị a trong thân hàm "<<a;
cout<<Giá trị b trong thân hàm "<<b;
}
void main()
{
int a, b;
a=1;
b=2;
cout<<"Giá trị a trước khi gọi hàm "<<a;
cout<<"Giá trị b trước khi gọi hàm "<<b;
Ham(a, b);
cout<<"Giá trị a sau khi gọi hàm "<<a;
cout<<"Giá trị b sau khi gọi hàm "<<b;
getch();
}
```

Vì a, b được truyền vào hàm dưới dạng tham trị nên mặc dù trong thân hàm các giá trị này đã bị thay đổi nhưng khi ra khỏi hàm nó lại giữ nguyên giá trị ban đầu. Nguyên nhân là do trong thân hàm, chỉ thay đổi giá trị trên các bản sao của biến truyền vào.

Nếu ta chỉ truyền địa chỉ của biến vào hàm thì việc truyền như vậy gọi là truyền tham chiếu. Khi đó hàm sẽ tham chiếu trực tiếp tới biến và thao tác trên vùng nhớ của biến truyền vào. Kết quả là giá trị của biến có thể bị thay đổi do tác động của hàm.

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

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

Dễ thấy khi gọi hàm ta chỉ truyền địa chỉ của n vào hàm (tang**(&n)**). Do vậy hàm int tang(int *a) sẽ sử dụng biến n (cho đối số *a) để thao tác. Kết quả sau khi ra khỏi hàm, biến n bị thay đổi giá trị (tăng lên 1 đơn vị).

Như vậy, nếu muốn truyền tham chiếu thì đối số tương ứng của hàm được định nghĩa trước đó phải là con trỏ.

Nếu trong hàm main, biến n là con trỏ thì bản thân con trỏ đã chứa địa chỉ của biến khác nên khi truyền n vào đã là truyền tham chiếu. Đây là điều dễ gây nhầm lẫn cần phải được chú ý.

Ví dụ: với hàm int tang(int *a) như trên, ta có hàm main sau:

```
void main()
{
        int *n; int a=1;
        n=&a; // n đang trỏ tới a – chứa địa chỉ của a
        cout<<"Giá trị trước khi gọi hàm "<<(*n);
tang(n); // truyền tham chiếu
        cout<<"Giá trị sau khi gọi hàm "<<(*n);
getch();
}
```

Trong lời gọi **tang(n);** ta truyền tham số n vào dưới dạng tham chiếu bởi vì n đang chứa địa chỉ của biến a.

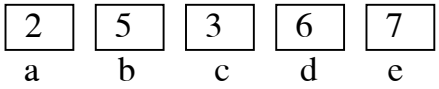
CHƯƠNG IV. KỸ THUẬT LẬP TRÌNH DÙNG MẢNG

I. MẢNG MỘT CHIỀU

I.1. Khai niệm và cách khai báo

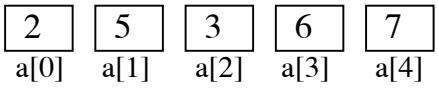
Bài toán: hãy lưu trữ một dãy số gồm 5 phần tử: {2, 5, 3, 6, 7}

Cách 1: Sử dụng 5 ô nhớ (5 biến) cùng kiểu. Các biến được đặt tên lần lượt là: a, b, c, d, e. Khi đó, các phần tử được chứa trong 5 ô nhớ này như sau:



Vì cần lưu trữ 5 giá trị khác nhau nên việc dùng 5 ô nhớ khác nhau là cần thiết. Tuy nhiên, phương pháp này tỏ không khả thi do sử dụng quá nhiều tên biến, dẫn tới khó kiểm soát các biến, đặc biệt trong trường hợp số phần tử của dãy quá lớn.

Cách 2: Vẫn sử dụng 5 ô nhớ cùng kiểu nhưng tất cả các ô được đặt chung một tên (a chẳng hạn). Để phân biệt các ô với nhau, người ta đánh chỉ số cho từng ô. Chỉ số là các số nguyên liên tiếp, tính từ 0. Như vậy ta thu được:



Kết quả ta có được một cấu trúc dữ liệu khắc phục được nhược điểm của cách 1. Cấu trúc dữ liệu này gọi là mảng một chiều.

Mảng là một cấu trúc bộ nhớ bao gồm một dãy liên tiếp các ô nhớ cùng tên, cùng kiểu nhưng khác nhau về chỉ số, dùng để lưu trữ một dãy các phần tử cùng kiểu.

Cú pháp khai báo mảng:

<Kiểu mảng> <Tên mảng> [<Số phần tử tối đa>];

Trong đó:

<Kiểu mảng>: Là kiểu dữ liệu của mỗi phần tử trong mảng, có thể là một kiểu dữ liệu chuẩn hoặc kiểu dữ liệu tự định nghĩa.

<Tên mảng>: Được đặt tuỳ ý tuân theo quy ước đặt tên biến trong C++.

<Số phần tử tối đa>: Là một hằng số chỉ ra số ô nhớ tối đa được dành cho mảng cũng như số phần tử tối đa mà mảng có thể chứa được.

Ví dụ: Khai báo int a[3]; sẽ cấp phát 3 ô nhớ liên tiếp cùng kiểu nguyên (2 byte) dành cho mảng a. Mảng này có thể chứa được tối đa 3 số nguyên.

I.2. Các thao tác cơ bản trên mảng một chiều

- Nhập mảng: Giả sử ta cần nhập mảng a gồm n phần tử. Cách duy nhất là nhập từng phần tử cho mảng. Do vậy, ta cần sử dụng một vòng lặp for duyệt qua từng phần tử và nhập dữ liệu cho chúng. Nhưng trước tiên, cần nhập số phần tử thực tế của mảng (n).

```
for(int i=0; i<n; i++)
{
    cout<< "a["<<i<< "]=";
    cin>>a[i];
}
```

- Xuất mảng: tương tự như nhập mảng, ta cũng cần sử dụng vòng lặp for để xuất từng phần tử của mảng lên màn hình.

```
for(i = 0; i<n; i++)
    cout<<a[i];
    cout<<a[i]<< "  ";
    cout<<a[i]<<endl;
```

- **Duyệt mảng**: là thao tác thăm lần lượt từng phần tử của mảng. Thao tác duyệt mảng có trong hầu hết các bài toán sử dụng mảng.

```
for(i = 0; i<n; i++)
{thăm phần tử a[i]}
```

I.3. Các bài toán cơ bản

a. Bài toán sắp xếp mảng

Bài toán: cho một dãy gồm n phần tử. Hãy sắp xếp dãy theo chiều tăng dần.

Để giải quyết bài toán này, trước tiên ta cần lưu trữ dãy các phần tử đã cho vào bộ nhớ, như vậy ta cần sử dụng một mảng một chiều. Sau đó, có rất nhiều phương pháp để sắp một mảng theo chiều tăng dần. Sau đây ta xem xét một số cách phổ biến:

- **Phương pháp 1:** Sắp xếp nổi bọt – bubble sort

Ý tưởng của phương pháp như sau:

- Sắp lần lượt từng phần tử của dãy, bắt đầy từ phần tử đầu tiên.

- Giả sử cần sắp phần tử thứ i, ta tiến hành duyệt lần lượt qua tất cả các phần tử đứng sau nó trong dãy, nếu gặp phần tử nào nhỏ hơn phần tử đang sắp thì đổi chỗ.

Giả sử ta sắp mảng a gồm n phần tử, giải thuật được mô tả chi tiết như sau:

```
for(i = 0; i < n; i++)// với mỗi phần tử a[i]
for(j = i+1; j<n; j++)
   if(a[j]  < a[i])    Đổi chỗ a[i] cho a[j]
```

Để đổi chỗ a[i] cho a[j], ta sử dụng một biến tg có cùng kiểu và gán một trong 2 giá trị (a[i] hoặc a[j]) vào đó. Sau đó gán giá trị còn lại sang giá ô nhớ vừa gán vào tg. Cuối cùng ta gán giá trị đang chứa trong tg vào ô nhớ này.

```
for(i = 0; i < n; i++)
for(j = i+1; j<n; j++)
if(a[j]  < a[i])
{
        tg = a[i];
        a[i] = a[j];
        a[j] = tg;
}
```

Sắp xếp bằng phương pháp này trung bình cần $n^2/2$ lần so sánh và $n^2/2$ lần hoán vị. Trong trường hợp tồi nhất ta cũng cần số lần so sánh và hoán vị như vậy.

Giả sử ta có mảng a = {3, 5, 2, 7, 4, 8}. Hình ảnh của a sau các lần lặp sắp xếp nổi bọt như sau:

Bắt đầu sắp i = 0	3	5	2	7	4	8
Hết 1 vòng lặp j; i = 1;	2	5	3	7	4	8
Hết một vòng j; i=2;	2	3	5	7	4	8
Hết một vòng j; i=3;	2	3	4	7	5	8
Hết một vòng j; i=4;	2	3	4	5	7	8

- **Phương pháp sắp xếp chọn** – Selection sort

Trong phương pháp sắp sếp nổi bọt, để sắp một phần tử nhiều khi ta phải đổi chỗ nhiều lần phần tử đang sắp với các phần tử đứng sau nó. Một ý tưởng rất hay là làm sao chỉ đổi chỗ 1 lần duy nhất khi sắp một phần tử trong dãy. Đây chính là ý tưởng của phương pháp sắp xếp chọn.

Để làm được điều này, khi sắp phần tử thứ i, người ta tiến hành tìm phần tử nhỏ nhất trong số các phần tử đứng sau nó kể cả phần tử đang sắp rồi tiến hành đổi chỗ phần tử nhỏ nhất tìm được với phần tử đang sắp.

Ví dụ: Với dãy a = {1, 6, 4, 2, 5, 7}, để sắp phần tử thứ 2 (6) người ta tiến hành tìm phần tử nhỏ nhất trong số các phần tử {6, 4, 2, 5, 7}. Khi đó Min(6, 4, 2, 5, 7) = 2 và phần tử 2 được đảo chỗ cho phần tử 6. Kết quả thu được:

1	2	4	6	5	7

Trước tiên ta xem xét bài toán tìm phần tử nhỏ nhất của một dãy các phần tử:

- Lấy một phần tử ngẫu nhiên trong dãy làm phần tử nhỏ nhất (Min) (thường lấy phần tử đầu tiên).

- Duyệt qua tất cả các phần tử của dãy, nếu gặp phần tử nào nhỏ hơn Min thì gán Min bằng phần tử đó.

Ví dụ: Tìm số nhỏ nhất trong mảng a gồm n phần tử:

```
Min = a[0];
for(i = 0; i<n; i++)
if (a[i] < Min) Min = a[i];
```

Khi kết thúc vòng lặp, ta thu được giá trị nhỏ nhất của dãy đang chứa trong biến Min.

Tuy nhiên, áp dụng giải thuật này vào phương pháp sắp xếp chọn ta cần phải lưu ý một số điểm. Chẳng hạn ta cần biết chính xác vị trí của phần tử Min nằm ở đâu để tiến hành đổi chỗ chứ không quan tâm tới giá trị Min là bao nhiêu. Tuy nhiên giải thuật tìm Min lại chỉ cho biết giá trị Min mà không cho biết vị trí. Nếu muốn biết, ta lại phải sử dụng 1 vòng lặp duyệt lại từ đầu để tìm vị trí Min. Do vậy, khi cài đặt phương pháp sắp xếp chọn, để tránh trường hợp sử dụng nhiều vòng lặp sẽ làm tăng độ phức tạp của giải thuật, ta chỉ chú ý tới việc tìm vị trí phần tử Min.

- Sắp xếp từng phần tử của dãy, bắt đầu từ phần tử đầu tiên.

- Giả sử sắp phần tử a[i], ta gán Min = i rồi duyệt qua các phần tử đứng sau nó (i+1 trở đi). Nếu phần tử a[j] nào nhỏ hơn phần tử a[Min] thì gán Min bằng vị trí của a[j] (tức Min=j). Cuối cùng ta đổi chỗ a[i] cho a[Min].

```
for(i = 0; i < n; i++)
{   Min = i;
    for(j = i+1; j<n; j++)
    if(a[j]  < a[Min]) Min = j;
        tg = a[i];
        a[i] = a[Min];
        a[Min] = tg;
}
```

Sắp xếp bằng phương pháp chọn cần $n^2/2$ lần so sánh và n lần hoán vị.

Giả sử ta có mảng a = {3, 5, 2, 7, 4, 8}. Hình ảnh của a qua các lần lặp sắp xếp chọn như sau:

3	5	2	7	4	8	Min = 2
2	5	3	7	4	8	Min = 3
2	3	5	7	4	8	Min = 4
2	3	4	7	5	8	Min = 5
2	3	4	5	7	8	Min = 7
2	3	4	5	7	8	Min = 8

Sau đây là hàm sắp xếp chọn với đối vào là mảng a gồm n phần tử:

```
void SapChon(int a[100], int n)
{
for (int i=0; i<n; i++)
        {
        int min = i;
        for (int j = i+1; j<n; j++)
                if (a[j] < a[min]) min = j;
        int tg = a[i];
        a[i] = a[min];
        a[min]=tg;
        }
}
```

- **Phương pháp sắp xếp chèn**

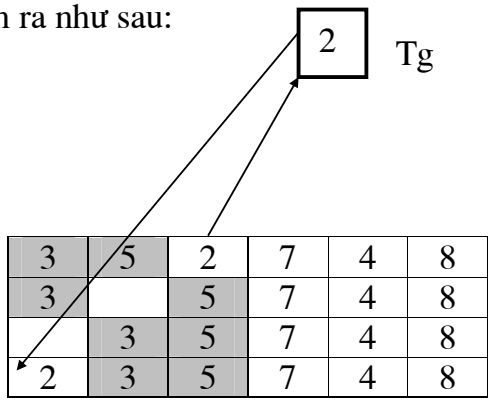
Một thuật toán gần như đơn giản ngang với thuật toán sắp xếp chọn nhưng có lẽ mềm dẻo hơn, đó là sắp xếp chèn. Đây là phương pháp người ta dùng để sắp xếp các thanh ngang cho một chiếc thang.

Đầu tiên, người ta rút ngẫu nhiên 1 thanh ngang và đặt vào 2 thanh dọc để làm thang. Tiếp đó, người ta lần lượt chèn từng thanh ngang vào sao cho không phá vỡ tính được sắp của các thanh đã được đặt trên 2 thanh dọc.

Giả sử với mảng a = {3, 5, 2, 7, 4, 8}. Giả sử các phần tử 3 và 5 đã được chèn vào đúng vị trí (đã được sắp):

3	**5**

Ta xem xét quá trình chèn phần tử tiếp theo vào mảng (giả sử chèn 2 vào). Khi đó, quá trình diễn ra như sau:



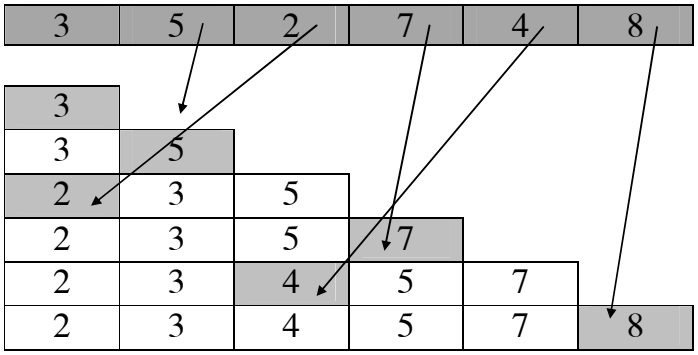
- Đặt phần tử 2 vào biến tg.

- Duyệt qua các phần tử đứng trước phần tử 2 (các phần tử đã được sắp). Nếu gặp phần tử nào lớn hơn 2 thì đẩy phần tử đó sang phải 1 vị trí. Ngược lại, nếu gặp phần tử nhỏ hơn 2 thì chèn 2 vào ngay sau phần tử nhỏ hơn này. Nếu đã duyệt hết các phần tử đứng trước mà vẫn chưa tìm thấy phần tử nhỏ hơn 2 thì chèn 2 vào đầu mảng.

Kết thúc quá trình này, phần tử 2 đã được chèn đúng vị trí và 3 phần tử đã được sắp là:

2	**3**	**5**

Toàn bộ quá trình sắp mảng a được biểu diễn trong bảng sau:



Như vậy trong quá trình thực hiện, để chèn 1 phần tử vào đúng vị trí của nó, ta luôn thực hiện 2 công việc: *Đẩy một phần tử sang phải 1 vị trí* hoặc *chèn phần tử cần chèn vào vị trí của nó*. Nếu gọi phần tử cần chèn là a[i] đang chứa trong biến tg và j là biến duyệt qua các phần tử đứng trước a[i] thì:

- Khi chưa hết mảng và gặp một phần tử lớn hơn phần tử cần chèn thì *đẩy nó sang phải 1 vị trí*: **while (j >= 0 && a[j] > tg) a[j+1] = a[j];**

- Ngược lại thì *chèn tg vào sau j*: **a[j+1] = tg;**

```
void SapChen(int a[100], int n)
{
for (int i=1; i< n; i++)
        {
        int Tg = a[i]; int j = i-1;
        while (j > = 0 && a[j] > Tg)
          {      a[j+1] = a[j];
                 j--;
           }
        a[j+1]=Tg;
        }
}
```

Sắp xếp bằng phương pháp chèn trong trường hợp trung bình cần $n^2/4$ lần so sánh và $n^2/8$ lần hoán vị. Trường hợp tồi nhất gấp đôi số lần này.

- **Phương pháp sắp xếp trộn: Merge sort**

Bài toán trộn: Cho mảng a gồm n phần tử và mảng b gồm m phần tử đã sắp tăng. Hãy trộn hai mảng để thu được một mảng thứ 3 cũng được sắp tăng.

Trước tiên, ta xét hai mảng a và b như ví dụ như sau:

a	2	3	5

b	1	4	6	7	9	10

Mảng c sau thu được sau khi trộn a và b là:

c	1	2	3	4	5	6	7	9	10

Để có được mảng c, ta làm như sau:

B1. Cho biến i xuất phát từ đầu mảng a (i=0) và biến j xuất phát từ đầu mảng b (j=0).

B2. Ta so sánh a[i] và a[j] rồi lấy phần tử nhỏ hơn trong hai phần tử đó đặt vào mảng c.

Nếu lấy a[i], ta phải tăng i lên 1 đơn vị (i++) và tương tự, nếu lấy b[j], ta tăng j lên 1 đơn vị (j++). Lặp lại B2 cho tới khi 1 trong 2 mảng đã được lấy hết.

Với mảng a,b ở trên, dễ thấy giải thuật trên sẽ dừng lại khi mảng a đã được lấy hết. Tuy nhiên, khi đó, mảng b vẫn còn các phần tử 6, 7, 9, 10 chưa được lấy. Công việc tiếp theo là chuyển toàn bộ các phần tử "còn thừa" này từ b sang c.

Hàm sau thực hiện trộn 2 mảng a, b theo thuật toán trên.

```
int c[100];
void Tron(int a[50],int n, int b[50], int m)
{
int i=0, j=0, k=0;
while(i<n&&j<m)
if(a[i]<b[j])
   {c[k]=a[i]; i++; k++;}
else
   {c[k]=b[j]; j++; k++;}
// Gắn đuôi----------------------------
if(i<n)
  for(int t=i; t<n; t++)  {c[k]=a[t]; k++;}
else
  for(int t=j; t<m; t++)  {c[k]=b[t]; k++;}
}
```

Dễ thấy quá trình cho i chạy trên a và j chạy trên b sẽ buộc phải dừng lại khi 1 trong 2 mảng đã được duyệt hết. Nếu không, biến i hoặc j sẽ chạy quá giới hạn của mảng a hoặc b. Khi dừng lại, một trong 2 mảng có thể chưa được duyệt

hết và còn thừa một số phần tử và quá trình chuyển các phần tử này sang c được diễn ra.

Một cải tiến nhỏ giúp cho i và j không bao giờ chạy vượt quá giới hạn của 2 mảng a và b cho tới khi ta lấy xong tất cả các phần tử của a và b đặt sang c. Muốn vậy, trước khi thực hiện giải thuật trên, ta làm như sau:

- Gọi Max là phần tử lớn nhất trong số các phần tử của cả a và b.
- Chèn giá trị Max + 1 vào vị trí cuối cùng của mảng a và mảng b.

Với mảng trên, giá trị Max = 10 và hai mảng sau khi chèn Max+1 vào cuối sẽ có dạng:

a	2	3	5	11

b	1	4	6	7	9	10	11

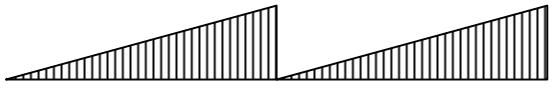
Khi i hoặc j chạy tới cuối mảng, ta gặp phải giá trị 11 là giá trị lớn hơn bất kỳ giá trị mào của hai mảng a và b ban đầu, do đó, theo giải thuật thì i và j sẽ bị dừng lại tại đó và không thể chạy vượt ra khỏi giới hạn của mảng.

```
int c[100];
void Tron2(int a[50],int n, int b[50], int m)
{
int Max=a[n-1];
if(Max<b[m-1]) Max=b[m-1];
a[n]=b[m]=Max+1;
//------------------------
int i=0, j=0;
for(int k=0; k<n+m; k++)
if(a[i]<b[j])
   {c[k]=a[i]; i++;}
else
   {c[k]=b[j]; j++;}
}
```

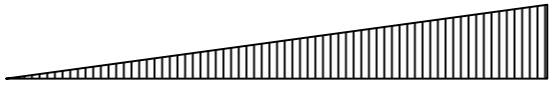
Ý tưởng của giải thuật sắp xếp trộn như sau:

- Giả sử có mảng a chưa sắp:

- Chia mảng a làm hai phần bằng nhau và sắp trên 2 nửa của a.



- Trộn 2 nửa đã sắp để thu được mảng a cũng được sắp:



Việc sắp trên 2 nửa của a cũng được áp dụng phương pháp sắp xếp trộn này, do đó 2 nửa lại tiếp tục được chia đôi, trộn…Như vậy ta có một giải thuật đệ quy.

Giải thuật sau sử dụng mảng b làm mảng trung gian. Mỗi khi cần trộn 2 nửa của mảng a ta sao một nửa đầu của a sang b, một nửa còn lại cũng đặt vào cuối của b nhưng theo thứ tự ngược lại và tiến hành trộn với i chạy từ đầu mảng b còn j chạy từ cuối mảng b. Việc trộn 2 nửa sau khi 2 nửa đã được đặt chung vào 1 mảng b như vậy sẽ không cần phải sử dụng phần tử chặn Max+1 như trên nữa.

```
int a[100], b[100];
void MergeSort(int l, int r)
{
if(r>l)
{
 int m=(l+r)/2;
 MergeSort(l,m); MergeSort(m+1, r);
//Sao chep nua dau cua a sang b
 for(int i=m; i>=l; i--) b[i]=a[i];
//Sao chep nua con lai cua a sang b theo thu tu nguoc lai
 for(int j=m+1; j<=r;j++) b[r+m+1-j]=a[j];
//i chay tu dau mang b, j chay tu cuoi mang b va tron
 i=l; j=r;
 for(int k=l; k<=r; k++)
 if(b[i]<b[j]) {a[k]=b[i];i++;}
 else       {a[k]=b[j];j--;}
}
}
```

Sắp xếp bằng trộn sử dụng khoảng n lg(n) lần so sánh để sắp bất kỳ một mảng nào gồm n phần tử.

Sắp xếp bằng trộn cũng sử dụng thêm một mảng b phụ trợ có kích thước n. Người ta đã chứng minh sắp xếp bằng trộn là ổn định và không bị ảnh hưởng bởi thứ tự ban đầu của dữ liệu.

b. Bài toán tìm kiếm

Cho một mảng a gồm n phần tử và một phần tử c cùng kiểu. Hỏi c có xuất hiện trong a không?

- **Phương pháp tìm kiếm tuần tự:**

Một phương pháp đơn giản để giải quyết bài toán trên là duyệt mảng. Giải thuật được trình bày như sau:

- Sử dụng một biến đếm d = 0;
- Duyệt qua các phần tử của mảng, nếu gặp c ta tăng biến đếm: d++;

Kết thúc quá trình duyệt mảng, nếu d bằng 0 chứng tỏ c không xuất hiện trong mảng và ngược lại.

Phương pháp trên cần thiết phải quyệt qua tất cả các phần tử của mảng một cách tuần tự, do vậy, độ phức tạp của giải thuật là O(n) với n là kích thước của mảng.

Nếu mảng a đã được sắp (tăng hoặc giảm) thì do tính chất đặc biệt này, ta có thể giải quyết bài toán mà không cần duyệt qua tất cả các phần tử của mảng. Phương pháp đó gọi là tìm kiếm nhị phân.

- **Phương pháp tìm kiếm nhị phân:**

Giả sử ta cần tìm kiếm c trong một đoạn từ vị trí L tới vị trí R trong mảng a (trường hợp tìm kiếm trên toàn bộ mảng thì L=0 và R=n-1). Ta làm như sau:

- Gọi M là vị trí giữa của đoạn mảng ta đang tìm kiếm (M = (L+R)/2). Trước tiên ta tiến hành kiểm tra a[M]. Khi đó, chỉ có thể xảy ra một trong 3 trường hợp sau:

a[M] = c: kết luận c có trong mảng a và ta có thể dừng quá trình tìm kiếm.

a[M] > c: vì mảng được sắp (giả sử sắp tăng) nên rõ ràng c (nếu có) chỉ nằm trong đoạn bên phải tức [M+1, R]. Ta tiến hành lặp lại quá trình tìm kiếm trên đoạn [M+1, R], tức cận trái L=M+1.

a[M] < c: khi đó c (nếu có) chỉ nằm trong đoạn bên trái của mảng tức [L, M-1]. Ta tiến hành lặp lại quá trình tìm kiếm trên đoạn [L, M-1], tức cận phải R = M-1.

Kết thúc quá trình tìm kiếm là khi xảy ra một trong hai trường hợp:

[1]. Nếu L > R chứng tỏ C không xuất hiện trong a.

[2]. Nếu a[M] = c chứng tỏ c xuất hiện trong a tại vị trí M.

Vậy quá trình chia đôi-tìm kiếm sẽ được lặp lại nếu: (a[M] !=c && L<=R). Hàm sau thực hiện việc tìm kiếm nhị phân trên một mảng a có kích thước n với một khoá c cần tìm, sử dụng vòng lặp:

```
void TKNP_Lap(int a[100], int n, int c)
{
int L=0, R=n-1, M;
do
{
M = (L+R)/2;
if (a[M]>c) R=M-1;
if (a[M]<c) L=M+1;
}
while(a[M]!=c && L<R);
if(a[M]==c)
     cout<<c<<" xuat hien tai "<<M;
```

```
else
     cout<<c<<" khong xuat hien";
}
```

Việc chia đôi và tìm kiếm trên một nửa của mảng được lặp đi lặp lại cho tới khi xảy ra 1 trong 2 trường hợp : *tìm thấy* và *không tìm thấy* gợi ý cho ta một cài đặt đệ quy cho hàm này:

- **Trường hợp suy biến:** là trường hợp a[M] = c hoặc L > R. Khi đó, nếu a[M]=c hàm trả về giá trị M là vị trí cuất hiện của c trong mảng; nếu L > R thì c không xuất hiện trong mảng và hàm trả về giá trị -1.

- **Trường hợp tổng quát:** là trường hợp a[M] > c hoặc a[M] < c. Khi đó việc gọi đệ quy là cần thiết với các cận L hoặc R được thay đổi cho phù hợp.

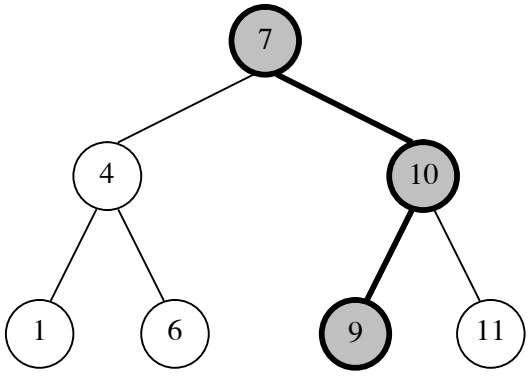
```
int TKNP_DQ(int a[100], int c, int L, int R)
{
int M=(L+R)/2;
if(a[M]==c)
       return M;
else if(L>R)
       return -1;
else //trường hợp tổng quát
       if(a[M]>c)  return TKNP_DQ(a,c,L,M-1);
       else        return TKNP_DQ(a,c,M+1,R);
}
```

Giải thuật trên giống như việc ta thăm phần tử chính giữa của đoạn mảng đang tìm kiếm, nếu không gặp c ta có thể "rẽ" sang bên trái hoặc bên phải để tìm tiếp tuỳ thuộc vào giá trị của phần tử chính giữa này. Việc này giống như việc tìm kiếm trên cây nhị phân (cây mà mỗi nút có 2 con sao cho các giá trị thuộc nhánh trái nhỏ hơn giá trị của nút và các giá trị của nhánh bên phải lớn hơn giá trị của nút).

Giả sử ta có mảng a như sau:

a	1	4	6	7	9	10	11

Các giá trị của a có thể biểu diễn trên cây nhị phân tìm kiếm như sau:



Nếu ta cần tìm một giá trị c bất kỳ, giả sử c = 9, ta chỉ cần đi hết chiều cao của cây. (Đường đi trong trường hợp này được tô đậm). Giả sử mảng có n phần tử, khi đó ta luôn có thể sử dụng một cây có chiều cao không quá $lg_2(n)$ để biểu diễn các giá trị của mảng trên cây. Do vậy, độ phức tạp trong trường của giải thuật này là $O(lg_2(n))$

II. MẢNG HAI CHIỀU

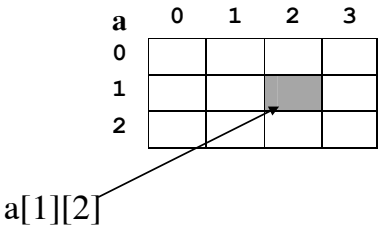
II.1. Các thao tác cơ bản trên mảng hai chiều

Mảng 2 chiều là một cấu trúc bộ nhớ gồm nhiều ô nhớ cùng tên, cùng kiểu nhưng khác nhau về chỉ số dùng để lưu trữ một bảng các phần tử cùng kiểu.

Mỗi bảng các phần tử bao gồm nhiều dòng và nhiều cột, do vậy, chỉ số của một phần tử của mảng được xác định bởi chỉ số của dòng và của cột tương ứng.

Giải sử bảng các phần tử có n dòng và m cột. Các dòng được đánh chỉ số bắt đầu từ 0: 0, 1, 2,…, n-1 và các cột cũng được đánh chỉ số từ 0: 0, 1, 2, …, m-1. Phần tử tại dòng i, cột j có chỉ số là [i][j].

Một mảng a gồm 3 dòng, 4 cột có dạng như sau:



Khai báo mảng 2 chiều:

<Kiểu mảng> <Tên mảng> [<Số dòng tối đa>] [<Số cột tối đa>];

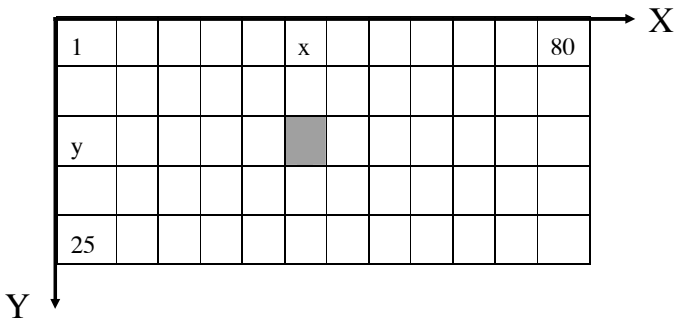
Ví dụ: int a[5][10]; khai báo một mảng a có tối đa 5 dòng và 10 cột. Mảng a có thể chứa được tối đa 50 phần tử kiểu nguyên.

Vấn đề tạo dáng ma trận:

Khi thể hiện mảng hai chiều lên màn hình, ta thường sử dụng câu lệnh gotoxy để tạo dáng ma trận cho mảng. Câu lệnh này có tác dụng đưa con trỏ màn hình tới tọa độ (x,y) đã chỉ ra trên màn hình để nhập hoặc xuất các phần tử của mảng. Cú pháp:

gotoxy(x, y);

Với x là tọa độ theo phương X (cột x) của màn hình, có giá trị từ 1 tới 80, y là tọa độ theo phương Y (dòng y) của màn hình, có giá trị từ 1 tới 25.



Toạ độ màn hình trong chế độ text mode

Như vậy, để tạo dáng ma trận cho mảng khi nhập, xuất ta sử dụng câu lệnh sau:

gotoxy(x + k*j, y + i);

Với (x, y) là toạ độ đỉnh trái trên của ma trận và k là bước nhảy theo trục x (hay khoảng cách giữa các cột của mảng - thường chọn k=3). Thông thường ta hay nhập xuất mảng tại 4 vị trí của màn hình tương ứng với các lệnh gotoxy sau:

```
gotoxy(5 + 3 *j, 5 + i);          gotoxy(35 + 3 *j, 5 + i);
gotoxy(5 + 3 *j, 15 + i);         gotoxy(35 + 3 *j, 15 + i);
```

Nhập mảng:

- Ta sử dụng hai vòng lặp for lồng nhau để nhập từng phần tử cho mảng. Đoạn trình sau nhập giá trị cho một mảng số a gồm n dòng, m cột tạo dáng tại vị trí có toạ độ (5,5) :

```
for(i=0; i<n; i++)
for(j=0; j<m; j++)
{
        gotoxy(5 + 3 * j, 5 + i);
        cin>>a[i][j];
}
```

Xuất mảng:

Ta cũng sử dụng hai vòng for lồng nhau và tạo dáng ma trận cho mảng khi xuất. Đoạn trình sau xuất mảng a gồm n dòng, m cột tạo dáng tại toạ độ (35, 5):

```
for(i=0; i<n; i++)
for(j=0; j<m; j++)
{
        gotoxy(35 + 3 * j, 5 + i);
        cout<<a[i][j];
}
```

Duyệt mảng:

Việc thăm lần lượt tất cả các phần tử của mảng cần phải sử dụng 2 vòng lặp for lồng nhau:

```
for(i=0; i<n; i++)
for(j=0; j<m; j++)
{
      //Thăm phần tử có chỉ số [i][j]
}
```

Thao tác duyệt mảng là thao tác thường xuyên sử dụng trong các bài tập về mảng. Ngay việc nhập, xuất mảng về bản chất cũng là thao tác duyệt mảng.

II.2. Các bài toán cơ bản trên mảng 2 chiều

- **Các bài toán trên ma trận:**

Về mặt hình thức, mảng có dạng ma trận. Do vậy, một dạng bài tập phổ biến về mảng là thực hiện các bài toán trên ma trận. Thuộc dạng này có các bài như: Cộng, trừ, nhân hai ma trận; chuyển vị ma trận, đổi dấu ma trận, nghịch đảo ma trận…

Ví dụ 1: Nhập vào một ma trận a (n x k) và b (k x m) các phần tử thực. Tính ma trận c = a * b.

Với hai ma trận a, b đã cho, tích của chúng là ma trận c có kích thước n x m với:

$$c[i][j] = \sum_{t=0}^{k-1} a[i][t] * b[t][j]$$

```
void main()
{
int a[50][50], b[50][50], c[50][50];
int n, m, k; clrscr();
  cout<<"n="; cin>>n;
  cout<<"m="; cin>>m;
  cout<<"k="; cin>>k;
for(int i=0; i<n; i++)
for(int j=0; j<k; j++)
{
     gotoxy(5+3*j, 5+i);
     cin>>a[i][j];
}
for(i=0; i<k; i++)
for(j=0; j<m; j++)
{
     gotoxy(35+3*j, 5+i);
     cin>>b[i][j];
}
   for(i=0; i<n; i++)
   for(j=0; j<m; j++)
   {  c[i][j]=0;
```

```
            for(int t=0; t<k; t++)c[i][j]+=a[i][t]*b[t][j];
            gotoxy(5+3*j, 15+i);
            cout<<c[i][j];
         }
getch();
}
```

- **Thống kê trên ma trận**

Một dạng bài toán khá phổ biến trên mảng hai chiều là thống kê trên ma trận. Việc thống kê một đại lượng nào đó nhằm mục đích kiểm tra một tính chất nào đó của mảng. Một vấn đề khó khăn đặt ra là xác định chính xác xem cần thống kê đại lượng nào nếu đề bài chưa nói rõ.

Ví dụ 1: Nhập vào một ma trận vuông a (n x n) các phần tử thực. Ma trận a được gọi là "hợp lệ" nếu tất cả các phần tử trên dường chéo chính đều bằng 0; các phần tử phía trên đường chéo chính đều dương và các phần tử còn lại đều âm. Hãy kiểm tra xem ma trận vừa nhập có hợp lệ không?

Ta biểu diễn điều kiện để ma trận vuông a là "hợp lệ" như sau:

$$\begin{cases} \forall i = j, a[i][j] = 0 \\ \forall i < j, a[i][j] > 0 \\ \forall i > j, a[i][j] < 0 \end{cases} \qquad \text{(IV.1)}$$

Lấy phủ định của điều kiện này ta thu được điều kiện để một ma trận vuông là "không hợp lệ" như sau:

$$\begin{cases} \exists i = j, a[i][j] \neq 0 \\ \exists i < j, a[i][j] \leq 0 \\ \exists i > j, a[i][j] \geq 0 \end{cases} \qquad \text{(IV.2)}$$

Như vậy việc giải bài toán sẽ trở nên dễ dàng. Theo đó, ta thống kê các phần tử a[i][j] thoả mãn điều kiện IV.2. Nếu không tồn tại phần tử nào thoả mãn IV.2 thì ma trận là hợp lệ.

```
void main()
{
int a[50][50];int n; clrscr();
   cout<<"n="; cin>>n;
for(int i=0; i<n; i++)
for(int j=0; j<n; j++)
{
   gotoxy(5+3*j, 5+i);
   cin>>a[i][j];
}
int d=0;
for(i=0; i<n; i++)
for(j=0; j<n; j++)
```

```
   {
      if(i==j && a[i][j]!=0) d++;
      if(i<j && a[i][j]<=0)  d++;
      if(i>j && a[i][j]>=0)  d++;
   }
   if(d==0)
        cout<<"Ma tran la hop le !";
   else
        cout<<"Ma tran la khong hop le !";
   getch();
   }
```

Ví dụ 2: Có n cửa hàng kinh doanh trong m tháng. Doanh thu của mỗi cửa hàng trong mỗi tháng đều được lưu trữ trong một ma trận có n dòng, m cột. Một cửa hàng sẽ bị đóng cửa nếu doanh thu của nó giảm liên tiếp trong m-1 tháng (trừ tháng đầu tiên). Hãy cho biết cửa hàng nào trong số n cửa hàng trên sẽ bị đóng cửa?

Bài toán được giải quyết bằng cách duyệt qua từng cửa hàng (0 -> n-1). Với mỗi của hàng i, ta đếm số lần giảm doanh thu của nó. Nếu cửa hàng nào có đúng m-1 lần giảm doanh thu, cửa hàng đó sẽ bị đóng cửa.

```
void main()
{
int a[50][50];int n,m; clrscr();
   cout<<"Nhap so cua hang n="; cin>>n;
   cout<<"Nhap so thang m="; cin>>m;
   cout<<"Nhap dt cua tung cua hang-tung thang:";
for(int i=0; i<n; i++)
for(int j=0; j<m; j++)
{
   gotoxy(5+3*j, 5+i);
   cin>>a[i][j];
}
for(i=0; i<n; i++)
{
   int d=0;
   for(j=0; j<m-1; j++) if(a[i][j]>a[i][j+1]) d++;
   if (d==m-1)
        cout<<"Cua hang "<<i+1<<" se bi dong cua !";
}
getch();
}
```

III. XÂU KÝ TỰ – MẢNG CÁC KÝ TỰ

III.1. Một số lưu ý khi sử dụng xâu ký tự

Xâu ký tự (hay chuỗi ký tự) là một dãy liên tiếp các ký tự. Như vậy, về bản chất xâu ký tự giống với một mảng một chiều mà mỗi phần tử của mảng là một ký tự.

Tuy nhiên, ngoài các đặc điểm như mảng, xâu ký tự còn có đặc thù riêng.

- **Khai báo biến kiểu xâu ký tự:** Có hai cách để khai báo biến xâu ký tự:

  - Khai báo như mảng:

    char <Tên biến xâu> [<số ký tự tối đa trong xâu>]; **(1)**

  - Khai báo như con trỏ xâu:

    char * <Tên biến xâu>; **(2)**

Với cách khai báo (1), ta cần chỉ ra độ dài tối đa của xâu. Độ dài này không vượt quá 256.

Với cách khai báo (2), độ dài xâu chưa xác định. Như vậy xâu có thể chứa tối đa 256 ký tự.

Ví dụ: ta khai báo: **char S[30]; char * T;**

Ta thu được hai biến xâu. Biến xâu S chỉ chứa được các xâu có độ dài không quá 30 ký tự. Biến xâu T có độ dài chưa xác định và nó có thể chứa được xâu có độ dài bất kỳ không quá 256 ký tự.

- **Nhập xâu:**

Mặc dù xâu có bản chất giống mảng như việc nhập xâu không phức tạp như nhập mảng (không cần sử dụng vòng lặp). Thông thường người ta hay sử dụng lệnh gets cho việc nhập xâu:

gets(<Biến xâu>);

Ví dụ: để nhập xâu S, ta viết: gets(S);

Ta cũng có thể sử dụng lệnh scanf cho việc nhập xâu hoặc thậm chí lệnh cin để nhập các xâu không chứa dấu cách. Nếu sử dụng liên tiếp nhiều lệnh gets để nhập nhiều xâu thì giữa các lệnh gets phải xoá bộ đệm bằng lệnh fflush(stdin);

- **Xuất xâu:**

Tương tự nhập xâu, việc xuất xâu cũng trở nên rất đơn giản bằng cách sử dụng một trong các lệnh xuất puts, printf, cout.

Ví dụ: Xuất xâu S lên màn hình, ta viết: cout<<S; hoặc puts(S); …

- **Duyệt xâu:**

Việc duyệt xâu cũng tương tự như duyệt mảng. Tuy nhiên, ta cần sử dụng hàm strlen(<Biến xâu>) trả về độ dài của xâu cần duyệt (hàm này có sẵn trong thư viện string.h). Giả sử duyệt xâu S, ta viết:

```
for(i=0; i < strlen(S); i++)
{
        //Thăm ký tự S[i];
}
```

- **Phép gán xâu:**

Nếu a và b là hai biến không phải kiểu xâu, ta dễ dàng gán a sang b và ngược lại bằng cách viết: a=b; hoặc b = a;

Tuy nhiên, nếu a và b là hai biến kiểu xâu, việc sử dụng phép gán trên là không hợp lệ. Để gán xâu b sang xâu a, ta phải sử dụng hàm copy xâu:

strcpy(a, b);

Một cách tổng quát, hàm copy xâu được viết như sau:

strcpy(S1, S2);

Trong đó, S1 là một biến xâu còn S2 có thể là một biến xâu hoặc một hằng xâu. Khi đó, xâu ký tự S2 sẽ được gán sang S1.

Ví dụ:

```
char S1[30], S2[30];
strcpy(S1, "Ha Noi"); //Gán "Ha Noi" vào S1
strcpy(S2, S1); //Gán S1 vào S2, S1 và S2 có cùng nội dung là Ha Noi
```

- **Phép so sánh xâu:**

Để so sánh 2 biến xâu, ta cũng không được phép sử dụng toán tử so sánh (==) mà sử dụng hàm so sánh xâu:

int strcmp(S1, S2);

Hàm này sẽ trả về giá trị bằng 0 nếu hai xâu bằng nhau; giá trị dương nếu S1 > S2 và giá trị âm nếu S1 <S2.

Ví dụ:

```
char S1[30], S2[30];
gets(S1); fflush(stdin);
gets(S2);
if (strcmp(S1, S2)==0)
        cout<< " Xau S1 bang xau S2";
else    cout<< "Xau S1 khac xau S2";
```

Hai hàm strcpy và strcmp có sẵn trong thư viện string.h.

- **Lấy mã ASCII của một ký tự trong xâu:**

Một số bài toán ta cần lấy mã ASCII của các ký tự. Công việc này trong C++ được thực hiện một cách dễ dàng mà không cần sử dụng tới hàm lấy mã ASCII. Để lấy mã ASCII của một ký tự S[i], ta chỉ cần đặt toán tử ép kiểu (int) ngay trước ký tự đó:

Ví dụ: Với S là một xâu, đương nhiên S[i] là một ký tự của xâu. Hai câu lệnh sau sẽ cho kết quả khác nhau:

```
cout<<S[i];          (1)
cout<<(int) S[i];    (2)
```

Câu lệnh (1) in ra màn hình ký tự S[i], còn câu lệnh (2) chỉ in ra mã ASCII của ký tự đó.

- **Chuyển mã ASCII thành ký tự:**

Tương tự như trên, việc chuyển mã ASCII thành ký tự cũng được thực hiện dễ dàng bằng toán tử ép kiểu (char).

Giả sử muốn in ra màn hình ký tự có mã 65 (chữ A), ta chỉ cần viết:

```
cout<<(char) 65;
```

Khi đó ký tự 'A' sẽ được in ra màn hình do nó có mã ASCII là 65.

III.2. Một số bài toán đặc thù trên xâu

Ngoài các bài toán tìm kiếm, sắp xếp như trên một mảng thông thường, các bài toán trên xâu còn được mở rộng do tính đặc thù của xâu. Sau đây là một số dạng phổ biến:

- **Thống kê trên xâu:** chẳng hạn thống kê số ký tự, số từ, số câu, số dấu chấm,….

- **Chuẩn hoá xâu:** Cắt các ký tự trống ở hai đầu xâu, xoá bớt dấu cách nếu có 2 dấu cách liền nhau trong thân xâu. Trước dấu chấm câu không có dấu cách, sau dấu chấm câu có 1 dấu cách; ký tự đầu câu viết hoa…

Ví dụ 1:

Nhập vào một xâu ký tự bất kỳ. Một từ trong xâu là một dãy liên tiếp, dài nhất các ký tự khác ký tự trống. Hãy cho biết xâu vừa nhập có bao nhiêu từ.

Dễ thấy với một xâu chưa chuẩn thì số từ không tỷ lệ thuận với số dấu cách. Do vậy việc đếm số dấu cách là không phù hợp.

Thay vào đó, ta đi đếm số lần bắt đầu của một từ, đó là số lần S[i] bằng dấu cách và S[i+1] khác dấu cách. Tuy nhiên, trong trường hợp S[0] khác dấu cách thì ta vẫn đếm thiếu 1 từ đầu tiên nên phải tăng biến đếm lên 1.

```
void main()
{
char S[30];
   cout<<"S="; gets(S);
int d=0;
for(int i=0; i<strlen(S)-1; i++)
if(S[i]==' ' && S[i+1] !=' ') d++;

     if(S[0]!=' ') d++;
     cout<<"Xau co: "<<d<<" tu !";
getch();
}
```

Ví dụ 2: Nhập một xâu ký tự S bất kỳ. Hãy đếm số lần xuất hiện của tất cả các chữ cái có trong S.

Nếu không phân biệt chữ hoa và chữ thường thì bảng mã ASCII có tổng cộng 26 chữ cái. Trường hợp tồi nhất là cả 26 chữ cái này đều xuất hiện trong S. Do vậy ta sử dụng 26 biến đếm (mảng d gồm 26 phần tử nguyên).

Nếu ta gặp một ký tự nào đó trong S thì biến đếm tương ứng của nó sẽ được tăng lên 1. Bảng sau chỉ ra sự tương ứng giữa biến đếm và ký tự:

Chữ cái hoa (mã ASCII)	**Biến đếm tương ứng**
A (65)	d[0] = d[65-65]
B (66)	d[1] = d[66-65]
C (67)	d[2] = d[67-65]
…	…
S[i] ((int) S[i])	**d[(int) S[i]-65]**

Chữ cái thường (mã ASCII)	**Biến đếm tương ứng**
a (97)	d[0] = d[97- 97]
b (98)	d[1] = d[98-97]
c (99)	d[2] = d[99-97]
…	…
S[i] ((int) S[i])	**d[(int) S[i]-97]**

Như vậy ta cần chia hai trường hợp:

Trường hợp thứ nhất: nếu chữ cái S[i] có mã ASCII nhỏ hơn 97 thì S[i] sẽ là chữ cái hoa. Mã ASCII của nó là k = (int) S[i]. Khi đó ta cần tăng biến đếm d[k-65].

Trường hợp thứ 2: nếu chữ cái S[i] có mã ASCII lớn hơn hoặc bằng 97 thì S[i] sẽ là chữ cái thường. Mã ASCII của nó là k = (int) S[i]. Khi đó ta cần tăng biến đếm d[k-97].

Kết thúc quá trình đếm, ta duyệt lại mảng d. Nếu thấy d[i] khác 0 thì đó chính là số lần xuất hiện của chữ cái (char) (i+65) hoặc (char) (i+97).

```
void main()
{
char S[30];
   cout<<"S="; gets(S);
int d[26];
for(int i=0; i<26; i++) d[i]=0;
for(i=0; i<strlen(S); i++)
{
   if((int)S[i] <97)     d[(int)S[i]-65]++;
   else                  d[(int)S[i]-97]++;
}
for(i=0; i<26; i++)
if(d[i]>0)
cout<<"ky tu "<<(char)(i+97)<<" xuat hien "<<d[i]<<" lan !"<<endl;
getch();
}
```

CHƯƠNG V. KỸ THUẬT LẬP TRÌNH DÙNG CON TRỎ

I. TỔNG QUAN VỀ CON TRỎ

I.1. Khái niệm và cách khai báo

- Mỗi byte trong bộ nhớ đều được đánh địa chỉ, là một con số hệ thập lục phân. Địa chỉ của biến là địa chỉ của byte đầu tiên trong vùng nhớ dành cho biến.

Thông thường, khi ta khai báo một biến, máy tính sẽ cấp phát một ô nhớ với kích thước tương ứng trong vùng 64Kb dành cho việc khai báo biến (mô hình tiny). Ô nhớ này có thể dùng để lưu trữ các giá trị khác nhau, gọi là "giá trị của biến". Bên cạnh đó, mỗi biến còn có một địa chỉ là một con số hệ thập lục phân.

Con trỏ (hay biến con trỏ) là một biến đặc biệt dùng để chứa địa chỉ của các biến khác.

Như vậy, con trỏ cũng giống như biến thường (tức cũng là một ô nhớ trong bộ nhớ) nhưng điểm khác biệt là nó không thể chứa các giá trị thông thường mà chỉ dùng để chứa địa chỉ của biến.

Con trỏ cũng có nhiều kiểu (nguyên, thực, ký tự…). **Con trỏ thuộc kiểu nào chỉ chứa địa chỉ của biến thuộc kiểu đó.**

Cú pháp khai báo:

<Kiểu con trỏ> * <Tên con trỏ>;

Trong đó: **<Kiểu con trỏ>** có thể là một trong các kiểu chuẩn của C++ hoặc kiểu tự định nghĩa. **<Tên con trỏ>** được đặt tuỳ ý theo quy ước đặt tên trong C++.

Ví dụ: dòng khai báo int a, *p; float b, *q; khai báo a và p kiểu nguyên, b và q kiểu thực, trong đó, p và q là hai con trỏ. Khi đó, p có thể chứa địa chỉ của a và q có thể chứa địa chỉ của b.

I.2. Một số thao tác cơ bản trên con trỏ

- **Lấy địa chỉ của biến đặt vào con trỏ:**

Giả sử a là một biến nguyên và p là một con trỏ cùng kiểu với a. Để lấy địa chỉ của a đặt vào p ta viết:

P = &a;

Toán tử & cho phép lấy địa chỉ của một biến bất kỳ. Khi đó, ta nói p đang trỏ tới a. Một cách tổng quát, để lấy địa chỉ của một biến đặt vào con trỏ cùng kiểu, ta viết:

<Tên con trỏ> = & <Tên biến>;

- **Phép gán con trỏ cho con trỏ:**

Nếu p và q là hai con trỏ cùng kiểu, ta có thể gán p sang q và ngược lại, ta viết: p = q; hoặc q = p; Khi đó, địa chỉ đang chứa trong con trỏ ở vế phải sẽ được đặt vào con trỏ ở vế trái và ta nói hai con trỏ cùng trỏ tới một biến.

Ví dụ:

```
int a, *p, *q;
p=&a; //cho p trỏ tới a
q = p; //p và q cùng trỏ tới a
```

- **Sử dụng con trỏ trong biểu thức:**

Khi sử dụng biến con trỏ trong biểu thức thì địa chỉ đang chứa trong con trỏ sẽ được sử dụng để tính toán giá trị của biểu thức.

Nếu muốn lấy *giá trị của biến mà con trỏ đang trỏ tới* để sử dụng trong biểu thức thì ta thêm dấu * vào đằng trước tên biến con trỏ.

Ví dụ:

```
int a=5, b=3, *p, *q;
p=&a;
q=&b;
int k = p + q;
int t = *p + *q;
```

Khi đó, hai biến k và t có giá trị khác nhau. Trong biểu thức k, địa chỉ đang chứa trong con trỏ p và q sẽ được cộng lại và đặt vào k; ngược lại t sẽ có giá trị = a + b = 8.

II. CON TRỎ - MẢNG VÀ HÀM

II.1. Con trỏ và mảng

- **Con trỏ là mảng**

Khi khai báo mảng, ta được cấp phát một dãy các ô nhớ liên tiếp, cùng kiểu được gọi là các phần tử của mảng. Điều đặc biệt là tên mảng chính là một con trỏ trỏ tới phần tử đầu tiên của mảng. Như vậy tên mảng nắm giữ địa chỉ của ô nhớ đầu tiên trong mảng và do vậy, ta có thể sử dụng tên mảng để quản lý toàn bộ các phần tử của mảng.

Ví dụ: giả sử ta khai báo: **int a[10];** Khi đó, 10 ô nhớ được cấp phát cho mảng a như sau:

Các phần tử lần lượt là a[0], a[1], a[2],….a[9]. Tuy nhiên, a là một ô nhớ riêng biệt và ô nhớ này đang chứa địa chỉ của a[0] (a trỏ tới a[0]):

Dễ thấy có sự tương ứng sau:

a	**là địa chỉ của a[0]**
a+1	**là địa chỉ của a[1]**
a+2	**là địa chỉ của a[2]**
…	
a+i	**là địa chỉ của a[i]**

Như vậy thì

***a**	**là a[0]**
***(a+1)**	**là a[1]**
***(a+2)**	**là a[2]**
…	
***(a+i)**	**là a[i]**

Vậy ta có thể sử dụng cách viết thứ hai cho các phần tử của mảng. Thay vì viết a[i], ta có thể viết *(a+i)

- **Con trỏ là mảng**

Một con trỏ p bất kỳ cũng tương đương với một mảng một chiều. Thật vậy, giả sử con trỏ p đang chứa địa chỉ của một ô nhớ a nào đó, khi đó ta có thể sử dụng p để quản lý một dãy các ô nhớ liên tiếp bắt đầu từ a.

Như vậy:

p	**là địa chỉ của a**
p+1	**là địa chỉ của ô nhớ ngay sau a**
…	
p+i	**là địa chỉ của ô nhớ thứ i kể từ a.**

Vậy, với p là con trỏ thì ta có thể coi nó như mảng một chiều và để truy cập tới phần tử thứ i của p ta có thể viết *(p+i) hoặc thậm chí viết p[i].

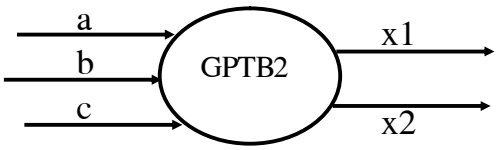
II.2. Con trỏ và hàm

- **Phân loại đối số**

Một hàm trong C++ có thể không trả về giá trị nào mà đơn giản chỉ thực hiện một công việc nào đó (hàm void). Tuy nhiên, với những hàm có giá trị trả về, giá trị đó sẽ được đặt vào tên hàm (trả về thông qua tên hàm) bằng lệnh return <giá trị trả về>;. Lệnh return này tương tự như việc ta gán <tên hàm> = <giá trị trả về>;

Với một hàm, tên hàm chỉ có một nên hàm chỉ có thể trả về duy nhất một giá trị thông qua tên hàm.

Tuy nhiên, có những hàm đòi hỏi phải trả về nhiều hơn một giá trị, chẳng hạn hàm giải phương trình bậc hai. Hàm này có các đối vào là các hệ số a, b, c của phương trình bậc hai và nếu có nghiệm thì hàm sẽ trả về 2 nghiệm x1 và x2.



Nếu viết hàm theo kiểu có giá trị trả về như cách thông thường (trả về qua tên hàm) ta sẽ gặp khó khăn do một tên hàm không thể chứa cùng lúc hai giá trị x1 và x2.

Để giải quyết khó khăn đó, ta sử dụng kỹ thuật "đối ra" cho hàm. Theo đó, các đối của hàm được chia làm hai loại:

- **Đối vào:** là các biến mang giá trị đầu vào cho hàm

- **Đối ra:** là các biến chứa giá trị đầu ra của hàm.

Nếu hàm trả về nhiều giá trị thì các giá trị đó thường không được đặt vào tên hàm (qua lệnh return) mà được trả về qua đối ra.

Nếu đối ra là biến thường thì khi sử dụng hàm, ta chỉ có thể truyền tham số dưới dạng tham trị. Điều đó có nghĩa là sau khi ra khỏi hàm, các tham số này lại quay trở về giá trị ban đầu như trước khi nó được truyền vào hàm. Như vậy chúng không thực hiện được "phận sự" của mình là mang các giá trị đầu ra ra khỏi hàm. Do vậy, chỉ có thể sử dụng cách truyền tham số theo kiểu tham chiếu, tức là:

Các đối ra bắt buộc phải là con trỏ.

Ví dụ 1: viết hàm trả về các nghiệm (nếu có) của phương trình bậc hai.

Hàm sau sẽ trả về giá trị -1 qua tên hàm nếu phương trình bậc hai vô nghiệm. Ngược lại, nó trả về giá trị +1. Khi đó, hai nghiệm được đặt vào hai đối ra. Như vậy hàm có 3 đối vào và 2 đối ra đồng thời hàm trả về giá trị nguyên qua tên hàm (hàm int):

```
int GPTB2(float a,float b,float c,float *x1,float *x2)
{
float DT=b*b-4*a*c;
if(DT<0)
     return -1;
else
     {
     *x1=(-b+sqrt(DT))/(2*a);
     *x2=(-b-sqrt(DT))/(2*a);
     return 1;
     }
}
void main()
{
float a,b,c;
     cout<<"a="; cin>>a;
     cout<<"b="; cin>>b;
     cout<<"c="; cin>>c;
float x1, x2;
int k=GPTB2(a,b,c,&x1,&x2);
if(k==-1)
     cout<<"Phuong trinh vo nghiem";
else
     cout<<"Pt co 2 nghiem x1="<<x1<<" x2="<<x2;
getch();
}
```

Ví dụ 2: Viết hàm trả về đồng thời 3 giá trị là tổng các số chẵn, tổng các số lẻ và tổng các số chia hết cho 3 trong một mảng n phần tử nguyên.

Các đối vào: mảng nguyên a, kích thước thực tế của mảng n.

Các đối ra: T1- tổng các số chẵn trong mảng; T2- tổng các số lẻ trong mảng; T3- tổng các số chia hết 3 trong mảng.

```
void TinhTong(int *a, int n, int *T1, int *T2, int *T3)
{
*T1=*T2=*T3=0;
for(int i=0; i<n; i++)
{
     if(a[i]%2==0)  *T1+=*(a+i);
     else           *T2+=*(a+i);
     if(a[i]%3==0)  *T3+=*(a+i);
}
}
void main()
{
int *a;int n;
     cout<<"n="; cin>>n;
for(int i=0; i<n; i++)
     cin>>a[i];
int T1, T2, T3;
     TinhTong(a,n,&T1,&T2,&T3);
     cout<<"Tong chan="<<T1<<endl;
```

```
        cout<<"Tong le="<<T2<<endl;
        cout<<"Tong chia het 3="<<T3;
    getch();
    }
```

Hàm TinhTong() ở trên không trả về giá trị nào thông qua tên hàm (hàm void) nhưng lại trả về đồng thời 3 giá trị thông qua 3 đối ra. Các tham số T1, T2, T3 được truyền vào hàm chỉ làm nhiệm vụ duy nhất là chứa các tổng tính được và "mang" ra khỏi hàm.

III. CẤP PHÁT VÀ GIẢI PHÓNG BỘ NHỚ CHO CON TRỎ

III.1. Cấp phát bộ nhớ động cho con trỏ

Việc sử dụng con trỏ thay cho mảng sẽ giúp tiết kiệm bộ nhớ nếu như ta cấp phát bộ nhớ động cho con trỏ (tức sử dụng tới đâu, cấp phát tới đó).

Việc cấp phát bộ nhớ cho con trỏ sử dụng các hàm định vị bộ nhớ (allocation memory) của C++. Có rất nhiều hàm làm công việc này, tuy nhiên ta hay sử dụng hai hàm calloc và malloc

- **Hàm calloc:**

Cú pháp:

<con trỏ> = (<Kiểu con trỏ>*) calloc(<n>, <size>);

Trong đó, <n> là số ô nhớ cần cấp phát (số phần tử của mảng); <size> là kích thước của một ô nhớ.

Hàm calloc nếu thực hiện thành công sẽ cấp phát một vùng nhớ có kích thước <n>*<size> Byte và <con trỏ> sẽ trỏ tới ô nhớ đầu tiên của vùng nhớ này. Ngược lại, nếu thực hiện không thành công (do không đủ bộ nhớ hoặc <n> hoặc <size> không hợp lệ) hàm sẽ trả về giá trị NULL (tức con trỏ trỏ tới NULL).

Giả sử p là một mảng nguyên, khi đó kích thước mỗi ô nhớ là 2 Byte (tức <size> = 2). Nếu p là mảng thực thì <size> =4 .v.v…Toán tử sizeof sẽ cho ta biết kích thước của mỗi ô nhớ thuộc một kiểu bất kỳ. Muốn vậy ta chỉ cần viết: sizeof(<kiểu>). Ví dụ sizeof(int) = 2; sizeof(float) = 4; .v.v…

Hàm calloc thuộc thư viện alloc.h.

Ví dụ 1: Nhập một mảng p gồm n phần tử nguyên, sử dụng hàm calloc cấp phát bộ nhớ động.

```
int *p, n;
        cout<< "Nhập n="; cin>>n;
p = (int *) calloc(n, sizeof(int));
if(p==NULL)
```

```
        cout<< “Cap phat bo nho khong thanh cong”;
    else //Nhap mang
    for(int i=0; i<n; i++)
    {
        cout<< “p[”<<i<< “]=”;
        cin>>p[i];
    }
```

- **Hàm malloc:**

Tương tự như hàm calloc, hàm malloc sẽ cấp phát một vùng nhớ cho con trỏ. Cú pháp như sau:

<Con trỏ> = (<Kiểu con trỏ>*) malloc(<size>);

Trong đó <size> là kích thước của ô nhớ cần cấp phát tính bằng Byte. Chẳng hạn ta cần cấp phát bộ nhớ cho một mảng a gồm 10 phần tử nguyên. Khi đó kích thước vùng nhớ cần cấp phát = 10 * sizeof(int) = 10*2=20 Byte, ta viết:

a = (int*) malloc(20); hoặc **a = (int*) malloc(10 * sizeof(int));**

Ví dụ 2: Nhập một mảng p gồm n phần tử nguyên, sử dụng hàm malloc cấp phát bộ nhớ động.

```
int *p, n;
        cout<< “Nhập n=”; cin>>n;
p = (int *) malloc(n*sizeof(int));
if(p==NULL)
        cout<< “Cap phat bo nho khong thanh cong”;
else //Nhap mang
for(int i=0; i<n; i++)
{
        cout<< “p[”<<i<< “]=”;
        cin>>p[i];
}
```

Ví dụ 3. Nhập một mảng a gồm n phần tử thực bằng cách sử dụng con trỏ và cấp phát bộ nhớ động. Tìm phần tử lớn nhất và lớn thứ nhì trong mảng.

```
void main()
{
float *a;int n;
     cout<<"n="; cin>>n;
     a = (float*) calloc(n, sizeof(float));
if(a==NULL)
```

```
        cout<<"cap phat bo nho that bai!";
    else
    {
    for(int i=0; i<n; i++)
    {
        cout<<"a["<<i<<"]=";
        cin>>*(a+i);
    }

    int Max1, Max2;
    //Tim phan tu lon nhat chua vao Max1
    Max1=a[0];
        for(i=0; i<n; i++)
        if(Max1<*(a+i)) Max1=*(a+i);
        //Tim phan tu lon thu nhi chua vao Max2
        i=0;
        while(a[i]==Max1) i++;
        Max2=a[i];
        for(i=0; i<n; i++)
            if(Max2<*(a+i) && *(a+i) !=Max1) Max2=*(a+i);
    //In ket qua ra man hinh
    cout<<"Phan tu lon nhat = "<<Max1<<endl;
    cout<<"Phan tu lon thu nhi = "<<Max2<<endl;
    }
    getch();
    }
```

Giải thuật trên sẽ không cho kết quả đúng khi tất cả các phần tử của mảng bằng nhau (không tồn tại số lớn thứ nhì). Ta có thể khắc phục điều đó bằng cách kiểm tra trước trường hợp này.

III.2. Cấp phát lại hoặc giải phóng bộ nhớ cho con trỏ

Đôi khi, trong quá trình hoạt động, kích thước của mảng lại thay đổi. Nếu ta sử dụng cấp phát bộ nhớ động thì kích thước của mảng "vừa đủ dùng" nên nếu kích thước này tăng hoặc giảm (khi chương trình thực thi mới phát sinh điều này) thì cần thiết phải cấp phát lại bộ nhớ cho con trỏ.

Để làm điều đó, ta sử dụng hàm realloc. Hàm này có nhiệm vụ cấp phát một vùng nhớ với kích thước mới cho mảng (con trỏ) nhưng vẫn giữ nguyên các giá trị vốn có của mảng.

Cú pháp:

<Con trỏ> = (<Kiểu con trỏ>*) realloc(<Con trỏ>, <Kích thước mới>);

Trong đó, <Kích thước mới> được tính bằng Byte.

Ví dụ: Nhập vào một mảng a gồm n phần tử nguyên. Hãy sao chép các giá trị chẵn của mảng đặt vào cuối mảng.

Giả sử ta có mảng a ban đầu gồm các phần tử như sau:

1	4	3	2	6	5

Sau khi sao chép các phần tử chẵn đặt vào cuối mảng, mảng a có dạng:

1	4	3	2	6	5	4	2	6

Rõ ràng kích thước của mảng a bị thay đổi (tăng lên). Mỗi khi có một phần tử chẵn được đặt vào cuối mảng thì kích thước của mảng được tăng lên 1. Do đó cần cấp phát lại bộ nhớ cho a với kích thước tăng thêm 1 ô nhớ (2 Byte).

```
void main()
{
int *a;int n;
     cout<<"n="; cin>>n;
     a = (int*) calloc(n, sizeof(int));
if(a==NULL)
     cout<<"cap phat bo nho that bai!";
else
{
for(int i=0; i<n; i++)
{
     cout<<"a["<<i<<"]=";
     cin>>*(a+i);
}
int m=n;
for(i=0; i<m; i++)
if(a[i]%2==0)
{
     a = (int*) realloc(a,(n+1)*sizeof(int));
     a[n]=a[i];n++;
}
}
for(int i=0; i<n; i++)
cout<<a[i]<<"  ";
getch();
}
```

- **Giải phóng bộ nhớ đang chiếm giữ bởi con trỏ**

Khi không sử dụng tới con trỏ nữa, nếu ta không giải phóng vùng nhớ đã cấp phát cho con trỏ thì hiển nhiên vùng nhớ này vẫn bị nó chiếm giữ và không thể cấp phát cho các con trỏ khác (nếu có). Đặc biệt trong các hàm có cấp phát bộ nhớ động cho con trỏ, khi mà việc gọi hàm xảy ra thường xuyên nhưng khi kết thúc hàm ta không giải phóng vùng nhớ đã cấp phát thì bộ nhớ sẽ bị chiếm dụng một cách nhanh chóng.

Giải phóng vùng nhớ đang bị con trỏ chiếm giữ đơn giản là xoá địa chỉ đang lưu trữ trong con trỏ đó. Việc này sẽ "cắt đứt" mối liên hệ giữa con trỏ và vùng nhớ mà nó quản lý. Để làm như vậy, hãy sử dụng lệnh free.

Cú pháp:

free(<Tên con trỏ muốn giải phóng>);

Ví dụ: Giả sử con trỏ p đã được cấp phát bộ nhớ. Muốn giải phóng nó, ta viết: **free(p);**

MỘT SỐ BÀI TẬP THỰC HÀNH

MÔN KỸ THUẬT LẬP TRÌNH

Hệ: Đại học

CHƯƠNG I: MỞ ĐẦU

1. Nhập hai số nguyên, tính tổng, hiệu, tích, thương, đồng dư.
2. Nhập một số nguyên <= 9999, in ra màn hình cách đọc số nguyên đó (VD: số 1523 đọc là: 1 ngàn 5 trăn 2 chục 3 đơn vị). Nhận xét về cách làm vừa áp dụng nếu số nguyên nhập vào không được giới hạn? Thử đưa ra phương án đọc số hoàn toàn? (Ví dụ: với số 1304 đọc là: một nghìn ba trăm linh tư?)
3. Viết chương trình tính giá trị biểu thức:

$$F(x) = (x^2+e^{|x|}+\sin^2(x))/\sqrt[5]{x^2+1}$$

CHƯƠNG II: CÁC CẤU TRÚC ĐIỀU KHIỂN

1. Viết chương trình nhập vào một số nguyên n. Kiểm tra xem n chẵn hay lẻ.
2. Viết chương trình giải và biện luận phương trình bậc nhất theo hai hệ số a, b nhập từ bàn phím.
3. Viết chương trình giải và biện luận phương trình bậc hai với các hệ số a, b, c nhập từ bàn phím.
4. Viết chương trình giải và biện luận hệ phương trình bậc nhất 2 ẩn bằng phương pháp định thức?
5. Viết chương trình nhập vào số tiền phải trả của khách hàng. In ra số tiền khuyến mại với quy định: nếu số tiền phải trả thuộc [200.000, 300.000) thì khuyến mại 20%. Nếu số tiền phải trả từ 300.000 trở lên thì khuyến mại 30%. Còn lại thì không khuyến mại.
6. Viết chương trình nhập vào điểm tổng kết của một học sinh và in ra xếp loại cho học sinh đó với quy định:
   - Xếp loại giỏi nếu tổng điểm từ 8.00 trở lên.
   - Xếp loại khá nếu tổng điểm từ 7.00 tới cận 8.00.
   - Xếp loại trung bình nếu tổng điểm từ 5.00 tới cận 7.00.
   - Còn lại, xếp loại yếu.

---

7. Viết chương trình nhập vào một tháng của một năm bất kỳ (dương lịch), sau đó in ra số ngày có trong tháng.

---

8. Viết chương trình tính n!. Hãy tìm các cách khác nhau để giải quyết bài toán.
9. Nhập vào một số nguyên n. Tính tổng các số nguyên tố trong đoạn [n, 2n]. Đánh giá độ phức tạp của giải thuật trong trường hợp tồi nhất?
10. Viết chương trình nhập vào một số nguyên n, sau đó tính giá trị biểu thức:

$$S = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n}$$

11. Viết chương trình nhập vào một số nguyên n, sau đó tính giá trị biểu thức

$$F = \begin{cases} 1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + ... + \frac{1}{2^n} & \text{nếu n chẵn} \\ \sqrt{n^2 + 1} & \text{nếu n lẻ} \end{cases}$$

12. Viết chương trình nhập vào một số thực x và số nguyên n, sau đó tính giá trị biểu thức:

$$S = \begin{cases} x + \frac{x^2}{3} + \frac{x^3}{3^2} + ... + \frac{x^n}{3^{n-1}} & \text{nếu n chẵn} \\ 0 & \text{nếu n lẻ} \end{cases}$$

13. Viết chương trình nhập vào một số nguyên n trong khoảng [10, 20] (nếu số nhập vào không thuộc khoảng đó thì yêu cầu nhập lại tới khi thoả mãn). Sau đó tính tổng các số liên tiếp từ 1 tới n.
14. Viết chương trình nhập vào một số nguyên dương n, sau đó tính tổng các giá trị chẵn, lẻ thuộc đoạn [1, n].
15. Viết chương trình nhập vào các số nguyên dương n, m, sau đó in ra:
    - Tổng các số chẵn dương trong khoảng [- n, m].
    - Tổng các số chẵn âm trong khoảng [- n, m].
    - Tổng các số lẻ dương trong khoảng [- n, m].
    - Tổng các số lẻ âm trong khoảng [- n, m].

    Hãy thực hiện chương trình bằng hai cách và đánh giá mỗi cách.
16. Viết chương trình nhập vào một số nguyên n, sau đó tính tổng các số nguyên tố thuộc đoạn [1..n]. Cho biết có bao nhiêu số nguyên tố thuộc đoạn đó.

17. Dùng while (sau đó viết lại, dùng do/ while) để viết chương trình in ra số là luỹ thừa 2 bé nhất lớn hơn 1000.

18. Cho dãy số x[] = { 12.3, -45.4, 12, 15, 10.1, 12.5}. Viết chương trình đảo ngược dãy số trên. Đánh giá độ phức tạp của giải thuật đảo ngược dãy số bất kỳ có n phần tử trong trường hợp tồi nhất?

19. Viết chương trình tìm số nguyên dương n nhỏ nhất thoả mãn: 1 + 2 + 3 + … + n > 1000.

20. Để tính căn bậc hai của một số dương a, ta sử dụng công thức lặp sau:

    x(0) = a;

    x(n+1) = (x(n) * x(n) + a)/ (2* x(n)) với n >=0.

    Quá trình lặp kết thúc khi abs((x(n+1) – x(n))/x(n)) < ε.

    và khi đó x(n+1) được xem là giá trị gần đúng của sqrt(a).

    Viết chương trình tính căn bậc hai của a với độ chính xác ε = 0.00001.

21. Lập trình để tính sin(x) với độ chính xác ε = 0.00001 theo công thức :

    $\sin(x) = x - x^3/3! + x^5/5! + \ldots + (-1)^n x^{(2n+1)}/(2n+1)!$.

22. Lập trình để tính tổ hợp chập m của n theo công thức:

    C(m, n) = (n(n-1)…(n-m+1))/ m!.

CHƯƠNG III: KỸ THUẬT LẬP TRÌNH ĐƠN THỂ

1. Viết hàm kiểm tra xem một số nguyên n có phải là số nguyên tố không. Sau đó, trong chương trình chính, nhập vào một số nguyên n, kiểm tra tính nguyên tố của số n và thông báo ra màn hình? Mở rộng bài toán bằng cách sử dụng hàm trên để tính tổng các số nguyên tố trong đoạn [1, n]?

2. Viết hàm tính n! sau đó, trong chương trình chính, nhập vào một số nguyên n và tính, in ra kết quả của biểu thức:

$$S = \frac{n!+1}{(n+1)!}$$

3. Viết hàm tính giá trị biểu thức F (trong bài số 10 chương II) với đối vào là n. Sau đó, trong chương trình chính, nhập vào hai số a, b, tính và in ra màn hình kết quả của biểu thức:

$$S = \frac{F(a)-F(b)}{F(a-b)}$$

4. Viết hàm sắp xếp một chuỗi ký tự (từ A->Z). Sau đó, trong chương trình chính, nhập vào một xâu ký tự bất kỳ, in xâu đã được sắp lên màn hình.
5. Viết chương trình giải phương trình trùng phương : $ax^4 + bx^2 + c = 0$.
6. USCLN của hai số a, b được định nghĩa như sau:

USCLN(a, b) = a nếu b = 0

= USCLN(b, a%b) nếu b <> 0

Viết hàm đệ quy tìm USCLN của hai số nguyên a, b. Trong chương trình chính, nhập vào hai số nguyên a, b. Tìm và in USCLN của hai số đó lên màn hình.

7. Viết hàm tìm kiếm đệ quy trên một dãy số nguyên đã được sắp.
8. Các số Fibonacci F[i] được định nghĩa đệ quy như sau:

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

F[i] = F[i-1] + F[i-2] (với i > 1);

(VD: 1, 1, 2, 3, 5, 8, 13…)

Viết hàm đệ quy tìm số Fibonacci thứ n trong dãy.

(Bài toán này có thể phát biểu cách khác như sau: *có một cặp thỏ con gồm 1 thỏ đực và một thỏ cái. Thỏ con bắt đầu để sau khi nuôi được hai tháng. Mỗi lần để chỉ được 1 cặp (cũng gồm một thỏ đực và một thỏ cái). Mỗi tháng thỏ để một lần. Hỏi sau 7 (hoặc 8, hoặc 9...) tháng ta có mấy đôi thỏ – giả định trường hợp lý tưởng thỏ không bị chết và đôi nào cũng để*).

9. Viết hàm đệ quy tính n!.
10. Viết hàm đệ quy tính $f(x, n) = x^n$.
11. Viết hàm đệ quy tính giá trị của biểu thức: $F(x, n) = x^n / n!$
12. Viết hàm đệ quy tính số chữ số trong 1 số nguyên? (ví dụ số 1423 có 4 chữ số)
13. Viết hàm đệ quy tìm số lớn nhất trong một dãy số n phần tử?

CHƯƠNG IV: KỸ THUẬT LẬP TRÌNH DÙNG MẢNG.

1. Viết chương trình nhập vào một mảng n số nguyên, sắp xếp mảng theo chiều tăng dần, in kết quả lên màn hình.
2. Viết chương trình nhập vào một mảng n số nguyên, tính tổng các phần tử chẵn, các phần tử lẻ, các phần tử chia hết cho 3 và in kết quả ra màn hình.

3. Viết chương trình nhập vào một dãy số thực, tìm phần tử lớn nhất (tương tự, tìm phần tử nhỏ nhất) của dãy và in kết quả ra màn hình.
4. Viết chương trình nhập vào một dãy số nguyên. Tính tổng của các số nguyên tố trong dãy và in kết quả ra màn hình.
5. Viết chương trình nhập vào một dãy số nguyên và một số nguyên c. Đếm số lần xuất hiện và vị trí xuất hiện của c trong dãy. In các kết quả ra màn hình.
6. Viết chương trình nhập vào một dãy n số nguyên. Tính trung bình cộng của dãy và in kết quả tính được ra màn hình.
7. Một dãy số a gọi là được sắp tăng nếu a[i] <= a[i+1] với mọi i;

   Dãy gọi là được sắp giảm nếu a[i] >= a[i+1] với mọi i;

   Dãy gọi là được sắp tăng ngặt nếu a[i] < a[i+1] với mọi i;

   Dãy gọi là được sắp giảm ngặt nếu a[i] > a[i+1] với mọi i;

   Viết chương trình nhập một dãy n số thực, kiểm tra xem dãy đã được sắp hay chưa. Nếu đã được sắp thì sắp theo trật tự nào (tăng, tăng ngặt, giảm, giảm ngặt?). Nếu chưa thì sắp xếp dãy theo chiều tăng dần. In các kết quả lên màn hình.
8. Cho hai vector x(x_1, x_2…x_n) và y(y_1, y_2…$y_{n)}$. Viết chương trình in ra Tích vô hướng của hai vector trên.
9. Cho hai mảng a và b có các phần tử đều đã được sắp tăng. Lập chương trình trộn hai mảng trên để thu được một mảng thứ 3 cũng sắp theo thứ tự tăng (bằng 2 cách). Hãy đưa ra phương án cải tiến 2 cách trên?
10. Cho một mảng a gồm n phần tử nguyên sao cho a[i] thuộc [1, n] và không có giá trị nào xuất hiện quá 1 lần trong mảng. Chỉ bằng 1 vòng lặp (For(int i=0; i<n; i++) hãy sắp mảng a theo chiều tăng dần (hoặc giảm dần). Có nhận xét gì về bài tập này?
11. Nhập một xâu ký tự vào biến S. Một đường đi trong xâu là dãy liên tiếp các ký tự giống nhau trong S (không phân biệt chữ hoa và chữ thường), độ dài của đường đi là số ký tự có trong đường đi. Hãy cho biết độ dài của đường đi dài nhất trong S?
12. Cho một biểu thức gồm toàn các dấu mở/ đóng ngoặc '(' và ')'. Một biểu thức được gọi là hợp lệ nếu các dấu mở/ đóng ngoặc được đặt phù hợp như khi nó đặt trong biểu thức toán học. Ví dụ biểu thức: (( )( )) hoặc ((( )))( ) là hợp lệ, biểu thức)( )) hoặc ((( ))…là không hợp lệ. Hãy cho biết biểu thức vừa nhập có hợp lệ không?

13. Một mảng a gồm n phần tử nguyên được gọi là hợp lệ nếu tất cả các phần tử có chỉ số lẻ đều nguyên tố. Hãy kiểm tra tính hợp lệ của mảng a?

14. Nhập vào một mảng a chỉ gồm các phần tử 0 hoặc 1. Một đường đi trên a là một dãy liên tiếp các phần tử 1. Độ dài của đường đi là số phần tử trên đường đi đó. Hãy cho biết:

    - Mảng vừa nhập có bao nhiêu đường đi?
    - Mảng vừa nhập có đường đi dài nhất xuất phát từ vị trí nào?
    - Độ dài của đường đi dài nhất trong mảng?
    - Độ dài trung bình của các đường đi trong mảng?

12. Viết chương trình nhập vào một ma trận m x n số nguyên. Tìm các phần tử lớn nhất và bé nhất trên các dòng (tương tự các cột) của ma trận. (sử dụng for sau đó dùng while, do/ while).

15. Viết chương trình tìm phần tử âm đầu tiên trong ma trận (theo chiều từ trái qua phải, từ trên xuống dưới).

16. Viết chương trình nhập vào một ma trận m x n số nguyên. Tìm phần tử lớn nhất (tương tự tìm phần tử nhỏ nhất) của ma trận vừa nhập. In kết quả ra màn hình.

17. Viết chương trình nhập vào hai ma trận A, B có n hàng, m cột. Tính ma trận C = A + B và in kết quả ra màn hình.

18. Viết chương trình nhập vào hai ma trận A, B, tính và in ra màn hình tích của hai ma trận đó.

19. Viết chương trình nhập vào một ma trận A có n dòng, m cột. In ra màn hình ma trận chuyển vị của A. (A' được gọi là ma trận chuyển vị của A nếu A'[i, j] = A[j, i] với mọi i, j).

20. Ma trận A được gọi là đối xứng qua đường chéo chính nếu A[i, j] = A[j, i] với mọi i khác j. Viết chương trình nhập vào một ma trận A, kiểm tra xem A có đối xứng qua đường chéo chính không. In kết luận lên màn hình.

21. Một ma trận số, vuông a được gọi là hợp lệ nếu tất cả các phần tử nằm trên đường chéo chính bằng 1, tất cả các phần tử nằm phía trên đường chép chính đều dương, tất cả các phần tử nằm phía dưới đường chéo chính đều âm. Hãy kiểm tra xem a có hợp lệ không?

CHƯƠNG V: KỸ THUẬT LẬP TRÌNH DÙNG CON TRỎ

1. Viết chương trình nhập vào một mảng a gồm n phần tử nguyên. Sắp xếp mảng theo chiều giảm dần (lưu ý sử dụng tên mảng như con trỏ và sử dụng con trỏ).
2. Hãy dùng một vòng for để nhập vào một ma trận vuông cấp n với các phần tử thực và tìm phần tử Max của ma trận này.
3. Viết hàm hoán vị hai biến thực a, b bằng cách sử dụng con trỏ (đối vào là hai con trỏ). Viết chương trình chính nhập hai số thực a, b. Sử dụng hàm trên để đổi chỗ a và b.
4. Viết hàm giải hệ phương trình bậc nhất với sáu đối vào là a, b, c, d, e, f và 2 đối ra là x và y.
5. Viết hàm tính giá trị đa thức:

   $f(x) = a_0x^n + \ldots + a_{n-1}x + a_n$. với đối vào là biến nguyên n và mảng thực a.
6. Viết hàm cộng hai ma trận vuông a và b cấp n (sử dụng con trỏ).
7. Viết hàm trả về đồng thời 3 giá trị là tổng chắn, tổng lẻ và tổng các số chia hết cho 3 có trong mảng a gồm n phần tử nguyên. Sử dụng hàm trên trong chương trình chính.
8. Viết chương trình tính tích phân của f(x) trên đoạn [a, b] bằng công thức hình thang. Theo đó, tích phân của f(x) trên [a, b] bằng: h * s. Trong đó:

h là độ dài khoảng phân hoạch đoạn [a, b] thành n khoảng.

s là tổng tất cả các f(a+i*h) với i từ 1 tới n.

Sử dụng hàm trên để tính tích phân trong đoạn [-1, 4] của:

$f(x) = (e^x-2\sin(x^2))/ (1+x^4)$. (nghiên cứu cách đưa con trỏ vào giải quyết bài toán).

MỘT SỐ CÂU HỎI VỀ MẢNG

Đề 1. Nhập vào một mảng a gồm n phần tử nguyên. In ra màn hình phần tử âm đầu tiên trong dãy (tính từ trái qua phải) và vị trí của phần tử âm đó (nếu có). Nếu mảng không có phần tử âm nào, hãy tính trung bình cộng các phần tử dương trong mảng và in kết quả ra màn hình.

Đề 2. Viết chương trình nhập vào một dãy số nguyên. Tính tổng của các số nguyên tố trong dãy và in kết quả ra màn hình. Nếu mảng không có số nguyên tố nào, hãy sắp mảng tăng dần bằng phương pháp nổi bọt.

Đề 3. Viết chương trình nhập vào một dãy số nguyên và một số nguyên c. Đếm số lần xuất hiện và vị trí xuất hiện của c trong dãy. In các kết quả ra màn hình. Nếu c không xuất hiện trong mảng, hãy chèn c vào giữa mảng.

Đề 4. Nhập một ma trận vuông n x n phần tử thực. Gọi P[i] là trung bình cộng của các phần tử trong dòng thứ i của ma trận và K là trung bình cộng của tất cả các phần tử trong ma trận. Hãy tính và in ra P[i] và K.

Đề 5. Nhập một ma trận vuông n x n phần tử nguyên. Kiểm tra xem ma trận vừa nhập có đối xứng không? nếu đối xứng, hãy in ra các phần tử trên đường chéo chính của ma trận ra màn hình đồng thời tính và in ra tổng các phần tử nằm phía trên đường chéo chính.

Đề 6. Nhập một mảng a gồm n phần tử nguyên, hãy đảo ngược mảng a và in mảng đã đảo ngược ra màn hình. Sắp sếp lại mảng a theo chiều tăng dần và in ra phần tử lớn nhất trong mảng a.

Đề 7. Nhập một mảng a gồm n phần tử nguyên. hãy sắp xếp mảng a sao cho: các phần tử lớn nhất ở đầu mảng, các phần tử bé nhất ở cuối mảng, các phần tử còn lại sắp tăng dần. In mảng đã sắp ra màn hình.

Đề 8. Nhập một mảng a gồm n phần tử nguyên. Mảng a được gọi là hợp lệ nếu tồn tại 3 phần tử liên tiếp đều là các phần tử lẻ. Hãy kiểm tra xem a có hợp lệ không? nếu không hợp lệ hãy dồn tất cả các phần tử lẻ của a lên đầu mảng.

Đề 9. Nhập một mảng a gồm n phần tử nguyên. Sắp a theo chiều tăng dần bằng phương pháp chọn. Xoá mọi phần tử lẻ trong mảng a và in mảng kết quả lên màn hình.

Đề 10. Nhập một mảng a gồm n phần tử nguyên. Mảng a được gọi là hợp lệ nếu nó chứa đúng 3 phần tử dương và 3 phần tử âm. Hãy cho biết a có hợp lệ không? nếu a không hợp lệ hãy sắp a theo chiều tăng dần bằng phương pháp chèn.

Đề 11. Nhập một ma trận a gồm n dòng, m cột. Hãy tính tổng các phần tử dương trên ma trận và in kết quả ra màn hình. Hãy tính tổng của các phần tử xung quanh ma trận đồng thời in ma trận vừa nhập ra màn hình.

Đề 12. Nhập một ma trận vuông n x n phần tử nguyên. Ma trận được gọi là hợp lệ nếu tổng các phần tử trên mỗi dòng của tất cả các dòng đều bằng nhau. Hãy

kiểm tra xem ma trận vừa nhập có hợp lệ không? In mảng vừa nhập ra màn hình.

Đề 13. Nhập một mảng a gồm n phần tử nguyên. Hãy tách các phần tử chẵn trong a ra một mảng b, tách các phần tử lẻ trong a ra một mảng c. In cả 3 mảng ra màn hình.

Đề 14. Nhập một mảng a gồm n phần tử nguyên. Hãy sắp xếp mảng a sao cho các phần tử lớn nhất về cuối dãy, các phần tử còn lại được sắp giảm dần. In kết quả ra màn hình. Cho biết mảng vừa sắp có bao nhiêu phần tử nhỏ nhất?

Đề 15. Nhập một ma trận vuông a gồm n x n phần tử nguyên. Ma trận được gọi là hợp lệ nếu tất cả các dòng của nó, mỗi dòng chỉ chứa đứng 1 phần tử âm. Hãy kiểm tra xem ma trận vừa nhập có hợp lệ không? in mảng vừa nhập ra màn hình.

MỤC LỤC

CHƯƠNG I. TỔNG QUAN VỀ C++ 2
I. QUY TRÌNH LÀM VIỆC TRONG C++ 2
I.1. Các bước để lập một trương trình bằng C++ 2
I.2. Cấu trúc một chương trình đơn giản trong C++ 3
II. BIẾN, BIỂU THỨC, CÁC LỆNH NHẬP XUẤT 5
II.1. Biến 5
II.2. Biểu thức 5
II.3. Các lệnh nhập-xuất 8
a. Các lệnh nhập xuất trong IOStream.h 8
b. Các lệnh nhập xuất trong Stdio.h 10
c. Các lệnh nhập xuất trong Conio.h 11
CHƯƠNG II. CÁC CẤU TRÚC ĐIỀU KHIỂN TRONG C++ 13
I. CẤU TRÚC RẼ NHÁNH VÀ CẤU TRÚC CHỌN 13
I.1. Cấu trúc rẽ nhánh 13
I.2. Cấu trúc chọn 16
II. CẤU TRÚC LẶP 18
II.1. Vòng lặp với số lần lặp xác định 18
II.2. Vòng lặp với số lần lặp không xác định 22
a. Lặp kiểm tra điều kiện trước: 22
b. Lặp kiểm tra điều kiện sau: 24
II.3. Các ví dụ minh hoạ sử dụng vòng lặp 26
CHƯƠNG III. KỸ THUẬT LẬP TRÌNH ĐƠN THỂ 28
I. ĐƠN THỂ VÀ LẬP TRÌNH ĐƠN THỂ 28
I.1. Khái niệm và phân loại đơn thể 28
I.2. Định nghĩa và sử dụng hàm 29
I.3. Tổ chức các hàm 32
I. 4. Phạm vi hoạt động của biến 36
II. KỸ THUẬT ĐỆ QUY 37
II.1. Khái niệm về đệ quy 37

II.2. Thiết kế hàm đệ quy 39
II.3. Đệ quy và các dãy truy hồi 41
II.4. Một số ví dụ về đệ quy 42
III. KỸ THUẬT TRUYỀN THAM SỐ 43
III.1. Khái niệm và phân loại tham số 43
III.2. Truyền tham số 44
CHƯƠNG IV. KỸ THUẬT LẬP TRÌNH DÙNG MẢNG 47
I. MẢNG MỘT CHIỀU 47
I.1. Khai niệm và cách khai báo 47
I.2. Các thao tác cơ bản trên mảng một chiều 48
I.3. Các bài toán cơ bản 48
a. Bài toán sắp xếp mảng 48
b. Bài toán tìm kiếm 54
II. MẢNG HAI CHIỀU 58
II.1. Các thao tác cơ bản trên mảng hai chiều 58
II.2. Các bài toán cơ bản trên mảng 2 chiều 60
III. XÂU KÝ TỰ – MẢNG CÁC KÝ TỰ 62
III.1. Một số lưu ý khi sử dụng xâu ký tự 62
III.2. Một số bài toán đặc thù trên xâu 65
CHƯƠNG V. KỸ THUẬT LẬP TRÌNH DÙNG CON TRỎ 68
I. TỔNG QUAN VỀ CON TRỎ 68
I.1. Khái niệm và cách khai báo 68
I.2. Một số thao tác cơ bản trên con trỏ 68
II. CON TRỎ - MẢNG VÀ HÀM 69
II.1. Con trỏ và mảng 69
II.2. Con trỏ và hàm 70
III. CẤP PHÁT VÀ GIẢI PHÓNG BỘ NHỚ CHO CON TRỎ 73
III.1. Cấp phát bộ nhớ động cho con trỏ 73
III.2. Cấp phát lại hoặc giải phóng bộ nhớ cho con trỏ 75
MỘT SỐ BÀI TẬP THỰC HÀNH 78