HỌC VIỆN KỸ THUẬT QUÂN SỰ
KHOA CÔNG NGHỆ THÔNG TIN
TS. DƯƠNG TỬ CƯỜNG

Ngôn ngữ lập trình C

HỌC VÀ SỬ DỤNG

NHÀ XUẤT BẢN
KHOA HỌC VÀ KỸ THUẬT

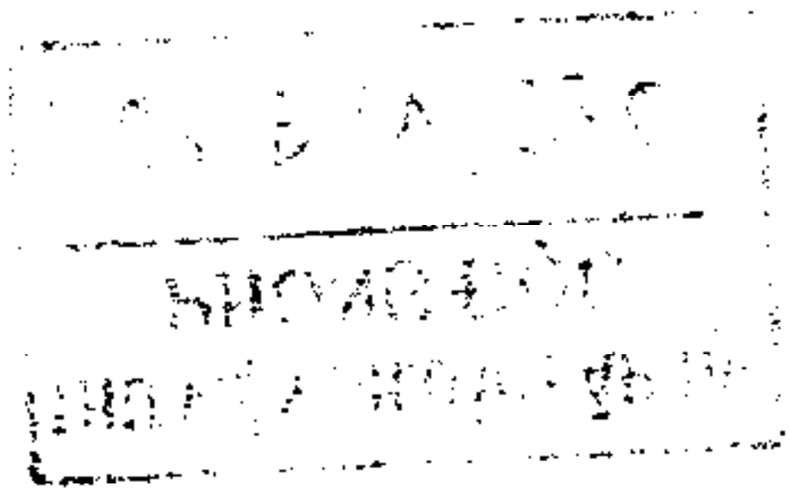
HỌC VIỆN KỸ THUẬT QUÂN SỰ
KHOA CÔNG NGHỆ THÔNG TIN

TS. DƯƠNG TỬ CƯỜNG

NGÔN NGỮ LẬP TRÌNH C
HỌC VÀ SỬ DỤNG

(In lần thứ ba có chỉnh lý)

NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ THUẬT
HÀ NỘI – 2001



MỞ ĐẦU

Có thể nói ngôn ngữ C là một ngôn ngữ "không truyền thống" theo sự phát sinh và quá trình phát triển của mình. Lịch sử phát triển của ngôn ngữ C được gắn liền với hệ điều hành *UNIX*. Đến lượt mình hệ điều hành *UNIX* lại liên quan chặt chẽ đến một hệ điều hành khác - hệ điều hành *MULTICS*, một hệ điều hành với sự phân chia thời gian. Hệ điều hành *MULTICS* là kết quả hợp tác của *AT & T Bell Laboratories, General Electric và Massachusetts Institute of Techonology*. Trên thực tế nguyên mẫu của hệ thống *MULTICS* đã đáp ứng được nhu cầu đặt ra nhưng lại tỏ ra chưa thuận tiện và khó khăn trong việc sử dụng. Đây là một trong các nguyên nhân để *AT & T* đình chỉ không tiếp tục phát triển hệ thống hơn nữa. Ảnh hưởng bởi ý tưởng đầu tiên của *MULTICS*, *Ken Thomcon* - một trong những người tham gia thiết kế hệ thống này đã xây dựng một hệ điều hành khác, mà một trong những điều khác biệt với *MULTICS* là sự thuận tiện và dễ dàng trong khi sử dụng. Hệ điều hành này là phiên bản đầu tiên của *UNIX* và được *Thomcon* viết trên ngôn ngữ *Assemblly* dành cho máy PDP-7 vào năm 1969. Mặc dù hệ điều hành này ban đầu còn chưa cho phép làm việc trong chế độ đa người sử dụng nhưng nó đã tỏ ra có hiệu quả, thuận tiện và xứng đáng được phát triển trên những máy tính điện tử khác. Để thực hiện được điều này, năm 1970 Thomcon đã xây dựng một ngôn ngữ hệ thống (thông dịch) với tên gọi là ngôn ngữ *B*. Khi xây dựng ngôn ngữ này, ông đã xuất phát từ một ngôn ngữ khác - ngôn ngữ BCPL được viết vào năm 1969 bởi *Martin Richards*. Với hệ thông dịch này hệ điều hành *UNIX* đã được đưa sang máy vi tính *PDP-11* vào năm 1971. Trong cùng giai đoạn này, Dennis Ritchie - một nhà lập trình nổi tiếng đã tham gia vào tập thể

các nhà lập trình của hệ thống trên. Năm 1972 Dennis Ritchie cùng các cộng sự đã phát triển ngôn ngữ *B* thành ngôn ngữ *C*. Bằng cách này cặp *UNIX - C* đã tỏ rõ những khả năng hiếm có của mình trong kỹ thuật lập trình. Với ngôn ngữ *C* hệ điều hành *UNIX* đã được xây dựng vào năm 1973 và được xem như hệ điều hành tương thích *(Portable Operating System)* đa người sử dụng. Năm 1975 phiên bản thứ 6 của hệ điều hành *UNIX* bắt đầu được phổ biến rộng rãi. Một điểm đặc biệt cần lưu ý là hầu như toàn bộ hệ điều hành *UNIX*, bao gồm các trình điều khiển thiết bị, trình biên dịch *C* đều được viết trên ngôn ngữ *C*. Trong thời gian này ngôn ngữ *C* cũng bắt đầu được phổ biến và bán trên thị trường. Cho đến năm 1978 khi *Brian Kernighan* và *Dennis Ritchie* cho xuất bản cuốn *The C Programming Language* thì có thể xem như *C* chính thức được ra đời.

Cùng với sự ra đời và phát triển của kỹ thuật vi mạch (các hệ vi xử lý *8080* và *Z80*) nhiều trình biên dịch của C đã ra đời và phần nào đã đáp ứng được nhu cầu của thực tế về tốc độ, khả năng biên dịch v.v. Ngày nay tối thiểu ta có thể đếm được đến 17 trình biên dịch C đang thực sự phổ biến trên thị trường PC.

Ngôn ngữ C đang là một trong các ngôn ngữ được đánh giá cao trên thực tế do nhiều nguyên nhân:

• Khả năng tương thích trên nhiều loại máy khác nhau có thể xem là ưu điểm được quảng cáo nhiều nhất của ngôn ngữ C. Một chương trình được viết tuân thủ theo các nguyên tắc của C và tránh sử dụng các đặc tính mở rộng của một thư viện phụ thuộc trình biên dịch đặc biệt, có thể được xem như một chương trình có khả năng thành công nhiều nhất khi mang sang sử dụng trong các môi trường trình biên dịch/hệ điều hành máy tính khác. Đây là một đặc tính được đánh giá ngày càng cao của ngôn ngữ C.

• Một ưu điểm thứ hai của C là sự súc tích và cô đọng của ngôn ngữ. Cú pháp của ngôn ngữ C ban đầu rất đơn giản và chỉ dựa trên 27 từ khóa. Điều này làm cho ngôn ngữ có thể được phát triển và thực hiện một cách dễ dàng ở trên máy tính lớn và máy vi tính.

• Ngôn ngữ C đưa ra những khả năng cao cho lập trình có cấu trúc, một cơ cấu truyền đạt, truyền thông (Communication) thuận tiện giữa các đơn vị chương trình khác nhau, khả năng biên dịch độc lập, tính đệ qui v.v...

• C còn được chú ý đến là do hiệu quả của việc kết sinh mã (Code Generation). Điều này bắt nguồn từ việc ngôn ngữ C đi sát với cấu trúc ký ức và thanh ghi của phần cứng, yếu tố mà dựa trên ngôn ngữ được thiết kế và phát triển. Người ta thường xem C như là một ngôn ngữ cấp cao như Pascal, Fortran v.v... bởi sự thuận tiện khi sử dụng và phương pháp lập trình có cấu trúc của nó. Tuy vậy ngoài những chức năng bậc cao đó, C lại cho phép thực hiện nhiều công việc ở "mức thấp" như ngôn ngữ *Assembly*. Các chuyên gia đã thống kê được rằng có đến 90% các trường hợp có thể sử dụng ngôn ngữ C thay thế cho ngôn ngữ Assembly, nhất là trong các lĩnh vực liên quan đến phần cứng và các bài toán điều khiển. Điều đó chứng tỏ rằng với các khả năng của một ngôn ngữ bậc cao, C có thể đạt đến kết quả đặc trưng cho ngôn ngữ *Assembly* trong điều kiện thuận tiện, dễ dàng hơn khi xây dựng và sửa đổi chương trình.

Mặt dù hiện nay trên thị trường đã tồn tại nhiều tài liệu viết về ngôn ngữ C nhưng có thể nói tài liệu này được biên soạn theo một phong cách khác hẳn. Một trong những đặc trưng đó là *tính hệ thống, đầy đủ* và được xây dựng theo mức độ *từ thấp đến cao* và do đó có thể được sử dụng bởi nhiều đối tượng với trình độ và khả năng lập trình khác nhau.

Các khái niệm và kiến thức trong tài tiệu được minh họa với

nhiều ví dụ, chương trình khác nhau. Các ví dụ được trình bày theo một ý tưởng thống nhất: mô tả các vấn đề đặt ra, các đại lượng cần sử dụng trong chương trình, giới thiệu thuật toán (nếu cần), Listing của chương trình, giải thích v.v... Trên thực tế khả năng lập trình không thể tách rời khỏi thực hành nên chúng tôi đề nghị các bạn nghiên cứu kỹ các ví dụ được đưa ra trong tài liệu.

Mục đích chính của tài liệu là cung cấp cho bạn đọc một cách hệ thống và đầy đủ nhất những vấn đề cơ bản liên quan đến ngôn ngữ C. Chính vì lý do này trong tài liệu không đề cập đến một số vấn đề nâng cao của C như đồ họa, truy nhập đến phần cứng của máy, các vấn đề về files cũng chỉ được đề cập ở mức độ cơ bản v.v... Tuy vậy, chúng tôi hy vọng rằng tài liệu sẽ là nền móng chắc chắn và tạo điều kiện dễ dàng để bạn đọc nghiên cứu các vấn đề chưa được đề cập đến ở giai đoạn tiếp theo.

Trong tài liệu có sử dụng các thuật ngữ và các ký hiệu chuẩn được công nhận và phổ biến chính thức trong các tài liệu viết về ngôn ngữ C. Các khái niệm và kiến thức về ngôn ngữ C được đưa ra trong tài liệu này mang ý nghĩa chung mà không phụ thuộc vào các khả năng riêng biệt của từng loại máy khác nhau.

Hiện nay, trên thực tế đang phổ biến nhiều trình biên dịch cao cấp được sử dụng trong môi trường *DOS* và *Windows* nhưng những vấn đề được đưa ra trong tài liệu là những kiến thức bạn không thể thiếu khi tiến tới làm việc với những phiên bản cao cấp của ngôn ngữ C.

Chúc các bạn thành công.

Chương I

NHỮNG KHÁI NIỆM CƠ BẢN CỦA NGÔN NGỮ C

Trong lĩnh vực tin học có một câu ngạn ngữ rất bổ ích dành cho các bạn muốn đi sâu vào kỹ thuật lập trình: "Máy tính chỉ thực hiện những điều mà con người mong muốn"... Chính vì vậy để máy tính thực hiện được những điều bạn muốn, bạn phải chỉ thị chính xác các thông tin cho máy. Thông thường các thông tin được đưa vào máy thông qua một chương trình viết bằng một ngôn ngữ lập trình nào đó. Để đạt được điều này người lập trình phải nắm chắc những khái niệm cơ bản, những nguyên tắc lập trình của ngôn ngữ. Một chương trình được tạo ra bao gồm các chỉ thị (các lệnh của máy) nhằm thực hiện một mục đích nào đó và thường được thiết lập dựa trên cơ sở của một thuật toán đã lập trước. Thuật toán này diễn tả các hành động khác nhau để đi đến mục đích cuối cùng.

Chương I đưa ra những khái niệm cơ bản của ngôn ngữ C - hằng số, chú giải, các kiểu dữ liệu, mảng và khởi tạo dữ liệu. Trong chương này cũng đưa ra những thông tin chung nhất về hàm, cấu trúc và cách thực hiện một chương trình viết bằng ngôn ngữ C. Để thuận tiện cho việc theo dõi các ví dụ trong chương trình, trong phần này chúng tôi xin giới thiệu một số hàm kết xuất thông tin vào ra thường được sử dụng - *printf, scanf, getchar*. Những bạn đã có ít nhiều kinh nghiệm trong lập trình C có thể bỏ qua chương này nhưng chắc chắn rằng một lúc nào đó bạn cũng sẽ cần tham khảo một số thông tin ở đây.

1.1 CÁC KÝ HIỆU CƠ BẢN CỦA C

Việc nắm vững các ký hiệu cơ bản của ngôn ngữ là điều đầu tiên không thể bỏ qua đối với những ai muốn đi vào lĩnh vực

lập trình. Các ký hiệu cơ bản bao gồm những ký tự cho phép dùng trong ngôn ngữ như chữ cái, số và tổ hợp các ký tự khác. Ngôn ngữ C sử dụng một số ký tự sau:

- 52 chữ cái in thường, in hoa và ký tự gạch nối:

a, b, c, ... z, A, B, C, ... Z và -

- Các chữ số:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

- Các ký tự đặc biệt được dùng riêng trong ngôn ngữ như:

, . : ; ? | .( ) [ ] + - * / = % & ! ^ - ' " { } # $

- Một số ký tự khác như dấu cách (ký tự trắng); dấu xuống hàng, dấu canh theo cột (TAB) và chú giải. Dấu xuống hàng là ký hiệu hay tập hợp các ký hiệu được đặt ở cuối dòng hiện thời và có tác dụng di chuyển con trỏ xuống đầu dòng sau. Chũ giải được đưa vào chương trình nhằm mục đích giải thích thêm cho chương trình được rõ ràng hơn và sẽ được đề cập chi tiết trong phần *1.6*.

Trong C có một số từ được dùng riêng gọi là các từ khóa. Các từ này được dùng để viết và xây dựng các toán lệnh của chương trình. Cần chú ý rằng trong C các từ này luôn được viết với các chữ thường và không được dùng chúng cho các mục đích riêng như đặt tên cho các biến, các hàm...

Các từ khóa trong C chuẩn bao gồm

auto	**break**	**case**	**char**
continue	**default**	**do**	**double**
else	**entry**	**enum**	**extern**
float	**for**	**goto**	**if**
int	**long**	**register**	**return**
sizeof	**short**	**static**	**struct**
switch	**typedef**	**union**	**unsigned**
while	**void**	**asm**	**fortran**
pascal	**ada**		

1.2 HẰNG SỐ

Hằng số là các thông tin được đưa vào chương trình và có giá trị không đổi trong quá trình thực hiện chương trình. Hằng số bao gồm các số hoặc dãy ký tự có giá trị không đổi. Tồn tại 3 loại hằng trong C: hằng số học, hằng ký tự và hằng kiểu chuỗi.

Hằng số học được phân làm hai loại: *hằng số nguyên* và *hằng số thực*. Giá trị của các hằng này bằng giá trị của các số mà chúng biểu diễn.

Hằng số nguyên có thể được biểu diễn dưới dạng cơ số 10, cơ số 8 và cơ số 16.

Các hằng cơ số 10 có thể là số dương hoặc số âm. Dấu của hằng được viết trước số đó, trong trường hợp số dương, dấu có thể được bỏ qua. Cần chú ý là không được dùng số 0 như là chữ số đầu tiên của hằng số nguyên.

Ví dụ 1.1

+14 - Hằng số nguyên dương có giá trị là 14;

-12 - Hằng số nguyên âm có giá trị là -12;

012 - Không phải là số nguyên dương.

Các hằng số nguyên cơ số 8 được viết không có dấu bằng các chữ số sử dụng trong hệ đếm cơ số 8 - các số từ 0 đến 7. Chữ số đầu tiên trong trường hợp này phải là số 0. Đây chính là nguyên nhân không cho phép sử dụng số 0 như là chữ số đầu tiên của hằng cơ số 10.

Ví dụ 1.2

014 - Hằng cơ số 8 với giá trị 12 (cơ số 10);

0114 - Hằng cơ số 8 với giá trị 76 (cơ số 10);

Các hằng số nguyên cơ số 16 được viết không có dấu và

luôn bắt đầu bởi *0X* hoặc *0x*. Trong hệ đếm cơ số 16 người ta dùng các số từ 0 đến 9. Ngoài ra các số từ *A* đến *F* hoặc từ *a* đến *f* không phân biệt chữ thường hay chữ viết hoa biểu diễn các số từ 10 đến 15.

Ví dụ 1.3

0XC hoặc 0xc - Hằng cơ số 16 có giá trị 12 (cơ số 10);

0XFF hoặc 0xff - Hằng cơ số 16 có giá trị 255 (cơ số 10);

Thông thường các hằng số nguyên được lưu trữ vào 2 byte của máy tính. Các hằng này có thể nhận các giá trị nguyên trong khoảng -32768 đến +32767. Trong trường hợp chúng ta muốn sử dụng các hằng số với giá trị ở ngoài khoảng này ta cần biểu diễn chúng bằng kiểu *long* (4 byte). Ở dạng này, sau giá trị của các số ta phải đặt thêm ký tự *L* hay *l*, ví dụ: -50000L, -40000L, 0XFE00L. Thực ra cách biểu diễn này sẽ làm cho chương trình được viết trở thành rõ ràng hơn, trong trường hợp hằng số có giá trị vượt quá giá trị cho phép nhưng không được biểu diễn bằng kiểu *long* máy sẽ tự động chuyển chúng về kiểu này.

Hằng số thực là một loại hằng số học khác dùng để biểu diễn các số thực có giá trị không đổi và chỉ được viết dưới dạng cơ số 10. Một hằng số thực có thể được viết dưới dạng thập phân hay số mũ.

Ví dụ 1.4

12.0 - Hằng số thực dương có giá trị là 12.0;

0.123 - Hằng số thực dương có giá trị là 0.123;

-12.15 - Hằng số thực âm có giá trị là -12.15.

Các hằng số thực có giá trị rất lớn hay rất nhỏ được biểu diễn dưới dạng cơ số mũ.

Ví dụ 1.5

1.2E1 hoặc 1.2e1	- Hằng số thực với giá trị 12;
120E- hoặc e-1	- Hằng số thực với giá trị 12;
0.123E2	- Hằng số thực với giá trị 12.3.

Hằng ký tự bao gồm các ký tự đơn được viết trong dấu nháy đơn, ví dụ 'A', 'B'. Mỗi ký tự có giá trị bằng mã *ASCII* của nó (mã *ASCII* là số thứ tự tương ứng của ký tự trong bảng mã *ASCII*). Ví dụ theo bảng này '*A*' có giá trị là 65 và '0' có giá trị là 48.

Một ký tự có thể được biểu diễn bằng 2 cách: bằng tên của ký tự và bằng mã *ASCII* của ký tự. Chẳng hạn để mô tả hằng ký tự *A* ta có thể viết '*A*' hoặc '*65*'. Cần chú ý rằng mã *ASCII* trong trường hợp này phải được viết sau dấu \ và cũng phải đặt trong dấu nháy.

Khi sử dụng hằng ký tự ta cũng phải chú ý đến sự khác biệt giữa hằng số học và hằng ký tự. Một hằng số học có giá trị bằng giá trị của số mà nó biểu diễn, trong khi hằng ký tự có giá trị bằng mã *ASCII* của nó. Ví dụ hằng số *0* có giá trị là *0* còn hằng ký tự '*0*' có giá trị là 48.

Hằng kiểu chuỗi là một dãy ký tự liên tục được đặt trong dấu nháy kép", ví dụ "This is a string" là một hằng kiểu chuỗi. Độ dài của hằng kiểu chuỗi (số lượng các ký tự trong chuỗi) là không hạn chế và có thể chỉ chứa một ký tự hoặc không chứa bất cứ ký tự nào. Trong quá trình biên dịch, sau ký tự cuối cùng của hằng kiểu chuỗi, trình biên dịch tự động đặt thêm một byte có giá trị là *0* (*NULL*). Để phân biệt với ký tự '*0*' Byte này được viết dưới dạng '*0*'. Như vậy ký tự này được dùng để xác định vị trí cuối cùng của chuỗi.

Các ký tự dùng riêng (là những ký tự không in ra được) có thể được sử dụng như những ký tự bình thường nhưng trước chúng phải đặt ký tự \. Chẳng hạn ký tự \ có thể được sử dụng trong chuỗi nếu viết \\.

Ví dụ 1.6

"*ví dụ về \"hàng chuỗi kiểu chuỗi \" trong C*\\" sẽ biểu diễn cho hằng kiểu chuỗi *ví dụ về "hằng kiểu chuỗi" trong C*\ được viết trong chương trình.

Hằng kiểu chuỗi được lưu trữ trong bộ nhớ của máy tính như là một mảng một chiều với phần tử cuối cùng là \0 do trình biên dịch tự động đưa vào. Như vậy số phần tử của mảng luôn lớn hơn 1 so với số ký tự của chuỗi.

Ở đây ta cũng phải chú ý rằng có sự khác biệt giữa hằng ký tự và chuỗi ký tự, chẳng hạn *'A'* và *"A"*. Sự khác biệt này được xác định ở chỗ xâu ký tự *"A"* có thể được xem như là một mảng gồm hai ký tự *'A'* và *'\0'*.

Có một số ký tự thuộc kiểu không in ra được (có giá trị *ASCII* từ 0 đến 31) nhưng trình biên dịch có thể nhận biết được các ký tự này thông qua một cặp ký tự thông thường (cặp này luôn luôn bắt đầu bằng ký tự \). Các ký tự này thường được dùng để điều khiển màn hình hoặc máy in. Bảng *1.1* mô tả một số ký tự đặc biệt thường dùng:

Bảng 1.1

Tác dụng	*Mô tả trong C*	*Mã ASCII*	*Ký hiệu*
Byte trắng	\0	0	NUL (Null Byte)
Lùi	\b	8	BS (Backspace)
Trở về đầu dòng	\r	13	CR (Cariage)
Canh theo cột	\t	9	T (Horizontal tab)
Xuống dòng mới	\n	10	NL (New line)
Xuống trang mới	\f	12	FF (Form Feed)
Tín hiệu chuông	\a	7	Bel (Bell)
Dấu nháy đơn	\'	39	(Single quote)
Dấu nháy kép	\"	34	(Double quote)
Dấu hỏi	\?	63	?
Mã ký tự (bit patten)	\ooo		

Mã ký tự *ooo* được dùng để biểu diễn một ký tự trong bảng *ASCII* bằng các số trong hệ đếm cơ số 8. Trong trường hợp này mã ký tự bao gồm ký tự điều khiển \ và mã cơ số 8 của ký tự. Chẳng hạn hằng kiểu chuỗi "*{ABCD}*" có thể được viết dưới dạng "*\173ABC\175*".

Việc sử dụng các ký tự đặc biệt trong bảng *1.1* sẽ được giải thích cụ thể trong các phần sau của giáo trình. Ngoài các ký tự này, tất cả các ký tự khác sẽ không thay đổi ý nghĩa của chúng nếu được viết sau ký tự \. Nói cách khác '*k*' có thể được viết dưới dạng '*\k*'.

1.3 BIẾN

Tất cả các chương trình được viết trong máy tính đều nhằm mục đích cuối cùng là xử lý thông tin. Những thông tin có giá trị không đổi trong suốt quá trình thực hiện chương trình được biểu diễn qua các hằng *(phần 1.2)*. Các thông tin mà giá trị có thể thay đổi được gọi là *biến*.

Biến là một đại lượng thuộc một kiểu nhất định (số nguyên, số thực v.v... mà giá trị của nó có thể thay đổi trong quá trình thực hiện chương trình. Giá trị của biến được sử dụng thông qua tên của biến. Thực ra biến là một ô ký ức (hoặc một số ô ký ức) trong bộ nhớ dùng để lưu trữ giá trị tức thời của nó. Bởi vì tất cả các ô ký ức của bộ nhớ đều có một địa chỉ nhất định, do đó việc sử dụng tên của một biến hoàn toàn đồng nhất với việc truy nhập một địa chỉ trong bộ nhớ và giá trị chứa trong địa chỉ đó.

Khi đặt tên cho biến cần phải thông qua các quy tắc sau:

1. Tên bao gồm một dãy các chữ cái và số và phải bắt đầu bằng chữ cái. Có thể dùng ký tự gạch dưới (_) để đặt tên cho biến, ví dụ *ho_ten* là một tên cho phép. Cũng cần lưu ý rằng độ dài của các tên biến cũng phải hạn chế và điều đó hoàn toàn phụ thuộc vào các trình biên dịch khác nhau.

2. Tên của biến không được trùng với từ khóa. Ví dụ không được dùng *for, while, unsigned* v.v để đặt tên cho biến.

3. Trong ngôn ngữ C có sự khác biệt giữa chữ thường và chữ hoa. Như vậy *FOR, WHILE* có thể được dùng để đặt tên cho biến vì chúng không trùng với các từ khóa *for, while*. Cũng vậy hai biến *NAME* và *name* là hoàn toàn khác nhau.

Về nguyên tắc ta có thể dùng một tên bất kỳ để đặt cho biến, tuy vậy để chương trình được dễ hiểu thông thường ta nên dùng các tên có thể biểu diễn tính chất vật lý của một biến ở một mức độ nào đó.

Ví dụ 1.7

count - Tên của biến được sử dụng làm bộ đếm.

center x, center y - Tọa độ x, y của tâm vòng tròn.

1.4. CÁC LOẠI DỮ LIỆU VÀ CÁCH KHAI BÁO

Tất cả các biến trong ngôn ngữ C trước khi sử dụng đều phải được khai báo. Tồn tại nhiều kiểu biến trong C và qua việc khai báo ta đã định ra kích thước của biến, tức là số lượng byte cần dùng để lưu trữ giá trị của biến trong bộ nhớ. Việc khai báo biến có cú pháp sau:

Kiểu_biến Tên_biến;

Trong C sử dụng một số từ khóa để khai báo kiểu của biến. Các từ khóa đó là:

char - Khai báo cho dữ liệu kiểu ký tự;

int - Khai báo cho dữ liệu kiểu số nguyên;

float - Khai báo cho dữ liệu kiểu số thực;

double - Khai báo cho dữ liệu kiểu số thực với độ chính xác gấp đôi.

Ví dụ, nếu hai biến *i, j* sẽ nhận giá trị là các số nguyên hay

i, *j* là kiểu *int*, còn biến *x* nhận giá trị thực - *float* thì chúng sẽ được khai báo như sau:

```
int i, j;
float x;
```

Như được chỉ ra trong ví dụ trên, các biến có cùng kiểu có thể được khai báo sau từ khóa (kiểu của biến) và được viết cách nhau bởi dấu phẩy. Tất nhiên chúng cũng có thể được khai báo trên các dòng khác nhau, chẳng hạn:

```
int i;
int j;
float x;
```

Loại biến *int* có thể được khai báo chính xác hơn nếu ta dùng thêm các từ khóa sau:

short - Dùng để khai báo biến với kích thước nhỏ hơn loại *int* thường dùng. Trên các máy 16 bits, *short* có kích thước giống *int*;

long - Dùng để khai báo biến với kích thước có độ dài lớn gấp 2 lần.

unsigned - Dùng để khai báo biến với giá trị không âm.

Khi sử dụng các từ khóa trên, từ khóa *int* có thể được bỏ qua; trong trường hợp được sử dụng, từ khóa *int* phải được viết sau các từ khóa trên. Chẳng hạn:

```
short int i;        tương đương với short i;
long int j;         tương đương với long j;
unsigned int k;     tương đương với unsigned k;
```

Ngoài ra *long* có thể được đặt trước *float*, khi đó biến được khai báo theo kiểu *double* (chú ý trong trường hợp này từ khóa *float* không được bỏ qua).

Qua việc khai báo biến chúng ta đã dành riêng một số *byte* cần thiết trong máy để lưu giữ giá trị của biến và cũng thông

qua đó chúng ta đã xác định giá trị lớn nhất mà biến có thể nhận được. Kích thước của biến (tính bằng *byte*) phụ thuộc vào kiểu của biến, loại trình biên dịch và loại máy tính. Trong bảng 1.2 có đưa ra kích thước của một số kiểu biến cho từng loại máy tính.

Bảng 1.2

Kiểu biến	Kích thước tính bằng byte		
	Motorola 6800	Intel 80280	IBM 360/370
char	8	8	8
int	32	16	32
short	16	16	16
long	32	32	32
float	32	32	32
double	32	64	64

Đối với các loại máy 16 *bit* các biến có kích thước được chỉ ra trong bảng *1.3*.

Bảng 1.3

Kiểu	Kích thước (bit)	Giá trị
char	8	[0, +255]
int hoặc short	16	[-32768, +32767]
unsigned int	16	[0, 65535]
long int	32	[-2e9, +2e9]
float	32	[± 10e-37, 10e37]
double hoặc long float	64	[± 10e-307, 10e 307]

Đối với một vài trình biên dịch, nếu giá trị của hằng số nguyên vượt quá giá trị lớn nhất của kiểu *int*, hằng số sẽ được tự động chuyển về kiểu *long* và tương tự kiểu *float* thành *double*.

1.5 MẢNG

Mảng là tập hợp các biến cùng kiểu được phân bố liên tục trong bộ nhớ và cũng như biến, mảng được gọi qua tên của nó. Tên của mảng cũng được đặt như tên của biến và mỗi biến trong mảng là một phần tử của mảng. Cũng như biến, trước khi được sử dụng mảng cũng phải được khai báo. Việc khai báo mảng tuân theo cú pháp sau:

Kiểu_mảng tên_mảng [*biểu thức*];

Biểu thức được đặt trong dấu ngoặc vuông [] dùng để xác định số phần tử của mảng. Nó phải là một hằng số và thường phải được viết ở dạng cơ số. Sau đây là một số ví dụ về khai báo mảng:

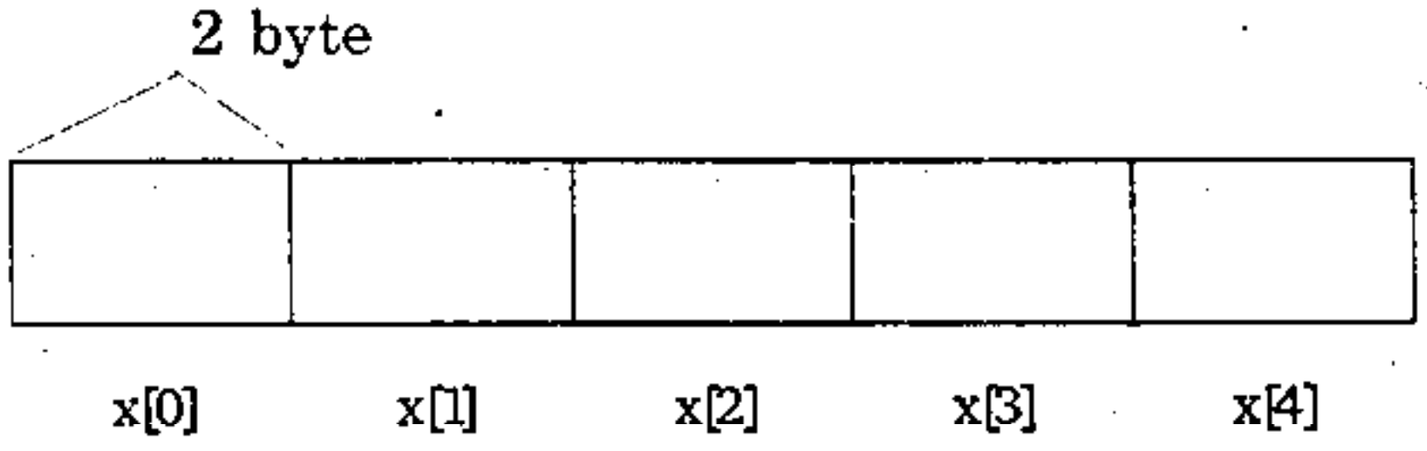
Ví dụ 1.8

int x [5]: - mảng *x* với 5 phần tử thuộc kiểu *int;*

char a[3], b[5]; - mảng *a* với 3 phần tử thuộc kiểu *char* và mảng *b* với 5 phần tử thuộc kiểu *char*.

float c[100]; mảng *c* với 100 phần tử thuộc kiểu *float*.

Mỗi phần tử của mảng có thể được xem và xử lý như là một biến, do đó chúng có thể nhận các giá trị khác nhau và tham gia vào các phép toán. Tất cả các phần tử của mảng đều có cùng một tên, đó là tên mảng. Để truy nhập đến một phần tử nào đó của mảng ta phải dùng tên của mảng và chỉ số *(index)* của phần tử đó trong mảng. Chỉ số của một phần tử chỉ ra vị trí của phần tử đó trong mảng. Cần chú ý rằng trong C, phần tử đầu tiên của mảng bao giờ cũng có chỉ số 0, như vậy *x[1]* sẽ là phần tử thứ 2 của mảng. Qua việc khai báo *int x[5]*, 5 phần tử của mảng *x* sẽ là *x[0], x[1], x[2], x[3], x[4]*. Trong bộ nhớ của máy các phần tử này sẽ sắp xếp theo thứ tự sau:



Thông thường các trình biên dịch của C không kiểm tra giá trị được gán cho chỉ số của các phần tử của mảng có vượt quá kích thước của mảng hay không. Điều này gây ra lỗi rất khó phát hiện trong khi lập trình bởi vì các phần tử với chỉ số vượt ra giới hạn cho phép sẽ thuộc về các ô ký ức trong bộ nhớ có chứa các giá trị ngẫu nhiên.

Trong C có thể định nghĩa nhiều kiểu mảng. Mảng một chiều là mảng trong đó các phần tử được phân biệt với nhau bằng một chỉ số, ví dụ $x[5]$ là mảng một chiều. Nếu để phân biệt các phần tử của mảng mà phải dùng đến 2 chỉ số, mảng được gọi là 2 chiều.

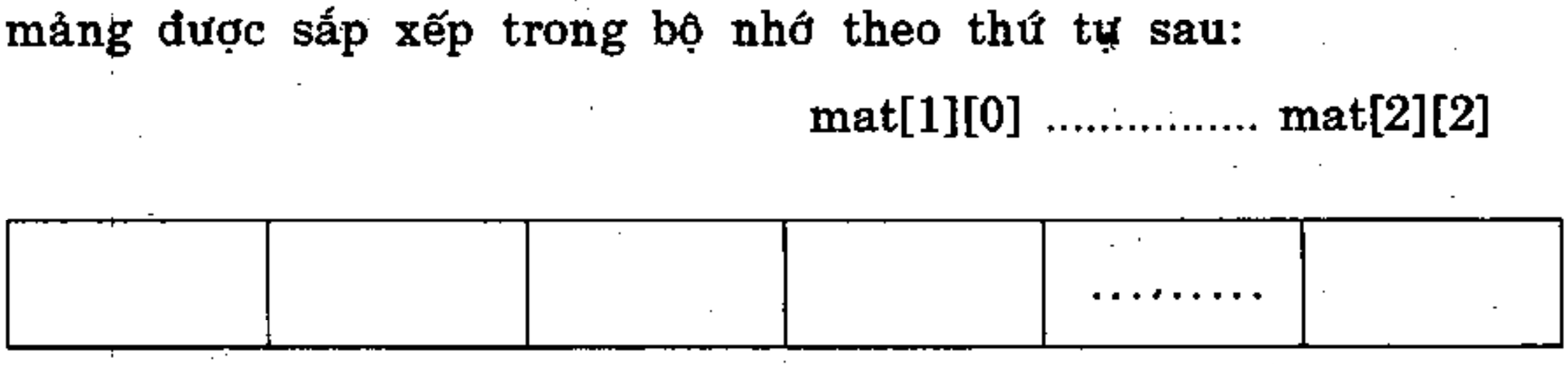
Ví dụ 1.9

```
int mat[3][3];
```

Mảng này gồm có 9 phần tử và tên của các phần tử lần lượt là:

mat[0][0], mat[0][1], mat[0][2],
mat[1][0], mat[1][1], mat[1][2],
mat[2][0], mat[2][1], mat[2][2].

Cũng như trong toán học, chỉ số đầu của phần tử dùng để chỉ hàng và chỉ số thứ hai dùng để chỉ cột. Các phần tử của mảng được sắp xếp trong bộ nhớ theo thứ tự sau:



Bằng cách tương tự ta có thể định nghĩa một mảng nhiều chiều, trong đó số các chỉ số hoàn toàn phụ thuộc vào loại trình biên dịch cũng như bộ nhớ còn giải phóng của máy.

Kích thước cần thiết cho một mảng bằng tích số các phần tử của mảng và kích thước của mỗi phần tử. Ví dụ nếu kiểu của mảng *x[5][2]* là *float* thì kích thước của mảng sẽ là 5*2*4 = 40 byte.

Theo định nghĩa, ta cũng có thể định nghĩa một mảng thuộc kiểu *char*, ví dụ:

char name[12];

Mảng thuộc kiểu *char* là một dãy liên tục các ký tự, trong mảng này ta có thể viết tên người, tên sách v.v. Một dãy liên tục các ký tự được gọi là *biến kiểu chuỗi*. Tương tự như các hằng kiểu chuỗi, các biến kiểu chuỗi phải được kết thúc bằng byte trắng (ký tự *'\0'*). Trong ví dụ trên mảng *name* sẽ được gọi là biến kiểu chuỗi nếu mảng có chứa 11 ký tự và ký tự cuối cùng (chỉ số 11) là byte trắng. Đây cũng là nguyên nhân thường phát sinh ra lỗi vì khi sử dụng mảng người lập trình thường quên phần tử cuối cùng của mảng khi khái báo kích thước của mảng.

1.6 CHÚ GIẢI

Tất cả các chương trình do những người lập trình có kinh nghiệm viết bao giờ cũng được giải thích bằng chú giải. Mục đích của việc đưa chú giải vào chương trình là để làm rõ thêm ý nghĩa của các hàm, câu lệnh và giải thích ý nghĩa của các biến được sử dụng, do đó chú giải sẽ làm cho chương trình thêm dễ hiểu và tiện cho việc tìm hiểu chương trình. Trong C chú giải là một đoạn văn bản được viết giữa 2 ký hiệu /* (bắt đầu) và */ (kết thúc), trong đó hai ký tự / và * phải được viết cạnh nhau. Ví dụ /* *Hàm vẽ vòng tròn* */ là một chú giải trong C. Chú giải có thể được đặt ở bất cứ vị trí nào trong chương

trình và khi biên dịch không chiếm bất cứ ô ký ức nào trong bộ nhớ. Điều đó cho phép ta có thể đưa các chú giải với độ dài bất kỳ vào chương trình. Tuy vậy cũng không nên lạm dụng điều này bởi vì các chú giải dài dòng chỉ làm cho chương trình thêm phức tạp và rắc rối.

1.7 CẤU TRÚC CỦA CHƯƠNG TRÌNH C

Một chương trình viết bằng ngôn ngữ C phải tuân theo các nguyên tắc, yêu cầu riêng của mình khi thiết lập. Tất cả các chương trình viết bằng ngôn ngữ C đều được xây dựng từ các chương trình con. Các chương trình này được gọi là hàm. Số lượng hàm trong chương trình là không hạn chế nhưng chương trình bao giờ cũng phải có ít nhất là một hàm. Có một hàm (và chỉ một) trong số các hàm được định nghĩa như là hàm chính và qua hàm này ta có thể liên hệ với hệ điều hành cũng như với các hàm khác của chương trình. Trong quá trình thực hiện chương trình, thông qua các hàm ta có thể trao đổi thông tin giữa chúng. Nhìn một cách tổng quát, một chương trình viết bằng ngôn ngữ C phải có cấu trúc như sau:

Khai báo các hàm thư viện chuẩn sẽ sử dụng

Định nghĩa các biến ngoài

Hàm chính (main)

Hàm 1

Hàm 2

.....

Trong khi lập trình ta có thể sử dụng một số hàm của thư viện chuẩn. Các hàm này thường được thiết lập sẵn để giải quyết một số bài toán nào đó mà không cần làm lại mỗi khi sử dụng. Các hàm này được khai báo trong thư viện chuẩn và được phân bố kèm theo trình biên dịch. Việc khai báo các hàm thư viện chuẩn thường đặt ở đầu chương trình thông qua việc sử dụng chỉ thị *#include*. Chỉ thị này sẽ được đề cập chi tiết ở phần 5.4. Các File có chứa các hàm này được gọi là File tiêu

đề *(Header file)*.

Hàm là tập hợp các toán lệnh được viết liên tiếp để thực hiện một nhiệm vụ trọn vẹn nào đó. Khi lập một hàm ta phải đặt tên cho nó. Cách đặt tên cho hàm phải tuân theo các nguyên tắc như khi đặt tên cho biến. Tên của hàm chính nhất thiết phải là *main*, còn các hàm còn lại có thể nhận một tên tùy ý. Các tên này tốt nhất là ở một mức độ nào đó nên phản ánh ý nghĩa, mục đích của hàm. Cấu trúc chung của một hàm trong C có dạng như sau:

```
Kiểu_Hàm Tên_Hàm (khai báo báo các tham số hình thức)
        {
        Khai báo các biến cục bộ;
        .....
        Các câu lệnh;
        .....
        }
```

Dòng đầu tiên mô tả tên hàm và khai báo danh sách các tham số hình thức nếu có. Các tham số này phải viết cách nhau bằng dấu phẩy *(,)*. Đây chính là các thông tin đầu vào và qua các đối số này ta có thể đưa dữ liệu vào cho hàm (giá trị của các tham số này sẽ được thiết lập thông qua việc gọi hàm từ chương trình triệu gọi). Các tham số hình thức này nếu không cần thiết thì có thể bỏ qua nhưng hai dấu ngoặc *()* thì trong mọi trường hợp đều phải viết. Chẳng hạn *getchar()* là tên của một hàm chuẩn, hàm này không chứa danh sách các tham số và sẽ được đề cập đến ở phần *1.8*. Kiểu của hàm cũng như biến bao gồm kiểu *char, int, float, double*. Ngoài ra các trình biên dịch mới còn đưa thêm một kiểu mới, đó là kiểu có thể nhận bất cứ giá trị nào sau khi thực hiện.

Ví dụ 1.10

Xây dựng một chương trình cụ thể cho C.

```
main()      /* Hàm chính bắt buộc phải có */
        {
        int i;/* Khai báo các biến cục bộ */
        i=i+1;/* Câu lệnh */
        printf ("i=%d",i); /* Hàm chuẩn dùng để biểu thị thông tin */
        }
```

Chương trình trên sử dụng một hàm duy nhất, đó là hàm *main*. Ví dụ này mô tả cách viết một hàm trọn vẹn, cách khai báo các biến và sử dụng chú giải trong chương trình. Câu lệnh cuối cùng dùng để hiển thị giá trị của *i* lên màn hình. Giá trị của *i* được hiển thị ở đây là bất kỳ vì trước đó *i* chưa được khởi tạo.

Ví dụ 1.11 dưới đây mô tả cách xây dựng một hàm trong C và cách gọi hàm này trong chương trình chính. Ở đây hàm *butler()* có kiểu *void* (không nhận giá trị trả về) và không có tham số truyền. Kết quả thực hiện của chương trình là ba dòng được hiển thị trên màn hình:

```
Se goi ham butler
Anh can gi o toi
Toi chi muon chao anh.
```

Ví dụ 1.11

```
#include <stdio.h>
void butler();
void main()
        {
        printf("Se goi ham butler\n");
        butler();
        printf("Toi chi muon chao anh\n");
```

```
        }
void butler()
        {
        printf("Anh can gi o toi\n");
        }
```

Trong 2 ví dụ trên, hàm *printf* dùng để hiển thị một chuỗi ký tự lên màn hình. Ký tự '\n' trong chuỗi này sẽ cho phép đưa con trỏ xuống dòng mới sau khi hiển thị.

Thông tin giữa các hàm trong chương trình có thể được trao đổi với nhau bằng cách sử dụng các tham số hình thức hoặc các biến ngoài. Các biến ngoài (biến tổng thể) luôn được định nghĩa trước các hàm và các hàm này có thể truy nhập đến chúng. Các biến ngoài và cục bộ được giải thích cụ thể ở các phần 4.5.

1.8 CÁC HÀM KẾT XUẤT THÔNG TIN ĐƠN GIẢN

Một trong những ý tưởng cơ bản khi thiết kế ngôn ngữ lập trình C là khi sử dụng phải có sự tương thích trên nhiều loại máy khác nhau và C phải có khả năng sử dụng trên các máy cá nhân. Để đạt các mục đích này, các cấu trúc của ngôn ngữ phải được hạn chế về số lượng và trong một chừng mực nào đó có thể hạn chế đến mức tối đa các toán lệnh phụ thuộc vào phần cứng của máy. Để giải quyết vấn đề này, trong ngôn ngữ không đưa ra các toán lệnh *output/input (vào/ra)* mà theo nguyên tắc thường phụ thuộc vào phần cứng của máy. Thay vào các câu lệnh này người ta thiết lập các hàm kết xuất thông tin vào, ra. Các hàm này được khai báo trong các thư viện chuẩn và là các hàm hay được sử dụng nhất trong chương trình. Trong phần này các hàm chỉ được đưa ra ở mức độ đơn giản nhất để thuận tiện cho việc theo dõi các ví dụ. Chi tiết về các hàm này bạn có thể xem ở phần 9.1, 9.2 của chương IX.

1.8.1 Hàm printf

Đây là một trong những hàm hay được sử dụng nhất dùng để hiển thị thông tin lên màn hình. Hàm này được định nghĩa trong tệp tiêu đề *stdio.h* trong thư viện chuẩn. Như vậy khi dùng hàm *printf* ta phải khai báo tệp *stdio.h* ở đầu chương trình (xem phần 5.4). Hàm *printf* có cú pháp sau:

printf ("Tham số định dạng", Các tham số);

Tham số định dạng được đặt trong dấu nháy kép và bao gồm các mã định dạng. Các mã định dạng này dùng để xác định số tham số và cách biểu thị các tham số này lên màn hình. Mỗi tham số định dạng luôn được bắt đầu bởi ký tự %. Ngoài các mã định dạng, trong tham số định dạng còn có thể đặt bất kỳ một đoạn văn bản ASCII nào đó để làm rõ ý nghĩa của các kết quả số hiện ra.

Tham số thường là các biến (thậm chí còn có thể là biểu thức) mà giá trị của nó cần biểu thị lên màn hình. Để đưa con trỏ xuống hàng mới sau khi hiển thị kết quả, tham số định dạng phải được kết thúc bởi "\n".

Ví dụ 1.12

```
printf("Ngôn ngữ C là một ngôn ngữ khó \n");
```

có thể được thay thế bởi các lệnh sau:

```
printf("Ngôn ngữ C");
printf("là một ngôn ngữ");
printf("khó\n");
```

Trong cả hai trường hợp, trên màn hình sẽ xuất hiện dòng chữ *Ngôn ngữ C là một ngôn ngữ khó.*

Ví dụ 1.13

```
printf("Giá trị của x là %3d, của y là %6.3 f\n",x,(x + y));
```

Ví dụ này trình bày phương pháp hiển thị giá trị của biến cũng như của biểu thức lên màn hình thông qua mã định dạng. Ở đây giá trị của biến x sẽ được hiển thị lên màn hình như là một số nguyên gồm 3 chữ số, trong khi kết quả của biểu thức $(x + y)$ sẽ được in lên như là một số thập phân gồm 6 chữ số (kể cả dấu chấm), trong đó có 3 số được hiển thị sau phần thập phân. Ký tự *d* trong mã số định dạng thứ nhất chỉ ra rằng tham số thứ nhất phải được hiển thị như là một số nguyên. Ký tự *f* trong mã số thứ hai xác định cách hiển thị của tham số thứ hai như là mã số thập phân. Trong ví dụ trên, nếu x có giá trị là 12 và y có giá trị là 8 thì kết quả được in lên màn hình sẽ là:

Giá trị của x là 12, của y là 20.000

1.8.2 Hàm scanf

Ngược lại với hàm *printf*, hàn *scanf* dùng để nhập dữ liệu từ bàn phím. Hàm được sử dụng qua cú pháp sau:

scanf("Tham số định dạng", Các đối số);

Cũng như hàm *printf*, trong hàm *scanf* tham số định dạng phải được đặt trong dấu nháy và dùng để xác định số tham số và phương pháp đưa dữ liệu cho các tham số. Các tham số định dạng luôn được bắt đầu bởi ký tự %. Khác với hàm *printf*, tham số trong trường hợp này phải là địa chỉ của các biến sẽ nhận các giá trị được đưa vào từ bàn phím.

Ví dụ 1.14

```
scanf("%2d%f", &x, &y);
```

Mã định dạng *%2d* xác định rằng ta phải nhập vào một số nguyên có độ dài tối đa là 2 chữ số và giá trị đó sẽ được gán cho biến x. Mã định dạng *%f* đòi hỏi ta phải nhập vào một số thực có độ dài tối đa của phần thập phân là 6 (ngầm định của *%f*) và giá trị đó sẽ được gán cho biến y.

. Để đưa địa chỉ của một biến trong danh sách tham số, trước đối số đó ta phải đặt ký tự &. Chẳng hạn:

&x là địa chỉ của biến *x*;

&y là địa chỉ của biến *y*

Trong ví dụ trên nếu ta đưa từ bàn phím các số sau 56 723 thì *x* và *y* sẽ có các giá trị tương ứng là 56 và 723.000000. Nhìn chung khi cần nhập và xuất dữ liệu theo một khuôn dạng nhất định có thể sử dụng các tham số định dạng sau:

%d: Số nguyên

%c: Ký tự

%s: Chuỗi

%f: Số float (dấu phẩy động)

%lf: Số double (dấy phẩy động với độ chính xác gấp đôi)

%u: Số nguyên không âm

%e: Số mũ

%o: Cơ số 8 không dấu

%x: Cơ số 16 không dấu

Ví dụ 1.15 dưới đây sẽ mô tả cách sử dụng 2 hàm *printf* và *scanf* cùng với các tham số định dạng.

Ví dụ 1.15

```
#include <stdio.h>
void main()
        {
        int tuoi;
        char ten[30];
        float trong_luong;
        printf("Hay dua ten, tuoi và trong luong cua ban\n");
        scanf("%s %d %f", ten, &tuoi, &trong_luong);
```

```
    printf("Ten cua ban la %s\n", ten);
    printf("Tuoi cua ban la %d\n", tuoi);
    printf("Trong luong cua ban la %f\n", trong_luong);
    }
```

Trong ví dụ này cần lưu ý:

- Sử dụng biến kiểu mảng để chứa chuỗi các ký tự tên người.

- Tên của mảng cũng chính là địa chỉ đầu của mảng đó, vì vậy trong chương trình để nhập tên ta chỉ sử dụng scanf("%s", ten) chứ không dùng scanf("%s", &ten);

- Có thể dùng một hàm scanf để đưa 3 giá trị có kiểu khác nhau: ten, tuoi và trong_luong. Để kết thúc nhập dữ liệu cho một biến có thể sử dụng dấu cách hoặc phím Enter.

1.8.3 Hàm getchar

Một hàm nữa mà ta hay sử dụng trong các ví dụ về sau, đó là hàm *getchar()*. Hàm này cho phép đưa một ký tự từ bàn phím (chính xác hơn là mã ASCII của ký tự). Hàm này không có tham số. Hàm được sử dụng qua cú pháp sau:

Tên_Biến = getchar();

Tên_biến nếu không cần thiết thì có thể bỏ qua. Cần chú ý là khi gặp hàm *getchar()* chương trình sẽ tạm ngừng hoạt động cho đến khi ta đưa một ký tự từ bàn phím. Chương trình sẽ tiếp tục hoạt động sau khi nhấn phím *Enter*. Sau khi thực hiện hàm, *Tên_biến* sẽ có giá trị bằng mã ASCII của ký tự được đưa vào từ bàn phím.

Ví dụ 1.16

```
c=getchar();
```

Sau khi thực hiện lệnh này, nếu ký tự được đưa vào từ bàn phím là *v* thì *c* sẽ có giá trị là 89 (mã ASCII của ký tự *v*).

Chương II

CÁC PHÉP TOÁN VÀ BIỂU THỨC

Hầu hết các lệnh dùng trong chương trình viết bằng ngôn ngữ C đều được thiết lập dựa trên các phép toán cơ bản của ngôn ngữ. Các phép toán này giữ vai trò quan trọng trong việc tính toán và so sánh các giá trị trong khi thực hiện chương trình và chiếm một tỷ lệ lớn trong các công việc cần làm của một chương trình. Phụ thuộc vào số toán hạng chịu tác động của phép toán, các phép toán được chia thành:

- Phép toán với một toán hạng;
- Phép toán với hai toán hạng;
- Phép toán với ba toán hạng;

Trong chương này chúng ta sẽ làm quen với một số phép toán cơ bản của ngôn ngữ như các phép toán số học, các phép toán quan hệ và logic, phép toán tăng giảm, chuyển kiểu bắt buộc và biểu thức điều kiện v.v.

2.1. CÁC PHÉP TOÁN SỐ HỌC

Trong C sử dụng các phép toán số học sau:

Phép toán số học	Cách viết	Ví dụ
Cộng	+	a = b + 2;
Trừ	-	a = b - 2;
Nhân	*	a = b*2;
Chia	/	a = b/2;
Chia lấy phần dư	%	a%5.

Các phép toán số học là những phép toán với hai toán hạng,

chúng có thể tham gia vào biểu thức, ví dụ: a*2 + (b - 2*3)/4; Các toán hạng tham gia vào phép toán số học có thể thuộc nhiều kiểu khác nhau. Kiểu của kết quả phụ thuộc vào kiểu của các toán hạng. Chẳng hạn kết quả của phép toán (*a* + *b*) sẽ có kiểu là *float* nếu một trong hai số có kiểu là *float* còn số kia có kiểu là *int* (xem phần 2.4). Cần chú ý rằng nếu *a* và *b* là hai số nguyên thì kết quả phép chia a/b sẽ là phần nguyên của phép chia đó.

Ví dụ 2.1

```
#include <stdio.h>
void main()
        {
        int a, b;
        a = 7;
        b = 3;
        printf("a/b là %d\n",a/b);
        }
```

Kết quả in lên màn hình sau khi thực hiện chương trình là *a/b là 2* (bằng phần nguyên của phép chia 7/3). Thông thường các phép chia với các toán hạng là số nguyên đòi hỏi ít bộ nhớ hơn và do đó tốc độ thực hiện cũng nhanh hơn.

Trong C tồn tại phép toán % - *phép lấy phần dư* của hai số nguyên trong phép chia.

Ví dụ 2.2

5%3 bằng 2;

7%2 bằng 1.

Phép toán lấy phần dư rất có lợi trong nhiều trường hợp, chẳng hạn khi cần kiểm tra một số chia hết cho số khác hay không. Phép toán này không cho phép thực hiện trên các toán hạng có kiểu khác *int*.

Ví dụ 2.3

```
#include <stdio.h>
main()
        {
        printf("Phần nguyên trong phép chia 1770/25 là %d\n",
                1770/25);
        printf("Phần dư trong phép chia 1770/25 là % d\n",
                1770%25);
        }
```

Ví dụ 2.4

Chuyển đơn vị thời gian bằng giây thành phút và giây. (Chẳng hạn 62 giây thành 1 phút và 2 giây).

```
#include <stdio.h>
main()
        {
        int sec, minute, left;/* khai báo biến giây và phút */
        printf("Hãy đưa đơn vị thời gian tính bằng giây\n");
        scanf("%d",&sec);
        minute = sec/60;/* Chuyển phần nguyên thành phút */
        left = sec%60; /* Phần dư là giây */
        printf("%d giây bằng %d phút và %d giây \n", sec,
                minute,left);
        }
```

2.2 CÁC PHÉP TOÁN QUAN HỆ VÀ LOGIC

Hai phép toán này thường được sử dụng cùng nhau và cho phép ta so sánh giá trị của các phép toán và sau đó chọn phương pháp cần giải quyết tiếp theo.

Các phép toán quan hệ và logic luôn cho giá trị đúng *(bằng 1)* hoặc giá trị sai *(bằng 0)*. Nói cách khác khi các điều kiện đưa ra được đánh giá là đúng thì phép toán cho giá trị bằng *1*, trong trường hợp ngược lại - giá trị *0*. Các phép toán quan hệ thường dùng để so sánh và chúng có các dạng sau:

>	lớn hơn	>=	lớn hơn hoặc bằng;
<	nhỏ hơn	<=	nhỏ hơn hoặc bằng;
==	bằng	!=	khác

Ví dụ 2.5

x>y Phép toán này sẽ có kết quả là 1 nếu x lớn hơn *y* hoặc *0* nếu *x* nhỏ hơn *y*;

x==y Phép toán này sẽ có kết quả là *1* nếu *x* bằng *y* hoặc *0* nếu *x* khác *y*;

Các phép toán logic dùng để liên hệ các phép toán quan hệ và đánh giá chúng trên quan điểm logic. Sau đây là một số phép toán logic dùng trong C:

Bảng 2.1

Giá trị của toán hạng		Kết quả		
Toán hạng 1	Toán hạng 2	&&	||	!
TRUE (khác 0)	TRUE (Khác 0)	1	1	0
TRUE (khác 0)	FALSE (bằng 0)	0	1	0
FALSE (bằng 0)	TRUE (Khác 0)	0	1	1
FALSE (bằng 0)	FALSE (bằng 0)	0	0	1

Ví dụ 2.6

x<y&&z>t Phép toán sẽ có kết quả bằng *1* nếu *x* nhỏ hơn *y* và *z* lớn hơn *t*;

!x<y	Phép toán sẽ có giá trị là *1* nếu *x* lớn hơn hoặc bằng *y*.

Cần chú ý rằng thứ tự ưu tiên khi thực hiện của phép toán logic luôn nhỏ hơn phép toán quan hệ và giá trị của phép toán logic và quan hệ luôn bằng *1* hay bằng *0*. Chẳng hạn *printf("%d", x>10)*; sẽ in giá trị *1* lên màn hình nếu *x* lớn hơn *10* và in giá trị *0* lên màn hình nếu *x* nhỏ hơn hoặc bằng *10*.

2.3 PHÉP GÁN

Đây là một trong các phép toán hay được sử dụng nhất trong khi lập trình. Phép toán này cho phép ta gán giá trị cho một biến và có dạng sau:

Tên_biến = Toán_hạng;

Chẳng hạn các dòng dưới đây

```
X1 = 10;
X2 = X1 + 12;
```

là ví dụ về các phép gán trong C.

Trong ngôn ngữ C việc sử dụng các phép gán liên tiếp nhau cũng được cho phép. Hãy xét ví dụ sau:

Ví dụ 2.7

```
void main()
      {
      int X1; X2
      X1 = X2 = 100;
      printf("Giá tri cua X1 la %d X2 la %d",X1,X2);
      }
```

Sau khi thực hiện chương trình, trên màn hình sẽ xuất hiện dòng

```
Gia tri cua X1 la 100 X2 la 100
```

Điều này là cho phép vì trong C mọi phép gán đều có giá trị và giá trị đó bằng giá trị của toán hạng bên phải phép gán. Vì vậy phép gán X1 = X2 = 100 sẽ tương đương với X1 = (X2 = 100) và do đó X1 sẽ có giá trị là 100.

Cần chú ý rằng 2 ký hiệu = và = = có ý nghĩa hoàn toàn khác nhau và không được dùng để thay thế cho nhau. Điều gì sẽ xảy ra khi một biến được gán cho một giá trị khác kiểu với nó? Trong các trường hợp này, trình biên dịch sẽ tự động thực hiện phép chuyển kiểu cho các toán hạng. Nguyên tắc chung được sử dụng khi chuyển kiểu là giá trị của phần bên phải phép gán sẽ được chuyển thành kiểu của phần bên trái, vì vậy khi chuyển kiểu có thể biến sẽ không nhận được giá trị chính xác cần gán cho nó do một số thông tin bị mất khi thực hiện phép chuyển kiểu. Bảng 2.2 cho biết các dạng thông tin bị mất khi thực hiện phép chuyển kiểu tự động.

Bảng 2.2

Kiểu biến	Kiểu của toán hạng	Thông tin bị mất
char	int	8 bit cao
char	long int	24 bit cao
int	long int	16 bit cao
int	float	phần thập phân
float	double	độ chính xác khi làm tròn

Ví dụ: sau phép gán

```
int x;
x = 1.25;
```

x sẽ có giá trị là *1*. Trong trường hợp này thông tin bị mất là phần thập phân *0.25*.

Thực tế của việc lập trình cho thấy các phép toán số học và các phép gán là hay được sử dụng thường xuyên hơn cả. Một

vài cấu trúc chung của chúng được sử dụng rất nhiều, do đó ngôn ngữ C đã đưa ra một vài cấu trúc đặc biệt để đơn giản hóa việc sử dụng chúng. Các cấu trúc thuộc các kiểu:

Biến_1 = Biến_1 Phép toán Toán hạng;

được viết ngắn gọn trong C theo dạng:

Biến_1 Phép toán = Toán hạng;

Toán hạng bên phải có thể là biến, hằng số hoặc tổ hợp cho phép của hằng, biến và các phép toán thực hiện trên chúng. Trong C có sử dụng những dạng viết ngắn gọn sau:

a += b; Tương đương với a = a+b;

a -= b; Tương đương với a = a-b;

a *= b; Tương đương với a = a*b;

a /= b; Tương đương với a = a/b;

a %= b; Tương đương với a = a%b;

Ví dụ 2.8

x -= 10; Tương đương với x = x-10;

y *= x; Tương đương với y = y*x;

2.4 PHÉP TOÁN TĂNG GIẢM MỘT ĐƠN VỊ

Các phép toán tăng hay giảm một đơn vị cũng là phép toán hay gặp, do đó C cũng đưa ra các cấu trúc đặc biệt để biểu diễn chúng. Các cấu trúc đó có dạng:

++ Tăng lên 1;

-- Giảm xuống 1.

Như vậy qua các phép biểu diễn này, *a++* hay *++a* là tương đương với *a += 1* và *--b* hay *b--* là tương đương với *b-=1*.

Việc đặt hai dấu trừ hoặc hai dấu cộng trước hoặc sau biến có một ý nghĩa nhất định. Ví dụ trong phép toán *a++* hoặc *a--* thì đầu tiên giá trị của *a* được sử dụng để tính toán sau đó mới được tăng hoặc giảm đi một đơn vị, trong khi ở phép toán *++a* hoặc *--a* thì đầu tiên giá trị của *a* tăng hoặc giảm đi *1* sau đó mới được sử dụng để tính toán.

Ví dụ 2.9

```
x = 15;          (1)
y = x++ +2;      (2)
z = ++x +3;      (3)
```

Trong ví dụ này ở phép toán *(1)*, *x* được gán cho giá trị bằng *15*, sau đó ở phép toán *(2)* sau khi thực hiện phép toán *y = 15 + 2 = 17*, giá trị của *x* được tăng lên một đơn vị. Trong phép toán *(3)* đầu tiên giá trị của *x* tăng lên *1* thành *17* sau đó thực hiện phép toán *z = 17+3 = 20*.

Ví dụ 2.10

```
#include <stdio.h>
void main()
      {
      int x,y,z,t;
      x=2; y=3;
      printf(" t =%d", x+ ++y);
      printf("z= %d\n", x+y);
      }
```

Kết quả của việc thực hiện chương trình này là: *t=5, z=6*.

Việc sử dụng các phép toán tăng hay giảm một đơn vị có thể dẫn đến kết quả không theo ý muốn, chẳng hạn trong ví dụ sau:

Ví dụ 2.11

```
#include <stdio.h>
main()
        {
        int a=10, b=10;
        printf(a=%d a*a =%d\n", a, a*a++);
        printf("b=%d\n", a+b);
        }
```

Trong ví dụ này đầu tiên ta muốn in giá trị của *a* và *a*a* lên màn hình sau đó giá trị của *a* được tăng lên *1* đơn vị và thực hiện phép toán *a+b*. Như vậy kết quả cần mong đợi khi thực hiện chương trình là:

```
a = 10 a*a = 100
b = 21
```

Tuy vậy để tăng hiệu quả làm việc của C, trình biên dịch được thiết kế sao cho có khả năng chọn bất cứ một tham số nào đó của hàm để thực hiện đầu tiên và chính ở điểm này đôi khi cũng sinh ra nhiều vấn đề cần chú ý. Chẳng hạn trong ví dụ trên khi thực hiện hàm *printf*, giá trị của tham số thứ hai của hàm có thể được tính đầu tiên và trước khi in giá trị của tham số thứ nhất lên màn hình giá trị của *a* đã được tăng lên một đơn vị và do đó kết quả của chương trình sẽ trở thành:

```
a = 11 a*a = 100
b = 21
```

Ví dụ 2.12

```
ans = num/2 + 5 * (1 + num++);
```

Cũng như trên, vấn đề ở đây có thể được nảy sinh vì trình biên dịch có thể thực hiện các phép toán không theo thứ tự như ta mong muốn. Khi viết biểu thức trên, ý định của ta là

đầu tiên xác định giá trị của biểu thức *num/2* và sau đó thực hiện các vế còn lại của biểu thức. Tuy vậy khi thực hiện, đầu tiên trình biên dịch có thể thực hiện việc tính toán trên toán hạng cuối cùng - tăng giá trị của *num* lên *1* đơn vị, sau đó sẽ sử dụng giá trị mới này của *num* khi tính giá trị của *num/2* và như vậy sẽ cho kết quả sai với điều ta mong muốn.

Để tránh các lỗi phát sinh do việc sử dụng các phép toán ++ và -- không hợp lý, ta cần chú ý:

- Không sử dụng phép toán tăng hay giảm cho các biến có mặt ở nhiều đối số của hàm (ví dụ 2.10);
- Không sử dụng phép toán tăng hay giảm cho các biến có mặt nhiều lần trong cùng một biểu thức (ví dụ 2.11).

2.5 BIỂU THỨC

Biểu thức là tổ hợp bao gồm các phép toán và các toán hạng (hằng số, biến, các phần tử của mảng, hàm v.v). Một biểu thức đơn giản có thể chỉ có một toán hạng hoặc bao gồm nhiều toán hạng. Sau đây là một số ví dụ về biểu thức:

```
6
4 + 21
(c/a) + (a*t) - (b+c)
(array[i] - a) *b
x = ++q%3
q>3
```

Biểu thức trong C bao giờ cũng có một giá trị nào đó. Để có thể xác định được giá trị này, thông thường các phép toán phải được thực hiện theo một thứ tự nhất định. Bảng *2.3* đưa ra thứ tự ưu tiên của các phép toán khi thực hiện. Các toán tử ở cột *thứ tự ưu tiên* có mức độ ưu tiên giảm dần từ dòng 1 đến cuối trong khi thực hiện các phép toán. Trong trường hợp các toán tử có cùng mức độ ưu tiên (các toán tử nằm trên

cùng một dòng) thứ tự thực hiện sẽ phụ thuộc vào vị trí tương đối của chúng. *Cột thứ tự thực hiện* chỉ ra thứ tự thực hiện của các toán tử có cùng cấp ưu tiên.

Các toán tử có cùng mức độ ưu tiên (các toán tử nằm trên cùng một dòng) luôn được thực hiện từ trái sang phải hoặc từ phải sang trái. Tuy vậy trong ngôn ngữ C chỉ có các toán tử && || ? : cho phép đảm bảo rằng các toán hạng bên trái luôn được thực hiện đầu tiên. Đối với những toán tử mà kết quả của biểu thức không bị thay đổi khi thay đổi vị trí của chúng thì thứ tự thực hiện sẽ không thể xác định trước. Chẳng hạn trong ví dụ:

```
y = (x = 5) + ( ++ x);
```

Bảng 2.3

Thứ tự ưu tiên	Toán tử	Thứ tự thực hiện
1	() []	Trái sang phải
2	++ -- - (kiểu) & sizeof	Phải sang trái
3	*/+	Trái sang phải
4	+-	Trái sang phải
5	< <= > >=	Trái sang phải
6	= = =!	Trái sang phải
7	&&	Trái sang phải
8	||	Trái sang phải
9	?:	Phải sang trái
10	= += -= *= /=	Phải sang trái
11	,	Trái sang phải

Phụ thuộc vào từng loại trình biên dịch, *y* sẽ có giá trị khác nhau nếu phép toán + (cộng) được thực hiện từ trái qua phải hay từ phải qua trái. Trong trường hợp thứ nhất (phép cộng được thực hiện từ trái qua phải) kết quả sẽ là $y = 5 + 6 = 11$. Trong trường hợp thứ hai (phép cộng được thực hiện từ phải qua trái), kết quả sẽ phụ thuộc vào giá trị ban đầu của *x*.

Nếu $x = 0$, kết quả sẽ là $y = 5 + 1 = 6$.

Mọi biểu thức trong C đều có một giá trị nhất định và đó là kết quả của việc thực hiện các phép toán trong biểu thức. Chính vì vậy biểu thức có thể tham gia vào các phép toán trong vai trò của một toán hạng và vì vậy khi tính toán ta phải xác định kiểu của biểu thức. Kiểu của biểu thức trong C chính là kiểu của kết quả mà nó nhận được. Tuy vậy trong biểu thức có thể tham gia nhiều kiểu dữ liệu khác nhau. Vậy kiểu của biểu thức trong trường hợp này là gì ? Khi trong một biểu thức có tham gia nhiều toán hạng với nhiều kiểu khác nhau, các toán hạng này sẽ tự động được chuyển về cùng một loại. Đó là kiểu dữ liệu cao nhất trong biểu thức khi thực hiện theo từng phép toán. Việc chuyển kiểu trong biểu thức tuân theo các nguyên tắc sau:

- Tất cả các toán hạng thuộc kiểu *char* và *short int* khi tính toán đều được tự động chuyển về loại *int*. Tất cả các toán hạng thuộc kiểu *float* - thành kiểu *double*.

- Đối với những phép toán có hai toán hạng, kiểu của các toán hạng được chuyển về kiểu của toán hạng có kiểu cao nhất. Chẳng hạn nếu kiểu cao nhất trong hai toán hạng là *double* thì cả hai toán hạng đều được chuyển về kiểu *double* v.v.

Ví dụ 2.13

```
char c;
int i, x;
float f;
double d;
x = (c*i) + (f-d) - (f*i);
```

Trong ví dụ này, khi tính toán, các toán hạng trong *(c*i)* sẽ được chuyển thành kiểu *int*, *f* trong *(f-d)* và *(f*i)* thành *double*,

và kết quả cuối cùng sẽ có kiểu là *double*, kiểu cao nhất trong các toán hạng. Kết quả của vế phải sẽ được chuyển thành kiểu *int* khi thực hiện phép gán cho *x* (xem phần 2.3).

2.6 CHUYỂN KIỂU BẮT BUỘC

Nếu các đại lượng trong biểu thức có kiểu khác nhau thì C sẽ tự động thực hiện phép chuyển đổi kiểu. Khi đó đại lượng có kiểu thấp hơn sẽ được chuyển thành kiểu cao hơn trước khi thực hiện phép toán. Tuy vậy việc chuyển kiểu có thể được thực hiện bằng cách sử dụng phép chuyển kiểu bắt buộc. Dạng của phép toán là:

(kiểu) Toán_hạng;

Sau phép toán này, toán hạng sẽ được chuyển thành kiểu được mô tả trong *kiểu*.

Ví dụ 2.14

```
(float) (x + (y/2));
```

Trong ví dụ này, kiểu của kết quả sẽ được chuyển thành kiểu *float*, không phụ thuộc vào kiểu của các biến *x*, *y* trong biểu thức.

Ví dụ 2.15

```
#include <stdio.h>
void main
      {
      int mice;
      mice = 1.6 + 1.7; /* (1) */
      printf("Gia tri cua mice =%d\n", mice);
      mice = int(1.6) + int(1.7); /* (2) */
      printf("Gia tri moi cua mice la %d\n", mice);
```

Trong biểu thức *(1)*, kết quả của vế phải sau khi tính toán là *3.3* sẽ tự động được chuyển thành kiểu của vế trái. Như vậy *mice* sẽ có giá trị là *3*. Ở biểu thức *(2)* C sẽ sử dụng phép chuyển kiểu bắt buộc cho các toán hạng nằm ở vế phải. Trong trường hợp này hai toán hạng *1.6* và *1.7* trước khi tham gia vào phép toán đều được chuyển thành kiểu *int* và chúng sẽ có giá trị là *1* và do đó *mice* sẽ có giá trị là *2*.

Ví dụ 2.16

```
.....
int x;
x = 7;
printf("%f", (float)x/2);
```

Trong ví dụ này *x* được khai báo theo kiểu *int* và như vậy phép chia *x/2* sẽ có giá trị là *3* (xem phần 2.1). Nếu muốn thu được kết quả chính xác của phép chia *x/2* (bằng 3.5) thì cần thực hiện phép chuyển kiểu bắt buộc. Phép chia *x/2* khi đó được viết thành *(float)x/2*. Theo bảng 2.3 phép chuyển kiểu luôn có mức độ ưu tiên lớn hơn phép chia, do đó trong biểu thức *x/2*, đầu tiên *x* sẽ được chuyển từ kiểu *int* sang kiểu *float* và theo nguyên tắc chung tất cả các phép toán với kiểu dữ liệu *float* đều được thực hiện với độ chính xác gấp đôi vì vậy giá trị của biến *x* và hằng số *2* sẽ được chuyển thành kiểu *double* và sau đó sẽ thực hiện phép toán. Như vậy với phép toán chuyển kiểu này kết quả của biểu thức *x/2* sẽ là *3.500000*.

Sau phép chuyển kiểu này ta cần lưu ý rằng kiểu của biến *x* sẽ không bị thay đổi và vẫn được dùng cho các phép toán về sau. Trong ngôn ngữ máy tính việc chuyển kiểu bắt buộc được gọi là *Casting*.

2.7 TÍNH KÍCH THƯỚC CỦA MỘT ĐỐI TƯỢNG

Phép toán này cho phép tính kích thước của một đối tượng

cụ thể trong ngôn ngữ C. Đối tượng được tính kích thước có thể là các biến được người sử dụng định nghĩa: biểu thức, cấu trúc, hợp, hoặc đơn giản là một từ khóa được dùng cho một kiểu dữ liệu nhất định. Phép toán được viết dưới dạng sau:

sizeof(toán hạng);

Ví dụ 2.17

```
char c;
void main()
{
int i;
long int l;
float f;
double d;
printf("%d",sizeof(c));
printf("%d",sizeof(i));
printf("%d",sizeof(l));
printf("%d",sizeof(f));
printf("%d",sizeof(sort));
}
```

Các lệnh *printf* trong ví dụ trên sẽ cho ta biết kích thước của từng loại dữ liệu được chỉ ra trong lệnh.

Việc sử dụng phép tính toán tính kích thước là một trong các kỹ thuật lập trình giúp cho ta xây dựng các chương trình có thể chạy trên các loại máy có cấu trúc khác nhau.

Điều cơ bản của phép toán này là nó được thực hiện trong quá trình biên dịch khi kích thước của từng đối tượng đã được biết rõ ràng, do đó phép toán không làm tăng thời gian thực hiện chương trình.

2.8 PHÉP TOÁN CÓ ĐIỀU KIỆN

Ngôn ngữ C đưa ra nhiều phép toán tiện lợi nhằm nâng cao hiệu quả của kỹ thuật lập trình. Một trong các phép toán đó là phép toán có điều kiện. Phép toán được ký hiệu bởi ?: và bao gồm 3 toán hạng. Phép toán có dạng:

Toán_hạng 1 ? Toán_hạng 2 : Toán_hạng 3;

Các toán hạng được chỉ ra trong phép toán có thể là các hằng số, các phần tử của mảng, biểu thức v.v... Phép toán được thực hiện thông qua các bước sau:

- Tính giá trị của toán hạng 1.
- Nếu giá trị của toán hạng 1 khác không (toán hạng 1 có giá trị đúng) thì kết quả của phép toán là giá trị của toán hạng 2.
- Nếu giá trị của toán hạng 1 bằng không (toán hạng 1 có giá trị sai) kết quả của phép toán sẽ là giá trị của toán hạng 3.

Điều cơ bản ở đây là chỉ một trong hai toán hạng 2 hoặc 3 được tính khi xác định kết quả của phép toán. Cần chú ý là kiểu kết quả của phép toán sẽ là kiểu cao nhất trong các toán hạng 2 và 3. Chẳng hạn nếu toán hạng 2 có kiểu là *int* và toán hạng 3 có kiểu là *float* thì kiểu của kết quả sẽ là *float*.

Ví dụ 2.18

```
x>y?10:100;
```

Trong ví dụ này nếu *x* có giá trị lớn hơn *y* thì kết quả của phép toán là *10* và ngược lại nếu x<y thì kết quả là *100*. Nếu bạn đã có ít nhiều kinh nghiệm trong việc lập trình thì bạn sẽ thấy rằng phép toán này có nhiều điểm giống với cấu trúc *if-else* (xem chương III) khi thực hiện. Tuy vậy, việc sử dụng phép toán trên sẽ làm cho chương trình thêm ngắn gọn và hiệu quả hơn. Chẳng hạn đoạn chương trình sau:

```
if(a>b)
        z=a;
else
        z=b;
```

có thể được viết lại ngắn gọn như sau:

```
z=(a>b)?a:b;
```

Tuy vậy khác với cấu trúc if-else, trong phép toán có điều kiện các toán hạng không thể bao gồm các toán lệnh khác của C.

Ví dụ 2.19

In ra màn hình giá trị lớn nhất và giá trị nhỏ nhất của 2 số được đưa vào từ bàn phím.

```
#include <stdio.h>
main()
        {
 float x,y,z,max,min;
 printf("Hãy nhập 3 số từ bàn phím\n");
 scanf("%f%f%f,"&x,&y,&z);
 max = x>y?x:y;
 max = max>z?max:z;
 min = x < y? x:y;
 min = min<z?min:z;
printf("Các số được nhập từ bàn phím là x =%f,y=%f,
        z=%f\n",x,y,z);
 printf("max= %f,min = %f\n",max,min);
 }/* Kết thúc chương trình */
```

Chương III

CÁC CÂU LỆNH CÓ ĐIỀU KIỆN

Thông thường nếu không chỉ ra điều gì đặc biệt thì các lệnh của chương trình sẽ được thực hiện liên tiếp nhau theo thứ tự được viết ra. Việc xây dựng một chương trình theo nguyên tắc trên sẽ không giải quyết được các bài toán trên thực tế trong đa số các trường hợp. Các bài toán ứng dụng cần giải nhìn chung đều được xây dựng trên các thuật toán. Các thuật toán đó phải cho phép người lập trình tìm ra các biện pháp giải quyết khác nhau tùy theo từng điều kiện cụ thể. Một chương trình được đánh giá là tốt nếu nó đảm bảo được các yếu tố sau:

- Cho phép các câu lệnh được thực hiện liên tiếp nhau theo thứ tự chúng được viết ra.
- Cho phép các câu lệnh viết tiếp nhau được thực hiện trong khi một điều kiện nào đó được đánh giá là đúng.
- Cho phép sử dụng các điều kiện để từ đó chọn các khả năng có thể xảy ra nhằm đề ra biện pháp giải quyết.

Để giải quyết các vấn đề trên, trong ngôn ngữ C có đưa ra các câu lệnh có điều kiện. Trong chương III này chúng ta sẽ đề cập chi tiết đến các câu lệnh đó.

3.1 BIỂU THỨC TRONG CÁC CÂU LỆNH ĐIỀU KHIỂN

Phần cơ bản trong các cấu lệnh thường cho phép chọn một trong hai khả năng có thể xảy ra. Việc chọn lựa này được tiến hành trên cơ sở đánh giá giá trị của biểu thức trong cấu trúc của lệnh. Phụ thuộc vào tác dụng của chúng các biểu thức này được dùng như các điều kiện để kiểm tra, điều kiện rẽ nhánh

hoặc điều kiện để ngừng chương trình.

Mọi biểu thức trong ngôn ngữ C đều có một giá trị nào đó. Tuy nhiên trong các câu lệnh điều khiển, giá trị cụ thể của biểu thức thường không được chú ý đến. Điều cơ bản ở đây là biểu thức đó có giá trị khác không hay bằng không. Nếu biểu thức trong các câu lệnh điều khiển có giá trị khác không, điều đó có nghĩa là điều kiện kiểm tra được đánh giá đúng và biểu thức có giá trị bằng 1. Trong trường hợp ngược lại - điều kiện là sai và biểu thức có giá trị bằng 0.

Giá trị của biểu thức được tính theo các nguyên tắc chỉ ra trong chương II. Khi tính các giá trị này ta phải thực hiện theo mức độ ưu tiên của các phép toán. Nói chung để làm cho chương trình dễ hiểu hơn, ở những chỗ cần thiết ta nên đặt các dấu ngoặc đơn () để xác định rõ thứ tự thực hiện của các phép toán trong biểu thức. Chẳng hạn hai biểu thức sau được trình biên dịch chấp nhận là đúng và không ra thông báo lỗi, mặc dù biểu thức thứ hai có gây cho ta một sự nghi ngờ nào đó:

11<12<13 (1) và 13<12<11 (2)

Trong các biểu thức trên, các phép toán được thực hiện theo thứ tự từ trái sang phải. Như vậy để cho rõ ràng hơn các phép toán trên có thể được viết thành:

(11<12)<13 (3) và (13<12)<11 (4)

Đối với biểu thức (3), phần đầu *(11<12)* được đánh giá là đúng và có giá trị là *1*, như vậy biểu thức (1) trở thành *1<13*. Trong biểu thức (4), phần đầu *(13<12)* được đánh giá là sai và khi đó biểu thức trở thành *0<11*. Như vậy đến đây ta có thể thấy rất rõ ràng rằng cả hai biểu thức đều được đánh giá là đúng.

Trong một vài trường hợp việc sử dụng phép toán gán trong biểu thức có thể dẫn đến các hiệu ứng phụ. Tuy vậy các hiệu ứng phụ này trong nhiều trường hợp cũng cho phép người lập trình phát huy các khả năng sáng tạo của mình để tạo ra các

chương trình ngắn gọn và hiệu quả hơn. Việc sử dụng các hiệu ứng phụ này về nguyên tắc đều do người lập trình kiểm soát, do vậy những người mới lập trình nên tránh sử dụng chúng. Trong trường hợp ngược lại có thể sinh ra các kết quả sai trong khi trình biên dịch không báo lỗi khi biên dịch.

Ví dụ sau sẽ minh họa cho việc phát sinh hiệu ứng phụ khi sử dụng phép gán. Ở đây nhờ sử dụng hiệu ứng phụ, đoạn chương trình sẽ trở nên ngắn gọn hơn. Đoạn chương trình:

```
int R;
R = getchar ();
if(R != 'Y' &&R ! = 'n')
.....
```

Có thể được viết thành như sau:

```
int R;
if(R = getchar())! = 'Y' &&R ! = 'n')
.....
```

Sau phép gán *R = getchar()*, biến *R* sẽ nhận được một giá trị mới. Mặt khác việc thực hiện phép toán && luôn bắt đầu từ toán hạng bên trái *(R = getchar())!= 'Y'*. Như vậy khi thực hiện vế phải của phép toán thì *R* đã được nhận giá trị mới sau phép toán *R = getchar()* trước đó.

3.2 CÂU LỆNH IF-ELSE

Câu lệnh if-else thường được dùng để rẽ nhánh trong chương trình. Phụ thuộc vào giá trị cần kiểm tra, câu lệnh sẽ chuyển quyền điều khiển của chương trình đến một nhóm lệnh nào đó. Cấu trúc hay được sử dụng nhất của câu lệnh có dạng sau:

```
if(Biểu thức)
    câu lệnh 1;
```

else

câu lệnh 2;

Trong đó *biểu thức* là điều kiện để kiểm tra và *câu lệnh 1* và *câu lệnh 2* có thể là lệnh đơn hay là một nhóm lệnh. Cần nhắc lại rằng, trong C mỗi câu lệnh luôn được kết thúc bởi dấu ; và các nhóm lệnh phải được đặt trong 2 dấu ngoặc {}.

Câu lệnh *if-else* thực hiện theo trình tự sau:

- Tính giá trị của *biểu thức*.
- Nếu *biểu thức* có giá trị khác 0, thực hiện *câu lệnh 1*, nếu bằng 0 - *câu lệnh 2*.
- Chuyển quyền điều khiển đến các câu lệnh viết ngay sau cấu trúc *if-else*.

Việc sử dụng câu lệnh *if-else* sẽ cho phép chọn một trong hai khả năng có thể xảy ra. Như vậy trong cấu trúc trên, một trong các câu lệnh - *câu lệnh 1* hoặc *câu lệnh 2* sẽ không được thực hiện. Mặc dù cú pháp của ngôn ngữ có thể cho phép ta viết cấu trúc *if-else* bằng nhiều cách nhưng cấu trúc hay được sử dụng nhất của câu lệnh *if-else* thường có dạng sau:

```
if(biểu thức)
        {
        Câu lệnh;
        ...
        }
else
        {
        Câu lệnh;
        ...
        {
```

Cách viết này đặc biệt có hiệu quả khi sử dụng cho những

cấu trúc rẽ nhánh phức tạp.

Ví dụ 3.1

Ví dụ này mô tả việc sử dụng câu lệnh if-else. Điều kiện rẽ nhánh là biểu thức logic có chứa phép toán &&.

```
...
int X,Y;
if(X==Y && X<10)
        X = 2*Y; /* Câu lệnh 1 */
else
        X = 10*Y; /* Câu lệnh 2 */
```

Giá trị của *X, Y* trong ví dụ này sẽ xác định câu lệnh nào sẽ được thực hiện. Ví dụ nếu *X=5* và *Y = 5* thì *câu lệnh 1* sẽ được thực hiện. Ở đây phép toán = = có tác dụng so sánh giá trị của hai biến *X* và *Y*. Cần phân biệt ý nghĩa của hai phép toán == và =. Biểu thức *(X=Y && X<10)* có ý nghĩa hoàn toàn khác. Trong biểu thức này đầu tiên biến *X* được gán giá trị của *Y* và sau đó nếu *Y* khác 0 sẽ bắt đầu kiểm tra phần hai của biểu thức *(phần X<10)*. Trong ví dụ *3.1* nếu *X = 7* và *Y = 15* thì *câu lệnh 2* sẽ được thực hiện.

Ví dụ 3.2

Từ bàn phím, bằng hàm scanf hãy đưa 2 số nguyên n1 và n2. Xác định số lớn nhất trong 2 số đó.

```
#include <stdio.h>
main()
{
int n1,n2,max;
printf("Hãy đưa số thứ nhất:\n");
scanf("%d", &n1);
printf("Hãy đưa số thứ hai:\n");
```

```
scanf("%d", &n2);
if(n1>n2)
        max = n1;
else
        max = n2;
printf("max(n1,n2) = %d\n",max);
}
```

Ví dụ 3.3

Từ bàn phím đưa một số x bất kỳ. Hãy tính giá trị tuyệt đối |x| theo công thức: |x| = x, nếu x>0, và |x| = -x, nếu x<0. Hãy so sánh giá trị thu được với giá trị tuyệt đối của x tính theo hàm chuẩn abs();

```
#include <stdio.h>
#include <math.h>
main()
{
float abs_user,x;
printf("Hãy đưa số bất kỳ\n");
scanf("%f", &x);
if(x>0)
        abs_user = x;
else
        abs_user= -x;
if(abs_user == abs(x))
        {
        printf("Giá trị tuyệt đối của số đưa vào là:\n");
        printf("abs(x)=%f", abs_user);
        }
```

```
        else
                printf("Trong chương trình có lỗi\n");
        }
```

Chú ý rằng trong chương trình có sử dụng hàm chuẩn *abs(x)* thuộc file tiêu đề *math.h*, do đó ở đầu chương trình cần phải sử dụng dòng lệnh *#include <math.h>*

Ví dụ 3.4

Cho 3 cạnh a,b,c của một tam giác. Xác định dạng của tam giác: vuông, cân, đều hay thường.

```
#include <stdio.h>
#include <math.h>
main()
        {
        int i,j;
        float a,b,c,max,s=1e-10, d;
        printf("Vào dữ liệu các cạnh của tam giác\n");
        scanf("%f%f%",&a,&b,&c);
        /*Kiểm tra điều kiện của 3 cạnh*/
        max = a>b?a:b;
        if(max<c)
                max = c;
        if(2*max-a-b-c) >=0)
                printf("a,b,c không phải là 3 cạnh của tam giác\n");
        else
        {
        if(a==b &&b ==c)
                printf("Tam giác đều\n");
        else
```

```
if(a==b || b==c || c==a)
        printf("Tam giác cân\n");
else
{
d = fabs(2*max*max-a*a-b*b-c*c);
if(d<s)
        printf("Tam giác vuông\n);
    else
        printf("Tam giác thường\n");
    }
  }
}
```

Trong nhiều trường hợp trong cấu trúc *if-else, câu lệnh 2* có thể được bỏ qua. Như vậy cấu trúc có thể được viết thành:

if(Biểu thức)

Câu lệnh 1

else

;

Tuy vậy bởi vì dấu ; đứng riêng không thực hiện bất cứ một nhiệm vụ nào do đó nó có thể bỏ qua và cấu trúc sẽ có dạng sau:

if(Biểu thức)

Câu lệnh 1;

Cấu trúc này của *if-else* sẽ thực hiện theo các bước sau: nếu điều kiện kiểm tra (biểu thức) được đánh giá là đúng, quyền điều khiển của chương trình sẽ được trao cho *câu lệnh 1*. Ngược lại quyền điều khiển sẽ được trao cho lệnh đứng ngay sau cấu trúc này.

Bằng hai câu lệnh *if* ta có thể thay thế cấu trúc của *if-else*. Cấu trúc sau:

```
if(x>y)
        x=10;
else
        x=12;
```

có thể được thay thế bởi:

```
if(x>y)
        x=10;
if(x<=y)
        x=20;
```

Trong trường hợp khả năng cần chọn nhiều hơn 2, cấu trúc *if-else* cần được mở rộng bằng *else-if*. Hãy xét một ví dụ cụ thể dưới đây - ví dụ 3.5.

Ví dụ 3.5

Hãy xác định tiền điện phải trả hàng tháng theo các trường hợp dưới đây:

1. Sử dụng đến 100 Kw/h: giá mỗi Kw là 450 đồng.

2. Từ 101 Kw/h đến 150 Kw/h: giá 750 đồng.

3. Trên 150 Kw/h: giá 950 đồng.

```
#include <stdio.h>
void main()
        {
int kwh; /* Luong dien tieu thu */
float tien; /* Tien phai tra */
printf("Hay dua luong dien da tieu thu:\n");
scanf("%d", &kwh);
if(kwh  >  150)
```

```
            tien = (kwh - 150) * 950 + 50 * 750 + 100 * 450
    else if (kwh > 100)
            tien = (kwh - 100) * 750 + 100 * 450
    else
            tien = kwh * 450
    printf("Tien dien phai tra là %f\n", tien);
}
```

3.3 CÂU LỆNH WHILE VÀ DO-WHILE

Bằng nhóm lệnh *while* và *do-while* ta có thể lặp lại việc thực hiên của một hoặc nhóm lệnh tùy theo điều kiện được đưa ra trong câu lệnh.

Cậu lệnh *while* có dạng sau:

while(biểu thức)

câu lệnh;

Ở đây biểu thức là điều kiện quyết định số lần thực hiện của câu lệnh. Câu lệnh *while* hoạt động theo nguyên tắc sau:

- Tính giá trị của *biểu thức*.
- Nếu giá trị của *biểu thức* khác 0, thực hiện *câu lệnh* và quay về bước 1.
- Nếu giá trị của *biểu thức* bằng 0, thoát khỏi chu trình và chuyển tới câu lệnh sau thân *while*.

Câu lệnh *do-while* có dạng sau:

do

Câu lệnh;

while(Biểu thức);

Câu lệnh thực hiện qua các bước sau:

- Thực hiện *câu lệnh* trong thân.

- Tính giá trị của *biểu thức*.
- Nếu giá trị của *biểu thức* khác 0, quay lại bước 1.
- Nếu giá trị của *biểu thức* bằng 0, thoát ra khỏi chu trình.

Câu lệnh *while* và *do-while* đều dùng để xây dựng các vòng lặp. Điểm khác nhau cơ bản giữa hai câu lệnh này là vị trí kiểm tra điều kiện để kết thúc vòng lặp. Khác với *while*, trong *do-while* điều kiện được kiểm tra sau thân vòng lặp, do đó thân của vòng lặp *do-while* bao giờ cũng được thực hiện ít nhất một lần. Một điểm nữa cần chú ý là khác với *while*, *do-while* phải được kết thúc bởi dấu ; được viết sau *while*. Nếu dấu ; này bị bỏ qua trình biên dịch sẽ ra thông báo rằng câu lệnh *do-while* bị viết thiếu *while*. Trong trường hợp này *while* bị viết thiếu dấu ; sẽ được liên kết với câu lệnh đứng sau nó và có tác dụng như câu lệnh *while* đứng riêng biệt

Ví dụ 3.6

Tính độ dài của xâu ký tự được nhập vào từ bàn phím.

```
#include <stdio.h>
void main()
      {
      int i=0;
      char s[20];
      printf("Hãy nhập xâu ký tự từ bàn phím\n");
      scanf("%s", s);
    while (s[i]!='\0') /* Kiem tra phần tử thứ i */
          ++i;
  printf("Độ dài của xâu ký tự là %d\n", i);
  }
```

Trong ví dụ trên, điều kiện để kết thúc vòng lặp là *s[i]* bằng *'\0'*. Đây là một byte được trình biên dịch tự động đặt vào cuối xâu ký tự sau khi đưa vào để đánh dấu vị trí kết thúc của câu (xem chương I).

Ví dụ:

Từ bàn phím hãy đưa m số nguyên (m < 10). Hãy sắp xếp chúng vào mảng int number[10] theo thứ tự ngược lại với thứ tự chúng được nhập vào.

```
#include <stdio.h>
void main()
    {
    int i,m,k;    /* i - biến điều khiển vòng lặp */
                  /* m - Số lượng các số đưa vào */
                  /* k - Biến trợ giúp */
    int number[10]; /* mảng dùng để sắp xếp các số */
    printf("Số lượng các số cần nhập\n");
    scanf("%d", &m),
    i=0;
    while(i<m)
        {
        printf("Hãy đưa số thứ %d\n",i+1);
        scanf("%d",& number[i]);
        i+=1;
        }
        printf("\n");
        i=0;
        while(i<m)
            {
```

```
        printf("Các số được nhập vào là %d",number[i]);
        i+=1;
        }
/* Sắp xếp các số được đưa vào theo thứ tự mới */
i=0;
while(i<m-i-1)
        {
        k = number[m-i-1];
        number[m-i-1] = number[i];
        number[i] = k;
        i+=1;
        }
printf("Thứ tự sắp xếp mới của dãy là:\n");
        i=0;
        while(i<m)
                {
                printf("%d",number[i]);
                i+=1;
                }
}
```

Ví dụ 3.7

*Từ bàn phím, bằng hàm getchar() hãy đưa một số ký tự, ký tự cuối cùng phải là *. Trong khi đưa các ký tự này hãy kiểm tra số lần xuất hiện của một ký tự nào đó đã được xác định trước bằng hàm getchar().*

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

```
{
int R;
char symbol;/* Ký tự cần kiểm tra */
    int i;
    printf("Hãy đưa ký tự cần kiểm tra\n");
    symbol = getchar();
    R = getchar(); /* Xóa Enter sau ký tự */
    printf("Hãy đưa các ký tự từ bàn phím \n")
    printf("Để kết thúc gõ * \n");
    i=0;
    while(R=getchar() !='*')
        if(R==symbol)
            i+=1;
    printf("\n");
}
```

Trong ví dụ trên ta hãy chú ý đến câu lệnh *R = getchar()*. Thông thường thông tin được đưa vào từ bàn phím sẽ được giữ trong bộ đệm và sau đó sẽ lần lượt được xử lý theo thứ tự chúng được đưa vào. Việc thực hiện của hàm *getchar()* được bắt đầu sau khi nhấn phím Enter. Như vậy sau lệnh *symbol = getchar()* ta cần xóa thông tin do phím Enter đưa vào và đang còn được lưu trong bộ đệm. Điều này được thực hiện bởi lệnh *R = getchar()* viết ngay sau đó.

Để kết thúc việc nhập các ký tự, có thể dùng phím F6 thay cho dấu '*'. Khi đó điều kiện kiểm tra được viết thành while (R = getchar() ! = EOF). Ở đây EOF là giá trị đã được định nghĩa trong file tiêu đề *stdio.h* và là giá trị nhận được khi gõ phím F6 hoặc Ctrl+Z.

Trong khi sử dụng các câu lệnh *while* và *do-while* ta cần phải chú ý đến một số điểm sau:

• Chỉ có câu lệnh nằm trong vòng lặp mới có thể thay đổi giá trị của biểu thức điều kiện. Chính vì vậy trong vòng lặp phải có câu lệnh để chương trình có thể thoát ra khỏi vòng lặp ở một thời điểm nhất định.

• Trong biểu thức điều kiện của câu lệnh *while* cần chú ý lựa chọn giá trị ban đầu của các biến để đảm bảo cho các câu lệnh trong thân được thực hiện.

Ví dụ 3.8

```
#include <stdio.h>
void main()
    {
    int x = 28;
    while(x<30)
        {
        a=b;
        b = 2*c+x;
        printf("Không thể thoát khỏi vòng lặp\n");
        }
}
```

Điều kiện $x<30$ luôn được đánh giá đúng trong ví dụ, do đó chương trình không thể chuyển quyền điều khiển ra khỏi các câu lệnh trong thân của *while*. Ở đây ta đã phạm phải sai lầm là không thay đổi giá trị của biến điều khiển trong thân của vòng lặp. Trong ví dụ này để có thể thoát khỏi vòng lặp cần có sự can thiệp bên ngoài bằng cách gõ phím Ctrl+C.

Điều kiện trong các câu lệnh có thể là biểu thức bất kỳ, do đó trong biểu thức có thể tham gia các biến thuộc kiểu *float*. Tuy nhiên việc sử dụng các biến thuộc kiểu *float* làm các biến điều khiển có thể làm cho chương trình không thể thoát ra

khỏi vòng lặp. Ví dụ sau sẽ minh họa cho vấn đề này.

Ví dụ 3.9

```
#include <stdio.h>
void main()
    {
    int i;
    float x;
    i=1;
    x = 0.005;
    while(x!=0.995 )
        {
        printf("Chu trình thứ %d, giá trị của x là %f\n", i, x);
        i += 1;
        x += 0.01;
        }
    }
```

Trong ví dụ này, sau mỗi chu trình giá trị của x tăng lên *0.01* đơn vị. Như vậy với giá trị ban đầu là *0.005* về lý thuyết sau *99* lần thực hiện x sẽ có giá trị *0.995* và chương trình có thể thoát ra khỏi vòng lặp. Tuy nhiên điều này có thể sẽ không xảy ra vì lý do được làm tròn số khi tính, x có thể sẽ không bao giờ đạt được giá trị *0.995* mà ta mong muốn. Đây chính là một trong các nguyên nhân để ta chỉ nên làm việc với các biểu thức điều kiện thuộc kiểu *int*.

3.4 CÂU LỆNH FOR

Câu lệnh *for* là câu lệnh phức tạp nhất trong số các câu lệnh thuộc nhóm này. Câu lệnh có dạng sau:

for(Biểu thức 1; Biểu thức 2; Biểu thức 3)

Câu lệnh;

Trong câu lệnh này:

- *Biểu thức 1* dùng để khởi tạo các biến điều khiển của vòng lặp.

- *Biểu thức 2* là điều kiện dùng để kiểm tra việc thực hiện của chu trình.

- *Biểu thức 3* dùng để thay đổi giá trị của các biến điều khiển. Biểu thức này thường chứa các công thức dùng để thay đổi giá trị của các biến điều khiển sau mỗi chu trình của vòng lặp.

Câu lệnh *for* thực hiện qua các bước sau:

• Tính giá trị của *biểu thức 1* (khởi tạo các biến điều khiển).

• Xác định giá trị của *biểu thức 2*.

• Nếu giá trị của *biểu thức 2* khác 0, thực hiện các lệnh trong thân *for*, sau đó tính giá trị của *biểu thức 3* và gán các giá trị cho các biến điều khiển, quay trở lại bước 2.

• Nếu giá trị của *biểu thức 2* bằng 0, thoát ra khỏi vòng lặp *for* và thực hiện các lệnh được viết sau thân *for*.

Ví dụ 3.10

Từ bàn phím, bằng hàm scanf hãy đưa hai số nguyên dương (số đầu nhỏ hơn số sau). Sử dụng dòng lặp for hãy tính tổng của tất cả các số nguyên nằm giữa 2 số đó, kể cả hai số được đưa vào.

```
#include <stdio.h>
void main()
    {
    int i;
```

```
int n1, n2;
int result;
int temp; /* biến tạm */
result=0;
printf("Hãy đưa số đầu\n);
scanf("%d", &n1);
printf("Hãy đưa số thứ hai\n) ;
scanf("%d", &n2 );
    if(n2 < n1)
        {
        temp=n1;
        n1=n2;
        n2=temp;
        }
    for(i=n1, i<=n2; i++)
        result +=i;
    print("Tổng các số nằm giữa %d và %d là %d\n",
        n1, n2, result);
}
```

Cần lưu ý là trong vòng lặp *for* giá trị của *i* tăng dần từ *n1* đến *n2* do đó trong chương trình cần phải có các dòng lệnh dùng để hoán vị *n1* và *n2* nếu *n2* < *n1*.

Ví dụ 3.11

Từ bàn phím hãy đưa một số nguyên bất kỳ và kiểm tra số đó có phải là nguyên tố hay không?

Để làm ví dụ này trước hết ta nhắc lại một số điều cơ bản - số nguyên tố là gì? Số nguyên tố là một số nguyên chỉ chia hết cho 1 và chính bản thân nó. Một phương pháp đơn giản

nhất dùng để kiểm tra một số có phải là số nguyên tố hay không là sử dụng phép chia của số đó cho tất cả các số nguyên nằm giữa số 1 và số cần kiểm tra. Nếu một trong các kết quả thu được là số nguyên thì số đó không phải là số nguyên tố. Nhằm thực hiện mục đích này trong ví dụ ta sẽ sử dụng phép chia lấy phần dư - phép toán %

```
#include <stdio.h>
main()
    {
    int number, divisor;
    printf("Đưa số nguyên cần kiểm tra\n");
    scanf("%d", &number);
    while(number < 2)
        {
        printf( "Không kiểm tra số nhỏ hơn 2\n);
        printf("Đưa số khác\n");
        scanf ("%d", &number);
        }
for(divisor = 2; number%divisor !=0; divisor++)
    ;
if(divisor == number)
    printf("%d là số nguyên tố\n", number);
else
        printf("%d không phải số nguyên tố\n", number);
    }
```

Phương pháp tìm số nguyên tố được đưa ra trong ví dụ trên không phải là phương pháp hiệu quả nhất, đây chỉ là thuật toán đơn giản dùng để minh họa cho việc sử dụng vòng lặp *for*. Bạn có thể tìm những thuật toán hiệu quả hơn để tìm một số

nguyên tố.

Ví dụ 3.12

Tính tổng $1 - \frac{1}{1*2} + \frac{1}{2*3} - \frac{1}{3*4} + \ldots + (-1)^{n*}\frac{1}{n*(n+1)}$

với *n* là một số nguyên dương.

```
#include <stdio.h>
void main()
    {
    int n, i;
    float kq=1; /* Chứa kết quả */
    int sign = 1; /* Dấu của toán hạng */
    printf("Hãy nhập số nguyên n:\n");
    scanf("%d", &n);
    for(i=1, i<=n; i++)
        {
        sign = -sign;
        kq = (float) sign / (i * (i + 1));
        }
    printf("Ket qua cua tong la %f\n", kq);
    }
```

Trong cú pháp của câu lệnh *for*, không nhất thiết phải tồn tại cả 3 biểu thức. Tuy vậy trong trường hợp một biểu thức nào đó bị bỏ qua, dấu chấm phẩy (;) đi kèm sau nó bắt buộc phải có. Chẳng hạn ta có thể viết:

```
for (;biểu thức 2;biểu thức 3)
    câu lệnh;
```

hay:

```
for (;;biểu thức 3)
```

câu lệnh;

hoặc

for (;;)

câu lệnh;

Trong trường hợp *biểu thức 2* bị bỏ qua, điều kiện được kiểm tra bởi biểu thức này luôn được đánh giá là đúng. Như vậy ta phải dùng thêm lệnh ở trong thân của vòng lặp để chuyển quyền điều khiển ra khỏi vòng lặp khi cần thiết.

Ví dụ 3.13

Từ bàn phím nhập một số nguyên kiểu long. Hãy tính tổng các chữ số của số này.

Để giải được bài toán này, giả sử số nguyên được đưa vào được viết dưới dạng $a_n a_{n-1} \ldots a_1 a_0$. Số nguyên này có thể được biểu diễn dưới dạng:

$(\ldots(a_n * 10 + a_{n-1}) * 10 \ldots) * 10 + a_0$;

Tổng các chữ số của số nguyên dễ dàng được xác định nếu ta xác định được từng chữ số của số nguyên đó. Theo công thức trên, nếu chia số nguyên cho 10 thì phần dư của phép chia sẽ là a_0 và nếu ta lấy phần nguyên của phép chia, chia tiếp cho 10 thì phần dư sẽ là a_1. Quá trình này sẽ được tiếp tục cho đến khi phần nguyên của phép chia cho 10 bằng 0. Dưới đây là chương trình giải quyết yêu cầu của bài toán.

```
#include <stdio.h>

void main()
{
    long n, number;
    long pn; /* phan nguyen cua phep chia cho 10 */
```

```
scanf("%ld", &n);
getchar();
number = n;
for(; (pn = number/10)! = 0; number = pn)
    kq+=number%10;
 kq+=number;
 printf("Tong cac chu so cua % ld la % d\n", n, kq);
}
```

Các *biểu thức 1, biểu thức 2, biểu thức 3* không chỉ là một biểu thức đơn giản mà còn có thể bao gồm nhiều biểu thức. Khi đó dãy các biểu thức này phải được viết cách nhau bởi dấu phẩy. Ví dụ để tính tổng các phần tử của mảng a ta có thể viết:

```
for (sum = 0, i = 0; i < 10; sum += a[i++])
    ;
```

Để tính tổng của 100 số nguyên đầu tiên ta có thể dùng vòng lặp sau:

```
for (sum = 1, i=1; i<100;i++,sum +=i)
    ;
```

Một vấn đề đặt ra là khi nào nên dùng câu lệnh *for* và khi nào nên dùng *while*. Trong một chừng mực nào đó, việc sử dụng các câu lệnh này hoàn toàn phụ thuộc vào thói quen của người lập trình bởi vì tất cả những gì mà câu lệnh *for* làm được đều có thể dùng *while* để thay thế và ngược lại. Chẳng hạn:

for(; biểu thức 2) có thể được thay thế bằng câu lệnh while bằng cách viết sau:

while(biểu thức 2)

hay dạng đầy đủ của câu lệnh *for: for(biểu thức 1; biểu thức 2; biểu thức 3)* có thể được thay thế bằng:

Biểu thức 1;

while(biểu thức 2)

{

Câu lệnh;

Biểu thức 3;

}

Biểu thức 1 thường dùng để khởi tạo biến và *biểu thức 2* là điều kiện để thực hiện vòng lặp. Qua 2 ví dụ so sánh trên ta có thể thấy vòng lặp *for* nên được chọn để sử dụng trong trường hợp cần thiết tạo các biến điều khiển vòng lặp và khi đã biết trước số lần cần thực hiện các lệnh trong vòng lặp. Trong hai trường hợp này, việc sử dụng câu lệnh *for* sẽ làm cho chương trình trở thành ngắn gọn và súc tích hơn.

3.5 CÂU LỆNH BREAK

Câu lệnh *break* cho phép thoát khỏi các chu trình của *for*, *while* và *do-while* ở vị trí bất kỳ trong thân vòng lặp và chuyển quyền điều khiển của chương trình cho câu lệnh nằm ngay sau các vòng lặp trên. Như vậy việc sử dụng *break* đưa ra khả năng mới cho phép chương trình thoát ra khỏi vòng lặp mà không cần kiểm tra điều kiện kết thúc của vòng lặp. Cú pháp của lệnh có dạng sau:

break;

Ví dụ 3.14

Từ bàn phím bằng hàm getchar hãy đưa một xâu ký tự. Ký tự cuối cùng phải là ''. Trong khi đưa các ký tự này hãy kiểm tra sự có mặt của một ký tự đã được đưa trước bằng bàn phím và xác định vị trí của ký tự đó trong xâu. Nếu ký tự được tìm thấy thì hãy ngừng quá trình kiểm tra bằng lệnh break.*

```
#include <stdio.h>
main()
    {
    int R, j, i;
    char symbol ;
    printf("Hãy đưa ký tự cần kiểm tra\n");
    symbol = getchar();
    R = getchar();
    printf("Hãy đưa xâu ký tự\n");
    printf("Để kết thúc hãy đưa '*'\n");
    j=0;
    i=1;
    while((R=getchar())!='*')
        if(R==symbol)
            {
            j=1;
            break;
            }
        else
            i+=1;
    if(j==1)
        printf("Ký tự %c xuất hiện ở vị trí %d của xâu\n",
                symbol,i);
    else
        printf("Trong xâu đưa ra không chứa ký tự %c\n",
                symbol);
```

Ví dụ 3.15

Từ bàn phím hãy đưa số nguyên bất kỳ và kiểm tra xem số đó có phải là nguyên tố không.

Ví dụ này đã được đưa ra trong khi xem xét câu lệnh *for* (ví dụ 3.11). Ở đây ta sẽ dùng ví dụ này để minh họa cho việc sử dụng *break.*

```
#include <stdio.h>
main()
    {
    int number,divisor;
    printf("Đưa số nguyên cần kiểm tra\n");
    scanf("%d", &number);
    while (number<2)
    {
    printf("Không kiểm tra số nhỏ hơn 2\n");
    printf("Đưa số khác\n");
    scanf("%d", &number);
    }
divisor = 2;
while(divisor <=number)
    if(number %divisor++==0)
        break;
if(divisor ==number)
    printf("%d là số nguyên tố \n", number);
else
    printf("%d không phải là số nguyên tố\n", number);
}
```

3.6 CÂU LỆNH CONTINUE

Câu lệnh *continue* cho phép kết thúc việc thực hiện chu trình hiện thời của các vòng lặp *for, while* và *do-while*. Câu lệnh được sử dụng qua dạng sau:

continue;

Cũng như *break, continue* có thể được viết ở vị trí bất kỳ trong thân vòng lặp. Câu lệnh *continue* thực hiện qua các bước sau:

- Kết thúc chu trình hiện thời của vòng lặp. Giá trị tức thời của các biến được lưu giữ.
- Đối với vòng lặp *while* và *do-while*, lệnh *continue* sẽ chuyển đến việc kiểm tra điều kiện dừng của vòng lặp, đối với *for* lệnh sẽ chuyển đến việc thực hiện biểu thức 3.
- Tiếp tục thực hiện vòng lặp.

Ví dụ 3.14

Từ bàn phím bằng vòng lặp for hãy đưa 10 số nguyên vào mảng int a[10]. Bằng vòng lặp for hãy hiển thị các số chẵn và thứ tự được đưa vào của nó. Nếu số được kiểm tra là số lẻ thì bằng lệnh continue kết thúc chu trình hiện thời và không in kết quả lên màn hình.

```
#include <stdio.h>
main()
    {
    int i;
    int  a[10];
    for(i=0;i<10;i++)
        a[i]=0;
    for(i=0;i<10;i++)
```

```
        {
        printf("Hãy đưa số a[%d]\n",i);
        scanf("%d", &a[i]);
        }
for(i=0;i<10;i++)
        {
        if(a[i]%2 !=0)
            continue; /*Kết thúc chu trình nếu là số lẻ*/
        printf("a[%d] = %d\n", i,a[i]); /* In kết quả nếu là
              số chẵn*/
        }
    }
```

3.7 CÂU LỆNH GOTO VÀ NHÃN

Câu lệnh *goto* cho phép chuyển quyền điều khiển của chương trình đến một lệnh nào đó trong chương trình. Cú pháp của câu lệnh có dạng sau:

goto nhãn;

Nhãn có cùng dạng như tên biến và có 2 dấu chấm đứng sau. *Nhãn* có thể đứng trước câu lệnh bất kỳ trong chương trình. Các ví dụ sau minh họa cho việc sử dụng *nhãn*:

```
end:printf("Đây là nhãn\n");
```

Hoặc:

```
a15:
        if(x>0)
            x=5;
```

Trong 2 ví dụ này *end* và *a15* là nhãn.

Khi gặp câu lệnh *goto nhãn*, chương trình sẽ chuyển quyền

điều khiển cho lệnh có *nhãn* viết sau từ khóa *goto*. Chú ý rằng *goto* và *nhãn* phải nằm trong cùng một hàm. Như vậy câu lệnh *goto* chỉ cho phép chuyển quyền điều khiển từ vị trí này đến vị trí khác trong cùng một hàm. Nếu hàm có chứa một khối lệnh ta có thể chuyển quyền điều khiển giữa các khối lệnh này bằng câu lệnh *goto*, chẳng hạn như trong trường hợp sau:

```
void main()
{
int i,j,k;
float x,y;
...
A1:
    i=i+1;
    if(i>10)
        goto A2;
    ...
/*Đầu khối lệnh */
{
A2:
    if(x<5)
        goto A3;
    ...
    } /* Cuối khối lệnh */
    /* Đầu khối lệnh */
    {
    ...
    A3:
        i=j+k;
```

```
        ...
        } /* Cuối khối lệnh */
    ...
    }
```

Việc sử dụng câu lệnh *goto* để chuyển quyền điều khiển trong chương trình về nguyên tắc không có gì bị hạn chế. Tuy vậy nếu dùng *goto* để chuyển quyền điều khiển vào trong một khối lệnh thì nhìn chung có thể dẫn đến sai sót. Chẳng hạn trong trường hợp sau:

```
...
goto E;
    if(x>15)
        E:printf("Cần cẩn thận khi dùng toán lệnh goto\n");
    else
        z=x+2;
```

lệnh *goto* sẽ cho phép *printf* luôn được thực hiện mà không phụ thuộc vào bất cứ điều kiện nào. Tuy vậy sau khi thực hiện lệnh này, quyền điều khiển sẽ được chuyển cho câu lệnh *else* mặc dù trước đó chương trình không hề kiểm tra điều kiện trong *if*. Điều này tất nhiên sẽ bị trình biên dịch thông báo lỗi khi biên dịch. Để tránh tất cả các trường hợp rắc rối có thể xảy ra, tuyệt đối không nên dùng câu lệnh *goto* để nhảy vào trong thân của các câu lệnh *if,while, do-while* và *for*. Tuy nhiên, việc nhảy từ trong ra ngoài khối lệnh là hoàn toàn cho phép.

Việc sử dụng câu lệnh *goto* đang là một vấn đề còn nhiều bàn cãi. Nhìn chung để chương trình được viết có cấu trúc rõ ràng và thuận tiện cho việc sửa đổi, nên tránh sử dụng câu lệnh *goto*. Trên thực tế việc sử dụng câu lệnh này trong khi có những câu lệnh đầy đủ khác là hoàn toàn không nên. Câu lệnh *goto* chỉ nên dùng khi cần phải ngưng chương trình trong những trường hợp cần thiết.

3.8 CÂU LỆNH SWITCH

Câu lệnh *switch* căn cứ vào giá trị của biểu thức nguyên để chọn một trong nhiều nhóm lệnh. Thông thường lệnh có dạng sau:

switch(biểu thức nguyên)

```
        case m1:
                câu lệnh 1;
                break;
        case m2:
                câu lệnh 2;
                break;
        ...
        default:
                câu lệnh;
                break;
```

Ở đây *biểu thức nguyên* là điều kiện để kiểm tra việc rẽ nhánh, *mi* có thể là các số nguyên, hằng ký tự hoặc biểu thức cho giá trị nguyên.

Câu lệnh hoạt động qua các bước sau:

- Tính giá trị của *biểu thức nguyên.*
- Giá trị tính được của biểu thức sẽ được so sánh với giá trị của các *mi*.
- Nếu giá trị của biểu thức trùng với một giá trị *m1* nào đó trong các *mi*. Chương trình sẽ thực hiện tất cả các lệnh được viết sau nhãn *case m1*. Sau đó bằng câu lệnh *break* quyền điều khiển của chương trình sẽ được trả cho câu lệnh viết ngay sau thân *switch*.
- Nếu giá trị của biểu thức không trùng với bất cứ một hằng số nào trong các số *mi*, chương trình sẽ nhảy đến câu

lệnh có nhãn *default* (trường hợp có *default*) hoặc ra thoát khỏi câu lệnh *switch* (trường hợp không có *default*).

Ví dụ 3.15

r có thể là bán kính của vòng tròn, cạnh của hình vuông hoặc tam giác đều. Với giá trị cho trước của r hãy tính chu vi của các hình trên. Mỗi hình trên được mã hóa bởi:

'c' nếu là vòng tròn;

's' nếu là hình vuông;

't' nếu là tam giác đều;

Các mã trên được đưa từ bàn phím vào biến sys.

```
#include <stdio.h>
void main()
    {
    char sys; /* Chứa mã của các hình */
    float r,x; /* r bán kính hoặc cạnh, x-chu vi */
    int i=0; /* Biến điều khiển vòng lặp */
    printf("Hay đua ma cua hinh\n");
    scanf("%c", &sys);
    getchar();
    printf("Hãy đưa giá trị của r\n");
    scanf("%f", &r); getchar();
    do
    {
    switch(sys)
        {
        case 'c':
            x=2*3.14*r;
            i=0;
```

```
            break;
        case 's':
            x=4*r;
            i=0;
            break;
        case 't':
            x=3*r;
            i=0;
            break;
        default:
            printf("Mã đưa sai, hãy đưa giá trị khác\n");
            scanf("%c", &sys);
            i=1;
            break;
        }
    }
    while(i==1);
    printf("Chu vi cua hinh là % f\n",x);
    }
```

Khi thiết lập câu lệnh switch cần phải chú ý đến một số vấn đề sau:

• Các biểu thức *mi* không được có giá trị trùng nhau.

• Thứ tự sắp xếp của nhãn *case* và *default* là hoàn toàn bất kỳ. Trình tự tính toán và so sánh giá trị của *biểu thức nguyên* với các số *mi* là hoàn toàn phụ thuộc vào trình biên dịch. Điều đó có nghĩa là nhãn *case* được viết đầu tiên có thể không được trình biên dịch so sánh trước.

• Các nhãn *case* không bị hạn chế về số lượng, tuy nhiên số lượng này không được vượt quá các giá trị mà *biểu thức*

nguyên có thể nhận được.

• Cấu trúc cơ bản của *switch* có thể được biến dạng trong nhiều trường hợp.

• Một vài nhãn *case* có thể có chung một nhóm lệnh. Chẳng hạn:

```
...
case m1:
case m2:
case mk:
    Câu lệnh;
...
```

Trong trường hợp này nếu giá trị của *biểu thức nguyên* trùng với một trong các giá trị *m1,m2,...,mk* thì *câu lệnh* sẽ được thực hiện.

• Các câu lệnh *break* trong cấu trúc *switch* có thể bị bỏ qua. Trong trường hợp này, sau khi thực hiện các lệnh của nhánh, nếu không gặp *break* chương trình sẽ thực hiện các lệnh của nhánh tiếp theo. Nếu không sử dụng một câu lệnh *break* nào trong cấu trúc *switch* thì các lệnh trong thân *switch* sẽ được thực hiện lần lượt từ đầu đến cuối.

• Nhãn *default* không bắt buộc phải có. Trong trường hợp không dùng nhãn nếu giá trị của *biểu thức nguyên* không trùng với bất cứ giá trị nào của *mi* thì sẽ không có lệnh nào của *switch* được thực hiện.

Chú ý rằng câu lệnh *switch* có thể được thay thế bởi cấu trúc *if-else* bằng cách sau:

```
if(biểu thức nguyên == m1)
```

câu lệnh;

else if(biểu thức nguyên == m3)

...

Trong đa số các trường hợp, việc sử dụng *switch* sẽ làm cho chương trình được rõ ràng hơn. Tuy vậy việc sử dụng cấu trúc *if-else* cũng có ưu điểm của nó. Cấu trúc này cho phép ta chọn sẵn thứ tự để thực hiện việc so sánh với các *mi*. Điều này cho phép ta chọn khả năng có thể xảy ra nhất để so sánh trước tiên và do vậy sẽ làm cho chương trình trở thành hiệu quả hơn. Tuy nhiên trong nhiều trường hợp ta có thể dùng câu lệnh *switch* để th y thế cho các cấu trúc *if-else* phức tạp.

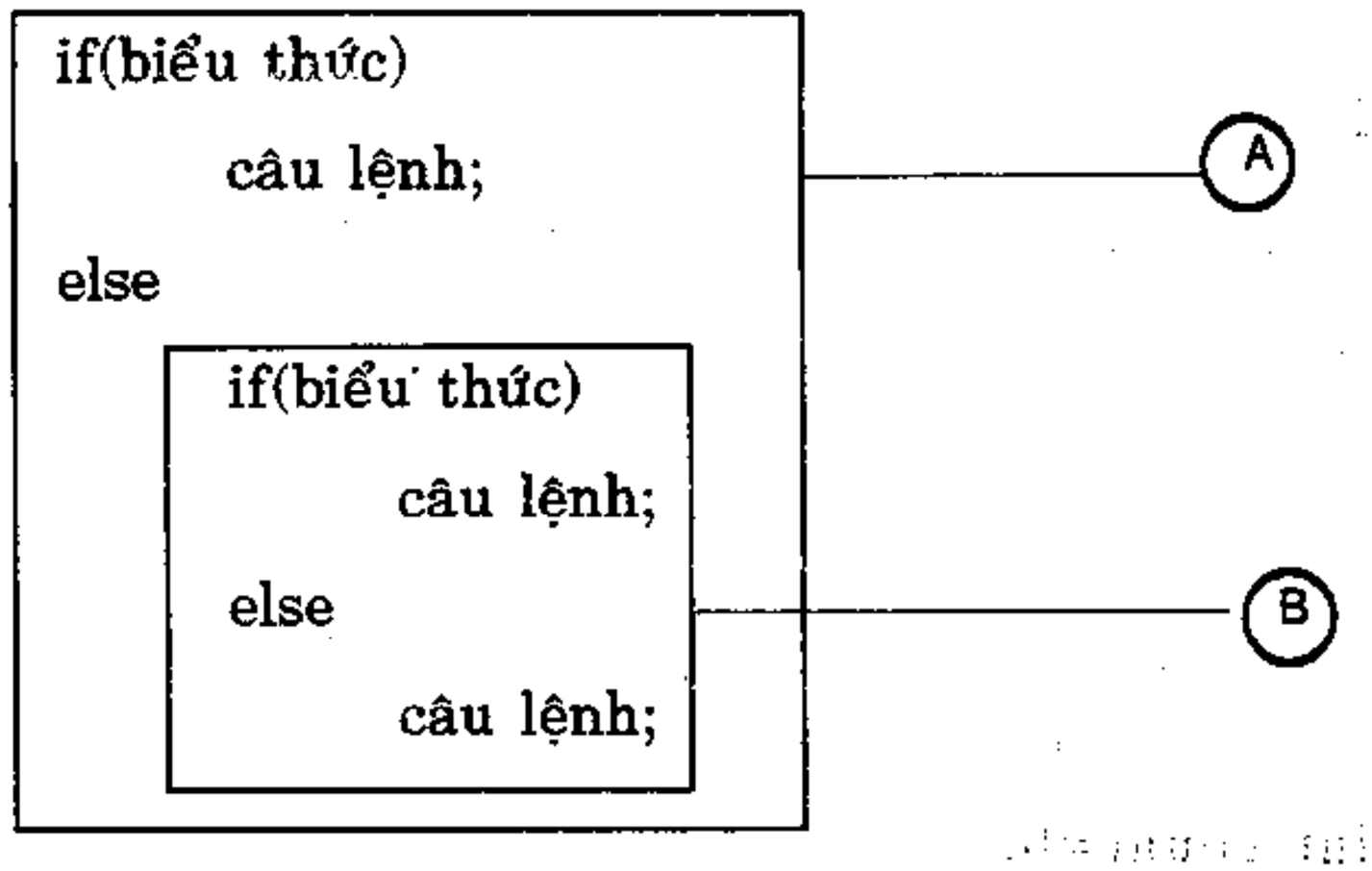
3.9 CÂU LỆNH LỒNG

Trong các câu lệnh đã xét ở trên, khi một câu lệnh này chuyển quyền điều khiển cho một câu lệnh khác cùng loại ta nói rằng câu lệnh thứ hai được lồng trong câu lệnh thứ nhất v.v. Trong ngôn ngữ C, số lượng các câu lệnh được viết lồng nhau là không hạn chế. Về sau minh họa cho việc sử dụng các câu lệnh viết lồng nhau.

if(biểu thức)
if(biểu thức)
if(biểu thức)
câu lệnh;
else
câu lệnh;
else
câu lệnh;

Hình 3.1a

```
if(biểu thức)
        câu lệnh;
else
        if(biểu thức)
                câu lệnh;
        else
                if(biểu thức)
                        câu lệnh;
                else
                        câu lệnh;
```



Hình 3.1b

Trong câu lệnh *if-else*, cấu trúc lồng có thể có trong cả 2 phần *if* và *else* hoặc là dạng tổ hợp của cả hai loại.

Trên thực tế cấu trúc ở dạng *3.1b* thường hay được sử dụng hơn vì dễ hiểu hơn.

Tương tự như *if-else*, *while* và *do-while* có thể được dùng trong cấu trúc lồng sau:

```
while(biểu thức)
        while(biểu thức)
                câu lệnh;
```

Trong kỹ thuật lập trình, cấu trúc lồng của câu lệnh *for* được sử dụng rất nhiều. Cấu trúc này có dạng sau:

```
for(biểu thức 1; biểu thức 2; biểu thức 3)
        for(biểu thức 1; biểu thức 2; biểu thức 3)
                for(biểu thức 1; biểu thức 2; biểu thức 3)
                        câu lệnh;
```

Trong cấu trúc này vòng lặp *for* trong cùng sẽ được thực hiện *k*l*m* lần nếu vòng lặp của chu trình *for* đầu là *k* lần, vòng lặp của chu trình *for* thứ 2 là *l* lần và vòng lặp của chu trình *for* trong cùng là *m* lần.

Ví dụ 3.15

Tìm tất cả các số nguyên tố nhỏ hơn một số được đưa vào từ bàn phím.

```
#include <stdio.h>
void main()
{
    int number,divisor,limit;
    int count=0;
    printf("Đưa giới hạn trên cần tìm\n");
    printf("Giới hạn trên phải >=2\n");
    scanf("%d", &limit);
  while(limit<2)
      {
          printf("Hãy đưa số khác\n");
          scanf("%d", &limit);
          }
      printf("Bắt đầu in các số nguyên tố nhỏ hơn %d\n",limit);
      for(number=2;number<=limit;number++)
          {
          for(divisor = 2;number%divisor !=0;divisor++)
          ;
          if(divisor == number)
```

```
                if(++count%10==0)/*In 10 số trên một dòng*/
                    printf("\n");
            }
        }
        printf("\nĐây là tất cả các số nguyên tố nhỏ hơn %d\n",
            limit);
    }
```

Trong chương trình trên có dùng thêm vòng lặp:

for (number=2; number<=limit;number++). Vòng lặp này có tác dụng kiểm tra tất cả các số nguyên nhỏ hơn *limit* có phải là số nguyên tố hay không.

Ví dụ 3.16

Cộng hai ma trận a và b kiểu int có kích thước là 3x3. Kết quả được đưa vào ma trận c. Giá trị các phần tử của ma trận a và b được đưa vào từ bàn phím.

```
#include <stdio.h>
main()
    {
    int a[3][3],b[3][3],c[3][3];
    int i,j,k;
    /* Nhập giá trị từ bàn phím */
    for(k=0;k<2;k++)
    {
    if(k==0)
        printf("Đưa ma trận a\n");
    else
        printf("Đưa ma trận b\n");
    printt("\n\n");
```

```
for(i=0;i<3;i++)
    for(j=0;j<3;j++)
        {
        printf("Hãy đưa giá trị các phần tử\n");
        printf("%d %d) của ma trận \n",i,j);
        scanf("%d",(k==0)? &a[i][i]: &b[i][i]);
        }
    }
/* Tính tổng hai ma trận */
for(i=0;i<3;i++)
    for(j=0;j<3;j++)
        c[i][j] = a[i][j] + b[i][j];
    printf("Các phần tử của ma trận C là:\n");
for(i=0;i<3;i++)
    for(j=0;j<3;j++)
        printf("c[%d][%d] = %d\n",i,j,c[i][i]);
}
```

Chương IV

HÀM VÀ CÁC CẤP LƯU TRỮ BIẾN

Hàm là đơn vị cơ bản trong ngôn ngữ C khi thực hiện chương trình. Sau khi thực hiện, hàm có thể được nhận một giá trị nào đó tương ứng với kiểu của hàm (xem chương I).

Việc sử dụng hàm cho phép ta chia chương trình thành những chương trình nhỏ (module) tương đối độc lập với nhau dựa trên các nguyên tắc của lập trình có cấu trúc. Chính các nguyên tắc này sẽ có tác dụng làm cho chương trình trở thành gọn gàng, dễ hiểu và thuận tiện khi cần phải sửa lỗi. Ngoài ra việc tạo ra các hàm còn cho phép ta sử dụng các hàm này trong các chương trình khác nhau khi cần thiết.

Trong chương này chúng ta sẽ đề cập đến các vấn đề liên quan đến hàm trong ngôn ngữ C như: khai báo các hàm, cơ cấu truyền thông tin giữa các hàm khác nhau và các loại biến khi sử dụng các hàm.

4.1 ĐỊNH NGHĨA HÀM

Ở chương I chúng ta đã đưa ra cấu trúc ngắn gọn của hàm được viết trong C. Cấu trúc đầy đủ của hàm được biểu diễn qua cách sau:

```
Kiểu_kết_quả Tên_hàm(khai báo danh sách các tham số)
    {
    Khai báo các biến cục bộ;
    Các lệnh;
        ...
        return(biểu thức);
```

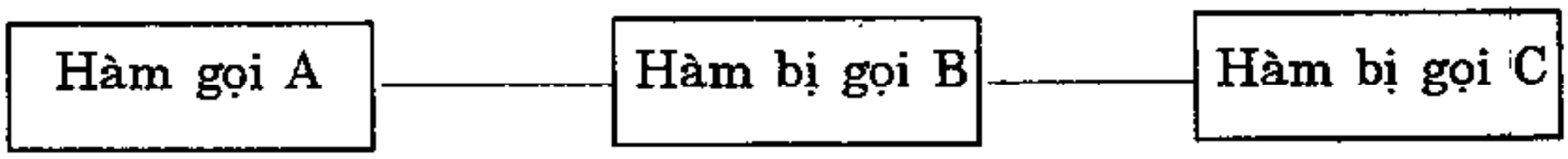
```
    Các lệnh;
        ...
        return(biểu thức);
    Các lệnh;
        ...
}
```

Qua cách khai báo này có thể thấy hàm được biểu diễn như một đơn vị độc lập trong chương trình vì nó có chứa việc khai báo các biến và một số lệnh nào đó cần phải thực hiện. Các phần này phải được viết trong 2 dấu ngoặc {} và được gọi là thân của hàm. Các biến được sử dụng trong hàm phải được khai báo trước lệnh đầu tiên được viết trong thân hàm, nghĩa là ngay sau dấu mở ngoặc {.

• Thông thường mỗi chương trình đều phải giải quyết một vấn đề tổng thể nào đó. Vấn đề này đến lượt mình thường được chia làm nhiều vấn đề và được xây dựng qua các hàm. Về nguyên tắc, các hàm trong C không nên quá lớn, tuy vậy để giải quyết vấn đề này thì số lượng các hàm trong chương trình lại trở thành quá nhiều, do đó trước khi giải một bài toán nào đó ta cần phải xây dựng một cấu trúc tối ưu cho chương trình.

• Hàm là đơn vị độc lập trong chương trình và vì vậy để các lệnh của hàm có thể thực hiện ta phải chuyển quyền điều khiển cho hàm từ một hàm khác. Hàm nhận quyền điều khiển được gọi là hàm bị gọi, hàm mà từ đó quyền điều khiển được trao cho hàm khác được gọi là *hàm gọi*. Một điểm cần lưu ý là sau khi *hàm bị gọi* kết thúc công việc của mình thì quyền điều khiển lại được trao cho hàm gọi nó.

Một hàm bị gọi lại có thể gọi một hàm khác v.v... Hình 4.1 minh họa cho quan hệ giữa 3 hàm A, B, C.



Hình 4.1

• Ở đây đối với hàm A thì hàm B là *hàm bị gọi*, tuy vậy đối với C thì B lại là hàm gọi.

• Trong chương trình phải có một hàm và chỉ được phép có một hàm với tên là *main* (hàm chính). Thông qua hàm chính này chương trình sẽ thực hiện việc trao đổi thông tin giữa các hàm khác nhau và với hệ điều hành. Trong C cũng đưa ra một khả năng cho phép thoát khỏi hàm và quay về hệ điều hành mà không cần phải về hàm chính, khả năng này được thực hiện thông qua lệnh *exit*.

• Một hàm được chuyển quyền điều khiển thông qua tên của nó. Tên của hàm phải là duy nhất và được đặt cho các nguyên tắc như khi đặt tên cho biến.

• Thông tin được trao đổi giữa các hàm khác nhau phải được khai báo trong danh sách các đối số. Các đối số này được khai báo sau tên của hàm số. Các đối số phải được viết cách nhau bởi dấu phẩy và phải nằm trong hai dấu ngoặc đơn.

• Thông thường sau khi thực hiện, hàm được nhận một giá trị nào đó tương ứng với kiểu khi khai báo. Khả năng này được thực hiện thông qua lệnh return.

Ví dụ 4.1

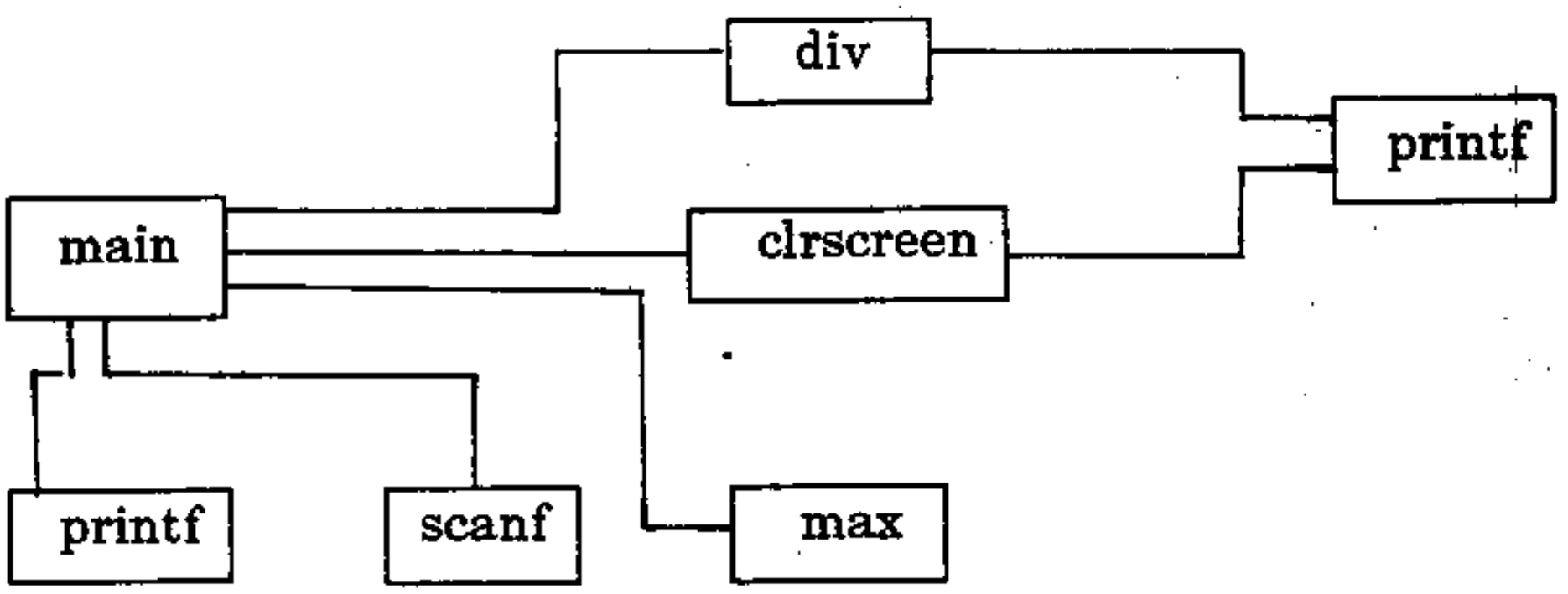
Chương trình sau có sử dụng 3 hàm. Hàm div dùng để xác định ước số chung lớn nhất của hai số và in kết quả ra màn hình. Hàm max xác định giá trị cực đại của hai số. Hàm clrscreen dùng để xóa màn hình. Hàm main lần lượt chuyển quyền điều khiển cho các hàm trên.

```
#include <stdio.h>
int max(int, int);
```

```
void main()
    {
    int x,n1,n2:
    clrscreen();
    printf("Hãy đưa số thứ nhất\n");
    scanf("%d", &n1);
    printf("Hãy đưa số thứ hai\n");
    scanf("%d", &n2);
    div(n1,n2);
    x = max(n1,n2);
    printf("Giá trị cực đại của hai số là %d\n",x);
    }
void div(int a, int b)
    {
    int t;
    while(b !=0)
        {
        t = a%b;
        a = b;
        b = t;
        }
    printf("Ước số chung lớn nhất là %d\n", a );
    }
clrscreen()
    {
    int br;
    for(br=0; br<=25; br++)
        printf("\n");
```

```
    }
int max(int c, int d)
    {
    return(c>d)?c:d;
    }
```

Trong ví dụ trên các hàm được liên kết với nhau theo sơ đồ sau:



Hình 4.2

Để ý thấy:

- Hàm *div* không trả lại giá trị. Giá trị của ước số chung được in ngay trong hàm.
- Hàm *max* trả lại giá trị sau khi được thực hiện.
- Hàm *clrscreen* không trả lại giá trị và không có tham số.

4.2 THAM SỐ VÀ CÁCH TRUYỀN THAM SỐ TRONG HÀM

4.2.1 Tham số

Khi xây dựng một hàm, các biến trong danh sách các đối số không nhận một giá trị cụ thể nào. Trong danh sách các đối số, qua việc khai báo ta chỉ xác định kiểu của các đối số được truyền cho hàm. Các biến này được gọi là các *tham số hình*

thức và sau khi khai báo, chúng được sử dụng như bất kỳ một biến cục bộ nào khác được khai báo trong thân hàm. Các biến có giá trị cụ thể sẽ thay đổi các tham số hình thức khi một hàm được chuyển quyền điều khiển được gọi là các *tham số thực*. Giữa tham số thực và tham số hình thức phải có sự tương ứng về kiểu cũng như về số lượng.

Một hàm có thể không chứa các tham số (giống như hàm *clrscreen* trong ví dụ 4.1), tuy vậy trong trường hợp đó vẫn bắt buộc phải có các dấu ngoặc đơn () sau tên của hàm.

Thông thường khi một hàm được gọi, tất cả các tham số thực thuộc kiểu *float* đều tự động được chuyển về kiểu *double*, các tham số khác thuộc kiểu *short int* hoặc *char* được chuyển về kiểu *int*. Nếu một tham số hình thức được khai báo theo kiểu *float* thì khi đổi kiểu, trình biên dịch sẽ không thông báo lỗi.

Sau đây là một ví dụ khi gọi hàm:

symbol ('a');	Hàm symbol có tham số thực là hằng kiểu ký tự.
z=2*p+err(s,rt);	Hàm err được sử dụng như một toán hạng khi tính giá trị của z.
call(10,20);	Các tham số của hàm call là các hằng thuộc kiểu số nguyên.

4.2.2 Truyền tham số

Trong ngôn ngữ C có nhiều cách truyền thông tin giữa các hàm khác nhau. Cụ thể có thể chia thành một số cách sau:

4.2.2.1. Truyền tham số theo trị

Trong cách truyền tham số này, hàm chỉ làm việc với các bản sao giá trị của các tham số thực khi chúng được truyền qua hàm. Khi hàm được gọi, giá trị của các tham số thực được sao chép sang ngăn xếp của chương trình và chính các giá trị này sẽ được sử dụng trong hàm chứ không phải là bản thân

các tham số thực. Chính vì nguyên nhân này mà mọi sự thay đổi giá trị trong hàm sẽ không dẫn đến việc thay đổi giá trị của các tham số thực được truyền cho hàm.

Ví dụ 4.2

```
#include <stdio.h>
void hoanvi(int, int);
void main()
    {
    int x=5, y = 7;
    printf("Giá trị của x và y trước khi hoán vị là: \n");
    printf("x = %d, y = %d\n", x, y);
    hoanvi(x,y);
    printf("Giá trị của x và y sau khi hoán vị là: \n");
    printf("x = %d y = %d\n", x, y);
    }
void hoanvi(int x, int y)
    {
    int temp;
    temp=x;
    x=y;
    y=temp;
    }
```

Sau khi thực hiện chương trình sẽ cho kết quả sau trên màn hình:

Giá trị của x và y trước khi hoán vị là:

x = 5 y = 7

Giá trị của x và y sau khi hoán vị là

x = 5 y = 7

Trong hàm *hoanvi* rõ ràng rằng giá trị của *x* và *y* được hoán vị cho nhau thông qua biến trung gian *temp*. Tuy vậy hai giá trị này chỉ là bản sao của *x* và *y* do đó không phản ánh lên hai biến này ở hàm gọi, vì thế giá trị của *x* và *y* ở hàm *main* không có gì thay đổi.

Ví dụ 4.3

```
#include <stdio.h>
int sqr(int x)
    {
    x=x*x;
    return(x)
    }
void main()
    {
    int y = 10;
    printf("Giá trị của y trước khi gọi là: %d\n", y);
    printf("Bình phương của y là: %d\n", sqr(y));
    printf("Giá trị của y sau khi gọi là: %d\n", y);
    }
```

Trong ví dụ này, tham số thực y có giá trị bằng 10 được truyền cho hàm *sqr* khi hàm *main* gọi hàm *sqr*. Mặc dù giá trị *x* trong hàm *sqr* bị thay đổi nhưng bởi vì giá trị của *x* chỉ là bản sao của *y* do đó giá trị của *y* sau khi thực hiện hàm số *sqr* vẫn không bị thay đổi. Dưới đây là kết quả nhận được sau khi thực hiện chương trình:

Giá trị của *y* trước khi gọi là: 10

Bình phương của *y* là: 100

Giá trị của *y* sau khi gọi là: 10

4.2.2.2. Truyền tham số qua cách khai báo

Tất cả các hàm trong chương trình đều có thể truy nhập đến một biến nào đó nếu biến được khai báo là *biến tổng thể*. Trong trường hợp này việc khai báo các biến tổng thể phải được tiến hành trước tất cả các hàm, kể cả hàm main.

Việc sử dụng các biến tổng thể khi truyền dữ liệu giữa các hàm khác nhau sẽ làm cho chương trình được viết dễ dàng hơn. Tuy vậy nó cũng có một số nhược điểm cơ bản sau:

• Tên của biến cục bộ có mức độ ưu tiên cao hơn biến tổng thể khi thực hiện, điều đó có nghĩa là nếu trong một hàm nào đó có biến cục bộ trùng tên với biến tổng thể thì hàm sẽ sử dụng giá trị của biến cục bộ, khi đó biến tổng thể được coi như là không được định nghĩa.

Hãy xét lại ví dụ 4.2. Để có thể hoán vị được giá trị của *x* và *y* ta có thể khai báo *x* và *y* là hai biến tổng thể. Cần lưu ý là một biến được gọi là tổng thể nếu nó được khai báo ở đầu chương trình, trên tất cả các hàm sử dụng nó.

```
#include <stdio.h>
int x,y;
void main()
    {
    x=5; y=7;
    printf("Giá trị của x và y trước khi hoán vị là: \n");
    printf( "x = %d y = %d\n", x, y);
    hoanvi();
    printf("Giá trị của x và y sau khi hoán vị là: \n");
    printf( "x = %d y = %d\n", x, y);
    }
```

```
void hoanvi()
    {
    int temp;
    temp=x;
    x=y;
    y=temp;
    }
```

Sau khi thực hiện, chương trình sẽ hiện kết quả sau lên màn hình:

Giá trị của x và y trước khi hoán vị là:

x = 5 y = 7

Giá trị của x và y sau khi hoán vị là:

x = 7 y = 5

Giá trị của *x* và *y* rõ ràng đã được hoán vị sau khi thực hiện chương trình. Điều này là do *x* và *y* là hai biến tổng thể và do đó hàm *hoanvi* có thể làm việc trực tiếp với hai biến này.

• Làm cho các hàm mất tính độc lập của chúng và gây ra nhiều khó khăn khi sửa đổi chương trình.

Hãy xét ví dụ dưới đây:

```
#include <stdio.h>
int x,y;
void main()
    {
    int m,n;
    x=5; y = 7;
    m=1; n=2;
    printf ("Giá trị của x và y trước khi hoán vị là: \n");
    printf ("x = %d y = %d\n", x, y);
```

```
hoanvi();
printf("Giá trị của x và y sau khi hoán vị là: \n");
printf("x = %d y = %d\n", x, y);
printf("Giá trị của m và n trước khi hoán vị là: \n");
printf("m = %d n = %d\n", m, n);
hoanvi();
printf("Giá trị của m và n sau khi hoán vị là: \n");
printf("m = %d n = %d\n", m, n);
printf("Giá trị của x và y là: \n");
printf("x = %d y = %d\n", x, y);
}
void hoanvi()
{
int temp;
temp=x;
x=y;
y=temp;
}
```

Sau khi thực hiện, chương trình cho kết quả sau trên màn hình:

Giá trị của x và y trước khi hoán vị là:

x = 5 y = 7

Giá trị của x và y sau khi hoán vị là:

x = 7 y = 5

Giá trị của m và n trước khi hoán vị là:

m = 1 n = 2

Giá trị của m và n sau khi hoán vị là:

m = 1 n = 2

Giá trị của x và y là:

x = 5 y = 7

Có thể nhận thấy rằng sau khi hàm *hoanvi( )* được gọi lần thứ hai với mục đích hoán vị giá trị của *m* và *n* thì giá trị của *m* và *n* vẫn không đổi mà thay vào đó giá trị của *x* và *y* lại được hoán vị cho nhau. Nguyên nhân là hàm này chỉ làm việc trực tiếp với hai biến *x* và *y* mà không làm việc với hai biến *m* và *n*. Như vậy với cách sử dụng biến tổng thể thì hàm sẽ phải phụ thuộc vào các biến tổng thể này và mất đi tính độc lập của mình. Điều này có thể bị loại bỏ nếu ta truyền tham số cho hàm dưới dạng con trỏ.

Ví dụ 4.4

Xây dựng chương trình giải phương trình bậc hai $ax^2+bx+c=0$. *Các hệ số của phương trình được đưa vào từ bàn phím bằng hàm scanf. Nghiệm của phương trình x1, x2.*

```
#include <stdio.h>
#include <math.h>
float x1,x2; /* x1 và x2 là nghiệm của phương trình */
int i;
void main()
    {
    float a,b,c; /*Hệ số của phương trình */
    printf ("Hãy đưa các hệ số a,b,c của phương trình \n");
    scanf ("%f%f", &a, &b, &c);
    esq(a,b,c);
    if(i==0)
        {
        printf("Phương trình có hai nghiệm khác nhau\n");
        printf("x1=%f x2=%f\n"x1,x2);
```

```
    }
else
    if(i==1)
        {
        printf("Phương trình có hai nghiệm kép\n");
        printf("x=%f\n",x1);
        }
    }
esq(double x, double y, double z)
    {
    double r,s,t; /* Các biến cục bộ */
    if((r=y*y) <(s=4*x*z))
        {
        printf("Phương trình vô nghiệm\n")
        i=2;
        goto end;
        }
t = 2*x;
if ((r=r-s) !=0) /* Tính hai nghiệm */
    {
    s=sqrt(r);
    x1=(-y+s)/t;
    x2=(-y-s)/t;
    i=0;
    }
else /* Tính nghiệm kép */
    {
    x1=- y/t;
```

```
            i=1;
            }
        end:
    }
```

Hai biến *x1* và *x2* dùng để ghi kết quả nghiệm của phương trình cùng với biến điều khiển *i* được khai báo như các biến tổng thể do đó các hàm *main, esq* có thể trực tiếp sử dụng giá trị của chúng mà không cần phải truyền tham số. Mặt khác mọi thay đổi về giá trị của các biến này trong bất cứ hàm nào sẽ dẫn đến làm thay đổi giá trị của chúng trong các hàm khác. Các hệ số *a,b,c* được khai báo trong hàm *main* và được sử dụng như tham số truyền cho hàm *esq*. Hàm *esq* dùng để tính nghiệm của phương trình phụ thuộc vào các hệ số của phương trình. Nghiệm *x1, x2* được tính trong hàm *esq* và được in ra từ hàm *main* tùy thuộc vào từng trường hợp (i=1 phương trình có nghiệm kép, i khác 1 phương trình có hai nghiệm khác nhau). Trong hàm *esq* có sử dụng lệnh *goto*, lệnh *goto* trong trường hợp này được dùng để kết thúc việc thực hiện hàm khi phương trình vô nghiệm.

4.2.2.3. Truyền tham số theo địa chỉ

Trong cách truyền tham số này, các tham số được truyền theo địa chỉ và trên thực tế ta đang làm việc với bản gốc của các tham số đó. Như vậy mọi sự thay đổi giá trị của các biến trong hàm bị gọi sẽ dẫn đến sự thay đổi giá trị của các bản gốc. Để thực hiện việc truyền tham số theo địa chỉ, trong ngôn ngữ C có sử dụng con trỏ (xem chương VI). Cách truyền tham số này kỳ thực ta đã sử dụng rất nhiều trong các ví dụ có sử dụng hàm *scanf* để đưa dữ liệu từ bàn phím. Các tham số truyền của hàm *scanf* là địa chỉ của các biến được đưa từ bàn phím. Khi đưa dữ liệu, hàm *scanf* sẽ thay đổi giá trị của các biến và bởi vì ở đây, hàm sử dụng địa chỉ của các biến nên sự

thay đổi này sẽ phản ánh lên các biến được truyền cho hàm.

Việc sử dụng con trỏ trong C sẽ được đề cập chi tiết hơn trong chương VI. Ở đây xin nêu một ví dụ đơn giản về cách sử dụng con trỏ thông qua chương trình hoán vị giá trị 2 số nguyên.

```
#include <stdio.h>
void hoanvi(int*,int*);
void main()
    {
    int x,y,m,n;
    x=5; y=7;
    m=1; n=2;
    printf("Giá trị của x và y trước khi hoán vị là:\n");
    printf("x=%d y=%d\n");
    printf("Giá trị của x và y sau khi hoán vị là:\n");
    hoanvi(&x;&y);
    printf("x=%d y=%d\n");
    printf("Giá trị của m và n trước khi hoán vị là:\n");
    printf("m=%d n=%d\n");
    hoanvi(&m,&n);
    printf("Giá trị của m và n sau khi hoán vị là:\n");
    printf("m=%d n=%d\n");
    }
void hoanvi(int* x,int* y)
    {
    int temp;
    temp=*x;
    *x=*y;
```

```
*y=temp;
}
```

Một vấn đề cũng cần lưu ý đó là sử dụng mảng như tham số truyền. Đối với các mảng một chiều, mảng được truyền qua tên của nó và bỏ các dấu ngoặc vuông sau tên mảng. Ví dụ mảng *int a[5]* gồm 5 phần tử thuộc kiểu *int* sẽ được truyền qua hàm bằng phương pháp sau: *example(a)*.

Đối với các mảng nhiều chiều, khi khai báo mảng như tham số hình thức, kích thước đầu của mảng có thể được bỏ qua, nhưng các kích thước khác bắt buộc phải được khai báo. Trong thân của hàm, các phần tử khác nhau được truy nhập theo phương pháp thông thường - qua chỉ số.

Bản chất của việc sử dụng mảng như tham số truyền đó là mảng được truyền qua địa chỉ của nó. Trong các trường hợp này, thực chất là ta thực hiện việc truyền địa chỉ của phần tử đầu của mảng cho hàm. Địa chỉ này được sử dụng như địa chỉ cơ sở và qua nó bằng phép dịch chuyển ta có thể truy nhập đến phần tử bất kỳ của mảng. Điều này có nghĩa là trong thân hàm ta đang làm việc trực tiếp với các phần tử gốc của mảng chứ không phải với các bản sao của chúng trong ngăn xếp. Chính vì vậy, khi giá trị của các thành phần của mảng trong thân hàm bị thay đổi thì giá trị gốc của chúng cũng sẽ thay đổi theo.

Ví dụ 4.5

Từ bàn phím hãy đưa các phần tử của ma trận vuông với kích thước 3x3. Hãy xây dựng hàm trans để tính phần tử của ma trận chuyển vị và ghi chúng vào mảng. Ma trận chuyển vị là ma trận được nhận từ một ma trận cho trước bằng cách đổi hàng thành cột.

Đây là một ví dụ minh họa cho việc sử dụng mảng như

tham số truyền trong hàm và cách khai báo chúng khi định nghĩa hàm.

```
#include <stdio.h>
#define DIM 3 /* Định nghĩa DIM = 3 */
void main()
{
int u[DIM][DIM],v[DIM][DIM]; /* u - ma trận ban đầu */
                             /* v- ma trận chuyển vị */
int i,j;
    /* đưa các phần tử của u */
for(i =0,i<DIM; i++)
    for(j=0, j <DIM; j++)
        {
        printf("Đưa phần tử");
        printf("%d,%d) của ma trận \n",i+1,j+1);
        scanf("%d", &u[i][j]);
        }
trans(u,v);
for(i=0;i<DIM;i++)
    {
    printf("hàng %d",i+1);
        for(j=0;j<DIM<;j++)
            printf("%d",v[i][j]);
            printf("\n");
        }
    }
trans int a[][DIM], int b[][DIM]
    {
```

```
    int m,n;
    for(m=0;m<DIM;++m)
        for(n=0;n<DIM;++n)
            b[n][m]=a[m][n];
    }
```

4.3 TRẢ LẠI GIÁ TRỊ CHO HÀM

Trong hàm ta có thể thay đổi giá trị của các biến như đã trình bày ở các phần trên. Ngoài ra hàm bị gọi có thể tính và trả lại một giá trị nào đó có thể có kiểu tương ứng với kiểu của hàm khi được định nghĩa cho hàm gọi. Để thực hiện được điều này giữa hàm gọi và hàm bị gọi phải có các mối liên hệ như sau:

• Hàm bị gọi phải được viết trong hàm gọi như là một toán hạng của biểu thức. Ví dụ m = max(x,y)+t.

• Hàm bị gọi thực hiện việc tính toán và giá trị đó được trả lại cho hàm gọi qua lệnh *return*.

• Kiểu của kết quả phải tương ứng với kiểu của hàm bị gọi.

Như vậy, khi một hàm được sử dụng như một toán hạng của một biểu thức, giá trị của hàm sẽ được sử dụng khi tính biểu thức. Để một hàm có thể trả lại một giá trị nào đó trong thân hàm phải sử dụng lệnh *return* với cú pháp sau:

return(biểu thức);

Khi sử dụng *return*, hai dấu ngoặc đơn không bắt buộc phải có, tuy vậy việc sử dụng chúng sẽ làm cho chương trình trở nên rõ ràng và dễ hiểu hơn. Trong biểu thức có thể tham gia các biến tổng thể, biến cục bộ hoặc các hằng số, thậm chí có thể chứa các lệnh gọi đến hàm khác hoặc biểu thức có điều kiện v.v.

Câu lệnh *return* có thể được viết tại vị trí bất kỳ trong hàm

và hàm có thể chứa một số lệnh *return*. Tuy nhiên cần chú ý rằng phụ thuộc vào các điều kiện được đưa ra bao giờ cũng chỉ có duy nhất một lệnh *return* trong số đó được thực hiện.

Để hàm trả lại giá trị cho hàm gọi nó, khi định nghĩa hàm ta phải khai báo kiểu cho hàm bị gọi. Nếu kiểu của hàm không được khai báo thì khi biên dịch, trình biên dịch sẽ tự gán cho hàm kiểu *int*.

Lệnh return thực hiện qua các bước sau:

- Tính kết quả của biểu thức được viết sau lệnh.
- Chuyển kiểu của kết quả khi cần thiết.
- Kết thúc việc thực hiện của hàm bị gọi.
- Quyền điều khiển được trao cho hàm gọi và trong hàm gọi, hàm bị gọi được thay thế bởi giá trị của nó.

Biểu thức không nhất thiết phải sử dụng sau lệnh *return*. Trong trường hợp đó hàm sẽ không trả lại kết quả cho hàm gọi và thông qua lệnh *return* đơn giản là quyền điều khiển sẽ được trao lại cho hàm gọi nó. Lệnh *return* rất hữu hiệu khi cần phải kết thúc việc thực hiện của hàm bị gọi trong một cấu trúc logic nào đó.

Ví dụ 4.6

Xây dựng hàm tính lũy thừa của một số nguyên và viết chương trình sử dụng hàm này.

```
#include <stdio.h>
#include <conio.h>
/* Hàm tính lũy thừa */
int power(int base, int exp)
    {
    int result;
    for (result=1; exp>0;exp--)
```

```
            result *=base;
        return(result);
        }
    /* Hàm chính main */
void main()
    {
    int x;
    clrscr(); /* Xóa màn hình */
    x=power(2,3);
    printf("%d\n",x);
    x=power(-3,3);
    printf("%d\n",x);
    x=power(4,-2);
    printf("%d\n",x);
    x=power(5,10);
    printf("%d\n",x);
    }
```

Sau khi thực hiện chương trình trên cho kết quả:

8

-27

1

761

và như vậy 2 lũy thừa 3 bằng 8

-3 lũy thừa 3 bằng -27

4 lũy thừa -2 bằng 1

5 lũy thừa 10 bằng 761

Trong kết quả này thì rõ ràng kết quả thứ 3 và thứ 4 là sai. Kết quả thứ 3 sai do hàm không được xây dựng để tính

lũy thừa của số âm. Kết quả 4 sai là do giá trị của x sau khi tính vượt quá giới hạn của nó khi khai báo.

Để có thể xác định chính xác lũy thừa của một số chúng ta cần mở rộng hàm *power* cho nhiều trường hợp và kiểu biến khác nhau. Hàm được xây dựng để tính cả lũy thừa âm của một số kiểu *double*.

Dưới đây là hàm *power*.

```
double power(double base, int exp)
    {
    double result;
        if(exp>0) /* lũy thừa > 0 */
            {
            for(result=1.0;exp>0; exp--)
                result*=base;
            return(result);
            }
        else if(base!=0) /* lũy thừa < 0 */
            {
            for(result=1.0;exp<0; exp++)
                result/=base;
            return(result);
            }
        else
            {
            printf("Không thực hiện phép lũy thừa %d của 0\n", e
            return(0);
            }
    }
```

Hàm này có thể được sử dụng thông qua hàm main dưới đây:

```
void main()
    {
    double x;
    clrscr();
    x=power(2.0,3);
    printf("2.0 lũy thừa 3 là %f\n", x);
    x=power(-3.0,3);
    printf("-3.0 lũy thừa 3 là %f\n", x);
    x=power(4.0,-2);
    printf("4.0 lũy thừa -2 là %f\n", x);
    x=power(5.0,10);
    printf("5.0 lũy thừa 10 là %f\n", x);
    }
```

Sau khi thực hiện hàm cho kết quả:

```
2.0 lũy thừa 3 là       8.000000
-3.0 lũy thừa 3 là      -27.000000
4.0 lũy thừa -2 là      0.062500
5.0 lũy thừa 10 là      9765625.000000
```

Ví dụ 4.7

Xây dựng hàm tính giá trị tuyệt đối của một số nguyên.

```
int user_abs(int x)
    {
    int y;
    return(y=x<0?-x:x);
    }
```

Trong hàm này biểu thức được viết sau lệnh *return* là một biểu thức có điều kiện. Hàm trên còn có thể được viết theo cách sau:

```
int user_abs(int x)
    {
    if(x<0)
        return(-x);
    else
        return(x);
    }
```

Theo cách viết này thì trong hàm được xây dựng có sử dụng hai câu lệnh *return*. Tuy vậy tùy theo từng điều kiện cụ thể bao giờ cũng chỉ có một lệnh *return* duy nhất được thực hiện.

Hàm tính giá trị tuyệt đối có thể được gọi từ các hàm khác theo cách sau:

```
#include <stdio.h>
void main()
    {
    int a=10,b=0,c=-10;
    int d,e,f;
    d=user_abs(a);
    e=user_abs(b);
    f=user_abs(c);
    printf ("Giá trị tuyệt đối của a,b,c là %d %d %d\n",d,e,f);
    }
```

Cần chú ý rằng cho đến nay cả hai hàm *main()* và *user_abs()* đều phải được viết trong cùng một file nguồn để biên dịch.

4.4 HÀM NGUYÊN MẪU (FUNCTION PROTOTYPE)

Việc khai báo kiểu của hàm (kết quả thu được qua lệnh *return*) làm cho chương trình trở thành chặt chẽ hơn. Tuy vậy nó cũng là nguyên nhân nảy sinh một số vấn đề nếu người lập trình không chú ý đến. Chẳng hạn trong một chương trình nào đó có một hàm được gọi trước vị trí hàm được định nghĩa, nếu hàm này có kiểu khác kiểu *int* thì khi biên dịch, trình biên dịch sẽ thông báo lỗi. Nguyên nhân là do khi biên dịch, C sẽ thực hiện quá trình biên dịch theo thứ tự từ trên xuống dưới theo từng lệnh. Khi gặp hàm bị gọi, bởi vì trình biên dịch không có thông tin về kiểu của hàm, do đó trình biên dịch sẽ xem như hàm bị gọi được định nghĩa theo kiểu *int*. Điều này sẽ làm phát sinh sự xung đột về kiểu khi biên dịch xuống dưới gặp định nghĩa của hàm với kiểu khác *int*.

Ví dụ 4.8

```
#include <stdio.h>
void main()
    {
    float x=12.3;
    int n=3;
    printf ("Lũy thừa bậc %d của %f là %f\n",n,x,power(x,n));
    }
```

Trong chương trình này ta có thể sử dụng hàm *power* đã được viết ở trên để tính lũy thừa của một số. Hàm này khi định nghĩa được khai báo theo kiểu *float* nhưng trước khi sử dụng hàm không được khai báo lại. Sau khi biên dịch, C sẽ thông báo lỗi "Type mismatch in redeclaration of luy thua" do kiểu của hàm trước và sau khi bị gọi không trùng nhau.

Để tránh các lỗi được nêu ra ở trên thì tất cả các hàm phải được định nghĩa trước khi chúng được gọi nếu kiểu của hàm

khác *int*. Nhằm tạo ra khả năng cho phép một hàm có thể được định nghĩa ở vị trí bất kỳ trong chương trình nguồn, trong ngôn ngữ C ta cần phải khai báo kiểu của hàm trước khi nó được sử dụng. Việc khai báo đó được thực hiện qua cách viết sau:

kiểu_hàm tên_hàm();

Trong kiểu khai báo này, danh sách các tham số không được viết sau tên hàm bởi vì việc khai báo chỉ nhằm mục đích báo trước cho trình biên dịch biết kiểu của hàm để tránh sự xung đột về kiểu khi biên dịch. Khai báo có thể được viết ở giữa hai hàm, đầu chương trình hoặc trong chương trình gọi trước khi hàm bị gọi được sử dụng. Khi được viết trong chương trình gọi, khai báo có thể nằm ở đầu trong phần khai báo chung hoặc ở đầu của một khối lệnh nào đó.

Ví dụ 4.9

Ví dụ này minh họa cho việc khai báo kiểu của hàm trước khi nó được sử dụng. Khi biên dịch chương trình này, C sẽ không thông báo lỗi.

```
#include <stdio.h>
main()
    {
    float x=12.3;
    int n=3;
    double power();  /* Khai báo hàm power trước khi sử
                        dụng */
    printf ("Lũy thừa bậc %d của %f là % f\n",n,x,power(x,n));
    }
double power(double base, int exp)
    {
```

```
    . . . . .
    . . . . .
    }
```

Ngoài cách khai báo trên, ta còn có thể thực hiện việc này bằng một khai báo mở rộng. Khi khai báo hàm mở rộng ngoài việc khai báo kiểu của kết quả mà hàm trả về ta còn phải khai báo kiểu và số lượng các đối số được sử dụng trong hàm. Việc khai báo này được gọi là khai báo hàm nguyên mẫu (function prototype). Bằng cách khai báo này C tạo ra khả năng chặt chẽ hơn nữa khi sử dụng hàm để kiểm soát các loại dữ liệu được truyền cho hàm. Khi sử dụng hàm, C chỉ cho phép làm việc với các kiểu dữ liệu của các đối số như đã khai báo trước đó.

Dạng khai báo của một hàm nguyên mẫu như sau:

Kiểu_hàm tên_hàm(danh sách các đối số);

Trong khi khai báo các đối số trong danh sách ta chỉ cần đưa ra kiểu của dữ liệu mà không cần đưa tên của chúng. Các kiểu này được viết cách nhau bởi dấu phẩy và được sắp xếp theo thứ tự chúng được viết khi định nghĩa.

Ví dụ 4.10

```
#include <stdio.h>
double power(double,int); /* Khai báo hàm lũy thừa trước
                             khi sử dụng */
void main()
    {
    float x=12.3;
    int n=3;
    printf ("Lũy thừa bậc %d của %f là %f\n",n,x,power(x,n));
    }
```

Dưới hàm main là định nghĩa của hàm power đã được viết ở trên.

Thậm chí ta cũng có thể không cần phải khai báo tất cả các đối số, trong trường hợp này ở cuối danh sách các đối số phải đặt ba dấu chấm. Chẳng hạn *printf(char format[],...)*.

Ngoài khả năng kiểm tra kiểu của các đối số và số lượng các đối số, việc sử dụng hàm nguyên mẫu còn đưa ra khả năng chuyển kiểu của đối số cho phù hợp với kiểu của chúng khi định nghĩa hàm. Ví dụ trong chương trình:

Ví dụ 4.11

```
#include <stdio.h>
double power(double,int);
void main()
    {
    float x = 12.3;
    int n=3;
    printf ("Lũy thừa bậc %d của %f là %f\n",n,x,power(x,n));
    }
double power(double base, int n)
    {
    . . . . .
    . . . . .
    }
```

Mặc dù khi gọi hàm *power*, *x* có kiểu khác với kiểu khi khai báo, song trình biên dịch cũng không thông báo lỗi. Nguyên nhân là do trình biên dịch đã tự động kiểm tra kiểu của đối số và tự động thực hiện sự chuyển kiểu bắt buộc khi kiểu của đối số lúc sử dụng khác với kiểu của đối số khi định nghĩa hàm.

4.5 CÁC CẤP LƯU TRỮ CỦA BIẾN

Trong C mỗi biến được đặc trưng bởi 3 thuộc tính sau:

- Kiểu của biến.
- Phạm vi làm việc của biến.
- Thời gian tồn tại của biến.

Kiểu của biến cho phép xác định giá trị mà biến có thể nhận được và thông qua đó trình biên dịch sẽ cấp phát số lượng byte cần thiết cho biến. Kiểu của biến được khai báo qua các từ khóa. *Phạm vi làm việc* của biến phụ thuộc vào vị trí của biến khi khai báo. Phạm vi hoạt động của biến có thể là một khối lệnh trong một chương trình, một hàm hoặc một tập tịn nguồn. *Thời gian tồn tại* của biến là khoảng thời gian làm việc của chương trình cho đến khi giá trị cuối cùng của biến được lưu giữ. Cần chú ý rằng thời gian tồn tại và phạm vi làm việc của biến có thể không bao trùm lên nhau, chẳng hạn một biến có thể vẫn tồn tại sau khi ra khỏi phạm vi hoạt động của biến (trong trường hợp biến được khai báo theo kiểu static như sẽ xem xét ở dưới). Như vậy thời gian tồn tại của biến có thể bằng thời gian thực hiện của chương trình hoặc bằng thời gian thực hiện chương trình khi đang ở phạm vi của biến.

Phụ thuộc vào thời gian tồn tại và phạm vi làm việc, các biến trong C được phân bố ở từng vị trí khác nhau trong bộ nhớ, hay nói cách khác là *các cấp lưu trữ* của chúng là khác nhau. Ngôn ngữ C cung cấp 4 biến từ dùng để chỉ định các cấp lưu trữ (Class Storage). Các biến từ này là *auto, extern, static và register*. Các biến từ này báo cho trình biên dịch được biết về cách lưu trữ của biến khi được khai báo. Một biến có thể được khai báo cấp lưu trữ của nó bằng cách sau:

Biến_từ kiểu_biến tên_biến;

4.5.1 Biến từ auto

Biến từ này thường được dùng để khai báo các biến cục bộ và các tham số hình thức của hàm. Các biến thuộc loại này được gọi là các biến tự động *(auto)*. Tuy nhiên, trong thực tế ta rất ít khi gặp các trường hợp có sử dụng biến từ auto bởi vì các biến cục bộ và các tham số hình thức của hàm mặc nhiên đã được xem là thuộc kiểu *auto*. Các biến được khai báo theo kiểu *auto* sẽ được cung cấp bộ nhớ cần thiết trong ngăn xếp trong quá trình thực hiện hàm mà chúng được khai báo. Các biến thuộc kiểu này đều được nhận giá trị bất kỳ có sẵn trong ngăn xếp cho đến khi chúng được gán cho một giá trị nào đó. Đây chính là nguyên nhân đòi hỏi ta phải khởi tạo biến trước khi chúng được sử dụng. Thời gian tồn tại và phạm vi của biến là hoàn toàn trùng nhau và giá trị làm việc của biến chỉ được lưu giữ trong quá trình thực hiện hàm hoặc khối mà chúng được khai báo. Chính vì vậy khi ta quay lại làm việc với hàm thì các biến kiểu này lại được nhận một giá trị bất kỳ nào đó.

Ví dụ 4.12

```
Fun(int x, int y) /* Các tham số hình thức thuộc kiểu auto */
    {
    float z; /* Các biến cục bộ luôn luôn thuộc kiểu auto */
        /* Phạm vi hoạt động của z là thân của hàm */
    ...
        {
        int p,q; /* Phạm vi hoạt động của p,q là khối lệnh có
                    chứa chúng */
        ...
        }
}
```

4.5.2 Biến từ extern

Thông thường loại biến từ này thường được dùng cho các biến ngoài và các biến này mặc nhiên được nhận cấp lưu trữ *extern* mà không cần phải khai báo *extern* trước tên biến. Các biến được khai báo theo kiểu này được lưu trữ trong vùng dữ liệu của chương trình và giá trị của chúng sẽ tồn tại trong suốt quá trình thực hiện chương trình.

Ví dụ 4.13

```
int i,j; /* i,j thuộc cấp extern */
void main()
    {
    ...
    }
float x,y; /* Các biến x,y thuộc loại extern */
fun()
{
...
}
...
```

Chú ý rằng các biến *x,y* và *i,j* có các phạm vi làm việc khác nhau. Cũng cần phải phân biệt sự khác nhau giữa biến từ *extern* và từ khóa *extern* được dùng trong ngôn ngữ C. Từ khóa *extern* được dùng khi một biến ngoài đã được định nghĩa ở một file nguồn khác. Khi gặp khai báo kiểu này trình biên dịch sẽ không cấp phát bộ nhớ cho biến nữa. Vấn đề này sẽ được trình bày cụ thể ở phần 4.6.

4.5.3 Biến từ static

Loại biến từ này phải được viết rõ ràng khi khai báo biến.

Các biến được khai báo theo kiểu *static* sẽ được cấp phát bộ nhớ trong vùng dữ liệu của chương trình và do vậy thời gian tồn tại của biến cũng chính là thời gian thực hiện của chương trình. Các biến cục bộ hoặc các biến ngoài đều có thể được khai báo theo kiểu *static* nhưng ý nghĩa trong hai trường hợp này là hoàn toàn khác nhau.

Khi một biến cục bộ được khai báo theo loại *static*, thời gian tồn tại của biến sẽ bị thay đổi và bằng thời gian thực hiện của chương trình. Trong ví dụ:

```
fun()
    {
    static int i;
    float z;
    ...
    i=i+2;
    }
```

Nếu không chỉ ra điều gì khác thì trước khi chương trình bắt đầu thực hiện tất cả các biến thuộc loại *static* sẽ có giá trị bằng 0 (điều này được thực hiện trong quá trình biên dịch). Khi hàm *fun* được thực hiện lần đầu thì sau khi quyền điều khiển được chuyển ra ngoài hàm, giá trị của *i* sẽ là 2. Giá trị này được lưu trữ trong suốt quá trình thực hiện chương trình cho đến khi quyền điều khiển lại được trao lại cho hàm *fun*. Khi hàm *fun* được gọi lại lần thứ 2 thì *i* vẫn có giá trị là 2 và sau khi ra khỏi hàm, giá trị này sẽ là 4.

Nếu một biến ngoài được khai báo theo loại *static* thì thời gian tồn tại của biến sẽ không được tăng lên. Việc khai báo này có ý nghĩa hoàn toàn khác. Trong trường hợp biên dịch riêng biệt (phần 4.6), phạm vi hoạt động của các biến được khai báo theo loại *static* sẽ chỉ nằm trong các tập tin nguồn mà biến được định nghĩa. Nếu biến ngoài không được khai báo

ở đầu tập tin nguồn thì các hàm được viết trước khi định nghĩa sẽ không có quyền truy nhập đến biến. Chẳng hạn trong ví dụ:

```
void main()
    {
    ...
    }
static char c;
fun()
    {
    ...
    }
```

Hàm *main* sẽ không được quyền truy nhập đến biến c.

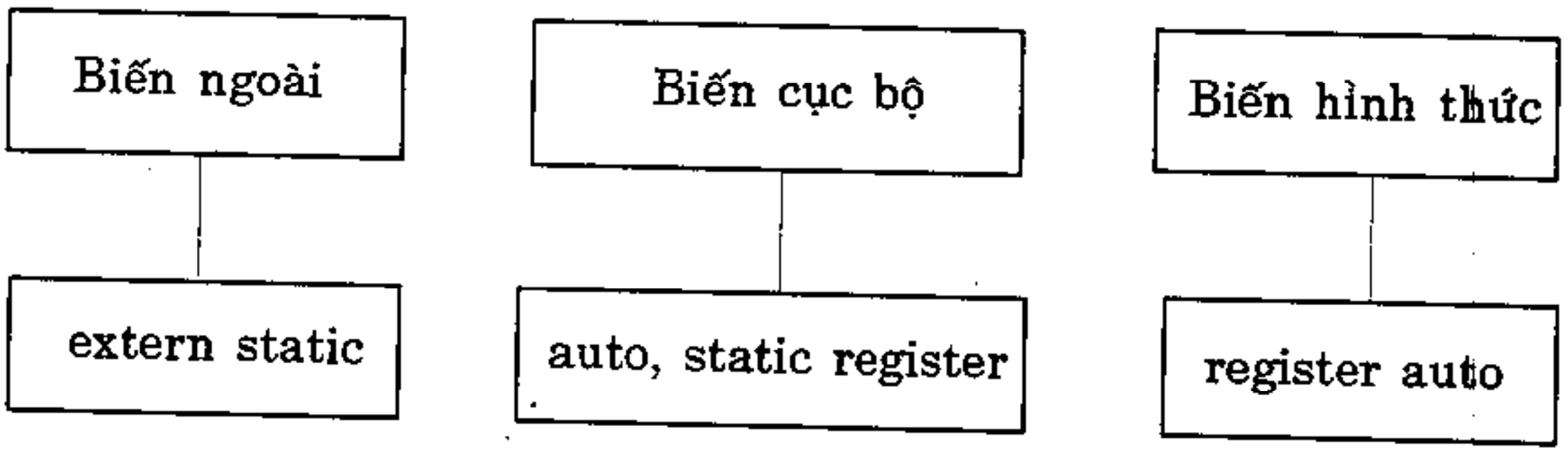
4.5.4 Biến từ register

Một loại biến từ quan trọng khác trong C được gọi là *register* và chỉ được áp dụng cho những biến kiểu *int* và *char*. Một biến thuộc loại *register* phải được khai báo rõ ràng và phải là các biến cục bộ hoặc là các biến hình thức. Trong quá trình thực hiện hàm, các biến *register* này được chứa trong các thanh ghi của bộ vi xử lý thay vì chứa trong bộ nhớ của máy. Những thao tác trên các biến thuộc kiểu *register* bao giờ cũng nhanh hơn so với các biến được lưu trữ trong ô ký ức và vì vậy nó sẽ làm cho thời gian thực hiện chương trình giảm xuống một cách đáng kể. Thông thường để thực hiện mục đích này người lập trình thường khai báo các biến hay được sử dụng trong chương trình, các biến điều khiển vòng lặp v.v. là thuộc loại *register*. Mặc dù trong C không đề cập đến số biến *register* có thể được khai báo là bao nhiêu nhưng bộ vi xử lý chỉ sử dụng có 2 thanh ghi để chứa các biến thuộc loại này, đó là các thanh ghi *DI* và *SI*. Chính vì nguyên nhân này nên người lập trình phải

quyết định nên chọn những biến nào sẽ là biến thuộc loại *register* để chương trình có thể được xây dựng ở mức độ tối ưu, bởi vì khi không còn thanh ghi trống để lưu trữ biến, biến sẽ được xem như là biến kiểu *auto* mặc dù trước đó biến được khai báo theo loại *register*. Việc sử dụng biến *register* có thể được minh họa qua ví dụ sau:

```
fun(register int x)
    {
    register k;
    ...
    }
```

Hình 4.3 mô tả quan hệ giữa các biến từ và phạm vi hoạt động của biến.



Hình 4.3

Việc phân chia các biến thành từng loại biến từ khác nhau sẽ cho phép ta điều khiển việc sử dụng bộ nhớ ở mức độ hợp lý nhất và trong nhiều trường hợp sẽ làm tăng tốc độ thực hiện của chương trình.

4.5.5. Biến từ const

Khi một biến được khai báo với biến từ *const* thì biến phải được khởi tạo giá trị cùng với lúc định nghĩa biến. Giá trị này sẽ không thể thay đổi trong quá trình thực hiện chương trình

bằng bất cứ cách nào. Chẳng hạn ta có thể gán giá trị cho biến no_change như sau:

```
const int no_change = 2;
```

Việc sử dụng biến từ *const* cho phép ta xác định hằng số và thông báo kiểu của biến để trình biên dịch cung cấp bộ nhớ dành riêng cho biến. Các biến thuộc loại *const* có ứng dụng rất quan trọng, nhất là trong trường hợp ta cần truyền biến qua hàm mà không muốn thay đổi giá trị của các biến đó trong thân của hàm.

4.5.6 Biến từ volatile

Biến từ volatile được sử dụng để báo cho C biết rằng giá trị của một biến có thể được thay đổi bằng một cách nào đó mà không được mô tả rõ ràng trong chương trình (chẳng hạn có thể bị thay đổi bởi các chương trình của Dos hay Bios...)

4.6 BIÊN DỊCH THEO MODULE

Trên thực tế khi thiết kế các chương trình dài và phức tạp, người ta thường chia các chương trình đó thành nhiều phần nhỏ gọi là các Module, cách lập trình này có rất nhiều ưu điểm:

- Dễ dàng kiểm soát và phát hiện các lỗi phát sinh ở các module khác nhau trong khi biên dịch.
- Tạo ra khả năng lập trình độc lập trên các phần khác nhau của chương trình.

Hầu hết các trình biên dịch của C đều cho phép thực hiện việc biên dịch theo từng Module, tuy vậy các lệnh để cho trình biên dịch thực hiện quá trình này thì lại phụ thuộc vào các loại máy với cấu trúc khác nhau. Mặt khác cơ cấu truyền thông tin giữa các Module với nhau lại được thực hiện qua các phương tiện lập trình và đã được chuẩn hóa theo trình biên dịch C chuẩn. Mỗi một Module của chương trình là một File nguồn độc

lập và có thể được soạn thảo, sửa và biên dịch riêng biệt. Trong quá trình liên kết (Link), tất cả các file đối tượng (Obj) đã được biên dịch sẽ được liên kết vào một file khả thi duy nhất (file có phần mở rộng là Exe). Các Module có thể bao gồm rất nhiều hàm khác nhau, tuy vậy chỉ có một và chỉ một Module có chứa hàm với tên *main*. Các hàm trong các Module khác nhau có thể trao đổi thông tin bằng hai cách sau:

- Khai báo các biến tổng thể chung cho các Module khác nhau.
- Sử dụng các hàm được viết trong các Module.

Theo cách thứ nhất, một biến tổng thể có thể được dùng chung cho các Module khác nếu biến được khai báo lại trong các Module đó bằng cách sau:

```
extern kiểu_biến_tên_biến;
```

Từ khóa *extern* bắt buộc phải được viết trước *kiểu của biến*.

Extern báo cho trình biên dịch biết là biến đã được định nghĩa ở một Module khác và trình biên dịch đã cấp phát cho nó một số ô ký ức cần thiết và do đó trong quá trình khai báo lại, trình biên dịch sẽ không cần cấp phát lại bộ nhớ cho biến nữa. Qua việc khai báo lại này, trong quá trình liên kết các Module các biến có cùng tên sẽ cùng sử dụng một địa chỉ nào đó trong bộ nhớ.

Khi một biến được dùng chung cho nhiều Module khác nhau, giá trị của các biến có thể được thay đổi trong các File chứa các Module đó. Nếu một biến được khai báo như một biến tổng thể (extern) phạm vi hoạt động của biến sẽ bao gồm tất cả các hàm được viết trong File đó. Biến cũng có thể được khai báo là tổng thể hoặc cục bộ trong hàm hoặc một khối lệnh, khi đó phạm vi hoạt động của biến chỉ là hàm hoặc khối lệnh đó.

Ta cần phải lưu ý trường hợp khi một biến kiểu mảng được khai báo theo kiểu tổng thể. Nếu biến là mảng một chiều có

thể bỏ qua kích thước của mảng khi khai báo. Chẳng hạn:

extern int array[]:

Nếu biến là mảng nhiều chiều, có thể bỏ qua kích thước đầu tiên của mảng:

extern int array[] [15];

Một điểm nữa cần lưu ý là khi một biến được dùng chung cho nhiều module khác nhau thì trước khi được sử dụng biến phải được khởi tạo trong File mà biến được định nghĩa.

Theo cách thứ hai, thông tin giữa các module có thể được trao đổi thông qua việc sử dụng các hàm. Thông thường tên của hàm được ngầm định là kiểu *int* nếu kiểu của hàm là *int*. Trong trường hợp này hàm có thể được sử dụng tự do trong các module khác nhau. Nếu kiểu của hàm không được khai báo theo kiểu *int* thì ta cần phải khai báo lại hàm trong module có sử dụng hàm việc khai báo này được tiến hành giống như trường hợp khai báo biến tổng thể. Tuy vậy trong nhiều trình biên dịch từ khóa *extern* có thể được bỏ qua khi khai báo lại hàm.

Nếu một biến tổng thể được định nghĩa theo kiểu *static* thì biến đó không thể được khai báo lại trong các module khác. Như vậy khi đó biến chỉ có thể được truy nhập trong File mà biến được định nghĩa. Bằng cách này ta có thể đảm bảo rằng phạm vi hoạt động của biến chỉ thuộc một module nhất định. Điều này cũng đúng cho các hàm được định nghĩa theo kiểu *static*.

Ví dụ 4.14

Thiết lập chương trình để xử lý các dữ liệu thực nghiệm x_i, $i = 1,2 \ldots j$, $x_i >= 0$, *trong đó j là số lần thực nghiệm theo phương pháp sau:*

- Xác định giá trị lớn nhất x^* và đối với mỗi giá trị thực

nghiệm hãy tính sai số theo công thức $(x^* - x_i)/x^*$.

• Các sai số trong các lần thực nghiệm được xếp theo thứ tự tăng dần và với một sai số cho phép lớn nhất ESP cho trước hãy xác định số lượng các sai số có giá trị nhỏ hơn ESP.

Chương trình được thiết kế theo kiểu module và được chia thành 4 File nguồn: *source1.c, source2.c, const.h* và *dimen.h*. Hai File source1.c, source2.c được biên dịch riêng biệt. File source1.c bao gồm hàm chính *main*, hàm *input* để nhập các dữ liệu, hàm *maxi* dùng để xác định giá trị lớn nhất của dữ liệu và hàm *ralat* tính các sai số. Các hàm *input, maxi, relat* chỉ được dùng trong File này do đó chúng được định nghĩa theo kiểu *static*. File source2.c bao gồm hàm *sort* dùng để sắp xếp các sai số theo thứ tự tăng dần và hàm *ret* - xác định các sai số có giá trị nhỏ hơn *ESP*. File cons.h và dimen.h là hai File tiêu đề và có chứa việc khai báo các thành phần của chương trình: số lượng lớn nhất của *xi* (số các thực nghiệm), giá trị của ESP (trong file cons.h). Các biến khác: *s[]* - dùng để chứa các sai số, *m* - giá trị lớn nhất của dữ liệu thực nghiệm *x**, *k* - số lượng các sai số có giá trị nhỏ hơn giá trị cho trước *ESP* được định nghĩa trong File *dimen.h*.

File const.h

```
#define L 10 /* Số lượng thực nghiệm lớn nhất được cho
                phép */
#define EPS 0.3 /* Giá trị sai số cho phép */
```

File dimen.h

```
#int k; /* Số lượng sai số dưới mức cho phép */
float m,s[L]; /* m - giá trị lớn nhất trong số các dữ liệu
                 thực nghiệm */
/* s[L] - mảng chứa các sai số */
```

File source1.1

```
#include"const.h"
#include"dimen.h"
main()
    {
    /* Khai báo các hàm ngoài */
    int sort();
    int ret();
    float maxi();
    float a[L]; /* mảng của các tham số đầu vào */
    int j; /* Số lượng các thực nghiệm */
    printf("Hãy đưa số lượng các thực nghiệm j\n");
    scanf("%d", &j);
    input(a,j);
    m = maxi(a,j);
    relat(a,j);
    sort(j);
    printf("Số lượng thực nghiệm với sai số nhỏ hơn sai số
        cho phép là k =%d\n",k);
    printf("%d");
    }
/* Hàm nhập dữ liệu từ thực nghiệm */
static int input(int t,float c[j])
    {
    int i;
    for(i=0;i<t;i++)
        {
        printf("Hãy đưa số thứ %d\n",i+1);
        scanf("%d", &c[i]);
```

```
        }
    }
/* Hàm maxi xác định phần tử lớn nhất trong mảng */
static float maxi(int n,float c[])
    {
int i;
float z;
z = c[0];
for(i=1;i<n;i++)
    if(z<c[])
        z=c[i];
        return z:
}
/* Hàm relat tính sai số tương đối */
static int relat(int n,float c[])
    {
    int i;
    for(i=0;i<n;i++)
        s[j] = (m-c[i])/m
    }
```

File source2.c

```
#include "const.h"
   extern float s[];
   /* Hàm sort sắp xếp các sai số tương đối theo thứ tự tăng dần */
   int sort(int n)
       {
       extern int k;
```

```
int ret();
int i,j;
float r;
for(i=1;i<n;i++)
    for(j=n-1;j>=1;--j)
        if(s[j-1]>s[j])
            {
            r = s[j-1];
            s[j-1] = s[j];
            s[j] = r;
            k=ret(n);
            }
/* Hàm tính số lượng sai số nhỏ hơn mức độ cho phép */
static int ret(int k)
    {
    int i;
    for(i=0;i<k;i++)
        if(s[i] > EPS)
            return i;
    return k;
    }
```

Chương V

BỘ TIỀN XỬ LÝ

Ngôn ngữ C có chứa một khả năng ít gặp đối với các ngôn ngữ bậc cao - khả năng điều khiển quá trình biên dịch. Khả năng này là một trong các đặc tính đưa ngôn ngữ lại gần với ngôn ngữ lập trình bậc thấp như Assembler. Quá trình biên dịch trong ngôn ngữ C được điều khiển bởi 1 chương trình riêng biệt gọi là bộ tiền xử lý. Bộ tiền xử lý đã đưa ra nhiều khả năng mới cho người lập trình. Cụ thể các khả năng đó bao gồm:

- Sử dụng Macro;
- Nối kết các file nguồn;
- Biên dịch có điều kiện.

Việc sử dụng bộ tiền xử lý đưa lại nhiều lợi thế trong lúc lập trình. Các chương trình được thiết kế sẽ trở nên sáng sủa và dễ hiểu hơn và nhất là tạo nhiều thuận lợi khi cần phải đánh giá một thuật toán hoặc lời giải của một bài toán thông qua chương trình. Ngoài các đặc điểm trên, chương trình có sử dụng bộ tiền xử lý trong nhiều điều kiện khác nhau có thể dễ dàng thay đổi cho phù hợp với các yêu cầu mới.

5.1 NGUYÊN TẮC LÀM VIỆC CỦA BỘ TIỀN XỬ LÝ

CHỈ THỊ CỦA BỘ TIỀN XỬ LÝ

Như đã nêu ra ở trên, bộ tiền xử lý có vai trò thực hiện các nhiệm vụ sau:

- Phân tích các chương trình nguồn như một file được viết bằng mã ASCII (file văn bản).
- Thay đổi file nguồn bằng cách thêm hay bớt một đoạn

nào đó vào file.

Khi phân tích file nguồn, bộ tiền xử lý sẽ thay thế tất cả các chú giải bằng dấu cách. Các dấu cách này không tham gia vào quá trình biên dịch và do đó không làm tăng độ dài của file thu được sau khi biên dịch. Đây là lý do cho phép người lập trình có thể sử dụng những chú giải với độ dài tùy ý. Mặc dù vậy người lập trình cũng không nên lạm dụng điều này để đưa ra những chú giải rườm rà, khó hiểu.

Khi sử dụng bộ tiền xử lý, mọi thay đổi trong khi file nguồn được xác định bởi các lệnh đặc biệt gọi là chỉ thị của bộ tiền xử lý. Khi sử dụng, mỗi chỉ thị phải được viết trên một hàng riêng biệt trong file nguồn và tuân theo cú pháp sau:

 # Từ_khóa Kiểu_thay_thế

Trong cú pháp này ký tự # bắt buộc phải đứng ở đầu cột của hàng. *Từ_khóa* là các từ đã được qui định và phải viết bằng chữ thường, giữa *#Từ_khóa* và *Kiểu_thay_thế* phải tồn tại ít nhất là một dấu cách.

Kiểu_thay_thế sẽ xác định các thay đổi trong file nguồn và có nhiều cách viết khác nhau tùy theo từng loại chỉ thị.

Bộ tiền xử lý của C có sử dụng 3 loại chỉ thị. Các chỉ thị này nhằm mục đích:

- Định nghĩa Macro;
- Nối kết các file
- Biên dịch có điều kiện.

Bảng 5.1 có đưa ra một số chỉ thị hay gặp trong chương trình C. Trong các phần sau sẽ trình bày chi tiết một số chỉ thị. Khi sử dụng các chỉ thị này cần lưu ý rằng chúng có thể được viết ở vị trí bất kỳ trong chương trình nhưng phải ở trên một hàng riêng biệt. Cần chú ý là các chỉ thị có bộ tiền xử lý có cú pháp riêng của mình và chúng không phải là các lệnh của chương trình C. Các chỉ thị của bộ tiền xử lý sẽ xem file

Bảng 5.1

Chỉ thị	Tác dụng
#define	Định nghĩa Macro
#undef	Hủy Macro đã định nghĩa
#include	Nối kết file
#ifdef	Cấu trúc "if-else-end" dùng
#indef	cho biên dịch có điều kiện phụ
#if	thuộc vào định nghĩa Macro
#else	(#ifdef) hoặc giá trị của biểu
#endif	thức hằng (# if)
#line	Gán số cho dòng chương trình

nguồn như một file text viết bằng mã ASCII mà không phân biệt đó là chương trình viết bằng C hay không. Về nguyên tắc một bộ tiền xử lý có thể được xem như là một file độc lập và có thể tác động lên các file nguồn viết bằng một ngôn ngữ bất kỳ.

5.2 ĐỊNH NGHĨA MACRO

Macro đươc định nghĩa thông qua cú pháp sau:

#define Tên_Macro Chuỗi

Sau khi gặp chỉ thị *define*, bộ tiền xử lý sẽ thay thế tất cả các *Tên macro* được gặp sau đó bằng *Chuỗi* được viết trong chỉ thị. Các *Chuỗi* này sẽ được sử dụng trong quá trình biên dịch do đó chúng phải bao gồm các ký tự được phép trong ngôn ngữ C và có thể là hằng số hay biểu thức. Do bộ tiền xử lý sẽ chèn các đoạn mà vào chương trình ở các vị trí khác nhau trước khi biên dịch bạn cần chú ý rằng việc sử dụng Macro sẽ cho phép chương trình của bạn trở thành ngắn gọn trong cách viết nhưng không làm giảm độ lớn của file khi biên dịch. Một lợi thế nữa của Macro là có thể tránh được việc gọi hàm số, một

công việc mà việc thực hiện có nhiều lúc đòi hỏi phải truyền các tham số vào ngăn xếp và làm giảm tốc độ thực hiện của toàn bộ chương trình (xem chương IV).

Ví dụ 5.1.

```
#define TWO 2
#define MSG "Message"
#define FOUR TWO * TWO
#define PX printf ("x=%d\n", x)
#define FMT "x=%d\n"
void main()
    {
    int x=TWO;
    PX;
    x = FOUR;
    printf(FMT,x);
    printf("%s\n",MSG);
    printf("TWO:MSG\n");
    }
```

Trước khi biên dịch, bộ tiền xử lý sẽ phân tích file nguồn và thay thế tên của Macro có sử dụng trong chương trình bằng các chuỗi được định nghĩa trong các chỉ thị. Chương trình *main* trước khi biên dịch sẽ có dạng sau:

```
void main()
    {
    int x=2; /* TWO được thay thế bởi 2 */
    printf("x=%d\n", x);/*thay thế PX*/
    x = 4; /* Thay thế Four = TWO * TWO = 4 */
    printf("=%d\n",x); /* Thay thế FMT bằng "x=%d\n" */
```

```
printf("%s\n","Message"); /* Thay thế MSG bằng Message */
printf("TWO:MSG\n");
}
```

Trong chương trình trên ta chỉ chú ý đến lệnh cuối cùng printf("TWO:MSG\n"). Trong lệnh này *TWO* và *MSG* được sử dụng như các chuỗi bình thường do đó sẽ không được xử lý như là một Macro.

Chỉ thị *define* có thể có cấu trúc lồng nhau. Chẳng hạn cấu trúc sau là hoàn toàn cho phép trong một chương trình viết bằng C.

```
#define Size 1000
...
#define Dimens (Size*5+1)
```

Trường hợp trong một chương trình nếu một Macro được định nghĩa nhiều hơn một lần thì chỉ thị đầu tiên sẽ có tác dụng cho đến khi gặp chỉ thị đó ở vị trí khác tiếp theo trong chương trình.

Ngoài ra, tác dụng của một Macro có thể bị loại bỏ từ một vị trí nào đó trong chương trình nếu tại vị trí đó ta sử dụng chỉ thị *#undef*. Chỉ thị có dạng sau:

#undef Tên_Macro

Tác dụng của chỉ thị *#undef* có thể được minh họa trong ví dụ sau:

```
#define GLOB 10
...
#define GLOB 100
...
#undef
```

Trong ví dụ này chỉ thị *#define GLOB 10* sẽ có tác dụng

cho đến khi gặp chỉ thị tiếp theo *#define GLOB 100*. Điều đó có nghĩa là sau khi gặp chỉ thị thứ 2, *GLOB* sẽ sử dụng giá trị *100* thay thế cho giá trị *10* được định nghĩa trước đó. Chỉ thị *#undef* sẽ loại bỏ tác dụng của chỉ thị *#define GLOB 100* và từ vị trí này trong chương trình chỉ thị *#define GLOB 10* lại có tác dụng trở lại.

Chỉ thị *#undef* được dùng tương đối ít trên thực tế và khi sử dụng cần phải thận trọng. Nếu để ý ta có thể thấy rằng mặc dù Macro đã được hủy bỏ tác dụng của nó nhưng trong chương trình tên của macro vẫn có thể tồn tại sau đó. Chẳng hạn đoạn chương trình sau có thể gây ra lỗi trong khi biên dịch.

```
#undef GLOB
x = GLOB*2;
```

Ta thấy ngay rằng nguyên nhân phát sinh lỗi khi biên dịch là Macro *GLOB* vẫn được sử dụng sau khi đã hủy bỏ tác dụng của nó.

5.3 MACRO CÓ SỬ DỤNG THAM SỐ

Ngoài các Macro thay thế đơn giản, bộ tiền xử lý của C còn cho phép sử dụng các Macro có dùng tham số. Các Macro này được sử dụng thông qua cú pháp sau:

#define Tên_Macro(danh sách tham số) chuỗi

Trong cú pháp trên, danh sách các đối số phải được viết trong hai dấu ngoặc đơn, giữa *Tên_Macro* và dấu mở ngoặc không được tồn tại dấu cách. Tên của Macro chỉ có tác dụng cho định nghĩa có chứa Macro đó, điều đó cho phép tên của Macro có thể trùng với các tên khác trong chương trình mà không xảy ra bất cứ một sự rắc rối, nhầm lẫn nào. Danh sách các tham số được viết trong định nghĩa được gọi là các tham số hình thức còn các tham số của Macro được sử dụng trong

chương trình gọi là các tham số thực. Khi bộ tiền xử lý gặp Macro có dùng tham số trong chương trình, bộ tiền xử lý sẽ thực hiện các công việc sau:

• Ở *Chuỗi* được viết trong chỉ thị *define* các tham số hình thức sẽ được thay thế bởi các tham số thực.

• Tên của *Macro* được viết trong chương trình sẽ được thay thế bởi chuỗi vừa được xử lý trước đó.

Ví dụ 5.2

Sử dụng Macro dùng tham số để tính diện tích hình tròn. Đối số được sử dụng cho bán kính vòng tròn là s:

```
#include<stdio.h>
#define PI 3.141492
#define CIRCLE(s)(PI*(s))
main()
    {
    float x,z;
    printf("Hãy đưa bán kính vòng tròn\n");
    scanf("%f",&x);
z = CIRCLE(x);
    printf("Diện tích của hình tròn là %f\n" z);
    }
```

Trong ví dụ trên khi định nghĩa Macro *CIRCLE* có sử dụng cấu trúc định nghĩa lồng (sử dụng tên của Macro đã được định nghĩa trước đó). Việc sử dụng các dấu ngoặc đơn trong *Chuỗi* (trong ví dụ trên đó là (PI*(S)*(S)) khi định nghĩa Macro là hoàn toàn cần thiết, chúng cho phép người lập trình xác định mức độ ưu tiên khi thực hiện các phép tính được viết trong *Chuỗi*. Cách viết này là bắt buộc trong các trường hợp khi Macro được sử dụng như là một biểu thức và khi đối số của

Macro lại là một *Chuỗi* được viết như một biểu thức. Chẳng hạn trong ví dụ trên nếu *CIRCLE* được định nghĩa bằng cách:

```
#define CIRCLE(s) PI*s*s
```

Thì khi gặp Macro với tham số là x + 1 bộ tiền xử lý sẽ thực hiện phép thay thế: z = CIRCLE (x + 1) bởi z = PI*x + 1* x + 1. Rõ ràng ta nhận thấy rằng biểu thức sau khi được thay thế không thể dùng để tính diện tích của hình tròn với bán kính $x + 1$. Việc sử dụng dấu ngoặc đơn sẽ cho phép ta thiết lập cách viết đúng cho một biểu thức:

```
z = PI* (x + 1)* (x + 1)
```

Một ví dụ khác có thể dùng để minh họa cho việc cần thiết phải sử dụng các dấu ngoặc khi định nghĩa Macro:

```
#define CONST 10

#define MACRO (s) CONST + s
```

Nếu bạn định sử dụng định nghĩa Macro trên để tính kết quả của biểu thức *z = 10/(CONST + x)* thì thông thường bạn sẽ đưa đến một kết quả sai nếu sử dụng Macro qua dạng sau:

```
z = 10/MACRO (x).
```

Nguyên nhân đưa đến kết quả sai là do khi thay thế tham số thực vào Macro ta sẽ thu được biểu thức:

```
z = 10/CONST + x
```

Biểu thức này không phải là biểu thức mà ta mong đợi. Để thu được biểu thức đúng ta chỉ cần sử dụng thêm dấu ngoặc đơn trong khi định nghĩa Macro:

```
#define MACRO (s) (CONST+s)
```

Trong nhiều trường hợp Macro đã được tham số hóa có thể được dùng để thay thế cho hàm số. Chẳng hạn để xác định giá trị cực đại của hai số ta có thể định nghĩa một Macro như là một hàm số.

```
#define max (a,b)((a)>(b) ? (a) : (b))
```

Với Macro này, việc tìm số lớn nhất trong hai số *x+y,z* trong chương trình có thể được sử dụng qua lệnh sau:

```
m = max(x+y,z);
```

Việc sử dụng Macro với tham số đòi hỏi chúng ta phải rất thận trọng. Với một sai sót nhỏ khi định nghĩa Macro có thể dẫn đến các hiệu ứng phụ và đưa đến kết quả sai. Chẳng hạn trong hai trường hợp sau:

- *Sử dụng Macro để in xâu ký tự*

```
#define PRINT(s) printf ("This is string %s",s);
```

Việc sử dụng Macro PRINT(sec) sẽ sinh ra lỗi cú pháp sau:

```
printf ("This is string% sec", sec);
```

Trong phép thay thế này tất cả các tham số *s* hình thức trong chuỗi được thay thế bởi tham số thực *sec* và do đó đưa đến lỗi trong khi sử dụng phép định dạng *%sec*.

- *Sử dụng Macro để tính bình phương của một số.*

```
#define SQR(x) (x)*(x)
```

Nếu trong chương trình ta sử dụng: *z = SQR(Y++)* thì phép thay thế sẽ đưa đến kết quả sai do giá trị của *Y* được tăng lên hai đơn vị trước khi thực hiện phép tính. Cụ thể biểu thức trên sẽ trở thành: `z = (Y++)*(Y++);`

Như vậy khi nào ta cần sử dụng Macro và khi nào ta nên dùng hàm? Để có thể sử dụng Macro một cách hợp lý trong nhiều điều kiện cụ thể khác nhau ta cần phải nắm rõ những lợi thế cũng như nhược điểm của Macro.

- Macro được thực hiện nhanh hơn hàm bởi vì trong quá trình thực hiện không phải truyền tham số.
- Macro sẽ đòi hỏi nhiều bộ nhớ bởi vì mỗi lần Macro được gọi thì tại vị trí đó được chèn thêm các dòng định nghĩa của Macro.
- Macro có thể sinh ra các hiệu ứng phụ.

• Macro có thể được dùng với đối số và các tham số này có thể được sử dụng với nhiều kiểu dữ liệu khác nhau *(float, int...)* trong khi đó kiểu của tham số khi gọi hàm là không thể thay đổi.

Như vậy sự giống nhau giữa Macro và hàm số chỉ là bên ngoài, về bản chất chúng hoàn toàn khác nhau. Việc sử dụng Macro sẽ dẫn đến việc tăng kích thước của file nguồn khi biên dịch nhưng sẽ giảm thời gian khi thực hiện chương trình, trong khi đó việc sử dụng hàm có thể sẽ đưa đến những hiệu quả ngược lại.

5.4 NỐI KẾT CÁC FILE NGUỒN

Khi xây dựng các chương trình bằng ngôn ngữ C thông thường ta cần phải nối kết một số file nguồn với nhau. Sự cần thiết này được phát sinh do nhiều nguyên nhân:

• Phương pháp lập trình bằng ngôn ngữ C cho thấy việc phân chia chương trình thành các chương trình con (MODULE) với những mục đích khác nhau sẽ làm cho chương trình trở nên dễ hiểu và rất thuận tiện. Trong các trường hợp này việc khai báo các đối tượng chung cho toàn bộ chương trình như các biến tổng thể, khai báo và định nghĩa các hằng số... nên đưa vào một file riêng biệt gọi là file tiêu đề *(Header file)*. Các đối tượng mà việc khai báo phụ thuộc vào các kiểu máy khác nhau cũng nên đưa vào *Header file*, điều này cho phép dễ dàng sửa đổi chương trình khi biên dịch trên các loại máy khác nhau.

• Nguyên nhân thứ 2 là các chương trình phức tạp thường được viết bởi nhiều thảo trình viên hoặc các file được viết trong các thời gian khác nhau nhằm tạo ra sự độc lập khi làm việc. Các file nguồn này sử dụng các thông tin chung cho toàn bộ chương trình thông qua việc sử dụng các *Header file* và thường được biên dịch độc lập với nhau. Phương pháp biên dịch riêng biệt các file có nhiều ưu điểm do có thể tiết kiệm bộ nhớ

khi biên dịch hoặc dễ dàng sửa đổi các file khi cần thiết. Khi sử dụng các phương pháp này, các hàm số được viết trong những file riêng biệt và được gọi là các hàm thư viện của người sử dụng. Một điều hết sức thuận tiện là các hàm sau khi được viết có thể được sử dụng với bất kỳ một chương trình nào đó khi ta liên kết file chứa nó với chương trình. Tuy vậy để có thể tiến hành biên dịch các file một cách độc lập với nhau, người lập trình phải hết sức thận trọng khi khai báo biến và các hàm. Vấn đề này sẽ được trình bày cụ thể ở phần sau.

Để chèn nội dung của một file tiêu đề vào các file khác, bộ tiền xử lý của C sử dụng chỉ thị *include*. Chỉ thị được sử dụng thông qua cú pháp sau:

#include <Tên file>

Hoặc: ***#include "Tên file"***

Thông thường file tiêu đề có phần mở rộng là *h* (từ Header file). Nếu tên của file được đóng trong dấu nháy kép (") thì file là cục bộ và bộ tiền xử lý đầu tiên sẽ tìm file đó trong thư mục nơi chứa chương trình nguồn. Trong trường hợp không tìm thấy, file sẽ được tìm trong những thư mục chuẩn, chẳng hạn thư mục \TC\INCLUDE. Nếu file tiêu đề được đóng trong dấu ngoặc nhọn thì đó là file hệ thống chuẩn và do đó chỉ được tìm kiếm trong các thư mục chuẩn.

Các trình biên dịch của ngôn ngữ C được xây dựng cùng với nhiều tệp tiêu đề chuẩn. Tên các tệp này hầu như không thay đổi trong tất cả các chương trình biên dịch nhưng phải chú ý rằng nội dung của chúng có thể khác nhau. Sau đây là tên của một số tệp chuẩn hay được sử dụng trong các chương trình: *stdio.h, math.h, string.h, ctype.h* v.v...

Các tệp tiêu đề hệ thống được sử dụng để xác định các cấu trúc dữ liệu và các hàm nguyên mẫu trong các thư việc chuẩn. Ví dụ trong tệp tiêu đề *stdio.h* có chứa các hàm nguyên mẫu

dùng để nhập và kết xuất dữ liệu.

Điều gì xảy ra nếu trong chương trình cần sử dụng hai Header file và trong 2 file này đều cần các định nghĩa chứa trong file khác. Chẳng hạn chương trình có sử dụng 2 file tiêu đề *part_a* và *part_b* thông qua các chỉ thị:

```
#include "part_a"
#include "part_b"
```

Tuy nhiên nếu trong hai file trên đều có sử dụng file tiêu đề *cdefs.h* thì các định nghĩa trong file *cdefs.h* sẽ được khai báo hai lần khi biên dịch và do đó sẽ phát sinh lỗi. Để giải quyết các vấn đề này trong bộ tiền xử lý của C có dùng các chỉ thị khác, các chỉ thị này là các chỉ thị biên dịch có điều kiện.

5.5 BIÊN DỊCH CÓ ĐIỀU KIỆN

Các chỉ thị biên dịch có điều kiện cho phép chọn một đoạn chương trình có được biên dịch không. Cơ cấu thực hiện các chỉ thị này được biểu diễn ở dạng *if - else - endif* quen thuộc.

1. Phần *if* trong chỉ thị thường có dạng:

#ifdef Tên Macro

đoạn chương trình C

hoặc:

#if Biểu thức hằng

Đoạn chương trình C

Tên của Macro trong chỉ thị *#ifdef* và biểu thức hằng trong chỉ thị *if* là các điều kiện xác định các lệnh tiếp đó có được thực hiện hay không. Nếu trước phần *#if* trong chương trình đã được định nghĩa Macro có tên trùng với Macro trong chỉ thị *#ifdef* chúng ta nói rằng điều kiện được thỏa mãn. Ngược lại nếu Macro chưa được định nghĩa thì tức là điều kiện không

thỏa mãn.

Trong trường hợp này ta chỉ cần quan tâm Macro có được định nghĩa hay là không, nên để thực hiện các chỉ thị biên dịch có điều kiện ta có thể sử dụng các chỉ thị *#define* trống, chẳng hạn *#define debug* (chỉ thị này có nghĩa là Macro *debug* đã được định nghĩa). Đối với chỉ thị *if* điều kiện là một biểu thức hằng và giá trị của nó được đánh giá theo một qui tắc chung. Nếu giá trị của biểu thức khác 0 thì điều kiện được thỏa mãn và ngược lại nếu giá trị của nó bằng 0 thì điều kiện là không thỏa mãn.

2. Phần *else* của chỉ thị có dạng:

#else

Đoạn chương trình C

3. Để kết thúc cho cấu trúc *if - else* trong C dùng chỉ thị:

#endif

Cấu trúc if - else có thể được ở vị trí bất kỳ trong chương trình và được viết theo cách sau:

#ifdef Tên Macro

Đoạn chương trình C

#else

Đoạn chương trình C

#endif

Bộ tiền xử lý sẽ kiểm tra các điều kiện được viết trong các chỉ thị *#ifdef* hoặc *#if*. Nếu điều kiện thỏa mãn, đoạn chương trình được viết trong phần *#if* sẽ được kết nối vào chương trình để biên dịch, đoạn chương trình được viết sau *#else* sẽ bị bỏ qua hoặc ngược lại.

Ví dụ 5.3

Một trong các phương pháp kiểm tra các chương trình là sử dụng hàm *printf* để in các kết quả trong quá trình thực hiện. Sau khi kiểm tra xong việc in, các kết quả này sẽ trở thành không cần thiết và cần phải loại bỏ ra khỏi chương trình khi biên dịch. Để giải quyết vấn đề này ta sẽ dùng chỉ thị biên dịch có điều kiện trong ví dụ:

```
#ifdef DEBUG
#define PRINT(x)x
#define PIX(x,y) printf(x,y)
#else
#define PRINT(x)
#define PIX(x,y)
#endif
test()
{
PRINT(printf ("Hàm đang được sử dụng để kiểm tra \n");)
...
for(i=0; i<m; i++)
    {
    ...
    PIX("Hàm test: Bước %d\n":i);
    }
   ...
}
```

Giải thích: Trong giai đoạn kiểm tra chương trình Macro DEBUG được định nghĩa bởi chỉ thị *#define DEBUG* và điều này sẽ dẫn đến việc bộ tiền xử lý sẽ chèn các toán lệnh cho phép in các kết quả trung gian qua việc thực hiện các Macro

đã được định nghĩa là PRINT và PIX. Sau khi đã loại bỏ các lỗi trong chương trình, chỉ thị *#define DEBUG* có thể được loại bỏ qua chỉ thị *#undef*. Trong quá trình biên dịch lại, tất cả các Macro đã được định nghĩa trước đó trong hàm số *test* sẽ được thay thế bởi dấu cách. Ưu điểm của phương pháp này là chỉ bằng một chỉ thị *#ifdef* ta có thể điều khiển việc in các kết quả trung gian thay bằng phương pháp thông thường là ở bất cứ vị trí nào cần in kết quả trung gian ta đều phải viết chỉ thị *#ifdef DEBUG*, gọi hàm *printf* và chỉ thị *#endif*.

Việc sử dụng các kỹ thuật trên có thể làm phát sinh một số vấn đề và nhiều khi phát sinh ra lỗi khi biên dịch. Chẳng hạn trong đoạn chương trình dưới đây trường hợp đó sẽ xảy ra nếu Macro DEBUG không được định nghĩa.

```
if(x>y)
    PRINT(printf ("Cấu trúc không đúng\n");)
else
    Đoạn chương trình C;
```

Lỗi phát sinh là do khi Macro DEBUG không được định nghĩa thì khi biên dịch đoạn chương trình trên sẽ trở thành:

```
if(x > y)
else
```

Và là một cú pháp sai của ngôn ngữ C

Bằng cách sử dụng các chỉ thị biên dịch có điều kiện, vấn đề được nêu trong phần 5.4 có thể được giải quyết một cách dễ dàng qua việc đặt các chỉ thị:

```
#ifdef _DEFS_H
#define _DEFS_H
```

ở đầu file cdefs.h và chỉ thị:

```
#endif
```

ở cuối file này.

Ngoài các chỉ thị được nêu ra ở trên, ngôn ngữ C còn sử dụng một số chỉ thị khác phục vụ cho việc biên dịch có điều kiện.

- Chỉ thị *#defined* dùng để kiểm tra một ký hiệu nào đó đã được định nghĩa hay chưa. Chỉ thị này có thể được dùng thay cho chỉ thị *#ifdef*. Ví dụ: *#defined DEBUG* là hoàn toàn tương đương với *#ifdef DEBUG*.

- Chỉ thị *#elif* được xem như cách viết ngắn gọn của cấu trúc "*else-if*". Nó có thể được sử dụng để tạo nên một tập hợp các điều kiện được lồng vào nhau dùng trong quá trình biên dịch. Ví dụ sau minh họa cho việc sử dụng *#elif*:

```
#if defined(PROC TABLE)
#define TableSize
#elifdefined(NAME_TABLE)
#define Table_Size_Name
#endif
```

5.6 CHỈ THỊ LINE

Cũng như các chỉ thị khác, *line* có thể được đặt ở vị trí bất kỳ trong chương trình. Cú pháp của chỉ thị *line* có dạng sau:

#line hằng_số

hoặc

#line hằng_số Tên_file

Trong hai cú pháp này *hằng_số* là một số nguyên được viết dưới dạng cơ số 10.

Trong quá trình biên dịch các dòng lệnh được viết sau chỉ thị *#line* sẽ được bắt đầu đánh số bằng hằng số được viết sau *#line* và các số dòng mới này sẽ được trình biên dịch sử dụng khi thông báo lỗi. Trong trường hợp trong chỉ thị *#line* có sử dụng *Tên_file*, lỗi sẽ được thông báo ở số dòng tương ứng trong

file này.

Ví dụ 5.4

```
void main()
    {
    int i,j,k;
    float x,s;
    i=5;
    #line 100
    s=x;
    printf ("sử dụng chỉ thị #line\n");
    }
```

Khi biên dịch chương trình trên, trình biên dịch sẽ thông báo lỗi ở hàng *3* và hàng *100*. Lỗi ở hàng 3 được phát sinh do ta sử dụng dấu chấm *(.)* thay dấu phẩy *(,)* khi khai báo. Lỗi ở hàng *100* phát sinh do ta sử dụng biến *z* trước khi nó được định nghĩa. Mặc dầu dòng chứa lỗi là dòng thứ *6* trong chương trình nhưng do chúng ta dùng chỉ thị *#line* nên trình biên dịch sẽ thông báo là lỗi được phát hiện ở hàng thứ *100*.