

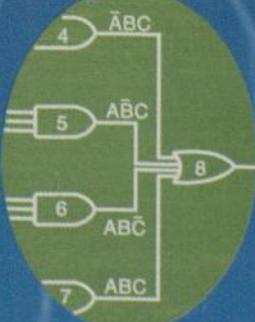
ĐẶNG THÀNH TÍN

HỆ THỐNG MÁY TÍNH VÀ NGÔN NGỮ C



main()
{
int a, b;

LDI R2, VALUE



100101
11001010
011001010
11001010



NHÀ XUẤT BẢN

ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH

**ĐẠI HỌC QUỐC GIA TP HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA**

Đặng Thành Tín

**HỆ THỐNG MÁY TÍNH
VÀ NGÔN NGỮ C**

(Tái bản lần thứ hai có sửa chữa, bổ sung)

**NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA
TP HỒ CHÍ MINH - 2011**

MỤC LỤC

LỜI NÓI ĐẦU

9

Chương 1 ÔN LẠI CÁC KIẾN THỨC CƠ BẢN VỀ MÁY TÍNH

1.1 Các hệ đếm	11
1.2 Các khái niệm cơ bản	11
1.3 Lịch sử phát triển của máy tính	17
1.4 Các thành phần cơ bản của máy tính	25
1.5 Phần mềm	30
1.6 Các cấp chuyển đổi	35
Bài tập cuối chương	38
	44

Chương 2 CÁC KIỂU DỮ LIỆU VÀ THAO TÁC

2.1 Kiểu dữ liệu số nguyên	46
2.2 Số nguyên bù 2	46
2.3 Phép toán trên bit - phép toán số học	47
2.4 Phép toán trên bit - phép toán luận lý	50
2.5 Kiểu dữ liệu dấu chấm động	52
Bài tập cuối chương	57
	60

Chương 3 CÁC CẤU TRÚC LUẬN LÝ SỐ

3.1 Transistor	64
3.2 Cổng luận lý	64
3.3 Mạch tổ hợp	65
3.4 Phần tử nhớ cơ bản	70
3.5 Bộ nhớ	74
3.6 Mạch tuần tự	77
3.7 Đường truyền dữ liệu LC 3	81
Bài tập cuối chương	88
	91

Chương 4 MÔ HÌNH VON NEUMANN VÀ KIẾN TRÚC TẬP LỆNH LC-3	95
4.1 Các thành phần cơ bản	95
4.2 Một ví dụ về mô hình von neumann: LC-3	99
4.3 Quá trình xử lý lệnh	101
4.4 Thay đổi quá trình xử lý lệnh	105
4.5 Khái niệm ISA LC-3	108
4.6 Nhóm lệnh thi hành	112
4.7 Nhóm lệnh di chuyển dữ liệu	115
4.8 Nhóm lệnh điều khiển	122
4.9 Ba cấu trúc lệnh trong LC-3	131
4.10 Một ví dụ	133
Bài tập cuối chương	138
Chương 5 LẬP TRÌNH HỢP NGỮ LC-3	141
5.1 Lập trình hợp ngữ	141
5.2 Các thành phần của một chương trình hợp ngữ	142
5.3 Quá trình hợp dịch	150
5.4 Chương trình với nhiều modul	155
5.5 Một số ví dụ	156
Bài tập cuối chương	164
Chương 6 CÁC VẤN ĐỀ KHÁC	166
6.1 Xuất nhập	166
6.2 Trap và chương trình con	180
6.3 Ngăn xếp	193
Bài tập cuối chương	206
Chương 7 GIỚI THIỆU NGÔN NGỮ LẬP TRÌNH C	209
7.1 Giới thiệu	209
7.2 Bộ dịch ngôn ngữ lập trình C	210
7.3 Các ví dụ	213
Bài tập cuối chương	219

Chương 8 CÁC THÀNH PHẦN CƠ BẢN VÀ CÁC KIỂU DỮ LIỆU CỦA C	220
8.1 Danh hiệu	220
8.2 Các kiểu dữ liệu chuẩn của C	223
8.3 Hằng	230
8.4 Biến	238
8.5 Biểu thức	242
8.6 Các phép toán của C	242
8.7 Cấu trúc tổng quát của một chương trình C	267
Bài tập cuối chương	269
Chương 9 CÁC LỆNH ĐIỀU KHIỂN VÀ VÒNG LẶP	270
9.1 Lệnh đơn và lệnh phức	270
9.2 Lệnh if	271
9.3 Lệnh switch-case	278
9.4 Lệnh while	281
9.5 Lệnh do-while	285
9.6 Lệnh for	290
9.7 Lệnh break và lệnh continue	296
9.8 Lệnh return	300
9.9 Lệnh goto	303
9.10 Lệnh rỗng	305
9.11 Một số ví dụ	307
Bài tập cuối chương	312
Chương 10 HÀM	314
10.1 Khái niệm hàm	314
10.2 Khai báo hàm	318
10.3 Đối số của hàm - đối số là tham trị	322
10.4 Kết quả trả về của hàm - lệnh return	327
10.5 Prototype của một hàm	334
10.6 Hàm đệ quy	338
10.7 Hiện thực hàm trong C	341
10.8 Kiểm tra và sửa lỗi	354
Bài tập cuối chương	370

Chương 11 LỚP LUU TRỮ CỦA BIẾN - SỰ CHUYỂN KIỂU	372
11.1 Khái niệm	372
11.2 Biến toàn cục và biến cục bộ	373
11.3 Biến tĩnh	383
11.4 Biến register	393
11.5 Khởi động trị cho biến ở các lớp	397
11.6 Sự chuyển kiểu	399
11.7 Định vị vùng nhớ cho các lớp lưu trữ	400
Bài tập cuối chương	408
Chương 12 MẢNG	409
12.1 Khái niệm	409
12.2 Khai báo mảng	409
12.3 Khởi động trị của mảng	420
12.4 Mảng là đối số của hàm, mảng là biến toàn cục	422
12.5 Các ứng dụng	428
Bài tập cuối chương	441
Chương 13 POINTER	443
13.1 Khái niệm	443
13.2 Thao tác trên pointer	446
13.3 Pointer và mảng	459
13.4 Đối số của hàm là pointer - truyền đối số theo dạng tham số biến	468
13.5 Hàm trả về pointer và mảng	475
13.6 Chuỗi ký tự	478
13.7 Pointer và việc định vị bộ nhớ động	487
13.8 Mảng các pointer	491
13.9 Pointer của pointer	498
13.10 Đối số của hàm main	504
13.11 Pointer trả đến hàm	506
13.12 Ứng dụng	509
Bài tập cuối chương	526

Chương 14 CÁC KIỂU DỮ LIỆU CÓ CẤU TRÚC VÀ KIỂU DỮ LIỆU TỰ ĐỊNH NGHĨA	527
14.1 Kiểu struct	527
14.2 Kiểu union	543
14.3 Kiểu enum	549
14.4 Định nghĩa kiểu bằng typedef	551
Bài tập cuối chương	554
Chương 15 TẬP TIN	555
15.1 Khái niệm	555
15.2 Các hàm truy xuất file	556
Bài tập cuối chương	568
Chương 16 CHẾ ĐỘ ĐỒ HỌA	569
16.1 Giới thiệu	569
16.2 Các hàm sử dụng trong chế độ đồ họa	569
Bài tập cuối chương	591
Chương 17 LỆNH TIỀN XỬ LÝ	592
17.1 #define và #undef	592
17.2 #include	597
17.3 #if-else #endif	599
17.4 #error	604
Bài tập cuối chương	605
Chương 18 CÁC HÀM TRONG THƯ VIỆN CHUẨN CỦA C	606
18.1 Thư viện của C	606
18.2 Thao tác xuất nhập chuẩn	607
18.3 Các hàm khác	619
18.4 Các hàm không chuẩn của các chương trình biên dịch	637

Chương 19	ĐỆ QUY	638
19.1	Đệ quy là gì?	638
19.2	Đệ quy và lặp	640
19.3	Tháp HÀ NỘI	640
19.4	Dãy số Fibonacci	647
19.5	Tìm kiếm nhị phân	651
19.6	Chuyển số nguyên sang dãy ký tự ASCII	653
19.7	Cấu trúc dữ liệu cây - cây nhị phân	656
Chương 20	GIỚI THIỆU LẬP TRÌNH C++	669
20.1	Lập trình hướng đối tượng	669
20.2	Constructor và destructor	672
20.3	Toán tử new và delete	674
20.4	Sự thừa kế dữ liệu	674
20.5	Từ khóa static	676
20.6	Hàm ảo (sự thừa kế theo chức năng)	677
20.7	Tham khảo trong C++	680
20.8	Một số điểm khác biệt chính giữa C và C++	681
20.9	Một số chương trình ví dụ	687
Phụ lục		707
Phụ lục A	Kiến trúc tập lệnh LC-3	707
Phụ lục B	Bảng mã ASCII	717
Phụ lục C	Các phím mở rộng trên bàn phím	727
Phụ lục D	Cách tạo project file	729
TÀI LIỆU THAM KHẢO		732

LỜI NÓI ĐẦU

Trong môn học TIN HỌC, sinh viên đã được cung cấp các kiến thức cơ bản về hệ điều hành và cǎn bǎn lập trình Visual Basic hoặc Pascal, qua đó đã có thể thao tác, vận hành máy và có khả năng lập trình trên ngôn ngữ Visual Basic hoặc Pascal. Tuy nhiên, hiện nay để thiết kế phần mềm, đặc biệt là phần mềm hệ thống để điều khiển robot, thiết kế kit vi xử lý, ..., nhiều lập trình viên sử dụng hợp ngữ hoặc ngôn ngữ C. Đây là các ngôn ngữ cấp thấp hoặc ngôn ngữ có khả năng giao tiếp được với hệ thống, nên được đánh giá là phù hợp để thiết kế phần mềm, từ các phần mềm ứng dụng thuận tiện đến các phần mềm hệ thống có giao tiếp với phần cứng.

Tài liệu HỆ THỐNG MÁY TÍNH và NGÔN NGỮ C này là lần ấn bản thứ tư có sửa chữa và bổ sung tinh từ các ấn bản TIN HỌC II trước đây. Nó cung cấp các kiến thức cơ bản về hệ thống máy tính từ các khái niệm ban đầu như máy đa cấp tới các cấp máy cụ thể như cấp luận lý số, cấp vi kiến trúc, cấp kiến trúc tập lệnh tới cấp hợp ngữ. Việc lập trình bằng mã máy và hợp ngữ với nền LC-3 sẽ được trình bày để minh họa cho lập trình cấp thấp vốn thường gây khó khăn cho người học. Ngôn ngữ C và cách lập trình trên nó với các vấn đề mở rộng, rồi các khác biệt giữa C và C++ cũng sẽ được đề cập, từ đó sinh viên có thể biết và vận dụng lập trình cấu trúc trên C và hướng đối tượng trên C++.

Cuốn HỆ THỐNG MÁY TÍNH và NGÔN NGỮ C này được chia ra làm 20 chương, tập trung vào bốn phần như sau:

Phần 1: gồm chương 1 và chương 2, trình bày một cách vấn tắt các kiến thức cơ bản về hệ thống máy tính như các hệ thống số, khái niệm phần cứng, phần mềm, các cấp chuyển đổi, ...

Phần 2: từ chương 3 tới chương 6, trình bày các cấp luận lý số, kiến trúc tập lệnh LC-3 với việc lập trình bằng mã máy và hợp ngữ, và một số vấn đề khác như stack, chương trình con, ngắt....

Phần 3: từ chương 7 đến chương 19, trình bày ngôn ngữ C và các thành phần như các kiểu dữ liệu, các lệnh, các cấu trúc, ... theo cả hai quan điểm là lập trình ứng dụng và lập trình hệ thống. Một số ứng dụng của C khi lập trình như chế độ văn bản, chế độ đồ

họa, truy xuất tập tin, các cấu trúc dữ liệu như stack, queue, tree cũng được trình bày.

Phân 4: chương 20, trình bày các khác biệt giữa C và C++. Đối tượng và lập trình hướng đối tượng cũng được nêu lên ở đây.

Trong mỗi chương đều có phần ví dụ minh họa cho mỗi trường hợp cụ thể để độc giả có thể tự học một cách dễ dàng, cuối mỗi chương đều có phần bài tập được sắp theo mức độ từ dễ tới khó để độc giả tự làm.

Đối tượng của giáo trình này là sinh viên năm 1, 2 của các trường kỹ thuật, vốn rất cần nắm chắc một ngôn ngữ lập trình giao tiếp phản ứng. Tuy nhiên, kiến thức là không hạn chế nên mọi độc giả có thể tìm được ở đây những kiến thức hữu ích về mã máy, lập trình hợp ngữ, lập trình ngôn ngữ C và C++.

Sau cùng, tác giả xin chân thành cảm ơn các thầy cô, các sinh viên cao học, sinh viên đại học đã giúp đỡ và cho rất nhiều ý kiến đóng góp hữu ích. Tác giả cũng rất cảm ơn cô con gái Ánh Châu, vốn đang là học sinh lớp 9³, trường THCS Nguyễn Gia Thiều, năm học 2010-2011 đã mất nhiều thời gian chế bản bìa sách và các hình trong sách này.

Tác giả rất mong nhận được nhiều ý kiến đóng góp của các đồng nghiệp và quý độc giả để lần tái bản sau sẽ được hoàn thiện hơn. Mọi góp ý xin gửi về:

Trường Đại học Bách khoa - Đại học Quốc gia TP.HCM

268 Lý Thường Kiệt Q.10, TP. HCM

Email: dttin@hcmut.edu.vn

cuu duong than cong . com

Tác giả

TS. ĐẶNG THÀNH TÍN

Chương 1

ÔN LẠI CÁC KIẾN THỨC CƠ BẢN VỀ MÁY TÍNH

1.1 CÁC HỆ ĐẾM

Khi sử dụng máy tính hay khi tính toán các vấn đề trong thực tế, con người đã rất quen thuộc với hệ đếm thập phân, còn gọi là hệ 10. Tuy nhiên, việc sử dụng hệ này sẽ gặp một số khó khăn khi sử dụng máy tính, đặc biệt là đối với các chương trình hệ thống, là các chương trình điều khiển phần cứng máy tính hay các chương trình phục vụ cho sự hoạt động của máy tính như hệ điều hành, ROM BIOS,..., hay các khái niệm về phần cứng của hệ thống máy tính.

Bên trong hệ thống máy tính, việc sử dụng hệ thống số dựa trên cơ số 2, vì máy tính hiện đang được sử dụng là hệ thống máy tính số (*digital computer*), trong đó các mạch số chỉ hoạt động dựa trên hai mức điện áp là thấp và cao, và từ đó mọi dữ liệu khác sẽ được mã hóa theo hệ 2. Tuy nhiên, hệ 2 lại dài dòng và phức tạp nên để dễ dàng trong việc ghi nhận và hiển thị, người ta cung cấp thêm hai hệ thống số đếm nữa là hệ bát phân (*octal system*), sử dụng cơ số 8, và hệ thập lục phân (*hexadecimal system*), hay còn được gọi tắt là hệ hex, sử dụng cơ số 16.

1.1.1 Hệ thập phân

Đây là hệ đếm thông dụng đối với con người. Hệ này dựa trên nguyên tắc đếm 10 ngón tay của con người, do đó có 10 ký số được sử dụng từ 0, ..., 9. Cơ số sử dụng là cơ số 10. Theo quy ước chung, số trong hệ thập phân sẽ được viết thêm ký tự D hay d phía sau, tức viết tắc từ từ tiếng Anh: *decimal* (tức *decimal system*), hoặc chỉ có số mà thôi.

Ví dụ 1.1 Các hằng số trong hệ 10:

102, 3098.34D, 198d

Một số trong hệ 10 sẽ được viết ở dạng phân tích như ví dụ 1.2 sau.

Ví dụ 1.2 Các số sau đây được viết ở dạng phân tích trong hệ thập phân

$$1986D = 1 \cdot 10^3 + 9 \cdot 10^2 + 8 \cdot 10^1 + 6 \cdot 10^0$$

$$234d = 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0$$

$$0.163 = 1 \cdot 10^{-1} + 6 \cdot 10^{-2} + 3 \cdot 10^{-3}$$

Lập trình viên khi viết chương trình có thể sử dụng các hằng số cơ số 10, và cách viết thường theo quy ước **không kèm theo D hay d** phía sau hằng số đó.

Chú ý phần thập phân được viết bắt đầu bằng dấu chấm (.)

1.1.2 Hệ nhị phân

Đây là hệ đếm chính thức dùng cho máy tính. Các mạch số trong máy tính sử dụng hai mức điện áp thấp và cao để quy định cho 2 trạng thái số làm việc là 0 và 1, thường thì mức điện áp cao quy định cho trạng thái số 1, và mức điện áp thấp quy định cho trạng thái số 0. Trạng thái số nhị phân được gọi là bit, viết tắt từ **binary digit**: Việc ghép các ký số 0 và 1 lại để mã hóa mọi dữ liệu để máy tính xử lý là điều cần thiết.

Hệ nhị phân, hay còn tắt là hệ 2, sử dụng hai ký số 0 và 1 để mã hóa dữ liệu, cơ số sử dụng là 2. Các số trong hệ 2 thường được viết có thêm ký tự **B** hay **b** phía sau.

Ví dụ 1.3 Các hằng số trong hệ 2

1011B, 101010b, 1010101.101B

Dạng phân tích của các số hệ 2 thường được sử dụng để đổi số từ hệ 2 sang hệ 10, trong đó cơ số sử dụng là 2.

Ví dụ 1.4

$$10101B = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 21D$$

$$11.01B = 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 3.25D$$

Khi lập trình bằng ngôn ngữ cấp cao như Pascal, các hằng số hệ 2 thường không được xử dụng trực tiếp mà phải qua trung gian hệ 8 hay hệ 16.

Một số nhị phân có 2 bit sẽ có $2^2 = 4$ trạng thái số đi từ 0 tới $2^2 - 1$, các trạng thái đó là 00, 01, 10, 11. Nếu số nhị phân có n bit thì ta sẽ có 2^n trạng thái số đi từ 0 tới $2^n - 1$, các trạng thái đó là

Trạng thái	Thập phân
0 0 ... 0	0
:	:
1 1 ... 1	$2^n - 1$
$\leftarrow n \text{ bit} \rightarrow$	

Trong máy tính số, việc mã hóa địa chỉ của bộ nhớ hay thiết bị đều theo hệ nhị phân. Một đường dây tín hiệu có thể có một trong hai trạng thái điện áp thấp hay điện áp cao, tương ứng bit 0 và bit 1. Nếu ta có 10 đường dây tín hiệu là địa chỉ thì nó có thể quản lý được $2^{10} = 1024$ ô nhớ khác nhau.

1.1.3 Hệ bát phân

Hệ này được coi như là dạng thức thu gọn của hệ 2, vì các số hệ 2 thường khó nhìn và khó nhớ (vì chỉ có 2 ký số 0 và 1, khá đơn điệu!), do đó người ta đưa ra quy ước dùng hệ 8 để ghi nhận dữ liệu hệ 2 một cách hiệu quả và dễ nhớ, đặc biệt là để biểu diễn dữ liệu của chương trình.

Hệ bát phân sử dụng cơ số 8, do đó có 8 ký số trong hệ này là 0, 1, ..., 7. Các hằng số trong hệ 8 khi viết thường có thêm ký tự O hay o (viết tắt từ octal) phía sau.

Ví dụ 1.5 Các hằng trong hệ bát phân:

734O, 123.56o, -34.23O

Dạng phân tích số hệ 8 cũng có nghĩa là đổi số hệ 8 ra hệ 10.

Ví dụ 1.6

$$705 \text{ O} = 7 \cdot 8^2 + 0 \cdot 8^1 + 5 \cdot 8^0 = 453 \text{ D}$$

$$123.56 \text{ O} = 1 \cdot 8^2 + 2 \cdot 8^1 + 3 \cdot 8^0 + 5 \cdot 8^{-1} + 6 \cdot 8^{-2}$$

Một ký số trong hệ bát phân có thể được biểu diễn bằng ba ký số trong hệ nhị phân, vì ở đây có sự tương ứng các ký số trong 2 hệ: 0, là ký số thấp nhất trong hệ 8, tương ứng với 000 trong hệ 2, và 7, là ký số lớn nhất trong hệ 8, tương ứng với 111 trong hệ 2. Bảng 1.1 sau đây trình bày cụ thể mối quan hệ này.

Bảng 1.1 Sự chuyển đổi giữa ký số trong hệ 8 và hệ 2

Ký số bát phân	Tương ứng nhị phân	Tương ứng thập phân
0	000	0
1	001	1
2	010	2
3	011	3
4	100	4
5	101	5
6	110	6
7	111	7

Do đó việc chuyển đổi từ hệ 8 qua hệ 2 và ngược lại thực tế là việc “xá” và “gom” bit, cứ mỗi ký số bát phân, ta sẽ có 3 bit, và cứ mỗi 3 bit ta sẽ có 1 ký số bát phân theo bảng 1.1.

Ví dụ 1.7 Chuyển số từ hệ 8 qua hệ 2 và ngược lại

$$\underline{1} \ \underline{101} \ \underline{011} \ \underline{011} \text{ B} = 1533 \text{ O}$$

1 5 3 3

$$245 \text{ O} = \underline{010} \ \underline{100} \ \underline{101} \text{ B} = 10100101 \text{ B}$$

1.1.4 Hệ thập lục phân

Hệ thống số 16 (còn được gọi tắt là hệ **hex**) này cũng có mục đích tương tự như hệ 8, tuy nhiên hiện nay nó được sử dụng rộng rãi hơn hệ 8, vì tính năng gọn và dễ viết hơn của nó. Trong chương trình, lập trình viên có thể viết trực tiếp các hằng số trong hệ này. Tùy vào ngôn ngữ mà việc viết hằng số 16 sẽ khác nhau. Hệ hex còn được sử dụng trong lãnh vực phần cứng, để quy ước việc mã hóa địa chỉ ô nhớ, thiết bị trong hệ thống...

Hệ 16 sử dụng cơ số 16, có 16 ký số khác nhau trong hệ thống số đếm này từ 0, 1, ..., 9, A, B, C, D, E, F. Trong đó các ký số từ A tới F quy ước cho các giá trị 10, tới 15. Các hằng hệ hex khi viết thường được viết thêm ký tự **H** hay **h** phía sau số đã có.

Ví dụ 1.8 Một số hằng trong hệ hex:

12A H, 234.907 H, B800 h

Dạng phân tích số hệ 16 cũng có nghĩa là đổi số hệ 16 ra hệ 10.

Ví dụ 1.9

$$F0 H = 15 \cdot 16^1 + 0 \cdot 16^0 = 240 D$$

$$FF H = 15 \cdot 16^1 + 15 \cdot 16^0 = 255 D$$

$$FFFF H = 15 \cdot 16^3 + 15 \cdot 16^2 + 15 \cdot 16^1 + 15 \cdot 16^0 = 65535$$

Một ký số trong hệ hex có thể được biểu diễn bằng 4 ký số trong hệ nhị phân, vì có sự tương ứng các ký số trong 2 hệ: 0, là ký số thấp nhất trong hệ 16, tương ứng với 0000 trong hệ 2, và F (có giá trị là 15), là ký số lớn nhất trong hệ 15, tương ứng với 1111 trong hệ 2. Bảng 1.2 sau đây trình bày cụ thể mối quan hệ này.

Bảng 1.2 Sự chuyển đổi giữa ký số trong hệ 16 và hệ 2

Ký số hệ hex	Tương ứng nhị phân	Tương ứng thập phân
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Việc chuyển đổi các số từ hệ 16 qua hệ 2 và ngược lại thực tế là việc “xả” và “gom” bit, cứ mỗi ký số hex, ta sẽ có 4 bit, và cứ mỗi 4 bit ta sẽ có 1 ký số hex theo bảng 1.2.

Ví dụ 1.10 Chuyển số từ hệ 16 qua hệ 2 và ngược lại

$$\underline{11} \underline{0101} \underline{1011} \text{ B} = 35\text{B H}$$

3 5 B

$$3\text{B H} = \underline{0011} \underline{1011} \text{ B} = 111011 \text{ B}$$

3 B

1.1.5 Sự chuyển đổi qua lại giữa các hệ thống số

Vì đặc tính liên thông trong việc gom và xả bit giữa hệ 2, và hệ 8, hệ 16 mà việc chuyển đổi qua lại giữa các số trong các hệ thống số khác nhau có thể quy về hai nhóm chuyển đổi chính: (1) chuyển từ số hệ 10 sang các hệ còn lại, mà chủ yếu là chuyển từ hệ 10 sang hệ 2, sau đó từ hệ 2 thực hiện việc gom bit để có số trong hệ 8 hay 16; (2) và ngược lại chuyển từ các hệ còn lại sang hệ 10, thực tế đây chính là dạng phân tích của các số trong mỗi hệ. Tuy nhiên, việc chuyển đổi cũng có thể được thực hiện trực tiếp từ hệ 10 sang các hệ còn lại, và việc thực hiện sẽ dựa vào nguyên tắc chia số hệ 10 cho cơ số hệ tương ứng cần đổi. Cụ thể để chuyển số trong hệ thập phân sang hệ nhị phân, ta thực hiện phép chia cho số 2, tương tự đối với hệ 8 và hệ 16.

Ví dụ 1.11 Chuyển số 27 trong hệ thập phân sang nhị phân

Ta thực hiện phép chia nguyên, lấy số dư như sau

cuu duong than cong . com

Cột biểu diễn thương số nguyên	$\frac{27}{13} \quad 2$ $13 \quad 1$ $6 \quad 1$ $3 \quad 0$ $1 \quad 1$ $0 \quad 1$	\Leftarrow Cột biểu diễn số dư
-----------------------------------	---	--

Kết quả của phép biến đổi sẽ là cột số dư đọc theo thứ tự từ dưới lên trên, vì khi chia ký số có trọng số lớn hơn sẽ xuất hiện sau. Do đó $27 = 11011 \text{ B}$.

Ví dụ 1.12 Chuyển số 367 trong hệ thập phân sang hệ bát phân

Ta thực hiện phép chia nguyên, lấy số dư như sau

Cột biểu diễn thương số nguyên	\Rightarrow	<table border="1" style="border-collapse: collapse; width: 100px;"> <tr><td style="width: 40px; height: 20px;"></td><td style="width: 40px; height: 20px; text-align: right;">8</td></tr> <tr><td style="width: 40px; height: 20px; text-align: right;">45</td><td style="width: 40px; height: 20px; text-align: right;">7</td></tr> <tr><td style="width: 40px; height: 20px; text-align: right;">5</td><td style="width: 40px; height: 20px; text-align: right;">5</td></tr> <tr><td style="width: 40px; height: 20px; text-align: right;">0</td><td style="width: 40px; height: 20px; text-align: right;">5</td></tr> </table>		8	45	7	5	5	0	5	\Leftarrow	Cột biểu diễn số dư
	8											
45	7											
5	5											
0	5											

Kết quả của phép biến đổi sẽ là cột số dư đọc theo thứ tự từ dưới lên trên, do đó $367 = 557_8$.

Ví dụ 1.13 Chuyển số 367 trong hệ thập phân sang hệ thập lục phân

Ta thực hiện phép chia nguyên, lấy số dư như sau

Cột biểu diễn thương số nguyên	\Rightarrow	<table border="1" style="border-collapse: collapse; width: 100px;"> <tr><td style="width: 40px; height: 20px;"></td><td style="width: 40px; height: 20px; text-align: right;">16</td></tr> <tr><td style="width: 40px; height: 20px; text-align: right;">22</td><td style="width: 40px; height: 20px; text-align: right;">15</td></tr> <tr><td style="width: 40px; height: 20px; text-align: right;">1</td><td style="width: 40px; height: 20px; text-align: right;">6</td></tr> <tr><td style="width: 40px; height: 20px; text-align: right;">0</td><td style="width: 40px; height: 20px; text-align: right;">1</td></tr> </table>		16	22	15	1	6	0	1	$\rightarrow F$	\Leftarrow	Cột biểu diễn số dư
	16												
22	15												
1	6												
0	1												

Kết quả của phép biến đổi sẽ là cột số dư đọc theo thứ tự từ dưới lên trên, do đó $367 = 16F_{16}$.

1.2 CÁC KHÁI NIỆM CƠ BẢN

1.2.1 Tin học

Tin học là ngành khoa học xử lý thông tin tự động bằng máy tính điện tử. Ở đây có ba khái niệm chính là xử lý, thông tin và máy tính.

Xử lý bao hàm khái niệm tính toán các dữ liệu mà thông tin cung cấp; thông tin là các dữ liệu đưa vào cho máy tính, đó chính là các dữ liệu mà người sử dụng máy tính hoặc từ thiết bị sử dụng ngoài nào đó đưa vào hay là dữ liệu do bản thân máy tính tạo ra; máy tính là thiết bị xử lý thông tin theo chương trình.

1.2.2 Đơn vị tin học

1.2.2.1 Bit

Bit là đơn vị cơ sở của thông tin. Một bit có thể có hai trạng thái. Đối với máy tính một bit có thể có hai trạng thái là 0 và 1.

Nếu coi thông tin là một cái nhà thì bit có thể được coi như là “viên gạch” để tạo nên thông tin.

1.2.2.2 Byte

Byte là đơn vị thông tin nhỏ nhất, nó có thể được dùng để lưu mã của ký tự. Một byte có 8 bit, do đó nó có thể biểu diễn được 256 trạng thái số nhị phân khác nhau. Hiện nay bộ nhớ máy tính cũng được tính theo đơn vị byte.

Các đơn vị bội của byte là KB (*kilo byte*), MB (*mega byte*), GB (*giga byte*) và TB (*tera byte*):

$$1\text{KB} = 2^{10} \text{ byte} = 1024 \text{ bytes}$$

$$1\text{-MB} = 2^{10} \text{ KB}$$

$$1\text{ GB} = 2^{10} \text{ MB}$$

$$1\text{ TB} = 2^{10} \text{ GB}$$

1.2.3 Máy tính

Máy tính là thiết bị hay công cụ dùng để lưu trữ và xử lý thông tin theo một chương trình định trước.

Tùy theo tính năng và mục đích sử dụng người ta phân ra bốn loại máy tính:

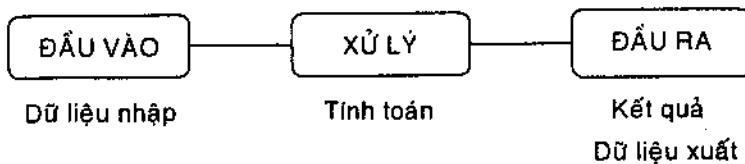
- *Siêu máy tính (super computer)*: các máy tính này có tính năng xử lý rất mạnh, tốc độ tính từ vài chục tới trăm Mips (*Million instruction per second*), thường được sử dụng trong các trung tâm tính toán hay mô phỏng lớn. Giá tiền của các máy tính này từ vài triệu đô la Mỹ trở lên.
- *Máy tính lớn (main frame)*: đây là các máy tính có tốc độ tính toán tương đương hay kém hơn siêu máy tính một chút, tốc độ tính từ vài Mips tới vài chục Mips, trăm Mips, nhưng thường được sử dụng làm máy tính chủ trong các hệ thống mạng lớn.
- *Máy tính trung (mini computer)*: có tốc độ tính toán kém hơn máy tính lớn. Hiện nay do công nghệ vi mạch ngày càng phát triển máy tính trung ngày càng rẻ tiền, và có cấu hình ngày càng mạnh hơn trước.

- *Máy vi tính (micro computer)*, hay còn gọi là máy tính cá nhân (*personal computer*) là máy tính được sử dụng rộng rãi trong gia đình hay công sở. Có hai họ máy tính cá nhân phổ biến là PC (do hãng IBM thiết kế) và Mac (do hãng Apple thiết kế).

1.2.4 Xử lý dữ liệu

Xử lý dữ liệu là quá trình tính toán theo một chương trình định trước. Bên trong máy tính, việc tính toán có thể được phân ra làm hai loại phép toán: phép toán số học và phép toán luận lý. Dữ liệu cần xử lý có được qua quá trình thu thập, phân loại, thống kê, hỏi đáp hoặc từ một quá trình nào đó, ..., nói chung là bằng bất kỳ hình thức nào miễn có dạng thích hợp để máy tính có thể hiểu được, tức dạng tín hiệu số, và phù hợp với yêu cầu chương trình. Sau khi xử lý xong, dữ liệu đầu ra sẽ là kết quả từ quá trình tính toán theo chương trình, với dữ liệu đầu vào đã cho. Dữ liệu đầu ra có thể là kết quả của quá trình phân tích dữ liệu đầu vào, có thể là dữ liệu đầu vào cho một quá trình điều khiển nào khác, ...

Quá trình xử lý dữ liệu có thể được tóm tắt theo sơ đồ sau:



1.2.5 Bộ mã ký tự

1.2.5.1 Khái niệm mã ký tự

Trong cuộc sống con người, các ký tự dùng để biểu thị chữ, số, các ký hiệu, các dấu ..., ký tự là cơ sở để tạo nên ngôn ngữ, gồm các số, các tên, các quan hệ, các cú pháp ..., do đó muốn máy tính hiểu được cần mã hóa chúng. Trong máy tính, do cấu trúc từ mạch số mà việc xử lý dữ liệu đều theo dạng nhị phân, do đó việc mã hóa ký tự sẽ theo một luật mã nhị phân xác định tùy theo bảng mã.

Một cách tổng quát, một bộ mã sẽ luôn bao gồm các nhóm ký tự sau:

- Nhóm ký tự điều khiển: gồm các ký tự điều khiển màn hình, bàn phím, quá trình giao nhận dữ liệu truyền nối tiếp, song song, ...
- Nhóm ký tự số và chữ: các ký tự trong bộ mẫu tự của ngôn ngữ, các ký số.

- Nhóm ký tự đặc biệt: gồm các ký tự như dấu chấm, chấm phẩy, ...
- Nhóm ký tự mở rộng đồ họa: gồm các ký tự mở rộng khác như các dấu tạo hình chữ nhật, dấu tích phân, dấu bình phương,

Hiện nay có nhiều bộ mã ký tự chuẩn đang được sử dụng, đó là EBCDIC, ASCII, UNICODE và sẽ được đề cập dưới đây, tuy nhiên mỗi quốc gia sẽ tùy theo hệ thống mẫu tự của nước mình mà có thể tạo riêng bộ mã ký tự thích hợp. Ngày nay, trong thời đại Internet, việc tìm ra một bộ mã chung là điều vô cùng quan trọng, và bộ mã Unicode ngày càng tỏ ra chiếm ưu thế so với một số mã khác.

1.2.5.2 Bộ mã EBCDIC (Extended Binary Coded Decimal Interchange Code)

Bộ mã này được sử dụng trên các máy tính lớn. Đặc trưng của bộ mã này là các ký tự alphabet không liên tục nhau. Mỗi ký tự trong bảng mã được mã hóa bằng 8 bit, do đó bộ mã này có 256 ký tự. Ví dụ trong bộ mã này:

Bảng 1.3 Một số ký tự trong bảng mã EBCDIC

Ký tự	Mã thập phân	Mã nhị phân
A	193	11000001
B	194	11000010
:	:	:
I	201	11001001
J	209	11010001
K	210	11010010
:	:	:
R	217	11011001
S	226	11100010
T	227	11100011
:	:	:
Z	233	11101001

1.2.5.3 Bộ mã ASCII (American Standard Code for Information Interchange)

Bộ mã này được sử dụng trên hầu hết các máy tính, máy in, Đặc trưng của bộ mã này là các ký tự alphabet liên tục nhau. Bộ mã này cũng dùng 8 bit để mã hóa cho một ký tự của bảng mã, do đó

cũng có 256 ký tự trong bảng mã này. Bảng 1.4 sau đây trình bày bộ mã này, lưu ý là bảng mã này không thể hiện dạng đồ họa của ký tự như Phụ lục B.

Bảng 1.4 Bảng mã ASCII

D	H	Ch	Ctrl	Mem	D	H	Ch	Ctrl	Mem	D	H	Ch	D	H	Ch
0	00	^@	NUL		16	10		^P	DLE	32	20	SP	48	30	0
1	01	^A	SOH		17	11		^Q	DC1	33	21	!	49	31	1
2	02	^B	STX		18	12		^R	DC2	34	22	"	50	32	2
3	03	^C	ETX		19	13		^S	DC3	35	23	#	5	33	3
4	04	^D	EOT		20	14		^T	DC4	36	24	\$	52	34	4
5	05	^E	ENQ		21	15		^U	NAK	37	25	%	53	35	5
6	06	^F	ACK		22	16		^V	SYN	38	26	&	54	36	6
7	07	^G	BEL		23	17		^W	ETB	39	27	'	55	37	7
8	08	^H	BS		24	18		^X	CAN	40	28	(56	38	8
9	09	^I	HT		25	19		^Y	EM	41	29)	57	39	9
10	0A	^J	LF		26	1A		^Z	SUB	42	2A	=	58	3A	:
11	0B	^K	VT		27	1B		^_	ESC	43	2B	+	59	3B	;
12	0C	^L	FF		28	1C		^`	PS	44	2C	-	60	3C	<
13	0D	^M	CR		29	1D		^]	GS	45	2D	.	61	3D	=
14	0E	^N	SO		30	1E		^^	RS	46	2E	,	62	3E	>
15	0F	^O	SI		31	1F		^~	US	47	2F	/	63	3F	?

D	H	Ch	D	H	Ch	D	H	Ch	D	H	Ch	D	H	Ch		
64	40	®	80	50	P	96	60	'	112	70	p	128	80	144	90	
65	41	A	81	51	Q	97	61	a	113	71	q	129	81	145	91	
66	42	B	82	52	R	98	62	b	114	72	r	130	82	146	92	
67	43	C	83	53	S	99	63	c	115	73	s	131	83	147	93	
68	44	D	84	54	T	100	64	d	116	74	t	132	84	148	94	
69	45	E	85	55	U	101	65	e	117	75	u	133	85	à	149	95
70	46	F	86	56	V	102	66	f	118	76	v	134	86	150	96	
71	47	G	87	57	W	103	67	g	119	77	w	135	87	151	97	
72	48	H	88	58	X	104	68	h	120	78	x	136	88	é	152	98
73	49	I	89	59	Y	105	69	i	121	79	y	137	89	153	99	
74	4A	J	90	5A	Z	106	6A	j	122	7A	z	138	8A	è	154	9A
75	4B	K	91	5B]	107	6B	k	123	7B	{	139	8B	155	9B	
76	4C	L	92	5C	\	108	6C	l	124	7C	}	140	8C	156	9C	
77	4D	M	93	5D]	109	6D	m	125	7D	ı	141	8D	157	9D	
78	4E	N	94	5E	^	110	6E	n	126	7E	~	142	8E	158	9E	
79	4F	O	95	5F	_	111	6F	o	127	7F	/	143	8F	159	9F	

D	H	Ch												
160	A0	á	176	B0		192	C0		208	D0		224	E0	
161	A1	í	177	B1		193	C1		209	D1		225	E1	
162	A2	ó	178	B2		194	C2		210	D2		226	E2	
163	A3	ú	179	B3		195	C3		211	D3		227	E3	
164	A4		180	B4		196	C4		212	D4		228	E4	
165	A5	à	181	B5		197	C5		213	D5		229	E5	
166	A6	ả	182	B6		198	C6		214	D6		230	E6	
167	A7		183	B7		199	C7		215	D7		231	E7	
168	A8		184	B8		200	C8		216	D8		232	E8	
169	A9		185	B9		201	C9		217	D9		233	E9	
170	AA		186	BA		202	CA		218	DA		234	EA	
171	AB		187	BB		203	CB		219	DB		235	EB	
172	AC		188	BC		204	CC		220	DC		236	EC	
173	AD		189	BD		205	CD		221	DD		237	ED	
174	AE		190	BE		206	CE		222	DE		238	EE	
175	AF		191	BF		207	CF		223	DF		239	EF	
												255	FF	

Lưu ý một số ký tự quy ước trong bảng mã:

D (Decimal) : mã ASCII của ký tự ở dạng thập phân.

H (Hexa) : mã ASCII của ký tự ở dạng thập lục phân.

Chr (Character) : ký tự được hiển thị thấy trên màn hình, máy in.

Ctrl (Control) : mã tương ứng được tạo ra khi ấn phím Control với phím quy định.

Mem (Memory) : từ gợi nhớ tính năng ký tự.

1.2.5.4 Bộ mã Unicode (Universal Code)

Bảng mã 8-bit như ASCII hay EBCDIC với 256 giá trị không thể đủ chỗ để mã các ký tự của các ngôn ngữ dùng chữ tượng hình như tiếng Hán, tiếng Nhật, Hàn Quốc, kể cả tiếng Việt của chúng ta... Từ trước đến nay đã có nhiều giải pháp khác nhau để mã hóa các ký tự của các ngôn ngữ này trên máy vi tính, tuy nhiên những giải pháp này thường dùng kỹ thuật tổ hợp hoặc các chuỗi ký tự điều khiển (ESC) khá phức tạp và quan trọng hơn cả là các giải pháp này không tương thích với nhau, dẫn tới việc sử dụng đồng thời các ngôn ngữ trong cùng một văn bản và trong cùng một font chữ thường không thể hoặc thực hiện rất khó khăn.

Unicode ra đời là nhằm khắc phục các nhược điểm nói trên và nhằm xây dựng một bộ mã chuẩn dùng chung cho hầu hết ngôn ngữ của các dân tộc trên thế giới.

Tổ chức Unicode được thành lập vào năm 1991 như một tổ chức phi lợi nhuận nhằm phát triển chuẩn Unicode, các thành viên của tổ chức này bao gồm các công ty hàng đầu của thế giới trong lĩnh vực phần mềm như Adobe, Aldus, Borland, Digital, GO, IBM, HP, Lotus, Metaphor, Microsoft, NeXT, Novell, Sun, Symantec, Taligent, Unisys, WordPerfect ...

Unicode là bộ mã ký tự 16-bit, tương thích hoàn toàn với chuẩn quốc tế ISO/IEC 10646-1 được đưa ra năm 1993. Với 65.536 ký tự ($= 2^{16}$), Unicode hầu như có thể mã hóa hầu hết các bộ mẫu tự của các ngôn ngữ trên thế giới. Ngoài ra với cơ chế mở rộng UTF-16, Unicode và chuẩn ISO 10646 còn cho phép mã hóa hơn một triệu ký tự mà không cần phải dùng đến mã điều khiển Escape.

Chuẩn Unicode mô tả các ký tự ngôn ngữ, các dấu chấm câu, dấu phụ, ký hiệu toán học, các dấu mũi tên và các ký hiệu Dingbats. Hiện nay Unicode đã định nghĩa khoảng 39 000 ký tự. Còn khoảng 18.000 ký tự sẽ được định nghĩa trong tương lai gần, 917 504 ký tự có thể được định nghĩa nếu dùng cơ chế mở rộng UTF-16. 6.400 chỗ được dành ra cho các hàng sử dụng với các mục đích riêng của mình. UTF-16 cũng dành ra 131.072 ký tự để dành cho những mục đích dùng riêng.

Các chuẩn mã hóa ký tự không chỉ định nghĩa các mã của các ký tự, giá trị số, và vị trí của các ký tự mà còn định nghĩa cả cách biểu diễn các mã ký tự dưới dạng bit. Unicode và ISO-10646 quy định 2 cơ chế, khuôn dạng chuyển đổi là UTF-8 và UTF-16.

Bộ mã Unicode có một số đặc điểm như sau:

- Mỗi ký tự trong bảng mã Unicode đều có độ dài cố định là 16 bit, nhờ đó việc xử lý các chuỗi ký tự Unicode rất đơn giản, không phức tạp như các giải pháp dùng chuỗi ký tự điều khiển, đòi hỏi những thuật toán tương đối phức tạp để nhận diện ký tự trong một chuỗi các byte. Trong khi đó với Unicode, mỗi ký tự có độ dài đúng 2 byte nên có thể định vị rất dễ dàng các vị trí của ký tự trong chuỗi byte cho trước.

- Unicode tránh đến mức tối đa việc định nghĩa dư thừa, trùng lặp. Ví dụ ký tự ‘é’ chỉ có một mã duy nhất dùng chung cho cả ngôn ngữ tiếng Việt, tiếng Czech,cũng chính vì thế nên hệ thống chữ Việt có các mã nằm rải rác ở nhiều ví trí không liền nhau. Tiếng Hán, Nhật và Hàn có khoảng 10 nghìn ký tự trùng nhau nên chúng được dùng chung cho cả 3, tuy nhiên trong Unicode vẫn có các vùng riêng để định nghĩa những ký tự đặc thù của 3 ngôn ngữ này.
- Unicode về cơ bản không quy định việc bố trí các ký tự theo quy định sắp xếp của các ngôn ngữ, điều này cũng là hệ quả của việc tránh định nghĩa các ký tự dư thừa do phải tận dụng các ký tự dùng chung nên không thể bố trí các ký tự theo từng vùng riêng cho từng ngôn ngữ. Hơn nữa thực tế với nhiều ngôn ngữ, người ta phải dùng những thuật toán riêng để sắp xếp, chứ không thể sắp xếp theo thứ tự của chúng trong bảng chữ cái (tiếng Việt là một điển hình). Chính vì vậy bảng mã tiếng Việt trong Unicode có các ký tự Việt nằm rải rác ở nhiều nơi và không theo một trình tự sắp xếp nào.

Unicode đã được cài đặt trong các hệ điều hành Windows NT, Windows 9.x, Macintosh (Mac OS), Linux....

Windows NT sử dụng Unicode như là nền tảng trong hệ điều hành, các chuỗi ký tự được xử lý như là chuỗi Unicode, Resource, tên File trong NTFS cũng là Unicode. Tuy nhiên, để tăng tính tương thích, Windows NT vẫn có các hàm API để xử lý ký tự mã 8-bit. Windows NT 5.0 hỗ trợ hơn 100 ngôn ngữ khác nhau trong đó có cả tiếng Việt.

Ngược lại, Windows 9.x không lấy Unicode làm nền tảng nội tại trong hệ điều hành, tuy nhiên Win9.x lại có một số hàm hỗ trợ cho việc xử lý và hiển thị mã Unicode.

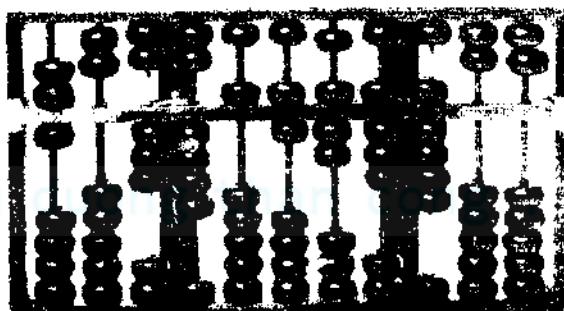
Muốn sử dụng được Unicode cần phải có những phần mềm hỗ trợ hiển thị hoặc cho phép gõ ký tự theo chuẩn Unicode, ngoài ra cũng cần phải có Font chữ Unicode được cài đặt trong hệ thống.

Hiện nay trong môi trường Windows, bộ MS Office 200x (Word, Excel, Powerpoint...) hỗ trợ rất tốt bộ mã Unicode. Trong môi trường mạng Internet, Expolore x.0 cũng cho phép hiển thị các trang Web được thiết kế theo chuẩn Unicode.

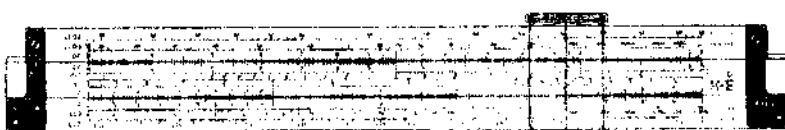
1.3 LỊCH SỬ PHÁT TRIỂN CỦA MÁY TÍNH

Có hai khái niệm quan trọng trong lịch sử tính toán của con người: cơ giới hóa các phép toán số học và khái niệm về chương trình được lưu trữ để điều khiển tự động quá trình tính toán.

Một trong các ví dụ của khái niệm đầu tiên là bàn tính (Hình 1.1) và thước kéo (Hình 1.2). Người Trung Hoa từ ba tới bốn ngàn năm trước đã biết phát minh ra bàn tính để thực hiện các phép tính trong công việc hằng ngày của họ, và ngày nay chúng vẫn còn được sử dụng. Ở châu Âu nhà toán học **John Napier** (1550 - 1617) đã thiết kế ra thiết bị Khung Napier (Hình 1.3) làm cơ sở để phát minh ra thước kéo (Hình 1.2) có thể thực hiện các phép toán số học và logarit.



Hình 1.1 Bàn tính



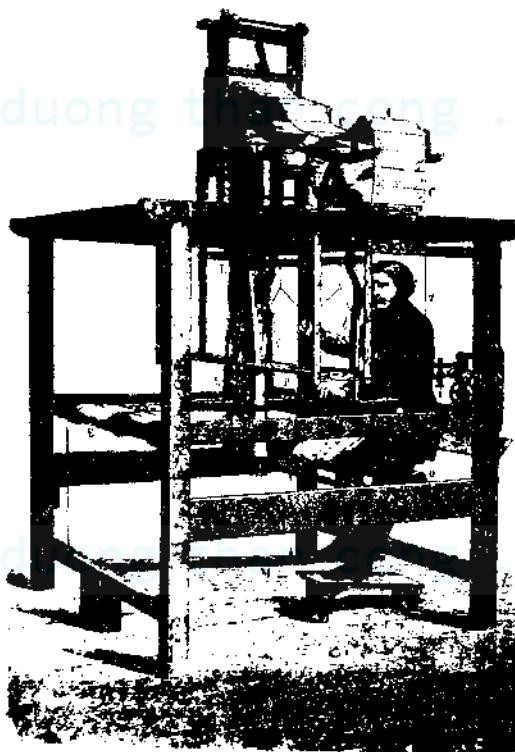
Hình 1.2 Khung Napier



Hình 1.3 Thước kéo

Vào năm 1642 nhà toán học trẻ người Pháp **Blaise Pascal** (1623-1662) đã sáng chế ra một trong những máy cộng cơ khí đầu tiên. Thiết bị này sử dụng hệ thống răng và bánh răng tương tự như trong đồng hồ đo kilomet và một số dụng cụ tính toán hiện đại khác. Thiết bị cộng này của Pascal có thể thực hiện được cả phép cộng và trừ, và ông dùng nó để tính thuế. Dù đã xuất hiện từ rất sớm, nhưng máy cộng của Pascal không được sử dụng rộng rãi do không đủ độ chính xác để dùng trong thực tế.

Thập niên 1670 nhà toán học Đức **Gottfried Wilhelm von Leibniz** (1646-1716) chế tạo ra một máy có thiết kế tương tự như thiết kế của Pascal nhưng tin cậy và chính xác hơn. Máy tính của Leibniz có thể thực hiện cả bốn phép toán cơ bản là cộng, trừ, nhân và chia. Sau đó cũng có nhiều thiết bị tính toán cơ khác nhưng chúng đều theo nguyên tắc của Pascal và Leibniz.

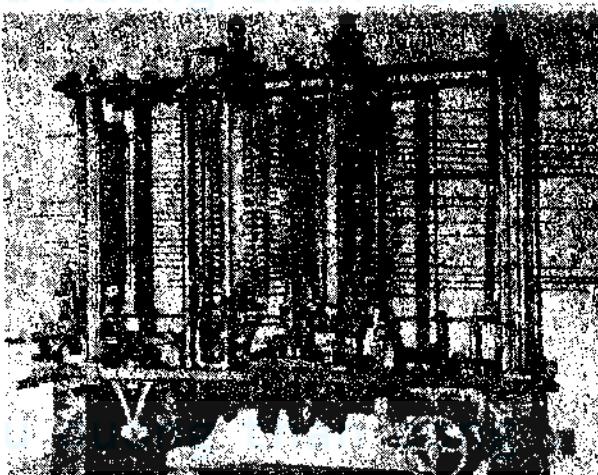


Hình 1.4 Máy dệt

Khái niệm tính toán thứ hai là xử lý việc tính toán theo chương trình điều khiển đã được lưu trữ trước. Một ví dụ về thiết bị này là máy dệt (Hình 1.4) được phát minh bởi một người Pháp tên **Joseph**

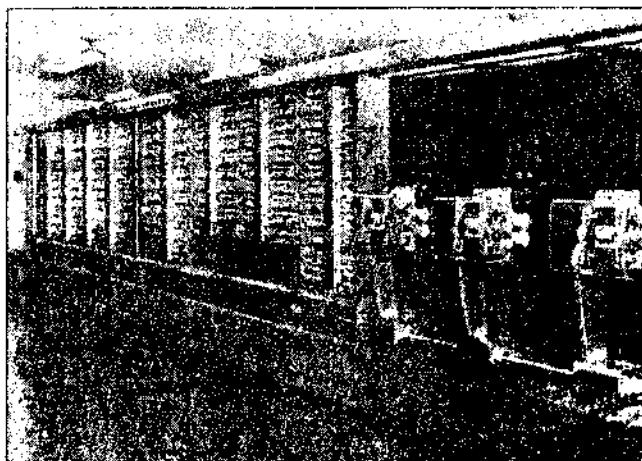
Marie Jacquard (1752-1834). Thiết bị này được giới thiệu trong một cuộc triển lãm ở Paris vào năm 1801, nó sử dụng các thẻ kim loại đục lỗ để định vị trí cho quá trình “dệt”. Tập hợp nhiều thẻ này tạo nên chương trình để điều khiển máy dệt. Trong vòng một thập niên, hơn 11.000 máy dệt theo chương trình như thế này đã được đưa vào sử dụng trong các nhà máy dệt của Pháp, và nó đã gây ra vấn đề về việc làm cho con người khi việc tự động hóa được sử dụng. Ngày nay máy dệt Jacquard vẫn còn được sử dụng và đã có những cải tiến như băng từ đã thay thế cho các thẻ đục lỗ.

Cả hai khái niệm, tính toán được cơ giới hóa và điều khiển theo chương trình đã lưu trữ, đã được nhà toán học Anh **Charles Babbage** (1792-1871) kết hợp lại. Kết quả là ông đã đưa ra một máy tính gọi là Máy Sai Biệt (Difference Engine) vào năm 1822. Máy này được thiết kế để tính đa thức theo những bảng toán học cho trước. Sau đó, ông đã đưa ra Máy Phân Tích (Analytical Engine) (Hình 1.5). Máy này có những bộ phận có chức năng đặc biệt có thể làm việc cùng nhau.



Hình 1.5 Máy phân tích

Ở Mỹ sự phát triển của các thiết bị tính toán diễn ra ở tốc độ rất nhanh. Những nhà tiên phong trong lĩnh vực này là Howard Aiken, J. P. Eckert, J. W. Mauchly và John von Neumann. Năm 1945 máy tính **Mark I** (Hình 1.6) ra đời. Đây là máy tính nổi tiếng nhất được chế tạo trước 1945, và có thể được xem như là sự thực hiện đầu tiên khái niệm máy phân tích của Babbage.



Hình 1.6 Máy Mark I

Bắt đầu từ giai đoạn này, máy tính phát triển theo những bậc gọi là các thế hệ máy tính. Tính tới thời điểm hiện nay (năm 2011), máy tính có thể được chia ra làm 4 thế hệ như sau:

Thế hệ 1: (1946 – 1958)

- Linh kiện chế tạo : đèn điện tử, máy ENIAC có hơn 18.000 đèn điện tử, 1.500 relay
- Bộ nhớ : từ tính, dung lượng nhỏ.
- Thời gian xử lý : mili giây, tốc độ tính toán nhanh hơn máy Mark I hàng ngàn lần
- Đặc điểm : ENIAC được đặt trong 1 phòng có kích thước xấp xỉ 10m × 20m, tiêu thụ rất nhiều năng lượng, hệ thống tỏa nhiệt rất tốn kém.
- Đại diện : ENIAC (Hình 1.7) và UNIVAC

Thế hệ 2: (1959 – 1963)

- Linh kiện chế tạo : bán dẫn, mạch in
- Bộ nhớ : xuyến ferit, dung lượng nhỏ.
- Thời gian xử lý : micro giây.
- Đặc điểm : độ tin cậy cao, kích thước nhỏ gọn hơn và công suất tiêu thụ ít hơn, tỏa nhiệt ít hơn.
- Đại diện : IBM 7090



Hình 1.7 Máy ENIAC

Thế hệ 3: (1964 – 1970)

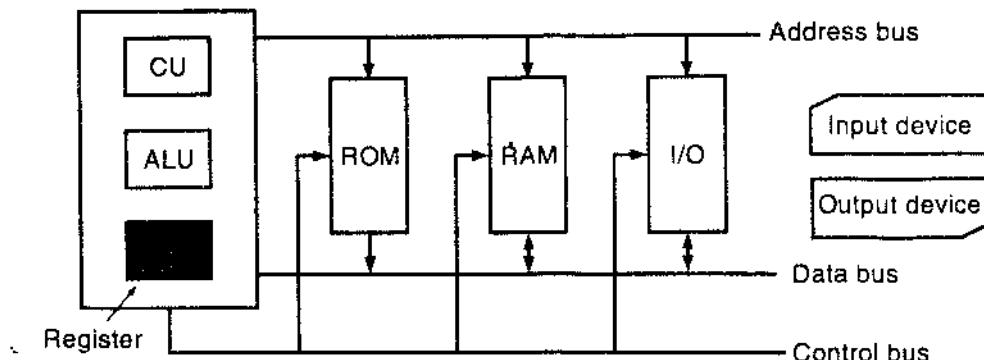
- Linh kiện chế tạo : mạch vi điện tử.
- Bộ nhớ : dây từ, xuyến ferit.
- Thời gian xử lý : micro giây đến nano giây.
- Đặc điểm : độ tin cậy cao, có sự hoàn thiện của hệ điều hành và ngôn ngữ lập trình, đặc biệt lập trình đa chương (multiprogramming) và có chia sẻ thời gian xử lý.
- Đại diện : IBM 360

Thế hệ 4: (1971 – nay)

- Linh kiện chế tạo : vi mạch cở lớn VLSI, bộ vi xử lý ra đời
- Bộ nhớ : màng mỏng, mos, cmos.
- Thời gian xử lý : micro giây đến nano giây.
- Đặc điểm : độ tin cậy cao, kích thước và công suất tiêu thụ nhỏ, ngôn ngữ lập trình và các hệ chuyên dụng phong phú đa dạng.
- Đại diện : máy tính cá nhân ra đời bên cạnh sự phát triển của dòng máy tính lớn.

1.4 CÁC THÀNH PHẦN CƠ BẢN CỦA MÁY TÍNH

Một cách tổng quát, bất kỳ máy tính nào cũng đều có các bộ phận cơ bản được biểu diễn qua sơ đồ khối sau (Hình 1.8).



Hình 1.8 Sơ đồ khái niệm tổng quát bên trong một máy tính

Trong sơ đồ này, máy tính sẽ gồm 4 thành phần cơ bản: đơn vị xử lý trung tâm (CPU), bộ nhớ chính (ROM, RAM), đơn vị xuất nhập (I/O) và các tuyến (bus).

1.4.1 Đơn vị xử lý trung tâm (CPU- Central Processing Unit)

Đây là bộ phận quan trọng nhất trong hệ thống máy tính, nó hoạt động theo một xung nhịp (xung clock) đã được cài sẵn trong máy tính. Nếu tần số hoạt động xung clock này của CPU càng lớn thì nó xử lý lệnh càng nhanh. Thuật ngữ CPU thường được gọi sử dụng tổng quát trong hệ thống máy tính từ máy tính lớn tới máy tính cá nhân, còn thuật ngữ bộ vi xử lý lại được sử dụng đối với máy tính cá nhân trong đó mọi thành phần bên trong CPU (CU, ALU và tập thanh ghi) đã được tích hợp trong một vi mạch.

CPU điều khiển mọi thao tác trong toàn hệ thống máy tính, thực hiện các thao tác số học và luận lý, lưu trữ và truy tìm lệnh và dữ liệu. CPU có ba thành phần là đơn vị điều khiển (CU), đơn vị số học luận lý (ALU) và các thanh ghi. Mỗi bộ phận đều có chức năng riêng.

Đơn vị điều khiển (CU - Control Unit) lấy lệnh từ bộ nhớ trong (ROM/RAM) phân tích và giải mã lệnh và đưa ra các tín hiệu điều khiển để điều khiển các bộ phận khác trong CPU và trong hệ máy tính hoạt động.

Đơn vị số học luận lý (ALU - *Arithmetic-Logic Unit*) thực hiện các phép toán số học hoặc logic. ALU thực hiện các phép toán với dữ liệu (*data*) trong các thanh ghi (*register*) mà đã được lấy từ bộ nhớ (*memory*), xử lý và sau đó đưa kết quả từ các thanh ghi về *memory*.

Tập các thanh ghi (*Register*) là bộ nhớ có dung lượng nhỏ nhưng tốc độ truy xuất cao, nằm bên trong CPU, có chức năng lưu trữ dữ liệu là toán hạng của ALU, hoặc lưu kết quả của phép toán mà ALU vừa xử lý xong, lưu địa chỉ ô nhớ.

Trong chương sau, chúng ta sẽ đi sâu vào việc phân tích quá trình thực hiện lệnh, trong đó các bộ phận trong CPU sẽ hoạt động theo từng chức năng cụ thể.

1.4.2 Bộ nhớ chính (ROM/RAM)

Bộ nhớ chính (*Main memory*) hay còn được gọi là bộ nhớ trong (*Internal memory*) hay bộ nhớ sơ cấp (*Primary memory*) của hệ thống máy tính là nơi dùng để lưu trữ thông tin, lệnh và dữ liệu của chương trình đang được thực thi.

Trong bộ nhớ chính có hai loại bộ nhớ là ROM và RAM. ROM (*Read Only Memory*) chứa thông tin cố định mà ta chỉ có thể đọc được dữ liệu, chứ không ghi được dữ liệu vào nó được, thường là chương trình điều khiển hay ký tự (được gọi là ROM BIOS – *Basic Input Output System*). Thông tin bên trong ROM sẽ không bị mất khi mất điện. Hiện nay, các hãng như AMI sản xuất chương trình cài đặt trong ROM còn cài thêm những chương trình tiện ích khác như tìm cấu hình đĩa cứng tự động, kiểm tra virus, cho phép cài đặt lại một số tham số của hệ thống, quy định quá trình khởi động máy, ...

RAM (*Random Access Memory*) là bộ nhớ ghi đọc ngẫu nhiên, theo nghĩa là ta có thể truy xuất thông tin ở RAM bất kỳ lúc nào và bất kỳ chỗ nào. Bộ nhớ này thường được sử dụng để lưu chương trình đang được CPU xử lý. Thông tin trong RAM sẽ bị mất đi khi mất điện.

Ngoài ra, trong máy tính còn có một loại RAM đặc biệt là CMOS-RAM, nó được sử dụng để lưu cấu hình của máy tính. Trong một hệ thống máy tính, có rất nhiều bộ phận và thiết bị khác nhau, khác nhau về tính năng sử dụng, khác nhau về chủng loại, khác nhau về khả năng hoạt động, ..., máy tính muốn hoạt động được cần phải biết tất cả các thiết bị này qua các thông số kỹ thuật, chẳng hạn đối

với đĩa cứng, máy tính cần phải biết số lượng đầu từ, số sector trên một track, số từ trụ, hay trong hệ thống có bao nhiêu ổ đĩa,... Tất cả các tham số này cần được khai báo và chứa trong CMOS trước khi máy tính hoạt động.

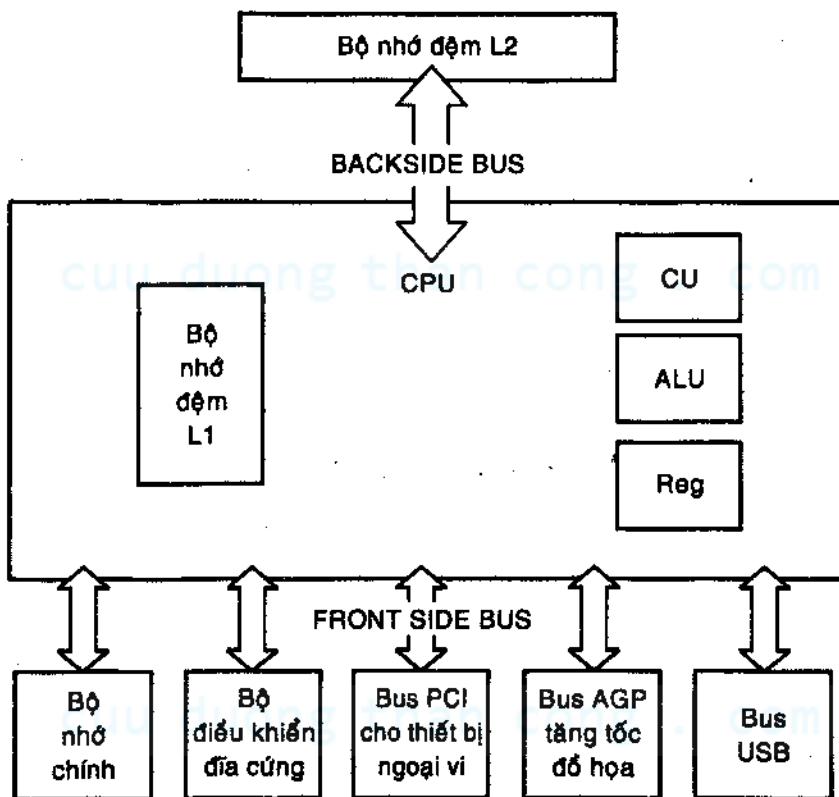
Có hai yếu tố xác định bộ nhớ là dung lượng bộ nhớ tính bằng byte và thời gian truy xuất tính bằng nano giây. Đối với máy tính cá nhân, bộ nhớ dung lượng vài chục tới vài trăm MB là bình thường. Bộ nhớ trong càng lớn, máy tính càng có khả năng lưu trữ nhiều chương trình hay chương trình càng lớn hơn, xử lý vấn đề phức tạp hơn. Thời gian truy xuất bộ nhớ được định nghĩa là thời gian từ lúc bộ nhớ được yêu cầu dữ liệu cho tới khi dữ liệu đó xuất hiện trên bus dữ liệu (*data bus*). Hiện nay, thời gian truy xuất nhanh nhất các vi mạch nhớ cở 40 ns.

Bộ nhớ có nhiều ô nhớ, mỗi ô nhớ có địa chỉ riêng không trùng với bất kỳ địa chỉ nào trong hệ thống. Trong hệ thống sẽ có những mạch thực hiện giải mã địa chỉ, để với một địa chỉ thì chỉ có một ô nhớ xác định tương ứng cần làm việc. Mỗi ô nhớ dài một byte, do đó nó có thể lưu giữ được một ký tự theo bảng mã ASCII.

Hiện nay, để tăng tốc độ làm việc của hệ thống, các kỹ sư thiết kế CPU của hãng Intel và ngay sau đó là AMD đưa ra một loại bộ nhớ đặc biệt gọi là bộ nhớ đệm (*cache memory*) vào khoảng giữa những năm 1990. Bộ nhớ đệm có chức năng lưu trữ những dữ liệu thường được hệ thống và chương trình dùng đến nhất. Các CPU Pentium II và Pentium Pro của Intel đều dùng bộ đệm ngoài chip. Bộ nhớ đệm này gần với CPU (cả về khoảng cách và tốc độ truy cập) hơn là bộ nhớ chính. Bus nối CPU với vùng đệm này là bus hai chiều, cho phép trao đổi dữ liệu với tốc độ của CPU.

Việc xuất hiện bộ nhớ đệm nằm ngoài chip CPU làm cho chi phí sản xuất máy tính tăng lên, vì vi mạch nhớ làm bộ nhớ đệm thường là RAM tĩnh rất đắt tiền, mặt khác lại chiếm chỗ trên bo mạch chính (main board), do đó gần đây các kỹ sư thiết kế đã thực hiện được một bước tiến trong giao tiếp giữa CPU và bộ đệm, đó là tích hợp bộ đệm vào trong chất nền bán dẫn của chính CPU. Bộ nhớ đệm nằm ngay trên chip CPU được gọi là bộ nhớ đệm L1 (Hình 1.9). Điều này cho phép thu nhỏ kích thước của phần xử lý trên bo chính, giảm chi phí đóng gói và cho phép các nhà thiết kế dùng loại RAM tĩnh có giá thành thấp hơn.

Tuy nhiên, bộ nhớ cache ngoài chip không hoàn toàn biến mất. Các bộ xử lý 400-500 MHz PowerPC G4 dùng trong các máy tính của Apple như Power Mac G4, máy tính xách tay của Cube và Titanium vẫn tiếp tục dùng cache ngoài chip, và nó được gọi là bộ nhớ đệm L2 (Hình 1.9). Khi đó các bus kết nối giữa cache ngoài và CPU gọi là back-side bus, để phân biệt với front-side bus nối CPU với bộ nhớ chính, và các bộ phận khác trong hệ vi xử lý. Bộ vi xử lý G4 dùng bộ nhớ đệm dung lượng 1 MB ngay trên bộ xử lý với backside bus 64 bit. Backside bus 64 bit này kết hợp với front-side bus 100 MHz tạo ra tốc độ truyền dữ liệu tối đa là 800 Mbit/s.



Hình 1.9 Sơ đồ khái niệm về sơ đồ khối của hệ vi xử lý có cache

1.4.3 Đơn vị xuất nhập và thiết bị ngoại vi

Đơn vị xuất nhập I/O (I/O Unit - Input/Output Unit) hay còn được gọi là đơn vị giao tiếp hay card giao tiếp là các thiết bị làm nhiệm vụ trung gian giao tiếp giữa CPU, bộ nhớ với các thiết bị ngoại vi. Máy tính hoạt động dựa vào các mạch số có các đáp ứng điện áp và dòng

điện, cũng như các tín hiệu điều khiển xác định. Một thiết bị khi muốn gắn vào hệ thống để CPU hiểu và làm việc thì cần phải có một bộ phận trung gian chuyển tín hiệu của thiết bị này với mức điện áp, dòng điện và các tín hiệu điều khiển sang mức điện áp, dòng điện và tín hiệu điều khiển của hệ thống. Do đó có thể nói là mọi thiết bị đều có thể được nối với máy tính để máy tính điều khiển tự động sự hoạt động của thiết bị đó theo chương trình đã viết miễn là ta có thể thiết kế được đơn vị xuất nhập cho nó. Ví dụ như màn hình hay bàn phím, cần phải các bộ phận thực hiện nhiệm vụ giao tiếp là card màn hình hay vi mạch thực hiện giao tiếp với bàn phím trên bảm mạch chính.

Thiết bị ngoại vi là các thiết bị làm nhiệm vụ giao tiếp giữa hệ vi xử lý và môi trường bên ngoài, được chia làm hai nhóm: thiết bị liên lạc và bộ nhớ phụ. Thiết bị liên lạc thực hiện việc trao đổi thông tin giữa hệ vi xử lý nói chung và máy tính nói riêng với người sử dụng như màn hình, máy in, bàn phím, modem ...; hay với một hệ cần được điều khiển nào đó như độ phun khí tạo sức đẩy của phi thuyền con thoi hay bộ phận cảm nhận độ ẩm trong lò lên men,

Bộ nhớ phụ (*Sub-memory*) còn được gọi là bộ nhớ ngoài (*External Memory*) hay bộ nhớ thứ cấp (*Secondary Memory*) là thiết bị lưu trữ thông tin như đĩa từ, băng từ, đĩa quang... Tốc độ truy xuất dữ liệu của thiết bị nhớ ngoài chậm hơn bộ nhớ trong.

Như vậy, nếu xem máy tính là một vật thể, con người là một vật thể khác thì để con người làm việc được với máy tính, điều cần thiết là phải có thiết bị giao tiếp giữa NGƯỜI - MÁY để con người có thể ra lệnh cho máy và máy hiểu được cũng như báo cho con người kết quả mà nó đã làm.

1.4.4 Các tuyến

Tuyến (*bus*) là tập các đường dây vật lý truyền các tín hiệu trong hệ thống máy tính. Các đường dây này thường được tập trung theo chức năng xác định nào đó, mỗi dây sẽ truyền một bit. Điện áp và dòng điện trên các dây tín hiệu này theo chuẩn, nếu theo quy định dương thì mức điện áp cao sẽ tương ứng với bit 1, và mức điện áp thấp tương ứng với bit 0; nếu theo quy ước âm thì ngược lại.

Trong máy tính có ba loại bus được phân loại theo chức năng là bus dữ liệu, bus địa chỉ và bus điều khiển. Bus dữ liệu truyền dữ liệu giữa

các bộ phận trong hệ với nhau, từ CPU vào ra bộ nhớ, từ bộ nhớ vào ra thiết bị ngoại vi, ... Bus điều khiển truyền các tín hiệu điều khiển từ CPU ra cho các bộ phận trong hệ thống như bộ nhớ, đơn vị xuất nhập, ..., đồng thời nhận các tín hiệu yêu cầu từ các thiết bị ngoại vi về CPU để yêu cầu một vấn đề gì đó như yêu cầu ngắt, yêu cầu truyền DMA, Mỗi bộ phận trong hệ thống máy tính đều có một địa chỉ xác định cho riêng nó, địa chỉ này được đưa ra từ CPU qua bus địa chỉ.

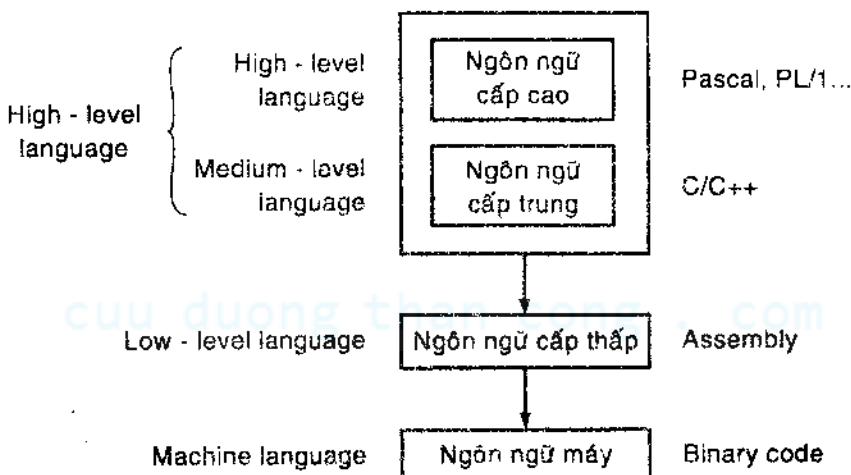
1.5 PHẦN MỀM

1.5.1 Định nghĩa

Phần mềm là toàn bộ các thủ tục đưa vào máy tính để máy thực hiện các chức năng xử lý theo mục tiêu của người lập trình.

1.5.2 Ngôn ngữ cho máy tính

Ngôn ngữ dùng cho máy tính còn gọi là ngôn ngữ lập trình, là toàn bộ các lệnh, các dữ liệu, các thủ tục... được kết hợp lại với nhau theo nguyên tắc kết cấu mã tin và hệ lệnh mà ta gọi là cú pháp (*syntax*), đưa vào máy tính để máy thực hiện các chức năng xử lý theo mục tiêu của người lập trình.



Hình 1.10 Các cấp của ngôn ngữ lập trình

Có thể chia ngôn ngữ máy tính ra làm ba cấp một cách tổng quát: cấp cao, cấp thấp và cấp máy (Hình 1.10). Cấp cao nhất là ngôn ngữ tự nhiên của con người. Các ngôn ngữ cấp cao được thiết kế cho

phép người viết chương trình không cần phải hiểu hoạt động bên trong của máy tính. Các lệnh của ngôn ngữ cấp cao thường sử dụng các từ tiếng Anh, các phép toán theo các ký hiệu toán học thông thường do đó rất dễ sử dụng. Ví dụ cho các ngôn ngữ này là Pascal, Basic, Java, C/C++, ... Người ta thường có xu hướng chia ngôn ngữ cấp cao ra làm hai cấp nhỏ là cấp cao và cấp trung gian vì đặc điểm của ngôn ngữ C cho phép lập trình cấp thấp hay mã máy trong nó. Đặc điểm của ngôn ngữ cấp cao là gần với con người, do đó chương trình viết bằng ngôn ngữ cấp cao có tính khả chuyen, tức có thể chạy trên nhiều hệ máy khác nhau, nhiều hệ điều hành khác nhau.

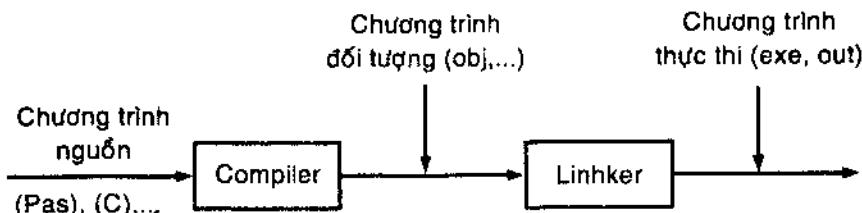
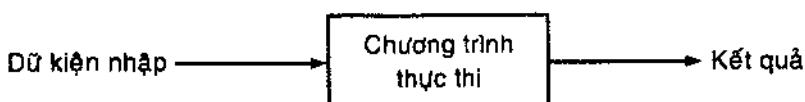
Ngôn ngữ cấp thấp là ngôn ngữ trong đó mỗi lệnh tương ứng với một lệnh của ngôn ngữ máy và tương ứng với tập lệnh của CPU. Các lệnh và phép toán của ngôn ngữ cấp thấp thường có tính gọi nhớ (*mnemonic*) tới một từ tiếng Anh nào đó. Mỗi hãng thiết kế CPU (như Intel hay Motorola) khi thiết kế ra một CPU mới đều quy định tập lệnh cho CPU đó, khi mua CPU thì ta cũng sẽ có luôn quyển sách quy định này. Hợp ngữ (*Assembly language*) là một ví dụ cho ngôn ngữ này.

Ngôn ngữ máy là ngôn ngữ trong đó mọi lệnh đều được viết dưới dạng mã nhị phân. Chương trình ở dạng này máy có thể thực thi được ngay.

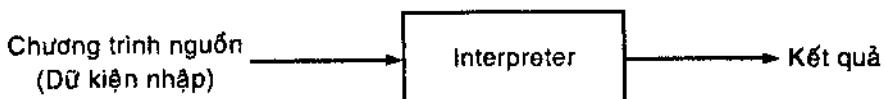
Chương trình viết bằng ngôn ngữ cấp thấp và ngôn ngữ máy chỉ có thể chạy trên một hệ máy xác định nào đó mà thôi vì mỗi họ CPU chỉ có thể hiểu được mã máy mà hãng đã quy định cho nó.

Chương trình viết dưới dạng văn bản (ngôn ngữ cấp cao hoặc cấp thấp) gọi là chương trình nguồn (*source*). Chương trình này máy không hiểu trực tiếp được. Vì thế muốn đưa vào thực hiện trên máy, chương trình nguồn phải được dịch sang ngôn ngữ máy (gọi là chương trình thực thi) bởi các chương trình dịch hay bộ dịch (*translator*). Có hai loại bộ dịch: chương trình biên dịch và chương trình diễn dịch hay thông dịch.

Chương trình biên dịch (*Compiler*) (Hình 1.11a, 1.11b) sẽ dịch chương trình nguồn một lần, thống kê và báo tất cả lỗi một lúc nếu có. Cách dịch này khó trong việc thống kê và sửa lỗi do toàn bộ lỗi được báo một lần, nhưng lại tiết kiệm nhiều thời gian khi chương trình đã được dịch và liên kết vì bản thân chương trình được chạy, không phải compiler được chạy.

**Hình 1.11a** Biên dịch chương trình**Hình 1.11b** Thực thi một chương trình đã được dịch

Chương trình diễn dịch hay thông dịch (*Interpreter*) (Hình 1.11c) xem từng lệnh của chương trình nguồn là dữ kiện để thực thi. Thực hiện xong một lệnh nếu có lỗi cú pháp (*Syntax*) thì báo, còn không thì thực thi lệnh tiếp theo. Cách dịch này đơn giản dùng để lập và sửa chữa chương trình rất tiện lợi nhưng tốn nhiều thời gian vì bản thân Interpreter được chạy, không phải chương trình nguồn được chạy.

**Hình 1.11c** Thực thi một chương trình của Interpreter

1.5.3 Chương trình

Chương trình là tập hợp các lệnh được sắp xếp theo một trình tự hợp logic để giải quyết một vấn đề nào đó trên máy tính. Sản phẩm của chương trình đã được dịch gọi là phần mềm (*software*).

Có hai loại chương trình: chương trình phục vụ và chương trình ứng dụng.

Chương trình phục vụ, còn gọi là chương trình hệ thống, là chương trình bảo đảm cho máy tính thực hiện các chức năng cơ bản, như điều khiển sự hoạt động của CPU, trao đổi giữa trung tâm (CPU) với ngoại vi, thực hiện các thủ tục xử lý dịch các ngôn ngữ ra mã máy, các thao tác điều hành cơ bản như sao chép đĩa, tập tin, các

dịch vụ truyền nhận dữ liệu ... Ví dụ các thủ tục trong ROM-BIOS và các ngắt của hệ điều hành DOS là các ví dụ điển hình cho chương trình phục vụ. Ngôn ngữ sử dụng để viết các chương trình phục vụ là ngôn ngữ cấp thấp hay ngôn ngữ C/C++.

Chương trình ứng dụng là chương trình do người sử dụng khai thác sử dụng máy lập ra để giải quyết các yêu cầu xử lý cụ thể. Ngôn ngữ sử dụng chủ yếu là ngôn ngữ bậc cao như Pascal, C, Java, ... Ví dụ chương trình giải phương trình bậc hai, chương trình tính độ bền sắt thép, các phần mềm trò chơi, ...

1.6 CÁC CẤP CHUYỂN ĐỔI

Hai mục trên đã trình bày sơ lược về kiến trúc tổng quan của máy tính và khái niệm về phần mềm. Tuy nhiên, làm sao một chương trình được viết bằng ngôn ngữ cấp cao như C/C++, Pascal lại có thể chạy được trên kiến trúc phần cứng như vậy? Hay nói cách khác, làm thế nào các lệnh của chương trình cấp cao điều khiển được phần cứng, gồm các mạch điện tử, hay nói rõ hơn là điều khiển các “điện tử” chạy theo ý muốn? Có thể hình dung, việc thực hiện lệnh này là một quá trình khá dài dòng và phức tạp, từ lúc ta gặp vấn đề cần giải quyết, xác định mục tiêu cụ thể, đến thiết kế giải thuật, rồi quyết định ngôn ngữ, dịch và thực thi. Nhưng cụ thể hơn quá trình này phải qua các cấp như hình 1.12 sau.

Vấn đề (Problems)

Giải thuật (Algorithms)

Ngôn ngữ (Language)

Kiến trúc (ISA) máy (Machine Architecture)

Vì kiến trúc (Microarchitecture)

Mạch (Circuits)

Thiết bị (Devices)

Hình 1.12 Các cấp chuyển đổi

Ở mỗi cấp chúng ta đều có sự chọn lựa, nếu chúng ta bỏ qua bất kỳ cấp nào, không quan tâm tới nó khi lập trình, thì có thể chúng ta sẽ không thể sử dụng được tốt nhất khả năng của máy tính.

1.6.1 Đặt vấn đề

Để viết được một chương trình, trước tiên chúng ta phải mô tả được vấn đề cần giải quyết bằng ngôn ngữ tự nhiên như tiếng Việt, tiếng Anh, Tuy nhiên, nếu chúng ta viết các lệnh đưa vào máy tính bằng các ngôn ngữ tự nhiên này thì không ổn, vì ngôn ngữ tự nhiên có rất nhiều sự không rõ ràng. Khi nghe một câu nói mà nếu không biết âm điệu hay ngữ cảnh thì chúng ta nhiều khi không hiểu câu đó có ý gì. Ví dụ, khi nghe câu: "Anh làm việc này quá tốt!", đó là một câu khen, hay một câu nói mỉa mai? Quả là ta không biết được ý nghĩa của nó khi ta chỉ biết nội dung như thế.

Và như vậy, không thể dùng ngôn ngữ tự nhiên để viết chương trình điều khiển các điện tử chạy được, vì khi đó nó sẽ tới đâu, đường nào với nhiều cách hiểu như vậy.

1.6.2 Giải thuật

Với sự không rõ ràng của ngôn ngữ tự nhiên như vậy, nên ta cần chuyển các lệnh mô tả vấn đề sang dạng giải thuật, và do đó đã đặc tính hoá cụ thể các lệnh, làm chúng xác định với thao tác trong máy tính. Một giải thuật là một thủ tục theo trình tự từng bước từ lúc bắt đầu cho tới lúc kết thúc. Mỗi bước đều được quy định trạng thái làm việc và được máy tính thực thi.

Mỗi vấn đề có thể có nhiều giải thuật giải khác nhau. Có giải thuật chỉ có ít bước thực hiện lệnh, có giải thuật nhiều bước hơn, nhưng thời gian thực thi ngắn hơn, có giải thuật lại sử dụng ít bộ nhớ hơn, Như vậy với một vấn đề, người viết có nhiều tiêu chí để thiết kế giải thuật sao cho tối ưu và thích hợp với yêu cầu của mình.

Thường có hai dạng giải thuật: lưu đồ (*flow chart*), và mã giả (*pseudo code*). Đặc giả có thể tham khảo thêm trong các tài liệu chuyên về thuật toán.

1.6.3 Chương trình

Bước kế tiếp là chuyển giải thuật thành chương trình máy tính bằng một trong các ngôn ngữ lập trình đã biết. Các ngôn ngữ lập trình là các ngôn ngữ thuộc về máy, chúng khác với ngôn ngữ tự nhiên là mỗi lệnh, mỗi câu đều có ý nghĩa bắt máy tính thực thi một công việc cụ thể, chứ không phải là các câu nói dài dòng nhưng nhiều khi ít ý nghĩa.

Đến nay có hơn 1000 ngôn ngữ lập trình, một vài cái trong chúng được thiết kế cho những ứng dụng riêng biệt, như **Fortran** chuyên giải các phép tính khoa học, **COBOL** lại chuyên giải các vấn đề thương mại, ngôn ngữ lập trình C được thiết kế chuyên xử lý các cấu trúc phần cứng cấp thấp,...

Có hai loại ngôn ngữ thuộc máy là ngôn ngữ cấp cao và ngôn ngữ cấp thấp. Các bộ dịch sẽ chuyển chương trình được viết ở ngôn ngữ tương ứng ra dạng kiến trúc tập lệnh của máy tính tương ứng đang muốn chạy chương trình.

1.6.4 Kiến trúc ISA

Chương trình ở ngôn ngữ cấp cao tiếp tục được dịch sang tập lệnh của một máy tính có kiến trúc đặc biệt mà từ đó nó sẽ được sử dụng để thực thi công việc của chương trình ở bước kế tiếp. Kiến trúc tập lệnh (*Instruction Set Architecture*) là sự quy định hoàn chỉnh cho sự tương tác giữa chương trình đã được viết và phần cứng máy tính để thực thi tác vụ của các chương trình.

Một kiến trúc tập lệnh cụ thể sẽ quy định tập lệnh, với những thao tác trên toán hạng với kiểu dữ liệu, khả năng định vị toán hạng trong bộ nhớ. Tùy vào số thao tác lệnh, kiểu dữ liệu, và các thao tác định vị bộ nhớ mà có nhiều kiến trúc tập lệnh khác nhau. Có những kiến trúc tập lệnh chỉ có vài thao tác, một, hai kiểu dữ liệu, nhưng lại cũng có những kiến trúc khác lên tới hàng trăm thao tác, hàng tá kiểu dữ liệu. Một vài kiến trúc tập lệnh chỉ có một hoặc hai trạng thái định vị bộ nhớ, trong khi những cái khác lại lên tới hơn 20. Kiến trúc tập lệnh x86 được dùng trong các PC có hơn 100 thao tác, hơn chục kiểu dữ liệu, và hơn hai chục trạng thái định vị bộ nhớ.

Ngày nay có nhiều kiến trúc tập lệnh đang được sử dụng, phổ biến nhất là ISA x86, được Công ty Intel thiết kế vào 1979, và hiện nay cũng đang được AMD và nhiều công ty khác sử dụng và phát triển. Những kiến trúc tập lệnh khác là Power PC (IBM và Motorola), PA-RISC (*Hewlett Packard*), và SPARC (*Sun Microsystems*).

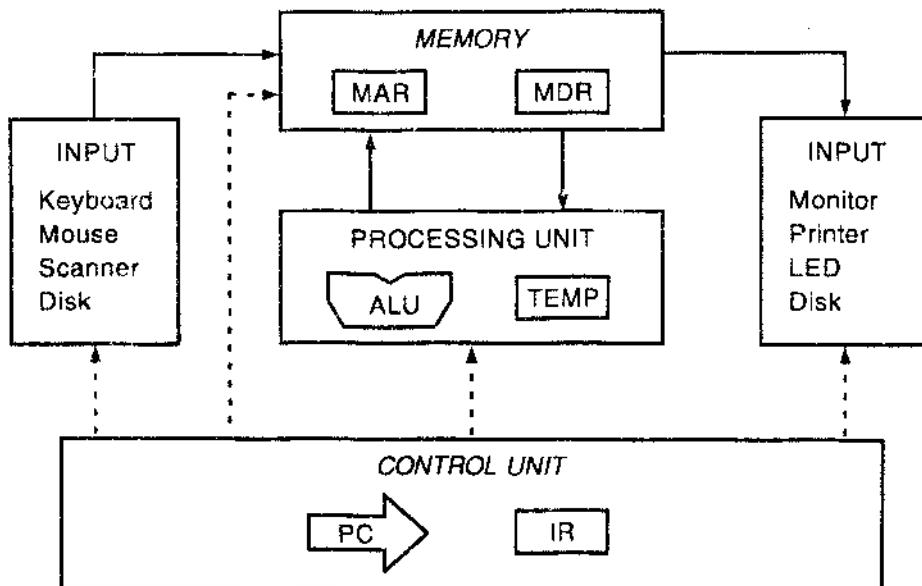
Việc dịch từ ngôn ngữ cấp cao, như C, sang kiến trúc tập lệnh của máy tính mà ở đó chương trình sẽ được thực thi, như kiến trúc x86, do **bộ dịch** đảm trách. Mỗi lệnh của ngôn ngữ cấp cao thường được dịch ra thành nhiều lệnh trong ISA. Với hợp ngữ, mỗi lệnh của nó chỉ được dịch ra thành một lệnh trong ISA mà thôi. Lập trình trên kiến trúc máy nào thì cần bộ dịch ngôn ngữ tương ứng với kiến trúc đó.

Như vậy, mỗi kiến trúc tập lệnh cho chúng ta khả năng hiểu kiến trúc và tổ chức máy tính qua các quy định về các thao tác lệnh, kiểu dữ liệu, các trạng thái định vị bộ nhớ, mà chúng không lệ thuộc vào tập lệnh thật lệ thuộc vào từng đời CPU, mà thường gọi là hợp ngữ. Chương 4 và 5 của giáo trình này sẽ trình bày cụ thể vấn đề này trên cơ sở khảo sát mô hình máy tính LC-3. Ngoài ra, độc giả có thể tham khảo sách “**Introduction to computing systems**”, ấn bản lần 2, của tác giả Yale N. Patt, và Sanjay J. Patel, nhà xuất bản Mc Graw Hill, 2005, hoặc “**Logic and Computer Design Fundamentals**”, ấn bản lần 3, của tác giả M. Morris Mano và Charles R. Kime, nhà xuất bản Prentice Hall, 2004 để biết thêm chi tiết.

1.6.5 Vi kiến trúc

Bước kế tiếp là chuyển lệnh ở kiến trúc tập lệnh sang dạng thực hiện. Việc tổ chức chi tiết của quá trình thực hiện lệnh này được gọi vi kiến trúc (*Microarchitecture*). Có thể hiểu, vi kiến trúc như là một máy tính “ảo” mà ngôn ngữ máy lúc này là kiến trúc tập lệnh tương ứng.

Có rất nhiều loại chip vi xử lý, mỗi chip đều có vi kiến trúc khác nhau, điều này kéo theo kiến trúc tập lệnh của từng loại vi xử lý là khác nhau. Tuy nhiên trên cùng một họ, như dòng x86, các vi kiến trúc đều được thiết kế theo cùng một dạng: mô hình máy Von Neumann (Hình 1.13). Hiện nay, máy tính ảo LC-3, mà trước đây là LC-2, có vi kiến trúc sử dụng kiến trúc tập lệnh cho các bước dịch của quá trình giao tiếp giữa phần mềm và phần cứng của máy tính.



Hình 1.13 Mô hình máy Von Neumann

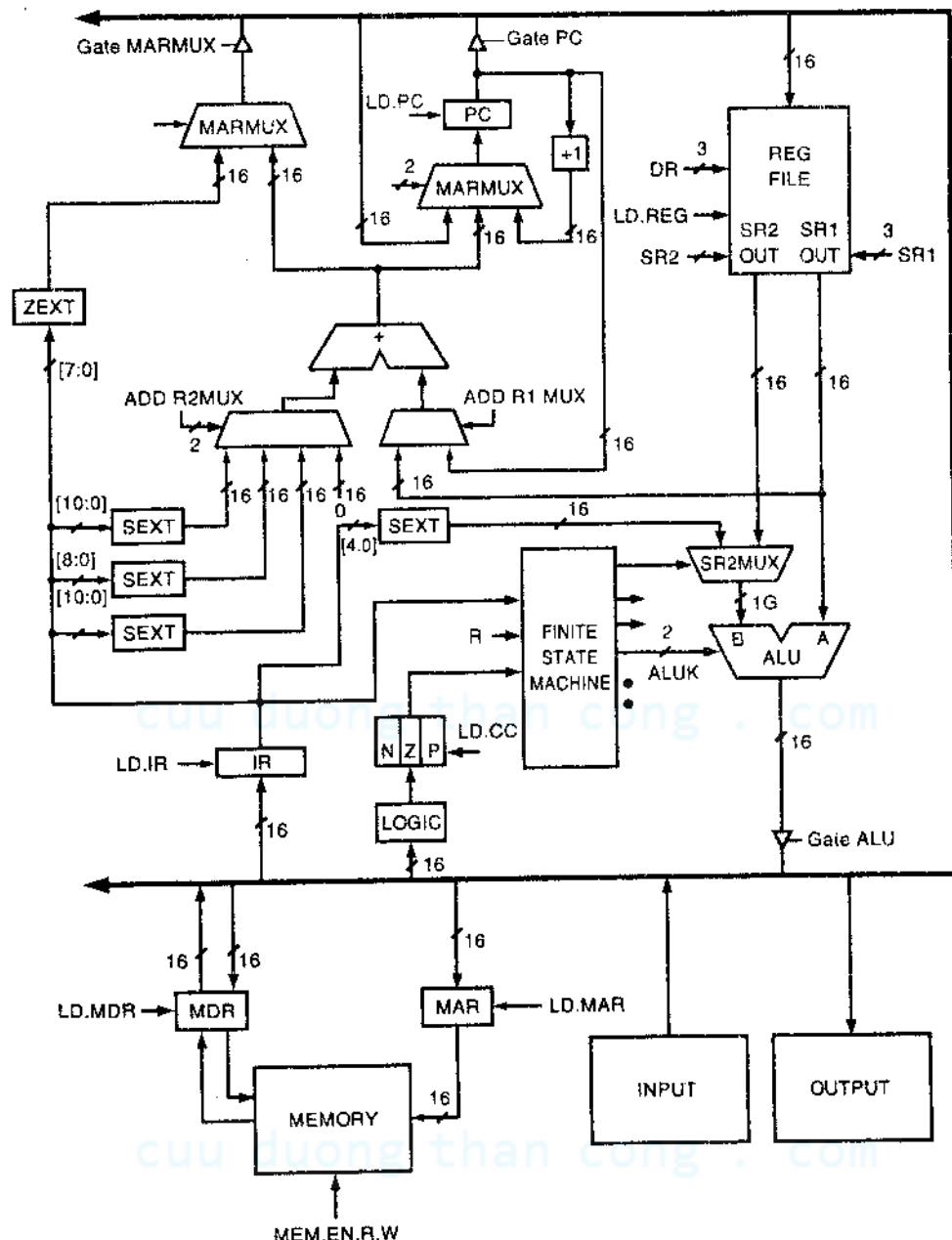
Chương 3 và 4 sẽ nêu cụ thể về vi kiến trúc và ISA của LC-3. Độc giả có thể download bộ dịch LC-3 và tập lệnh của nó từ mạng để lập trình, tìm hiểu cấp ISA vốn rất quan trọng trong máy tính.

Mỗi vi kiến trúc đều được thể hiện bằng một đường truyền dữ liệu (*data path*) với sự kết hợp các mạch luận lý đơn giản ở cấp mạch dưới đây.

1.6.6 Mạch luận lý

Bước này sẽ hiện thực mỗi phần tử của vi kiến trúc thành những mạch luận lý đơn giản. Ở bước này, các nhà thiết kế sẽ phải lựa chọn để máy tính khi được thiết kế phải được thị trường chấp nhận, tức phải có sự phù hợp về giá cả và khả năng xử lý của máy tính.

Hình 1.14 trình bày một ví dụ về sự kết hợp các mạch luận lý mà máy tính ảo LC-3 sẽ thực hiện lệnh ở cấp vi kiến trúc. Độc giả có thể tham khảo thêm vấn đề này trong hai tài liệu đã nêu và chương 3 và 4 về cách hoạt động và lập trình của máy tính LC-3.



Hình 1.14 Một ví dụ về mạch luận lý: đường truyền dữ liệu của máy LC-3

1.6.7 Thiết bị

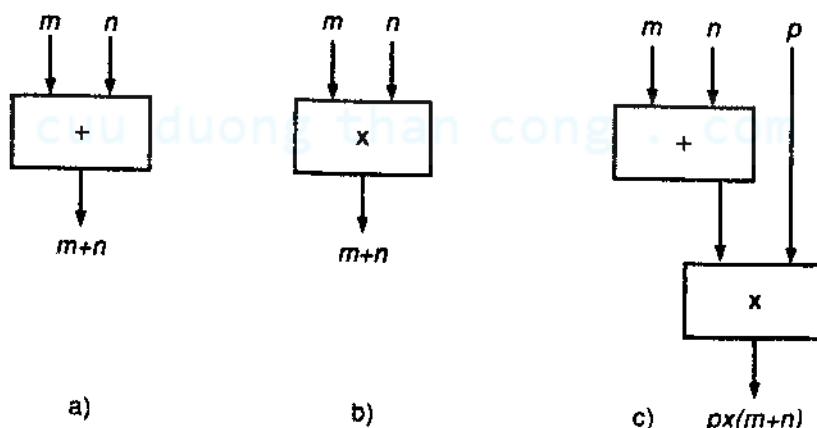
Sau cùng, mỗi mạch luận lý cơ bản ở cấp luận lý sẽ được hiện thực tương ứng bằng các mạch điện tử cụ thể. Khi đó, tùy theo sự chọn lựa, các mạch luận lý sẽ được ráp theo công nghệ CMOS, và

công nghệ này sẽ tạo ra mạch khác nếu ta ráp mạch theo NMOS, và tương tự như vậy cho các mạch khác. Với các mạch cứng này, các lệnh cấp cao qua nhiều công đoạn sẽ trở thành các bit 0 và 1, điều khiển các điện tử đóng mở các thành phần mạch, từ đó quá trình thực hiện lệnh sẽ diễn ra.

BÀI TẬP CUỐI CHƯƠNG

- 1.1. Một ngôn ngữ lập trình cấp cao hơn có thể chỉ thị cho một máy tính tính toán tốt hơn một ngôn ngữ cấp thấp hơn hay không? Hãy bàn luận dựa trên quan điểm phân tích các cấp chuyển đổi.
- 1.2. Các khó khăn nào của máy tính tương tự làm cho các nhà thiết kế muốn sử dụng máy tính số?
- 1.3. Chấp nhận chúng ta đã có một “hộp đen” nhập hai số và xuất ra tổng của chúng, và một “hộp đen” khác tính tích của hai số như trong hình 1.15a và 1.15b sau. Chúng ta có thể nối các hộp này lại với nhau để tính $p \times (m + n)$ như hình 1.15c. Nếu cho phép sử dụng số hộp này không hạn chế, hãy nối chúng lại để tính;
 - a. $ax + b$
 - b. Trung bình của bốn giá trị nhập w, x, y , và z .
 - c. $a^2 + 2ab + b^2$

Bạn có thể thực hiện câu c chỉ với một hộp cộng và một hộp nhân được không?



Hình 1.15 Các hộp đen cho bài tập 1.3

(a) Cộng, (b) Nhân, và (c) Cộng và nhân

- 1.4 Viết một mệnh đề bằng ngôn ngữ tự nhiên và đưa ra hai cách thông dịch (interpretation) khác nhau cho nó.
- 1.5 Một cô giáo dạy môn Sinh học, khi phòng học lý thuyết bị bật đột suất, cô liền thông báo: “Các em chuyển qua phòng thí nghiệm!”. Hãy đưa ra tất cả các cách thông dịch có thể để có thể hiểu mệnh đề trên.
- 1.6 Ngôn ngữ tự nhiên có khả năng diễn tả giải thuật không?
- 1.7 Cho một số a , hãy thực hiện các bước sau theo thứ tự:
1. Nhân nó với 4
 2. Cộng 4
 3. Chia cho 2
 4. Trừ 2
 5. Chia cho 2
 6. Trừ 1

Hãy viết ra biểu thức tương đương mà bạn có. Nhận xét sự khác biệt với bạn mình. Từ đó nêu nhận xét khi dùng ngôn ngữ tự nhiên biểu diễn giải thuật.

1.8

- a. Giả sử có khoảng 400 sinh viên trong nhóm sinh hoạt chuyên đề. Nếu mỗi sinh viên được cho một mã nhị phân riêng. Số bit tối thiểu để làm việc này là bao nhiêu?
 - b. Với số bit đã nêu trên, có bao nhiêu sinh viên có thể được thêm vào nhóm trên mà vẫn bảo đảm sự phân biệt về mã nhị phân?
- 1.9 Trong tiếng Anh có 26 mẫu tự alphabet. Cần tối thiểu bao nhiêu bit để biểu diễn cho một ký tự. Biết các mẫu bit này phải phân biệt nhau.

Hãy thực hiện yêu cầu trên với tiếng Việt. Chú ý, cần tìm số lượng tất cả các mẫu tự tiếng Việt (tức các mẫu tự có sự kết hợp với dấu sắc, huyền, hỏi ngã, nặng, ...) trước.

Chương 2

CÁC KIỂU DỮ LIỆU VÀ THAO TÁC

2.1 KIỂU DỮ LIỆU SỐ NGUYÊN

2.1.1 Số nguyên không dấu (*unsigned integer*)

Dạng biểu diễn thông tin, hay kiểu dữ liệu, đầu tiên mà chúng ta xét là kiểu số nguyên không dấu, kiểu dữ liệu này được sử dụng rất nhiều trong máy tính. Chẳng hạn, khi chúng ta muốn thực hiện một tác vụ nào đó một số lần xác định, chúng ta sẽ phải nghĩ ngay tới các giá trị nguyên không dấu là giới hạn số lần lặp. Số nguyên không dấu cũng được sử dụng để chỉ định vị trí của ô nhớ trong bộ nhớ máy tính, điều này cũng tương tự như địa chỉ số nhà, để sao cho mỗi ô nhớ đều có địa chỉ riêng biệt.

Ví dụ, trong số nguyên không dấu 329, số 3 “đáng giá” hơn số 2, và số 2 “đáng giá” hơn số 9, vì số 3 có “giá” $300 (3 \cdot 10^2)$, số 2 “trị giá” $20, (2 \cdot 10^1)$ trong khi số 9 chỉ có giá $9 (9 \cdot 10^0)$.

Như chúng ta đã thấy trong chương 1, số nguyên không dấu có thể được biểu diễn ở dạng nhị phân. Ví dụ số 6 được biểu diễn ở dạng nhị phân 5 bit là 00110, tương ứng với sự khai triển sau

$$0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$

Với k bit, chúng ta sẽ có 2^k trị nguyên không dấu từ 0 tới $2^k - 1$.

2.1.2 Số nguyên có dấu (*signed integer*)

Tuy nhiên, khi thực hiện các phép toán, chúng ta thường (mặc dù không phải là luôn luôn) gấp các đại lượng âm cũng như dương, mà máy tính làm việc với số nhị phân, có nghĩa là máy tính luôn dùng số không dấu mà ta thường biết cho mọi thao tác, nên số âm sẽ được đối xử ở dạng không dấu theo một quy luật xác định. Như vậy,

2^k trị khác nhau với độ dài k bit sẽ được chia ra làm hai nửa để quy định số có dấu, một nửa cho các số dương, và một nửa cho các số âm. Tuy nhiên, ở đây có vấn đề: với 2^k mã không dấu, mã nào sẽ biểu diễn trị nào? Để cho tiện, các số dương sẽ có trị từ dạng biến diễn của nó (giống như không dấu), còn với số âm thì có một số quy định như: bit trọng số cao nhất là bit dấu, các bit còn lại là trị tuyệt đối của số, dạng bù 1, và dạng bù 2. Bảng 2.1 trình bày cả ba cách biểu diễn số có dấu.

Ở dạng biểu diễn số âm dùng bit dấu và trị tuyệt đối, bit có trọng số cao nhất sẽ quy định dấu cho số có trị tuyệt đối ngay sau, nếu bằng 0, số là dương, 1 là âm. Dạng bù 1 sẽ biểu diễn số âm bằng việc đảo các trạng thái bit của số dương tương ứng, đảo từ 1 qua 0, và ngược lại. Còn dạng bù 2 (sẽ được xét cụ thể ở mục dưới) sẽ biểu diễn số âm bằng dạng bù 1 của nó cộng thêm 1. Trong ba cách, 2 cách đầu đơn giản về tư duy, nhưng không có lợi cho việc thực hiện phép toán hoặc mất trị trong tầm (2 trị 0, và -0, thực ra là 1 trị).

2.2 SỐ NGUYÊN BÙ 2

Từ bảng 2.1 ta thấy cách biểu diễn các số nguyên từ -16 tới +15 ở dạng bù 2. Nhưng tại sao cách biểu diễn này lại được chọn sử dụng để biểu diễn số âm trong máy tính?

Bảng 2.1 Ba cách biểu diễn số có dấu

Dạng biến diễn	Trị được biểu diễn		
	Trị tuyệt đối có dấu	Bù 1	Bù 2
00000	0	0	0
00001	1	1	1
00010	2	2	2
00011	3	3	3
00100	4	4	4
00101	5	5	5
00110	6	6	6
00111	7	7	7
01000	8	8	8
01001	9	9	9
01010	10	10	10
01011	11	11	11

01100	12	12	12
01101	13	13	13
01110	14	14	14
01111	15	15	15
10000	-0	-15	-16
10001	-1	-14	-15
10010	-2	-13	-14
10011	-3	-12	-13
10100	-4	-11	-12
10101	-5	-10	-11
10110	-6	-9	-10
10111	-7	-8	-9
11000	-8	-7	-8
11001	-9	-6	-7
11010	-10	-5	-6
11011	-11	-4	-5
11100	-12	-3	-4
11101	-13	-2	-3
11110	-14	-1	-2
11111	-15	-0	-1

Bảng 2.1 cho ta thấy các số dương được biểu diễn theo đúng mã nhị phân của chúng. Ví dụ với mã 5 bit, ta có phân nửa của 2^5 mã từ 0 tới $2^4 - 1$ là các số dương tương ứng từ 0 tới 15.

Khi thiết kế, các mạch luận lý phải càng đơn giản càng tốt, nên hầu hết máy tính đều sử dụng bộ cộng cho các thao tác số học. Bộ phận thực hiện thao tác này là đơn vị số học luận lý, hay gọi tắt là ALU (xin xem lại chương 1). Hai ngõ vào của ALU sẽ cộng các mẫu bit ở đầu vào, và tạo ra mẫu bit là tổng của hai mẫu bit này ở đầu ra. Ví dụ, ta có ALU xử lý mẫu 5 bit, thực hiện phép cộng hai chuỗi 00110 và 00101, kết quả là 01011. Phép cộng sẽ được thực hiện như sau:

$$\begin{array}{r}
 00110 \\
 00101 \\
 \hline
 01011
 \end{array}$$

Như vậy, ALU sẽ cộng từng bit theo cột từ phải qua trái, và có nhớ để đem qua trái, mà không biết (hay không quan tâm) hai mẫu

bit mà nó đang cộng là gì. Điều gì sẽ xảy ra khi ALU thực hiện cộng một số nguyên A bất kỳ với số nguyên có cùng biên độ nhưng trái dấu với số nguyên ban đầu $-A$? Kết quả đầu ra của ALU sẽ là 0, ở mẫu 5 bit là 00000. Có thể thấy dạng biểu diễn bù 2 cho từng số âm khi được ALU cộng với số dương cùng biên độ sẽ tạo ra kết quả là 0 ở cùng chiều dài bit, ví dụ vì 00101 là dạng biểu diễn của +5, thì 11011 sẽ được chọn là dạng biểu diễn cho -5, vì

$$\begin{array}{r} 00101 \\ 11011 \\ \hline 1\ 00000 \end{array}$$

↑

Bit nhơ sẽ bị bỏ qua trong mẫu kết quả

Chúng ta cần chú ý là ở kết quả đầu ra của ALU sẽ có bit nhơ, nhưng không ảnh hưởng tới kết quả phép cộng vì chúng ta chỉ quan tâm tới mẫu có cùng chiều dài bit với toán hạng đầu vào ALU mà thôi. Như vậy, dạng bù 2 cho từng số âm rất thích hợp cho ALU thực hiện phép trừ ở thao tác cộng với số bù 2 của số bị trừ.

Có hai bước trong quy luật tạo số bù 2 của một số:

1. Lật ngược trạng thái bit biểu diễn từ 1 qua 0, từ 0 qua 1 trong mẫu, còn gọi là phép bù 1.
2. Cộng 1 vào mẫu kết quả ở bước 1, để có mẫu kết quả sau cùng.

Ví dụ 2.1 Tìm dạng bù 2 cho số -12

Mẫu nhị phân của trị tuyệt đối của toán hạng 12 là 01100.

Ta thực hiện hai bước như sau:

1. Tìm bù 1 của 01100: 10011
2. Cộng 1 vào dạng bù 1: 10100

Kết quả dạng bù 2 của -12 là 10100, vì khi cộng 12 và -12 ở dạng bù 2, ta có kết quả là 0:

$$\begin{array}{r} 01100 \\ 10100 \\ \hline 1\ 00000 \end{array}$$

↔

Kết quả là 0

2.3 PHÉP TOÁN TRÊN BIT - PHÉP TOÁN SỐ HỌC

2.3.1 Cộng và trừ

Các phép toán số học trên các số bù 2 hoàn toàn tương tự như trên các số thập phân mà độc giả đã biết. Phép cộng sẽ thực hiện từ phải qua trái, từng ký số, mỗi lần như vậy ta có một ký số tổng và một ký số nhớ. Thay vì tạo ra một ký số nhớ khi có kết quả lớn hơn 9 (khi cộng trong hệ thập phân), thì ta chỉ tạo ra bit nhớ khi có kết quả lớn hơn 1 (vì 1 là ký số nhị phân lớn nhất).

Ví dụ 2.2 Tính biểu thức $11+3$.

Ta có:

Trị thập phân 11 được biểu diễn dưới dạng 01011

Trị thập phân 3 được biểu diễn ở dạng 00011

Tổng, có trị 14, là 01110

Ví dụ 2.3 Mô phỏng thực hiện phép trừ ở thao tác cộng ở ALU, tính biểu thức: $12 - 19$.

Trước tiên, CPU phân tích để tính biểu thức trên ở dạng: $12 + (-19)$, sau đó tính bù 2 của 19 (010011) để có -19 , tức 101101. Cộng 12, 001100, với -19 , tức 101101:

$$\begin{array}{r} 001100 \\ + \quad 101101 \\ \hline 111001 \end{array}$$

Kết quả trên có bit trọng số lớn nhất là 1, nên là số âm. Muốn tính là “âm” máy, ta cũng sẽ dùng phép bù 2 để tính trị tuyệt đối. Bù 1 của 111001 là 000110, cộng thêm 1 để có 000111, tức là 7. Tức kết quả sau cùng sẽ là -7 , đúng với thực tế!

Ví dụ 2.4 Cộng một số với chính nó ($x + x$), tính $6 + 6$.

Giả sử ta xét các mẫu có chiều dài 5 bit.

Mẫu nhị phân 5 bit của 6 là 00110, tức dạng khai triển là

$$0.2^4 + 0.2^3 + 1.2^2 + 1.2^1 + 0.2^0$$

Khi ta thực hiện $6 + 6$, tức ta có 2.6 , như vậy biểu thức khai triển sẽ là

$$2 \cdot (0.2^4 + 0.2^3 + 1.2^2 + 1.2^1 + 0.2^0)$$

hay $0.2^5 + 0.2^4 + 1.2^3 + 1.2^2 + 0.2^1$

Ở dạng 5 bit, biểu thức trên là

$$0.2^4 + 1.2^3 + 1.2^2 + 0.2^1 + 0.2^0$$

Ta có kết quả: 01100, tức dịch toán hạng ban đầu từng bit sang trái một vị trí. Như vậy, thao tác cộng một số với chính nó (tức nhân 2) tương đương với dời trái một bit.

2.3.2 Mở rộng dấu

Trong thực tế, thường người ta biểu diễn các số có giá trị nhỏ chỉ với một ít bit, ví dụ số 5 có dạng biểu diễn 6 bit là 000101, tuy nhiên khi ALU tính toán, nó sẽ biến số 5 dạng 6 bit này thành mẫu có chiều dài bit bằng với chiều dài bit của thanh ghi toán hạng đầu vào của nó, nếu thanh ghi toán hạng đầu vào dài 16 bit, mẫu 6 bit của trị 5 sẽ được mở rộng thêm các ký số 0, tức bit dấu, vào phía trước để được nạp vào thanh ghi này và tính toán. Việc mở rộng bit dấu này sẽ không ảnh hưởng tới trị của số.

Với số âm, vấn đề cũng tương tự. Vì trong máy tính, dạng biểu diễn số âm là bù 2, nên không thể thêm các ký số 0 vào đầu được, vì khi đó sẽ dẫn tới sai trị. Ví dụ, đang có số 11110, tức -2, nếu thêm ba bit 0 vào trước sẽ có 00011110, tức 30! Nhưng nếu ta mở rộng thêm các bit 1, tức bit dấu, vào trước thì trị số vẫn không thay đổi, ví dụ mở rộng thêm ba bit 1 vào trước dạng bù 2 của -2 sẽ cho 11111110, số này do có bit trọng số lớn nhất là 1, nên là số âm, còn trị tuyệt đối của nó là:

- Bù 1 cho: 00000001
- Cộng 1 cho: 00000010, tức 2

Như vậy, thao tác mở rộng thêm bit dấu (0 với số dương và 1 với số âm) vào phía trước dạng bù 2 sẽ không làm thay đổi giá trị của số ban đầu. Thao tác này được gọi là thao tác mở rộng dấu (SIGN-EXTENSION), và thường được viết tắt là SEXT.

2.3.3 Tràn số

Điều gì xảy ra khi đồng hồ đo Km với năm ký số đang chỉ 99998 mà chúng ta chạy thêm 10 Km nữa? Khi đó chúng ta thấy chỉ số là 00008, tức xe của chúng ta là mới toanh à? Dĩ nhiên là không rồi, vì ký số thứ sáu đã bị bỏ đi, nên chúng ta không thấy nó mà thôi. Để tráng tình trạng này, người ta thường tăng số ký số trên đồng hồ lên tám hay nhiều hơn nữa, tuy nhiên dù tăng bao nhiêu đi nữa, đến một lúc nào đó, chúng ta cũng sẽ bị tình trạng tràn số như trên mà thôi.

Trong máy tính cũng vậy, vì bộ nhớ máy tính là hữu hạn, nên chiều dài bit của các kiểu dữ liệu cũng bị hạn chế, tình trạng tràn số sẽ xảy ra nếu chúng ta không lưu ý đến giới hạn của toán hạng đầu vào sẽ gây ra kết quả sai ở đầu ra. Ví dụ, tính biểu thức sau với chiều dài toán hạng là 5 bit: $9 + 11$, chấp nhận hai trị này được lưu ở dạng 5 bit bù 2, tức trị lớn nhất có thể lưu đúng với dạng lưu trữ là 15; ta có

$$\begin{array}{r} 01001 \\ + \quad 01011 \\ \hline 10100 \end{array}$$

cuuduongthancong . com

Kết quả ai cũng biết là 21, nhưng ta lại nhận được một số âm, do bit trọng số lớn nhất là 1, tức -12!

Tương tự như vậy, khi cộng hai số âm theo dạng như trên, trị nhỏ nhất có thể nhận được là -16, nên khi tính biểu thức: $-12 - 6$; ta có

$$\begin{array}{r} 10100 \\ + \quad 11010 \\ \hline 01110 \end{array}$$

Tức kết quả nhận được là 14, trong khi đúng ra phải là -18!

Còn khi cộng một số âm với một số dương thì sao? Vấn đề là kết quả có tràn ra khỏi dạng lưu trữ hay không, nếu kết quả tràn số có khả năng xảy ra thì chúng ta phải đổi sang kiểu dữ liệu lớn hơn và phải trả giá về tốc độ chương trình và tốn bộ nhớ nhiều hơn.

2.4 PHÉP TOÁN TRÊN BIT – PHÉP TOÁN LUẬN LÝ

Các mạch điện tử được điều khiển bởi các trạng thái điện thế tương trưng bằng bit 1, hoặc 0, nên trong các ngôn ngữ cấp thấp hay C đều cung cấp các phép toán trên bit. Và hơn nữa, hai trị 1 và 0 còn

được xem là hai trạng thái *đúng (true)* và *sai (false)*, nên các phép toán trên bit còn được mở rộng thành các thao tác luận lý như trong một số ngôn ngữ lập trình khác (Pascal,...). Do đó hiểu được phép toán trên bit, mà mở rộng thêm là phép toán luận lý, sẽ giúp người lập trình nắm vững thao tác mà ALU thực thi với các bit trong toán hạng đầu vào. Như vậy, một cách tổng quát, khi đề cập tới trạng thái luận lý *đúng*, thì ta có thể nghĩ ngay nó là bit 1, và ngược lại; còn nếu gặp trạng thái luận lý *sai*, thì cũng có nghĩa là ta có bit 0.

2.4.1 Phép toán AND

AND là một hàm luận lý nhị phân, nó đòi hỏi hai toán hạng nhập, mỗi toán hạng là một trị luận lý 0 hoặc 1. Ta có thể hình dung toán hạng này hoạt động theo kiểu: cả hai đúng thì nó mới đúng. Một số tác giả xem toán hạng này là toán hạng “tất cả” (ALL). Nó chỉ cho kết quả đầu ra là 1 khi cả hai dữ liệu đầu vào đều là 1.

Bảng sự thật, một cơ chế thuận lợi để thấy sự hoạt động của toán hạng trên bit, được trình bày như sau:

A	B	AND
0	0	0
0	1	0
1	0	0
1	1	1

Từ bảng trên, ta thấy phải cả hai toán hạng nhập A và B đều cùng là 1, thì kết quả đầu ra mới là 1, còn nếu chỉ có một trong hai thì kết quả là 0. Toán hạng này có thể tổng quát cho các mảng n bit, tức khi tất cả bit nhập là 1, thì kết quả đầu ra mới là 1.

Ta hãy xét một số ví dụ sau đây.

Ví dụ 2.5

Nếu c là kết quả AND của a và b , với $a = 0011\ 1101$ và $b = 0100\ 0001$, thì c bằng bao nhiêu?

Trong trường hợp này, toán hạng trên bit AND sẽ được thực hiện giữa hai giá trị theo từng bit ở vị trí tương ứng, ta có kết quả

phép AND như sau:

$$\begin{array}{r} a : 0011\ 1101 \\ b : 0100\ 0001 \\ \hline c : 0000\ 0001 \end{array}$$

Ví dụ 2.6

Giả sử chúng ta có một mẫu nhị phân 8 bit được gọi là A , trong đó hai bit trọng số nhỏ nhất bên phải của A có ý nghĩa quan trọng. Máy tính sẽ tùy vào trị của hai bit này mà thực hiện một trong bốn tác vụ có thể. Làm sao cách ly bốn bit này để xét?

Trong trường hợp này chúng ta dùng *mặt nạ bit*. Một *mặt nạ bit* là một mẫu nhị phân mà có thể làm cho ta thấy được hai phần khác nhau trong các bit của A , phần ta cần quan tâm và phần ta muốn bỏ qua. Trong trường hợp này, mặt nạ bit 0000 0011 khi được AND với A sẽ tạo ra các bit 0 trong các bit từ vị trí 7 tới vị trí 2, còn các bit ở vị trí 1 và 0 thì sẽ được giữ nguyên. Một mặt nạ bit như vậy được gọi là *đã che* các bit ở vị trí 7 tới 2.

Cụ thể, nếu A là 0101 0110, còn mặt nạ bit là 0000 0011, thì kết quả AND sẽ cho 0000 0010. Có nghĩa là kết quả phép AND của bất kỳ mẫu 8 bit nào với mặt nạ 0000 0011 sẽ là một trong bốn mẫu 0000 0000, 0000 0001, 0000 0010, hoặc 0000 0011.

2.4.2 Phép toán OR

OR cũng là một phép toán luận lý nhị phân. Nó yêu cầu hai toán hạng đầu vào là hai trị luận lý. Khác với AND, chỉ cần một trong hai toán hạng đầu vào là 1 thì kết quả đầu ra của OR đã là 1. Bảng sự thật cho phép OR như sau:

A	B	OR
0	0	0
0	1	1
1	0	1
1	1	1

Tương tự như phép AND, chúng ta có thể áp dụng phép OR với nhiều toán hạng đầu vào, chỉ cần một trong các đầu vào bằng 1, thì kết quả đầu ra là 1.

Ví dụ 2.7

Nếu c là kết quả OR của a và b, với $a = 0011\ 1101$ và $b = 0100\ 0001$, thì c bằng bao nhiêu?

Trong trường hợp này, toán hạng trên bit OR được thực hiện giữa hai giá trị theo từng bit ở vị trí tương ứng, ta có kết quả phép OR:

$$\begin{array}{r} a : 0011\ 1101 \\ b : 0100\ 0001 \\ \hline c : 0111\ 1101 \end{array}$$

Tương tự như phép AND, ta cũng có khái niệm mặt nạ bit từ phép OR, xét ví dụ sau.

CuuDuongThanCong . com

Ví dụ 2.8

Với một trạng thái bit đã có, ta muốn hai bit trọng số nhỏ nhất của nó phải có trạng thái xác định là 11, thì mặt nạ xxxx xx11 sẽ được OR với trạng thái bit đã có, ví dụ

$$\begin{array}{r} 0011\ 1101 \\ 0000\ 0011 \\ \hline 0011\ 1111 \end{array}$$

Đi nhiên vì ta chỉ quan tâm tới các bit mà ta mong muốn, nên các bit còn lại sẽ không được xét tới.

Còn nếu như ta muốn hai bit này có trạng thái là 10 (hay 01) thì sao? Đây là một bài tập nhỏ, xin mời độc giả tự nghiên cứu.

2.4.3 Phép toán NOT

NOT là một hàm luận lý đơn toán hạng, nó chỉ cần một toán hạng nhập. Toán hạng này còn được gọi là toán hạng bù, vì nó thực hiện thao tác lật ngược trạng thái luận lý từ 1 qua 0, hoặc từ 0 qua 1.

Bảng sự thật cho toán tử này là

A	NOT
0	1
1	0

Toán tử NOT có thể được áp dụng cho mẫu nhiều bit, khi đó nó sẽ đảo trạng thái của từng bit một, ví dụ ta có

$$\begin{array}{l} a: 0100\ 0001 \\ \text{thì } c = \text{NOT } a: \underline{10111110} \end{array}$$

2.4.4 Phép toán Exclusive-OR (EX-OR)

Phép toán này còn được gọi ngắn gọn là XOR. Đây là toán tử hai toán hạng. Đầu ra của XOR sẽ là 1 nếu hai đầu vào là khác nhau.

Bảng sự thật cho toán tử XOR là

A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Tương tự như AND và OR, toán tử XOR cũng có thể được áp dụng cho hai mẫu n bit. Khi đó nó sẽ thực hiện XOR trên từng bit trọng số tương ứng.

Ví dụ 2.9

Nếu c là kết quả XOR của a và b , với $a = 0011\ 1101$ và $b = 0100\ 0001$, thì c bằng bao nhiêu?

Trong trường hợp này, toán hạng trên bit XOR được thực hiện giữa hai giá trị theo từng bit ở vị trí tương ứng, ta có kết quả phép XOR:

$$\begin{array}{l} a : 0011\ 1101 \\ b : 0100\ 0001 \\ \hline c : \underline{0111\ 1100} \end{array}$$

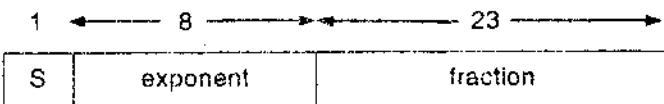
Toán tử XOR tỏ ra rất lợi hại khi ta muốn so sánh hai mẫu nào đó có đồng nhất hay không. Toán tử XOR sẽ cho tất cả các bit bằng 0 trong mẫu kết quả nếu hai mẫu nhập là đồng nhất.

2.5 KIỂU DỮ LIỆU DẤU CHẤM ĐỘNG (*Floating point data type*)

Rất nhiều phép toán số học chúng ta dùng số thực, để dễ hình dung ta thường sử dụng số thực dấu chấm tĩnh, ví dụ 2.73, -0.01256, ... Đối với những số giá trị nhỏ, việc viết theo dạng này sẽ làm chương trình đơn giản, tuy nhiên với những số rất nhỏ hoặc rất lớn, nếu viết theo cách này sẽ tốn rất nhiều bộ nhớ, thậm chí không thể mô tả được giá trị vì ta không đủ bit để biểu diễn tầm trị.

Kiểu dữ liệu dấu chấm động là cách giải quyết cho vấn đề. Hiện nay, hầu hết các kiến trúc tập lệnh (ISA) đều có một vài kiểu dữ liệu dấu chấm động theo định dạng chuẩn IEEE 754, một trong chúng là kiểu *float*, chiều dài 32 bit, có cấu trúc như sau:

- 1 bit cho dấu (dương hay âm)
- 8 bit cho tầm (vùng số mũ-exponent)
- 23 bit cho độ chính xác (fraction)



$$N = (-1)^S \times 1.\text{fraction} \times 2^{\text{exponent}-127}, \quad 1 \leq \text{exponent} \leq 254$$

Hình 2.1 Kiểu số thực dấu chấm động *float*

Dạng biểu diễn số thực tương ứng như trong hình 2.1 là một phần của chuẩn IEEE cho số học dấu chấm động. Dạng biểu diễn này rất giống với cách biểu diễn số khoa học mà ta đã học ở trung học, ví dụ 6.023×10^{23} , trong đó phần dấu là dương, phần trị là 6.023, và phần mũ là 23. Chú ý là phần trị được chuẩn hóa, nên chỉ có một (và chỉ một) ký số thập phân khác 0 nằm bên trái dấu chấm thập phân mà thôi.

Theo cách biểu diễn của IEEE ở hình 2.1, kiểu dữ liệu này có cấu trúc gồm ba phần, phần dấu chỉ có một bit, nếu nó là 0 ($S = 0$), ta có số dương, và nó là 1 ($S = 1$), số là số âm.

Phần mũ dài 8 bit nhị phân, biểu diễn 256 trị số không dấu, nhưng ta chỉ sử dụng 254 trị số trong đó mà thôi. Vùng mũ chứa 0000 0000 (tức 0), hay 1111 1111 (tức 255) sẽ cho một ý nghĩa đặc biệt khác mà ta sẽ xét sau.

Phân trị dài 23 bit cũng được chuẩn hoá, chỉ được có một (và chỉ một) ký số nhị phân khác không nằm bên trái dấu chấm nhị phân. Vì hệ nhị phân chỉ có một ký số khác 0 là 1 mà thôi nên ký số này luôn là 1, do đó trong dạng biểu diễn ta không cần phải viết nó ra vì nó được xem là mặc nhiên. Như vậy phân trị tổng cộng có 24 bit, 23 bit từ kiểu dữ liệu và một bit mặc nhiên (là bit 1) phía trước bên trái dấu chấm nhị phân không cần biểu diễn tường minh.

Ví dụ 2.10 Hãy biểu diễn số $-6\frac{5}{8}$ ở dạng kiểu dữ liệu dấu chấm động.

Trước tiên, chúng ta biểu diễn số $-6\frac{5}{8}$ ra dạng nhị phân: -110.101, biểu thức khai triển là

$$= (1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3})$$

Chuẩn hóa tri, tao ra -1.10101 . 2².

Số là âm nên bit dấu sẽ là 1. Trị mũ là 2, nên vùng mũ sẽ chứa trị sao cho hiệu của nó là số mũ 2, tức vùng mũ chứa trị 129 ($129 - 127 = +2$), tức $1000\ 0001$. Vùng trị gồm 23 bit (không kể bit 1 mặc nhiên) là 101010000000000000000000 . Kết quả là số $-6\frac{5}{8}$ được biểu diễn ở dạng IEEE là

1 10000001 1010100000000000000000000000

Ví dụ 2.11 Hãy tìm trị cho dạng biểu diễn thuộc kiểu dấu chấm dông sau:

Dạng biểu diễn trên có bit 0 bắt đầu, nên số sẽ là dương. Tám bit kế tiếp cho trị không dấu là 123, trừ với 127, ta có số mũ là -4. 23 bit cuối đều là 0. Vì vậy số được biểu diễn sẽ là

+1.000000000000000000000000000000 . 2⁻⁴, tức $\frac{1}{16}$.

Như đã nói ở trên, phần mũ của kiểu dữ liệu dấu chấm động IEEE không được chứa các mảng 00000000 hoặc 11111111, vì chúng có một ý nghĩa khác.

Nếu phần mũ chứa 00000000 thì số mũ sẽ được xem là -126, phần trị mантissa bắt đầu bằng bit 0 bên trái dấu chấm nhị phân, tới dấu chấm nhị phân, và theo sau là 23 bit phần trị bình thường, cụ thể

$$(-1)^S \times 0.\text{fraction} \times 2^{-126}$$

Ví dụ, dạng biểu diễn dấu chấm động

0 00000000 000010000000000000000000

có bit dấu bằng 0, nên là số dương, tám bit kế tiếp bằng 0, nên số mũ là -126, 23 bit cuối tạo ra dạng số 0.00001000000000000000000, tức bằng 2^{-5} . Như vậy, số được biểu diễn là $2^{-5} \cdot 2^{-126}$, tức 2^{-131} . Đây là một số rất nhỏ, nếu biểu diễn theo dạng dấu chấm tĩnh thì thật là phiền phức. Độc giả có thể kiểm chứng các ví dụ sau đây.

Ví dụ 2.12 Kiểm chứng trị kiếp dấu chấm động của các mảng sau:

0 10000011 001010000000000000000000 là $1.00101 \times 2^4 = 18.5$

1 10000010 001010000000000000000000 là $-1 \times 1.00101 \times 2^3 = -9.25$

0 11111110 111111111111111111111111 là $1.111\dots11 \times 2^{127} \sim 2^{128}$

1 00000000 000000000000000000000001 là -2^{-149}

0 00000000 000000000000000000000000 là 0^+

1 00000000 000000000000000000000000 là 0^-

Nếu phần mũ chứa 11111111 thì ta sẽ có hai khả năng xảy ra:

- Nếu phần trị bằng 0, số sẽ là dương vô cực ($+\infty$) hay âm vô cực ($-\infty$) tùy vào bit dấu.
- Nếu phần trị khác 0, lúc này việc biểu diễn số dấu chấm động sẽ không là một số (Not a Number - NaN), không quan tâm tới bit dấu. Dạng NaN này báo hiệu những thao tác không hợp lệ như nhân zero (0) với vô cực (∞).

Tương tự, kiểu *double* có chiều dài 64 bit theo định dạng sau:

1	11	52
S	exponent	fraction

$$N = (-1)^S \times 1.\text{fraction} \times 2^{\text{exponent}-1023}, 1 \leq \text{exponent} \leq 1022$$

Hình 2.2 Kiểu số thực dấu chấm động double

BÀI TẬP CUỐI CHƯƠNG

- 2.1** Chuyển các số nhị phân bù 2 sau đây ra trị thập phân.
- 1010
 - 01011010
 - 11111110
 - 001101100011
- 2.2** Chuyển các số thập phân sau đây ra trị nhị phân bù 2 8 bit.
- 102
 - 64
 - 33
 - 127
- 2.3** Nếu chữ số cuối cùng của một trị nhị phân bù 2 là 0, thì số là chẵn. Nếu hai chữ số cuối của dạng biểu diễn nhị phân là 00 (ví dụ như 10100), thì ta nói số đó có đặc điểm gì? Tổng quát cho 3, 4, .. bit 0 cuối?
- 2.4** Không thay đổi trị, không dùng các hệ thống số trung gian, chuyển các trị nhị phân bù 2 sau ra thành dạng bù 2 8 bit.
- 1010
 - 011001
 - 1111111000
 - 01
- 2.5** Cộng các số nhị phân bù 2 sau. Viết lại thao tác cộng ở dạng số thập phân.

- a. 01 + 1011
 - b. 11 + 01010101
 - c. 0101 + 110
 - d. 01 + 10
- 2.6 Thực hiện phép cộng cho các trị nhị phân không dấu với biểu diễn như trong bài 2.5.
- 2.7 Thực hiện phép cộng nhị phân cho các trị nhị phân bù 2 4 bit sau. Kết quả nào bị tràn? Hãy đề nghị phương án điều chỉnh để nhận kết quả đúng.
- a. 1100 + 0011
 - b. 1100 + 0100
 - c. 0111 + 0001
 - d. 1000 + 0001
 - e. 0111 + 1001
- 2.8 Hãy nêu các điều kiện xảy ra tràn số khi cộng hai số nhị phân bù hai. Nêu các ví dụ cụ thể minh họa cho vấn đề này.
- 2.9 Hãy giải thích tại sao tổng của hai số gồm một trị nhị phân bù 2 âm và một trị nhị phân bù 2 dương không bao giờ gây tràn số. Kết quả nhận được có đúng không? Nêu ví dụ minh họa cụ thể.
- 2.10 LC-3 là một máy tính 16 bit, khi cộng hai số bù 2 01010101010101 và 0011100111001111, thì nhận được 1000111100100100. Có sai sót gì ở đây không? Nếu có, sai sót là gì?
- Nếu không, tại sao?
- 2.11 Tính các biểu thức sau, ghi kết quả ở dạng nhị phân.
- a. 01010101 AND 10010110
 - b. 10010011 OR 10110000
 - c. 1011 AND (1010 AND 1011)

- d. 01001001 OR ((NOT 10110110) AND 10101000)
e. 01001001 OR (NOT (10110110 AND 10101000))

2.12 Hãy chứng minh định lý De Morgan 1 và 2 với các giá trị nhị phân cụ thể sau:

$$A = 10001110 \text{ và } B = 01001001$$

2.13 Viết các số thập phân sau ra dạng số thực dấu chấm động IEEE.

- a. 5.75
- b. -75
- c. 3.1415927
- d. 64000

2.14 Hãy viết các số thực dấu chấm động ở dạng IEEE sau thành số nhị phân.

- a. 0 10000000 0000000000000000000000000000
- b. 1 10000011 0001000000000000000000000000
- c. 0 11111111 0000000000000000000000000000
- d. 1 10000000 1001000000000000000000000000

2.15 Chuyển dãy các mã ASCII sau ra thành chuỗi các ký tự bằng cách gồm nhóm từng 8 bit để có một ký tự ASCII.

- a. x48656c7f5d
- b. x31495693
- c. x4c5d6e7f
- d. x347566

2.16 Xét hai số hệ thập lục phân x23FD34 và xEEAB124. Các giá trị chúng biểu diễn trong năm kiểu dữ liệu sau đây là gì?

Kiểu dữ liệu	x23FD34	xEEAB124
Nhị phân không dấu	.	
Bù 1		
Bù 2		
Dấu chấm động IEEE 754		
Chuỗi ký tự ASCII		

2.17 Điền vào bảng sự thật với các biểu thức đã được cho. Dòng đầu tiên trong bảng là một ví dụ.

$$Q_1 = \text{NOT} (\text{A AND B})$$

$$Q_2 = \text{NOT} (\text{NOT} (\text{A}) \text{ AND } \text{NOT} (\text{B}))$$

A	B	Q_1	Q_2
0	0	1	0

2.18 Điền vào bảng sự thật với các biểu thức đã được cho. Dòng đầu tiên trong bảng là một ví dụ.

$$Q_1 = \text{NOT} (\text{NOT}(\text{X}) \text{ OR } (\text{X AND Y AND Z}))$$

$$Q_2 = \text{NOT} ((\text{Y OR Z}) \text{ AND } (\text{X AND Y AND Z}))$$

X	Y	Z	Q_1	Q_2
0	0	0	0	1

Chương 3

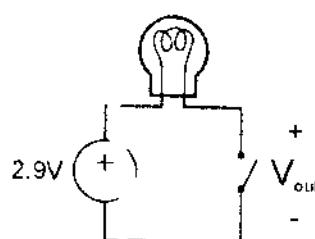
CÁC CẤU TRÚC LUẬN LÝ SỐ

Như trong chương 1 đã giới thiệu, máy tính được tạo ra từ rất nhiều các cấu trúc rất đơn giản. Ví dụ như bộ vi xử lý Intel Pentium IV, được đưa ra thị trường vào năm 2000, được tạo ra từ hơn 42 triệu transistor loại MOS, còn bộ vi xử lý IBM Power PC 750FX, được trình làng vào năm 2002, bao gồm hơn 38 triệu transistor MOS. Trong chương này, chúng tôi sẽ giải thích nguyên tắc làm việc của transistor MOS, cách thức ghép nối các transistor lại để tạo ra các công luận lý như AND, OR, NOT, và rồi phương thức ghép các công luận lý lại để tạo ra các cấu trúc lớn hơn như ADDER, MULTIPLEXER, DECODER, thanh ghi, sau cùng hình thành nên các bộ vi xử lý như LC3, Pentium IV,....

Nhưng đầu tiên chúng ta hãy làm quen với transistor.

3.1 TRANSISTOR

Đa số máy tính ngày nay sử dụng các bộ vi xử lý (*microprocessor*) được tạo từ các transistor họ MOS. MOS viết tắt từ *metal-oxide-semiconductor*. Nguyên lý hoạt động của MOS nằm ngoài giáo trình này, vì ở đây chúng ta chỉ cần hiểu phương thức hoạt động của nó để hình thành nên các công, và sau là mạch số lớn hơn.

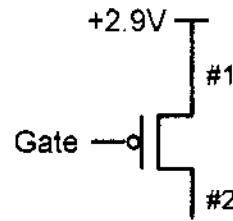
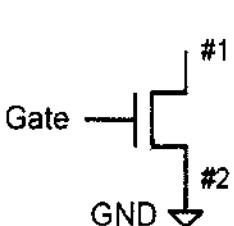


Hình 3.1 Một công tắc điện đơn giản

Có hai loại transistor MOS: loại p (*positive*) và loại n (*negative*). Cả hai đều hoạt động một cách luận lý như một công tắc điện như trên hình 3.1.

Khi khóa mở, không có dòng điện qua mạch nên đèn tắt, điện thế $V_{out} = 2,9V$, tức điện thế ra ở transistor ở mức cao, ta có mức logic “1”. Khi khóa đóng, có dòng chạy qua mạch, đèn sáng, điện thế $V_{out} = 0V$, khi đó điện thế ra ở transistor ở mức thấp, mức logic “0”.

Có hai loại transistor như hình dưới đây.



Hình 3.2 Transistor loại n Hình 3.3 Transistor loại p

Với transistor loại n, khi Gate có điện thế dương, tồn tại dòng điện qua hai đầu #1 và #2, tức xảy ra sự ngắn mạch. Lúc này transistor như một công tắc đóng. Và ngược lại, khi Gate có điện thế bằng 0, không có dòng điện qua hai đầu #1 và #2, tức mạch hở. Lúc này transistor như một công tắc mở. Ta có thể thấy trạng thái logic ở đầu ra của transistor loại n ngược lại với trạng thái đầu vào ở Gate.

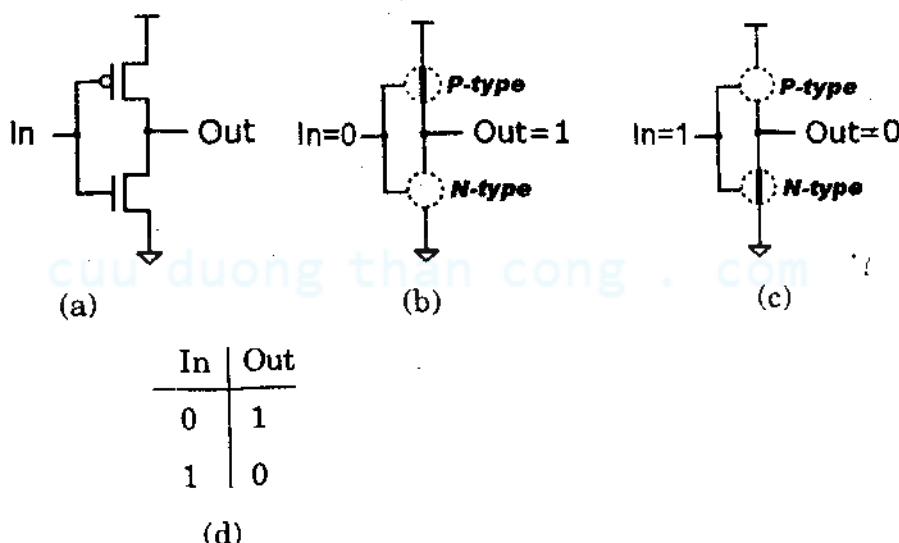
Trong khi đó, trạng thái mức logic ở đầu vào Gate và đầu ra của transistor loại p là như nhau, tức khi Gate có mức điện thế cao, thì transistor sẽ hở mạch; còn khi Gate có mức điện thế là 0 volt, thì có dòng điện giữa #1 và #2.

3.2 CÔNG LUẬN LÝ (*LOGIC GATE*)

Các transistor là các thành phần cơ bản để tạo nên các công luận lý. Sau đây chúng ta sẽ hiểu cách tạo ra các công luận lý cơ bản AND, OR, và NOT từ các transistor MOS. Cũng cần lưu ý là, với tầm trị điện áp analog từ 0-2,9V, khi ta có trị điện thế từ 0-0,5V, khi đó ta có mức logic 0; còn ứng với mức logic 1 là mức điện áp từ 2,4V – 2,9V. Sau này để dễ hình dung ta có thể nói hai mức điện áp là 0V và 2,9V cho hai mức logic khác nhau.

3.2.1 Cổng NOT (hay Inverter)

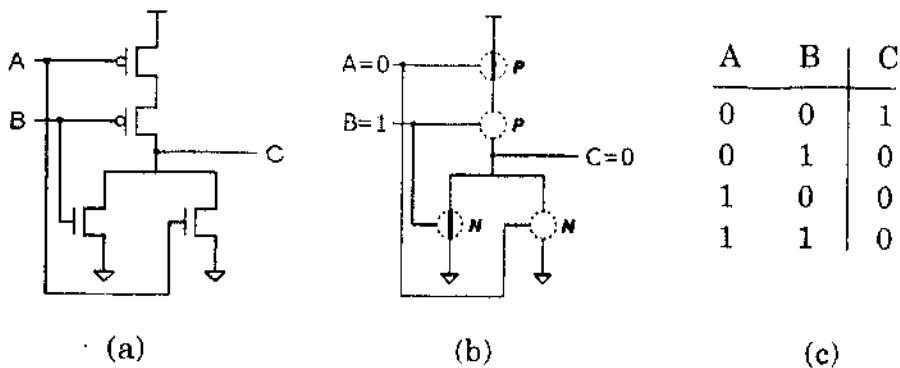
Hình 3.4 cho thấy cách tạo cổng NOT từ hai transistor MOS, một loại p và một loại n. Hình 3.4a biểu diễn sơ đồ mạch. Hình 3.4b trình bày trường hợp đầu vào có mức logic 0, đầu ra có mức logic 1 do transistor loại p dẫn, transistor loại n không dẫn, đầu ra vì vậy được nối với nguồn, tức ta có mức logic 1. Khi đầu vào có mức logic 1 như ở hình 3.4c thì transistor loại p không dẫn, mạch hở, trong khi transistor loại n dẫn nên đầu ra trong trường hợp này được nối với đất, tức 0 volt, ta có mức logic 0. Hình 3.4 cho ta bảng sự thật hoạt động của cổng NOT.



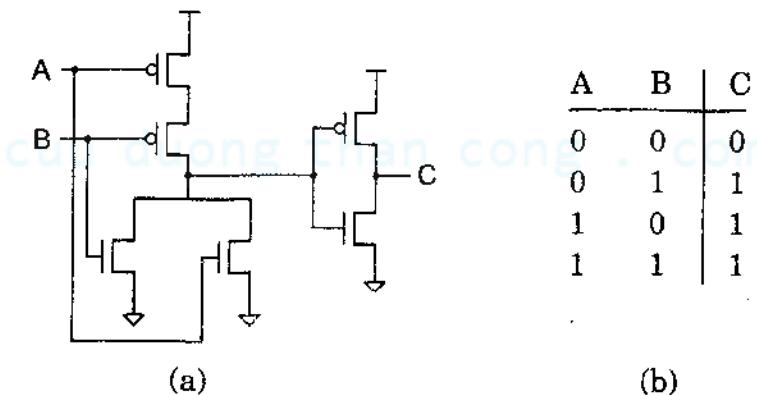
Hình 3.4 Cổng NOT

3.2.2 Cổng OR và NOR

Hình 3.5 minh họa cổng NOR. Hình 3.5a là sơ đồ mạch của cổng NOR, gồm hai transistor loại p và hai transistor loại n. Hình 3.5b trình bày sự hoạt động của mạch khi đầu vào A là 0 volt (bit 0) và B là 2,9 volt (bit 1). Khi đó, transistor loại p phía trên dẫn điện, trong khi transistor loại p dưới thì hở mạch, nên đầu ra C không nối với nguồn 2,9 volt. Nhưng transistor loại n bên trái lại dẫn, làm cho đầu ra bị nối đất, 0 volt. Và cũng tương tự như vậy cho các trường hợp còn lại. Hình 3.5c trình bày bảng sự thật của cổng NOR. Trong trường hợp này ta cũng chấp nhận mức điện áp 0 volt tương ứng với bit 0, và mức 2,9 volt tương ứng bit 1.

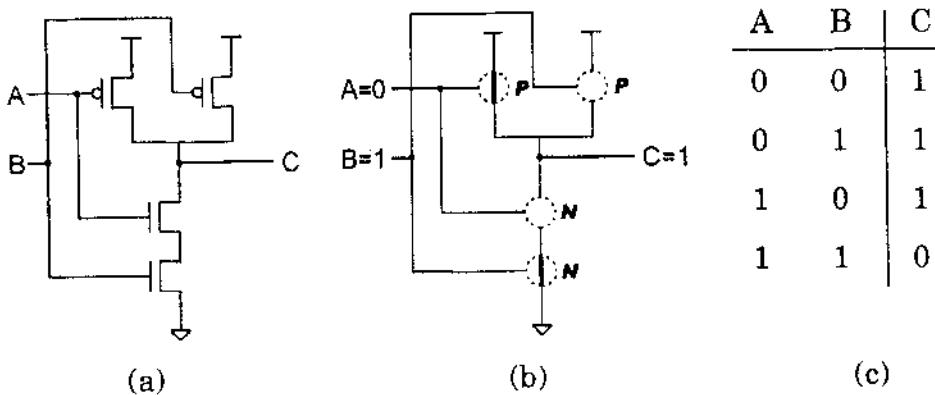
**Hình 3.5 Cổng NOR**

Từ mạch cổng NOR hình 3.5a, nếu ta thêm cổng NOT ở đầu ra như hình 3.6a, ta có mạch của cổng OR với bảng sự thật được trình bày trong hình 3.6b.

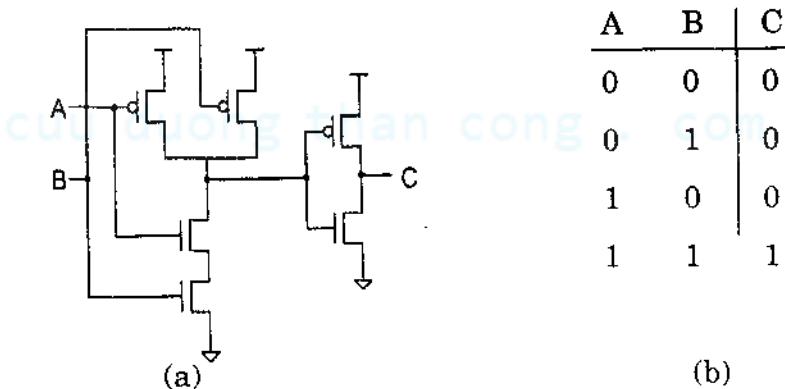
**Hình 3.6 Cổng OR**

3.2.3 Cổng AND và NAND

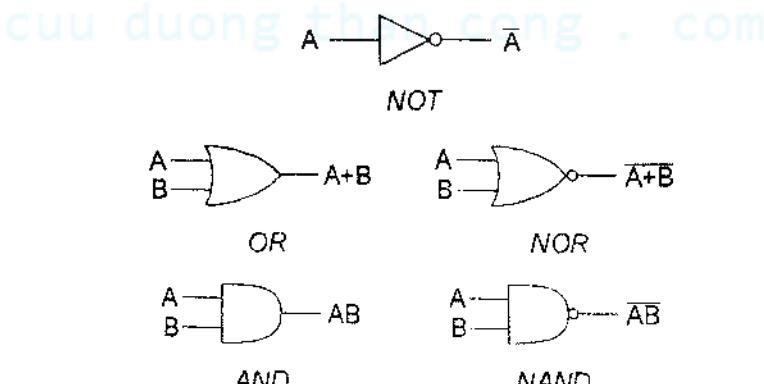
Hình 3.7 minh họa cổng NAND. Hình 3.7a là sơ đồ mạch của cổng NAND, gồm hai transistor loại p và hai transistor loại n. Hình 3.7b trình bày sự hoạt động của mạch khi đầu vào A là 0 volt (bit 0) và B là 2,9 volt (bit 1). Khi đó, transistor loại p phía trên trái dẫn điện, trong khi transistor loại p phía trên phải thì hở mạch, nên đầu ra C được nối với nguồn 2,9 volt. Trong khi đó, transistor loại n phía dưới lại dẫn, nhưng transistor loại n phía trên lại hở mạch, nên cuối cùng đầu ra C được nối với nguồn, tức 2,9 volt, bit 1. Và cũng tương tự như vậy cho các trường hợp còn lại. Hình 3.7c trình bày bảng sự thật của cổng NAND. Trong trường hợp này ta cũng chấp nhận mức điện áp 0 volt tương ứng với bit 0, và mức 2,9 volt tương ứng bit 1.

**Hình 3.7 Cổng NAND**

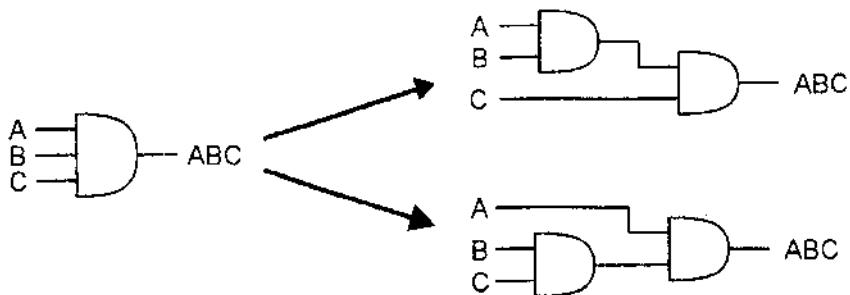
Từ mạch cổng NAND hình 3.7a, nếu ta thêm cổng NOT ở đầu ra như hình 3.8a, ta có mạch của cổng AND với bảng sự thật được trình bày trong hình 3.8b.

**Hình 3.8 Cổng AND**

Với các loại cổng đã được nêu, khi sử dụng để thiết kế mạch, ta dùng các ký hiệu theo quy ước như hình 3.9 dưới đây.

**Hình 3.9 Các cổng logic cơ bản**

Khi muốn biểu diễn nhiều đầu vào, chúng ta có thể sử dụng quy ước như hình 3.10, thay vì dùng nhiều tầng cổng AND. Các cổng khác cũng có sự tương tự.



Hình 3.10 Cổng logic nhiều đầu vào

3.2.4 Định luật De Morgan

Luật De Morgan cho phép chúng ta biểu diễn cổng OR bằng cổng AND kèm theo một số cổng NOT, hay ngược lại. Có hai luật De Morgan 1 và De Morgan 2 như sau:

$$\overline{A + B} = \overline{A} \cdot \overline{B} \quad (1) \quad \text{và} \quad \overline{A \cdot B} = \overline{A} + \overline{B} \quad (2)$$

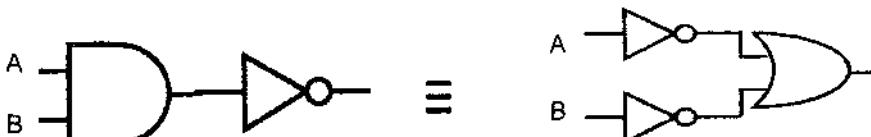
hay viết ở dạng khác là

$$A + B = \overline{\overline{A} \cdot \overline{B}} \quad A \cdot B = \overline{\overline{A} + \overline{B}}$$

Với ký hiệu ‘+’ đặc trưng cho phép OR, và ‘.’ cho phép AND. Hình 3.11 và 3.12 biểu diễn luật De Morgan 1 với (1), và luật De Morgan 2 với (2).



Hình 3.11 Luật De Morgan 1



Hình 3.12 Luật De Morgan 2

Độc giả có thể chứng minh 2 luật trên bằng các tính chất của đại số Boole hay bằng công cụ bảng sự thật một cách dễ dàng.

3.3 MẠCH TỔ HỢP (COMBINATIONAL CIRCUIT)

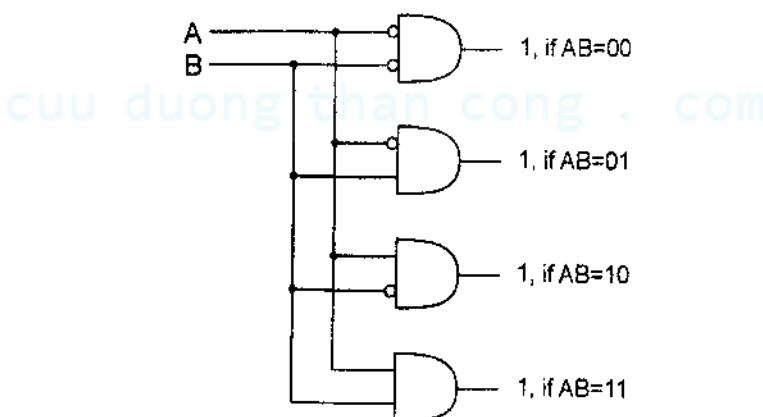
Sau khi đã hiểu cách hoạt động của các cổng luận lý cơ bản, bây giờ chúng ta chuyển qua bước kế tiếp là xây dựng một số cấu trúc logic quan trọng trong vi kiến trúc của một máy tính.

Có hai loại cấu trúc luận lý cơ bản là mạch tổ hợp và mạch tuần tự. Cấu trúc mạch tổ hợp là mạch luận lý mà các giá trị đầu ra của nó phụ thuộc vào tổ hợp các giá trị đầu vào của nó ở cùng thời điểm. Trong khi đó, mạch tuần tự có thể giữ được thông tin, và làm cơ sở cho cấu trúc bộ nhớ của máy tính. Phần này đề cập mạch tổ hợp, còn mạch tuần tự sẽ được nêu trong các mục tiếp theo.

Có ba loại tổ hợp mà chúng ta xét trong phần này: mạch giải mã, mạch phân kênh, và bộ cộng toàn phần.

3.3.1 Mạch giải mã (Decoder)

Mạch giải mã là một mạch luận lý số có đặc tính là tương ứng với một tổ hợp tín hiệu đầu vào thì chỉ có duy nhất một đầu ra tương ứng được tích cực, ví dụ ở mức logic 1, còn các đầu ra khác đều không tích cực, có mức logic 0. Hình 3.13 chỉ ra một bộ giải mã hai đầu vào, tương ứng với nó là $2^2 = 4$ đầu ra. Ví dụ, khi đầu vào AB là 01, thì đầu ra thứ hai sẽ tích cực.

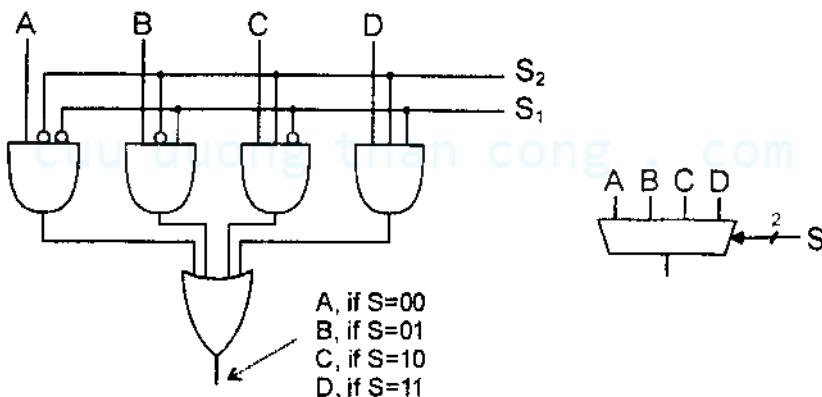


Hình 3.13 Mạch giải mã 2 bit

Tổng quát, với một mạch giải mã có n đầu vào thì chúng ta sẽ có 2^n đầu ra. Chúng ta sẽ thấy bộ giải mã rất có lợi trong việc giải mã địa chỉ bộ nhớ, giải mã lệnh tương ứng với mã lệnh (*opcode*) mà ta sẽ thấy sau này khi học LC3.

3.3.2 Mạch phân kênh (*Multiplexer*)

Khi có nhiều kênh dữ liệu (như A, B, C, D) muốn truyền thông tin ra một kênh truyền bên ngoài thì việc chọn kênh tất yếu phải diễn ra. Mạch số giải quyết việc này là mạch phân kênh. Hình 3.14 là một mạch phân kênh có hai tín hiệu điều khiển S_1S_2 , bốn đầu vào A, B, C, D. Khi $S_1S_2 = 10$, cổng AND tương ứng đầu vào C mở (vì $S_1S_2 = 11$), tín hiệu từ đầu vào C sẽ được chọn truyền qua mạch phân kênh.

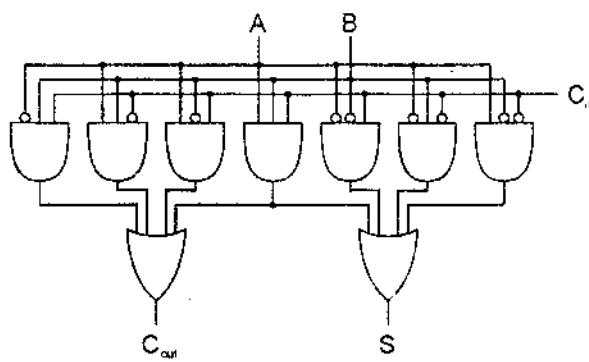


Hình 3.14 Mạch phân kênh 4 đầu vào

Tổng quát, một mạch phân kênh gồm 2^n đầu vào và n đường mã chọn kênh.

3.3.3 Mạch cộng toàn phần (*Full adder*)

Chương hai đã nêu phép cộng nhị phân. Cũng tương tự như phép cộng thập phân, thao tác cộng nhị phân cũng được thực hiện từ phải qua trái, từ chữ số có trọng số nhỏ qua chữ số có trọng số lớn, nếu có nhớ, bit nhớ sẽ được đem qua cộng vào thao tác cộng với trọng số lớn hơn. Hình 3.15a là mạch cộng toàn phần thực hiện thao tác cộng nhị phân giữa hai bit A và B với bit nhớ nếu có từ thao tác cộng hai bit trọng số nhỏ hơn; bảng sự thật của thao tác cộng này được trình bày trong hình 3.15b.

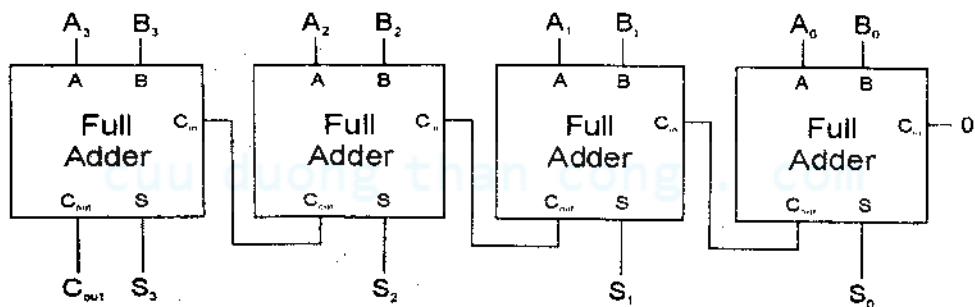


A	B	C _{in}	S	C _{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(a) (b)

Theo bảng sự thật hình 3.15b, có tám tổ hợp với ba bit ABC_{in} , tuy nhiên khi $ABC_{in} = 000$ thì đầu ra của mạch cộng $SC_{out} = 00$ do tất cả đầu ra của bảy cổng AND đều ở mức logic 0, do đó ta không cần cổng AND cho riêng tổ hợp này.

Hình 3.16 minh họa mạch cộng hai số nhị phân 4 bit, dùng bốn mạch cộng toàn phần ở hình 3.15a. Chú ý đầu vào $C_{in} = 0$ ở bộ cộng đầu tiên bên phải ứng với trọng số nhỏ nhất, còn đầu ra C_{out} ở phần tử thứ i được nối với đầu vào C_{in} của phần tử thứ $i + 1$.



Hình 3.16 Mạch cộng hai số nhị phân 4 bit

3.3.4 Một ví dụ về thiết kế mạch tổ hợp

Như đã biết, mạch tổ hợp là mạch luận lý mà các giá trị đầu ra của nó phụ thuộc vào tổ hợp các giá trị đầu vào của nó ở cùng thời điểm. Để biểu diễn mối liên quan này ta dùng một khái niệm được gọi là bảng sự

thật (*truth table*). Trong bảng này, tất cả các đầu vào (*input*), và đầu ra (*output*) đều được trình bày một cách cụ thể, chi tiết với tất cả các trạng thái có thể có. Sau đó, qua việc phân tích các trạng thái biến đổi từ đầu vào cho tới đầu ra, ta sẽ thiết lập hàm luận lý tương ứng với các trạng thái mong muốn, rồi với các cổng NOT, AND, OR ta có mạch logic của mạch cần thiết kế. Ta hãy xét ví dụ cụ thể sau.

Ví dụ 3.1 Thiết kế mạch kiểm tra sự chiếm đa số của bit 1 từ ba bit đầu vào (*majority function*).

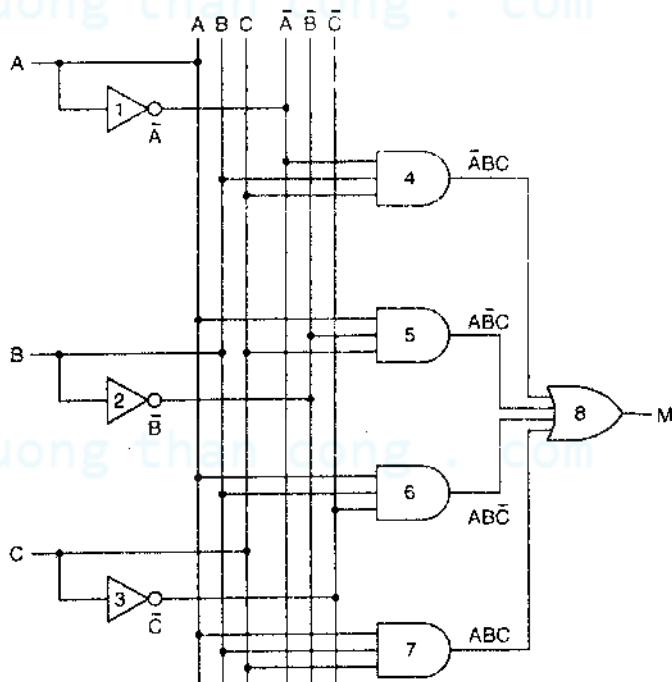
Ta có bảng sự thật ở hình 3.17a. Vì yêu cầu kiểm tra sự chiếm đa số của bit 1 từ ba bit, nên đầu ra (*M*) sẽ là 1 khi số bit 1 ở đầu vào là từ 2 trở lên. Như vậy, ta có hàm logic sau đây:

$$M = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

Hình 3.17 trình bày mạch logic để cụ thể hóa hàm logic trên. Trong hình này, các cổng được đánh số để dễ phân biệt, và theo trình tự từ đầu vào tới đầu ra.

A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(a)



(b)

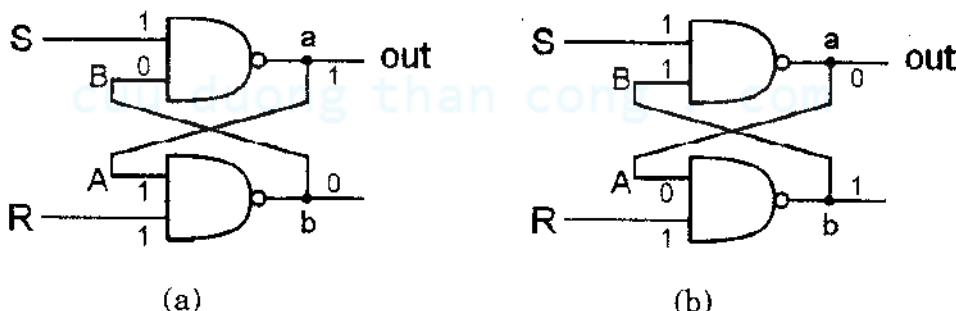
Hình 3.17 Bảng sự thật (a) và mạch logic (b) của mạch kiểm tra đa số đầu vào (*A, B, C*)

3.4 PHẦN TỬ NHỚ CƠ BẢN

Phần này trình bày các cấu trúc logic có thể lưu trữ thông tin, còn gọi là mạch nhớ. Khác với mạch tổ hợp, mạch nhớ có thể lưu trữ và thay đổi thông tin khi cần.

3.4.1 Mạch cài R-S (*R-S latch*)

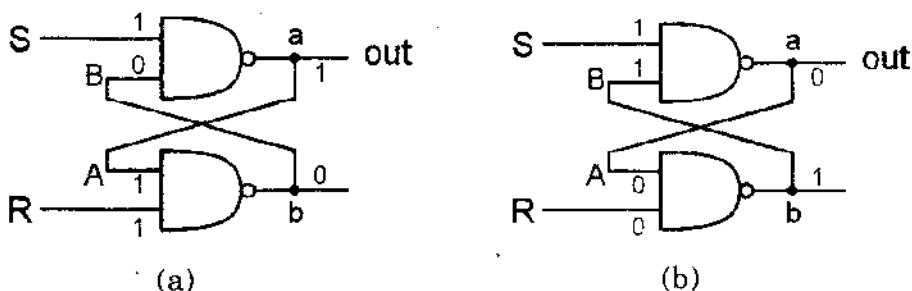
Một phần tử lưu trữ cơ bản là mạch cài R-S, nó có thể lưu trữ một bit thông tin. Mạch cài này có cấu trúc bằng hai cổng NAND 2 đầu vào, hai cổng NAND này được nối với nhau sao cho đầu ra của cái này được nối vào đầu vào của cái kia, hai đầu vào còn lại S và R thường được giữ ở mức logic 1 khi mạch cài đang giữ thông tin. Hình 3.18 trình bày hai trạng thái lưu trữ bit của mạch R-S, hình 3.18a cho bit 1, và 3.18b cho bit 0.



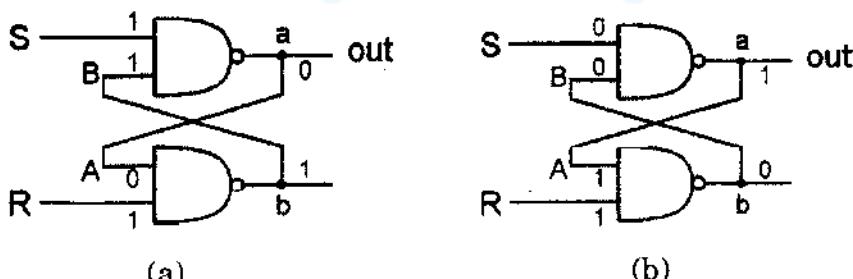
Hình 3.18 Mạch cài S-R

Mạch R-S làm việc như sau: đầu tiên chúng ta bắt đầu với trạng thái lặng khi cả hai đầu vào S và R có mức logic 1. Ở hình 3.18a, đầu ra a (tức bit đang lưu giữ) là 1, nên đầu vào A bằng 1. Cổng NAND dưới có hai đầu vào R và A bằng 1, nên đầu ra b bằng 0. Khi đó, đầu vào B cũng bằng 0, cổng NAND trên có hai đầu vào S = 1 và B = 0, nên đầu ra a bằng 1, tức mạch đã giữ được thông tin. Tương tự như vậy, hình 3.18b trình bày trạng thái lặng đang giữ thông tin là bit 0 của mạch cài R-S. Đầu tiên, giả sử đầu ra a đang là bit 0, làm cho đầu vào A cũng là 0. Cổng NAND dưới có hai đầu vào là A = 0, R = 1, nên đầu ra b là 1. Điều này kéo theo đầu vào B = 1. Cổng NAND trên có hai đầu vào S = B = 1, nên đầu ra a = 0. Tức mạch cài đang chứa bit 0.

Khi mạch cài đang chứa thông tin, tức $S = R = 1$, nếu ta muốn lưu bit khác vào cho nó, ta có thể đổi một trong hai đầu vào S (nếu muốn chứa bit 1) hoặc R (nếu muốn chứa bit 0) từ 1 qua 0 trong khi vẫn giữ nguyên đầu vào kia là 1. Hình 3.19a cho thấy bit đang chứa là 1, khi đổi R từ 1 qua 0, trong khi vẫn giữ nguyên $S = 1$, thì đầu tiên cổng NAND dưới có hai đầu vào $R = 0, A = 1$ (bit đang chứa, và cần đổi từ a), nên đầu ra của nó $b = 1$; kéo theo đầu vào $B = 1$. Cổng NAND trên có hai đầu vào $S = B = 1$, nên đầu ra $a = 0$, đây là trạng thái bit mà ta cần mạch cài chứa như trong hình 3.19b. Tương tự, hình 3.20 trình bày quá trình đổi bit chứa từ 0 qua 1 bằng cách thay đổi trạng thái đầu vào S từ 1 qua 0, trong khi vẫn giữ nguyên trạng thái đầu vào $R = 1$. Hình 3.20a cho thấy bit đang chứa là 0, khi đổi S từ 1 qua 0, trong khi vẫn giữ nguyên $R = 1$, thì đầu tiên cổng NAND trên có hai đầu vào $S = 0, B = 1$ (not bit đang chứa, và cần đổi từ b), nên đầu ra của nó $a = 1$; kéo theo đầu vào $A = 1$. Cổng NAND dưới có hai đầu vào $R = A = 1$, nên đầu ra $b = 0$, mạch cài ổn định và giữ bit 1. Đây là trạng thái bit mà ta cần mạch cài chứa như trong hình 3.20b.



Hình 3.19 Mạch cài R-S đổi trạng thái bit lưu trữ từ 1 qua 0



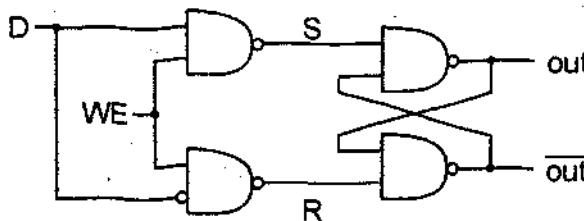
Hình 3.20 Mạch cài R-S đổi trạng thái bit lưu trữ từ 0 qua 1

Như vậy mạch cài có ba trạng thái cho cặp tín hiệu nhập S và R là 11 khi giữ thông tin, 10 khi muốn chứa bit 0, và 01 nếu muốn chứa bit 1. Không thể có trạng thái 00, khi đó mạch hoạt động không ổn định. Để tránh khả năng mạch bị trạng thái SR = 00, người ta đưa ra mạch cài D.

3.4.2 Mạch cài D (*D latch*)

Hình 3.21 cho thấy mạch cài D gồm một mạch cài S-R và hai cổng NAND thêm vào, hai cổng này cho phép mạch cài S-R lấy giá trị từ D chỉ khi tín hiệu WE (Write Enable) được xác lập, là 1.

Khi WE = 0, cả hai đầu ra S và R đều bằng 1, làm cho bộ cài R-S giữ được trạng thái bit hiện có. Khi WE được đặt là 1, chỉ có một trong hai đầu ra S hoặc R bằng 0 tùy vào tín hiệu D. Nếu D bằng 1, thì S được đặt về 0 vì hai đầu vào cổng NAND phía trên: D = WE = 1, còn R vẫn bằng 1 vì hai đầu vào cổng NAND phía dưới: not D = 0, WE = 1; khi đó bộ cài R-S sẽ chứa bit 1 ở đầu ra out của nó như mục 3.4.1 đã trình bày. Tương tự, khi D là 0, khi đó bộ cài R-S sẽ chứa bit 0 ở đầu ra out.

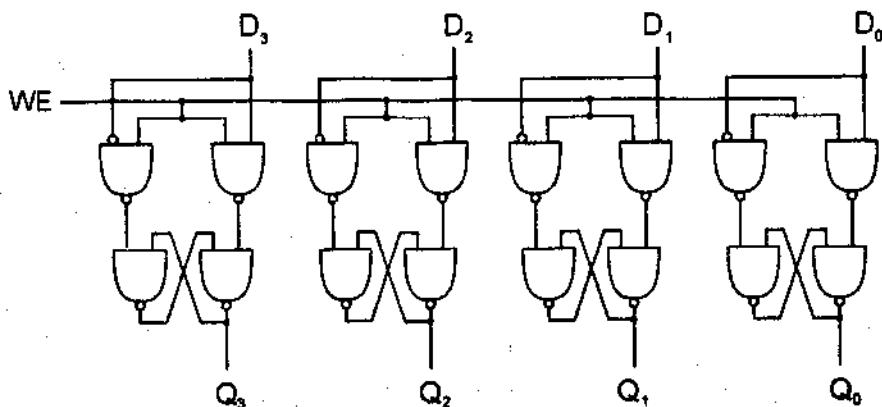


Hình 3.21 Mạch cài D

3.4.3 Thanh ghi (Register)

Như đã trình bày trong chương 2, các thao tác tính toán thường diễn ra trên số nhị phân có chiều dài lớn hơn 1 bit. Sau này, khi học máy tính LC3, chúng ta sẽ làm việc trên các giá trị 16 bit. Để có thể chứa được các giá trị dài như vậy, cấu trúc được sử dụng là thanh ghi trong các CPU. Tùy vào chiều dài toán hạng sử dụng mà ta có thanh ghi có chiều dài thích hợp. Trong LC3, chúng ta cần nhiều thanh ghi có chiều dài 16 bit như PC, IR, MAR, ..., nhưng cũng cần một số thanh ghi có chiều dài 1 bit như N, Z, và P.

Hình 3.22 là thanh ghi bốn bit được tạo từ bốn bộ cài D. Giá trị bốn bit được chứa trong thanh ghi là Q₃, Q₂, Q₁, và Q₀. Giá trị D₃, D₂, D₁, và D₀ được ghi vào thanh ghi khi WE được xác lập, bằng 1.



Hình 3.22 Thanh ghi bốn bit

Khi cần truy xuất chuỗi bit trong thanh ghi Q, ta dùng quy ước Q[chỉ số trái: chỉ số phải]. Cũng cần lưu ý rằng, các bit trong số nhị phân được đánh số theo trọng số của nó, như vậy bit bên phải cùng sẽ là bit [0], và việc đánh số sẽ tiếp tục từ phải qua trái. Nếu số nhị phân có n bit, thì bit ngoài cùng bên trái sẽ là bit [n-1]. Ví dụ, trong mẫu Q 16 bit sau:

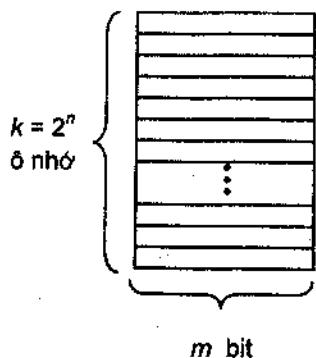
0011101100011110

thì bit Q[15] là 0, bit Q[14] là 0, bit Q[13] là 1, Q[15 :13] là 001.

3.5 BỘ NHỚ (MEMORY)

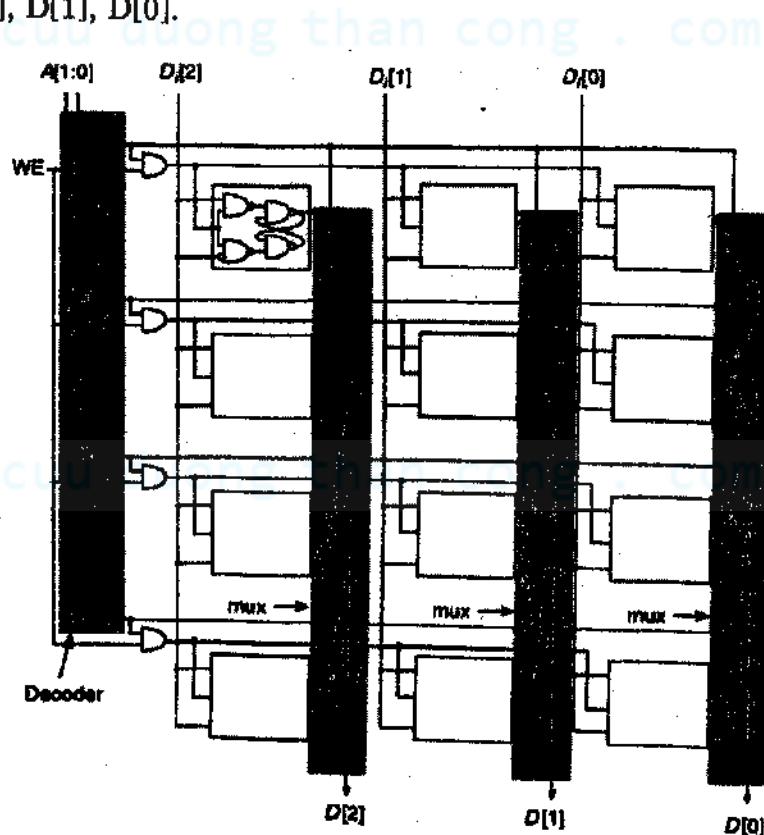
Sau khi có mạch cài để chứa thông tin, ta có thể mô tả một trong những cấu trúc quan trọng trong máy tính là bộ nhớ. Bộ nhớ gồm nhiều ô nhớ (*memory location*), còn được gọi là từ, mỗi ô nhớ có địa chỉ riêng. Mỗi ô nhớ có nhiều bit, và mỗi bit là một bộ cài D. Như vậy, bộ nhớ được đặc trưng bởi hai yếu tố: số ô nhớ và số bit trong mỗi ô nhớ như hình 3.23 trình bày.

Vì mỗi ô nhớ có địa chỉ riêng, nên nếu ta có n bit địa chỉ cho tổ hợp địa chỉ nhị phân của ô nhớ thì dung lượng bộ nhớ sẽ là $k = 2^n$ ô nhớ. Ví dụ, khi nói máy tính có dung lượng 4 GB, có nghĩa là máy tính đó có số ô nhớ 4×2^{30} và mỗi ô nhớ là một byte, tức bộ nhớ máy tính đó có hơn 4 tỷ byte.



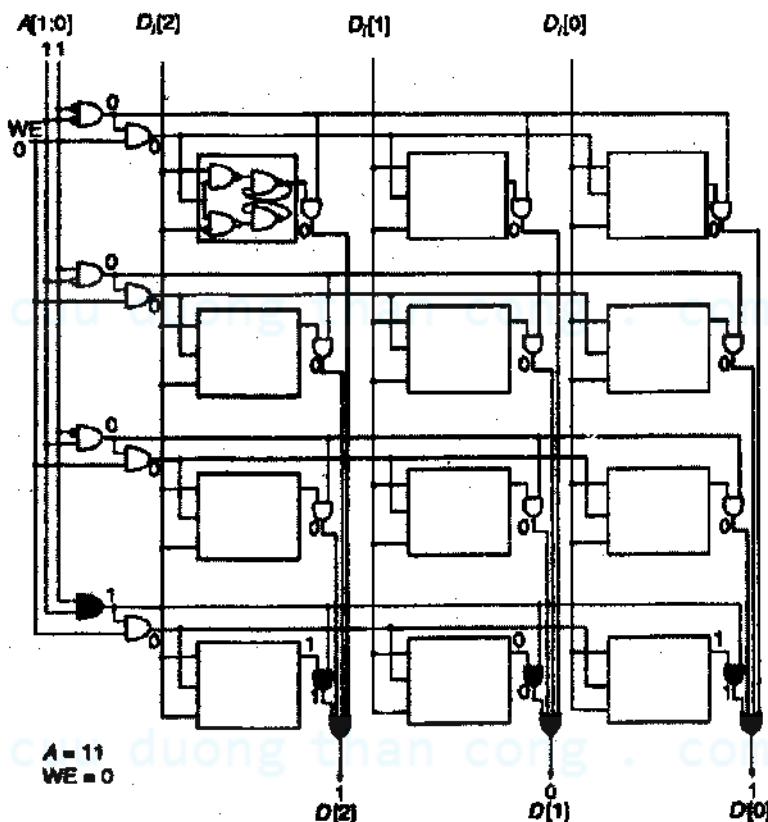
Hình 3.23 Hai đặc trưng của bộ nhớ

Hình 3.24 trình bày bộ nhớ 4 từ (word), mỗi từ có 3 bit. Để truy xuất từng từ nhớ ta cần một bộ decoder hai bit đầu vào $A[1:0]$, xác lập chính xác cho một đầu ra tương ứng với từ nhớ có địa chỉ tổ hợp trong $A[1:0]$. Tín hiệu WE cho phép thao tác ghi các bit từ $D_i[2]$, $D_i[1]$, $D_i[0]$ vào các bộ cài D . Ở đầu ra của các bit cùng trọng số, tức các bộ cài D cùng cột, có các bộ phân kênh để chọn tín hiệu xuất ra các $D[2]$, $D[1]$, $D[0]$.



Hình 3.24 Bộ nhớ 4 từ 3 bit

Khi muốn đọc dữ liệu từ bộ nhớ (Hình 3.25), ta cần biết địa chỉ ô nhớ cần truy xuất. Ví dụ, muốn đọc dữ liệu từ ô nhớ 3 (tức 11 nhị phân), khi đó $A[1:0] = 11$ nên cỗng AND tô đen tương ứng sẽ tích cực, cho phép mở các cỗng AND ở đầu ra bộ cài D tương ứng trong bộ phân kêt, cho phép các bit đang được chứa trong các bộ cài D được truyền ra ngoài $D_i[2]$, $D_i[1]$, $D_i[0]$ qua các cỗng OR. Lưu ý, trong quá trình đọc thông tin, tín hiệu WE = 0, nên các cỗng AND ở các đầu ra của bộ decoder điều khiển quá trình ghi bị đóng, các bit không thể từ $D_i[2]$, $D_i[1]$, $D_i[0]$ vào các bộ cài D được.

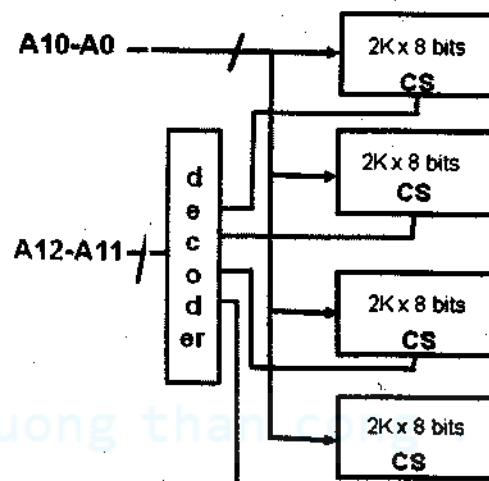


Hình 3.25 Đọc dữ liệu ở ô nhớ 3 trong bộ nhớ 4 từ 3 bit

Các mạch logic như trên được gọi là các chip nhớ. Mỗi chip nhớ sẽ có dung lượng và cấu trúc cụ thể. Khi muốn ghép các chip nhớ này lại để tạo một bộ nhớ có dung lượng lớn hơn, có chiều dài từ mong muốn, chúng ta phải sử dụng các bộ giải mã để cung cấp các tín hiệu chọn chip (chip select) thích hợp. Hãy xét hai ví dụ cụ thể sau.

Ví dụ 3.2 Thiết kế bộ nhớ 8K bằng 4 chip 2K x 8 bit.

Với 4 chip nhớ này, mỗi chip nhớ có dung lượng 2K, nên số đường địa chỉ cần để mã hóa cho mỗi ô nhớ trong từng chip là 2^{11} (=2K), tức từ A₀-A₁₀. Hơn nữa, tổng dung lượng là 8K, tức cần 2^{13} đường địa chỉ, tức từ A₀-A₁₃. Như vậy, các đường địa chỉ A₁₁-A₁₂ sẽ được dùng để giải mã chọn chip (CS-Chip select) như trong hình 3.26.

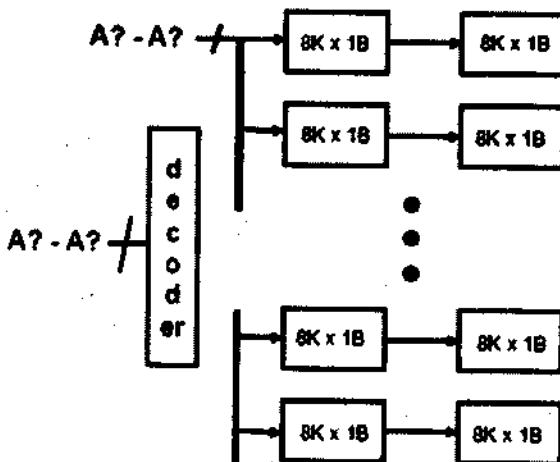


Hình 3.26 Bộ nhớ 8K từ 4 chip 8 bit

Ví dụ 3.3 Thiết kế bộ nhớ 64K x16 bit bằng 16 chip 8K x 8 bit.

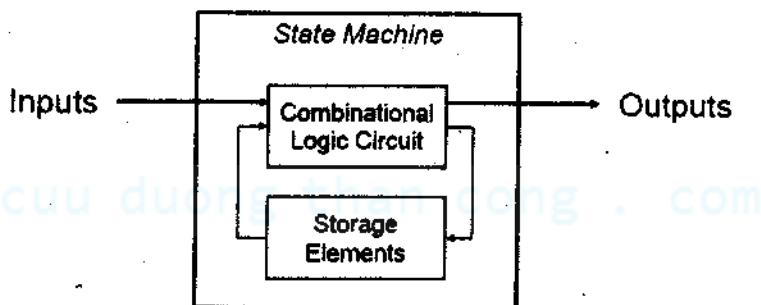
Trong ví dụ này, mỗi từ nhớ trong bộ nhớ có độ dài từ 16 bit (2B), nên để có bộ nhớ 64K như vậy, ta cần phải có 16 chip nhớ 8K x 8 bit. Tổng dung lượng là 64K (2^{16}), nên ta cần 16 đường địa chỉ, từ A₀-A₁₅. Mỗi chip nhớ có dung lượng 8K (=2¹³), nên số đường địa chỉ để giải mã cho mỗi tế bào nhớ là 13, tức từ A₀-A₁₂. Như vậy, số đường địa chỉ để giải mã chọn chip là A₁₃- A₁₅. Lưu ý, ta sử dụng thuật ngữ tế bào nhớ (memory cell) để chỉ đó là thành phần của từ nhớ (word) mà ta cần thiết kế. Trong trường hợp này, một tế bào nhớ là một byte. Độc giả hãy hoàn thành hình 3.27 trên với các ý đã nêu để hiểu rõ vấn đề.

Trong những trường hợp phức tạp hơn, như khi thiết kế bộ nhớ với các chip không cùng dung lượng, ta cần phải viết ra bảng phân bổ địa chỉ cụ thể cho từng chip, rồi sau đó mới chọn các đường địa chỉ để giải mã chip.

**Hình 3.27** Bộ nhớ $64K \times 16$ bit từ 16 chip 8 bit

3.6 MẠCH TUẦN TỰ (SEQUENTIAL LOGIC CIRCUIT)

Không như mạch tổ hợp, đầu ra của mạch tuần tự không chỉ phụ thuộc vào đầu vào hiện tại mà còn phụ thuộc vào trạng thái hiện tại của các phần tử nhớ trong mạch. Thông tin nhị phân đang có trong các phần tử nhớ của mạch xác định trạng thái của mạch ở bất kỳ thời điểm nào trong quá khứ. Như vậy, mạch tuần tự tiêu biểu sẽ gồm hai thành phần là mạch tổ hợp và các phần tử nhớ để trữ thông tin là trạng thái của mạch. Hình 3.28 minh họa cho mạch dạng này.

**Hình 3.28** Sơ đồ khối của mạch logic tuần tự

Trong hình 3.28 ta thấy trạng thái hiện thời của mạch được đưa vào lưu trữ trong các phần tử nhớ, và trở thành đầu vào thêm cho mạch tổ hợp của mạch cho lần khảo sát kế tiếp. Thao tác này gọi là hồi tiếp (feedback).

Mạch logic tuần tự được dùng để thực hiện một trong những cơ chế quan trọng là máy hay bộ điều khiển trạng thái hữu hạn (*finite state machine*). Ví dụ, một bộ điều khiển đèn giao thông bật đèn đỏ, vàng, hay xanh tùy thuộc vào đèn hiện thời đang sáng (thông tin trong quá khứ) và thông tin nhập từ các bộ cảm ứng về xe trên đường và các thiết bị quang đang điều khiển lưu lượng xe.

Chúng ta sẽ thấy trong chương 4 mô hình von Neumann của một máy tính trong đó bộ điều khiển có trạng thái hữu hạn là trái tim của máy tính. Nó điều khiển quá trình xử lý thông tin của máy tính do quá trình này phải qua các bước theo trình tự quy định.

3.6.1 Khái niệm về trạng thái

Trạng thái là một khái niệm rất quan trọng trong kỹ thuật máy tính và trong nhiều ngành kỹ thuật khác. Trạng thái của một hệ thống là một bức tranh chụp nhanh mà ở đó tất cả các thành phần thích hợp đều được biểu diễn một cách rõ ràng.

Nghĩa là: trạng thái của một hệ thống là một bức ảnh chụp nhanh tất cả các phần tử của hệ thống tại thời điểm bức tranh được chụp.

Trong thực tế cuộc sống hằng ngày, ta có rất nhiều khái niệm về trạng thái như trạng thái của một trận đá banh có thể được biểu diễn bằng bảng tỷ số trận đấu, thời gian còn lại của hiệp đấu, tên hai đội,... ; hoặc trong trò chơi carô, trạng thái của trò chơi giữa hai người chơi là hình ảnh về tiến trình ở đó người chơi thứ nhất đi được mẩy 'X', người chơi thứ hai đi được mẩy 'O' ở các vị trí nào,....

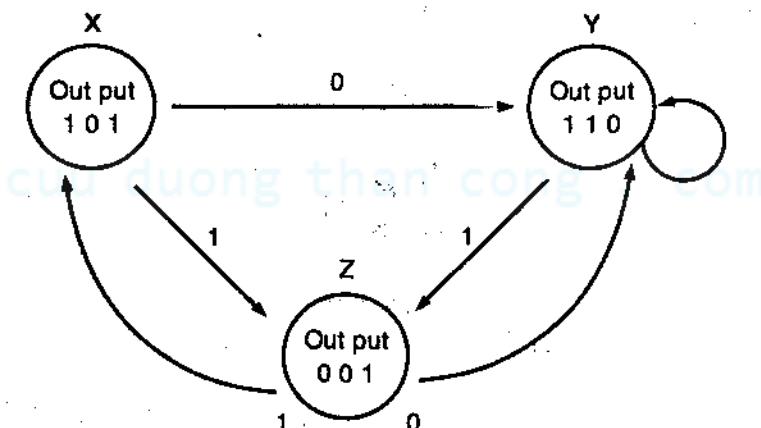
3.6.2 Máy trạng thái hữu hạn

Chúng ta đã biết một trạng thái là một bức tranh chụp nhanh thể hiện tất cả các phần tử có giá trị của hệ thống ở một thời điểm đặc biệt nào đó. Ở một thời điểm khác, hệ thống đó có thể ở trong một trạng thái khác. Việc một hệ thống thay đổi từ trạng thái này sang trạng thái khác với một số lượng trạng thái xác định hữu hạn biểu diễn tiến trình làm việc của hệ thống. Lúc này, ta nói hệ thống là một máy hay bộ điều khiển trạng thái hữu hạn (*finite state machine*).

Một máy trạng thái hữu hạn bao gồm năm thành phần:

1. Một số hữu hạn các trạng thái
2. Một số hữu hạn các đầu vào từ bên ngoài
3. Một số hữu hạn các tín hiệu xuất (hay đầu ra) ra bên ngoài
4. Một chỉ định rõ tất cả các chuyển trạng thái
5. Một chỉ định rõ thành phần mỗi giá trị đầu ra.

Tập hợp các trạng thái của hệ thống biểu diễn tất cả các tình huống có khả năng diễn ra của hệ thống. Mỗi một chuyển trạng thái mô tả cái cần có (hay cần thực hiện) để hệ thống chuyển từ trạng thái này qua trạng thái khác. Để biểu diễn rõ ràng sự hoạt động của một máy trạng thái, chúng ta sử dụng sơ đồ trạng thái. Hình 3.29 là một ví dụ về sơ đồ trạng thái. Một sơ đồ trạng thái là một hình vẽ gồm một tập các hình tròn, mỗi hình tròn tương ứng với một trạng thái, và một tập các đường nối giữa các trạng thái, mỗi đường nối là một cung được vẽ dưới dạng mũi tên thẳng hay cong hình cánh cung. Mỗi cung xác định sự chuyển từ trạng thái này sang trạng thái khác. Đi theo cung, trạng thái ở đầu cung là trạng thái hiện thời, còn trạng thái mà mũi tên chỉ tới là trạng thái kế tiếp phù hợp với chuyển trạng thái tương ứng. Máy trạng thái được biểu diễn ở hình 3.29 bao gồm ba trạng thái, với sáu chuyển trạng thái. Chú ý rằng không có chuyển trạng thái từ trạng thái Y sang trạng thái X.



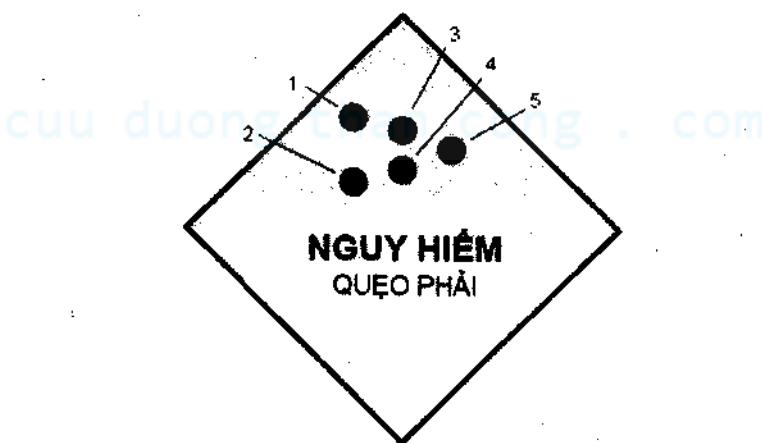
Hình 3.29 Sơ đồ trạng thái

Thông thường, từ một trạng thái hiện thời ta có nhiều chuyển trạng thái tới các trạng thái kế tiếp. Chuyển trạng thái phụ thuộc vào các giá trị đầu vào từ bên ngoài. Như trong hình 3.29, nếu trạng thái hiện thời là X và đầu vào là 0, thì trạng thái kế tiếp là Y, nhưng nếu đầu vào có giá trị bằng 1, thì trạng thái kế tiếp lại là Z. Tóm lại, trạng thái kế tiếp được quyết định bởi tổ hợp của trạng thái hiện thời và đầu vào hiện thời.

Mỗi trạng thái hiện thời quyết định giá trị đầu ra ở trạng thái đó. Trong hình 3.29, đầu ra là 101 khi hệ thống ở trạng thái X, đầu ra là 110 khi hệ thống ở trạng thái Y, và là 001 khi hệ thống ở trạng thái Z.

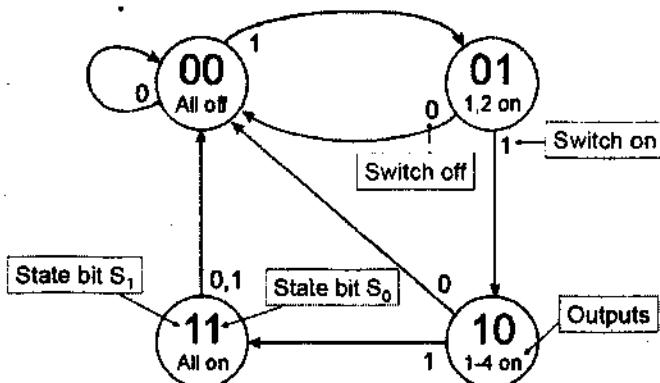
3.6.3 Một ví dụ về hiện thực một máy trạng thái hữu hạn

Trong phần này, chúng ta xét việc thiết kế một mạch logic tuần tự để hiện thực một máy trạng thái hữu hạn. Đó là bộ điều khiển báo nguy hiểm trong giao thông mà ta thường gặp trên đường, "Nguy hiểm, quẹo phải". Để bảng này được người đi đường chú ý, người ta thường thiết kế năm bóng đèn 1, 2, 3, 4, 5 như hình 3.30. Các bóng đèn này sáng theo kiểu chu kỳ đầu tiên đèn 1 và 2 sáng, chu kỳ kế thêm đèn 3, 4 sáng (tức các đèn 1, 2, 3, 4 cùng sáng), chu kỳ kế tiếp nữa thêm đèn 5 sáng (tức tất cả đèn 1, 2, 3, 4, 5 đều sáng), và chu kỳ sau nữa tắt cả đèn đều tắt, thường một chu kỳ như vậy kéo dài $\frac{1}{2}$ giây. Quá trình lặp lại như thế, điều này tạo ra hình mũi tên có chớp tắt chỉ theo hướng cần thiết.



Hình 3.30 Tín hiệu báo nguy hiểm trong giao thông

Hình 3.31 là sơ đồ trạng thái của bộ điều khiển báo hiệu nguy hiểm giao thông. Ta thấy có bốn trạng thái, mỗi trạng thái tương ứng với từng chu kỳ đèn sáng. Các chuyển trạng thái là các tình huống bật công tắc (switch on = 1) hay tắt công tắc (switch off = 0).



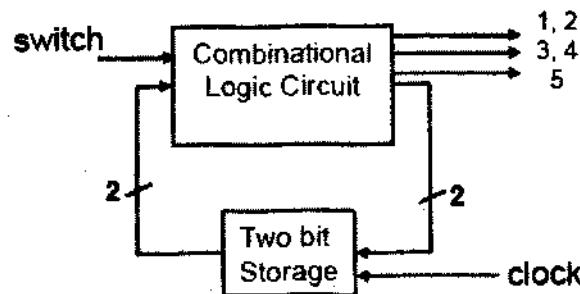
Hình 3.31 Sơ đồ trạng thái của bộ điều khiển báo hiệu nguy hiểm giao thông

Hình 3.32 dưới đây cho thấy việc thực hiện mạch logic tuần tự là máy trạng thái hữu hạn ở hình 3.30. Hình 3.32a là sơ đồ khối tương tự như hình 3.28. Chỉ có một đầu vào, đó là tín hiệu từ công tắc xác định xem đèn có sáng hay không. Có ba đầu ra, một để điều khiển cho đèn 1 và 2 sáng, một để điều khiển cho đèn 3 và 4 sáng, và một để điều khiển đèn 5. Vì ta có tất cả bốn trạng thái, nên ta cần dùng hai bit để mã cho mỗi trạng thái. Hai bit đó, S_1 và S_0 , được lưu trong hai phần tử lưu trữ trong (*storage element*) 1 và 0 tương ứng để giữ vết của trạng thái của hệ thống, vết này được xác định từ tình trạng trong quá khứ của bộ báo nguy hiểm.

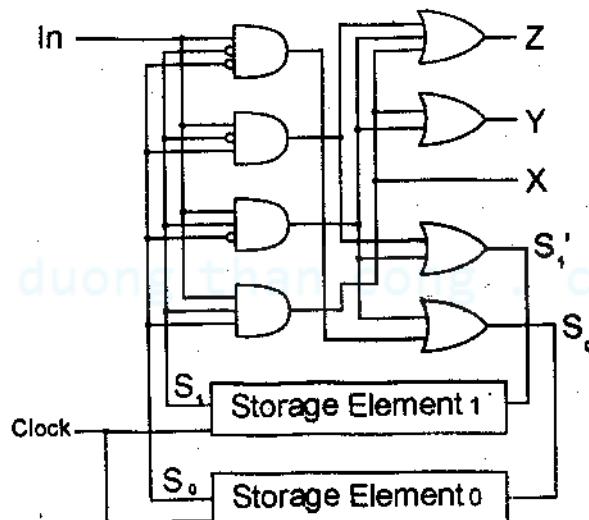
Bây giờ ta hãy xét từng phần trong bộ điều khiển.

* Mạch logic tổ hợp

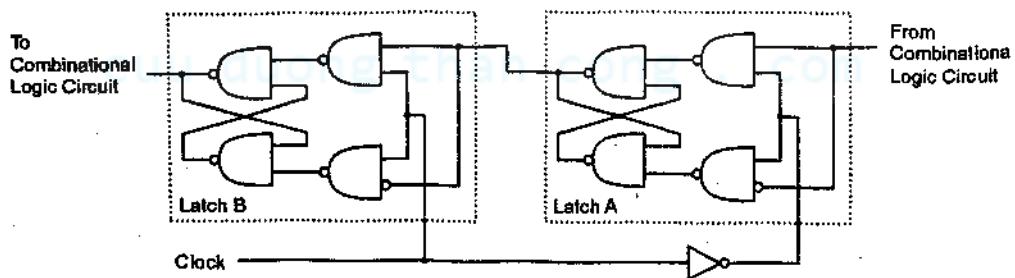
Hình 3.32b cho ta mạch logic tổ hợp để hiện thực hóa bộ điều khiển báo nguy giao thông. Hai tập hợp đầu ra của mạch logic tuần tự đảm bảo cho bộ điều khiển hoạt động hợp lý: một tập hợp đầu ra bên ngoài để điều khiển đèn, một tập hợp đầu ra hồi tiếp để xác định đầu vào cho hai phần tử nhớ đang giữ vết trạng thái của hệ thống.



(a) Sơ đồ khối



(b) Mạch logic tổ hợp



(c) Một phần tử nhớ

Hình 3.32 Hiện thực mạch logic tuần tự của biển báo giao thông

Trước tiên, chúng ta hãy xem xét các đầu ra điều khiển đèn. Như đã biết, chỉ có ba đầu ra cần thiết để điều khiển năm đèn, đó là X (điều khiển đèn 5), Y (điều khiển đèn 3 và 4) và Z (điều khiển đèn 1 và 2). Có bốn trạng thái sáng cho năm đèn: 00 khi tắt cả đèn tắt (tức $ZYX = 000$), 01 khi hai đèn 1 và 2 sáng (tức $Z = 1$), 10 khi bốn đèn 1, 2, 3, 4 sáng (tức $ZY = 11$), và 11 khi tất cả các đèn 1-5 cùng sáng (tức $ZYX = 111$). Trạng thái hiện thời của hệ được ký hiệu là S_1S_0 , trạng thái kế của hệ là $S'_1S'_0$ như bảng sự thật ở hình 3.33.

		Đầu ra			Trạng thái kế: $S'_1S'_0$	
		(phụ thuộc vào trạng thái: S_1S_0)			(tùy thuộc trạng thái và đầu vào)	
			Đèn 1 và 2		Công tắc	
				Đèn 3 và 4	In	S_1
						S_0
				Đèn 5		S'_1
			Z	Y		S'_0
S_1	S_0					
0	0	0	0	0	0	0
0	1	1	0	0	1	0
1	0	1	1	0	1	1
1	1	1	1	1	0	0

		Công tắc		
In	S_1	S_0	S'_1	S'_0
0	x	x	0	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	0	0

Khi In=0, trạng thái kế là 00

Hình 3.33 Bảng sự thật về hoạt động của bộ báo nguy giao thông

Trong bảng sự thật hình 3.33, công tắc In là đầu vào, khi nó tắt (bằng 0) bất chấp trạng thái hiện thời ra sao đi nữa, tức $S_1S_0 = xx$, ta luôn có trạng thái kế tiếp là $S'_1S'_0 = 00$, tức tắt cả các đèn đều tắt do mạch không hoạt động.

Từ hai bảng sự thật trên hình 3.33, ta thấy đèn 5 sáng, tức $X = 1$, khi công tắc đầu vào mở ($In = 1$), và bộ điều khiển ở trạng thái $S_1S_0 = 11$, như vậy đầu ra điều khiển đèn 5 sẽ là đầu ra của cổng AND từ ba bit này. Đèn 3, 4 sáng, tức $Y = 1$, khi công tắc đầu vào mở ($In = 1$), và bộ điều khiển ở một trong hai trạng thái $S_1S_0 = 11$ hoặc 10, nên đầu ra Y sẽ là OR của hai đầu vào từ hai kết quả AND InS_1S_0 của hai trạng thái trên: 111 hoặc 110, tức $Y = (InS_1S_0 = 111) \text{ OR } (InS_1S_0 = 110)$.

Độc giả có thể tự giải thích đầu ra Z để điều khiển đèn 1 và 2 một cách tương tự.

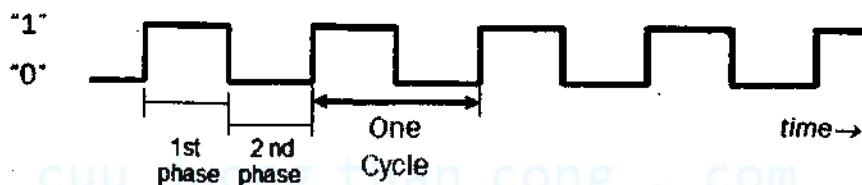
Bây giờ, chúng ta hãy xem xét các đầu ra điều khiển các phần tử nhớ. Từ bảng sự thật, ta thấy phần tử nhớ 0 là 1 chỉ khi công tắc

là mở, tức $In = 1$ và trạng thái kế tiếp là 01 hay 11, điều này chỉ xảy ra khi trạng thái hiện thời phải là 00 hay 10, như vậy đầu ra S_0' sẽ là OR của hai đầu ra từ hai kết quả AND của hai trạng thái: 100 hoặc 110, tức $S_0' = (InS_1S_0 = 100) \text{ OR } (InS_1S_0 = 110)$.

Một cách hoàn toàn tương tự ta cũng có đầu ra S_1' : $S_1' = (InS_1S_0 = 101) \text{ OR } (InS_1S_0 = 110)$

* Phần tử nhớ

Thành phần còn lại trong bộ điều khiển đèn báo hiệu nguy hiểm giao thông là mạch logic của hai phần tử nhớ. Hình 3.32c trình bày cấu trúc của một phần tử nhớ gồm cặp hai mạch cài D, và được gọi là mạch lật chủ tớ (*master-slave flip-flop*).



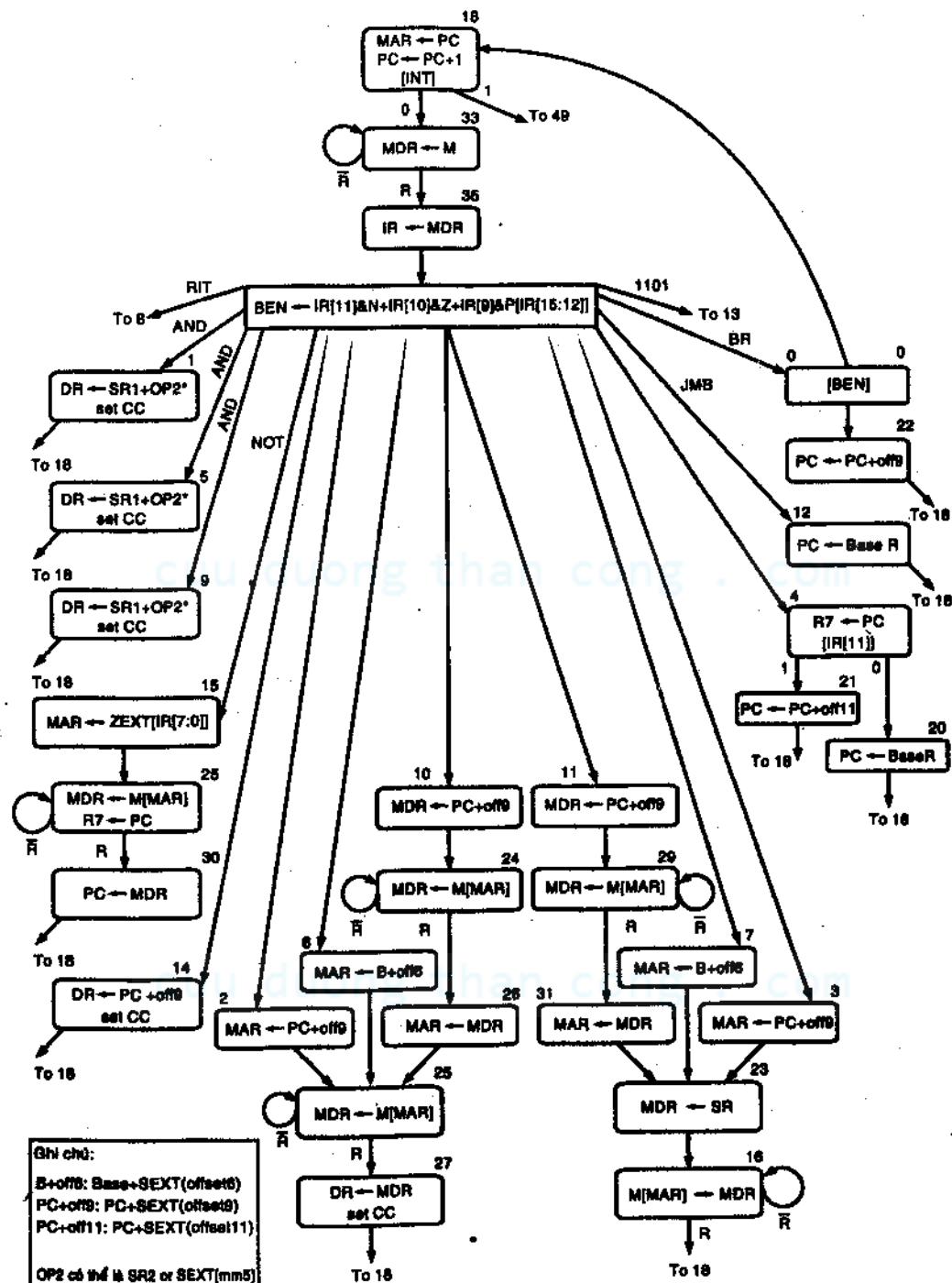
Hình 3.34 Tín hiệu xung clock (CK)

Theo hình 3.34, trong nửa đầu của chu kỳ xung clock (phase 1) xung CK = 1, ta không thể thay đổi giá trị bit đã lưu trong mạch cài A từ trạng thái trước đó, mà bit trạng thái này sẽ được chuyển sang cho mạch cài B và trở thành trạng thái hiện thời. Ở nửa sau của chu kỳ xung clock (phase 2), xung CK = 0, giá trị bit từ đầu ra của mạch điều khiển giao thông (tức S_1' hoặc S_0') sẽ được lưu vào mạch cài A để trở thành trạng thái kế tiếp ở chu kỳ sau. Như vậy, ta thấy mạch lật chủ tớ cho phép trạng thái hiện thời được giữ nguyên trong toàn chu kỳ, trong khi trạng thái kế tiếp được tạo bởi mạch logic tuần tự để thay đổi mạch cài A trong phase 2 của chu kỳ, và vì vậy bit này sẽ được chuyển vào mạch cài B lúc bắt đầu chu kỳ kế tiếp.

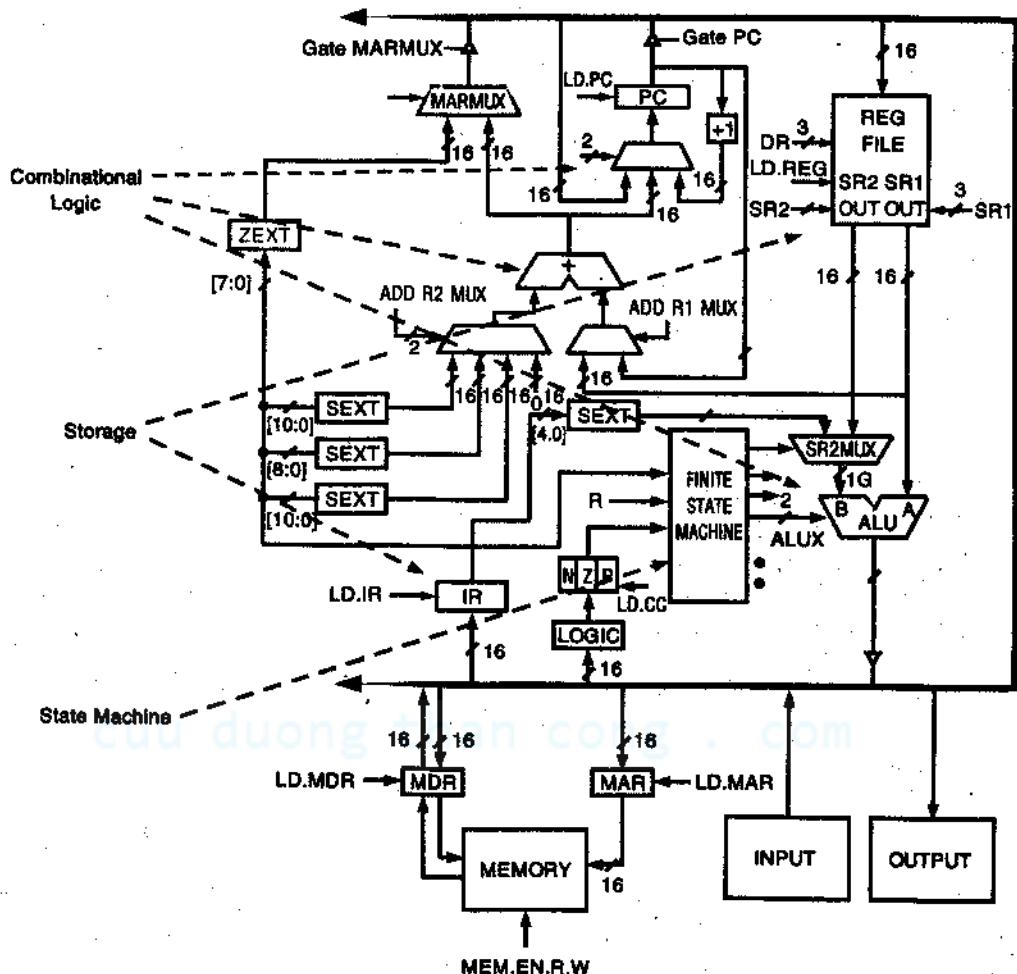
3.7 ĐƯỜNG TRUYỀN DỮ LIỆU LC 3

Trong chương 4, chúng ta sẽ học một máy tính đơn giản, gọi là LC3, và bạn sẽ có dịp để viết các chương trình máy tính để thực thi trên nó. Trước hết, chúng ta có thể thấy đã là một máy tính, nên LC3 sẽ hoạt động theo một chu trình thực thi lệnh xác định như một máy trạng thái hữu hạn. Hình 3.35 trình bày sơ đồ trạng thái của

qua trình thực thi lệnh kể từ lúc lấy lệnh từ bộ nhớ tới phân tích, giải mã, lấy dữ liệu, thực thi lệnh,...



Hình 3.35 Sự hoạt động của LC3 như một máy trạng thái hữu hạn



Hình 3.36 Đường truyền dữ liệu của máy tính LC3

Để hiểu rõ hơn việc thực thi của một chương trình, người ta đưa ra khái niệm đường truyền dữ liệu (*data path*) của một bộ vi xử lý. Đường truyền dữ liệu thực ra là các mạch logic được trình bày theo chức năng để xử lý thông tin. Do đó nó còn được gọi là vi kiến trúc của bộ vi xử lý. Hình 3.36 sau là đường truyền dữ liệu của máy tính ảo LC3, cũng tương tự như hình 1.14, có nhiều cấu trúc cơ bản tạo thành máy tính và rất quen thuộc với chúng ta như các thanh ghi 16 bit PC, IR, MAR, và MDR. Mỗi đường dây có đường gạch chéo nhỏ kèm theo số 16 biểu diễn 16 đường dây, mỗi dây mang một bit thông tin. N, Z, P là các thanh ghi một bit, chúng có thể được thực hiện bằng các mạch lật chủ từ. Có năm bộ phân kêtch, một bộ cung cấp một giá trị 16 bit cho thanh ghi PC (PCMUX), một bộ cung cấp địa

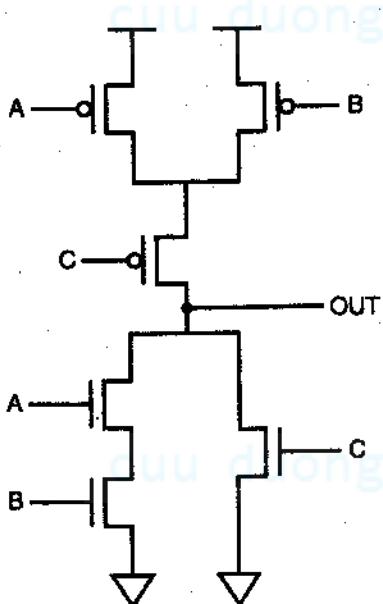
chỉ cho thanh ghi MAR (MARMUX), một bộ để chọn dữ liệu nhập vào đầu B của ALU (SR2MUX), và hai bộ để chọn toán hạng nhập cho bộ cộng 16 bit (ADDR1MUX và ADDR2MUX). Để điều khiển các thành phần trong đường truyền dữ liệu hoạt động, ta cần các vi lệnh với các bit quy định cụ thể. Tập hợp các vi lệnh là một vi chương trình mà trong chương sau chúng ta sẽ hiểu rõ hơn khi học cấp kiến trúc tập lệnh (IAS) của CPU LC3.

BÀI TẬP CUỐI CHƯƠNG

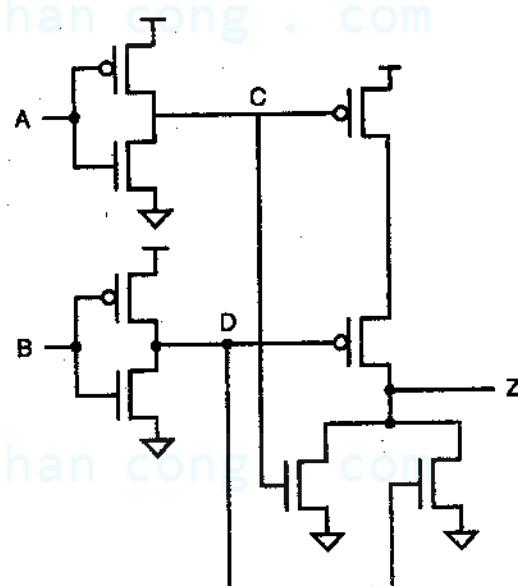
- 3.1** Viết bảng sự thật cho mạch ở cấp transistor như trong hình sau và cho biết chức năng của mạch.

Có thể thay thế các N transistor và P transistor với nhau để có cùng trạng thái đầu ra không? Tại sao?

- 3.2** Cho mạch transistor như hình dưới đây. Hoàn thành bảng sự thật. Cho biết đầu ra Z thể hiện chức năng gì của mạch?



BT 3.1

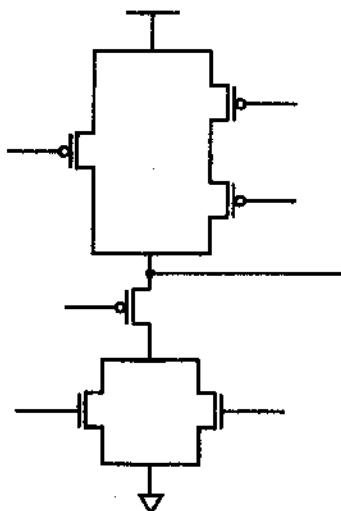


BT 3.2

- 3.3** Mạch transistor dưới đây hiện thực hóa phương trình logic

$$Y = \text{NOT}(\text{A AND}(\text{B OR C}))$$

Cho biết vị trí các đầu vào, đầu ra và hoàn thành bảng sự thật mạch này.



- 3.4 Nếu A và B là các số nguyên không dấu 4 bit, 0111 và 1011, hoàn thành bảng dưới khi dùng bộ cộng full adder để tính tổng của hai số A và B. Kiểm tra kết quả có được sau khi cộng hai giá trị ở dạng thập phân A và B và so sánh với S. Kết quả có giống nhau không? Giải thích.

C_{in}				0
A	0	1	1	1
B	1	0	1	1
S				
C_{out}				

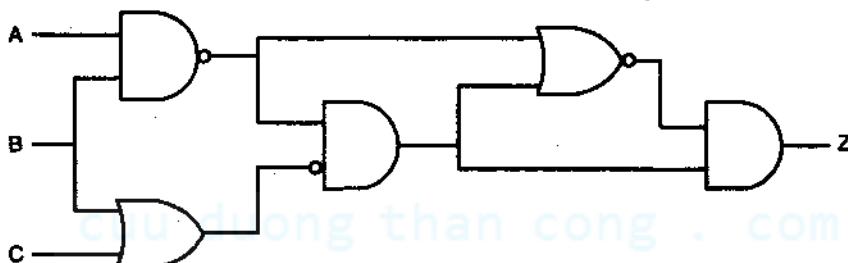
- 3.5 Cho bảng sự thật như dưới đây. Cho biết mạch logic tương ứng và nêu chức năng của mạch này.

Hãy chỉnh lại mạch trên để mạch có thể kiểm tra được trị thập phân đầu vào (tạo từ ba bit A, B, C) là số chẵn.

A	B	C	Z
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

- 3.6 Cho mạch logic như hình dưới đây. Hoàn thành bảng sự thật dưới đây khi đầu vào là ba bit A, B, C và đầu ra là Z.

A	B	C	Z
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	



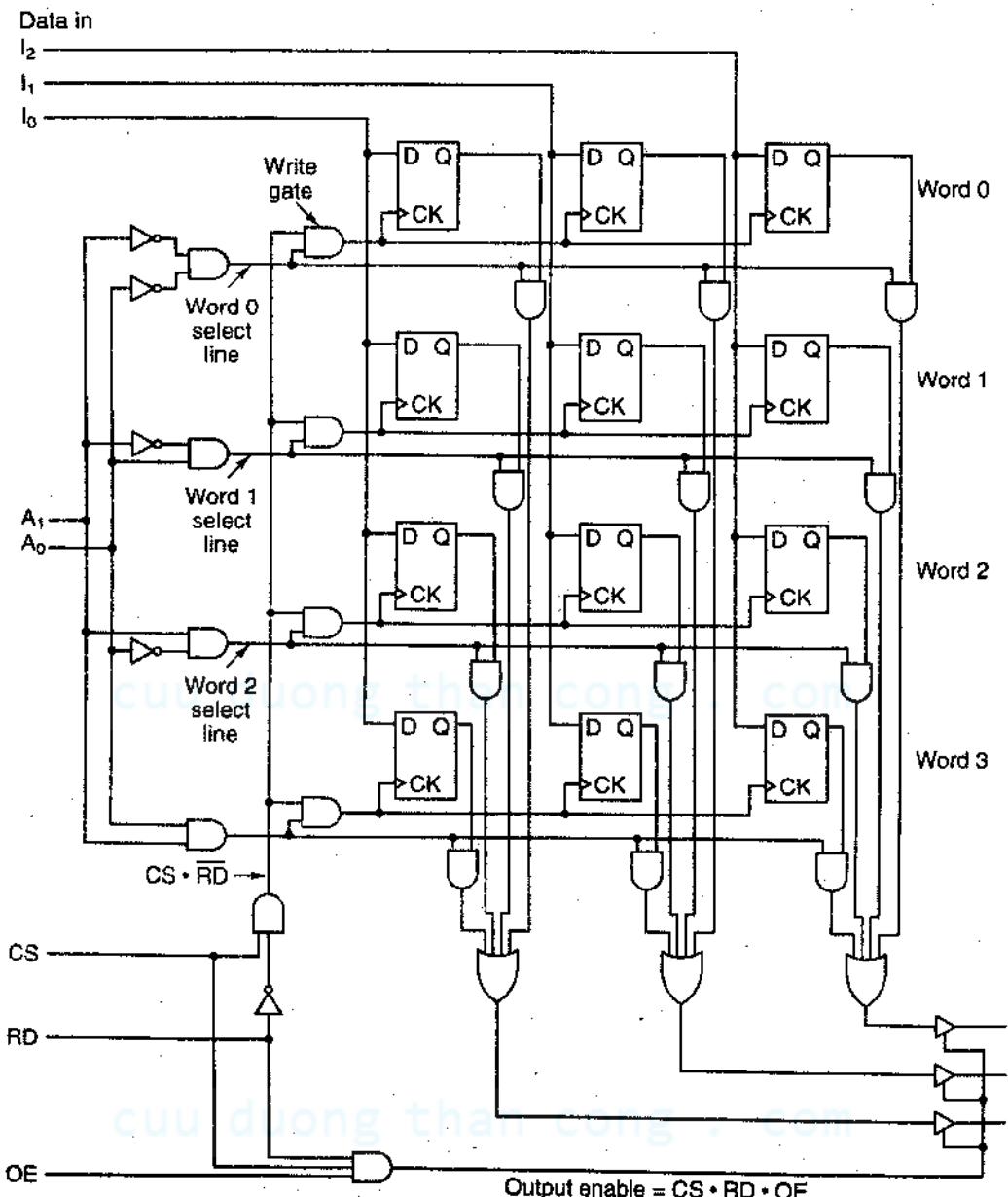
- 3.7 Trên thực tế, một chip nhớ sẽ có cấu trúc gồm các D Flip- Flop như hình sau. Một số tín hiệu được dùng để điều khiển như:

- CS (Chip Select): chọn chip.
- RD (Read): RD = 1: đọc chip, = 0: ghi chip.
- OE (Output Enable): Tín hiệu điều khiển, cho phép xuất dữ liệu ra Data bus.
- Dữ liệu nhập vào chip qua Data in: I2, I1, I0.
- Dữ liệu xuất từ chip qua Data out: O2, O1, O0.

D Flip-Flop được nhập liệu khi xung CK (Clock) chuyển trạng thái từ 0 sang 1.

Cho biết:

1. Không gian địa chỉ của chip nhớ này.
2. Chiều dài từ nhớ của chip nhớ này.
3. Trình bày quá trình ghi dữ liệu vào chip từ Data in.
4. Trình bày quá trình đọc dữ liệu từ chip và xuất ra Data out.



Chương 4

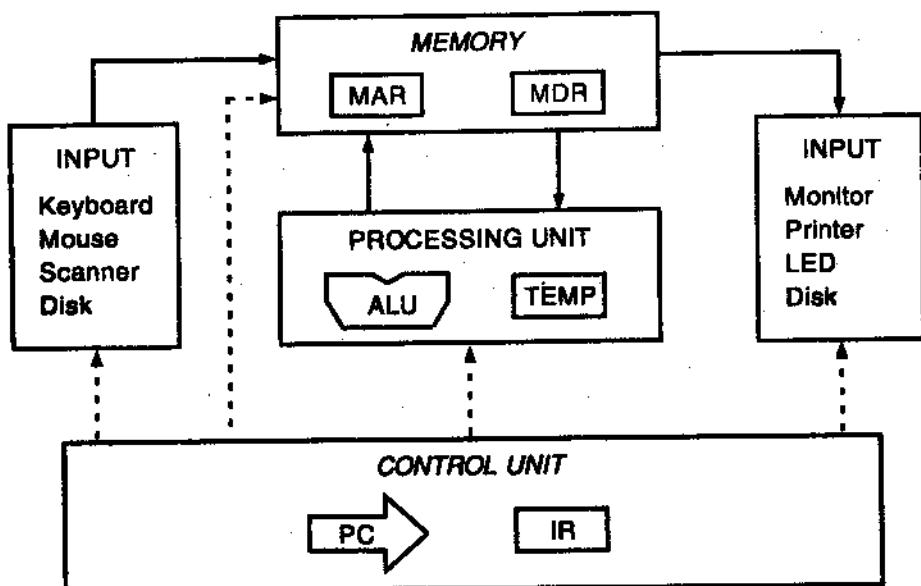
MÔ HÌNH VON NEUMANN VÀ KIẾN TRÚC TẬP LỆNH LC-3

Sau khi đã có khái niệm về tất cả các phần tử logic cơ bản trong chương 3, trong chương này, chúng ta sẽ học cấu trúc mô hình máy tính cơ bản đầu tiên do John von Neumann đề xuất vào 1946. Sau đó, một máy tính có cấu trúc tương ứng là LC-3 với các đặc tính quan trọng và cách viết chương trình ở dạng ngôn ngữ máy LC-3 (*Little Computer 3*) sẽ được đưa ra minh họa ở dạng kiến trúc tập lệnh (ISA - *Instruction Set Architecture*). Như đã được trình bày trong chương 1, ISA là cấp giao tiếp giữa lệnh ở phần mềm và quá trình thực hiện lệnh của phần cứng.

4.1 CÁC THÀNH PHẦN CƠ BẢN

Để thực hiện một nhiệm vụ được thực hiện bởi máy tính chúng ta cần hai thứ: chương trình và máy tính. Một chương trình máy tính gồm một tập hợp các lệnh, mà mỗi lệnh là một phần công việc máy tính cần thực thi. Lệnh (*Instruction*) là phần việc nhỏ nhất trong một chương trình máy tính. Máy tính không thể thực hiện một phần của lệnh được.

Vào năm 1946, John von Neumann đã đưa ra một mô hình máy tính cơ bản để xử lý các chương trình máy tính gồm các bộ phận cơ bản như trong hình 4.1. Có năm bộ phận: bộ nhớ (*memory*), đơn vị xử lý (*processing unit*), thiết bị nhập (*input*), thiết bị xuất (*output*), và một đơn vị điều khiển (*control unit*). Chương trình máy tính được chứa trong bộ nhớ của máy tính. Việc điều khiển thứ tự các lệnh cần thực hiện sẽ do đơn vị điều khiển đảm trách.



Hình 4.1 Sơ đồ khái quát mô hình von Neumann

4.1.1 Bộ nhớ (Memory)

Trong chương 3, chúng ta đã khảo sát một bộ nhớ có $2^2 = 4$ phần tử, mỗi phần tử 3 bit được cấu tạo từ các cổng và bộ cài. Với các máy tính thực ngày nay, bộ nhớ có thể là 2^{30} phần tử, mỗi phần tử có thể lưu trữ 8 bit thông tin. Khi đó, ta nói máy tính đó có không gian địa chỉ là 2^{30} vị trí xác định phân biệt, số bit địa chỉ là 30, mỗi vị trí là một ô nhớ dài 8 bit. Bộ nhớ đó có dung lượng 1 gigabyte (viết tắt là 1 GB). “1 giga” nghĩa là 2^{30} vị trí, và “byte” là từ nhớ dài 8 bit.

Tổng quát, với số bit địa chỉ là k, chúng ta có thể biểu diễn được 2^k phần tử nhớ. Với kiến trúc tập lệnh của máy tính LC-3, chúng ta có không gian địa chỉ là 2^{16} , và mỗi phần tử dài 16 bit.

Có hai thao tác truy xuất bộ nhớ là đọc và ghi. Để đọc thông tin của một ô nhớ, đầu tiên chúng ta cần đặt địa chỉ của ô nhớ đó vào thanh ghi địa chỉ bộ nhớ MAR (*Memory Address Register*), và hỏi thông tin từ bộ nhớ máy tính bằng tín hiệu Read. Sau một thời gian, thông tin từ ô nhớ có địa chỉ trên sẽ được đặt vào thanh ghi dữ liệu bộ nhớ MDR (*Memory Data Register*). Còn nếu muốn lưu một giá trị vào một ô nhớ, đầu tiên chúng ta phải ghi địa chỉ của ô nhớ đó vào thanh ghi MAR, và giá trị cần lưu vào thanh ghi MDR. Sau đó, chúng

ta yêu cầu ghi thông tin bằng tín hiệu Write Enable tích cực. Khi đó, thông tin đang ở trong thanh ghi MDR sẽ được ghi vào ô nhớ có địa chỉ trong thanh ghi MAR.

Hình 4.2 cho ta một bộ nhớ có mười sáu ô nhớ có địa chỉ từ 0 tới 15 (tức từ 0000_2 tới 1111_2). Mỗi vị trí chứa 8 bit thông tin. Trong hình, ô nhớ 3 (0011_2) đang chứa trị 00101101.

0000	
0001	
0010	
0011	00101101
0100	
0101	
0110	
	⋮
1101	
1110	10100010
1111	

Hình 4.2 Bộ nhớ 16 phần tử, mỗi phần tử 8 bit

4.1.2 Đơn vị xử lý (Processing Unit)

Đơn vị xử lý là bộ phận thực sự trong máy tính xử lý thông tin. Trong một máy tính hiện đại ngày nay có nhiều thành phần tinh vi phức tạp có chức năng riêng, như chia, căn bậc hai, ... Theo mô hình von Neumann, bộ phận xử lý chính là đơn vị số học luận lý ALU (*Arithmetic Logic Unit*) vì nó có thể thực hiện các phép tính số học như cộng, trừ, và các thao tác logic cơ bản như AND, OR, và NOT. Các thao tác mà ALU của LC-3 có thể thực hiện là ADD, AND, và NOT.

Kích thước của các toán hạng được ALU xử lý thường được xem như là chiều dài từ máy của máy tính. Mỗi toán hạng được xem là một từ. Trong LC-3, ALU xử lý toán hạng 16 bit. Chúng ta nói LC-3 có chiều dài từ 16 bit. Mỗi kiến trúc tập lệnh có chiều dài riêng của nó. Hầu hết các bộ vi xử lý trong các máy PC hay máy trạm đều có chiều dài từ máy hoặc 32 bit như Intel Pentium 4 hoặc 64 bit như SUN SPARC-V9 và Intel Itanium. Còn trong một số ứng dụng khác như máy nhắn tin, điện thoại di động, ... thì các bộ vi xử lý 8 bit cũng đã đủ cho sự hoạt động của chúng rồi.

Ngoài ra, để thực hiện tốt thao tác trong thời gian ngắn nhất, trong đơn vị xử lý còn có một bộ nhớ tạm, đó là tập các thanh ghi, mỗi thanh ghi có cấu trúc như trong mục 3.4.3. Kích thước của thanh ghi luôn bằng với kích thước của toán hạng đầu vào của ALU, có nghĩa là mỗi thanh ghi chứa một từ máy. LC-3 có tám thanh ghi (R0, R1, ..., R7), mỗi thanh ghi dài 16 bit. Cấp ISA của SPARC-V9 có 32 thanh ghi (R0, R1, ..., R31), mỗi thanh ghi dài 64 bit.

4.1.3 Xuất và nhập

Để một máy tính xử lý thông tin, thông tin phải được đưa vào trong máy tính. Để sử dụng được kết quả đã được xử lý, các kết quả này phải được thể hiện bằng một cách nào đó ra bên ngoài máy tính. Các thiết bị làm các việc như vậy gọi là các thiết bị xuất nhập, còn được gọi là các thiết bị ngoại vi.

Trong LC-3, chúng ta có hai thiết bị xuất nhập, đó là bàn phím và màn hình. Dĩ nhiên, còn có nhiều thiết bị xuất nhập khác nữa trong hệ thống máy tính ngày nay. Ví dụ, để nhập trị ta có mouse, scanner, đĩa cứng, đĩa quang. Để xuất liệu, ta có máy in, màn hiển thị LED, đĩa cứng.

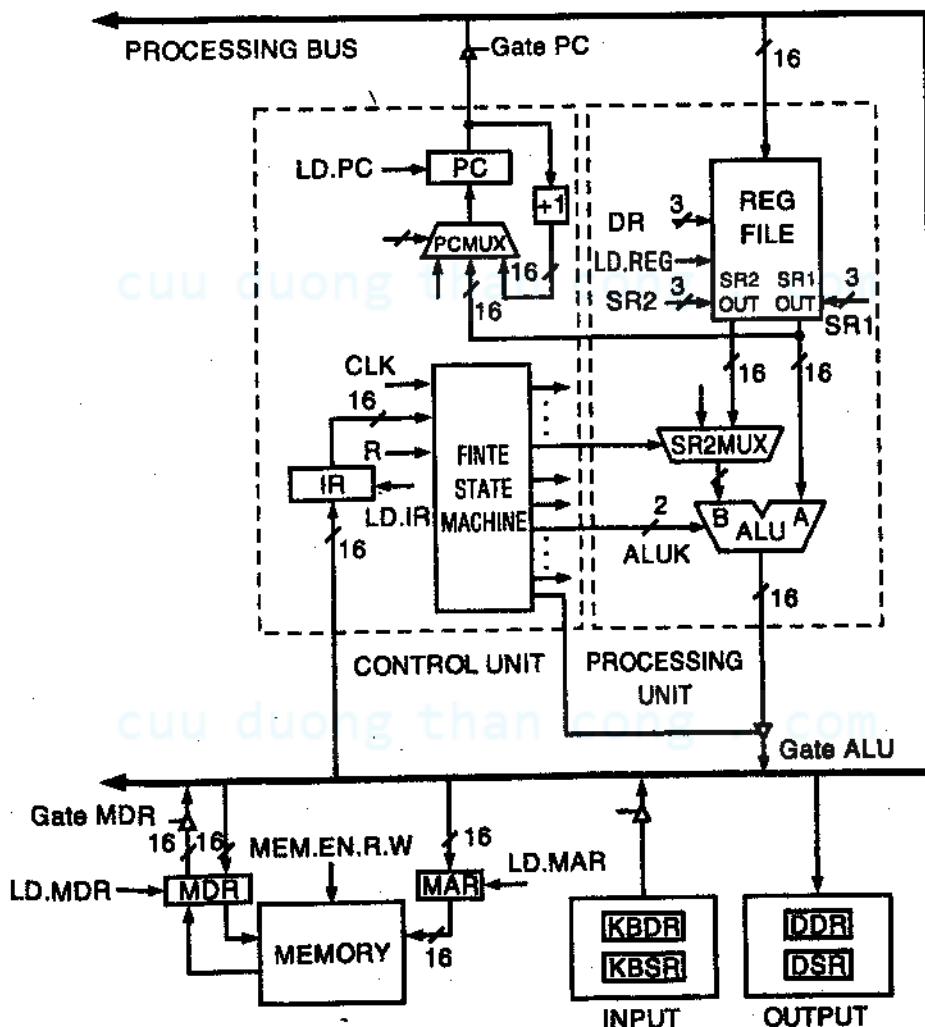
4.1.4 Đơn vị điều khiển

Đơn vị điều khiển cũng như nhạc trưởng của một dàn nhạc, nó có nhiệm vụ điều phối tất cả các bộ phận khác làm việc với nhau. Như chúng ta sẽ thấy trong quá trình thực hiện một chương trình máy tính, đơn vị điều khiển sẽ theo dõi cả hai quá trình, quá trình thực hiện chương trình và quá trình thực hiện từng lệnh.

Để theo dõi lệnh nào đang được thực thi, đơn vị điều khiển có thanh ghi lệnh IR (*instruction register*) để chứa lệnh đó. Để theo dõi lệnh cần được thực thi kế tiếp, đơn vị điều khiển có một thanh ghi chứa địa chỉ của lệnh kế đó. Vì nhiều lý do lịch sử, thanh ghi đó được gọi là thanh ghi đếm chương trình PC (*program counter*), dù tên thanh ghi con trả lệnh IP (*instruction pointer*) là thích hợp hơn PC, vì nội dung của thanh ghi này “chỉ” tới lệnh kế tiếp cần được thực thi. Trong thực tế, hãng Intel đã sử dụng tên gọi này để nói về thanh ghi PC. Tuy nhiên, theo cách chung chúng ta gọi nó là PC.

4.2 MỘT VÍ DỤ VỀ MÔ HÌNH VON NEUMANN: LC-3

Trong chương 3, chúng ta đã học các thành phần cơ bản của một CPU, và một máy tính có cấu trúc đơn giản là LC-3 đã được đề cập. Trong phần này, chúng ta sẽ mô tả LC-3 như là một máy tính theo mô hình von Neumann với tất cả các bộ phận trong đường truyền dữ liệu của nó như trong hình 4.3. Lưu ý là trong hình 4.3 chúng ta đã xóa các bộ phận không trọng yếu để nêu bật năm bộ phận cơ bản (*Memory*, *Input/Output*, *Processing unit*, và *Control unit*) trong mô hình von Neumann.



Hình 4.3 LC-3 là một ví dụ cho mô hình von Neumann

Có hai loại đầu mũi tên trong hình 4.3: tô đặc và không tô đặc. Các đầu mũi tên tô đặc ký hiệu cho các phần tử dữ liệu chạy theo đường truyền tương ứng. Các đầu mũi tên không tô đặc ký hiệu cho các tín hiệu điều khiển dùng để điều khiển các phần tử khác hoạt động. Ví dụ với bộ phận xử lý ALU, hai toán hạng 16 bit đầu vào và giá trị kết quả 16 bit đều dùng các đầu mũi tên đặc. Trong khi đó, thao tác thực hiện trên hai toán hạng này (ký hiệu là ALUK) dùng đầu mũi tên rỗng.

Các bộ phận trong mô hình von Neumann của LC-3 là:

1. Bộ nhớ (*Memory*) gồm các phần tử lưu trữ, với thanh ghi MAR chỉ tới ô nhớ riêng biệt, và thanh ghi MDR giữ nội dung của ô nhớ trong quá trình ghi/đọc bộ nhớ. Thanh ghi MAR dài 16 bit phản ánh không gian địa chỉ bộ nhớ của LC-3 là 2^{16} ô nhớ. Thanh ghi MDR dài 16 bit, cho biết thông tin trong mỗi ô nhớ là 16 bit.

2. Xuất/ Nhập (*Input/Output*)

Gồm bàn phím và màn hình. Để thao tác với bàn phím, ta có hai thanh ghi, thanh ghi dữ liệu KBDR (*Keyboard Data Register*) giữ mã ASCII của các phím đã được nhấn, và thanh ghi trạng thái KBSR (*Keyboard Status Register*) lưu thông tin về trạng thái của phím được ấn. Màn hình cũng cần hai thanh ghi để làm việc, thanh ghi DDR (*Display Data Register*) giữ mã ASCII của ký tự cần hiển thị, và thanh ghi DSR (*Display Status Register*) giữ thông tin về trạng thái hoạt động của màn hình.

3. Đơn vị xử lý (*Processing unit*)

Gồm đơn vị số học luận lý ALU và tám thanh ghi (R0, ..., R7) để lưu các giá trị tạm thời cần cho quá trình tham khảo, tính toán trong tương lai. ALU của LC-3 có thể thực hiện một phép tính số học (cộng) và hai thao tác luận lý (AND và bù 1).

4. Đơn vị điều khiển (*Control unit*) gồm tất cả các phần tử cần thiết để quản lý quá trình đang được máy tính xử lý. Cấu trúc quan trọng nhất là máy trạng thái hữu hạn (*Finite state machine*), điều khiển tất cả các hoạt động. Nó hoạt động theo từng bước, từ chu kỳ xung clock này qua chu kỳ xung clock khác. Do đó, trong hình 4.3 ta thấy đầu vào CLK, ký hiệu cho xung clock. Thanh ghi IR (*instruction*

register) cũng là một đầu vào của máy trạng thái hữu hạn, để xác định các thao tác cần thực hiện trong quá trình thực thi lệnh LC-3 đang có trong thanh ghi IR. Thanh ghi PC (*program counter*) cũng là một phần của đơn vị điều khiển, nó theo dõi lệnh kế cần được thực thi sau khi lệnh hiện thời hoàn thành.

Chú ý, tất cả đều ra từ máy trạng thái hữu hạn trong hình 4.3 đều là các tín hiệu điều khiển, nên chúng đều có đầu mũi tên rỗng. Ví dụ, một trong các đầu ra là 2 bit ALUK, dùng để quy định thao tác mà ALU cần thực hiện (add, and, và not) trong chu kỳ xung clock hiện hành. Đầu ra khác là GateALU, để quyết định việc xuất dữ liệu ra processor bus.

4.3 QUÁ TRÌNH XỬ LÝ LỆNH

Một vấn đề quan trọng trong mô hình von Neumann là quá trình xử lý chương trình và dữ liệu được lưu giữ dưới dạng chuỗi các bit trong bộ nhớ máy tính. Quá trình gồm nhiều bước, mỗi bước có chức năng riêng mà ta cần tìm hiểu.

4.3.1 Lệnh

Đơn vị cơ bản nhất của quá trình xử lý của máy tính là lệnh. Lệnh gồm hai phần, mã lệnh (*opcode*) và toán hạng (*operand*). Với LC-3, mỗi lệnh gồm 16 bit, được đánh số từ trái qua phải từ bit[15] tới bit[0]. Bit[15:12] chứa opcode. Điều này có nghĩa là có tổng cộng 2^4 mã lệnh khác nhau. Các bit từ bit[11:0] được dùng để xác định toán hạng.

Hai ví dụ dưới đây cho chúng ta cái nhìn rõ hơn về lệnh của ISA LC-3.

Ví dụ 4.1 Lệnh ADD (cộng) có ba toán hạng gồm hai toán hạng nguồn (dữ liệu từ đó được cộng) và một toán hạng đích (giữ tổng sau khi phép cộng được thực thi). Vì ISA LC-3 có tám thanh ghi nên có ba bit cần dùng để mã cho một thanh ghi. Lệnh cộng ADD có dạng như sau:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD		Dst		Src1		0	0	0		Src2					
0	0	0	1	1	1	0	0	1	0	0	0	0	1	1	0

Bốn bit, bit[15:12], là mã lệnh của lệnh cộng này: 0001. Các bit bit[11:9] quy định vị trí lưu kết quả, trong ví dụ trên là thanh ghi R6 (110). Các bit bit[8:6] và bit[2:0] xác định thanh ghi chứa hai toán hạng nguồn, trong trường hợp này là R2 (010) và R6 (110). Các bit bit[5:3] sẽ được sử dụng cho mục đích khác, sẽ được đề cập sau. Như vậy, lệnh trên thực tế có nghĩa là một lệnh gán trị: $R6 = R2 + R6$.

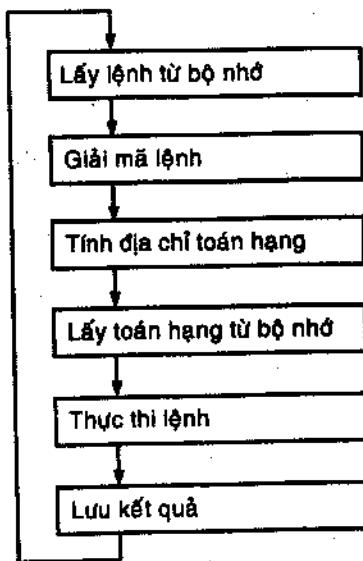
Ví dụ 4.2 Lệnh LDR (LD xuất phát từ Load) sẽ vào ô nhớ được xác định, đọc dữ liệu và lưu nó vào thanh ghi. Lệnh này có hai toán hạng là giá trị đọc được từ ô nhớ và thanh ghi đích. Ký tự R trong LDR viết tắt từ register, cho biết cơ chế kiểu địa chỉ (*addressing mode*) xác định toán hạng là ô nhớ lưu dữ liệu, cụ thể là Base + offset. Lệnh này có dạng như sau:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LDR				Dst			Base			Offset					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	1	0	0	1	1	0	0	0	1	1	0

Bốn bit mã lệnh của LDR là 0110. Các bit bit[11:9] quy định thanh ghi đích lưu dữ liệu đọc được từ ô nhớ. Các bit bit[8:0] dùng để tính địa chỉ của ô nhớ lưu toán hạng nguồn. Trong trường hợp này, với kiểu địa chỉ base + offset, các bit bit[8:6] chứa 011, tức thanh ghi R3 chứa địa chỉ Base, còn bit[5:0] chứa địa chỉ offset theo dạng bù 2, tức $000110_2 = 6$. Như vậy lệnh này có nghĩa là: $R2 = M[R3 + 6]$, với $M[x]$ có nghĩa là ô nhớ có địa chỉ x.

4.3.2 Chu kỳ lệnh

Dưới sự điều phối của đơn vị điều khiển, các lệnh được xử lý một cách hệ thống theo từng bước. Chuỗi các bước thực hiện lệnh được gọi là chu kỳ lệnh (*instruction cycle*). Mỗi bước được xem như một pha (*phase*). Một cách cơ bản, có sáu pha trong chu kỳ lệnh, dù trong thực tế nhiều máy tính được thiết kế với chu kỳ lệnh không đòi hỏi cả sáu pha này. Sáu pha của chu kỳ lệnh như sau:

**Hình 4.4** Sáu pha của chu kỳ lệnh

Cụ thể như sau:

1. Lấy lệnh từ bộ nhớ (Fetch)

Pha này lấy lệnh kế tiếp từ bộ nhớ mà địa chỉ đang được chứa trong thanh ghi PC và nạp vào thanh ghi lệnh IR của đơn vị điều khiển. Mỗi lệnh của chương trình máy tính là chuỗi các bit, và toàn bộ chương trình đang được thực thi theo sơ đồ máy tính von Neumann được chứa trong bộ nhớ máy tính. Như vậy pha FETCH này gồm có nhiều bước con như sau:

- Đầu tiên thanh ghi MAR được nạp trị là nội dung thanh ghi PC
- Kế tiếp, bộ nhớ được yêu cầu, để rồi lệnh kế tiếp được lấy ra từ bộ nhớ và đặt vào thanh ghi MDR
- Sau cùng, thanh ghi IR được nạp trị là nội dung của thanh ghi MDR

Và sau đó, chúng ta qua bước kế tiếp trong chu kỳ lệnh. Tuy nhiên, khi chu kỳ lệnh kết thúc, chúng ta muốn lấy lệnh kế, tức thanh ghi PC phải đang lưu địa chỉ của lệnh kế. Vì vậy một bước con cần thêm vào để thực hiện tăng PC. Như vậy, sau khi thực hiện xong lệnh, pha FETCH của lệnh kế tiếp sẽ nạp vào thanh ghi IR nội dung của ô nhớ kế tiếp, và sự thực thi lệnh hiện hành không làm thay đổi giá trị trong thanh ghi PC.

Tóm lại, bước FETCH sẽ gồm các thao tác sau:

Bước 1: Nạp nội dung của thanh ghi PC vào thanh ghi MAR, đồng thời tăng PC

Bước 2: Yêu cầu bộ nhớ để lấy lệnh đặt vào thanh ghi MDR

Bước 3: Nạp vào thanh ghi IR nội dung của thanh ghi MDR

Mỗi bước thao tác như vậy chiếm một hay nhiều chu kỳ máy (hay chu kỳ xung clock của CPU), như bước 1 và 3 mỗi bước chiếm một chu kỳ máy, trong khi bước 2 có thể chiếm hơn một chu kỳ máy tùy theo thời gian truy xuất bộ nhớ.

2. Giải mã lệnh (*Decode*)

Pha này khảo sát lệnh để đưa ra các yêu cầu cho vi kiến trúc thực hiện. Lưu ý, trong LC-3, bộ giải mã 4 ra 16 xác định mã lệnh nào trong 16 mã lệnh cần được xử lý. Đầu vào là bốn bit mã lệnh IR[15:12]. Đường đầu ra tích cực tương ứng với mã lệnh ở đầu vào. Tùy thuộc vào đầu ra nào của bộ giải mã được tích cực, 12 bit còn lại trong lệnh sẽ xác định cái cần phải làm để thực hiện lệnh đó.

3. Tính địa chỉ toán hạng (*Evaluate address*)

Pha này sẽ tính địa chỉ của ô nhớ cần thiết để xử lý lệnh. Như lệnh LDR trong ví dụ 4.2 sẽ lấy một trị từ một ô nhớ và nạp vào thanh ghi. Trong ví dụ này, địa chỉ ô nhớ sẽ được tính từ trị 6 cộng với nội dung thanh ghi R3. Thao tác tính toán này được thực hiện trong pha này.

4. Lấy toán hạng (*Fetch Operands*)

Pha này thực hiện việc lấy các toán hạng cần thiết để xử lý lệnh. Trong ví dụ 4.2 về lệnh LDR, pha này gồm hai bước: nạp thanh ghi MAR giá trị địa chỉ tính toán được từ pha Tính địa chỉ toán hạng, và đọc bộ nhớ để có toán hạng nguồn được đặt trong thanh ghi MDR.

Trong ví dụ 4.1 về lệnh ADD, pha này cần hai toán hạng từ hai thanh ghi R2 và R6. Với nhiều bộ vi xử lý hiện đại, việc lấy toán hạng từ thanh ghi có thể được thực hiện cùng lúc khi lệnh được giải mã để tăng tốc quá trình xử lý lệnh.

5. Thực thi lệnh (*Execute*)

Pha này thực thi lệnh sau khi đã có đủ tất cả toán hạng và mã lệnh. Như trong ví dụ về lệnh ADD, pha này chỉ gồm bước đơn giản là thực thi việc cộng trong ALU.

6. Lưu kết quả (*Store result*)

Đây là bước sau cùng của quá trình thực thi lệnh. Kết quả của lệnh sẽ được ghi vào thanh ghi đích đã được xác định.

Một khi pha thứ sáu này được hoàn thành, đơn vị điều khiển bắt đầu một chu kỳ lệnh mới, từ đầu pha Lấy lệnh.

4.4 THAY ĐỔI QUÁ TRÌNH XỬ LÝ LỆNH

Bình thường một chương trình máy tính được thực hiện theo trình tự, nghĩa là lệnh đầu tiên được thực thi, sau đó tới lệnh thứ hai, rồi thứ ba, ... Tuy nhiên, có một nhóm lệnh đặc biệt gọi là lệnh điều khiển, nó có thể thay đổi trình tự thực thi lệnh. Ví dụ, chúng ta muốn lệnh thứ nhất được thực hiện, sau đó tới lệnh thứ hai, rồi thứ ba, và rồi quay lên thực hiện lại lệnh thứ nhất, rồi thứ hai, thứ ba, rồi lại quay lên thực hiện lại lệnh thứ nhất, thứ hai rồi thứ ba, cứ như vậy nhiều lần. Như chúng ta biết, mỗi chu kỳ lệnh bắt đầu bằng việc nạp thanh ghi PC vào thanh ghi MAR. Như vậy, nếu chúng ta muốn thay đổi trình tự thực thi lệnh, chúng ta phải thay đổi thanh ghi PC trong khoảng thời gian giữa lúc nó được tăng lên (trong pha Lấy lệnh của một lệnh) và sự bắt đầu của pha Lấy lệnh của lệnh kế.

Các lệnh điều khiển thực hiện chức năng đó bằng việc nạp thanh ghi PC trong pha Thực thi lệnh, việc này sẽ xóa trị đã được tăng trong thanh ghi PC trong pha Lấy lệnh trước đó. Kết quả là ở đầu chu kỳ lệnh kế, khi máy tính truy xuất thanh ghi PC để lấy địa chỉ của lệnh, nó sẽ lấy địa chỉ của lệnh đã được nạp từ pha Thực thi lệnh trước đó, chứ không lấy lệnh kế theo trình tự trong chương trình máy tính.

Ví dụ 4.3 Lệnh JMP của ISA LC-3 có định dạng như sau. Giả sử lệnh này đang được chứa trong bộ nhớ ở địa chỉ x36A2.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP	0	0	0	Base	0	0	0	Q	0	0					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0

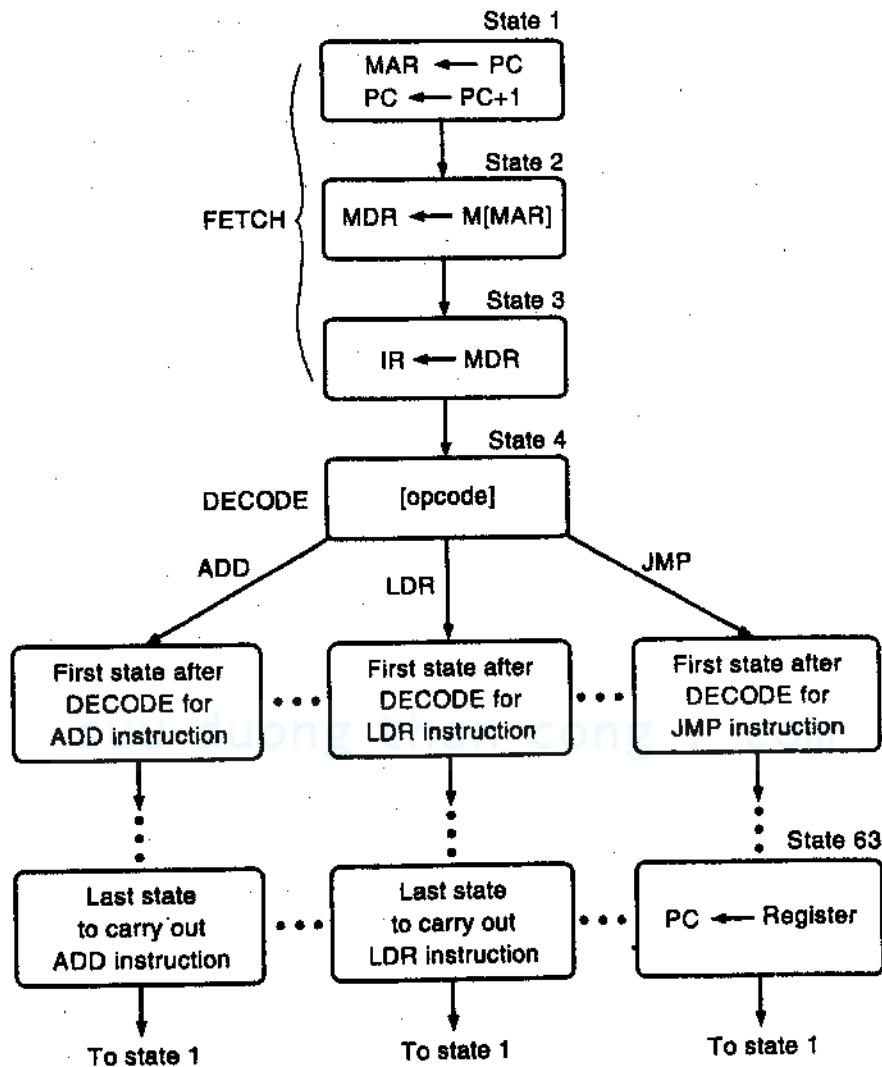
Bốn bit mã lệnh của JMP là 1100. Các bit bit[8:6] xác định thanh ghi chứa địa chỉ của lệnh kế tiếp cần xử lý, trong ví dụ này là thanh ghi R3. Như vậy, có thể hiểu lệnh này sẽ nạp thanh ghi PC (trong pha Thực thi lệnh) giá trị là nội dung của thanh ghi R3 để lệnh kế tiếp cần thực thi là lệnh có địa chỉ trong thanh ghi R3. Cụ thể, lúc bắt đầu chu kỳ lệnh, thanh ghi PC = x36A2. Pha Lấy lệnh nạp lệnh JMP vào thanh ghi IR và thanh ghi PC được cập nhật để chứa địa chỉ x36A3. Giả sử thanh ghi R3 đang chứa x5446, trong pha Thực thi lệnh, thanh ghi PC sẽ được nạp trị x5446 này. Do đó trong chu kỳ lệnh kế tiếp, lệnh được xử lý là lệnh ở địa chỉ x5446 chứ không phải lệnh ở địa chỉ x36A3.

Hình 4.5 trình bày một phần sơ đồ trạng thái tương ứng với máy trạng thái hữu hạn, được dùng để điều khiển tất cả các pha trong chu kỳ lệnh. Mỗi trạng thái tương ứng với một chu kỳ xung clock. Việc xử lý được điều khiển bởi mỗi trạng thái được mô tả trong mỗi nút biểu diễn trạng thái đó. Các cung chỉ ra các chuyển trạng thái.

Việc xử lý lệnh bắt đầu ở state 1. Pha lấy lệnh FETCH cần ba chu kỳ xung clock. Trong chu kỳ xung clock đầu tiên, thanh ghi MAR được nạp nội dung của PC, và PC được tăng lên. Để nội dung của thanh ghi PC được nạp vào MAR (hình 4.3), máy trạng thái hữu hạn cần phải có hai tín hiệu GatePC và LD.MAR. GatePC cho phép nối PC ra bus của bộ xử lý. LD.MAR là tín hiệu cho phép ghi MAR, cài thông tin trên bus trong quá trình chép vào MAR ở cuối chu kỳ xung clock hiện hành.

Để thanh ghi PC được tăng lên, máy trạng thái hữu hạn phải yêu cầu PCMUX chọn đầu vào để đầu ra được chọn là hộp có nhãn +1, đồng thời cũng yêu cầu tín hiệu LD.PC cài đầu ra của PCMUX ở cuối chu kỳ xung clock hiện hành.

Sau đó, máy trạng thái hữu hạn chuyển qua state 2, lúc này thanh ghi MDR được nạp mã lệnh đọc được từ bộ nhớ.

**Hình 4.5** Sơ đồ trạng thái thu gọn của LC-3

Trong state 3, dữ liệu được chuyển từ thanh ghi MDR sang thanh ghi lệnh IR. Việc này cần hai tín hiệu GateMDR và LD.IR từ máy trạng thái hữu hạn, và thanh ghi IR được cài dữ liệu vào cuối chu kỳ xung clock, kết thúc pha FETCH của chu kỳ lệnh.

Pha giải mã lệnh DECODE chiếm một chu kỳ xung clock. Trong state 4, tùy thuộc vào mã thao tác opcode trong các bit IR[15:12], máy trạng thái hữu hạn sẽ quyết định trạng thái kế để xử lý lệnh. Việc xử lý này cứ tiếp tục từ chu kỳ này sang chu kỳ khác cho tới khi lệnh được thi xong, sau đó máy trạng thái hữu hạn quay lại trạng thái 1 cho việc thực thi lệnh kế tiếp.

Như đã được đề cập, đôi khi chúng ta không thực thi lệnh kế tiếp theo trình tự, mà lại nhảy tới một vị trí khác để tìm lệnh kế cần thực thi. Việc này có thể được thực hiện dễ dàng bằng việc nạp thanh ghi PC trong pha thực thi lệnh EXECUTE của lệnh điều khiển (như JMP), như trong trạng thái 63 của hình 4.5.

4.5 KHÁI NIỆM ISA LC-3

Kiến trúc tập lệnh (ISA - *Instruction Set Architecture*) xác định tất cả thông tin về máy tính mà phần mềm cần phải nhận biết. Nói một cách khác, kiến trúc tập lệnh (ISA) xác định mọi thứ trong máy tính mà người lập trình sử dụng khi viết chương trình bằng ngôn ngữ máy. Như vậy, kiến trúc tập lệnh cũng xác định mọi thứ trong máy tính để một người có thể sử dụng chúng để dịch các chương trình được viết bằng ngôn ngữ cấp cao như C, Pascal, Fortran hay Cobol sang ngôn ngữ máy của máy tính đó. Có nghĩa, ISA xác định tổ chức bộ nhớ, tập thanh ghi, và tập lệnh gồm mã lệnh, loại dữ liệu, và các kiểu định vị địa chỉ.

4.5.1 Tổ chức bộ nhớ

Bộ nhớ của LC-3 có dung lượng 2^{16} (65536) ô nhớ, mỗi ô có chiều dài 16 bit. Tuy nhiên, không phải tất cả 65536 ô nhớ đều được sử dụng, vì có một số vùng nhớ được dùng để lưu các thông tin hệ thống như bảng các vector ngắn, biến hệ thống, ... Vì đơn vị lưu trữ chuẩn được LC-3 xử lý là 16 bit, nên chúng ta gọi 16 bit là một từ (*word*), và do đó chúng ta nói LC-3 định vị theo từ.

4.5.2 Thanh ghi

Như hầu hết các máy tính, LC-3 thường tồn hơn một chu kỳ xung clock để lấy dữ liệu từ bộ nhớ, nên nó cũng cung cấp các thanh ghi để chứa dữ liệu tạm thời mà có thể được truy xuất chỉ trong một chu kỳ xung clock.

Loại thanh ghi mà hầu hết các máy tính đều có là tập thanh ghi đa dụng. Mỗi thanh ghi trong tập này đều có thể được sử dụng cho nhiều mục đích khác nhau. Các thanh ghi có tính chất giống như ô nhớ, tức nó được sử dụng để chứa thông tin mà có thể được truy tìm sau đó. Trong LC-3, mỗi thanh ghi dài một từ, tức 16 bit, và có 8 thanh ghi đa dụng. Mỗi thanh ghi có một chỉ định riêng, nên cần dùng 3 bit để mã cho số hiệu của một thanh ghi, đó là các thanh ghi R0, R1, ..., R7.

4.5.3 Tập lệnh

Một lệnh được tạo từ hai thứ, mã thao tác (*opcode*) là cái mà lệnh bắt máy tính thực thi, và toán hạng (*operands*) là cái mà máy tính cần để thực thi lệnh. Tập lệnh của một ISA được định nghĩa bằng tập các mã thao tác, kiểu dữ liệu và các kiểu định vị để xác định chỗ của toán hạng. Trong ví dụ 4.1 trên, ta có lệnh $R6 = R2 + R6$, thì kiểu định vị của toán hạng là thanh ghi vì các toán hạng đều là thanh ghi.

4.5.4 Mã thao tác

Một vài kiến trúc tập lệnh có tập các mã thao tác rất lớn, một số khác lại có tập mã thao tác rất nhỏ. Kiến trúc tập lệnh của LC-3 có 15 lệnh, mỗi lệnh được chỉ định mã thao tác riêng. Mã thao tác được quy định trong bốn bit [15:12] của lệnh, nên sẽ có 16 mã thao tác khác nhau. Tuy nhiên, trong thực tế ISA LC-3 chỉ sử dụng 15 mã thao tác. Mã 1101 chưa được quy định mã thao tác, nó được để dành cho các nhu cầu cần thiết trong tương lai.

Có ba loại lệnh khác nhau, tức có ba loại mã thao tác: thi hành, chuyển dữ liệu, và điều khiển. Các lệnh thi hành xử lý thông tin, như lệnh ADD. Các lệnh chuyển dữ liệu chuyển thông tin qua lại giữa bộ nhớ và các thanh ghi, giữa các thanh ghi với nhau, giữa các thiết bị xuất nhập, như lệnh LDR. Các lệnh điều khiển thay đổi trình tự các lệnh sẽ được thực thi, như lệnh JMP.

Hình 4.3 trình bày toàn bộ tập lệnh LC-3 với định danh từng nhóm bit theo quy ước. Tất cả các lệnh này sẽ được trình bày chi tiết trong các mục từ 4.6 dưới đây.

cuu duong than cong . com

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD ⁺		0001		DR		SR1		0	00			SR2				
ADD ⁺		0001		DR		SR1		1				imm5				
AND ⁺		0101		DR		SR1		0	00			SR2				
AND ⁺		0101		DR		SR1		1				imm5				
BR		0000	n	z	p		SR1					PC offset9				
JMP		1100		000		BaseR						000000				
JSR		0100	1									PC offset11				
JSRR		0100	0	00		BaseR						000000				
LD ⁺		0010		DR								PC offset9				
LDI ⁺		1010		DR								PC offset9				
LDR ⁺		0110		DR		BaseR						offset6				
LEA ⁺		1110		DR								PC offset9				
NOT ⁺		1001		DR		SR						111111				
RET		1100		000		111						000000				
RTI		1100		000								0000000000000000				
ST		0011		SR								PC offset9				
STI		1011		SR								PC offset9				
STR		0111		SR		BaseR						offset6				
TRAP		1111		0000								trapvect8				
Reserved		1101														

Hình 4.6 Toàn bộ tập lệnh LC-3

4.5.5 Các kiểu dữ liệu

Một kiểu dữ liệu là một sự miêu tả về thông tin để ISA có các mã thao tác thực thi với miêu tả đó. Có rất nhiều cách để biểu diễn một thông tin trên máy tính, ví dụ với một số nguyên ta có thể viết số ở hệ thập phân, hệ bát phân hay hệ thập lục phân. Hoặc với số thực, ta có thể dùng số thực dấu chấm cố định hay số thực dấu chấm động.

Trong kiến trúc tập lệnh LC-3, kiểu dữ liệu duy nhất được sử dụng là dạng số nguyên bù 2.

4.5.6 Các kiểu định vị địa chỉ

Một kiểu định vị địa chỉ là một cơ chế để xác định toán hạng ở đâu. Tổng quát, một toán hạng có thể được tìm ở một trong ba chỗ: trong bộ nhớ, trong một thanh ghi, hoặc là một phần của lệnh. Nếu nó là một phần của lệnh, ta gọi nó là toán hạng tức thời.

LC-3 sử dụng năm kiểu định vị địa chỉ: tức thời, thanh ghi, và ba kiểu định vị địa chỉ bộ nhớ là PC-relative, gián tiếp và Base+offset. Trong mục 4.7, chúng ta sẽ thấy các lệnh chuyển dữ liệu sẽ dùng cả năm kiểu định vị.

4.5.7 Các mã điều kiện

Hầu như tất cả các ISA đều cho phép quá trình thực hiện lệnh được thay đổi tùy thuộc vào kết quả được tạo ra trước đó. LC-3 có ba thanh ghi một bit được đặt (đặt lên 1) hay xóa (xóa về 0) mỗi khi một trong tám thanh ghi đa dụng được ghi trị. Ba thanh ghi một bit này được gọi là các thanh ghi trạng thái N, Z, và P tương ứng với nghĩa của chúng: âm, zero, và dương. Khi một thanh ghi đa dụng được ghi trị, các thanh ghi trạng thái tương ứng sẽ được đặt về 0 hay 1 tương ứng với kết quả được ghi vào thanh ghi đó là âm, zero, hay dương. Có nghĩa là nếu kết quả là âm, thì thanh ghi N sẽ được đặt là 1, các thanh ghi kia bị xóa (là 0). Tương tự cho các thanh ghi Z và P.

Mỗi thanh ghi trạng thái được coi như là một mã điều kiện vì nó có thể được các lệnh điều khiển sử dụng để kiểm tra và thay đổi quá trình thực thi lệnh.

4.6 NHÓM LỆNH THI HÀNH

Các lệnh thi hành xử lý dữ liệu, gồm các phép số học (ví dụ như ADD, SUB, MUL, và DIV) và các phép toán luận lý (ví dụ như AND, OR, NOT, XOR). LC-3 có ba lệnh thi hành gồm ADD, AND, và NOT.

Lệnh NOT (mã thao tác là 1001) là lệnh thi hành duy nhất có một toán hạng nguồn. Lệnh này thực hiện bù 1 (đảo bit) toán hạng nguồn 16 bit và lưu trữ kết quả đó vào toán hạng đích. Lệnh NOT sử dụng kiểu định vị địa chỉ thanh ghi cho cả hai toán hạng nguồn (bit [8:6]) và đích (bit [11:9]), các bit [5:0] phải chứa các bit 1.

Ví dụ 4.4 Nếu R5 đang chứa 0101000011110000, thì sau khi thực hiện xong lệnh sau:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	1	1	1	0	1	1	1	1	1	1	1

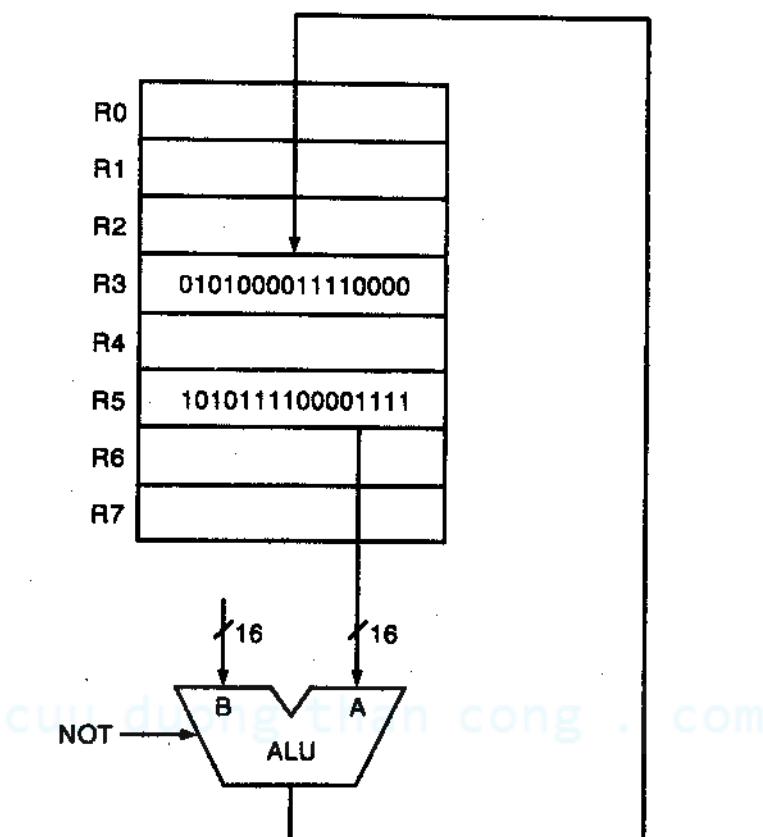
NOT R3 R5

thanh ghi R3 sẽ lưu 10101111000001111.

Hình 4.7 trình bày các phần quan trọng của đường truyền dữ liệu được sử dụng để thực thi lệnh NOT. Vì NOT là thao tác đơn toán hạng, nên chỉ có đầu vào A của ALU là thích hợp, đó là toán hạng nguồn, thanh ghi R5. Đích là thanh ghi R3, lưu kết quả từ đầu ra của ALU. Tín hiệu điều khiển tới ALU sẽ quy định thao tác là bù 1.

Lệnh ADD (opcode = 0001) và AND (opcode = 0101) đều yêu cầu các hai toán hạng nguồn 16 bit. Lệnh ADD thực hiện cộng bù hai hai toán hạng nguồn của nó. Lệnh AND thực hiện AND bit cho mỗi cặp bit trong hai toán hạng nguồn 16 bit của nó. Cũng như lệnh NOT, các lệnh ADD và AND cũng dùng kiểu định vị địa chỉ thanh ghi cho toán hạng đích và một trong các toán hạng nguồn. Các bit [11:9] xác định thanh ghi đích, còn các bit [8:6] quy định toán hạng nguồn thứ nhất là thanh ghi.

Toán hạng nguồn thứ hai của hai lệnh này có thể là thanh ghi hoặc một trị tức thời tùy vào bit [5], nếu bit này bằng 0 toán hạng nguồn thứ hai là thanh ghi, khi đó các bit [2:0] xác định thanh ghi cần dùng, còn các bit [4:3] được đặt về 0 để hoàn thành việc xác định lệnh.



Hình 4.7 Đường truyền dữ liệu tương ứng với sự thực thi lệnh NOT R3, R5

Ví dụ 4.5 Nếu R4 chứa trị 6 và R5 chứa trị -18, sau khi lệnh dưới được thực thi

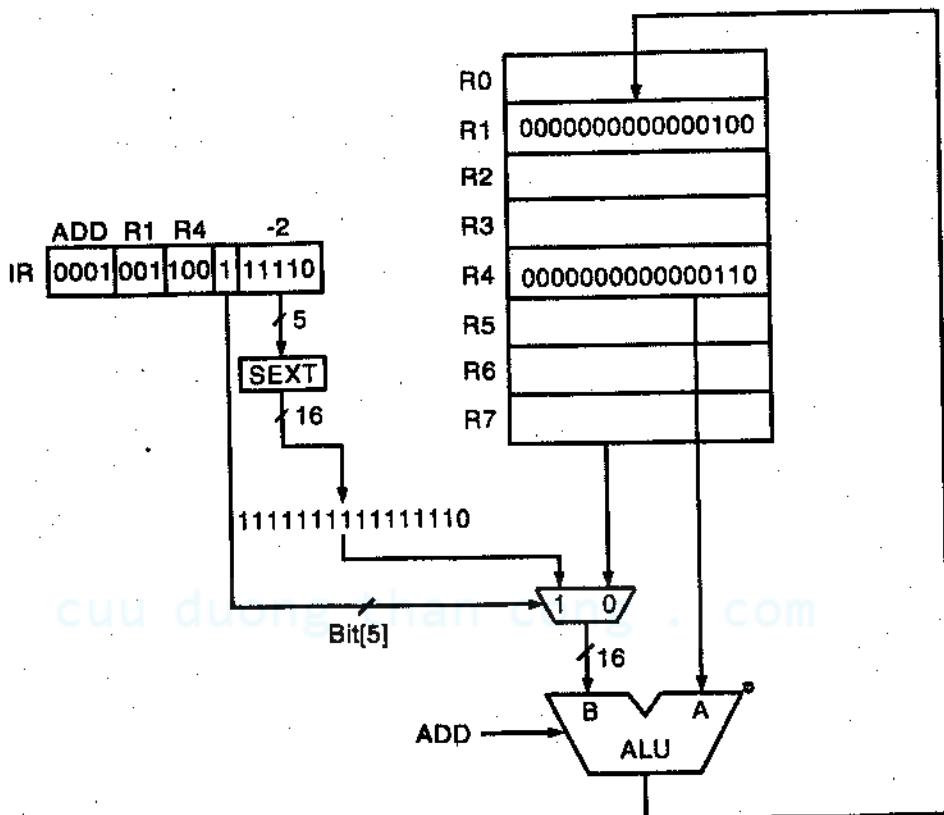
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	1	0	0	0	0	0	1	0	1

ADD R1 R4 R5

thì thanh ghi R1 chứa trị -12.

Nếu bit [5] là 1, toán hạng nguồn thứ hai được chứa ngay bên trong lệnh, và có được bằng phép mở rộng dấu 5 bit [4:0] thành 16 bit trước khi thực hiện phép ADD hoặc AND. Hình 4.8 trình bày các phần quan trọng của đường truyền dữ liệu để thực hiện lệnh ADD R1, R4, #-2.

Cũng cần lưu ý là vùng trị tức thời này chỉ có 5 bit ở dạng bù 2, nên nó chỉ có thể chứa trị trong khoảng -16..15 mà thôi.



Hình 4.8 Đường truyền dữ liệu với sự thực thi lệnh ADD R1, R4, #-2

Ví dụ 4.6 Lệnh AND R2, R2, 0 sẽ xóa thanh ghi R2 về 0.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	0

AND R2 R2 0

Ví dụ 4.7 Lệnh ADD R6, R6, 1 sẽ tăng thanh ghi R6 lên 1.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	1	1	0	1	0	0	0	0	1

ADD R6 R6 1

Ví dụ 4.8 Ba lệnh dưới đây thực hiện việc tính hiệu hai trị đang nằm trong hai thanh ghi R0 và R1.

Hai lệnh đầu (NOT R1, R1 và ADD R2, R1, 1) sẽ tính bù 2 của trị đang có trong thanh ghi R1.

Lệnh ADD R2, R0, R2 tính hiệu giữa hai trị ban đầu trong R0 và R1 theo yêu cầu.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	0	1	0	0	1	1	1	1	1	1	1
NOT				R1				R1							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	0	0	1	1	0	0	0	0	1
ADD				R2				R1							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	0	0	0	0	0	0	0	1	0
ADD				R2				R0				R2			

4.7 NHÓM LỆNH DI CHUYỂN DỮ LIỆU

Nhóm lệnh chuyển dữ liệu thực hiện việc sao chép thông tin qua lại giữa các thanh ghi đa dụng và bộ nhớ, giữa các thanh ghi và các thiết bị xuất nhập.

Quá trình chép thông tin từ bộ nhớ vào thanh ghi được gọi là nạp (*load*), và quá trình chép thông tin từ thanh ghi vào bộ nhớ gọi là lưu (*store*). Trong cả hai trường hợp, thông tin trong toán hạng nguồn không đổi, còn thông tin cũ trong toán hạng đích đã bị ghi đè bởi thông tin mới sau quá trình chép.

LC-3 có bảy lệnh thực hiện việc sao chép dữ liệu: LD, LDR, LDI, LEA, ST, STR, và STI.

Định dạng chung cho các lệnh này là

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode				DR or SR				Address generation bits							

Các lệnh chuyển dữ liệu cần hai toán hạng, một nguồn và một đích. Toán hạng nguồn là nơi dữ liệu cần được chép, còn toán hạng đích là nơi cần chép thông tin vào.

Trong định dạng trên, các bit [15:12] xác định mã thao tác opcode, các bit [11:9] quy định một trong hai toán hạng, là thanh ghi có mã theo quy ước. Các bit [8:0] chứa các bit tạo địa chỉ (*Address generation bits*), mã hóa thông tin để tính ra địa chỉ 16 bit của toán hạng thứ hai. Với LC-3, có bốn cách để tính ra địa chỉ 16 bit từ vùng tin này tùy theo kiểu định vị địa chỉ (*addressing modes*) được quy định từ opcode. Có bốn kiểu định vị địa chỉ: định vị tương đối từ PC (*PC-Relative mode*), định vị gián tiếp (*Indirect mode*), định vị base + offset (*base+offset mode*), và định vị tức thời (*immediate mode*).

4.7.1 PC-relative mode

Lệnh LD (opcode = 0010) và ST (opcode = 0011) dùng kiểu định vị tương đối PC. Các bit [8:0] của lệnh xác định một độ dời (*offset*) tính từ thanh ghi PC. Địa chỉ bộ nhớ được tính từ tổng của 16 bit địa chỉ sau phép mở rộng dấu từ các bit [8:0] và thanh ghi PC đã được tăng 1 sau pha FETCH. Với lệnh LD, ô nhớ tương ứng với địa chỉ bộ nhớ đã tính được đọc, và ghi vào thanh ghi được xác định bởi các bit [11:9] của lệnh.

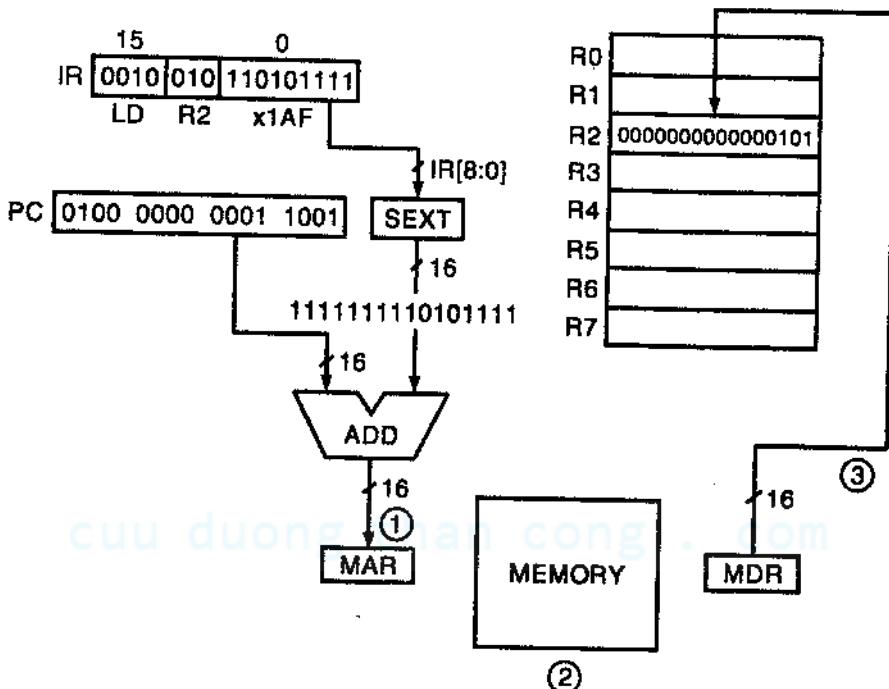
Ví dụ 4.9 Nếu lệnh nằm ở ô nhớ có địa chỉ x4018, thì nội dung ô nhớ có địa chỉ x4019 + SEXT (x1AF) = x4019 + xFFAF = x3FC8 (bỏ bit 1 do vượt quá chiều dài 16 bit) sẽ được đọc và nạp vào thanh ghi R2.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1	1	0	1	0	1	1	1	1

LD R2 x1AF

Hình 4.9 trình bày đường truyền dữ liệu với các bộ phận quan trọng cho việc thực thi lệnh này. Có ba bước thực hiện. Trong bước 1, thanh ghi PC đã được tăng 1 (x4019) được cộng với trị đã được mở rộng bit dấu từ IR[8:0] (xFFAF), và kết quả (là x3FC8) được nạp vào thanh ghi MAR. Trong bước 2, bộ nhớ được đọc và nội dung của ô nhớ x3FC8 được nạp vào thanh ghi MDR. Giá trị giá trị đó bằng 5. Trong bước 3, giá trị 5 được chép vào thanh ghi R2, hoàn thành chu kỳ lệnh.

Chú ý, vì vùng tin chứa các bit tạo địa chỉ có 9 bit, [8:0], nên ta chỉ có tầm địa chỉ nhỏ tính từ thanh ghi PC, -256 .. +255, cho cả hai lệnh LD và ST.



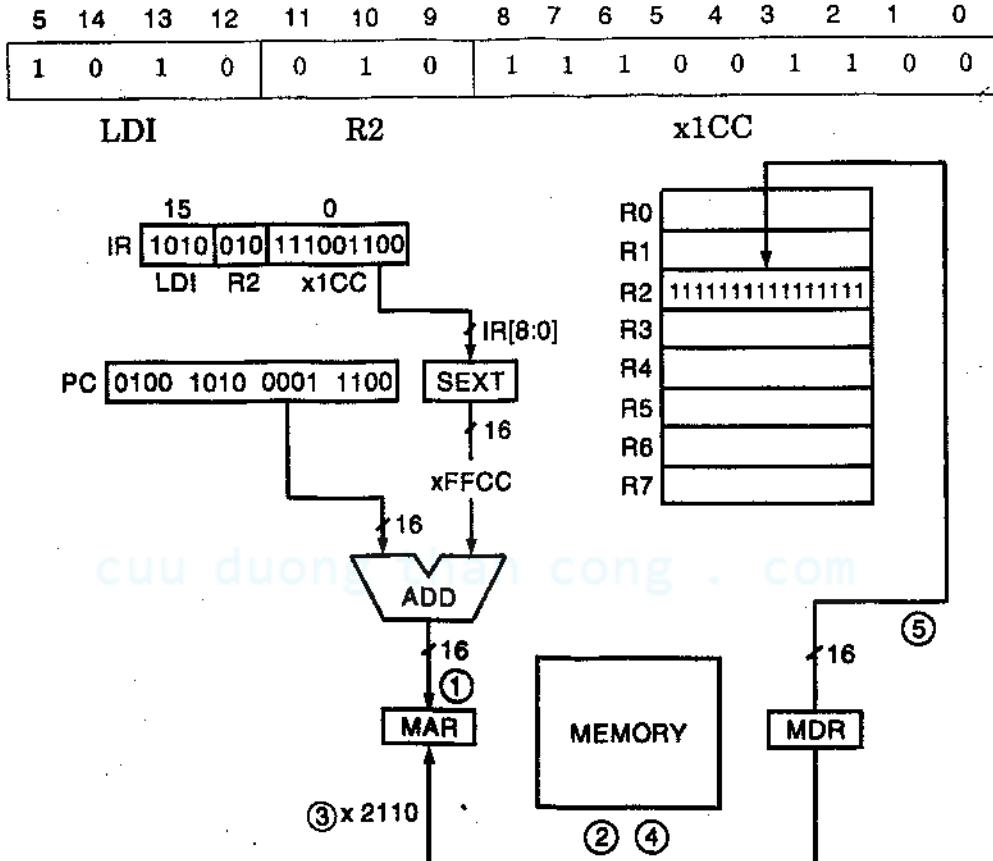
Hình 4.9 Đường truyền dữ liệu tương ứng lệnh LD R2, x1AF

4.7.2 Indirect mode

Lệnh LDI (opcode = 1010) và lệnh STI (opcode = 1011) sử dụng kiểu định vị địa chỉ gián tiếp. Với kiểu định vị này, địa chỉ đầu tiên được tạo ra cũng tương tự như với lệnh LD và ST.

Tuy nhiên, thay vì đây là địa chỉ của toán hạng chứa dữ liệu cần được nạp hay lưu, thì toán hạng này lại chứa địa chỉ của toán hạng là dữ liệu cần được nạp hay lưu. Vì phải qua trung gian như vậy nên ta mới có kiểu định vị địa chỉ gián tiếp. Với kiểu định vị này, ta thấy toán hạng chứa dữ liệu cần làm việc có thể ở bất cứ đâu trong bộ nhớ 64K, chứ không bị giới hạn trong tầm 9 bit [8:0] như trong trường hợp lệnh LD và ST. Thanh ghi đích cho LDI và thanh ghi nguồn trong STI cũng được xác định bằng các bit [11:9] trong lệnh.

Ví dụ 4.10 Nếu lệnh ở địa chỉ x4A1B, và chứa đựng của x49E8 là x2110, việc thực thi lệnh này sẽ lấy dữ liệu từ ô nhớ có địa chỉ x2110 nạp vào thanh ghi R2.



Hình 4.10 Đường truyền dữ liệu cho việc thực thi lệnh LDI R2, x1CC

Hình 4.10 chỉ ra đường truyền dữ liệu với các bộ phận quan trọng cho việc thực thi lệnh. Đầu tiên thanh ghi PC đã được tăng (x4A1C) sẽ được cộng với 9 bit sau thao tác mở rộng bit dấu trong thanh ghi lệnh IR[8:0] (xFFCC) để có kết quả (x49E8) nạp vào thanh ghi MAR. Trong bước 2, bộ nhớ được đọc và nội dung ô nhớ x49E8 (x2110) được nạp vào thanh ghi MDR. Sang bước 3, vì x2110 chưa phải là dữ liệu cần đọc, nên nó lại tiếp tục được nạp tiếp vào thanh ghi MAR. Trong bước 4, bộ nhớ tiếp tục được đọc, và thanh ghi MDR được nạp trở lại với chứa đựng là nội dung của ô nhớ x2110, giả sử là -1. Sang bước 5, nội dung của thanh ghi MDR (tức -1) được nạp vào thanh ghi R2, hoàn tất chu kỳ lệnh.

4.7.3 Base+offset mode

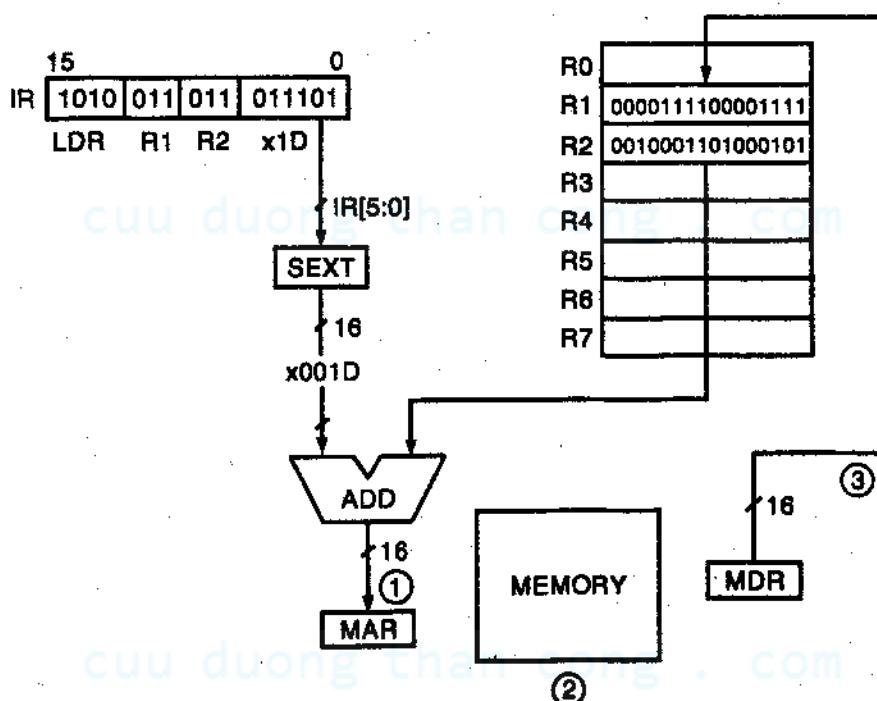
Lệnh LDR (opcode = 0110) và STR (opcode = 0111) dùng kiểu định vị Base+offset. Kiểu định vị này xác định địa chỉ toán hạng trong bộ nhớ 64K bằng cách lấy 6 bit offset được mở rộng dấu IR[5:0] cộng với thanh ghi nền trong vùng IR[8:6].

Ví dụ 4.11 Nếu R2 đang chứa trị 16 bit x2345, lệnh

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	1	0	1	0	0	1	1	1	0	1

LDR R1 R2 x1D

nạp R1 nội dung của ô nhớ x2362 (x2345 + x1D).



Hình 4.11 Đường truyền dữ liệu khi thực hiện lệnh LDR R1, R2, x1D

Hình 4.11 trình bày đường truyền dữ liệu với các bộ phận liên quan khi thực hiện lệnh này. Đầu tiên nội dung của thanh ghi R2 (x2345) được cộng với trị 6 bit được mở rộng dấu trong vùng IR[5:0] (x001D), kết quả (x2362) được nạp vào thanh ghi MAR. Sau đó, bộ nhớ được đọc, và nội dung của ô nhớ x2362 được nạp vào thanh ghi MDR. Giá trị giá trị trong ô nhớ x2362 là x0F0F. Sau cùng, nội dung của MDR, tức x0F0F, được nạp vào R1.

4.7.4 Immediate mode

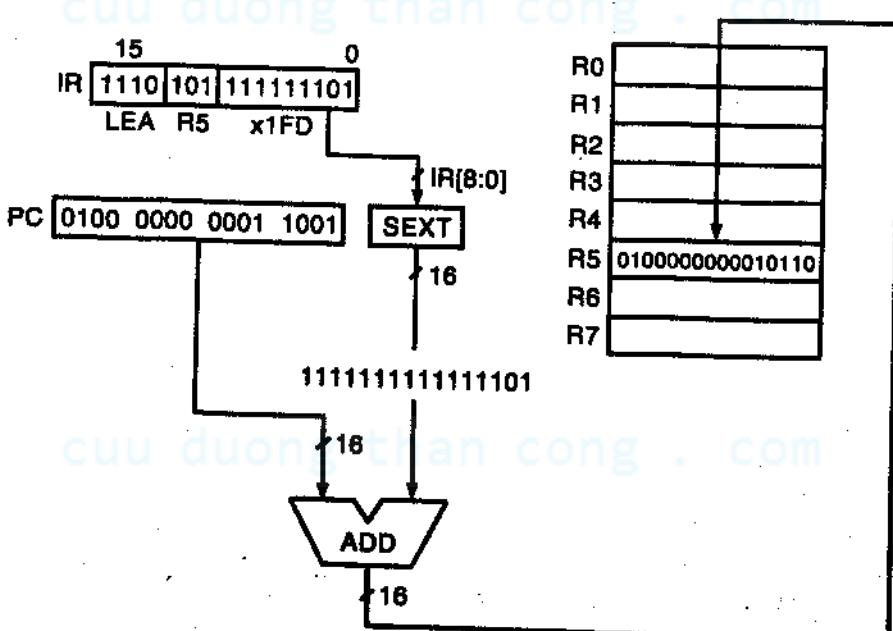
Đây là kiểu định vị cuối cùng được các lệnh sao chép dữ liệu sử dụng. Nó chỉ được sử dụng trong lệnh LEA (opcode = 1110). Lệnh này nạp vào thanh ghi được xác định trong IR[11:9] giá trị được tạo ra từ thanh ghi PC đã tăng với 9 bit mở rộng dấu của lệnh, tức IR[8:0]. Số đỉ có tên là kiểu định vị tức thời vì toán hạng được nạp vào thanh ghi đích được lấy trực tiếp từ lệnh, không qua truy cập bộ nhớ. Lệnh LEA rất hữu ích để khởi tạo thanh ghi.

Ví dụ 4.12 Nếu ô nhớ x4018 chứa lệnh LEA R5, #-3, và thanh ghi PC chứa x4018, ta có lệnh

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	1	1	1	1	1	1	1	0	1

LDI R5 -3

thì R5 sẽ chứa x4016 sau khi lệnh ở x4018 được thực thi.



Hình 4.12 Đường truyền dữ liệu khi thực hiện lệnh LEA R5, #-3

Hình 4.9 trình bày đường truyền dữ liệu khi thực hiện lệnh LEA R5, #-3 trên. Độc giả có thể dễ dàng tự giải thích quá trình thực hiện lệnh này.

4.7.5 Ví dụ

Để hiểu rõ hơn các kiểu định vị địa chỉ, chúng ta hãy xét ví dụ sau. Giả sử các lệnh được chứa trong các ô nhớ từ địa chỉ x30F6 tới x30FC như hình 4.13 sau.

Địa chỉ lệnh	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	Thao tác
x30F6	1 1 1 0 0 0 1 1 1 1 1 1 1 1 0 1	R1 ← PC - 3
x30F7	0 0 0 1 0 1 0 0 0 1 1 0 1 1 1 0	R2 ← R1 + 14
x30F8	0 0 1 1 0 1 0 1 1 1 1 1 1 0 1 1	M[x30F4] ← R2
x30F9	0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0	R2 ← 0
x30FA	0 0 0 1 0 1 0 0 1 0 1 0 0 1 0 1	R2 ← R2 + 5
x30FB	0 1 1 1 0 1 0 0 0 1 0 0 1 1 1 0	M[R1+14] ← R2
x30FC	1 0 1 0 0 1 1 1 1 1 1 1 0 1 1 1	R3 ← M[M[x30F4]]

Hình 4.13 Ví dụ về các kiểu định vị

Lúc này thanh ghi PC chứa x30F6, nên lệnh đầu tiên cần được thực thi là ở vị trí x30F6. Mã thao tác của lệnh này là 1110, tức lệnh Load Effective Address (LEA). Lệnh này nạp thanh ghi được xác định bởi các bit [11:9] (thanh ghi R1) bằng địa chỉ được tạo bởi tổng của 9 bit được mở rộng bit dấu [8:0] (xFFFFD) với thanh ghi PC đã tăng trị (x30F7). Kết quả là sau khi lệnh này được thực thi, thanh ghi R1 chứa x30F4, và PC đang lưu x30F7.

Lệnh thứ hai ở địa chỉ x30F7, với opcode = 0001 là lệnh ADD, thực hiện tính tổng giữa thanh ghi xác định bởi các bit [8:6] (thanh ghi R1) với năm bit [4:0] sau phép mở rộng bit dấu (x000E) (vì bit [5] bằng 1), và lưu kết quả vào thanh ghi đích được xác định bởi bit [11:9] (thanh ghi R2). Lệnh trước đã nạp trị x30F4 vào thanh ghi R1, nên lệnh này tạo ra kết quả x3012 và nạp vào thanh ghi R2. Sau khi lệnh này được thực thi xong, R2 chứa x3012, PC chứa x30F8, và R1 vẫn đang chứa x30F4.

Lệnh thứ ba ở địa chỉ x30F8, opcode = 0011, là lệnh ST. Lệnh này lưu giá trị từ thanh ghi nguồn xác định bởi các bit [11:9] vào ô nhớ có địa chỉ xác định từ tổng của thanh ghi PC đã được tăng trị (x30F9) và chín bit [8:0] được mở rộng bit dấu (xFFFFB), tức địa chỉ x30F4. Cuối lệnh này, ô nhớ địa chỉ x30F4 chứa x3102, và PC chứa x30F9.

Lệnh kế ở địa chỉ x30F9 có opcode là 0101, tức lệnh AND. Với các vùng bit tương tự, lệnh này nạp trị 0 vào thanh ghi R2, và cuối lệnh này PC chứa x30FA.

Lệnh ở địa chỉ x30FA có opcode là 0001, tức lệnh ADD. Với bit [5] là 1, lệnh này thực hiện tính tổng giữa R2 và 5, kết quả R2 chứa trị 5, và PC bằng x30FB.

Lệnh ở ô nhớ x30FB có opcode = 0111, tức lệnh STR. Lệnh này tương tự lệnh LDR, dùng kiểu định vị địa chỉ Base+offset, với Base là thanh ghi xác định từ các bit [8:6] (thanh ghi R1), và offset là vùng bit [5:0] được mở rộng bit dấu (tức x000E). R1 vẫn đang chứa x30F4, nên địa chỉ ô nhớ là x30F4 + x000E, tức x3102. Lệnh STR sẽ lưu trị từ thanh ghi R2 (xác định từ các bit [11:9]), tức 5, vào ô nhớ địa chỉ x3102. Sau khi lệnh này được thực thi xong, ô nhớ x3102 chứa trị 5, PC lưu trị x30FC.

Ở ô nhớ x30FC, opcode là 1010, ta có lệnh LDI. Tương tự lệnh STI, lệnh này dùng kiểu định vị gián tiếp. Đầu tiên, thanh ghi PC đã được tăng (x30FD) sẽ được cộng với chín bit [8:0] sau phép mở rộng bit dấu (-9 = xFFF7) theo kiểu định vị PC relative để có x30F4. Đây là địa chỉ của ô nhớ chứa toán hạng cần làm việc. Mặt khác, từ cuối lệnh thứ ba, ta có ô nhớ x30F4 chứa x3102, và ở cuối lệnh trên, ta có ô nhớ x3102 đang chứa trị 5. Như vậy, cuối lệnh này, thanh ghi được xác định từ các bit [11:9], tức thanh ghi R3, được nạp trị là 5, và thanh ghi PC chứa x30FD.

4.8 NHÓM LỆNH ĐIỀU KHIỂN

Các lệnh điều khiển thay đổi trình tự các lệnh thực thi trong chương trình. LC-3 có năm mã lệnh thực hiện việc này: lệnh rẽ nhánh (*Branch*) có điều kiện, lệnh nhảy (*Jump*) không điều kiện, gọi chương trình con, TRAP, và lệnh trả về từ ngắt (*Interrupt*). Trong mục này chủ yếu chúng ta nói về lệnh rẽ nhánh có điều kiện, lệnh nhảy không điều kiện, và TRAP.

4.8.1 Lệnh rẽ nhánh có điều kiện

Định dạng của lệnh rẽ nhánh có điều kiện (opcode = 0000) như sau:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	N	Z	P									PCoffset

Các bit [11], [10], và [9] tương ứng với ba mã điều kiện đã được nêu trong mục 4.5.7. Lưu ý rằng trong LC-3, tất cả các lệnh mà có ghi trị vào các thanh ghi đa dụng sẽ đặt trị cho ba mã điều kiện (tức các thanh ghi một bit) tương ứng với trạng thái giá trị được ghi vào thanh ghi. Các lệnh đó là ADD, AND, NOT, LD, LDI, LDR, và LEA.

Các mã điều kiện được lệnh rẽ nhánh có điều kiện sử dụng để xác định xem có thay đổi trình tự lệnh hay không; nghĩa là, xem sự thực thi lệnh có theo trình tự thường thấy như là kết quả của việc tăng thanh ghi PC trong phase FETCH của mỗi lệnh hay không.

Trong khi thực hiện phase EXECUTE, bộ xử lý khảo sát các mã điều kiện mà các bit tương ứng của nó trong lệnh là 1. Nếu bit [11] là 1, mã điều kiện N được xem xét. Nếu bit [10] là 1, mã điều kiện Z sẽ được kiểm tra. Nếu bit [9] là 1, mã điều kiện P phải được xét. Nếu các bit [11:9] trong lệnh đều là 0, không có mã điều kiện tương ứng nào được kiểm tra. Nếu có một mã điều kiện mà được kiểm tra và thấy bằng 1, thanh ghi PC được nạp bằng địa chỉ có được trong phase EVALUATE ADDRESS. Ngược lại, khi không có một mã điều kiện mà được kiểm tra và thấy bằng 1, PC giữ trị không đổi. Trong trường hợp này, lệnh theo trình tự kế tiếp sẽ được lấy trong chu kỳ lệnh kế.

Ví dụ 4.13 Nếu giá trị cuối cùng được nạp vào một thanh ghi đa dụng nào đó là 0, thì lệnh hiện thời (ở ô nhớ x4027) dưới đây sẽ nạp thanh ghi PC bằng trị x4101 ($x4028 + x0D9$), tức lệnh kế cần thực thi sẽ ở ô nhớ x4101, chứ không phải ở ô nhớ x4028.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	1	1	0	1	1	0	0	1

BRuu long han cong . com

Hình 4.14 nêu các phần tử trong đường truyền dữ liệu khi thực hiện lệnh này. Trạng thái logic được yêu cầu để kiểm tra xem dòng lệnh theo trình tự có bị thay đổi hay không. Trong ví dụ này, câu trả lời là có (Yes), nên thanh ghi PC được nạp trị x4101 thay cho trị hiện thời x4028, vốn đã được nạp trong phase FETCH của lệnh rẽ nhánh có điều kiện.

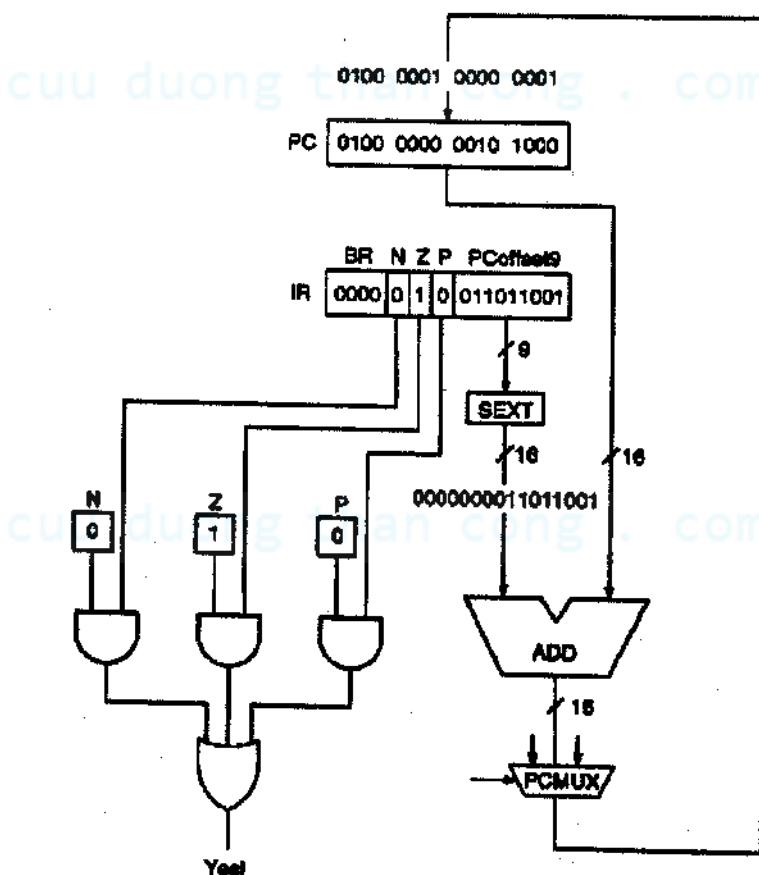
Nếu tất cả ba bit [11:9] đều là 1, thì cả ba mã điều kiện đều được kiểm tra. Khi đó, vì kết quả cuối cùng được lưu vào một thanh ghi phải

hoặc là âm, không, hoặc dương (không có trường hợp khác), nên ta phải có một trong ba mã điều kiện phải ở trạng thái 1. Kết quả so sánh cho câu trả lời Yes, thanh ghi PC được nạp trị bằng địa chỉ tính được trong phase EVALUATE ADDRESS. Chúng ta gọi trường hợp này là rẽ nhánh không điều kiện vì trình tự lệnh bị thay đổi bất chấp điều kiện (vì lúc này điều kiện để rẽ nhánh chương trình luôn đúng).

Ví dụ 4.14 Nếu lệnh sau đây

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	1	0	0	0	0	1	0	1
BR	n	z	p										x185		

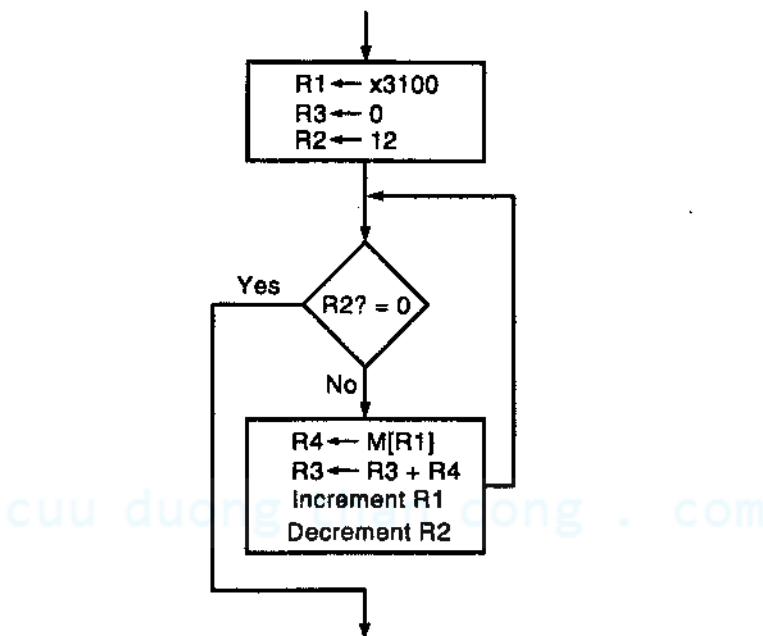
Ở ô nhớ x507B được thực thi, thì thanh ghi PC được nạp trị x5201. Đọc giả hãy suy nghĩ xem nếu tất cả ba bit [11:9] trong lệnh BR đều bằng không, điều gì sẽ xảy ra?



Hình 4.14 Đường truyền dữ liệu khi thực thi lệnh `BRz x0D9`

4.8.2 Ví dụ

Giả sử chúng ta đã có 12 ô nhớ từ x3100 tới x310B chứa 12 số nguyên mà chúng ta cần tính tổng. Lưu đồ của giải thuật cho vấn đề được trình bày trong hình 4.15 dưới đây.



Hình 4.15 Lưu đồ cộng 12 số nguyên

Đầu tiên, ta cần khởi động trị cho các biến cần sử dụng. Ở đây có ba biến: biến lưu địa chỉ của số nguyên kế trong mảng cần cộng (thanh ghi R1), biến lưu tổng (thanh ghi R3), và số số nguyên chưa cộng còn lại (thanh ghi R2). Các biến này được khởi động trị như sau: địa chỉ của số nguyên đầu tiên cần cộng được gán vào R1, tức sau khởi động, R1 bằng x3100; thanh ghi R3 giữ tổng, nên cần được khởi động trị bằng 0; thanh ghi R2 lưu số phần tử số nguyên còn lại cần được cộng, nên được khởi động trị là 12. Sau đó quá trình cộng bắt đầu.

Chương trình lặp lại quá trình nạp thanh ghi R4 từng giá trị trong 12 số nguyên để được cộng vào R3. Mỗi lần thực hiện phép cộng ADD, chúng ta tăng R1 để nó chỉ tới (tức chứa địa chỉ của) phần tử số nguyên kế tiếp cần được cộng, và giảm thanh ghi R2. Khi đó qua R2, chúng ta biết được còn bao nhiêu phần tử còn lại cần cộng tiếp. Khi R2 tới 0, mã điều kiện Z được thiết lập (bằng 1), và chúng ta thấy là đã hoàn thành chương trình.

Chương trình gồm 10 lệnh thực hiện việc cộng này được trình bày trong hình 4.16.

Địa chỉ lệnh	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	Thao tác
x3000	1 1 1 0 0 0 1 0 1 1 1 1 1 1 1 1	R1 ← 3100
x3001	0 1 0 1 0 1 1 0 1 1 1 0 0 0 0 0	R3 ← 0
x3002	0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0	R2 ← 0
x3003	0 0 0 1 0 1 0 0 1 0 1 0 1 1 0 0	R2 ← 12
x3004	0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 1	BRz x300A
x3005	0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 0	R4 ← M[R1]
x3006	0 0 0 1 0 1 1 0 1 1 0 0 0 1 0 0	R3 ← R3 + R4
x3007	0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1	R1 ← R1 + 1
x3008	0 0 0 1 0 1 0 0 1 0 1 1 1 1 1 1	R2 ← R2 - 1
x3009	0 0 0 0 1 1 1 1 1 1 1 1 1 0 1 0	BRnzp x3004

Hình 4.16 Chương trình hiện thực giải thuật ở hình 4.15

Chương trình này bắt đầu với thanh ghi PC = x3000. Lệnh đầu tiên (ở địa chỉ x3000) nạp thanh ghi R1 bằng trị x3100. Thanh ghi PC đã tăng lên x3001, vùng PCoffset được mở rộng dấu là x00FF.

Lệnh ở x3001 xóa R3 về 0, vốn được sử dụng để giữ tổng.

Lệnh ở x3002 và x3003 gán trị 12 cho thanh ghi R2, tức số phần tử cần được cộng. R2 theo dõi số phần tử này cũng đồng thời là biến điều khiển vòng lặp, nên sau khi đã cộng phần tử vào biến tổng (R3), thanh ghi R2 sẽ bị giảm, tiến tới kết thúc lặp.

Lệnh ở x3004 là một lệnh rẽ nhánh có điều kiện, ta thấy bit [10] bằng 1, tức mã điều kiện Z được kiểm tra. Nếu mã này bằng 1, khi đó R2 đã bằng 0, tức không còn số nào cần được cộng nữa, và ta đã hoàn thành nhiệm vụ. Nếu mã này bằng 0 (điều kiện kiểm tra là sai), chúng ta tiếp tục thực hiện công việc.

Lệnh ở x3005 nạp nội dung của ô nhớ ở x3100 (tức số nguyên đầu tiên) vào thanh ghi R4, và lệnh ở x3006 cộng nó vào thanh ghi R3.

Lệnh ở x3007 và x3008 thực hiện các thao tác cần thiết khác. Cụ thể, lệnh ở x3007 tăng R1 để nó chỉ tới ô nhớ kế chứa số nguyên cần cộng (tức x3101). Còn lệnh ở x3008 giảm R2, vốn đang theo dõi

số phần tử nguyên còn lại cần được cộng, và thiết lập trị cho các mã điều kiện N, Z, P.

Lệnh ở x3009 là lệnh rẽ nhánh không điều kiện, vì với ba bit [11:9] được bật, việc kiểm tra đương nhiên đúng để rẽ nhánh. Lệnh này nạp PC bằng giá trị x3004, và không ảnh hưởng tới trạng thái đang có của các mã điều kiện. Khi lệnh kế tiếp được thực hiện (tức lệnh rẽ nhánh có điều kiện ở x3004), các trạng thái đang có của các mã điều kiện này được kiểm tra, tức trạng thái có được từ lệnh ở x3008 trước đó. Điều đó có nghĩa là lệnh x3008 trước đó thay đổi trị của R2, xác định số phần tử còn lại trong mảng cần cộng. Lệnh này có thể làm cho R2 bằng 0 khi không còn phần tử nào cần cộng, khi đó mã điều kiện Z = 1, còn N và P bằng 0; ngược lại, nếu R2 còn khác 0 (và lớn hơn 0), tức còn phần tử cần cộng, thì mã điều kiện sẽ là N và Z = 0, P = 1. Các mã điều kiện này được giữ nguyên để lệnh BRz ở x3004 kiểm tra. Trong trường hợp đầu, khi R2 = 0, tức Z = 1, lệnh BRz với bit [10] là 1 sẽ nạp thanh ghi PC trị x300A, tức qua tác vụ khác. Còn khi R2 lớn hơn 0, tức Z = 0, lệnh BRz với bit [10] là 1 sẽ không nạp trị x300A vào PC, PC vẫn đang giữ x3005 và lệnh ở vị trí này theo trình tự sẽ được thực thi.

Như vậy, lệnh rẽ nhánh có điều kiện sẽ quy định trình tự các lệnh được thực thi là ở x3000, x3001, x3002, x3003, x3004, x3005, x3006, x3007, x3008, x3009, x3004, x3005, x3006, x3007, x3008, x3009, x3004, x3005, ... cho tới khi trị trong thanh ghi R2 bằng 0. Sau đó, thanh ghi PC được nạp trị x300A, và chương trình qua một tác vụ khác.

4.8.3 Hai phương pháp điều khiển lặp

Chúng ta dùng thuật ngữ lặp (*loop*) để mô tả chuỗi lệnh cần được thực thi lặp đi lặp lại theo một cơ chế điều khiển nào đó. Trong ví dụ cộng 12 số nguyên ở trên, vòng lặp đã được sử dụng. Mỗi khi thân vòng lặp được thực thi, một số nguyên nữa được cộng vào tổng, và bộ đếm được giảm. Điều này làm chúng ta biết được còn lại bao nhiêu trị cần được cộng. Mỗi khi thân vòng lặp được thực hiện ta gọi là sự lặp lại (*iteration*) của vòng lặp.

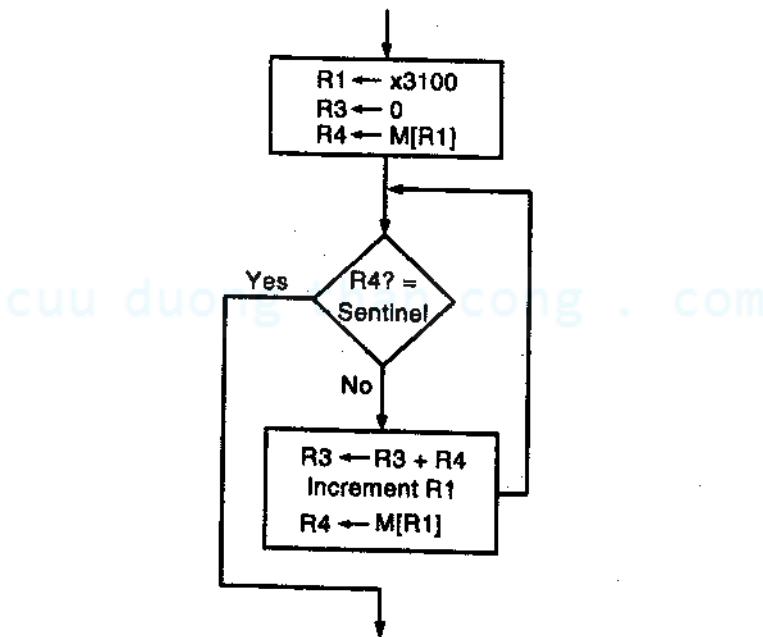
Có hai phương pháp chung để điều khiển số lần lặp lại của thân vòng lặp. Trong ví dụ ở mục trên, phương pháp biến đếm được sử dụng. Nếu chúng ta biết chính xác số lần lặp là n lần, thì đơn giản

chúng ta chỉ cần gán cho biến đếm trị n, và sau mỗi lần lặp, chúng ta cần giảm biến đếm này và kiểm tra nó xem có là 0 hay chưa. Nếu chưa là 0, chúng ta đặt trị cho thanh ghi PC bằng địa chỉ của lệnh đầu thân vòng lặp, và tiếp tục sự lặp lại.

Phương pháp thứ hai là dùng trị canh (*sentinel*). Cách này đặc biệt hiệu quả khi chúng ta không biết trước có bao nhiêu sự lặp lại cần phải được thực hiện. Thông thường quá trình lặp sẽ là dãy các sự kiện cần xử lý, chúng ta cần thêm vào dãy sự kiện này một sự kiện là kiểm tra một giá trị mà chúng ta biết trước hay theo quy ước là không bao giờ xuất hiện trong các sự kiện gốc. Nếu việc kiểm tra này xảy ra, tức việc lặp kết thúc. Giá trị đó được gọi là trị canh. Ví dụ, để bài yêu cầu tính tổng các trị nguyên dương nhập từ bàn phím. Việc nhập trị kết thúc khi gặp trị 0 (hoặc một trị âm). Khi đó trị 0 (hay trị âm) được gọi là trị canh.

4.8.4 Ví dụ

Ví dụ này có yêu cầu giống như mục 4.8.2, tức tính tổng các số, nhưng dùng trị canh trong giải thuật. Giả sử chúng ta cần cộng các trị nguyên dương từ ô nhớ x3100 tới x310B. Khi đó chúng ta có thể dùng bất kỳ trị âm nào để làm trị canh. Giả sử trị canh được chứa ở ô nhớ x310C là -1. Lưu đồ và chương trình được nêu ở hình 4.17 và 4.18 dưới đây.



Hình 4.17 Giải thuật sử dụng một trị canh điều khiển lặp

Địa chỉ lệnh	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	Thao tác
x3000	1 1 1 0 0 0 1 0 1 1 1 1 1 1 1 1	R1 ← x3100
x3001	0 1 0 1 0 1 1 0 1 1 1 0 0 0 0 0	R3 ← 0
x3002	0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0	R4 ← M[R1]
x3003	0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0	BRn x3008
x3004	0 0 0 1 0 1 1 0 1 1 0 0 0 1 0 0	R3 ← R3 + R4
x3005	0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1	R1 ← R1 + 1
x3006	0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 0	R4 ← M[R1]
x3007	0 0 0 0 1 1 1 1 1 1 1 1 1 0 1 1	BRnzp x3003

Hình 4.18 Chương trình hiện thực giải thuật ở hình 4.17

Cũng giống như trước, lệnh ở x3000 nạp thanh ghi R1 bằng địa chỉ của trị nguyên đầu tiên cần cộng, lệnh ở x3001 khởi tạo R3 (giữ tổng) trị 0.

Ở ô nhớ x3002, chúng ta nạp trị nguyên đầu tiên cần tính tổng vào thanh ghi R4. Nếu trị canh được nạp, thanh ghi một bit mã điều kiện N được dựng trị bằng 1. Ngược lại, N bằng 0.

Lệnh rẽ nhánh có điều kiện ở x3003 kiểm tra mã điều kiện N, nếu N đã được dựng bằng 1, thì PC được đặt trị x3008 để qua tác vụ kế, tức tổng đã được tính xong. Còn nếu mã N đang bị xóa bằng 0, R4 đang chứa trị cần cộng là dương, nên sẽ được cộng vào R3 do lệnh ở x3004. R1 cần được tăng lên để chỉ tới ô nhớ chứa số nguyên kế tiếp ở lệnh x3005. Sau đó R4 được nạp trị nguyên mới này ở lệnh x3006, còn PC được nạp trị x3003 để bắt đầu chu trình lặp mới.

4.8.5 Lệnh JMP

Lệnh rẽ nhánh có điều kiện có giới hạn cho lệnh kế tiếp vì địa chỉ lệnh này được tạo ra từ tổng của trị PC đã tăng với các bit [8:0] offset ở dạng bù 2 được mở rộng bit dấu thành 16 bit. Nếu coi trị trong thanh ghi PC là mốc (tức địa chỉ của lệnh hiện thời cộng 1), thì do offset có 9 bit có dấu, nên lệnh kế tiếp sẽ nằm trong tầm PC - 256 → PC + 255. Nhưng nếu chúng ta muốn thực thi một lệnh mà lệnh đó cách lệnh hiện thời 1000 vị trí thì sao? Với chín bit offset trong lệnh rẽ nhánh có điều kiện ta không thể làm được điều này.

ISA LC-3 cung cấp lệnh JMP (opcode = 1100) làm được việc này. Lệnh có định dạng như ví dụ sau:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0

JMP

R2

Lệnh JMP nạp thanh ghi PC bằng trị của thanh ghi xác định bởi các bit [8:6] của lệnh. Nếu lệnh JMP nằm ở địa chỉ x4000, R2 chứa trị x6600, và PC chứa x4000, thì lệnh JMP ở x4000 sẽ được thực thi, làm cho lệnh kế tiếp là lệnh ở x6600. Vì các thanh ghi đều dài 16 bit, có thể mã hóa địa chỉ cho cả bộ nhớ 64K của ISA LC-3, nên lệnh JMP không có giới hạn cho lệnh kế cần được thực thi.

4.8.6 Lệnh TRAP

Lệnh TRAP (opcode = 1111) cho phép lấy dữ liệu vào và xuất dữ liệu ra khỏi máy tính và có định dạng như sau:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0								Trap vector

Lệnh này thay đổi trị của thanh ghi PC, làm nó chỉ tới một vị trí trong bộ nhớ mà là một phần của hệ điều hành để hệ điều hành thực hiện một tác vụ nào đó nhân danh chương trình đang được thực thi. Theo kiểu nói của ngôn ngữ hệ điều hành, chúng ta nói lệnh TRAP gọi một dịch vụ của hệ điều hành (service call). Các bit [7:0] của lệnh TRAP tạo nên một trapvector 8 bit, xác định dịch vụ cần gọi mà chương trình muốn hệ điều hành thực hiện. Một số dịch vụ chúng ta cần biết để sử dụng ngay là:

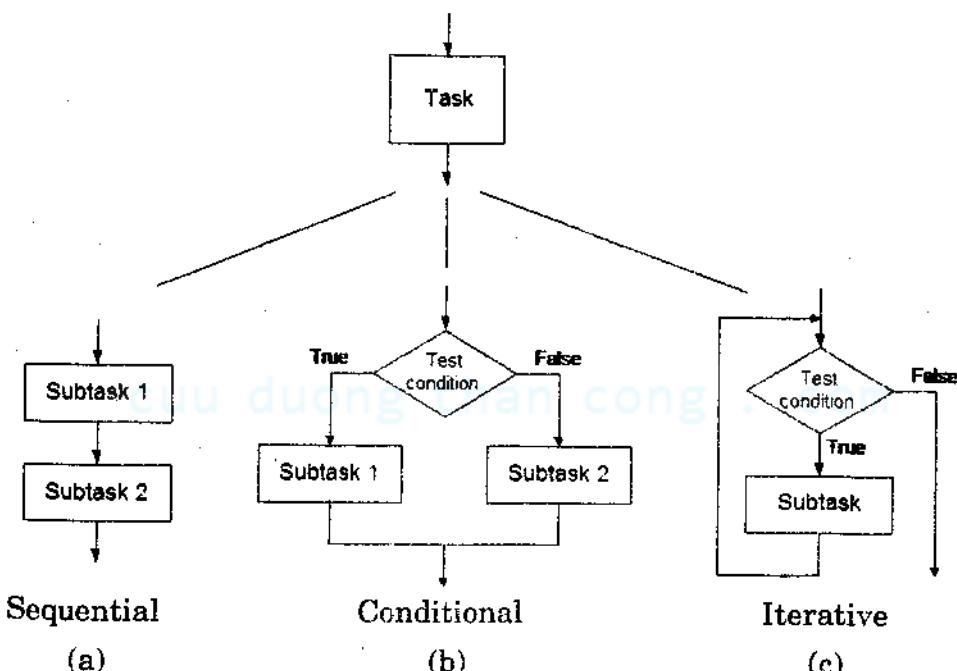
- Nhập một ký tự từ bàn phím: trapvector = x23
- Xuất một ký tự ra màn hình: trapvector = x21
- Kết thúc chương trình: trapvector = x25.

Khi hệ điều hành hoàn tất dịch vụ, bộ đếm chương trình PC sẽ được đặt trở lại địa chỉ của lệnh ngay sau lệnh TRAP trong chương trình đang được thực thi, và chương trình tiếp tục chạy. Đây cũng là khả năng thực hiện giao tiếp của LC-3 giữa hệ điều hành và chương trình đang được thực thi.

4.9 BA CẤU TRÚC LỆNH TRONG LC-3

4.9.1 Ba cấu trúc cơ bản trong lập trình có cấu trúc

Trong lập trình có ba cấu trúc lệnh cơ bản để thực hiện các tác vụ cần thiết, đó là các cấu trúc tuần tự (*sequential*), điều kiện (*conditional*), và lặp (*iterative*). Hình 4.19 trình bày ba cấu trúc cơ bản triển khai tác vụ cần thực thi (*task*).



Hình 4.19 Ba cấu trúc lệnh của lập trình cấu trúc

Cấu trúc tuần tự (Hình 4.19a) được dùng khi tác vụ (*task*) cần thực hiện được chia ra làm hai tác vụ con, tác vụ này làm trước, tác vụ kia làm sau. Có nghĩa là, máy tính thực thi hoàn toàn tác vụ con đầu tiên (*subtask 1*) trước, và rồi tiếp tới thực thi hoàn toàn tác vụ con thứ hai (*subtask 2*) mà không bao giờ quay trở lại tác vụ con đầu tiên sau khi bắt đầu tác vụ con thứ hai.

Cấu trúc điều kiện (Hình 4.19b) được dùng khi tác vụ cần thực thi cần phải chọn một trong hai tác vụ con cần thực thi, tùy vào điều kiện (*condition*) nào đó, chứ không phải cả hai. Nếu điều kiện là đúng (*true*), máy tính thực hiện tác vụ 1 (*subtask 1*). Nếu điều kiện là sai (*false*), máy tính thực hiện tác vụ 2 (*subtask 2*). Tác vụ 2 có thể rỗng,

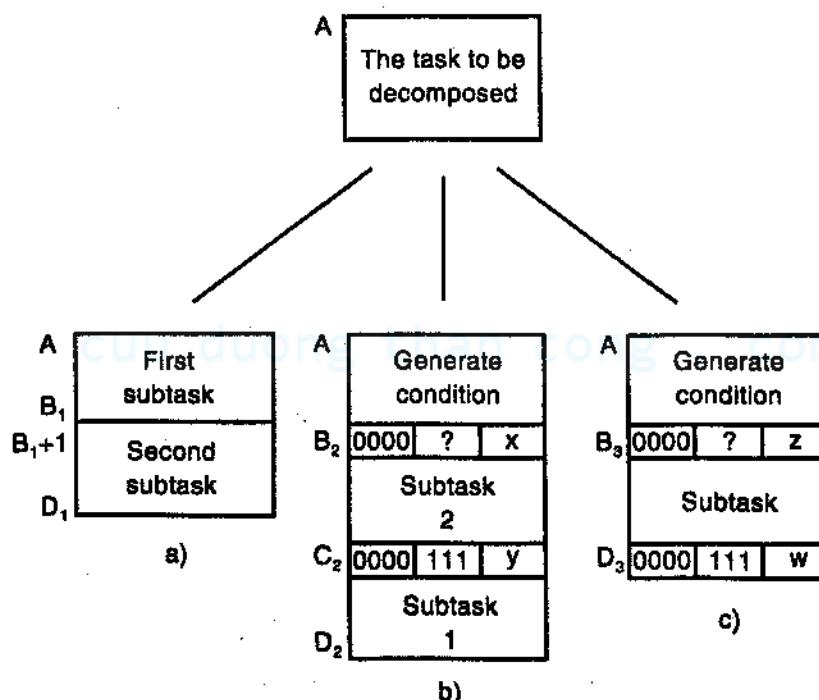
tức nó không làm gì cả. Sau khi tác vụ con ứng với điều kiện thích hợp được thực thi hoàn chỉnh, chương trình chuyển qua thực thi lệnh kế, và không bao giờ quay lại, kiểm tra điều kiện.

Cấu trúc lặp (Hình 4.19c) được sử dụng khi tác vụ cần thực thi cần phải thực hiện tác vụ con một số lần khi điều kiện xác định là đúng. Sau khi tác vụ con được hoàn thành, điều khiển chương trình quay trở lại và kiểm tra lại điều kiện. Khi kết quả kiểm tra điều kiện này là đúng, chương trình tiếp tục thực hiện tác vụ con. Lần đầu tiên khi việc kiểm tra này là sai, thì chương trình thực thi lệnh kế.

4.9.2 Ba cấu trúc trong LC-3

Üng với ba cấu trúc cơ bản của lập trình có cấu trúc trên, việc sử dụng các lệnh điều khiển của LC-3 để điều khiển thanh ghi đếm chương trình PC cũng cho phép thực hiện ba cấu trúc này. Hình 4.19a, 4.19b, và 4.19c tương ứng với ba cấu trúc ở hình 4.20a, 4.20b, và 4.20c.

Chúng ta dùng các ký tự A, B, C, và D biểu diễn địa chỉ trong bộ nhớ chứa các lệnh LC-3. Ví dụ, A được dùng trong cả ba trường hợp để biểu diễn địa chỉ của lệnh LC-3 đầu tiên cần thực thi.



Hình 4.20 Ba cấu trúc lệnh trong LC-3

Hình 4.20a trình bày dòng điều khiển của chương trình theo trình tự phân tích. Ở đây không có các lệnh điều khiển nào mà chỉ có hai tác vụ con. Sau khi xong tác vụ con đầu tiên, thanh ghi PC được tăng từ địa chỉ B1 qua địa chỉ $B1 + 1$. Chương trình tiếp tục thực hiện các lệnh tới địa chỉ D1. Nó không quay lại tác vụ con đầu tiên.

Hình 4.20b cho thấy dòng điều khiển chương trình với phân tích có điều kiện. Đầu tiên, một điều kiện được tạo ra, kết quả làm dựng một trong các mã điều kiện. Điều kiện này với mã điều kiện được kiểm tra bằng lệnh rẽ nhánh có điều kiện ở địa chỉ B2. Nếu điều kiện là đúng, PC được đặt tới địa chỉ C2 + 1, và tác vụ con 1 (*subtask 1*) được thực thi. Chú ý rằng, x trong lệnh ở B2 tương ứng với số lệnh trong tác vụ con 2 (*subtask 2*). Nếu điều kiện là sai, PC (vốn đã được tăng trong phase FETCH của lệnh rẽ nhánh) lấy lệnh ở địa chỉ $B2 + 1$, và tác vụ con 2 (*subtask 2*) được thực thi. Tác vụ con 2 kết thúc bằng lệnh rẽ nhánh không điều kiện ở địa chỉ C2 để nhảy tới lệnh ở địa chỉ D2 + 1.

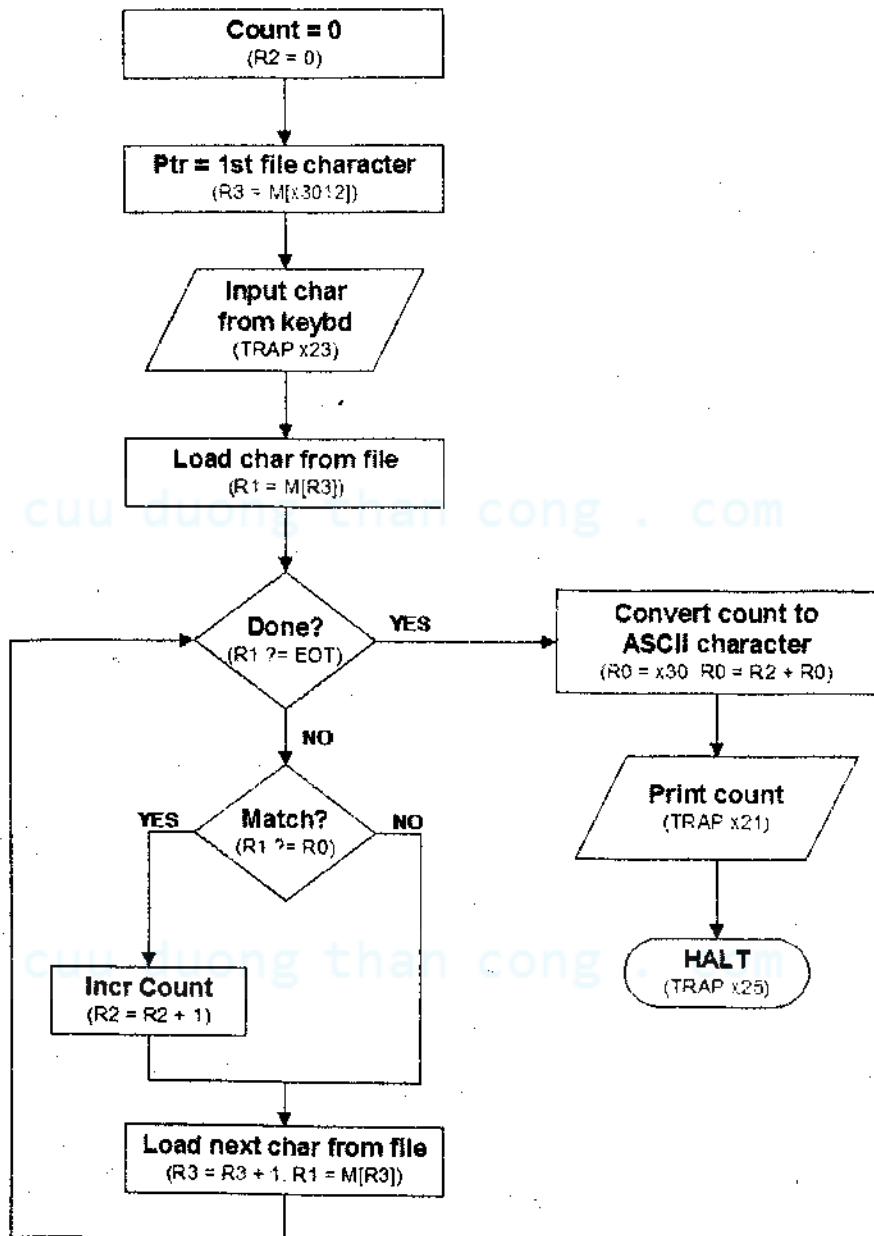
Hình 4.20c minh họa dòng điều khiển của chương trình khi phân tích lặp. Như trong trường hợp cấu trúc có điều kiện, đầu tiên một điều kiện được tạo ra, mã điều kiện tương ứng được dựng, và một lệnh rẽ nhánh có điều kiện được thực thi. Trong trường hợp này, các bit điều kiện của lệnh ở địa chỉ B3 được dựng để tạo ra một rẽ nhánh có điều kiện nếu điều kiện được tạo ra là đúng. Khi đó, thanh ghi PC được đặt tới địa chỉ $D3 + 1$. Chú ý rằng, z trong lệnh ở B3 tương ứng với số lệnh trong tác vụ con (*subtask*) ở hình 4.20c. Mặt khác, khi điều kiện là sai, thanh ghi PC vốn đã được tăng tới $B3 + 1$ (trong phase FETCH của lệnh rẽ nhánh B3), và tác vụ con (*subtask*) bắt đầu từ đây được thực thi cho tới lệnh rẽ nhánh không điều kiện ở địa chỉ D3. Lệnh này đặt PC tới A để tạo và kiểm tra điều kiện trở lại. Chú ý rằng, w trong lệnh ở D3 tương ứng với toàn bộ số lệnh trong hình 4.20c.

Ví dụ sau đây minh họa cho tất cả các vấn đề này.

4.10 MỘT VÍ DỤ

Mục này trình bày một ví dụ hoàn chỉnh về ISA LC-3: đếm số lần xuất hiện của một ký tự xác định trước (được nhập từ bàn phím) trong một mảng ký tự (*file*) cho trước. Sau đó hiển thị số lần xuất hiện này ra màn hình. Nếu ví dụ này được giải một cách tổng quát

thì số lần xuất hiện là tùy ý, tuy nhiên để đơn giản vấn đề, ta hãy chấp nhận số lần xuất hiện tối đa của một ký tự là 9 để chúng ta không lo lắng về các thủ tục chuyển đổi phức tạp giữa số lần đếm ở số nhị phân và dạng hiển thị ASCII ra màn hình của nó.



Hình 4.21 Giải thuật đếm số lần xuất hiện của ký tự

Hình 4.21 là lưu đồ của giải thuật giải quyết vấn đề. Chú ý rằng mỗi bước trong lưu đồ được diễn ta bằng tiếng Anh và bằng mã LC-3 (phần trong ngoặc).

Bước đầu tiên luôn luôn là khởi động trị cho tất cả các biến. Ở đây là các thanh ghi R0, R1, R2, và R3. Thanh ghi R2 giữ số lần xuất hiện của ký tự, trong lưu đồ là count. Nó được khởi động trị là 0. Thanh ghi R3 lưu địa chỉ của ký tự kế trong file cần được kiểm tra, nên nó sẽ được khởi động bằng trị địa chỉ (của ký tự đầu tiên) đang được chứa trong ô nhớ ở cuối chương trình, tức M[x3012]. Cần lưu ý rằng, giả sử chương trình của chúng ta bắt đầu ở địa chỉ x3000, thì để hợp lý địa chỉ của file chứa các ký tự có thể bắt đầu ở địa chỉ x9000, file dữ liệu này cần được khởi tạo xong trước khi chạy chương trình. Nên thanh ghi R3 được khởi động trị là x9000. Thanh ghi R0 giữ ký tự cần đếm xuất hiện trong file và có được từ việc nhập vào từ bàn phím. Thanh ghi R1 giữ từng ký tự đọc được từ file để kiểm tra xem có đúng là ký tự cần đếm hay không. Do đó, nó được khởi động trị bằng ký tự đầu tiên trong file được chỉ bởi R3.

Bước kế tiếp là đếm số lần xuất hiện của ký tự đã được nhập. Việc này được làm bằng cách đọc từng ký tự trong file đang được khảo sát cho tới khi gặp ký tự quy ước kết thúc file. Quá trình này cần một chu trình lặp. Vòng lặp ở đây sử dụng phương pháp trị canh, là ký tự EOT (End of Text) có mã ASCII là x04 (00000100).

Trong mỗi chu trình lặp, nội dung của thanh ghi R1 luôn được so sánh với mã ASCII của ký tự EOT. Nếu chúng bằng nhau, vòng lặp không được thực thi tiếp, và chương trình chuyển tới bước cuối cùng là hiển thị ra màn hình số lần xuất hiện của ký tự. Nếu việc so sánh cho kết quả không bằng nhau, R1 (chứa ký tự hiện thời cần kiểm tra) được so sánh với R0 (ký tự nhập vào từ bàn phím). Nếu chúng giống nhau, R2 được tăng lên. Nếu không, chúng ta lấy ký tự kế, nghĩa là R3 được tăng lên, ký tự kế được nạp vào R1, và chương trình quay lại kiểm tra xem gặp trị canh ở cuối file hay chưa.

Khi đã tới cuối file, tức tất cả các ký tự đã được kiểm tra, và số lượng ký tự đã được chứa trong R2 ở dạng nhị phân. Để hiển thị số lượng này ra màn hình, chúng ta cần chuyển nó sang dạng mã ASCII. Vì chúng ta đã chấp nhận số lượng là nhỏ hơn 10 cho đơn giản, nên để chuyển nó sang mã ASCII, chúng ta chỉ cần cộng nó với x30

(001100002) là xong. Ví dụ, đang có số 5, muốn có ký tự '5' (mã ASCII là x35) để hiển thị ra màn hình, chúng ta cần có $5 + x30 = x35$. Trong chương trình, chúng ta có thể đặt mẫu cho để có bảng ASCII cho các ký tự ở một địa chỉ (như x3013 chẳng hạn). Khi cần sẽ lấy mẫu ra sử dụng.

Hình 4.22 là chương trình bằng ngôn ngữ máy hiện thực lưu đồ ở hình 4.21.

Địa chỉ lệnh	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	Thao tác
x3000	0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0	R2 ← 0
x3001	0 0 1 0 0 1 1 0 0 0 0 1 0 0 0 0	R3 ← M[x3012]
x3002	1 1 1 1 0 0 0 0 0 0 1 0 0 0 1 1	TRAP x23
x3003	0 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0	R1 ← M[R3]
x3004	0 0 0 1 1 0 0 0 0 1 1 1 1 1 0 0	R4 ← R1 - 4
x3005	0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0	BRz x300E
x3006	1 0 0 1 0 0 1 0 0 1 1 1 1 1 1 1	R1 ← NOT R1
x3007	0 0 0 1 0 0 1 0 0 1 1 0 0 0 0 1	R1 ← R1 + 1
x3008	0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0	R1 ← R1 + R0
x3009	0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1	BRnp x300B
x300A	0 0 0 1 0 1 0 0 1 0 1 0 0 0 0 1	R2 ← R2 + 1
x300B	0 0 0 1 0 1 1 0 1 1 1 0 0 0 0 1	R3 ← R3 + 1
x300C	0 1 1 0 0 0 1 0 1 1 0 0 0 0 0 0	R1 ← M[R3]
x300D	0 0 0 0 1 1 1 1 1 1 1 1 0 1 1 0	BRnzp x3004
x300E	0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0	R0 ← M[x3013]
x300F	0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0	R0 ← R0 + R2
x3010	1 1 1 1 0 0 0 0 0 0 1 0 0 0 0 1	TRAP x21
x3011	1 1 1 1 0 0 0 0 0 0 1 0 0 1 0 1	TRAP x25
x3012	Địa chỉ bắt đầu của file	
x3013	0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0	Bảng ASCII

Hình 4.22 Chương trình ngôn ngữ máy hiện thực
giải thuật hình 4.16

Đầu tiên là các bước khởi động. Lệnh ở x3000 xóa R2 bằng việc AND nó x0000; lệnh ở x3001 nạp trị ở x3012 vào R3. Đây là địa chỉ của ký tự đầu tiên trong file cần kiểm tra số lần xuất hiện của ký tự mong muốn. Một lần nữa, xin nhắc lại file ký tự là mảng dữ liệu này có thể ở bất kỳ đâu trong bộ nhớ, nó cần phải được tạo ra, và địa chỉ của file này phải được chứa vào ô nhớ địa chỉ x3012 trước khi lệnh ở x3000 được thực hiện. Ở nhớ x3002 chứa lệnh TRAP, yêu cầu hệ điều hành thực hiện gọi dịch vụ nhân danh chương trình. Hàm theo yêu cầu, như được chỉ định trong trapvector 8 bit 00100011 (tức x23), nhập một ký tự từ bàn phím và nạp nó vào thanh ghi R0. Lệnh ở x3003 nạp ký tự được chỉ bởi R3 vào R1.

Lúc này quá trình kiểm tra các ký tự bắt đầu. Chúng ta ở lệnh x3004, kiểm tra ký tự EOT (mã ASCII là 4) trong R1 bằng phép trừ, kết quả được lưu vào R4. Nếu kết quả là 0, ta đã tới cuối file, và đây là lúc xuất số lượng lý tự. Lệnh ở x3005 rẽ nhánh có điều kiện tới x300E, với quá trình xuất số trị bắt đầu.

Nếu R4 không bằng 0, ký tự trong R1 cần được kiểm tra. Các lệnh từ x3006, x3007, và x3008 xác định xem nội dung của R1 và R0 có đồng nhất hay không. Các lệnh này thực hiện thao tác sau:

$$R0 + (\text{NOT } (R1) + 1)$$

và tạo ra tất cả các bit 0 nếu R1 và R0 là giống nhau. Nếu các mẫu bit không giống nhau, lệnh rẽ nhánh có điều kiện ở x3009 điều khiển nhảy tới x300B, tức bỏ qua lệnh x300A, tăng R2, tức bộ đếm (ký tự không thích hợp, không đếm!).

Lệnh ở x300B tăng R3, nên nó chỉ tới ký tự kế trong file cần kiểm tra. Lệnh ở x300C nạp ký tự này vào R1, và lệnh ở x300D nhảy không điều kiện trở lại x3004 để bắt đầu quá trình xử lý ký tự đó.

Khi trị canh (EOT) được tìm thấy, quá trình xuất số lượng bắt đầu ở x300E. Lệnh ở x300E nạp 00110000 (x30) vào R0, và lệnh ở x300F cộng số lượng cần hiển thị vào R0. Việc này tạo ra mã ASCII của số cần xuất ra màn hình như đã trình bày. Lệnh ở x3010, TRAP, yêu cầu hệ điều hành xuất nội dung của R0 ra màn hình. Khi việc này hoàn thành xong và chương trình trở lại quá trình thực thi, lệnh ở x3011 thực hiện TRAP để kết thúc chương trình.

BÀI TẬP CUỐI CHƯƠNG

- 4.1 Trình bày vấn tắt sự giao tiếp giữa bộ nhớ và đơn vị xử lý. Nghĩa là, phương pháp để bộ nhớ và bộ xử lý liên lạc với nhau.
- 4.2 Hai thành phần của lệnh là gì? Hai thành phần chứa các thông tin gì?
- 4.3 Giả sử một lệnh 32 bit có dạng sau:

OPCODE	SR	DR	IMM
--------	----	----	-----

Nếu có 64 opcode và 32 thanh ghi. Hãy nêu tầm trị của vùng tin tức thời IMM. Chấp nhận vùng IMM chứa trị theo dạng bù 2.

- 4.4 Nếu từ nhớ của một bộ nhớ dài 64 bit. Hãy cho biết kích thước của MAR và MDR. Giải thích.
- 4.5 Có hai cách để kết thúc vòng lặp. Một cách là dùng một bộ đếm để theo dõi số lần lặp. Cách kia là dùng một phần tử gọi là(?) Nếu tính chất của phần tử này để kết thúc lặp.
- 4.6 Lệnh NOP (NO OPERATION) được nhiều ISA thiết kế để bộ xử lý không làm gì cả. Lệnh này được lấy, giải mã, và thực thi với pha thực thi lệnh là không thực hiện thao tác nào cả. Lệnh nào trong ba lệnh sau đây có thể được dùng cho NOP và làm cho chương trình vẫn chạy đúng? Giải thích.
- 0001 001 001 1 00000
 - 0000 111 000000001
 - 0000 000 000000000

- 4.7 Cho hai lệnh LC-3 A và B như sau:

A: 0000111101010101

B: 0100111101010101

Cho biết sự khác nhau giữa hai lệnh này. Chúng giống nhau như thế nào? Chúng khác nhau ra sao?

4.8 Chúng ta muốn có một lệnh LC-3 để trừ số thập phân 20 từ thanh ghi 1 và đặt kết quả vào thanh ghi 2. Chúng ta có thể làm được điều này không? Nếu được, viết ra lệnh này. Nếu không được, giải thích lý do.

4.9 Hoàn thành dây bốn lệnh sau với hai lệnh chưa biết để thực hiện thoa tác OR giữa hai thanh ghi 1 và thanh ghi 2, đặt kết quả vào thanh ghi 3.

(1): 1001 100 001 111111

(2): ...

(3): 0101 110 100 000 101

(4): ...

4.10 Trong LC-3, thanh ghi IR (*Instruction Register*) dài 16 bit, trong đó 9 bit trọng số thấp [8:0] biểu diễn thành phần PC-relative offset cho lệnh LD. Nếu chúng ta thay đổi ISA để chỉ có 7 bit cho vùng PC-relative offset, cho biết tầm địa chỉ mới mà chúng ta có thể nạp dữ liệu khi dùng lệnh LD mới này.

4.11 Thanh ghi PC chứa x3010. Các ô nhớ sau chứa các giá trị tương ứng như sau:

x3050: x70A4

x70A2: x70A3

x70A3:xFFFF

x70A4: x123B

Hãy cho biết trị trong thanh ghi R6 sau khi ba lệnh LC-3 sau đây được thực thi:

x3010 1110 0110 0011 1111

x3011 0110 1000 1000 0000

x3012 0110 1101 0000 0000

Chúng ta có thể thay thế chuỗi ba lệnh trên bằng một lệnh đơn hay không? Nếu được, đó là lệnh gì?

- 4.12 Viết một chương trình LC-3 so sánh hai số trong R2 và R3, đặt số lớn hơn vào R1. Nếu hai số bằng nhau, R1 được đặt trị 0.

4.13 Hình sau trình bày thông tin trong tám thanh ghi của LC-3 trước và sau khi lệnh ở vị trí x1000 được thực thi. Điền vào các bit của lệnh ở vị trí x1000.

- 4.14** Cho biết số chương trình dịch vụ TRAP lớn nhất mà LC-3 có thể cung cấp. Giải thích chi tiết.

Chương 5

LẬP TRÌNH HỢP NGỮ LC-3

5.1 LẬP TRÌNH HỢP NGỮ

Việc lập trình bằng ngôn ngữ máy với các dãy bit 1 và 0 là khá mệt mỏi và dễ sai sót. Có cách nào tham khảo một ô nhớ mà không cần sử dụng địa chỉ 16 bit của nó, mà chỉ cần dùng tên thay thế không? Chương này sẽ giúp ta giải quyết vấn đề này.

Khi lập trình chúng ta sử dụng một ngôn ngữ máy tính. Tổng quát, có hai cấp ngôn ngữ máy tính mà chúng ta đã biết, đó là ngôn ngữ cấp cao và cấp thấp. Ngôn ngữ cấp cao như C, C++, Java, Fortran, COBOL, Pascal, ... dùng các mệnh đề gần với ngôn ngữ tự nhiên như tiếng Anh. Chúng có khuynh hướng độc lập với kiến trúc tập lệnh (ISA). Có nghĩa là, một khi đã biết cách viết chương trình C (hoặc Fortran hay Pascal) cho một kiến trúc tập lệnh (ISA) nào đó, thì thật dễ để chúng ta viết chương trình C (hoặc Fortran hay Pascal) cho một ISA khác vì sự khác biệt là rất ít.

Trước khi một chương trình bằng ngôn ngữ cấp cao nào đó được thực thi, nó phải được dịch thành chương trình ở cấp kiến trúc tập lệnh của máy tính đang chạy chương trình đó. Thông thường mỗi lệnh trong ngôn ngữ cấp cao xác định một số lệnh trong kiến trúc tập lệnh của máy tính đó. Từ chương 7, chúng ta bắt đầu học ngôn ngữ lập trình C. Chúng ta sẽ thấy có mối liên hệ giữa các mệnh đề khác nhau trong C với các lệnh đã được dịch là mã LC-3.

Một bước nâng cấp nhỏ cho ISA của một máy tính là hợp ngữ của ISA đó. Hợp ngữ là một ngôn ngữ cấp thấp. Mỗi lệnh hợp ngữ thường xác định một lệnh đơn trong ISA. Không như ngôn ngữ cấp cao, ngôn ngữ cấp thấp phụ thuộc rất nhiều vào ISA. Thực tế, ta sẽ thấy là mỗi kiến trúc tập lệnh ISA chỉ có duy nhất một hợp ngữ.

Mục đích của hợp ngữ là làm cho quá trình lập trình bằng ngôn ngữ máy được thân thiện hơn với con người, trong khi vẫn cung cấp cho lập trình viên các lệnh cần thiết để viết chương trình mà máy tính có thể thực thi. Một cách cụ thể, con người không cần phải nhớ chi tiết mã thao tác 0001 là gì, rồi mã 1001 là thao tác gì, hay cái gì đang được chứa trong ô nhớ 0100000111100001. Các hợp ngữ quy định các từ gọi nhớ cho mã thao tác, như ADD và NOT, và chúng bắt chúng ta quy định các tên biến tượng có ý nghĩa cho các ô nhớ, như SUM hay PRODUCT, chứ không dùng các địa chỉ 16 bit. Điều này làm cho ta dễ dàng phân biệt giữa biến giữ tổng SUM và biến giữ tích PRODUCT khi sử dụng trong chương trình. Chúng ta gọi các tên này là các địa chỉ tượng trưng.

5.2 CÁC THÀNH PHẦN CỦA MỘT CHƯƠNG TRÌNH HỢP NGỮ

Để hiểu rõ hợp ngữ LC-3, ta hãy xét chương trình ví dụ sau.

Ví dụ 5.1 Chương trình nhân số nguyên với hằng số 6

```

01      ;
02      ; Chương trình nhân một số với 6
03      ;
04          .ORIG     x3050
05          LD        R1, SIX
06          LD        R2, NUMBER
07          AND       R3, R3, #0      ; Xóa R3 để giữ tích
08                      ; qua việc tính tổng cộng dần
09      ; Vòng lặp
0A      ;
0B      AGAIN    ADD     R3, R3, R2
0C          ADD     R1, R1, #-1   ; biến theo dõi số lần lặp
0D          BRp    AGAIN    ; lặp lại
0E      ;
0F          HALT
10      ;
11      NUMBER  .BLKW    1
12      SIX     .FILL x0006
13      ;
14          END

```

Chương trình này nhân số nguyên được khởi tạo trong biến NUMBER với 6 bằng việc cộng số nguyên đó 6 lần. Ví dụ, nếu số nguyên đó là 123, chương trình sẽ tính tích bằng việc cộng $123 + 123 + 123 + 123 + 123 + 123$.

Chương trình gồm 21 dòng mã. Chúng ta thêm số hiệu dòng trước mỗi dòng của chương trình để có thể tham khảo tới mỗi dòng một cách dễ dàng. Các số hiệu dòng này không phải là phần của chương trình. Chín dòng bắt đầu bằng dấu chấm phẩy (;) cung cấp các ghi chú thuận lợi cho người đọc. Bảy dòng (05, 06, 07, 0B, 0C, 0D, và 0F) xác định các lệnh hợp ngữ được dịch sang các lệnh ngôn ngữ máy của LC-3, mà thực sự được thực thi khi chương trình chạy. Bốn lệnh còn lại (04, 11, 12, và 14) chứa các mã giả, là các thông điệp từ lập trình viên tới chương trình dịch để giúp quá trình dịch. Chương trình dịch được gọi là một bộ hợp dịch (*assembler*), trong trường hợp này là assembler LC-3, và quá trình dịch được gọi là hợp dịch.

5.2.1 Lệnh

Thay vì dùng dãy 16 bit 0 và 1 để biểu diễn một lệnh như trong trường hợp ISA LC-3, một lệnh hợp ngữ bao gồm bốn phần theo cấu trúc sau:

LABEL OPCODE OPERANDS ; COMMENTS

Hai phần LABEL và COMMENTS là tùy chọn. Còn OPCODE và OPERANDS là bắt buộc.

1. OpCodes và Operands

Hai phần này phải có trong lệnh. Một lệnh phải quy định một mã thao tác OPCODE, tức là cái mà lệnh cần phải làm, và giá trị thích hợp của toán hạng OPERANDS, tức là cái mà lệnh sẽ dùng với tác vụ đã có. Đây là những thứ mà chúng ta đã gặp khi học LC-3.

OPCODE là tên tượng trưng cho mã tác vụ của lệnh LC-3 tương ứng. Với tên tượng trưng này, lập trình viên dễ dàng nhớ thao tác qua các tên như ADD, AND, hay LDR hơn là 4 bit 0001, 0101, hay 0110. Hình 4.3 liệt kê toàn bộ các OPCODES của 15 lệnh LC-3.

Số lượng các toán hạng phụ thuộc vào thao tác được thực thi. Ví dụ, lệnh ADD ở dòng 0B trong chương trình trên

AGAIN ADD R3, R3, R2

đòi hỏi ba toán hạng, gồm hai toán hạng nguồn chứa các số cần cộng (hai thanh ghi R3 và R2) và một toán hạng đích (thanh ghi R3) để lưu kết quả. Tất cả ba toán hạng phải được chỉ định tường minh trong lệnh.

Lệnh LD ở dòng 06

LD R2, NUMBER

đòi hỏi hai toán hạng, gồm một vị trí ô nhớ chứa trị cần đọc và thanh ghi đích chứa trị sau khi lệnh hoàn thành xong. Chúng ta thấy là các ô nhớ được cho bởi các địa chỉ tượng trưng được gọi là các nhãn (*labels*). Trong trường hợp này, vị trí từ đó giá trị được đọc là nhãn NUMBER. Đích để ghi thông tin vào là thanh ghi R2.

Như đã được trình bày trong chương 4, các toán hạng có thể từ các thanh ghi, từ bộ nhớ, hay chúng có thể là các giá trị tức thời trong lệnh. Trong trường hợp toán hạng là thanh ghi, thanh ghi sẽ được biểu diễn tường minh (như R2 và R3 trong dòng 0B). Trong trường hợp toán hạng là bộ nhớ, tên tượng trưng cho ô nhớ cũng được nêu rõ ràng (như NUMBER trong dòng 06 và SIX trong dòng 05). Trong trường hợp toán hạng tức thời, các giá trị thực cần được ghi rõ trong lệnh (như trị 0 trong dòng 07).

AND R3, R3, #0 ; xóa R3 để giữ tích

Giá trị tức thời sử dụng trong lệnh phải chứa một ký hiệu xác định cơ số biểu diễn số đó. Chúng ta dùng dấu # cho số thập phân, x cho thập lục phân, và b cho nhị phân. Nói chung chúng ta luôn nên dùng các ký hiệu quy ước này để viết các hằng trong hệ thống số mong muốn. Nếu không có sự nhầm lẫn nào khác, chúng ta có thể viết và hiểu ngay cơ số sử dụng, như 3F0A trong hệ hex.

2. Nhãn

Nhãn là các tên tượng trưng được dùng để xác định các ô nhớ được tham khảo tới trong chương trình. Trong hợp ngữ LC-3, một nhãn có thể được tạo từ một tới 20 ký số hay ký tự, và bắt đầu bằng một ký tự, như LAPLAI, KETTHUC, LAP100,....

Có hai lý do cần cho việc tham khảo một vị trí trong bộ nhớ, đó là

- Ô nhớ vị trí đó chứa đích của một lệnh rẽ nhánh, ví dụ AGAIN trong dòng 0B.
- Ô nhớ vị trí đó chứa một giá trị cần được nạp hay lưu, ví dụ, NUMBER ở dòng 11, và SIX ở dòng 12.

Vị trí tương ứng nhãn AGAIN được tham khảo bởi lệnh rẽ nhánh ở dòng 0E,

BRp AGAIN

Nếu kết quả của lệnh ADD R1, R1, #-1 là dương ($R1 > 0$) do mã điều kiện P được dựng, thì chương trình rẽ nhánh tới vị trí được nêu AGAIN để thực hiện chu trình lặp khác.

Vị trí NUMBER được tham khảo bởi lệnh load ở dòng 06. Giá trị được lưu trong vị trí nhớ NUMBER được nạp vào R2.

Nếu một vị trí trong chương trình không cần được tham khảo, thì ta không cần cho nó một nhãn.

3. Ghi chú

Ghi chú là các thông điệp chỉ cần thiết với con người. Các ghi chú không có bất kỳ ảnh hưởng nào trong quá trình dịch và cũng không chịu tác động nào từ bộ dịch hợp ngữ LC-3. Chúng được quy định trong chương trình bằng các dấu chấm phẩy đặt trước, phần sau dấu chấm phẩy (nếu có) là một ghi chú và được bộ dịch bỏ qua.

Mục đích của ghi chú là làm chương trình dễ hiểu hơn cho người đọc. Chúng giúp giải thích mục đích sử dụng lệnh hay nhóm lệnh một cách rõ ràng. Trong dòng 07 và 08, ghi chú "Xóa R3 để giữ tích qua việc tính tổng cộng dồn" làm cho người đọc biết được lệnh ở dòng 07 là để khởi động R3 trước khi cộng tích lũy để có tích từ hai số. Vì trong thực tế, ta có thể thấy việc viết dòng lệnh này và hiểu mục đích thao tác của nó là quá dễ dàng cho ngày hôm nay, nhưng sáu tháng sau, khi người lập trình đã viết thêm 10 000 dòng lệnh nữa, thì họ không thể nhớ lệnh AND R3, R3, #0 là để làm gì, và tại sao, vì thông thường muốn tính tích người ta thường khởi động trị 1 cho biến giữ tích, chứ không phải 0. Hoặc có thể gấp trường hợp hai năm sau, lập trình viên viết chương trình này đã nghĩ làm việc cho công ty, và công ty hiện

nay muốn nâng cấp, chỉnh sửa chương trình, và lập trình viên mới khi đọc chương trình này có thể không hiểu nếu không nhờ các ghi chú.

Cũng cần phải lưu ý là việc viết các ghi chú là để cung cấp thêm thông tin nhìn rõ hơn vấn đề, chứ không phải là nói lại những cái hiển nhiên, vì việc nói lại này là, mất thời gian cho người đọc, mà không cung cấp thêm thông tin có ích. Hơn nữa, các ghi chú loại này còn làm mờ đi ý nghĩa của các ghi chú quan trọng mà người đọc cần nắm bắt để hiểu rõ chương trình. Ví dụ, nếu dòng 0C có ghi chú là “Giảm trị thanh ghi R1” thì thật là không cần thiết, vì nó không cung cấp thêm thông tin nào, mà còn rối khi đọc.

Mục đích khác của ghi chú, với cách dùng khôn ngoan các dòng ghi chú trắng, làm cho việc thể hiện chương trình dễ hiểu hơn. Ví dụ, các ghi chú chỉ với các dấu chấm phẩy được dùng để phân cách giữa các phần có mục đích khác nhau của chương trình. Ví dụ, các dòng từ 0B tới 0D được cách biệt với phần mã còn lại của chương trình bằng các dòng ghi chú 0A và 0E. Không có bất cứ thứ gì khác ngoài các dấu chấm phẩy.

5.2.2 Mã giả (Các hướng dẫn dịch)

Bộ hợp dịch LC-3 là một chương trình lấy đầu vào là chuỗi ký tự biểu diễn một chương trình được viết bằng hợp ngữ LC-3, và dịch nó ra thành một chương trình ở cấp kiến trúc tập lệnh (ISA) của LC-3. Mã giả (*pseudo-ops*) giúp cho bộ dịch thực hiện nhiệm vụ này.

Trong thực tế, mã giả còn được gọi bằng một tên khác là hướng dẫn dịch (*assembler directives*). Sở dĩ gọi là các mã giả vì chúng không quy định cho một thao tác nào cần được chương trình thực thi. Mã giả chính là một thông điệp cho bộ hợp dịch để giúp nó trong quá trình hợp dịch chương trình. Một khi bộ hợp dịch xử lý thông điệp xong, mã giả sẽ được bỏ qua. Bộ hợp dịch LC-3 gồm năm mã giả: .ORG, .FILL, .BLKW, .STRINGZ, và .END. Tất cả mã giả này đều có dấu chấm như là ký tự đầu tiên của nó.

.ORIG

.ORIG cho bộ dịch biết nơi bắt đầu chương trình LC-3 trong bộ nhớ. Ở dòng 04, .ORIG x3050 nói rằng, chương trình bắt đầu ở vị trí x3050. Và tất nhiên, lệnh LD R1, SIX sẽ được đặt ở vị trí x3050.

.FILL

.FILL nói cho bộ hợp dịch biết việc cần dùng vị trí kế trong chương trình (và tất nhiên là sau này là bộ nhớ khi chạy chương trình), và khởi động nó bằng giá trị của toán hạng. Ở dòng 12, vị trí thứ 9 (tính từ lệnh đầu tiên) trong chương trình LC-3 được khởi động trị x0006.

.BLKW

.BLKW bắt bộ dịch để dành một số ô nhớ (tức BLocK Words) trong chương trình. Số ô nhớ thực sự là toán hạng của mã giả .BLKW. Ở dòng 11, mã giả yêu cầu bộ dịch để dành một ô nhớ với nhẫn là NUMBER.

Mã giả .BLKW đặc biệt hữu ích khi chúng ta chưa biết giá trị của toán hạng. Ví dụ, ta có thể khai báo để dành một ô nhớ để lưu ký tự được nhập vào từ bàn phím mà đến lúc chạy chương trình ta mới biết ký tự là gì sau khi phím được gõ.

.STRINGZ

.STRINGZ bắt bộ dịch khởi tạo một chuỗi $n + 1$ ô nhớ. Đôi số là dãy n ký tự, bên trong cặp dấu nháy kép. Khi đó, n từ nhớ đầu tiên được khởi động bằng các ký tự mã ASCII 8 bit được mở rộng zero (để có 16 bit) trong chuỗi. Từ nhớ cuối cùng được khởi tạo là 0, tức x0000, là trị canh để truy xuất chuỗi các mã ASCII.

Ví dụ 5.2 Đoạn mã sau:

```
    .ORIG      x3010
    HELLO     .STRINGZ   "Hello, World!"
```

tạo ra sự khởi động hợp dịch trong các ô nhớ từ x3010 tới x301D như sau

x3010: x0048

x3011: x0065

x3012: x006C

x3013: x006C

x3014: x006F

x3015: x002C

```
x3016: x0020
x3017: x0057
x3018: x006F
x3019: x0072
x301A: x006C
x301B: x0064
x301C: x0021
x301D: x0000
```

.END

.END nói cho bộ dịch biết chương trình kết thúc ở đâu. Bất kỳ ký tự nào đứng sau .END sẽ bị bộ hợp dịch bỏ qua. Như vậy, thực ra .END chỉ đơn giản là một quy định giới hạn, nó đánh dấu sự kết thúc của chương trình nguồn.

5.2.3 Một ví dụ

Trong mục này, chúng ta xét lại ví dụ ở mục 4.10, tính số lần xuất hiện của một ký tự trong một file cho trước. Ký tự cần kiểm tra được vào từ bàn phím, file ký tự được xem là mảng ký tự cần được khởi tạo trước khi chạy chương trình. Giải thuật ở dạng lưu đồ và chương trình ở dạng ISA LC-3 được trình bày trong hình 4.18 và 4.19.

Bây giờ, thay vì viết chương trình ở ISA LC-3 dưới dạng các chuỗi nhị phân 1 và 0, nay chúng ta viết lại nó dưới dạng hợp ngữ LC-3. Chương trình được cho trong ví dụ 5.3 dưới đây.

Ví dụ 5.3 Chương trình đếm số lần xuất hiện ký tự trong file ở dạng hợp ngữ LC-3.

```
01 ;
02 ; Chương trình đếm số lần xuất hiện ký tự trong file.
03 ; Ký tự cần kiểm tra được nhập từ bàn phím.
04 ; Kết quả được hiển thị ra màn hình.
05 ; Chương trình chỉ làm việc đúng khi số lần xuất hiện ký tự không quá 9.
06 ;
07 ;
```

```

08 ; Khởi động
09 ;
0A .ORIG x3000
0B AND R2, R2, #0 ; R2 là bộ đếm, được khởi động bằng 0
0C LD R3, PTR ; R3 là pointer tới các ký tự trong file
0D TRAP x23 ; R0 lưu ký tự được nhập vào
0E LDR R1, R3, #0 ; R1 giữ ký tự đầu tiên
0F ;
10 ; Kiểm tra ký tự kết thúc file EOT
11 ;
12 TEST ADD R4, R1, #-4 ; Kiểm tra EOT (ASCII x04)
13 BRz OUTPUT ; Nếu đúng, chuẩn bị xuất
14 ;
15 ; Kiểm tra sự thích hợp của ký tự. Nếu có, tăng biến đếm.
16 ;          cuu duong than cong . com
17 NOT R1, R1
18 ADD R1, R1, R0 ; Nếu đúng là ký tự cần kiểm tra, R1 = xFFFF
19 NOT R1, R1 ; Nếu đúng là ký tự cần kiểm tra, R1 = x0000
1A BRnp GETCHAR ; Nếu không đúng, không tăng đếm, lấy ký tự kế
1B ADD R2, R2, #1
1C ;
1D ; Lấy ký tự kế tiếp trong file.
1E ;
1F GETCHAR ADD R3, R3, #1 ; Chỉ tới ký tự kế.
20 LDR R1, R3, #0 ; R1 lấy ký tự kế để kiểm tra
21 BRnzp TEST
22 ;
23 ; Xuất số lần xuất hiện.
24 ;
25 OUTPUT LD R0, ASCII ; Nạp mẫu ASCII
26 ADD R0, R0, R2 ; Chuyển số trị từ nhị phân ra ký tự ASCII
27 TRAP x21 ; Mã ASCII trong R0 được hiển thị.

```

```

28          TRAP x25      ; Chấm dứt chương trình.
29  ;
2A  ; Phần lưu trữ pointer và mẫu ASCII
2B  ;
2C  ASCII    .FILL  x0030
2D  PTR      .FILL  x4000
2E      .END

```

Trong chương trình trên, có ba lần chương trình yêu cầu gọi dịch vụ từ hệ điều hành. Đó là lệnh TRAP x23 nhập một ký tự từ bàn phím và đặt vào R0 (dòng 0D), lệnh TRAP x21 để hiển thị mã ASCII đang có trong thanh ghi R0 ra màn hình (dòng 27), và lệnh TRAP x25 để kết thúc chương trình (dòng 28).

Chúng ta cũng cần lưu ý là các ký số thập phân từ 0 tới 9 (0000 tới 1001 nhị phân) là từ x30 tới x39. Việc đổi từ dạng nhị phân thành mã ASCII chỉ đơn giản là cộng x30 vào trị nhị phân đang biểu diễn ký số thập phân. Dòng 2C cho nhãn ASCII xác định vị trí ô nhớ chứa x0030.

File là mảng ký tự cần được khởi tạo ở địa chỉ x4000 (dòng 2D). Thường địa chỉ bắt đầu này không được lập trình viên biết trước vì chúng ta muốn chương trình xử lý trên các file dữ liệu ở vị trí chúng ta muốn tại thời điểm chạy chương trình. Chúng ta sẽ bàn rõ hơn vấn đề này sau.

5.3 QUÁ TRÌNH HỢP DỊCH

5.3.1 Giới thiệu

Trước khi một chương trình hợp ngữ LC-3 được thực thi, nó phải được dịch ra thành một chương trình ngôn ngữ máy, có nghĩa là từng lệnh trong đó sẽ là từng lệnh ở ISA LC-3. Đây là công việc của bộ dịch hợp ngữ LC-3.

Với bộ hợp dịch LC-3 (mà chúng ta có thể download từ mạng), ta có thể dịch từ chương trình hợp ngữ ra chương trình ngôn ngữ máy. Trong giáo trình này, các chương trình hợp ngữ có thể được viết và được dịch ra dạng ISA bằng LC-3 Simulator mà chúng ta có thể tìm thấy trên mạng.

5.3.2 Quá trình dịch

Bộ hợp dịch chuyển một chương trình hợp ngữ thành chương trình ngôn ngữ máy qua quá trình dịch hai bước. Để dễ hình dung, chúng ta hãy quan sát lại chương trình ở ví dụ 5.3 trên.

Tổng quát, có một sự tương ứng một-một giữa các lệnh trong chương trình hợp ngữ và các lệnh trong chương trình sau cùng bằng ngôn ngữ máy. Để hiểu rõ hơn về quá trình hợp dịch, chúng ta hãy thử thực hiện dịch chương trình hợp ngữ đã nêu ở ví dụ 5.3 chỉ với một bước dịch. Bắt đầu từ đầu chương trình, bộ hợp dịch bỏ qua các dòng từ 01 tới 09, vì chúng chỉ chứa các ghi chú. Kế tiếp, bộ dịch tới dòng 0A, đây là mã giả, nó nói cho bộ dịch biết là chương trình ngôn ngữ máy bắt đầu ở địa chỉ x3000. Tới dòng 0B, bộ dịch dễ dàng dịch lệnh này ra mã máy LC-3. Lúc này, ta có

x3000: 0101010010100000

Bộ dịch LC-3 tiếp tục chuyển qua lệnh kế tiếp ở dòng 0C. Và thật không may là nó không thể dịch được, vì nó không biết ý nghĩa của địa chỉ tượng trưng PTR. Lúc này, bộ dịch bị mắc kẹt, và quá trình dịch bị sai.

Để tránh điều này xảy ra, quá trình hợp dịch được thực hiện qua hai bước (từ đầu chương trình tới .END) qua toàn bộ chương trình hợp ngữ. Mục tiêu của bước đầu tiên là xác định các địa chỉ nhị phân thực sự tinh từ đầu chương trình tương ứng với các tên tượng trưng (nhãn). Tập hợp của các tương ứng này được gọi là bảng biểu trưng. Trong bước 1, chúng ta tạo bảng biểu trưng. Trong bước 2, chúng ta dịch các lệnh hợp ngữ riêng lẻ thành các lệnh ngôn ngữ máy tương ứng.

Như vậy, khi bộ dịch khảo sát dòng 0C

LD R3, PTR

để dịch trong bước thứ hai, nó đã biết sự tương ứng giữa PTR và x3013 (từ bước thứ nhất). Như vậy, nó có thể dịch dòng 0C thành

x3001: 0010011000010001

Chúng hãy thử xem tại sao như vậy. Trong lệnh ở x3001, ta có lệnh LD với opcode là 0010, kế tiếp thanh ghi R3 với mã là 011 xác định toán hạng đích nhận dữ liệu. Nếu lệnh ở x3001 là lệnh đang được thực thi, qua phase FETCH của nó, PC là x3002. Khi đó độ lệch

nhi phân tính từ PC tới nhãn PTR là 17 thập phân (10001 nhị phân), tức qua 17 lệnh, nên phần PCoffset9 là 000010001.

Hai mục dưới đây sẽ giúp ta hiểu sâu hơn về hai bước dịch trong quá trình hợp dịch này.

5.3.3 Bước đầu tiên: Tạo bảng biểu trưng

Bảng biểu trưng là một sự tương ứng giữa các tên tượng trưng với các địa chỉ 16 bit của chúng tính từ đầu chương trình. Nên nhớ rằng, chúng ta cần các nhãn ở những chỗ cần được tham khảo, hoặc đó là đích của một lệnh rẽ nhánh hoặc nơi đó chứa dữ liệu cần được nạp hay lưu. Vì vậy, nếu chúng ta không có bất kỳ một lối lập trình nào, và nếu chúng ta xác định được tất cả các nhãn, chúng ta hẳn sẽ xác định được tất cả các địa chỉ tượng trưng được dùng trong chương trình.

Trong bước dịch đầu tiên, sau khi bỏ qua các dòng ghi chú từ 01 tới 09, bộ dịch qua dòng 0A xác định dòng lệnh đầu tiên ở địa chỉ x3000. Chúng ta hãy theo dõi vị trí được gán cho mỗi lệnh bằng một bộ đếm vị trí LC (*Location counter*). LC được khởi động bằng địa chỉ được xác định sau .ORIG, nghĩa là x3000 với chương trình ví dụ 5.3.

Bộ hợp dịch khảo sát từng lệnh trong dãy, và tăng LC khi gặp một lệnh hợp ngữ. Nếu lệnh được khảo sát chứa một nhãn, một đầu vào (*entry*) trong bảng biểu trưng được tạo ra cho nhãn đó, xác định các nội dung hiện thời của LC là địa chỉ của nhãn này.

Lệnh đầu tiên có nhãn là lệnh ở dòng 12. Vì nó là lệnh thứ năm của chương trình, nên lúc này LC chứa x3004, một đầu vào trong bảng biểu trưng được tạo ra như sau:

Symbol	Address
TEST	x3004

Lệnh thứ hai có nhãn là lệnh ở dòng 1F. Tại đây, LC đã được tăng lên tới x300B. Như vậy một đầu vào trong bảng đã được tạo ra thêm như sau:

Symbol	Address
--------	---------

GETCHAR	x300B
---------	-------

Tới lúc cuối của bước dịch đầu tiên, bảng biểu trưng có các đầu vào như sau:

Symbol	Address
--------	---------

TEST	x3004
GETCHAR	x300B
OUTPUT	x300E
ASCII	x3012
PTR	x3013

5.3.4 Bước thứ hai: tạo chương trình ngôn ngữ máy

Bước dịch thứ hai gồm việc duyệt qua chương trình hợp ngữ lần thứ hai, theo từng dòng, lúc này với sự trợ giúp của bảng biểu trưng. Ở mỗi dòng, lệnh hợp ngữ được dịch ra lệnh ngôn ngữ máy LC-3.

Bắt đầu trở lại đầu chương trình, bộ hợp dịch bỏ qua các dòng từ 01 tới 09 vì chúng chỉ chứa các ghi chú. Dòng 0A là mã giả .ORIG, được bộ dịch dùng để khởi động LC bằng x3000. Bộ dịch chuyển tới dòng 0B và tạo ra lệnh ngôn ngữ máy 0101010010100000. Và rồi bộ dịch chuyển tới dòng 0C.

Lần này, khi bộ dịch lấy lệnh ở dòng 0C, nó có thể hoàn toàn dịch lệnh này vì nó biết nhãn PTR tương ứng với x3013. Lệnh là LD, có opcode là 0010. Thanh ghi đích là R3, nghĩa là 011.

PCoffset được tính như sau: chúng ta biết rằng PTR là nhãn cho địa chỉ x3013, và thanh ghi PC đã tăng là LC + 1, tức x3002. Vì PTR (x3013) phải là tổng của PC đã tăng (x3002) và PCoffset được mở rộng dấu, nên PCoffset phải là x0011. Ghép tất cả điều này lại với nhau, ta thấy lệnh ở x3001 là 0010011000010001, và LC được tăng lên x3002.

Chú ý rằng, để dùng lệnh LD, nhãn là địa chỉ của toán hạng nguồn không được vượt quá +256 hay -255 vị trí ô nhớ tính từ bản thân lệnh LD. Nếu địa chỉ của PTR mà lớn hơn LC+1+255 hay nhỏ hơn LC+1-256 thì offset sẽ không đặt được vào chín bit [8:0] của lệnh LD. Trong trường hợp này, một lỗi hợp dịch xuất hiện, ngăn cản quá trình dịch thành công. Thật may là trong chương trình 5.3, PTR không bị vấn đề này, nên lệnh được dịch đúng.

Bước thứ hai tiếp tục. Ở mỗi bước, LC được tăng lên và vị trí được LC xác định được gán thành một lệnh LC-3, hay trong trường hợp .FILL, là giá trị được quy định ngay sau đó. Khi bước dịch thứ hai này gặp .END, sự hợp dịch kết thúc.

Chương trình đã được dịch và nhận được là

Address	Binary
	0011000000000000
x3000	0101010010100000
x3001	0010011000010001
x3002	111100000100011
x3003	0110001011000000
x3004	0001100001111100
x3005	0000010000001000
x3006	1001001001111111
x3007	0001001001000000
x3008	1001001001111111
x3009	0000101000000001
x300A	0001010010100001
x300B	0001011011100001
x300C	0110001011000000
x300D	000011111110110
x300E	0010000000000011
x300F	0001000000000010
x3010	111100000100001
x3011	111100000100101
x3012	000000000110000
x3013	0100000000000000

Chương trình ở dạng mã máy này rất khó đọc và chỉnh sửa. Tuy nhiên, chúng ta sẽ dùng dạng hợp ngữ LC-3 với các mã giả và các lệnh để viết, và sau đó dùng bộ dịch hợp ngữ để dịch ra dạng mã máy này cho máy tính LC-3 thực thi.

5.4 CHƯƠNG TRÌNH VỚI NHIỀU MODUL

5.4.1 Bản thực thi

Khi máy tính bắt đầu thực thi một chương trình, tập tin thực thi của chương trình được gọi là bản thực thi (*Executable image*). Bản thực thi thường được tạo ra từ nhiều modul do nhiều lập trình viên thiết kế ra một cách độc lập. Mỗi modul được dịch một cách riêng biệt và tạo thành một tập tin đối tượng (*object*). Nếu các modul được viết bằng hợp ngữ LC-3, chúng sẽ được dịch bằng bộ dịch hợp ngữ LC-3. Những modul được viết bằng C sẽ được dịch bằng bộ dịch C. Có những modul do lập trình viên viết khi thiết kế chương trình, và cũng có những modul là các chương trình con được cung cấp bởi hệ điều hành. Mỗi tập tin đối tượng bao gồm các lệnh trong kiến trúc tập lệnh (ISA) của máy tính đang được sử dụng, cùng với các dữ liệu liên quan.

Bước cuối cùng là liên kết (*link*) tất cả các modul lại với nhau để có một tập tin gọi là bản thực thi. Trong suốt quá trình thực thi, các chu kỳ lệnh FETCH, DECODE, ... được áp dụng cho các lệnh trong bản thực thi.

5.4.2 Thiết kế với nhiều tập tin đối tượng

Khi thiết kế một chương trình, chúng ta thường dùng thư viện của hệ điều hành cũng như các modul được viết bởi các lập trình viên khác trong nhóm. Do đó, việc bản thực thi được tạo ra từ nhiều tập tin đối tượng khác nhau là rất phổ biến.

Trong chương trình ví dụ đếm số ký tự xuất hiện trong một tập tin là mảng, ta có thể thấy một áp dụng tiêu biểu của chương trình với hai modul, gồm modul chương trình và modul là tập tin dữ liệu. Với ví dụ 5.3, địa chỉ bắt đầu của tập tin mảng dữ liệu là x4000 ở dòng 2D không được quan tâm khi chương trình được viết. Nếu chúng ta thay thế dòng 2D này bằng

PTR .FILL STARTofFILE

thì chương trình ví dụ này sẽ không được hợp dịch vì không có đầu vào cho STARTofFILE trong bảng biểu trưng. Chúng ta giải quyết việc này ra sao?

Mặt khác, nếu hợp ngữ LC-3 có mã giả .EXTERNAL, chúng ta có thể xác định STARofFILE như là một tên biểu trưng của một địa chỉ không được biết lúc chương trình 5.3 được dịch.

Điều này có thể được thực hiện bằng dòng sau

.EXTERNAL STARTofFILE

mà sẽ báo cho bộ dịch LC-3 rằng sự vắng mặt của nhãn STARTofFILE không phải là một lỗi trong chương trình. Hơn nữa, STARTofFILE là một nhãn trong modul khác và modul này sẽ được dịch một cách độc lập. Trong ví dụ 5.3, đó chính là nhãn của vị trí của ký tự đầu tiên trong tập tin mảng mà sẽ được chương trình đếm ký tự của chúng ta khảo sát.

Nếu hợp ngữ LC-3 có được mã giả .EXTERNAL, và nếu chúng ta đã thiết kế nhãn STARTofFILE theo .EXTERNAL, LC-3 có khả năng tạo một đầu vào trong bảng biểu trưng cho STARTofFILE, và thay vì gán cho nhãn này một địa chỉ, LC-3 sẽ đánh dấu biểu trưng tùy thuộc modul khác. Lúc liên kết, khi tất cả các modul được kết nối lại, bộ liên kết (tức chương trình phụ trách việc nối này) sẽ dùng đầu vào cho STARTofFILE trong bảng biểu trưng trong modul khác để hoàn tất việc dịch dòng 2D.

Theo cách này, mã giả .EXTERNAL cho phép việc tham khảo của một modul tới các vị trí biểu trưng trong một modul khác một cách dễ dàng. Quá trình dịch phù hợp được bộ liên kết giải quyết.

5.5 MỘT SỐ VÍ DỤ

5.5.1 Ví dụ 1

Viết chương trình nhập một ký tự từ bàn phím. Kiểm tra ký tự này, nếu ký tự là thường thì đổi nó ra hoa, rồi in ra màn hình.

Chương trình:

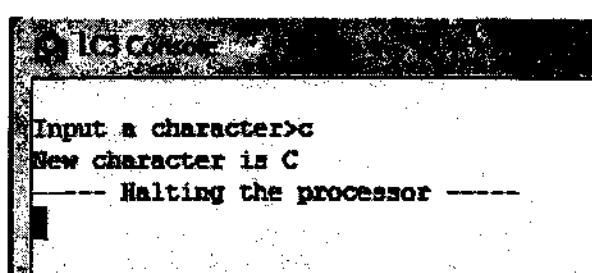
```
00 .ORIG x1000
01 IN          ; Nhập ký tự vào R0
02
```

```

03 LD R1, Nega_a ; Kiểm tra ký tự trong R0 < 'a'?
04 ADD R1, R0, R1
05 BRn ENDING ; Có, nhảy tới kết thúc
06 ; Không, kiểm tra tiếp
07 LD R1, Nega_z ; Kiểm tra ký tự trong R0 > 'z'?
08 ADD R1, R0, R1
09 BRp ENDING ; Có, nhảy tới kết thúc
0A ; Không, ký tự trong R0 thuộc a-z
0B LD R1, Nega_a_A ; R1=-khoảng cách thường-hoa, -x20
0C ADD R1, R0, R1 ; R1: ký tự sau khi trừ x20, ký tự hoa
0D
0E LEA R0, P_out ; In ra P_out
0F PUTS
10
11 ADD R0, R1, #0 ; In ký tự mới
12 OUT
13
14 ENDING
15 HALT
16
17 P_out .STRINGZ "New character is "
18 Nega_a .FILL xFF9F ; '-a'
19 Nega_z .FILL xFF86 ; '-z'
1A Nega_a_A .FILL xFFE0 ; -x20
1B
1C .END

```

Xuất liệu ví dụ:



Chương trình trên cho phép ta nhập một ký tự từ bàn phím bằng mã Trap (IN) ở dòng 01. Lệnh này mặc nhiên xuất ra câu “**Input a character>**” dù ta không muốn, ký tự nhập được sẽ được hiển thị ra màn hình. Khác với lệnh IN, lệnh GETC không in ra câu nhắc nhở nào cả, và cũng không hiển thị ký tự vừa gõ ra màn hình. Cả hai lệnh đều không cần đợi ENTER để xác nhận việc nhập. Các nhãn Nega_a, Nega_z, và Nega_a_A là địa chỉ của các ô nhớ chứa bù hai mã ASCII của các ký tự ‘a’, ‘z’ và khoảng cách giữa hai tập ký tự hoa-thường. Để xuất chuỗi ra màn hình, lệnh PUTS, tức lệnh Trap x22, được sử dụng sau khi thanh ghi R0 đã được nạp bằng địa chỉ của chuỗi cần in qua lệnh LEA. Độc giả có thể xem thêm các chú thích ngay sau mỗi lệnh để hiểu thêm chi tiết.

5.5.2 Ví dụ 2

Nhập hai số nguyên dương từ 0-9. Tính và in ra tổng của hai số đó.

Chương trình:

```

00 .ORIG x3000
01 LEA    R0, P_1      ; In ra P_1
02 PPUTS
03
04 IN          ; Nhập số thứ nhất, R0 = '8', ví dụ
05 LD    R1, N_ASCII ; R1 = -x30
06 ADD   R0, R0, R1 ; R0 = 8
07 ADD   R2, R0, #0 ; sao lưu R0 vào R2, R2 = 8
08
09 LEA    R0, P_2      ; In ra P_2
0A PPUTS
0B
0C IN          ; Nhập số thứ hai, R0 = '9', ví dụ
0D ADD   R0, R0, R1 ; R0 = 9; ví dụ, R1 = -x30
0E
0F ADD   R3, R0, R2 ; R3 = R0 + R2 = 17, ví dụ
10
11 LEA    R0, P_out    ; In ra P_out

```

```

12 PUTS
13
14 ADD R2, R3, #0 ; Nạp R2 từ R3
15 ADD R2, R2, #-10 ; So sánh xem tổng >= 10?
16
17 BRzp Greater_10 ; Có, R3 = 17, R2 = 7
18      )
19 Less_10 .           ; Không, R3 = 4: ví dụ
1A ADD R0, R3, #0 ; R0 = R3 = 4
1B LD   R1, ASCII    ; R1 = '0'
1C ADD R0, R0, R1    ; R0 = '0' + 4 ='4'
1D OUT .             ; In ra tổng khi có 1 chữ số hàng đơn vị
1E LEA  R1, Ending   ; Kết thúc
1F JMP  R1
20
21 Greater_10 .       ; cuu duong than cong . com
22 NOT R2, R2         ; Tính -R2
23 ADD R2, R2, #1     ; bằng bù hai 2, R2 = -7
24 ADD R4, R3, R2     ; R4 = R3 - R2 = 17 - 7 = 10
25 ADD R4, R4, #-9    ; R4 = 1
26
27 LD   R1, ASCII    ; R1 = '0'
28 ADD R0, R4, R1     ; R0 = '1'
29
2A OUT .             ; In ra chữ số hàng chục
2B .                   ; cuu duong than cong . com
2C ADD R3, R3, #-10  ; R3 = 17 - 10 = 7
2D
2E LEA  R1, Less_10   ; Tới chỗ in ra chữ số hàng đơn vị
2F JMP  R1
30
31 Ending
32 HALT
33 P_1 .STRINGZ "Input the first number: "

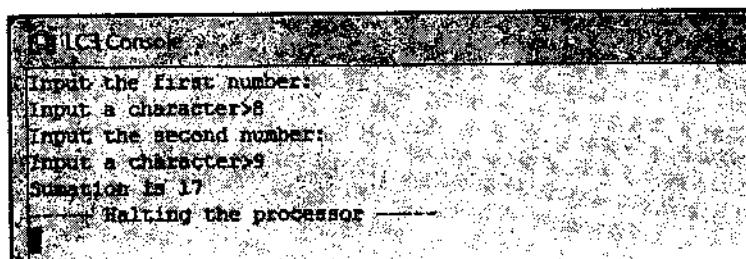
```

```

34 P_2 .STRINGZ "Input the second number: "
35 P_out .STRINGZ "Sumation is "
36
37 N_ASCII .FILL xFFD0
38 ASCII .fill x0030
39
3A .END

```

5.5.3 Xuất liệu ví dụ



Trong chương trình trên, lệnh PUTS ở dòng 02 in ra chuỗi “**Input the first number:**” ra màn hình, sau đó lệnh IN ở dòng 04 lại in ra thêm câu “Input a character” dù ta không mong muốn. Việc này sẽ được gấp lại ở dòng 0A và 0C mà ta phải chấp nhận do đây là thao tác của bộ dịch. Ký tự số được nhập vào sẽ được chuyển qua số ở các dòng 05, 06 và 0D bằng việc trừ đi cho x30, mà trong chương trình là tính tổng với -x30 ở nhãn N_ASCII. Để in ra các ký số, các số cần được chuyển qua các ký số bằng thao tác cộng thêm cho x30 ở nhãn ASCII ở các dòng 1B, 1C cho chữ số hàng chục, và 27, 28 cho chữ số hàng đơn vị. Chi tiết thao tác lệnh được trình bày bằng các ghi chú ngay sau từng lệnh.

Ví dụ 3

Viết chương trình tính tổng từ n số nguyên từ 0-9 nhập từ bàn phím, với n là một số nguyên dương được nhập từ bàn phím và có trị từ 1-9.

Chương trình:

```

00 .ORIG x1000
01 LEA R0, P_in ; In chuỗi P_in

```

```

02    PUTS
03    GETC          ; Nhập số lượng số nguyên, không
04                  ; hiện ký số ra màn hình, R0='5': VD
05    LD      R1, N_Ascii   ; R1 = -x30
06    LEA     R3, Array     ; R3 = pointer tới mảng Array
07    ADD     R2, R0, R1     ; .2 = số phần tử trong mảng, R2=5 : VD
08    ST      R2, Temp       ; Sao lưu R2
09
10
11
12
13
14
15 Tinh_tong LEA  R3, Array ; R3: địa chỉ mảng
16 AND    R0, R0, #0       ; Khởi động R0 để giữ tổng
17 LD     R2, Temp        ; Nạp lại R2: số phần tử trong mảng
18 Lap_tong LDR
           R1, R3, #0       ; Đọc từng phần tử
19 ADD    R0, R0, R1       ; Cộng vào R0
1A ADD    R3, R3, #1       ; Tăng pointer
1B ADD    R2, R2, #-1      ; Giảm số đếm
1C BRp   Lap_tong        ; Số đếm > 0, cc. lặp lại tính tổng
1D
1E
1F
20 AND   R2, R2, #0       ; Khởi động R2 để chứa số lần của 10

```

21 Lap_chuoi ADD

```

        R0, R0, #-10      ; Trừ R0 cho 10
22    BRn   In_ra          ; Nếu R0 < 0, in ra
23    ADD   R2, R2, #1      ; R0>=0, tăng R2 để tìm số hàng chục
24    LEA   R3, Lap_chuoi ; Nhảy tới Lap_chuoi
25    JMP   R3
26
27 In_ra
        LD    R3, Ascii     ; R3 = x30 = '0'
28    ADD   R1, R0, #10    ; Thoát khỏi vòng lặp,
29                      ; R1 = số ở hàng đơn vị
2A                      ; R2 = số ở hàng chục
2B    LEA   R0, P_out     ; In ra P_out
2C    PUTS
2D
2E    ADD   R1, R1, R3    ; Tạo ký số hàng đơn vị
2F    ADD   R0, R2, R3    ; Tạo ký số hàng chục
30    OUT
31    ADD   R0, R1, #0    ; Tạo ký số hàng đơn vị
32    OUT
33
34    HALT
35
36    Array .BLKW 9       ; Khai báo mảng có 9 phần tử
37    P_in  .STRINGZ "Nhập hai số nguyên từ 0-9: "
38    P_out .STRINGZ "Tổng là: "
39    N_Ascii .FILL xFFD0
3A    Ascii  .FILL x0030
3B    Temp   .FILL 0      ; Biến tạm
3C
3D    .END

```

Xuất liệu ví dụ:

```

Z80 LC3 Console - Version 1.0.0.0

Nhập hai số nguyên từ 0-9;
Input a character>2
Input a character>3
Input a character>2
Input a character>9
Input a character>9
Tổng là: 31
----- Halting the processor -----

```

Chương trình trên cho phép nhập vào một số sau khi in ra câu “Nhập hai số nguyên từ 0-9;” ở dòng 01 và 02 bằng lệnh GETC ở dòng 03, nên ký số nhập không hiện ra màn hình, và ta không thấy câu nhắc tự động “Input a character>” như khi thực hiện lệnh IN.

Từ dòng 20 tới dòng 25 là vòng lặp để tìm các số ở hàng chục và đơn vị của tổng vừa mới tìm được ở các dòng từ 15 tới dòng 1C. Giải thuật tổng quát của vòng lặp này với ví dụ tổng của các số trên là $8+3+2+9+9 = 32$ như sau:

Khởi động: R0 = 32, R2 = 0

$$- R0 = 32 - 10 = 22 > 0 \quad : \text{tiếp tục lặp}$$

↓

$$R2 = R2 + 1 = 1$$

$$- R0 = 22 - 10 = 12 > 0 \quad : \text{tiếp tục lặp}$$

↓

$$R2 = R2 + 1 = 2$$

$$- R0 = 12 - 10 = 2 > 0 \quad : \text{tiếp tục lặp}$$

↓

$$R2 = R2 + 1 = 3 \quad : \text{số hàng chục}$$

$$- R0 = 2 - 10 = -8 < 0 \quad : \text{kết thúc lặp}$$

↓

$$R1 = R0 + 10 = -8 + 10 = 2 : \text{số hàng đơn vị}$$

Chi tiết các lệnh được trình bày trong phần ghi chú sau mỗi lệnh, độc giả có thể đọc để hiểu thêm chi tiết.

BÀI TẬP CUỐI CHƯƠNG

5.1 Hãy giải thích xem chương trình sau đây làm gì?

```

.ORIG      x3000
LD         R2, ZERO
LD         R0, M0
LD         R1, M1
LOOP      BRz      DONE
          ADD      R2, R2, R0
          ADD      R1, R1, #-1
          BR       LOOP .
DONE      ST       R2, RESULT
          HALT
RESULT    .FILL   x0000
ZERO     .FILL   x0000
M0       .FILL   x0004
M1       .FILL   x0803
.END

```

Sau khi chương trình được thực hiện xong, biến RESULT chứa trị bao nhiêu?

- 5.2** Mục đích của mã giả .END là gì? Nó khác với lệnh HALT như thế nào? Nếu trong chương trình hợp ngữ LC-3 mà không có .END thì sẽ có chuyện gì xảy ra? Giải thích.
- 5.3** Viết chương trình hợp ngữ LC-3 nhập một hằng số ở hệ thập phân (ví dụ: #12) hay ở hệ thập lục phân (ví dụ: x2A) và in ra màn hình ở dạng biểu diễn nhị phân của hằng số.
- 5.4** Viết chương trình hợp ngữ LC-3 đếm và in ra màn hình số lần xuất hiện của ký tự đã cho (nhập từ bàn phím) trong một file ký tự ở vị trí xác định trước trong bộ nhớ.
- 5.5** Viết chương trình hợp ngữ LC-3 thực hiện các việc sau:
- Nhập hai số nguyên 0 – 9 từ bàn phím.

- Tính tích của hai số trên.
- In ra màn hình kết quả.

Lưu ý: cần thực hiện tất cả các thông báo nhắc nhở khi có sai sót, hay khi nhập xuất trị.

- 5.6 Thực hiện lại bài 5.2 với yêu cầu là hai số nguyên bất kỳ có thể âm hay dương.

Gợi ý: dữ liệu nhập, xuất và khi tính toán đều hoàn toàn ở dạng chuỗi.

- 5.7 Thực hiện tương tự bài 5.2 nhưng tính tổng, hiệu, tích và thương của hai số từ 0 – 9. Lưu ý, cho phép tạo menu chọn thao tác tính toán theo dạng:

Phép tính:

1. Cộng
2. Trừ
3. Nhân
4. Chia
5. Kết thúc

Chọn thao tác (1-5):

- 5.8 Thực hiện tương tự bài 5.3 và 5.4 nhưng tính tổng, hiệu, tích và thương của hai số từ bất kỳ.

- 5.9 Viết chương trình nhập trị vào một mảng các số nguyên dương. Tính tổng tất cả các phần tử có trong mảng và in ra màn hình tổng này.

- 5.10 Viết một chương trình hợp ngữ LC-3 đếm số bit 1 trong giá trị được chứa trong thanh ghi R0, và kết quả này được chứa trong R1. In ra màn hình trị này.

Ví dụ: Nếu R0 chứa 0001110010011111, thì sau khi chương trình thực thi, kết quả được chứa trong R1 là 0000 0000 0000 1001 (= 9).

Chương 6

CÁC VẤN ĐỀ KHÁC

6.1 XUẤT NHẬP

Vấn đề xuất nhập (*Input/Output*) là quan trọng với máy tính nói chung và mô hình von Neumann nói riêng. Vì phải có cách để đưa thông tin vào máy tính để xử lý, và cũng phải có cách để lấy kết quả từ quá trình xử lý đưa ra cho con người sử dụng. Như đã biết, trong LC-3, chúng ta có lệnh TRAP, yêu cầu hệ điều hành thực hiện tác vụ cụ thể. Cơ chế xuất nhập là tổng quát, việc tìm hiểu chúng là cần thiết, đặc biệt là các thiết bị bàn phím và màn hình.

Có một số vấn đề cơ bản về xuất nhập của máy tính mà chúng ta cần phải biết, đó là thanh ghi thiết bị, cơ chế quản lý thanh ghi thiết bị, cơ chế truyền dữ liệu, quản lý giao tiếp.

Một thiết bị xuất nhập được coi là một thiết bị đơn lẻ, và để tương tác với nó ta thường sử dụng các thanh ghi thiết bị (*device register*) tương ứng. Các thiết bị xuất nhập đơn giản nhất thường có tối thiểu hai thanh ghi thiết bị: một để chứa dữ liệu được truyền giữa thiết bị và máy tính, và một để cho biết thông tin về trạng thái làm việc của thiết bị.

Khi có một lệnh cần giao tiếp với một thiết bị xuất nhập thì nó cần làm việc với thanh ghi thiết bị tương ứng. Việc sử dụng thanh ghi thiết bị phải cần cơ chế quản lý thanh ghi thiết bị. Tổng quát, có hai cách thức quản lý thanh ghi thiết bị đã được sử dụng cho tới nay, đó là lệnh xuất nhập đặc biệt (*Special Input/Output Instructions*) và xuất nhập qua bộ nhớ (*Memory-Mapped I/O*). Một số máy tính dùng các lệnh xuất nhập đặc biệt với sự chỉ định thanh ghi thiết bị cần

dùng (tức thiết bị xuất nhập cần làm việc) và thao tác cần làm với thiết bị này, ví dụ như các máy tính DEC PDP-8. Phần lớn máy tính hiện nay lại sử dụng các lệnh chuyển dữ liệu để chuyển dữ liệu vào và ra khỏi bộ nhớ tương ứng với vị trí của thanh ghi thiết bị, mà không cần thêm các lệnh đặc biệt để đối phó với các thao tác xuất nhập như cách thức một. Khi đó, các lệnh chuyển dữ liệu được dùng để nạp và lưu dữ liệu giữa bộ nhớ và các thanh ghi đa dụng. Thí dụ, một lệnh nạp (load), trong đó địa chỉ nguồn xác định thanh ghi thiết bị nhập trong bộ nhớ, là một lệnh nhập. Tương tự, lệnh lưu (store), trong đó địa chỉ đích xác định thanh ghi thiết bị xuất trong bộ nhớ, là một lệnh xuất. Như vậy, để quản lý được tất cả các thiết bị trong hệ thống, mỗi thanh ghi thiết bị cần phải được sắp xếp vị trí trong không gian địa chỉ bộ nhớ ở cấp kiến trúc tập lệnh (ISA). Máy tính LC-3 sử dụng cách thức này. Cụ thể, không gian địa chỉ từ x0000 tới xFDFF là các ô nhớ lưu chương trình và dữ liệu, còn từ xFE00 tới xFFFF được dành riêng cho các thanh ghi xuất nhập như sau:

Địa chỉ	Tên thanh ghi xuất nhập
xFE00	Thanh ghi trạng thái của bàn phím (Keyboard Status Register) (bit [15] = 1 khi có phím mới được nhấn)
xFE02	Thanh ghi dữ liệu của bàn phím (Keyboard Data Register) (bit[7:0] chứa mã của ký tự mới được nhấn)
xFE04	Thanh ghi trạng thái của màn hình (Display Status Register) (bit [15] = 1 khi thiết bị sẵn sàng cho việc hiển thị ký tự mới)
xFE06	Thanh ghi dữ liệu của màn hình (Display Data Register) (bit[7:0] chứa mã của ký tự sẽ được hiển thị ra màn hình)
xFFFFE	Thanh ghi điều khiển máy tính (Machine Control Register)

Hầu như các thiết bị xuất nhập đều có tốc độ hoạt động rất chậm so với tốc độ của bộ xử lý. Ví dụ, tốc độ gõ ký tự từ bàn phím từ bàn tay con người tối đa vào khoảng 10 ký tự một giây, tốc độ của bộ vi xử lý loại đời cũ 300 MHz, tức khoảng 3,3 nano giây cho một chu kỳ xung clock. Nếu vi xử lý này thực thi một lệnh tốn 10 chu kỳ xung clock, tức 33 nano giây/lệnh, thì nó có khả năng nhận được khoảng 30 triệu ký tự một giây. Đây là quá một con số khổng lồ so với khả năng gõ của con người.

Có hai cách thức truyền nhận dữ liệu giữa thiết bị xuất nhập và máy tính: bất đồng bộ (*asynchronous*) và đồng bộ (*synchronous*). Nếu tốc độ hoạt động của thiết bị xuất nhập khác với tốc độ của bộ xử lý, và việc truyền dữ liệu của thiết bị xuất nhập là không xác định trước thì ta có cơ chế truyền nhận bất đồng bộ. Lúc này, việc truyền nhận dữ liệu giữa bộ xử lý và thiết bị xuất nhập đòi hỏi phải có nghi thức (*protocol*) hay cơ chế bắt tay (*handshaking*). Với bàn phím, chúng ta cần một thanh ghi trạng thái 1 bit, gọi là cờ (*flag*), để báo có một ký tự đã được gõ. Với màn hình, thanh ghi trạng thái 1 bit cũng cần được sử dụng để báo ký tự vừa mới được gõ đã được hiển thị hay chưa.

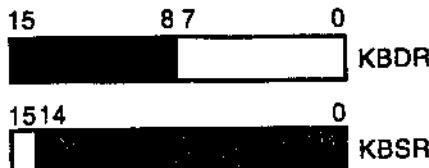
Nếu việc truyền nhận dữ liệu là xác định theo một quy định phần cứng, ví dụ người gõ với một tốc độ cố định, thì cứ sau một thời khoảng xác định máy tính cứ đọc một ký tự, thì ta có quá trình truyền nhận là đồng bộ. Khi đó ta không cần thêm bất kỳ một cờ điều khiển nào cho quá trình truyền nhận nữa.

Nếu trong khi bộ xử lý đang tính toán, và người dùng máy tính lại gõ một ký tự, thì ký tự được gõ cần phải được đưa vào máy tính để xử lý. Có hai cách để điều khiển việc giao tiếp giữa máy tính và thiết bị xuất nhập: ngắt (*interrupt-driven*) và thu thập (*polling*). Ta hãy xem cơ chế ngắt áp dụng cho thiết bị nhập là bàn phím. Trong khi bộ xử lý đang làm việc riêng của nó, thì có một phím được nhấn, lúc này bàn phím chủ động “la” lên với bộ xử lý: “Ê, có một phím vừa mới được nhấn kia. Mã ASCII đang ở trong thanh ghi của thiết bị nhập đó.” Và bộ xử lý sẽ tạm dừng việc riêng đang xử lý để nhận ký tự này. Như vậy, với ngắt, thiết bị xuất nhập đóng vai trò điều khiển việc giao tiếp. Trong khi đó, với cơ chế thu thập, bộ xử lý điều khiển việc giao tiếp với thiết bị xuất nhập, nó lặp đi lặp lại việc hỏi (tức thu thập) bit trạng thái cho tới khi nó thấy bit trạng thái đã được thiết lập. Và lúc này, bộ xử lý sẽ đọc ký tự từ thanh ghi thiết bị.

6.1.1 Nhập từ bàn phím

Để thực hiện nhập ký tự từ bàn phím, ta cần hai thứ: một thanh ghi dữ liệu chứa ký tự được nhập, và một cơ chế đồng bộ để bộ xử lý biết khi nào có việc nhập xảy ra, đó chính là thanh ghi trạng thái của bàn phím. Hai thanh ghi này, như đã biết, là thanh ghi dữ liệu bàn phím (KBDR) và thanh ghi trạng thái bàn phím (KBSR) như

trong hình 6.1. Chúng được sắp xếp trong bộ nhớ ở địa chỉ tương ứng xFE02 và xFE00 với các bit theo quy định.



Hình 6.1 Các thanh ghi thiết bị bàn phím

Khi một phím trên bàn phím được ấn, mã ASCII của phím đó được nạp vào các bit KBDR[7:0] và các mạch điện tử liên quan với bàn phím tự động bật bit KBSR[15] (còn được gọi là bit *Ready*) lên 1. Khi LC-3 đọc KBDR (và đã đọc xong!), các mạch điện tử liên quan với bàn phím tự động xóa bit KBSR[15] về 0, cho phép phím khác được ấn. Nếu KBSR[15] đang là 1, mã ASCII tương ứng với phím vừa được nhấn hãy còn chưa được đọc, và do vậy bàn phím vẫn chưa thể tiếp tục làm việc được.

Nếu việc xuất nhập được bộ xử lý điều khiển (tức *polling*), thì một chương trình có thể kiểm tra lặp đi lặp lại bit KBSR[15] cho tới khi nó thấy bit này được bật. Lúc này, bộ xử lý có thể nạp mã ASCII trong KBDR vào một trong các thanh ghi LC-3.

Thủ tục trong ví dụ sau đây minh họa cho việc nạp thanh ghi R0 bằng mã ASCII của phím vừa được nhập.

Ví dụ 6.1

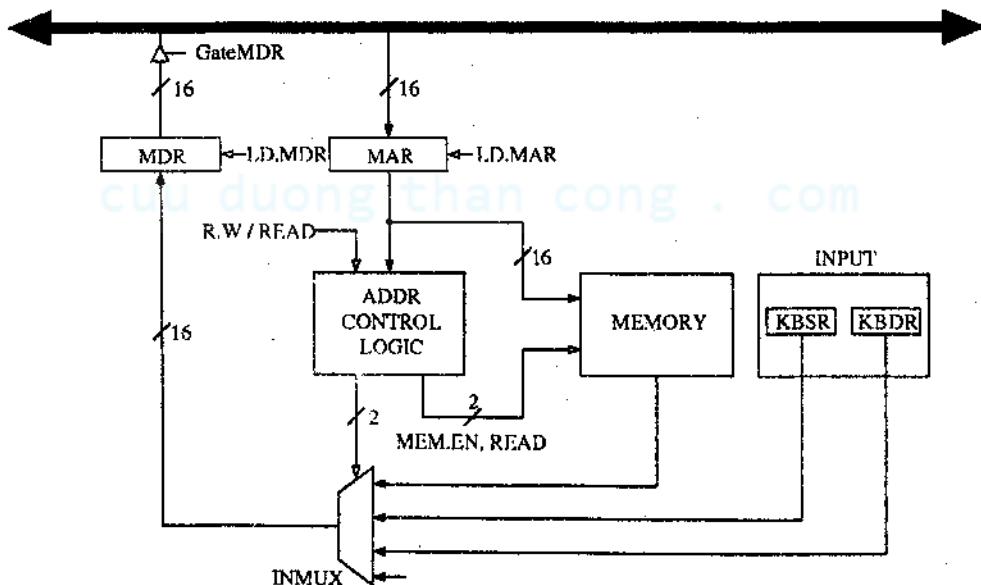
```

01 POLL      LDI R0, KBSRPtr
02          BRzp POLL
03          LDI R0, KBDRPtr
04 ...
05 KBSRPtr .FILL xFE00
06 KBDRPtr .FILL xFE02

```

Dòng 01 và 02 thành lập một vòng lặp để kiểm tra bit KBSR[15] có bằng 0 (tức cả thanh ghi KBSR bằng 0) hay không, nếu nó bằng 0, tức chưa có phím nào được nhấn kể từ khi bộ xử lý đọc thanh ghi dữ

liệu KBDR. Chú ý, ta dùng lệnh LDI để nạp vào thanh ghi R0 nội dung của ô nhớ xFE00 (chứ không phải bản thân trị xFE00 như khi dùng lệnh LD), địa chỉ trong bộ nhớ của thanh ghi KBSR. Nếu bit[15] bị xóa (bằng 0), lệnh BRzp sẽ chuyển điều khiển tới POLL và bắt đầu một chu trình lặp khác. Khi có một phím nào đó được nhấn, KBDR sẽ được nạp mã ASCII của phím đó và bit Ready, KBSR[15], sẽ được bật lên 1, tức thanh ghi KBSR là một số âm (vì theo quy ước, LC-3 chứa số nguyên dạng bù 2), làm cho hai thanh ghi trạng thái Z = 0 và P = 0. Điều này làm lệnh rẽ nhánh bị sai, và lệnh ở dòng 03 được thực thi. Lúc này, một lần nữa lệnh LDI được sử dụng để nạp vào thanh ghi R0 nội dung của ô nhớ xFE02, là địa chỉ qua bộ nhớ của KBDR. Thủ tục nhập đã hoàn tất, chương trình sẽ tiếp tục ở lệnh kế.



Hình 6.2 Nhập qua bộ nhớ

Hình 6.2 trình bày đường truyền dữ liệu thêm vào để hiện thực hóa việc nhập qua bộ nhớ. Các thao tác cũng tương tự như trong phase EXECUTE của lệnh Load. Có ba bước chủ yếu như sau:

1. Thanh ghi MAR được nạp bằng địa chỉ của ô nhớ cần đọc.
2. Bộ nhớ được đọc, dữ liệu từ ô nhớ theo yêu cầu được nạp vào thanh ghi MDR.
3. Thanh ghi đích (DR) được nạp bằng nội dung của MDR.

Trong trường hợp nhập qua bộ nhớ, các bước cũng hoàn toàn tương tự, chỉ trừ thay vì MAR được nạp bằng địa chỉ ô nhớ, thì nó lại được nạp bằng địa chỉ của thanh ghi thiết bị. Và thay vì bộ logic điều khiển địa chỉ kích hoạt bộ nhớ để đọc, nó sẽ chọn thanh ghi thiết bị tương ứng cho việc nhập trị vào thanh ghi MDR.

6.1.2 Xuất ra màn hình

Thao tác xuất liệu cũng tương tự như việc nhập, với hai thanh ghi DDR (*Display Data Register*) và DSR (*Display Status Register*) thay thế vai trò cho hai thanh ghi KBDR và KBSR một cách tương ứng. Trong LC-3, DDR được quy định ở địa chỉ xFE06, còn DSR ở xFE04. Với DDR, các bit [7:0] được dùng để chứa dữ liệu, các bit [15:8] chứa x00. Còn với DSR, bit[15] quy định cơ chế đồng bộ, nghĩa là bit Ready. Hình 6.3 cho thấy điều này.



Hình 6.3 Các thanh ghi thiết bị màn hình

Bit DSR[15] điều khiển sự đồng bộ giữa một bộ xử lý nhanh và một màn hình hiển thị chậm. Khi LC-3 chuyển một mã ASCII vào DDR[7:0] để xuất, các mạch điện tử của màn hình tự động xóa bit DSR[15] cho quá trình xử lý xuất của ký tự trong DDR[7:0] được bắt đầu. Khi màn hình hoàn tất quá trình hiển thị ký tự ra màn hình, nó tự động bật lại DSR[15] lên 1 để báo cho bộ xử lý rằng bộ xử lý có thể chuyển một mã ASCII khác cho DDR để hiển thị. Chừng nào mà DSR[15] còn đang bị xóa (bằng 0), thì khi đó màn hình hãy còn đang xử lý ký tự trước đó, nên nó không thể nhận một ký tự nào để xuất từ bộ xử lý được.

Nếu việc xuất nhập được bộ xử lý điều khiển (tức *polling*), thì một chương trình có thể kiểm tra lặp đi lặp lại bit DSR[15] cho tới khi nó thấy bit này được bật, cho biết việc ghi một ký tự ra màn hình là đã sẵn sàng. Lúc này, bộ xử lý cần lưu mã ASCII của ký tự cần hiển thị vào DDR[7:0] để phân cứng (vốn quét liên tục vùng nhớ này) hiển thị ký tự ra màn hình.

Thủ tục ví dụ sau đây minh họa cho việc in ra màn hình ký tự có mã ASCII đang được chứa trong thanh ghi R0.

Ví dụ 6.2

```

01      POLL      LDI R1, DSRPtr
02                  BRzp POLL
03                  STI R0, DDRPtr
04                  ...
05      DSRPtr    .FILL xFE04
06      DDRPtr    .FILL xFE06

```

Tương tự như thủ tục nhập ký tự từ bàn phím, dòng 01 và 02 lặp đi lặp lại việc kiểm tra bit DSR[15] xem mạch điện tử của màn hình đã hoàn tất việc hiển thị ký tự vừa rồi hay chưa. Chú ý rằng lệnh LDI dùng kiểu truy xuất gián tiếp cho ô nhớ xFE04, tức địa chỉ qua bộ nhớ của DSR. Chừng nào DSR[15] còn bằng 0 (tức DSR bằng 0), mạch điện tử để hiển thị ký tự vẫn còn đang bận xử lý ký tự này, tức khi đó lệnh BRzp sẽ điều khiển rẽ nhánh tới nhánh POLL cho chu trình lặp khác. Khi mạch điện tử của màn hình hoàn tất quá trình hiển thị ký tự, nó tự động bật bit DSR[15] lên 1 (làm DSR lưu một số âm), lệnh rẽ nhánh bị sai nên lệnh ở dòng 03 được thực thi. Chú ý, việc dùng lệnh STI, lưu R0 vào xFE06, địa chỉ qua bộ nhớ của DDR cũng theo kiểu định vị gián tiếp. Việc ghi mã ASCII vào DDR cũng xóa bit DSR[15], tức không cho việc hiển thị ký tự khác cho tới khi ký tự này được hiển thị xong.

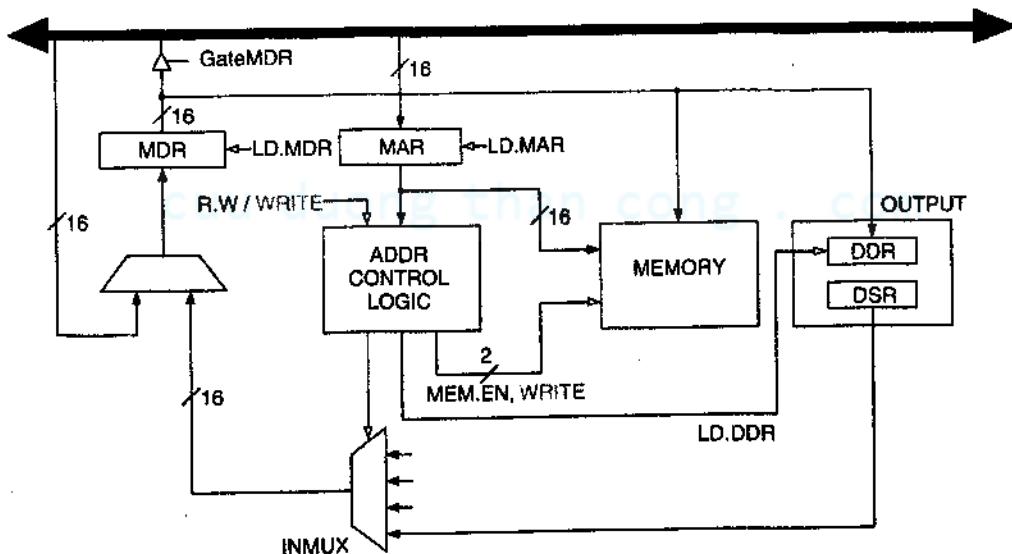
Hình 6.4 cũng trình bày đường truyền dữ liệu thêm vào để hiện thực hóa việc xuất qua bộ nhớ. Các thao tác cũng tương tự như trong phase EXECUTE của lệnh Store. Có ba bước chủ yếu như sau:

1. Thanh ghi MAR được nạp bằng địa chỉ của ô nhớ cần ghi.
2. MDR được nạp bằng dữ liệu để ghi vào ô nhớ.
3. Ô nhớ được ghi, tức dữ liệu trong MDR được ghi vào ô nhớ đã được xác định.

Trong trường hợp hiển thị qua bộ nhớ, các bước cũng hoàn toàn tương tự, chỉ trừ thay vì MAR được nạp bằng địa chỉ ô nhớ, thì nó lại được nạp bằng địa chỉ của thanh ghi thiết bị. Và thay vì bộ logic điều

khiến địa chỉ kích hoạt bộ nhớ để ghi, nó sẽ tác động chọn thanh ghi thiết bị tương ứng cho việc hiển thị DDR nhận ký tự từ MDR.

Việc xuất qua bộ nhớ cũng yêu cầu khả năng đọc các thanh ghi thiết bị xuất. Cụ thể, trước khi DDR được nạp, bit Ready đã phải ở trạng thái 1, chỉ ra ký tự vừa rồi đã hoàn toàn được ghi ra màn hình. Lệnh LDI và BRzp ở các dòng 01 và 02 thực hiện việc kiểm tra. Để làm điều này, LDI đọc thanh ghi thiết bị xuất DSR, và BRzp kiểm tra bit[15]. Nếu MAR được nạp bằng trị xFE04, tức địa chỉ qua bộ nhớ của DSR, bộ logic điều khiển địa chỉ chọn DSR là đầu vào cho MDR, để sau đó nó được nạp vào R1 và các mã điều kiện sẽ được kiểm tra.



Hình 6.4 Xuất qua bộ nhớ

Đoạn chương trình ví dụ sau minh họa việc kết hợp thủ tục nhập một ký tự và xuất in ra màn hình ký tự này.

Ví dụ 6.3

POLL1	LDI R0, KBSRPtr ; kiểm tra nhập ký tự
	BRzp POLL1
	LDI R0, KBDRPtr
POLL2	LDI R1, DSRPtr ; kiểm tra sự sẵn sàng của thanh ghi xuất
	BRzp POLL2

STI R0, DDRPtr

	...	
KBSRPtr	.FILL xFE00	; Địa chỉ của KBSR
KBDRPtr	.FILL xFE02	; Địa chỉ của KBDR
DSRPtr	.FILL xFE04	; Địa chỉ của DSR
DDRPtr	.FILL xFE06	; Địa chỉ của DDR

Chương trình ví dụ sau in một câu nhắc nhở ra màn hình để nhập ký tự, sau đó nhập ký tự và in ra màn hình ký tự vừa nhập.

Ví dụ 6.4

01	START	ST	R1, SaveR1	; Lưu trữ các thanh ghi được dùng trong
02		ST	R2, SaveR2	; thủ tục này
03		ST	R3, SaveR3	
04	;			
05		LD	R2, NewLine	
06	POLL1	LDI	R3, DSR	; Lặp cho tới khi
07		BRzp	POLL1	; màn hình sẵn sàng
08		STI	R2, DDR	; Dời cursor xuống hàng mới
09	;			
0A		LEA	R1, Prompt	; Địa chỉ bắt đầu của chuỗi
0B	LOOP	LDR	R0, R1, #0	; Kiểm tra ký tự cuối chuỗi
0C		BRz	Input	; Đã in xong chuỗi, nhập ký tự
0D	POLL2	LDI	R3, DSR	; Lặp cho tới khi
0E		BRzp	POLL2	; màn hình sẵn sàng
0F		STI	R0, DDR	; In từng ký tự trong chuỗi
10		ADD	R1, R1, #1	; Tăng con trỏ chỉ tới ký tự kế
11		BRnzp	LOOP	; Lặp để in từng ký tự
12	;			
13	Input	LDI	R3, KBSR	; Kiểm tra cho tới khi
14		BRzp	Input	; một ký tự được ấn
15		LDI	R0, KBDR	; Nhận mã của ký tự vừa được nhấn
16	POLL3	LDI	R3, DSR	; Lặp cho tới khi
17		BRzp	POLL3	; màn hình sẵn sàng
18		STI	R0, DDR	; In ký tự vừa nhấn
19	;			

```

1A POLL4 LDI R3, DSR ; Lắp cho tới khi
1B BRzp POLL4 ; màn hình sẵn sàng
1C STI R2, DDR ; Dời cursor sang dòng mới
1D LD R1, SaveR1 ; Khôi phục lại các thanh ghi
1E LD R2, SaveR2 ; đã dùng trong thủ tục
1F LD R3, SaveR3 ; bằng các trị ban đầu
20 BRnzp NEXT_TASK ; Thực hiện tác vụ kế
21 ;
22 SaveR1 .BLKW 1 ; Biến bộ nhớ lưu các thanh ghi
23 SaveR2 .BLKW 1
24 SaveR3 .BLKW 1
25 DSR .FILL xFE04
26 DDR .FILL xFE06
27 KBSR .FILL xFE00
28 KBDR .FILL xFE02
29 Newline .FILL x000A ; mã ASCII của ký tự xuống dòng
2A Prompt .STRINGZ "Input a character: "

```

Xin mời độc giả tự giải thích chương trình trên với các ghi chú sau mỗi dòng lệnh. Chú ý, tác vụ kế tiếp (nếu có) là phần chương trình cần thực hiện sau khi thủ tục này hoàn tất. Nếu không có tác vụ nào, thì ta có thể xóa dòng 20 này đi. Ký tự có mã ASCII x0A là ký tự xuống hàng, mỗi khi in ký tự này, phần cứng điều khiển màn hình sẽ điều khiển cho cursor màn hình dời xuống dòng mới. Chuỗi ký tự khai báo bằng mã giả .STRINGZ luôn kết thúc bằng trị x0000, nên dùng nó để kiểm tra tới cuối chuỗi hay chưa ở dòng 0B và 0C.

6.1.3 Xuất nhập qua ngắt

Trong phần trên, chúng ta đã hiểu việc bộ xử lý điều khiển xuất nhập bằng polling với các chương trình ví dụ cụ thể, trong đó bộ xử lý phải kiểm tra bit Ready của các thanh ghi trạng thái để rẽ nhánh tới lệnh nhập hay xuất. Nếu việc xuất nhập được các thiết bị điều khiển, chúng ta có cơ chế ngắt (interrupt). Như vậy ngắt là gì? Đó là một tín hiệu không biết trước (bất đồng bộ) chỉ ra sự cần thiết phải chú ý hoặc là một sự kiện định trước (đồng bộ) trong phần mềm chỉ ra sự cần thiết phải thay đổi trong quá trình thực thi. Như vậy, bình thường khi chưa có ngắt, thiết bị xuất nhập

không có việc gì làm với chương trình đang được bộ xử lý thực thi. Nhưng khi có ngắt, nó có thể bắt chương trình đang chạy dừng lại, bắt bộ xử lý thực hiện các nhu cầu của thiết bị xuất nhập, và rồi bắt chương trình đang bị dừng chạy lại như là không có gì xảy ra. Khác với polling, vốn luôn yêu cầu bộ xử lý dành nhiều thời gian để kiểm tra bit Ready cho tới khi nó được bật, ngắt không ảnh hưởng tới quá trình thực thi chương trình của bộ xử lý cho tới khi nó yêu cầu bộ xử lý chú ý nhu cầu của thiết bị xuất nhập.

Có hai thành phần cho xuất nhập qua ngắt: (1) cơ chế khởi động cho phép thiết bị xuất nhập ngắt bộ xử lý, gọi là cơ chế tạo ngắt; và (2) cơ chế quản lý việc truyền dữ liệu xuất nhập, gọi là cơ chế xử lý ngắt. Phần đầu (1), cơ chế tạo ngắt, sẽ được trình bày ngay dưới đây, còn phần thứ hai (2), cơ chế quản lý ngắt, sẽ được nêu trong mục 6.3 khi học về stack.

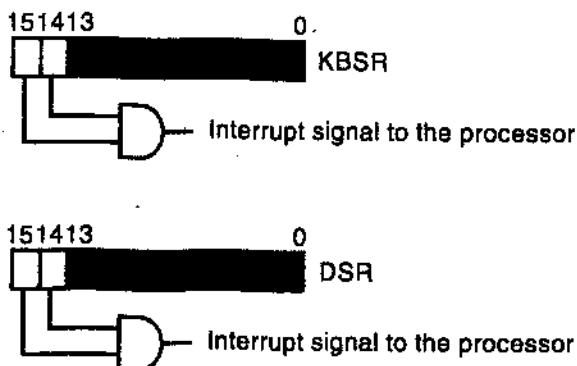
Để cơ chế tạo ngắt được thực hiện, các việc sau đây phải thỏa:

1. Thiết bị xuất nhập phải muốn dịch vụ.
2. Thiết bị phải có quyền yêu cầu dịch vụ.
3. Nhu cầu của thiết bị phải khẩn cấp hơn chương trình đang được bộ xử lý thực thi.

Nếu cả ba thứ trên đều hiện diện, bộ xử lý sẽ dừng thực thi chương trình đang chạy và quay sang xử lý ngắt.

Để một thiết bị xuất nhập tạo được yêu cầu ngắt, hai việc đầu tiên trong danh sách trên phải đúng: thiết bị phải muốn có dịch vụ, và nó phải có quyền yêu cầu dịch vụ đó.

Với bàn phím và màn hình, khi bit Ready trong KBSR hoặc DSR được bật, có nghĩa là thiết bị xuất nhập muốn dịch vụ nhập ký tự hay xuất ký tự ra màn hình, tức vấn đề thứ nhất đã thỏa. Việc thứ hai là bit cho phép ngắt, mà có thể được bộ xử lý bật hay xóa, tùy thuộc vào việc bộ xử lý có cho phép thiết bị xuất nhập được quyền yêu cầu dịch vụ ngắt hay không. Trong tất cả các thiết bị xuất nhập, bit cho phép ngắt (*Interrupt Enable - IE*) là một phần của thanh ghi trạng thái thiết bị. Trong các thanh ghi KBSR và DSR trong hình 6.5, bit IE là bit [14], nên yêu cầu ngắt từ thiết bị xuất nhập chính là phép AND luận lý giữa bit IE và bit Ready như trong hình 6.5.



Hình 6.5 Các bit cho phép ngắt và cách dùng

Nếu bit IE bị xóa, nó không ảnh hưởng gì tới việc bit Ready được bật; khi đó, thiết bị xuất nhập không có khả năng ngắt bộ xử lý. Trong trường hợp này, chương trình sẽ phải kiểm tra (*polling*) thiết bị xuất nhập để xác định xem nó có sẵn sàng hay chưa.

Nếu bit [14] này được bật, xuất nhập bằng ngắt được cho phép. Trong trường hợp này, ngay khi người nhập gõ một ký tự (hay ngay khi màn hình hoàn tất in ký tự vừa nhập), bit [15] được bật. Điều này làm đầu ra cống AND lên 1, xuất hiện một yêu cầu ngắt được tạo ra từ thiết bị xuất nhập.

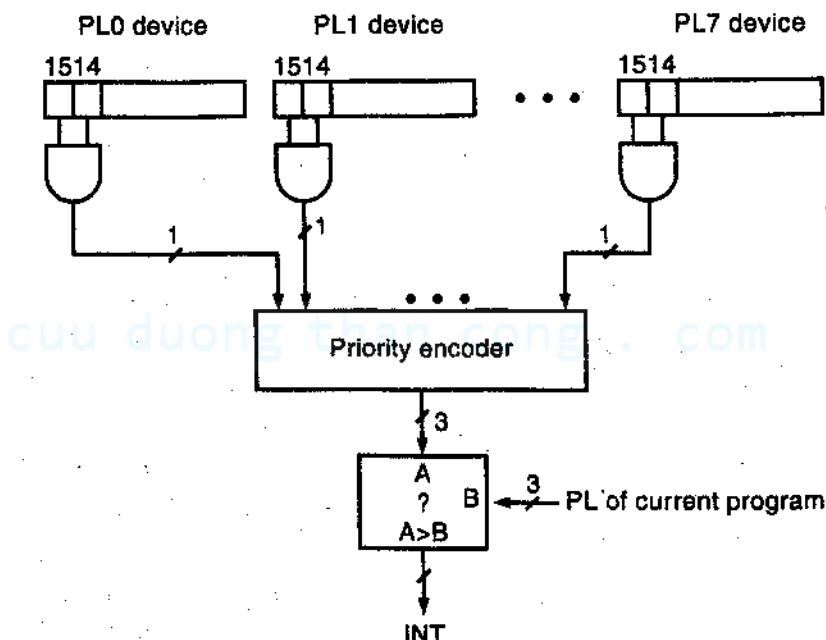
Vấn đề thứ ba trong danh sách các việc phải thỏa để một thiết bị xuất nhập thật sự ngắt được bộ xử lý, đó là yêu cầu ngắt phải ở mức độ khẩn cấp. Để quy định mức độ khẩn cấp ta có độ ưu tiên.

Bình thường, một chương trình đang được thực thi đang ở một mức ưu tiên xác định. Hầu hết các máy tính đều có tập các mức ưu tiên mà các chương trình có thể chạy. LC-3 có tám mức ưu tiên, từ PL0 tới PL7. Trị càng lớn, càng có độ ưu tiên cao hơn. Độ ưu tiên của chương trình thường được xem là độ khẩn cấp của yêu cầu để chạy chương trình đó. Nếu có một chương trình đang chạy ở một mức ưu tiên PL nào đó, mà lại có một chương trình ở mức ưu tiên PL cao hơn yêu cầu làm việc với máy tính, thì chương trình ở mức ưu tiên thấp hơn sẽ được tạm dừng cho tới khi chương trình ở mức độ cao hơn chạy xong, máy tính sẽ quay lại chạy tiếp chương trình mức độ ưu tiên thấp hơn. Ví dụ, máy tính đang chạy chương trình tính bảng lương ở mức PL0, việc này mất cả đêm để hoàn thành và không có tính khẩn cấp. Đột nhiên, trong đêm chương trình điều khiển nhà

máy hạt nhân cần máy tính xử lý ngay một vấn đề quan trọng, và chương trình này được gán cho mức PL6. Tất nhiên, máy tính sẽ ngưng ngay việc tính lương để xử ngay vấn đề nguy hiểm kia, và chúng ta cũng không nề hà gì về chuyện lương bị trễ đi vài giờ, hay vài ngày.

Như vậy, để một thiết bị xuất nhập dững bộ xử lý và bắt đầu một yêu cầu xuất nhập qua ngắt, độ ưu tiên của yêu cầu này phải cao hơn độ ưu tiên của chương trình đang chạy mà nó muốn ngắt.

Chúng ta sẽ thấy ngay là bộ xử lý sẽ dừng thực thi chương trình đang chạy và phục vụ một yêu cầu ngắt khi có tín hiệu ngắt INT được xác nhận. Hình 6.6 trình bày việc này. Các thanh ghi trạng thái của thiết bị ngoại vi hoạt động ở nhiều mức độ ưu tiên khác nhau. Các bit [15] và [14] xác nhận tín hiệu yêu cầu ngắt của thiết bị, chúng sẽ được đưa vào bộ mã hóa ưu tiên. Đây chính là một cấu trúc luận lý tổ hợp, nó sẽ chọn yêu cầu ngắt cao nhất từ những cái được xác nhận. Nếu yêu cầu ngắt đó (A) cao hơn độ ưu tiên PL của chương trình đang chạy, tín hiệu INT được xác lập và chương trình đang thực thi dừng lại.

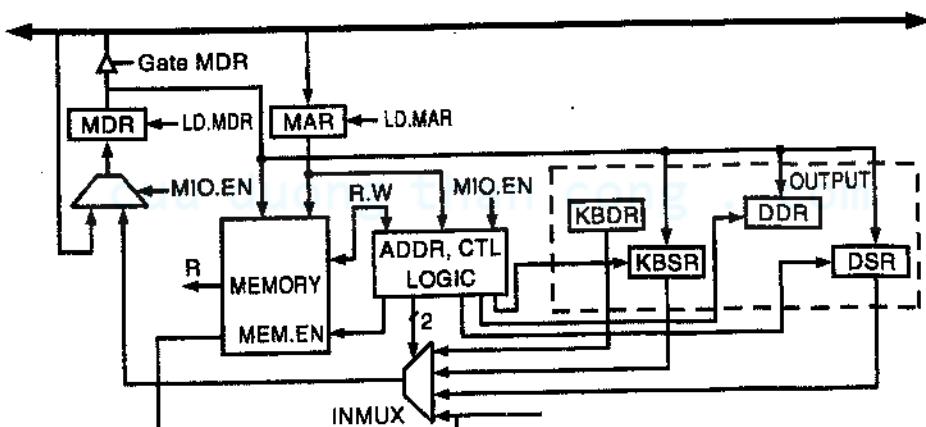


Hình 6.6 Tạo tín hiệu ngắt INT

Hình 6.2 và 6.4 có thể kết hợp lại để có một đường truyền dữ liệu như trong hình 6.7, để mô tả toàn bộ đặc tính cho việc xuất nhập qua ngắt mà chúng ta đã đề cập. Khối Address Control Logic điều khiển thao tác nhập hay xuất. Có ba tín hiệu nhập cho khối này. MIO.EN cho biết hoặc việc chuyển dữ liệu từ/đến bộ nhớ hoặc việc chuyển dữ liệu để xuất nhập diễn ra trong chu kỳ xong clock này. MAR chứa địa chỉ của ô nhớ hoặc địa chỉ qua bộ nhớ của thanh ghi thiết bị xuất nhập. R.W cho biết thao tác là nạp hay lưu sẽ diễn ra. Tùy theo trị của ba đầu vào này, bộ Address Control Logic sẽ không làm gì cả khi MIO.EN = 0, hoặc cung cấp các tín hiệu để điều khiển việc chuyển dữ liệu giữa MDR và bộ nhớ hay các thanh ghi xuất nhập.

Nếu R.W quy định nạp (bằng 1), việc chuyển dữ liệu sẽ từ bộ nhớ hay thiết bị xuất nhập tới MDR. Bộ Address Control Logic cung cấp tín hiệu điều khiển để chọn lựa đầu vào cho INMUX để tạo nguồn cho thanh ghi thiết bị xuất nhập thích hợp hay bộ nhớ (tùy thuộc vào MAR) và cũng tích cực bộ nhớ nếu MAR chứa địa chỉ của một ô nhớ.

Nếu R.W quy định lưu (bằng 0), nội dung của MDR sẽ được ghi vào ô nhớ hoặc một trong các thanh ghi thiết bị. Bộ Address Control Logic cho phép ghi bộ nhớ hoặc nó xác nhận tín hiệu cho phép nạp của thanh ghi thiết bị được xác định bởi MAR.



Hình 6.7 Phân đường truyền dữ liệu biểu diễn xuất nhập qua bộ nhớ

6.2 TRAP VÀ CHƯƠNG TRÌNH CON

6.2.1 Thủ tục TRAP của LC-3

Trong chương 4, chúng ta đã học lệnh TRAP. Với nó, một chương trình có thể yêu cầu hệ điều hành thực hiện một tác vụ nhân danh chương trình. Khi đó, hệ điều hành sẽ lấy quyền điều khiển máy tính, xử lý yêu cầu được quy định trong lệnh TRAP, và rồi trả điều khiển về lại cho chương trình người dùng. Chúng ta thường gọi yêu cầu này của chương trình người dùng là một cuộc gọi dịch vụ (*service call*) hay một cuộc gọi hệ thống (*system call*).

Cơ chế của lệnh TRAP gồm nhiều thứ như sau:

1. Một tập hợp các thủ tục dịch vụ nhân danh chương trình người dùng được thực thi bởi hệ điều hành. Các thủ tục này là một phần của hệ điều hành, chúng được chứa trong một vùng nhớ bắt đầu ở một địa chỉ bất kỳ được quy định trước trong bộ nhớ. LC-3 được thiết kế để có thể định nghĩa đến 256 thủ tục dịch vụ. Trong thực tế, nó có sáu thủ tục dịch vụ của hệ điều hành như sau:

Trap vector	Tên hợp ngữ	Chức năng
x20	GETC	Nhập một ký tự từ bàn phím, không hiển thị ra màn hình. Mã ASCII của ký tự được chứa trong R0.
x21	OUT	Ghi ký tự trong R0[7:0] ra màn hình.
x22	PUTS	In chuỗi ký tự kết thúc bằng x0000 có địa chỉ được chứa trong R0 ra màn hình.
x23	IN	Nhập một ký tự từ bàn phím vào R0, có hiển thị ký tự.
x24	PUTSP	In chuỗi ký tự kết thúc bằng x0000 có địa chỉ được chứa trong R0 với hai ký tự ASCII trong một ô nhớ 16 bit ra màn hình.
x25	HALT	Kết thúc thực thi và in một thông điệp ra màn hình.

2. Một bảng các địa chỉ bắt đầu của 256 thủ tục dịch vụ này. Bảng này được lưu trong bộ nhớ ở địa chỉ từ x0000 tới x00FF. Bảng này có nhiều tên gọi khác nhau như Khối điều khiển hệ thống (*System Control Block*), hay Bảng vector Trap (*Trap vector table*). Hình 6.8 là một phần của Bảng vector Trap của

LC-3. Các giá trị x0020, x0021, ... là số hiệu của các Trap vector. Các trị x0400, x04030, ... là địa chỉ các thủ tục dịch vụ tương ứng Trap vector.

⋮	⋮
x0020	x0400
x0021	x0430
x0022	x0450
x0023	x04A0
x0024	x04E0
x0025	xFD70
⋮	⋮

Hình 6.8 Bảng vector Trap

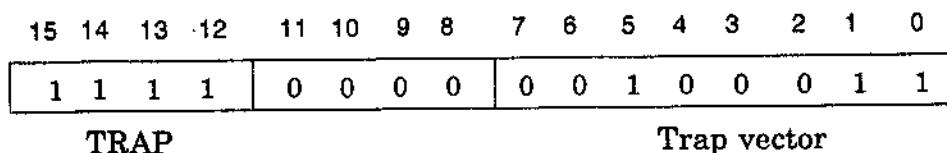
3. Lệnh Trap. Khi chương trình muốn hệ điều hành thực thi một thủ tục dịch vụ xác định nào đó nhân danh chương trình, và sau đó trả điều khiển về lại cho chương trình ban đầu, chương trình sử dụng lệnh TRAP. Lệnh Trap sẽ thay đổi PC bằng địa chỉ bắt đầu của thủ tục dịch vụ thích hợp tương ứng Trap vector trong lệnh.
4. Sự tái kết hợp với chương trình ban đầu. Thủ tục dịch vụ cần có cơ chế để trả điều khiển về cho chương trình.

Như trong lệnh TRAP dưới đây, mã thao tác là 1111, Trap vector là các bit [7:0], có trị là x23. Các bit [11:8] phải bằng 0. Với lệnh TRAP này, pha EXECUTE của nó sẽ làm bốn việc:

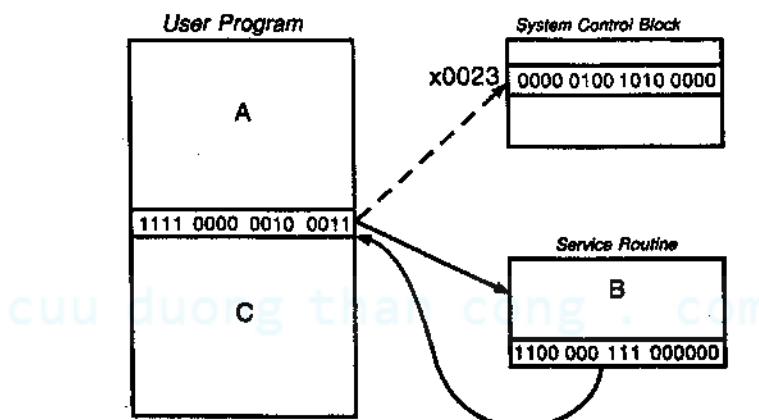
1. 8 bit Trap vector được mở rộng bằng các bit 0 để có 16 bit địa chỉ x0023, sau đó được nạp vào MAR. Đây chính là địa chỉ của một đầu vào trong bảng vector Trap.
2. Nội dung của ô nhớ x0023 sẽ được đọc, và trong trường này là x04A0 (xem hình 6.8) sẽ được nạp vào MDR.

3. Thanh ghi đa dụng R7 được nạp bằng nội dung hiện thời của thanh ghi PC. Đây là cách để trở về khi đã hoàn tất thủ tục dịch vụ.
4. Nội dung của MDR được nạp vào PC, hoàn tất pha và chu kỳ lệnh.

Vì PC đang chứa x04A0, nên việc xử lý tiếp tục ở địa chỉ x04A0. Đây là địa chỉ bắt đầu của thủ tục dịch vụ của hệ điều hành nhập một ký tự từ bàn phím. Chúng ta nói vector Trap “chỉ tới” địa chỉ bắt đầu của thủ tục Trap.



Hình 6.9 dưới đây trình bày cơ chế trở về từ lệnh Trap của LC-3. Sau khi thực hiện xong thủ tục dịch vụ B, lệnh JMP R7 (1100 000 111 000000) được dùng để trở lại chương trình chính C từ nội dung của R7, vốn là nội dung của thanh ghi PC được lưu. Trong hợp ngữ, mã giả gọi nhứ cho lệnh JMP R7 là RET.



Hình 6.9 Quá trình sử dụng lệnh Trap

Chương trình ví dụ 6.5 sau đây minh họa việc dùng lệnh Trap. Người dùng liên tục gõ các phím ký tự hoa, chương trình sẽ in ra ký tự thường tương ứng. Nếu người dùng gõ ký số ‘7’, chương trình sẽ kết thúc. Muốn hiểu được chương trình này một cách dễ dàng, độc giả cần tìm lại các vector Trap ở trang trước.

Ví dụ 6.5

.ORIG x3000

	LD	R2, TERM	; Trị âm của ASCII '7'
	LD	R3, ASCII	; Khoảng cách giữa ký tự ASCII hoa và thường
AGAIN	TRAP	x23	; Nhập ký tự
	ADD	R1, R2, R0	; Kiểm tra xem đã nhập '7' hay chưa
	BRz	EXIT	; Nếu đúng, kết thúc
	ADD	R0, R0, R3	; Đổi ký tự từ hoa sang thường
	TRAP	x21	; Xuất ra màn hình
	BRnzp	AGAIN	; Gõ lại ký tự khác
TERM	.FILL	xFFC9	; -'7'
ASCII	.FILL	x0020	; Khoảng cách ký tự hoa - thường
EXIT	TRAP	x25	; Kết thúc
	.END		

Chương trình ví dụ 6.6 dưới đây trình bày thủ tục dịch vụ nhập một ký tự minh họa cho hình 6.9, mà nếu ta gọi Trap x23, ta phải sử dụng thủ tục này. Lưu ý, đây là một dạng khác của ví dụ 6.4. Chúng ta có sử dụng một số mã giả gợi nhớ như .ORIG (dòng 03), RET (dòng 26), .END (dòng 31). Thủ tục này phải được đặt ở địa chỉ x04A0, nên lệnh .ORIG sẽ có dạng .ORIG x04A0. Tất nhiên, giá trị x04A0 cần phải được đặt trước vào ô nhớ có đầu vào x23 trong System control block, để khi Trap x23 được gọi thì thủ tục này sẽ được thực thi.

Ví dụ 6.6

01 ; Thủ tục dịch vụ để nhập ký tự

02 ;

03 .ORIG x04A0

04 START ST R1,SaveR1 ; Lưu trữ các thanh ghi

05 ST R2,SaveR2 ; để sử dụng sau khi

06 ST R3,SaveR3 ; trả về từ lệnh RET

07 ;

08 LD R2,Newline

09 L1 LDI R3,DSR ; Kiểm tra xem DDR đang tự do?

0A BRzp L1 ; Màn hình chưa rãnh

0B STI R2,DDR ; Chuyển cursor sang dòng mới

0C ;

0D		LEA	R1,Prompt	; Prompt là địa chỉ bắt đầu của chuỗi nhắc
0E				
0F	Loop	LDR	R0,R1,#0	; Lấy ký tự kế trong chuỗi nhắc
10		BRz	Input	; Kiểm tra ký tự kết thúc chuỗi
11	L2	LDI	R3,DSR	; Đợi màn hình rảnh
12		BRzp	L2	; Màn hình chưa rảnh
13		STI	R0,DDR	; In ký tự kế này ra màn hình
14				; từ chuỗi nhắc
15		ADD	R1,R1,#1	; tăng pointer chỉ tới chuỗi Prompt
16		BRnzp	Loop	; Xử lý ký tự kế
17	:			
18	Input	LDI	R3,KBSR	; Có một phím đã được gõ?
19		BRzp	Input	; Chưa có phím được gõ
1A		LDI	R0,KBDR	; Phím vừa gõ được nạp vào R0
1B	L3	LDI	R3,DSR	; Bàn phím
1C		BRzp	L3	; đang rảnh? Không
1D		STI	R0,DDR	; Rảnh rồi, hiển thị ký tự vừa nhập
1E				; ra màn hình
1F	:			
20	L4	LDI	R3,DSR	; Màn hình rảnh
21		BRzp	L4	; không? Không
22		STI	R2,DDR	; Rảnh rồi, dời cursor xuống hàng mới
23		LD	R1,SaveR1	; Thủ tục dịch vụ đã xong, phục hồi lại
24		LD	R2,SaveR2	; trị ban đầu cho các thanh ghi.
25		LD	R3,SaveR3	
26		RET		; Trả về từ Trap (tức JMP R7)
27	:			
28	SaveR1	.BLKW	1	
29	SaveR2	.BLKW	1	
2A	SaveR3	.BLKW	1	
2B	DSR	.FILL	xFE04	
2C	DDR	.FILL	xFE06	
2D	KBSR	.FILL	xFE00	
2E	KBDR	.FILL	xFE02	
2F	Newline	.FILL	x000A	; Mã ASCII cho hàng mới
30	Prompt	.STRINGZ	"Nhập một ký tự: "	
31		.END		

Chương trình ví dụ sau đây thực hiện thủ tục dịch vụ hiển thị ký tự ra màn hình, ứng với Trap vector x21. Trong bảng vector Trap, đầu vào x21 chứa trị x0430, nên dòng 01 trong thủ tục này là .ORIG x0430 cho biết thủ tục này được đặt từ ô nhớ có địa chỉ x0430.

Ví dụ 6.7

```

01          .ORIG    x0430      ; Địa chỉ bắt đầu của thủ tục
02          ST        R1, SaveR1 ; R1 được dùng cho polling
03
04          ;In ký tự
05  TryWrite LD       R1, DSR   ; Kiểm tra xem màn hình rãnh chưa
06          BRzp    TryWrite ; bit 15 = 1 => rãnh rồi
07  Writelt  STI     R0, DDR   ; In ký tự trong R0
08
09          ;Return from TRAP
0A  Return   LD       R1, SaveR1 ; Khôi phục trị ban đầu
0B          RET     ; Trở về (tức JMP R7)
0C  DSR     .FILL    xFE04
0D  DDR     .FILL    xFE06
0E  SaveR1  .BLKW    1
0F          .END

```

Khi đang thực hiện một chương trình, nếu ta muốn bộ xử lý dừng quá trình đang xử lý, ta cần dừng xung clock điều khiển máy tính. Trong LC-3, xung này được điều khiển bởi bit[15] trong thanh ghi Machine Control Register (MCR), vốn được sáp trong bộ nhớ ở địa chỉ xFFFE. Bình thường, MCR[15] bằng 1 (tức MCR = xFFFF), khi nó bị xóa (tức MCR = x7FFF), thì thủ tục dịch vụ Trap x25 được gọi để dừng bộ xử lý. Chương trình ở ví dụ 6.8 sau sẽ trình bày thủ tục dịch vụ của hệ thống Trap x25. Lưu ý, trong LC-3 đầu vào x25 trong bảng System control block sẽ chứa trị xFD70, tức thủ tục dịch vụ này của hệ thống được đặt từ ô nhớ xFD70, nên ta có dòng 01 là .ORIG xFD70. Chú ý, việc kiểm tra xem màn hình có rãnh để thực hiện Trap x21 hay không có thể được hiện dễ dàng, và không được sử dụng ở đây.

Ví dụ 6.8

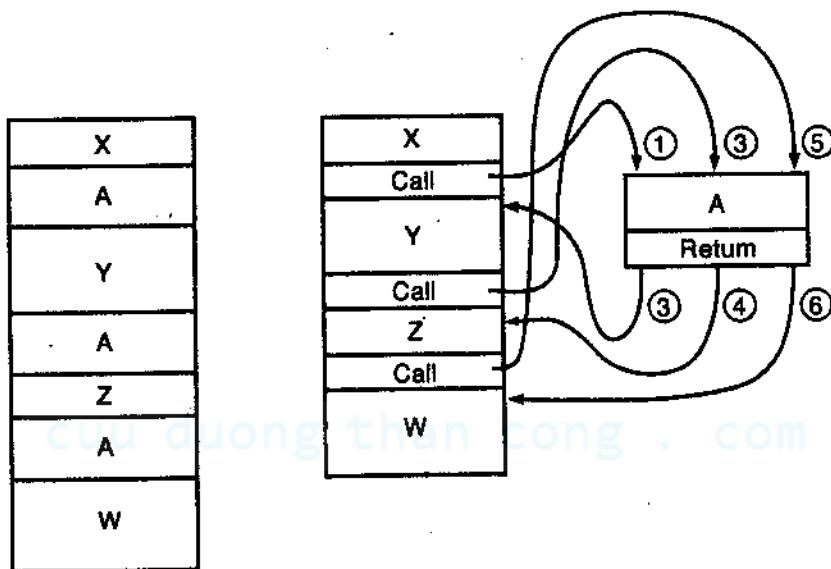
```

01      .ORIG    xFD7G          ; Địa chỉ bắt đầu của thủ tục hệ thống
02      ST       R0, SaveR0      ; Sao lưu các thanh ghi
03      ST       R1, SaveR1      ; cần dùng trong thủ tục
04      ST       R7, SaveR7      ;
05
06      ; Hiển thị câu thông báo khi máy tính bị dừng
07      LD       R0, ASCIINewLine
08      TRAP    x21              ; Đặt vị trí cursor ở dòng mới
09      LEA     R0, Message      ; Lấy địa chỉ của câu thông báo
0A      TRAP    x22              ; và in nó ra màn hình
0B      LD       R0, ASCIINewLine
0C      TRAP    x21
0D
0E      ; Xóa bit MCR[15] để dừng xung clock
0F      LDI     R1, MCR          ; Nạp thanh ghi R1 bằng MCR
10     LD       R0, MASK         ; MASK = x7FFF (tức bit 15 = 0)
11     AND     R0, R1, R0        ; Xóa bit 15 của bản sao copy của MCR
12     STI     R0, MCR          ; và nạp ngược lại nó vào MCR
13     ; Trở về chương trình chính từ thủ tục Halt
14     ;
15     LD       R7, SaveR7      ; Khôi phục lại trị ban đầu cho các
16     LD       R1, SaveR1      ; thanh ghi trước khi trở về
17     LD       R0, SaveR0
18     RET                  ; JMP R7
19
1A     ; Các hằng
1B     ASCIINewLine .FILL x000A
1C     SaveR0   .BLKW           1
1D     SaveR1   .BLKW           1
1E     SaveR7   .BLKW           1
1F     Message   .STRINGZ      "Halting the machine"
20     MCR    .FILL xFFFF
21     MASK.FILL x7FFF
22     .END

```

6.2.2 Chương trình con

Trong mục trên, chương trình con là thủ tục dịch vụ của hệ thống đã được thiết kế, và khi muốn trở về chương trình chính, nơi gọi lệnh Trap, ta sử dụng lệnh JMP R7. Việc thiết kế chương trình con của người dùng cũng tương tự như thủ tục của lệnh Trap. Hình 6.10 trình bày

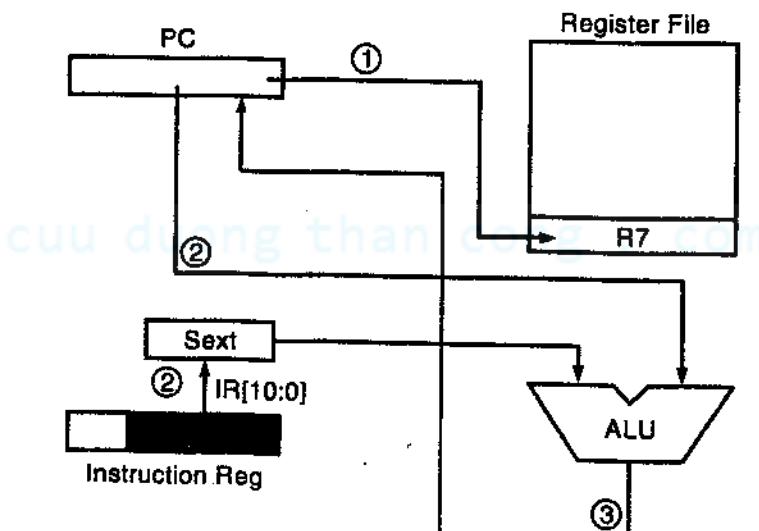


Hình 6.10 Dòng thực thi lệnh khi không có (a) và khi có chương trình con (b)

quá trình thực thi lệnh khi không sử dụng chương trình con 6.10a, và 6.10b khi có sử dụng chương trình con A ba lần. Việc gọi (Call) từ chương trình chính và trở về (Return) từ chương trình con tạo nên cơ chế Gọi/Trở về cũng như khi gọi và trả về từ lệnh Trap. Tuy nhiên, có sự khác nhau giữa chương trình con và thủ tục dịch vụ của hệ thống khi gọi bằng lệnh Trap, đó là thủ tục dịch vụ là do lập trình viên hệ thống thiết kế sẵn, và được đặt vào thư viện của hệ thống. Người dùng khi cần sẽ gọi vector Trap tương ứng là sử dụng, không cần thiết kế lại. Còn chương trình con thông thường, người viết chương trình phải tự thiết kế, và mã nguồn của nó có thể được đặt trong cùng modul với chương trình chính.

LC-3 quy định mã thao tác 0100 để gọi chương trình con. Lệnh này dùng một trong hai trạng thái địa chỉ để tính địa chỉ bắt đầu của chương trình con, đó là định vị tương đối PC (*PC-relative addressing*) hoặc định vị nền (*Base addressing*). Tương ứng, ta có các lệnh JSR và JSRR.

Lệnh JSR tính địa chỉ của chương trình con bằng việc mở rộng 11 bit offset [10:0] của lệnh thành 16 bit và cộng vào trị thanh ghi PC đã được tăng. Kiểu định vị này gần như giống với kiểu định vị của lệnh LD và ST, ngoại trừ việc sử dụng 11 bit offset. Hình 6.11 trình bày đường truyền dữ liệu với các thành phần cần thiết khi thực hiện lệnh JSR.



Hình 6.11 Đường truyền dữ liệu đơn giản cho lệnh JSR

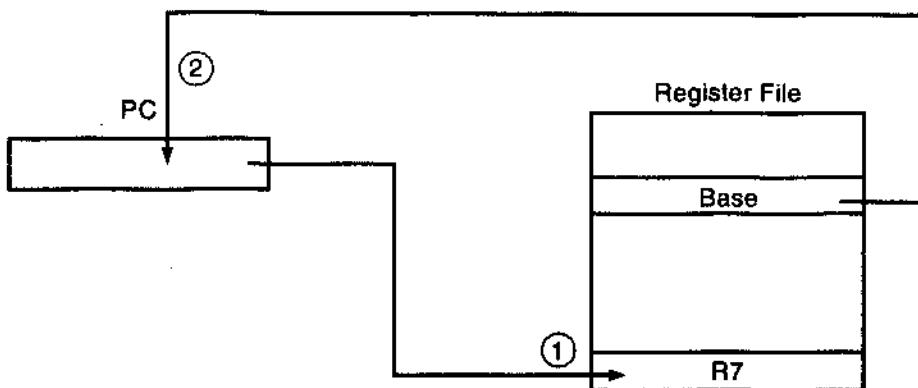
Nếu lệnh JSR được chứa ở ô nhớ x4200, sau khi nó được thực thi, PC sẽ được nạp bằng x3E05 (x4001 + xFC04) và R7 được nạp bằng x4201 như sau:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	0	0	0	0	0	1	0	0

JSR A PCoffset11

Lệnh JSRR giống như lệnh JSR trừ kiểu định vị địa chỉ. JSRR tính địa chỉ của chương trình con bằng cách dùng nội dung của thanh

ghi được xác định bởi các bit[8:6] của lệnh làm thanh ghi nền. Hình 6.12 trình bày đường truyền dữ liệu với các thành phần cần thiết để thực thi lệnh JSR.



Hình 6.12 Đường truyền dữ liệu đơn giản cho lệnh JSRR

Nếu lệnh JSRR sau đây được chứa trong ô nhớ x420A, và thanh ghi R5 đang chứa x3002, thì việc thực thi lệnh JSRR sau sẽ làm cho thanh ghi R7 được nạp bằng x420B, và PC được nạp bằng x3002.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0

JSRR A BaseR

Chương trình ở ví dụ 6.9 trình bày lại thủ tục dịch vụ nhập từ bàn phím ở ví dụ 6.6 với việc khai báo và sử dụng các chương trình con. Trong chương trình này, chuỗi ba lệnh ở các địa chỉ biểu tượng L1, L2, L3, và L4 (nói chung là LABEL)

LABEL	LDI	R3, DSR
BRzp	LABEL	
	STI	

được thay bằng chương trình con WriteChar như sau

WriteChar	LDI	R3, DSR
	BRzp	WriteChar
	STI	R2, DDR
	RET	

và khi cần gọi chương trình con này, ta sử dụng lệnh JSR WriteChar. Chú ý, để đảm bảo các thanh ghi được sử dụng đúng với giá trị, ta có thể thay đổi thanh ghi đã được dùng trong ví dụ 6.6. Cụ thể, chúng ta dùng lệnh

LDR R2, R1, #0

thay vì lệnh

LDR R0, R1, #0

Ví dụ 6.9

```

01          .ORIG      x04A0
02  START    ST         R7,SaveR7
03          JSR        SaveReg
04          LD          R2,Newline
05          JSR        WriteChar
06          LEA        R1,PROMPT
07  ;
08  ;
09  Loop     LDR        R2,R1,#0      ; Lấy ký tự kế trong chuỗi
10          BRz        Input
11          JSR        WriteChar
12          ADD        R1,R1,#1
13          BRnzp     Loop
14  ;
15          Input    JSR        ReadChar
16          ADD        R2,R0,#0      ; Ghi ký tự vào R2 để hiển thị
17          JSR        WriteChar      ; Hiển thị ký tự
18  ;
19          LD         R2, Newline
20          JSR        WriteChar
21          JSR        RestoreReg
22          LD         R7,SaveR7
23          RET        ; JMP R7 kết thúc
24          ;           ; thủ tục TRAP
25  SaveR7   .FILL      x0000
26  Newline  .FILL      x000A
27  Prompt   .STRINGZ "Nhập một ký tự > "
28  ;
29  WriteChar LDI        R3,DSR
30          BRzp      WriteChar
31          STI        R2,DDR

```

```

20           RET          ; JMP R7 kết thúc chương trình con
21   DSR     .FILL       xFE04
22   DDR     .FILL       xFE06
23   ;
24   ReadChar LDI        R3,KBSR
25           BRzp       ReadChar
26           LDI        R0,KBDR
27           RET
28   KBSR    .FILL       xFE00
29   KBDR    .FILL       xFE02
2A   ;
2B   SaveReg ST         R1,SaveR1
2C           ST         R2,SaveR2
2D           ST         R3,SaveR3
2E           ST         R4,SaveR4
2F           ST         R5,SaveR5
30           ST         R6,SaveR6
31           RET
32   ;
33   RestoreReg LD        R1,SaveR1
34           LD        R2,SaveR2
35           LD        R3,SaveR3
36           LD        R4,SaveR4
37           LD        R5,SaveR5
38           LD        R6,SaveR6
39           RET
3A   SaveR1  .FILL       x0000
3B   SaveR2  .FILL       x0000
3C   SaveR3  .FILL       x0000
3D   SaveR4  .FILL       x0000
3E   SaveR5  .FILL       x0000
3F   SaveR6  .FILL       x0000
40           END

```

Chương trình ví dụ 6.10 sau đây trình bày thủ tục dịch vụ TRAP x22 để in ra một chuỗi kết thúc bằng ký tự NULL (mã ASCII là 0). Trong chương trình này, chương trình con Return được dùng để khôi phục lại trị các thanh ghi được sử dụng trong thủ tục dịch vụ, mà có ý nghĩa sử dụng trong chương trình gọi thủ tục dịch vụ này.

Ví dụ 6.10

; Thủ tục dịch vụ in một chuỗi kết thúc bằng NULL ra màn hình

; PUTS (TRAP x22).

; Thông số nhập: R0 là con trỏ tới chuỗi cần in.

; Các thanh ghi cần lưu: R0, R1, R3, và R7

; khi nhảy vào thủ tục này

;

.ORIG	x0450	; Địa chỉ của thủ tục dịch vụ
ST	R7, SaveR7	; Lưu R7 để phục vụ trả về
ST	R0, SaveR0	; Lưu các thanh ghi bị thay đổi
ST	R1, SaveR1	; trị và được sử dụng trong thủ tục này
ST	R3, SaveR3	;

;

; Lặp qua từng ký tự của mảng

;

;

Loop LDR R1, R0, #0 ; Lấy từng ký tự

L2 BRz Return ; Nếu là ký tự NULL, xong !!!

LDI R3,DSR

BRzp L2

STI R1, DDR ; Ghi ký tự

ADD R0, R0, #1 ; Tăng con trỏ

BRnzp Loop ; Lặp để in ký tự kế

;

; Trở về từ việc gọi thủ tục dịch vụ

Return LD R3, SaveR3

LD R1, SaveR1

LD R0, SaveR0

LD R7, SaveR7

RET

;

; Vị trí các thanh ghi

DSR .FILL xFE04

DDR .FILL xFE06

SaveR0 .FILL x0000

SaveR1 .FILL x0000

SaveR3 .FILL x0000

SaveR7 .FILL x0000

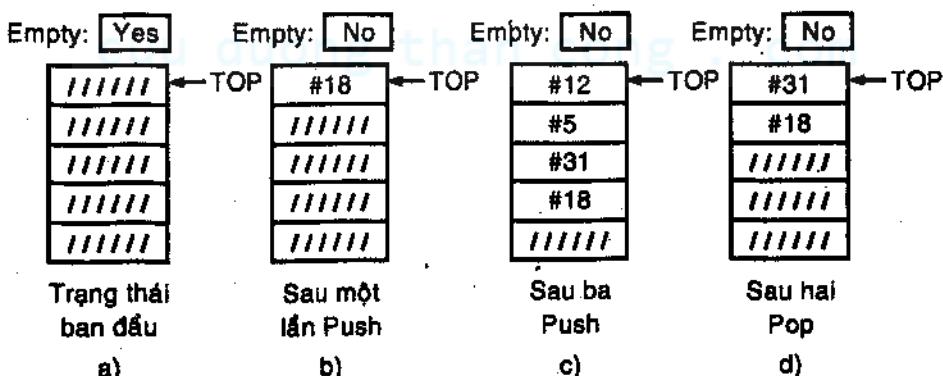
.END

6.3 NGĂN XẾP (Stack)

6.3.1 Các thao tác trong stack

Trong mục 6.1 chúng ta đã đề cập tới cơ chế ngắt với hai thành phần là cơ chế tạo ngắt và cơ chế xử lý ngắt. Với cơ chế xử lý ngắt, stack được sử dụng để quản lý việc truyền dữ liệu xuất nhập. Nên stack đóng vai trò quan trọng trong các thao tác hệ thống của máy tính. Hơn nữa, stack cũng là một cấu trúc dữ liệu quan trọng trong lập trình vì nó cho khái niệm lưu dữ liệu theo kiểu vào sau ra trước (LIFO-Last In First Out), vốn rất thuận lợi cho một bài toán lập trình, đặc biệt là các cấu trúc đệ quy.

Hình 6.13 trình bày một ví dụ về stack. Trong stack này có năm phần tử, mỗi phần tử có thể là ô nhớ hay thanh ghi, và có thể chứa được một giá trị quy định. Mỗi giá trị có thể được đưa vào hay lấy ra khỏi stack.



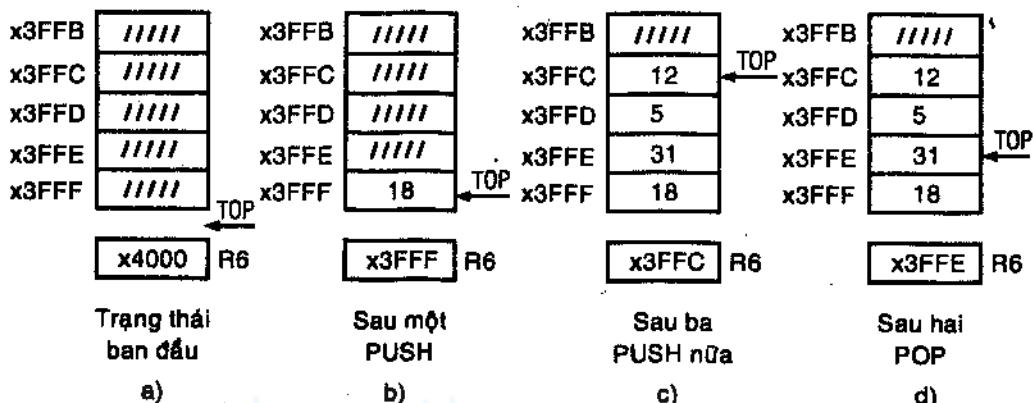
Hình 6.13 Các thao tác trên stack, dữ liệu bị đổi chỗ

Trong hình 6.13a, stack vừa được khởi tạo, nên đang trống. Việc truy xuất stack luôn bắt đầu từ phần tử đầu tiên, được chỉ bởi biến TOP. Nếu giá trị 18 được đẩy (PUSH) vào stack, chúng ta có hình 6.13b. Nếu tiếp tục có ba giá trị 31, 5, và 12 theo thứ tự được đưa thêm vào stack thì kết quả là hình 6.13c. Cuối cùng, khi có hai phần tử được lấy ra (POP) khỏi stack, chúng ta có hình 6.13d. Lưu ý, với stack này, các trị khi được đưa vào sẽ thực sự thay đổi vị trí chứa trị khi có thao tác trên stack.

Trong bộ nhớ máy tính, stack gồm nhiều ô nhớ liên tiếp nhau, và được quản lý bởi một con trỏ stack (*stack pointer*). Con trỏ này sẽ theo

dổi đỉnh stack, tức ô nhớ chứa trị vừa được đưa vào stack. Lưu ý, mỗi giá trị khi được đẩy vào trong stack được đưa vào một ô nhớ, và dữ liệu là không có sự di chuyển vật lý nào, chỉ có đỉnh stack là thay đổi theo TOP, quy định ô nhớ chứa dữ liệu sau cùng được đưa vào stack.

Hình 6.14 nêu ví dụ về stack này. Trong stack có năm vị trí địa chỉ từ x3FFF tới x3FFB. Thanh ghi R6 là con trỏ stack.



Hình 6.14 Các thao tác trên stack, dữ liệu không bị đổi chỗ

Hình 6.14a cho thấy trạng thái ban đầu trống của stack, lúc này R6 đang chứa x4000. Để chứa một trị vào stack, con trỏ stack R6 đầu tiên phải được giảm để chỉ tới ô nhớ địa chỉ x3FFF trong stack, sau đó trị 18 sẽ được chép vào đây. Hình 6.14b cho thấy stack sau khi đẩy trị 18 vào stack. Hình 6.14c chỉ ra stack sau khi ta đẩy thêm vào stack các trị theo thứ tự 31, 5, và 12. Lúc này R6 chứa x3FFC, và ta muốn lấy hai trị từ stack ra. Thao tác lấy trị POP ngược lại với thao tác đẩy trị vào stack PUSH. Đầu tiên, căn cứ vào R6, nội dung ô nhớ địa chỉ tương ứng x3FFC sẽ được chép ra (là 12), sau đó R6 sẽ giảm, chỉ tới ô nhớ là đỉnh stack mới, tức vị trí x3FFD. Tiếp tục lấy một phần tử nữa, tức ta có trị 5, lúc này R6 chứa x3FFE. Chú ý, thao tác POP chỉ đơn giản là căn cứ vào địa chỉ trong con trỏ stack mà chép nội dung từ ô nhớ tương ứng ra, nội dung ô nhớ đó không mất đi, tuy nhiên, với chúng ta, vì ta không quản lý chúng nữa, nên nội dung này dù còn nhưng cũng không có ý nghĩa sử dụng nữa. Hình 6.14d cho thấy stack sau 2 lần POP.

Với ví dụ và giải thích trên, thao tác PUSH và POP được viết ở dạng LC-3 như sau:

```
PUSH ADD R6, R6, #-1 ; giảm con trỏ stack
STR R0, R6, #0 ; lưu dữ liệu vào ô nhớ (R0)
```

và

```
POP LDR R0,R6,#0 ; Nạp dữ liệu từ đỉnh stack TOS
ADD R6, R6, #1 ; Tăng con trỏ stack
```

Tuy nhiên, chúng ta có thể thấy có vấn đề với hai đoạn chương trình trên vì chúng không thực hiện kiểm tra xem stack đầy hay chưa khi PUSH phần tử mới vào stack, cũng như không kiểm tra xem stack có trống không trước khi POP một phần tử ra khỏi stack. Hãy ví dụ của hình 6.14 để xây dựng hoàn chỉnh hai thủ tục này. Lưu ý, để kiểm tra stack có đầy hay không, ta so sánh con trỏ stack R6 với địa chỉ của ô nhớ giới hạn trên của stack, tức địa chỉ x3FFB, mà bù 2 của nó là xC005. Còn để kiểm tra xem stack có đang trống hay không, con trỏ R6 sẽ được so sánh với địa chỉ của ô nhớ là đáy stack, tức x4000, mà bù 2 của nó là xC000. Khi thao tác PUSH hay POP thành công, thanh ghi R5 sẽ cho trị là 0, còn nếu thất bại, R5 cho trị là 1. Hai thao tác hoàn chỉnh được trình bày như trong hai thủ tục dưới đây.

```
PUSH LD R1, MAX ; MAX = -x3FFB
ADD R2, R6, R1 ; So sánh xem con trỏ stack đang chỉ tới
; đỉnh hay chưa
BRz FAIL ; bằng cách so sánh với -x3FFB
ADD R6, R6, #-1
STR R0, R6, #0
AND R5, R5, #0 ; SUCCESS: R5 = 0
RET
FAIL AND R5, R5, #0 ; FAIL: R5 = 1
ADD R5, R5, #1
RET
MAX .FILL xC005 ; Bù 2 của x3FFB, tức -x3FFB
```

và

```
POP LD R1, EMPTY ; EMPTY = -x4000
ADD R2, R6, R1 ; So sánh con trỏ stack
BRz FAIL ; với x4000, xem stack trống hay không
LDR R0, R6, #0
ADD R6, R6, #1
```

```

        AND R5, R5, #0 ; SUCCESS: R5 = 0
        RET
FAIL    AND R5, R5, #0 ; FAIL: R5 = 1
        ADD R5, R5, #1
        RET
EMPTY   .FILL xC000           ; Bù 2 của x4000, tức -x4000

```

Chương trình con sau đây là một minh họa hoàn chỉnh cho việc sử dụng stack, gồm hai thao tác PUSH và POP, với các kiểm tra cần thiết khi không thực hiện được các thao tác này. Stack trong chương trình con này gồm năm phần tử, có địa chỉ từ x3FFF tới x3FFB. Nếu muốn đẩy một trị vào stack, chúng ta chỉ cần nạp trị này vào thanh ghi R0 và thực hiện JSR PUSH. Để lấy một trị ra khỏi stack và cất vào R0, chúng ta thực hiện JSR POP. Nếu muốn thay đổi vị trí hay kích thước stack, chúng ta chỉ cần thay đổi BASE và MAX tương ứng. Các thao tác PUSH và POP, nếu thành công sẽ trả về trị 0 trong thanh ghi R5; nếu thất bại, R5 chứa trị 1.

; cuu duong than cong . com
; Chương trình con thực hiện hàm PUSH và POP. Chương trình này
; làm việc với một stack chứa các ô nhớ từ địa chỉ x3FFF
; (BASE) tới x3FFB (MAX). R6 là stack pointer.

```

; cuu duong than cong . com
POP    ST     R2,Save2      ; Lưu các trị cho POP.
        ST     R1,Save1
        LD     R1,BASE       ; BASE chứa -x3FFF.
        ADD   R1,R1,#-1      ; R1 chứa -x4000.
        ADD   R2,R6,R1       ; So sánh stack pointer với x4000
        BRz  fail_exit      ; Nhảy nếu stack là trống (empty).
        LDR   R0,R6,#0       ; "pop".
        ADD   R6,R6,#1       ; Thay đổi stack pointer
        BRnzp success_exit

PUSH   ST     R2,Save2      ; Lưu trị các register
        ST     R1,Save1      ; dùng trong PUSH.
        LD     R1,MAX         ; MAX chứa -x3FFB
        ADD   R2,R6,R1       ; So sánh stack pointer với -x3FFB
        BRz  fail_exit      ; Nhảy nếu stack là đầy (full).
        ADD   R6,R6,#-1      ; Thay đổi stack pointer
        STR   R0,R6,#0       ; "push"

success_exit

```

```

LD      R1,Save1    ; Khôi phục lại trị ban đầu
LD      R2,Save2    ; cho các register.
AND    R5,R5,#0    ; Thành công, R5 <- 0.
RET

fail_exit
LD      R1,Save1    ; Khôi phục lại trị ban đầu
LD      R2,Save2    ; cho các register.
AND    R5,R5,#0    ; Thành công, R5 <- 0.
ADD    R5,R5,#1    ; Thất bại, R5 <- 1.
RET

BASE   .FILL    xC001    ; BASE chứa -x3FFF.
MAX    .FILL    xC005
Save1  .FILL    x0000
Save2  .FILL    x0000

```

6.3.2 Điều khiển xuất nhập qua ngắt

Như đã trình bày trong mục 6.1.3, ta có hai thành phần để điều khiển xuất nhập qua ngắt, đó là cơ chế tạo ngắt (1) và cơ chế quản lý việc truyền dữ liệu xuất nhập, gọi là cơ chế xử lý ngắt (2). Phần đầu (1), cơ chế tạo ngắt, đã được trình bày trong mục 6.1.3, còn phần thứ hai (2), cơ chế quản lý ngắt, sẽ được trình bày ngay sau đây do nó liên quan tới stack.

Việc truyền dữ liệu xuất nhập có ba giai đoạn: khởi tạo ngắt, phục vụ ngắt, và trở về từ ngắt. Từng giai đoạn sẽ được khao sát ngay sau đây.

1- Khởi tạo ngắt và phục vụ ngắt

Như đã biết, ngắt được khởi tạo do một thiết bị xuất nhập có độ ưu tiên cao hơn chương trình đang chạy và tạo ra một tín hiệu INT. Bộ xử lý kiểm tra sự có mặt của tín hiệu INT mỗi khi nó hoàn tất một chu kỳ lệnh. Nếu không thấy, nó sẽ tiếp tục công việc đang chạy, tức lệnh kế của chương trình đang chạy sẽ được lấy. Nếu việc kiểm tra là có, lệnh kế không được lấy, và thay vào đó là các thao tác chuẩn bị để ngắt chương trình đang chạy và thực thi chương trình phục vụ ngắt liên quan tới nhu cầu của thiết bị xuất nhập đã yêu cầu ngắt. Có hai bước cần được thực hiện: (1) lưu giữ trạng thái chương trình đang chạy phục vụ trở về từ ngắt, (2) nạp trạng thái của chương trình phục vụ ngắt để phục vụ cho yêu cầu ngắt.

a- Trạng thái của một chương trình

Trạng thái của một chương trình là một bức tranh nhanh về các thông tin về tài nguyên mà chương trình tác động tới. Nó bao gồm nội dung các ô nhớ dùng trong chương trình và nội dung các thanh ghi đa dụng. Nó cũng bao gồm hai thanh ghi quan trọng, PC và PSR. Chúng ta đã quen thuộc với thanh ghi PC, còn thanh ghi PSR (*Program Status Register*) chứa các thông tin quan trọng về trạng thái của chương trình đang chạy như hình dưới.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Pr				PL						N	Z		P		

Đặc quyền Priv	Độ ưu tiên Priority	Mã điều kiện cond code
-------------------	------------------------	---------------------------

Bit PSR[15] quy định chương trình đang chạy ở kiểu đặc quyền (Privileged mode) hay không đặc quyền (Unprivileged mode). Với kiểu đặc quyền, chương trình có quyền truy xuất các tài nguyên quan trọng mà chương trình người dùng không sử dụng được. Các bit PSR[10:8] xác định mức độ ưu tiên (PL) hay khẩn cấp của việc thực thi chương trình. Như đã được đề cập trước đây, có tám mức độ ưu tiên từ thấp tới cao là PL0 tới PL7. Sau cùng, các bit PSR[2:0] chứa các mã điều kiện N, Z, P tương ứng từ PSR[2] tới PSR[0].

b- Lưu trạng thái của chương trình bị ngắt

Bước đầu tiên để khởi tạo ngắt là lưu đủ trạng thái của chương trình đang chạy để nó có thể được tiếp tục thực thi ngay sau khi yêu cầu của thiết bị xuất nhập được đáp ứng. Nghĩa là, trong trường hợp LC-3, lưu PC và PSR. Thanh ghi PC phải được lưu vì nó biết lệnh nào cần được thực thi kế tiếp khi chương trình bị ngắt phục hồi việc thực thi. Các mã điều kiện (các cờ N, Z, P) phải được lưu vì chúng có thể được các lệnh *rẽ nhánh* trong chương trình bị ngắt sau này sử dụng khi chương trình này được phục hồi thực thi. Mức độ ưu tiên của chương trình bị ngắt phải được lưu vì nó xác định tính khẩn cấp của chương trình này so với tất cả các chương trình khác. Khi chương trình bị ngắt phục hồi thực thi, việc biết được chương trình độ ưu tiên nào có thể ngắt nó trở lại và chương trình nào không thể ngắt nó là rất quan trọng. Sau cùng, mức độ đặc quyền của chương trình cũng

phải được lưu vì nó chứa thông tin về tài nguyên của bộ xử lý mà chương trình bị ngắt có thể và không thể truy xuất.

Việc lưu các thanh ghi đa dụng là không cần thiết vì chúng ta chấp nhận rằng chương trình con phục vụ ngắt sẽ lưu nội dung của thanh ghi đa dụng mà nó cần trước khi dùng, và sẽ phục hồi lại trị ban đầu cho thanh ghi trước khi quay lại chương trình bị ngắt.

LC-3 sao lưu tất cả thông tin trạng thái này vào một stack đặc biệt, được gọi là Supervisor Stack. Stack này chỉ được các chương trình thực thi ở kiểu đặc quyền sử dụng. Một vùng của bộ nhớ được dành riêng cho mục đích này. Stack này tách biệt với User Stack, được truy xuất bởi chương trình người dùng. Các chương trình truy xuất cả hai stack bằng thanh ghi R6 như là stack pointer. Khi truy xuất Supervisor stack, R6 là Supervisor stack pointer. Khi truy xuất User stack, R6 là User stack pointer. Hai thanh ghi trong, Saved.SSP và Saved.USP, được dùng để sao lưu stack pointer chưa được dùng tới. Khi kiểu đặc quyền chuyển từ user (tức R6 đang là User stack pointer) sang supervisor, nội dung thanh ghi R6 sẽ được sao lưu trong Saved.USP, và R6 được nạp bằng nội dung của Saved.SSP trước khi việc xử lý bắt đầu. Nghĩa là, trước khi chương trình phục vụ ngắt bắt đầu chạy, R6 được nạp bằng nội dung của Saved.SSP. Rồi PC và PSR của chương trình bị ngắt được đẩy vào Supervisor stack, nơi đó chúng được giữ nguyên vị trí trong khi chương trình phục vụ ngắt thực thi.

c- Nạp trạng thái của chương trình phục vụ ngắt

Một khi trạng thái của chương trình bị ngắt đã được lưu an toàn vào Supervisor stack, bước thứ hai là nạp PC và PSR của chương trình phục vụ ngắt. Các chương trình phục vụ ngắt tương tự như các chương trình phục vụ “Trap” mà ta đã biết. Chúng là các phân khúc chương trình được lưu trong bộ nhớ ở những vị trí được sắp xếp trước. Chúng phục vụ các yêu cầu ngắt.

Tất cả các bộ xử lý đều dùng cơ chế ngắt theo vector (*vectored interrupt*), cũng tương tự như trap vector trong lệnh TRAP. Trong trường hợp ngắt, vector 8 bit được thiết bị yêu cầu ngắt bộ xử lý cung cấp. Nghĩa là, thiết bị xuất nhập chuyển cho bộ xử lý một vector ngắt 8 bit cùng với tín hiệu yêu cầu ngắt và độ ưu tiên của nó. Vector ngắt tương ứng với yêu cầu ngắt độ ưu tiên cao nhất được bộ xử lý sử dụng.

Nó được đặt tên là INTV. Nếu ngắt này được sử dụng, bộ xử lý mở rộng vector ngắt 8 bit INTV thành 16 bit địa chỉ, chính là đầu vào trong bảng vector ngắt. Như trong mục 6.2 có đề cập, bảng vector Trap gồm các ô nhớ từ x0000 tới x00FF, mỗi cái chứa địa chỉ bắt đầu của thủ tục phục vụ “trap”. Bảng vector ngắt gồm các ô nhớ từ địa chỉ x0100 tới x01FF, mỗi cái chứa địa chỉ bắt đầu của thủ tục phục vụ ngắt tương ứng. Bộ xử lý nạp thanh ghi PC bằng nội dung của địa chỉ hình thành bởi sự mở rộng vector ngắt INTV.

Thanh ghi trạng thái PSR được nạp theo quy tắc sau: khi chưa có lệnh nào trong thủ tục phục vụ ngắt thực thi, PSR[2:0] được khởi tạo bằng các bit 0. Vì thủ tục phục vụ ngắt chạy trong kiểu đặc quyền, nên PSR[15] được đặt trị 0. PSR[10:8] được đặt trị là mức ưu tiên liên quan tới yêu cầu ngắt. Sau tất cả các thao tác này, thủ tục phục vụ ngắt đã có thể sẵn sàng để chạy.

d. Phục vụ ngắt

Vì thanh ghi PC chứa địa chỉ bắt đầu của thủ tục phục vụ ngắt, thủ tục phục vụ sẽ được thực thi, và các yêu cầu của thiết bị xuất nhập sẽ được thực hiện.

Ví dụ, bàn phím LC-3 có thể ngắt bộ xử lý mỗi khi có một phím được ấn. Vector ngắt bàn phím sẽ cho biết chương trình điều khiển cần gọi. Chương trình điều khiển sẽ chép nội dung của thanh ghi dữ liệu vào ô nhớ nào đó được quy định trước.

2. Trở về từ ngắt

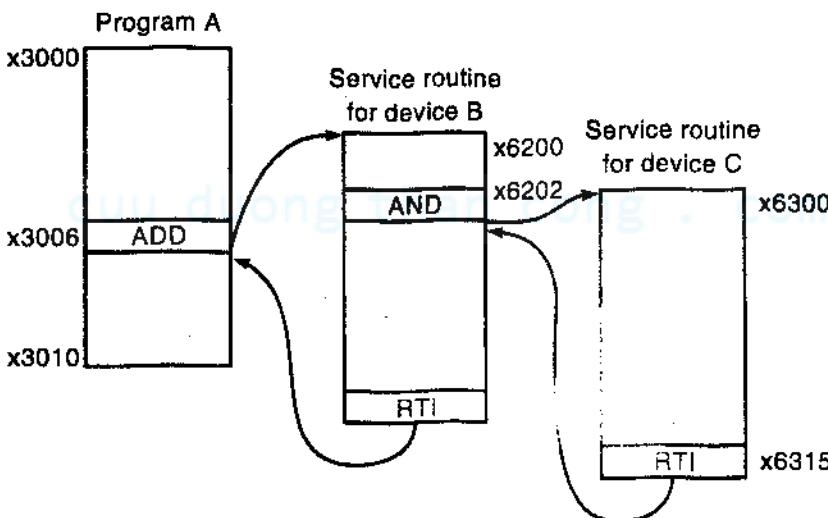
Lệnh sau cùng trong mỗi thủ tục phục vụ ngắt là RTI, trở về từ ngắt. Sau cùng, trước khi bộ xử lý truy xuất lệnh RTI, tất cả những yêu cầu của thiết bị xuất nhập đã được đáp ứng. Việc thực thi lệnh RTI (mã thao tác =1000) bao gồm việc pop PSR và PC từ Supervisor Stack, nơi chúng đang được lưu giữ, và nạp đúng chỗ trở lại vào các nơi trong bộ xử lý. Các mã điều kiện được phục hồi lại trị như lúc chương trình bị ngắt, trong trường hợp chúng được một lệnh rẽ nhánh BR nào đó ở dưới trong chương trình cần đến. PSR[15] và PSR[10:8] phản ảnh mức đặc quyền và độ ưu tiên của chương trình vừa mới được phục hồi chạy. Tương tự, PC được phục hồi trở lại, là địa chỉ của lệnh đáng lẽ đã được thực thi kế tiếp nếu chương trình không bị ngắt.

Với tất cả những điều này khi ngắt xảy ra, chương trình có thể phục hồi chạy trở lại như không có gì xảy ra.

6.3.3 Ví dụ

Chúng ta hãy khảo sát ví dụ sau đây để hiểu rõ hơn về xuất nhập qua ngắt.

Giả sử trong khi chương trình A đang chạy thì thiết bị B, có độ ưu tiên cao hơn A, yêu cầu phục vụ. Trong quá trình thực thi thủ tục phục vụ cho thiết bị xuất nhập B, hãy còn thiết bị C khẩn cấp hơn yêu cầu phục vụ. Hình 6.15 trình bày quá trình này thực hiện ví dụ trên.

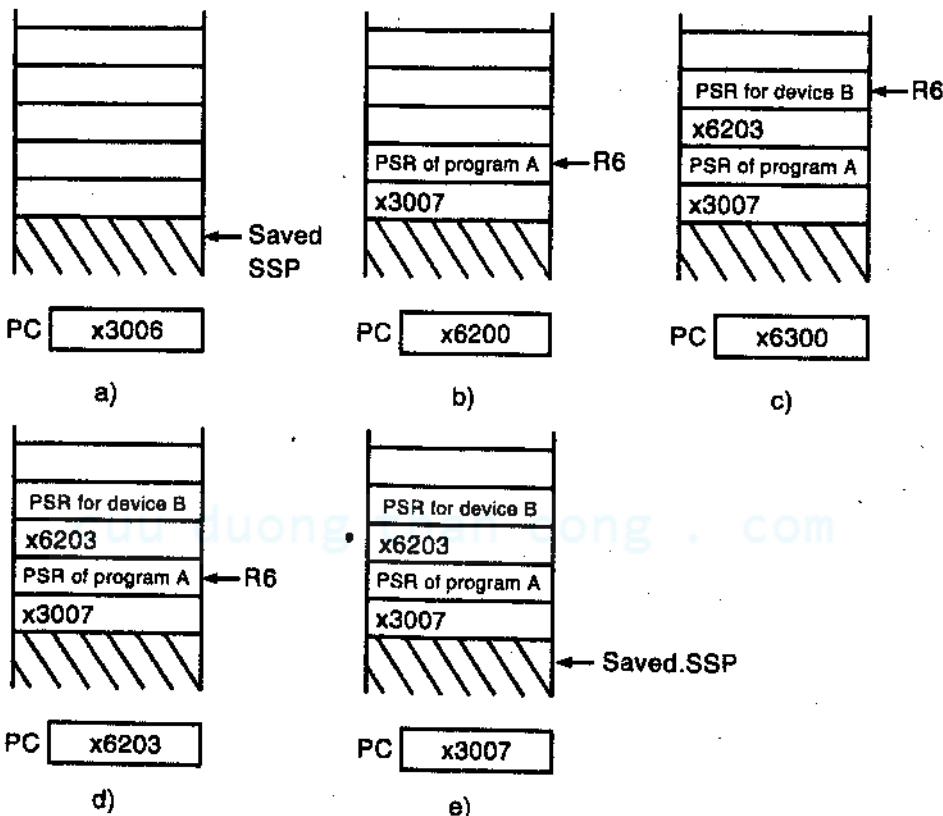


Hình 6.15 Quá trình thực hiện xuất nhập qua ngắt

Chương trình A gồm các lệnh ở ô nhớ từ x3000 tới x3010 và nó đang trong quá trình thực thi lệnh ADD ở x3006. Lúc này thiết bị B gửi tín hiệu yêu cầu ngắt kèm theo vector ngắt xF1, làm tín hiệu ngắt INT được xác nhận.

Thủ tục phục vụ ngắt cho thiết bị B được chứa ở địa chỉ từ x6200 tới x6210; ô nhớ x6210 chứa lệnh RTI. Giả sử khi thủ tục phục vụ ngắt cho thiết bị B này đang thực hiện lệnh AND ở địa chỉ x6202 thì thiết bị C gửi tín hiệu yêu cầu ngắt kèm theo vector ngắt xF2. Vì yêu cầu ngắt liên quan tới thiết bị C có độ ưu tiên cao hơn của thiết bị B, tín hiệu ngắt INT lại được xác lập.

Thủ tục phục vụ ngắt cho thiết bị C được chứa trong bộ nhớ từ x6300 tới x6315; ô nhớ x6315 chứa lệnh RTI. Chúng ta hãy khảo sát thứ tự thực hiện của bộ xử lý. Hình 6.16 trình bày nội dung của Supervisor Stack và thanh ghi PC trong khi thực hiện ví dụ này.



Hình 6.16 Các bức chụp nhanh về nội dung của Supervisor Stack và PC khi điều khiển xuất nhập bằng ngắt

Bộ xử lý thực hiện như sau: Hình 6.16a cho hình ảnh của Supervisor Stack và PC trước khi chương trình A lấy lệnh ở x3006. Chú ý lúc này ngắt chưa xảy ra, nên stack pointer R6 đang chỉ tới nội dung hiện thời trong User Stack. Còn Saved.SSP đang chứa địa chỉ của Supervisor Stack còn chưa được sử dụng. Tín hiệu INT, gây ra bởi một ngắt từ thiết bị B, xuất hiện vào cuối quá trình thực thi lệnh ở x3006. Vì trạng thái của chương trình A phải được lưu lại trong Supervisor Stack, nên bước đầu tiên cần làm là khởi động việc sử dụng Supervisor Stack. Điều này được thực hiện bằng việc chép lưu R6 vào thanh ghi Saved.USP, và nạp R6 bằng nội dung của thanh ghi Saved.SSP. Địa chỉ

x3007, đang được chứa trong PC, tức địa chỉ lệnh kế tiếp trong chương trình A, được đẩy vào stack (tức *Supervisor Stack*). Thanh ghi PSR của chương trình A, bao gồm các mã điều kiện của tạo bởi lệnh ADD, cũng được lưu vào stack. Vector ngắt liên quan tới thiết B được mở rộng ra 16 bit để có x01F1, và nội dung của ô nhớ x01F1, tức x6200, được nạp vào thanh ghi PC. Hình 6.16d cho thấy stack và PC lúc này.

Thủ tục ngắt cho thiết bị B thực hiện cho tới khi một ngắt độ ưu tiên cao hơn được tìm thấy ở cuối quá trình thực thi lệnh ở x6202. Địa chỉ x6203 được đưa vào stack cùng với PSR của thủ tục phục vụ cho thiết bị B, vốn đang chứa các mã điều kiện tạo bởi lệnh AND. Vector ngắt liên quan tới thiết bị C được mở rộng ra để có 16 bit x01F2, và nội dung của x01F2, là x6300, được nạp vào thanh ghi PC. Hình 6.16c cho thấy stack và PC lúc này.

Thủ tục phục vụ ngắt cho thiết bị C thực thi tới lúc hoàn tất, tức tới lệnh ở x6315. Sau đó, Supervisor Stack được lấy ra hai lần, để khôi phục lại PSR của thủ tục phục vụ cho thiết bị B, bao gồm các mã điều khiển tạo bởi lệnh AND ở x6202, và khôi phục lại PC chứa x6203. Hình 6.16d cho thấy điều này.

Thủ tục phục vụ ngắt cho thiết bị B khôi phục thực hiện từ x6203 và chạy tới khi hoàn thành, tức tới lệnh RTI ở x6210. Supervisor Stack tiếp tục được lấy ra hai lần nữa nhằm khôi phục lại PSR của chương trình A, vốn chứa các mã điều kiện tạo bởi lệnh ADD ở x3006, và khôi phục lại PC chứa x3007. Sau cùng, vì chương trình A chạy trong User mode, mà R6 đang là stack pointer tới Supervisor Stack, nên nội dung của R6 cần được lưu vào thanh ghi Saved.SSP để R6 được nạp trở lại từ nội dung của thanh ghi Saved.USP. Hình 6.16e cho thấy điều này. Sau cùng, chương trình A khôi phục thực hiện và chạy tới lệnh kết thúc ở x3007.

Để hiểu rõ hơn vấn đề, độc giả có thể tự giải thích ví dụ sau đây. Trong ví dụ này, chương trình sẽ nhập một chuỗi và đổi chuỗi từ ký tự thường sang ký tự hoa bằng C và hợp ngữ LC-3.

Ví dụ 6.11 Chương trình bằng C

```
#include <stdio.h>
void UpcaseString(char inputString[]);
main ()
```

```

{
    char string[8];
    scanf("%s", string);
    UpcaseString(string);
}

void UpcaseString(char inputString[])
{
    int i = 0;

    while(inputString[i])
    {
        if (('a' <= inputString[i]) && (inputString[i] <= 'z'))
            inputString[i] = inputString[i] - ('a' - 'A');
        i++;
    }
}

```

. cuu duong than cong . com

Chương trình bằng LC-3:

; Chương trình chuyển từ chuỗi ký tự thường sang chuỗi ký tự hoa

	.ORIG	x3000
	LEA	R6, STACK
MAIN	ADD	R1, R6, #3
READCHAR	IN	; nhập chuỗi: scanf
	OUT	
	STR	R0, R1, #0
	ADD	R1, R1, #1
	ADD	R2, R0, x-A
	BRnp	READCHAR
	ADD	R1, R1, #-1
	STR	R2, R1, #0 ; ký tự NULL đánh dấu cuối chuỗi
	ADD	R1, R6, #3 ; lấy ký tự đầu của chuỗi
	STR	R1, R6, #14 ; truyền tham số
	STR	R6, R6, #13

	ADD	R6, R6, #11	
	JSR	UPPERCASE	
	HALT		
UPPERCASE	STR	R7, R6, #1	
	AND	R1, R1, #0	
	STR	R1, R6, #4	
	LDR	R2, R6, #3	
CONVERT	ADD	R3, R1, R2	; thêm chỉ số tới địa chỉ đầu của chuỗi
	LDR	R4, R3, #0	
	BRz	DONE	; Xong, nếu tới ký tự NULL
	LD	R5, a	
	ADD	R5, R5, R4	; so sánh nếu là 'a'
	BRn	NEXT	
	LD	R5, z	
	ADD	R5, R4, R5	; nếu là 'z'
	BRp	NEXT	
	LD	R5, asubA	; chuyển sang hoa
	ADD	R4, R4, R5	
	STR	R4, R3, #0	
NEXT	ADD	R1, R1, #1	; tăng chỉ số tới mảng, i
	STR	R1, R6, #4	
	BRnzp	CONVERT	
DONE	LDR	R7, R6, #1	
	LDR	R6, R6, #2	
	RET		
a	.FILL	#-97	
z	.FILL	#-122	
asubA	.FILL	#-32	
STACK	.BLKW	100	
	.END		

Độc giả có thể chỉnh sửa lại chương trình trên bằng cách thêm vào đầy đủ các khai báo stack, thủ tục thao tác trên stack như đã trình bày để hiểu rõ vấn đề hơn.

BÀI TẬP CUỐI CHƯƠNG

- 6.1 Mục đích của bit[15] trong thanh ghi KBSR là gì?
- 6.2 Chuyện gì xảy ra khi một chương trình không kiểm tra Ready bit của KBSR trước khi đọc KBDR?
- 6.3 Viết chương trình kiểm tra trị ban đầu trong ô nhớ x4000 để xem nó là một mã ASCII hợp lệ hay không. Nếu đúng, in ra ký tự. Nếu không, chương trình không in ra gì cả.
- 6.4 Chuyện gì xảy ra khi phần cứng bàn phím không kiểm tra KBSR trước khi ghi vào KBDR?
- 6.5 Chuyện gì xảy ra khi phần cứng màn hình không kiểm tra DSR trước khi ghi vào DDR?
- 6.6 Chương trình LC-3 sau đây làm gì?

```

        .ORIG x3000
        LD    R0, ASCII
        LD    R1, NEG
AGAIN   LDI    R2, DSR
        BRzp AGAIN
        STI    R0, DDR
        ADD   R0, R0, #1
        ADD   R2, R0, #1
        BRnp AGAIN
        HALT
ASCII   .FILL  x0041
NEG    .FILL  xFFB6 ; -x004A
DSR    .FILL  xFE04
DDR    .FILL  xFE06

```

- 6.7 Liệt kê các ưu điểm của thao tác xuất nhập bằng thủ tục TRAP thay vì chúng ta phải tự viết một thủ tục để mỗi khi cần chương trình của chúng ta có thể sử dụng để thực hiện xuất nhập.

6.8 Xét chương trình hợp ngữ LC-3 sau đây

```

.ORIG      x3000
L1        LEA      R1, L1
          AND     R2, R2, x0
          ADD     R2, R2, x2
          LD      R3, P1
L2        LDR      R0, L1, xC
          OUT
          ADD     R3, R3, #-1
          BRz    GLUE
          ADD     R1, R1, R2
          BR      L2
GLUE      HALT
P1        .FILL   xB
.STRINGZ  "Hello, World!"
.END

```

- a. Sau khi chương trình này được dịch và nạp vào bộ nhớ để chạy, mã nhị phân nào được chứa vào ô nhớ x3005?
- b. Lệnh nào (cung cấp địa chỉ ô nhớ) được thực hiện sau khi lệnh ở x3005 được thực thi?
- c. Lệnh nào (cung cấp địa chỉ ô nhớ) được thực hiện trước khi lệnh ở x3006 được thực thi?
- d. Xuất liệu của chương trình này là gì?
- 6.9** Hãy cho biết sự khác nhau khi sử dụng lệnh RET và RTI khi thiết kế chương trình con và chương trình phục vụ ngắt.
- 6.10** Viết chương trình với bốn chương trình con tính tổng, hiệu, tích, và thương của hai số từ 0 - 9 nhập vào từ bàn phím. In kết quả ra màn hình.
- 6.11** Hãy viết lại thủ tục PUSH và POP để thao tác trên stack có kích thước tùy ý.
- 6.12** Chương trình LC-3 sau đây làm gì?

```

.ORIG x3000
LEA R6, STACKBASE
LEA R0, PROMPT
TRAP x22 ; PUTS
AND R1, R1, #0
LOOP TRAP x20 ; IN
TRAP x21
ADD R3, R0, #-10 ; Check for newline
BRz INPUTDONE
JSR PUSH
ADD R1, R1, #1
BRnzp LOOP
INPUTDONE ADD R1, R1, #0 ; Check for newline
BRz DONE
LOOP2 JSR POP
TRAP x21
ADD R1, R1, #-1
BRp LOOP2
DONE TRAP x25 ; HALT

PUSH ADD R6, R6, #-2
STR R0, R6, #0
RET

POP LDR R0, R6, #0
ADD R6, R6, #2
RET

PROMPT .STRINGZ "Mời nhập một câu:"
STACKSPAC .BLKW #50
STACKBASE .FILL #50
.END

```

GIỚI THIỆU NGÔN NGỮ LẬP TRÌNH C

7.1 GIỚI THIỆU

7.1.1 Sơ lược lịch sử của ngôn ngữ C

Lịch sử phát triển của ngôn ngữ C gắn liền với sự phát triển của hệ điều hành UNIX. Vào năm 1969, UNIX được phát triển và được cài đặt vào máy DEC PDP-7 ở Bell Laboratories, New Jersey (Hoa Kỳ), nó được viết hoàn toàn bằng hợp ngữ PDP-7. Hệ điều hành UNIX đã tỏ ra có nhiều ưu điểm như thân thiện với người sử dụng, cung cấp nhiều công cụ giúp đỡ hữu hiệu và đặc biệt là môi trường mở để mọi người có thể cài đặt và viết chương trình trên nó một cách dễ dàng. Sau khi UNIX đã được phát triển và vận hành thì Ken Thompson đã đưa ra một bộ dịch của một ngôn ngữ mới, gọi là B. B chịu ảnh hưởng sâu sắc của BCPL, là ngôn ngữ chuyên dùng để thiết kế phần mềm hệ thống. Sau đó máy DEC PDP-11 ra đời, việc sử dụng UNIX trên máy mới cho thấy có nhiều trực trặc, do đó nhu cầu thiết kế lại UNIX lại được đặt ra và ngôn ngữ được chọn là B. Tuy nhiên, B lúc này lại không hoàn toàn phù hợp với máy PDP-11 (vì một số lý do như B là ngôn ngữ làm việc theo word trong khi PDP-11 làm việc theo byte,...), do đó nhu cầu cải tiến B xuất hiện vào 1971 và ngôn ngữ C ra đời từ sự cải tiến này, để rồi sau đó C đã được sử dụng để viết trở lại hệ điều hành UNIX.

Sau khi máy vi tính ra đời, C nhanh chóng được cài đặt, sử dụng trên máy vi tính và trở thành một ngôn ngữ lập trình khá mạnh, hiện nay đang có khuynh hướng trở thành ngôn ngữ chính cho máy vi tính trên thế giới. C vẫn còn đang được phát triển và đã được tiêu chuẩn hóa bởi ANSI từ năm 1983.

Như vậy, C là một ngôn ngữ lập trình đa dụng, cấp cao, nhưng lại có những khả năng thực hiện những thao tác khá sâu như của hợp ngữ (*assembly language*). Chính nhờ tính tổng quát và sự linh hoạt đó mà C được xem là ngôn ngữ lập trình chuyên nghiệp rất hiệu quả và tiện lợi. Chính vì thế mà nó thường được gọi là ngôn ngữ cấp trung gian (*Middle level language*).

7.1.2 Đặc điểm của ngôn ngữ C

C là ngôn ngữ không nhạy kiểu. Khi các biến, hằng được khai báo theo các kiểu dữ liệu nào đó của C, thì chúng có thể nhận được trị không cùng kiểu với kiểu mà biến, hằng đã được khai báo.

C có nhiều kiểu dữ liệu phong phú, với nhiều kiểu số nguyên và số thực. Ngoài ra, C còn cho phép người lập trình tự xây dựng những kiểu dữ liệu khác tùy theo yêu cầu của mình.

C có các phép toán đặc biệt cho phép lập trình viên thực hiện thao tác lệnh hiệu quả nhất. Hiệu quả đó có được do C có nhiều toán tử khá gần với các lệnh của ngôn ngữ máy. Ngoài ra, C còn cung cấp các toán tử xử lý đến từng bit, byte, đến cả địa chỉ của bộ nhớ.

C có các lệnh điều khiển và vòng lặp rất thoáng và khá logic và phù hợp với phương pháp lập trình có cấu trúc.

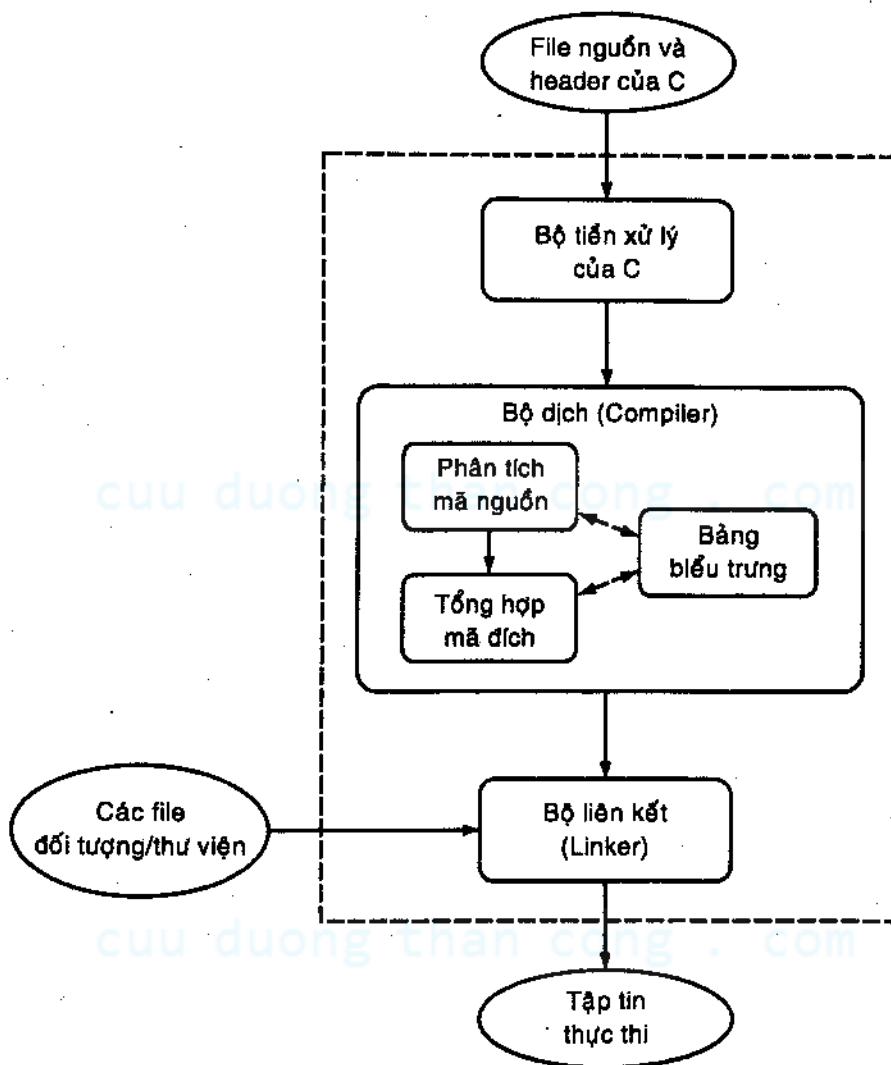
C cho phép khai báo các pointer chỉ tới các biến và hàm, đặc biệt là dùng pointer để quản lý biến động, điều này làm cho một chương trình C rất linh động trong việc khai báo, sử dụng biến và hàm.

C không cung cấp trực tiếp các thao tác đặc biệt như xuất nhập các thiết bị ngoại vi, xử lý chuỗi, mảng. Các thao tác này chỉ được cung cấp dưới dạng những *hàm thư viện*.

7.2 BỘ DỊCH NGÔN NGỮ LẬP TRÌNH C

Bộ dịch (*Compiler*) C là thành phần tiêu biểu trong quá trình dịch từ một chương trình nguồn C sang bản thực thi của nó. Tập tin thực thi này là dạng biểu diễn mã máy của chương trình nên nó sẵn sàng được nạp vào bộ nhớ để thực thi. Toàn bộ quá trình dịch bao gồm bộ tiền xử lý (*Preprocessor*), bản thân bộ biên dịch (*Compiler*), và bộ liên kết (*Linker*). Thông thường, cơ chế dịch được xem như là

quá trình biên dịch vì khi chúng ta dùng bộ biên dịch C, bộ tiền xử lý và bộ liên kết thường được tự động thực hiện. Hình 7.1 cho thấy các thành phần khác nhau trong quá trình biên dịch. Có ba thành phần chính mà chúng ta cần khảo sát kỹ hơn là bộ tiền xử lý, bộ biên dịch, và bộ liên kết.



Hình 7.1 Quá trình dịch tổng quát

- **Bộ tiền xử lý (preprocessor)**

Theo như tên của nó, bộ tiền xử lý C xử lý chương trình C một số việc trước khi đưa nó cho bộ biên dịch. Nó quét qua tất cả các tập

tin nguồn của chương trình C để tìm và thực hiện các hướng dẫn dịch (*directives*). Các hướng dẫn dịch này tương tự như các mã giả trong hợp ngữ LC-3. Chúng chỉ thị bộ tiền xử lý chuyển tập tin nguồn C sang thành dạng điều khiển nào đó. Ví dụ, chúng ta có thể ra lệnh cho bộ tiền xử lý thay thế chuỗi ký tự **DO_DAI_TOI_DA** bằng chuỗi **30**, hay điều khiển nó chèn vào tập tin nguồn chương trình C nội dung của cả tập tin **stdio.h** ở tại vị trí có dòng

```
#include <stdio.h>
```

Chúng ta sẽ thấy các thao tác này rất có lợi khi viết C trong các chương sau. Lưu ý là tất cả các hướng dẫn dịch đều bắt đầu bằng dấu #.

- **Bộ biên dịch (Compiler)**

Sau khi bộ tiền xử lý chuyển đổi tập tin chương trình nguồn xong, lúc này chương trình đã sẵn sàng để được biên dịch. Bộ biên dịch chuyển chương trình đã được tiền xử lý thành một modul đối tượng (*object modul*). Như trong mục 5.4.2 đã đề cập, một tập tin đối tượng là mã máy của một phần của chương trình tổng thể. Có hai công đoạn chính trong quá trình dịch: (1) phân tích, trong đó chương trình nguồn được chia ra làm các phần tử cấu thành theo cú pháp, và (2) tổng hợp, trong đó dạng mã máy của chương trình được tạo ra. Công việc của công đoạn phân tích là đọc vào, phân tích cú pháp, và tạo ra một dạng biểu diễn nội của chương trình ban đầu. Công đoạn tổng hợp tạo ra mã máy và, nếu được yêu cầu, sẽ thử tối ưu hóa mã này để nó có thể thực thi nhanh và hiệu quả hơn trên máy tính mà nó sẽ chạy. Mỗi công đoạn này đều được chia ra làm các công đoạn nhỏ cụ thể hơn ở đó các tác vụ xác định, như phân tích cú pháp, định vị thanh ghi, hay liệt kê lệnh, được hoàn tất. Một vài bộ dịch tạo ra mã hợp ngữ và dùng bộ hợp dịch để hoàn tất quá trình dịch sang mã máy.

Một trong những cơ chế theo dõi trong quan trọng nhất mà các bộ dịch dùng trong việc dịch một chương trình là bảng biểu trưng (*symbol table*). Một bảng biểu trưng là cách thức mà bộ biên dịch theo dõi tất cả các tên biểu trưng được người lập trình dùng trong chương trình. Bảng biểu trưng của bộ dịch C hoàn toàn tương tự như bảng biểu trưng được dùng trong bộ hợp dịch LC-3.

• **Bộ liên kết (Linker)**

Bộ liên kết hoạt động ngay sau khi bộ biên dịch dịch tập tin nguồn sang mã đối tượng. Công việc của bộ liên kết là kết nối tất cả các modul đối tượng lại để tạo ra bản thực thi của chương trình. Bản thực thi là một dạng thức khác của chương trình mà có thể được nạp vào bộ nhớ và thực thi bởi phần cứng. Thí dụ, khi chúng ta nhấp vào biểu tượng của bộ duyệt web trên máy tính của chúng ta, tức chúng ta đang ra lệnh cho hệ điều hành đọc bản thực thi của bộ duyệt web từ đĩa cứng, nạp nó vào bộ nhớ, bắt đầu thực thi nó.

Thông thường, các chương trình C đều tin vào các thủ tục thư viện (*library routines*). Các thủ tục thư viện thi hành các tác vụ chung như các thao tác xuất nhập và chúng được chuẩn bị để các lập trình viên phát triển phần mềm hệ thống sử dụng cho các mục đích chung như phát triển hệ điều hành, bộ biên dịch chẳng hạn. Nếu một chương trình dùng một thủ tục thư viện, thì bộ liên kết sẽ tìm mã đối tượng tương ứng với thủ tục đó và kết hợp nó và bản thực thi cuối cùng của chương trình. Quá trình liên kết các đối tượng thư viện này với chúng ta là không mới, nó đã được mô tả trong chương 5, mục 5.3 khi chúng ta nghiên cứu quá trình dịch của LC-3.

7.3 CÁC VÍ DỤ

Để hiểu một cách tổng quát một chương trình C có cấu trúc và hoạt động như thế nào, hãy xét một vài ví dụ để bạn có thể hình dung ngay được thế nào là một chương trình C, qua đó phần nào thấy được những đặc điểm điển hình của ngôn ngữ C. Sau mỗi chương trình ví dụ, là một số phân tích và giải thích sơ lược các điểm quan trọng của chương trình.

7.3.1 Ví dụ 1

Hãy xét chương trình C đơn giản: In ra màn hình câu **Hello, world** và xuống hàng. Đây là một chương trình C rất quen thuộc cho những người mới làm quen với ngôn ngữ C.

```
main()
{
    printf ("Hello, world\n");
}
```

Giải thích

Một chương trình C, bao giờ cũng có một hàm main(), trong hàm này các lệnh, các hàm, các biến... sẽ được gọi sử dụng theo một trình tự hợp logic để giải quyết bài toán. Đối với các hàm khác ta có thể đặt một tên tùy ý cho nó, còn hàm main() lại là một hàm đặc biệt, đây là hàm có tên cố định và được thực hiện đầu tiên khi vào chương trình. Như vậy, mỗi chương trình C đều phải có một hàm main(), các lệnh, các hàm, các biến,... sẽ được đặt bên trong cặp dấu mở/dóng ngoặc nhọn ({ }) như sau:

```
main()
{
    <các biến, hàm, lệnh...>
}
```

Hàm của C có thể chuyển dữ liệu qua đối số của nó. Các đối số này được đặt ở giữa cặp dấu ngoặc () theo sau tên hàm. Trong trường hợp ví dụ trên, hàm main() không có đối số nên chỉ có cặp dấu ngoặc trống mà thôi.

Trong ví dụ trên ta chỉ có một lệnh

```
printf ("Hello, world\n");
```

trong hàm main(), đây là một lệnh gọi hàm: cần nêu tên của hàm được gọi với các đối số là chuỗi cần in, printf() là một hàm, không phải là từ khóa của ngôn ngữ C, mà chỉ là một hàm trong thư viện chuẩn của C, hàm printf () dùng để in những thông báo (được gởi ở đối số của hàm) lên thiết bị ra chuẩn (*standard output*) mà thông thường là màn hình.

Khi xuất dữ liệu xong, hàm printf() không tự động dời con nháy (*cursor*) xuống hàng, con nháy chỉ xuống hàng khi hàm printf thực hiện việc in một ký tự đặc biệt là '\n', đây là chuỗi thoát (*escape sequence*) biểu diễn cho ký tự LF (*linefeed*), là ký tự không thấy được trên màn ảnh (trong bảng mã ASCII, ký tự này có mã số là 10 thập phân). ANSI đã có những quy định về những chuỗi thoát để biểu diễn một số ký tự không thấy được khác trong C.

Một lệnh của C luôn dùng dấu chấm phẩy (;) để kết thúc lệnh. Việc thiếu dấu chấm phẩy này sẽ gây ra một lỗi cho chương trình, do đó cuối lệnh printf (...) là dấu chấm phẩy.

Để kiểm soát các đối số thật đưa vào cho hàm có phù hợp kiểu với các đối số giả mà hàm yêu cầu không, C đã đưa ra khái niệm prototype của hàm. Đối với các hàm chuẩn, prototype nằm trong các file.h mà C đã thiết kế sẵn, do đó khi lập trình viên sử dụng hàm nào trong chương trình thì phải bao hàm prototype của hàm đó vào chương trình bằng lệnh tiền xử lý #include theo cú pháp sau

```
#include <file.h>
```

Do đó, chương trình trên nên được viết lại như sau:

```
#include <stdio.h>
main()
{
    printf ("Hello, world\n");
}
```

Hàm printf được dùng trong hàm main(), prototype của nó nằm trong file stdio.h.

7.3.2 Ví dụ 2

Xét một chương trình khác: nhập hai số từ bàn phím, in ra màn hình kết quả so sánh của hai số đó. Ta có chương trình sau:

```
#include <stdio.h>
#include <conio.h>
main()
{
    double a, b;
    clrscr();
    printf ("Mời nhập hai số nguyên: ");
    scanf ("%lf%lf", &a, &b);
    if (a < b)
        printf ("%5.2lf nhỏ hơn %5.2lf", a, b);
    else if (a == b)
```

```

    printf ("%5.2lf bằng %5.2lf", a, b);
else /* a > b */
    printf ("%5.2lf lớn hơn %5.2lf", a, b);
}

```

Đầu chương trình là hai lệnh include bao gồm hai file stdio.h và conio.h, vì trong hàm main() ta sử dụng hàm chuẩn clrscr(), hàm này có prototype nằm trong file conio.h; còn hai hàm printf(), scanf() có prototype nằm trong file stdio.h, trong đó hàm scanf() dùng để nhập giá trị từ bàn phím vào cho hai biến *a* và *b*. Chú ý, việc nhập trị bằng hàm scanf() vào cho hai biến *a* và *b* đòi hỏi phải nhập theo địa chỉ của biến, vì vậy toán tử lấy địa chỉ của biến "&" trong C đã được sử dụng.

Trong hàm main() lệnh đầu tiên **double a, b;** là lệnh khai báo và định nghĩa các biến *a* và *b*, nó có nghĩa:

- Thông báo rằng có các danh hiệu *a* và *b* được sử dụng trong hàm main().
- Hai biến này được cung cấp vùng nhớ tương ứng với kiểu mà nó được khai báo **double**. C có nhiều dữ liệu số khác nhau, kiểu **double** là một kiểu số thực 64 bit có dấu, tức là một biến thuộc loại này có thể lưu một trị thực nằm giữa tầm dương 1.7×10^{-308} đến 1.7×10^{308} hoặc tầm âm tương ứng.

Lệnh kế tiếp clrscr() là lệnh gọi hàm xóa màn hình, hàm này khi gọi không cần đưa đối số. Lệnh printf() in ra màn hình câu "Mời... nguyên: ", sau đó lệnh scanf() cho phép nhập trị vào cho hai biến *a* và *b*. Khi nhập trị (hoặc khi xuất trị) trị nhập vào phải tuân theo chuỗi định dạng nhập mà hàm scanf đã quy định cho từng kiểu biến khác nhau, trong trường hợp này biến thuộc kiểu **double** thì scanf() quy định chuỗi định dạng nhập là **%lf** (*lf* viết tắt từ *long float*).

Lệnh if... else thực hiện việc kiểm tra và rẽ nhánh thực hiện chương trình, trong chương 4 ta sẽ xét cụ thể lệnh if, còn trong ví dụ này nếu *a* nhỏ hơn *b* (*a* < *b*) là đúng thì lệnh printf ("%lf nhỏ hơn %lf", *a*, *b*); được thực hiện; nếu *a* bằng *b* (*a* == *b*) thì lệnh printf ("%lf bằng %lf", *a*, *b*); được thực hiện; còn nếu *a* lớn hơn *b* (*a* > *b*) thì lệnh printf ("%lf lớn hơn %lf", *a*, *b*); được thực hiện. C cho phép ghi chú

thích nằm giữa cặp dấu /* và */; trong ví dụ trên, lệnh rẽ nhánh **else** sau cùng sẽ là trường hợp a lớn hơn b , do đó để chương trình rõ ràng dòng ghi chú sau được thêm vào sau **else /* a > b */**. Chú ý, khi in trị a , b ra màn hình, chuỗi định dạng xuất %lf lại được sử dụng.

7.3.3 Ví dụ 3

Hãy xét một chương trình khác: In bảng lũy thừa 2 của các số nguyên từ 1 đến 10. Chương trình sau sẽ in ra một bảng có dạng:

1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100

/* In ra bảng bình phương của các số từ 1 đến 10 */

```
#include <stdio.h>
#include <conio.h>

main()
{
    int n, n2;
    clrscr();
    n = 1;
    while (n <= 10)
    {
        n2 = n * n;
        printf ("%2d%5d\n", n, n2);
        n = n + 1;
    }
}
```

Dòng đầu tiên của chương trình này là một lời ghi chú. Những lời ghi chú thường rất có ích cho việc đọc lại, xem và sửa đổi chương trình. Các ghi chú này có thể được viết ở những vị trí nào cho phép đặt dấu cách (dấu khoảng trắng, hoặc dấu xuống hàng).

Lệnh **int n, n2;** là lệnh khai báo, và định nghĩa cho các biến *n* và *n2*; nó có nghĩa:

- Thông báo rằng có các danh hiệu *n* và *n2* được sử dụng trong hàm main().
- Cung cấp vùng nhớ cho hai biến này với kích thước của kiểu nguyên int.

Lệnh **n = 1;** mới thực sự là lệnh thao tác đầu tiên của chương trình, đây là một lệnh gán đơn giản, cho *n* một giá trị ban đầu là 1.

Vì cần phải lặp đi lặp lại việc tính và in ra giá trị bình phương của mỗi trị từ 1 đến 10 ở mỗi hàng của bảng, ta phải dùng một vòng lặp để thực hiện điều đó, lệnh **while** là một lệnh điều khiển của C thực hiện việc lặp. Theo sau while là điều kiện (*n <= 10*) và khối lệnh:

```
{
    n2 = n * n;
    printf ("%2d%5d \n", n, n2);
    n = n + 1;
}
```

Để thực hiện lệnh lặp while, đầu tiên điều kiện (*n <= 10*) sẽ được kiểm tra, nếu đúng (*n* còn nhỏ hơn hoặc bằng 10) khối lệnh tương ứng sẽ được thực hiện (tính *n2*, in kết quả ra màn hình, tăng *n* thêm 1). Sau đó điều kiện lại được kiểm tra lại để quyết định xem có thực hiện khối lệnh tương ứng nữa không. Khi điều kiện sai (*n* lớn hơn 10) thì vòng lặp kết thúc và chương trình sẽ được tiếp tục ở lệnh kế tiếp sau khối lệnh của lệnh while. Chú ý, cặp dấu (()) còn được dùng để gộp một nhóm các lệnh thành một khối lệnh, khối lệnh này có thể được dùng làm thân cho một lệnh điều khiển nào đó (chẳng hạn như while trong ví dụ trên).

BÀI TẬP CUỐI CHƯƠNG

- 7.1 Viết chương trình in ra màn hình hình sau:

```
* * * * * * *  
*           *  
*           *  
* * * * * * *
```

- 7.2 Viết chương trình in ra màn hình các thông tin sau:

Họ tên :

Tuổi :

Nghề nghiệp :

Địa chỉ :

- 7.3 Viết chương trình nhập một số từ bàn phím, kiểm tra nếu số đó lớn hơn hoặc bằng 0 thì tính căn bậc hai của nó, còn nếu số đó nhỏ hơn 0 thì báo lỗi và kết thúc chương trình. Với sqrt là hàm trong C có prototype nằm trong file math.h như sau:

```
double sqrt (double x);
```

- 7.4 Nhập ba số từ bàn phím, in ra màn hình số lớn nhất và số nhỏ nhất trong ba số đó.

- 7.5 Viết lại chương trình ở bài 7.2 ở dạng hợp ngữ LC-3 để so sánh.

- 7.6 Viết lại chương trình ở bài 7.4 ở dạng hợp ngữ LC-3 để so sánh.

cuu duong than cong . com

Chương 8

CÁC THÀNH PHẦN CƠ BẢN VÀ CÁC KIỂU DỮ LIỆU CỦA C

8.1 DANH HIỆU

Danh hiệu là tên của hằng, biến, hàm... hoặc các ký hiệu đã được quy định đặc trưng cho một thao tác nào đó. Danh hiệu có hai loại: ký hiệu và danh hiệu.

Ký hiệu (*symbol*) là các dấu đã được C quy định để biểu diễn cho một thao tác nào đó.

Nếu dùng một dấu để biểu diễn cho một thao tác thì ta có ký hiệu đơn (*single symbol*).

Ví dụ 8.1

dấu + đặc trưng cho phép cộng

dấu – đặc trưng cho phép trừ

dấu * đặc trưng cho phép nhân

dấu / đặc trưng cho phép chia

dấu % đặc trưng cho phép lấy số dư của phép chia nguyên

dấu = đặc trưng cho phép gán

dấu > đặc trưng cho phép so sánh lớn hơn

...

Nếu dùng hai dấu trở lên biểu diễn cho một thao tác thì ta có ký hiệu kép (*compound symbol*).

Ví dụ 8.2

dấu == đặc trưng cho phép so sánh bằng
 dấu /* bắt đầu cho một ghi chú
 dấu */ kết thúc ghi chú
 dấu >= đặc trưng cho phép so sánh lớn hơn hay bằng
 dấu ++ đặc trưng cho phép tăng trị lên một
 dấu && đặc trưng cho phép and luận lý
 ...

Danh hiệu (*Identifier*) là các từ khóa của ngôn ngữ hoặc tên của các hằng, biến, hàm trong C. Danh hiệu bao hàm từ khóa và danh hiệu.

Từ khóa (*keyword*) là các danh hiệu mà C đã định nghĩa sẵn cho lập trình viên sử dụng để thiết kế chương trình, tập các từ khóa của C sẽ được liệt kê trong phần phụ lục.

Ví dụ 8.3

từ khóa *if* là một thành tố tạo ra lệnh if
 từ khóa *for* là một thành tố tạo ra lệnh for
 từ khóa *while* là một thành tố tạo ra lệnh while
 ...

Chú ý rằng C là một ngôn ngữ nhạy cảm với sự phân biệt giữa ký tự hoa và ký tự thường, do đó khi viết While sẽ hoàn toàn phân biệt với while. Các từ khóa của C đều ở dạng chữ thường.

Danh hiệu là tên của các hằng, biến, hàm... Nếu các hằng, biến, hàm... này do C đã khai báo và thiết kế sẵn thì các danh hiệu có được gọi là các danh hiệu chuẩn.

Ví dụ 8.4

danh hiệu *main* là danh hiệu của hàm main()
 danh hiệu *scanf* là danh hiệu của hàm scanf()
 danh hiệu *printf* là danh hiệu của hàm printf()
 ...

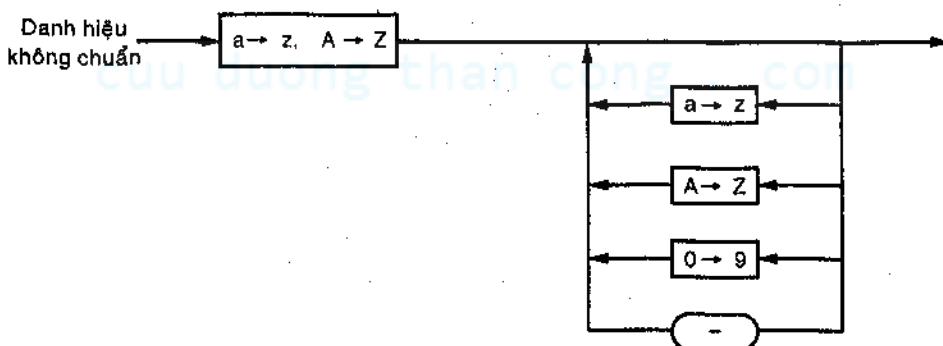
Nếu các hằng, biến, hàm... này do lập trình viên khai báo và định nghĩa trong quá trình thiết kế chương trình thì các danh hiệu đó được gọi là các danh hiệu không chuẩn.

Ví dụ 8.5

Trong ví dụ 2 của chương 7, khi khai báo `double a, b;` thì `a`, `b` là hai danh hiệu không chuẩn.

Như vậy danh hiệu không chuẩn là tên của các hằng, biến, hàm... do lập trình viên tự đặt, do đó nguyên tắc đặt tên của danh hiệu không chuẩn cũng cần phải được nêu cụ thể:

- Danh hiệu không chuẩn không trùng với từ khóa
- Danh hiệu không chuẩn không trùng với danh hiệu chuẩn
- Khi đặt tên cho danh hiệu không chuẩn cần phải theo sơ đồ cú pháp sau:



Chú ý:

- Đối với ô vuông: khi đi ngang qua ta cần phải lấy một phần tử trong nó.
- Đối với ô tròn: khi đi ngang qua ta phải lấy phần tử trong nó.
- Một danh hiệu có thể được bắt đầu bằng dấu gạch dưới

Ví dụ 8.6

Xét các danh hiệu sau:

Main là danh hiệu không chuẩn hợp lệ

- **bắt đầu** là danh hiệu không hợp lệ vì có dấu trừ

2 tháng 9 là danh hiệu không hợp lệ vì có số bắt đầu

kết thúc là danh hiệu không hợp lệ vì có khoảng trắng

Chiều dài một danh hiệu không bị hạn chế, mỗi bộ dịch C sẽ có quy định về chiều dài danh hiệu khác nhau, đối với các bộ dịch C/C++ thì danh hiệu có thể dài tùy ý, tuy nhiên trong các bộ dịch Borland C/C++ có quy định một giá trị xác định số ký tự đầu có nghĩa để phân biệt sự giống nhau và khác nhau giữa hai danh hiệu. Trong Turbo C 2.0, giá trị này là 31, trong Borland C++ 5.02, giá trị này là 55.

Ví dụ 8.7

Xét hai danh hiệu sau

```
kết_thúc_vòng_lặp_in_ra_ký_tự_khoảng_trắng
kết_thúc_vòng_lặp_in_ra_ký_tự_k
```

Hai danh hiệu này có 31 ký tự đầu giống nhau do đó C xem chúng là một, như vậy việc gán trị và tính toán sẽ diễn ra nhầm lẫn giữa hai biến này.

8.2 CÁC KIỂU DỮ LIỆU CHUẨN CỦA C

C có bốn kiểu dữ liệu chuẩn: char, int, float và double, mỗi kiểu sẽ có yêu cầu về bộ nhớ và tầm trị như sau:

Kiểu	Kích thước	Tầm trị biểu diễn
char	8 bit	-128 .. + 127
int	16 bit (hoặc 32 bit)	-32768 .. + 32767 (16 bit) -2147483648 .. +2147483647 (32 bit)
float	32 bit	3.4E38 .. 3.4E+38
double	64 bit	1.7E308.. 1.7E+308

1- Kiểu char

char là kiểu nguyên một byte, kiểu này có thể được sử dụng để khai báo biến, biến đó sẽ chiếm kích thước trong bộ nhớ là 1 byte và có thể giữ một ký tự hoặc một giá trị 8 bit. Mỗi bộ dịch C sẽ có quy định khác nhau về tầm trị của kiểu **char**, đối với bộ dịch TURBO C VERSION 2.0 và Borland C++5.0x kiểu char là kiểu có dấu. Xét hai chương trình ví dụ sau:

Ví dụ 8.8 Biến kiểu char lưu trữ hằng ký tự

```
#include <stdio.h>
main()
{
    char c;
    c = 'a';
    printf ("Ký tự trong biến c là %c ", c);
}
```

Ví dụ 8.9 Biến kiểu char lưu trữ số nguyên

```
#include <stdio.h>
main()
{
    char c;
    c = 89;
    printf ("Trị trong biến c là %d ", c);
}
```

Tùy lập trình viên sử dụng mà biến kiểu char có thể lưu trữ ký tự hoặc trị số nguyên. Trong bộ nhớ, dữ liệu luôn được lưu giữ theo dạng nhị phân và có thể được sử dụng để tính toán như số nguyên bình thường. Tùy cách định dạng xuất nhập mà dữ liệu sẽ hiện ra màn hình dưới dạng ký tự hay số nguyên.

2- Kiểu int

Kiểu int là một kiểu số nguyên, kiểu này có thể được sử dụng để khai báo biến, biến đó có kích thước trong bộ nhớ là kích thước của số nguyên mà máy quy định, đối với máy PC, là các máy vi tính đang phổ biến hiện nay, và bộ dịch Borland C/C++ thì chiều dài của kiểu int là 16/32 bit có dấu, như vậy một biến hay hằng thuộc kiểu này có tầm trị biểu diễn từ -32768 đến 32767 (tức từ -2^{15} đến $2^{15} - 1$) với kiểu 16 bit, từ -2147.483648 đến $-2.147.483.647$ với kiểu 32 bit.

Xét chương trình ví dụ sau

Ví dụ 8.10

```
#include <stdio.h>
main()
{
    int i;
    i = 1234;
    i = i + 123;
    printf ("Trị trong biến i là %d ", i);
}
```

Có thể dùng chuỗi định dạng xuất nhập "%d" để xuất hay nhập trị cho hằng, biến hoặc biểu thức kiểu int.

3- Kiểu float và double

float là kiểu số thực dấu chấm động, có độ chính xác đơn (7 ký số sau dấu chấm thập phân), **double** là kiểu số thực, dấu chấm động, có độ chính xác kép (15 ký số sau dấu chấm thập phân). Khi dùng các kiểu này khai báo cho biến thì biến có thể lưu trị là số thực mà tầm trị đã được cho trong bảng trên.

Ví dụ 8.11

```
float a;
double b;
```

Kiểu **double** còn có thể được khai báo là **long float**, do đó khi khai báo **double b**; thì cũng hoàn toàn tương đương với **long float b**;

Để xuất nhập cho hằng, biến, biểu thức **float** chuỗi định dạng được sử dụng là "%f" đối với kiểu **double** thì chuỗi định dạng là "%lf" cho các hàm **printf** và **scanf**.

Ví dụ 8.12 Viết chương trình nhập hai số thực lớn hơn 0 bất kỳ, sau đó tính lũy thừa của số đầu đối với số sau. Yêu cầu có kiểm tra và báo lỗi.

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
main()
```

```

{
    double x, y, luy_thua;
    clrscr();
    printf ("Moi nhap 2 so:");
    scanf ("%lf %lf", &x, &y);
    if (x < 0 && (y - (int)y != 0))
        printf ("Ban da nhap sai tri");
    else
    {
        luy_thua = pow (x, y);
        printf ("Luy thua cua %5.2lf voi %5. 2lf la %5.2lf", x, y, luy_thua);
    }
}

```

Trong ví dụ trên ta sử dụng hàm **pow** để tính lũy thừa, hàm này có prototype trong file math.h. Chú ý hàm **pow** không tính được lũy thừa x^y khi $x < 0$ và y không phải là số nguyên (*whole number*).

Ngoài ra, ANSI còn đưa thêm một kiểu dữ liệu nữa là **void**. Đây là kiểu không trị, chỉ dùng để biểu diễn kết quả trả về của hàm và khai báo pointer không trỏ đến một kiểu dữ liệu xác định nào cả. Kiểu này sẽ được nói chi tiết hơn ở các phần sau.

Để bổ sung cho bốn kiểu dữ liệu cơ bản, C còn đưa ra các dạng bổ sung **signed**, **unsigned**, **short**, **long** để kết hợp với bốn kiểu trên tạo ra thêm các kiểu mở rộng, ý nghĩa của các dạng này như sau:

- **signed** dùng để xác định kiểu khai báo là kiểu có dấu.
- **unsigned** dùng để xác định kiểu khai báo là kiểu không dấu.
- **short** dùng để xác định kiểu khai báo là kiểu ngắn của kiểu cơ bản.
- **long** dùng để xác định kiểu khai báo là kiểu dài của kiểu cơ bản.

Khả năng kết hợp giữa các kiểu cơ bản và các dạng mở rộng kiểu được cho trong bảng sau:

Bảng 8.1 Sự kết hợp giữa các kiểu dữ liệu chuẩn và các dạng mở rộng kiểu

Dạng Kiểu	signed	unsigned	short	long
char	signed char → char	unsigned char	x	x
int	signed int → int	unsigned int → unsigned	short int → int/short	long int → long
float	x	x	x	long float → double
double	x	x	x	long double

Có thể hiểu:

- **short int:** kiểu số nguyên ngắn.
- **long int:** kiểu số nguyên dài gấp đôi short int.
- **unsigned int, unsigned char:** kiểu số nguyên hoặc ký tự không dấu (luôn biểu diễn trị dương).
- **signed int, signed char:** kiểu số nguyên hoặc ký tự có dấu (có thể âm hoặc dương, bit cao nhất được dùng làm bit dấu).
- **long double:** kiểu số thực có độ chính xác hơn số double (19 ký số sau dấu chấm thập).

Hơn nữa **signed** và **unsigned** còn có thể kết hợp với **short int** và **long int** để cho thêm các kiểu sau:

signed short int → short int

signed long int → long

unsigned short int → unsigned short

unsigned long int → unsigned long

Chú ý rằng trong bộ nhớ máy tính, các giá trị đều được lưu trữ dưới dạng mã nhị phân có nhiều bit (8, 16 hoặc 32 bit tùy theo kiểu của biến hoặc giá trị), trong đó số thứ tự của các bit được đánh số từ phải sang trái bắt đầu từ 0, số hiệu này được gọi là **vị trí** của bit. Mỗi bit như vậy có một trọng số là $2^m n$, với n là vị trí của bit đó. Do đó, một số mà nhị phân là 10011110 sẽ được biểu diễn như sau:

	7	6	5	4	3	2	1	0	← Vị trí
1	0	0	1	1	1	1	0		
↑	↑	↑	↑	↑	↑	↑	↑	↑	
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0		← Trọng số (Weight hay Significance)

và sẽ có giá trị là:

$$\begin{aligned} & 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\ & = 2^7 + 2^4 + 2^3 + 2^2 + 2^1 = 128 + 16 + 8 + 4 + 2 = 158 \end{aligned}$$

Tuy nhiên, bit có trọng số lớn nhất của một số ở dạng nhị phân, còn gọi là MSB (the most significant bit), là bit đầu tiên bên trái của dạng lưu trữ đó, có thể là bit dấu, hoặc không là bit dấu. Nếu MSB không phải là bit dấu, thì nó được dùng để chỉ giá trị như các bit khác, và số được hiểu luôn luôn là số không dấu (tức số luôn luôn dương). Ngược lại, nếu MSB là bit dấu, thì chính nó sẽ cho biết số đang được biểu diễn là âm hay dương, nếu bit dấu = 0, số là dương, và sẽ có một giá trị được tính theo nguyên tắc bình thường như ví dụ trên, nếu bit dấu bằng 1, số được hiểu là số âm, và có giá trị bằng giá trị tính được trừ cho 2^m , với m là vị trí của bit dấu trong dạng mã nhị phân hiện hành. Như vậy, nếu một số 16 bit, có bit dấu, có biểu diễn nhị phân là:

$$\begin{array}{ccccccccccccccccc} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ & \uparrow & & & & & & & & & & & & & & & \\ \text{Bit dấu} & 15 & & & & 2^9 & 2^7 & & 2^3 & 2^1 & \leftarrow \text{trọng số các bit } 1 \end{array}$$

thì sẽ có giá trị

$$2^9 + 2^7 + 2^3 + 2^1 = 512 + 128 + 8 + 2 = 650$$

còn nếu có biểu diễn là:

$$\begin{array}{ccccccccccccccccc} & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ & \uparrow & & & & & & & & & & & & & & & \\ \text{bit dấu} & 15 & & & & 2^9 & 2^7 & & 2^3 & 2^1 & & & & & & & \end{array}$$

sẽ có giá trị là:

$$\begin{aligned} & 2^9 + 2^7 + 2^3 + 2^1 - 2^{15} = 512 + 128 + 8 + 2 - 32768 \\ & = 650 - 32768 = -32118 \end{aligned}$$

Ta cũng có thể tính ra số âm của một số có dấu được biểu diễn ở dạng nhị phân bằng cách lấy bù 2 (*2-complement*) của số đó (bù 2 sẽ là bù 1 cộng thêm 1), tức lấy đảo các bit của số nhị phân rồi cộng thêm 1, ta cũng sẽ tính được giá trị của số âm đó như ví dụ trên.

Chú ý, nếu đây là một số không dấu (*unsigned*) thì với cùng dạng mã nhị phân

1	0	0	0	0	1	0	0	1	0
15	9	7			3	1			

sẽ có giá trị là:

$$2^{15} + 2^9 + 2^7 + 2^3 + 2^1 = 32768 + 512 + 128 + 8 = 33416$$

Do đó, việc khai báo biến là **signed** hoặc **unsigned** là rất quan trọng, ta cần phải lưu ý và có sự phân biệt rõ ràng giữa hai dạng biến trên.

Ví dụ 8.13 Khi cần khai báo một biến n có kiểu là **unsigned int** ta chỉ cần viết:

CuuDuongThanCong . com

unsigned int n;

hoặc gọn hơn

unsigned n;

hoặc chỉ cần viết:

long p;

là đủ để khai báo cho biến p có kiểu là **signed long int**.

Riêng kiểu **char** tùy thuộc vào chương trình biên dịch mà kiểu mặc nhiên (default) là **unsigned** hay **signed**. Hiện nay nhiều chương trình biên dịch cho phép chúng ta quy định kiểu mặc nhiên của kiểu **char** là **signed** hay **unsigned** tùy ý. Do đó ta cần chú ý là khi muốn khai báo đây là một **char** có bit dấu thì nên khai báo đầy đủ.

Ví dụ 8.14

signed char c;

Tuy nhiên, thông thường **char** được hiểu là **signed char**. Do đó, khi muốn khai báo một biến có kiểu **unsigned char**, ta cần phải khai báo đầy đủ như sau.

Ví dụ 8.15

```
unsigned char c;
```

Mỗi kiểu dữ liệu chỉ biểu diễn được các giá trị nằm trong một giới hạn nhất định. Giới hạn này phụ thuộc vào số bit mà kiểu dữ liệu đó quy định khi khai báo biến và do đó còn tùy thuộc vào loại máy. Giới hạn này ta gọi là tầm trị của kiểu. Đối với Turbo C dùng trên IBM PC và PS/2, các kiểu dữ liệu này quy định kích thước và tầm trị như sau.

Bảng 8.2 Các kiểu dữ liệu, tầm trị và kích thước

Kiểu	Kích thước	Tầm trị biểu diễn
unsigned char	8 bit	0..255
char	8 bit	-128..+ 127
unsigned short	16 bit	0..65535
short	16 bit	-32768..+ 32767
unsigned int	16 bit	0..65535
int	16 bit	-32768..+ 32767
unsigned long	32 bit	0.. 4294967295
long	32 bit	-2147483648..+ 2147483647
float	32 bit	3.4 E-38..3.4 E + 38
double	64 bit	1.7 E-308..1.7 E + 308
long double	80 bit	3.4 E- 4932..1.1 E + 4932

8.3 HẰNG (Constant)

Hằng là những giá trị cố định có trị hoàn toàn xác định và không thể thay đổi được chúng trong quá trình thực thi chương trình. Trong C, mỗi hằng đều có một kiểu dữ liệu riêng mà căn cứ vào kiểu dữ liệu ta có các loại hằng sau:

- Hằng số
- Hằng ký tự
- Chuỗi ký tự
- Biểu thức hằng

1- Hằng số

Hằng số là các trị số đã xác định, một hằng số có thể là số nguyên hoặc số thực.

Hằng số nguyên

Trong ngôn ngữ C, hằng số nguyên có thể thuộc một trong hai kiểu là *integer* hoặc *long integer*. Ứng với mỗi kiểu, hằng số có thể được biểu diễn ở dạng thập phân, bát phân hay thập lục phân.

- Hằng số nguyên được viết một cách bình thường và thường chiếm một từ (word) trong bộ nhớ, do đó nó có giá trị đi từ -32768 đến 32767 hoặc từ -2.147483648 đến +2147483647, có nghĩa là MSB của dạng lưu trữ nhị phân của một số nguyên luôn là bit dấu.

Ví dụ 8.16

Các hằng số nguyên 10 -4 -23456

Do đó, ta cần lưu ý khi sử dụng hằng số nguyên vượt quá tầm quy định.

Ví dụ 8.17 Xét chương trình sau khi hằng số nguyên là 16 bit:

```
#include <stdio.h>
main()
{
    printf ("%d %d %d", 32767, 32767 + 1, 32767 + 2);
}
```

Chương trình này sẽ in ra màn hình ba trị sau:

32767 -32768 -32767

(Xin mời độc giả tự giải thích việc in ra ba trị trên)

- Hằng số nguyên dạng long integer lại được lưu trữ trong bộ nhớ với chiều dài 32 bit, có nghĩa là nó có thể có trị nằm trong khoảng -2147483648 đến +2147483647, và khi viết các hằng số nguyên dạng này ta cần phải thêm l hay L ngay sau số cần làm việc.

Ví dụ 8.18

```
#include <stdio.h>
#include <conio.h>
main()
{
    clrscr();
    printf ("%ld %ld %ld", 32767L, 32768L, 32769L);
    getch();
}
```

Hằng số nguyên có thể ở dạng *unsigned*, khi đó ta sẽ thêm u hoặc U vào ngay sau số đang làm việc (số đó có thể đang ở kiểu integer hoặc long integer).

Ví dụ 8.19 Các hằng số sau đây ở dạng unsigned

123U 234u 24UL

Ngoài ra C còn cho phép viết hằng số nguyên ở hệ bát phân (*octal*) hoặc thập lục phân (*hexadecimal* gọi tắt là *hex*). Muốn viết số ở hệ bát phân ta cần phải thêm chữ số 0 ngay trước hằng nguyên, số ở hệ thập lục phân ta cần phải thêm 0x hoặc 0X ngay trước hằng nguyên (số đó có thể đang ở kiểu *integer* hoặc *long integer*).

Ví dụ 8.20 Xem các hằng nguyên sau

<i>Hằng nguyên</i>	<i>Dạng biểu diễn</i>	<i>Tương đương thập phân</i>
12347	decimal	12347
-153	decimal	-153
034	octal	28 ($3 \cdot 8^1 + 4 \cdot 8^0$)
0xa5	hex	165 ($10 \cdot 16^1 + 5 \cdot 16^0$)
0XA2	hex	162 ($10 \cdot 16^1 + 2 \cdot 16^0$)
123L	decimal (dạng long)	123
034L	octal (dạng long)	28
0xa2L	hex (dạng long)	162

Hằng thực

Trong ngôn ngữ C, số thực có thể ở dạng dấu chấm tinh hoặc dấu chấm động.

Ví dụ 8.21 Các số thực sau ở dạng dấu chấm tinh

1.4 -2.34 -10.0234

Một hằng thực ở dạng số dấu chấm động có thể có các thành phần sau:

- Phần nguyên: phần này là tùy yêu cầu.
- Dấu chấm thập phân: bắt buộc phải có.
- Phần lẻ: tùy yêu cầu.
- Các ký tự "e" hoặc "E" và một số mũ. Số mũ bắn thân nó có thể âm. Nếu số mũ là xác định thì số ở các phần nguyên và phần lẻ sẽ được nhân với 10 lũy thừa số mũ tương ứng.

Ví dụ 8.22 Nếu có số thực ở dạng dấu chấm động

123. 45e2

thì ta sẽ có số thực

123.45×10^2

Sau đây ta xét một số ví dụ về số dấu chấm động.

Ví dụ 8.23

Hằng thực dấu chấm động	Hằng thực tương đương
2.1415e4	21415.0
0.2344e-4	0.00002344
.2344e3	234.4

Cần lưu ý:

- Các hằng số được viết không có dấu thập phân hoặc số mũ, sẽ được hiểu là nguyên và được lưu trữ theo kiểu int, ngược lại sẽ được lưu trữ theo kiểu double.
- Các hằng số nguyên lớn hơn khả năng một int được tự động lưu trữ theo kiểu long.
- Các hằng số nguyên lớn hơn một long được lưu trữ theo kiểu double

2. Hằng ký tự

Hằng ký tự biểu diễn một giá trị ký tự đơn, ký tự này phải được viết giữa cặp dấu nháy đơn (""), mỗi ký tự có một mã số tương ứng trong bảng mã ký tự của máy, bình thường là mã ASCII.

Ví dụ 8.24 'a' '/' '9' là những hằng ký tự

Ví dụ 8.25 'A' có mã là 65 trong bảng mã ASCII.

'0' có mã là 48 (0×30) trong bảng mã ASCII.

(Xin độc giả xem thêm bảng mã ASCII phần phụ lục cuối sách)

Một ký tự khi in ra màn hình có thể theo nhiều dạng khác nhau tùy vào chuỗi định dạng xuất trong chuỗi ký tự điều khiển in của hàm printf. Ta hãy xét ví dụ sau đây:

Ví dụ 8.26

```
#include <stdio.h>
main()
{
    printf ("Ký tự: %c %c %c \n", 'A', '$', '1');
    printf ("Mã ASCII (Octal): %o %o %o \n", 'A', '$', '1');
    printf ("Mã ASCII (Decimal): %d %d %d \n", 'A', '$', '1');
}
```

Chương trình in ra màn hình kết quả sau:

Ký tự: A \$ 1

Mã ASCII (Octal):	101	44	61
-------------------	-----	----	----

Mã ASCII (Decimal):	65	36	49
---------------------	----	----	----

Có một số ký tự đặc biệt không in được lên màn hình (*nonprinting character*) như các ký tự điều khiển (ví dụ ký tự xuống dòng), hoặc ký tự mà ta chỉ biết giá trị mã ASCII thì ANSI đã quy định cách viết các hằng ký tự này ở dạng chuỗi thoát (*escape sequence*), đây chính là các biểu diễn thấy được của các ký tự đặc biệt này.

Ví dụ 8.27

'\n' là ký tự xuống dòng (*line feed*)

'\45' là ký tự ASCII có mã octal là 45 hay 37 decimal

'\0' là ký tự NUL

Bảng sau đây sẽ cho ta các chuỗi thoát của hằng ký tự đặc biệt.

Bảng 8.3 Chuỗi thoát của các ký tự không hiển thị được

Chuỗi	Giá trị	Ký tự được hiểu	Tác dụng
\a	0 x 07	BEL	Phát tiếng còi
\b	0 x 08	BS	Xóa ký tự bên trái
\f	0 x 0C	FF	Sang trang
\n	0 x 0A	LF	Xuống dòng
\r	0 x 0D	CR	Enter
\t	0 x 09	HT	Tab theo cột
\v	0 x 0B	VT	Tab theo bảng
\\\	0 x 5C	\	Dấu backslash
\'	0 x 2C	'	Dấu nháy đơn '
*	0 x 22	*	Dấu nháy kép *
\?	0 x 3F	?	Dấu chấm hỏi?
\ddd	ddd	Ký tự có mã theo có số 8 là ddd trong bảng mã ASCII	
\x HH *	0 x HH	Ký tự có mã theo có số 16 là HH trong bảng mã ASCII	

* Chú ý là bắt buộc phải dùng \x, chứ không được \X

Các ký tự bình thường như dấu \, dấu ?, dấu nháy đơn và kép, thực ra vẫn là những ký tự nhìn thấy được, nhưng vì khi viết chúng lại có một ý nghĩa đặc biệt đối với C, nên muốn biểu diễn bản thân chúng, ta vẫn phải dùng chuỗi thoát.

Nếu sau dấu '\' là một ký tự nào đó, không nằm trong các chuỗi thoát quy định trên thì C sẽ bỏ qua cả cặp ký tự (dấu \' và ký tự sau nó) coi như không hề có chúng.

3- Chuỗi ký tự

Trong ngôn ngữ C, một chuỗi ký tự là một loạt các ký tự nằm trong cặp dấu nháy kép (" "); các ký tự này có thể là ký tự được biểu diễn bằng chuỗi thoát.

Ví dụ 8.28

"Một chuỗi ký tự"

"Chuỗi ký tự có chuỗi thoát: i can't go to school \n\\"

Trong bộ nhớ, chuỗi ký tự đang được sử dụng được lưu trữ tại địa chỉ bắt đầu xác định như một dãy ký tự liên tiếp tận cùng bằng một ký tự kết thúc chuỗi, ký tự NUL (tức ký tự '\0', có mã số là 0 trong bảng mã ASCII).

Ví dụ 8.29 Chuỗi String được lưu trữ trong bộ nhớ như sau:

S	t	r	i	n	g	\0
---	---	---	---	---	---	----

Chúng ta không nên nhầm lẫn giữa hằng ký tự và chuỗi ký tự như trong trường hợp sau: "A" được lưu trữ là:

A	\0
---	----

tức tốn 2 byte cho "A", còn 'A' chỉ được lưu trữ trong một byte mà thôi:

A

Một chuỗi ký tự cũng có thể là chuỗi rỗng ("") và trong bộ nhớ chuỗi này được lưu trữ chỉ có một ký tự '\0'.

Một chuỗi ký tự có thể được viết trên nhiều hàng, khi đó ở cuối mỗi hàng của chuỗi chúng ta có dấu \.

Ví dụ 8.30 Chuỗi ký tự sau đây được viết trên nhiều hàng nhưng vẫn được hiểu là một chuỗi ký tự liền nhau, các dấu \ cuối mỗi hàng sẽ không xuất hiện lúc in chuỗi ra màn hình:

"Chuỗi ký tự này được viết trên nhiều hàng \n nhưng nó sẽ
được in ra \a trên màn hình thành \n
các hàng \n khác với chuỗi mà chúng ta ghi ở đây \n"

Cần biết rằng, thực sự một chuỗi ký tự không phải là một hằng chuỗi, nếu trong cùng một chương trình có một chuỗi được sử dụng đi lại nhiều lần ở nhiều chỗ khác nhau trong chương trình thì

mỗi bản sao của chuỗi như vậy đều được lưu trong bộ nhớ ở những vị trí khác nhau, do đó việc thay đổi nội dung của chuỗi là hoàn toàn có khả năng tuy điều này không được khuyến khích.

Để hiểu rõ hơn ta hãy xét chương trình in ra màn hình vị trí trong bộ nhớ của hai chuỗi ký tự "*i go to school*". Chương trình có sử dụng chuỗi định dạng xuất "%p" để in ra địa chỉ của hai chuỗi này, điều này sẽ được nhắc lại trong chương đề cập đến con trỏ (pointer).

Ví dụ 8.31

```
#include <stdio.h>
#include <conio.h>
main()
{
    clrscr();
    printf ("Chuoi 1 co vi tri: %p \n", "i go to school");
    printf ("Chuoi 2 co vi tri: %p \n", "i go to school");
    getch();
}
```

Chương trình sẽ in ra màn hình vị trí bắt đầu của hai chuỗi "*i go to shool*". Ví dụ trên máy đang soạn thảo giáo trình này thì màn hình xuất hiện:

Chuoi 1 co vi tri: 00C3

Chuoi 2 co vi tri: 00EB

4- Biểu thức hằng

Một biểu thức được xem là một biểu thức hằng nếu giá trị của biểu thức hoàn toàn xác định, như vậy một biểu thức toán học là một biểu thức hằng khi trong biểu thức đó các toán hạng đều là những hằng số hoặc hằng ký tự. Khi đó, biểu thức hằng sẽ được chương trình biên dịch tính trước ra một trị bằng số xác định và trị này được ghi vào chương trình đã dịch từ chương trình nguồn.

Ví dụ 8.32 Xét các biểu thức hằng sau

$10 - 13 \% 3$ sẽ được tính trước và được ghi là 9

'a' - 'A' sẽ được tính trước và được ghi là 32

$1 < 8$ sẽ được tính trước và được ghi là 1 (true)

Chính vì khả năng này một chương trình C sau khi dịch xong được thực thi thì máy sẽ không cần tính lại những biểu thức này nữa, điều này làm giảm thời gian chạy chương trình, mà chương trình nguồn vẫn giữ được biểu thức ban đầu, tiện cho việc kiểm tra, đọc lại và sửa đổi chúng.

8.4 BIẾN (Variable)

8.4.1 Khai báo biến

Tất cả các biến được sử dụng trong một chương trình C đều phải được khai báo trước. Việc khai báo này giúp cho chương trình biên dịch có thể biết được kích thước của biến đó, vị trí của chúng trong bộ nhớ và sự tồn tại của chúng trong chương trình, khi muốn sử dụng biến ta chỉ cần gọi tên biến, dĩ nhiên tên biến phải là một danh hiệu không chuẩn hợp lệ (Xin độc giả xem lại mục 4.1).

Ví dụ 8.33

bat_dau if _hoten Thu_thang_1_68	}	đều là các danh hiệu hợp lệ
printf pow 31_thang_12 ket thuc		đều là các danh hiệu không hợp lệ

C là ngôn ngữ nhạy cảm với chữ hoa và chữ thường, do đó nếu hai tên biến hợp lệ khác nhau ở kiểu chữ hoa hoặc thường thì hai biến đó là khác nhau.

Ví dụ 8.34

BIEN BiEN Bien bIEN bien	}	đều là những biến khác nhau

Cú pháp khai báo biến

kiểu dsach_tenbien;

Giải thích:

kiểu	kiểu của các biến cần khai báo
dsach_tenbien	danh sách liệt kê các tên biến cần khai báo, các biến cách nhau bằng dấu ","

Ví dụ 8.35

```
int lap, count, max;
double he_so_1, he_so_2, delta;
```

Trị của biến có thể bị thay đổi qua phép gán trị, trong C phép gán được đặc trưng bằng một dấu "=", hoặc các toán tử tăng, giảm của C.

Ví dụ 8.36

```
#include <stdio.h>
main()
{
    int thu;
    thu = 65;
    printf ("Trị trong biến thu là %d \n", thu);
    thu = thu + 10;
    printf ("Sau khi gán, trị trong biến thu là %d \n", thu);
}
```

Chương trình sẽ in ra màn hình như sau:

Trị trong biến thu là 65

Sau khi gán, trị trong biến thu là 75

Biến của một chương trình C có thể được khai báo ở một trong ba vị trí sau:

- Ngoài tất cả các hàm (gọi là khai báo biến ngoài), khi đó ta có biến toàn cục.

Ví dụ 8.37

```
#include <stdio.h>
int a, b; ← khai báo biến ngoài
main()
{
    ...
    a = 10;
    b = a + 24;
    ...
}
```

- Đầu phần thân của một hàm hoặc một khối lệnh (gọi là khai báo biến trong), khi đó ta có biến cục bộ.

Ví dụ 8.38

```
int tong()
{
    int i, tam; ← khai báo biến trong
    ...
    for (i = 10; ...
    ...
}
```

- Trong phần định nghĩa đối số của hàm (gọi là đối số hàm hoặc tham số hàm).

Ví dụ 8.39

```
int luy_thua (int n, char ket_qua)
{
    ...
    for (i = 1; ...
    ...
}
```

Về ý nghĩa, khai báo và cách sử dụng các biến trong, ngoài ta sẽ có một chương riêng để cập đến chúng (đó là chương *Lớp lưu trữ của biến*).

Một biến có thể được khởi động trị ngay sau khi khai báo bằng phép gán với giá trị tương ứng, khi đó cú pháp khai báo biến như sau:

kiểu biến1 = trị1, biến2 = trị2;

với các trị1 và trị2 phải là những giá trị đã xác định.

Ví dụ 8.40

double he_so_1 = 20.37;

char ky_tu = 'A', ky_tu_moi = ky_tu;

8.4.2 Các bổ túc kiểu const và volatile

Trong ngôn ngữ C có hai từ khóa **const** và **volatile** dùng để bổ túc cho khai báo biến.

Từ khóa const: khi được khai báo cho biến thì nó xác định rằng biến sẽ không bị thay đổi trị trong suốt quá trình thực thi chương trình, mọi sự thay đổi trị đều gây ra lỗi, biến đó ta gọi là biến hằng.

Cú pháp:

const kiểu tên biến [= trị];

Ví dụ 8.41

const double bat_dau = 3.1415;

const int max = 100;

Nếu **kiểu** của biến hằng không nêu cụ thể thì biến hằng đó sẽ thuộc loại **int**, ngay cả nếu trị là một trị khác **int** thì chỉ phần nguyên được sử dụng và lưu vào biến hằng mà thôi.

Ví dụ 8.42

const max = 100;

const pi = 3.14; khi đó biến pi chỉ là 3 mà thôi

Dĩ nhiên biến hằng là "biến" nên nó hoàn toàn tuân theo quy luật khai báo và hoạt động của biến, mặt khác nó lại là "hằng" nên không thể thay đổi trị của nó được.

Cần lưu ý rằng, khi khai báo biến hằng thì biến này cần có trị xác định, trị này không đổi trong suốt quá trình chương trình thực thi và biến hằng này có một địa chỉ xác định trong bộ nhớ.

Từ khóa volatile: chỉ ra rằng một biến có thể bị thay đổi từ một tác nhân không nằm trong chương trình. Từ khóa này làm cho biến của C có tính linh động rất cao, ví dụ như biến của C có thể thay đổi theo đồng hồ hệ thống hay theo một chương trình nền nào đó.

Cú pháp:

volatile kiểu <tên_var>;

8.5 BIỂU THỨC

Biểu thức là một sự kết hợp của các toán hạng là các biến, hằng hoặc phép gọi hàm bằng các toán tử xác định của C để tạo ra được một trị, trị này có thể được sử dụng hoặc không được sử dụng tùy nhu cầu của lập trình viên. Do đó, khái niệm biểu thức trong lập trình trên ngôn ngữ C là rất tổng quát, nó có thể chỉ đơn giản là một phép gọi hàm (mà ta có thể không cần quan tâm tới trị trả về của hàm dù hàm có trả về), hoặc nó có thể là một biểu thức gán hay có thể là một biểu thức luận lý trả về trị 1 (nếu biểu thức đó là đúng) hoặc 0 (nếu biểu thức đó sai)...

Chính vì vậy một biểu thức của C sẽ rất phức tạp, nó có thể bao hàm rất nhiều phép toán, nhiều loại phép toán trong đó.

Ví dụ 8.43 Đối với C, một biểu thức như sau là hợp lệ:

$$a = (x = 10) - (y = a + 1) * ((b += 1) > 12);$$

Do đó việc xét chi tiết từng phép toán của C là quan trọng và cần thiết.

8.6 CÁC PHÉP TOÁN CỦA C

8.6.1 Toán tử số học

C có các toán tử số học bình thường giữa hai toán hạng, đó là

- Toán tử cộng (+) : thực hiện phép toán cộng
- Toán tử trừ (-) : thực hiện phép toán trừ

- Toán tử nhân (*): thực hiện phép nhân
- Toán tử chia (/): thực hiện phép chia
- Toán tử modulo (%): thực hiện phép toán lấy số dư của phép chia nguyên.

Các phép toán này đều thực hiện được trên tất cả các toán hạng là hằng, biến hoặc biểu thức có kiểu dữ liệu char, int, long, float, double, trừ toán tử modulo chỉ thực hiện phép toán trên các dữ liệu thuộc các kiểu nguyên (char, int, long, unsigned). Thứ tự kết hợp theo nguyên tắc toán học: nhân, chia, modulo trước; cộng, trừ sau, nếu các toán hạng đều ngang cấp thì kết hợp từ trái sang phải.

Ví dụ 8.44

```
float a, b, c;
double y;
y = a * b - c;
```

Phép toán cộng, trừ cũng có thể được dùng như phép toán đơn hạng (lấy dương và âm của một toán hạng).

Ví dụ 8.45

```
b = - b;
a = + b;
```

Phép nhân và chia các số nguyên sẽ chỉ cho kết quả nguyên (C tự động cắt bỏ phần thập phân nếu có).

Ví dụ 8.46

int a = 10, b = 3, c;
thì c = a/b;
sẽ cho kết quả là số nguyên: c = 3
và nếu có
 c = a % b;
thì kết quả là c = 1
trong khi nếu có:

double x = 10., y = 3, z;
 thì z = x/y;
 sẽ cho kết quả là: z = 3.33333...

Do phép toán * và + là những phép toán có tính kết hợp và giao hoán nên đổi với chúng trật tự tính toán không được đặt ra, và các trình biên dịch có thể thay đổi trật tự tính toán của chúng để tối ưu hóa phép tính của biểu thức. Việc này thường không tạo ra sự khác biệt về kết quả.

Khi thực hiện các phép toán số học, một vấn đề đặt ra là nếu có nhiều toán hạng khác kiểu nhau thì C sẽ thực hiện việc tính toán biểu thức ra sao? Để giải quyết vấn đề này, C sẽ thực hiện việc chuyển kiểu tự động theo quy luật sau: toán hạng thuộc kiểu có trị số nhỏ hơn sẽ được chuyển sang kiểu có trị số lớn hơn, ví dụ như nếu trong một biểu thức có:

- một toán hạng thuộc kiểu char và toán hạng kia thuộc kiểu short thì hai toán hạng sẽ được chuyển kiểu về int trước khi tính;
- float được tự động chuyển kiểu về double trước khi tính;
-

Nói chung, ta có quy cách đổi như sau:

signed char	→ int	bằng cách mở rộng bit dấu
unsigned char	→ int	diền 0 vào phần cao
short	→ int	giữ nguyên
unsigned short	→ unsigned	giữ nguyên
int	→ long	mở rộng bit dấu
unsigned	→ unsigned long	diền 0 vào phần cao
int	→ unsigned	giữ nguyên
long	→ unsigned long	giữ nguyên
float	→ double	diền 0 vào phần định trị

Qui tắc đổi như trên chỉ nhằm mục đích đảm bảo được giá trị của toán hạng sau khi đổi là chính xác phù hợp với cả biểu thức (ngay cả khi các giá trị đó là số âm).

Ví dụ 8.47 Nếu ta có khai báo biến như sau:

```
double a = 10.;
```

```
int b = 3;
```

Thì phép toán a/b sẽ đổi giá trị lấy từ b (3) thành số double (3.00...) trước, rồi thực hiện phép toán $10./3.0$, kết quả (3.333...) sẽ là số double.

8.6.2 Toán tử quan hệ

Trong C có các toán tử quan hệ chỉ ra mối quan hệ giữa hai toán hạng là lớn nhỏ, bằng hay không bằng ra sao, do đó chúng còn được gọi là toán tử so sánh:

- Toán tử bằng ($==$): chỉ ra mối quan hệ bằng giữa hai toán hạng
- Toán tử khác ($!=$): chỉ ra mối quan hệ khác giữa hai toán hạng
- Toán tử lớn hơn ($>$): chỉ ra mối quan hệ lớn hơn giữa hai toán hạng
- Toán tử nhỏ hơn ($<$): chỉ ra mối quan hệ nhỏ hơn giữa hai toán hạng
- Toán tử lớn hơn hoặc bằng (\geq): chỉ ra mối quan hệ lớn hơn hoặc bằng giữa hai toán hạng
- Toán tử nhỏ hơn hoặc bằng (\leq): chỉ ra mối quan hệ nhỏ hơn hoặc bằng giữa hai toán hạng.

Khi mối quan hệ giữa hai toán hạng theo toán tử quan hệ trong biểu thức là **ĐÚNG** thì biểu thức đó sẽ trả về một trị nguyên là 1, còn ngược lại mối quan hệ đó là **SAI** thì biểu thức đó sẽ trả về một trị nguyên là 0, đây là các trị nguyên bình thường mà ta có thể dùng nó để tính toán. Đây chính là điểm khác nhau giữa ngôn ngữ C và các ngôn ngữ khác, và cũng chính vì điểm này mà lập trình trên ngôn ngữ C đôi khi trở nên khó nhìn thậm chí lặp đi, tuy rằng đây thật sự là điểm mạnh của ngôn ngữ C.

Ví dụ 8.48

```

if (delta > 0)
{
    x1 = ... ;
    x2 = ... ;
    printf ("Nghiem cua phuong trinh la ..."); ] (*) 
}
else
...

```

Trong ví dụ trên, nếu *delta* lớn hơn 0 thì khi đó biểu thức *delta > 0* là đúng, tức biểu thức $(\text{delta} > 0) = 1$, lệnh phức (*) sẽ được thực hiện, còn nếu *delta* nhỏ hơn hoặc bằng 0 thì biểu thức trên sẽ cho kết quả là sai (0), khi đó lệnh trong else sẽ được thực hiện.

Ví dụ 8.49 Xét các khai báo biến và biểu thức sau

```

int a, b, c; duongthancong . com
char kt;
a = 1;
b = 2;
c = -3;
kt = (a >= 4) * 2 + (b < 3) - c;

```

Biểu thức $(a \geq 4)$ sai (0) vì $a = 1$ nên $(a \geq 4) = 0$, do đó biểu thức $(a \geq 4) * 2 = 0$. Biểu thức $(b < 3)$ đúng (1), do đó sau cùng $kt = 0 + 1 - (-3) = 4$.

Cần lưu ý phép toán so sánh **Bằng ==** thường bị quen đánh là " $=$ ", tức lệnh gán, điều này không bị chương trình biên dịch báo lỗi vì C vẫn hiểu được, nhưng khi đó biểu thức lại có một ý nghĩa khác, để hiểu rõ ta hãy xét ví dụ sau.

Ví dụ 8.50 Xét hai trường hợp sau:

1/ $a = (b == 4);$

Nếu b bằng 4, thì gán
 $a = 1$, còn không $a = 0$

2/ $a = (b = 4);$

Gán $b = 4$ (như vậy b đã có trị là 4),
lấy trị đang có của b gán vào cho a ,
như vậy a luôn bằng 4

Phép chuyển kiểu giữa các toán hạng trong các biểu thức có toán tử quan hệ tương tự như ở các biểu thức có toán tử số học đã nói mục 8.6.1 trên. Để rõ phần này ta xét chương trình ví dụ sau:

Ví dụ 8.51

```
#include <stdio.h>
#include <conio.h>
main()
{
    int x, y;
    clrscr();
    x = 10;
    y = 24;
    printf ("Tri cua bien thuc x == y la %d \n"; x == y );
    printf ("Tri cua bien thuc x <= y la %d \n"; x <= y );
    printf ("Tri cua bien thuc x >= y la %d \n"; x >= y );
    printf ("Tri cua bien thuc x != y la %d \n\n", x != y );
    if (x == y)
        printf ("Tri cua bien thuc x == y la true \n");
    else
        printf ("Tri cua bien thuc x == y la false \n");
    if (x = y)
        printf ("Tri cua bien thuc x = y la true \n\n");
    else
        printf ("Tri cua bien thuc x = y là false \n");
    printf ("Voi x la %d va y la %d \n", x, y);
    getch();
}
```

Chương trình này sẽ in ra màn hình kết quả như sau:

```
Tri cua bien thuc x == y la 0
Tri cua bien thuc x <= y la 1
Tri cua bien thuc x >= y la 0
Tri cua bien thuc x != y la 1
Tri cua bien thuc x == y la false
Tri cua bien thuc x = y la true
Voi x la 24 va y la 24
```

Mời độc giả tự giải thích kết quả này.

8.6.3 Toán tử logic

C có các toán tử logic not (!), and (&&), or (||). Các toán tử này cho phép lập trình viên liên kết các biểu thức quan hệ đơn lẻ với nhau để tạo các biểu thức phức tạp hơn.

Bảng sự thật cho thấy cách hoạt động của các toán tử này:

Bảng 8.3 Bảng hoạt động của các toán tử logic

Toán hạng 1	Toán hạng 2	Kết quả		
A	B	! A	A && B	A B
bằng 0	bằng 0	1	0	0
bằng 0	khác 0	1	0	1
khác 0	bằng 0	0	0	1
khác 0	khác 0	0	1	1

Các toán tử này có độ ưu tiên là not, and, or; nếu trong biểu thức các toán tử đều ngang cấp nhau thì thứ tự tính toán từ trái sang phải.

- Toán tử not, đặc trưng bằng dấu "!" : đảo ngược giá trị thật của biểu thức theo sau nó từ 0 qua 1, từ khác 0 về 0, có nghĩa là nếu một biến x có trị khác 0 (chẳng hạn bằng 1, 5.67, -9.469...) thì biểu thức

x

có trị true, nhưng biểu thức

!x có trị false (0)

Đây là toán tử đơn toán hạng, do đó cú pháp tổng quát của nó như sau:

! biểu thức

với biểu thức có thể là hằng, biến hoặc biểu thức.

Toán tử and, đặc trưng bằng hai dấu &, tức &&, cú pháp sử dụng biểu_thức_1 && biểu_thức_2

Như vậy khi hai biểu thức được and với nhau thì biểu thức đó chỉ đúng (1) khi cả hai biểu thức đều cùng đúng (khác 0).

Ví dụ 8.52

```
int a, b;
a = 4;
b = 5;
```

thì biểu thức

$(a > 5) \&\& (b < 3)$

có trị là false (0).

Toán tử or, đặc trưng bằng hai dấu ";" tức "||", cú pháp sử dụng

biểu_thức_1 || biểu_thức_2

Kết quả của biểu thức "or" là true (1) khi chỉ cần một trong hai biểu thức của nó là true (khác 0)

Ví dụ 8.53

```
int a, b;
a = 4;
b = 5;
```

thì biểu thức

$(a > 5) || (b > 3)$

có trị là true (1) vì $(a > 5)$ là false, trong khi $(b > 3)$ là true nên cả biểu thức là true.

Cần lưu ý rằng, toán hạng của các toán tử này có thể là một biểu thức hoặc giá trị bất kỳ nào và ở một trong hai trạng thái: "bằng 0" (false) hoặc "khác 0" (true). Và kết quả của các biểu thức có các toán tử logic này cũng là một giá trị int: 1 hoặc 0, các kết quả này cũng có thể được dùng để tính toán tiếp theo.

Ví dụ 8.54 Trong mệnh đề if sau đây thay vì viết

```
if (delta == 0)
{
    ...
}
```

ta có thể viết gọn hơn như sau:

```
if (!delta)
{
    ...
}
```

khi đó nếu *delta* là một số bằng 0 thì biểu thức (!delta) sẽ có giá trị bằng 1, và lệnh if sẽ được thực hiện, còn ngược lại, nếu *delta* khác 0, thì (!delta) có trị = 0, lệnh if sẽ không được thi hành.

Ví dụ 8.55 Xét chương trình ví dụ sau

```
#include <stdio.h>
#include <conio.h>

main()
{
    int min = 10, max = 10000; /* các giới hạn trên và dưới */
    int m1, m2; /* hai trị nhập để kiểm tra */
    clrscr();
    printf("Moi nhap hai so: ");
    scanf ("%d %d", &m1, &m2);
    if ((m1 < max) && (m1 > min))||(m2 < max) && (m2 > min))
        printf ("Toi thieu co mot tri trong tam (min, max)\n");
    if ((m1 < max && m1 > min) && (m2 < max && m2 > min))
        printf ("Ca hai tri deu nam trong tam (min, max)\n");
    getch();
}
```

Trong biểu thức điều kiện của if đầu thì hai phép toán and (**&&**) được kết hợp trước, sau đó là phép toán or (**||**), or hai kết quả từ hai phép toán **&&** mới tính, kết quả sau cùng nếu là 1 thì các lệnh trong if được thực hiện; đối với if sau, thì hai phép toán and (**&&**) trong ngoặc (...) được tính trước, sau đó phép toán and ngoài sẽ and hai kết quả này lại với nhau, tương tự nếu kết quả là 1 thì các lệnh trong if được thực hiện.

Điều cần lưu ý là các phép toán and (**&&**) và or (**||**) được thực hiện từ trái sang phải và việc tính toán sẽ dừng lại ngay khi giá trị của cả biểu thức logic đã xác định trị mà không cần tính tiếp các phần còn lại của biểu thức nữa.

Ví dụ 8.56 Khi tính giá trị của biểu thức:

$$(c >= 'A') \&\& (c <= 'Z') || (c >= 'a') \&\& (c <= 'z')$$

thì $(c >= 'A')$ được tính toán trước, nếu sai thì giá trị của biểu thức $(c <= 'Z')$ không cần tính nữa, (có nghĩa là biểu thức $(c >= 'A') \&\& (c <= 'Z')$ cũng đã được xác định là sai), mà việc tính toán được chuyển sang về phải của phép toán or (**||**) là $(c >= 'a') \&\& (c <= 'z')$, việc tính toán tương tự sẽ diễn ra; ngược lại, nếu $(c >= 'A')$ là đúng thì $(c <= 'Z')$ sẽ được tính, và nếu kết quả này cũng đúng thì giá trị của cả biểu thức and đầu xem như đã xác định (là đúng) nên phần còn lại của biểu thức or không cần tính nữa, và như vậy cả biểu thức or đã xác định trị (là đúng).

Đây có thể nói là một sự tối ưu hóa tốt của C nhằm làm giảm thời gian tính toán một biểu thức logic. Cũng chính vì vậy, chúng ta cần lưu ý khi muốn kết hợp nhiều biểu thức vào một biểu thức logic, có thể có một số biểu thức sẽ không được thực hiện (nhất là khi các biểu thức logic lại là các biểu thức gán, biểu thức có sự gọi hàm,...) nếu kết quả các phép toán logic trước đó đã làm cho kết quả của toàn biểu thức được xác định, mà điều này có thể dẫn tới kết quả bài toán bị sai.

Ví dụ 8.57

```
scanf ("%d", &x);
if ( (x >= 10) && (x <= (y = 100)) )
{
    ...
}
```

Trong biểu thức logic của if ta kết hợp việc gán trị cho biến y

$y = 100$

với việc tính giá trị biểu thức logic

$$(x >= 10) \&\& (x <= y)$$

nếu điều kiện đầu tiên ($x \geq 10$) bị sai thì kết quả của biểu thức logic đã hoàn toàn xác định là sai, khi đó việc tính toán biểu thức này kết thúc, mà không thực hiện biểu thức gán trị cho biến y nữa. Như vậy, lệnh if thực hiện đúng, tuy nhiên y lại không được gán trị nên điều này có thể sẽ dẫn tới sai khi chương trình chạy tiếp.

8.6.4 Toán tử trên bit

Có sáu toán tử đặc biệt cho phép lập trình viên xử lý dữ liệu kiểu nguyên (như char, int, long, unsigned hoặc signed) ở cấp độ bit, các toán tử này được gọi là toán tử trên bit. Đó là các toán tử:

- NOT : ~, thực hiện việc đảo bit, từ bit 0 qua 1 và ngược lại
- AND : &, thực hiện việc *and bit*
- OR : |, thực hiện việc *or bit*
- XOR : ^, thực hiện việc *xor bit*
- Dịch trái : <<, dịch các bit sang trái
- Dịch phải : >>, dịch các bit sang phải

Ta có bảng hoạt động của các toán tử này như sau:

Bảng 8.4 Bảng hoạt động của các toán tử trên bit

Bit	Bit	Phép toán			
		~ A	A & B	A B	A ^ B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Ví dụ 8.58

a = 1030 có mã nhị phân là 0000 0100 0000 0110

b = 224 có mã nhị phân là 0000 0000 1110 0000

thì

-a	1111	1011	1111	1001
a & b	0000	0000	0000	0000
a b	0000	0100	1110	0110
a ^ b	0000	0100	1110	0110

Các toán tử trên bit có thể được sử dụng để thực hiện các thao tác trên từng bit của dữ liệu đang xét.

Phép toán AND (&) sẽ kết hợp với một trị mà ta gọi là mặt nạ che các bit dữ liệu không cần quan tâm, như vậy trong giá trị mặt nạ tại những vị trí cần che, bit sẽ có trị là 0, các bit còn lại là 1, và qua đó ta có thể đánh giá được dữ liệu đang làm việc một cách chính xác.

Ví dụ 8.59 Nhập vào một số nguyên, xét xem bit có vị trí là 9 có bằng 1 hay không.

```
#include <stdio.h>
#include <conio.h>
#define MASK 0x0200
main()
{
    int a;
    clrscr();
    printf ("Moi nhap mot so: ");
    scanf ("%d", &a);
    if (a & MASK)
        printf ("Bit 9 bang 1 \n");
    else
        printf ("Bit 9 bang 0 \n");
    getch();
}
```

Chú ý rằng, ở đây mặt nạ số hex 0x200 che tất cả các bit trừ bit có vị trí 9 để xét, ở đây số nguyên nhập vào thuộc kiểu int dài 16 bit nên mặt nạ cũng dài tương ứng 16 bit.

9 ← vị trí

0000 0010 0000 0000

0 2 0 0

Phép toán OR (|) được dùng kết hợp với một mặt nạ để bật các bit cần thiết trong một giá trị lên một 1 và giữ nguyên các bit khác. Để tạo ra một trị mặt nạ, các bit tại các vị trí cần thiết sẽ được bật lên 1, còn lại giữ 0. Kết quả của phép OR thu được từ một giá trị và một mặt nạ là một giá trị mà trong đó các bit quan tâm sẽ được gán bằng 1.

Ví dụ 8.60 Nhập vào một số nguyên, xét xem bit có vị trí là 9 có bằng 1 hay không, nếu nó bằng 0 thì bật bit đó lên 1 và in ra kết quả.

```
#include <stdio.h>
#include <conio.h>
#define MASK 0x0200
main()
{
    int a;
    clrscr();
    printf ("Moi nhap mot so: ");
    scanf ("%d", &a);
    if (a & MASK)
    {
        printf ("Bit 9 bang 1 \n");
        printf ("Tri cua bien a la %d \n", a);
    }
    else
    {
        a = a | MASK;
        printf ("Bit 9 bang 0 va da duoc bat len 1\n");
        printf ("Tri cua bien a la %d \n", a);
    }
    getch();
}
```

Phép toán XOR (^) kết hợp với một mặt nạ dùng để đảo các bit cần thiết trong một giá trị từ 0 lên 1 và từ 1 về 0, giữ nguyên các bit khác. Mặt nạ này có các bit tại các vị trí cần thiết sẽ được bật lên 1, các bit còn lại giữ nguyên.

Ví dụ 8.61 Nhập vào một số nguyên, xét xem bit có vị trí là 9 có bằng 1 hay không, nếu nó bằng 0 thì bật đó lên 1, nếu bằng 1 thì đưa về 0 và in ra kết quả.

```
#include <stdio.h>
#include <conio.h>
#define MASK 0x0200

main()
{
    int a;
    clrscr();
    printf ("Moi nhap mot so: ");
    scanf ("%d", &a);
    if (a & MASK)
        printf ("Bit 9 bang 1 \n");
    else
        printf ("Bit 9 bang 0 \n");
    printf ("Tri cua bien a truoc khi dao bit 9 la %d \n", a);
    a = a ^ MASK;
    if (a & MASK)
        printf ("Bit 9 sau khi dao bang 1 \n");
    else
        printf ("Bit 9 sau khi dao bang 0 \n");
    printf ("Tri cua bien a sau khi dao bit 9 la %d \n", a);
    getch();
}
```

Toán tử NOT (~) cho phép thực hiện việc đảo tất cả các bit của một giá trị từ 0 lên 1 và từ 1 xuống 0. Phép toán như vậy được gọi là phép bù 1 giá trị hiện hành.

Ví dụ 8.62 Hãy tạo ra một mặt nạ gồm toàn các bit có giá trị 1, trừ các bit 0, 2, 5 bằng 0. Muốn có mặt nạ này ta có thể lấy bù 1 của một giá trị mà các bit có vị trí 0, 2, 5 đều bằng 1 như sau:

Có m

0	0	0	0	0	0	0	0	0	1	0	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

thì MASK =~m sẽ là

1	1	1	1	1	1	1	1	1	1	0	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Cần lưu ý rằng, phép lấy bù 1 của một giá trị không lệ thuộc độ dài thực sự của giá trị đó là bao nhiêu bit, đều đó sẽ rất có lợi khi ta muốn viết một chương trình tổng quát không lệ thuộc vào kiểu dữ liệu nhập hay vào loại máy.

Toán tử dịch trái (<<) và dịch phải (>>) cho phép thực hiện việc dời các bit của toán hạng sang bên trái hoặc sang phải. Cú pháp như sau:

biểu_thức_nguyên << số_bit_dời

biểu_thức_nguyên >> số_bit_dời

Với **biểu_thức_nguyên** và **số_bit_dời** có thể là hằng, biến hoặc biểu thức kiểu nguyên.

Trong phép dịch trái, các bit ở bên phải của toán hạng sẽ được ghi vào các giá trị là 0. Còn trong phép dịch phải, thì tùy theo kiểu dữ liệu của toán hạng bên trái mà ta có hai trường hợp sau:

- Nếu toán hạng bên trái có dữ liệu thuộc dạng unsigned (unsigned int, unsigned long, unsigned char) thì phép dịch phải sẽ ghi 0 vào các bit bên trái của kết quả.
- Còn nếu toán hạng bên trái có dữ liệu thuộc dạng signed (int, long, char) thì phép dịch phải sẽ ghi bit dấu vào các bit bên trái của kết quả.

Ví dụ 8.63

Xét biến *n* được khai báo

int n;

đang có trị *n* = 469 (tức 0x01d5), biểu diễn trong bộ nhớ dưới dạng nhị phân là

0	0	0	0	0	0	0	1	1	1	0	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Khi dịch trái 2 bit, ta có:

$n \ll 2$

0	0	0	0	0	1	1	1	0	1	0	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Giá trị $n \ll 2$ có trị hex là 0x0754, trị thập phân là 1876 (trị này chính là $469 * 4$, có nghĩa phép dịch trái 2 bit này tương đương với việc nhân n cho 4, và phép dịch trái m bit sẽ tương đương với việc nhân trị n cho 2^m).

Ví dụ 8.64 Nếu biến n được khai báo là

unsigned n;

Giả sử n đang lưu giá trị $n = 42569$ (tức 0xA649), tức trong bộ nhớ ta có n

1	0	1	0	0	1	1	0	0	1	0	0	1	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Khi phép dịch phải 2 bit xảy ra ($n \gg 2$), kết quả như sau:

0	0	1	0	1	0	0	1	1	0	0	1	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$0 \rightarrow$

Ví dụ 8.65 Nếu biến n được khai báo là

int n;

Giả sử n đang lưu giá trị $n = -32766$ (tức 0x8002), tức trong bộ nhớ ta có n

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

bit dấu

Khi phép dịch phải 2 bit xảy ra ($n \gg 2$), kết quả như sau:

1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$1 \rightarrow$

Ta cũng dễ dàng kiểm tra thấy rằng phép dịch phải m bit tương đương với phép chia trong cả hai trường hợp số dương lẫn số âm.

Ví dụ 8.66 Xét chương trình ví dụ sau

```
#include <stdio.h>
#include <conio.h>
main()
{
    int a;
    clrscr();
    printf ("Moi nhap mot so: ");
    scanf ("%d", &a);
    printf ("Tri tren co dang hex: %x \n", a);
    printf ("Tri tren dich trai 2 bit cho ket qua: \n"
           - Dang hex: %x \n\
           - Dang thap phan %d\n", a << 2, a << 2);
    printf ("Tri tren dich phai 2 bit cho ket qua: \n"
           - Dang hex: %x \n\
           - Dang thap phan %d\n", a >> 2, a >> 2);
    getch();
}
```

8.6.5 Toán tử tăng giảm

Trong C có hai toán tử đặc biệt gọi là toán tử tăng (`++`) và toán tử giảm (`--`) dùng để tăng hoặc giảm một biến nào đó đi 1. Việc tăng giảm này theo kiểu mà biến được khai báo.

Cú pháp sử dụng:

`++ biến`

`biến ++`

`-- biến`

`biến --`

với biến có thể thuộc kiểu bất kỳ hoặc pointer.

Ký hiệu `++` và `--` có thể được đặt trước (gọi là *tiền tố*) hoặc sau (gọi là *hậu tố*) một tên biến.

Ví dụ 8.67

`++ a;`

hoặc

`a ++;`

chính là

`a = a + 1;`

Tuy nhiên, đôi khi việc đặt toán tử trước hoặc sau một biến lại gây ra kết quả khác nhau, nhất là khi dùng phép toán này trong một biểu thức kết hợp với các phép toán khác, khi đó nếu dùng `++` hoặc `--` là tiền tố thì việc tăng hoặc giảm biến sẽ diễn ra trước khi trị của biến được sử dụng để tính toán trong cả biểu thức, còn nếu `++` hoặc `--` là hậu tố thì việc tính toán biểu thức sẽ diễn ra trên trị hiện thời của biến, sau khi biểu thức được tính xong thì biến sẽ được tăng hoặc giảm trị.

Ví dụ 8.68

`int i = 12;`

`double b;`

thì lệnh

`b = i++; (1)`

sẽ gán `b = 12`, sau đó `i` tăng 1, tức `i = 13`, như vậy lệnh (1) sẽ tương đương với hai lệnh

`b = i;`

`i = i + 1;`

Còn lệnh

`b = ++ i;`

sẽ tăng `i` trước, tức `i = 13`, sau đó việc gán trị cho `b` sẽ diễn ra, tức lệnh (2) lại tương đương với hai lệnh

`i = i + 1;`

`b = i;`

Cần lưu ý rằng, cặp dấu ++ hoặc -- phải được viết liền nhau và cần phải rõ ràng, trong những trường hợp có thể lầm lẫn, ví dụ như 2 biểu thức:

`a++ + b`

và

`a + ++b`

là khác nhau.

Các phép toán tăng giảm này nhanh hơn nhiều so với thao tác bình thường là cộng hoặc trừ biến đó với 1 rồi gán trở lại cho biến đó vì chúng rất gần với các phép toán tăng giảm của ngôn ngữ máy. Do đó, người ta thường sử dụng các toán tử này để tối ưu hóa thời gian thực thi của chương trình, nhất là khi việc tăng giảm trị lại lặp đi lặp lại trong các vòng lặp.

Ví dụ 8.69 Trong thân vòng while của chương trình tính tổng từ 1 tới n , ta có thể viết như sau

```
s = 0;
so = 1;
while (so <= n)
{
    s = s + so;
    so++;
}
```

Đi nhiên ta hoàn toàn có thể ghép như sau

```
while (so <= n)
    s = s + so++;
```

8.6.6 Toán tử gán

Phép toán gán là phép toán cơ bản trong mỗi ngôn ngữ lập trình, vì nhờ nó lập trình viên có thể thay đổi trị của biến trong quá trình tính toán.

Trong C phép gán có hai dạng theo cú pháp sau đây.

Gán đơn giản

biến = trị

Gán phức tạp

biến op = trị

Với **trị** có thể là hằng, biến hoặc là biểu thức

op có thể là * / % + - hoặc << >> & ^ |

Phép gán trị phức tạp

biến op = trị

chính là

biến = biến op trị

Ví dụ 8.70

```
int a, b = 2;
a = 4;           →      a = 4
b *= a * 3;     →      b = 24
```

Trong phép gán đơn, nếu hai toán hạng có cùng kiểu thì toán hạng bên phải sẽ được gán vào toán hạng bên trái. Nếu không, trước khi gán toán hạng bên phải sẽ được chuyển theo kiểu của toán hạng bên trái, điều này có thể sẽ gây ra kết quả sai hoặc không chính xác, nếu như kiểu của toán hạng bên trái thấp hơn kiểu của toán hạng bên phải.

Bảng 8.5 Bảng biểu diễn sự tương quan các kiểu trong phép gán

Kiểu toán hạng trái	Kiểu toán hạng phải	Trị có thể mất sau khi gán
signed char	unsigned char	Giá trị > 127, thành số âm
char	short int	Mất trị từ bit 8 trở đi
char	int	Mất trị từ bit 8 trở đi
char	long	Mất trị từ bit 8 trở đi
short int	long int	Mất 16 bit cao (một int)
int	float	Mất phần thập phân và phần trị lớn hơn một int
float	double	Độ chính xác do làm tròn

Đối với phép gán phức hợp, việc chuyển kiểu được thực hiện theo việc chuyển kiểu tự động trong khi thực hiện việc tính toán biểu thức và việc chuyển kiểu của phép gán đơn giản.

Phép gán phức hợp này tỏ ra rất hiệu quả nhất là khi các toán hạng bên trái là những biến khá dài.

Ví dụ 8.71

Thay vì viết:

$$n = n * (x + 5) + n * (a + 8);$$

ta chỉ cần viết:

$$n *= x + 5 + a + 8;$$

Hoặc phức tạp hơn

$$a[i][j] := b[i][j];$$

thay vì phải viết dài dòng

$$a[i][j] = a[i][j] - b[i][j];$$

Đối với C, nếu một biểu thức gán được kết thúc bằng một dấu ";" thì ta có một lệnh gán; còn nếu biểu thức gán này được sử dụng trong một biểu thức phức hợp khác thì biểu thức gán sẽ có trị là trị của biến sau khi gán.

Ví dụ 8.72

```
int a = 4, b = 3;
b += (a = 2 * b) + (a *= b);
```

đầu tiên a được gán bằng $2 * b$, như vậy $a = 6$, biểu thức $(a = 2 * b)$ có trị là 6, sau đó a tiếp tục được gán là $a * b$, vậy $a = 18$, do đó $b = 3 + 6 + 18 = 27$.

8.6.7 Toán tử phẩy - Biểu thức phẩy

Trong ngôn ngữ C có một toán tử đặc biệt gọi là toán tử phẩy, ký hiệu là "," toán tử này cho phép tạo ra biểu thức phẩy gồm hai biểu thức cách nhau bằng dấu phẩy, việc tính toán biểu thức phẩy sẽ được thực hiện từ trái sang phải, kết quả của cả biểu thức phẩy sẽ là giá trị và kiểu của kết quả của biểu thức bên phải.

Cú pháp:

`biểu_thức_1, biểu_thức_kết_quả`

Với `biểu_thức_1` và `biểu_thức_kết_quả` là hai biểu thức bất kỳ.

Ví dụ 8.73

`m = (a = 2, t = a + 3);` sẽ cho `a = 2, t = 5` và `m = t = 5`

hoặc `x = (t = 1, t + 4);` sẽ cho `t = 1` và `x = 5`

8.6.8 Toán tử điều kiện - biểu thức điều kiện

Trong ngôn ngữ C có một toán tử khá đặc biệt gọi là toán tử điều kiện, ký hiệu của toán tử điều kiện là hai dấu "?" và ":" theo cú pháp sau:

`dieu-kien? bieu-thuc1: bieu-thuc2`

với `dieu-kien` là một biểu thức bất kỳ có kết quả thuộc kiểu chuẩn (scalar type)

`biau-thuc1` và `biau-thuc2`

là hai biểu thức bất kỳ và dĩ nhiên có thể là một biểu thức điều kiện khác.

Ta gọi một biểu thức như vậy là biểu thức điều kiện. Để tính toán biểu thức điều kiện, điều kiện sẽ được tính toán trước, nếu khác 0, `biau-thuc1` sẽ được tính và được lấy làm kết quả, còn nếu bằng 0 thì kết quả sẽ là kết quả của `biau-thuc2`.

Ví dụ 8.74

Thay vì phải viết dài dòng

`if (i > 0)`

`n = 1;`

`else`

`n = 0;`

ta chỉ cần dùng biểu thức điều kiện

`n = (i > 0)? 1: 0;`

chương trình sẽ trở nên rất gọn và dễ nhìn.

Ví dụ 8.75 Viết chương trình nhập một ký tự, đổi ký tự đó sang ký tự hoa nếu đó là ký tự thường.

```
#include <stdio.h>
#include <conio.h>

main()
{
    char c;
    clrscr();
    printf ("Nhập một ký tự: ");
    c = getchar(); (1)
    c = (c >= 'a' && c <= 'z')? c - 32: c; (2)
    printf ("Ký tự đã được đổi là: ");
    putchar (c);
    getch();
}
```

Chương trình có sử dụng hai hàm chuẩn là getchar () để nhập ký tự còn putchar () để xuất ký tự, cả hai hàm đều có prototype nằm trong file stdio.h. Hai lệnh được đánh dấu (1) và (2) trong chương trình có thể được viết gọn lại như sau:

$c = ((c = getchar()) >= 'a' \&\& c <= 'z')? c - 32: c;$

hoặc hơn nữa

$((c = getchar()) >= 'a' \&\& c <= 'z')? c - 32: c;$

Đây có thể nói là một toán tử rất lợi hại của C, nếu biết tận dụng nó một cách thích hợp chương trình sẽ trở nên rất gọn, tuy nhiên nếu tận dụng thái quá thì chương trình có thể trở nên hơi khó nhìn, điều này tùy lập trình viên.

8.6.9 Toán tử sizeof

Toán tử cuối cùng của C mà ta sẽ nói đến ở đây là sizeof. Đây là một toán tử cho ta kích thước của một biến hoặc một kiểu dữ liệu nào đó. Do phạm vi sử dụng của sizeof rất rộng và thường được dùng để lấy kích thước các kiểu dữ liệu phức hợp như struct, union... Việc sử dụng toán tử này cho phép ta không phải quan tâm đến chiều dài cụ thể của các biến.

Toán hạng của sizeof là một biến hoặc một kiểu dữ liệu bất kỳ nào đó đã định nghĩa. Toán tử này được dùng dưới dạng:

`sizeof (biến)`

`sizeof biến`

`sizeof (kiểu)`

Kết quả của toán tử này là một giá trị nguyên chỉ kích thước (tính bằng byte hoặc char) của kiểu dữ liệu hoặc của biến đó. Biến hoặc kiểu này có thể là một biến hoặc một kiểu đơn giản hay phức hợp.

Ví dụ 8.76

Nếu có một biến đã khai báo:

`double f;`

thì:

`sizeof (f)`

`sizeof f`

hoặc:

`sizeof (double)`

sẽ cho ta kích thước của biến `f` hoặc của một số `double` nói chung, trong trường hợp này trị số là 8 (byte).

Toán tử này có thể được sử dụng kết hợp với các hàm `malloc()`, hoặc `calloc()` để cấp phát biến động, vì các hàm này đòi hỏi kích thước vùng cần cấp tính bằng byte.

Ví dụ 8.77

`int *p;`

`p = (int *) malloc (10 * sizeof(int));`

thì `10*sizeof(int)` là kích thước của 10 số `int`.

Ví dụ 8.78 Hãy xét đoạn chương trình sau: in ra kích thước của một biến có kiểu struct date.

```

struct date
{
    int day;
    int month;
    int year;
    char weekday [4];
} d;
main()
{
    int dsize;
    dsize = sizeof (struct date);
    printf ("Kich thuoc cua struct date la %d byte\n", dsize);
    dsize = sizeof d;
    printf ("Kich thuoc cua bien d la %d byte\n", dsize);
}

```

Toán tử sizeof cũng có một độ ưu tiên nhất định so với các toán tử khác, do đó phải đóng ngoặc đầy đủ khi làm việc với nhiều toán tử khác.

Toán tử sizeof cũng có thể lấy kích thước của một mảng, kết quả của nó là số byte của mảng cho dù mảng có phần tử thuộc bất kỳ kiểu gì. Cũng có thể lấy kích thước của một biến pointer.

Toán tử sizeof được sử dụng đặc biệt cho trường hợp xin cấp phát động một vùng nhớ, vì khi đó cần phải biết số lượng byte cần thiết để có thể cấp phát. Việc sử dụng sizeof sẽ giúp cho chúng ta không phải bận tâm về cách sắp xếp và chiều dài của từng kiểu dữ liệu... nhất là trong trường hợp các chương trình biên dịch tự động sắp xếp các thành phần, thêm các byte để đệm cho chẵn số byte.

8.6.10 Độ ưu tiên của các toán tử

Trong một biểu thức, các phép toán luôn được thực hiện theo một mức độ ưu tiên khác nhau, và nếu cùng một độ ưu tiên thì các phép toán sẽ được kết hợp với nhau theo một trật tự nhất định. Như vậy, việc năm được độ ưu tiên và thứ tự kết hợp này của các toán tử rất quan trọng vì nó ảnh hưởng đến kết quả tính toán của biểu thức.

Dưới đây là bảng ưu tiên và thứ tự kết hợp của các toán tử của C được phân theo từng phân lớp từ phân lớp 1 (có độ ưu tiên cao nhất) đến phân lớp 15 (có độ ưu tiên thấp nhất), mỗi phân lớp có thể có nhiều toán tử và các toán tử này đều ngang cấp nhau. Nếu trong một biểu thức chỉ có các toán tử ngang cấp nhau thì thứ tự kết hợp là từ trái sang phải, trừ ba phân lớp 2, 13 và 14 là từ phải sang trái.

Bảng 8.6 Độ ưu tiên của các toán tử và thứ tự kết hợp trong C

Độ ưu tiên	Phép toán	Thứ tự kết hợp
1	$() [] ->$	Trái qua phải
2	$! ~ ++ -- \rightarrow (\text{type})^* \& \text{sizeof}$	Phải qua trái *
3	$* / %$	Trái qua phải
4	$+ -$	Trái qua phải
5	$<< >>$	Trái qua phải
6	$< <= > >=$	Trái qua phải
7	$== !=$	Trái qua phải
8	$\&$	Trái qua phải
9	$^$	Trái qua phải
10	$ $	Trái qua phải
11	$\&\&$	Trái qua phải
12	$\ $	Trái qua phải
13	$? :$	Trái qua Phải *
14	$= += -= *= /= \% = <<= >>= \&= = ^=$	Phải qua trái *
15	$,$	Trái qua phải

Lưu ý rằng lớp toán tử thứ hai gọi là lớp toán tử đơn toán hạng, vì chúng chỉ cần một toán hạng trong phép toán của chúng.

Ví dụ 8.79

$i = b;$

$a = +t;$

8.7 CẤU TRÚC TỔNG QUÁT CỦA MỘT CHƯƠNG TRÌNH C

Một chương trình C tổng quát bao hàm hai phần: phần khai báo đầu (*header*) và phần hàm (*function*).

Phần khai báo đầu của một chương trình C bao gồm:

- Các lệnh tiền xử lý: include, define...
- Các khai báo hằng, biến ngoài...
- Các prototype của các hàm được sử dụng trong chương trình

Phần hàm của một chương trình C là phần định nghĩa các hàm sử dụng trong chương trình, trong các hàm này phải có hàm main().

Ngoài ra, khi lập trình cụ thể có thể lập trình viên sẽ thêm hoặc bớt các phần khi cần thiết. Để hình dung, ta hãy xét cụ thể một chương trình.

Ví dụ 8.80 Nhập một số kiểm tra số đó là chẵn hay lẻ.

```
#include <stdio.h>
#include <conio.h>
int kiem_tra (int so);
/* ham kiem_tra nhan vao doi so
   la mot so nguyen, tra ve tri
   - 0 la so chan
   - 1 la so le
*/
main()
{
    int n;
    clrscr();
    printf ("Nhap mot so: ");
    scanf ("%d", &n);
    if (kiem_tra(n))
        printf ("So da nhap la so le \n");
    else
        printf ("So da nhap la so chan \n");
    getch();
}
int kiem_tra (int so)
{
    return (so % 2 == 0)? 0:1;
}
```

→ phần khai báo

→ phần hàm

BÀI TẬP CUỐI CHƯƠNG

- 8.1 Viết chương trình in ra màn hình trị thập phân của các hằng sau đây

067	01234	0x1al	0x89ad
0xfb	'h'	022	02365

- 8.2 Nhập ba số, tìm số lớn nhất và nhỏ nhất trong ba số đó.
- 8.3 Nhập bốn số, sắp xếp theo thứ tự từ lớn tới nhỏ và từ nhỏ tới lớn theo menu sau:

1. Từ lớn tới nhỏ
2. Từ nhỏ tới lớn
3. Kết thúc

Mời bạn chọn thao tác (1...3):

- 8.4 Nhập ba cạnh tam giác, kiểm tra ba cạnh đó có thỏa điều kiện hình thành tam giác không, in kết quả kiểm tra.
- 8.5 Nhập ba cạnh tam giác, kiểm tra ba cạnh đó có thỏa điều kiện hình thành tam giác không, nếu thỏa in ra kết quả xem tam giác đó là tam giác gì (vuông thì vuông tại đâu, cân thì cân tại đâu...)?
- 8.6 Viết chương trình nhập một ký tự và xử lý theo yêu cầu sau:
- Nếu ký tự là hoa thì đổi sang thường, in kết quả đổi
 - Nếu ký tự là thường thì không làm gì cả, in kết quả
 - Nếu ký tự là ký số thì in ra màn hình câu: "Day la mot ky so".
- 8.7 Dùng hàm pow () để tính bình phương và lũy thừa ba của một số nhập từ bàn phím.
- 8.8 Viết chương trình đổi từ độ Fahrenheit (F) sang độ Celcius (C) theo công thức sau:

$$\frac{F - 32}{C} = \frac{9}{5}$$

CÁC LỆNH ĐIỀU KHIỂN VÀ VÒNG LẶP

9.1 LỆNH ĐƠN VÀ LỆNH PHỨC (*Simple statement* và *compound statement*)

Trong ngôn ngữ C có hai loại lệnh (hay mệnh đề): lệnh đơn và lệnh phức.

- Lệnh đơn được tạo ra từ một biểu thức thuộc loại bất kỳ hoặc là một lệnh điều khiển của C (break, continue, ...) theo sau nó là một dấu chấm phẩy (;). Nếu lệnh đơn được tạo từ biểu thức thì được gọi là lệnh biểu thức.

Ví dụ 9.1 Các lệnh sau đây là các lệnh đơn

```
a = a + 1;  
b >>= 3;  
printf (...);  
break;
```

đây là bốn lệnh riêng biệt, ba lệnh đầu là lệnh biểu thức, mỗi lệnh đều kết thúc bằng một dấu ";".

- Lệnh phức bao hàm một hay nhiều lệnh đơn được bao bên trong cặp dấu ngoặc nhọn ({}) và được bộ dịch C xem như là một lệnh tương đương (với lệnh đơn). Các lệnh phức này thường được dùng trong các câu lệnh điều khiển và vòng lặp của C để xác định lệnh thực thi của các lệnh điều khiển này.

Ví dụ 9.2 Xét lệnh if sau

```
if (a > 0)  
{  
    i += 2;  
    a++;  
    n = a * i;  
}
```

→ *lệnh phức, được xem là một lệnh*

Chú ý: nếu trên một hàng trong chương trình C chỉ có một dấu chấm phẩy (;) thì C vẫn xem đó là một lệnh, nhưng lại là lệnh rỗng (*null statement*), tức lệnh không làm gì cả. Lệnh này nhiều khi có ý nghĩa quan trọng trong giải thuật của chúng ta, nó bảo đảm tính cấu trúc của lệnh và giải thuật của chương trình. Lệnh này sẽ được đề cập kỹ hơn ở cuối chương này.

Khi một chương trình C được thực thi, các lệnh sẽ được thực hiện từ lệnh này đến lệnh khác theo các cấu trúc điều khiển chương trình. Nếu lệnh thực thi là lệnh đơn thì biểu thức sẽ được tính toán, hoặc hàm sẽ được thực hiện, còn nếu là lệnh phức thì việc tính toán sẽ được thực hiện với từng lệnh con một trong lệnh phức đó, việc tính toán theo trình tự từ trên xuống dưới.

Để phối hợp các lệnh đơn và phức theo một trình tự hợp logic để giải quyết bài toán, C đã thiết kế sẵn các cấu trúc điều khiển chương trình (mà ta thường gọi là các lệnh điều khiển chương trình), mỗi lệnh đều có cú pháp riêng của nó. Các lệnh điều khiển này có thể được chia ra làm hai nhánh:

- Nhóm lệnh liên quan đến việc rẽ nhánh chương trình: if-else, switch-case, goto,...
- Nhóm lệnh lặp: while, for, do_while

Do đó, việc nắm vững cú pháp của từng lệnh để ứng dụng vào lập trình là điều hết sức cần thiết.

9.2 LỆNH IF

Lệnh **if** cho phép lập trình viên thực hiện một lệnh đơn hay một lệnh phức tùy theo biểu thức điều kiện, nếu biểu thức có trị khác 0 thì lệnh được thực thi.

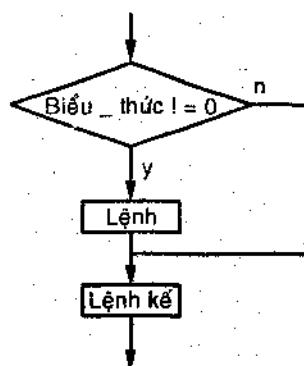
Cú pháp của lệnh if như sau:

Dạng 1:

if (biểu_thức)

 lệnh

Lưu đồ như hình 9.1 sau:



Hình 9.1 Lưu đồ dạng 1 của lệnh if

Với:

- Điều kiện **Biểu_thức** là một biểu thức bất kỳ, có thể có hằng, biến hoặc gọi hàm trong đó và sau cùng là biểu thức này sẽ có trị 0 hoặc khác 0 (tương ứng với hai trạng thái false và true của điều kiện).
- **Lệnh** là lệnh thực thi của if, có thể là lệnh đơn, phức hoặc lệnh rỗng.

Khi gặp lệnh này, biểu thức điều kiện **Biểu_thức** sẽ được tính toán, nếu kết quả khác 0 thì lệnh **lệnh** sẽ được thực hiện, còn nếu kết quả bằng 0 thì chương trình sẽ bỏ qua **lệnh** này.

Dạng 2:

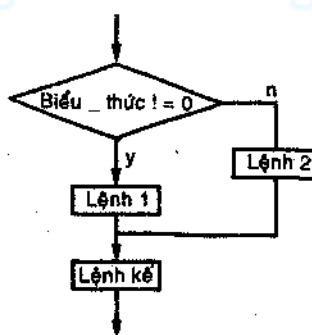
```
if (Biểu_thức)
```

```
  Lệnh_1
```

```
else
```

```
  Lệnh_2
```

Lưu đồ tương ứng trong hình 9.2 như sau:



Hình 9.2 Lưu đồ dạng 2 của lệnh if

Với:

- điều kiện **Biểu_thức** là một biểu thức bất kỳ, có thể có hằng, biến hoặc gọi hàm và biểu thức này sẽ có trị 0 hoặc khác 0 (tương ứng với hai trạng thái false và true của điều kiện).
- **Lệnh_1, Lệnh_2** là lệnh thực thi của if, có thể là lệnh đơn, phức hoặc lệnh rỗng.

Khi gặp lệnh này, biểu thức điều kiện **Bieu_thuc** sẽ được tính toán, nếu kết quả khác 0 thì lệnh **Lệnh_1** sẽ được thực hiện, còn nếu kết quả bằng 0 thì **Lệnh_2** sẽ được thực hiện.

Đĩ nhiên, trong cả hai trường hợp dấu chấm phẩy kết thúc lệnh trong **Lệnh_1** và **Lệnh_2** chỉ cần thiết khi lệnh là lệnh đơn hoặc lệnh rỗng.

Cũng cần lưu ý là thông thường các biểu thức là điều kiện trong lệnh if được sử dụng với các toán tử so sánh ($>$, \geq , $<$, ...). Trong các toán tử này, toán tử so sánh bằng ($=$) thường được viết chỉ với một dấu '='; điều này dẫn tới sai sót trong giải thuật của chương trình như ví dụ dưới đây.

Ví dụ 9.3

```
if (x = 2)      /* Điều kiện này luôn luôn đúng */
    y = 5;        /* Lệnh này luôn luôn được thực thi */
```

Trong ví dụ trên, lệnh if sẽ kiểm tra và thấy biểu thức **x=2** luôn khác 0, tức điều kiện luôn đúng, và lệnh **y = 5;** luôn được thực thi, tức lệnh if là vô nghĩa (!). Trong thực tế, khi gặp lệnh if như thế này, các bộ dịch C thường sẽ cảnh báo để chúng ta xem xét lại nếu cần. Và trong trường hợp này, chúng ta sẽ sửa lại như ví dụ sau.

Ví dụ 9.4

```
if (x == 2)
    y = 5;
```

Chúng ta hãy xem mã LC-3 được tạo ra cho đoạn code trên, chấp nhận rằng hai biến **x** và **y** là hai biến nguyên đã được khai báo cục bộ. Điều này có nghĩa là thanh ghi R5 sẽ chỉ tới biến **x**, và R5-1 sẽ chỉ tới biến **y**.

```

LDR R0, R5, #0 ; nạp x vô R0
ADD R0, R0, #-2 ; trừ R0, tức x, cho 2
BRnp NOT_EQUAL ; nếu R0, tức x, không bằng 2
AND R0, R0, #0 ; gán 0 vô R0
ADD R0, R0, #5 ; gán 5 vô R0
STR R0, R5, #-1 ; y = 5
NOT_EQUAL: ; phần còn lại của chương trình

```

Ví dụ 9.5 Xét chương trình sau đây

```

#include <stdio.h>
#include <conio.h>
main()
{
    int n;
    clrscr();
    printf ("Moi nhap mot so: ");
    scanf ("%d", &n);
    if (n % 2 == 0)
        printf ("So la so chan \n");
    printf ("Moi ban nhan mot phim de ket thuc \n");
    getch();
}

```

Chương trình trên sẽ kiểm tra một số có phải là số chẵn hay không, nếu là số chẵn thì xác nhận, còn không thì thôi.

Ta hãy xét một số ví dụ khác cho dạng 2, if-else và dạng LC-3 của nó.

Ví dụ 9.6 Xét đoạn chương trình sau

```

if (x)
{
    y++;
    z--;
}
else
{
    y--;
    z++;
}

```

Trong đoạn chương trình này, x , y , và z là ba biến cục bộ đã được khai báo với kiểu nguyên. Nếu trị x khác 0, điều kiện của lệnh if là đúng, y được tăng lên, và z được giảm đi. Ngược lại, y giảm đi, và z tăng lên. Dạng LC-3 tương ứng của nó:

LDR R0, R5, #0	; lấy giá trị của x
BRz ELSE	; nếu x bằng 0, nhảy tới phần else
	; x khác 0
LDR R1, R5, #-1	; lấy trị từ y
ADD R1, R1, #1	; tăng lên 1
STR R1, R5, #-1	; chứa vào y
LDR R1, R5, #-2	; lấy trị từ z
ADD R1, R1, #1	; tăng thêm 1
STR R1, R5, #-2	; chứa vào z
JMP DONE	; lệnh kế, sau if
	; x bằng 0
ELSE: LDR R1, R5, #1	; nạp y vào R1
ADD R1, R1, #-1	; giảm đi 1
STR R1, R5, #-1	; chứa vào y
LDR R1, R5, #-2	; lấy z
ADD R1, R1, #1	; tăng lên 1
STR R1, R5, #-2	; chứa vào z
DONE: ...	; lệnh kế

Ví dụ 9.7 Xét chương trình sau đây

```
#include <stdio.h>
#include <conio.h>
main()
{
    int n;
    clrscr();
    printf ("Moi nhap mot so: ");
    scanf ("%d", &n);
    if (n % 2 == 0)
        printf ("So la so chan \n");
    else
        printf ("So la so le \n");
    printf ("Moi ban nhan mot phim de ket thuc \n");
    getch();
}
```

vẫn có dấu chấm phẩy

Chương trình trên sẽ kiểm tra một số có phải là số chẵn không, nếu là số chẵn thì báo chẵn, còn ngược lại thì báo là số lẻ.

Cần chú ý rằng, trong dạng 2 của lệnh if, else sẽ luôn kết hợp với if gần nhất phía trên và chưa có else. Do đó, khi sử dụng if-else lồng nhau thì phải lưu ý việc kết hợp giữa các if và các else cho đúng với giải thuật bài toán.

Ví dụ 9.8 Hãy xem hai đoạn chương trình sau:

```

if (a > 0)           if (a > 0)
    if (b > 0)       {
        c = b + a;
    else             if (b > 0)
        c = b - a;
    c = b - a;       }
else
    c = b - a;

```

Hai đoạn chương trình này sẽ được thực thi hoàn toàn khác nhau, dù cho ta đã cố ý viết else ngang với if để biểu thị cặp if-else cần phải đi với nhau, vì C thực sự không quan tâm đến điều đó. Trong ví dụ trên bên trái, else sẽ được C hiểu là của lệnh if ($b > 0$) vì đây là lệnh if gần nó nhất mà chưa có else, và điều này là một sai lầm về giải thuật vì ta muốn else là của if ($a > 0$). Muốn diễn tả

```

else
    c = b - a;

```

là một phần của lệnh

```
if (a > 0)
```

ta phải bao lệnh

```
if (b > 0)
```

```
c = b + a;
```

trong cặp () để biết đây là một lệnh phức của lệnh if ($a > 0$) như cách viết bên phải của ví dụ trên.

Khi kết hợp nhiều lệnh if-else lại với nhau, chúng ta có một dạng lệnh điều kiện là if-else-if cho phép rẽ nhánh chương trình theo nhiều điều kiện khác nhau. Cú pháp của lệnh này như sau:

```

if (biểu_thức_1)
    lệnh_1
else if (biểu_thức_2)
    lệnh_2
else if (biểu_thức_3)
    lệnh_3
.....
else
    lệnh_n

```

Khi thực hiện lệnh if_else lồng nhau như thế này các biểu thức sẽ được tính lần lượt từ trên xuống dưới nếu có biểu thức nào khác 0, lệnh tương ứng với if đó sẽ được thi hành và toàn bộ phần còn lại của lệnh if-else được bỏ qua.

Ví dụ 9.9 Chương trình ví dụ sau nhập vào một ký tự, kiểm tra ký tự đó là thường, hoa, ký số hoặc ký tự kết thúc file hay ký tự khác.

```

#include <stdio.h>
#include <conio.h>
main()
{
    char c;
    clrscr();
    printf ("Nhập một ký tự: ");
    c = getchar();
    if (c == EOF)
        printf ("Đã đến cuối file \n");
    else if (c >= 'a' && c <= 'z')
        printf ("ký tự thường\n");
    else if (c >= 'A' && c <= 'Z')
        printf ("ký tự hoa\n");
    else if (c >= '0' && c <= '9')
        printf ("ký tự số\n");
    else
        printf ("ký tự khác\n");
    getch();
}

```

9.3 LỆNH SWITCH-CASE

Đây cũng là một lệnh rẽ nhánh có điều kiện, lệnh này được sử dụng khi giá trị của biểu thức cần kiểm tra có bằng với một trị cụ thể nào đó được nêu trong danh sách các hằng không, nếu bằng thì lệnh tương ứng sẽ được thực thi.

Cú pháp của lệnh như sau:

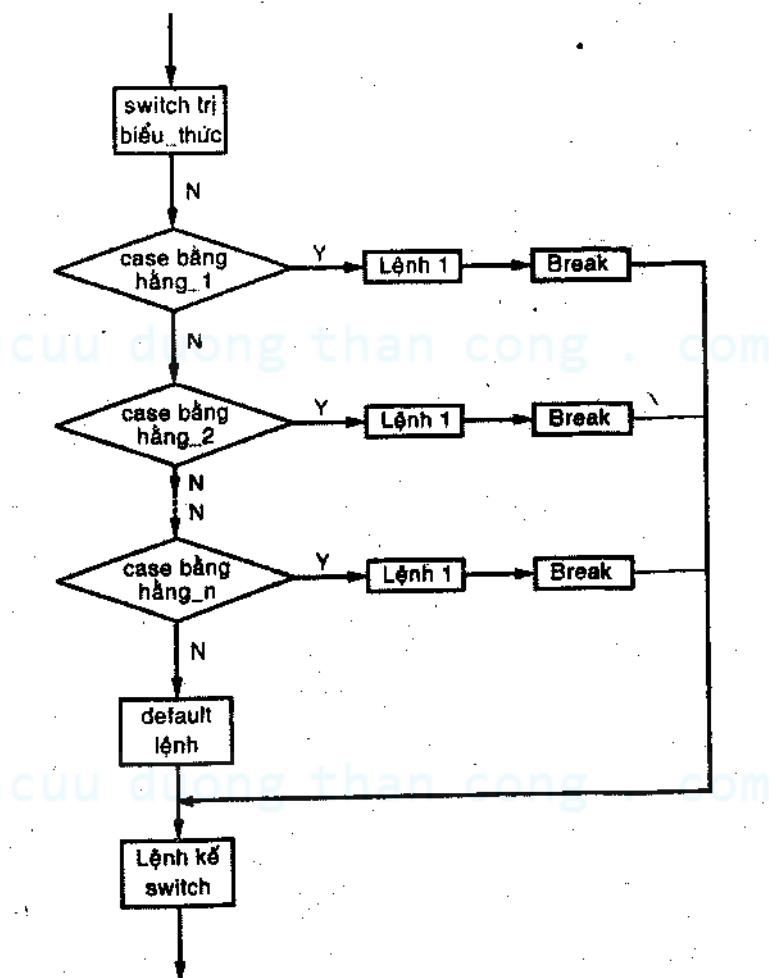
```
switch (biểu_thức)
{
    case hằng_1:
        lệnh_1;
        break;
    case hằng_2:
        lệnh_2;
        break;
    case hằng_n:
        lệnh_n;
        break;
    default:
        lệnh;
        break;
}
```

Với:

- **biểu_thức** là một biểu thức sau khi tính toán có kết quả thuộc kiểu nguyên (char, int, long, unsigned...).
- **hằng_1,...** là những hằng số, hằng ký tự, hoặc hằng biểu thức có kết quả nguyên, và chúng không trùng nhau.
- các lệnh **lệnh_1**, **lệnh_2**..., **lệnh_i**, **lệnh_n** là các lệnh đơn hoặc tương đương của từng trường hợp cụ thể, trong một **case** như vậy có thể có nhiều lệnh thực thi bất kỳ và kết thúc các lệnh này là lệnh **break** để nhảy ra khỏi ra khỏi **switch**.

Khi gặp lệnh **switch**, đầu tiên biểu thức sẽ được tính, nếu kết quả phép tính này bằng với một trị **hằng_i** nào đó được liệt kê ra trong các **case** thì việc thực hiện lệnh sẽ bắt đầu từ **lệnh_i** tương ứng, tuần tự xuống cho đến khi gặp một lệnh **break** thì nhảy ra khỏi lệnh **switch** hoặc cho đến khi hết **switch**, thì việc thực hiện lệnh **switch** chấm dứt, chương trình sẽ thực hiện xuống các lệnh tiếp theo sau **switch**.

Lưu đồ lệnh switch được thể hiện trong hình 9.3.



Hình 9.3 Lưu đồ lệnh switch

Nếu khi so sánh trị của **biểu_thức** switch với các **hằng_i** mà không có một **hằng_i** nào bằng với trị của **biểu_thức** cả thì lệnh

switch sẽ kết thúc, còn nếu trong lệnh **switch** có **default** thì lệnh trong **default**, lệnh sẽ được thực thi. Lệnh **default** có thể không có trong **switch** và nếu có thì có thể nằm tùy ý: đầu, giữa hoặc cuối các **case**.

Ví dụ 9.10 Viết chương trình nhập một trị, nếu trị đó chia hết cho 5 thì cộng thêm 5 vào cho số đó, nếu trị đó chia cho 5 dư 1 thì cộng thêm 1, tương tự cho 3, nếu là số khác thì báo không thỏa.

```
#include <stdio.h>
#include <conio.h>
main()
{
    int so;
    clrscr();
    printf ("Nhap mot so: ");
    scanf ("%d", &so);
    switch (so % 5)
    {
        case 0:
            so += 5;
            printf ("Tri la: %d\n", so);
            break;
        case 1:
            so += 1;
            printf ("Tri la: %d\n", so);
            break;
        case 3:
            so += 3;
            printf ("Tri la: %d\n", so);
            break;
        default:
            printf ("Khong thoai\n");
            break;
    }
    getch();
}
```

Lệnh **break** cuối mỗi case sẽ chuyển điều khiển chương trình ra khỏi lệnh switch. Nếu không có break, các lệnh tiếp ngay sau sẽ được thực thi dù các lệnh này có thể là của một case khác.

Ví dụ 9.11 Xét ví dụ nhập tháng và năm, kiểm tra số ngày trong tháng.

```
switch (thang)
{
    case 4:
    case 6:
    case 9:
    case 11:
        so_ngay = 30;
        break;
    case 2:
        if (nam % 4 == 0)
            so_ngay = 29;
        else
            so_ngay = 28;
        break;
    default:
        so_ngay = 31;
        break;
}
printf("Thang %d nam %d co %d ngay\n", thang, nam, so_ngay);
```

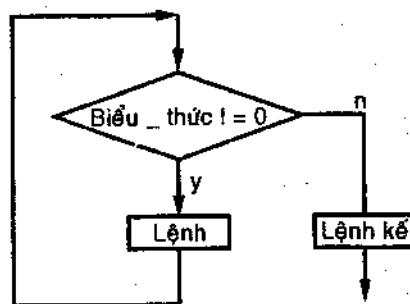
Trong ví dụ trên, ta giả sử rằng hai biến thang và nam đã được nhập trị phù hợp và điều kiện năm nhuận đơn giản là năm chia hết cho 4.

9.4 LỆNH WHILE

Đây là lệnh lặp đầu tiên mà ta sẽ đề cập trong chương này. Có thể nói while là lệnh lặp cơ bản của ngôn ngữ lập trình có cấu trúc, nó cho phép chúng ta lặp lại một lệnh hay một nhóm lệnh **trong khi** điều kiện còn đúng (true-tức khác 0). Cú pháp của lệnh while:

while (bieu_thuc) lenh

Lưu đồ được trình bày trong hình 9.4 dưới đây.

**Hình 9.4 Lưu đồ lệnh while**

Với:

- **biểu_thức** là một biểu thức bất kỳ trong C, nó sẽ có một trong hai trị bằng 0 hoặc khác 0.
- **lệnh** là lệnh thực thi trong while có thể là lệnh đơn hay lệnh phức hoặc lệnh rỗng.

Ví dụ 9.12 Chương trình sau sử dụng vòng lặp while để đếm và in ra trị từ 0 tới 9.

```
#include <stdio.h>
main()
{
    int x = 0;
    while (x < 10)
    {
        printf("%d ", x);
        x = x + 1;
    }
}
```

<http://cuuduongthancong.com>

Mã LC-3 tương ứng cho chương trình trên là:

AND R0, R0, #0	
STR R0, R5, #0	; gán x = 0
	; kiểm tra
LOOP: LDR R0, R5, #0	; lấy x
ADD R0, R0, #-10	; cộng -10 vào R0
BRzp DONE	; nếu x-10 >= 0, hay x >= 10 đúng, tới DONE
	; thàn vòng lặp

```

LDR R0, R5, #0      ; lấy x để sử dụng: in ra
...
<printf>
...
ADD R0, R0, #1      ; tăng x
STR R0, R5, #0      ; chứa vào x
JMP LOOP            ; kiểm tra tiếp
DONE:               ; lệnh kế

```

Ví dụ 9.13 Chương trình sau đây sẽ in ra màn hình 10 số ngẫu nhiên từ 0 đến 99.

```

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>>
#include <time.h>
main()
{
    int i = 1;
    clrscr();
    randomize();
    printf ("So ngau nhien trong khoang 0-99 la: ");
    while (i <= 10)
    {
        printf ("%d ", random(100));
        i++;
    }
    getch();
}

```

Trong chương trình trên hàm randomize() có chức năng khởi động bộ phát số ngẫu nhiên để sử dụng hàm random(), hàm random() với đối số là n sẽ trả về một trị ngẫu nhiên trong khoảng 0 tới $n - 1$, lệnh while sẽ kiểm tra điều kiện nếu i nhỏ hơn hay bằng 10 thì in ra số ngẫu nhiên, nếu điều kiện sai thì lệnh while sẽ không lặp được nữa, chương trình sẽ kết thúc.

Trong ví dụ trên chúng ta có thể viết lại vòng lặp trên với i giảm dần từ 10 về 0 như sau:

```

int i = 10;
clrscr();
randomize();
printf ("So ngau nhien trong khoang 0-99 la: ");
while (i)
{
    printf ("%d", random(100));
    i--;
}

```

Nếu **bieu_thuc** là một hằng số khác 0 (số đó có thể là âm, dương hoặc lẻ) thì vòng lặp **while** coi như lặp mãi mãi, khi đó chỉ có thể thoát ra khỏi while bằng các lệnh break để sang lệnh kế tiếp.

Ví dụ 9.14 Nhập các ký tự cho đến khi nào nhận được ký tự ESC có mã ASCII là 27 thì kết thúc chương trình.

```

#include <stdio.h>
#include <conio.h>
#define ESC 27
main()
{
    char c;
    clrscr();
    printf ("Cac ky tu duoc nhap la: ");
    while (1)
    {
        c = getch();
        if (c == ESC)
            break;
    }
}

```

Chú ý rằng, lệnh thực thi của lệnh while cũng có thể là lệnh **rỗng**, khi đó thường lệnh thực thi của while đã nằm trong biểu thức điều kiện, do đó ví dụ trên có thể được viết lại như sau:

```
#include <stdio.h>
```

```

#include <conio.h>
#define ESC 27
main() {
    char c;
    clrscr();
    printf ("Cac ky tu duoc nhap la: ");
    while (getche() - ESC)
        ;
}

```

lệnh thực thi rỗng

Biểu thức **getche()** - ESC cho phép nhận ký tự và trừ ký tự đó (thực chất là lấy mã ASCII của ký tự đó) với 27 (là mã ASCII của phím ESC), nếu trị của biểu thức này khác 0, tức ký tự nhập không là phím ESC, thì lệnh while sẽ lặp lệnh rỗng, thực tế là không làm gì cả, và sau đó quay lên kiểm tra điều kiện, thực tế là tính lại biểu thức getche() - ESC, cứ như vậy việc lặp sẽ diễn ra.

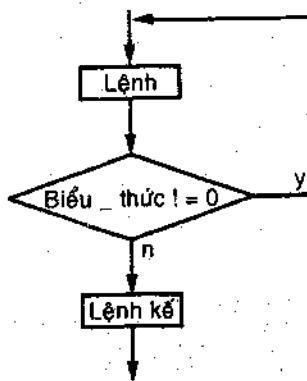
9.5 LỆNH DO-WHILE

Nếu lệnh while cho phép kiểm tra điều kiện trước rồi thực thi lệnh sau, như vậy nếu ngay từ đầu mà điều kiện đã sai thì lệnh của while không được thực thi, thì lệnh lặp do-while lại thực thi lệnh trước rồi mới kiểm tra điều kiện sau.

Cú pháp của lệnh **do-while** như sau:

do
 lệnh
 while (biểu_thức);

Lưu đồ như sau:

**Hình 9.5 Lưu đồ lệnh do-while**

Với:

- điều kiện **biểu_thức** là một biểu thức bất kỳ, có thể có hằng biến hoặc gọi hàm và biểu thức này sẽ có trị bằng 0 hoặc khác 0 (tương ứng với hai trạng thái false và true của điều kiện). Chú ý theo sau while (**biểu_thức**) phải luôn là dấu chấm phẩy.
- **lệnh** là lệnh thực thi của while, có thể là lệnh đơn, phức hoặc lệnh rỗng.

Đầu tiên, lệnh trong do_while được thực thi, sau đó **biểu_thức** sẽ được tính, nếu kết quả khác 0 thì lệnh lại được thực thi trở lại, cứ thế tiến trình lặp sẽ diễn ra đến khi điều kiện sai.

Ví dụ 9.15 Viết chương trình cho phép kiểm tra và in ra thông báo cho biết phím mũi tên nào đã được nhấn.

```

#include <stdio.h>
#include <conio.h>
#define ESC 27
main()
{
    char c;
    clrscr();
    printf ("\n Moi an cac phim mui ten \n");
    do
    {
        c = getch();
        if (c == ESC)
            break;
        else
            printf ("Phim ban da nhap la %c\n", c);
    }
}
  
```

```

if (c == 0)
{
    c = getch();
    switch(c)
    {
        case 'H':
            printf ("Ban da an mui ten len\n");
            break;
        case 'P':
            printf ("Ban da an mui ten xuong\n");
            break;
        case 'K':
            printf ("Ban da an mui ten qua trai\n");
            break;
        case 'M':
            printf ("Ban da an mui ten qua phai\n");
            break;
    } /* end switch */
}
while (c != 27);
}

```

Chú ý rằng mỗi phím mũi tên khi được ấn đều sinh ra hai ký tự: ký tự đầu luôn là ký tự có mã ASCII là 0 (tức ký tự NUL), ký tự thứ hai là các mã ASCII tương ứng với phím, trong ví dụ trên thì

- + Phím mũi tên lên (\uparrow) có mã là 0 và 'H'
- + Phím mũi tên xuống (\downarrow) có mã là 0 và 'P'
- + Phím mũi tên qua trái (\leftarrow) có mã là 0 và 'K'
- + Phím mũi tên (\rightarrow) có mã là 0 và 'M'.

và tùy trường hợp mũi tên được ấn là gì thì chương trình sẽ in ra câu thông báo phù hợp.

Vòng lặp do-while thường được kết hợp với switch-case để viết ra các chương trình điều khiển menu hay các chương trình trắc nghiệm, trong đó việc chọn câu trả lời chỉ có thể chọn trong các câu được ra mà thôi. Để rõ hơn ta hãy xét ví dụ sau đây:

Ví dụ 9.16

```
#include <stdio.h>
#include <conio.h>
main()
{
    char c;
    int danh_gia = 0;
    textbackground (BLUE);
    clrscr();
    cprintf ("Cau 1: Xin ban cho biet ban thich mau gi:\n\r");
    cprintf ("      1. Vang\n\r");
    cprintf ("      2. Xanh\n\r");
    cprintf ("      3. Tim \n\r");
    cprintf ("      4. Do \n\r");
    /* an mot trong cac ky so 1..4 */
    do
    {
        c = getch();
        switch (c)
        {
            case '1':
                danh_gia += 1;
                break;
            case '2':
                danh_gia += 2;
                break;
            case '3':
                danh_gia += 3;
                break;
            case '4':
                danh_gia += 4;
                break;
        }
    }
    while (c != '1' && c != '2' && c != '3' && c != '4');
    cprintf ("Cau 2: Xin ban cho biet mon ban thich hon ca:\n\r");
    cprintf ("      1. Hanh \n\r");
```

```

        printf (" 2. Toi \n\r");
        printf (" 3. Tieu \n\r");
        printf (" 4. Ot \n\r");
/* an mot trong cac ky so 1..4 */
do
{
    c =getch();
    switch(c)
    {
        case '1':
            danh_gia += 1;
            break;
        case '2':
            danh_gia += 2;
            break;
        case '3':
            danh_gia += 3;
            break;
        case '4':
            danh_gia += 4;
            break;
    }
}
while (c != '1' && c != '2' && c != '3' && c != '4');
if ( danh_gia >= 2 && danh_gia <= 4 )
    printf ("Ban la nguoi thoai mai, khong thich dieu sai\n\r");
else if ( danh_gia >= 4 && danh_gia <= 6 )
    printf ("Ban la nguoi nhanh thanh kien\n\r");
getch();
}

```

Trong chương trình ví dụ trên, vòng do-while cho phép nhận các ký tự từ '1' đến '4', vòng lặp này chỉ thoát ra khi ta chọn đúng các ký tự quy định, như vậy ta cần phải có ký tự nhập rồi mới kiểm tra điều kiện xem ký tự nhập có đúng không, vì vậy, việc dùng vòng lặp do-while là dễ hiểu.

Vì vòng lặp do-while cho phép thực thi lệnh rồi mới kiểm tra điều kiện, nếu ngay từ đầu điều kiện đã sai thì các lệnh trong vòng lặp cũng được thực thi một lần, do đó tùy hoàn cảnh mà lập trình viên nên quyết định xem chọn kiểu vòng lặp nào cho thích hợp. Ngoài hai vòng lặp trên C còn có một vòng lặp khác rất mạnh mà ta sẽ đề cập sau đây, đó là vòng lặp for.

9.6 LỆNH FOR

Tương tự như ngôn ngữ PASCAL, trong ngôn ngữ C cũng có vòng lặp **for**, đây cũng là một lệnh lặp cho phép kiểm tra điều kiện trước, giống như while, nhưng lại có thêm nhiều điểm mạnh khác mà lệnh **while** của C (cũng như lệnh for của ngôn ngữ PASCAL) không có. Cú pháp của lệnh for như sau:

for (biểu_thức1; biểu_thức2; biểu_thức3)

lệnh

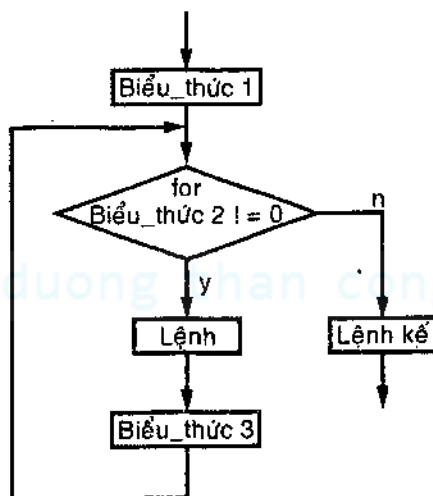
Với: *cuu duong than cong . com*

- **biểu_thức1, biểu_thức2, biểu_thức3** là những biểu thức bất kỳ, có thể là biểu thức phẩy, biểu thức điều kiện, biểu gán hay có sự gọi hàm Tuy nhiên mỗi biểu thức đều có một ý nghĩa riêng khi sử dụng trong for.
- **biểu_thức1** có ý nghĩa là biểu thức để khởi động trị đầu cho biến điều khiển vòng for, nó có thể là biểu thức gán hay biểu thức phẩy, nếu biểu thức này không có thì vòng for không cần khởi động trị đầu trước trước khi lặp.
- **biểu_thức2** có ý nghĩa là biểu thức cho phép kiểm tra xem vòng lặp có được tiếp tục lặp nữa hay không, nếu biểu thức này khác 0 vòng lặp tiếp tục chu kỳ lặp mới, còn ngược lại biểu thức này bằng 0 thì vòng lặp kết thúc. Nếu biểu thức này không có thì vòng lặp for sẽ không có điều kiện để kiểm tra, do đó ta có vòng lặp lặp vô tận, khi đó chỉ có thể nhảy ra khỏi nó bằng lệnh break.
- **biểu_thức3** là biểu thức có ý nghĩa cho phép thay đổi biến điều khiển vòng lặp để vòng lặp tiến dần đến kết thúc. Biểu thức này được tính sau khi các lệnh thực thi trong thân vòng

for được thực hiện xong, nếu biểu thức này không có, vòng for có thể bị lặp vô tận, và khi đó ta chỉ có thể thoát ra khỏi nó bằng lệnh break.

Chú ý rằng, các biểu thức biểu_thức1, biểu_thức2, biểu_thức3 là biểu thức rất tổng quát nên có thể kết hợp các biểu thức tính toán của lệnh thực thi của vòng for vào chúng, khi đó vòng for sẽ trở nên rất ngắn gọn, thậm chí hơi lập dị.

Lưu đồ cho lệnh for được trình bày trong hình 9.6.



Hình 9.6 Lưu đồ lệnh for

Như vậy, có thể nói lệnh for tương đương với nhóm lệnh sau:

```

biểu_thức1;
while (biểu_thức2)
{
    lệnh
    biểu_thức3;
}
  
```

Ta hãy xét một ví dụ đơn giản về lệnh for và đoạn code tương ứng được tạo tự bộ dịch LC-3.

Ví dụ 9.17 Chương trình tính tổng các số nguyên từ 0-9

```
#include <stdio.h>
int main()
{
    int x;
    int sum = 0;
    for (x = 0; x < 10; x++)
        sum = sum + x;
}
```

Đoạn mã LC-3 tương ứng

```
AND R0, R0, #0      ; xóa R0
STR R0, R5, #-1     ; xóa sum về 0
AND R0, R0, #0      ; xóa R0
STR R0, R5, #0      ; xóa x về 0
                      ; kiểm tra
LOOP: LDR R0, R5, #0      ; lấy trị của x
      ADD R0, R0, #-10    ; cộng với -10 để so sánh
      BRzp DONE           ; x >= 10
                      ; thân vòng lặp
      LDR R0, R5, #0      ; lấy trị hiện thời của x
      LDR R1, R5, #-1     ; lấy trị hiện thời của sum
      ADD R1, R1, R0      ; tính tổng sum + x
      STR R1, R0, #-1     ; sum = sum + x
                      ; thay đổi trị
      LDR R0, R5, #0      ; lấy x
      ADD R0, R0, #1      ; x + 1
      STR R0, R5, #0      ; x = x + 1
      BR LOOP
DONE:          -          ; lệnh kế for
```

Ta có thể viết lệnh for một cách mềm dẻo theo các ví dụ sau.

Ví dụ 9.18 Vòng lặp for để tính tổng từ 1 tới n như sau

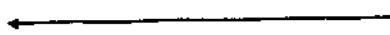
```
s = 0;
for (i = 1; i <= n; i++)
    s += i;
```

Có thể viết ngắn gọn hơn như sau

```
for (i = 1, s = 0; i <= n; i++)
    s += i;
```

trong trường hợp này, ta đã kết hợp biểu thức khởi động trị s và i vào biểu_thức1, lúc này biểu_thức1 là biểu thức phẩy; ta có thể thu gọn hơn nữa vòng lặp trên như sau:

```
for (i = 1, s = 0; i <= n; s += i++)
```

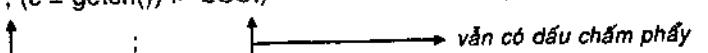


trong trường hợp này lệnh thực thi lại được để trong biểu_thức3 và như vậy lệnh thực thi của vòng for chỉ đơn giản là một lệnh rỗng.

Ví dụ 9.19 Xét chương trình ở ví dụ 9.14, nhưng thiết kế lại theo vòng for.

```
#include <stdio.h>
#include <conio.h>
#define ESC 27
main()
{
    char c;
    clrscr();
    printf ("Cac ky tu duoc nhap la: ");
    for ( ; (c = getch()) != ESC; )
        ;
    }

```



Trong ví dụ trên, ta không cần phải khởi động trị cho biến nào cả, nên biểu_thức1 là không cần thiết, mặt khác việc nhận ký tự có thể được ghép vào biểu_thức2 nên biểu_thức3 thực tế cũng lại không cần, như vậy lệnh for không có biểu_thức1 và biểu_thức3 nhưng dấu ";" để ngăn cách các biểu thức thì vẫn phải tồn tại.

Nếu biến _thức2 không có, hoặc là một hằng số khác 0, thì vòng lặp for coi như lặp lại mãi mãi, và chỉ có thể thoát ra khỏi nó bằng lệnh break.

Ví dụ 9.20 Vòng lặp for ở ví dụ 9.14 cũng có thể được viết lại như sau:

```
for (;)
{
    c = getche();
    if (c == ESC)
        break;
}
```

hoặc

```
for (;;)
{
    c = getche();
    if (c == ESC)
        break;
}
```

Như vậy, có thể nói lệnh lặp for rất mạnh, nó có thể thay thế hoàn toàn các lệnh lặp khác trong C, tuy nhiên việc chọn lựa giữa for, do-while và while là hoàn toàn tùy thuộc vào lập trình viên. Đối với các kiểu dữ liệu có cấu trúc (như mảng) thì việc dùng for để làm việc là rất tiện lợi.

Chương trình ví dụ sau đây sẽ dùng vòng for để truy xuất mảng, giải thuật sắp xếp mảng là **select sort**.

Ví dụ 9.21 Viết chương trình nhập một dãy số nguyên, sắp xếp lại dãy số đó theo thứ tự từ lớn tới nhỏ.

```
#include <stdio.h>
#include <conio.h>
main()
{
    int a[100];
    int i, j, n, max, vtmax, tam;
    char c;
    do
```

```

{
    clrscr();
    printf ("Chuong trinh sap xep mang dung for\n");
    printf ("theo thu tu tu lon toi nho\n");
    printf ("Co bao nhieu so can sap xep: ");
    scanf ("%d", &n);
    printf ("Nhap cac so can sap xep: ");
    for (i = 0; i < n; i++)
        scanf ("%d", &a[i]);
    for (i = 0; i < n-1; i++)
    {
        max = a[i];
        vtmax = i;
        for (j = i + 1; j < n; j++)
            if (max < a[j])
            {
                max = a[j];
                vtmax = j;
            }
        if (vtmax != i)
        {
            tam = a[i];
            a[i] = a[vtmax];
            a[vtmax] = tam;
        }
    }
    printf ("Cac so da duoc sap xep:");
    for (i = 0; i < n; i++)
        printf ("%d", a[i]);
    printf ("\nBan co muon tiep tuc khong (Y/N)?");
    c = getche();
}
while (c != 'n' && c != 'N');
}

```

Lưu ý rằng, mảng trong C có chỉ số luôn đi từ 0, do đó vòng for ta luôn có trị khởi động cho biến i là 0, và biến i sẽ chạy từ 0 tới $n-1$.

9.7 LỆNH BREAK VÀ LỆNH CONTINUE

Đây là hai lệnh nhảy không điều kiện của C, chúng cho phép lập trình viên có thể thay đổi tiến trình lặp của các cấu trúc lặp mà ta đã biết: for, while, do-while.

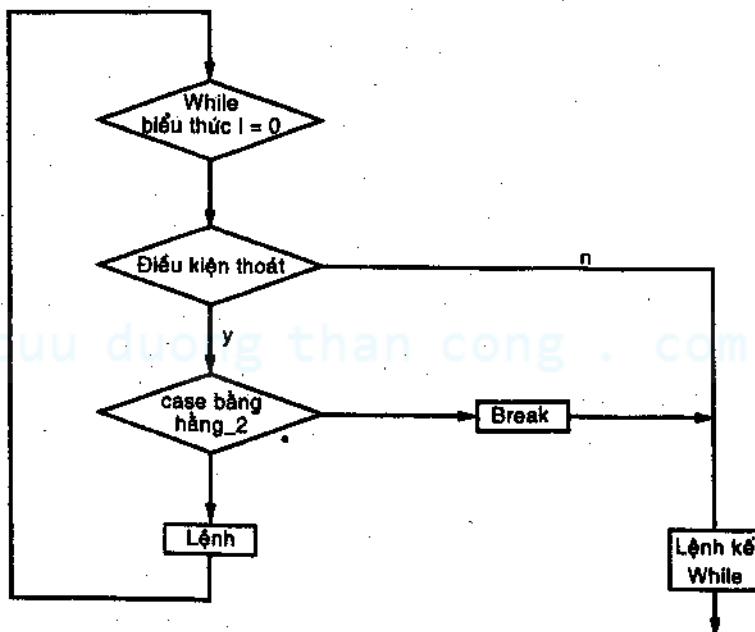
9.7.1 Lệnh break

Như ta đã thấy trong cấu trúc switch-case, lệnh break sẽ kết thúc lệnh switch-case, chuyển điều khiển chương trình đến cuối cấu trúc, sang lệnh kế; còn trong các cấu trúc lặp thì lệnh break cho phép thoát sớm ra khỏi vòng lặp (while, for hoặc do-while) chưa nó mà không cần xét điều kiện của lệnh kế tiếp sau vòng lặp. Như vậy, lệnh break sẽ có thể là lỗi thoát bắt buộc trong các vòng lặp vô tận hoặc trong việc xử lý lỗi.

Cú pháp của lệnh break:

break;

Sau đây là lưu đồ biểu diễn lệnh break sử dụng trong vòng lặp while (hình 9.7)



Hình 9.7 Lưu đồ sử dụng lệnh break trong lệnh while

Ví dụ 9.22 Hãy xem xét các lệnh break trong đoạn chương trình sau:

```
#include <stdio.h>
#include <conio.h>
#define TAB '\t'
#define ESC '\x1b'
#define ENTER '\r'

main()
{
    char c;
    clrscr();
    while (1)
    {
        printf ("Moi ban nhap day cac ky tu: ");
        switch (c = getch())
        {
            case TAB:
                printf("Ban da an phim TAB \n");
                break;
            case ENTER:
                printf ("Ban da an phim ENTER \n");
                break;
            case ESC:
                printf ("Ban da an phim ESC \n");
                break;
            default:
                printf ("Ban da an phim khac \n");
                break;
        } /* kết thúc switch */
        if (c == ESC)
            break;           → break để thoát khỏi while
    } /* kết thúc while */
} /* kết thúc chương trình */

```

*break để
thoát ra
khỏi switch*

Trong ví dụ trên lệnh break trong switch có tác dụng chuyển điều khiển ra khỏi switch, như vậy lệnh if (`c == ESC`) ... (là lệnh ngay sau lệnh switch) sẽ được thực hiện khi lệnh switch được thực hiện xong; lệnh break trong if (`c == ESC`) ... có tác dụng chuyển điều khiển ra khỏi vòng lặp while, nếu phím được nhấn là phím ESC thì điều kiện của if được thỏa mãn như vậy điều khiển sẽ được chuyển ra khỏi vòng while và kết thúc chương trình.

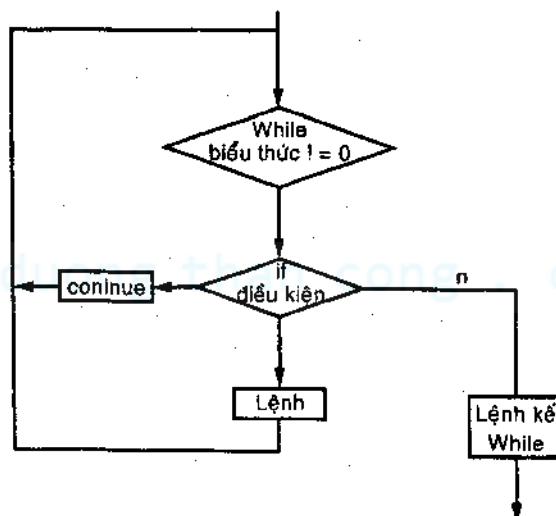
9.7.2 Lệnh continue

Khác với lệnh break, lệnh continue có tác dụng chuyển điều khiển chương trình về đầu vòng lặp chuẩn bị cho chu kỳ lặp mới, bỏ qua các lệnh còn lại nằm ngay sau lệnh continue trong chu kỳ lặp hiện hành. Lệnh này chỉ được dùng trong các vòng lặp, để bỏ qua các lệnh không cần thực thi trong vòng lặp trong các trường hợp đặc biệt nào đó.

Cú pháp lệnh continue

[cuuduongthancong.com](https://tailieu.tucn.com)

Lưu đồ biểu diễn lệnh continue trong vòng lặp while được cho trong hình 9.8 sau.



Hình 9.8 Lưu đồ sử dụng lệnh continue trong lệnh while

Ví dụ 9.23

```
i = 0;
while (i <= 10)
{
    i++;
    if (i >= 6 && i <= 8)
        continue;
    printf ("Trị hiện thời của i là %d\n", i);
}
```

Đoạn chương trình trên in ra các giá trị của i từ 1 tới 5 và từ 9 đến 11 vì nếu điều kiện ($i \geq 6 \text{ && } i \leq 8$) thỏa thì lệnh **continue** được thực thi và khi đó lệnh `printf(...)` được bỏ qua.

Ví dụ 9.24 Viết chương trình nhập một dãy số, tính tổng của các số dương trong dãy số đó và thương số của tổng đó với từng số dương này.

```
#include <stdio.h>
#include <conio.h>
main()
{
    double a[100];
    double tong;
    int i, n;
    clrscr();
    printf ("Co bao nhieu so can tinh: ");
    scanf ("%d", &n);
    printf ("Nhap cac so can tinh tong: ");
    for (i = 0; i < n; i++)
        scanf ("%lf", &a[i]);
    for (i = 0, tong = 0; i < n; i++)
    {
        if (a[i] <= 0)
            continue;
```

```

    tong += a[i];
}

printf ("Tong cua cac so duong la %. 2f\n", tong);
for (i = 0; i < n; i++)
{
    if (a[i] <= 0)
        continue;
    printf("Thuong cua tong voi so thu %d la %5.2f\n",i,tong/a[i]);
}
getch();
}

```

Trong chương trình trên khi điều kiện ($a[i] \leq 0$) thỏa thì các lệnh còn lại nằm ngay sau lệnh continue bị bỏ qua, mà cụ thể là các lệnh

$tong += a[i];$

trong vòng lặp for thứ nhất, và lệnh

$printf("Thuong cua tong voi so thu %d la %5.2f\n",i,tong/a[i]);$

trong vòng lặp for thứ hai.

Như vậy, đối với lệnh do-while hoặc while, lệnh continue sẽ chuyển điều khiển về kiểm tra điều kiện của vòng lặp, còn đối với lệnh for, lệnh continue sẽ chuyển điều khiển chương trình trở lên thực hiện biểu thức thứ 3 của for, sau đó mới kiểm tra điều kiện của for.

9.8 LỆNH RETURN

Lệnh này dùng để thoát ra khỏi hàm hiện thời trở về hàm đã gọi nó, có thể trả về cho hàm gọi một trị. Lệnh này sẽ kết thúc hàm dù nó nằm ở đâu trong thân hàm. Khi gặp lệnh này C sẽ không thực hiện bất cứ lệnh nào sau lệnh return nữa.

Các cú pháp của lệnh return như sau:

```

return;
return (biểu-thức);
return biểu-thức;

```

Với biểu-thức là một biểu thức tổng quát của C, biểu thức sẽ được tính và trị của nó sẽ được trả về cho nơi gọi hàm.

Lệnh return không có biểu thức nào đi theo cả chỉ đơn thuần là kết thúc hàm mà không cần trả về trị cho nơi gọi hàm.

Ví dụ 9.25

Thiết kế hàm trả về kết quả so sánh hai số theo quy tắc sau đây:

số đầu > số sau: hàm trả về trị 1

số đầu = số sau: hàm trả về trị 0

số đầu < số sau: hàm trả về trị -1

Hàm được viết như sau:

```
int so_sanh (int a, int b)
{
    if (a > b)      /* Lệnh return kết thúc hàm, trả về trị 1 cho */
        return 1;   /* nơi đã gọi hàm */
    else if (a == b)
        return 0; /* Trả về trị 0 cho nơi gọi hàm khi a = b */
    else /* a < b */
        return -1; /* Trả về trị -1 cho nơi gọi hàm khi a < b */
}
```

Trong hàm trên có nhiều lệnh return và điều này là hoàn toàn bình thường vì khi gặp lệnh này thì hàm sẽ kết thúc mà tùy trường hợp hàm sẽ kết thúc và sẽ trả về trị khác nhau. Do đó hàm trên có thể được viết lại như sau:

```
int so_sanh (int a, int b)
{
    if (a > b)      /* Lệnh return kết thúc hàm, trả về trị 1 cho */
        return 1;   /* nơi đã gọi hàm */
    else if (a == b)
        return 0; /* Trả về trị 0 cho nơi gọi hàm khi a = b */
    return -1;     /* Trả về trị -1 cho nơi gọi hàm khi a < b */
}
```

Ta cũng có thể viết lại hàm trên nhưng lại dùng biểu thức điều kiện như sau:

```
int so_sanh (int a, int b)
{
    return (a > b) ? 1 : (a == b) ? 0 : -1;
}
```

C sẽ tính giá trị của biểu thức trong lệnh return và trả giá trị này về cho nơi gọi hàm này. Giá trị này được gọi là kết quả trả về của hàm và sẽ được đặt vào chỗ tên hàm trong các biểu thức hay lệnh có sự gọi hàm. Ví dụ như lệnh gọi hàm sau

```
ket_qua = so_sanh (4, 5);
```

hàm so_sanh (4, 5) sẽ được gọi với 4 và 5 là hai trị cần so sánh, hàm sẽ trả về một trị tùy theo đối số đưa vào cho hàm. Trong trường hợp này hàm sẽ trả về trị là -1, trị này sẽ được gán vào cho biến ket_qua.

Lệnh return cũng có thể được dùng thay cho lệnh break khi việc thoát ra khỏi vòng lặp đồng nghĩa với việc thoát ra khỏi hàm đang chứa vòng lặp vì khi gặp return, chương trình sẽ bỏ qua các lệnh sau nó để thoát ra khỏi hàm hiện hành, trả điều khiển về cho nơi gọi hàm.

Ví dụ 9.26 Chương trình sau dùng lệnh return để kết thúc vòng lặp lặp vô tận khi điều kiện thỏa (là phím ESC được nhấn).

```
#include <stdio.h>
#include <conio.h>
#define ESC 'x1b'
void nhan_ky_tu (void); ← prototype của hàm
main()
{
    char c;
    clrscr();
    printf ("Moi ban nhap cac ky tu: ");
    nhan_ky_tu (); ← gọi hàm
}
```

```

void nhan_ky_tu (void) ← định nghĩa hàm
{
    while (1)
        if (getche() == ESC)
            return;
}

```

Trong chương sau ta sẽ nói rõ hơn về cách khai báo, sử dụng hàm như thế nào, còn hiện thời trong ví dụ trên, hàm nhan_ky_tu không trả về trị nên kiểu trả về khi khai báo hàm là void, hàm cũng không nhận đối số truyền vào nên kiểu của đối số là void.

9.9 LỆNH GOTO

Mặc dù không ủng hộ cho việc lập trình có goto nhưng C vẫn có lệnh rẽ nhánh không điều kiện goto, lệnh này cho phép chuyển điều khiển chương trình cho một lệnh nào đó.

Cú pháp của lệnh goto:

goto nhãn;

Với nhãn là một danh hiệu không chuẩn, danh hiệu này sẽ được đặt ở trước lệnh mà ta muốn nhảy đến theo cú pháp sau:

nhãn: lệnh

nhãn mà lệnh goto muốn nhảy đến phải nằm trong cùng một hàm với lệnh goto đó, do đó trong những hàm khác nhau có thể có các tên nhãn giống nhau, nhưng trong cùng một hàm các tên nhãn này phải khác nhau.

Ví dụ 9.27 Cách sử dụng lệnh goto trong một chương trình C

```

main()
{
    lap_lai: ——————
        clrscr();
        ...
        if ((c = getch()) != ESC)
            goto lap_lai;
}

```

chuyển điều khiển về lệnh
ngay sau nhãn khi goto được
thực thi

Sự tồn tại lệnh goto trong C thực tế chỉ có ý nghĩa lịch sử, nó không được khuyến khích sử dụng trong các chương trình được viết theo phương pháp lập trình có cấu trúc (structured programming), trong các chương trình này lệnh goto được hạn chế sử dụng vì thực tế C đã có cấu trúc điều khiển hoàn toàn thay thế được goto, mặt khác việc sử dụng nhiều lệnh goto sẽ làm cho chương trình trở nên rắc rối, khó hiểu và khó sửa sai, phá vỡ cấu trúc của một chương trình đang dùng các cấu trúc điều khiển mạnh như trong C. Do đó, chúng ta chỉ nên sử dụng lệnh goto trong những trường hợp kiểm tra lỗi, nhập lại trị hoặc cần thoát sớm.

Ví dụ 9.28 Chương trình sau đây sẽ dùng goto để kiểm tra việc nhập trị để tính căn.

```
#include <stdio.h>
#include <conio.h>
main()
{
    double a;
    clrscr();
    nhap:
    printf ("Moi ban nhap mot tri > 0: ");
    scanf ("%lf", &a);
    if (a <= 0)
    {
        printf ("Ban da nhap sai tri. Nhap lai \n");
        goto nhap;
    }
    /* khi chương trình đã ra tới đây thì a > 0 rồi */
    printf ("Can bac hai cua %5.2f la %5.2f\n", a, sqrt(a));
    getch();
}
```

Trong ví dụ trên lệnh do-while hoàn toàn có thể thay thế cho lệnh goto trong việc kiểm tra trị để tính căn, ta có thể viết lại chương trình trên như sau:

```

#include <stdio.h>
#include <conio.h>
main()
{
    double a;
    clrscr();
    do
    {
        printf ("Moi ban nhap mot so trai > 0: ");
        scanf ("%lf", &a);
        if (a <= 0)
            printf ("Ban da nhap sai so. Nhap lai \n");
    }
    while (a <= 0);
    /* khi chuong trinh da ra toi day thi a > 0 roi */
    printf ("Can bac hai cua %5.2f la %5.2f\n", a, sqrt(a));
    getch();
}

```

9.10 LỆNH RỖNG

Trong C có khái niệm lệnh rỗng, lệnh này chỉ có một dấu chấm phẩy (;), nó rất cần thiết trong nhiều trường hợp, ví dụ như đối với các vòng lặp, khi ta đặt các lệnh biểu thức thực thi vào trong các biểu thức của lệnh (như đặt vào trong biểu-thức3 của vòng for chẳng hạn) thì ta không cần có thêm lệnh thực thi làm thân cho chúng nữa, khi đó nếu để trống, C sẽ hiểu nhầm rằng lệnh kế tiếp sẽ là thân của vòng lặp, do đó chỉ còn cách cho một lệnh rỗng làm thân của chúng.

Ta hãy xét các ví dụ sau:

Ví dụ 9.29

Vòng lặp for để tính giai thừa từ 1 tới n như sau

```

for (i = gt = 1; i <= n; gt *= i++)
;
printf ("Giai thua %d! = %d\n", n, gt);

```

Trong ví dụ trên biểu thức `i = gt = 1` sẽ thực hiện việc gán trị vào biến `gt`, trị của biểu thức `gt = 1` sẽ là trị của biến `gt` sau khi gán, tức bằng 1, trị này đến lượt nó lại được tính xong thì trị của biến `i` và `gt` đều bằng 1 cả.

Lệnh rỗng; làm lệnh của thân vòng for vì lệnh biểu thức của for đã được để vào biểu thức 3 của vòng for

```
gt *= i++;
```

do đó ta phải để thêm một lệnh rỗng cho for để tránh hiểu nhầm lệnh `printf (...);` là lệnh của thân vòng for.

Thiết kế hàm `strlen (s)` trả về chiều dài của một chuỗi ký tự `s`. Ta sẽ dùng một vòng lặp cụ thể là vòng while lặp cho đến khi gặp ký tự kết thúc chuỗi '`\0`' thì kết thúc hàm, trả về trị là chiều dài của chuỗi. Hàm có thể được viết như sau:

```
int strlen (char s[])
{
    int i = 0;
    while ( s[i++] != '\0' )
        ; ← Lệnh rỗng
    return (i);
}
```

Bản thân lệnh while này không cần thêm một lệnh nào nữa, nhưng vì cú pháp đòi hỏi phải có một lệnh đi theo, nên ta phải đặt một lệnh rỗng làm lệnh của thân vòng while.

Các nhãn của lệnh goto cũng cần lệnh rỗng, khi các nhãn này được đặt ở cuối của một hàm để lệnh goto nhảy đến và kết thúc hàm, vì theo cú pháp đặt nhãn, theo sau nhãn luôn phải có một lệnh.

Ví dụ 9.30 Xét hàm như sau

```
void kiem_tra (void)
{
    int n;
    scanf ("%d", &n);
    if (n < 0)
```

```
    goto thoát;
    ...
thoát:
    ;   ← Lệnh rỗng
}
```

Do lệnh rỗng trong C vẫn có một ý nghĩa nhất định, nên khi sử dụng cũng cần phải lưu ý, vì nếu dư một dấu chấm phẩy C cũng hiểu và tưởng rằng có lệnh rỗng, mà không thực thi đúng theo ý đồ của lập trình viên.

Ví dụ 9.31 Xét đoạn chương trình sau đây

```
for (i = 1, s = 0; i < 10; i++) ;
    s += i;           ←———— Lệnh của vòng for là lệnh rỗng
    printf("Tong la %d \n",s);
```

đoạn chương trình trên mục đích là sử dụng vòng lặp for để tính tổng từ 1 tới 9, tuy nhiên vì lệnh lặp for lại có lệnh rỗng nên lệnh thực thi của thân vòng lặp là:

```
s += i;
```

không được thực thi đúng, vòng lặp for chỉ làm có một việc là tăng trị của i lên 1 sau mỗi chu kỳ lặp, có thể nói là vòng lặp đã chạy "không tải"; sau khi vòng lặp được thực thi xong, trị của biến i là 10, trị này sẽ được cộng vào cho biến s , nên sau cùng s chỉ bằng 10 mà thôi.

9.11 MỘT SỐ VÍ DỤ

Trong mục này chúng ta hãy xét một số ví dụ sử dụng các lệnh điều khiển và vòng lặp trên.

9.11.1 Ví dụ 1

Chương trình sau đây nhập một số nguyên dương ≥ 2 . Kiểm tra xem số đó có phải là số nguyên tố không.

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

```

int n, i;
// Nhập số nguyên
do
{
    printf ("Mời nhập một số nguyên dương >=2: ");
    scanf ("%d", &n);
    if (n < 0)
        printf ("Trị %d < 2. Sai trị. Nhập lại.\n", n);
} while (n < 2);
// Kiểm tra số nguyên tố
i = 2;
while (n % i)
{
    i++;
}
// In kết quả
if (i == n)
    printf ("Số %d là số nguyên tố", n);
else
    printf ("Số %d không phải là số nguyên tố", n);
getch();
}

```

Trong chương trình trên, để nhập số nguyên n thỏa lớn hơn hoặc bằng 2, ta sử dụng lệnh do-while để thực hiện việc kiểm tra, nếu có trị sai thì việc nhập được tiếp tục cho tới khi nhận được trị đúng theo yêu cầu. Trong thực tế, ta thường dùng cấu trúc này để nhập trị vì nó rất thuận lợi do trị nhận được sau khi nhập đã thỏa điều kiện ban đầu.

Để kiểm tra số nguyên tố, ta dùng lệnh while. Biểu thức $n \% i$ bằng 0 chỉ khi n là bội số của i (tức n không phải là số nguyên tố) hoặc khi n bằng i (tức khi n là số nguyên tố). Trong cả hai trường hợp lệnh while sẽ không tiếp tục được thực thi (do $n \% i == 0$), và điều khiển sẽ tiếp tục thực thi lệnh kế.

Việc kiểm tra số nguyên tố sau đó sẽ dễ dàng với lệnh if-else, nếu i nhỏ hơn n khi thoát ra vòng while, tức ta tìm được một số là ước số của n , nên n không là số nguyên tố; ngược lại (tức $i == n$) thì n là số nguyên tố.

9.11.2 Ví dụ 2

Chương trình sau đây tính tổng trong khai triển Maclaurin để tính xấp xỉ sin (x), với x là trị radian nhập từ bàn phím, n là một số nguyên dương ≥ 1 . Chương trình sử dụng hàm để tính chuỗi xấp xỉ. Đặc giả có thể xem thêm khái niệm chương trình con là hàm trong C trong chương 10.

Chuỗi xấp xỉ tính sin (x) và chương trình được cho như sau. Chú ý, chuỗi này đã được chứng minh hội tụ.

$$\sin(x) = x - \frac{x^3}{3!} + \cdots - \frac{x^n}{n!}$$

```
#include <stdio.h>
#include <conio.h>
double sin_x (double x, int n); // Khai báo prototype của hàm
int main()
{
    double x, sinx;
    int n;
    printf ("Chương trình tính sin x theo khai triển Maclaurin.\n");
    // Nhập x radian bất kỳ
    printf ("Mời nhập trị bất kỳ: ");
    scanf ("%lf", &x);
    // Nhập trị n >=1
    do
    {
        printf ("Mời nhập một số nguyên dương >=1: ");
        scanf ("%d", &n);
        if (n < 1)
            printf ("Trị %d < 1. Sai trị. Nhập lại.\n", n);
    } while (n < 1);
    // Gọi hàm tính sin x
    sinx = sin_x (x, n);
    // In ra kết quả
    printf ("sin (%.5lf) = %.5lf\n", x, sinx);
```

```

    getch();
}

double sin_x (double x, int n)
{
    int i,                      // biến i điều khiển vòng lặp
    dau = 1;                    // biến dấu để thay đổi dấu điều hòa trong chuỗi sin x
    double s = 0,                // biến s để giữ trị sin x trong hàm
    t = x;                      // biến t là biến phụ
                                // Tính tổng của chuỗi

    for (i = 1; i <= n; i+= 2)
    {
        s += dau * t;
        t *= x*x/(i+1)/(i+2);
        dau = -dau;
    }
    return s;
}

```

Chương trình trên có khai báo và sử dụng hàm với prototype trong phần khai báo của chương trình C và phần định nghĩa hàm nằm trong hàm của chương trình C (sau hàm main()) mà độc giả có thể tham khảo trước ở chương 10, Hàm.

Việc nhập trị trong hàm main() cho x sẽ đơn giản là trị thực bất kỳ, còn n chỉ cần thỏa lớn hơn hoặc bằng 1. Chuỗi sin (x) sẽ được tính trong hàm sin_x (). Để thực hiện chuỗi điều hoàn đan dấu, ta dùng biến dau (tức dấu) thay đổi trị từ 1 sang -1 và ngược lại sau mỗi lần lặp để có được dấu mong muốn. Biến t giữ trị là từng số hạng riêng lẻ trong chuỗi mà chưa có dấu. Lưu ý, do i được khởi động ban đầu là 1, sau mỗi lần lặp sẽ tăng thêm 2 đơn vị, nên vòng lặp sẽ kết thúc khi $i > n$ bất chấp tính chẵn lẻ của n .

Để kết thúc hàm và trả về trị cho nơi gọi, trong trường hợp ví dụ này là hàm main(), lệnh return được sử dụng theo sau đó là một trị, tức biến s . Trị này sẽ được biến cục bộ sinx trong hàm main() nhận và được sử dụng để in ra màn hình trong hàm printf().

9.11.3 Ví dụ 3

Chương trình sau đây sẽ tạo và in ra màn hình hình tam giác ngược các dấu hoa thị như sau:



Giá trị nhập vào n là số nguyên dương ≥ 3 . Ta có chương trình như sau:

```

#include <stdio.h>
#include <conio.h>
int main()
{
    int n, i, j;
    // Nhập số nguyên n
    do
    {
        printf ("Mời nhập một số nguyên dương >=3: ");
        scanf ("%d", &n);
        if (n < 0)
            printf ("Trị %d < 3. Sai trị. Nhập lại.\n", n);
    } while (n < 3);
    // In ra hình tam giác ngược
    for (i = 1; i <= n-1; i++)
    {
        if (i == 1)
        {
            for (j = 1; j <= 2*n - 1; j++)
                printf ("*");
            printf ("\n");
            continue;
        }
        printf ("%*c%*c\n", i, '*', 2*n - 2*i, '*');
    }
    printf ("%*c\n", i, '*')
    getch();
}
  
```

Trong chương trình trên, việc nhập và kiểm tra trị vẫn dùng lệnh do-while kết hợp với if. Để in ra hình tam giác này, dòng đầu tiên (với i bằng 1) sẽ dùng vòng lặp for với biến j. Lệnh continue; sẽ bỏ qua không thực hiện lệnh in ra các dòng kế khi dòng i bằng 1 đã được in xong. Để in các dòng kế tiếp, ta sử dụng lệnh

```
printf ("%*c%*c", i, "*", 2*n - 2*i, "*");
```

với định dạng "%*c", dấu "*" trong định dạng này xác định số vị trí dành để in ký tự ngay sau trong định dạng "%c". Ví dụ, khi ta có "%*c" cho đối số là

```
5, "
```

thì khi in ra màn hình, dấu "*" sẽ được in ra trong định dạng 4 khoảng trắng phía trước.

Xin mời độc giả hãy thử phân tích để hiểu giải thuật của chương trình trên.

BÀI TẬP CUỐI CHƯƠNG

9.1 Viết một chương trình nhập 4 số và in ra

- a) số lớn nhất trong 4 số đó
- b) số nhỏ nhất trong 4 số đó

9.2 Viết chương trình tìm số nguyên tố từ 1 tới 100

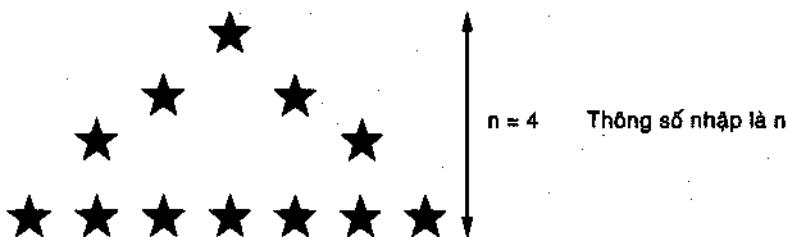
9.3 Nhập một số nguyên từ bàn phím, in ra màn hình theo thứ tự ngược lại.

Ví dụ nhập: 54321
 xuất : 12345

9.4 In ra màn hình bản cửu chương cần biết.

9.5 In ra màn hình các bản cửu chương từ 2 đến 9.

9.6 Vẽ ra màn hình hình sau:



9.7 Tính biểu thức sau đây

a) $T = 1! + 2! + \dots + n!$ thông số nhập là n

b) $T = \frac{1! + (1+2)! + \dots + (1+\dots+n)!}{n!}$ thông số nhập là n

c) $T = \frac{e^1}{1!} + \frac{e^2}{2!} + \dots + \frac{e^n}{n!}$

Biết trong C có hàm $\exp(x)$ để tính e^x , prototype hàm này nằm trong file math.h.

9.8 Tính biểu thức sau:

$$s = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots + \frac{1}{n} < 2$$

Hãy viết chương trình nhập một số a thỏa: $1 < a < 2$, sau đó tìm số n thỏa điều kiện (1): $s < a$

9.9 Một người muốn gửi một số tiền vào ngân hàng, hãy viết chương trình tính tổng số tiền mà người đó có được sau khi đã gửi ngân hàng theo một trong hai cách gửi:

- Gửi từng tháng rút tiền lãi
- Gửi không rút lãi từng tháng, mà nhập lãi vào vốn

Thông số nhập cần thiết:

- Số tiền gửi lúc đầu
- Thời gian gửi (theo tháng)
- Lãi suất/tháng

9.10 Viết chương trình tính $\cos(x)$, e^x , $\log_{10}x$ và $\tan(x)$ theo khai triển MacLaurin

Chương 10

HÀM

10.1 KHÁI NIỆM HÀM

Bất kỳ ngôn ngữ lập trình nào cũng đều có khái niệm chương trình con (*subroutine*), mỗi chương trình con như vậy sẽ đảm nhận thực hiện một thao tác nhất định. Đối với C, chương trình con chỉ ở một dạng là hàm (*function*), không có khái niệm thủ tục (*procedure*). Tất cả các thao tác chuẩn của C đều là các hàm, tuy nhiên hàm trong C có khả năng sử dụng rất mạnh và đặc biệt, mỗi hàm có thể trả về một trị nhất định, tuy nhiên nơi chương trình gọi hàm có sử dụng trả về từ hàm hay không là tùy theo nhu cầu lập trình.

Nếu các ngôn ngữ khác, như Pascal, sẽ gọi hàm trong chương trình chính và sử dụng hàm thì đối với C, chương trình chính cũng là một hàm, đó là hàm main () và là hàm đặc biệt của C. Nó là một hàm mà trong đó các thao tác lệnh (bao gồm các biểu thức tính toán, gọi hàm,...) được C thực hiện theo một trình tự hợp logic để giải quyết bài toán được đặt ra.

Việc sử dụng hàm trong C sẽ làm cho chương trình trở nên rất dễ quản lý, dễ sửa sai, khi đó chương trình trong hàm main () thực tế chỉ để quản lý các hàm khác theo giải thuật của bài toán mà không cần biết các thao tác thực sự bên trong từng hàm, khái niệm hàm như vậy ta gọi là hộp đen (*black box*). Khi có sự sửa đổi trong hộp đen nào đó mà gây ra kết quả xấu thì ta có thể biết ngay chính hộp đen đó là thủ phạm, vì thế việc phát hiện sai và sửa sai là rất dễ dàng và chính xác. Khi gọi sử dụng hàm, ta chỉ cần cung cấp đối số cần thiết mà hàm yêu cầu, lấy ra các giá trị mà hàm cung cấp ngược trở lại để sử dụng, do đó khi sử dụng hàm hay khi thiết kế hàm ta cũng cần phải có những quy định về cách sử dụng hàm sao cho thống nhất và tổng quát.

Tất cả các hàm trong C đều ngang cấp nhau, không có khái niệm hàm con của một hàm, tức khi thiết kế hàm, mỗi hàm đều có một thao tác riêng, trong mỗi hàm đó ta không thể khai báo các hàm con của nó như trong ngôn ngữ Pascal. Các hàm đều có thể gọi lẫn nhau, dĩ nhiên hàm được gọi phải được khai báo trước hàm gọi (hoặc sau này ta sẽ biết thêm là hàm được gọi có thể được đặt sau hàm gọi cũng được, tuy nhiên khi đó prototype của hàm được gọi phải được đặt trước hàm gọi).

Các hàm trong một chương trình có thể nằm trên các tập tin khác nhau và khác với tập tin chính (chứa hàm main()), mỗi tập tin như vậy được gọi là một module chương trình. Các module chương trình sẽ được dịch riêng rẽ và sau đó được liên kết (*link*) lại với nhau để tạo ra được một tập tin thực thi duy nhất. Cách tạo chương trình theo kiểu nhiều module như vậy trong C là project; ta sẽ đề cập về vấn đề này kỹ càng hơn trong phần phục lục của cuốn sách này.

Để minh họa cho những ý kiến trên ta hãy xét hai chương trình sau, cả hai đều giải phương trình bậc hai, nhưng một chương trình sử dụng hàm còn chương trình kia thì không.

Ví dụ 10.1

Chương trình 1

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
main ()
{
    double a, b, c, delta, n1, n2;
    clrscr();
    printf ("Nhập 3 hệ số phuong trình bậc hai: ");
    scanf ("%lf %lf %lf", &a, &b, &c);
    if (a == 0)
        /* phuong trình suy biến về bậc nhất */
    {
        printf ("Phuong trình suy biến về bậc nhất và ");
        if (b == 0)
            if (c == 0)
                printf ("vô số nghiệm\n");
            else /* c != 0 */

```

```

        printf ("vo nghiem\n");
    else /* b != 0 */
    {
        n1 = -c/b;
        printf ("co 1 nghiem: = %5.2f \n", n1);
    }
}
else /* a != 0 */
{
    printf ("Phuong trinh bac hai va ");
    delta = b*b - 4*a*c;
    if (delta < 0)
        printf ("vo nghiem thuc\n");
    else if (delta == 0)
    {
        n1 = n2 = -b/2/a;
        printf ("co nghiem kep x1 = x2 = %5.2f \n", n1);
    }
    else /* delta > 0 */
    {
        n1 = (-b + sqrt(delta))/2/a;
        n2 = (-b - sqrt(delta))/2/a;
        printf ("co hai nghiem phan biet; \n");
        printf ("x1 = %5.2f \n", n1);
        printf ("x2 = %5.2f \n", n2);
    }
}
getch();
}

```

Chương trình 2

```

#include <stdio.h>
#include <conio.h>
#include <math.h>
void gptb1 (double a, double b);
void gptb2 (double a, double b, double c);
void gptb1 (double a, double b)
{

```

```
printf ("Phuong trinh suy bien ve bac nhat va ");
if (a == 0)
    if (b == 0)
        printf ("vo so nghiem\n");
    else /* b != 0 */
        printf ("vo nghiem\n");
else
    printf ("co 1 nghiem: x = %5.2f \n", -b/a);
}

void gptb2 (double a,double b,double c)
{
    double delta, x1, x2;
    printf ("Phuong trinh bac hai va ");
    delta = b*b - 4*a*c;
    if (delta < 0)
        printf ("vo nghiem thuc\n");
    else if (delta == 0)
        printf ("co nghiem kep x1 = x2 = %5.2f \n", -b/2/a);
    else /* delta > 0 */
    {
        x1 = (-b + sqrt(delta))/2/a;
        x2 = (-b - sqrt(delta))/2/a;
        printf ("co hai nghiem phan biet: \n");
        printf ("x1 = %5.2f \n ", x1);
        printf ("x2 = %5.2f \n ", x2);
    }
}
main()
{
    double a, b, c;
    clrscr();
    printf ("Nhap 3 he so phuong trinh bac hai: ");
    scanf ("%lf %lf %lf", &a, &b, &c);
    if (a == 0)
        /* phuong trinh suy bien ve bac nhat */
        gptb1 (b, c);
    else /* a != 0 */
        gptb2 (a, b, c);
    getch();
}
```

Chương trình 1 giải phương trình bậc hai theo kiểu bình thường, chương trình này không thiết kế hàm, do đó nhìn vào chương trình trong hàm main() ta thấy không rõ ràng bằng cấu trúc trong hàm chính của chương trình 2. Chương trình 2 chỉ đơn giản là gọi hai hàm: một hàm là hàm giải phương trình bậc nhất, một hàm là hàm giải phương trình hai, cả hai hàm này được gọi sử dụng trong hàm main() tùy theo trường hợp trị của hệ số a.

10.2 KHAI BÁO HÀM

Khai báo một hàm có nghĩa là chỉ ra rõ rằng trị trả về vị trí kiểu gì, đối số đưa vào cho hàm có bao nhiêu đối số, mỗi đối số có kiểu như thế nào và các lệnh bên trong thân hàm xác định thao tác của hàm. Như vậy, sẽ có hai loại hàm: *hàm trong thư viện của C* và *hàm do lập trình viên tự định nghĩa*.

- Nếu hàm sử dụng là hàm chuẩn trong thư viện thì việc khai báo hàm chỉ đơn giản là khai báo prototype của hàm, các prototype này đã được phân loại và ở trong các file .h, lập trình viên cần ra lệnh **#include** bao hàm các file này vào chương trình hoặc module chương trình sử dụng nó.
- Nếu các hàm sử dụng là do lập trình viên tự định nghĩa thì việc khai báo hàm bao gồm hai việc: khai báo prototype của hàm đầu chương trình và định nghĩa các lệnh bên trong thân hàm (hay thường được gọi tắt là định nghĩa hàm).

Phần prototype của một hàm sẽ đề cập sau, trong mục này ta chỉ xét cấu trúc của một hàm, cách định nghĩa hàm và gọi hàm.

Để định nghĩa một hàm hiện nay có hai cú pháp sau:

Dạng 1:

```

kiểu tên_hàm (danh_sách_khai_báo_đối_số) → đầu hàm
{
    khai_báo_biến_cục_bộ → thân hàm
    lệnh
}

```

Đối với dạng 1, hàm có thể được chia ra làm hai phần: phần đầu và phần thân; phần đầu gồm phần khai báo kiểu của trị trả về, tên

hàm và danh sách khai báo đối số; phần thân sẽ là phần khai báo biến cục bộ được sử dụng trong thân hàm và các lệnh trong hàm.

Với:

- **kiểu** là kiểu của kết quả trả về từ hàm, kiểu này có thể là bất kỳ kiểu nào trong các kiểu cơ bản (char, int, float, double, long, short,...) hoặc con trỏ đến bất kỳ kiểu cơ bản nào hoặc đến bất kỳ kiểu cấu trúc nào (*structure* hay *union*). Nếu hàm không trả về bất cứ trị nào thì kiểu trả về nên để là void, còn nếu không để kiểu thì C tự hiểu là trị trả về thuộc kiểu int.
- **tên_hàm** là danh hiệu không chuẩn do lập trình viên tự đặt theo nguyên tắc ở chương 1.
- **danh_sách_khai_báo đối số** sẽ liệt kê các khai báo đối số giả, mỗi đối số đều được khai báo riêng biệt, các khai báo này cách nhau một dấu phẩy và theo cú pháp của khai báo biến. Nếu không có đối số cho hàm thì nên để là void.
- Thân hàm nằm giữa cặp dấu móc { và }, trong đó có thể có phần **khai_báo biến_cục_bộ** và các **lệnh** trong thân hàm.

Dạng 2 hiện nay được xem là lạc hậu và ít được sử dụng vì kém hiệu quả hơn dạng 1 trong việc kiểm soát trị đưa vào cho hàm, cú pháp dạng này như sau:

```

kiểu tên_hàm (danh_sách đối số)
khai_báo đối số
{
    khai_báo biến_cục_bộ
    lệnh
}

```

Cú pháp này hoàn toàn tương tự dạng 1, nhưng phần danh sách đối số và phần khai báo đối số được để ở hai chỗ khác nhau, cách viết như thế này sẽ gọn nhẹ cho phần đầu của hàm vì phần danh sách đối số chỉ là tên các đối số giả được sử dụng trong thân hàm, còn phần khai báo có đối số như khai báo biến, tức khai báo kiểu cho tất cả các đối số đã được liệt kê trong phần danh sách đối số.

Ví dụ 10.2 Xét hai dạng của hàm so sánh hai số sau:

Dạng 1:

```
int so_sanh (int a, int b) → phần đầu hàm
{
    int ket_qua;
    if (a >b)
        ket_qua = 1;
    else if (a == b)
        ket_qua = 0;
    else if (a < b)
        ket_qua = -1;
    return ket_qua;
}
```

Dạng 2:

```
int so_sanh (a, b) → phần đầu hàm
int a, b;
{
    int ket_qua;
    if (a >b)
        ket_qua = 1;
    else if (a == b)
        ket_qua = 0;
    else if (a < b)
        ket_qua = -1;
    return ket_qua;
}
```

Ta thấy rằng hàm ở dạng 1 dễ quan sát hơn ở dạng 2, dạng 2 hiện nay ít được sử dụng.

Do đó, chương trình so sánh hai số viết có sử dụng hàm như sau:

Ví dụ 10.3

```
#include <stdio.h>
#include <conio.h>
int so_sanh (int a, int b); → prototype của hàm so_sanh
```

```

main()
{
    int a, b, ket_qua;
    clrscr();
    printf ("Moi nhap hai so ");
    scanf ("%d %d" , &a, &b);
    ket_qua = so_sanh (a, b);
    switch (ket_qua)
    {
        case -1:
            printf ("So %d nho hon so %d \n" , a, b);
            break;
        case 0:
            printf ("So %d bang so %d \n" , a, b);
            break;
        case 1:
            printf ("So %d lon hon so %d \n" , a, b);
            break;
    }
    getch();
}

int so_sanh (int a, int b)
{
    int ket_qua;
    if (a >b)
        ket_qua = 1;
    else if (a == b)
        ket_qua = 0;
    else if (a < b)
        ket_qua = -1;
    return ket_qua;
}

```

→ định nghĩa hàm

10.3 ĐỐI SỐ CỦA HÀM - ĐỐI SỐ LÀ THAM TRỊ

Một cách bình thường, khi gọi hàm thì đối số thật cần gởi cho hàm chỉ được gởi dưới dạng tham số trị, có nghĩa là các biến, trị hoặc biểu thức được gởi đến cho một hàm qua đối số của nó, sẽ được lấy trị để tính toán trong thân hàm. Như vậy về khía cạnh biến có thể nói trị của biến thật bên ngoài khi gọi hàm đã được chép sang đối số giả, mà ta có thể xem như là biến cục bộ của hàm, và mọi việc tính toán chỉ được thực hiện trên biến cục bộ này mà thôi. Để dễ hình dung ta hãy xét Ví dụ 10.4 sau đây.

Ví dụ 10.4

Viết chương trình tính lũy thừa n của x (x^n), với n nguyên và thực.

```
#include <stdio.h>
#include <conio.h>
double luy_thua(double x, int n);
main()
{
    int n;
    double x, xn;
    clrscr();
    printf ("Moi nhap so tinh luy thua: ");
    scanf ("%lf", &x);
    printf ("Moi nhap so luy thua: ");
    scanf ("%d", &n);
    xn =luy_thua (x, n);
    printf("Ket qua: %5.2f luy thua %d bang: %7.2f\n", x,n,xn);
    printf ("Tri cua so mu la %d", n);
    getch();
}
double luy_thua(double x, int n)
{
    double t = 1;
    for ( ; n > 0; n--)
        t *= x;
    return t;
}
```

Khi gọi hàm luythua() trong hàm main(), ta thấy rằng đối số thật là n sẽ được truyền đi cho đối số giả cùng tên (đối số giả này có thể trùng hoặc khác tên), nếu số n nhập vào là 3 thì trị 3 này cũng được chép sang cho đối số giả n của hàm để tính toán, như vậy mọi việc tính toán trong hàm đều tính toán trên biến cục bộ n của hàm, chứ không trực tiếp tính toán trên biến n thật của hàm main(), do đó sau khi gọi hàm luythua() ở dòng lệnh:

```
xn = luy_thua(x, n);
```

thì biến n của hàm main(), trước đó ví dụ là 3, thì bây giờ cũng là 3, chứ không phải là 0 do việc trừ dần trị n trong hàm luy_thua(), nên lệnh

```
printf ("Tri cua so mu la %d", n);
```

vẫn in giá trị của n là 3.

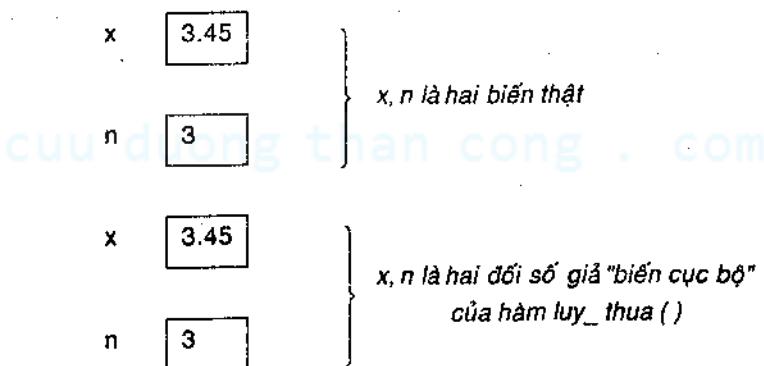
Để dễ hình dung ta có thể xem các hình biểu diễn các biến trước, trong khi và sau khi gọi hàm luy_thua():

Trước khi gọi hàm lũy thừa, ví dụ x và n đã được nhập trị:

$x = 3.45$

$n = 3$

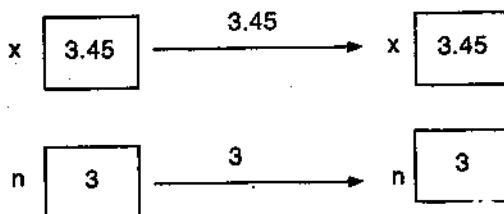
do đó ta có hai hộp biến:



Khi gọi hàm lũy thừa, trị của biến x và n sẽ được chép vào cho hai đối số giả x và n , do đó ta có đồng thời các hộp biến như sau khi

vào trong hàm luy_thua():

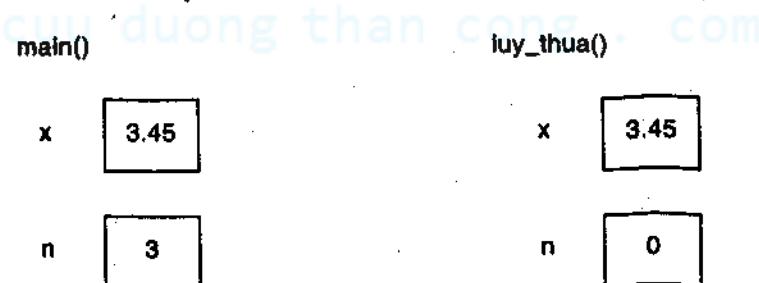
Biến của hàm main() Biến của hàm luy_thua()



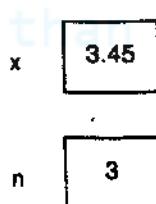
Như vậy, mọi việc tính toán biến *n* trong hàm luy_thua đều sử dụng đối số giả *n*, mà có thể được xem như là biến cục bộ của hàm, do đó sau khi kết thúc vòng for

```
for ( ; n >0; n--)
    t *= x;
```

thì các biến sẽ có trị như sau:



Như vậy, lúc này biến *n* cục bộ của hàm *luy_thua* có trị 0, còn biến *n* của chương trình chính thì vẫn có trị là 3.



Khi kết thúc hàm *luy_thua()*, các đối số *x* và *n* của hàm *luy_thua()* không còn ý nghĩa nữa, mọi việc tính toán nếu có trên *x* và *n* chính là trên *x* và *n* của hàm *main()*.

Như vậy, với cách chuyển đổi số như thế này, biến gốc chỉ truyền trị cho đối số của hàm, mà sẽ không bị thay đổi trị bởi các phép tính toán bên trong hàm được gọi. Cách chuyển đổi số như thế này được gọi là chuyển đổi số theo giá trị (*call by value*). Từ đó mở rộng ra đối với cách chuyển đổi số là tham trị, C chấp nhận đối số thật có thể là một hằng, biến hoặc biểu thức bất kỳ, miễn rằng biểu thức đó có trị thuộc kiểu mà hàm chấp nhận cho truyền vào hàm.

Ví dụ 10.5

Ta có thể gọi hàm `luy_thua()` và truyền cho hàm này một biểu thức:

`xn = luy_thua(3*a + x , 5);`

Lúc đó, C sẽ tính toán giá trị của biểu thức $(3*a + x)$ trước, sau đó chép kết quả vào biến `x` của hàm `luy_thua()`, và quá trình tính toán trong hàm `luy_thua()` sẽ diễn ra bình thường.

Tuy nhiên, cách truyền tham số như trên không thể thay đổi trị của biến, mà điều này đôi khi lại cần thiết; ví dụ như ta cần thiết kế một hàm nhập các số vào cho biến, nếu viết hàm như ví dụ sau đây thì sẽ thu được kết quả không mong muốn.

Ví dụ 10.6 Viết chương trình sử dụng hàm nhập số liệu

```
#include <stdio.h>
#include <conio.h>
void nhap_tri (int a, int b);
main()
{
    int a = 0, b = 0;
    clrscr();
    printf ("Truoc khi goi ham nhap_tri: a = %d, b = %d\n", a, b);
    nhap_tri (a, b);
    printf ("Sau khi goi ham nhap_tri a = %d, b = %d\n", a, b);
    getch();
}
void nhap_tri (int a, int b)
{
    printf ("Moi nhap hai so: ");
    scanf ("%d %d", &a, &b);
}
```

Chương trình này sẽ in ra màn hình các dòng sau:

Trước khi gọi hàm nhap_tri: a = 0, b = 0

Mời nhập hai số: 5 4

Sau khi gọi hàm nhap_tri: a = 0, b = 0

Như vậy, thay vì thu được $a = 5$, $b = 4$ thì ta cũng chỉ nhận được $a = 0$, $b = 0$ mà thôi. Trong trường hợp này, muốn đạt được kết quả như mong muốn, ta không thể truyền theo giá trị mà cần truyền theo địa chỉ của biến mà ta thực sự muốn thay đổi trị. Cụ thể cách truyền như thế nào ta sẽ đề cập cụ thể trong chương 14, chương đề cập về con trỏ (pointer), trong chương này ta sẽ hiểu rõ hơn tại sao đối với mảng, việc truyền đối số luôn ở dạng tham số biến. Và như vậy, mọi sự thay đổi mà hàm thực hiện trên mảng thì chính là thay đổi trên mảng đối số thật.

Ví dụ 10.7

Thiết kế chương trình dùng hàm nhập mảng, tính tổng các phần tử và in ra màn hình kết quả.

```
#include <stdio.h>
#include <conio.h>
void nhap_tri (int a[], int n);
int tong (int a[], int n);
main()
{
    int a[100], n;
    int sum;
    clrscr();
    printf ("Mời nhập số phần tử của mảng: ");
    scanf ("%d", &n);
    nhap_tri (a, n);
    sum = tong (a, n);
    printf ("Tổng các phần tử của mảng là: %d \n", sum);
    getch();
}
void nhap_tri (int a[], int n)
{
```

```

int i;
printf ("Moi nhap cac phan tu cua mang: ");
for (i = 0; i <n; i++)
    scanf ("%d", &a[i]);
}
int tong (int a[], int n)
{
    int i, s = 0;
    for (i = 0; i < n; s += a[i++])
        ;
    return s;
}

```

Trong chương trình trên mảng *a[]* được truyền cho hàm với dạng tương tự như truyền theo kiểu tham trị, nhưng trị của mảng hoàn toàn có thể thay đổi được. Do đó, khi làm việc trên mảng ta cần phải lưu ý điều này.

10.4 KẾT QUẢ TRẢ VỀ CỦA HÀM - LỆNH RETURN

C không có sự phân biệt giữa thủ tục (*procedure*) và hàm (*function*), mà thủ tục cũng được xem là một hàm không trả về giá trị nào cả. Để khai báo kiểu trả về từ hàm như vậy, C đưa ra kiểu *void*, tạm gọi là kiểu không kiểu.

Thật ra, C không quan tâm một hàm nào đó có trả về trị hay không, hoặc trị trả về từ hàm có dùng đến không. Việc không quan tâm đến trị trả về từ hàm làm cho cách viết chương trình trên ngôn ngữ C trở nên rất linh động, ví dụ có một lúc nào đó ta cần nhận một ký tự và chương trình cần ký tự này để xử lý, ta có thể dùng hàm

`getch()`

hoặc

`getche()`,

khi thực hiện phép gán

`c = getch();`

hoặc

`c = getche();`

để biến c lưu ký tự cần xử lý, tuy nhiên nếu có một lúc nào đó ta muốn chương trình nhận một ký tự mà ký tự này không cần được xử lý thì ta cũng gọi được các hàm trên, nhưng không cần phải gán trị vào cho biến khi gọi hàm, tức là chỉ gọi một cách đơn giản

`getch();`

hoặc

`getche();`

và C hoàn toàn chấp nhận điều này.

Trong chương trình, ta cũng biết lệnh **return** dùng để thực hiện việc trả trị của hàm về cho nơi gọi nó, dù trị này có được sử dụng hay không tùy nơi gọi. Do đó nếu ta có một hàm sau đây thì ta có thể gọi nó theo mục đích sử dụng.

Ví dụ 10.8

Thiết kế hàm so sánh hai số.

```
int so_sanh (int a, int b)
{
    if (a >b)
    {
        printf ("So %d lon hon so %d", a, b);
        return 1;
    }
    else if (a == b)
    {
        printf ("So %d bang so %d", a, b);
        return 0;
    }
    else /* a <b */
    {
        printf ("So %d nho hon so %d", a, b);
        return -1;
    }
}
```

Trong chương trình 1 sau đây, trị trả về của hàm `so_sanh()` sẽ không được sử dụng, trong khi chương trình 2 lại sử dụng trị này vào mục đích cụ thể và tính trị tuyệt đối của hiệu hai số.

Chương trình 1:

```
#include <stdio.h>
#include <conio.h>
int so_sanh (int a, int b);
main()
{
    int so1, so2;
    clrscr();
    printf("Moi nhap hai so: ");
    scanf ("%d %d", &so1, &so2);
    so_sanh (so1, so2);           —————→ hàm so_sanh() được gọi
    getch();                      không cần giá trị trả về
}
int so_sanh (int a, int b)
{
    if (a >b)
    {
        printf ("So %d lon hon so %d", a, b);
        return 1;
    }
    else if (a == b)
    {
        printf ("So %d bang so %d", a, b);
        return 0;
    }
    else /* a < b */
    {
        printf ("So %d nho hon so %d", a, b);
        return -1;
    }
}
```

Trong chương trình 1, việc gọi hàm `so_sanh()` chỉ để in ra kết quả so sánh của hai số mà không cần tính trị tuyệt đối của hiệu hai số nên việc gọi hàm có dạng như gọi thủ tục.

Chương trình 2:

```

#include <stdio.h>
#include <conio.h>
int so_sanh (int a, int b);
main()
{
    int so1, so2;
    int kq;
    clrscr();
    printf ("Moi nhap hai so: ");
    scanf ("%d %d", &so1, &so2); ————— hàm so_sanh() được gọi, và có
    kq = so_sanh (so1, so2);           biến nhận trị trả về từ hàm
    switch (kq)
    {
        case -1:
            printf (" , nen tri tuyet doi hieu hai so la: %d\n,so2-so1");
            break;
        case 0:
        case 1:
            printf (" , tri tuyet doi hieu hai so la %d\n , so1-so2");
            break;
    }
    getch();
}
int so_sanh (int a, int b)
{
    if (a > b)
    {
        printf ("So %d lon hon so %d", a, b);
        return 1;
    }
    else if (a == b)
    {

```

Cuu duong than cong . com

```

    printf ("So %d bang so %d", a, b);

    return 0;

}

else /* a < b */

{
    printf ("So %d nho hon so %d", a, b);

    return -1;

}
}

```

Trong chương trình 2 vì cần phải biết xem trị nào lớn hơn nên chương trình gọi phải xét đến trị trả về từ hàm, và như vậy việc gọi hàm cần phải gán trị trả về cho một biến để xét.

Đi nhiên, nếu hàm không trả về giá trị nào cả, thì có thể không cần lệnh return, khi đó lúc khai báo hàm kiểu trả về là void.

CuuDuongThanCong . com

Ví dụ 10.9 Xét hàm sau:

```

void so_sanh (int a, int b)

{
    if (a > b)
        printf ("So %d lon hon so %d", a, b);

    else if (a == b)
        printf ("So %d bang so %d", a, b);

    else /* a < b */
        printf ("So %d nho hon so %d", a, b);
}

```

Khi khai báo hàm mà ta không nêu cụ thể kiểu trả về của hàm, C mặc nhiên xem như hàm trả về kết quả là int. Do đó khi một hàm mà có kiểu trả về là int thì có thể ta không cần nêu kiểu trả về cụ thể. Tuy nhiên đối với ngôn ngữ C, là một ngôn ngữ có tính linh động rất cao, khi khai báo hàm ta cần nêu cụ thể kiểu trả về của hàm để thuận tiện cho việc tính toán khi gọi hàm tính toán trong một biểu thức nào đó.

Ví dụ 10.10 Thay vì khai báo hàm

```
int so_sanh (int a, int b)
{
    if (a > b)
        return 1;
    else if (a == b)
        return 0;
    else /* a < b */
        return -1;
}
```

thì ta có thể khai báo hàm như sau

```
so_sanh (int a, int b)
{
    if (a > b)
        return 1;
    else if (a == b)
        return 0;
    else /* a < b */
        return -1;
}
```

Đối với các hàm có kiểu trả về trị khác int, thì khi khai báo ta cần phải trình bày đầy đủ các thành phần của hàm, và khi gọi sử dụng hàm thì trong hàm gọi cần phải nêu kết quả trả về của các hàm được gọi trong đó. Kiểu khai báo kết quả này có thể được đặt bên ngoài tất cả các hàm để thông báo cho tất cả các hàm về trị trả về của nó, hoặc có thể được đặt trong hàm mà hàm sử dụng được gọi, kiểu khai báo này tương tự như khai báo biến, cú pháp như sau:

kiểu tên_hàm();

trong đó cặp dấu () chứng tỏ đây là một khai báo hàm.

Ví dụ 10.11

Trong hàm main() sử dụng hàm so_sanh() để so sánh hai số thì ta cần khai báo hàm này trong phần khai báo biến của hàm main() như sau:

int so_sanh();

do đó, chương trình 1 ở Ví dụ 10.8 theo cách khai báo hàm này sẽ được viết như sau:

```
#include <stdio.h>
#include <conio.h>
main()
{
    int so_sanh (); → nơi khai báo hàm sử dụng
    int so1, so2;
    clrscr();
    printf ("Moi nhap hai so: ");
    scanf ("%d %d", &so1, &so2);
    so_sanh (so1, so2);
    getch();
}
int so_sanh (int a, int b)
{
    if (a > b)
    {
        printf ("So %d lon hon so %d", a, b);
        return 1;
    }
    else if (a == b)
    {
        printf ("So %d bang so %d", a, b);
        return 0;
    }
    else /* a < b */
    {
        printf ("So %d nho hon so %d", a, b)
        return -1;
    }
}
```

Chú ý rằng, ta có thể kết hợp khai báo chung hàm và biến

```
int so_sanh(), so1, so2;
```

Chương trình trong trường hợp này vẫn chạy đúng, nhưng cách viết và khai báo hàm như thế này sẽ không kiểm soát được kiểu trị

cũng như số đối số được đưa vào cho hàm, như trong ví dụ trên, nếu ta gọi hàm mà chỉ cung cấp một đối số như sau:

```
so_sanh(so2);
```

thì C hoàn toàn không bắt lỗi, tuy nhiên chương trình lại chạy sai, nên trong thực tế C đưa ra khái niệm prototype để thay thế cho kiểu khai báo hàm như trên.

10.5 PROTOTYPE CỦA MỘT HÀM

Như vậy, để một hàm có thể sử dụng trong một hàm khác thì trong hàm sử dụng phải có khai báo hàm cần sử dụng, khai báo này bình thường tương tự như khai báo biến mà ta đã xét ở mục trên, tuy nhiên khai báo này rất hạn chế ở chỗ không cho phép kiểm tra số đối số thật đưa vào hàm cũng như kiểu của đối số có phù hợp không. Để khắc phục những lỗi trên, trong những phát triển sau này của C theo ANSI, người ta đưa ra khái niệm prototype của một hàm, đây thật sự là một dạng khai báo hàm mở rộng hơn, có dạng tổng quát như sau:

```
kiểu tên_hàm (danh_sách_khai_báo đối_số);
```

với:

- **kiểu** bắt buộc phải có, là kiểu của trị trả về từ hàm có tên là `tên_hàm()`. Kiểu này có thể là một trong các kiểu dữ liệu cơ bản của C hoặc có thể là kiểu void, nếu muốn nhấn mạnh rằng hàm không trả về giá trị gì cả, hoặc kiểu con trỏ (sẽ được đề cập trong chương kiểu con trỏ) hoặc kiểu tự định nghĩa.
- **danh_sách_khai_báo đối_số** là danh sách các khai báo đối số của hàm, mỗi khai báo có dạng một khai báo biến, các khai báo đối số này cách nhau bằng dấu phẩy (,).

Ví dụ 10.12

```
int so_sanh (int a, int b);
void gptb1 (double a, double b, double c);
char kiem_tra (double n);
```

C cho phép khai báo prototype của hàm trong phần khai báo đối số chỉ cần có kiểu mà không cần có tên của đối số giả.

Ví dụ 10.13

```
int so_sanh (int, int);
```

Khi khai báo prototype của hàm, **danh_sách_khai_báo_đối_số** có thể không cần khai báo hết các đối số, khi đó C cho phép dùng dấu (...) để kết thúc danh sách khai báo đối số.

Ví dụ 10.14

Như prototype của hàm chuẩn printf() trong file stdio.h của C:

```
int printf (char format[], ...);
```

- Công dụng của prototype của hàm: prototype của một hàm ngoài việc dùng để khai báo kiểu của kết quả trả về từ một hàm, nó còn được dùng để kiểm tra số đối số. Việc viết prototype trước khi sử dụng giúp cho chương trình biên dịch có thể kiểm tra và báo lỗi khi có sự không phù hợp diễn ra, do đó việc nắm vững công dụng của prototype là cần thiết.
- Prototype của một hàm thực tế là một dạng khai báo hàm mở rộng, do đó nó có tác dụng hoàn toàn như một khai báo hàm: khai báo kết quả trả về từ một hàm mà kết quả này đến lượt nó lại được dùng để tính toán trong một biểu thức, nếu kết quả này là thích hợp kiểu với các toán hạng khác trong biểu thức thì C sẽ tính toán biểu thức, còn nếu kiểu của kết quả là không phù hợp thì C sẽ báo lỗi.
- Kiểm tra số đối số được đưa vào cho hàm: khi một hàm đã có khai báo prototype, mỗi khi hàm được gọi sử dụng từ một hàm khác thì chương trình biên dịch sẽ kiểm tra số đối số thật được gởi đến cho hàm với số đối số giả khai báo trong prototype. Nếu có một sự không phù hợp thì thông báo lỗi. Nếu không khai báo prototype hoặc trong prototype chỉ khai báo một phần số đối số, (phần còn lại không nêu rõ bằng dấu (...)) thì các phần còn lại sẽ không được kiểm tra.

Ví dụ 10.15

Nếu đã khai báo prototype

```
int so_sanh (int a, int b);
```

mà khi gọi hàm ta chỉ gửi một đối số như sau:

so_sanh (so2);

thì sẽ bị C phát hiện và báo lỗi. Trong trường hợp không có prototype này thì C hoàn toàn không bắt lỗi, tuy nhiên điều đó sẽ làm cho chương trình bị sai kết quả mà lý do cụ thể không thể giải thích được. Mời độc giả hãy xem chương trình sau đây:

```
#include <stdio.h>
#include <conio.h>
main()
{
    int so1, so2;
    int n;
    clrscr();
    printf("Mời nhập hai số: ");
    scanf ("%d %d", &so1, &so2);
    so_sanh (so2);           → hàm so_sanh() được gọi chỉ
    getch();                 với một đối số là so2
}
int so_sanh (int a, int b)
{
    if (a > b)
        printf ("So %d lon hon so %d", a, b);
    else if (a == b)
        printf ("So %d bang so %d", a, b);
    else /* a < b */
        printf ("So %d nho hon so %d", a, b);
}
```

Chương trình này ví dụ cho ra màn hình các thông tin sau đây:

Moi nhap hai so: 4 8

So 8 bang so 8

Như vậy, luôn luôn chương trình in ra kết quả bằng dù việc nhập có như thế nào đi nữa.

- Chuyển kiểu của đối số: khi một hàm được gọi, mà hàm đó có prototype, các đối số được gởi cho hàm sẽ được chuyển kiểu bắt buộc theo kiểu của các đối số được khai báo trong prototype, sự chuyển kiểu này làm cho các đối số được sử dụng phù hợp với các phép toán trong thân hàm.

Ví dụ 10.16

Xét chương trình sau:

```
#include <stdio.h>
#include <conio.h>
double chia (double a, double b);
main()
{
    int so1, so2;
    double kq;
    clrscr();
    printf ("Moi nhap hai so: ");
    scanf ("%d %d", &so1, &so2);
    kq = chia (so1, so2);
    printf ("ket qua la %.2f", kq);
    getch();
}
double chia (double a, double b)
{
    return a/b;
}
```

Trong chương trình trên, hai biến *so1* và *so2* dù là *int* vẫn được chuyển sang kiểu *double* khi chúng được gọi đến làm đối số của hàm, do đó thao tác chia trong hàm là chia hai số thực chứ không phải chia hai số nguyên, vì vậy kết quả của trị trả về là một số thực, nên nếu nhập vào hai số 10 và 3 thì kết quả in là 3.33 (chứ không phải là 3.00).

Đĩ nhiên, trong những trường hợp mà sự chuyển kiểu không cho phép thực hiện thì C sẽ đưa ra các thông báo lỗi, hoặc một lời cảnh báo (*warning*) rằng sự chuyển kiểu có thể gây ra sai lầm cho bài toán.

Qua những công dụng trên, ta thấy prototype thực sự rất có ích, nó giúp chúng ta tránh được những sai sót không đáng và tiết kiệm được nhiều thời gian để tìm và sửa lỗi. Do đó, khi lập trình trên C, việc viết hàm nên luôn đi đôi với việc khai báo prototype của hàm.

Cần lưu ý rằng, đối với các hàm chuẩn trong thư viện C, prototype của chúng đã được C viết sẵn và để trong các file có phần mở rộng là .h, muốn lấy các prototype này vào chương trình ta cần ra chỉ thị bao hàm file.h chứa prototype của các hàm cần sử dụng vào đầu chương trình bằng lệnh tiền xử lý #include theo cú pháp sau:

```
# include <file. h>
```

Ví dụ 10.17 Khi ra lệnh

```
#include <stdio.h>
```

vào đầu chương trình là chúng ta đã lấy toàn bộ nội dung của file stdio.h được "include" vào file chương trình, trong đó có các prototypte của các hàm cần sử dụng (và cả các hàm không cần sử dụng và cả các hằng không cần sử dụng!), khi đó chúng ta có thể yên tâm sử dụng các hàm thư viện mà prototype đã được khai báo trong file này.

10.6 HÀM ĐỆ QUY

Trong thực tế, ta rất hay gặp khái niệm đệ quy (*recursion*), ví dụ như khi xem truyền hình ta sẽ thấy màn ảnh truyền hình đang phát hình chính nó, và trong ảnh của máy thu hình trong hình ta lại thấy một ảnh nữa của máy thu hình nữa, cứ thế trên màn hình có rất nhiều máy thu hình, cái này lồng vào cái kia. Hoặc khái niệm một số nguyên, người ta định nghĩa một số nguyên bằng khái niệm đệ quy như sau: số nguyên là một số mà theo sau nó là một... số nguyên, và tập hợp số nguyên là vô hạn nhưng ta có thể biểu diễn nó bằng một định nghĩa hữu hạn: chỉ dựa vào hai số nguyên để định nghĩa số nguyên.

C được gọi là một ngôn ngữ đệ quy vì C cho phép một hàm có thể gọi đến chính nó một cách trực tiếp, hoặc gián tiếp (tức là gọi qua trung gian một hàm khác), khi đó ta nói hàm đó có tính đệ quy (*recursive*). Như vậy, một giải thuật đệ quy sẽ dẫn đến một sự lặp đi lặp lại không kết thúc các thao tác, nhưng trong thực tế, chúng cần phải được kết thúc. Sự kết thúc đó cũng tương tự như việc ta chỉ chấp nhận thấy đến ảnh đệ quy cấp n nào đó trong ảnh hiện thời mà thôi (vì các ảnh đệ quy cấp nhỏ hơn nữa sẽ không còn được thấy rõ nữa), đối với giải thuật đệ quy ta gọi là điều kiện kết thúc đệ quy. Như vậy, việc gọi đệ quy phải có một lúc nào đó không diễn ra nữa, điều kiện này cũng giống như điều kiện cần thiết để kết thúc một vòng lặp do while.

Để dễ hình dung, ta hãy xét ví dụ cụ thể hàm trong chương trình C tính giai thừa. Trong toán học, giai thừa có thể được định nghĩa theo khái niệm đệ quy như sau:

$$n! = n^* (n-1)!$$

Chương trình ví dụ cụ thể như sau:

Ví dụ 10.18

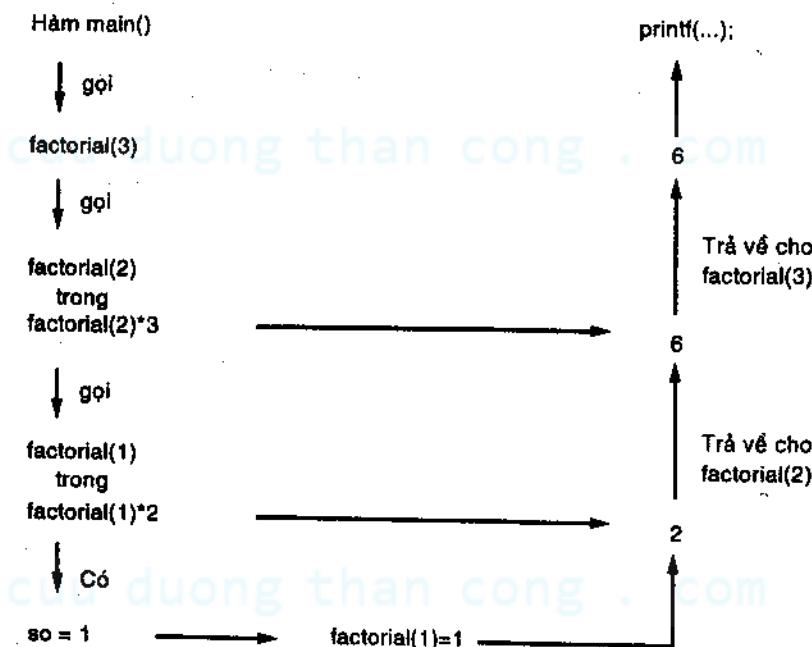
```
#include <stdio.h>
#include <conio.h>
long factorial (long so);
int main()
{
    long so, kq = 0;
    clrscr();
    printf ("Moi nhap mot so 0: ");
    scanf ("%ld", &so);
    kq = factorial (so);
    printf ("Ket qua %ld! la %ld \n", so, kq);
    getch();
    return 0;
}
long factorial (long so)
{
```

```

if (so > 1)
    return (factorial(so - 1) * so);
else
    return 1;
}

```

Trong ví dụ trên, ta thấy hàm factorial () được thiết kế đệ quy, trong thân nó lại gọi đến nó, tuy nhiên ở đây độc giả cần lưu ý điều kiện kết thúc đệ quy, khi biến so trong hàm mà lớn hơn 1 thì việc tính toán vẫn tiếp tục, tức vẫn tiếp tục gọi đệ quy, còn khi so là bằng 1 thì việc gọi đệ quy kết thúc, lúc này lại đến giai đoạn trả trị về cho lần gọi đệ quy trước đó. Ta có thể hình dung quá trình gọi đệ quy của chương trình trên với số được nhập để tính đệ quy là 3 như sau:



Đệ quy là một khả năng khá mạnh của C. Tuy nhiên, khi thiết kế đệ quy, điều quan trọng là phải nắm được quá trình hàm gọi đệ quy, và điều kiện kết thúc đệ quy, nếu vì lý do gì đó mà điều kiện này không có hoặc sai thì chương trình sẽ chạy mãi cho đến khi máy bị "đóng băng", hay ta có thể gọi theo kiểu vòng lặp là chương trình lặp vô tận. Do đó, trong những trường hợp mà việc dùng vòng lặp còn hiệu quả thì nên dùng vòng lặp, khi đó chương trình sẽ dễ

kiểm soát hơn. Một khác khi sử dụng đệ quy, các biến xin trong hàm cũng như các địa chỉ trả về hàm gọi trước đều được lưu vào stack, các biến này tồn tại cho đến khi nào việc gọi đệ quy chấm dứt, như vậy nếu trong một hàm mà có nhiều biến cục bộ quá chương trình có thể bị thiếu bộ nhớ và không chạy được.

Trong thực tế, người ta thường thiết kế đệ quy cho các kiểu dữ liệu được định nghĩa theo kiểu đệ quy, như cấu trúc cây hoặc một số giải thuật đặc biệt trong đó viết bằng đệ quy sẽ đơn giản hơn nhiều. Vấn đề này sẽ được nêu chi tiết trong chương 19.

10.7 HIỆN THỰC HÀM TRONG C

Bây giờ chúng ta hãy xem làm thế nào hàm trong C được thực hiện ở cấp máy. Các hàm trong C tương đương các chương trình con trong hợp ngữ LC-3 (mà ta đã được biết trong chương 6), và lối thao tác của chúng là như nhau. Trong C, để gọi hàm ta cần ba bước cơ bản: (1) các tham số từ nơi gọi được chuyển cho hàm được gọi và điều khiển được chuyển cho hàm được gọi, (2) hàm được gọi thực hiện tác vụ, (3) một giá trị trả về được gửi ngược lại cho nơi gọi hàm, và điều khiển được trả về cho nơi gọi. Một quy ước quan trọng mà chúng ta quy định cho cơ chế gọi là hàm được gọi nên độc lập với nơi gọi. Nghĩa là, một hàm cần được viết để có thể được gọi từ bất cứ hàm nào. Trong mục này chúng ta sẽ xem làm thế nào điều này được thực hiện bằng LC-3.

10.7.1 Ngăn xếp thực thi (Run-time stack)

Trước khi tiến hành, đầu tiên chúng ta cần thảo luận về một thành phần rất quan trọng của hàm trong C và các ngôn ngữ lập trình hiện đại khác, đó là cách thức kích hoạt một hàm khi nó được gọi. Nghĩa là, khi một hàm bắt đầu thực thi, các biến cục bộ của nó phải được cấp các vị trí trong bộ nhớ. Có thể hiểu là mỗi hàm đều có một phần vùng nhớ dùng để chứa các biến cục bộ của nó và một số thành phần khác, mà ta gọi là mẫu tin kích hoạt. Mỗi biến cục bộ được khai báo trong hàm đều sẽ có một vị trí trong mẫu tin kích hoạt mà vị trí đầu của nó được thanh ghi (R5) con trả khung chỉ tới. Như vậy câu hỏi là: mẫu tin kích hoạt được định vị ở đâu trong bộ nhớ? Có hai chọn lựa như sau.

Chọn lựa 1: Bộ dịch có thể quy định một cách hệ thống một số vùng trống trong bộ nhớ cho mỗi hàm để chứa mẫu tin kích hoạt. Hàm A có thể được gán cho vùng bộ nhớ X để đặt mẫu tin kích hoạt của nó, hàm B lại có thể được gán cho vùng nhớ Y, và cứ thế, miễn là các vùng nhớ này là không trùng lấp nhau. Ta có thể thấy cách này hoàn toàn không phức tạp, nhưng lại có một giới hạn cho chọn lựa này, đó là cái gì xảy ra khi hàm A gọi chính nó? Tức khi có đệ quy diễn ra, và đây là một khái niệm lập trình quan trọng mà độc giả có thể xem ở chương 19. Nếu một hàm gọi chính nó, như hàm A chẳng hạn, thì bản được gọi của hàm A sẽ ghi đè các biến cục bộ của bản gọi của hàm A, và chương trình sẽ không chạy như ta mong đợi. Với ngôn ngữ lập trình C, vốn cho phép gọi đệ quy, chọn lựa 1 sẽ không được sử dụng.

Chọn lựa 2: Mỗi lần một hàm được gọi, một mẫu tin kích hoạt được định vị cho riêng nó trong bộ nhớ. Khi hàm đó trở về nơi gọi, vùng nhớ của mẫu tin kích hoạt đó sẽ được dời lại để gán cho các hàm khác sau này. Chọn lựa này về khái niệm có vẻ khó hơn chọn lựa 1, nó cho phép gọi đệ quy. Mỗi lần gọi một hàm đều lấy một vùng nhớ riêng cho các biến cục bộ của nó. Ví dụ, nếu hàm A gọi hàm A, mẫu tin kích hoạt của bản được gọi sẽ được định vị ở một vị trí khác với vị trí của bản gọi trong bộ nhớ, và dĩ nhiên là hai mẫu tin kích hoạt này độc lập nhau. Có một tác nhân làm giảm bớt tính phức tạp của việc thực hiện chọn lựa 2: quá trình gọi của hàm (tức hàm A gọi hàm B, hàm B lại gọi hàm C, ...) có thể được theo dõi bằng một cấu trúc dữ liệu ngăn xếp (Chương 6). Chúng ta hãy minh họa vấn đề này bằng ví dụ 10.19 sau. Đoạn code ở ví dụ này có ba hàm: main, Watt, và Volta. Thao tác trong của từng hàm là không quan trọng cho ví dụ này, nên chúng ta bỏ qua chúng nhưng các dòng code quan trọng thể hiện việc gọi hàm, biến cục bộ, ... vẫn có đủ để chúng ta thấy rõ quá trình gọi và thực thi hàm.

Ví dụ 10.19

```

1 int main()
2 {
3     int a;
4     int b;
```

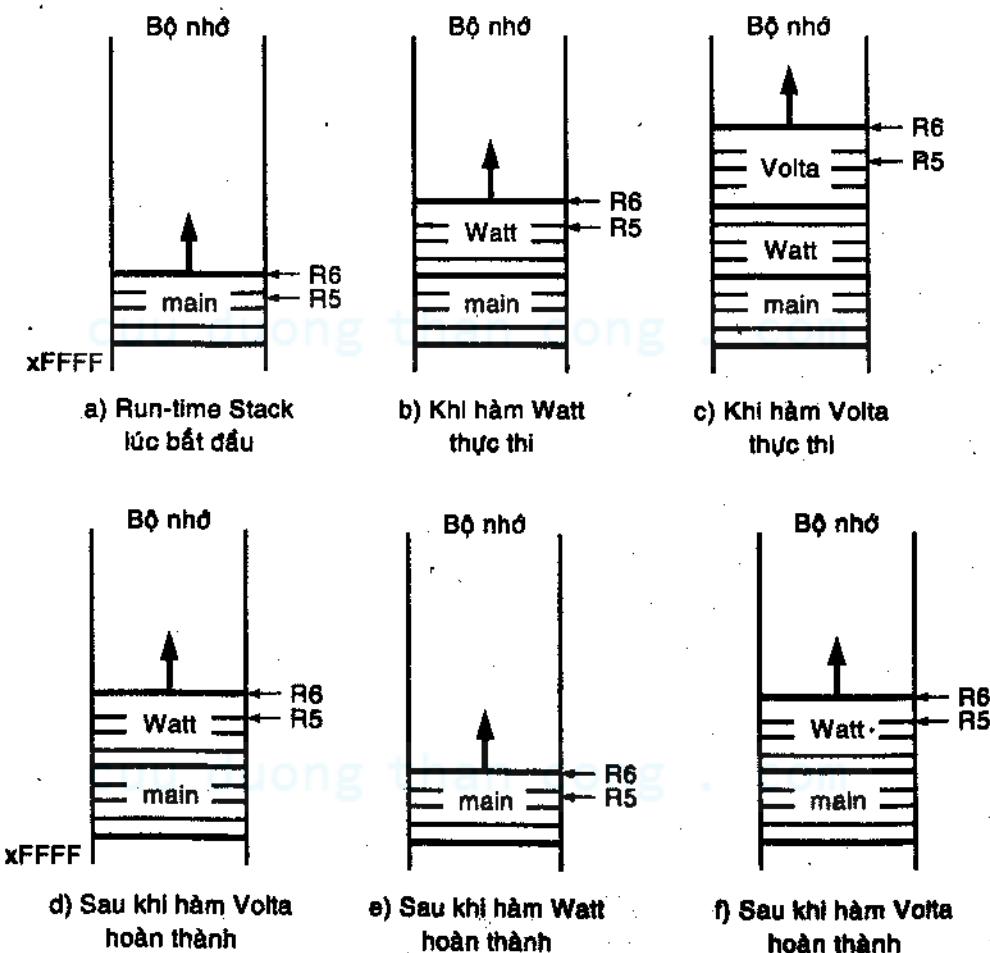
```
5
6      ..
7      b = Watt (a);
8      b = Volta (a, b);
9  }
10
11 int Watt (int a)
12 {
13     int w;
14
15     ..
16     w = Volta (w, 20);
17
18     return w;
19 }
20
21 int Volta (int q, int r)
22 {
23     int k;
24     int m;
25
26     ..
27     return w;
28 }
```

Trong ví dụ trên, hàm main() gọi hàm Watt() ở dòng 7, hàm Watt() gọi hàm Volta() ở dòng 16. Sau đó, điều khiển được trả về cho hàm main(), và rồi hàm main() tiếp tục gọi hàm Volta().

Mỗi hàm đều có một mẫu tin kích hoạt riêng chứa các biến cục bộ của nó, thông tin hàm, và các tham số nhập từ nơi gọi (chúng ta sẽ đề cập nhiều hơn về các tham số và thông tin hàm trong những phần kế). Bất cứ khi nào một hàm được gọi, mẫu tin kích hoạt của nó sẽ được định vị ở một chỗ nào đó trong bộ nhớ, và như chúng ta đã

nêu trong mục trước, theo kiểu như stack. Điều này được minh họa trong sơ đồ ở hình 10.1

Mỗi vùng có gạch ngang biểu diễn mảng tin kích hoạt của một lần gọi hàm riêng biệt. Trình tự các hình cho thấy cách thức một Ngăn xếp thực thi (run-time stack) cấp và thu lại vùng nhớ cho mảng tin kích hoạt tùy vào hàm được gọi và trả về nơi gọi nó. Chúng ta hãy nhớ là, mỗi khi đẩy một phần tử vào stack, thì đỉnh stack sẽ di chuyển “tăng lên”, theo hướng các ô nhớ có địa chỉ thấp dần.



Hình 10.1 Một số hình trạng thái của Ngăn xếp thực thi (run-time stack) trong khi chương trình ví dụ 10.19 thực thi

Hình 10.1(a) là một hình của ngăn xếp thực thi (run-time stack) khi chương trình bắt đầu thực thi. Vì việc thực thi của một chương

trình C bắt đầu từ trong hàm main(), nên mẫu tin kích hoạt của hàm main() là cái đầu tiên được định vị trong stack. Hình 10.1(b) cho thấy ngăn xếp thực thi (*run-time stack*) ngay sau khi hàm Watt() được hàm main() gọi. Chú ý là các mẫu tin kích hoạt được định vị theo kiểu như stack. Nghĩa là, bất cứ khi nào một hàm được gọi, thì mẫu tin kích hoạt của nó được đưa vào stack. Bất kỳ lúc nào khi hàm trở về, mẫu tin kích hoạt của nó được lấy ra khỏi stack. Từ Hình 10.1(c) tới (f) cho thấy trạng thái của ngăn xếp thực thi (*run-time stack*) ở những thời điểm khác nhau trong quá trình thực thi của ví dụ trên. Lưu ý là thanh ghi R5 chỉ tới vị trí nào đó bên trong mẫu tin kích hoạt (nó chỉ tới nhóm biến cục bộ). Còn thanh ghi R6 luôn luôn chỉ tới đỉnh của stack, và thường được gọi là con trỏ stack (*stack pointer*). Tổng quát mà nói, cả hai thanh ghi này có vai trò quan trọng trong việc hiện thực ngăn xếp thực thi (*run-time stack*) và các hàm trong C.

10.7.2 Quá trình hiện thực

Tới đây ta có thể thấy là có nhiều thứ diễn ra ở cấp máy khi mà một hàm được gọi. Các tham số phải được truyền vào, các mẫu tin kích hoạt được đưa vào và lấy ra, điều khiển chương trình được chuyển từ hàm này qua hàm kia. Một số trong các thứ này được noi gọi hàm hoàn thành, trong khi một số thứ khác lại do hàm được gọi thực hiện.

Để hoàn thành tất cả các việc này, các bước sau cần phải được thực hiện: đầu tiên, code của hàm gọi (*caller*) sẽ chép các đối số của nó vào vùng bộ nhớ để hàm được gọi (*callee*) có thể truy xuất được. Thứ hai, code ở nơi bắt đầu trong hàm được gọi đẩy mẫu tin kích hoạt của nó vào stack và lưu các thông tin trạng thái của biến cục bộ, thanh ghi, ... để khi điều khiển trả về cho nơi gọi, thì đối với nơi gọi mọi thứ như không có gì thay đổi, từ các biến cục bộ tới các thanh ghi. Thứ ba, hàm được gọi thực thi tác vụ của nó. Thứ tư, khi hàm được gọi hoàn thành việc của nó, mẫu tin kích hoạt của nó được lấy ra khỏi ngăn xếp thực thi (*run-time stack*) và điều khiển được trả về cho nơi gọi. Sau cùng, một khi điều khiển được trả về cho code nơi gọi, code thực thi sẽ truy tìm trị mà hàm được gọi trả về.

Bây giờ chúng ta sẽ khảo sát code LC-3 để thực hiện các thao tác này. Chúng ta hãy khảo sát đoạn code LC-3 liên quan tới việc gọi hàm như sau: $w = \text{Volta}(w, 20)$; ở dòng 16 trong ví dụ 10.19.

• Gọi hàm

Trong mệnh đề $w = \text{Volta}(w, 20)$, hàm Volta() được gọi với hai đối số. Sau đó, giá trị trả về từ hàm Volta() sẽ được gán cho biến nguyên cục bộ w. Trong khi dịch việc gọi hàm này, bộ dịch tạo ra code LC-3 làm các việc sau:

1. Truyền giá trị của hai đối số của hàm Volta() bằng việc đẩy chúng trực tiếp vào đỉnh của ngăn xếp thực thi (*run-time stack*) mà địa chỉ đang được chứa trong thanh ghi R6. Nghĩa là, thanh ghi R6 đang chứa địa chỉ của phần tử dữ liệu hiện thời trên cùng của ngăn xếp thực thi. Để đẩy một phần tử vào stack, đầu tiên chúng ta giảm thanh ghi R6 và rồi lưu giá trị dữ liệu vào ô nhớ được chỉ bởi R6, tức R6 được xem là địa chỉ nền cho thao tác này. Trong LC-3, các đối số của một lần gọi hàm C được đẩy vào stack từ **phải sang trái** theo thứ tự chúng xuất hiện trong hàm khi gọi. Trong trường hợp hàm Volta(), đầu tiên sẽ đẩy trị 20 (tức đối số bên phải nhất) và rồi tới trị w.
2. Chuyển điều khiển sang hàm Volta() nhờ lệnh JSR.

Mã LC-3 thực hiện gọi hàm này như sau:

```
; đẩy đối số thứ hai vào stack: 20
AND    R0, R0, #0          ; R0 ← 0
ADD    R0, R0, #20         ; R0 ← 20
ADD    R6, R6, #-1
STR    R0, R6, #0          ; Push 20
```

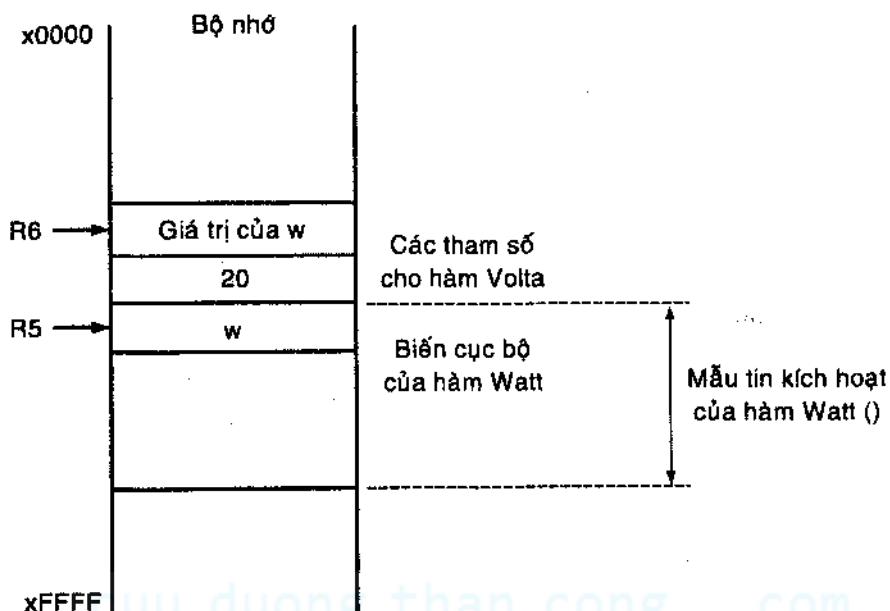
```
; đẩy đối số thứ nhất vào stack: w
LDR    R0, R5, #0          ; Lấy trị của biến cục bộ w
ADD    R6, R6, #-1
STR    R0, R6, #0          ; Push trị này vào stack
```

```
; gọi chương trình con
```

```
JSR    Volta
```

Hình 10.2 minh họa quá trình này. Chú ý là giá trị các đối số được đẩy ngay vào đỉnh của mẫu tin kích hoạt của hàm gọi (Watt).

Mẫu tin kích hoạt cho hàm được gọi (Volta) sẽ được tạo trong stack ngay trên đỉnh của mẫu tin của hàm gọi.



Hình 10.2 Stack thực thi của Watt() đẩy các trị để truyền cho hàm Volta()

- **Thực hiện hàm được gọi**

Lệnh được thực hiện ngay sau lệnh JSR trong hàm Watt() là lệnh đầu tiên trong hàm được gọi Volta().

Code ở chỗ bắt đầu của hàm được gọi xử lý một số thao tác liên quan tới việc gọi hàm. Việc đầu tiên là định vị bộ nhớ cho trị trả về bằng cách hàm được gọi sẽ đẩy một ô nhớ vào stack để chiếm chỗ qua việc giảm con trỏ stack. Và vị trí này sẽ được ghi vào giá trị cần trả về trước khi điều khiển trả về cho hàm gọi.

Kế tiếp, hàm được gọi lưu các thông tin về hàm gọi để khi việc gọi kết thúc, hàm gọi sẽ lấy lại điều khiển chương trình một cách đúng đắn. Đặc biệt, chúng ta cần lưu địa chỉ trả về của hàm gọi đang được chứa trong thanh ghi R7 (độc giả cần xem lại cách làm việc của lệnh JSR) và con trỏ khung của hàm gọi đang được chứa trong thanh ghi R5. Một điều quan trọng là cần chép sao con trỏ khung của hàm gọi mà ta gọi là liên kết động, để khi điều khiển trả về cho nơi gọi thì nơi gọi sẽ có thể truy xuất trở lại các biến cục bộ của nó. Nếu một

trong địa chỉ trả về hoặc liên kết động bị sai, thì chúng ta sẽ gặp sai sót khi tiếp tục thực thi hàm gọi khi hàm được gọi hoàn thành. Nên chúng ta cần phải sao lưu cả hai thứ này vô bộ nhớ.

Sau cùng, khi tất cả điều này được thực thi xong, hàm được gọi sẽ định vị đủ không gian trong stack cho các biến cục bộ của nó bằng việc chỉnh trị cho R6, và nó sẽ đặt R5 chỉ tới nền của các biến cục bộ này.

Tóm lại, dưới đây là danh sách các tác vụ phải diễn ra lúc bắt đầu một hàm:

1. Hàm được gọi lưu không gian trong stack cho trị trả về. Trị trả về được định vị ngay trên đỉnh các tham số của hàm được gọi.
2. Hàm được gọi đẩy một bản sao của địa chỉ trả về trong thanh ghi R7 vô stack.
3. Hàm được gọi đẩy một bản sao của liên kết động (con trả khung của hàm gọi) trong R5 vô stack.
4. Hàm được gọi định vị đủ không gian trong stack cho các biến cục bộ của nó và chỉnh thanh ghi R5 chỉ tới nền của danh sách biến cục bộ và thanh ghi R6 chỉ tới đỉnh stack.

Đoạn code hoàn thành việc này cho hàm Volta() như sau:

```

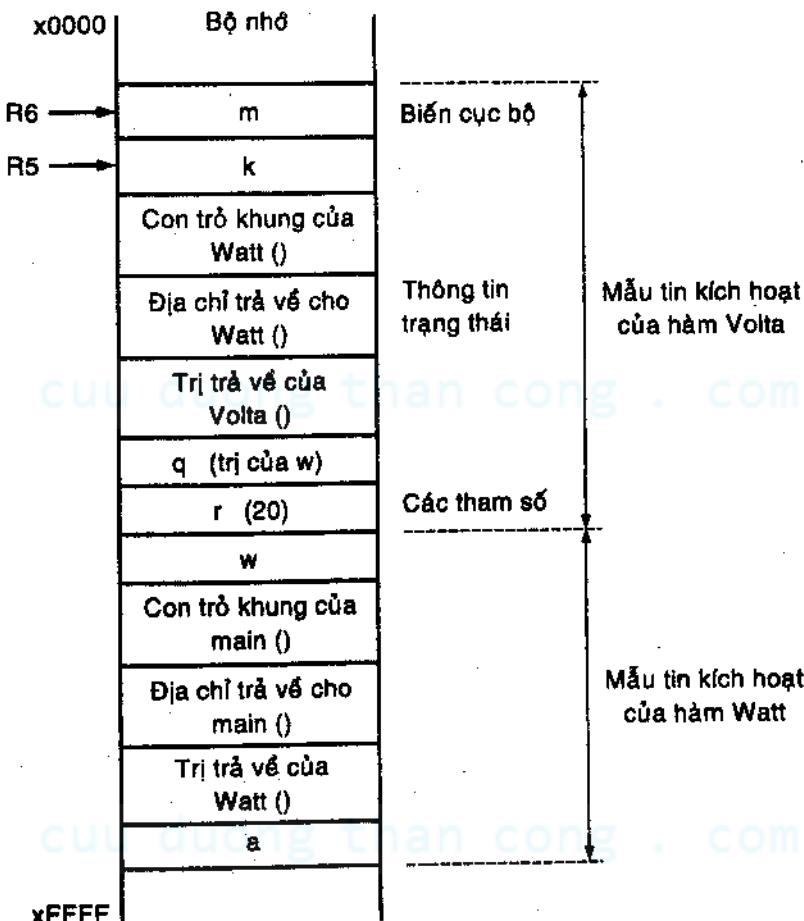
ADD  R6,  R6,  #-1 ; dành không gian cho trị trả về
ADD  R6,  R6,  #-1
STR  R7,  R6,  # 0 ; đẩy R7 (địa chỉ trả về) vô stack
ADD  R6,  R6,  #-1 ; đẩy liên kết động vô stack (con trả khung của hàm gọi)
STR  R5,  R6,  # 0 ; đẩy R5 vô stack
ADD  R5,  R6,  #-1 ; đặt con trả khung mới
ADD  R6,  R6,  #-2 ; định vị không gian cho các biến cục bộ

```

Hình 10.3 tóm tắt các thay đổi trong bộ nhớ được hoàn thành bởi đoạn code trên đây. Theo đó có hai mẫu tin kích hoạt trong bộ nhớ: một cho hàm Watt() và một hàm Volta(). Chú ý là có một số đầu vào trong mẫu tin kích hoạt của hàm Volta() được ghi trị bởi hàm Watt(). Đặc biệt, các đầu vào này là các vùng tin tham số của mẫu tin kích hoạt của hàm Volta(). Theo cách gọi, hàm Watt() đẩy trị 20 vào stack trước (đó nó nằm bên phải), sau đó tới trị của biến cục bộ w . Do đó, ta có thể thấy trị của w xuất hiện phía trên trị 20. Một khi được kích hoạt, hàm Volta() sẽ tham khảo các giá trị ở các vị trí này dưới

các tên q và r . Cũng nên nhắc lại là sau con trỏ khung của hàm Watt(), các biến cục bộ của hàm Watt(), k và m sẽ lần lượt được tạo ra, nên chúng ở đỉnh của mẫu tin kích hoạt.

Chú ý là các mẫu tin kích hoạt trong stack đều có cùng cấu trúc, trong đó có các vị trí cho các biến cục bộ của hàm, cho thông tin trạng thái trả về (gồm địa chỉ trả về của hàm gọi và liên kết động), trị trả về, và các tham số của hàm.



Hình 10.3 Stack thực thi của mẫu tin kích hoạt cho hàm Volta() được đẩy vào stack

- **Kết thúc hàm được gọi**

Một khi hàm được gọi hoàn thành công việc của nó, nó phải làm thêm một số tác vụ nữa trước khi trả điều khiển về cho hàm gọi. Đầu

tiên, ta cần có cơ chế để một hàm trả về trị một cách thích hợp cho nơi gọi. Thứ hai, hàm được gọi phải lấy ra khỏi stack mẫu tin kích hoạt của nó. Như vậy, ta có các việc như sau:

1. Nếu có một trị trả về, nó cần phải được ghi vô đầu vào trị trả về của mẫu tin kích hoạt.
2. Các biến cục bộ phải được lấy ra khỏi stack.
3. Liên kết động cần được phục hồi.
4. Địa chỉ trả về phải được phục hồi.
5. Lệnh RET trả điều khiển về cho hàm gọi.

Các lệnh LC-3 tương ứng với các thao tác trên cho hàm Volta() khi lệnh return k; được thi hành như sau:

```

LDR R0, R5, #0 ; nạp biến cục bộ k
STR R0, R5, #3 ; lưu nó vô đầu vào trị trả về
ADD R6, R5, #1 ; pop các biến cục bộ
                  ; (thực tế là gán trị để R6 chỉ tới vị trí con trỏ khung)
LDR R5, R6, #0 ; pop liên kết động, con trỏ khung -> R5
ADD R6, R6, #1 ; R6 chỉ tới ô địa chỉ trả về
LDR R7, R6, #0 ; lưu địa chỉ trả về cho R7
ADD R6, R6, #1 ; R6 chỉ tới ô trị trả về
RET              ; trả điều khiển về cho nơi gọi

```

Hai lệnh đầu tiên ghi trị trả về, trong ví dụ này là biến cục bộ k, vô đầu vào trị trả về của mẫu tin kích hoạt của hàm Volta(). Kế đó, các biến cục bộ bị lấy ra khỏi stack bằng việc dời con trỏ stack tới vị trí ngay dưới, con trỏ khung. Liên kết động được khôi phục, rồi tới địa chỉ trả về, và sau cùng chúng ta quay về nơi gọi.

Chúng ta cũng cần lưu ý là mặc dù mẫu tin kích hoạt của hàm (trong ví dụ này là hàm Volta()) đã được lấy ra khỏi stack, nhưng các giá trị vẫn còn giữ trong bộ nhớ, chỉ có vấn đề là chúng trở thành các trị rác, do ta không quản lý chúng nữa.

• Trở về hàm gọi

Sau khi hàm được gọi thực thi lệnh RET, điều khiển được trả ngược về cho hàm gọi. Trong một số trường hợp, không có trị trả về

(tức hàm được gọi được khai báo với kiểu *void*) và, trong một số trường hợp khác, hàm gọi bỏ qua các trị trả về (như khi ta gọi *getch();*). Trong ví dụ của chúng ta, trị trả về được gán vào biến *w* trong hàm *Watt()*.

Đặc biệt, có hai thao tác phải được thi hành:

1. Trị trả về (nếu có) được lấy ra khỏi stack.
2. Các đối số cần được lấy ra khỏi stack.

Đoạn code LC-3 sau lệnh JSR sẽ như sau:

```
JSR Volta
LDR R0, R6, #0 ; nạp trị trả về ở đỉnh stack
STR R0, R5, #0 ; w = Volta(w, 20);
ADD R6, R6, #1 ; pop trị trả về
ADD R6, R6, #2 ; pop các đối số
```

Một khi đoạn code trên được thực hiện, việc gọi hoàn thành và hàm gọi thực hiện tiếp tục các thao tác bình thường của nó. Chú ý là trước khi trở về nơi gọi, hàm được gọi sẽ khôi phục lại môi trường của nơi gọi. Với hàm gọi, có thể xem không có thay đổi gì xảy ra ngoại trừ việc có một trị mới (trị trả về) đã được đẩy vào stack.

• Sao lưu ở nơi gọi và sao lưu ở nơi được gọi

Trong khi thực thi một hàm, R0 tới R3 có thể chứa các giá trị tạm thời, là một phần của các thao tác tính toán đang chạy. Thanh ghi từ R4 tới R7 được dành cho các mục đích khác: thanh ghi R4 là con trỏ chỉ tới vùng dữ liệu toàn cục, R5 là con trỏ khung, R6 là con trỏ stack, và R7 được dùng để giữ địa chỉ trả về. Nếu chúng ta gọi một hàm, dựa theo quy ước gọi hàm chúng ta đã mô tả, thanh ghi R4 tới R7 không thay đổi hay thay đổi theo các cách đã được xác định trước. Nhưng cái gì xảy ra cho các thanh ghi R0, R1, R2, và R3? Tổng quát mà nói, chúng ta muốn chắc chắn rằng hàm được gọi sẽ không chép đè chúng. Để làm được điều này, các quy ước gọi hàm cụ thể theo một trong hai phương cách: (1) Nơi gọi sẽ sao lưu các thanh ghi bằng cách đẩy chúng vô mẫu tin kích hoạt của nó. Đây được gọi là sao lưu nơi gọi (*caller save*). Khi điều khiển được trả về cho nơi gọi,

nơi gọi sẽ khôi phục lại các thanh ghi này bằng việc lấy chúng ra khỏi stack. (2) Mặt khác, hàm được gọi có thể sao lưu các thanh ghi này bằng cách thêm vô bốn vùng tin trong vùng thông tin trạng thái của mẫu tin của nó. Đây chính là sao lưu nơi được gọi (*callee save*). Khi nơi được gọi được khởi tạo, nó sẽ sao lưu R0 tới R3 và R5 và R7 vô vùng thông tin trạng thái và khôi phục các thanh ghi này lại trước khi trở về nơi gọi.

10.7.3 Tóm lại

Code cho việc gọi hàm trong Watt() và đoạn đầu và cuối của hàm Volta() được trình bày trong đoạn mã dưới đây. Các đoạn code LC-3 được nêu trong các mục trước đây giờ tất cả được nối lại cho thấy cấu trúc chung của code. Đoạn code này được tối ưu hóa hơn các đoạn code riêng rẽ trước đây. Chúng ta đã đưa thao tác liên quan tới thanh ghi con trả stack R6, đẩy và lấy trị trả về vào các lệnh đơn.

Watt:

			; đẩy đối số thứ hai vào stack
AND	R0, R0, #0		; R0 ← 0
ADD	R0, R0, #20		; R0 ← 20
ADD	R6, R6, #-1		
STR	R0, R6, #0		; Push 20
			; đẩy đối số thứ nhất vào stack
LDR	R0, R5, #0		; Lấy trị của biến cục bộ w
ADD	R6, R6, #-1		
STR	R0, R6, #0		; Push trị này vào stack
			; gọi chương trình con
JSR	Volta		
LDR	R0, R6, #0		; Nạp trị trả về ở đỉnh stack
STR	R0, R5, #0		; w = Volta(w, 20);
ADD	R6, R6, #3		; lấy ra trị trả về và các đối số

Volta:

ADD	R6, R6, #-2	; dành không gian cho trị trả về
STR	R7, R6, #0	; đẩy R7 (địa chỉ trả về) vô stack

ADD	R6,	R6,	#-1	; đẩy liên kết động vô stack (con trỏ khung của hàm gọi)
STR	R5,	R6,	#0	; đẩy R5 vô stack
ADD	R5,	R6,	#-1	; đặt con trỏ khung mới
ADD	R6,	R6,	#-2	; định vị không gian cho các biến cục bộ
LDR	R0,	R5,	#0	; nạp biến cục bộ k
STR	R0,	R5,	#3	; lưu nó vô dấu vào trị trả về
ADD	R6,	R5,	#1	; pop các biến cục bộ
LDR	R5,	R6,	#0	; pop liên kết động, con trỏ khung -> R5
ADD	R6,	R6,	#1	; R6 chỉ tới ô địa chỉ trả về
LDR	R7,	R6,	#0	; lưu địa chỉ trả về cho R7
ADD	R6,	R6,	#1	; R6 chỉ tới ô trị trả về
RET				; trả điều khiển về cho nơi gọi

Tóm lại, thao tác một hàm gọi một hàm khác với LC-3 bao gồm một loạt các bước được thực hiện. Hàm gọi đẩy giá trị của mỗi tham số vô stack và thực hiện một cái Nhảy tới Thủ tục (*Jump to Subroutine-JSR*) tới hàm được gọi. Hàm được gọi định vị một không gian cho trị trả về, sao lưu các thông tin trạng thái về hàm gọi, và rồi định vị không gian trong stack cho các biến cục bộ của nó. Tiếp theo, hàm được gọi tiến hành thực thi nhiệm vụ của nó. Khi nhiệm vụ hoàn thành, hàm được gọi ghi trị trả về vô chỗ dành riêng cho nó, lấy ra và khôi phục lại thông tin trạng thái, và trở về nơi gọi. Sau đó, nơi gọi lấy ra trị trả về và các tham số nó đã đặt vô stack và khôi phục lại quá trình thực thi trước đây bị tạm dừng.

Có thể bạn thắc mắc là tại sao chúng ta phải hiểu tất cả các bước trên khi gọi một hàm. Nghĩa là, tất cả các code trên là thật sự cần thiết và không thể thực hiện gọi hàm một cách đơn giản hơn? Một trong các đặc điểm của việc gọi hàm trong thực tế là trong trường hợp tổng quát, bất cứ hàm nào cũng đều có thể gọi tới một hàm khác. Để điều này có thể, việc gọi hàm cần được tổ chức để nơi gọi không cần phải biết bất cứ gì về hàm được gọi ngoài giao diện của nó (tức là, kiểu của trị mà hàm được gọi trả về và kiểu của các trị là tham số hàm). Theo đó, hàm được gọi được thiết kế độc lập với hàm gọi nó. Vì tính tổng quát này, việc gọi các hàm trong C đòi hỏi các bước đã trình bày như trên.

10.8 KIỂM TRA VÀ SỬA LỖI

10.8.1 Giới thiệu

Vào tháng 12 năm 1999, cơ quan NASA đã mất liên lạc với thiết bị thăm dò Sao hỏa ngay sau khi nó tiếp xúc bề mặt Sao Hỏa. Thiết bị này có nhiệm vụ nghiên cứu vùng cực nam của Hành tinh Đỏ. Liên lạc đã không bao giờ được tái lập, và NASA đã tuyên bố tàu vũ trụ này đã va vào bề mặt hành tinh trong quá trình tiếp đất. Qua quá trình điều tra, các điều tra viên đã kết luận rằng nguyên nhân gần như chắc chắn là do phần mềm kiểm soát lỗi đã tắt các động cơ quá sớm khi thiết bị thăm dò cách bề mặt hành tinh 40 mét thay vì tắt khi đã tiếp đất xong. Sự phức tạp về phương diện vật lý của việc gởi các thiết bị thăm dò vào không gian thật đáng kinh ngạc, và những hệ phần mềm điều khiển các tàu vũ trụ này cũng phức tạp không kém. Phần mềm là một thành phần không thể tách rời trong hệ thống cũng như bất kỳ hệ thống cơ hay điện nào, hơn nữa để chỉnh sửa nó lại càng khó khăn vì nó “vô hình”. Chúng ta không thể quan sát thấy nó một cách dễ dàng như hệ thống dây hay hệ thống tiếp đất.

Ngày nay phần mềm có mặt ở khắp mọi nơi. Nó có trong điện thoại di động, trong xe hơi, ngay cả văn bản của cuốn sách này cũng được xử lý bởi một phần mềm xử lý văn bản trước khi nó được in ra và ở trước mặt các bạn. Phần mềm là một phần sống còn trong thế giới của chúng ta, việc thiết kế chương trình cho phần mềm không tự động được mà tùy thuộc các chi tiết cụ thể, tức không phải khi chương trình đã được viết xong thì nó chạy đúng. Chúng ta phải kiểm tra và sửa lỗi nó càng nhiều càng tốt trước khi chúng ta nghĩ nó đã hoàn chỉnh. Các lập trình viên thường tiêu tốn nhiều thời gian cho việc sửa lỗi chương trình hơn là cho việc viết ra chúng. Các chuyên gia đã quan sát và thấy rằng một lập trình viên có kinh nghiệm sử dụng thời gian để sửa lỗi mã chương trình cũng nhiều như thời gian để viết chương trình. Vì mối quan hệ không thể tách rời giữa viết mã, kiểm tra và sửa lỗi, trong chương này chúng tôi xin giới thiệu với các bạn một số khái niệm cơ bản về các vấn đề này, đặc biệt là sau khi học chương về hàm trong ngôn ngữ C.

Kiểm tra là một quá trình phát hiện lỗi, còn sửa lỗi là quá trình sửa mã chương trình. Kiểm tra một đoạn mã bao gồm việc đưa ra cho

nó càng nhiều càng tốt các dữ liệu nhập, đặc biệt là các điều kiện bờ, để phần mềm bộc lộ ra những lỗi có thể có. Ví dụ, giả sử chúng ta cần thiết kế hàm Toupper để nhận được ký tự chữ hoa của ký tự cung cấp trong đối số hàm, đối hàm chúng ta cung cấp có thể là bất kỳ ký tự nào trong bảng ASCII, tuy nhiên không phải ký tự nào đưa vào cho hàm chúng ta cũng nhận được ký tự hoa của nó, vì hàm này chỉ trả về ký tự hoa của những ký tự chữ thường mà thôi. Như vậy, việc xem xét đầu ra của hàm với những ký tự đầu vào đặc biệt sẽ giúp chúng ta phát hiện ra lỗi. Nếu hàm tạo ra một xuất liệu không thích hợp với dữ liệu nhập, thì chúng ta sẽ phát hiện ra lỗi. Mọi việc chắc hẳn sẽ tốt hơn cho các kĩ sư phần mềm NASA để tìm ra lỗi trong thiết bị hạ cánh Sao Hỏa trên bề mặt của Trái Đất hơn là gấp phải lỗi ở độ cao 40 mét trên bề mặt Sao Hỏa.

Dùng thông tin về một chương trình và sự thực thi của nó, một lập trình viên có thể áp dụng một cách thức chung để giảm cái gây ra lỗi. Việc sửa lỗi một chương trình thì hơi giống trò chơi giải đoán ô chữ một chút. Như là một thám tử truy tìm tội phạm, một lập trình viên phải khảo sát những khả năng có thể xảy ra để diễn tả thành chương trình nguồn của vấn đề. Việc sửa lỗi mã chương trình sẽ thực sự dễ dàng hơn nếu bạn biết cách thu thập thông tin về lỗi, chẳng hạn như giá trị các biến quan trọng trong quá trình thực thi chương trình, theo một cách thức có hệ thống.

Trong mục này, chúng tôi sẽ mô tả một số kỹ thuật để độc giả có thể sử dụng để tìm ra và sửa lỗi trong một chương trình, đặc biệt là chương trình được thiết kế với các hàm. Đầu tiên chúng tôi sẽ mô tả một số loại lỗi chung nhất có thể gặp phải trong chương trình, rồi chúng tôi sẽ mô tả các phương pháp kiểm tra để tìm ra các lỗi này một cách nhanh chóng. Sau cùng chúng tôi sẽ trình bày một vài kĩ thuật để cách ly và sửa các lỗi này, chúng tôi cũng cung cấp một vài kĩ thuật lập trình để hạn chế tối đa các lỗi trong mã mà bạn viết.

10.8.2 Các dạng lỗi

Để hiểu tốt hơn cách thức tìm ra và sửa lỗi trong chương trình, việc tìm hiểu các loại lỗi có thể gặp phải trong các chương trình mà chúng ta viết thì thật là hữu ích. Có ba loại lỗi lớn mà bạn thường gặp phải. Lỗi cú pháp (*Syntactic errors*) là những cái dễ gặp nhất bởi vì chúng được bộ dịch phát hiện ra. Bộ dịch lưu ý chúng ta các

lỗi như vậy khi nó thử dịch mã nguồn ra mã máy, thường các lỗi này được chỉ ra một cách chính xác theo dòng gấp phải. Lỗi ngữ cảnh (*Semantic errors*) lại khó sửa hơn, nó xảy ra khi cú pháp lệnh đã đúng nhưng lại chạy không theo ý chúng ta, như sự sơ suất khi gõ làm mất một số ký hiệu của ngôn ngữ làm ảnh hưởng tới cả sự đúng đắn của chương trình. Cả hai lỗi trên thường xảy ra trong quá trình gõ chương trình. Lỗi giải thuật (*Algorithmic errors*) phát sinh khi chúng ta phân tích và giải quyết vấn đề sai hay thiếu sót. Lỗi này thường rất khó để tìm ra và sửa chữa, nhiều khi chúng chỉ phát sinh sau nhiều năm sử dụng.

1- Lỗi cú pháp

Trong C, lỗi cú pháp (còn gọi là lỗi *syntax*) được bộ dịch tìm ra trong quá trình dịch khi có những mã không theo quy định của C. Xét ví dụ 10.19 dưới đây, in ra màn hình bình phương các số từ 0 - 9.

Ví dụ 10.20

```

1 #include <stdio.h>
2
3 main()
4 {
5     int i, i2;
6
7     for (i = 0; i < 10; i++)
8     {
9         i2 = i * i;
10        printf ("%d x %d = %d\n", i, i, i2);
11    }
12 }
```

Trong chương trình ví dụ này, lỗi cú pháp xảy ra ở hai chỗ: dòng 5 với khai báo biến không có dấu chấm phẩy kết thúc, và dòng 7 với lệnh for bị viết sai thành fo.

Nói chung, lỗi dạng này khá dễ dàng để tìm ra do bộ dịch đã xác định, chỉ ra lỗi sai và nơi bị lỗi. Lỗi này được sửa chữa khá dễ với các quy định của ngôn ngữ lập trình.

2- Lỗi ngữ cảnh

Lỗi này xuất hiện khi chúng ta không chú ý lúc gõ chương trình, nhiều khi chúng ta nghĩ phải gõ thế này, mà tay chúng ta lại gõ ra thứ khác. Xét chương trình ví dụ 10.20, chính là ví dụ 10.19 với các lỗi cú pháp đã được sửa.

Ví dụ 10.21

```

1   #include <stdio.h>
2
3   main()
4   {
5       int i, i2;
6
7       for (i = 0; i < 10; i++)
8           i2 = i * i;
9       printf ("%d x %d = %d\n", i, i, i2);
10 }
```

Trong chương trình trên, chúng ta muốn in ra bình phương các số từ 0-9, và độc giả hãy xem chương trình in ra cái gì? Chắc các bạn sẽ khá ngạc nhiên khi thấy nó chỉ in ra:

$$10 \times 10 = 81$$

Lỗi này xuất hiện do có sai sót trong dòng điều khiển chương trình. Thay vì lặp 10 lần với biến i lần lượt từ 0 tới 9, và với mỗi trị của i chương trình in ra bình phương tương ứng, thì do không có hai dấu “{”, “}” (do quên chăng?) để bao hai dòng 8 và 9 cho vòng for nên chương trình chỉ in ra sau khi kết thúc vòng for với i khi đó bằng 10!

Chúng ta hãy xét một ví dụ khác, để minh họa cho lỗi ngữ cảnh này.

Ví dụ 10.22 Chương trình tính tổng $1+2+\dots+n$, với n là trị nguyên dương được nhập từ bàn phím. Chương trình được thiết kế với hàm tính tổng.

```

#include <stdio.h>
int tinh_tong (int n);           // prototype của hàm
main()
{
    int so;
    int tong;
    printf ("Moi nhap mot so nguyen duong: ");
    scanf ("%d", &so);
    tong = tinh_tong (n);
    printf ("Tong tu 1 + ... + %d = %d\n", n, tong);
}
int tinh_tong (int n)
{
    int tong;
    int i;
    for (i = 1; i <= n; i++)
        tong += i;
    return tong;
}

```

Độc giả hãy thử giải thích tại sao chương trình in ra sai, dù trị n nhập vào là thỏa nguyên dương.

3- Lỗi giải thuật

Lỗi này xuất phát từ việc phân tích và giải quyết vấn đề ban đầu không đúng, tức là chương trình chạy đúng theo những gì ta đã thiết kế, nhưng việc thiết kế bản thân nó lại có sai sót. Lỗi này thường rất khó được phát hiện nếu chúng ta không thử chương trình ở tất cả những trị mà chương trình được quy định phải thực thi. Muốn sửa những lỗi dạng này, thường chúng ta thực hiện việc cách ly từng biến với từng trị có thể có. Sau khi chúng chạy đúng, chúng ta mới gắn chúng vào chương trình lớn. Việc thiết kế từng giai đoạn, với việc hoạch định hợp lý sự thay đổi trị trước khi viết code có thể hạn chế được lỗi dạng này.

Trong thực tế, chúng ta rất thường gặp lỗi dạng này. Hãy xét chương trình giải phương trình bậc nhất sau.

Ví dụ 10.23

```

1   #include <stdio.h>
2   #include <conio.h>
3
4   void gptb1 (double a, double b);
5
6   main()
7   {
8       double a, b;
9
10      clrscr();
11      printf ("Nhập 2 hệ số phương trình bậc nhất: ");
12      scanf ("%lf %lf", &a, &b);
13      gptb1 (a, b);
14      getch();
15
16  void gptb1 (double a, double b)
17  {
18      printf ("Phương trình bậc nhất ");
19      printf ("có 1 nghiệm: x = %.2f \n", -b/a);
20  }

```

Nếu nhìn không kỹ, có thể chúng ta sẽ nói chương trình không sai. Tuy nhiên chương trình lại chỉ chạy đúng khi hệ số a khác 0, còn khi a bằng 0 thì sao? Tất nhiên, chương trình sẽ có lỗi do chia cho số bằng 0! Khi đó, chúng ta cần sửa lại hàm từ dòng 15, trong đó cần kiểm tra trị của biến a rồi quyết định nghiệm. Độc giả có thể xem lại chương trình 10.1.

10.8.3 Kiểm tra chương trình

Kỹ thuật kiểm tra chương trình tốt đóng vai trò quan trọng trong việc viết được phần mềm tốt. Vậy kiểm tra chương trình là gì? Với việc kiểm tra chương trình, chúng ta cần phải đưa cho phần mềm những mẫu thử đầu vào như là dữ liệu thật ở tất cả các trạng thái, hoàn cảnh có thể xảy ra, và sau đó chúng ta kiểm tra tính đúng đắn từ dữ liệu xuất của chương trình. Trong thực tế, các phần mềm

thường phải trải qua hàng triệu lần thử trước khi chúng được phát hành chính thức.

Một cách lý tưởng, việc kiểm tra chương trình nên được thực hiện với mọi dữ liệu nhập. Tuy nhiên, trong thực tế, nhiều khi chúng ta không thể làm được điều này do chúng ta không thể thử dữ liệu nhập với số lượng khổng lồ được. Ví dụ, với chương trình tìm tất cả các số nguyên tố giữa hai số nguyên 32 bit A và B, có $(2^{32})^2$ tổ hợp số, nếu chúng ta chuẩn bị dữ liệu nhập bằng các tập tin chứa tất cả các tổ hợp trị A và B, và việc kiểm tra có thể được thực hiện với tốc độ rất nhanh là 1 triệu phép thử trong một giây, thì phải mất gần nửa triệu năm (!) để hoàn thành việc kiểm tra. Tức, việc kiểm tra chương trình với mỗi tổ hợp nhập không phải là một lựa chọn hay. Như vậy, chúng ta cần dùng tổ hợp nhập nào để kiểm tra? Chúng ta có thể cho ngẫu nhiên các trị nhập, và hy vọng rằng các mẫu ngẫu nhiên đó gây ra lỗi cho chương trình để chúng ta có thể sửa. Các kỹ sư phần mềm có các phương pháp hệ thống hơn để kiểm tra code của họ. Kỹ thuật kiểm tra dùng hộp đen dùng để xem chương trình có tuân thủ các quy định hay không, còn kiểm tra bằng hộp trắng nhắm tới các khía cạnh hiện thực của chương trình để bảo đảm mỗi hàng code đều đúng.

1- Kiểm tra hộp đen

Với kỹ thuật hộp đen này, chúng ta có thể kiểm tra xem dữ liệu nhập và xuất của chương trình có phù hợp không, mà không cần phải biết cấu trúc lệnh trong chương trình. Nghĩa là, với kỹ thuật kiểm tra này, chúng ta chỉ quan tâm tới việc chương trình làm gì, chứ không cần quan tâm tới việc nó làm việc đó như thế nào. Ví dụ, chương trình tính tổng trong ví dụ 10.21, hộp đen để kiểm tra chương trình gồm các thao tác: chạy chương trình, nhập trị đầu vào, và so sánh kết quả xuất với cái mà ta tính bằng tay. Nếu hai cái không phù hợp, thì hoặc là chương trình có lỗi hay kỹ năng số học của ta có vấn đề. Chúng ta có thể tiếp tục thử và sửa nhiều lần cho tới khi chúng ta thấy chương trình thực hiện đúng chức năng của nó.

Với những chương trình lớn hơn, quá trình thử được tự động hóa để chạy nhiều lần kiểm tra trong một đơn vị thời gian. Nghĩa là, chúng ta tạo ra một chương trình khác chạy tự động chương trình gốc cần kiểm tra, cung cấp một vài trị nhập ngẫu nhiên, kiểm tra tính đúng đắn của trị xuất, và lặp lại quá trình này. Rõ ràng là với một

quá trình như vậy, chúng ta có thể chạy được nhiều lần thử hơn là nếu chúng ta thực hiện trực tiếp bằng tay.

Cụ thể, để tự động hóa quá trình kiểm tra hộp đen, chúng ta cần một cách thức kiểm tra tự động xem xuất liệu của chương trình có đúng hay không. Lúc này, chúng ta cần xây dựng một chương trình kiểm tra khác chương trình gốc, nhưng lại có chức năng tính toán tương tự như chương trình gốc. Các lập trình viên thiết kế các chương trình kiểm tra thường không được phép biết code của hộp đen là chương trình gốc mà họ đang kiểm tra để chúng ta có được một bộ kiểm tra độc lập thực sự.

2- Kiểm tra hộp trắng

Với những hệ thống phần mềm lớn hơn, kỹ thuật kiểm tra hộp đen còn không đủ, nó không thể biết được dòng code nào đã được kiểm tra rồi, dòng nào chưa. Hơn nữa, kiểm tra hộp đen đôi lúc gặp khó khăn khi đặc tính dữ liệu nhập hay xuất của chương trình không cụ thể. Ví dụ, kiểm tra hộp đen của các phần mềm đa phương tiện (như nghe MP3 chẳng hạn) sẽ khó khăn do bản chất không chính xác của dữ liệu xuất là âm thanh. Kiểm tra hộp đen cũng chỉ có thể bắt đầu sau khi phần mềm đã hoàn thành, phải được dịch và thỏa một phần đặc trưng để kiểm tra.

Ngoài kiểm tra hộp đen, các kỹ sư phần mềm còn bổ sung thêm các phép kiểm tra hộp trắng. Các phép kiểm tra này cách ly các thành phần bên trong khác nhau của phần mềm, và kiểm tra xem các thành phần này có theo đúng yêu cầu thiết kế không. Ví dụ, việc kiểm tra mỗi hàm xem nó có thực hiện đúng theo thiết kế không là một phép kiểm tra hộp trắng. Chúng ta có thể áp dụng loại kiểm tra giống như vậy cho các vòng lặp và các cấu trúc khác trong hàm.

Một phép kiểm tra hộp trắng có cấu trúc ra sao? Chúng ta nhiều lúc cần phải hiệu chỉnh code. Ví dụ, để xem một hàm có làm việc đúng hay không, chúng ta có thể thêm vào các dòng mã phụ để gọi hàm thêm một vài lần với các dữ liệu nhập khác nhau và kiểm tra dữ liệu xuất. Chúng ta có thể thêm vào các mệnh đề với hàm printf để in ra các trị mà chúng ta muốn quan sát trong các biến để xem mọi thứ có chạy đúng không. Một khi code đã chạy đúng, chúng ta sẽ xóa các lệnh printf này.

Kỹ thuật kiểm tra hộp trắng tổng quát chính là sử dụng các code tìm lỗi được đặt trong chương trình một cách có tính toán. Các code này có thể kiểm tra các điều kiện để chỉ ra lỗi sai làm chương trình không thực thi đúng. Khi có một lỗi sai được tìm ra, code sẽ in ra một thông điệp cảnh báo, hiển thị một vài thông tin có giá trị về lỗi, hoặc là làm cho chương trình kết thúc sớm. Vì các code tìm lỗi này xác nhận các điều kiện cụ thể cần giữ trong suốt quá trình chương trình thực thi, nên chúng ta gọi các code này là **sự xác nhận**.

Ví dụ, các xác nhận có thể được dùng để kiểm tra xem một hàm có trả về trị trong tầm mong muốn hay không. Nếu giá trị trả về là ngoài tầm, một thông điệp báo lỗi được hiển thị. Trong ví dụ 10.24 sau, chúng ta muốn kiểm tra xem thao tác tính toán trong hàm ThueThunhap() có nằm trong khoảng hợp lý không. Ở đây chúng ta không quan tâm cụ thể tới các thao tác tính thuế, chúng ta chỉ cần biết là nếu thuế tính ra từ hàm này mà là số âm hoặc lớn hơn thu nhập của chúng ta, thì hàm này đã sai, khi đó một thông điệp cần in ra xác nhận lỗi này.

Ví dụ 10.24

```
thue = ThueThunhap (thunhap);
If (thue < 0 || thue > thunhap)
    Printf("Có lỗi trong hàm ThueThunhap!\n");
```

Tóm lại, một kỹ thuật kiểm tra tổng thể đòi hỏi cả kiểm tra hộp đen và kiểm tra hộp trắng. Một điều quan trọng cần phải biết là kiểm tra hộp trắng một mình nó không bao hàm được hết chức năng của phần mềm, tức là ngay cả nếu tất cả thao tác kiểm tra hộp trắng đều đúng, thì cũng có thể có một phần đặc trưng nào đó bị bỏ sót. Tương tự, các kiểm tra hộp đen bản thân chúng không bảo đảm mọi dòng code đều được kiểm tra.

10.8.4 Gỡ rối chương trình (debuging)

Một khi một lỗi nào đó được tìm thấy, chúng ta bắt đầu quá trình sửa sai, quá trình này có thể cần nhiều mèo nhỏ. Gỡ rối một lỗi đòi hỏi việc sử dụng đầy đủ các kỹ năng hợp lý: quan sát triệu chứng lỗi, như xuất liệu sai; nhận xét, đánh giá các thông tin liên quan, như vị trí lỗi trong code; suy luận để cách ly nguồn lỗi. Chìa khóa để gỡ

rồi hiệu quả là khả năng tập hợp các thông tin có giá trị, dẫn tới việc xác định chính xác lỗi, tương tự như cách một thám tử tập hợp bằng chứng phạm tội hay cách một bác sĩ thực hiện một loạt kiểm tra để chuẩn đoán bệnh lý của bệnh nhân.

Có một số cách thu thập thông tin để chẩn đoán lỗi, từ các kỹ thuật phi thể thức nhưng nhanh và cần một chút ... thủ đoạn cho tới các kỹ thuật có tính hệ thống hơn bao gồm việc sử dụng các công cụ phần mềm gõ rồi chuyên nghiệp.

1. Kỹ thuật phi thể thức

Cách đơn giản nhất mà chúng ta thường làm một khi chương trình của chúng ta có vấn đề là xem xét mã nguồn (*source code*). Đôi khi bản chất của lỗi sai phát sinh từ việc bạn bỏ một đoạn code trong vùng mà bạn nghĩ là có lỗi.

Một kỹ thuật đơn giản khác là chèn vào code một số mệnh đề để in ra các thông tin cần thiết trong quá trình thực thi chương trình. Với C, ta có thể dùng các mệnh đề với hàm printf để in ra trị của một số biến quan trọng mà ta nghĩ là có tác dụng trong việc tìm ra lỗi. Ta cũng có thể thêm mệnh đề với hàm printf ở những điểm khác nhau trong code để xem dòng điều khiển của chương trình có làm việc đúng không. Ví dụ, nếu bạn muốn nhanh chóng xác định xem một vòng lặp với biến đếm điều khiển đang lặp ở lần lặp thứ mấy, bạn có thể đặt mệnh đề printf trong thân vòng lặp. Với những chương trình đơn giản, những kỹ thuật phi thể thức như vậy là dễ dàng và hợp lý khi được đặt đúng chỗ. Các chương trình lớn với các lỗi khó hiểu đòi hỏi việc sử dụng các kỹ thuật mạnh hơn nữa.

2- Gõ rối cấp nguồn

Các kỹ thuật phi thể thức thường không thể cung cấp đủ thông tin để cho biết nguồn lỗi. Trong những trường hợp này, lập trình viên thường chuyển qua dùng bộ gõ rối cấp nguồn (*source-level debugger*) để cách ly lỗi. Bộ gõ rối cấp nguồn là một công cụ cho phép một chương trình thực thi trong một môi trường có điều khiển, ở đó tất cả các khía cạnh thực thi của chương trình đều có thể được lập trình viên điều khiển và giám sát. Ví dụ, một bộ gõ rối có thể cho phép chúng ta thực thi chương trình theo kiểu mỗi lần một lệnh và khảo

sát trị của các biến (cũng như của các ô nhớ và thanh ghi mà chúng ta đã chọn) theo quá trình thực thi đó.

Để một bộ gỡ rối cấp nguồn chạy một chương trình, chương trình phải được dịch sao cho bộ dịch có thể làm tăng lên ảnh thực thi với các thông tin thêm vào đây đủ để bộ gỡ rối làm đúng chức năng của mình. Mặt khác, bộ gỡ rối cũng cần thông từ quá trình dịch để lấy đối ứng từng lệnh ở cấp ngôn ngữ máy với mệnh đề tương ứng trong chương trình nguồn cấp cao. Bộ gỡ rối cũng cần thông tin về tên các biến và vị trí của chúng trong bộ nhớ (tức bằng các biểu trưng-symbol). Điều này cần thiết để người lập trình có thể khảo sát trị của mỗi biến trong chương trình với tên tương ứng trong mã nguồn.

Hầu hết các bộ gỡ rối đều có tập các thao tác cơ bản giống nhau và rơi vào một trong hai loại sau: các lệnh điều khiển chương trình và các lệnh cho phép lập trình viên khảo sát trị của biến và ô nhớ, ... trong quá trình thực thi. Ta hãy xét một trong số các thao tác này.

1- Điểm ngắt (Breakpoints)

Các điểm ngắt cho phép chúng ta quy định các vị trí trong quá trình thực thi chương trình mà ở đó chương trình có thể tạm thời dừng lại để chúng ta kiểm tra hay hiệu chỉnh trạng thái hiện có của chương trình. Việc này rất hữu ích vì nó giúp chúng ta kiểm tra việc thực thi của chương trình trong vùng code có lỗi xuất hiện.

Ví dụ, chúng ta có thể quy định điểm ngắt là một dòng lệnh đặc biệt nào đó trong mã nguồn hay là một hàm nào đặc biệt nào đó. Khi chương trình thực thi tới dòng được quy định là điểm ngắt, việc thực thi sẽ tạm dừng lại, và chúng ta có thể khảo sát mọi thứ của chương trình tại thời điểm đó để tìm ra lỗi nếu có. Làm thế nào quy định một điểm ngắt thì lại tùy thuộc vào giao diện với người dùng của bộ gỡ rối. Một số bộ debugger cho phép quy định các vị trí ngắt bằng việc click vào dòng mã bằng tổ hợp phím quy ước. Một số khác lại yêu cầu phải cung cấp số hiệu của dòng mà ta muốn quy định điểm ngắt.

Đôi khi dừng điều khiển chương trình ở một dòng code khi một điều kiện cụ thể đúng nào đó là rất hữu dụng. Những điểm ngắt theo điều kiện như vậy có lợi cho việc cách ly các trạng thái mà chúng ta

nghi ngờ là gây ra lỗi. Ví dụ, nếu chúng ta nghi ngờ hàm TinhTong làm việc không đúng khi tham số nhập vào là 12, thì chúng ta có lẽ cần có một điểm ngắt để dừng quá trình thực thi chỉ khi đổi số x là 12 trong đoạn code sau:

```
for (x = 1; x < 100; x++)
```

```
TinhTong (x);
```

Mặt khác, chúng ta có thể đặt một điểm quan sát (*watchpoint*) để dừng chương trình ở bất kỳ điểm nào mà ở đó có một điều kiện xác định là đúng. Ví dụ, chúng ta có thể dùng một điểm quan sát để dừng sự thực thi bất kỳ khi nào biến Ketthuc bằng 6. Việc này sẽ làm cho bộ gỡ rối tạm dừng quá trình thực thi ở mệnh đề làm cho biến Ketthuc bằng 6. Không như các điểm ngắt, điểm quan sát không gắn liền với một dòng lệnh đơn nào, nó được áp dụng cho mọi lệnh.

2- Thực thi từng bước (Single-Stepping)

Một khi bộ gỡ rối chạy tới điểm ngắt hay điểm quan sát, nó tạm thời treo quá trình thực thi chương trình lại và chờ lệnh kế tiếp từ chúng ta. Lúc này chúng ta có thể khảo sát trạng thái chương trình, như trị của các biến, hay chạy tiếp chương trình.

Nếu từ điểm ngắt mà mỗi lần ta thực thi một lệnh thì ta có quá trình thực thi từng bước. Bộ gỡ rối LC-3 có một lệnh cho phép thực thi từng lệnh đơn LC-3, nó tương tự bộ gỡ rối cấp nguồn, tức cho phép tiến hành mỗi lần một lệnh. Lệnh thực thi từng bước thực thi dòng mã nguồn hiện thời và rồi tạm thời treo chương trình lại. Hầu hết các bộ gỡ rối đều hiển thị mã nguồn trong một cửa sổ riêng nên chúng ta có thể quan sát nơi mà chương trình đang bị dừng lại. Việc thực hiện từng bước qua chương trình rất có ích, đặc biệt là thực thi trong vùng nghi ngờ có lỗi. Chúng ta có thể đặt một điểm ngắt gần vùng bị nghi ngờ và sau đó kiểm tra các giá trị của biến qua quá trình thực thi từng bước để tìm ra lỗi.

Cách sử dụng thực thi từng bước tổng quát là kiểm tra dòng điều khiển chương trình thực hiện cái mà chúng ta mong muốn. Chúng ta có thể chạy từng bước qua vòng lặp để kiểm tra xem nó có thực hiện đúng số lần lặp không, hay chạy từng bước qua một mệnh đề if-else để kiểm tra xem chúng ta đã lập trình đúng điều kiện hay không.

Quá trình thực thi từng bước có thể được thay đổi và cho phép chúng ta bỏ qua thực thi từng bước vào hàm nếu muốn, hay chạy tới chu trình lặp cuối của vòng lặp. Các thay đổi này rất có ích khi chúng ta muốn bỏ qua các code mà chúng ta chắc là chúng không sinh lỗi.

3- Hiển thị giá trị

Nghệ thuật gỡ rối liên quan trực tiếp tới việc thu thập thông tin yêu cầu để suy luận một cách hợp lý ra nguồn lỗi. Bộ gỡ rối là công cụ chọn lựa cho việc thu thập thông tin khi gỡ rối các chương trình lớn. Trong khi quá trình thực thi bị treo lại ở một điểm ngắt, chúng ta có thể tập hợp thông tin về lỗi bằng việc khảo sát các giá trị của các biến liên quan tới lỗi bị nghi ngờ. Nói một cách tổng quát, chúng ta có thể kiểm tra tất cả các trạng thái thực thi của chương trình tại điểm ngắt. Chúng ta có thể khảo sát trị của biến, bộ nhớ, ngăn sếp, và thậm chí thanh ghi. Làm cách nào việc này được thực hiện là đặc trưng của từng bộ gỡ rối. Một số bộ debugger cho phép chúng ta dùng chuột chỉ tới biến trong cửa sổ mã nguồn, rồi mở ra một cửa sổ pop-up để hiển thị trị hiện thời của biến. Một bộ debugger khác lại yêu cầu ta gõ trong dòng lệnh tên biến ta muốn kiểm tra.

Các bạn cần sử dụng các bộ gỡ rối cho quen thuộc vì sự ích lợi của chúng. Dĩ nhiên là tùy vào ngôn ngữ mà chúng ta có các bộ gỡ rối khác nhau, như bộ gỡ rối LC-3, bộ Turbo Debugger cho hợp ngữ 80x86, bộ gỡ rối cho C/C++ trên Borland C ver 5.0x,...

10.8.5 Tính đúng đắn khi lập trình

Biết cách để kiểm tra trị và gỡ rối code là một đòi hỏi trước tiên cho một lập trình viên giỏi. Một lập trình viên giỏi biết làm thế nào để tránh các tình huống gây ra lỗi, anh ta phải biết chọn lựa code để giảm thời gian thực thi và gây ra lỗi có thể có. Việc hạn chế lỗi cần được bắt đầu trước khi ta thực hiện viết code. Có ba phương pháp chung để bắt lỗi ngay cả trước khi chúng trở thành lỗi.

1- Xác định thật rõ các chi tiết

Có nhiều lỗi xuất phát từ các chi tiết trong chương trình không rõ ràng hay không hoàn chỉnh. Các chi tiết đôi khi không bao hàm được trong tất cả các ngữ cảnh có thể xảy ra, và như vậy chúng để lại một vài điều kiện mở cho lập trình viên diễn dịch. Ví dụ, chương trình tính

giai thừa của một số nhập vào từ bàn phím. Bạn có thể nghĩ chi tiết cho chương trình trên khi đề bài như sau: "Viết một chương trình nhập một số nguyên từ bàn phím và tính giai thừa của nó". Với đề bài đã cho, ta thấy chi tiết xác định chưa hoàn chỉnh. Nếu người dùng nhập một trị âm thì sao? Còn nếu nhập zero? Hoặc người dùng nhập một số quá lớn gây tràn bộ nhớ thì sao? Trong những trường hợp này, nếu không cẩn thận mã nguồn có thể bị sai. Do đó, để tránh các lỗi sai, chúng ta cần xác định rõ chi tiết của chương trình, như nếu dữ liệu nhập n trong chương trình tính giai thừa trên mà nhỏ hơn hay bằng 0, hay nếu nó tạo ra $n! > 2^{31}$ (số nguyên 4 byte có dấu), tức n khoảng lớn hơn 31 thì chương trình sẽ xử lý ra sao. Hàm tính giai thừa sau đây đưa ra một đề nghị cho cách giải quyết này.

Ví dụ 10.25

```
int Giaithua (int n)
{
    int i;
    int ketqua = 1;
    // Kiểm tra trị nhập
    if (n < 1 || n > 31)
    {
        printf ("Trị sai, cần trị trong khoảng 1-31.\n");
        ketqua = -1;
    }
    else
        for (i = 1; i <= n; i++)
            ketqua *= i;
    return ketqua;
}
```

Từ hàm này ta thấy khi hàm trả về trị bằng -1, và trước đó có câu báo sai, tức trị nhập vào không thỏa chi tiết về dữ liệu. Còn khi hàm trả về trị khác, tức đó chính là giai thừa cần tìm.

2- Thiết kế theo từng khối

Các hàm rất có lợi cho việc mở rộng chức năng của ngôn ngữ lập trình. Với hàm, chúng ta có thể thêm vào các thao tác và cấu trúc

mới rất hữu dụng cho một tác vụ lập trình đặc biệt. Theo đó, thiết kế với hàm giúp chúng ta có thể viết chương trình theo các modul.

Khi một hàm là hoàn chỉnh, chúng ta có thể kiểm tra nó một cách độc lập theo kiểu cách ly, như phép kiểm tra hộp trắng, và xác định xem nó có làm việc như chúng ta mong đợi không. Vì một hàm thường đảm nhận một tác vụ cụ thể chứ không phải một chương trình hoàn chỉnh, nên kiểm tra nó dễ hơn là kiểm tra toàn bộ chương trình. Một khi chúng ta đã kiểm tra và tìm lỗi mỗi hàm theo kiểu cách ly, chúng ta sẽ dễ dàng bắt chương trình làm việc khi mọi thứ được tích hợp lại.

Khái niệm thiết kế theo modul để xây dựng chương trình bằng các thành phần đơn giản, đã được kiểm tra trước là một khái niệm cơ bản trong thiết kế hệ thống. Trong các ngôn ngữ lập trình có khái niệm thư viện. Một thư viện là một tập hợp các thành phần đã được kiểm tra mà mọi lập trình viên đều có thể dùng chúng trong mã nguồn của mình. Ngày nay, lập trình hiện đại luôn gắn liền với việc sử dụng thư viện vì các ích lợi vốn có của thiết kế theo modul. Chúng ta thiết kế không chỉ phần mềm, mà còn cả mạch, phần cứng, và nhiều thứ khác trong hệ thống máy tính cũng dùng triết lý thiết kế modul tương tự như vậy.

3- Lập trình dự phòng

Tất cả lập trình viên dày dạn kinh nghiệm đều có các kỹ thuật để tránh lỗi lọt vào code của họ. Họ xây dựng code theo cách sao cho các lỗi mà họ ngờ ngờ là ảnh hưởng tới chương trình sẽ bị loại trừ ngay từ khâu thiết kế. Nghĩa là, họ lập trình một cách dự phòng. Dưới đây chúng ta có một danh sách ngắn các kỹ thuật lập trình dự phòng tổng quát mà chúng ta có thể làm theo để tránh các vấn đề với chương trình mà ta viết.

- Ghi chú code. Việc viết các ghi chú làm cho bạn hiểu rõ hơn code mà bạn viết. Nó không chỉ là cách để báo cho người khác biết về cách mà code của bạn làm việc, mà nó còn là một quá trình bắt bạn ngẫm nghĩ và xem xét lại code của bạn. Trong suốt quá trình này, bạn có thể khám phá ra rằng bạn đã quên một trường hợp quan trọng hay một điều kiện đặc biệt nào đó mà nó sẽ phá vỡ code của bạn nếu gặp phải.

- Giữ một phong cách viết code không đổi. Ví dụ, canh ngay cột hai dấu mở và đóng mốc nhọn trong C/C++ sẽ làm bạn không bị mắc các lỗi ngữ cảnh đơn giản do thiếu dấu mốc. Hay với tên biến, tên của biến nên mang một thông tin có nghĩa, phản ánh ý nghĩa của giá trị mà nó chứa.
- Tránh các chấp nhận. Khi lập trình các chấp nhận đơn giản hay vô hại dễ làm cho ta thỏa mãn, nhưng chính những cái này lại dễ dẫn tới lỗi làm hư chương trình. Ví dụ, khi viết hàm, chúng ta nhiều khi dễ dàng chấp nhận tham số nhập luôn luôn nằm trong khoảng xác định. Nếu gặp trường hợp trị đưa vào hàm nằm ngoài tầm ta chấp nhận theo chi tiết chương trình, khi đó lỗi sẽ có khả năng phát sinh. Do đó, bạn cần nhớ, viết code tức là ta không được có những chấp nhận trước, như khi viết hàm, ta luôn phải kiểm tra trị cho các tham số nhập, dù có thể việc này là thừa do tham số nhập đã được kiểm tra trước khi gọi hàm.
- Tránh các biến toàn cục. Trong khi một số lập trình viên kinh nghiệm tin tưởng chắc chắn vào biến toàn cục, thì nhiều kỹ sư phần mềm lại chủ trương tránh dùng chúng bất cứ khi nào có thể. Biến toàn cục có thể làm cho một số tác vụ lập trình dễ dàng hơn. Tuy nhiên, chúng thường làm cho code khó hiểu hơn do việc dùng biến khai báo ngoài trong hàm, và khi đó, nếu có lỗi phát sinh, sửa chữa sẽ khó khăn hơn.
- Tin tưởng vào bộ dịch. Hầu hết các bộ dịch tốt đều có một lựa chọn để kiểm tra cẩn thận chương trình của bạn đối với các code nghi ngờ (ví dụ như một biến chưa được khởi động trị) hay các cấu trúc code không phù hợp (ví dụ, thay vì so sánh bằng dùng hai dấu == trong lệnh if, ta lại chỉ viết có một dấu =, tức phép gán trong C). Các kiểm tra này là chưa hoàn hảo, nhưng chúng thật sự giúp chỉ ra một số lỗi lập trình chung ở dạng cảnh báo (*warning*), các cảnh báo này là đúng hay sai tùy theo kinh nghiệm và kiến thức của người lập trình.

Kỹ thuật lập trình dự phòng lỗi đề cập ở đây rất có lợi cho lập trình viên. Chúng chỉ là một số kỹ thuật tiêu biểu. Ngoài ra, lập trình viên có thể rút ra cho mình những kinh nghiệm khác để việc viết chương trình và sửa lỗi nếu có được tiện lợi và dễ dàng.

BÀI TẬP CUỐI CHƯƠNG

- 10.1** Thiết hàm giải phương trình $ax^2 + bx + c = 0$ tổng quát. Viết chương trình chính sử dụng hàm này.
- 10.2** Thiết hàm in ra màn hình chuỗi số Fibonacci, với thông số nhập là một số ngẫu nhiên trong số từ 1 đến 100 biểu diễn số lượng số trong chuỗi.

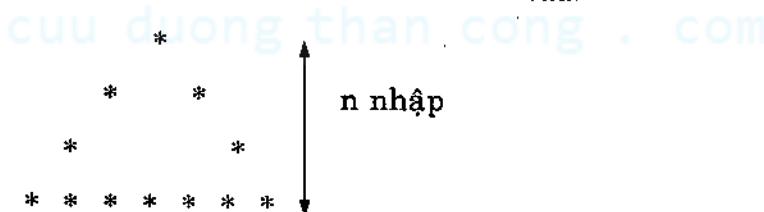
Hướng dẫn: Sử dụng hàm random() và randomize() để tạo số ngẫu nhiên.

- 10.3** Thiết kế hàm tính các biểu thức sau đây:

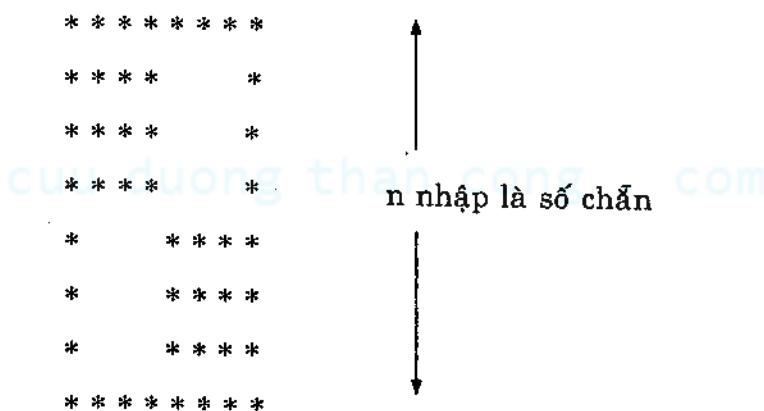
$$T = \frac{1}{1!} + \frac{1+2}{2!} + \dots + \frac{1+2+\dots+n}{n!}$$

$$S = (1)! + (1+2)! + \dots + (1+\dots+n)!$$

- 10.4** Thiết kế hàm vẽ ra màn hình hình sau:



- 10.5** Thiết kế hàm và vẽ ra màn hình hình sau:



- 10.6** Viết chương trình thiết kế hai hàm max() và min() cho phép tìm số lớn nhất và nhỏ nhất trong loạt số đã nhập. Loạt số nhập này sẽ kết thúc việc nhập bằng việc nhấn phím <F6>

- 10.7 Viết một chương trình cho phép nhập một chuỗi. Hãy thiết kế một hàm cho phép cắt chuỗi đó ra làm nhiều từ và in ra màn hình mỗi từ trên một hàng.
- 10.8 Viết một hàm nhận một chuỗi và cho phép đảo chuỗi đó, in ra kết quả.
- 10.9 Viết một hàm cho phép nhận một số nguyên dương. In ra màn hình ký số thứ n tính từ bên phải qua của số đó.

Ví dụ :

Nhập : 12345 4

Xuất : 2

- 10.10 Thiết kế một hàm đệ quy cho phép nhận một số nguyên dương, in ra màn hình số đó ở dạng nhị phân.
- 10.11 Viết hàm đệ quy tính x^n .
- 10.12 Viết một hàm nhận một số dương có phần lẻ và in ra màn hình phần nguyên và phần lẻ riêng biệt.
- 10.13 Viết chương trình với hàm tính tổng sau:

$$S = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots - \frac{x^n}{n!}$$

với x (thực) và n (nguyên dương) nhập từ bàn phím.

- 10.14 Tương tự như bài 10.13, nhưng tính tổng sau:

$$S = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots$$

với tham số nhập vào là x và sai số tuyệt đối ε.

LỚP LƯU TRỮ CỦA BIẾN SỰ CHUYỂN KIỂU

11.1 KHÁI NIỆM

Mỗi biến khi được sử dụng trong chương trình đều phải được khai báo, tuy nhiên biến có thể được khai báo ở nhiều chỗ trong chương trình, biến có thể được khai báo trong hàm, ngoài hàm..., mỗi chỗ như vậy sẽ làm cho biến có khả năng sử dụng khác nhau, từ đó hình thành nên các lớp lưu trữ biến. Đối với C, dựa vào cách mà biến được lưu trữ và sử dụng, biến sẽ ở một trong các lớp lưu trữ khác nhau sau đây:

- Lớp biến tự động
- Lớp biến toàn cục và biến cục bộ
- Lớp biến tĩnh
- Lớp biến thanh ghi

Có hai đặc tính quan trọng của một biến: tầm sử dụng của biến và thời gian tồn tại của biến. Chính tầm sử dụng và thời gian tồn tại khác nhau của các biến hình thành nên các lớp lưu trữ biến (*storage classes*) khác nhau của C. Vậy tầm sử dụng và thời gian tồn tại của biến là gì?

- Tầm sử dụng của biến (*scope*) là nơi mà biến có thể được sử dụng trong các lệnh của chương trình. Do đặc tính này mà ta có hai lớp lưu trữ khác nhau là lớp lưu trữ biến toàn cục (*global storage class*) và lớp lưu trữ biến cục bộ (*local storage class*).
- Thời gian tồn tại của biến (*time life*) xác định rằng biến với giá trị đang tồn tại trong nó sẽ có ý nghĩa đến lúc nào. Thật ra thời gian tồn tại của biến tùy thuộc vào việc biến số đó tồn tại như thế nào trong bộ nhớ (tạm thời trong suốt thời

gian chương trình được thực thi). Nếu biến được khai báo và sử dụng tạm thời trong một khối lệnh, nó sẽ mất đi khi khối lệnh được thực thi xong, thì biến đó thuộc lớp biến tự động, nhưng nếu biến lại được khai báo và sử dụng trong suốt thời gian chạy chương trình thì ta lại có lớp biến tĩnh.

Như vậy có hai yếu tố tạo ra các lớp lưu trữ biến khác nhau:

- Tầm sử dụng sinh ra hai lớp lưu trữ: lớp biến toàn cục và lớp biến cục bộ
- Thời gian tồn tại sinh ra hai lớp: lớp biến tự động và lớp biến tĩnh

Sự kết hợp giữa các lớp biến này với nhau sẽ tạo ra các biến mà khi ta khai báo và sử dụng sẽ có những đặc điểm riêng. Ta có thể tóm tắt sự kết hợp ấy bằng một bảng như sau:

Lớp biến	Tự động	Tĩnh
Toàn cục	(không kết hợp được)	Biến toàn cục tĩnh
Cục bộ	Biến cục bộ tự động (hay biến tự động)	Biến cục bộ tĩnh

Như vậy, ta cần phải nắm rõ cách hoạt động của từng lớp để khai báo và sử dụng, cũng như tối ưu hóa chương trình.

11.2 BIẾN TOÀN CỤC VÀ BIẾN CỤC BỘ

11.2.1 Biến cục bộ

Biến cục bộ, còn gọi là biến tự động (auto), là các biến được khai báo ngay sau cặp dấu móc {và} (cặp dấu này như đã biết để bắt đầu cho một lệnh phức hoặc một tham hàm), hoặc là các biến được khai báo trong danh sách đối số của hàm.

Khi khai báo biến cục bộ ta có thể đặt hoặc không đặt từ khóa auto phía trước khai báo biến cục bộ theo cú pháp như sau:

[auto] **kiểu danh_sách_tên_bien:**

Từ khóa auto trong cú pháp được đặt trong dấu ngoặc cho thấy rằng nó có thể có hoặc không có thì biến cục bộ được khai báo luôn là biến auto.

Ví dụ 11.1

```
int tong (int n)
{
    auto int i;      → biến i là biến cục bộ trong hàm tong(),
    ...            biến này chỉ sử dụng được trong hàm này mà thôi
}
```

Nhưng trong thực tế khai báo, từ khóa auto này luôn luôn được hiểu ngầm, và do đó cũng không cần thiết phải viết nó ra nữa.

Ví dụ 11.2

Xét chương trình sắp xếp hai số, in ra kết quả theo thứ tự từ lớn tới nhỏ

```
#include <stdio.h>
#include <conio.h>
main()
{
    auto int a, b;
    clrscr();
    printf ("Moi nhap hai so: ");
    scanf ("%d %d", &a, &b);
    if (b > a)
    {
        auto int temp;
        temp = a;
        a = b;
        b = temp;
    }
    printf("Ket qua sap xep hai so: %d %d \n", a, b);
    getch();
}
```

tầm sử dụng của
hai biến a và b

Trong chương trình trên, biến *a* và *b* là hai biến tự động được khai báo trong hàm main(), nên chúng chỉ có thể được sử dụng bởi các lệnh trong hàm main() mà thôi. Biến *temp* là biến cục bộ trong khối lệnh của lệnh if (*b* > *a*), nên nó chỉ có nghĩa trong khối lệnh này mà thôi.

Đặc điểm của các lớp biến tự động:

- Biến tự động chỉ có thể sử dụng trong phần chương trình mà nó được khai báo, bắt đầu từ chỗ nó được khai báo đến khi kết thúc khối lệnh mà trong đó nó được khai báo. Như vậy, chỉ có các lệnh bên trong thân hàm hoặc khối lệnh mà biến được khai báo mới sử dụng được nó mà thôi.
- Mỗi khi gặp khai báo biến cục bộ, C sẽ cấp chỗ cho các biến này trong vùng nhớ stack; khi khối lệnh hoặc hàm được thực hiện xong, các biến trong vùng nhớ stack này được (tự động) giải phóng. Như vậy, biến cục bộ không chiếm những vị trí cố định và thường xuyên trong bộ nhớ, chúng chỉ tồn tại một cách tạm thời, từ lúc C bắt đầu thực hiện lệnh khai báo chung cho đến khi C kết thúc khối lệnh chứa khai báo ấy. Ngoài khoảng thời gian đó, các biến này không được biết đến.

Đây có thể nói là một ưu điểm của C, vì khi chương trình được thực thi không phải tất cả các biến đều được sử dụng đồng thời, ta nên chia biến cục bộ nào được khai báo ở đâu để không tốn bộ nhớ (vì tất cả các biến đều được cấp bộ nhớ cùng một lúc); mặt khác biến đó là cục bộ trong khối lệnh mà nó được khai báo nên chúng không thể bị các hàm khác truy xuất đến; tuy nhiên, các biến này sẽ không giữ được giá trị để tham khảo sau khi hàm hoặc khối lệnh được thực hiện xong.

Đối với hàm cũng vậy, hàm nếu được khai báo sử dụng (tức khai báo prototype) bên trong một hàm nào đó thì hàm cũng chỉ có ý nghĩa bên trong hàm này mà thôi.

Ví dụ 11.3 Xét chương trình sau đây trong đó hàm chia() được đặt trong hàm main(), nên chỉ có các lệnh trong hàm main() mới gọi được hàm chia() này mà thôi.

```
#include <stdio.h>
#include <conio.h>
main()
{
    int so1, so2;
    double kq;
```

```

double chia (double a, double b); → hàm chia được khai báo tại đây
clrscr();
printf ("Moi nhap hai so:");
scanf ("%d %d", &so1, &so2);
kq = chia (so1, so2); → hàm chia chỉ được gọi trong
printf ("ket qua la %.2lf", kq);     hàm main() mà thôi
getch();
}

double chia (double a, double b)
{
    return a/b;
}

```

11.2.2 Biến toàn cục

Biến toàn cục (*global*) hay còn gọi là biến ngoài là biến được khai báo ở bên ngoài tất cả các hàm. Biến này có thể được sử dụng để liên kết trị giữa các hàm khác nhau mà việc truyền theo tham số trở nên rắc rối và phức tạp. Các hàm sử dụng chung biến toàn cục có thể nằm trong cùng một tập tin hoặc có thể nằm trong các tập tin khác nhau.

Ví dụ 11.4 Xét chương trình ví dụ sau:

```

#include <stdio.h>
#include <conio.h>
int a, b;      → biến toàn cục
void swap(void);
main()
{
    clrscr();
    /* bắt đầu nhập trị cho hai biến toàn cục a và b */
    printf ("Moi nhap hai so: ");
    scanf ("%d %d", &a, &b);
    /* gọi hàm swap() để hoán đổi trị giữa hai biến này */
    swap();
    /* In trị của hai biến a và b */
    printf ("Ket qua sap xep hai so: %d %d \n", a, b);
}

```

```

    getch();
}

void swap(void)
{
    /* kiểm tra trị của hai biến a và b trong hàm swap() */
    if (b > a)
    {
        /* khai báo thêm biến temp làm biến tạm để đổi trị */
        auto int temp;
        temp = a;
        a = b;
        b = temp;
    }
}

```

Trong chương trình ví dụ trên, hai biến *a*, *b* là hai biến toàn cục, chúng được khai báo trước ngoài hàm main() và swap(), biến này có thể được sử dụng trong cả hai hàm này. Trong hàm main(), hai biến này được nhập trị, và in trị, còn trong hàm swap(), hai biến này được kiểm tra để hoán đổi trị. Như vậy, hai biến *a*, *b* hoàn toàn có khả năng truyền trị giữa hai hàm main() và swap(): trị nhập từ hàm main(), được đưa cho hàm swap() để hoán đổi trị (nếu thỏa điều kiện), sau đó trong hàm main() lại dùng hàm printf() để in trị của hai biến này ra.

Biến toàn cục có khả năng sử dụng rộng rãi và mạnh, khi sử dụng ta cần phải nắm các đặc điểm của lớp biến này cho chắc chắn để phối hợp sử dụng với biến cục bộ. Hai đặc điểm của lớp biến ta cần nắm là *tầm sử dụng* và *thời gian tồn tại*.

- Tầm sử dụng của biến toàn cục là toàn bộ chương trình, điều đó có nghĩa là biến toàn cục có thể được sử dụng trong tất cả các hàm (nếu trong các hàm này không có biến cục bộ trùng tên). Chính vì ưu điểm này mà biến toàn cục có thể được dùng để truyền trị giữa nhiều hàm khác nhau khi việc sử dụng truyền theo đối số trở nên dài dòng và phức tạp, khó theo dõi, nhất là khi số trị cần dùng chung lại trở nên nhiều trị.

- Các biến toàn cục sau khi khai báo sẽ được cấp vùng nhớ riêng, kích thước của biến sẽ tùy thuộc vào kiểu mà biến được khai báo. Vùng nhớ cung cấp cho mỗi biến toàn cục sẽ được dành riêng cho biến trong suốt thời gian chương trình được thực thi, vì vậy giá trị của biến không bị mất đi trong suốt quá trình làm việc. Điều này rất thuận lợi cho việc tham khảo, sử dụng trị của biến toàn cục. Các biến này có thể bị thay đổi trị bởi bất cứ hàm nào, vào bất kỳ lúc nào.

Tuy nhiên, không phải chỉ các điểm mạnh trên đây mà ta sử dụng biến toàn cục một cách tràn lan, vì có một số lý do sau đây:

- Các biến toàn cục chiếm chỗ trong bộ nhớ suốt khoảng thời gian chương trình được thực thi. Như vậy, theo nguyên tắc sử dụng biến của C thì biến nào cần sử dụng sẽ được khai báo, có những biến chưa cần sử dụng đến cũng được khai báo trước, và như vậy sẽ chiếm chỗ trước trong bộ nhớ, nếu biến toàn cục này quá nhiều có thể sinh ra việc thiếu bộ nhớ, chương trình không thực thi được.
- Điểm mạnh của biến toàn cục là cho phép các hàm sử dụng chung biến toàn cục để trao đổi trị, tuy nhiên đây thực ra không phải là việc nên làm vì các hàm trong đó có sử dụng biến toàn cục sẽ mất tính tổng quát của hàm. Một hàm khi được thiết kế thì có thể sử dụng trong nhiều chương trình khác nhau, nếu có biến toàn cục trong hàm thì trong mỗi chương trình sử dụng ta đều phải khai báo biến toàn cục có tên đúng với tên biến toàn cục sử dụng trong hàm. Và đây thực sự là một bất tiện. Trong thực tế, người ta rất hạn chế khai báo và sử dụng biến toàn cục, nếu muốn thay đổi trị giữa các hàm thì cố gắng truyền qua đối số.

Ngoài ra, khi dùng biến toàn cục trong hàm có thể gây ra các tác động không mong muốn cho chương trình, mà điều này có thể xảy ra những lỗi rất khó sửa.

Ví dụ 11.5

Xét hai chương trình ví dụ sau đây:

Chương trình 1:

```
#include <stdio.h>
#include <conio.h>
double so1, so2;
double kq;
void chia (void);
main()
{
    clrscr();
    printf ("Moi nhap hai so: ");
    scanf ("%lf %lf", &so1, &so2);
    chia ();
    printf ("ket qua la %.2lf", kq);
    getch();
}
void chia (void)
{
    kq = so1/so2;
}
```

Chương trình 2:

```
#include <stdio.h>
#include <conio.h>
double chia (double a, double b);
main()
{
    int so1, so2;
    double kq;
    clrscr();
    printf ("Moi nhap hai so ");
    scanf ("%d %d", &so1, &so2);
    kq = chia (so1, so2);
    printf ("ket qua la %.2lf", kq);
    getch();
}
double chia (double a, double b)
{
    return a/b;
}
```

Trong chương trình 1, hàm chia() sử dụng biến toàn cục *kq* và *so1*, *so2* để tính toán và trả trị về trực tiếp cho biến toàn cục, như vậy hàm này không thể được sử dụng cho chương trình khác. Trong khi đó chương trình 2, hàm chia() sử dụng đối số để nhận trị truyền vào, tính toán và trả trị cho bên ngoài bằng lệnh **return**. Hàm này có thể được sử dụng cho các chương trình khác. Như vậy hàm chia() trong chương trình 2 có tính tổng quát hơn trong chương trình 1 vì ta không cần quan tâm đến biến nào được khai báo và sử dụng trong hàm chia().

Như trên đã đề cập, biến toàn cục có thể được các hàm trong cùng một tập tin chương trình sử dụng, nhưng do khả năng của C, một chương trình có thể nằm trên nhiều tập tin nguồn khác nhau, mỗi tập tin như vậy ta gọi là một module chương trình, biến toàn cục còn có thể được các hàm ở các module khác nhau sử dụng. Mỗi module này được dịch riêng rẽ và sau cùng chúng sẽ được liên kết (*link*) lại với nhau để tạo ra được một tập tin thực thi duy nhất (đối với C, tập tin thực thi luôn có thuộc tính .exe).

Biến toàn cục là biến mà nó chỉ được khai báo một lần duy nhất trong chương trình, như vậy nếu một chương trình lại được thiết kế thành nhiều module chương trình thì biến toàn cục phải được khai báo trong một module chương trình nào đó, nhưng nó lại có thể được sử dụng bởi tất cả các hàm khác ở module khác của chương trình.

Như vậy, nếu có một biến toàn cục nào đó đã được khai báo trong một module của chương trình, và một hàm trong một module khác lại muốn sử dụng biến này để truyền trị, C đưa ra cú pháp sau đây:

extern kiểu tên_bien_toan_cuc;

Khai báo này giống như một khai báo biến nhưng có thêm từ khóa **extern** phía trước, và tên biến chỉ là sự liệt kê các tên biến, đối với biến là chuỗi hay mảng thì ta không cần nêu rõ số lượng phần tử của mảng.

Khai báo này được đặt đầu module chương trình chứa hàm sử dụng biến toàn cục, nó có tác dụng báo cho chương trình dịch khi dịch biết rằng các biến có tên đặt sau từ khóa **extern** là các biến toàn cục, và các biến này đã được khai báo và cấp chỗ trong bộ nhớ ở một module nào đó của chương trình. Nếu không có khai báo này, việc sử dụng các biến này trong các hàm sẽ gây ra lỗi.

Đĩ nhiên khai báo **extern** chỉ được đặt ở đầu các module cần sử dụng biến toàn cục mà thôi, còn trong bản thân module khai báo và định nghĩa biến toàn cục thì không cần từ khóa này.

Đối với hàm cũng vậy, khi một hàm được khai báo và định nghĩa trong một module, thì trong một module khác muốn sử dụng nó ta cần phải khai báo hàm trong module này theo cú pháp sau:

extern kiểu tên_hàm (danh_sách_khai_báo_đối_số);

Khai báo này thật sự chỉ là prototype của hàm thêm từ khóa **extern** phía trước.

Ví dụ 11.6

Xét hai chương trình có hai module khác nhau MAIN.C và FUNC.C nhập vào một chuỗi, thực hiện một trong hai thao tác lật ngược chuỗi hoặc kiểm tra ký tự thường đổi ra ký tự hoa tùy theo menu chọn của người sử dụng.

Module MAIN.C:

```
#include <stdio.h>
#include <conio.h>
#define MAX 100
char s[MAX];
extern void dao_chuoi (char s[]);
extern void doi_chuoi (char s[]);
main()
{
    char thao_tac;
    clrscr();
    printf ("Moi nhap mot chuoi: ");
    gets (s);
    printf ("Moi chon thao tac: \n");
    printf ("1. Dao chuoi \n");
    printf ("2. Doi chuoi \n");
    printf ("Thao tac: ");
    thao_tac = getch();
    if (thao_tac == '1')
```

→ biến chuỗi s được khai báo tại đây
khai báo hai hàm cần sử dụng ở đây,
nhưng chúng lại được định nghĩa trong
module FUNC.C

```

{
    dao_chuoi (s);
    printf ("\n Chuoi ban da dao la: ");
    puts (s);
}
else if (thao_tac == '2')
{
    doi_chuoi (s);
    printf ("\n Chuoi ban da doi la: ");
    puts (s);
}
else
    printf ("\n Ban chon thao tac sai \n");
getch();
}

```

Module FUNC.C:

```

#include <string.h>
extern char s[];           → biến chuỗi s được khai báo cần sử dụng tại đây
void dao_chuoi (char s[])   → định nghĩa hàm dao_chuoi()
{
    int i, j;
    char tam;
    for (i = 0, j = strlen(s) - 1; i < j; i++, j--)
    {
        tam = s[i];
        s[i] = s[j];
        s[j] = tam;
    }
}
void doi_chuoi (char s[])   → định nghĩa hàm doi_chuoi()
{
    int i;
    for (i = 0; s[i] != '\0'; i++)
        s[i] = (s[i] >= 'a' && s[i] <= 'z')? s[i]-32: s[i];
}

```

Trong ví dụ trên, khai báo biến

```
char s [MAX];
```

nằm trong tập tin MAIN.C, theo nguyên tắc biến s này là toàn cục và các hàm trong module MAIN.C hoàn toàn có khả năng sử dụng biến này, nhưng các hàm trong module khác của chương trình thì lại không thể truy xuất đến nó được, muốn vậy thì trong module chương trình FUNC.C ta phải có thêm một khai báo rằng biến s đã có khai báo rồi, nhưng ở module khác của chương trình, bằng khai báo **extern** như sau:

```
extern char s[];
```

Trong khai báo này, ta không cần phải nêu rõ chuỗi s có MAX ký tự. Tương tự như vậy, các hàm doi_chuoi() và dao_chuoi() được khai báo và định nghĩa trong tập tin FUNC.C, muốn sử dụng các hàm này trong hàm main() trong module MAIN.C, thì đầu module chương trình MAIN.C ta cần có khai báo extern cho các hàm như sau:

```
extern void dao_chuoi (char s[]);
extern void doi_chuoi (char s[]);
```

Chú ý:

Ta có thể tạo và dịch lại tập tin riêng rẽ, sau đó liên kết chúng lại bằng khái niệm **project** trong môi trường IDE của C/C++. (Xin xem thêm phụ lục cuối sách).

11.3 BIẾN TĨNH (*Static*)

Ngoài hai lớp biến toàn cục và biến cục bộ, C còn có loại biến khác gọi là biến tĩnh (*static*). Đây là một loại biến rất đặc biệt của C so với hai lớp biến trên. Để khai báo biến tĩnh ta cần thêm từ khóa **static** trước khai báo biến bình thường, cú pháp như sau:

```
static kiểu danh_sách_tên_biến;
```

Khai báo này là một khai báo biến bình thường, sau khai báo này, các biến trong danh sách tên biến được cấp chỗ để sử dụng, tùy khai báo này đặt trong hay ngoài hàm mà ta có biến toàn cục tĩnh hoặc biến cục bộ tĩnh.

- **Biến toàn cục tĩnh** là biến khai báo ngoài tất cả các hàm, trong một module chương trình nào đó và chỉ có ý nghĩa sử dụng bởi các hàm trong cùng module đó mà thôi. Các hàm trong các module khác của chương trình không thể sử dụng được các biến toàn cục dạng static như thế này.
- **Biến cục bộ tĩnh** là các biến được khai báo trong hàm và chỉ có ý nghĩa sử dụng trong hàm có khai báo đó mà thôi. Nhưng các biến cục bộ tĩnh khác với biến cục bộ (hay tự động) ở thời gian tồn tại, biến tĩnh tồn tại suốt trong bộ nhớ từ lúc nó được sử dụng lần đầu tiên cho đến khi kết thúc chương trình, và giá trị của chúng không hề mất đi khi ra khỏi hoặc trở vào hàm chứa nó.

Ví dụ 11.7

```

static int a;           ← biến toàn cục tĩnh
main()
{
    clrscr();
    ...
}

int func(void)
{
    static int b;       ← biến cục bộ tĩnh
    ...
}

```

Đặc điểm của biến tĩnh:

- Biến static có tầm sử dụng là toàn bộ module hoặc khối lệnh mà nó được khai báo trong đó, như vậy biến static chỉ có thể được truy xuất bởi các lệnh trong cùng một hàm, một khối lệnh với nó nếu là biến cục bộ tĩnh, còn nếu là biến toàn cục tĩnh thì tất cả các hàm trong cùng một module với nó đều có thể sử dụng nó để trao đổi trị.
- Như trên đã có đề cập, biến tĩnh có thời gian tồn tại từ lần sử dụng đầu tiên đến khi kết thúc chương trình, vùng nhớ cấp cho nó không bị hủy khi hàm hoặc khối lệnh kết thúc,

như vậy giá trị của nó không bị mất đi sau mỗi lần thực hiện hàm hoặc khởi lệnh.

Ví dụ 11.8 Xét chương trình tính tổng

$s = 1 + \dots + n$

dùng hàm trong đó có khai báo biến static.

```
#include <stdio.h>
#include <conio.h>
int tong (int a);
main()
{
    int n, i, kq;
    clrscr();
    printf ("Nhập trị n: ");
    scanf ("%d", &n);
    for (i = 1; i <= n; i++)
        kq = tong (i);
    printf ("Kết quả: %d", kq);
    getch();
}
```

```
int tong (int a)
{
    static int tam = 0;
    tam += a;
    return t;
}
```

Chương trình trên có dữ liệu xuất ví dụ như sau:

```
Nhập trị n : 4
Kết quả : 10
```

Trong chương trình trên, trong hàm tong(), ta có khai báo một biến cục bộ tĩnh, biến *tam*, biến này chỉ được khởi động trị một lần đầu chương trình, trị 0, sau đó trị của biến này luôn được giữ lại cho lần sử dụng sau. Hàm tong() được truyền đối số là *i*, ý nghĩa là mỗi trị số từ 1 đến *n*, trị *i* này được truyền vào cho hàm và nó được cộng

vào cho biến static *tam* trong hàm *tong()*, trị của biến *tam* được lưu giữ sau mỗi lần gọi, điều này có nghĩa là biến *tam* trong hàm *tong()* lưu trị cộng dồn từ 1 đến *n*. Việc nhận trị trả về của hàm *tong()* sau mỗi lần gọi

kq = tong (i);

hoàn toàn mang tính hình thức, vì thực ra chỉ quan tâm đến trị sau cùng mà hàm trả về, vì trị đó mới thật là tổng của dãy từ 1 đến *n*. Như vậy, một câu hỏi được đặt ra là sau khi tính tổng xong, trị của biến static *tam* trong hàm *tong()* luôn khác 0, thế thì làm sao dùng trở lại hàm này để tính tổng khác? Nếu muốn điều đó thì ta có nhiều cách, một trong những cách đơn giản là ta thiết kế thêm một hàm dùng để xóa biến cục bộ tĩnh. Trong ví dụ trên, hàm xóa này được viết như sau:

```
void xoa (void)
{
    int temp;
    if ( (temp = tong(0)) != 0 )
        tong(-temp);
}
```

Như vậy, chương trình trên nếu gọi sử dụng nhiều lần hàm *tong()*, thì trước khi gọi lần sau ta chỉ cần gọi hàm *xoa()* là trị của biến cục bộ tĩnh trong hàm *tong()* sẽ bị đặt về 0.

Ví dụ 11.9

```
#include <stdio.h>
#include <conio.h>
int tong (int a);
void xoa (void);
main()
{
    int n, i, kq;
    clrscr();
    printf ("Nhập trị n: ");
    scanf ("%d", &n);
    for (i = 1; i <= n; i++)
        kq = tong (i);
    xoa ();
    getch();
}
```

```

kq = tong (i);
printf ("Ket qua %d\n", kq);
getch();
xoa();
printf ("Nhập lại trị n: ");
scanf ("%d", &n);
for (i = 1; i <= n; i++)
    kq = tong (i);
printf ("Kết quả: %d", kq);
getch();
}

int tong (int a)
{
    static int t = 0;
    t += a;
    return t;
}

void xoa (void)
{
    int temp;
    if ( (temp = tong(0)) != 0 )
        tong(-temp);
}

```

gọi tong(0) để biết trị hiện thời
của biến static tam đang có trị
thể nào, nếu trị này khác 0, trừ
biến static tam với đối số của
nó: -temp

Như vậy, biến cục bộ tĩnh là biến cục bộ, nên ngoài các đặc điểm riêng của biến cục bộ, nó còn có các đặc điểm mà ta đã đề cập ở trên. Ví dụ sau đây sẽ trình bày về cách sử dụng biến toàn cục tĩnh, là một cách sử dụng khác của biến tĩnh.

Ví dụ 11.10

Xét chương trình sau đây có hai tập tin MAIN.C và FUNC.C, mỗi module chương trình sẽ lưu các hàm riêng, trong tập tin module FUNC.C, ta có khai báo ba biến toàn cục tĩnh trong module này, nhằm mục đích sử dụng chung trị giữa các hàm trong module FUNC.C.

Module MAIN.C

```
#include <stdio.h>
#include <conio.h>

/* khai báo các hàm sử dụng trong module này, nhưng lại được khai báo
và định nghĩa trong một module chương trình nào khác */
extern void nho_lon (int a, int b, int c);
extern void lon_hon (int a, int b, int c);

/* hàm chương trình chính */
main()
{
    int a, b, c;
    char thao_tac;
    clrscr();
    printf ("Chuong trinh sap xep 3 tri a, b, c \n");
    printf ("Nhap tri a, b, c: ");
    scanf ("%d %d %d",&a, &b, &c);
    printf ("Chon thao tac: \n");
    printf ("1. Tu lon nho \n");
    printf ("2. Tu nho lon \n");
    printf ("Moi chon: ");
    thao_tac = getch();
    switch (thao_tac)
    {
        case '1':
            /* nếu thao tác chọn là '1', tức sắp xếp theo thứ tự
            từ lớn tới nhỏ, thì cần gọi hàm lon_hon () trong module FUNC.C */
            lon_hon (a, b, c);
            break;
        case '2':
            /* nếu thao tác chọn là '2', tức sắp xếp theo thứ tự từ
            nhỏ tới lớn, thì cần gọi hàm nho_lon () trong module FUNC.C */
            nho_lon (a, b, c);
            break;
    }
}
```

default:

```
/* nếu chọn sai thao tác, chương trình cần phải báo ra
màn hình rằng thao tác đã được chọn sai */
printf ("\nBan da nhap sai thao tac\n");
break;
}
getch();
}
```

Module FUNC.C

```
#include <stdio.h>

static int max, min, med;

int lon_nhat(int a, int b, int c)    → các biến toàn cục tĩnh này chỉ
{                                         được sử dụng trong module
                                         FUNC.C này mà thôi
    int temp;
    temp = a;
    if (temp < b)
        temp = b;
    if (temp < c)
        temp = c;
    return temp;
}

int nho_nhat(int a, int b, int c)
{
    int temp;
    temp = a;
    if (temp > b)
        temp = b;
    if (temp > c)
        temp = c;
    return temp;
}

void lon_nho(int a, int b, int c)
{
```

```

max = lon_nhat (a, b, c);
min = nho_nhat (a, b, c);
med = a + b + c - max - min;
printf ("\nBa so theo thu tu lon toi nho: %d %d %d\n", max, med, min);
}

void nho_lon(int a, int b, int c)
{
    max = lon_nhat (a, b, c);
    min = nho_nhat (a, b, c);
    med = a + b + c - max - min;
    printf ("\nBa so theo thu tu tu nho toi lon: %d %d %d\n", min, med, max);
}

```

Trong chương trình trên, ba biến *max*, *med* và *min* là ba biến toàn cục chỉ có nghĩa trong module chương trình FUNC.C, các hàm *lon_nho()* và *nho_lon()* đều có thể sử dụng các biến này để tính toán. Chú ý rằng, trong các hàm của module chương trình FUNC.C, ta có sử dụng các hàm của thư viện chuẩn của C, đó là hàm *printf*, vì vậy cần phải khai báo prototype của hàm này vào đầu module chương trình qua lệnh tiền xử lý

```
#include <stdio.h>
```

Chú ý rằng, biến static được C xem như biến riêng của hàm hay của khối lệnh mà nó được khai báo nên việc đặt tên biến không sợ bị trùng và báo lỗi gì cả.

Ví dụ 11.11

Trong MAIN.C có

...

```

static int a;           → biến a toàn cục của module MAIN.C
void thu(void)
main()
{
    ...
}
```

Trong FUNC.C có

```
...
static int a;           → biến a toàn cục của module FUNC.C
int tinh (...)

{
    ...
}
```

Trong ví dụ trên, hai biến tĩnh *a* được khai báo trong hai module khác nhau nên mỗi biến chỉ có ý nghĩa sử dụng trong module mà nó được khai báo.

Cũng tương tự như biến static, hàm cũng có thể là hàm static. Hàm được khai báo là static thì nó chỉ có thể được sử dụng trong module mà nó được khai báo và định nghĩa mà thôi, điều này hạn chế việc trùng hàm khi một chương trình lớn nhiều người viết trên nhiều tập tin module chương trình khác nhau.

Cú pháp khai báo và định nghĩa hàm static như sau:

```
static kiểu tên_hàm (danh_sách_khai_báo_đối_số)
{
    ...
}
```

Ví dụ 11.12 Trong chương trình ví dụ 11.10, hai hàm lon_nho() và nho_lon () là hai hàm bình thường, chúng có thể được gọi bởi các hàm trong cùng một module hoặc khác module với chúng. Tuy nhiên, nếu ta đặt thêm các từ khóa static trước chúng thì chúng trở nên các hàm tĩnh, và các hàm này chỉ có thể được truy xuất bởi các hàm trong cùng module FUNC.C với nó mà thôi.

```
static int lon_nhat(int a, int b, int c)
{
    ...
}

static int nho_nhat(int a, int b, int c)
{
    ...
}
```

Như vậy chương trình trong ví dụ 11.10 có thể được viết lại với module MAIN.C vẫn giữ không đổi, còn module FUNC.C có thể được viết lại như sau:

Module FUNC.C

```
#include <stdio.h>

static int max, min, med; ←
static int lon_nhat (int a, int b, int c)
{
    int temp;
    temp = a;
    if (temp < b)
        temp = b;
    if (temp < c)
        temp = c;
    return temp;
} ←
static int nho_nhat(int a, int b, int c) ←
{
    int temp;
    temp = a;
    if (temp > b)
        temp = b;
    if (temp > c)
        temp = c;
    return temp;
} ←
void lon_nho(int a, int b, int c)
{
    max = lon_nhat (a, a, c); ←
    min = nho_nhat (a, b, c); ←
    med = a + b + c - max - min;
    printf ("\nBa so theo thu tu lon toi nho: %d %d %d\n", max, med, min);
}
```

*các biến toàn cục tĩnh
và hàm tĩnh này chỉ có
thể được sử dụng trong
module này mà thôi*

gọi sử dụng các hàm

```

void nho_lon(int a, int b, int c)      gọi sử dụng các hàm
{
    max = lon_nhat (a, b, c); ←
    min = nho_nhat (a, b, c); ←
    med = a + b + c - max - min;
    printf ("\nBa so theo thu tu tu nho toi lon: %d %d %d\n", min, med, max);
}

```

Mọi sự truy cập sử dụng các hàm lon_nhat() và nho_nhat(), bởi các hàm trong module chương trình MAIN.C đều bị báo lỗi.

11.4 BIẾN REGISTER

C là một ngôn ngữ được đánh giá là cấp cao nhưng có thể viết được các lệnh theo cách của cấp thấp, hoặc viết trực tiếp lệnh hợp ngữ trong chương trình C. Bộ dịch C cho phép tận dụng các tài nguyên có sẵn của máy để tối ưu hóa chương trình, một trong các tối ưu này là C cho phép lập trình viên sử dụng một số thanh ghi của bộ vi xử lý để khai báo biến, biến này gọi là biến thanh ghi (register). Đây là một lớp lưu trữ biến đặc biệt.

Để khai báo biến thanh ghi ta cần đặt thêm từ khóa register trước khai báo biến bình thường theo cú pháp sau:

register kiểu danh_sách_tên_bien;

với kiểu là kiểu khai báo cho biến, kiểu này chỉ có thể là int, char, unsigned, long hoặc pointer. Nếu kiểu không có cụ thể thì mặc nhiên là kiểu int.

Ví dụ 11.13

```

register int l;
register char c;
register unsigned u;
register long l;
register int *r;
register t;

```

Các khai báo biến thanh ghi chỉ có thể được đặt bên trong một hàm hoặc đầu một lệnh phức (khối lệnh) như khai báo biến cục bộ hoặc khai báo đối số hàm. Như vậy, tầm sử dụng và thời gian tồn tại của các biến thanh ghi tương tự như các biến cục bộ, nhưng chúng được truy xuất nhanh hơn các biến cục bộ bình thường vì chúng chính là các thanh ghi của bộ vi xử lý. Do đó, các biến thanh ghi thường được sử dụng làm các biến điều khiển trong các vòng lặp hoặc các biến phải truy xuất nhiều lần trong chương trình.

Ví dụ 11.14

Xét chương trình sau đây, trong chương trình ta có hai vòng lặp for trong đó không có lệnh thực hiện, chương trình sẽ in ra màn hình kết quả so sánh giữa hai lớp biến: biến cục bộ và biến thanh ghi nhằm mục đích tham khảo.

```
#include <stdio.h>
#include <conio.h>
#include <dos.h>
main()
{
    register unsigned i, j;
    unsigned u, v;
    int how_long;
    char h, m, s;
    struct time t;
    clrscr();
    printf ("Chuong trinh thu bien register \n");
    printf ("Theo dong ho cua may nay, bay goi la \n");
    gettime(&t);
    h = t.ti_hour;
    m = t.ti_min;
    s = t.ti_sec;
    printf ("%d gio %d phut %d giay\n", t.ti_hour, t.ti_min, t.ti_sec);
    for (i = 1; i <= 10000; i++)
        for (j = 10000; j > 0; j--)
```

```

    ;
gettime(&t);

printf ("Sau khi thuc hien xong vong lap, bay goi la \n");
printf ("%d gio %d phut %d giay\n", t.ti_hour, t.ti_min, t.ti_sec);
how_long = (t.ti_hour-h)*3600 + (t.ti_min-m)*60 +t.ti_sec-s;
printf ("Vong lap dung bien register ton khoang thoi gian:\n");
textcolor (YELLOW);
cprintf ("%d giay \n\r", how_long);
printf ("\nChuong trinh thu bien cuc bo \n");
printf ("Theo dong ho cua may nay, bay goi la \n");
gettime(&t);
h = t.ti_hour;
m = t.ti_min;
s = t.ti_sec;
printf ("%d gio %d phut %d giay\n", t.ti_hour, t.ti_min, t.ti_sec);
for (u = 1; u <= 10000; u++)
    for (v = 10000; v > 0; v--)
        ;
gettime(&t);
printf ("Sau khi thuc hien xong vong lap, bay goi la \n");
printf ("%d gio %d phut %d giay\n", t.ti_hour, t.ti_min, t.ti_sec);
how_long = (t.ti_hour-h)*3600 + (t.ti_min-m)*60 +t.ti_sec-s;
printf ("Vong lap khong dung bien register ton khoang thoi gian:\n");
textcolor (YELLOW);
cprintf ("%d giay\n\r", how_long);
getch();
}

```

Chương trình trên cho dữ liệu xuất ví dụ trên máy chạy chương trình này như sau:

Chuong trinh thu bien register

Theo dong ho cua may nay, bay goi la.

9 gio 42 phut 31 giay

Sau khi thuc hien xong vong lap, bay goi la

9 gio 43 phut 1 giay

Vong lap dung bien register ton khoang thoi gian:

30 giay

Chuong trinh thu bien cuc bo

Theo dong ho cua may nay, bay goi la

9 gio 43 phut 1 giay

Sau khi thuc hien xong vong lap, bay goi la

9 gio 43 phut 56 giay

Vong lap khong dung bien register ton khoang thoi gian:

55 giay

Chương trình trên có sử dụng một số vấn đề mới như:

- kiểu **struct time**, kiểu này đã được khai báo sẵn trong file DOS.H, với cấu trúc như sau:

```
struct time {
    unsigned char ti_min; /* minutes */
    unsigned char ti_hours; /* hours */
    unsigned char ti_hund; /* hundredths of seconds */
    unsigned char ti_sec; /* seconds */
};
```

- Hàm **gettime()** dùng để lấy ra thời gian hiện hành trong đồng hồ của máy, hàm này có prototype nằm trong file DOS.H.

Qua chương trình ví dụ trên ta thấy rằng, trong những trường hợp biến cần sử dụng đi lại nhiều lần, thì biến thanh ghi nên được sử dụng để thời gian chạy chương trình được rút ngắn. Tuy nhiên, vì số lượng thanh ghi trong bộ vi xử lý có giới hạn nên việc sử dụng biến này cũng cần một số chú ý như sau:

- Biến thanh ghi chỉ chấp nhận một số kiểu biến nguyên như int, char, unsigned, long và pointer mà thôi.
- Do số thanh ghi trong mỗi bộ vi xử lý là có giới hạn (tùy thuộc vào họ vi xử lý) và tùy lúc chạy chương trình, mà các biến thanh ghi có thể thật sự là thanh ghi của bộ vi xử lý hay không. Nếu số biến thanh ghi xin trong phần khai báo là nhiều

quá, thì biến không được chấp nhận là thanh ghi sẽ được cấp chỗ trong bộ nhớ nên chúng thành biến tự động bình thường. C hoàn toàn không thông báo rằng có bao nhiêu biến và biến nào là biến thanh ghi thật sự. Như vậy, để đảm bảo các biến được khai báo là biến thanh ghi thật sự, ta không nên khai báo quá nhiều biến thanh ghi. Đối với máy PC hoặc tương thích, số biến thanh ghi thật sự thay đổi từ 0 đến 2.

- Biến thanh ghi sử dụng thanh ghi lưu trữ dữ liệu, vì vậy không thể lấy được địa chỉ của biến thanh ghi.

11.5 KHỞI ĐỘNG TRỊ CHO BIẾN Ở CÁC LỚP

Trên ta đã xét các lớp biến lưu trữ, mỗi lớp có một đặc điểm riêng và tùy lớp mà C có khả năng tự động gán giá trị ban đầu cho biến lúc chúng được khai báo. Như vậy cần hiểu rõ việc khởi động trị này của các lớp để không phải tốn thời gian khởi động trị của chúng trong chương trình. Đối với các lớp biến không được C khởi động trị thì lập trình viên phải tự khởi động trị lúc khai báo hoặc trước khi cần sử dụng.

- Đối với biến toàn cục hoặc biến tĩnh, ngay sau khi được khai báo, mỗi biến sẽ được C tự động gán trị là 0. Trong khi đó biến tự động và biến thanh ghi sẽ có giá trị không xác định (gọi là trị rác). Nếu muốn các biến sau khai báo có trị theo yêu cầu thì lập trình viên phải khởi động trị cho biến qua phép gán trong khai báo biến.

Ví dụ 11.15

```
#include <stdio.h>
```

```
...
```

```
int n; → biến n được C tự động khởi động trị là 0
```

```
main()
```

```
{
```

```
    double b; → trị đang có trong biến b là không xác định
```

```
...
```

```
}
```

trong khi đó

```
#include <stdio.h>
...
int n = 10; → biến n được lập trình viên khởi động trị là 10 lúc khai báo
main()
{
    double b = 3.45; → biến b được lập trình viên khởi động trị
    ...
}
```

- Trong suốt quá trình chạy chương trình, biến toàn cục và biến tĩnh chỉ có thể được khởi động trị một lần, đó là lần đầu tiên mà khai báo biến đó được thực thi. Sau lần đầu lệnh khai báo biến đó được thực hiện, lệnh này được C bỏ qua trong những lần gọi hàm sau (chính vì vậy mà trị của biến toàn cục hay biến tĩnh không bị thay đổi trong những lần gọi hàm sau). Đối với biến tự động và biến thành ghi, lệnh khai báo biến lại được C thực thi mỗi lần hàm hay khối lệnh được gọi sử dụng. Như vậy, các giá trị khởi động do lập trình viên quy định, được C gán cho biến và hàm được thực thi bình thường.
- Biến toàn cục và biến tĩnh có thể được khởi động trị bằng một biểu thức hằng.

Ví dụ 11.16

```
int i = -234;
char c = 'U';
long double ld = 123. / 34578.56;
```

- Biến tự động và biến thành ghi có thể được khởi động trị bằng một biểu thức mà giá trị của biểu thức tới lúc đó đã xác định, trong biểu thức đó có thể có gọi hàm.

Ví dụ 11.17

```
main()
{
    int x = 10, y = 3;
    int i = random(10);
    double a = i + x/y;
    ...
}
```

- Việc tự động khởi động trị ban đầu bằng 0 cho các biến thuộc kiểu dữ liệu có cấu trúc như mảng (array), struct và union chỉ có thể thực hiện được đối với các biến toàn cục hoặc biến tĩnh mà thôi. Trong các chương về mảng và struct, union ta sẽ đề cập lại vấn đề này.

11.6 SỰ CHUYỂN KIỂU

Khi thực hiện các phép toán số học hoặc luận lý, C luôn thực hiện sự chuyển kiểu tự động, mà quy luật chuyển kiểu này đã được đề cập trong chương 8 khi đề cập đến các kiểu dữ liệu cơ bản. Tuy nhiên, trong quá trình tính toán, nếu chỉ dựa vào việc chuyển kiểu tự động không thôi chưa đủ, mà C còn cho phép lập trình viên thực hiện việc chuyển kiểu bắt buộc, mà ta có thể gọi nôm na là ép kiểu (*type casting*). Cú pháp để ép kiểu một biến, hằng hay biểu thức:

`(type) giá_trị`

với `type` là kiểu mà ta muốn ép về cho `giá_trị` để thực hiện phép toán.

Ví dụ 11.18

Cho khai báo biến sau:

```
int a = 10, b = 3;
double d;
```

Nếu thực hiện

```
d = a/b;
```

thì trị `d` sau khi gán là `d = 3.00`, trong khi ta lại muốn trị này là `3.33`. Muốn vậy, ta cần phải ép một trong hai biến `a` hoặc `b` theo kiểu `double` trong biểu thức tính toán `a/b`, như vậy lệnh chia nên viết lại như sau:

```
d = (double)a/b;
```

hoặc

```
d = a/(double)b;
```

11.7 ĐỊNH VỊ VÙNG NHỚ CHO CÁC LỐP LƯU TRỮ

Trong chương 5, chúng ta đã biết được cách thức một bộ dịch dịch code chứa các biến và toán tử sang mã máy. Có hai cơ chế cơ bản giúp bộ dịch làm công việc này. Bộ dịch cần dùng một cách đúng dắn bảng biểu trưng để theo dõi các biến trong quá trình dịch. Bộ dịch cũng theo một sự phân chia bộ nhớ hệ thống, nó cần thận định vị bộ nhớ cho các biến dựa vào các đặc tính cụ thể, với các vùng nhớ xác định dành riêng cho các đối tượng đặc biệt. Trong mục này chúng ta sẽ nhìn kỹ hơn hai quá trình này.

11.7.1 Bảng biểu trưng

Trong chương 5, chúng ta đã khảo sát làm thế nào bộ hợp dịch theo dõi các nhãn trong một chương trình hợp ngữ một cách hệ thống. Cũng như bộ hợp dịch, bộ dịch C theo dõi các biến trong một chương trình với một bảng biểu trưng. Bất kỳ lúc nào khi bộ dịch đọc một khai báo biến, nó tạo ra một đầu vào mới trong bảng biểu trưng của nó tương ứng với biến được khai báo. Đầu vào này chứa đủ thông tin để bộ dịch quản lý vùng nhớ cấp cho biến và sự phát ra trình tự mã máy thích hợp bất kỳ khi nào biến được dùng trong chương trình. Mỗi đầu vào của bảng biểu trưng cho biến chứa (1) tên của biến, (2) kiểu của biến, (3) vị trí trong bộ nhớ mà biến đó được định vị, và (4) một danh hiệu chỉ định khu vực mà trong đó biến được khai báo (tức tầm vực của biến đó). Ta hãy xét một chương trình ví dụ sau.

Ví dụ 11.19 Chương trình tính tốc độ mạng

```
#include <stdio.h>
int main()
{
    int amount;
    int rate;
    int time;
    int hours;
    int minutes;
    int seconds;
    // Nhập: số lượng byte và tốc độ truyền của mạng
```

```

printf ("Có bao nhiêu byte dữ liệu được truyền? ");
scanf ("%d", &amount);
printf ("Tốc độ truyền (bytes/giây)? ");
scanf ("%d", &rate);
// Tính thời gian theo số giây
time = amount / rate;
// Chuyển thời gian sang giờ, phút giây
hours = time / 3600;           // 3600 giây trong một giờ
minutes = (time % 3600) / 60;   // 60 giây trong một phút
seconds = (time % 3600) % 60;   // phần dư còn lại là giây
// Xuất ra kết quả
printf("Thời gian: %dh %dm %ds\n", hours, minutes, seconds);
}

```

Bảng 11.1 trình bày các đầu vào của bảng biểu trưng tương ứng với các biến được khai báo trong chương trình tính tốc độ mạng trong ví dụ 11.19. Vì chương trình này chứa sáu khai báo biến, nên trong bảng biểu trưng có sáu đầu vào được sắp theo thứ tự *a*, *b*, *c*. Chú ý là bộ dịch ghi vị trí các biến trong bộ nhớ theo thứ tự khai báo trong chương trình theo các offset tính từ đầu vùng nhớ mà nó định vị. Biến đầu tiên *amount* có offset 0, biến kế tiếp *rate* có offset -1, ...

Bảng 11.1 Bảng biểu trưng khi bộ dịch dịch chương trình

Danh hiệu	Kiểu	Vị trí (offset)	Tầm vóc
Amount	int	0	Main
Hours	int	-3	Main
Minutes	int	-4	Main
Rate	int	-1	Main
Seconds	int	-5	Main
Time	int	-2	Main

11.7.2 Định vị vùng nhớ cho biến

Có hai vùng nhớ mà các biến C được định vị ở đó: vùng dữ liệu toàn cục (*global data section*) và ngăn xếp thực thi (*run-time stack*)

(trong thực tế C còn định vị tối ưu một số biến trong thanh ghi của bộ xử lý để tối ưu thời gian thực thi lệnh). Vùng biến toàn cục là nơi chứa tất cả các biến toàn cục. Tổng quát hơn, nó cũng là nơi chứa các biến tĩnh. Vùng stack thực thi là nơi chứa các biến cục bộ (hay lớp biến lưu trữ tự động).

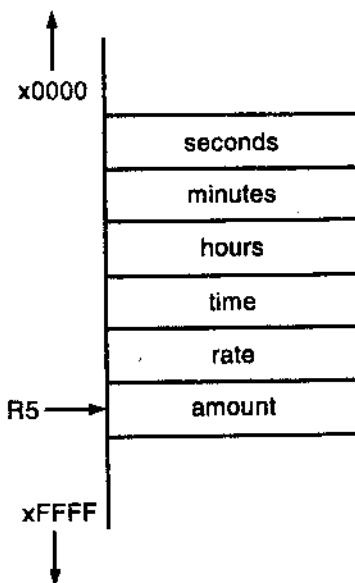
Vùng offset trong bảng biểu trưng cung cấp thông tin chính xác về vị trí trong bộ nhớ của các biến. Nó cho biết số ô nhớ tính từ địa chỉ nền của vùng nhớ chứa biến.

Ví dụ, nếu biến toàn cục *earth* có offset là 4 và vùng nhớ toàn cục bắt đầu từ địa chỉ 0x5000, thì *earth* được lưu giữ ở vị trí 0x5004. Trong các ví dụ về mã may được tạo bởi bộ dịch, thanh ghi R4 được dùng để chứa địa chỉ bắt đầu của vùng nhớ toàn cục, ta gọi nó là con trỏ toàn cục (*global pointer*). Ví dụ, để nạp trị từ biến *earth* vào thanh ghi R3, ta dùng lệnh LC-3 sau:

```
LDR R3, R4, #4
```

Nếu *earth* là biến cục bộ, ví dụ như trong hàm *main()*, thì vấn đề trở nên phức tạp hơn. Tất cả các biến cục bộ trong một hàm được định vị trong một vùng nhớ được gọi là mẫu tin kích hoạt (*activation record*) hay còn gọi là khung ngăn xếp (*stack frame*). Bây giờ chúng ta hãy xem lại định dạng của mẫu tin kích hoạt mà nó đã được sử dụng trong chương 10 khi khảo sát hàm. Một mẫu tin kích hoạt là một vùng nhớ có nhiều ô nhớ liên tục nhau dùng để chứa tất cả các biến cục bộ trong hàm. Mỗi hàm đều có một mẫu tin kích hoạt riêng (hay nói chính xác hơn là mỗi một lần gọi hàm đều sinh ra một mẫu tin kích hoạt khác nhau).

Bất cứ khi nào chúng ta thực thi một hàm, địa chỉ bộ nhớ cao nhất của mẫu tin kích hoạt sẽ được chứa trong thanh ghi R5, nên R5 được gọi là con trỏ khung (*frame pointer*). Ví dụ, mẫu tin kích hoạt cho hàm *main()* trong chương trình ví dụ 11.19 được cho trong hình 11.19. Chú ý là các biến trong mẫu tin được sắp theo thứ tự ngược với thứ tự khai báo. Vì biến *amount* được khai báo đầu tiên, nên nó xuất hiện gần với con trỏ khung R5 nhất.



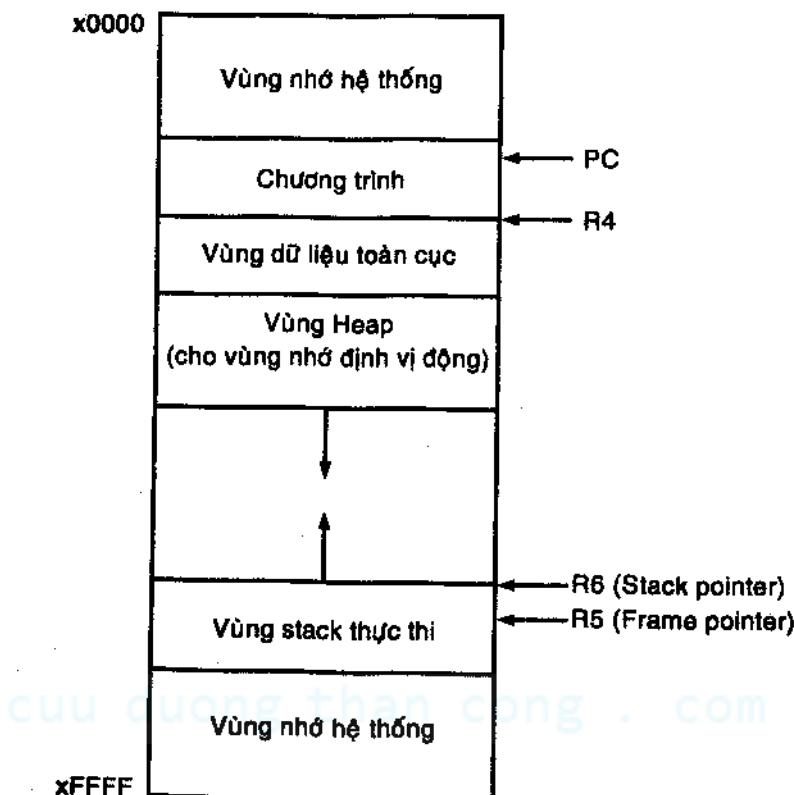
Hình 11.1 Mẫu tin kích hoạt của chương trình ví dụ 11.19

Nếu chúng ta muốn tham khảo tới một biến cục bộ nào đó, bộ dịch sẽ dùng đầu vào trong bảng biểu trưng của biến đó để tạo ra code thích hợp cho việc truy xuất nó. Đặc biệt, offset trong đầu vào ở bảng biểu trưng chỉ ra chỗ trong mẫu tin kích hoạt mà biến đã được định vị. Để truy xuất biến *seconds*, bộ dịch tạo ra lệnh sau:

LDR R0, R5, # -5

Có thể tóm tắt như sau: khi chúng ta gọi một hàm trong C, mẫu tin kích hoạt cho hàm được khởi tạo và được đẩy vào stack thực thi, lúc này nó nằm ở đỉnh stack. Thanh ghi R5 tương ứng được thay đổi để chỉ tới nền của mẫu tin này, vì vậy bất kỳ code nào trong hàm mà muốn truy xuất các biến cục bộ thì đều làm việc đúng. Khi hàm hoàn thành và điều khiển được trả về cho nơi gọi, mẫu tin kích hoạt được lấy ra khỏi stack. Thanh ghi R5 được thay đổi chỉ tới mẫu tin kích hoạt của hàm nơi gọi. Trong suốt quá trình trên, thanh ghi R6 luôn luôn chứa địa chỉ của đỉnh stack thực thi, nó được gọi là con trỏ stack (*stack pointer*). Chúng ta đã thấy vấn đề này trong chương 10-Hàm.

Hình 11.2 cho thấy tổ chức của bộ nhớ LC-3 khi một chương trình đang chạy. Nhiều hệ điều hành như UNIX, Windows sắp xếp bộ nhớ tương tự như vậy.



Hình 11.2 Bản đồ bộ nhớ LC-3 khi một chương trình đang thực thi

Chương trình bắn thân nó chiếm một vùng nhớ (vùng nhớ với nhãn “Chương trình”); vùng stack thực thi và vùng dữ liệu toàn cục cũng vậy. Có một vùng khác được dành riêng cho dữ liệu định vị động, được gọi là vùng heap. Vùng heap này được dùng để định vị các đối tượng được tạo ra trong quá trình thực thi chương trình. Cả hai vùng stack thực thi và heap đều có thể thay đổi kích thước khi chương trình thực thi. Ví dụ, khi một hàm gọi một hàm khác, stack thực thi sẽ lớn lên vì chúng ta đẩy thêm mẫu tin kích hoạt khác vào stack theo hướng địa chỉ bộ nhớ x0000. Ngược lại, vùng heap nở lớn theo hướng địa chỉ xFFFF. Vì stack nở lớn theo hướng x0000, nên các thành phần của mẫu tin kích hoạt xuất hiện theo chiều “từ trên xuống”, nghĩa là biến cục bộ đầu tiên định vị ở ô nhớ chỉ bởi R5, biến kế ở R5-1, kế nữa là R5-2, và cứ thế.

Trong quá trình thực thi, thanh ghi PC luôn chỉ tới một vị trí nào đó trong vùng “Chương trình”, R4 chỉ tới đầu vùng nhớ dữ liệu toàn

cục, R5 chỉ tới ô nhớ trong vùng stack thực thi, còn thanh ghi R6 luôn chỉ tới đỉnh trên cùng của stack thực thi. Có các vùng nhớ quy định, như vùng nhớ hệ thống (*System space*) được hệ điều hành sử dụng chứa các thứ như các thủ tục TRAP, bảng vector, các thanh ghi I/O và mã khởi động.

11.7.3 Ví dụ

Bây giờ chúng ta hãy khảo sát các kỹ thuật của bộ dịch LC-3 để theo dõi và định vị biến trong bộ nhớ từ một chương trình ví dụ cụ thể với ngôn ngữ C và mã LC-3 của nó sau khi đã được dịch.

Ví dụ 11.20 là một chương trình C với một số thao tác tính toán đơn giản trên các biến nguyên (kiểu int) và xuất ra các kết quả của các phép tính này. Chương trình bao gồm một biến toàn cục, inGlobal, và ba biến cục bộ, inLocal, outLocalA, và outLocalB, trong hàm main().

Chương trình bắt đầu bằng việc gán các trị ban đầu cho inGlobal và inLocal. Sau bước này, các biến outLocalA và outlocalB được gán trị bằng hai phép toán được thực hiện bằng việc dùng trị từ các biến inLocal và inGlobal tính ra. Sau đó, các giá trị của các biến outLocalA và outlocalB được xuất ra màn hình bằng hàm thư viện printf(). Chú ý, vì chúng ta dùng hàm printf(), nên chúng ta phải bao hàm tập tin đầu thư viện xuất nhập chuẩn của C, stdio.h.

Ví dụ 11.20

```
#include <stdio.h>
int inGlobal; /* toàn cục */
main()
{
    int inLocal; /* cục bộ trong hàm main() */
    int outLocalA;
    int outLocalB;
    /* khởi tạo trị */
    inLocal = 5;
    inGlobal = 3;
    /* thực hiện phép tính */
    outLocalA = inLocal++ & ~inGlobal;
```

```

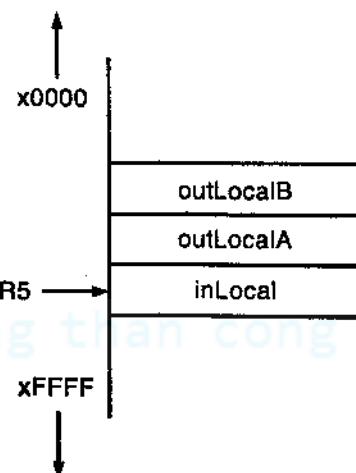
outLocalB = (inLocal + inGlobal) - (inLocal - inGlobal);
/* In kết quả */
printf("Kết quả là outLocalA = %d, outLocalB = %d\n", outLocalA, outLocalB);
}

```

Khi phân tích mã C trên, bộ dịch C của LC-3 sẽ dùng vùng nhớ khả dụng đầu tiên ở offset 0 trong vùng dữ liệu toàn cục cho biến toàn cục *inGlobal*. Khi phân tích hàm *main()*, bộ dịch sẽ *inLocalA* cho vùng offset 0, *outLocalA* cho vùng offset -1 và *outLocalB* cho vùng offset -2 trong mẫu tin kích hoạt của hàm *main()*. Một hình ảnh về bảng biểu trưng của bộ dịch tương ứng với chương trình này cùng với mẫu tin kích hoạt của hàm *main()* được cho trong hình 11.3 dưới.

Danh hiệu	Kiểu	Vị trí (offset)	Tầm vực
<i>inGlobal</i>	int	0	global
<i>inLocal</i>	int	0	Main
<i>outLocalA</i>	int	-1	Main
<i>outLocalB</i>	int	-2	Main

a)



b)

Hình 11.3 Bảng biểu trưng (a) của bộ dịch C từ LC-3 khi dịch chương trình ví dụ 11.20 và định dạng của mẫu tin kích hoạt cho hàm *main()* của nó

Mã hợp ngữ tạo bởi bộ dịch C LC-3 được trình bày trong ví dụ 11.21 dưới đây. Việc thực thi bắt đầu từ lệnh có nhãn main.

Ví dụ 11.21

main:

; khởi động trị cho biến

```
AND R0, R0, #0
ADD R0, R0, #5 ; inLocal = 5
STR R0, R5, #0 ; (offset = 0)
AND R0, R0, #0
ADD R0, R0, #3 ; inGlobal = 3
STR R0, R4, #0 ; (offset = 0)
```

; lệnh: outLocalA = inLocal++ & ~inGlobal;

```
LDR R0, R5, #0 ; inLocal
ADD R1, R0, #1 ; tăng 1
STR R1, R5, #0 ; lưu lại
LDR R1, R4, #0 ; inGlobal
NOT R1, R1 ; ~inGlobal
AND R2, R0, R1 ; inLocal & ~inGlobal
STR R2, R5, #-1 ; lưu vào outLocalA (offset = -1)
```

; lệnh kế: outLocalB = (inLocal + inGlobal) - (inLocal - inGlobal);

```
LDR R0, R5, #0 ; inLocal
LDR R1, R4, #0 ; inGlobal
ADD R0, R0, R1 ; R0 là tổng
LDR R2, R5, #0 ; inLocal
LDR R3, R5, #0 ; inGlobal
NOT R3, R3
ADD R3, R3, #1
ADD R2, R2, R3 ; R2 là hiệu
NOT R2, R2 ; bù 1
ADD R2, R2, #1
ADD R0, R0, R2 ; R0 = R0 - R2
STR R0, R5, #-2 ; outLocalB (offset = -2)
```

<code cho hàm printf()>

BÀI TẬP CUỐI CHƯƠNG

11.1 Viết một hàm sao cho mỗi lần gọi hàm thì hàm sẽ trả về một trị số ngay sau trị trước đó trong dãy số Fibonacci.

11.2 Viết chương trình với các hàm tính các biểu thức sau đây:

$$t = \frac{1 + \dots + n}{n!} - \frac{1 + \dots + (n-1)}{(n-1)!} + \frac{1 + \dots + (n-2)}{(n-2)!} - \dots + \frac{1}{1!}$$

$$T = \frac{(1 + \dots + n) + (1 + \dots + (n-1)) + (1 + \dots + (n-2)) + \dots + 1!}{n! + (n-1)! + (n-2)! + \dots + 1!}$$

Hàm tính biểu thức trên được thiết kế theo hai cách dùng và không dùng biến thanh ghi. In thời gian kiểm tra.

11.3 Viết một chương trình gồm hai module: main.c và func.c, trong đó module main.c lưu hàm main() có các lệnh gọi nhập ba hệ số của tam thức bậc hai, kiểm tra in ấn; còn trong module func.c lưu các hàm cần thiết để giải phương trình bậc hai và biện luận tam thức bậc hai.

11.4 In ra màn hình các số nguyên tố từ 1 đến 1.000. Dùng biến thanh ghi và không dùng biến thanh ghi. Kiểm tra thời gian.

Chương 12

MẢNG

12.1 KHÁI NIỆM

Bất kỳ ngôn ngữ nào cũng đều có khái niệm mảng, đây là một khái niệm quen thuộc trong kỹ thuật lập trình.

Mảng là một biến cấu trúc trong đó có nhiều phần tử cùng kiểu, mỗi phần tử là một biến thành phần của mảng. Mỗi biến thành phần này là một biến bình thường và có cước số (*subscript*) để phân biệt giữa phần tử này và phần tử kia. Như vậy, để truy xuất một phần tử của mảng, ta cần biết được cước số của nó.

Trong bộ nhớ, các phần tử của mảng được cấp phát ô nhớ có địa chỉ liên tiếp nhau.

C cũng cho phép lập trình viên khai báo và làm việc trên mảng một chiều (*singledimensional array*) và mảng nhiều chiều (*multidimensional array*). Số phần tử trên mảng một chiều được gọi là kích thước của chiều đó.

12.2 KHAI BÁO MẢNG

Một mảng của C được khai báo giống như khai báo biến, nhưng có thêm cặp dấu [] xác định đó là một mảng.

Để dễ khảo sát, ta chia mảng ra làm hai loại: mảng một chiều và mảng nhiều chiều để xét.

1- Mảng một chiều

Cú pháp khai báo mảng một chiều như sau:

kiểu tên_mảng [kích_thước];

với:

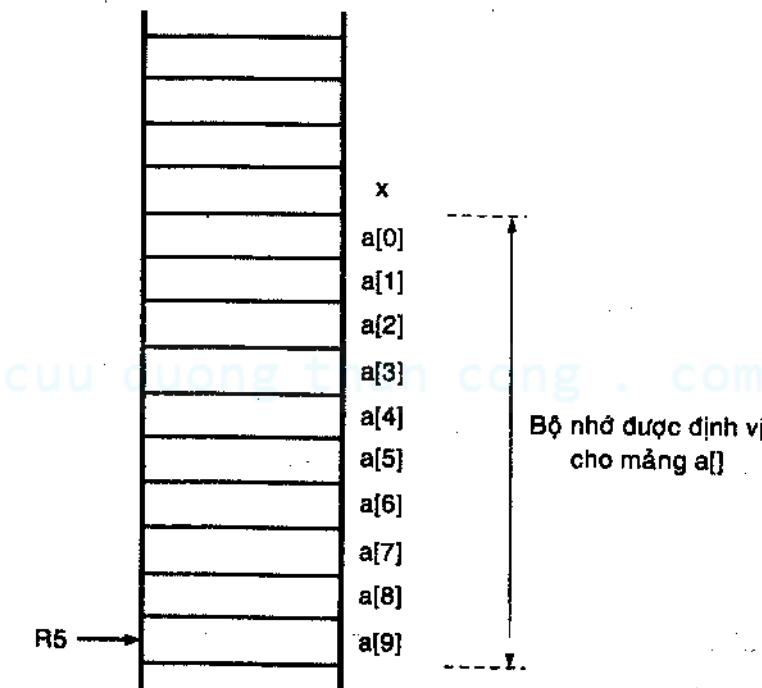
- **kiểu** là kiểu dữ liệu của các phần tử của mảng.
- **tên_mảng** là một danh hiệu không chuẩn, được dùng để truy xuất mảng.
- **kích_thước** là một hằng số nguyên cụ thể, cho biết số phần tử trên chiều đang xét.

Trong C, cước số các phần tử của mảng luôn đi từ 0 trở đi, nên mảng một chiều có **kích_thước** phần tử thì cước số các phần tử của mảng là 0,..., (**kích_thước** - 1).

Ví dụ 12.1 Cho khai báo sau:

```
int a[10], x;
```

Như vậy mảng a có 10 phần tử int, các phần tử đó là a[0], a[1], ..., a[9]. Các phần tử này được cấp phát vị trí trong bộ nhớ như hình 12.1 sau. Cần lưu ý, ngay sau khi khai báo biến trên xong, thanh ghi R6 chỉ tới đỉnh stack thực thi (tức R6 chứa địa chỉ của biến x), còn thanh ghi R5 chứa địa chỉ nền của hai biến cục bộ là mảng a, và biến x, tức R5 đang chỉ tới phần tử sau cùng của mảng, tức a[9] trong stack thực thi.



Hình 12.1 Mảng a[10] trong bộ nhớ máy tính

Lệnh

```
a[5] = a[3] + 1;
```

có mã LC-3 như sau:

```
; a[5] = a[3] + 1
ADD R0, R5, #-9 ; R0 = &a[0]: địa chỉ của a[0]
LDR R1, R0, #3 ; R1 = a[3]
ADD R1, R1, #1 ; tăng 1
STR R1, R0, #5 ; a[5] = R1, tức a[5] = a[3] + 1
```

Lệnh

```
a[5] = 7;
```

có mã LC-3 như sau:

```
; a[5] = 7;
AND R0, R0, #0
ADD R0, R0, #7 ; R0 = 7
ADD R1, R5, #-9 ; R1 = &a[0]: địa chỉ của phần tử a[0]
STR R0, R1, #5 ; a[5] = R0
```

Còn lệnh

```
a[x+1] = a[x] + 2;
```

với x là biến đang chứa trị là chỉ số nào đó cần làm việc, có mã LC-3 như sau:

```
; a[x+1] = a[x] + 2
LDR R0, R5, #-10 ; R0 = x
ADD R1, R5, #-9 ; R1 = &a[0]
ADD R1, R0, R1 ; R1 = &a[x]
LDR R2, R1, #0 ; R2 = a[x]
ADD R2, R2, #2 ; cộng thêm 2
LDR R0, R5, #-10 ; R0 = x
ADD R0, R0, #1 ; R0 = x+1
ADD R1, R5, #-9 ; R1 = &a[0]
ADD R1, R0, R1 ; R1 = &a[x+1]
STR R2, R1, #0 ; a[x+1] = R2
```

Ví dụ 12.2

Viết chương trình nhập một dãy các số nguyên, tìm số lớn nhất trong dãy số đó.

```
#include <stdio.h>
#include <conio.h>
main()
{
    int i, n, max, vtmax;
    int a[100];
    clrscr();
    printf ("Chuong trinh thu mang \n");
    printf ("Moi ban nhap so phan tu cua mang: ");
    scanf ("%d", &n);
    printf ("Moi nhap cac phan tu cua mang:");
    for (i = 0; i < n; i++)
        scanf ("%d", &a[i]);
    max = a[0];
    vtmax = 0;
    for (i = 1; i < n; i++)
        if (max < a[i])
    {
        max = a[i];
        vtmax = i;
    }
    printf ("Phan tu co cuoc so %d co tri lon nhat la %d\n", vtmax, max);
    getch();
}
```

Để nhập trị cho các phần tử của mảng, ta dùng một vòng lặp for có biến điều khiển i đi từ 0 đến $(n-1)$, ứng với mỗi trị của i ta có một phần tử $a[i]$ cần được nhập trị. Việc nhập trị cho biến thành phần $a[i]$ vẫn dùng hàm `scanf()`, để lấy địa chỉ của các phần tử của mảng ta vẫn dùng dấu `&` như đã biết khi dùng hàm `scanf`.

2- Mảng nhiều chiều

C cho phép khai báo mảng nhiều chiều, việc khai báo bình thường như mảng một chiều.

Cú pháp khai báo mảng nhiều chiều như sau:

kiểu tên_mảng [kích_thước_chiều1] [kích_thước_chiều2] [...];

với:

- **kiểu** là kiểu dữ liệu của các phần tử của mảng, tương tự như mảng một chiều.
- **tên_mảng** là một danh hiệu không chuẩn, được dùng để truy xuất mảng.
- **kích_thước_chiều1, kích_thước_chiều2,...** là kích thước của các chiều trong mảng. Số kích thước này theo lý thuyết thì không hạn chế, tuy nhiên trong thực tế bộ nhớ máy tính lại có giới hạn, nên số kích thước cũng như độ lớn của từng kích thước tùy thuộc vào bộ nhớ máy tính, nếu khi dịch C báo lỗi

Array size too large

có nghĩa là C đã không chấp nhận các kích thước này, do đó chương trình sẽ không được dịch.

Cũng tương tự như trong mảng một chiều, cước số các phần tử của mảng nhiều chiều luôn đi từ 0 trở đi, nên một chiều nào đó, ví dụ chiều 2, có kích_thước_chiều2 phần tử thì cước số các phần tử của mảng trong chiều đó là 0,..., (kích_thước_chiều2 - 1).

Ví dụ 12.3 Khai báo mảng hai chiều a

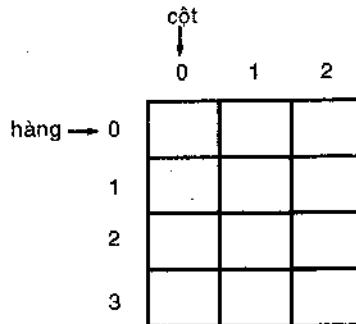
```
int a[4][3];
```

Như vậy mảng a có 4×3 phần tử int, các phần tử đó là

```
a[0][0] a[0][1] a[0][2]
a[1][0] a[1][1] a[1][2]
a[2][0] a[2][1] a[2][2]
a[3][0] a[3][1] a[3][2]
```

Các phần tử này được sắp trong bộ nhớ theo thứ tự $a[0][0]$, $a[0][1]$, $a[0][2]$, $a[1][0]$, $a[1][1]$, $a[1][2]$, $a[2][0]$, $a[2][1]$, $a[2][2]$, ...

Mảng a này có thể được xem như ma trận hai chiều 4×3 sau:



Ví dụ 12.4

Viết chương trình tạo và in ra màn hình ma trận có dạng sau:

```

    1   0   0   0
    0   1   0   0
    0   0   1   0
    0   0   0   1
  
```

Chương trình như sau:

```

#include <stdio.h>
#include <conio.h>
#define MAX 20
main()
{
    int i, j; /* hai biến điều khiển truy xuất ma trận */
    int a[MAX][MAX]; /* mảng hai chiều lưu ma trận */
    int n; /* cấp của ma trận */
    clrscr();
    printf ("Chuong trinh thu mang \n");
    printf ("Moi ban nhap cap cua ma tran: ");
    scanf ("%d", &n);
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if (i == j)
                a[i][j] = 1;
            else
                a[i][j] = 0;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            printf ("%d ", a[i][j]);
    getch();
}
  
```

```

for (j = 0; j < n; j++)
    if (i == j)
        a[i][i] = 1;
    else
        a[i][j] = 0;
printf ("Ma tran duoc tao la: \n");
for (i = 0; i < n; i++)
{
    for (j = 0; j < n; j++) → vòng for in ra từng hàng
        printf ("%d ", a[i][j]); → của ma trận
    printf ("\n"); /* xuống hàng về đầu dòng */
}
getch ();
}

```

Trong chương trình trên, hai biến i và j được dùng để làm các chỉ số đến các hàng i và cột j , từ đó ta sẽ kiểm tra nếu i bằng j , tức đang truy xuất đến phần tử nằm trên đường chéo thì trị 1 sẽ được gán cho biến thành phần $a[i][j]$, còn không biến này sẽ được gán trị 0.

Thông thường việc khai báo mảng thường được khai báo trừ hao, tức ta thường khai báo với số nhiều nhất có thể, tới khi sử dụng ta mới nhập số phần tử hay số hàng cột thật sự cần làm việc. Ta hãy xét ví dụ sau với hình vẽ ở cấp LC-3.

Ví dụ 12.5 Cho các khai báo sau:

```

#define MAX 4
int a[MAX][MAX];
int n = 3;           /* cấp thực sự cần làm việc của ma trận */
int i, j;           /* biến là chỉ số mảng */
// Nhập trị cho mảng
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        scanf ("%d", &a[i][j]);

```

Giá trị nhập vào là:

0	1	2
3	4	5
6	7	8
9	10	11

Khi đó ma trận thực sự cần làm việc là mảng $a[3][3]$, là một phần của ma trận $a[MAX][MAX]$ theo hình dưới:

$a[0][0]$	$a[0][1]$	$a[0][2]$	$a[0][3]$
$a[1][0]$	$a[1][1]$	$a[1][2]$	$a[1][3]$
$a[2][0]$	$a[2][1]$	$a[2][2]$	$a[2][3]$
$a[3][0]$	$a[3][1]$	$a[3][2]$	$a[3][3]$

Trong bộ nhớ, các phần tử của mảng sắp xếp theo trình tự địa chỉ liên tiếp nhau trong stack thực thi tương ứng. Với khai báo trên, ta có Hình 12.2 như sau.

	Stack thực thi
R6 →	3
xEFE9	3
xEFEA	3
xEFEB	0
xEFEC	1
xEFED	2
xEFEE	?
xEFEF	3
xEFF0	4
xEFF1	5
xEFF2	?
xEFF3	6
xEFF4	7
xEFF5	8
xEFF6	?
xEFF7	9
xEFF8	10
xEFF9	11
R6 → xEFFA	?

Hình 12.2 Mảng trong stack thực thi

Chú ý rằng, C sẽ hoàn toàn không báo lỗi khi việc truy xuất mảng vượt quá kích thước cho phép của mảng, và điều này sẽ dẫn đến sai lầm cho chương trình sau này.

Ví dụ 12.6 Có khai báo

```
int a[10];
```

mà ta lại thực hiện lệnh

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

thì trong thực tế không có phần tử a[10], nhưng việc gán cũng được thực hiện, và ô nhớ kế tiếp phần tử a[9] được gán trị, điều này có thể gây ra lỗi cho chương trình (vì biết đâu ô nhớ được gán trị bất đắc dĩ này lại chứa một dữ liệu quan trọng của chương trình đang được thực hiện!).

C không có sự phân biệt giữa một biến chuỗi và một mảng các ký tự. Cả hai trường hợp đều được khai báo

```
char tên [chiều_dài];
```

trong đó tên là một danh hiệu không chuẩn, và chiều dài là kích thước mảng hay chiều dài vật lý của chuỗi. Như đã đề cập trong chương 1, chuỗi là một dãy các ký tự được kết thúc bằng ký tự NUL ('\0'), nếu trong chương trình sử dụng các hàm chuẩn của C để truy xuất mảng các ký tự, thì C tự động thêm ký tự NUL này vào chuỗi để báo kết thúc chuỗi cho các hàm khác biết, hoặc C sẽ truy xuất chuỗi cho đến khi gặp ký tự này thì kết thúc. Còn nếu là chuỗi do lập trình viên tự tạo do lập trình, thì sau cùng lập trình viên cần phải thêm ký tự NUL này vào cuối chuỗi (nếu muốn chuỗi này có thể làm việc được với các hàm chuẩn truy xuất chuỗi của C); nếu lập trình viên không thêm ký tự NUL, thì thực tế ta có một mảng ký tự mà thôi.

C có hai hàm chuẩn để truy xuất chuỗi:

- **Hàm gets()** cho phép nhập một chuỗi có tên để trong đối số hàm này.

Ví dụ 12.7

```
char s[20];
gets (s);
```

- **Hàm puts()** cho phép xuất một chuỗi có tên để trong đối số hàm này ra màn hình.

Ví dụ 12.8

```
char s[20];
puts (s);
```

Cả hai hàm đều có prototype nằm trong file stdio.h.

Ví dụ 12.9

Chương trình truy xuất chuỗi dùng hàm chuẩn của C.

```
#include <stdio.h>
#include <conio.h>
main()
{
    char s[100];
    clrscr();
    printf ("Moi nhap mot chuoi: ");
    /* nhap chuoi dung ham chuẩn gets() cua C */
    gets (s);
    printf ("Chuoi da nhap la: ");
    /* xuat chuoi dung ham chuẩn puts() cua C */
    puts (s);
    getch();
}
```

Nhưng trong chương trình sau, việc nhập chuỗi do lập trình viên tự nhập bằng vòng lặp.

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

```

{
    char s[100];
    int chieu_dai = 0;
    char c;
    s[0] = '\0';
    clrscr();
    printf ("Moi nhap mot chuoi: ");
    /* nhap chuoi bang vong lap */
    do
    {
        c = getch();
        if (c != 'r')
        {
            s [chieu_dai++] = c;
            putch(c);
        }
    } while (c != 'r');
    s [chieu_dai] = '\0';
    printf ("\n");
    printf ("chuoi da nhap la: ");
    /* xuat chuoi dung ham chuan gets() cua C */
    puts (s);
    getch();
}

```

Trong chương trình trên, để nhập trị cho chuỗi, một vòng lặp do-while được sử dụng, việc nhập sẽ kết thúc khi phím Enter được nhấn. Ký tự NUL đầu tiên được gán cho ký tự s[0] để phòng trường hợp ngay từ đầu đã nhấn phím Enter. Như vậy sau khi lặp tạo chuỗi xong, cần phải gán ký tự NUL vào chuỗi qua lệnh

s [chieu_dai] = '\0';

Vì vậy mà chuỗi tự tạo này có thể được in ra màn hình bằng lệnh gọi hàm chuẩn của C puts(s).

12.3 KHỞI ĐỘNG TRỊ CỦA MẢNG

Khi khai báo mảng là biến toàn cục hoặc tĩnh thì mảng có thể được khởi động trị bằng các giá trị hằng.

Đối với mảng một chiều, việc khởi động được thực hiện bằng một phép gán một loạt các giá trị hằng nằm giữa cặp dấu móc {}, vào cho các phần tử của mảng theo thứ tự được liệt kê, các trị hằng cách nhau bằng dấu phẩy (,).

Ví dụ 12.10

```
int a[5] = {1, 3, 5, 7, 9};
```

Sau khi khởi động xong, các phần tử sẽ có trị

```
a[0] = 1, a[1] = 3, a[2] = 5, a[3] = 7, a[4] = 9
```

Nếu số trị ít hơn số phần tử mảng thì các phần tử còn lại không được khởi động trị, có nghĩa các phần tử này có trị là 0.

Ví dụ 12.11

```
int a[10] = {1, 2, 3, 4, 5};
```

thì các phần tử a[0] tới a[4] được khởi động trị, còn từ a[5] tới a[9] có trị 0 (nếu mảng được khởi động là biến toàn cục hay tĩnh).

Khi khởi động, kích thước mảng có thể không cần xác định, khi đó C sẽ tự động xem mảng có kích thước là số trị để khởi động cho các phần tử của mảng.

Ví dụ 12.12

```
double a[] = {1.23, -5.67, 9.87, 1.34};
```

Mảng a được C xem là có 4 phần tử, mỗi phần tử có trị khởi động theo thứ tự được liệt kê.

Với mảng nhiều chiều, việc khởi động cũng tương tự, kích thước chiều đầu tiên không cần cung cấp, các kích thước còn lại cần phải để đầy đủ với các trị hằng cụ thể.

Ví dụ 12.13

Cho khai báo mảng và khởi động trị như sau:

```
int a[][3] = {
    { 11, 12, 13},
    { 21, 22, 23},
    { 31, 32, 33}
};
```

Với khai báo này, mảng a sẽ có 9 phần tử trong 3 hàng với các trị được động như trên.

Đối với chuỗi ký tự, ta có thể khởi động trị bằng một hàng chuỗi có chiều dài không vượt quá kích thước khai báo (kể cả ký tự kết thúc chuỗi NUL '\0'). Khi khởi động, chiều dài chuỗi lúc khai báo không xác định thì chiều dài của chuỗi chính là chiều dài chuỗi đã khởi động cho biến chuỗi.

Ví dụ 12.14

```
char s[30] = "I go to school \n";
char ch[] = "Hello, World!";
```

Chú ý cần có sự phân biệt giữa chuỗi và mảng các ký tự.

Ví dụ 12.15

Chuỗi

```
char s[] = "Hello";
```

được lưu trong bộ nhớ

H	e	I	I	o	\0
---	---	---	---	---	----

trong khi đó, mảng

```
char ch[] = {'H', 'e', 'I', 'I', 'o'};
```

được lưu trong bộ nhớ

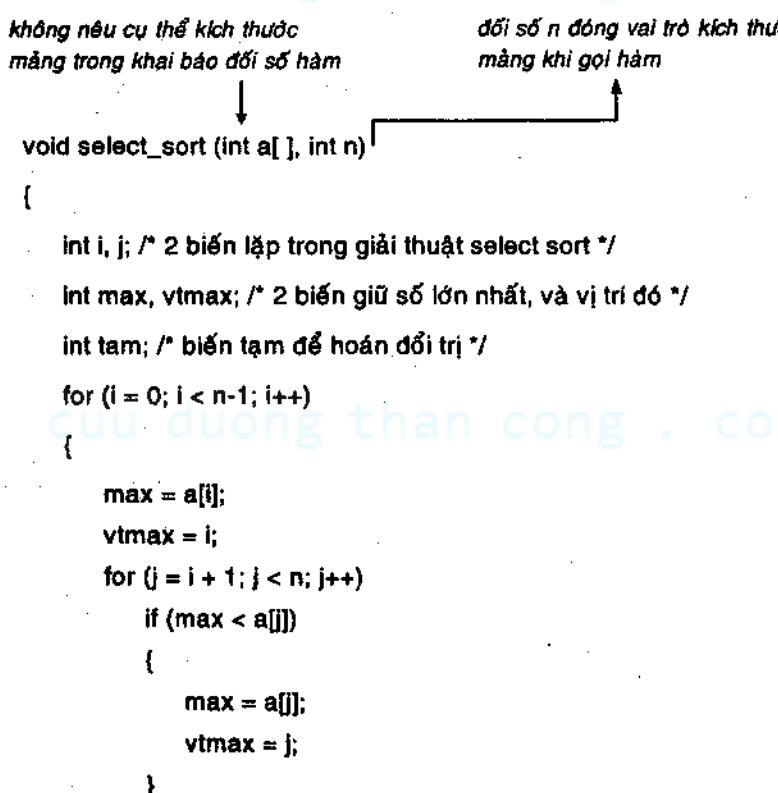
H	e	I	I	o
---	---	---	---	---

12.4 MẢNG LÀ ĐỐI SỐ CỦA HÀM, MẢNG LÀ BIẾN TOÀN CỤC

Khi khai báo đối số của hàm là mảng, thì kích thước của chiều đầu tiên của mảng không cần xác định cụ thể. Đây là một ưu điểm của C, vì một hàm có thể được dùng cho nhiều mảng, mỗi mảng có thể có kích thước khác nhau. Tuy nhiên từ chiều thứ hai trở đi, kích thước mảng phải xác định. Như vậy, đối với mảng một chiều, khi khai báo là đối số hàm, kích thước mảng không cần truyền cụ thể. Mặt khác, mảng truyền cho hàm thì truyền theo dạng tham số biến. Điều này có thể được giải thích rõ hơn trong chương con trỏ, tuy nhiên có thể nói vấn tắt, tên mảng chính là địa chỉ của mảng, nên việc truyền tên mảng cho hàm chính là truyền địa chỉ thực của mảng nên mọi thay đổi trên mảng trong hàm cũng chính là thay đổi trên mảng thật.

Ví dụ 12.16

Thiết kế hàm sắp xếp mảng một chiều theo thứ tự từ lớn tới nhỏ bằng phương pháp select sort.



```

    if (vtmax != i)
    {
        tam = a[i];
        a[i] = max;
        a[vtmax] = tam;
    }
}
}

```

Ví dụ 12.17

Thiết kế chương trình nhập ma trận hai chiều, tính tổng từng hàng của ma trận.

```

#include <stdio.h>
#include <conio.h>
#define MAX 20
void nhap_ma_tran (int a[][MAX], int n);
int tong_hang (int a[], int n);
main()
{
    int n, i;
    int a[MAX][MAX];
    clrscr();
    printf ("chuong trinh thu ma tran \n");
    printf ("Moi ban nhap cap cua ma tran: ");
    scanf ("%d", &n);
    printf ("Moi nhap cac phan tu cua ma tran:\n");
    nhap_ma_tran (a, n);
    printf ("Tong theo tung hang cua ma tran:\n");
    for (i = 0; i < n; i++)
        printf ("Hang thu %d co tong la %d \n", i+1, tong_hang(a[i], n));
    getch();
}
void nhap_ma_tran (int a[][MAX], int n)
{
    int i, j;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)

```

```

        scanf ("%d", &a[i][j]);
    }
int tong_hang (int a[], int n)
{
    int i, s = 0;
    for (i = 0; i < n; i++)
        s += a[i];
    return s;
}

```

Chương trình trên sẽ cho màn hình xuất liệu ví dụ là:

Chuong trinh thu ma tran

Moi ban nhap cap cua ma tran: 4

Moi nhap cac phan tu cua ma tran:

```

3   4   5   8
2   3   5   1
27  9   0   -18
2   3   3   8

```

Tong theo tung hang cua ma tran:

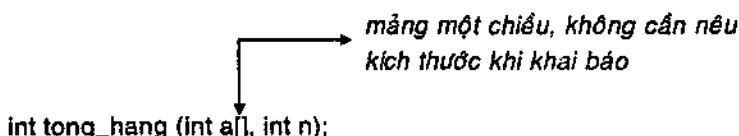
Hang thu 1 co tong la 20

Hang thu 2 co tong la 11

Hang thu 3 co tong la 18

Hang thu 4 co tong la 16

Trong chương trình trên, hai hàm nhap_ma_tran() và tong_hang() có đối số là mảng hai chiều và một chiều, khi khai báo không cần phải nêu cụ thể kích thước chiều đầu tiên của mảng. Đối với hàm tong_hang(), đối số là mảng một chiều.



nên khi gọi hàm, cần truyền đối số thật là mảng một chiều.

tong_hang (a[i], n);

↑
mảng một chiều a[i][0], ..., a[i][n-1]

Cũng tương tự như biến, khi khai báo mảng là biến toàn cục và nó được khai báo trong một module chương trình, muốn sử dụng mảng này trong module chương trình kia ta cần khai báo **extern** cho mảng này đầu module sử dụng.

Ví dụ 12.18

Trong module MAIN.C

```
#include <...>
```

```
int a[20];
```



```
main()
```

```
{
```

```
...
```

```
}
```

Trong module FUNC.C

```
#include <...>
```

```
extern int a[];
```



```
void func1 ()
```

```
{
```

```
...
```

```
}
```

mảng a[] được sử dụng
trong module FUNC.C,
nhưng được khai báo
trong module MAIN.C

Để hiểu rõ hơn quá trình truyền và nhận trị khi mảng là đối số của hàm, ta hãy xét ví dụ sau ở cấp LC-3.

Ví dụ 12.19 Xét chương trình tính trung bình của các số như sau:

```
#include <stdio.h>
#define MAX 10
int Average (int values[]);
main()
{
    int Index;
    int mean;
    int a[MAX];
```

```

printf ("Mời nhập %d số nguyên: ", MAX);
// Nhập trị cho mảng
for (index = 0; index < MAX; index++)
    scanf ("%d", &a[index]);
mean = Average (a);
printf ("Trung bình của các số này là %d.\n", mean);
}

int Average (int values[])
{
    int index;
    int sum = 0;
    for (index = 0; index < MAX; index++)
        sum += values[index];
    return (sum/MAX);
}

```

Trong chương trình này, hàm main() có ba biến cục bộ theo thứ tự khai báo là *index*, *mean* và mảng *a*. Nên trong mẫu tin kích hoạt của nó, các biến xuất hiện trong stack thực thi từ dưới lên theo chiều địa chỉ giảm dần là *index*, *mean* và mảng *a*. Hình 12.3 minh họa trạng thái của bộ nhớ ngay trước khi hàm Average() với đối số thực là mảng kết thúc.

Mẫu tin kích hoạt được khởi tạo khi hàm Average() được thực thi với đối số thực là tên mảng *a* được truyền cho đối số giả là *values*. LC-3 sẽ dành một phần tử ngay trên đỉnh stack (vốn đang được chỉ bởi R6) vào coi nó là biến *values* để chứa địa chỉ của mảng *a* truyền vào. Ví dụ theo Hình 12.3, ô nhớ xEFEE là biến *values* sẽ chứa trị xEFEF (chỉ được áp dụng khi khai báo đối số là mảng), là địa chỉ của mảng *a*, tức địa chỉ phần tử đầu tiên của mảng *a*, *a[0]*; như vậy, khi vào hàm Average(), địa chỉ của mảng *a* được cung cấp trong biến *values*, nên biến mảng *a* được sử dụng ở hình thức biến mảng *values*, đây là lý do việc truyền mảng cho hàm luôn ở dạng tham biến. Kế đó, các thành phần khác được khởi tạo, sau cùng trên đỉnh của mẫu tin kích hoạt của hàm Average() sẽ có hai biến cục bộ của hàm là *index* và *sum*.

Stack thực thi	
R6	416
R5	10
xEFEB	Con trả khung của main()
xEFEC	Địa chỉ trả về cho main()
xEFED	Trị trả về cho main()
xEFEE	xEFEEF
xEFEEF	49
xEFF0	54
xEFF1	84
xEFF2	12
xEFF3	12
xEFF4	6
xEFF5	63
xEFF6	92
xEFF7	20
xEFF8	24
xEFF9	xx
xEFFA	10

sum
index

Mẫu tin kích hoạt
của hàm Average()

Values

a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]
a[8]
a[9]
mean
index

Mẫu tin kích hoạt
của hàm main()

Hình 12.3 Stack thực thi khi hàm Average() đang được thực hiện

Như vậy, nếu ta hiệu chỉnh chương trình trên để tăng tính cơ động của chương trình, và có chương trình sau trong ví dụ 12.20, độc giả hãy vẽ ra cấu trúc bộ nhớ với các mẫu tin kích hoạt thích hợp.

Ví dụ 12.20

```
#include <stdio.h>
#define MAX 10
int Average (int values[], int number);
main()
{
    int index;
    int mean;
    int n;
```

```

int a[MAX];
// Nhập số số nguyên cần làm việc
do
{
    printf ("Bạn muốn làm việc với bao nhiêu số nguyên? (0 < n <= 10): ");
    scanf ("%d", &n);
    if (n <= 0 || n > 10)
        printf ("Sai trị. Nhập lại.\n");
} while (n <= 0 || n > 10);
printf ("Mời nhập %d số nguyên: ", n);
// Nhập trị cho mảng
for (index = 0; index < n; index++)
{
    scanf ("%d", &a[index]);
}
mean = Average (a, n);
printf ("Trung bình của các số này là %.2f\n", mean);
}

int Average (int values[], int number)
{
    int index;
    int sum = 0;
    for (index = 0; index < number; index++)
        sum += values[index];
    return (sum/number);
}

```

12.5 CÁC ỨNG DỤNG

Trong mục này, ta sẽ xét một số ứng dụng trên mảng, qua đó một số kỹ thuật lập trình cơ bản và các kiểu cấu trúc dữ liệu đặc biệt sẽ được nêu ra.

12.5.1 Sắp xếp mảng

Trong phần trên, để sắp xếp một mảng, giải thuật được nêu ra là giải thuật **select sort**, trong mục này ta sẽ xét thêm hai giải thuật nữa là giải thuật **bubble sort** và **quick sort**.

1- Bubble sort

Giải thuật sort này dựa vào nguyên tắc: phần tử nhỏ hơn sẽ "nhẹ hơn" và vì vậy sẽ "nổi" lên trên. Như vậy, đây là phương pháp so sánh trực tiếp hai phần tử trong mảng với nhau, nếu phần tử nào nhỏ sẽ được đổi chỗ sang chỗ có chỉ số (cúc số) thấp hơn (nếu việc sắp xếp theo thứ tự từ nhỏ tới lớn).

Ví dụ 12.21

Xét chương trình sau:

```
#include <stdio.h>
#include <conio.h>
#define MAX 20
void nhap_ma_tran (int a[], int n);
void in_ma_tran (int a[], int n);
void bubble_sort (int a[], int n);
main()
{
    int n, i;
    int a[MAX];
    clrscr();
    printf ("Chuong trinh thu ma tran \n");
    printf ("Moi ban nhap kich thuoc day: ");
    scanf ("%d", &n);
    printf ("Moi nhap cac phan tu cua ma tran:\n");
    nhap_ma_tran (a, n);
    bubble_sort (a, n);
    printf ("Ma tran duoc sap xep la:\n");
    in_ma_tran (a, n);
    getch();
}
void nhap_ma_tran (int a[], int n)
{
```

```

int i;
for (i = 0; i < n; i++)
    scanf ("%d", &a[i]);
}

void in_ma_tran (int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf ("%d", a[i]);
}

void bubble_sort (int a[], int n)
{
    int i, j, temp; /* 2 biến lặp trong giải thuật bubble sort */
    for (i = 0; i < n - 1; i++)
    {
        printf ("Lan lặp thứ %d\n", i);
        for (j = n - 1; j > i; j--)
            if (a[j] < a[j - 1])
            {
                temp = a[j];
                a[j] = a[j - 1];
                a[j - 1] = temp;
                in_ma_tran (a, n);
                printf ("\n");
            }
    }
}

```

Chương trình sẽ cho xuất liệu ví dụ:

Chuong trinh thu ma tran

Moi ban nhap kich thuoc day: 5

Moi nhap cac phan tu cua ma tran:

9 -5 7 0 1 ← mảng nhập

Lan lap thu 0

9 -5 7 0 1 } → vòng lặp thứ 0 của i
-5 9 7 0 1 }

Lan lap thu 1

-5 9 0 1 7 } → vòng lặp thứ 1 của i
-5 0 9 1 7 }

Lan lap thu 2

-5 0 1 9 7 } → vòng lặp thứ 2 của i

Lan lap thu 3

-5 0 1 7 9 } → vòng lặp thứ 3 của i

Ma tran duoc sap xep la:

-5 0 1 7 9

2- Quick sort

Đây là giải thuật sort được đánh giá là nhanh nhất trong các giải thuật sắp xếp. Giải thuật này được C khuyến khích sử dụng. Nguyên tắc của giải thuật này như sau:

Giải thuật **quick sort** bắt đầu bằng việc tìm một giá trị giữa tầm cho mảng. Nếu một mảng chứa các số từ 1 tới 10, giá trị giữa tầm có thể là 5 hoặc 6. Trị chính xác của giá trị giữa tầm là không quan trọng, giải thuật sẽ làm việc với giá trị giữa tầm có giá trị bất kỳ. Tuy nhiên, nếu giá trị giữa tầm càng chính xác (càng gần với giá trị giữa tầm thật của mảng) thì quá trình sắp xếp sẽ diễn ra nhanh hơn.

Giá trị giữa tầm có thể được tính bằng trung bình cộng của hai phần tử đầu tiên và sau cùng trong phần mảng đang được sắp xếp. Mỗi khi chọn xong giá trị giữa tầm này, giải thuật sẽ chuyển tất cả các phần tử có giá trị nhỏ hơn giá trị giữa tầm sang phần có chỉ số thấp của mảng, và chuyển tất cả các phần tử có giá trị lớn hơn giá trị giữa tầm sang phần có chỉ số cao của mảng. Điều này được minh họa trong ví dụ sắp xếp theo thứ tự từ nhỏ tới lớn mảng sau:

86 3 10 23 12 67 59 47 31 24

Trị giữa tầm	Vị trí trong mảng									
	0	1	2	3	4	5	6	7	8	9
Bước 1: 55	86	3	10	23	12	67	59	47	31	24
Bước 2: 35	24	3	10	23	12	31	47	59	67	86
Bước 3: 27	24	3	10	23	12	31	47	59	67	86
Bước 4: 18	24	3	10	23	12	31	47	59	67	86
Bước 5: 11	12	3	10	23	24	31	47	59	67	86
Bước 6: 6	10	3	12	23	24	31	47	59	67	86
Bước 7: 23	3	10	12	23	24	31	47	59	67	86
Bước 8: 72	3	10	12	23	24	31	47	59	67	86
Bước 9: 63	3	10	12	23	24	31	47	59	67	86
Bước 10: 63	3	10	12	23	24	31	47	59	67	86

Trong bước 1, giá trị giữa tầm là 55, trị này chính là trung bình cộng của 86 và 24. Trong bước 2, phần mảng được sắp xếp từ trị 24 đến trị 47, phần mảng này lại có trị giữa tầm là 35. Cứ như vậy tiến trình sắp xếp diễn ra trong từng đoạn mảng. Trong mỗi bước xử lý, các phần tử của đoạn mảng cần sắp xếp sẽ được sắp xếp xung quanh trị giữa tầm của nó. Các đoạn mảng cần sắp xếp ngày càng nhỏ dần, đến khi mảng được sắp xếp xong. Về chương trình minh họa cho giải thuật quick sort xin mời độc giả tự giải quyết, xem như một bài tập.

Do đó, chính vì điểm mạnh của giải thuật này mà C cung cấp trong thư viện hàm chuẩn qsort(), có prototype trong file stdlib.h như sau:

```
void qsort(void *base, size_t nelem, size_t width,  
          int (*fcmp)(const void *, const void *));
```

với: - **base** là tên mảng cần sắp xếp
- **nelem** là số phần tử của mảng cần sắp xếp
- **width** là kích thước của một phần tử của mảng
- ***fcmp** là hàm xác định quy luật sắp xếp mảng.

Xét ví dụ sau:

Ví dụ 12.22

```
#include <stdio.h>  
#include <conio.h>  
#include <stdlib.h>  
  
#define MAX 20  
  
void nhap_ma_tran (int a[], int n);  
void in_ma_tran (int a[], int n);  
int sort_function (const void *a, const void *b);  
  
main()  
{  
    int n;  
    int a[MAX];  
    clrscr();  
    printf ("Chuong trinh sap xep ma tran bang QUICK SORT \n");  
    printf ("Moi ban nhap kich thuoc day: ");  
    scanf ("%d", &n);  
    printf ("Moi nhap cac phan tu cua ma tran:\n");  
    nhap_ma_tran (a, n);  
    qsort (a, n, sizeof(a[0]), sort_function);  
    printf ("Ma tran duoc sap xep la:\n");  
    in_ma_tran (a, n);
```

```

    getch();
}

void nhap_ma_tran (int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        scanf ("%d", &a[i]);
}

void in_ma_tran (int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf ("%d", a[i]);
}

int sort_function (const void *a, const void *b)
{
    if (*(int*)a > *(int*)b)
        return 1;
    else if (*(int *)a == *(int *)b)
        return 0;
    else
        return -1;
}

```

Chương trình này cho xuất liệu, ví dụ:

Chuong trinh sap xep ma tran bang QUICK SORT

Moi ban nhap kich thuoc day: 10

Moi nhap cac phan tu cua ma tran:

86 3 10 23 12 67 59 47 31 24

Ma tran duoc sap xep la:

3 10 12 23 24 31 47 59 67 86

Chương trình trên có sử dụng một số khái niệm về biến con trả, về địa chỉ của mảng... mà trong chương sau độc giả sẽ biết rõ hơn.

12.5.2 Stack

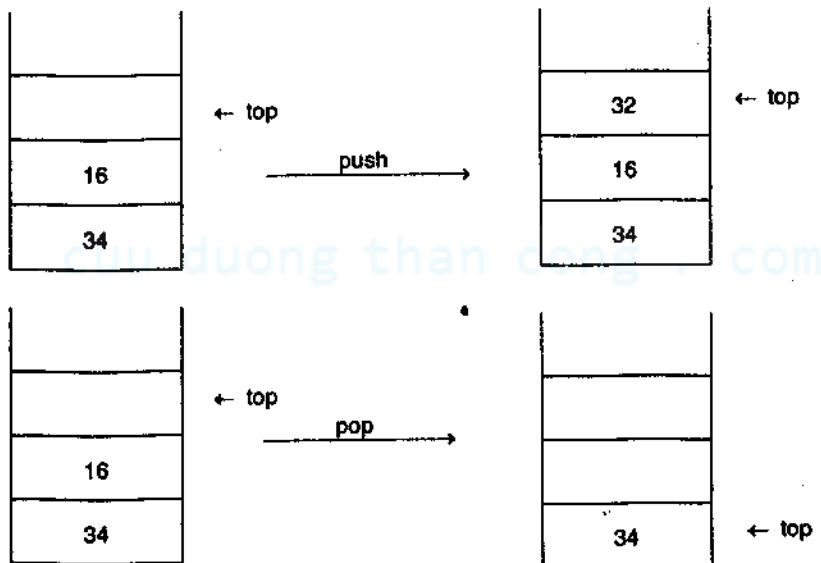
Stack (tạm dịch là *ngăn xếp*) trong lập trình là một kiểu cấu trúc dữ liệu do lập trình viên tự lập ra. Cũng tương tự như stack ở cấp thấp (chương 6), khi cần, lập trình viên có thể thêm một phần tử vào stack, hoặc xóa một phần tử ra khỏi stack. Đặc điểm của cấu trúc dữ liệu này là dữ liệu được ghi vào hoặc lấy ra khỏi stack theo trật tự **vào trước ra sau** (*last-in first-out*), có nghĩa là phần tử được ghi vào stack trước sẽ được lấy ra sau và ngược lại.

Các thao tác cần có để làm việc trên stack:

- Khởi động stack, tương ứng với hàm init_stack() cần thiết kế.
- Các hàm để xem stack rỗng, đầy, hay xem trị trên đỉnh stack.
- Đẩy một phần tử vào stack, tương ứng hàm push() cần thiết kế.
- Lấy một phần tử từ đỉnh stack ra, tương ứng với hàm pop() cần thiết kế.

Ví dụ 12.23

Xét một stack đang có dữ liệu, sau khi thực hiện hai thao tác push và pop, stack trở thành:



Để truy xuất stack, ta dùng một biến để quản lý đỉnh stack, biến đó là top. Khi biến này có trị là MAXSTACK -1 (là một trị đã được khai báo sẵn, cho biết kích thước tối đa của stack) thì stack đang đầy, không thể đẩy thêm một phần nào vào stack nữa được. Còn nếu trị top này bằng 0 thì stack đang rỗng, không thể lấy trị từ stack ra được.

Xét chương trình ví dụ sau đây

```
#include <stdio.h>
#include <conio.h>
#define MAXSTACK 100
int top;
void push (int x, int stack[]);
void pop (int *x, int stack[]);
int empty (void);
int full (void);
void init_stack (void);
main()
{
    int s[MAXSTACK], i, so_tri, tri;
    clrscr();
    printf (Nhập bao nhiêu trị vào stack: );
    scanf ("%d", &so_tri);
    init_stack();
    printf("Stack đã được khởi động, đề nghị nhập %d trị\n", so_tri);
    for (i = 0; i < so_tri; i++)
    {
        scanf("%d", &tri);
        push(tri, s);
    }
    printf ("tri trong stack: ");
    for (i = 0; i < so_tri; i++)
    {
        pop(&tri, s);
        printf("%d", tri);
    }
    getch();
}
void push (int x, int stack[])
{

```

```

if (top == MAXSTACK -1)
    printf ("Stack day \n");
else
{
    top++;
    stack[top] = x;
}
}

void pop (int *x, int stack[])
{
    if (top == -1)
        printf ("Stack rong \n");
    else
    {
        *x = stack[top];
        top--;
    }
}
int empty (void)
{
    return top == -1;
}
int full (void)
{
    return top == MAXSTACK;
}
void init_stack (void)
{
    top = -1;
}

```

Chương trình sẽ cho một xuất liệu ví dụ:

Nhap bao nhieu tri vao stack: 9

Stack da duoc khai dong, de nghi nhap 9 tri

4 0 -4 7 97 3 1 34 123

Tri trong stack: 123 34 1 3 97 7 -4 0 4

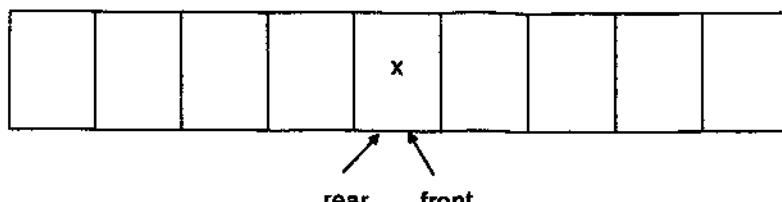
Chương trình trên cho một ví dụ về cách khai báo và sử dụng stack, qua đó lập trình viên có thể dùng cấu trúc dữ liệu này để giải quyết một số bài toán cụ thể trong lập trình.

12.5.3 Queue

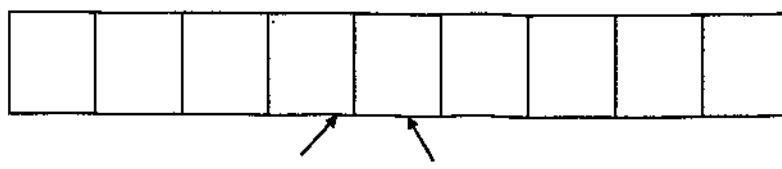
Queue là một cấu trúc dữ liệu, trong đó việc thêm dữ liệu vào được thực hiện ở một đầu, còn việc lấy một phần tử ra khỏi queue được thực hiện ở đầu kia. Dữ liệu vào ra queue theo trật tự **vào đầu tiên ra đầu tiên** (*first-in first-out*). Có thể quy ước rằng, phần tử đầu tiên ra khỏi queue gọi là **front**, phần tử sau cùng ra khỏi queue gọi là **rear**. Queue có nhiều loại, tuy nhiên loại queue được sử dụng trong lập trình nhiều vẫn là queue vòng. Để quản lý queue vòng, người ta dùng một biến đếm **count** để biết được số phần tử đang có trong queue. Cũng tương tự như đối với stack, các thao tác cần có để làm việc trên queue:

- Khởi động queue, tương ứng với hàm `init_queue()` cần thiết kế.
- Các hàm để xem queue rỗng, đầy.
- Thêm một phần tử vào queue, tương ứng hàm `addqueue()` cần thiết kế.
- Lấy một phần tử ra khỏi queue, tương ứng với hàm `deletequeue()` cần thiết kế.

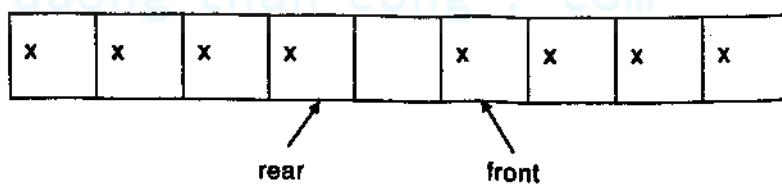
Queue vòng
Queue có một
phần tử



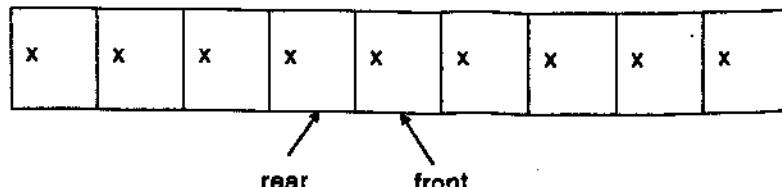
Queue rỗng



Queue với một
phần tử rỗng



Queue đầy



Ví dụ 12.24

Xét chương trình ví dụ sau:

```
#include <stdio.h>
#include <conio.h>
#define MAXQUEUE 100
int count, front, rear;
void addqueue (int x, int queue[]);
void deletequeue (int *x, int queue[]);
int empty (void);
int full (void);
void init_queue (void);
main()
{
    int q[MAXQUEUE], i, so_tri, tri;
    clrscr();
    printf ("Nhập bao nhiêu trị số vào queue: ");
    scanf ("%d", &so_tri);
    init_queue();
    printf("Queue đã được khởi động, đề nghị nhập %d trị số", so_tri);
    for (i = 0; i < so_tri; i++)
    {
        scanf("%d", &tri);
        addqueue(tri, q);
    }
    printf ("Trị số trong queue được lấy ra: ");
    for (i = 0; i < so_tri; i++)
    {
        deletequeue (&tri, q);
        printf("%d", tri);
    }
    getch();
}
void addqueue (int x, int queue[])
{
    if (count == MAXQUEUE)
        printf ("QUEUE đầy\n");
    else
```

```

    {
        count++;
        rear++;
        if (rear == MAXQUEUE)
            rear = 0;
        queue[rear] = x;
    }
}

void deletequeue (int *x, int queue[])
{
    if (count == 0)
        printf ("Queue rong \n");
    else
    {
        count--;
        *x = queue[front];
        if (front == MAXQUEUE -1)
            front = 0;
        else
            front++;
    }
}
int empty (void)
{
    return count == 0;
}
int full (void)
{
    return count == MAXQUEUE;
}
void init_queue (void)
{
    count = 0;
    front = 0;
    rear = -1;
}

```

Trong chương trình trên, biến *count* được sử dụng để theo dõi số lượng phần tử có trong queue. Khi queue rỗng, *count* bằng 0, báo không có phần tử nào trong queue, khi queue đầy, *count* bằng MAXQUEUE, báo queue đang đầy, không thể ghi thêm một phần tử nào khác vào queue nữa. Chú ý rằng, biến *rear* đang chỉ đến phần tử được ghi sau cùng vào queue nên muốn ghi một phần tử mới vào queue, trước tiên cần tăng trị của *rear* để *rear* chỉ đến ô trống, gán trị vào. Đối với *front*, *front* đang chỉ đến phần tử cần lấy ra, sau khi lấy trị ra khỏi *front*, cần thay đổi trị của *front* xuống để nó chỉ đến trị kế cần lấy ra.

BÀI TẬP CUỐI CHƯƠNG

- 12.1 Nhập một ma trận $n \times n$ bất kỳ, sắp xếp lại ma trận sao cho các trị lớn nhất trên từng hàng, nằm trên đường chéo của ma trận.
- 12.2 Viết chương trình tạo và in ra màn hình ma trận có dạng sau:

a	b	c	d
b	c	d	a
c	d	a	b
d	a	b	c

← n →

n: tham số nhập

- 12.3 Nhập một chuỗi bất kỳ từ bàn phím, xóa tất cả các ký tự khoảng trắng thừa của chuỗi và in ra màn hình chuỗi mới.
- 12.4 Nhập một dãy số từ bàn phím. Viết hai hàm để in ra màn hình biểu đồ ngang và biểu đồ dọc của các dấu * tương ứng với các số nhập trong dãy số.

Ví dụ: Nhập 2 4 3;

Xuất:

* *	* * *
* * * *	* * *
* * *	* *
	*

- 12.5 Viết chương trình tạo và in ra màn hình tam giác PASCAL cấp n, với n nhập từ bàn phím.
- 12.6 Viết chương trình tạo ma trận nghịch đảo n x n.
- 12.7 Viết chương trình giải hệ phương trình tuyến tính bằng phương pháp Gauss.
- 12.8 Nhập một ma trận vuông bất kỳ, tính tổng các hàng, các cột, các đường chéo.
- 12.9 Nhập một ma trận bất kỳ. In ra màn hình các trị và vị trí của các số nguyên tố có trong mảng đó.
- 12.10 Viết các hàm đổi từ số sang chuỗi, và từ chuỗi sang số.

cuu duong than cong . com

cuu duong than cong . com

Chương 13

POINTER

13.1 KHÁI NIỆM

Trong ngôn ngữ C, mỗi biến và chuỗi ký tự đều được lưu trữ trong bộ nhớ và có địa chỉ riêng, địa chỉ này xác định vị trí của chúng trong bộ nhớ. Khi lập trình trong C, nhiều lúc chúng ta cần làm việc với các địa chỉ này, và C ủng hộ điều đó khi đưa ra kiểu dữ liệu pointer (tạm dịch là con trỏ) để khai báo cho các biến lưu địa chỉ.

Đây là một kiểu dữ liệu đặc biệt và được sử dụng nhiều trong các chương trình C. Một biến có kiểu pointer có thể lưu được dữ liệu trong nó, là địa chỉ của một đối tượng đang khảo sát. Đối tượng đó có thể là một biến, một chuỗi hoặc một hàm. Một hằng kiểu pointer là địa chỉ của đối tượng đang khảo sát.

Việc sử dụng biến pointer khéo léo trong một chương trình C, cho phép lập trình viên nâng cao tính hiệu quả khi viết C. Đây có thể nói là một công cụ rất mạnh, nếu thiếu nó một số thao tác sẽ rất khó, thậm chí không thể thực hiện được. Tuy nhiên, nếu không khéo sử dụng và quản lý pointer, chúng ta có thể viết ra những chương trình khó hiểu hoặc khó phát hiện ra những lỗi sai. Cũng vì lý do này mà từ các phiên bản sau của TC 2.0, ta thấy có nhiều ràng buộc hơn khi sử dụng pointer khiến việc viết chương trình trở nên không mềm dẻo như trước, nhưng bù lại khả năng phát hiện sai sót nếu có sẽ được cải thiện đáng kể.

Để dễ hình dung về khái niệm pointer và khả năng của nó, ta xét chương trình ví dụ sau với hàm đổi trị cho hai đối số truyền cho hàm. Với cách truyền trị như vậy, ta không thể đổi trị được cho hai đối số của hàm như đã biết trong chương 10-Hàm.

Ví dụ 13.1 Chương trình đổi trị

```

1   #include <stdio.h>
2
3   void Swap (int doi_1, int doi_2);
4
5   main()
6   {
7       int a = 3, b = 4;    // Khai báo và khởi động trị
8
9       // In trị trước khi gọi hàm
10      printf ("Trước khi gọi hàm, trị của biến a = %d, b = %d.\n");
11
12      // Gọi hàm đổi trị
13      Swap (a, b);
14
15      // In trị sau khi gọi hàm
16      printf ("Sau khi gọi hàm, trị của biến a = %d, b = %d.\n");
17  }
18
19  void Swap (int doi_1, int doi_2)
20  {
21      int temp = doi_1;
22
23      doi_1 = doi_2;
24      doi_2 = temp;
25 }
```

Chương trình này cho xuất liệu ra màn hình như sau:

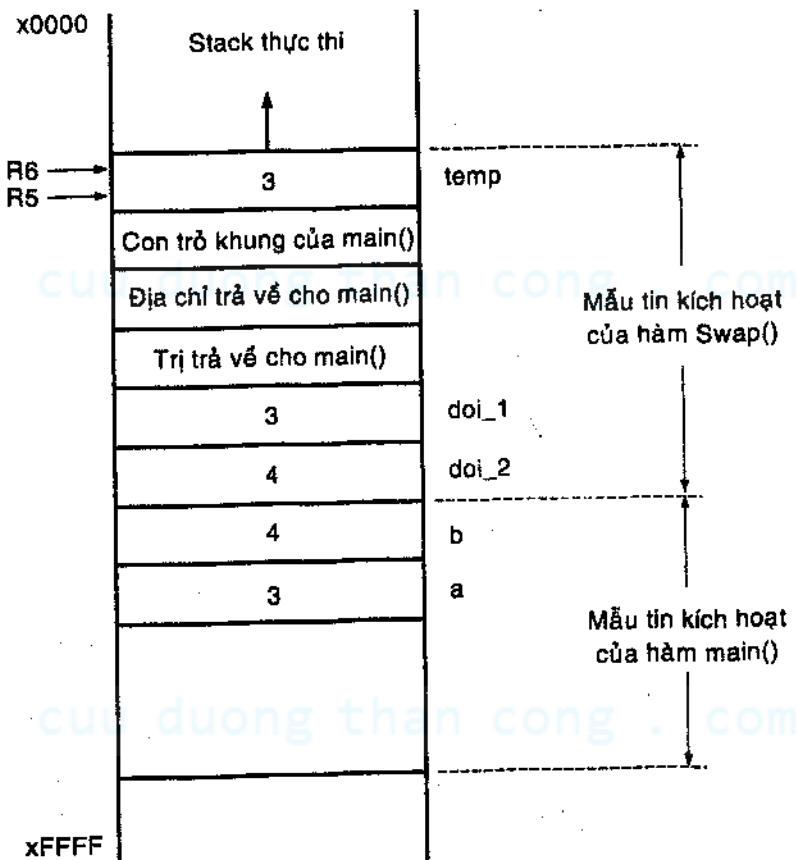
Trước khi gọi hàm, trị của biến a = 3, b = 4.

Sau khi gọi hàm, trị của biến a = 3, b = 4.

Hình 13.1 và 13.2 cho chúng ta thấy rõ ràng vấn đề. Trong Hình 13.1, hai đối số thực *a*, *b* sẽ được lấy trị và chép vào hai biến cục bộ là *doi_1* và *doi_2* trong hàm Swap() (dòng 13). Khi vào hàm, biến cục bộ *temp* được khai báo và khởi động trị bằng *doi_1* (= 3)

(dòng 21), nó nằm trong mẫu tin kích hoạt của hàm Swap(), ở đỉnh stack, và đang được chỉ bởi R6, và cũng được chỉ bởi R5 (do vùng biến cục bộ này chỉ có mỗi một biến cục bộ mà thôi). Như vậy, lúc này hai biến *a* và *b* vẫn đang có trị là 3 và 4; còn hai biến cục bộ *doi_1* và *doi_2* cũng có trị là 3 và 4.

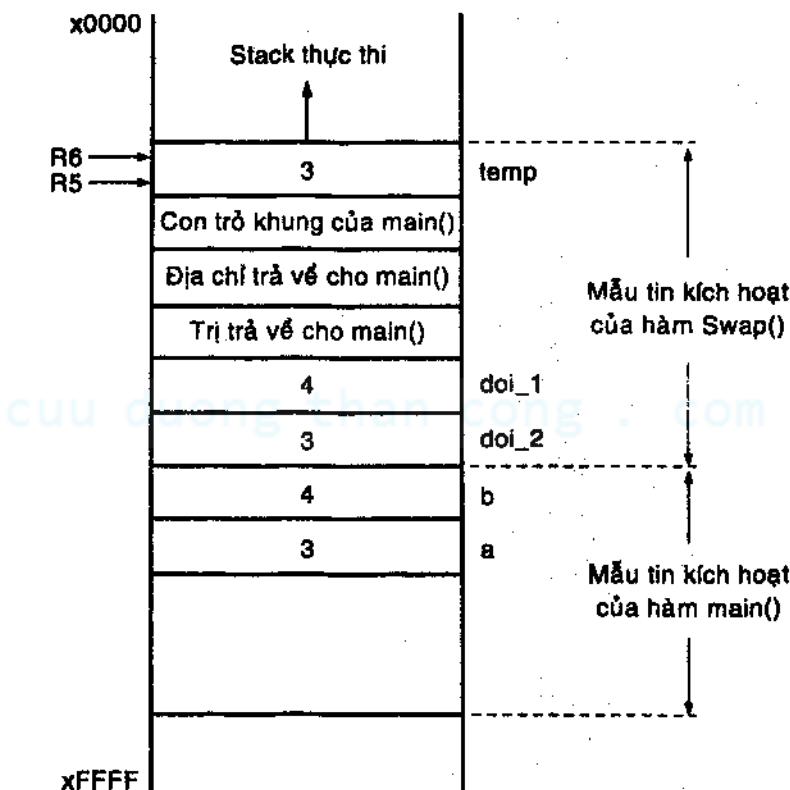
Sau khi lệnh ở dòng 24 được thực thi xong, Hình 13.2 cho thấy trạng thái của stack thực thi lúc đó. Lúc này hai biến *a* và *b* vẫn đang có trị là 3 và 4; còn hai biến cục bộ *doi_1* và *doi_2* có trị là 4 và 3 do hiệu ứng của hai dòng lệnh 23 và 24 trong hàm Swap().



Hình 13.1 Hình ảnh stack thực thi khi điều khiển chương trình đang ở dòng 22

Sau khi điều khiển được trả về cho hàm main(), hai trị *a* và *b* được in ra, và tất nhiên là hai trị 3 và 4 như đã biết.

Như vậy, vấn đề ở đây là cách truyền trị theo kiểu tham trị như vậy không thể thay đổi được trị như mong muốn. Làm sao giải quyết vấn đề này? Vấn đề này sẽ được giải quyết dễ dàng khi ta truyền tham số là địa chỉ của đối số thật, tức pointer. Đặc giả có thể xem lại ví dụ 12.7 với hình 12.2, đối số giả *values* thực chất là biến cục bộ trong hàm, nó chứa trị là địa chỉ của mảng, mà mảng này là đối số thực của nó.



Hình 13.2 Hình ảnh stack thực thi khi điều khiển chương trình đang ở dòng 25

13.2 THAO TÁC TRÊN POINTER

13.2.1 Khai báo biến pointer - pointer hằng

- Trong ngôn ngữ C có một toán tử lấy địa chỉ của một biến đang làm việc, toán tử này là một dấu & (ampersand), tạm gọi là toán tử lấy địa chỉ. Cú pháp như sau:

& biến

với **biến** là một biến thuộc kiểu bất kỳ, nhưng không được là biến thanh ghi.

Ví dụ 13.2 Nếu có một biến đã được khai báo là

```
int hेस्त_a;
```

thì

```
& hेस्त_a
```

sẽ là **địa chỉ** của biến *hेस्त_a*.

Có nghĩa là khi khai báo, biến *hेस्त_a* thì được cấp phát một vùng nhớ trong bộ nhớ máy tính. Địa chỉ đầu của vùng nhớ này chính là **địa chỉ** của biến *hेस्त_a*: `&hेस्त_a`.

Kết quả của phép toán lấy **địa chỉ** của một biến là một **pointer** **hằng** chỉ tới biến đó, địa chỉ hằng này có thể được xem như một giá trị để gán vào biến pointer.

Ví dụ 13.3

Nếu biến con trỏ *pint* là một biến pointer chỉ để lưu **địa chỉ** của đối tượng thuộc kiểu `int`, thì việc gán **địa chỉ** của một biến thuộc kiểu `int`, ví dụ biến *sô_trang*, vào cho biến *pint* là hoàn toàn hợp lệ:

```
pint = &sô_trang;
```

Dĩ nhiên ta không thể thực hiện việc gán sau:

```
pint = &(sô_trang + 2);
```

vì $(sô_trang + 2)$ không phải là một biến nên không có **địa chỉ** trong bộ nhớ.

Ngược lại, để khai báo một biến pointer để chứa **địa chỉ** của một đối tượng hoặc lấy một nội dung của đối tượng mà con trỏ đang chỉ tới, C dùng toán tử `"*"` (dấu hoa thị- asterisk). Cú pháp để khai báo biến pointer:

```
kiểu * tên_bien_pointer
```

với:

- **kiểu** có thể là kiểu bất kỳ, xác định kiểu dữ liệu có thể được ghi vào đối tượng mà con trỏ đang chỉ tới.
- **tên_bien_pointer** là tên của biến con trỏ, một danh hiệu hợp lệ.

Ví dụ 13.4 Xét khai báo

```
int *pint;
```

thì *pint* là một biến pointer, sẽ chỉ đến một đối tượng thuộc kiểu int. Biến này có chiều dài từ hai đến bốn byte (tùy biến con trỏ là gần hay xa).

Tuy nhiên, ngay sau khi khai báo biến *pint*, trị trong các ô nhớ máy tính cung cấp cho *pint* là "trị rác", do đó đối tượng mà *pint* đang chỉ tới là "đối tượng rác", đối tượng này không thể được lấy sử dụng để tính toán hoặc gán trị trong chương trình.

Biến hoặc đối tượng mà con trỏ đang chỉ tới có thể được truy xuất qua tên của biến con trỏ và dấu "*" đi ngay trước biến con trỏ, cú pháp cụ thể như sau:

*** tên_bien_con_tró**

Đây là một biến bình thường, nếu biến này được lấy tính toán trong một biểu thức nào đó, thì kiểu của biến này sẽ là kiểu có được khi khai báo biến pointer. Và vì vậy, ta hoàn toàn có thể thực hiện mọi thao tác trên biến.

Ví dụ 13.5 Xét ví dụ sau:

```
int object;
int *pint;

object = 5;
pint = &object;
```

Mã LC-3 cho đoạn code trên như sau:

```
AND R0, R0 ; xóa R0
ADD R0, R0, #5 ; R0 = 5
```

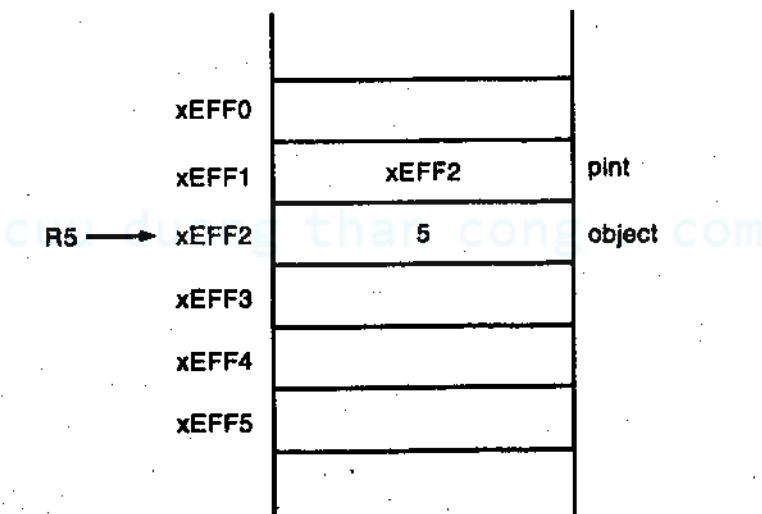
```

STR R0, R5, #0 ; object = 5
ADD R0, R5, #0 ; R0 = R5 + 0; R0 chứa địa chỉ của biến object
STR R0, R5, #-1 ; R5 - 1: địa chỉ của biến pint, pint <- R0

```

Hình 13.3 cho thấy một bản chụp của stack thực thi ngay sau khi lệnh `pint = &object;` vừa được thực thi. Lưu ý, giả sử có hai biến cục bộ trong hàm đang xét, thì thanh ghi R5 đang chỉ tới biến `object`.

Với các khai báo biến trong C như trên, biến `object` sẽ chứa trị `int`, còn biến `pint` sẽ chứa trị là địa chỉ của một đối tượng `int`. Và hai lệnh kế tiếp cho ta dữ liệu như trong hình 13.3, `object` chứa trị 5, còn `pint` chứa trị `xEFF2`, tức là địa chỉ của biến `object`.



Hình 13.3 Trạng thái stack thực thi chứa `pint` và `object` sau khi lệnh `pint = &object;` vừa được thực thi

Nếu ta phát triển ví dụ trên, bằng cách thêm dòng lệnh để tăng trị cho biến `object` lên 1, ta có thể viết theo một trong hai dạng như sau:

```

int object;
int *pint;
object = 5;
pint = &object;
object = object + 1;

```

hoặc

```
int object;
int *pint;
object = 5;
pint = &object;
*pint = *pint + 1;
```

Mã LC-3 tương ứng cho chúng sẽ khác nhau cho dòng cuối này:

```
; với object = object + 1;
LDR R0, R5, #0      ; R0 <- object
ADD R0, R0, #1
STR R0, R5, #0      ; R0 > object
```

và

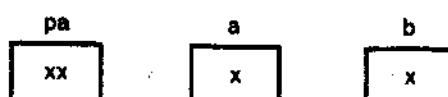
```
; với *pint = *pint + 1;
LDR R0, R5, #-1     ; R0 <- pint
LDR R1, R0, #0       ; R1 <- *pint, tức R1 <- object
ADD R1, R1, #1       ; tăng R1
STR R1, R0, #0       ; R1 > *pint, tức R1 > object
```

Ta có thể thấy, tùy vào mã cấp cao được viết, mà bộ dịch sẽ cho ra các mã cấp thấp khác nhau, và điều này sẽ ảnh hưởng tới thời gian thực thi, bộ nhớ sử dụng và chiều dài mã của chương trình thực thi.

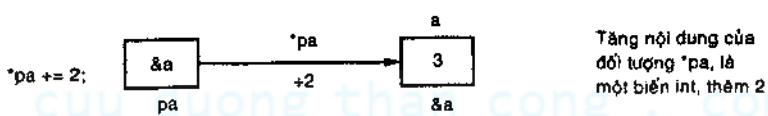
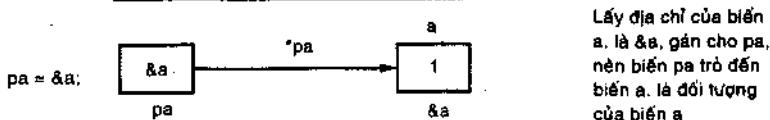
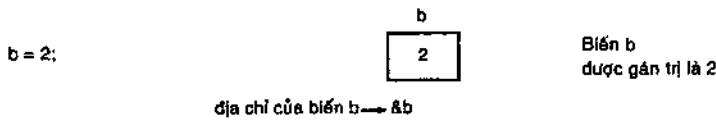
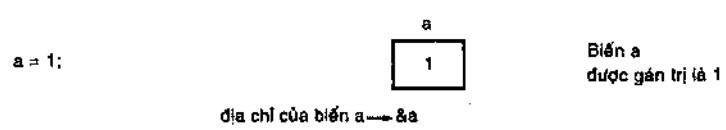
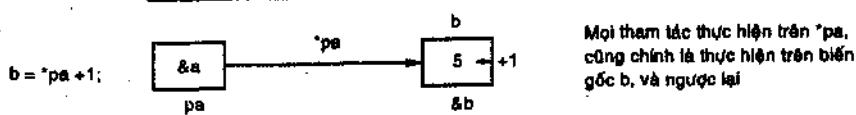
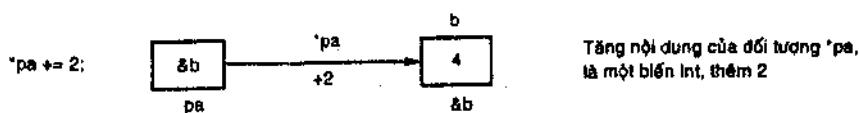
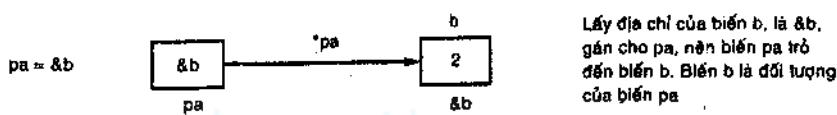
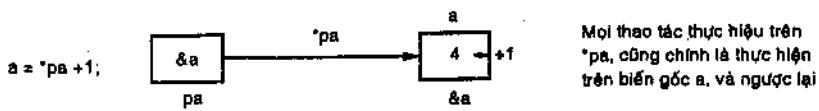
Ví dụ 13.6 Xét các khai báo sau:

```
int a, b;      /* Khai báo biến a, b thuộc kiểu int */
int *pa;       /* Biến con trỏ pa sẽ chỉ tới một đối tượng int */
```

Sau khi khai báo, ta có ba ô nhớ cho ba biến *a*, *b* và *pa* như sau:



Lúc này, trong cả ba biến, đều đang có trị rác. Nếu thực hiện các phép toán sau đây thì sự thay đổi trị trong mỗi ô nhớ của mỗi biến có thể được biểu diễn qua các hình 13.4 như sau.

**Hình 13.4 Một số thao tác (a)****Hình 13.4 Một số thao tác (b)**

Như vậy ta thấy rằng, biến pointer không có kiểu riêng cho nó, nên nó phải được khai báo qua kiểu của đối tượng mà nó sẽ chỉ tới. Do đó, trước khi quyết định khai báo biến pointer, ta cần phải xác định đối tượng của nó thuộc kiểu gì. Nếu chưa xác định được kiểu của đối tượng cần khai báo, C cho phép khai báo một biến pointer chỉ tới đối tượng thuộc kiểu void (không kiểu) như Ví dụ 13.7 sau.

Ví dụ 13.7

```
void * pvoid;
```

Với khai báo trên, biến *pvoid* đã được xác định danh hiệu trong chương trình, biến này được C dành ra 2 hoặc 4 byte trong bộ nhớ để lưu địa chỉ của đối tượng mà nó sẽ chỉ tới. Tuy nhiên, đối tượng mà *pvoid* chỉ tới lại là kiểu không kiểu, nên địa chỉ lưu trong biến *pvoid* chỉ đơn giản có ý nghĩa là một địa chỉ bình thường, muốn sử dụng biến này để truy xuất dữ liệu từ địa chỉ lưu trong nó, ta phải thực hiện sự ép kiểu bắt buộc như trong ví dụ sau:

Ví dụ 13.8

```
void * pvoid;
int a, * pint;
double b, * pdouble;
pvoid = (void *) &a;
pint = (int *) pvoid;
(*pint)++;
pvoid = (void *) &b;
pdouble = (double *) pvoid;
(*pdouble) --;
```

Trong ví dụ trên, địa chỉ của biến *a* (*&a*) là địa chỉ của một int, địa chỉ này muốn gán vào cho biến *pvoid* thì cần phải được ép kiểu sang kiểu của biến mà địa chỉ này được gán vào, đó là con chỉ tới void. Như vậy sau khi thực hiện lệnh gán

```
pvoid =(void *) &a;
```

biến *pvoid* có lưu một địa chỉ, đây chỉ là một trị giá địa chỉ bình thường, không có ý nghĩa trong việc truy xuất dữ liệu. Địa chỉ này có thể được

sử dụng để nạp lại trị vào một biến con trỏ khác, dĩ nhiên việc ép kiểu cho phù hợp là điều cần thiết

```
pint = (int *) pvoid;
```

Sau khi gán xong, biến con trỏ mới này có thể được sử dụng để truy xuất dữ liệu, thực hiện thao tác

```
(*pint) ++;
```

Biến con trỏ *pvoid* không chỉ tới kiểu dữ liệu xác định nào cả, nên có thể sử dụng nó để lưu địa chỉ như một trị cần giữ. Như vậy địa chỉ gán vào cho biến *pvoid* có thể thuộc kiểu bất kỳ, miễn là khi gán có sự ép kiểu cho phù hợp. Do đó, trong ví dụ trên, biến *pvoid* lúc đầu lưu địa chỉ của biến *a* (*&a*), sau đó lại lưu địa chỉ của biến *b* (*&b*); điều này là hoàn toàn hợp lệ.

Như vậy, bằng các phép lấy địa chỉ của một biến và lấy đối tượng của một pointer, ta có thể truy xuất gián tiếp đến một biến nào đó, mà không cần sử dụng đến tên biến.

13.2.2 Các phép toán trên pointer

Khi đưa ra khái niệm con trỏ, C cho phép thực hiện các phép toán trên các con trỏ nhằm tăng cường khả năng sử dụng nó. Các phép toán này có thể được tóm tắt như sau:

- Có thể cộng, trừ một pointer với một số nguyên (int, long,...). Kết quả là một pointer. Pointer có được từ phép cộng hoặc trừ trên, sẽ chỉ đến một đối tượng mới lệch với đối tượng cũ n phần tử, nếu tính theo byte thì đối tượng cũ lệch với đối tượng mới số byte bằng n lần kích thước byte của kiểu đối tượng mà con trỏ đang chỉ tới.

Ví dụ 13.9

```
int *pi1, *pi2, n;
pi1 = &n;
pi2 = pi1 + 3;
```

Trong ví dụ trên, biến con trỏ *pi1* sau khi được gán trị *&n*, sẽ chỉ đến biến *n*, biến *pi2* được gán trị từ *pi1* cộng thêm 3, điều này có

ý nghĩa là biến *pi2* chỉ tới một đối tượng mới, lệch với biến *n* ba phần tử int, tức $3 \times 2 = 6$ byte. Tuy nhiên, trong thực tế, đối tượng mà biến *pi2* chỉ tới có thể không có ý nghĩa sử dụng trong chương trình, nếu việc cộng và trừ này không mang một ý nghĩa nào đó.

Khi lập trình, thao tác cộng và trừ này có thể được sử dụng để truy xuất mảng vì mảng là một cấu trúc trong đó các phần tử được sắp xếp liên tiếp nhau trong bộ nhớ. Do đó, nếu một biến con trả được gán trị là địa chỉ của một biến thành phần nào đó của mảng, thì việc cộng hay trừ biến này với một số nguyên sẽ cho ra địa chỉ của biến thành phần mới, lệch với biến thành phần cũ n phần tử.

Ví dụ 13.10 Cho khai báo

```
int a[20];           /* Mảng a có 20 phần tử int liên tiếp */
int *p;              /* Biến pointer chỉ đến một đối tượng int */
p = &a[0];            /* p là địa chỉ của phần tử đầu tiên của mảng a */
p += 3;              /* p lưu địa chỉ phần tử a[0 + 3], tức &a[3] */
```

Do đó, ta có thể dùng biến con trả để truy xuất mảng, địa chỉ trong biến con trả lần lượt sẽ là địa chỉ của từng phần tử của mảng.

Ví dụ 13.11

```
#include <stdio.h>
#include <conio.h>
main()
{
    int n, *p1, *p2, i;
    int a[20];
    clrscr();
    printf ("Mang co bao nhieu phan tu:");
    scanf ("%d", &n);
    p1 = p2 = &a[0];
    printf("Nhap cac phan tu cua mang: \n");
    for (i = 0; i < n; i++)
    {
        scanf("%d", p1);
        p1++;
    }
}
```

```

printf ("Cac tri trong mang la: ");
for (i =0; i <n; i++)
{
    printf("%d", *p2);
    p2++;
}
getch();
}

```

Chương trình cho xuất liệu ví dụ:

```

Mang co bao nhieu phan tu: 5
Nhap cac phan tu cua mang: 5 7 9 0 5
Cac tri trong mang la: 5 7 9 0 5

```

Trong chương trình ví dụ trên, ta dùng hai biến con trỏ để lưu địa chỉ đầu của mảng *p1* và *p2*. Biến *p1* được sử dụng trong vòng lặp nhập trị cho các phần tử của mảng, biến *p2* lại được sử dụng trong vòng lặp truy xuất mảng, vì sau các thao tác cộng trị, biến con trỏ đã bị thay đổi trị.

- Không thể thực hiện các phép toán nhân, chia, hoặc lấy dư một pointer với một số, vì pointer lưu địa chỉ, nên nếu thực hiện được điều này cũng không có một ý nghĩa nào cả.
- Phép trừ giữa hai pointer vẫn là một phép toán hợp lệ, kết quả là một trị thuộc kiểu int biểu thị khoảng cách (số phần tử) giữa hai pointer đó.

Ví dụ 13.12

Xét chương trình ví dụ sau:

```

#include <stdio.h>
#include <conio.h>
main()
{
    int *p1, *p2;
    int a[10];

    clrscr();
    p1 = &a[0];

```

```

p2 = &a[5];
printf ("Dia chi cua bien a[0] la: %p\n", p1);
printf ("Dia chi cua bien a[5] la: %p\n", p2);
printf ("Khoang cach giua hai phan tu la %d int\n", p2 - p1);
getch();
}

```

Chương trình sẽ cho xuất liệu ví dụ:

```

Dia chi cua bien a[0] la: FFE2
Dia chi cua bien a[5] la: FFEC
Khoang cach giua hai phan tu la 5 int

```

Trong chương trình trên, ta sử dụng một định dạng xuất cho các trị con trỏ là $p1$ và $p2$, để in ra màn hình giá trị mà chúng đang giữ, đó là chuỗi $\%p$. Hiệu ($p2 - p1$) sẽ là trị lệch tính theo số phần tử int mà ta đã gán địa chỉ vào cho hai biến $p1$ và $p2$.

- Không thể thực hiện việc cộng hai pointer lại với nhau.
- Ngoài ra, biến pointer có thể được xử lý như một biến bình thường, có nghĩa là ta có thể thực hiện các phép toán so sánh, gán, tăng, giảm Các phép toán này đòi hỏi các pointer là toán hạng phải có cùng kiểu. Nếu việc thực hiện phép toán trên con trỏ là không rõ ràng, bộ dịch sẽ nhắc nhở ngay.

Ví dụ 13.13 Cho các khai báo sau:

```

int * a1;
char * a2;

a1 = 0;           /* Chương trình dịch sẽ nhắc nhở lệnh này */
a2 = (char *)0;
if (a1 != a2)    /* Chương trình dịch sẽ nhắc nhở kiểu của đối tượng */
{
    a1 = (int *) a2; /* Hợp lệ vì đã ép kiểu */
}

```

- Có thể áp dụng phép toán chuyển kiểu bắt buộc để chuyển kiểu một pointer, chuyển một pointer đang chỉ đến một đối tượng thuộc kiểu nào đó thành pointer chỉ đến một đối tượng thuộc kiểu khác, cả hai đối tượng đều cùng địa chỉ. Phép

chuyển kiểu này thường được dùng để đổi kiểu của một pointer thành một kiểu pointer khác để có thể lấy đối tượng của nó với kiểu khác mà ta mong muốn.

Ví dụ 13.14

```
#include <stdio.h>
#include <conio.h>
main()
{
    int *pint, a = 0x6141;
    char *pchar;

    clrscr();
    pint = &a;
    pchar = (char *) &a;
    printf ("Tri cua bien pint la: %p\n", pint);
    printf ("Tri cua bien pchar la: %p\n", pchar);
    printf ("Doi tuong pint dang quan ly la %X \n", *pint);
    printf ("Doi tuong pchar dang quan ly la %c \n", *pchar);
    pchar++;
    printf ("Doi tuong pchar dang quan ly la %c \n", *pchar);
    getch();
}
```

Chương trình cho xuất liệu ví dụ:

```
Tri cua bien pint la: FFF4
Tri cua bien pchar la: FFF4
Doi tuong pint dang quan ly la 6141
Doi tuong pchar dang quan ly la A
Doi tuong pchar dang quan ly la a
```

Trong chương trình trên, cả hai biến con trỏ đều cùng lưu một địa chỉ, nhưng đối tượng mà chúng quản lý khác nhau. Biến *pint* quản lý đối tượng là một int tại địa chỉ đó, trong khi biến *pchar* lưu cùng một địa chỉ như *pint* nhưng lại quản lý đối tượng theo kiểu char. Chính vì vậy mà việc in đối tượng của *pint* sẽ cho ta một int, trong khi cũng cùng địa chỉ đó, việc in đối tượng của *pchar* lại cho các ký tự 'A' và 'a'.

- C cho phép khai báo một biến pointer là hằng hoặc đối tượng của một pointer là hằng. Lúc đó, việc gán, hoặc tăng giảm trị lưu trong pointer hằng là không hợp lệ, hoặc thay đổi đối tượng của pointer khi pointer được khai báo là chỉ đến một đối tượng hằng đều bị C báo lỗi.

Ví dụ 13.15

Các khai báo sau đây là biến pointer hằng:

1. int a, b;

int * const pint = &a;



biến pointer pint chỉ lưu địa chỉ của biến a, và
được khởi động trị lúc mới khai báo mà thôi

pint = &b;

2. char * const ps = "ABCD";



ps là một pointer hằng chỉ đến chuỗi ký tự

ps là một pointer hằng chỉ đến một chuỗi ký tự

Các khai báo sau đây cho thấy đối tượng của một pointer là hằng:

1. int i;

const int * pint;



khai báo này cho thấy pint sau này sẽ là
một đối tượng hằng

pint = &i;

i = 1;

(*pint) ++;

lệnh này sẽ báo lỗi, vì không thể thay đổi
trị của một đối tượng hằng của con trỏ

i++; → hoàn toàn hợp lệ

2. const char * s = "ABCD" hoặc
char const * s = "ABCD";



s là một biến pointer chỉ đến một chuỗi ký tự hằng

Như vậy, việc đặt từ **const** trước hay sau dấu * trong khai báo pointer có ý nghĩa khác nhau:

* **const** trước *: pointer chỉ đến đối tượng hằng

* trước **const**: pointer hằng

Các khai báo biến pointer là một **const** hoặc pointer chỉ đến đối tượng là **const** thường được dùng để kiểm soát trị, vì đôi khi lập trình viên không muốn thay đổi đối tượng đang được chỉ tới hoặc thay đổi bản thân đối tượng. Dùng cách khai báo này trong khai báo tham số của mảng, lập trình viên sẽ không cho phép mọi sự thay đổi mảng hay chuỗi trong hàm, nói cách khác, đây là hình thức truyền mảng hay chuỗi vào cho hàm dưới dạng tham số trị.

13.3 POINTER VÀ MẢNG

Trong C, *tên mảng là một hằng pointer tới phần tử có kiểu là kiểu của biến thành phần dưới nó một bậc trong mảng*, ví dụ tên của mảng một chiều của các int là pointer chỉ tới int, tên của mảng hai chiều của các int là pointer chỉ tới mảng một chiều là hàng các int trong mảng. Trong trường hợp mảng một chiều, tên mảng chính là địa chỉ của phần tử đầu tiên của mảng. Do đó, ta hoàn toàn có thể truy xuất mảng bằng một pointer.

Ta có thể lấy địa chỉ của một mảng bằng toán tử "&" và khai báo biến pointer chỉ tới mảng một chiều như trong ví dụ sau.

Ví dụ 13.16

```
int a[3];
```

```
int (*p)[3];
```

```
p = &a;
```

Với lệnh này, con trỏ *p* chỉ có thể được gán trị từ một địa chỉ của một mảng gồm 3 phần tử int mà thôi. Khi thao tác với con trỏ *p* này, ta lưu ý là nó luôn quản lý từng 3 phần tử int một. Tất nhiên, hằng pointer *&a* cũng là pointer quản lý từng 3 phần tử int một theo như khai báo, trong khi đó bản thân tên mảng *a* lại là hằng pointer chỉ tới int đơn. Ta hãy xem ví dụ sau:

Ví dụ 13.17

Cho khai báo mảng một chiều và biến con trỏ sau:

```
int a[10];
int *pa;
```

thì *a* là một hằng pointer, do đó ta có thể dùng tên mảng là pointer để truy xuất mảng, đối tượng

```
*(a + 0) chính là a[0],
*(a + 1) chính là a[1],
...
*(a + i) chính là a[i].
```

Mặt khác, *a* là một hằng pointer nên ta hoàn toàn có khả năng gán trị này vào cho biến *pa*, hoặc địa chỉ của phần tử đầu tiên *&a[0]*, để *pa* chỉ tới mảng

```
pa = a;
hoặc pa = &a[0];
```

Khi đó,

```
(pa + 0) sẽ chỉ đến phần tử a[0],
(pa + 1) sẽ chỉ đến phần tử a[1],
...
(pa + i) sẽ chỉ đến phần tử a[i].
```

Như vậy, các đối tượng

```
*(pa + 0) chính là *(a + 0), cũng là a[0]
*(pa + 1) chính là *(a + 1), cũng là a[1]
...
*(pa + i) chính là *(a + i), cũng là a[i]
```

Chính vì điều này mà ta có thể dùng *a* hoặc *pa* để truy xuất mảng *a*. Nếu biến con trỏ *pa* đã chỉ tới đầu mảng và kiểu đối tượng mà con trỏ đang quản lý là cùng kiểu với các phần tử mảng thì C cho phép viết một cách ngắn gọn hơn:

pa[0] chính là phần tử a[0]

pa[1] chính là phần tử a[1]

...

pa[i] chính là phần tử a[i].

Ví dụ 13.18 Xét chương trình sau:

```
#include <stdio.h>
#include <conio.h>
main()
{
    int *pa, a[10];
    int i, s = 0;

    clrscr();
    pa = a;
    printf ("Moi nhap 10 phan tu can tinh tong \n");
    for (i = 0; i < 10; i++)
        scanf ("%d", pa + i);
    for (i = 0; i < 10; i++)
        s += pa[i];
    printf ("Tong cac phan tu cua mang la: %d \n", s);
    getch();
}
```

Chương trình cho xuất liệu ví dụ:

Moi nhap 10 phan tu can tinh tong

2 3 5 9 0 5 7 3 7 5

Tong cac phan tu cua mang la: 46

Tuy nhiên, cũng cần có sự phân biệt rõ ràng giữa mảng và pointer.

- Một mảng, sau khi được khai báo và định nghĩa, đã được cấp một vùng nhớ trong bộ nhớ trong của máy tính, vùng nhớ này có kích thước tùy thuộc vào kích thước của mảng, và địa chỉ chính là tên mảng.

Ví dụ 13.19

Sau khi khai báo

```
int a[10];
```

thì trong bộ nhớ, a là một hằng pointer, hay một địa chỉ cố định



Các phần tử trong mảng đang có trị rác.

Vì tên mảng là một hằng pointer, ta không thể thay đổi trị của nó được nên việc gán hay thay đổi trị

```
a++;
```

là không hợp lệ.

- Còn một biến pointer, sau khi được khai báo, thì vùng nhớ được cấp chỉ là bản thân biến pointer, còn trị bên trong nó là một địa chỉ rác, tức biến con trỏ chưa chỉ tới một đối tượng cụ thể để sử dụng.

Ví dụ 13.20 Sau khi khai báo

```
int *pa;
```

thì trong bộ nhớ



do đó lệnh gán

```
*pa = 2;
```

là không có ý nghĩa, dù C không báo lỗi trong trường hợp này.

Như vậy, nếu sau khi gán địa chỉ của mảng (tên mảng) vào cho biến pointer, để biến pointer chỉ tới mảng thì việc truy xuất mảng qua biến pointer mới có thể diễn ra bình thường. Và ta hoàn toàn có thể thay trị bên trong nó để nó chỉ tới một đối tượng mới.

- Ngoài ra, biến pointer có một địa chỉ trong bộ nhớ, còn không thể lấy địa chỉ của tên mảng.

Ví dụ 13.21

```
#include <stdio.h>
#include <conio.h>
main()
{
    int *pint, a;

    clrscr();
    pint = &a;
    printf ("Dia chi cua bien pint la: %p\n", &pint);
    printf ("Dia cua bien a, pint dang tro toi la: %p\n", pint);
    getch();
}
```

Chương trình sẽ cho xuất liệu ví dụ:

```
Dia chi cua bien pint la: FFF4
Dia cua bien a, pint dang tro toi la: FFF2
```

Qua các trình bày trên, ta thấy tên mảng là một hằng pointer chỉ tới đầu mảng, nếu mảng là đối số của hàm thì trong danh sách khai báo đối số của hàm ta có thể khai báo đối số giả dưới dạng mảng như sau:

```
int a[];
```

hoặc có thể khai báo dưới dạng pointer như:

```
int *a;
```

Trong cả hai trường hợp, đối số thật đưa vào cho hàm đều là địa chỉ của mảng, tức tên mảng, hoặc địa chỉ của phần tử đầu tiên của mảng. Do đó, nếu không quy định mảng là **const**, thì trong hàm, các lệnh hoàn toàn có khả năng thay đổi trị của các phần tử của mảng gốc. Hãy xem ví dụ hàm nhập trị cho mảng và chương trình sử dụng hàm này.

Ví dụ 13.22

```
void nhap_tru_mang (int a[], int n)
{
    int i;
    for (i=0; i <n; i++)
        scanf ("%d", &a[i]);
}
```

↑
đối số giả dưới dạng mảng

Hàm này có thể được viết lại như sau:

```
void nhap_tri_mang (int *a, int n)
{
    int i;
    for (i = 0; i < n; i++)
        scanf ("%d", &a[i]);
}
```

↑
đối số giả dưới dạng pointer

Chương trình sử dụng:

```
#include <stdio.h>
#include <conio.h>
void nhap_tri_mang (int *a, int n);
main()
{
    int a[10], s =0, i;
    clrscr();
    printf ("Moi nhap 10 tri vao mang: ");
    nhap_tri_mang (a, 10); /* gọi sử dụng mảng, truyền theo địa chỉ */
    for (i = 0; i < 10; i++)
        s += a[i];
    printf ("Tong cac phan tu cua mang la: %d \n", s);
    getch();
}
void nhap_tri_mang (int *a, int n)
{
    int i;
    for (i = 0; i < n; i++)
        scanf ("%d", a + i);
}
```

Chương trình cho xuất liệu ví dụ:

```
Moi nhap 10 tri vao mang: 1 2 3 4 5 6 7 8 9 0
Tong cac phan tu cua mang la: 45
```

Trong chương trình trên, ta có thể sử dụng hàm nhap_tri_mang() có đối số khai báo là pointer, hoặc đối số khai báo là mảng đều được.

Từ mảng hai chiều trở đi, việc dùng biến pointer để truy xuất mảng là khá phức tạp, chúng ta cần phải luôn nhớ là các thao tác trên pointer luôn diễn ra trên cùng một bậc quản lý đối tượng, nghĩa là chúng ta phải luôn biết pointer mà chúng ta sử dụng đang quản lý đối tượng kiểu nào. Để dễ hình dung, ta hãy xét mảng hai chiều với hai danh hiệu đã khai báo thay thế qua **#define** là MAX_ROW và MAX_COL, ta có ví dụ sau.

Ví dụ 13.23

```
#define MAX_ROW 20
#define MAX_COL 30
int array[MAX_ROW][MAX_COL];
int row, col; /* hai biến cho chỉ số hàng và chỉ số cột */
int (*parr) [MAX_ROW][MAX_COL]; /* biến con trỏ mảng hai chiều */
parr = &array; /* gán trị cho biến pointer mảng hai chiều */
```

Với khai báo trên, danh hiệu array là hằng pointer hai lần pointer chỉ tới phần tử đầu tiên của mảng array, tức ta có ****array** chính là **array[0][0]**. Với mảng hai chiều, ta có **array[0]** (= ***array** hay ***(array + 0)**) là con trỏ chỉ tới hàng 0 trong mảng (và dĩ nhiên **array[row]** là con trỏ chỉ tới hàng **row** trong mảng, ...). Như vậy, ta có ***array[0]** chính là ****array** và cũng chính là **array[0][0]**.

Theo quy định mảng một chiều, **array[row]** chính là ***(array+row)**, tức

$$\text{array}[row] \equiv \text{*(array} + \text{row}\text{)}$$

phần tử cột **col** trong hàng **row** của ma trận sẽ là

$$\text{array}[row][col] \equiv \text{*(array}[row]\text{ + col}), \text{tức}$$

$$\text{array}[row][col] \equiv \text{*(int } (\text{*}) \text{ array} + \text{row} \text{* MAX_COL} + \text{col})$$

$$\text{array}[row][col] \equiv \text{*(} (\text{*array} + \text{row}) + \text{col})$$

hay $\text{array}[row][col] \equiv *(*(\text{int } (*)[\text{MAX_COL}]) \text{ parr} + row) + col$

Ở dòng trên, con trỏ *parr* vốn chỉ tới đối tượng là mảng *MAX_ROW* x *MAX_COL* phần tử, ta cần phải chuyển nó về pointer chỉ tới hàng có *MAX_COL* phần tử, nên phép chuyển kiểu

(int ()[MAX_COL]) parr*

được sử dụng. Thao tác cộng thêm *row* để xác định địa chỉ hàng cần làm việc. Thao tác kế tiếp

**((int (*)[MAX_COL]) parr + row)*

xác định địa chỉ của phần tử đầu tiên trên hàng (là mảng một chiều) cần làm việc. Kế tiếp, thao tác cộng thêm *col* xác định địa chỉ phần tử hàng *row*, cột *col*. Sau cùng, thao tác

**(*((int (*)[MAX_COL]) parr + row) + col)*

xác định đối tượng là phần tử trên cột *col* và hàng *row* đó.

Như vậy, ta có thể thấy việc sử dụng biến pointer để truy xuất mảng là rất phức tạp, đặc biệt là từ mảng hai chiều trở đi. Trong thực tế, việc sử dụng pointer để truy xuất mảng chỉ được dùng cho mảng một chiều mà thôi.

Để hiểu rõ về địa chỉ của mảng cũng như cách quản lý các phần tử của mảng qua pointer, xin mời độc giả tham khảo ví dụ 13.24 sau đây.

Ví dụ 13.24

```
#include <stdio.h>
#define MAX_ROW 3
#define MAX_COL 3

main()
{
    int a;
    int a1d [MAX_COL] = {0, 1, 2};
    int *pint1;
    int (*pa1d)[MAX_COL];
```

```
int a2d [MAX_ROW][MAX_COL] = { {0, 1, 2},  
                                {10, 11, 12},  
                                {20, 21, 22}  
};  
  
int **pint2;  
int (*pa2d)[MAX_ROW][MAX_COL];  
int b;  
  
/* Thu dia chi cua pointer va mang 1 chieu */  
pint1 = &a;  
printf ("pint1 = &a = %p\n", pint1);  
  
pint1 = &b;  
printf ("pint1 = &b = %p\n", pint1);  
  
pint1 = a1d;  
printf ("pint1 = a1d = %p\n", pint1);  
  
pa1d = &a1d;  
printf ("pa1d = &a1d = %p\n", pa1d);  
  
printf ("a1d [2] = %d\n", pint1[2]);  
printf ("(*pa1d)[2] = %d\n", (*pa1d)[2]);  
  
/* Thu dia chi cua pointer va mang 2 chieu */  
  
pint1 = a2d[1];  
printf ("pint1 = a2d[1] = %p\n", pint1);  
pa2d = &a2d;  
  
printf ("*( *(a2d + 1) + 2) = %d\n", *( *(a2d + 1) + 2));  
printf ("*pint1[2] = %d\n", pint1[2]);  
printf ("(*pa2d)[1][2] = %d\n", (*pa2d)[1][2]);  
  
getchar();  
}
```

Chương trình này xuất liệu ví dụ như sau:

```

E:\BC5\B\N\thu pointer Z.exe
pint1 = &a = 0012FF88
pint1 = &b = 0012FF84
pint1 = &id = 0012FF78
pa1d = &a1d = 0012FF78
a1d [2] = 2
(*pa1d)[2] = 2
mint1 = a2d[1] = 0012FF60
*(a2d + 1) + 2 = 12
*pint1[2] = 12
(*pa2d)[1][2] = 12

```

Độc giả có thể dễ dàng giải thích sự xuất hiện các giá trị trên. Từ xuất liệu trên ta thấy:

- C luôn tối ưu việc cấp phát biến trong stack thực thi, như hai biến *a* và *b* dù được khai báo xa nhau, nhưng do cùng là biến vô hướng nên được cấp phát bộ nhớ gần nhau.
- Một biến int trong C lúc này dài 4 byte, nên địa chỉ của chúng cách nhau 4 đơn vị.
- Việc sử dụng biến pointer để truy xuất mảng là khá phức tạp, tuy nhiên nó sẽ làm chương trình mềm dẻo hơn trong việc sử dụng và quản lý dữ liệu khi lượng dữ liệu là nhiều.

13.4 ĐỔI SỐ CỦA HÀM LÀ POINTER - TRUYỀN ĐỔI SỐ THEO SỐ DẠNG THAM SỐ BIẾN

Bình thường, đối số của hàm được khai báo ở dạng tham số trị, khi đó hàm sẽ lấy giá trị của các đối số thật gởi cho hàm khi gọi hàm để tính toán trong hàm, mọi sự thay đổi trị của đối số thật trong hàm đều không thể thực hiện được, mà điều này trong nhiều trường hợp lại cần thiết.

Ta hãy xem lại Ví dụ 13.1 về hàm hoán đổi trị giữa hai biến và xem kết quả sau khi hàm này được gọi.

Ví dụ 13.25

```

#include <stdio.h>
#include <conio.h>

void Swap (int doi_1, int doi_2);
main()

```

```

{
    int a = 3, b = 4;

    clrscr();
    printf ("Tri cua hai bien a va b la: %d %d\n", a, b);
    Swap (a, b);
    printf("Sau khi goi ham Swap, tri cua a va b la: %d %d\n", a, b);
    getch();
}

void Swap (int doi_1, int doi_2)
{
    int temp;

    temp = doi_1;
    doi_1 = doi_2;
    doi_2 = temp;
}

```

Chương trình sẽ cho xuất liệu ví dụ:

Tri cua hai bien a va b la: 3 4

Sau khi goi ham Swap, tri cua a va b la: 3 4

Chương trình này không cho kết quả mong muốn, vì ta muốn sau khi gọi hàm, hai trị trong hai biến *a* và *b* sẽ được thay đổi cho nhau.

Để thực hiện được điều này, ta cần phải truyền địa chỉ của đối số thật vào cho hàm, khi đó việc khai báo danh sách đối số trong hàm ứng với đối số được truyền theo địa chỉ sẽ là pointer. Khi đó mọi sự thay đổi trên đối tượng mà con trỏ đang chỉ tới trong hàm chính là thay đổi trên đối số thật.

Do đó, hàm Swap () trên có thể được viết lại như sau:

```

void Swap (int *doi_1, int *doi_2)
{
    int temp;

    temp = * doi_1;
    * doi_1 = * doi_2;
    * doi_2 = temp;
}

```

*pointer là đối số của hàm, để nhận
địa chỉ của đối số thật*

Khi đó chương trình hoán đổi trị của hai biến có thể được viết lại như sau:

```

1   #include <stdio.h>
2   #include <conio.h>
3
4   void Swap (int *doi_1, int *doi_2);
5   main ()
6   {
7       int a = 3, b = 4;
8
9       clrscr();
10      printf ("Tri cua hai bien a va b la: %d %d\n", a, b);
11      Swap (&a, &b);
12      printf ("Sau khi goi ham Swap, tri cua a va b la: %d %d\n", a, b);
13      getch();
14  }
15  void Swap (int *doi_1, int *doi_2)
16  {
17      int temp;
18
19      temp = * doi_1;
20      * doi_1 = * doi_2;
21      * doi_2 = temp;
22  }

```

Chương trình sẽ cho xuất liệu ví dụ:

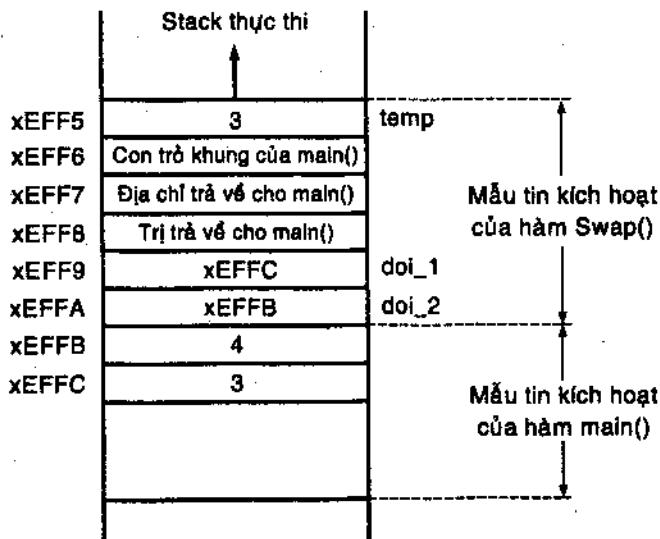
Tri cua hai bien a và b la: 3 4

Sau khi goi ham Swap, tri cua a va b la: 4 3

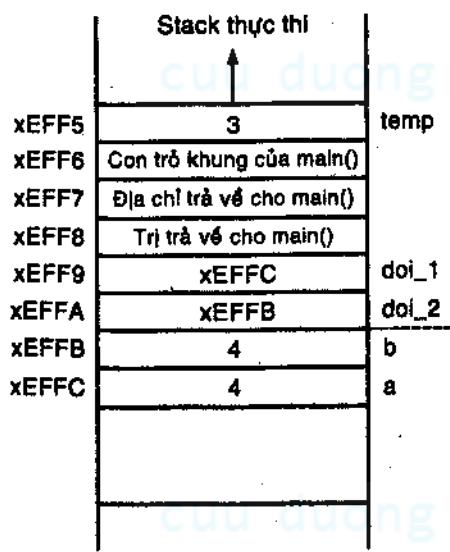
Chương trình chính trong hàm main() sẽ gọi sử dụng hàm Swap(), các đối số thật truyền cho hàm này sẽ là hai địa chỉ của hai biến *a* và *b*, *&a* và *&b*, do đó hàm gọi sẽ là

Swap (&a, &b);

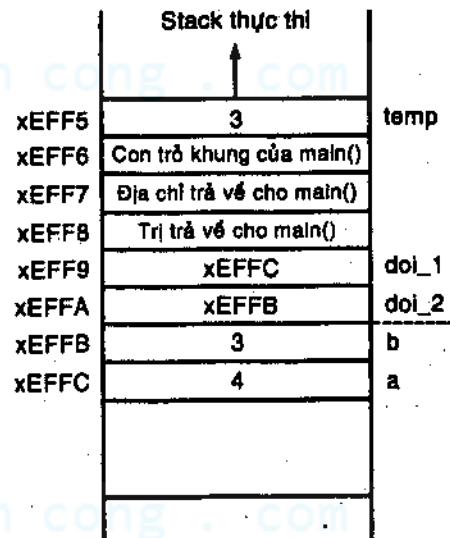
cách truyền theo địa chỉ này thật ra chính là hình thức truyền đối số hàm theo tham số biến, sau khi gọi hàm, ta có thể nhận được trị trả về từ hàm qua các đối số được truyền theo địa chỉ.



a)



b)



c)

Hình 13.5 Các trạng thái của stack thực thi ngay sau:

(a) lệnh 19, (b) lệnh 20, (c) lệnh 21

Hình 13.5 cho thấy rõ hơn cấu trúc bộ nhớ khi chương trình được dịch sang LC-3. Chú ý là hai biến *doi_1* và *doi_2* là pointer nên chúng chứa địa chỉ của hai biến *a* và *b*, nên **doi_1* chính là *a* và **doi_2* chính là *b*.

Trong thư viện chuẩn của C cũng có nhiều hàm nhận đối số vào theo địa chỉ, ví dụ hàm `scanf()`, nhập trị vào cho biến từ bàn phím.

Ví dụ 13.26

```
#include <stdio.h>
#include <conio.h>
main()
{
    int n;

    clrscr();
    printf("Moi nhap mot so nguyen : ");
    scanf("%d", &n);
    printf("\n Binh phuong cua %d la %d\n", n, n*n);
    getch();
}
```

Khi đó địa chỉ của biến `n` sẽ được gởi cho hàm `scanf()`, hàm này với các lệnh trong nó mới có thể biết được vị trí của `n` trong bộ nhớ, mà đưa giá trị nhận được vào đó.

Việc truyền đối số là địa chỉ cho hàm rất thuận lợi trong các trường hợp cần hàm trả về nhiều trị cho nơi gọi, mà bản thân tên hàm không thôi không đủ. Hãy xét ví dụ cụ thể, cần thiết kế hàm giải phương trình bậc nhất $ax + b = 0$. Để thiết kế hàm này, đối số nhập cần thiết là hai tham trị đóng vai trò hai hệ số a và b ; khi có nghiệm, trị của nghiệm sẽ được trả về qua một địa chỉ của biến là đối số thật của nơi gọi, như vậy, đối số thật đóng vai trò nghiệm nhận trị trả về từ hàm, cần phải được truyền theo địa chỉ vào cho hàm để hàm tính toán xong, có trị, trị này sẽ được đưa vào địa chỉ của biến được cung cấp; tuy nhiên, tùy vào trị của hai hệ số a và b mà phương trình có các trạng thái nghiệm: vô số nghiệm, vô nghiệm hoặc có một nghiệm. Để biết được lúc nào phương trình ở trạng thái nghiệm nào, lập trình viên cần quy định một trị trả về, có thể trả về qua tên hàm, theo trạng thái nghiệm của phương trình, ví dụ, nếu hàm trả về trị.

- 1 : phương trình vô nghiệm
- 0 : phương trình vô số nghiệm
- 1 : phương trình có một nghiệm

Như vậy, hàm giải phương trình sẽ có prototype đề nghị như sau:

```
int ptb1 (double a, double b, double *x);
```

Hàm này sẽ được định nghĩa như sau:

```
int ptb1 (double a, double b, double *x)
{
    if (a == 0)
        if (b == 0)
            return 0;          /* phương trình vô số nghiệm */
        else
            return -1;        /* phương trình vô nghiệm */
    else
    {
        *x = -b/a;
        return 1;          /* phương trình có một nghiệm */
    }
}
```

Chương trình giải phương trình bậc nhất được viết như sau:

Ví dụ 13.27

```
#include <stdio.h>
#include <conio.h>
int ptb1 (double a, double b, double *x);
main()
{
    double h1, h2, nghiem;
    int trang_thai;

    clrscr();
    printf("Moi nhap hai he so: ");
    scanf("%lf%lf", &h1, &h2);
```

```

trang_thai = ptb1 (h1, h2, &nghiem);
switch (trang_thai)
{
    case -1:
        printf("Phuong trinh vo nghiem \n");
        break;
    case 0:
        printf("Phuong trinh vo so nghiem \n");
        break;
    case 1:
        printf("Phuong trinh co nghiem la %5.2f\n", nghiem);
        break;
}
getch();
}

int ptb1 (double a, double b, double *x)
{
    if (a == 0)
        if (b == 0)
            return 0; /* phuong trinh vo so nghiem */
        else
            return -1; /* phuong trinh vo nghiem */
    else
    {
        *x = -b/a;
        return 1; /* phuong trinh co mot nghiem */
    }
}

```

Như vậy chương trình trên sử dụng hàm ptb1(), hàm này trả về hai trị: một trị trả qua tên hàm, một trị qua địa chỉ của đối số truyền cho hàm. Việc trả nào trả về nơi đâu là lập trình viên quyết định khi lập trình, không có một nguyên tắc nào nhất định, tuy nhiên, thường những trị có cùng chức năng nên được trả về trong cùng một danh sách đối số để dễ theo dõi sau này.

13.5 HÀM TRẢ VỀ POINTER VÀ MẢNG

Cũng bình thường như các kiểu khác, một pointer có thể được trả về từ hàm, nếu pointer chỉ tới đối tượng là mảng, thì hàm trả về mảng; nếu pointer chỉ tới biến bình thường, thì hàm chỉ trả về con trỏ chỉ tới biến thường, cú pháp khai báo hàm như sau:

kiểu * tên_hàm (danh_sách_khai_báo_dối_số);

với **kiểu** là kiểu của đối tượng mà pointer được hàm trả về chỉ tới.

Kiểu này là bất kỳ, nếu chưa xác định được kiểu của đối tượng trả về, có thể void. Khi muốn sử dụng pointer chỉ đến kiểu void, ta cần ép kiểu về kiểu ta cần làm việc.

Ví dụ 13.28 Có khai báo

```
int * lon_nhat (int a, int b, int c);
```

thì hàm lon_nhat() trả về một địa chỉ, địa chỉ đó có thể là địa chỉ của một int đơn hoặc địa chỉ của một int bắt đầu cho mảng các int, việc sử dụng địa chỉ theo đối tượng nào là do nơi gọi.

Ví dụ 13.29 Thiết kế hàm nhập trị cho mảng các int

```
int *nhap_tri (int *num)
{
    static int a[10];
    int i, n;

    printf ("Nhập kích thước mảng: ");
    scanf ("%d", &n);
    *num = n;
    printf ("Nhập trị cho %d phần tử của mảng: ", n);
    for (i = 0; i < n; i++)
        scanf ("%d", &a[i]);
    return a; /* a là địa chỉ đầu mảng cần trả về */
}
```

Hàm trên trả về một pointer chỉ tới đối tượng int, đối tượng này khi nơi gọi sử dụng phải đang tồn tại, do đó ta cần khai báo mảng là mảng tĩnh trong hàm. Pointer được trả qua lệnh return

chính là tên mảng, là địa chỉ đầu mảng. Hàm nhap_tri() còn trả về trị là số phần tử của mảng đã nhập trị, đối số thật đưa vào cho hàm là một địa chỉ của một biến int để nhận trị này. Chương trình chính sử dụng hàm trên có thể được viết như sau:

```
#include <stdio.h>
#include <conio.h>
int *nhap_tri(int *num);
main()
{
    int *pint, so_phan_tu, i;

    clrscr();
    pint = nhap_tri (&so_phan_tu);
    printf ("Cac phan tu cua mang la:");
    for (i =0; i <so_phan_tu; i++) /* in trị của mảng */
        printf ("%d", pint[i]);
    getch();
}
int *nhap_tri (int *num)
{
    static int a[10];
    int i, n;

    printf ("Nhập kích thước mảng: ");
    scanf ("%d", &n);
    *num = n;
    printf ("Nhập trị %d phan tu cua mang: ", n);
    for (i = 0; i < n; i++)
        scanf ("%d", &a[i]);
    return a;
}
```

Chương trình sẽ cho xuất liệu ví dụ:

Nhập kích thước mảng: 8

Nhập trị cho 8 phần tử của mảng: 1 2 3 4 5 6 7 8

Các phần tử của mảng là: 1 2 3 4 5 6 7 8

Để trả trị về từ hàm, ta chỉ cần cung cấp địa chỉ cần được trả về trong lệnh **return**. Dĩ nhiên, địa chỉ này khi được nơi gọi sử dụng thì nó phải là địa chỉ của một đối tượng đang tồn tại trong bộ nhớ mà nơi gọi dùng được. Như vậy, địa chỉ này có thể là:

- địa chỉ của biến tĩnh, được khai báo trong hàm
- địa chỉ được truyền qua đối số hàm
- địa chỉ của biến toàn cục

Ví dụ 13.30

Hàm sau đây nhập trị cho mảng, trả về pointer chỉ tới đầu mảng này và số phần tử của mảng:

```
int *nhap_tri (int a[], int *num)
{
    int i;
    printf ("Nhập kích thước mảng: ");
    scanf ("%d", num);
    printf ("Nhập trị %d phần tử của mảng: ", *num);
    for (i = 0; i < *num; i++)
        scanf ("%d", &a[i]);
    return a;
}
```

Nơi gọi sử dụng hàm này có thể lấy được mảng *a* (đã được nhập trị) qua đối số hàm, hoặc địa chỉ của mảng này mà hàm trả về qua tên hàm. Khi gọi có thể là

nhap_tri (a, &so_phan_tu);

hoặc

pint = nhap_tri (a, &so_phan_tu);

Tùy mục đích sử dụng, ta có thể dùng biến *pint* hoặc mảng *a* để tính toán.

13.6 CHUỖI KÝ TỰ

Như đã đề cập trong phần chuỗi, một chuỗi, khi được sử dụng, đều có một địa chỉ riêng trong bộ nhớ, địa chỉ này là pointer chỉ tới đầu chuỗi.

Ví dụ 13.31 Khi khai báo

"Hello, world!"

thì chuỗi này sẽ được C ghi vào một nơi nào đó trong bộ nhớ và có địa chỉ xác định. Địa chỉ này có thể được gán vào cho một biến con trỏ chỉ tới ký tự để quản lý chuỗi.

Ví dụ 13.32 Cho khai báo

char *s;

ta có thể gán trị cho biến pointer s, là một biến pointer chỉ tới ký tự, địa chỉ của chuỗi "Hello, world!";

Chú ý rằng lệnh trên không thực hiện việc chép từng ký tự của chuỗi đã biết vào cho biến s, mà chỉ đơn giản là lấy địa chỉ xác định của chuỗi đã biết chép vào biến s mà thôi.

Như vậy, các thao tác về chuỗi đều dựa trên pointer chỉ tới chuỗi, do đó việc gán trị là chuỗi vào cho mảng các ký tự để tạo ra một biến chuỗi như ví dụ sau đây là không hợp lệ:

Ví dụ 13.33 Cho khai báo

char s[20];

s = "Hello, world!"; → không hợp lệ

Thao tác trên không hợp lệ vì s, tên mảng, là một hằng pointer, nên không thể nhận trị tử phép gán được. Để thực hiện thao tác mong muốn, C cung cấp một số hàm để tiện cho việc truy xuất chuỗi. Ta hãy xét đến một số hàm này.

I- Nhập trị chuỗi

Việc nhập trị cho chuỗi bao gồm hai bước: đầu tiên cần khai báo một nơi trống để chứa chuỗi, sau đó dùng một hàm nhập trị để lấy chuỗi.

Để có một nơi lưu chuỗi, cách đơn giản nhất là ta hãy khai báo một mảng ký tự, mảng này có thể là tĩnh hay tự động.

Ví dụ 13.34 Khai báo mảng

```
char s [40];
```

Việc nhập chuỗi vào mảng *s* có thể dùng một trong hai hàm: **gets()** hoặc **scanf()**. Cả hai hàm đều có prototype trong file stdio.h.

- **Hàm gets()** được sử dụng nhiều trong các chương trình C để nhập chuỗi, nó dùng để lấy một chuỗi từ thiết bị nhập chuẩn (như bàn phím). Khi nhập trị, hàm này đọc các ký tự đến khi nào gặp ký tự quy định hàng mới (tức ký tự '\n', tức khi ta ấn phím ENTER) thì kết thúc việc nhập. Sau đó hàm này lấy tất cả các ký tự đã nhập trước ký tự '\n', gắn thêm vào cuối chuỗi một ký tự NUL ('\0') và trả chuỗi cho chương trình gọi. Prototype của hàm này trong file stdio.h:

```
char * gets (char * s);
```

Hàm này trả về một pointer chỉ tới chuỗi, pointer này chính là tên mảng, là đối số sau khi gán chuỗi.

Ví dụ 13.35 Xét chương trình ví dụ sau đây

```
#include <stdio.h>
#include <conio.h>

main()
{
    char ten[41];
    char *pten;

    clrscr();
    printf ("Ban ten gi?\n");
    pten = gets (ten);
    printf("%s? Ai Chao ban %s\n", ten, pten);
    getch();
}
```

Chương trình cho xuất liệu ví dụ:

Ban ten gi?

Dang Thanh Tin

Al Chao ban Dang Thanh Tin

Chương trình trên cho phép ta nhập một chuỗi dài tối đa 40 ký tự, vì ký tự 41 (nếu dùng tới) được dùng để chứa ký tự NUL.

- **Hàm scanf()** cũng cho phép nhập chuỗi qua định dạng nhập "%s". Việc nhập chuỗi sẽ kết thúc khi hàm scanf() gặp một trong các ký tự khoảng trắng, ký tự tab hay ký tự xuống hàng đầu tiên mà nó gặp. Đây chính là điểm khác nhau giữa hai hàm nhập chuỗi gets() và scanf().

Ví dụ 13.36

```
#include <stdio.h>
#include <conio.h>
main()
{
    char ten1[41], ten2[41];

    clrscr();
    printf("Moi ban nhap hai ten: ");
    scanf ("%s %s", ten1, ten2);
    printf("Al Chao hai ban %s va %s \n", ten1, ten2);
    getch();
}
```

Chương trình cho xuất liệu ví dụ như sau:



Nếu ta có định dạng nhập cho scanf(), ví dụ "%5s", thì C chỉ cho phép nhập chuỗi có tối đa 5 ký tự, phần còn lại sẽ được nhập vào cho chuỗi ngay sau nó nếu có.

2. Xuất chuỗi

Để xuất chuỗi, hai hàm thường hay được dùng là puts() và printf(). Cả hai hàm có prototype trong file stdio.h.

- Hàm puts() là một hàm xuất chuỗi rất dễ sử dụng. Ta chỉ cần cung cấp cho hàm đối số là địa chỉ của chuỗi cần in. Hàm này sẽ đọc từng ký tự của chuỗi và in ra màn hình cho đến khi gặp ký tự NUL thì in ra màn hình thêm một ký tự xuống hàng nữa. Prototype của hàm này như sau:

```
int puts (char * s);
```

Hàm này sẽ về trị là một số không âm.

Ví dụ 13.37

Xét chương trình ví dụ sau đây:

```
#include <stdio.h>
#include <conio.h>
main()
{
    char ten[41];
    clrscr();
    printf("Moi ban nhap ten: ");
    gets(ten);
    printf("AI Chao ban: ");
    puts (ten);
    getch();
}
```

Chương trình cho xuất liệu ví dụ:

Moi ban nhap ten: Dang Thanh Tin

AI Chao ban: Dang Thanh Tin



dấu nháy (cursor) sau khi xuất chuỗi

- **Hàm printf()** cũng cho phép xuất chuỗi ra màn hình nếu ta dùng định dạng xuất "%s" cho nó. Hàm này sẽ không tự động in thêm ký tự xuống hàng mới như hàm puts().

Ví dụ 13.38

```
#include <stdio.h>
#include <conio.h>
main()
{
    char ten[41];

    clrscr();
    printf ("Moi ban nhap ten: ");
    gets(ten);
    printf ("AI Chao ban: %s", ten);
    getch();
}
```

Chương trình cho xuất liệu ví dụ:

Moi ban nhap ten: Dang Thanh Tin

AI Chao ban: Dang Thanh Tin

3- Gán trị cho chuỗi

Việc gán trị cho biến chuỗi thực tế là việc chép từng ký tự từ hằng chuỗi hoặc biến chuỗi đã biết sang một biến chuỗi khác. Trong C, thao tác này được thực hiện nhờ hàm **strcpy()**, hàm này có prototype trong file string.h như sau:

```
char *strcpy(char *dest, const char *src);
```

Hàm này thực hiện việc chép từng ký tự cho đến khi gặp ký tự NUL từ chuỗi ký tự có địa chỉ là pointer *src* (source) (chuỗi này sẽ không được thay đổi trị trong hàm) sang chuỗi có địa chỉ là pointer *dest*, hàm này trả về trị là địa chỉ của biến chuỗi *dest*. Nếu *dest* và *src* phủ lấp, cách làm việc của strcpy không được biết trước. Ta có thể hiểu việc chép trị này qua việc mô phỏng hàm strcpy như sau:

Ví dụ 13.39

```
char *strcpy(char *dest, const char *src)
{
    int i;

    for (i = 0; (dest[i] = scr[i]) != '\0'; i++)

    return dest;
}
```

Ví dụ sau đây minh họa cách sử dụng hàm này:

Ví dụ 13.40

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
main()
{
    char ten1[41], ten2[41];

    clrscr();
    printf("Moi ban nhap ten: ");
    gets(ten1);
    strcpy (ten2, ten1);
    printf("AI Chao ban: %s", ten2);
    getch();
}
```

Chương trình cho xuất liệu ví dụ:

Moi ban nhap ten: Dang Thanh Tin

AI Chao ban: Dang Thanh Tin

Chú ý, nếu chiều dài chuỗi đích không đủ để nhận hết dữ liệu từ chuỗi nguồn thì C vẫn thực hiện việc gán trị cho chuỗi đích mà không báo lỗi nào, tuy nhiên chương trình sẽ chạy sai cho các lệnh dưới hoặc không kết thúc được.

4- Lấy chiều dài chuỗi

Trong C, để lấy chiều dài chuỗi ta dùng hàm `strlen()`. Hàm này sẽ đếm số ký tự trong chuỗi cho đến khi gặp ký tự NUL thì kết thúc và trả về một số nguyên cho biết số ký tự này không kể ký tự NUL kết thúc chuỗi. Prototype của hàm này trong file `string.h`:

```
size_t strlen(const char *s);
```

với `size_t` là một kiểu nguyên, C quy định đây là `unsigned`.

Ví dụ 13.41

Xét chương trình nhập một chuỗi, đổi các ký tự thường của chuỗi đó thành ký tự hoa, in chuỗi đó ra lại màn hình.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
main()
{
    char ten[41];
    int i;

    clrscr();
    printf("Moi ban nhap ten: ");
    gets(ten);
    for ( i = 0; i < strlen (ten); i++)
        if (ten[i] >= 'a' && ten[i] <= 'z')
            ten[i] -= 32;
    printf ("AI Chao ban: %s", ten);
    getch();
}
```

Chương trình cho xuất liệu ví dụ như sau:

Moi ban nhap ten: dang thanh tin

AI Chao ban: DANG THANH TIN

5- Nối chuỗi

Để nối hai chuỗi lại, C có hàm chuẩn **strcat()**. Hàm này nhận hai chuỗi làm đối số, và một bản sao của chuỗi thứ hai sẽ được chép nối vào cuối của chuỗi thứ nhất, để tạo ra chuỗi mới. Chuỗi thứ hai vẫn không có gì thay đổi. Prototype của hàm này trong file string.h:

```
char *strcat(char *dest, const char *src);
```

Hàm này sẽ trả về trị là pointer chỉ tới chuỗi thứ nhất *dest*.

Ví dụ 13.42

Thiết kế chương trình nối hai chuỗi nhập từ bàn phím.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#define MAX 41

main()
{
    char hoten[41], ten[10];
    int i;

    clrscr();
    printf("Moi ban nhap Ho: ");
    gets(hoten);
    printf("Moi ban nhap Ten: ");
    gets(ten);
    if ( strlen (hoten) + strlen (ten) <MAX )
    {
        strcat ( hoten, ten );
        printf("A! Chao ban: %s", hoten);
    }
    else
        printf ("Khong du chieu dai chuoi! \n");
    getch();
}
```

Chương trình cho xuất liệu ví dụ:

```
Moi ban nhap Ho: Dang Thanh
Moi ban nhap Ten: Tin
Al Chao ban: Dang Thanh Tin
```

*Khi nhập có thêm khoảng
ký tự trắng tại đây*

Chú ý, hàm strcat() sẽ không kiểm tra chuỗi thứ hai có nối vào được chuỗi thứ nhất không, do đó, trước khi sử dụng hàm strcat(), ta cần phải kiểm tra chiều dài của chuỗi thứ nhất trước, nếu chiều dài cho phép thì hãy thực hiện hàm strcat().

6- So sánh chuỗi

Không như Pascal, C không cho phép so sánh chuỗi bằng các toán tử quan hệ bình thường, mà phải qua hàm so sánh chuỗi. Hàm này là strcmp(), có prototype trong file string.h:

```
int strcmp(const char *s1, const char*s2);
```

Hàm này so sánh hai chuỗi s1 và s2 và trả về một trị là:

- số dương nếu s1 > s2
- số âm nếu s1 < s2
- số 0 nếu s1 == s2

Ví dụ 13.43

Xét chương trình ví dụ sau đây:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
main()
{
    clrscr();
    printf("%d \n", strcmp("QUAN", quan));
    printf("%d \n", strcmp("QUAN", "QUAN"));
    printf("%d \n", strcmp("quan", "QUAN"));
    printf("%d \n", strcmp("quang", quanG));
    printf("%d \n", strcmp("quang", quan));
    getch();
}
```

Chương trình cho xuất liệu:

-32

0

32

32

103

Nhìn vào xuất liệu của chương trình trên, ta có nhận xét khi chuỗi thứ nhất lớn hơn chuỗi thứ hai thì hàm strcmp() trả về trị dương ($32 > 0$), như hai chuỗi "quan" và "QUAN", còn nếu hai chuỗi bằng nhau, hàm trả về trị là 0, như hai chuỗi "QUAN" và "QUAN", khi chuỗi thứ nhất nhỏ hơn chuỗi thứ hai thì hàm trả về trị âm ($-32 < 0$). Tuy nhiên, các số 32 và -32 có ý nghĩa gì trong việc so sánh hai chuỗi? Thật ra, khi so sánh, hàm strcmp() sẽ so sánh lần lượt từng ký tự của hai chuỗi, hiệu số hai mã ASCII của cặp ký tự đầu tiên khác nhau quyết định trị của hàm strcmp(), và C luôn luôn lấy ký tự trong chuỗi thứ nhất trừ ký tự trong chuỗi thứ hai. Điều này giải thích tại sao khi so sánh hai chuỗi "quan" và "QUAN", hàm trả về trị số 32 vì cặp ký tự đầu tiên khác nhau có hiệu là ' q ' - ' Q ' = 32. Cũng tương tự như vậy, khi dùng hàm strcmp() so sánh hai chuỗi "quang" và "quan" thì trị trả về là

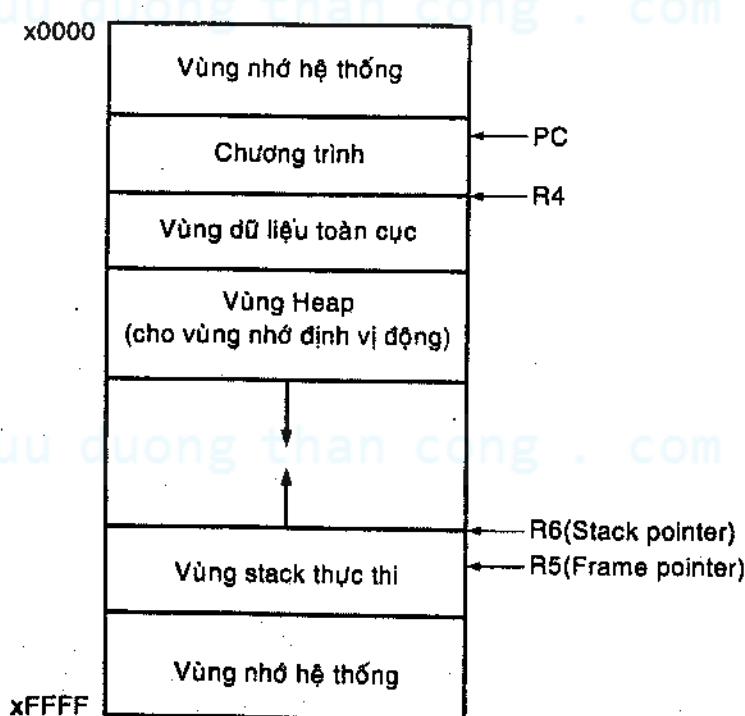
$$'g' - 'Q' = 103$$

103 chính là mã ASCII của ký tự 'g'

13.7 POINTER VÀ VIỆC ĐỊNH VỊ BỘ NHỚ ĐỘNG

Khi thiết kế chương trình, một trong những hạn chế cần lưu ý là bộ nhớ máy tính. Bộ nhớ là có giới hạn, nhưng nhiều chương trình lại khai báo và sử dụng biến nhiều, dẫn đến có thể chương trình không thể thực thi được. Có những biến chỉ sử dụng một vài lần, sau đó không còn đóng một vai trò nào trong chương trình nữa, nhưng chúng vẫn chiếm chỗ trong bộ nhớ. Để khắc phục tình trạng này, C cho phép khai báo các biến động, các biến này khi cần thì xin chỗ, không cần thì giải phóng vùng nhớ cho chương trình sử dụng vào mục đích khác. Các biến động này được cấp phát trong vùng nhớ **heap**, là vùng đáy bộ nhớ (vùng còn lại sau khi đã nạp các chương trình khác xong), và được quản lý bởi các biến pointer.

Hình 13.6 trình bày bộ nhớ khi dịch C sang LC-3. Từ hình này, chúng ta có thể thấy vùng nhớ heap nằm ngay sau vùng nhớ toàn cục, nó sẽ được cấp phát theo hướng địa chỉ tăng dần, hướng về vùng nhớ dành cho các stack thực thi. Đặc điểm của các biến động xin trong vùng nhớ heap là chúng được cấp phát thông qua các thường trình của hệ điều hành, nên khi một hàm của C kết thúc mà không báo hủy biến động đã được khai báo xin trước đó, thì biến đó vẫn được hệ điều hành tưởng rằng biến động đó đang được sử dụng. Và nếu điều này tiếp diễn, vùng heap sẽ bị tràn, tức chương trình C sẽ không thể được thực thi được do bộ nhớ không còn chỗ để cấp phát cho stack thực thi, chứa các mẫu tin kích hoạt của các hàm đang cần hoạt động trong chương trình. Mặt khác, do biến động được xin trong quá trình chạy chương trình, nên nhiều lúc hệ điều hành không tìm được vùng nhớ thích hợp cấp cho nó, khi đó một con trỏ NULL sẽ được trả về từ hàm xin cấp phát để báo điều này.



Hình 13.6 Cấu trúc bộ nhớ LC-3 với vùng bộ nhớ heap

Do đó, chúng ta nhớ là khi xin biến động rồi, khi không cần dùng nó nữa, hoặc trước khi kết thúc chương trình, thì cần phải báo xóa nó để giải phóng vùng nhớ đã đăng ký sử dụng.

Trong C có hai hàm chuẩn để xin cấp phát bộ nhớ động `malloc()` và `calloc()`, cả hai hàm đều có prototype nằm trong file `alloc.h` hoặc `stdlib.h` như sau:

```
void *malloc(size_t size);
void *calloc(size_t nitems, size_t size);
```

với:

- **`size_t`** là kiểu để khai báo kích thước khối bộ nhớ cần xin là `size` byte, `size_t` có thể là `int` hoặc `unsigned`.
- **`nitems`** là số phần tử cần xin.

Nếu hàm này xin được khối bộ nhớ cần thiết thì chúng sẽ trả về một pointer chỉ tới đầu khối này, nếu không xin được khối bộ nhớ cần thiết, hàm sẽ về trị là một con trỏ `NULL`, tức con trỏ không chỉ tới đâu cả, và như vậy ta không thể sử dụng được biến động.

Ví dụ 13.44

Cần xin một khối bộ nhớ có 10 phần tử `int`, ta viết như sau:

```
int *pint;
pint = (int *) malloc (10 * sizeof (int));
```

hoặc

```
pint = (int *) calloc (10, sizeof (int));
```

Tuy nhiên, biến động là biến được xin trong vùng nhớ heap, biến này sẽ không bị mất đi bình thường khi kết thúc chương trình, nên C đưa ra hàm `free()` để giải phóng khối bộ nhớ được xin bằng hàm `malloc()` hoặc `calloc()`. Nếu biến động được xin, sau khi dùng xong, vùng nhớ của nó không được giải phóng thì nó vẫn chiếm chỗ trong bộ nhớ, mặc dù chương trình đã kết thúc. Điều này sẽ ảnh hưởng đến việc cấp phát bộ nhớ cho việc thực hiện chương trình sau.

Prototype của hàm `free()` trong file `stdlib.h` hoặc `alloc.h` như sau:

```
void free (void * block);
```

với:

- *block* là pointer chỉ tới đầu khối bộ nhớ trong vùng nhớ heap cần giải phóng.

Chương trình ví dụ sau đây sử dụng hàm calloc(), malloc() và free():

Ví dụ 13.45

```
#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <alloc.h>
main()
{
    int *pint, s = 0, i;

    pint = (int *) calloc (10, sizeof (int)); /* xin biến động */
    /* hoặc pint = (int *) malloc (10 * sizeof (int)); */
    if (pint == NULL) /* kiểm tra sự tồn tại */
        /* của biến động */
        printf ("Khong du bo nho \n");
        exit (1);
    }
    clrscr();
    printf ("Moi nhap 10 tri vao mang: ");
    for (i = 0; i < 10; i++) /* Thao tác trên biến */
        scanf ("%d", &pint[i]); /* động */
    for (i = 0; i < 10; i++)
        s += pint[i];
    printf ("Tong cac phan tu cua mang la: %d \n", s);
    getch();
    free (pint); /* giải phóng biến động */
}
```

Chương trình cho xuất liệu ví dụ:

Moi nhap 10 tri vao mang: 1 2 3 4 5 6 7 8 9 0

Tong cac phan tu cua mang la: 45

Chương trình có sử dụng hàm **exit()**, hàm này sẽ kết thúc chương trình và trả về trị là đối số của hàm. Prototype của hàm này trong file process.h. Toán tử **sizeof** cho phép lấy kích thước của kiểu hoặc biến theo byte. Trong ví dụ trên, ta cần xin 10 phần tử int, như vậy kích thước vùng nhớ cần xin là $10 \times 2 = 20$ byte, vì đối với máy PC, thường biến kiểu int sẽ chiếm 2 byte (hoặc 4 byte) trong bộ nhớ. Do đó, để tổng quát hóa chương trình, ta không nên để cụ thể là 20 mà nên để là

`10 * sizeof (int)`

Cần chú ý, hàm **calloc()** có một đặc điểm là sau khi xin xong biến động, tất cả các phần tử biến động đều được khởi động trị là 0; có nghĩa **calloc()** là hàm xin biến động có khởi động trị.

Do đó, việc dùng các biến động giúp lập trình viên linh động trong việc xin biến, khi cần thì xin biến, không cần thì giải phóng vùng nhớ đó cho mục đích khác. Đây là ưu điểm của một chương trình C. [cuu duong than cong . com](http://cuuduongthancong.com)

13.8 MẢNG CÁC POINTER

C cho phép khai báo mảng các pointer, việc khai báo và sử dụng mảng các pointer làm cho một chương trình C rất linh động trong việc quản lý dữ liệu, nhất là các dữ liệu có kích thước lớn. Ví dụ, ta muốn sắp xếp các dữ liệu là các cấu trúc (struct), đây là một cấu trúc dữ liệu sẽ được đề cập trong chương sau, việc thay đổi vị trí trực tiếp trên các **struct** là rất mất thời gian; thay vì thay đổi trực tiếp như vậy, ta nên dùng một mảng các pointer, mỗi pointer chỉ tới một cấu trúc, thứ tự sắp xếp các pointer trong mảng là thứ tự luận lý cần truy xuất các cấu trúc, như vậy việc sắp xếp các cấu trúc thực tế trở thành việc sắp xếp các pointer trong mảng các pointer. Vì vậy, ta không cần phải thay đổi trực tiếp trên các cấu trúc.

Cú pháp khai báo mảng các pointer:

kiểu * tên_mảng [Kích_thước];

Ví dụ 13.46 Khi khai báo

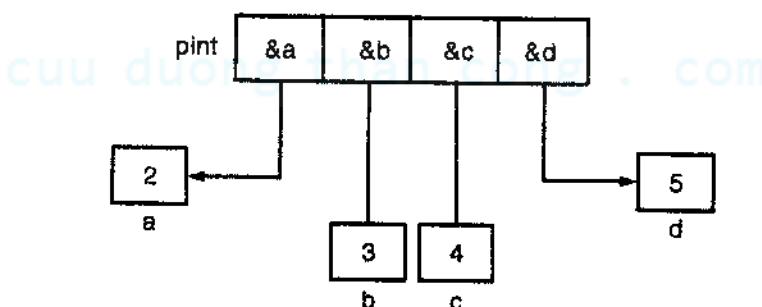
`int * pnt[4];`

thì pint là tên một mảng, mảng này được cấp chỗ trong bộ nhớ để chứa bốn pointer, trị trong các phần tử này là trị rác nên đối tượng mà chúng chỉ tới không thể sử dụng được. Sau này khi được gán trị thì mỗi pointer chỉ tới một biến int hoặc mảng các int, mảng này có thể là mảng bình thường hoặc mảng là biến động.

Ví dụ 13.47 Khi khai báo

```
int * pint[4];
int a = 2, b = 3, c = 4, d = 5;
pint[0] = &a;
pint[1] = &b;
pint[2] = &c;
pint[3] = &d;
```

Ta có thể hình dung việc gán trên như sau:

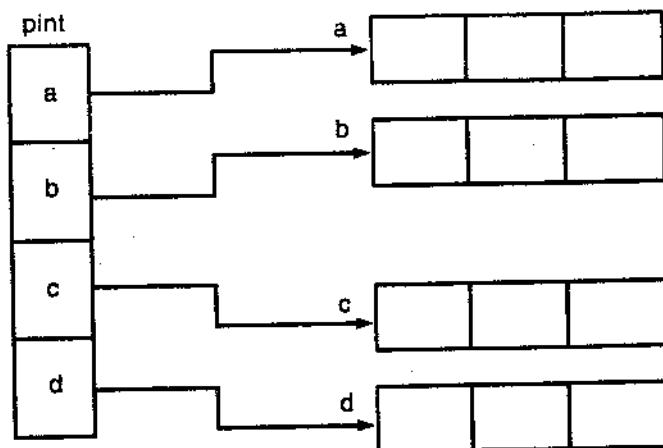


thì mảng pint chứa các pointer chỉ tới các đối tượng int.

Nhưng, khi khai báo

```
int * pint[4];
int a[3], b[3], c[3], d[3];
pint[0] = a;
pint[1] = b;
pint[2] = c;
pint[3] = d;
```

thì mảng pint lại chứa các pointer chỉ tới mảng các int. Ta có thể hình dung việc gán trên như sau:



Như vậy, khi truy xuất, ta có thể sử dụng `pint[0][1]` thay vì `a[1]`, và tương tự cho các phần tử khác.

Ta hãy xét chương trình sắp xếp các biến int với địa chỉ được chứa trong mảng các pointer.

cuuduongthancong.com

Ví dụ 13.48

```
#include <stdio.h>
#include <conio.h>
void sap_xep(int *pi[], int num);
main()
{
    int *pint[4], i;
    int a = 2, b = 3, c = 4, d = 5;

    clrscr();
    pint[0] = &a;
    pint[1] = &b;
    pint[2] = &c;
    pint[3] = &d;
    sap_xep (pint, 4);
    printf ("Cac phan tu cua mang la: ");
    for (i = 0; i < 4; i++)
        printf ("%d", *pint[i]);
    getch();
}
```

```

}

void sap_xep (int *pi[], int num)
{
    int i, j, max, vtmax, *ptam;

    for (i = 0; i < num-1; i++)
    {
        max = *pi[i];
        vtmax = i;
        for (j = i+1; j < num; j++)
            if (max < *pi[j])
            {
                max = *pi[j];
                vtmax = j;
            }
        if (i != vtmax)
        {
            ptam = pi[i];
            pi[i] = pi[vtmax];
            pi[vtmax] = ptam;
        }
    }
}

```

Chương trình cho xuất liệu:

Các phần tử của mảng là: 5 4 3 2

Trong chương trình trên, mảng *pint* gồm bốn phần tử *pint[0]*, ..., *pint[3]*, mỗi phần tử chứa địa chỉ của các biến *a*, *b*, *c*, *d*, và như vậy, mỗi phần tử của mảng *pint* là một pointer chỉ tới một biến *int* khác nhau. Việc so sánh vẫn là so sánh trên bản thân các đối tượng mà các pointer này chỉ tới, tuy nhiên việc hoán đổi lại là hoán đổi trên mảng pointer. Sau khi sắp xếp xong, pointer đầu tiên của mảng, *pint[0]*, luôn chỉ đến đối tượng chứa giá trị lớn nhất, và kế tiếp.

Xét chương trình ví dụ sau đây, chương trình này tạo một mảng pointer, mỗi pointer chỉ tới một chuỗi ký tự mà vùng nhớ cung cấp cho các chuỗi ký tự này là biến động, xin bằng *malloc()*.

Ví dụ 13.49

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <alloc.h>
#include <process.h>
#define MAX 81
void sap_xep(char *ps[], int num);
main()
{
    char *s[4];
    int i;

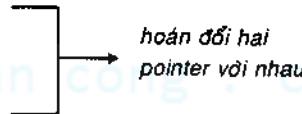
    clrscr();
    for (i = 0; i < 4; i++)
    {
        s[i] = (char *) malloc (MAX * sizeof(char));
        if ( s[i] == NULL )
        {
            printf ("Khong du bo nho dong cho thao tac \n");
            exit (0);
        }
        printf ("Moi nhap chuoi thu %d: ", i+1);
        gets (s[i]);
    }
    sap_xep(s, 4);
    printf ("Cac chuoi duoc sap xep la: \n");
    for (i = 0; i < 4; i++)
        puts (s[i]);
    getch();
    for (i = 0; i < 4; i++)
        free(s[i]);
}
void sap_xep(char *ps[], int num)
{
```

```

int i, j, vtmax;
char *max, *ptam;

for (i = 0; i < num-1; i++)
{
    max = ps[i];
    vtmax = i;
    for (j = i+1; j < num; j++)
        if (strcmp (max, ps[j]) < 0 )
    {
        max = ps[j];
        vtmax = j;
    }
    if (i != vtmax)
    {
        ptam = ps[i];
        ps[i] = ps[vtmax];
        ps[vtmax] = ptam;
    }
}
}

```



Chương trình trên cho xuất liệu ví dụ:

```

Moi nhap chuoi thu 1: Dang Thanh Tin
Moi nhap chuoi thu 2: Vo Van Danh
Moi nhap chuoi thu 3: Doan Sy Tri
Moi nhap chuoi thu 4: Ngo Hung Phuong
Cac chuoi duoc sap xep la:Vo Van Danh
Ngo Hung Phuong
Doan Sy Tri
Dang Thanh Tin

```

Chương trình trên có thể được chia ra làm bốn đoạn, mỗi đoạn có ý nghĩa riêng. Đoạn đầu tiên là việc xin vùng nhớ động cho các biến chuỗi. Việc xin này cần một vòng **for**. Sau khi xin xong, gọi hàm **gets()** để nhập trị cho biến chuỗi. Đoạn kế tiếp là gọi hàm **sap_xep()**, đưa vào đối số là mảng các pointer đến các chuỗi theo thứ

tự từ lớn tới nhỏ, việc so sánh vẫn so sánh trên các chuỗi, nhưng khi hoán đổi lại hoán đổi vị trí trên mảng các pointer, để pointer đầu tiên, $ps[0]$, luôn chứa chuỗi lớn nhất, và kế tiếp sau thì giảm dần. Kế tiếp là in các chuỗi được sắp xếp ra màn hình, việc in này cứ theo trình tự từ $ps[0]$ đến $ps[3]$, và như vậy các chuỗi được in ra theo thứ tự so sánh giảm dần. Sau cùng là đoạn chương trình giải phóng vùng nhớ động đã được xin cho các biến chuỗi.

Qua ví dụ trên, ta có thể thấy khái niệm sử dụng mảng pointer tương tự như mảng hai chiều.

Ví dụ 13.50 Khi khai báo và thực hiện các lệnh gán sau:

```
int a[4][5];
```

```
int *b[4];
```

```
b[0] = a[0];
```

```
b[1] = a[1];
```

```
b[2] = a[2];
```

```
b[3] = a[3];
```

thì khi truy xuất $a[2][3]$ và $b[2][3]$, C đều hiểu đây là các biến int. Tuy nhiên cũng cần lưu ý, việc khai báo mảng các pointer chỉ được C cấp chỗ cho các pointer mà thôi, còn đối tượng chưa xác định. Trong khi đó, một mảng nhiều chiều khi được khai báo thì đã được cấp chỗ trong bộ nhớ để sẵn sàng lưu dữ liệu.

Một mảng các pointer cũng có thể được khởi động trị nếu mảng là mảng toàn cục hay mảng tĩnh.

Ví dụ 13.51

```
static char *thu[7] = {
    "Thu 2", "Thu 3", "Thu 4", "Thu 5",
    "Thu 6", "Thu 7", "Chua nhat"
};
```

thì các pointer $thu[0]$, ... chính là các chuỗi "Thu 2", ..., vì các chuỗi "Thu 2", ... luôn cho địa chỉ đến đầu chuỗi, các địa chỉ này được khởi động gán lần lượt vào $thu[0]$, ...

13.9 POINTER CỦA POINTER

Như đã đề cập trong các phần trên, biến pointer khi được khai báo thì được cấp chỗ trong bộ nhớ, chính vì vậy mà bản thân biến pointer cũng có địa chỉ, địa chỉ này là địa chỉ của một biến pointer. Cho phép khai báo một pointer chỉ đến một đối tượng là một pointer, và dĩ nhiên pointer này chỉ đến một đối tượng thuộc kiểu tùy ý. Cú pháp khai báo pointer này như sau:

kiểu ** tên_pointer

Chú ý rằng, trước tên_pointer là hai dấu * biểu thị tên_pointer là pointer của pointer chỉ đến một đối tượng có kiểu là kiểu.

Ví dụ 13.52

```
int **pint;
int *p;
int a[4][4];
```

thì biến *pint* là một biến pointer chỉ đến đối tượng là một pointer, pointer này lại chỉ đến đối tượng là một int, hoặc mảng các int, tức các thao tác sau là hợp lệ:

pint = &p;

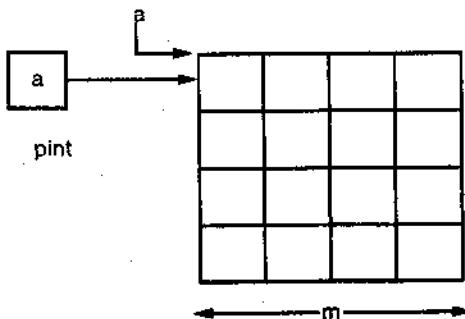
hoặc

*pint = (int **) &a;*

Chú ý, ta có thể dùng biến pointer của pointer để quản lý mảng hai chiều, khi đó, địa chỉ của mảng hai chiều là một hằng pointer, hằng này có thể được sử dụng để gán vào cho biến pointer của pointer sau khi ép kiểu thích hợp.

Ta có thể hình dung việc gán trên qua hình sau:

thay vì truy xuất *a[i][j]*, ta có thể truy xuất **(* (int (*[4])) pint + i) + j*. C không tự động tính ra được *pint[i][j]* là *a[i][j]*, nên thao tác trên *pint[i][j]* tạo ra những lỗi luận lý khó sửa. Chương trình ví dụ sau minh họa cho khả năng truy xuất này.



Ví dụ 13.53

```
#include <stdio.h>
#define MAX_ROW 3
#define MAX_COL 3

main()
{
    int row, col;
    int *pint1;
    int a2d [MAX_ROW][MAX_COL] = { {0, 1, 2},
                                    {10, 11, 12},
                                    {20, 21, 22}
                                };
    int **pint2;
    int (*pa2d)[MAX_ROW][MAX_COL];
    /* Thu dia chi cua pointer va mang 2 chieu */

    pint1 = a2d[1];
    pa2d = &a2d;
    pint2 = (int **) &a2d;

    printf ("pint1 = a2d[1] = %p\n", pint1);
    printf ("*( *( ( int (*)[MAX_COL] ) pint2 + 1)+ 2)= %d\n",
           *( *( ( int (*)[MAX_COL] ) pint2 + 1)+ 2));
}
```

```

*( *( ( int (*)[MAX_COL] ) pint2 + 1)+ 2));
printf ("*( *(a2d + 1) + 2) = %d\n", *( *(a2d + 1) + 2));
printf ("*pint1[2] = %d\n", pint1[2]);
printf ("(*pa2d)[1][2] = %d\n", (*pa2d)[1][2]);

printf ("Tri cua cac phan tu trong mang truy xuat qua pointer 2 lan:\n");
for (row = 0; row < MAX_ROW; row++)
{
    for (col = 0; col < MAX_COL; col++)
        printf ("%d ", *( *( ( int (*)[MAX_COL] ) pint2 + row)+ col));
    printf ("\n");
}
getchar();
}

```

Chương trình trên cho xuất liệu ví dụ như sau:

```

E:\BC5\BIN\thu_pointer_chom_2_chieu.exe
pint1 = a2d[1] = 0012FF74
*( *( ( int (*)[MAX_COL] ) pint2 + 1)+ 2) = 12
*( *(a2d + 1) + 2) = 12
*pint1[2] = 12
(*pa2d)[1][2] = 12
Tri cua cac phan tu trong mang truy xuat qua pointer 2 lan:
0 1 2
10 11 12
20 21 22

```

Ví dụ sau đây, pointer của pointer chỉ đến mảng các pointer, mảng này chứa các địa chỉ của các biến int.

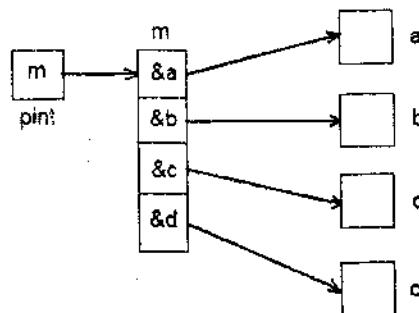
Ví dụ 13.54

```

int *m[4];
int a = 1, b = 2, c = 3, d = 4;
int **pint;
pint = m;
m[0] = &a;
m[1] = &b;
m[2] = &c;
m[3] = &d;

```

Ta có thể hình dung việc gán trên qua hình sau:



Thay vì truy xuất trực tiếp a , b , ..., ta có thể dùng pointer, các $*(pint[i])$ là các biến tương ứng: a là $*(pint[0])$, ...

Pointer của pointer cũng có thể chỉ đến mảng các pointer, mảng này lại chứa các pointer chỉ đến mảng int một chiều.

Xét ví dụ sau đây:

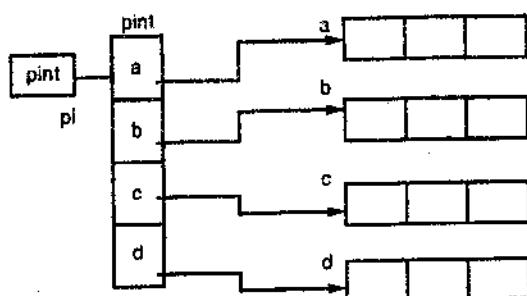
Ví dụ 13.55

Cho khai báo sau:

```

int ** pi;
int * pint[4];
int a[3], b[3], c[3], d[3];
pi = pint;
pint[0] = a;
pint[1] = b;
pint[2] = c;
pint[3] = d;
  
```

Ta có thể hình dung việc gán trên qua hình sau:



Xét chương trình ví dụ sau đây, chương trình này dùng một biến pointer của pointer để quản lý mảng động một chiều, mảng này là mảng các pointer chỉ đến các biến động lưu chuỗi ký tự.

Ví dụ 13.56

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
#include <string.h>
#include <process.h>
#define MAX 81
main()
{
    char **ps, l, n;
    char *tam;
    clrscr();
    tam = (char *) malloc (MAX * sizeof(char));
    if( tam == NULL )
    {
        printf("Khong du bo nho 1 \n");
        exit (0);
    }
    printf("So chuoi ky tu can lam: ");
    scanf("%d", &n);
    flushall(); → xóa bộ đệm bàn phím
    ps = (char **) malloc (n * sizeof (char *));
    if (ps == NULL)
    {
        printf("Khong du bo nho 2 \n");
        exit (0);
    }
    for (l = 0; l < n; l++)
}
```

*xin vùng nhớ cho
biến chuỗi tam để
làm mới trường
nhập chuỗi*

*xin vùng nhớ cho
mảng động ps để
chứa các pointer
đến chuỗi*

```

{
    printf ("Nhập chuỗi thu %d: ", i+1);
    gets(tam);
    ps[i] = (char *) malloc ( strlen(tam)*sizeof(char) );
    if (ps[i] == NULL)
    {
        printf("Không đủ bộ nhớ 3 \n");
        exit (0);
    }
    strcpy (ps[i], tam);
}

for (i = 0; i < n; i++)
    printf ("Chuỗi thu %d: %s \n", i+1, ps[i]);
free (tam);
for (i = 0; i < n; i++)
    free (ps[i]);
free (ps);
getch();
}

```

nhập
và lưu
chuỗi

giải phóng
biến động

Chương trình cho xuất liệu ví dụ:

Nhập chuỗi ký tự cần làm: 4

Nhập chuỗi thu 1: Tin

Nhập chuỗi thu 2: Thu

Nhập chuỗi thu 3: Trí

Nhập chuỗi thu 4: Phuong

Chuỗi thu 1: Tin

Chuỗi thu 2: Thu

Chuỗi thu 3: Trí

Chuỗi thu 4: Phuong

Chương trình trên có sử dụng hàm flushall(), hàm này có tác dụng xóa bỏ các ký tự hiện thời trong bộ đệm bàn phím, nếu không có hàm này, việc nhập chuỗi không được thực hiện như ý muốn, vì

trước khi nhập chuỗi, lệnh `scanf("%d", &n);` nhập trị vào cho biến `n`, nhưng lại để lại ký tự xuống hàng (tương đương việc ấn phím ENTER) trong bộ đệm bàn phím, đến khi nhập chuỗi, ký tự này lại được lấy ra ngay và ta được chuỗi rỗng.

13.10 ĐỐI SỐ CỦA HÀM MAIN

Trước đây khi lập trình, ta không cần quan tâm đến hàm `main()` có thể nhận được đối số vào hay không, và nếu nhận được thì như thế nào? C hoàn toàn cho phép việc nhận đối số vào hàm `main()`, có hai đối số C đã quy định theo thứ tự:

int argc: đối số cho biết số tham số đã nhập, kể cả tên chương trình.

char *argv[]: mảng các pointer chỉ tới các chuỗi là tham số đi theo sau tên chương trình khi chạy chương trình từ DOS. Chuỗi là tên chương trình luôn được chỉ bởi `argv[0]`.

Ví dụ 13.57

Một chương trình có tên là `thu.C`, hàm `main()` được khai báo:

```
main (int argc, char *argv[])
{
    ...
}
```

Khi chạy chương trình từ DOS, ví dụ nhập từ dấu nhắc

D:\>thu doi_so_1 doi_so_2 123

Khoảng cách giữa các đối số

thì trong chương trình ta có:

- `argc = 4` vì có 4 đối số được truyền vào cho chương trình.
- `argv[0]` chỉ chuỗi "thu"
- `argv[1]` chỉ chuỗi "doi_so_1"
- `argv[2]` chỉ chuỗi "doi_so_2"

argv[3] chỉ chuỗi "123"

argv[4] là pointer NULL: hết đối số

Như vậy, khi chạy chương trình, ta luôn gọi tên chương trình, nên argc tối thiểu là 1.

Ví dụ 13.58 Xét chương trình ví dụ sau:

```
#include <stdio.h>
#include <conio.h>
main (int argc, char *argv[])
{
    int i;

    clrscr();
    printf ("Cac doi so cua chuong trinh la: \n");
    printf ("Ten chuong trinh la: %s \n", argv[0]);
    if ( argc >1 )
        for (i = 1; i < argc; i++)
            printf ("Doi so thu %d: %s \n", i, argv[i]);
    getch();
}
```

Nếu nhập từ bàn phím như sau

D:\>thu_main tin thu 123 ↵

thì chương trình cho xuất liệu là:

```
Cac doi so cua chuong trinh la:
Ten chuong trinh la: D:\thu_main.exe
Doi so thu 1: tin
Doi so thu 2: thu
Doi so thu 3: 123
```

Ta hãy xét một chương trình khác, chương trình tính tổng, với các số cần tính nằm ngay sau tên chương trình khi chạy.

Ví dụ 13.59

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
main (int argc, char *argv[])
{
    int i, tong = 0;

    if ( argc > 1 )
    {
        for ( i = 1; i < argc; i++ )
            tong += atoi(argv[i]);
        printf ("Tong la %d \n",tong);
    }
    else
        printf ("Chuong trinh can doi so di theo \n");
    getch();
}
```

Khi chạy chương trình từ Command line:Run trong Windows, ta có xuất liệu ví dụ:

C:\>main 1 2 3 4 5

Tong la 15

Trong chương trình trên, ta có sử dụng hàm atoi(), đây là hàm đổi từ chuỗi sang số int, hàm có prototype trong file stdlib.h như sau:

`int atoi (const char *s);`

với s là chuỗi cần đổi sang số.

13.11 POINTER CHỈ TỚI HÀM

C đã có quy định, một hàm của C, nếu đã được khai báo, thì tên hàm là một hằng pointer, trị này có thể được gán cho một biến pointer, và ta có được pointer trỏ tới hàm. Cú pháp khai báo một pointer chỉ tới hàm:

`kiểu (* tên_pointer) (kiểu_các_dổi_số);`

với:

- **kiểu** là kiểu bất kỳ, đây là kiểu dữ liệu mà hàm trả về cho nơi gọi. Kiểu này bắt buộc phải có, dù hàm trả về dữ liệu int.
- **tên_pointer** là tên biến pointer chỉ tới hàm, đây là một khai báo biến bình thường, nên sau khai báo, C cấp vùng nhớ cho biến tên_pointer.
- **kiểu_các đối số** xác định kiểu của các đối số cần đưa vào cho hàm, mà hàm này đang được pointer chỉ tới. Kiểu_các đối số này có thể có hay không cũng được, nếu có, C sẽ kiểm tra đối số đưa vào có phù hợp không, nếu không, C không cần quan tâm đến các đối số khi gọi hàm.

Ví dụ 13.60

Nếu khai báo

```
int (* p_function) (int, int);
```

thì C xác định một biến pointer tên *p_function*, đây là một pointer chỉ tới một hàm trả về một trị int, nhận vào hai đối số int. Tuy nhiên giá trị trong biến này chưa được xác định, muốn sử dụng pointer này ta cần gán trị là địa chỉ của một hàm nào đó, hàm này cũng phải trả về một int, nhận vào hai số int.

Ví dụ 13.61

Nếu khai báo

```
int (* p_function) (int, int);
```

và đã có hàm

```
int cong (int a, int b)
```

```
{
```

```
...
```

```
}
```

thì ta có thể thực hiện phép gán

```
p_function = cong;
```

và sau đó có thể dùng pointer này gọi hàm

```
tong = (*p_function) (m, n);
```

Chú ý, không được nhầm lẫn giữa pointer tới hàm và hàm trả về pointer, khi viết

int * p_function (int, int) → hàm trả về pointer
 int (* p_function)(int, int) → pointer tới hàm

Pointer chỉ tới hàm là một biến, nên khi cần ta có thể gán trị khác, là địa chỉ của hàm khác vào cho nó, khi đó chỉ đến hàm này.

Việc cho phép khai báo pointer chỉ tới hàm là một điểm mạnh của C, vì nhiều lúc khi lập trình ta cần gọi hàm qua chỉ số đã định, chứ không cần phải gọi tên qua hàm. Để dễ hình dung, ta hãy xét ví dụ sau:

Ví dụ 13.62

```
#include <stdio.h>
#include <conio.h>
double cong(double a, double b);
double tru(double a, double b);
double nhan(double a, double b);
main()
{
    double (*pf[3])(double, double); → khai báo mảng các pointer
    double a, b, ket_qua;
    int thao_tac;
    pf[0] = cong;
    pf[1] = tru;
    pf[2] = nhan; → Khởi động vị trí cho các pointer
}
```

```
clrscr();
printf ("Moi ban nhap hai so: ");
scanf ("%lf %lf", &a, &b);
printf ("Cac thao tac: \n");
printf ("1. Cong hai so \n");
printf ("2. Tru hai so \n");
printf ("3. Nhan hai so \n");
printf ("Moi ban chon thao tac: ");
scanf ("%d", &thao_tac);
```

```

ket_qua = pf[thao_tac-1](a, b); → gọi hàm qua pointer
printf ("Kết quả của phép toán trên là %.2lf\n", ket_qua);
getch( );
}

double cong(double a, double b)
{
    return a + b;
}

double tru(double a, double b)
{
    return a - b;
}

double nhan(double a, double b)
{
    return a * b;
}

```

Trong chương trình trên, ta khai báo mảng các pointer chỉ đến hàm, các pointer này được gán trị là địa chỉ của các hàm cong(), tru(), nhan(). Khi gọi sử dụng hàm, ta chỉ cần cung cấp chỉ số để gọi hàm tương ứng mà thôi.

13.12 ỨNG DỤNG

Như đã trình bày trong chương mảng, có hai ứng dụng thường được lập trình viên sử dụng trong chương trình, đó là hai cấu trúc dữ liệu stack và queue. Tuy nhiên, nếu sử dụng mảng bình thường thì ta sẽ bị hạn chế về bộ nhớ, kích thước của stack và queue là cố định không thể tăng hoặc giảm khi cần. Để khắc phục hạn chế này, ta nên dùng cấu trúc dữ liệu stack và queue là các danh sách liên kết, đây là sự kết nối giữa các thành phần có cấu trúc, mà cụ thể trong từng trường hợp ta sẽ nêu ra sau đây. Phần này có sử dụng kiến thức về struct, xin độc giả xem chương sau để hiểu rõ hơn.

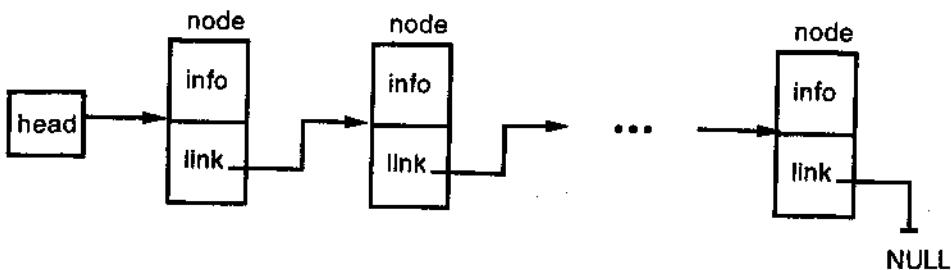
13.12.1 Danh sách liên kết là stack

Có thể hiểu danh sách liên kết dạng stack là một cấu trúc dữ liệu động, trong đó phần tử đưa vào stack sau sẽ được lấy ra trước

và ngược lại, các thông tin đưa vào stack sẽ được cất vào trong một node, mà vùng nhớ cho node này được xin bằng các hàm cấp phát biến động `calloc()` hoặc `malloc()`.

Để quản lý một stack, cần dùng một biến pointer trỏ đến node, biến này gọi là **head**, nó có chức năng chỉ đến phần tử node cần truy xuất trong stack.

Có thể hiểu một danh sách liên kết dạng stack qua hình vẽ sau:



Cấu trúc một node thành phần của stack bao gồm hai phần:

- Phần thông tin, tạm gọi là **info**, lưu các dữ liệu đã được đưa vào stack
- Phần liên kết, tạm gọi là phần **link** hoặc **next**, là pointer chỉ đến node kế. Ví dụ cấu trúc của node trong danh sách liên kết dạng stack lưu các dữ liệu là chuỗi các node như sau:

```

struct node
{
    char *message;
    struct node *next;
};
  
```

Như vậy biến **head** để quản lý stack được khai báo là

```
struct node *head;
```

Các thao tác cần thực hiện trên stack:

1- *Dưa một phần tử vào stack - Thao tác push*

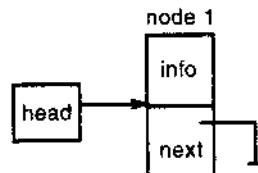
Để đưa một phần tử vào stack ta cần thực hiện các thao tác sau:

- Xin một biến động là node mới để lưu dữ liệu
- Cắt dữ liệu cần đưa vào stack vào thành phần info của node mới này.
- Tạo mốc nối giữa stack đang có và node mới.

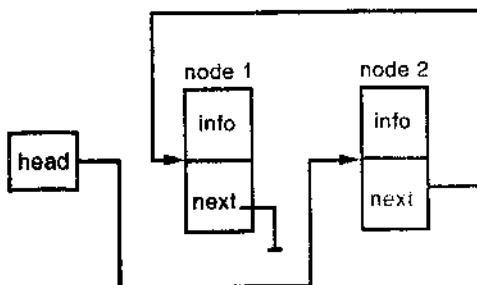
Node mới này được đưa vào stack và được head chỉ đến, vì nó là node đưa vào stack sau cùng nên cần được truy xuất trước tiên.

Ta có thể hiểu thao tác push qua hình vẽ sau:

Trước khi push



Sau khi push



Ví dụ sau đây cho hàm push đối với dữ liệu là một chuỗi, trong đó phần info có tên cụ thể là message, phần link có tên cụ thể là next.

```
void push (const char *s)
{
    struct node *pnode;
    /* xin biến động chỉ tới node */
    pnode = (struct node *) malloc ( sizeof( struct node ) );
    if ( pnode == (struct node *) NULL )
    {
        printf ("Khong du bo nho \n");
        exit (0);
    }
}
```

```

}

/* xin biến động để lưu thông tin */
pnode->message = (char *) malloc ( strlen(s) + 1 );
if ( pnode->message == (char *) NULL )
{
    printf ("Khong du bo nho \n");
    exit (0);
}

/* lưu thông tin vào stack */
strcpy ( pnode->message, s );
/* Tạo mốc nối mới */
pnode->next = head;
head = pnode;
}

```

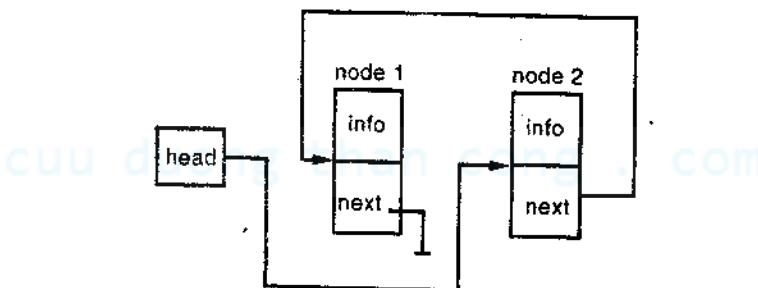
2- Lấy một phần tử từ stack - Thao tác pop

Để lấy một phần tử từ stack, ta cần thực hiện các thao tác sau:

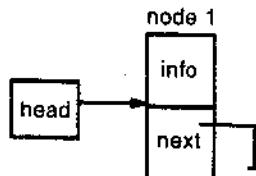
- Lấy thông tin trong node đang được chỉ bởi head cất vào biến lưu info, là đối số hàm pop().
- Tạo mốc nối mới trong stack đang có, bỏ node đã lấy thông tin.
- Giải phóng các vùng nhớ động đã xin cho node

Ta có thể hiểu thao tác pop qua hình vẽ sau:

Trước khi pop



Sau khi pop



Ví dụ sau đây cho hàm pop đối với dữ liệu là một chuỗi:

```
char *pop (char *s )
{
    struct node *temp_node;
    if ( head == (struct node *) NULL )
    {
        printf ("stack rong \n");
        strcpy (s, (char *) NULL);
        return ( (char *) NULL );
    }
    else
    {
        strcpy (s, head->message);
        temp_node = head->next;
        free (head->message);
        free ( (char *) head );
        head = temp_node;
        return s;
    }
}
```

3- Xem stack

Thao tác này chỉ đơn giản là xem thông tin của phần tử hiện hành cần truy xuất trong stack có trị như thế nào. Để thực hiện thao tác này, ta cần thiết kế hàm look() như sau:

```
char *look ( char *s )
{
    if ( head == (struct node *) NULL )
    {
        printf ("stack rong \n");
        strcpy (s, (char *) NULL);
        return ( (char *) NULL );
    }
    else
    {
        strcpy (s, head->message);
        return s;
    }
}
```

4- Khởi động stack

Khi sử dụng stack ta cần khởi động stack, nếu stack đang có sẵn, hàm khởi động cần xóa các phần tử cũ, và khởi động biến head trị khởi thủy là NULL, còn nếu stack chưa có thì việc khởi động chỉ đơn giản là khởi động trị cho head trị đầu là NULL. Hàm này có thể được viết như sau:

```
void init (struct node *head)
{
    struct node *pnode;
    for (pnode = head; pnode != (struct node *)NULL ; )
    {
        struct node *temp_node;
        free (pnode->message);
        temp_node = pnode->next;
        free(pnode);
        pnode = temp_node;
    }
    head = (struct node *)NULL;
}
```

Xét chương trình ví dụ sau đây:

Ví dụ 13.58

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
#include <process.h>
struct node
{
    char *message;
    struct node *next;
};

struct node *head;
void push (const char *s);
char *pop ( char *s);
char *look ( char *s );
void init (struct node *head);
```

```

main()
{
    char s[81];
    clrscr();
    init (head);    → khởi động stack
    do
    {
        printf ("Moi ban nhap mot chuoi: ");
        gets(s);
        if ( strcmp (s, (char *)NULL) != 0 )
            push (s);
    }
    while ( strcmp (s, (char *)NULL ) );
    printf ("Cac chuoi trong stack la: \n");
    for ( ; pop (s) != (char *)NULL ; )
        puts (s);
    getch();
}
void push (const char *s)
{
    struct node *pnode;
    /* xin biến động chỉ tới node */
    pnode = (struct node *) malloc ( sizeof( struct node ) );
    if ( pnode == (struct node *) NULL )
    {
        printf (Khong du bo nho \n);
        exit (0);
    }
    /* xin biến động để lưu thông tin */
    pnode->message = (char *) malloc ( strlen(s) + 1 );
    if ( pnode->message == (char *) NULL )
    {
        printf (Khong du bo nho \n);
        exit (0);
    }
    /* lưu thông tin vào stack */
    strcpy ( pnode->message, s);
}

```

nhập trị vào stack

```

/* Tạo mốc nối mới */
pnode->next = head;
head = pnode;
}
char *pop ( char *s )
{
    struct node *temp_node;
    if ( head == (struct node *) NULL )
    {
        strcpy ( s, (char *) NULL );
        return ( (char *) NULL );
    }
    else
    {
        strcpy ( s, head->message );
        temp_node = pnode->next;
        free(pnode);
        pnode = temp_node;
        return s;
    }
}
char *look ( char *s )
{
    if ( head == (struct node *) NULL )
    {
        printf ("stack rong \n");
        strcpy ( s, (char *) NULL );
        return ( (char *) NULL );
    }
    else
    {
        strcpy ( s, head->message );
        return s;
    }
}
void init (struct node *head)
{
    struct node *pnode;
    for(pnode = head; pnode != (struct node *) NULL; )
    {

```

```

    struct node *temp_node;
    free (pnode->message);
    temp_node = pnode -> next;
    free(pnode);
    pnode = temp_node;
}
head = (struct node *)NULL;
}

```

Chương trình cho xuất liệu ví dụ:

Moi ban nhap mot chuoi: Doan Si Tri

Moi ban nhap mot chuoi: Ngo Hung Phuong

Moi ban nhap mot chuoi: Dang Thanh Tin

Moi ban nhap mot chuoi:

Cac chuoi trong stack la:

Dang Thanh Tin

Ngo Hung Phuong

Doan Si Tri

13.12.2 Danh sách liên kết là queue

Khác với danh sách liên kết dạng stack, danh sách liên kết dạng queue lưu trữ liệu vào theo thứ tự dữ liệu được nhập vào queue. Việc quản lý queue bằng danh sách liên kết tỏ ra linh động và hữu hiệu hơn so với việc dùng mảng thường.

Cũng tương tự như stack, mỗi thành phần của queue là một node có hai thành phần: phần thông tin info và phần để liên kết, tạm gọi là phần link hay next.

Để quản lý queue ta cần hai biến, một biến pointer first trỏ đến node ở đầu queue, một trỏ đến node ở cuối queue, last. Ví dụ các cấu trúc cần khai báo cho queue để quản lý các thông tin là chuỗi:

```

struct queue
{
    char *message;
    struct queue *next;
};

struct q_head
{

```

```

    struct queue *first;
    struct queue *last;
};


```

Như vậy để quản lý queue, cần khai báo đầu chương trình

```
struct q_head head;
```

Các thao tác cần thực hiện trên queue:

1- Thêm một thông tin vào queue - Thao tác add

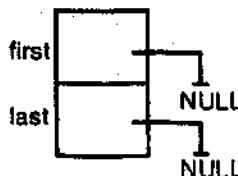
Thao tác này cho phép thêm một thông tin mới vào queue, trong hàm thực hiện thao tác này ta cần thực hiện các việc sau:

- Xin một biến động để chèp thông tin vào vùng info của nó
- Móc biến động này vào queue

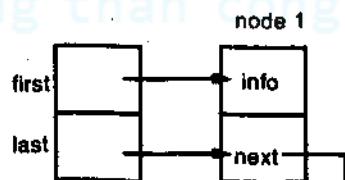
Node mới thêm vào này được đặt cuối queue, nó được chỉ bởi biến last.

Ta có thể xem thao tác add qua hình vẽ sau:

Trước khi thêm node đầu tiên



Sau khi thêm node đầu tiên



Ví dụ sau đây cho hàm add() đối với dữ liệu là một chuỗi, trong đó phần info có tên cụ thể là message, phần link có tên cụ thể là next.

```

void add (const char *s)
{
    struct queue *pnode;
    /* xin biến động chỉ tới node */
    pnode = (struct queue *) malloc ( sizeof( struct queue ) );
    if ( pnode == (struct queue *) NULL )
    {
        printf ("Khong du bo nho \n");
        exit (0);
    }
    /* xin biến động để lưu thông tin */
    pnode->message = (char *) malloc ( strlen(s) + 1 );
    if (pnode->message == (char *) NULL)
    {
        printf ("Khong du bo nho \n");
        exit (0);
    }
    /* lưu thông tin vào queue */
    strcpy ( pnode->message,s );
    pnode->next =( struct queue *) NULL;
    /* Tạo mốc nối mới */
    if ( head.first == ( struct queue *) NULL )
    {
        head.first = pnode;
        head.last = pnode;
    }
    else
    {
        head.last->next = pnode;
        head.last = pnode;
    }
}

```

2- Lấy một thông tin khỏi queue - Thao tác delete

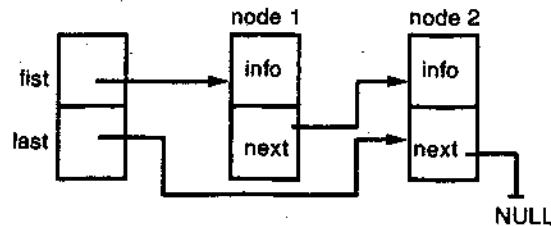
Thao tác này cho phép lấy một thông tin từ queue, trong hàm thực hiện thao tác này ta cần thực hiện các việc sau:

- Lấy thông tin trong vùng info lưu vào đối số của hàm
- Giải phóng biến động là node đã bị lấy thông tin

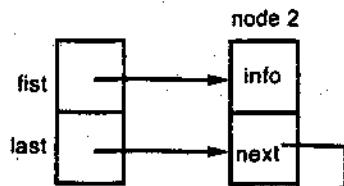
- Tạo mối nối trong queue

Ta có thể xem thao tác delete qua hình vẽ sau:

Trước khi lấy ra node đầu tiên



Sau khi lấy node đầu tiên ra



Ví dụ sau đây cho hàm delete() đối với dữ liệu là một chuỗi, trong đó phần info có tên cụ thể là message, phần link có tên cụ thể là next.

```
char *delete ( char *s )
{
    struct queue *temp_node;
    if ( head.first != (struct queue *) NULL )
    {
        strcpy ( s, head.first->message );
        free ( head.first->message );
        temp_node = head.first;
        head.first = temp_node->next;
        free( (char *) temp_node );
        return ( s );
    }
    else
    {
        strcpy ( s, (char *) NULL );
        return ( (char *) NULL );
    }
}
```

3- Xem trị hiện hành của phần tử đầu của queue

Thao tác này chỉ đơn giản là xem thông tin của phần tử hiện hành cần truy xuất trong queue có trị như thế nào. Để thực hiện thao tác này, ta cần thiết kế hàm look() như sau:

```
char *look ( char *s )
{
    if ( head.first == (struct node *) NULL )
    {
        strcpy ( s, (char*)NULL );
        return ( (char *) NULL );
    }
    else
    {
        strcpy ( s, head.first->message );
        return s;
    }
}
```

4- Khởi động queue

Khi sử dụng queue ta cần khởi động queue, nếu queue đang có sẵn, hàm khởi động cần xóa các phần tử cũ, và khởi động biến first và last trị khởi thủy là NULL, còn nếu queue chưa có thì việc khởi động chỉ đơn giản là khởi động trị cho first và last trị đầu là NULL. Hàm này có thể được viết như sau:

```
void init (struct q_head *head)
{
    struct queue *q_pointer, *next_node;
    for (q_pointer = head->first; q_pointer != (struct queue *)NULL;
         q_pointer = next_node )
    {
        next_node = q_pointer ->next;
        free (q_pointer->message);
        free( (char *) q_pointer);
    }
    head.first = (struct queue *)NULL;
    head.last = (struct queue *)NULL;
}
```

Xét chương trình ví dụ sau đây:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <alloc.h>
#include <process.h>
struct queue
{
    char *message
    struct queue *next;
};
struct q_head
{
    struct queue *first;
    struct queue *last;
};
struct q_head head;
void add (const char *s);
char *delete ( char *s );
char *look (char *s);
void init (void);
main()
{
    char s[81];
    clrscr();
    init ();           → khởi động queue
    do
    {
        printf ("Moi ban nhap mot chuoi: ");
        gets(s);
        if ( strcmp (s, (char *)NULL) != 0 )
            add (s);
    }
    while ( strcmp (s, (char *)NULL));
    printf ("Cac chuoi trong queue la: \n");
    for ( ; delete (s) != (char *)NULL ; )   → in trj từ queue
        puts (s);
}
```

nhập trj vào queue

```
getch();
}

void add (const char *s)

{
    struct queue *pnode;
    /* xin bien dong chi toi node */
    pnode = (struct queue *) malloc ( sizeof( struct queue ) );
    if ( pnode == (struct queue *) NULL )
    {
        printf ("Khong du bo nho \n");
        exit (0);
    }
    /* xin bien dong de luu thong tin */
    pnode->message = (char *) malloc (strlen(s) + 1);
    if (pnode->message == (char *) NULL)
    {
        printf ("Khong du bo nho \n");
        exit (0);
    }
    /* luu thong tin vao queue */
    strcpy ( pnode->message,s);
    pnode->next= ( struct queue *) NULL;
    /* Tạo mốc nối mới */
    if ( head.first == ( struct queue *) NULL )
    {
        head.first = pnode;
        head.last = pnode;
    }
    else
    {
        head.last->next = pnode;
        head.last = pnode;
    }
}

char *delete ( char *s )
```

```

{
    struct queue *temp_node;
    if ( head.first != (struct queue *) NULL )
    {
        strcpy (s, head.first->message);
        free (head.first->message);
        temp_node = head.first;
        head.first = temp_node->next;
        free( (char *) temp_node );
        return ( s );
    }
    else
    {
        strcpy (s, (char *)NULL);
        return ( (char *) NULL );
    }
}
char *look ( char *s )
{
    if ( head.first == (struct queue *) NULL )
    {
        strcpy (s, (char *)NULL);
        return ( (char *) NULL );
    }
    else
    {
        strcpy (s, head.first->message);
        return s;
    }
}
void init (void)
{
    struct queue *q_pointer, *next_node;
    for (q_pointer = head.first; q_pointer != (struct queue *)NULL;
        q_pointer = next_node )

```

```
{  
    next_node = q_pointer->next;  
    free (q_pointer->message);  
    free( (char *) q_pointer);  
}  
head.first = (struct queue *)NULL;  
head.last = (struct queue *)NULL;  
}
```

Chương trình cho xuất liệu ví dụ:

Moi ban nhap mot chuoi: Dang Thanh Tin

Moi ban nhap mot chuoi: Doan Si Tri

Moi ban nhap mot chuoi: Ngo Hung Phuong

Moi ban nhap mot chuoi:

Cac chuoi trong queue la:

Dang Thanh Tin

Doan Si Tri

Ngo Hung Phuong

Ngoài ra, còn có các cấu trúc dữ liệu khác, như cây (*tree*), danh sách vòng... xin xem chương 19 của tài liệu này.

BÀI TẬP CUỐI CHƯƠNG

- 13.1** Viết chương trình với một hàm cho phép truy xuất chuỗi trong stack (danh sách liên kết và mảng) và in ra màn hình thông tin theo thứ tự alphabet.
- 13.2** Dùng cấu trúc dữ liệu queue dạng danh sách liên kết, tính biểu thức dạng đa thức bất kỳ sau:

$$f(x) = a_0x^n + a_1x^{n-1} + \dots + a_{n-1}x^1 + a_n$$

trong phần thông tin có hai vùng biến

- hệ số
- số mũ

- 13.3** Viết chương trình với một hàm duyệt toàn bộ các phần tử trong queue, trả về số phần tử trong queue.

- 13.4** Viết chương trình tạo một danh sách liên kết lưu các thông tin là các số nguyên theo thứ tự từ lớn tới nhỏ. Thiết kế hàm insert() cho phép chèn một phần tử lưu thông tin số vào vị trí có thứ tự phù hợp trong chuỗi.

- 13.5** Viết chương trình nhập vào một số số nguyên (chưa biết có bao nhiêu số nguyên). Loại bỏ các số nguyên bị lặp lại. In ra dãy số mới này.

Ví dụ: Nhập: 5 4 10 8 5 4 10 2 8

In ra: 5 4 10 8 2

Chương 14

CÁC KIỂU DỮ LIỆU CÓ CẤU TRÚC VÀ KIỂU DỮ LIỆU TỰ ĐỊNH NGHĨA

Khi lập trình, nhiều lúc ta cần có một biến trong đó có các biến thành phần, mỗi thành phần lại có kiểu khác nhau. Để khai báo cho các biến phức hợp như vậy, C đưa ra các kiểu dữ liệu có cấu trúc: struct, union; và kiểu dữ liệu tự định nghĩa: enum. Trong các mục sau đây, các kiểu dữ liệu phức hợp trên sẽ được trình bày cụ thể, ngoài ra sẽ có một số ví dụ thực tế thường gặp cũng được nêu ra.

14.1 KIỂU STRUCT

14.1.1 Khái niệm - Khai báo struct

Struct (tạm dịch là cấu trúc) là một kiểu dữ liệu phức hợp được tạo từ các kiểu dữ liệu khác, các kiểu dữ liệu này được sử dụng khai báo cho các biến thành phần của biến kiểu struct.

Việc khai báo một cấu trúc với tên các biến thành phần và kiểu của chúng được gọi là định nghĩa một cấu trúc.

Cú pháp để định nghĩa một cấu trúc như sau:

```
struct tên_cấu_trúc
{
    Khai báo các biến thành phần
};
```

với:

- **struct** là từ khóa của C, xác định kiểu struct.
- **tên_cấu_trúc** là một danh hiệu không chuẩn được dùng làm tên cấu trúc, tên này có thể có hoặc không.

- phần khai báo các biến thành phần là một danh sách các khai báo biến của các thành phần tạo nên cấu trúc này.

Ví dụ 14.1

Khai báo một struct có các thành phần xác định một sinh viên:

- mã số sinh viên
- họ tên sinh viên
- tuổi
- địa chỉ

Cấu trúc sẽ có cấu tạo như sau:

```
struct sinh_vien
```

```
{
```

```
    char ma_so[10];
```

```
    char ho_ten[40];
```

```
    int tuoi;
```

```
    char dia_chi[80];
```

```
}
```

Với khai báo struct trên, sinh_vien là tên cấu trúc, các danh hiệu ma_so, ho_ten, tuoi, dia_chi là tên các biến thành phần của cấu trúc. Như vậy, một biến bình thường có tên là **tuoi** cũng được sử dụng bình thường mà không gây ra nhầm lẫn gì với thành phần có tên **tuoi** của cấu trúc sinh_vien ở trên, vì C sẽ phân biệt được chúng theo cách mà các tên này được sử dụng.

Việc định nghĩa một cấu trúc như trên chỉ mới báo cho bộ dịch C biết rằng có một cấu trúc đã được định nghĩa, chứ chưa cấp chỗ cho biến cấu trúc. Để có được một biến cấu trúc ta phải khai báo biến với kiểu là cấu trúc đó.

Cú pháp của một khai báo biến cấu trúc giống như khai báo biến bình thường:

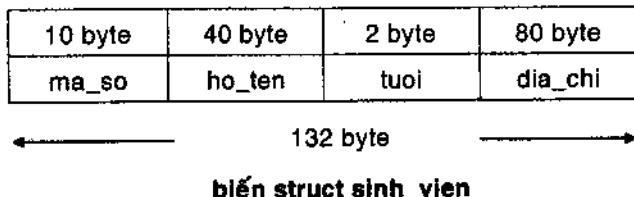
```
struct tên_struct tên_bien;
```

Ví dụ 14.2

Để khai báo các biến struct sinh_vien với định nghĩa đã có như trên, ta có lệnh sau trong chương trình

```
struct sinh_vien sv1, sv2;
```

Sau khai báo trên, các biến ***sv1*** và ***sv2*** đều được cấp chỗ trong bộ nhớ, mỗi biến như thế này sẽ có cấu trúc như hình sau:



Đi nhiên, một khai báo biến cấu trúc chỉ hợp lệ khi cấu trúc dùng trong khai báo này đã được định nghĩa trước. Tuy nhiên, C cho phép ngay sau định nghĩa cấu trúc (tức là sau dấu đóng ngoặc nhọn kết thúc cấu trúc) ta có thể khai báo các biến có cấu trúc này.

Ví dụ 14.3

Ta có thể khai báo biến cấu trúc ngay sau định nghĩa cấu trúc:

```
struct sinh_vien
```

```
{
    char ma_so[10];
    char ho_ten[40];
    int tuoi;
    char dia_chi[80];
} sv1, sv2;
```

đây có thể được xem như một sự kết hợp định nghĩa cấu trúc và khai báo biến cấu trúc.

Một biến kiểu struct nếu là biến toàn cục hay biến tĩnh thì có thể được khởi động trị. Việc khởi động trị cho biến cấu trúc cũng tương tự như khởi động mảng, được thực hiện bằng phép gán biến đó, tương ứng với các biến thành phần là các trị cần gán vào cho chúng. Các trị này cách nhau bằng dấu phẩy (,) và được đặt giữa cặp dấu ngoặc nhọn.

Ví dụ 14.4

```
struct sinh_vien sv1 = { "4950897", "Tran van Vinh", 21, "42 Truong Cong Dinh
p.13 q.TB";}
```

Để truy xuất một thành phần của biến cấu trúc, C có toán tử chấm(.) để lấy từng thành phần. Toán tử này được đặt sau một tên biến cấu trúc và theo sau nó là tên của biến thành phần cần truy xuất.

Ví dụ 14.5

```
strcpy(sv1.ma_so, "4950897");
strcpy(sv1.ho_ten, "Tran van Dinh");
sv1.tuoi = 21;
strcpy(sv1.dia_chi, "42 Truong Cong Dinh p.13 q.TB")
```

Một cấu trúc có thể có thành phần là một cấu trúc khác, khi đó ta có struct lồng nhau. Việc truy xuất đến các biến thành phần trong các struct lồng nhau cũng sử dụng toán tử chấm đến thành phần sau cùng.

Ví dụ 14.6

Trong struct sinh_vien trên, ta thêm một thành phần là một struct để xác định ngày tháng năm sinh của sinh viên.

```
struct ngay
{
    int day;
    int month;
    int year;
};

struct sinh_vien
{
    char ma_so[10];
    char ho_ten[40];
    struct ngay ngay_sinh;
    char dia_chi[80];
} sv;
```

struct ngay được khai báo trước struct sinh_vien và được sử dụng để khai báo cho biến thành phần ngay_sinh của struct sinh_vien

Khi đó, các biến thành phần có thể truy xuất, ví dụ:

```
strcpy(sv.ma_so, "4050897");
strcpy(sv.ho_ten, "Tran Van Vinh");
strcpy(sv.dia_chi, "42 Truong Cong Dinh p.13 q.TB");
sv.ngay_sinh.day = 31;
sv.ngay_sinh.month = 12;
sv.ngay_sinh.year = 1967;
```

Để thực hiện việc gán từ biến cấu trúc này sang biến cấu trúc khác, C cho phép gán các cấu trúc cùng kiểu cho nhau qua tên biến cấu trúc thay vì phải gán từng thành phần cho nhau.

Ví dụ 14.7

```
struct sinh_vien sv1, sv2;
strcpy (sv1.ma_so, "4950897");
strcpy (sv1.ho_ten, "Tran Van Dinh");
strcpy (sv1.dia_chi, "42 Truong Cong Dinh p.13 q.TB");
sv1.ngay_sinh.day = 31;
sv1.ngay_sinh.month = 12;
sv1.ngay_sinh.year = 1967;
sv2 = sv1;
```

C cho phép struct là đối số của hàm. Việc gởi struct là đối số của hàm cũng được truyền theo tham số trị, như vậy hàm sẽ chép toàn bộ các trị trong các biến thành phần của biến struct là đối số thật sang cho đối số giả để tham khảo trị trong hàm. Mọi sự thay đổi trên biến struct trong hàm đều không thực hiện được.

Ví dụ 14.8 Xét chương trình ví dụ sau đây:

```
#include <stdio.h>
#include <conio.h>
struct sinh_vien
{
    char ma_so[10];
    char ho_ten[40];
    int tuoi;
    char dia_chi[80];
};
void in_tri_struct (struct sinh_vien a);
main()
{
    struct sinh_vien sv = {"4954098",
                          "Tran Quang Trung",
                          25,
                          "1030 Hau Giang Q.6"};
    clrscr();
    in_tri_struct(sv);
    getch();
```

```

}

void in_tri_struct (struct sinh_vien a)
{
    printf ("Cac bien thanh phan co tri: \n");
    printf ("Ma_so: %s \n", a.ma_so);
    printf ("Ho_ten: %s \n", a.ho_ten);
    printf ("Tuoi: %d \n", a.tuoi);
    printf ("Dia_chi: %s \n", a.dia_chi);
}

```

Chương trình này cho xuất liệu là:

Cac bien thanh phan co tri:

Ma_so: 4954098

Ho_ten: Tran Quang Trung

Tuoi: 25

Dia_chi: 1030 Hau Giang Q.6

Các thành phần của biến struct cũng là biến bình thường, nên ta có thể lấy địa chỉ của chúng, địa chỉ này là một hằng pointer chỉ đến thành phần tương ứng. Chương trình sau đây cho thấy các thao tác trên các thành phần biến struct.

Ví dụ 14.9 Xét chương trình ví dụ sau:

```

#include <stdio.h>
#include <conio.h>
struct so_phuc
{
    double r, i;
};

main()
{
    struct so_phuc a, b, kq;
    clrscr();
    printf ("Chuong trinh tinh toan tren bien struct \n");
    printf ("Moi nhap phan thuc va ao cua bien a: ");
    scanf ("%lf %lf", &a.r, &a.i);
    printf ("Moi nhap phan thuc va ao cua bien b: ");
    scanf ("%lf %lf", &b.r, &b.i);
    kq.r = a.r + b.r;
}

```

```

kq.i = a.i + b.i;
printf ("Ket qua cong hai so phuc: %5.2f + i*%-.5.2f\n", kq.r, kq.i);
getch();
}

```

Chương trình cho xuất liệu, ví dụ:

Chuong trinh tinh toan tren bien struct

Moi nhap phan thuc va ao cua bien a: 4 7

Moi nhap phan thuc va ao cua bien b: 4 2

Ket qua cong hai so phuc: 8.00 + i* 9.00

Trong chương trình trên, phép lấy địa chỉ của các thành phần của struct được thực hiện bằng toán tử địa chỉ (&) qua biểu thức &a,..., trong biểu thức &a., toán tử chấm(.) có độ ưu tiên cao hơn toán tử địa chỉ, thay vì viết &(a.r) ta có thể viết ngắn hơn là &a.r.

Chú ý: Kiểu struct có thể được lấy kích thước tính theo byte nhờ toán tử sizeof, ví dụ:

sizeof (struct sinh_vien)

14.1.2 Mảng các struct

C cho phép khai báo mảng các struct như là mảng các kiểu dữ liệu bình thường khác. Cú pháp khai báo mảng các struct:

struct ten_cau_truc ten_mang [kich_thuoc]:

Ví dụ 14.10

Cho khai báo sau:

struct sinh_vien sv[50];

Với khai báo trên, mảng sv có 50 phần tử, mỗi phần tử là một biến kiểu struct sinh_vien, và ta hoàn toàn có thể truy xuất đến từng thành phần trong struct qua toán tử chấm(.) bình thường.

Ví dụ 14.11

strcpy (sv[0].ho_ten, "Dang thanh Tin");

sv[0].tuoi = 28;

Chương trình ví dụ sau đây cho thấy cách truy xuất và làm việc trên mảng các struct.

Ví dụ 14.12

```

#include <stdio.h>
#include <conio.h>
#include <string.h>
#define MAX 50
struct sinh_vien
{
    char ma_so[10];
    char ho_ten[40];
    int tuoi;
    char dia_chi[80];
};
main()
{
    struct sinh_vien sv[MAX];
    int i = 0, shs = 0;
    char ma_so[10];
    clrscr();
    printf ("Chuong trinh thu mang struct \n");
    do
    {
        printf ("Sinh vien %d \n", i + 1);
        printf ("Ma so: ");
        gets (sv[i].ma_so);
        if (strcmp (sv[i].ma_so, (char *)NULL) != 0)
        {
            printf ("Ho ten: ");
            gets (sv[i].ho_ten);
            scanf ("%d", &sv[i].tuoi);
            flushall();
            printf ("Dia chi: ");
            gets (sv[i].dia_chi);
        }
    } while (strcmp (sv[i++].ma_so, (char *)NULL) != 0);
    shs = --i; /* trừ i đi 1 để ra số sinh viên thực */
    printf ("Trong danh sach co %d sinh vien \n", shs);
    printf ("Moi ban nhap ma so sinh vien can tim: ");
    gets (ma_so);
}

```

```

i = 0;
while (strcmp (sv[i].ma_so, ma_so) != 0 && i < shs)
    i++;
/* tìm thấy mã số thì thoát ra, khi đó i < shs,
nếu không thấy mã số, khi thoát ra, i == shs */
if (i == shs )
    printf ("Ma so nay khong ton tai! \n");
else
{
    printf ("Ma so sinh vien la: %s \n", sv[i].ma_so);
    printf ("Ho ten sinh vien la: %s \n", sv[i].ho_ten);
    printf ("Tuoi: %d \n", sv[i].tuoi);
    printf ("Dia chi sinh vien la: %s \n", sv[i].dia_chi);
}
getch();
}

```

Chương trình cho xuất liệu, ví dụ:

Chuong trinh thu mang struct

Sinh vien 1

Ma so: 4951234

Ho ten: Tran Van Thinh

Tuoi: 21

Dia chi: 123 Hai Ba Trung

Sinh vien 2

Ma so: 4942345

Ho ten: Dang Van Sau

Tuoi: 23

Dia chi: 152 Hau Giang

Sinh vien 3

Ma so: 4953456

Ho ten: Lam Van Tai

Tuoi: 24

Ma so:

Trong danh sach co 3 sinh vien

Moi ban nhap ma so sinh vien can tim: 4942345

Ma so sinh vien la: 4942345

Ho ten sinh vien la: Dang Van Sau

Tuoi: 23

Dia chi sinh vien la: 152 Hau Giang

14.1.3 Pointer tới một struct

Một biến struct cũng là một biến trong bộ nhớ, nó có một địa chỉ xác định sau khai báo, ta có thể lấy địa chỉ này bằng toán tử lấy địa chỉ (&). Địa chỉ này là một hằng pointer chỉ đến đầu của cấu trúc. Ta có thể khai báo một biến pointer chỉ đến một cấu trúc, biến pointer này lưu địa chỉ đầu của biến struct đã được khai báo.

Cú pháp khai báo biến pointer này như sau:

```
struct tên_cấu_trúc *tên_pointer;
```

Ví dụ 14.13

```
struct sinh_vien *psv;
```

Khi đó, bản thân pointer *psv* này được cấp chỗ trong bộ nhớ, và được ghi nhận là pointer chỉ đến struct *sinh_vien*, còn đối tượng mà pointer này chỉ đến vẫn chưa xác định, và do đó chưa thể dùng được đối tượng đó.

Trị có thể gán được vào cho biến *psv* có thể là địa chỉ của biến thuộc kiểu struct *sinh_vien*, hoặc là tên mảng các struct *sinh_vien*.

Ví dụ 14.14

```
struct sinh_vien a, *psv;
```

```
psv = &a;
```

hoặc

```
struct sinh_vien sv[MAX], *psv;
```

```
psv = sv;
```

Việc sử dụng pointer chỉ đến cấu trúc thường được dùng để gửi cấu trúc cho hàm, điều này làm tiết kiệm được thời gian để chép lại toàn bộ cấu trúc đó để gởi cho hàm. Mặt khác, ta có thể thay đổi nội dung của cấu trúc từ trong hàm.

Việc truy xuất đến một thành phần của một cấu trúc thông qua một pointer được thực hiện bằng toán tử lấy thành phần của đối tượng của pointer, ký hiệu là -> (có thể gọi là toán tử mũi tên).

Ví dụ 14.15

```
printf ("Ho ten sinh vien: %s \n", psv -> ho_ten);
```

hay ta có thể viết

```
printf ("Ho ten sinh vien: %s \n", (*psv).ho_ten);
```

Tuy nhiên, dạng viết sau (*psv).ho_ten hiện nay ít được sử dụng, dạng này cho thấy ta đang truy xuất trên thành phần *ho_ten* của đối tượng đang được pointer *psv* chỉ đến.

Ví dụ 14.16

Hãy xem cách sử dụng của pointer chỉ đến mảng các struct trong chương trình sau, chương trình này là sự sửa đổi chương trình trên.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#define MAX 50
struct sinh_vien
{
    char ma_so[10];
    char ho_ten[40];
    int tuoi;
    char dia_chi[80];
};

int nhap_mang_struct (struct sinh_vien *psv );
void xuat_struct ( struct sinh_vien sv );
main()
{
    struct sinh_vien sv[MAX];
    int i = 0, shs = 0;
    char ma_so[10];
    clrscr();
    printf ("Chuong trinh thu mang struct \n");
    shs = nhap_mang_struct(sv);
    printf ("Trong danh sach co %d sinh vien \n", shs);
    printf ("Moi ban nhap ma so sinh vien can tim: ");
    gets (ma_so);
    i = 0;
    while (strcmp (sv[i].ma_so, ma_so) != 0 && i < shs)
        i++;
    if (i == shs)
        printf ("Ma so nay khong ton tai ! \n");
    else
```

```

        xuat_struct (sv[i]);
    getch();
}

int nhap_mang_struct ( struct sinh_vien *psv)
{
    int i = 0;
    do
    {
        printf ("Sinh vien %d \n", i + 1);
        printf ("Ma so: ");
        gets ( (psv + i) ->ma_so);
        if ( strcmp (psv + i)->ma_so, (char *) NULL) != 0
        {
            printf ("Ho ten: ");
            gets ( (psv + i)->ho_ten);
            printf ("Tuoi: ");
            scanf ("%d", & (psv + i)->tuoi);
            fflush();
            printf ("Dia chi: ");
            gets (( psv + i)->dia_chi);
        }
        } while ( strcmp ((psv + i++)->ma_so, (char *) NULL) != 0);
    return (i);
}

void xuat_struct (struct sinh_vien sv)
{
    printf ("Ma so sinh vien la: %s \n", sv.ma_so);
    printf ("Ho ten sinh vien la: %s\n", sv.ho_Ten);
    printf ("Tuoi: %d \n", sv.tuoi);
    printf ("Dia chi sinh vien la: %s \n", sv.dia_chi);
}

```

Chương trình này cho xuất liệu tương tự như chương trình trong ví dụ 14.12.

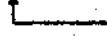
Trong chương trình trên, để truy xuất đến từng phần tử trong mảng các struct, ta sử dụng con trỏ (*psv + i*) để chỉ đến từng struct trong mảng. Để truy xuất đến từng thành phần trong struct, ta sử

dụng toán tử mũi tên \rightarrow , ví dụ $(psv + i) \rightarrow ho_ten$. Tuy nhiên, đối với mảng một chiều, ta hoàn toàn có thể dùng biến con trả theo kiểu mảng, có nghĩa là thay vì để $(psv + i) \rightarrow ho_ten$, ta có thể để là $psv[i].ho_ten$.

Bình thường, không thể định nghĩa một cấu trúc lại lồng chính cấu trúc đó được, vì khi đó xảy ra sự lồng nhau vô hạn. Tuy nhiên, C lại cho phép khai báo struct mà trong các thành phần của nó lại có các pointer chỉ đến một cấu trúc cùng kiểu. Đây là một "cấu trúc tham khảo đến chính nó".

Ví dụ 14.17

```
struct node
{
    char message[81];
    struct node *next;
};
```

 *next là một pointer chỉ đến
một cấu trúc cùng kiểu*

Cấu trúc này có thể được sử dụng để tạo ra các cấu trúc dữ liệu phức tạp như danh sách liên kết (*linked list*) dạng stack hay dạng queue hoặc cây (*tree*) với nhiều ứng dụng trong lập trình. Các cấu trúc dữ liệu này đã được trình bày trong chương 13 - Pointer, đề nghị độc giả xem lại.

14.1.4 Struct dạng field

C cho phép ta khai báo các thành phần của struct theo bit hoặc một nhóm bit. Một thành phần như vậy được gọi là một **field** (tạm dịch là **vùng**).

Khai báo một cấu trúc có các thành phần là các field được thực hiện tương tự như khai báo cấu trúc bình thường, cú pháp cụ thể là:

```
struct tên_cấu_trúc
{
    kiểu tên_vùng 1: số_bit1;
    kiểu tên_vùng 2: số_bit2;
    ...
} tên_biel;
```

với:

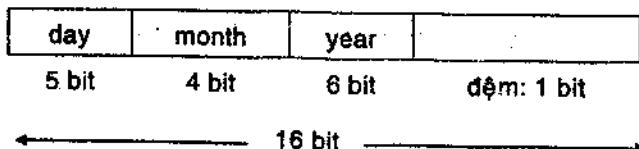
- **kiểu** chỉ có thể là unsigned, signed hoặc int, là kiểu của field tương ứng. Field này có thể lưu số có dấu hay không dấu.
- **tên_vùng** 1, tên_vùng 2,... là các danh hiệu không chuẩn, tên của các field thành phần của struct. Một field có thể không có tên, khi đó số bit được khai báo sau chỉ để đếm cho hết 16 bit của một int hay unsigned. Field này chúng ta không thể truy xuất được.
- **số_bit1, số_bit2,...** là số nguyên từ 0-16, quy định số bit được nhóm thành vùng 1, vùng 2.... Nếu là 0, các bit còn lại của int hoặc unsigned đang cấp bit cho các field trên sẽ được bỏ qua và vùng kế tiếp được cấp chỗ qua phần tử int kế. Như vậy, vùng có số bit bằng 0 thường chỉ dùng để đếm, bỏ qua các bit cuối một int.

Ví dụ 14.18

Một biến có cấu trúc như sau có thể dùng để lưu trữ thông tin liên quan đến ngày tháng năm

```
struct date
{
    unsigned day: 5; /* Field có 5 bit: 0-31 */
    unsigned month: 4; /* Field có 4 bit: 0-15 */
    unsigned year: 6; /* Field có 6 bit: 0-63 */
    int: 0;
} ngay;
```

Khi đó biến ngày được cấp chỗ và các thành phần sẽ được sắp xếp như sau:



Trong struct trên, vùng **day** có chiều dài 5 bit, như vậy nó có thể chứa được một giá trị trong khoảng 0 đến $2^5 - 1 = 31$, vì các ngày trong tháng chỉ đi từ 1 đến 31 nên vùng **day** chứa được một giá trị quy định cho ngày.

Tương tự, với tháng, tháng đi từ 1 đến 12, chiều dài field **month** 4 bit là có thể chứa được giá trị quy định cho tháng.

Đối với năm, nếu giả sử rằng 0 tương ứng với năm 1980 thì field **year** có 6 bit, tương ứng chúng ta có thể biểu diễn được 64 năm từ 1980 - 2043. Như vậy, thật sự vùng year chỉ lưu trữ trị từ 0 - 63, nên trước khi in ra năm, chúng ta phải cộng vùng này với 1980 mới có được năm đúng.

Với khai báo struct như trên, biến **ngay** chỉ chiếm một byte trong bộ nhớ, vì vậy giảm được đáng kể số byte cần dùng để lưu trữ các thông tin này như theo dạng struct bình thường.

Ví dụ 14.19

```
ngay.day = 31;
ngay.year = năm - 1980;
```

Trong một struct vừa có thể có các thành phần là field vừa có thể có các thành phần có kiểu bình thường khác. Khi đó, các thành phần bình thường sẽ được cấp chỗ trong các int khác với các thành phần field. Do đó, khi khai báo định nghĩa cấu trúc trong đó có nhiều field, ta cần sắp xếp cho các field nằm gần nhau, sao cho số bit các field gần nhau gần chẵn một int để đỡ tốn bộ nhớ.

Chú ý:

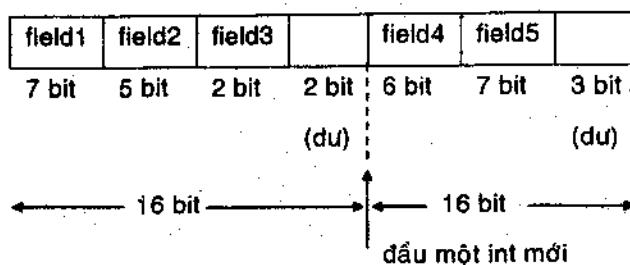
- 1- Mỗi vùng chỉ có thể dài tối đa 16 bit (một int) và được cấp chỗ trong một int, chứ không thể nằm trên hai int khác nhau được, phần bit dư trong một int không được cấp chỗ cho field nào cả (vì không đủ cho một field hoặc hết struct) sẽ được C tự động bỏ qua.

Ví dụ 14.20

Khi khai báo

```
struct vi_du
{
    unsigned field1: 7;
    unsigned field2: 5;
    unsigned field3: 2;
    unsigned field4: 6;
    unsigned field5: 7;
} vd;
```

thì bộ dịch C sẽ cấp chỗ như sau:



2- Sự phân bổ bit cho các field trong một int của struct (từ trái sang phải hay ngược lại) tùy thuộc vào quy định của bộ dịch ngôn ngữ. Do đó, khi viết chương trình ta nên khai báo tên field sao cho khi truy xuất ta không cần quan tâm đến thứ tự thật.

Ví dụ 14.21

Một biến có cấu trúc định nghĩa như sau:

```
struct byte
{
    unsigned bit0: 1;
    unsigned bit1: 1;
    unsigned bit2: 1;
    unsigned bit3: 1;
    unsigned bit4: 1;
    unsigned bit5: 1;
    unsigned bit6: 1;
    unsigned bit7: 1;
} byte1;
```

thì field bit0 được bộ dịch C cấp chỗ là bit có trọng số lớn nhất hay bit có trọng số nhỏ nhất trong biến byte 1 là không được biết trước và cũng không cần quan tâm, vì khi xử lý, ta luôn luôn xem field bit0 là bit có trọng số nhỏ nhất.

3- Kiểu field là một kiểu đặc biệt thường được cấp chỗ trong bộ nhớ, do đó, mọi thao tác thực hiện trên biến kiểu field có liên quan đến địa chỉ đều không được thực hiện. Như vậy, ta không thể lấy địa chỉ của một biến kiểu field, khai báo mảng các field hay trả trị của field về cho hàm qua lệnh return. Các thao tác này đều phải qua biến trung gian.

Ví dụ 14.22 Hãy xem đoạn chương trình nhập ngày như sau:

```
struct date ngay;
unsigned tam_d, tam_m, tam_y;
printf ("Nhập ngày: ");
scanf ("%d", &tam_d);
ngay.day = tam_d;
printf ("Nhập tháng: ");
scanf ("%d", &tam_m);
ngay.month = tam_m;
printf ("Nhập năm: ");
scanf ("%d", &tam_y);
ngay.year = tam_y - 1980;
```

Ví dụ 14.23

```
unsigned tam_d;
tam_d = ngay.day;
return (tam_d);
```

4- Nếu field được khai báo là có dấu (signed) mà chỉ có một bit thì nó chỉ có hai giá trị là 0 và -1.

14.2 KIỂU UNION

Trong ngôn ngữ C có kiểu dữ liệu union (tạm dịch là kiểu hợp nhất), đây là một kiểu dữ liệu đặc biệt mà nếu được khai báo thì ứng với một vùng nhớ, giá trị ở mỗi thời điểm khác nhau thì có thể có kiểu khác nhau tùy vào việc sử dụng biến thành phần trong nó.

Để khai báo và định nghĩa một union, ta cũng theo cú pháp tương tự như struct:

```
union tên_union
{
    khai_báo_biến_thành_phần
};
```

với:

- **union** là một từ khóa, xác định đang có khai báo union.
- **tên_union** là một danh hiệu do lập trình viên tự đặt. Tên này được sử dụng để khai biến union.

- Khai báo biến thành phần là danh sách liệt kê các tên biến cùng các kiểu có thể sử dụng chung vùng nhớ được cấp.

Union là một kiểu dữ liệu đặc biệt, một biến thuộc kiểu này sẽ có các thành phần luôn luôn bắt đầu tại cùng một vị trí trong bộ nhớ. Điều đó có nghĩa là ở mỗi thời điểm sử dụng union, ta chỉ có thể sử dụng được một biến thành phần có kiểu xác định trong **union** và biến **union** có chiều dài là chiều dài của thành phần có kích thước lớn nhất trong các thành phần của union.

Ví dụ 14.24

Có khai báo union như sau:

```
union thu /* biến kiểu này chiếm 8 byte trong bộ nhớ */
```

```
{
```

```
    char c; /* biến kiểu này chiếm 1 byte trong bộ nhớ */
```

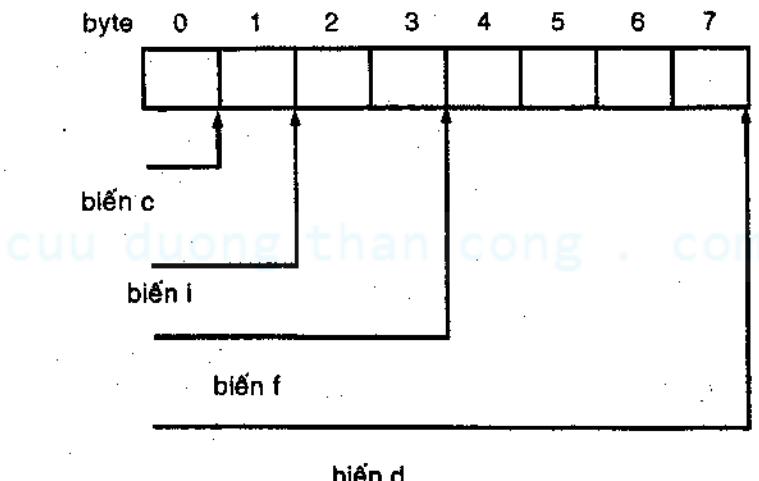
```
    int i; /* biến kiểu này chiếm 2 byte trong bộ nhớ */
```

```
    float f; /* biến kiểu này chiếm 4 byte trong bộ nhớ */
```

```
    double d; /* biến kiểu này chiếm 8 byte trong bộ nhớ */
```

```
};
```

thì một biến kiểu union thu có thể hiểu qua hình sau:



Tất cả các biến thành phần *c*, *i*, *f*, *d* đều cùng một vùng nhớ được cấp nên mọi phép gán, thay đổi trị trên một thành phần đều ảnh hưởng đến thành phần khác.

Cũng tương tự như struct, ta có thể khai báo một biến union ngay khi định nghĩa union đó, hoặc có thể sử dụng tên_union để khai báo theo cú pháp:

```
union tên_union
{
    khai_báo_biến_thành_phần
} biến, biến [...];
```

hoặc

```
union tên_union biến, biến [...];
```

Ví dụ 14.25

```
union thu
{
    char c;
    int i;
    float f;
    double d;
} a, b;
```

hoặc

```
union thu a, b;
```

Nếu **struct** là một cấu trúc gồm nhiều thành phần có kiểu khác nhau kết hợp lại, các thành phần này là độc lập nhau, việc sử dụng một thành phần không làm ảnh hưởng đến các thành phần khác của cùng **struct** thì biến **union** thực chất chỉ là biến một phần tử, nhưng ở mỗi thời điểm sử dụng khác nhau, phần tử này có thể được hiểu là một thành phần nào đó trong các thành phần đã định nghĩa của **union**. Và như vậy ta chỉ có một chỗ duy nhất, mà tại đó vào lúc này ta có thể lưu một trị có kiểu dữ liệu này, và lúc khác thì lại sử dụng chỗ đó cho một trị có kiểu dữ liệu khác. Như biến **a** được khai báo thuộc kiểu **union** trên có thể được sử dụng như một biến **char** nếu ta dùng thành phần **a.c** để tính toán, hoặc xem biến **a** như một số **double** nếu xử lý tính toán trên thành phần **a.d**.

Do việc sử dụng chung một vị trí cho các giá trị có thể có kiểu khác nhau, nên khi sử dụng union cần phải chú ý đến kiểu dữ liệu của trị đang làm việc để sử dụng đúng giá trị đang được lưu

trong union. Nếu cất vào union một giá trị double mà lại sử dụng giá trị đó như một int thì không thể nào thu được kết quả như đã gán vào được.

Ví dụ 14.26

```
a.d = 12.345;
printf ("Trị int trong union a là %d \n", a.i);
```

thì kết quả in ra sẽ không là 12.345 nữa.

Để truy xuất đến từng thành phần của một union, toán tử chấm (.) cũng được sử dụng như khi truy xuất struct.

Ví dụ 14.27

```
union thu a;
a.c = 'a';
```

Một biến union cũng được cấp chỗ trong bộ nhớ nên nó cũng có địa chỉ, địa chỉ này là một hằng pointer chỉ đến union, địa chỉ này thật sự cũng là địa chỉ đến từng thành phần của union. Ta có thể khai báo một biến pointer chỉ đến một union tương tự như khai báo pointer chỉ đến struct.

Ví dụ 14.28

```
union thu *pthu, a;
pthu = &a;
```

Khi đó, việc truy xuất đến một thành phần của union qua pointer cũng được thực hiện bằng toán tử mũi tên, để lấy thành phần của union đang được pointer chỉ đến.

Ví dụ 14.29

```
pthu->c = 'A';
```

Chương trình ví dụ sau đây minh họa cho cách sử dụng union là một thành phần của struct, trong struct ta có khai báo thêm một thành phần để lưu thông tin về kiểu dữ liệu đang sử dụng trong union, tại mỗi thời điểm, ta có thể kiểm tra thành phần này để biết kiểu đang lưu mà lấy ra cho đúng.

Ví dụ 14.30

```

#include <stdio.h>
#include <conio.h>
#include <string.h>
struct date
{
    char ngay;
    char thang;
    int nam;
};

struct fdi
{
    char type;
    union
    {
        char *sval; /* Kiểu chuỗi ký tự */
        double fval; /* Kiểu số thực */
        int ival; /* Kiểu số nguyên */
        struct date dval; /* Kiểu ngày tháng năm */
    } value;
};

main()
{
    struct fdi a;
    int x, y, x1, y1;
    clrscr();
    printf ("Ban thuc hien thao tac tren du lieu: ");
    x = wherex();
    y = wherey();
    printf ("\n1. Chuoi \n");
    printf ("2. So thuc \n");
    printf ("3. So nguyen \n");
    printf ("4. Ngay thang nam\n");
    x1 = wherex();
    y1 = wherey();
    gotoxy (x, y);
    a.type = getchar();
    /* nhap du lieu theo kieu da chon */
    switch (a.type)

```

```

{
    case '1':
        a.value.sval = "Chao ban \n";
        break;
    case '2':
        a.value.fval = 14.25;
        break;
    case '3':
        a.value.ival = 45;
        break;
    case '4':
        a.value.dval.ngay = 31;
        a.value.dval.thang = 12;
        a.value.dval.nam = 1967;
        break;
}
gotoxy (x1, y1);
/* xuat du lieu theo kieu dang luu */
switch (a.type)
{
    case '1':
        printf ("Chuoi dang luu la: ");
        puts (a.value.sval);
        break;
    case '2':
        printf ("Tri so thuc dang luu la %5.2f \n",a.value.fval);
        break;
    case '3':
        printf ("Tri so nguyen dang luu la %5d \n",a.value.ival);
        break;
    case '4':
        printf ("Ngay thang nam dang luu la: %d/%d%d\n");
        a.value.dval.ngay,a.value.dval.thang, a.value.dval.nam);
        break;
}
getch();
}

```

Chương trình sẽ cho xuất liệu, ví dụ:

Ban thực hiện thao tác trên dữ liệu: 4

1. Chuỗi
2. Số thực
3. Số nguyên
4. Ngày tháng năm

Ngày tháng năm đang lưu là: 31/12/1967

Trong chương trình trên, ta quy định thành phần ***type*** chứa các ký số '1', '2', '3' hoặc '4' sẽ tương ứng với các kiểu dữ liệu trong thành phần ***value*** là chuỗi, số thực, số nguyên hoặc struct ngày tháng năm.

Chú ý, kiểu union có thể được lấy kích thước tính theo byte qua toán tử **sizeof**, ví dụ:

sizeof (union thu)

14.3 KIỂU ENUM (*Enumerated*)

Sau này, ANSI đã đưa vào cho C thêm một kiểu dữ liệu là **enum**. Kiểu dữ liệu này tương tự như loại liệt kê trong ngôn ngữ Pascal, tức khi khai báo kiểu, ta cần liệt kê toàn bộ trị mà sau này một biến kiểu enum có thể nhận được.

Cú pháp để khai báo kiểu dữ liệu **enum** như sau:

```
enum tên_kiểu {
    danh_sách_trị
} biến, biến [...];
```

với **danh_sách_trị** liệt kê tất cả những danh hiệu đại diện cho các giá trị có thể được gán cho biến, mỗi tên ngăn cách nhau bằng dấu phẩy.

Ví dụ 14.31 Cho khai báo

```
enum weekday { sunday, monday, tuesday, wednesday, thursday, friday, saturday};
```

Khi đó, các tên được liệt kê ra tương ứng với các giá trị int từ 0, 1,... trở đi. Như vậy, một biến có trị kiểu **enum weekday** nếu nhận trị **sunday**, thì cũng có nghĩa nó được gán trị int là 0, **monday** là 1,..., **saturday** là 6.

Tuy nhiên, ta hoàn toàn có khả năng gán tên bắt đầu hoặc tên bắt kỳ nào trong danh sách khai báo **enum** một trị int khác 0. Khi đó, các tên theo sau sẽ tương ứng với trị int này tăng dần lên 1.

Ví dụ 14.32 Cho khai báo

```
enum boolean {
    false = 0, true,
    no = 0, yes,
    sai = 0, dung
};
```

thì các trị true, yes, dung đương nhiên có trị là 1.

Tương tự như kiểu struct và union, biến kiểu enum có thể được khai báo ngay sau khi khai báo xong kiểu hoặc có thể sử dụng cú pháp sau:

```
enum tên_kiểu biến, biến [...];
```

Ví dụ 14.33

```
enum boolean tri_luan_ly;
enum weekday ngay;
```

thì *tri_luan_ly* và *ngay* là các biến thuộc kiểu enum, và chúng có thể được gán trị

```
triluan_ly = yes;
ngay = sunday;
```

Thật ra, C xem các tên liệt kê trong danh sách khai báo kiểu enum là các hằng int, nên một biến int có thể nhận trị là tên trong danh sách khai báo enum, ngược lại, một biến kiểu enum hoàn toàn có thể được gán trị int. Do vậy, có thể nói enum là một dạng đặc biệt của kiểu int, và việc sử dụng kiểu enum làm cho chương trình dễ nhìn và dễ hiểu vì các trị số đều được thay bằng các tên gợi nhớ hơn.

Ví dụ 14.34

Xét chương trình ví dụ sau

```
#include <stdio.h>
#include <conio.h>
enum weekday { sunday, monday, tuesday, wednesday, thursday, friday, saturday };
main()
{
    enum weekday en = 1;
    int i = 2;
```

```

    i += monday + 1;
    en += 2;
    clrscr();
    printf ("Tri cua bien int i la %d\n", i);
    printf ("Tri cua bien enum en la %d\n", en);
    getch();
}

```

Chương trình cho xuất liệu là:

```

Tri cua bien int i la 4
Tri cua bien enum en la 3

```

Chú ý rằng các tên được liệt kê ra trong danh sách trị của kiểu **enum** là các danh hiệu quy định cho các hằng số, ta không thể xuất, nhập trị trên chúng được.

Ví dụ 14.35

Không thể thực hiện các lệnh sau:

```

scanf ("%s", &en);
printf ("To day is %s", monday);

```

vì biến **en** được hiểu là biến kiểu int, còn **monday** là một hằng int.

14.4 ĐỊNH NGHĨA KIỂU BẰNG TYPEDEF

C cho phép lập trình viên khai báo kiểu mới theo cú pháp:

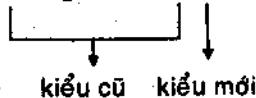
```
typedef kiểu_cũ kiểu_mới;
```

với:

- **typedef (type definition)** là một từ khóa của C, dùng để định nghĩa kiểu
- **Kiểu_cũ** là kiểu dữ liệu mà ta muốn đặt tên mới.
- **Kiểu_mới** là tên mới của kiểu cũ mà ta muốn đặt. Đây là một danh hiệu mà ta có thể dùng để khai báo biến.

Ví dụ 14.36

```
typedef unsigned char byte;
```



Khi đó byte được xem là một kiểu dữ liệu tương đương và có thể được sử dụng trong các khai báo biến như các kiểu dữ liệu khác.

Ví dụ 14.37

byte C;

Nếu nhìn kỹ, cú pháp của **typedef** giống như một khai báo biến có từ khóa **typedef** đặt trước và tên biến được thay bằng **kiểu_mới**.

Chương trình sau đây cho thấy sử dụng khai báo kiểu mới bằng **typedef** như trên:

Ví dụ 14.38

```
#include <stdio.h>
#include <conio.h>
typedef unsigned char byte;
main()
{
    byte c;
    clrscr();
    printf ("Moi nhap mot ky tu: ");
    c = getchar();
    if (c >= 'A' && c <= 'Z')
        printf ("%c la ky tu hoa \n",c);
    else
        printf ("%c khong phai la ky tu hoa \n", c);
    getch();
}
```

Chương trình cho xuất liệu, ví dụ:

Moi nhap mot ky tu: T

T la ky tu hoa

Thường **typedef** được sử dụng để định nghĩa các kiểu dữ liệu phức hợp (**pointer, struct, union, enum,...**) thành một tên duy nhất để dễ viết hơn khi khai báo.

Ví dụ 14.39

Ta có thể định nghĩa một kiểu mới

```
typedef int * INT_POINTER;
```

để sau đó có thể khai báo một cách ngắn gọn

```
INT_POINTER pi, a[10];
```

Đi nhiên khai báo trên hoàn toàn tương đương với khai báo

```
int *pi; *a[10];
```

Ví dụ 14.40

```
typedef struct sinh_vien
```

```
{
```

```
    char ma_so[10];
```

```
    char ho_ten[40];
```

```
    int tuoi;
```

```
    char dia_chi[80];
```

```
} SV, *SV_PTR;
```

thì sau này việc khai báo biến struct sinh_vien, hoặc pointer chỉ đến dữ liệu là một struct sinh_vien có thể được khai báo như sau:

```
SV sinh_vien1, sinh_vien2, lop[50];
```

```
SV_PTR psv;
```

Chú ý rằng, **typedef** có tác dụng đặt tên lại một kiểu dữ liệu phức hợp đã có, nó không đơn thuần là việc thay thế chuỗi như trong lệnh tiền xử lý **#define**.

BÀI TẬP CUỐI CHƯƠNG

- 14.1** Tạo một danh sách các sinh viên có các thông tin họ tên, mã số, điểm trung bình. Sắp xếp các học sinh này lại theo thứ tự điểm trung bình từ cao xuống thấp. In ra màn hình danh sách mới này theo hai dạng: không có xếp hạng và có xếp hạng.
- 14.2** Viết một chương trình với các hàm cho phép thực hiện các thao tác trên số phức: nhập, xuất, cộng, trừ, nhân, chia.
- 14.3** Viết chương trình với các hàm cho phép quản lý tọa độ một điểm trong mặt phẳng và trong không gian ba chiều.
- 14.4** Viết một chương trình để quản lý thông tin thời tiết của ngày, cụ thể là tạo mảng các cấu trúc có các thông tin sau
- ngày
 - tháng
 - năm
 - nhiệt độ cao nhất trong ngày
 - nhiệt độ thấp nhất trong ngày
 - tốc độ gió lớn nhất
- Tìm ngày có nhiệt độ cao nhất trong tháng và trong năm.
- 14.5** Dựa vào bài 4, hãy phát triển và viết chương trình dự báo thời tiết dựa vào hai yếu tố:
- Thời tiết hiện tại
 - Bảng thông tin về thời tiết của năm trước.

Chương 15

TẬP TIN

Trong ngôn ngữ C có nhiều hàm thao tác trên file. Những hàm này thật ra không phải là một phần của ngôn ngữ C, mà chúng nằm trong thư viện riêng biệt. Trong chương này ta sẽ khảo sát về file, các hàm mở file, đóng file, các hàm cho phép ghi đọc file một cách tuân tự hay ngẫu nhiên.

15.1 KHÁI NIỆM

Khi lập trình, các dữ liệu cần thiết cho chương trình bình thường có thể được nhập từ bàn phím và xuất kết quả ra màn hình. Tuy nhiên, trong thực tế có nhiều trường hợp dữ liệu nhập vào cho chương trình lại đang nằm trong một file nào đó do một tiến trình xử lý khác đưa đến, hoặc kết quả là các dữ liệu cần phải lưu để tham khảo, hay lại là đầu vào cho một tiến trình xử lý sau nữa. Như vậy, dữ liệu và kết quả trong những trường hợp như vậy không thể chỉ được in ra màn hình hay được nhập từ bàn phím mà còn phải được ghi vào đĩa từ dưới dạng tập tin (*file*) nào đó.

Để truy xuất một file, cần có bốn thao tác cơ bản sau:

- *Mở file:* thao tác này phải có trước khi cần làm việc trên file, nó cho phép truy xuất file với tên tương ứng và định vị trí dữ liệu trên file để thao tác.
- *Đọc file:* thao tác này dùng để lấy thông tin trong file, thông tin có thể được lấy theo từng chuỗi ký tự hoặc kiểu dữ liệu nào đó (integer, character, float, double, hay struct).
- *Ghi file:* thao tác này cho phép thêm thông tin vào file hoặc thay thế thông tin đã có trong file.

- **Dóng file:** thao tác này kết thúc việc truy xuất file, cho phép đóng file. Số lượng file được một chương trình đang chạy mở là có giới hạn, bằng việc đóng file hợp lý, chương trình đang chạy có thể mở file khác để truy xuất.

Cú pháp cụ thể của các thao tác này và các thao tác khác sẽ được đề cập trong mục sau.

15.2 CÁC HÀM TRUY XUẤT FILE

Các hàm truy xuất file đều có prototype trong file stdio.h.

1- Mở file - hàm fopen()

Để mở một file cho việc đọc ghi, thì đầu tiên file đó phải được mở bằng hàm fopen(). Prototype của hàm này nằm trong file stdio.h như sau:

```
FILE *fopen (const char *filename, const char *mode);
```

với:

- **FILE** là một kiểu struct để truy xuất file do C quy định.
- **filename** là một hằng hay một biến chuỗi, xác định tên file cần làm việc.
- **mode** là một chuỗi, xác định chế độ làm việc trên file. Các chế độ này có thể là:
 - . **r** : nếu mở file theo chế độ này thì file chỉ có thể được đọc. Việc đọc file bắt đầu từ đầu file trở đi. Nếu file chưa có trên đĩa, C sẽ báo lỗi.
 - . **w** : mở một file chỉ để ghi thông tin vào đó. Nếu file đó chưa có trên đĩa, C sẽ tạo file mới. Nếu file đó đã có, nội dung file đó bị xóa đi để chuẩn bị cho việc ghi thông tin mới.
 - . **a** : mở file để ghi thông tin tiếp nối vào đó.
 - . **r+** : mở file đang tồn tại theo cả hai chế độ ghi đọc mà nội dung file không bị mất đi. Chế độ này thường được dùng để mở và cập nhật file. Vị trí đầu tiên để truy xuất file là đầu file.
 - . **w+** : mở file theo cả hai chế độ ghi đọc, nội dung file cũ sẽ bị mất đi. Nếu file chưa có trên đĩa, file sẽ được tạo ra. Vị trí đầu tiên để truy xuất file là đầu file.

- . a+ : mở file đang tồn tại để cập nhật. File có thể được ghi và đọc. Nội dung file cũ không bị mất đi. Vị trí đầu tiên để truy xuất file là cuối file vừa được mở. Ngoài ra, chúng ta có thể quy định chế độ mở file để truy xuất theo dạng văn bản hay nhị phân bằng cách thêm các ký tự sau vào chuỗi mode trên:
- . t : mở file theo kiểu văn bản.
- . b : mở file theo kiểu nhị phân.

Ví dụ 15.1

```
FILE *fptr;
fptr = fopen ("myfile", "r+t");
```

C cho phép viết "r+t" hoặc "rt+" với cùng một mode.

Nếu không xác định dạng mở file cụ thể, C sẽ mở file theo dạng hiện thời, là dạng đang được lưu trong biến *fmode*.

Hàm này trả về một pointer đến cấu trúc FILE mà C đã định nghĩa sẵn trong file stdio.h. Cấu trúc này chứa các thông tin về việc mở file. Nếu sau khi gọi hàm, pointer trả về là khác pointer NULL, chứng tỏ file đã được mở thành công, pointer này đang lưu địa chỉ của một struct FILE về file đang làm việc. Nếu sau khi gọi hàm, pointer trả về pointer NULL, chứng tỏ không có một struct kiểu FILE nào được tạo, vì vậy file không được mở. Do đó, trong mọi thao tác file, ta luôn cần phải kiểm tra việc mở file bằng con trả chỉ đến kiểu FILE, nếu biến con trả này bằng NULL sau khi gọi fopen, thì cần báo lỗi, ngược lại sẽ cho xử lý.

Ví dụ 15.2

```
FILE *fptr;
fptr = fopen ("myfile", "rt+");
if (fptr == (FILE *)NULL)
{
    /* báo lỗi */
    ...
}
```

với myfile là tên file cần mở để đọc.

2- Đọc file

Cũng tương tự như các hàm thao tác trên thiết bị nhập chuẩn `getchar()`, `scanf()` và `gets()`, C cung cấp các hàm `fgetc()`, `fscanf()` và `fgets()` để nhập liệu từ file, prototype và chức năng của từng hàm này như sau:

- Hàm `fgetc()`

```
int fgetc (FILE *stream);
```

Hàm này trả về từng ký tự từ dòng xuất nhập *stream*. Nếu đã đến cuối file hoặc có lỗi thì hàm trả về trị EOF (end of file). Đây là một trị đã được define trong file stdio.h.

Hàm này cũng tương tự như `getc()`, tuy nhiên `getc()` là một macro, còn `fgetc()` là một hàm.

Ví dụ 15.3

```
c = fgetc (fptr);
```

- Hàm `fscanf()`

```
int fscanf (FILE *stream, const char *format [, address,...]);
```

Hàm này cũng tương tự như `scanf()` cho phép đọc từ tập tin *stream* các chuỗi ký tự và chuyển đổi thành các giá trị theo định dạng trong chuỗi *format*, gán vào cho các biến có địa chỉ *address* được nêu.

Nếu hàm `fscanf()` đọc được trị thì trả về từ hàm là khác 0, nếu không đọc được, hàm trả về trị số 0.

Ví dụ 15.4

```
fscanf (fptr, "%d", &i);
```

- Hàm `fgets()`

```
char *fgets(char *s, int n, FILE *stream);
```

Hàm này đọc một chuỗi ký tự từ tập tin *stream* cho đến khi gặp ký tự xuống hàng ('\n') hoặc khi đã đọc được *n*-1 ký tự. Hàm sẽ tự động thêm ký tự kết thúc chuỗi NUL vào cuối chuỗi mà không bỏ qua ký tự xuống hàng, lưu chuỗi đọc được vào biến *s*.

Hàm này trả về trị là một con trỏ NULL nếu không đọc được chuỗi, ngược lại, hàm trả về trị là con trỏ chỉ đến chuỗi s.

Ví dụ 15.5 Cho các khai báo

```
char s[81];
FILE *fptr;
thì
fgets (s, 81, fptr);
```

Chương trình sau đây đọc một file TAM. nào đó đã có trên đĩa, kết quả in các chuỗi có được ra màn hình.

Ví dụ 15.6

```
#include <string.h>
#include <stdio.h>
#include <conio.h>
main()
{
    FILE *stream;
    char msg[81];
    clrscr();
    /* mở file để đọc */
    stream = fopen ("D:\\VR\\TAM.", "r");
    /* đọc một chuỗi */
    fgets(msg, 81, stream);
    /* hiển thị chuỗi */
    printf("%s", msg);
    /* đọc một chuỗi từ file */
    fgets(msg, 81, stream);
    /* hiển thị chuỗi */
    printf("%s", msg);
    fclose(stream);
}
```

Chương trình này cho xuất liệu là:

```
#include <string.h>
#include <stdio.h>
```

Đây là hai dòng dữ liệu trong file văn bản TAM.

Chú ý rằng trong chuỗi quy định cho tên file để mở, nếu có thư mục thì dùng dấu hai dấu \\ để phân biệt thư mục, nếu dùng một dấu \, có thể lầm với các chuỗi thoát của ký tự.

3- Ghi file

C có các hàm fputc(), fprintf() và fputs() để ghi dữ liệu vào file đang được mở. Prototype và chức năng của từng hàm như sau:

- Hàm fputc()

```
int fputc(int c, FILE *stream);
```

Hàm này cho phép ghi một ký tự vào tập tin *stream*. Nếu việc ghi thành công hàm sẽ trả về chính ký tự đó, còn nếu đến cuối file hay có lỗi ghi, hàm sẽ trả về trị EOF. Hàm này khác với putc(), vì putc() là macro, trong khi fputc() là một hàm.

Ví dụ 15.7

```
cuu duong than cong . com
```

- Hàm fprintf()

```
int fprintf(FILE *stream, const char *format [, argument,...]);
```

Giống như hàm printf(), hàm fprintf() cho phép ghi chuỗi ký tự có định dạng vào tập tin *stream*. Hàm này trả về trị là số ký tự ghi được vào file, nếu ghi không được hàm sẽ trả về trị 0.

Ví dụ 15.8

```
fprintf(fptr, "Cac so la: %d %d \n", 45, 53);
```

- Hàm fputs()

```
int fputs(const char *s, FILE *stream);
```

Hàm này ghi vào tập tin *stream* chuỗi dữ liệu *s*. Nếu hàm ghi vào file thành công, hàm trả về trị là ký tự sau cùng được ghi vào file, nếu có lỗi trong quá trình ghi, hàm trả về trị EOF.

Ví dụ 15.9

```
fputs ("Tol di hoc", fptr);
```

Chương trình ví dụ sau đây cho phép ghi dữ liệu vào file với tên file được nhập từ bàn phím.

Ví dụ 15.10

```
#include <string.h>
#include <stdio.h>
#include <conio.h>
#include <process.h>
main()
{
    FILE *fptr;
    char filename[81], chuoi[81];
    int c;
    clrscr();
    /* mo file */
    printf ("Moi nhap ten file can mo: ");
    gets(filename);
    fptr = fopen(filename, "r");
    if ( fptr != NULL ) /* File dang ton tai */
    {
        printf ("File dang ton tai, muon ghi de khong? (y/n)");
        c = getchar();
        if ( c == 'y' || c == 'Y')
            /* mo lai file theo che do khac */
            freopen ( filename, "w+", fptr);
        else
            exit(1);
    }
    else /* File chua ton tai */
    {
        fclose (fptr);
        fptr = fopen ( filename, "w+");
    }
    fprintf (fptr, "Hang 1 la: %d %d\n", 45, 56);
    fprintf (fptr, "Hang 2 la: %d %d\n", 49, 59);
```

```

    rewind (fptr); /* bat dau thao tac tu dau file */
    fgets (chuoi, 81, fptr);
    printf ("Chuoi 1 doc duoc trong file la: %s", chuoi);
    fgets (chuoi, 81, fptr);
    printf ("Chuoi 2 doc duoc trong file la: %s", chuoi);
    fclose(fptr);
}

```

Chương trình tạo ra một file với tên file nhập từ bàn phím, ví dụ tên file nhập là vi_du, chương trình cho kết quả là

```

Moi nhap ten file can mo: vi_du
File dang ton tai, muon ghi de khong? (y/n)y
Chuoi 1 doc duoc trong file la: Hang 1 la: 45 56
Chuoi 2 doc duoc trong file la: Hang 2 la: 49 59

```

Với file vi_du có nội dung là

```

Hang 1 la: 45 56
Hang 2 la: 49 59

```

Trong chương trình, hàm freopen () dùng để đóng một file đang được mở, và gán lại dòng xuất nhập để mở file lại theo chế độ khác. Hàm này chỉ dùng cho file đang tồn tại trên đĩa.

Hàm rewind () dùng để đưa thao tác file về đầu file.

4. Ghi đọc file theo struct

Cung cấp các hàm fread() và fwrite () dùng để đọc và ghi file theo struct. Các hàm này có prototype trong file stdio.h như sau:

```

size_t fread(void *ptr, size_t size, size_t n, FILE *file_ptr);
size_t fwrite(const void *ptr, size_t size, size_t n, FILE *file_ptr);

```

với:

- *size_t* là kiểu unsigned mà C đã định nghĩa trong file stdio.h.
- *ptr* là con trỏ hay địa chỉ của biến hay mảng các struct.
- *size* là số byte một lần đọc/ghi
- *n* là số struct một lần đọc/ghi file
- *file_ptr* là file để ghi/đọc dữ liệu.

Các hàm này trả về trị là một số int cho biết số lượng struct ghi/đọc vào/từ file. Nếu không ghi/đọc được vì gặp cuối file hay lỗi truy xuất (ví dụ như dĩa đầy) thì trị trả về từ hàm là 0.

Chương trình ví dụ sau đây tạo một file là danh sách các sinh viên, mỗi sinh viên có các thông tin: mã số, họ tên, điểm trung bình. Tìm sinh viên có mã số nhập vào từ bàn phím. File tạo ra có dạng binary.

Ví dụ 15.11

```
#include <string.h>
#include <stdio.h>
#include <conio.h>
#include <process.h>
typedef struct
{
    char ma_so[9];
    char ho_ten[40];
    double diem_trung_binh;
} SV;
main ()
{
    FILE *fptr;
    char filename[81], ma_so_nhap[9];
    SV a;
    int c, i, n;
    clrscr();
    /* open a file for update */
    printf ("Moi nhap ten file can mo: ");
    gets (filename);
    fptr = fopen(filename, "r");
    if ( fptr != NULL ) /* File dang ton tai */
    {
        printf ("File dang ton tai, muon ghi de khong? (y/n)");
        c = getchar();
        if ( c == 'y' || c == 'Y')
            freopen (filename, "w+b", fptr);
        else
    }
```

```

        exit(1);
    }
else /* File chua ton tai */
{
    fclose (fptr);
    fptr = fopen ( filename, "w+b");
}
printf ("Ban nhap bao nhieu sinh vien: ");
scanf ("%d", &n);
flushall();
/* nhap lieu vao struct, va ghi vao file */
for (i = 0; i < n; i++)
{
    printf ("Sinh vien thu %d \n", i+1);
    printf ("Ma so: ");
    gets (a.ma_so);
    printf ("Ho ten: ");
    gets (a.ho_ten);
    printf ("Diem trung binh: ");
    scanf ("%lf", &a.diem_trung_binh);
    fwrite (&a, sizeof(SV), 1, fptr);
    flushall();
}
/* dua thao tac file ve dau file */
rewind (fptr);
printf ("Moi nhap ma so sinh vien can tim: ");
gets (ma_so_nhap);
while ( fread (&a, sizeof(SV), 1, fptr) )
{
    if ( strcmp (a.ma_so, ma_so_nhap) ==0 )
    {
        printf ("Hoc sinh can tim co cac thong tin sau: \n");
        printf ("Ma so: %s \n", a.ma_so);
        printf ("Ho ten: %s \n", a.ho_ten);
        printf ("Diem trung binh: %.2f \n", a.diem_trung_binh);
    }
    printf ("Da het file \n");
    fclose(fptr);
}

```

Chương trình cho kết quả là:

Moi nhap ten file can mo: vi_du.dbf

File dang ton tai, muon ghi de khong? (y/n)y

Ban nhap bao nhieu sinh vien: 4

Sinh vien thu 1

Ma so: 4956781

Ho ten: Tran Than Tam

Diem trung binh: 7.89

Sinh vien thu 2

Ma so: 4943421

Ho ten: Nguyen Thanh Binh

Diem trung binh: 5.45

Sinh vien thu 3

Ma so: 4950987

Ho ten: Dang Trung Hieu

Diem trung binh: 8.21

Sinh vien thu 4

Ma so: 4931256

Ho ten: Quach Thanh Tung

Diem trung binh: 7.80

Moi nhap ma so sinh vien can tim: 4943421

Hoc sinh can tim co cac thong tin sau:

Ma so: 4943421

Ho ten: Nguyen Thanh Binh

Diem trung binh: 5.45

Da het file

5- **Dóng file**

Để đóng file, C cung cấp các hàm sau đây trong thư viện:

- **Hàm fclose()** dùng để đóng một file đang được mở, prototype của hàm này như sau:

```
int fclose(FILE *stream);
```

Nếu đóng lại thành công, hàm trả về trị 0, nếu xảy ra lỗi, hàm trả về trị EOF.

- **Hàm fcloseall()** dùng để đóng tất cả các dòng xuất nhập file đang được mở. Prototype của hàm này như sau:

```
int fcloseall(void);
```

6- Các thao tác khác

Ngoài các thao tác trên ra, trong thư viện C còn nhiều hàm khác truy xuất file nữa, ví dụ hàm:

- **Hàm fseek ()** cho phép chuyển vị trí đang truy xuất trong file đến vị trí mong muốn. Prototype của hàm này như sau:

```
int fseek(FILE *stream, long offset, int where);
```

với:

- + *offset* là độ dời tính bằng byte, kể từ đầu file hoặc cuối file (tùy theo thông số *where*) đến vị trí mong muốn.
- + *where* là thông số xác định gốc tính độ dời, nếu bằng 0, vị trí cần đến truy xuất trong file lệch đầu file là offset byte; nếu bằng 1, gốc tính là vị trí hiện hành trong file đang truy xuất tính xuống, nếu bằng 2, gốc tính là cuối file tính lên.

Hàm này trả về trị 0 nếu kết quả thành công, còn ngược lại, hàm trả về trị khác 0.

Ví dụ 15.12

```
num_struct =10 * sizeof (struct sinh_vien);
if (fseek (fptr, num_struct, 0) ==0)
{
    /* dời vị trí cần truy xuất đến struct thứ 11 trong file */
    if ( fread( &var_struct, sizeof(struct sinh_vien), 1,fptr ) != 0)
        /* đọc thành công - truy xuất struct */
}
```

Hàm này thường được sử dụng khi mở file theo dạng nhị phân.

- **Hàm ftell ()** dùng để cho biết vị trí đang truy xuất file hiện thời trong dòng xuất nhập file. Hàm này nên dùng khi mở file theo dạng nhị phân. Prototype của hàm này như sau:

```
long ftell(FILE *stream);
```

Nếu file được mở theo dạng nhị phân thì hàm này trả về trị tính theo byte kể từ đầu file. Trị trả về này có thể được sử dụng trong hàm fseek để truy xuất file.

Ngoài ra, còn nhiều hàm khác có thể truy xuất file. Trong chương 18 của sách này, có một số hàm được nêu ra. Độc giả có thể tham khảo thêm các sách tra cứu thư viện của C.

Chú ý: C đã định nghĩa sẵn các dòng xuất nhập chuẩn, các dòng này sẽ được C tự động mở khi chạy chương trình. Các dòng xuất nhập đó là

- `stdin`: dòng xuất nhập cho thiết bị nhập chuẩn, thường là bàn phím.
- `stdout`: dòng xuất nhập cho thiết bị xuất chuẩn, thường là màn hình.
- `stderr`: dòng xuất nhập cho thiết bị xuất báo lỗi chuẩn, thường là màn hình.
- `stdaux`: dòng xuất nhập cho thiết bị xuất nhập phụ.
- `stdprn`: dòng xuất nhập cho máy in, thường máy in khi đó được đặt ở port LPT1.

Do đó, mọi thao tác file trên các dòng xuất nhập chuẩn đều có nghĩa là thao tác trên thiết bị đó.

Ví dụ 15.13

In một chuỗi ra màn hình, ta có thể viết như sau:

```
printf (stdout, "I go to school\n");
```

Nhập trị từ bàn phím

```
fscanf (stdin, "%d", &a);
```

BÀI TẬP CUỐI CHƯƠNG

15.1 Viết chương trình nhập danh sách các học sinh vào một file, mỗi học sinh có các thông tin sau:

- Mã số
- Họ tên
- Ngày tháng năm sinh
- Điểm trung bình

Tìm học sinh có điểm trung bình lớn nhất. Nếu có trùng điểm trung bình thì chỉ in học sinh trẻ nhất.

15.2 Viết chương trình tạo file chứa dãy số các số nguyên là số nguyên tố từ 1 đến 1000 theo hai dạng nhị phân và văn bản. Nhận xét về hai file này. Viết chương trình khác để đọc file trả lại, in dãy này ra màn hình.

15.3 Giả sử đã có file dữ liệu chứa danh sách học sinh như bài 1. Hãy viết chương trình sắp xếp file lại theo điểm trung bình từ cao xuống thấp. In file này ra màn hình.

Lưu ý: bài này sắp xếp trực tiếp trên file gốc.

15.4 Tương tự như bài 3, nhưng không sắp xếp trên file gốc mà tạo ra một file chỉ mục, mọi việc truy xuất trên file dữ liệu đều phải qua file chỉ mục quy định phần tử cần truy xuất.

15.5 Mô phỏng lệnh type của DOS

15.6 Mô phỏng lệnh copy con của DOS

15.7 Mô phỏng lệnh ren của DOS

15.8 Viết chương trình đọc một file theo dạng văn bản. Tạo file mới từ file cũ với yêu cầu chuyển những ký tự thường sang ký tự hoa.

15.9 Viết chương trình đọc file ảnh .BMP. Với mỗi entry trong phần data, tăng/giảm đi 50 trị. Ghi dữ liệu mới và tạo ra file .BMP mới này. Hiển thị ảnh thị ảnh ra màn hình bằng một phần mềm nào đó. Nhận xét sự khác nhau giữa 2 ảnh ban đầu và ảnh mới.

Chương 16

CHẾ ĐỘ ĐỒ HỌA

16.1 GIỚI THIỆU

C cho phép lập trình viên thiết kế chương trình trong chế độ đồ họa để vẽ các hình hay đồ thị theo ý muốn mặc dù hiện nay lập trình trên giao diện đồ họa của MS Window là rất phổ biến. Các bước cần có thể để chuyển màn hình sang chế độ đồ họa:

- *Khởi động chế độ đồ họa:* để chuyển màn hình đang hoạt động trong chế độ văn bản sang chế độ đồ họa, ta cần có file .BGI tương ứng với màn hình đang có. Ví dụ, với màn hình EGA hoặc VGA, tập tin cần thiết chứa các chương trình điều khiển đồ họa là tập tin EGAVGA.BGI.
- *Thao tác trong chế độ đồ họa:* các hàm thư viện mà C đã thiết kế sẵn cho các lập trình viên sử dụng để vẽ hình theo ý muốn.
- *Đóng chế độ đồ họa:* sau khi thực hiện các thao tác vẽ hình cần thiết xong, ta cần phải đóng chế độ đồ họa lại, đưa màn hình về chế độ văn bản chuẩn.

Mục sau đây trình bày các hàm, biến cần thiết mà C đã thiết kế sẵn cho lập trình viên sử dụng.

16.2 CÁC HÀM SỬ DỤNG TRONG CHẾ ĐỘ ĐỒ HỌA

Các hàm được sử dụng trong chế độ đồ họa đều có prototype nằm trong file graphics.h.

1- Hàm khởi động chế độ đồ họa initgraph()

Hàm này dùng để khởi động chế độ đồ họa. Prototype của hàm này nằm trong file graphics.h như sau:

```
void Initgraph (int *graph_driver,int *graph_mode,
char*pathtodriver);
```

với: - *graph_driver* là một con trỏ chỉ đến một biến nguyên, giá trị trong số này là số hiệu đặc trưng cho bộ điều khiển đồ họa phần cứng. **graph_driver* có thể nhận được một trong các trị sau đây, các trị này đã được định nghĩa trong file graphics.h:

Bảng 16.1 Số hiệu của các bộ điều khiển đồ họa được sử dụng trong TC 2.0 và BC 5.OX

Hàng	Giá trị
DETECT	0 (truy tìm tự động)
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7
ATT400	8
VGA	9
PC3270	10

Nếu **graph_driver* được gán trị là DETECT khi gọi hàm initgraph(), thì hàm này sẽ tự động truy tìm bộ điều khiển đồ họa và khởi động chế độ đồ họa ở chế độ mặc nhiên, thường là chế độ có độ phân giải cao nhất mà bộ điều khiển đó cho phép.

- *graph_mode* là một con trỏ chỉ đến một biến nguyên, giá trị biến này là số hiệu của kiểu (*mode*) màn hình đồ họa đang làm việc. **graph_mode* có thể nhận được một trong các trị sau đây, các trị này đã được định nghĩa trong file graphics.h:

Bảng 16.2 Các chế độ đồ họa

Bộ điều khiển	Chế độ đồ họa	Giá trị	Cột hàng	Càng màu	Số trang
CGA	CGAC0	0	320x 200	C0	1
	CGAC1	1	320x 200	C1	1
	CGAC2	2	320x 200	C2	1
	CGAC3	3	320x200	C3	1
	CGAHI	4	640x200	2 màu	1
MCGA	MCGAC0	0	320x 200	C0	1
	MCGAC1	1	320x 200	C1	1
	MCGAC2	2	320x 200	C2	1
	MCGAC3	3	320x 200	C3	1
	MCGAMED	4	640x 200	2 màu	1
EGA	MCGAHI	5	640x 480	2 màu	1
	EGAL0	0	640x 200	16 màu	4
EGA64	EGAH1	1	640x 350	16 màu	2
	EGA64L0	0	640x200	16 màu	1
EGA-MONO	EGA64HI	1	640x 350	4 màu	1
	EGAMONOH1	3	640x 350	2 màu	1 (1)
HERC	EGAMONOH1	3	640x 350	2 màu	2 (2)
	HERCMONOH1	0	720x 348	2 màu	2
ATT400	ATT400C0	0	320x 200	C0	1
	ATT400C1	1	320x 200	C1	1
	ATT400C2	2	320x 200	C2	1
	ATT400C3	3	320x 200	C3	1
	ATT400MED	4	640x 200	2 màu	1
	ATT400HI	5	640x 400	2 màu	1
VGA	VGAL0	0	640x 200	16 màu	2
	VGAMED	1	640x 350	16 màu	2
	VGAHI	2	640x 480	16 màu	1
PC3270	PC3270HI	0	720x 350	2 màu	1
IBM8514	IBM8514HI	1	1024x 768	256 màu	
	IBM8514L0	0	640 x480	256 màu	

(1) 64K trên EGAMONO card; (2) 256K trên EGAMONO card

Nếu **graph_driver* được gán trị là DETECT khi gọi hàm *initgraph()* thì sau khi gọi *initgraph()*, **graph_mode* được gán trị tương ứng với chế độ đồ họa mà *initgraph()* tìm được.

- *pathtodriver*: tên đầy đủ của thư mục trong đó có các file .BGI (và các file .CHR).

Nếu *initgraph()* không tìm thấy các file .BGI trong thư mục trên thì nó sẽ tìm các file .BGI trong thư mục hiện hành.

Nếu ngay từ đầu, *pathtodriver* là chuỗi rỗng thì việc truy tìm file .BGI sẽ được thực hiện trong thư mục hiện hành.

Ta có thể gán trị cho **graph_driver* và **graph_mode* các giá trị tương ứng với card điều khiển và chế độ muốn mở và sau đó gọi *initgraph()*; nếu để *initgraph()* truy tìm tự động card điều khiển và sau đó khởi động chế độ đồ họa mặc nhiên thì cần gán trị DETECT vào cho **graph_driver* trước khi gọi *initgraph()*.

Nếu *initgraph()* gặp một lỗi khi mở chế độ đồ họa thì mã lỗi được gán vào biến nguyên **graph_driver*, các trị này đã được định nghĩa như sau:

Mã lỗi	Trị bằng tương ứng với mã lỗi	Lỗi
-2	grNotDetected	Không tìm thấy phần cứng điều khiển đồ họa
-3	grFileNotFoundException	File điều khiển đồ họa (.BGI) không thấy
-4	grInvalidDriver	File điều khiển đồ họa (.BGI) không hợp lệ
-5	grNoLoadMem	Không đủ bộ nhớ để nạp file .BGI

Tuy nhiên, trong thực tế để kiểm tra việc mở chế độ đồ họa, C cung cấp hàm *graphresult()*, prototype của hàm này như sau:

```
int graphresult(void)
```

Hàm này trả về một trị nguyên đặc trưng cho lỗi (*graphics_error*) đã gặp, các lỗi này đã được C định nghĩa như sau:

Bảng 16.3 Các lỗi trong chế độ đồ họa được trả về từ hàm graphresult()

Mã lỗi	Trị bảng tương ứng với mã lỗi	Lỗi
0	grOk	Không có lỗi
-1	grNoInitGraph	File .BGI chưa được cài đặt
-2	grNotDetected	Không tìm thấy phần cứng điều khiển đồ họa
-3	grFileNotFoundException	File điều khiển đồ họa (.BGI) không thấy
-4	grInvalidDriver	File điều khiển đồ họa (.BGI) không hợp lệ
-5	grNoLoadMem	Không đủ bộ nhớ để nạp file .BGI
-6	grNoScanMem	Không đủ bộ nhớ để vẽ hình (scan fill)
-7	grNoFloodMem	Không đủ bộ nhớ để tô hình (flood fill)
-8	grFontNotFound	File phông chữ không thấy
-9	grNoFontMem	Không đủ bộ nhớ để nạp phông chữ
-10	grInvalidMode	Chế độ bộ đồ họa không phù hợp bộ điều khiển
-11	grError	Lỗi đồ họa
-12	grIOerror	Lỗi đồ họa do việc truy xuất thiết bị
-13	grInvalidFont	File phông chữ không hợp lệ
-14	grInvalidFontNum	Số hiệu phông không hợp lệ
-15	grInvalidDeviceNum	Số hiệu thiết bị không hợp lệ
-16	grInvalidVersion	Ấn bản không hợp lệ

Để in ra câu thông báo lỗi tương ứng lỗi *graphics_error* đã gặp, C cung cấp hàm *grapherrmsg()*, prototype của hàm này như sau:

```
char * grapherrmsg(int errorcode);
```

Hàm này trả về câu thông báo lỗi tương ứng lỗi *errorcode* đã gặp.

Ví dụ 16.1

Cho các khai báo sau:

```
int gd = DETECT, gm;
int error_code;
initgraph (&gd, &gm, "D:\\TC20");
error_code = graphresult();
if ( error_code != grOk )
```

```

{
    printf ("Graphics error: %s\n", grapherrmsg(error_code));
    exit (1);
}
/* thực hiện các thao tác về hình tại đây */
...

```

Nếu đã biết rõ card điều khiển đồ họa và chế độ cần mở, ta có thể gán trị trực tiếp như sau:

```

int gd = VGA, gm = VGAHI;
int error_code;
initgraph (&gd, &gm, "D:\\TC20");
error_code = graphresult();
...

```

2- *Hàm xác định card điều khiển đồ họa detectgraph()*

Hàm này có chức năng dùng để xác định bộ phối ghép điều khiển đồ họa và chế độ hoạt động của màn hình tương ứng với độ phân giải cao nhất mà bộ phối ghép cho phép. Prototype của hàm này như sau:

```

void detectgraph(int *graph_driver, int *graph_mode);

```

với các biến **graph_driver*, **graph_mode* cũng tương tự như trong *initgraph()*.

Sau khi gọi hàm *detectgraph()*, nếu hệ thống phần cứng không cho phép màn hình hoạt động ở chế độ graph thì biến **graph_driver* có trị là -2. Chương trình ví dụ sau đây để xác định bộ điều khiển đồ họa phần cứng và báo ra màn hình kết quả tìm được.

Ví dụ 16.2

```

#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
/* Tên của các loại card khác nhau */
char *dname[] = { "Chế độ tìm tự động",
                  "Bo phoi ghep CGA",
                  "Bo phoi ghep MCGA",

```

```

    "Bo phoi ghep EGA",
    "Bo phoi ghep EGA 64K tren card",
    "Bo phoi ghep monochrome EGA",
    "Bo phoi ghep IBM 8514",
    "Bo phoi ghep Hercules monochrome",
    "Bo phoi ghep AT&T",
    "Bo phoi ghep VGA",
    "Bo phoi ghep IBM PC 3270"
};

main()
{

```

```

    int gdriver, gmode, errorcode;
    /* tim phan cứng ứng hộ cho chế độ đồ họa */
    detectgraph(&gdriver, &gmode);
    if (gdriver < 0) /* an error occurred */
    {
        printf("Phần cứng không ứng hộ cho chế độ đồ họa");
        printf("An phím bất kỳ để thoát");
        getch();
        exit(1);
    }
    /* hiển thị thông tin tìm được */
    clrscr();
    printf("Bạn đang có card màn hình là %s\n", dname[gdriver]);
    printf("An phím bất kỳ để thoát");
    getch();
}

```

Chương trình cho xuất liệu, ví dụ:

Bạn đang có card màn hình là bo phoi ghep VGA
An phím bất kỳ để thoát

Hàm này thường được gọi sử dụng trước khi gọi initgraph().
Bằng việc gọi hàm detectgraph(), lập trình viên có thể thay đổi chế độ đồ họa của màn hình, không theo thông số mặc nhiên detectgraph() trả về, mà khởi động chế độ đồ họa theo chế độ lập trình viên muốn.

Ví dụ 16.3

```

int gd, gm;
detectgraph (&gd, &gm);
if (gd == VGA && gm == VGAHI)
    gm = VGAMED;
initgraph (&gd, &gm, "");

```

Nếu khi gọi initgraph (), với đối số **graph_driver* là DETECT, thì hàm initgraph() tự động gọi hàm detectgraph() để truy tìm các thông số phần cứng và sau đó khởi động chế độ đồ họa.

3- Hàm đóng chế độ đồ họa closegraph()

Hàm này dùng để đóng chế độ đồ họa, chuyển màn hình về chế độ văn bản. Prototype của hàm như sau:

```
void closegraph(void);
```

Ví dụ 16.4

```

initgraph (&gd, &gm, );
...
closegraph();

```

Hàm closegraph() thoát ra khỏi hẳn chế độ đồ họa về chế độ văn bản. C cũng cung cấp thêm hàm restorecrtmode() để lưu màn hình graph hiện thời và tạm thời về chế độ văn bản, sau đó để quay lại chế độ đồ họa, cần gọi hàm setgraphmode() với thông số chế độ mà getgraphmode () đã lưu. Prototype của các hàm restorecrtmode (), setgraphmode () và getgraphmode() như sau:

```

void restorecrtmode(void);
int getgraphmode(void);
void setgraphmode(int mode);

```

Chương trình ví dụ sau đây cho cách sử dụng các hàm trên:

Ví dụ 16.5

```

#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>

```

```

#include<conio.h>
main()
{
    /* Truy tìm tự động */
    int gdriver = DETECT, gmode, errorcode;
    int x, y;
    clrscr();
    gotoxy(20, 12);
    printf ("Dang trong che do van ban. An mot phim \n");
    getch();
    /* Khởi động chế độ đồ họa */
    initgraph(&gdriver, &gmode, );
    /* đọc kết quả của việc khởi động */
    errorcode = graphresult();
    if (errorcode != grOk) /* có xảy ra lỗi */
    {
        printf("Loi: %s \n", grapherrmsg(errorcode));
        printf("An mot phim de ket thuc:");
        getch();
        exit(1); /* Kết thúc chương trình, trả về mã lỗi */
    }
    x = getmaxx() / 2;
    y = getmaxy() / 2;
    /* vẽ ra màn hình một hình khối 3 chiều */
    bar3d (x - 20, y - 20, x + 20, y + 20, 10, 1);
    getch();
    /* về chế độ văn bản */
    restorecrtmode();
    printf("Chung ta dang trong che do van ban.\n");
    printf("An mot phim bat ky ve che do do hoa:");
    getch();
    /* về lại chế độ đồ họa */
    setgraphmode( getgraphmode() );
    /* in câu thông báo ra màn hình */
    settextjustify(CENTER_TEXT, CENTER_TEXT);
    outtextxy(x; y, "Chung ta dang trong che do do hoa.");
    outtextxy(x, y+20, "An mot phim de ket thuc:");
    getch();
    closegraph();
}

```

Lưu ý rằng, khi sử dụng hàm restorecrtmode() để quay về chế độ văn bản thì tất cả chưa dựng trước đó (nếu có) của màn hình đều bị mất. Tương tự, nếu quay trở lại chế độ đồ họa từ setgraphmode() thì mọi hình vẽ lúc đó trên màn hình đều bị mất.

4. Hàm đặt màu vẽ setcolor()

Hàm này dùng để đặt màu vẽ cho các lệnh vẽ hình tròn, hình chữ nhật, ký tự... Prototype của hàm này như sau:

```
void setcolor(int color);
```

với color là một trị nguyên quy định màu vẽ cho các lệnh vẽ theo nét: rectangle(), line(), arc(), ellipse()... color có thể nhận được một trong các trị sau:

Bảng 16.4 Bảng các màu được sử dụng với setcolor().

Hàng	Trị
BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
LIGHTGRAY	7
DARKGRAY	8
LIGHTBLUE	9
LIGHTGREEN	10
LIGHTCYAN	11
LIGHTRED	12
LIGHTMAGENTA	13
YELLOW	14
WHITE	15

Cần biết màu vẽ hiện thời, C cung cấp hàm getcolor(), hàm này có prototype như sau:

```
int getcolor(void);
```

Hàm này trả về trị là màu đang được sử dụng để vẽ.

5- Hàm đặt màu nền setbkcolor()

Hàm này dùng để đặt màu nền hiện thời là màu trong đối số của hàm. Prototype của hàm này như sau:

```
void setbkcolor(int color);
```

với color là một trong 16 màu ở bảng 16.4.

Hàm trả về màu nền hiện thời là getbkcolor(), prototype của hàm này như sau:

```
int getbkcolor(void)
```

6- Hàm xuất ký tự ra màn hình trong chế độ đồ họa outtextxy()

Hàm này dùng để in một chuỗi ký tự ra màn hình, có thể chọn phông, kích thước, hướng ký tự để in ký tự. Prototype của hàm này như sau:

```
void outtextxy(int x, int y, char *textstring);
```

với

- *x, y* là vị trí để in chuỗi.

- *textstring* là con trỏ chỉ đến chuỗi cần in.

Các hàm sau đây thường dùng với outtextxy() để xác định thêm về chuỗi ký tự cần xuất:

- **Hàm settextjustify()** dùng để điều chỉnh chuỗi ký tự cần in lệch như thế nào so với tọa độ (*x, y*) mà ta đặt trong outtextxy(). Prototype của hàm này như sau

```
void settextjustify(int horiz, int vert);
```

với *horiz, vert* là các trị nguyên, xác định việc canh chỉnh theo chiều ngang và chiều đứng của chuỗi văn bản cần xuất. Chúng có thể nhận được các trị sau đây:

Bảng 16.5 Trị hằng quy định cho hàm `settextjustify()`

Đối số	Hằng	Trị	Ý nghĩa
horiz	LEFT_TEXT	0	Chuỗi được canh từ trái
	CENTER_TEXT	1	Chuỗi được canh giữa
	RIGHT_TEXT	2	Chuỗi được canh từ phải
vert	BOTTOM_TEXT	0	Chuỗi được canh từ dưới
	CENTER_TEXT	1	Chuỗi được canh giữa
	TOP_TEXT	2	Chuỗi được canh từ trên

Chú ý việc canh chuỗi theo chiều ngang (*horiz*), hay chiều dọc (*vert*) là tính tương đối so với vị trí (x, y) mà ta đặt trong `outtextxy()`. Ví dụ khi in chuỗi ra đât.

```
settextjustify(RIGHT_TEXT, TOP_TEXT);
```

thì chuỗi được in ra sao cho (x, y) nằm bên phải (RIGHT_TEXT) và phía trên chuỗi (TOP_TEXT).

- **Hàm `settextstyle()`** dùng để chọn phông, kích thước và hướng xuất chuỗi. Prototype của hàm này như sau:

```
void settextstyle(int font, int direction, int charsize);
```

với: - *font* là một số nguyên tương ứng loại phông ký tự để xuất chuỗi. Nó có thể nhận được một trong các trị sau đây:

Bảng 16.6 Các loại phông ký tự và loại hằng tương ứng

Hằng	Trị	Loại phông	Tên file
DEFAULT_FONT	0	Phông chuẩn 8x8	-
TRIPLEX_FONT	1	Phông triplex	TRIP.CHR
SMALL_FONT	2	Phông ký tự nhỏ	LITT.CHR
SANS_SERIF_FONT	3	Phông sans-serif	SANG.CHR
GOTHIC_FONT	4	Phông gothic	GOTH.CHR

- *direction* là một số nguyên quy định việc in chuỗi theo chiều dọc hay chiều ngang. Nó có thể nhận được một trong các trị sau:

Bảng 16.7 Các hướng xuất ký tự

Hàng	Trị	Hướng xuất chuỗi
HORIZ_DIR	0	In chuỗi theo chiều ngang
VERT_DIR	1	In chuỗi theo chiều dọc

- *charsize* là một trị nguyên định kích thước in chuỗi. Kích thước này được tính từ kích thước chuẩn của ma trận ký tự 8×8 nhân với số nguyên *charsize*. Như vậy nếu *charsize* bằng 1 thì ký tự được in ra sẽ là ma trận 32×32 , tức gấp bốn kích thước thường. Nếu *charsize* bằng 0, kích thước ký tự in ra được xác định từ hàm *setusercharsize()*, hàm này có prototype như sau:

```
void setusercharsize(int multx, int divx, int multy, int divy);
```

- với
- *multx/divx* xác định độ rộng ký tự.
 - *multy/divy* xác định độ cao ký tự.

Ví dụ 16.6

```
settextstyle(SANS_SERIF_FONT, HORIZ_DIR, 0);
setusercharsize(10, 3, 7, 3);
outtextxy (20, 20, "I go to school");
```

7- *Hàm vẽ hình chữ nhật rectangle()*

Hàm này dùng để vẽ ra màn hình hình chữ nhật. Prototype của hàm này như sau:

```
void rectangle (int left, int top, int right, int bottom);
```

- với:
- (*left, top*) là tọa độ góc trên bên trái của hình chữ nhật.
 - (*right, bottom*) là tọa độ góc dưới bên phải của hình chữ nhật.

Ví dụ 16.7

```
rectangle (10, 10, 20, 50);
```

Khi vẽ hình chữ nhật, màu cạnh của hình chữ nhật được quy định từ *setcolor()*, độ dày của cạnh được quy định từ hàm *setlinestyle()*, hàm này có prototype như sau:

```
void setlinestyle(int linestyle, unsigned upattern, int thickness);
```

với: - *linestyle* là một trị nguyên quy định kiểu mẫu đường vẽ. Nó có thể nhận được một trong các trị sau đây:

Bảng 16.8 Các mẫu đường vẽ

Hàng	Trị	Ý nghĩa
SOLID_LINE	0	Vẽ đường liền nét
DOTTED_LINE	1	Vẽ đường chấm
CENTER_LINE	2	Vẽ đường gạch
DASHED_LINE	3	Vẽ đường gạch chéo
USERBIT_LINE	4	Vẽ đường theo mẫu quy định trong upattern

- *upattern* là 16 bit mẫu vẽ cho lập trình viên tự quy định. Nếu đổi số *linestyle* là USERBIT_LINE thì mẫu vẽ đường lấy từ *upattern*. Một số ví dụ về *upattern*:

16-bit mẫu	Trị của upattern
..xx..xx..xx..xx	Ox3333 (gạch ngắn)
....xxxx....xxxx	OxOFOF (gạch dài)
..xxxxxxxx.....xxxxxx	Ox3F3F (gạch dài hơn)
xxxxxxxxxxxxxxxxxx	OxFFFF (gạch liền nét)

- *thickness* là số nguyên quy định cho độ dày của đường vẽ. Nó có thể nhận được một trong hai trị sau:

Bảng 16.9 Độ dày các đường vẽ

Hàng	Trị	Độ dày
NORM_WIDTH	1	1 pixel
THICK_WIDTH	3	3 pixels

Ví dụ 16.8

```
setlinestyle(CENTER_LINE, 1, NORM_WIDTH);
rectangle (10, 10, 20, 50);
```

8. Hàm vẽ đường tròn *circle()*, cung tròn *arc()* và hình cánh quạt *pieslice()*

Prototype của các hàm này như sau:

```
void arc(int x, int y, int stangle, int endangle, int radius);
```

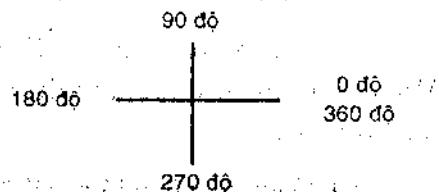
```
void circle(int x, int y, int radius);
void pieslice(int x, int y, int stangle, int endangle, int radius);
```

với: - **(x, y)** là tọa độ tâm

- **stangle** là góc bắt đầu vẽ

- **entangle** là góc kết thúc vẽ

- **radius** là bán kính hình.



Chú ý: *stangle* và *entangle* được tính theo độ, có nghĩa là chúng nhận trị theo tọa độ hình bên.

Màu của các đường vẽ được đặt từ `setcolor()`. Riêng hàm vẽ hình cánh quạt `pieslice()` có mẫu vẽ và màu tô được cho trong hàm `setfillstyle()`, hàm này dùng để đặt mẫu vẽ và màu tô cho các hàm vẽ hình đặc: `bar()`, `bar3d()`, `pieslice()`, ... Hàm này có prototype như sau:

```
void setfillstyle(int pattern, int color);
```

với: **pattern** là mẫu vẽ, Nó có thể nhận được một trong các trị sau đây:

Bảng 16.10 Các mẫu tô đặc

Hàng	Trị	Ý nghĩa tô đầy
EMPTY_FILL	0	Tô đầy với màu nền
SOLID_FILL	1	Tô đầy với màu được đặt
LINE_FILL	2	Tô đầy với -
LTSLASH_FILL	3	Tô đầy với //
SLASH_FILL	4	Tô đầy với //, nét dày
BKSLASH_FILL	5	Tô đầy với \\, nét dày
LTBKSLASH_FILL	6	Tô đầy với \\
HATCH_FILL	7	Tô đầy với các hình ca-ro đứng
XHATCH_FILL	8	Tô đầy với các hình ca rô chéo
INTERLEAVE_FILL	9	Tô đầy với các đường thẳng giao
WIDE_DOT_FILL	10	Tô đầy với các điểm chấm xa nhau
CLOSE_DOT_FILL	11	Tô đầy với các điểm chấm gần nhau
USER_FILL	12	Tô đầy với mẫu do lập trình viên quy định

Khi chọn pattern là `USER_FILL` (12) thì mẫu vẽ do lập trình viên quy định trong hàm `setfillpatern()`.

- **color** là màu vẽ, có thể lấy từ bảng 16.4.

9- Hàm vẽ thanh bar() và bar3d()

Các hàm này dùng để vẽ ra màn hình khối hai và ba chiều. Mẫu và màu tô được quy định từ setfillstyle(). Prototype của hai hàm này như sau:

```
void bar( int left, int top, int right, int bottom );
```

```
void bar3d( int left, int top, int right, int bottom, int depth, int topflag );
```

với:

- (**left, top**) là tọa độ trên bên trái của hình cần vẽ

- (**right, bottom**) là tọa độ dưới bên phải của hình cần vẽ

- **depth** là độ sâu của hình 3 chiều, tính bằng pixel

- **topflag** là trị số quy định cho hình khối ba chiều có hay không có nắp tùy theo trị nhận là khác 0 hay bằng 0.

Hàm bar() vẽ hình không có đường viền, trong khi hàm bar3d() vẽ hình có đường viền. Màu của đường viền lại được quy định từ setcolor().

10- Hàm lấy tọa độ hiện thời (CP: current position) của con trỏ getx() và gety()

Dù trong màn hình đồ họa ta không thấy con nháy (cursor) chỉ ra vị trí làm việc hiện thời (CP) trên màn hình, nhưng trong thực tế vẫn có CP. Vị trí hiện thời của CP có thể lấy từ hai hàm getx() và gety(), hai hàm này có prototype như sau:

```
int getx(void);
int gety(void);
```

Hàm getx(), gety() trả về tọa độ theo thứ tự (x, y) của CP.

Ngoài ra, C còn cung cấp hai hàm để lấy ra tọa độ x và tọa độ y lớn nhất trong mode màn hình hiện hành: getmaxx() và getmaxy(), prototype hai hàm này như sau:

```
int getmaxx(void);
int getmaxy(void);
```

Chẳng hạn khi ta đã đặt

```
gd = VGA;
gm = VGAMED; /* kiểu màn hình có độ phân giải: 640x350 */
initgraph( &gd, &gm, "" );
/* nếu khởi động sang chế độ đồ họa không có lỗi thì */
/* tọa độ x lớn nhất là */
x = getmaxx(); /* x = 639 */
/* tọa độ y lớn nhất là */
y = getmaxy(); /* y = 349 */
```

11- Hàm di chuyển CP moveto() và moverel()

Hàm moveto() dùng để di chuyển CP đến vị trí (x, y) là đối số.

Hàm moverel() dùng để di chuyển CP đến vị trí mới, cách vị trí cũ dx pixel theo trục x và dy pixel theo trục y.

Prototype hai hàm này như sau:

```
void moverel(int dx, int dy);
void moveto(int x, int y);
```

Các hàm moveto() và moverel() đều tính tọa độ tương đối từ cửa sổ hiện hành, cửa sổ này được đặt từ hàm setviewprot().

Chương trình ví dụ sau đây cho thấy cách sử dụng của hàm moveto() và moverel().

Ví dụ 16.9

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
main()
{
    int gdriver = DETECT, gmode, errorcode;
    char msg[80];
    initgraph(&gdriver, &gmode, "");
    errorcode = graphresult();
    if (errorcode != grOk) /* an error occurred */
```

```
{\n    printf("Loi: %s\\n", grapherrmsg(errorcode));\n\n    printf("An mot phim de ket thuc:\\n");\n\n    getch();\n\n    exit(1);\n}\n\n/* di chuyen CP den vi tri (10, 20) */\nmoveto(10, 20);\n\n/* Vẽ một điểm sáng tại CP */\nputpixel(getx(), gety(), getmaxcolor());\n\n/* tao va xuat mot cau thong bao tai CP */\nsprintf(msg, "(%d, %d)", getx(), gety());\nouttextxy(10, 20, msg);\n\n/* di chuyen CP den vi tri (10 + 100, 20 + 100) */\nmoverel(100, 100);\n\n/* Vẽ một điểm sáng tại CP */\nputpixel(getx(), gety(), getmaxcolor());\n\n/* tao va xuat mot cau thong bao tai CP */\nsprintf(msg, "(%d, %d)", getx(), gety());\nouttext(msg);\n\ngetch();\nclosegraph();
```

12- Hàm tạo cửa sổ setviewport()

Để tạo một cửa sổ vẽ hình, Cung cấp hàm `setviewport()`. Prototype của hàm này như sau:

void setviewport(int left, int top, int right, int bottom, int clip);

với:

- (*left, top*) là tọa độ góc trên bên trái của cửa sổ.
- (*right, bottom*) là tọa độ góc dưới bên phải của cửa sổ.
- *clip* là một số khác 0 hay bằng 0, quy định hình vẽ có bị cắt hay không bị cắt khi vượt quá cửa sổ hiện hành.

Sau khi đặt cửa sổ xong, CP được đặt tính từ tọa độ (0, 0) của cửa sổ mới. Để xóa nội dung của cửa sổ hiện hành, C cung cấp hàm clearviewport() hàm này có prototype như sau:

void clearviewport(void);

Sau khi xóa cửa sổ xong, CP được dời về vị trí (0, 0) trong cửa sổ hiện hành.

Chú ý:

- 1- Để lấy ra thông tin về cửa sổ hiện thời, C cung cấp hàm getviewsettings(). Hàm này có prototype như sau:

void getviewsettings (struct viewporttype *viewport);

với *viewport* là một biến con trỏ chỉ đến cấu trúc *viewporttype* như sau:

```
struct viewporttype
{
    int left;
    int top;
    int right;
    int bottom;
    int clip;
};
```

- 2- Để xóa toàn bộ màn hình đồ họa bằng màu nền hiện thời. C cung cấp hàm cleardevice(), prototype hàm này như sau:

void cleardevice(void);

Sau khi xóa màn hình xong, CP được dời về tọa độ (0, 0) của màn hình.

13- Hàm vẽ một điểm sáng putpixel()

Hàm này dùng để vẽ một điểm ảnh trên màn hình trong cửa sổ hiện hành. Prototype hàm này như sau:

void putpixel(int x, int y, int color);

với: - **(x, y)** là tọa độ điểm ảnh cần vẽ.

- **color** là màu vẽ điểm ảnh đó. Nó có thể nhận trị trong bảng 12.4. Để lấy thông tin về màu của điểm ảnh tại vị trí **(x, y)**. Cung cấp hàm **getpixel()**, prototype hàm này như sau:

unsigned getpixel(int x, int y);

14- Hàm vẽ đường thẳng line(), lineto() và linerel()

Cung cấp các hàm để vẽ đường thẳng, prototype của các hàm này như sau:

void line(int x1, int y1, int x2, int y2);

void linerel(int dx, int dy);

void lineto(int x, int y);

- **line()** vẽ đường thẳng từ tọa độ **(x1, y1)** đến **(x2, y2)**, không thay đổi CP.
- **linerel()** vẽ đường thẳng từ CP đến vị trí cách nó **dx** theo trục x, và **dy** theo trục y, hàm này có thể cập nhật vị trí CP về vị trí mới.
- **lineto()** vẽ đường thẳng từ CP đến điểm có tọa độ **(x, y)**.
- Màu và mẫu vẽ đường thẳng do **setcolor()** và **setlinestyle()** quy định.

Chương trình sau đây vẽ màn hình hình sin với hàm số:

$$x = a \cdot \sin(2\pi ft)$$

với

- tần số tín hiệu liên tục: **f = 50 Hz**
- thời gian khảo sát: **0 - 1 giây**

Ví dụ 16.10

```
/* chương trình vẽ đồ thị hàm số
x = a . sin (2πft)
```

với:

```
- f = 50 Hz
- t = 0 - 1 giây
*/
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <graphics.h>
#define PI 3.14159
#define X_MAX 500 /* chiều dài trục x: so màu */
#define Y_MAX 400 /* chiều dài trục y */
#define THOI_GIAN_KHAO_SAT 1 /* Thời gian khảo sát: 1 giây */
#define TAN_SO 50
void trucxy (int x, int y);
void sin_t (int Ox, int Oy, double a);
main()
{
    int gd = DETECT, gm;
    int error_code;
    int Ox, Oy;
    initgraph (&gd, &gm, "D:\\bc31\\bgi");
    error_code = graphresult();
    if (error_code != grOk)
    {
        printf("Loi che do hoa: %s \n", grapherrmsg(error_code));
        exit(1);
    }
    Ox = 80;
    Oy = getmaxy()/2;
    trucxy (Ox, Oy);
    sin_t (Ox, Oy, 2);
```

```

getch();
closegraph();
}

void trucxy (int x, int y)
{
    line (x, y, x + X_MAX, y); /* trục x */
    line (x + X_MAX, y, x + X_MAX - 6, y - 4); /* mũi tên trục x */
    line (x + X_MAX, y, x + X_MAX - 6, y + 4);
    outtextxy (x + X_MAX, y + 2, x);
    line (x, y - Y_MAX / 2, x, y + Y_MAX / 2); /* trục y */
    line (x, y - Y_MAX / 2, x - 4, y - Y_MAX / 2 + 5); /* mũi tên trục y */
    line (x, y - Y_MAX / 2, x + 4, y - Y_MAX / 2 + 5);
    outtextxy (x, y - Y_MAX / 2, y);
    outtextxy (x, y + 5, (0, 0)); /* gốc toa do */
}

void sin_t (int Ox, Int Oy, double a)
{
    int i, x, y;
    double t, x_t;
    setcolor (YELLOW);
    moveto (Ox, Oy);
    for (i =0; i <=X_MAX; i++)
    {
        t = (double)i * THOI_GIAN_KHAO_SAT / X_MAX;
        x_t = a * sin (2 * PI * TAN_SO * t);
        x = Ox + i;
        y = (int) (Oy + (x_t / a) * (Y_MAX / 2));
        lineto (x, y);
    }
}

```

Nhìn vào hình vẽ của chương trình ta có thể thấy có 50 chu kỳ dao động sin.

Ngoài ra, C còn cung cấp nhiều hàm khác nữa để vẽ hình và các thao tác xử lý khác. Độc giả có thể tham khảo thêm trong các sách tra cứu về C.

BÀI TẬP CUỐI CHƯƠNG

- 16.1** Viết chương trình vẽ ra màn hình các hàm $\sin x$, $\cos x$, và $\tan x$.
- 16.2** Viết chương trình cho phép nhập một đa thức bất kỳ. Vẽ hàm tạo từ đa thức này.
- Hướng dẫn:* Dùng cấu trúc dữ liệu danh sách liên kết các struct, mỗi struct lưu các thông tin về số hạng thứ i bất kỳ như sau;
- hệ số
 - số mũ
- 16.3** Vẽ ra màn hình chữ nhật.Ấn phím mũi tên để di chuyển hình chữ nhật này.
- 16.4** Viết một chương trình demo tất cả các hàm vẽ hình với các màu đã biết.
- 16.5** Viết chương trình cho phép vẽ ra màn hình một mạch điện bất kỳ, cho phép nhập liệu trong chế độ đồ họa. Giải tích mạch tìm đáp ứng của mạch theo thông số nhập đầu vào.
- 16.6** Viết phần mềm cho phép vẽ một hình không gian bất kỳ. Chỉ rõ phần thấy và phần khuất của hình.
- 16.7** Viết chương trình tìm nghiệm gần đúng của hai hàm f_1 và f_2 bất kỳ. Dùng phương pháp đồ thị vẽ hai hàm và hai nghiệm này. Phương trình được tạo từ $f_1 - f_2 = 0$ có nghiệm không?
- 16.8** Vẽ một hình không gian và quay nó theo phương bất kỳ.
- 16.9** Vẽ một hình bất kỳ, lưu dữ liệu về hình đó lên đĩa dưới dạng một file nhị phân. Sau đó chạy lại phần mềm để hiện lại hình đó.
- Hướng dẫn:* đọc thêm về hàm `getpixel()`.
- 16.10** Đọc một file .GIF (hay một file hình ảnh có dạng lưu trữ đã biết), hiện ra màn hình hình mà file dữ liệu đó đang lưu.

Chương 17

LỆNH TIỀN XỬ LÝ

Trong C có một loại lệnh đặc biệt được C thực hiện trước khi dịch chương trình, đó là lệnh tiền xử lý.

17.1 #DEFINE VÀ #UNDEF

Lệnh tiền xử lý này cho phép định nghĩa một chuỗi hay biểu thức bằng một tên nào đó, tên này gọi là macro. Trong chương trình, bất kỳ chỗ nào muốn sử dụng chuỗi hay biểu thức này, ta chỉ cần ghi tên macro đặc trưng của nó. Trước khi dịch chương trình, bộ dịch C sẽ thay thế những tên macro đã được định nghĩa bằng biểu thức thật của nó. Việc thay thế này được thực hiện trong toàn bộ tập tin kể từ chỗ bắt đầu có nó cho đến hết tập tin.

Cú pháp của lệnh tiền xử lý này như sau:

```
# define tên_macro biểu_thức_được_thay_thế
```

với:

- **tên_macro** là một danh hiệu không chuẩn, là tên sẽ được gọi sử dụng, thay thế cho chuỗi hay biểu thức trong chương trình. Theo sau tên macro có thể có tham số.
- **biểu_thức_được_thay_thế** cách tên_macro ít nhất một khoảng trắng (blank hoặc tab). Biểu thức này có thể không có, khi đó lệnh define chỉ đơn giản là khai báo một kiểu mà thôi.

Ta có thể có hai dạng thay thế macro: thay thế macro đơn giản và thay thế macro có tham số

- Thay thế macro đơn giản: một hình thức thay thế macro một cách máy móc.

Ví dụ 17.1

```
#define MAX 100
```

Khi đó, lúc dịch nếu gặp MAX ở bất kỳ vị trí nào trong chương trình (mà không nằm trong một chuỗi hoặc không là một phần của một tên nào đó) thì C sẽ lập tức thay thế nó bằng 100.

Ví dụ 17.2

Lệnh

```
for (i = 0; i < MAX; i++)...
```

sẽ là

```
for (i = 0; i < 100; i++)...
```

Ví dụ 17.3

Nếu đã định nghĩa ở đầu chương trình:

```
#define HO_TEN "Nguyen Van An\n"
```

thì sau đó lệnh

```
printf(HO_TEN);
```

khi được dịch sẽ là:

```
printf ("Nguyen Van An \n");
```

- Thay thế macro có tham số: C có thể định nghĩa một macro có tham số thay thế cho một biểu thức được đặt trong cặp dấu ngoặc theo sau nó. Khi sử dụng macro, ta có thể đưa đổi số thật của nó. Như vậy về hình thức sử dụng, macro dạng này tương tự như hàm, nên nó còn được gọi là hàm trực tiếp (inline function).

Ví dụ 17.4

```
#define MAX(x,y) ( (x) > (y) ? (x) : (y) )
```

thì MAX(x,y) được gọi là macro trong đó x và y được xem như đối số của nó. Việc thay thế macro này sẽ được C thực hiện với cả các đối số này.

Ví dụ 17.5

Lệnh gán sau đây:

```
c = MAX(a,b);
```

sẽ được thay thế thành

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

Với macro này, ta gần như đã có một hàm tìm số lớn hơn trong hai số mà không cần định nghĩa một hàm cụ thể nào nữa. Mặt khác, so với hàm, macro không cần biết kiểu của các đối số là gì, do đó nó có khả năng sử dụng rộng rãi cho mọi kiểu dữ liệu phù hợp.

Chú ý rằng các dấu ngoặc bao quanh từng đối số và cả biểu thức thực ra rất quan trọng vì nếu không có chúng thì trong nhiều trường hợp kết quả thay thế có thể không đúng.

Ví dụ 17.6

Nếu viết một macro như sau:

```
#define MAX(x,y) x > y? x: y
```

Khi sử dụng macro này trong một lệnh như sau:

```
c = MAX(a,b) + 4;
```

thì kết quả sẽ là:

```
c = x > y? x: y + 4;
```

khi đó biến c sẽ được gán trị là x (nếu $x > y$) hoặc $y + 4$ (nếu $x \leq y$) và điều này sai với ý đồ là nếu $x > y$ thì c sẽ được gán trị là $x + 4$, còn khi $x \leq y$ thì c sẽ được gán trị $y + 4$.

Các chú ý khi viết macro.

- Không thể có khoảng trống giữa tên macro và dấu mở ngoặc đối số

Ví dụ 17.7

Không thể viết:

```
#define MUL (x,y) ((x)*(y))
```

↳ *khoảng trống cần loại bỏ*

- Không quan tâm kiểu đối số, cũng như ngữ pháp của biểu thức

Ví dụ 17.8

Ta có thể viết một macro như sau:

```
#define CONG (a+      ← C không báo lỗi tại đây
```

và dùng nó trong lệnh:

```
x = CONG (b);
```

khi đó lúc dịch sẽ có:

```
x = (a+b);
```

và lệnh này hoàn toàn đúng cú pháp nên C không báo lỗi gì cả.

- Khi sử dụng macro có đối số, việc thay thế các đối số sẽ diễn ra một cách máy móc, điều này không cho phép chúng ta thực hiện những phép toán kết hợp tùy tiện.

Ví dụ 17.9

Xét macro sau đây

```
#define kt_hoa(c) ( ((c) <= 'z' && (c) >= 'a')? (c)-32:(c) )
```

khi dùng nó, ta không thể kết hợp nó với các lệnh gọi hàm hay gán lại.

Ví dụ 17.10

```
c = kt_hoa(getch());
```

thì khi dịch nó sẽ trở thành:

```
c =(((getch()) <= 'z' && (getch()) = 'a')?(getch ()) - 32: (getch()));
```

điều này hoàn toàn sai ý nghĩa vì lệnh trên khi gọi macro có đối số sẽ được gọi hàm getch() 4 lần khác nhau, trong khi ta chỉ muốn một ký tự được kiểm tra mà thôi.

- Trong macro bình thường không có dấu chấm phẩy (trừ khi bản thân macro cần thiết phải có) theo sau một định nghĩa macro, vì bản thân dấu này cũng sẽ được đặt vào thay thế, và có thể gây ra sai lầm.
- Khi viết macro trên nhiều hàng, ta phải có dấu \ trước khi xuống hàng mới.

Ví dụ 17.11

```
#define isalpha(c) ((c) >= 'A' && (c) <= 'Z') || \
                    ((c) >= 'a' && (c) <= 'z')

#define swap(a,b) { int temp = a; \
                    a = b; \
                    b = temp; \
                }
```

- Trong biểu_thức_được_thay_thế của một macro có thể dùng một tên macro nào đó đã được định nghĩa trước đó. Khi đó, sẽ xảy ra sự thay thế lồng nhau.

Ví dụ 17.12

```
#define PI 3.14159
#define cv(r) (PI*2*(r))
```

Khi đó lệnh:

```
printf ("Chu vi hình tròn có bán kính %5.2f là %5.2f", a, cv(a));
```

sẽ trở thành

```
printf ("Chu vi hình tròn có bán kính %5.2f là %5.2f", a, (PI*2*(a)));
```

Trong những mở rộng của C, người ta còn đưa ra một lệnh tiền xử lý đi cặp với **#define** là lệnh **#undef**. Lệnh **#undef** sẽ hủy bỏ việc định nghĩa một macro trước đó đã được định nghĩa bằng một lệnh define nào đó.

Ví dụ 17.13

Ta có chương trình ví dụ sau:

```
#include <stdio.h>
#include <conio.h>
main()
{
    clrscr();
    #define ht "Đang\n"
    printf (ht);
    #undef ht
```

```
#define ht "Nguyen\n"
printf (ht);
getch();
}
```

Chương trình cho xuất liệu là:

```
Dang
Nguyen
```

Lệnh `#undef` thường được dùng khi người ta muốn hủy bỏ những tác dụng của các lệnh `#define` trước đó để define lại một macro cùng tên bằng một biểu thức khác.

- Các tên macro có thể được viết chữ in hay chữ thường tùy ý, nhưng chúng thường được viết là chữ in, để dễ phân biệt với các tên biến và hàm trong chương trình.

17.2 #INCLUDE

C cho phép đưa một tập tin khác lồng vào tập tin đang làm việc dưới dạng nguồn bằng lệnh tiền xử lý `#include`.

Cú pháp lệnh này như sau:

```
#include "têntậptin"
#include <têntậptin>
```

với `têntậptin` là tên của tập tin nào đó mà ta muốn đưa vào tập tin đang làm việc.

Có hai cách viết khác nhau cho lệnh `include`, như

```
#include "stdio.h"
```

và

```
#include <stdio.h>
```

Ở cách viết đầu, tập tin `stdio.h` chỉ được tìm trong thư mục hiện thời. Còn trong cách viết dưới, C tìm tập tin `stdio.h` trong thư mục hiện thời, nếu không thấy C sẽ tìm tập tin này trên thư mục riêng cho các tập tin `include`, mà chúng ta đã xác định trước cho chương trình biên dịch. Do vậy, cách viết này thường được dùng khi tập tin chương trình cần kéo vào một tập tin header chuẩn của chương trình biên dịch cung cấp, như `stdio.h`, `conio.h`,...

Ví dụ 17.14

```
#include <conio.h>
#include "myheader.h"
```

Khi gặp lệnh này, C sẽ thay dòng lệnh này bằng nội dung của tập tin có tên tương ứng vào chương trình của chúng ta, lúc đó chương trình có thể xem như được viết có sẵn nội dung của tập tin đã được include.

Lệnh này thường được dùng để kéo các phần khai báo chung đã viết sẵn cho nhiều chương trình vào chương trình của mình mà không phải viết lại chúng. Ví dụ như các file.h chứa các **prototype** của hàm và các định nghĩa kiểu cần thiết cho hàm thư viện, có thể được đưa vào chương trình của chúng ta bằng lệnh include với tên file tương ứng. Các khai báo hàm dùng chung cho nhiều chương trình hoặc các định nghĩa kiểu do lập trình viên tự khai báo có thể được để trong một file nào đó, khi chương trình nào sử dụng các khai báo đó sẽ ra lệnh include file này vào chương trình, ví dụ: #include "myheader".

Ví dụ 17.15 Có hai chương trình được lưu trong hai tập tin ct1.C và ct2.c dùng một số khai báo hằng, biến, kiểu giống nhau. Để gọn cho các chương trình, ta có thể viết các khai báo giống nhau này ở một tập tin .h, ví dụ header.h như sau:

```
#define HO_TEN "Nguyen Van Nam\n"
#define TRUE 1
#define FALSE 0
typedef struct date
{
    int day;
    int month;
    int year;
    char dayofweek[8];
} DATE;
int a, b, c;
```

và ra lệnh đưa tập tin này vào đầu các tập tin ct1.c và ct2.c bằng lệnh #include

```

ct1.c           ct2.c
#include <stdio.h>      #include <conio.h>
#include "header.h"      #include "header.h"
main()          main()
{
    DATE dt;        a = 2;
                    b = 3;
    dt.month = 2;...
    dt.year = 1990; }

}

```

Khi đó nội dung của file header.h đã được đưa vào chương trình trong các file ct1.c và ct2.c lúc dịch, và ta có thể sử dụng các biến, hằng a, b, c, DATE,... trong hàm main().

Các chú ý:

- Phần mở rộng của các file dùng chung là bất kỳ, có thể là .c, .bak,... tuy nhiên ta thường để là .c hoặc .h theo ý nghĩa mà tập tin này được viết: nếu .c: hàm ý rằng modul này được viết bằng ngôn ngữ C, nếu .h: hàm ý modul này chứa các khai báo đầu như biến, hằng, kiểu, prototype của hàm...

Ví dụ 17.16

```

#include "begin.c"
#include "myheader.h"

```

- Tập tin được kéo vào một chương trình bằng lệnh #include cũng có thể có các lệnh include những tập tin khác nữa trong nó.

17.3 #IF-ELSE #ENDIF

Ngoài các lệnh tiền xử lý #define và #include rất thường được sử dụng, C còn có các lệnh:

```

#if hoặc #ifdef hoặc #ifndef
#elif
#else
#endif

```

Các lệnh tiền xử lý này phục vụ cho khả năng biên dịch có điều kiện của C.

1- Lệnh #ifdef, #else và #endif

Lệnh tiền xử lý #ifdef cho phép nếu danh hiệu theo sau nó đã được định nghĩa trước đó bằng lệnh tiền xử lý #define nào đó thì tất cả các lệnh tiền xử lý, các chỉ thị dịch nằm trước #else hoặc trước #endif đều được C dịch, và mã của các lệnh này sẽ được dùng trong chương trình. Nếu có một #else thì mọi lệnh từ #else đến #endif sẽ được C thực hiện nếu danh hiệu theo sau #ifdef chưa được định nghĩa. Cú pháp cụ thể như sau:

```
#ifdef danh_hiệu
    lệnh, khai báo...
#else
    lệnh, khai báo...
#endif
```

Phần #else có thể không có.

Ví dụ 17.17

```
#ifdef NGUYEN
    #include "danh.h"      → phần này được thực hiện nếu
    #define TUOI 5          NGUYEN đã được định nghĩa
#else
    #include "anh.h"        → phần này được thực hiện nếu
    #define TUOI 15          NGUYEN chưa được định nghĩa
#endif
```

Xét chương trình ví dụ sau đây

Ví dụ 17.18

```
#include <stdio.h>
#include <conio.h>
#define CHECK
#ifndef CHECK
    int tong (int a, int b)
```

```

    {
        return a + b;
    }
#else
    int hieu (int a, int b)
    {
        return a - b;
    }
#endif
main()
{
    int a = 6, b = 9;
    clrscr();
    #ifdef CHECK
        printf ("Tong hai so la: %d", tong (a, b));
    #else
        printf ("Hieu hai so la: %d", hieu (a, b) );
    #endif
    getch();
}

```

Chương trình sẽ cho xuất liệu

Tong hai so la: 15

Tuy nhiên nếu bỏ lệnh `#define CHECK`, hoặc thêm lệnh `#undef CHECK` vào chương trình, chương trình thành

```

#include <stdio.h>
#include <conio.h>
#define CHECK
#undef CHECK
#ifndef CHECK
    int tong (int a, int b)
    {
        return a + b;
    }
#else
    int hieu (int a, int b)
    {
        return a - b;
    }

```

```

    }
#endif
main()
{
    int a =6, b =9;
    clrscr();
    #ifdef CHECK
        printf ("Tong hai so la: %d", tong (a, b) );
    #else
        printf ("Hieu hai so la: %d", hieu (a, b) );
    #endif
    getch();
}

```

thì chương trình sẽ in ra kết quả

Hieu hai so la: -3

Một dạng khác của lệnh `#ifdef` là lệnh `#ifndef`. Lệnh này ngược lại với lệnh `#ifdef`, nếu danh hiệu chưa được định nghĩa thì các lệnh ngay sau nó sẽ được thực hiện. Cú pháp của lệnh này như sau:

```

#ifndef danh_hieu
    lệnh, khai báo...
#else
    lệnh, khai báo...
#endif

```

Ví dụ 17.19

```

#ifndef NGUYEN
    #include "danh.h"      → phần này được thực hiện nếu
    #define TUOI 5          NGUYEN chưa được định nghĩa

#else
    #include "anh.h"       → phần này được thực hiện nếu
    #define TUOI 15         NGUYEN đã được định nghĩa

#endif

```

C còn cung cấp thêm một cách thứ hai để kiểm tra xem một danh hiệu đã được khai báo chưa, đó là toán tử tiền xử lý `defined`.

Thay vì viết

```
#ifdef TEN
```

ta có thể viết

```
#if defined TEN
```

Nhờ các lệnh tiền xử lý này, khi viết chương trình chúng ta rất linh động trong việc chọn lựa mã lệnh cho chương trình.

2- Lệnh `#if`, `#elif` và `#endif`

Lệnh `#if` cũng tương tự như lệnh `if` thường của C, theo sau nó sẽ là một biểu thức hằng. Nếu biểu thức này có trị số khác 0 thì các lệnh ngay sau nó sẽ được dịch, còn ngược lại, các lệnh ngay sau `#elif` (nếu có) sẽ được dịch.

Cú pháp lệnh

```
#if <biểu_thức_hằng>
    lệnh
#else
    lệnh
#endif
#if <biểu_thức_hằng>
    lệnh
#elif <biểu_thức_hằng>
    lệnh
#endif
```

phần `#else` hoặc `#elif` có thể không có.

Ví dụ 17.20

```
#define MODE 1 ← Định nghĩa MODE là 1
#if MODE ==1 ← Nếu MODE là 1 thì phần chương trình này được biên dịch
    #include "myhead1.h"
#elif MODE == 2 ← Nếu MODE là 1 thì phần chương trình này được biên dịch
    #include "myhead2.h"
#else ← Nếu MODE khác thì phần chương trình này được biên dịch
    #include "myhead3.h"
#endif
```

Ví dụ 17.21

```
#define IBMPC
#if defined (IBMPC)
    #include "myhead1.h"
#elif defined (MAC)
    #include "myhead2.h"
#else
    #include "myhead3.h"
#endif
```

17.4 #ERROR

Lệnh này bắt buộc chương trình dịch thôi biên dịch chương trình khi gặp nó và báo ra câu thông báo lỗi theo sau nó. Cú pháp lệnh này như sau:

```
#error thông_báo_lỗi
```

với **thông_báo_lỗi** được viết không có nháy kép;

Khi gặp lệnh này, chương trình dịch sẽ dừng lại, in thông báo lỗi ra rồi không dịch tiếp nữa. Lệnh này thường được dùng kèm với **#if #else #endif** để điều khiển quá trình dịch chương trình.

Ví dụ 17.22

```
#if defined BC5
    #error Dịch lại chương trình, bộ dịch là VC6
#endif
```

Khi đó, trong lúc dịch chương trình, C báo lỗi

Dịch lại chương trình, bộ dịch là VC6

Ngoài ra, C còn có số lệnh tiền xử lý khác nữa như **#line**, **#pragma**, độc giả có thể đọc thêm trong các sách về lập trình C.

BÀI TẬP CUỐI CHƯƠNG

- 17.1 Định nghĩa một hàm macro trả về số nhỏ nhất trong hai số.
- 17.2 Định nghĩa một hàm macro tính tổng nghịch đảo của hai số.
- 17.3 Viết một chương trình với các macro cần thiết để chuyển tọa độ một vector trong tọa độ cực sang tọa độ Descartes. Cho các biểu thức quan hệ:

$$x = r \cos (\theta)$$

$$y = r \sin (\theta)$$

- 17.4 Xem các macro sau đây macro nào viết đúng, macro nào sai?

a. #define FPM 5280

 distance = FPM * miles;

b. #define FEET 4

 #define POD FEET + FEET

 plot = FEET * POD;

c. #define SEVEN = 7

 next = SEVEN;

Chương 18

CÁC HÀM TRONG THƯ VIỆN CHUẨN CỦA C

Chương này trình bày thêm các hàm trong thư viện chuẩn của C, các hàm này đều có trong mỗi bộ dịch C. Tuy nhiên, để tăng sức mạnh của từng bộ dịch, các hãng thiết kế phần mềm đã tăng cường thêm nhiều hàm khác nữa, các hàm mở rộng này không được trình bày trong giáo trình này.

18.1 THƯ VIỆN CỦA C

Các hàm thư viện của các chương trình biên dịch C nằm trong các tập tin thư viện riêng, đó là các tập tin có phần mở rộng .LIB, các tập tin này nằm trong thư mục \LIB của thư mục chứa bộ dịch C.

Như vậy, các chương trình C được viết ra bình thường sẽ phụ thuộc vào các hàm thư viện của một bộ dịch C nào đó. Do đó, để việc sử dụng thư viện trở nên thống nhất hơn và làm cho các chương trình dễ mang đi từ máy này sang máy khác, từ bộ dịch này sang bộ dịch khác, các thư viện C ngày nay thường có các chức năng, tên hàm, đối số giống nhau, các chức năng này gần như đã được chuẩn hóa. Các chức năng đó tạo ra thư viện chuẩn của C, bao gồm các nhóm chức năng sau:

- Truy xuất thiết bị xuất nhập, tập tin.
- Giao tiếp với hệ điều hành
- Xử lý chuỗi
- Xử lý toán học
- Quản lý bộ nhớ
- Xử lý ngày giờ.

...

Bên cạnh những hàm chuẩn, các nhà sản xuất chương trình biên dịch thường đưa ra thêm các hàm xử lý đặc biệt hơn như xử lý màn ảnh graphics, cửa sổ,... hoặc mở rộng thêm các chức năng chuẩn đã quy định. Việc sử dụng các hàm thư viện như vậy, trong nhiều trường hợp làm cho chương trình dễ viết hơn, hoặc xử lý được những chức năng đặc biệt hơn. Nhưng khi đó việc mang chương trình sang một chương trình biên dịch khác sẽ không được chấp nhận.

Lập trình viên có thể tạo cho mình một thư viện riêng, chứa các hàm xử lý đặc biệt mà trong thư viện chuẩn không có. Để làm được việc này, độc giả có thể sử dụng các chương trình khảo sát và quản lý thư viện được bán kèm theo bộ dịch C (như chương trình LIB.EXE).

18.2 THAO TÁC XUẤT NHẬP CHUẨN

Để hiểu rõ hơn về các hàm xuất nhập chuẩn, ta cần nắm hai khái niệm: dòng xuất nhập (*I/O stream*) và tập tin (*file*).

Để có thể giao tiếp với một thiết bị xuất nhập một cách tổng quát, không phụ thuộc vào loại thiết bị mà chương trình C sẽ giao tiếp, người ta đưa ra khái niệm dòng xuất nhập (*I/O stream*).

Có hai loại dòng xuất nhập: dòng xuất nhập kiểu văn bản và dòng xuất nhập kiểu nhị phân.

1- Dòng xuất nhập kiểu văn bản (*text stream*): một text stream là một dòng xuất nhập cho phép truy xuất một loạt các ký tự được xếp thành một hàng tận cùng bằng ký tự xuống hàng ('\n'). Ở kiểu văn bản, có thể có sự chuyển đổi ký tự khi in ra thiết bị thực bên ngoài, ví dụ ký tự xuống hàng ('\n') khi xuất ra sẽ được chuyển thành một cặp ký tự: về đầu dòng ('\r') và xuống hàng ('\n'): "\n\r". Như vậy trên thiết bị thực, chúng ta có thể không hoàn toàn nhận đúng số ký tự mà chúng ta đã truy xuất.

Cũng tương tự, khi truy xuất một thiết bị thực bên ngoài (ví dụ ghi vào file) qua một dòng xuất nhập kiểu văn bản, nếu nhận được cặp ký tự về đầu dòng và xuống hàng ("\r\n") thì dòng xuất nhập này sẽ chuyển đổi thành một ký tự duy nhất: xuống hàng ('\n').



2. Dòng xuất nhập kiểu nhị phân (binary stream): một binary stream là một dòng xuất nhập cho phép chuyển một loạt các ký tự mã ra thiết bị thực mà không có một sự chuyển đổi ký tự nào. Và như vậy số ký tự xuất nhập giữa chương trình với thiết bị thực luôn luôn giống nhau.

Tập tin (*file*) lại là một khái niệm để chỉ một môi trường cụ thể mà trên đó ta thực hiện các thao tác xuất nhập. Đối với C, tập tin có thể được hiểu là bất cứ thiết bị giao tiếp gì, từ đĩa cho đến bàn phím và màn hình. Để truy xuất một file, chúng ta phải gán một dòng xuất nhập cho nó trước khi chương trình thao tác trên tập tin đó và ra lệnh mở tập tin lại, để bắt tất cả các thông tin xuất nhập phải được lấy hết khỏi dòng xuất nhập, điều này hạn chế mất thông tin khi truy xuất file.

Mỗi khi chạy chương trình, C đã quy định các dòng xuất nhập chuẩn. Các dòng xuất nhập này được C tự động mở mỗi khi chạy chương trình. Các dòng xuất nhập này là:

- **stdin** : nối với thiết bị nhập chuẩn theo hệ điều hành, thông thường thiết bị này là **bàn phím**.
- **stdout** : nối với thiết bị xuất chuẩn theo hệ điều hành, thường thiết bị này là **màn hình**.
- **stderr** : nối với thiết bị báo lỗi chuẩn theo hệ điều hành, thường thiết bị này là **màn hình**.
- **stdprn**: nối với thiết bị in chuẩn, thường là **máy in**.
- **stdaux**: nối với thiết bị giao tiếp phụ.

Như vậy, tổng quát C xem các thiết bị xuất nhập trên cũng như các tập tin khác và các tập tin này không cần mở cũng không cần đóng lại vì C đã tự động làm các công việc này mỗi khi chạy chương trình. Các dòng xuất nhập này luôn luôn được mở theo kiểu văn bản, có nghĩa là luôn luôn xảy ra việc chuyển đổi giữa '\n' và '\r\n'.

18.2.1 Các hàm xử lý xuất nhập chuẩn

Các hàm getchar(), putchar(), scanf() và printf() là các hàm xuất hoặc nhập trên các thiết bị chuẩn stdin và stdout mà ta xét ở đây.

1- Hàm getchar()

Hàm getchar() cho phép đọc vào từ stdin một ký tự và cho phép trả về ký tự đó như kết quả của nó, getchar() sẽ cho kết quả là EOF (định nghĩa trong tập tin stdio.h) khi nó đọc đến cuối tập tin (gặp ký tự kết thúc tập tin). Việc trả ký tự về từ hàm chỉ xảy ra khi gặp ký tự xuống hàng, nếu nhập ký tự từ bàn phím ta cần ấn phím Enter để hàm getchar() trả về ký tự đã nhập.

2- Hàm putchar()

Hàm putchar(c) xuất ký tự c ra stdout, mà thông thường là ra màn hình. Dùng getchar() và putchar() nói chung ta có thể làm việc trên từng ký tự của stdin và stdout.

Hàm getchar() và putchar() thực ra chỉ là các macro được định nghĩa trong stdio.h.

3- Hàm printf() và scanf()

Các hàm printf() và scanf() là hai hàm xuất nhập chuẩn từ stdin hoặc ra stdout. Các hàm này cho phép in ra hoặc đọc vào các giá trị theo một dạng đã được quy định trước trong một chuỗi định dạng xuất nhập nào đó.

a) *Hàm printf() được viết tổng quát như sau:*

```
int printf(chuỗi_định_dạng, tham_số1, tham_số2,...)
```

Hàm **printf()** theo chuỗi_định_dạng, lấy giá trị các tham_số đặt vào theo yêu cầu của chuỗi_định_dạng và gửi ra thiết bị stdout.

chuỗi_định_dạng là một chuỗi ký tự bình thường trong đó có những ký tự hằng cần xuất ra nguyên vẹn, và có thể có những ký tự thay thế cho các trị lấy từ các tham số sau đó để in ra.

Các ký tự thay thế cho các trị để in ra luôn được bắt đầu bằng dấu phần trăm (%) và ta gọi là chuỗi định dạng xuất nhập. Dạng của chuỗi này như sau:

```
% [ flag ] [ width ] [ .prec ] [ F: N: h: l: L ] type
```

với: - *type* là kiểu của tham số theo sau chuỗi_định_dạng, để lấy giá trị xuất nhập.

type có thể là một trong các ký tự sau:

- d, i số nguyên dạng cơ số 10
- o số nguyên dạng cơ số 8
- u số nguyên dạng cơ số 10, không dấu
- x số nguyên dạng cơ số 16, chữ thường (a, b,... f)
- X số nguyên dạng cơ số 16, chữ hoa (A, B,... F)
- f số thực có dạng [-]ddd.dddd...
- e số thực có dạng [-]d.ddd e[+/-]ddd
- E số thực có dạng [-]d.ddd E[+/-]ddd
- g số thực có dạng theo e hay f tùy theo độ chính xác
- G số thực có dạng theo E hay f tùy theo độ chính xác
- c ký tự có mã tương ứng
- s chuỗi ký tự tận cùng bằng '\0'
- % dấu % cần in
- p pointer
- n số ký tự in ra được gửi trả ra cho * tham_số

- flag canh chỉnh lề khi xuất liệu

Nếu flag là

- (không có) in ra theo dạng canh phải
- in ra theo dạng canh trái
- + luôn luôn bắt đầu bằng dấu + hoặc dấu -
- (trống) chỉ ra dấu trừ khi số âm.
- # in ra tùy theo type
- c, s, d, i, u không ảnh hưởng
- o chèn thêm số 0 đằng trước cho giá trị >0
- x, X chèn thêm 0x hoặc 0X đằng trước số này
- e, E, f luôn luôn có dấu chấm thập phân
- G, g như trên nhưng không có các số 0 đi sau

- *width* định kích thước in ra

Nếu width là

- | | |
|----|--|
| n | dành chỗ xuất liệu ít nhất là n ký tự, điền
khoảng trắng vào các vị trí còn dư. |
| 0n | dành chỗ ít nhất là n ký tự, điền số 0
vào các vị trí còn dư. |

* Số ký tự ít nhất cần in nằm ở tham_số tương ứng.

- *prec* định số lẻ sau dấu chấm thập phân cần in ra

(không có) độ chính xác theo dạng máy lưu trữ

- | | | |
|----|---------------|-----------------------------|
| .0 | d, i, o, u, x | độ chính xác mặc nhiên |
| | e, E, f | không in dấu chấm thập phân |

- | | |
|----|------------------------|
| .n | nhiều nhất là n ký tự. |
|----|------------------------|

- | | |
|----|--|
| .* | số nhiều nhất cần in ở tham_số tương ứng |
|----|--|

- Các bối cảnh kiểu khi xuất nhập

F tham_số là far pointer XXXX:YYYY

N tham_số là near pointer YYYY

h tham_số là short int

l tham_số là long int đối với các kiểu: d, i, o, u, x, X

tham_số là double đối với các kiểu: e, E, f, g, G

L tham_số là long double đối với các kiểu: e, E, f, g, G

tham_số: là trị số hằng hay biến cần in ra theo chuỗi định dạng, số tham số tùy theo chuỗi định dạng đòi hỏi và sẽ được lấy theo đúng thứ tự mà chuỗi đang cần.

Hàm printf() sẽ trả về số ký tự đã in ra được

b) Hàm scanf() có dạng tương tự như printf()

int scanf(chuỗi định dạng, địa chỉ tham số1, địa chỉ tham số2,...)

Trong đó chuỗi định dạng cũng là một chuỗi ký tự có chứa các chuỗi ký tự điều khiển nhập giống như printf(). Chuỗi này dùng để

xác định cách lấy các ký tự phù hợp từ chuỗi nhập vào từ stdin, biến đổi theo kiểu trong chuỗi điều khiển, tạo trị và cất trị vào các ô nhớ có địa chỉ cung cấp: `địa_chỉ_tham_số1,...` theo sau.

`chuỗi định dạng` của `scanf()` gồm có ba loại:

- Chuỗi điều khiển
- Ký tự trắng
- Ký tự khác ký tự trắng
- Chuỗi điều khiển của `scanf()` quyết định kiểu của giá trị sẽ được gán vào cho biến, có dạng:

`% [width][h:1] type`

với: `type` xác định kiểu của giá trị nhập vào biến có địa chỉ là các đối số `địa_chỉ_tham_số` theo sau chuỗi định dạng. Nó có thể là:

- d, i nhập vào số nguyên dạng cơ số 10
- o nhập vào số nguyên dạng cơ số 8
- u nhập vào số nguyên dạng cơ số 10, không dấu
- x nhập vào số nguyên dạng cơ số 16
- f, c nhập vào số thực
- c nhập vào một ký tự
- s nhập vào một chuỗi ký tự
- p nhập vào pointer
- n trả lại số ký tự đã nhập được từ đầu đến đó

`width` xác định số ký tự tối đa sẽ được nhập vào

`scanf()` sẽ chỉ nhận số ký tự cho đến khi đủ `width` ký tự hoặc cho đến khi gặp ký tự trắng đầu tiên thì số ký tự trước đó sẽ được nhận, được biến đổi và ghi vào biến. Nếu chuỗi nhập vào nhiều hơn, phần còn lại sẽ được dành lại cho lần gọi `scanf()` kế tiếp.

Ví dụ 18.1

```
scanf("%4s",str); /* str đã khai báo là char str[10] */
```

thì `scanf()` sẽ chờ nhận vào tối đa là 4 ký tự, và cất vào mảng `str`. Nếu nhập vào chuỗi:

ABCDEF ←
G

thì chuỗi ABCD được cất vào mảng str, còn EFG sẽ được lấy cất vào mảng kế tiếp nếu như sau đó lại có một lệnh gọi scanf("%s", str) khác.

- Ký tự trắng nếu có trong chuỗi định dạng sẽ yêu cầu scanf() bỏ qua một hoặc nhiều ký tự trắng trong chuỗi nhập vào (ký tự trắng được hiểu là ký tự blank (' '), ký tự tab ('\t') hoặc ký tự xuống hàng '\n'). Một ký tự trắng trong chuỗi định dạng sẽ được hiểu là chờ nhập đến ký tự khác trắng kế tiếp.

Ví dụ 18.2

Nếu chúng ta viết một lệnh gọi hàm scanf() như sau:

```
scanf("%d", &n);
      ↑
      |   có một khoảng trắng
```

Thì hàm scanf tiếp tục chờ nhận cho đến khi ta đánh một ký tự trắng rồi thì mới thoát ra khỏi hàm scanf() và trở về chương trình chính. Ký tự trắng đó vẫn còn trong vùng đệm và được lấy bởi các hàm scanf() hoặc gets() sau đó.

- Ký tự khác trắng nếu có trong chuỗi định dạng sẽ làm cho scanf() nhận vào và bỏ qua đúng ký tự như thế.

Ví dụ 18.3

Nếu chúng ta viết một lệnh gọi hàm scanf() như sau:

```
scanf("%d/%d", &d, &m);
```

thì ta cần nhập vào một số nguyên, trị này được cất vào biến d, kế tiếp cần nhập một dấu /, scanf() bỏ dấu này đi, và chờ nhận một số nguyên kế tiếp để cất vào m. Nếu không không gặp dấu / kế tiếp số nguyên đầu tiên thì scanf() chấm dứt việc nhập trị.

Hàm scanf() trả về số trị đã được đổi và ghi vào biến.

c) Hàm sprintf() và sscanf()

Có hai hàm tương tự như printf() và scanf() là sprintf() và sscanf(). Hai hàm này thực hiện việc gửi ra hoặc nhận vào từ một chuỗi ký tự, chứ không phải từ thiết bị vào hoặc thiết bị ra chuẩn.

- Hàm sprintf() cũng có dạng giống như printf() nhưng có thêm một chuỗi_ra để nhận kết quả tạo được từ chuỗi định_dạng và các tham số. Dạng gọi tổng quát của hàm này như sau:

. sprintf(chuỗi_ra, chuỗi định_dạng, tham_số1, tham_số2,...)

Ví dụ 18.4 Có khai báo

```
char str[20];
int i = 2, j = 3;
```

Nếu gọi hàm như sau:

```
 sprintf(str, "%d %d", i, j);
```

thì chuỗi str sẽ là "2 3".

- Hàm sscanf() lại tương tự như hàm scanf(), nhưng cũng có thêm một chuỗi_vào, để từ đó lấy ra các thành phần theo chuỗi định_dạng và cất vào các biến có địa chỉ là các tham số địa_chỉ_tham_số theo sau. Dạng tổng quát của hàm như sau:

. sscanf(chuỗi_vào, chuỗi định_dạng, địa_chỉ_tham_số1,...)

Hàm này sẽ phân tích chuỗi_vào (thay vì phân tích chuỗi ký tự nhận vào từ stdin như ở scanf()) và tách thành các vùng theo chuỗi_dạng, biến đổi ra các trị phù hợp và cất vào đối tượng được chỉ bởi các địa chỉ tương ứng.

Ví dụ 18.5

Nếu str là chuỗi ký tự myprog.c và name, ext là các mảng ký tự, thì lệnh gọi:

```
 sscanf(str, "%s.%s", name, ext);
```

sẽ cất vào name chuỗi myprog và cất vào ext chuỗi c.

18.2.2 Các hàm xử lý xuất nhập tập tin

Các hàm xử lý chuẩn, xử lý tập tin bình thường của C gồm có:

`fopen()`

`fscanf()`

`fclose()`

`feof()`
`putc()`
`ferror()`
`getc()`
`rewind()`
`fseek()`
`fprintf()`

Các hàm này đã được trình bày trong chương 15, kiểu tập tin (*file*). Ngoài ra có một số hàm khác cho phép truy xuất file được trình bày dưới đây:

int remove(char * tentaptin);

Hàm này cho phép ta xóa một tập tin nào đó. Hàm này sẽ trả về 0 nếu xóa được, nếu có lỗi, thì trả về giá trị khác 0.

void clearerr(FILE * fp)

Hàm này dùng để xóa các cờ báo lỗi khi thao tác trên các tập tin, vì các cờ này không tự xóa đi mỗi khi kết thúc truy xuất file, trừ khi chúng được xóa bởi clearerr() hoặc rewind().

int fflush(FILE * fp)

Hàm này bỏ những dữ liệu đang có trong vùng đệm lưu trữ dữ liệu xuất nhập để chuẩn bị cho thao tác truy xuất file sau đó. Hàm này sẽ trả về 0, nếu tốt đẹp, trả về EOF nếu có lỗi.

int fgetpos(FILE * fp, fpos_t * pos)

Hàm này dùng để lưu lại vị trí định vị trên tập tin *fp* vào biến ** pos* kiểu *fpos_t* là kiểu được định nghĩa bằng *typedef* trong *stdio.h* (thường là kiểu *long*). Trả về 0, nếu tốt đẹp. Trả về số khác 0, nếu có lỗi, khi đó biến báo lỗi *errno* sẽ có trị là *EBADF* hoặc *EINVAL*.

int fsetpos(FILE * fp, const fpos_t *pos)

Định vị lại vị trí cần truy xuất trên tập tin *fp* đến vị trí có giá trị là **pos*. Giá trị này phải là giá trị mà lần gọi fgetpos () trước đó cung cấp. Hàm này sẽ xóa các cờ báo end-of-file, và hủy tác dụng của hàm ungetc() trên tập tin này. Trả về 0, nếu tốt đẹp. Trả về số khác 0 và đặt biến errno bằng EBADF hoặc EINVAL nếu có lỗi.

int rename(const char * oldname, const char * newname)

Đổi tên tập tin *oldname* thành *newname*. Trả về 0 nếu đổi được, nếu có lỗi hàm sẽ trả về số khác 0 và biến errno có trị:

EACCES không dùng path hoặc tên tập tin

ENOENT không tìm thấy tên *oldname*

EXDEV không thể đổi tập tin thành thiết bị khác

void setbuf(FILE * fp, char * buf)

Đổi vùng đệm xuất nhập của một dòng xuất nhập bằng vùng đệm do lập trình viên quy định. Bằng cách sử dụng setbuf () gởi cho dòng xuất nhập *fp* một pointer *buf* chỉ đến một vùng đệm có kích thước BUFSIZ (định nghĩa trong stdio.h). Vùng này sẽ là vùng đệm khi xuất nhập trên tập tin này. Nếu *buf* == NULL thì dòng xuất nhập này xem như không có vùng đệm.

int setvbuf(FILE * fp, char * buf, int type, size_t size)

Hàm này cho phép chúng ta kiểm soát cả vùng đệm của dòng xuất nhập *fp* lẫn kích thước của nó. *fp* là dòng xuất nhập đang mở, *buf* là pointer chỉ đến vùng đệm, nếu bằng NULL thì một vùng nhớ sẽ tự động được cấp phát theo kích thước yêu cầu với:

size là kích thước yêu cầu của vùng đệm, *size* > 0

type là một trong các trị sau:

_IOFBF đếm đầy vùng đệm

_IOLBF đếm vùng đệm theo hàng

_IONBF không dùng vùng đệm

Trả về 0 nếu tốt, hoặc khác 0 nếu có tham số nào đó không đúng quy định.

char * tmpnam(char * s)

Hàm tmpnam() dùng để lấy một tên tập tin tạm thời nào đó. Tên tập tin này được cất vào *s* và trả về pointer chỉ đến đầu vùng này nếu tốt, hoặc trả về NULL nếu không thể đặt được tên tập tin tạm thời nữa.

FILE * tmpfile(void)

Hàm tmpfile() dùng để mở một tập tin tạm thời và lấy dòng xuất nhập mà hàm này trả về để xuất nhập trên tập tin này. Tập tin này sẽ tự động đóng lại khi hết chương trình. Tập tin được mở theo kiểu cập nhật ("w+b"). Hàm trả về NULL nếu không mở được tập tin.

int ungetc (int c, FILE * fp)

Hàm ungetc() được dùng để đẩy một ký tự *c* trở ngược lại vào vùng đệm để lần đọc tập tin sau đó sẽ bắt đầu bằng *c*. Chúng ta sẽ không thể ungetc (EOF) được vì ungetc() sẽ bỏ qua ký tự này. Ký tự được ungetc() có thể bị hủy nếu sau đó ta gọi đến các hàm fflush, fseek, fsetpos, hay rewind. Hàm trả về ký tự *c* nếu ungetc() được, hoặc trả về EOF nếu không thành công.

Kế là các hàm mà macro cho phép chúng ta viết được các hàm có số đối số thay đổi. Các hàm và macro này được khai báo trong stdarg.h mà stdio.h đã include vào trong nó.

int vfprintf(FILE * fp, const char * format, va_list arglist)***int vprintf(const char * format, va_list arglist)******int vsprintf(char * buffer, const char * format, va_list arglist)***

Các hàm này dùng để ghi vào tập tin, viết ra stdout hoặc ra chuỗi *buffer* các chuỗi định dạng theo *format*. Các hàm này có dạng giống sprintf, printf và sprintf, nhưng chúng lại nhận một pointer chỉ đến danh sách các tham số gửi theo sau kiểu *va_list*. Các hàm này có ích cho các trường hợp cần viết những hàm có số đối số thay đổi. *va_arg*, *va_list*, *va_end* là những macro được định nghĩa trong stdarg.h được dùng để viết các hàm có số đối số thay đổi của C.

va_list arg_ptr

Kiểu pointer chỉ đến các danh sách đối số

void va_start (arg_ptr, prev)

Hàm bắt đầu việc lấy tham số

type va_arg (arg_ptr, type)

Hàm lấy tham số

void va_end (arg_ptr)

Hàm chấm dứt việc lấy tham số

va_start()

sẽ đặt *arg_ptr* chỉ đến đối số thay đổi đầu tiên trong các đối số được gửi đến cho hàm, *prev* là tên của đối số đứng ngay trước đối số đầu tiên trong các đối số hàm

va_arg()

Chỉ được gọi sau khi đã gọi *va_start()*, sẽ lấy một giá trị theo kiểu *type* ra khỏi danh sách, và tăng *arg_ptr* đến vị trí tham số kế tiếp. Hàm này sẽ được gọi nhiều lần để khai thác liên tiếp các tham số tiếp theo.

va_end()

sẽ đưa *arg_ptr* trở về NULL. Hàm này sẽ được gọi sau khi đã lấy hết các tham số cần thiết.

Với các hàm này, chúng ta có thể thiết kế hàm với số đối số không cần viết trước (như hàm printf hay scanf). Đây là một điểm rất đặc biệt đối với hàm của ngôn ngữ C.

Ví dụ 18.6

Ta có thể viết hàm mean() nhận nhiều đối số là số nguyên và lấy trung bình cộng của chúng để trả về. Ta có thể quy định rằng danh sách các số cần lấy trung bình này sẽ phải tận cùng bằng 0, vì ta không biết trước rằng người ta sẽ gởi đến cho hàm bao nhiêu số.

```
#include <stdio.h>
#include <stdarg.h>
```

```

int mean ( int num1,...) → định nghĩa một hàm có đối số thay đổi
{
    int value, total, count = 0;
    va_list list;
    if (num1 == 0)
        return(0);
    va_start (list,num1); ← Khởi động list
    total = num1;
    for (count++; (value = va_arg(list, int)) != 0; count++)
        total += value;
    va_end (list);
    return (total/count);
}

main()
{
    int n;
    clrscr();
    n = mean(5,9,90,14,15,0); /* 6 đối số */
    printf ("Trung bình là %d\n",n);
    n = mean(60,1,14,72,0); /* 5 đối số */
    printf ("Trung bình là %d\n",n);
    getch();
}

```

Chương trình cho xuất liệu:

Trung bình là 26

Trung bình là 36

18.3 CÁC HÀM KHÁC

18.3.1 Các hàm xử lý chuỗi và mảng

Như ta đã biết, C không trực tiếp xử lý các chuỗi ký tự hoặc mảng bằng các phép toán của mình, mà việc xử lý này phải được thực hiện qua các hàm thư viện.

Các khai báo hàm, các kiểu dữ liệu đặc biệt đã được khai báo đầy đủ trong tập tin string.h.

Ngoài các hàm đã biết: strcat(), strcpy(), strcmp(), strlen(), các hàm sau đây được nêu thêm:

char * strchr (const char * s, int c)

Dò một chuỗi theo chiều thuận, tìm ký tự *c* đầu tiên xuất hiện trong chuỗi *s*. Trả về pointer chỉ đến vị trí đầu tiên tìm thấy ký tự này, hoặc trả về NULL nếu không tìm thấy.

size_t strcspn (const char * s1, const char * s2)

Dò một chuỗi theo chiều thuận và đếm ký tự cho đến khi gặp bất kỳ một ký tự nào có trong chuỗi *s2* thì dừng lại. Trả về chiều dài của chuỗi con đã đếm qua của chuỗi *s1*, chuỗi con này chỉ toàn chứa các ký tự không có trong chuỗi *s2*.

char * strerror (int errnum)

Trả về một pointer chỉ đến thông báo lỗi ứng với lỗi thứ *errnum*, để ta có thể in ra sau đó.

char * strncat (char * dest, const char * src, size_t maxlen)

Chép tối đa là *maxlen* ký tự vào cuối chuỗi *dest*, và tự thêm ký tự NULL vào cuối chuỗi. Chiều dài tối đa của chuỗi kết quả sẽ là *strlen (dest) + maxlen*. Trả về pointer chỉ đến chuỗi *dest*.

int * strncmp (const char * s1, const char * s2, size_t maxlen)

So sánh hai chuỗi *s1* và *s2* theo cách so sánh sắp xếp của tự điển, nhưng chỉ so sánh tối đa *maxlen* ký tự đầu tiên mà thôi. Nếu phần đó của chuỗi *s1* đứng trước phần chuỗi tương ứng của *s2*, ta gọi *s1* là nhỏ hơn *s2*. Trả về giá trị

- < 0 nếu chuỗi *s1* nhỏ hơn *s2*
- == 0 nếu chuỗi *s1* giống như *s2*
- > 0 nếu chuỗi *s1* lớn hơn chuỗi *s2*

char * strncpy (char * dest, const char * src, size_t maxlen)

Chép *maxlen* ký tự từ chuỗi *src* vào chuỗi *dest* nếu chuỗi *src* dài hơn hoặc bằng *maxlen*, hoặc ghi thêm ký tự NULL cho đủ *maxlen* vào chuỗi *dest* nếu chuỗi *src* ngắn hơn *maxlen* ký tự. Trả về pointer chỉ đến *dest*.

char * strrchr (const char * s, int c);

Dò chuỗi *s*, để tìm ký tự *c* xuất hiện cuối cùng trong chuỗi *s*. Trả về pointer chỉ đến vị trí tìm được, hoặc trả về NULL nếu không tìm thấy.

size_t strspn (const char * s1, const char * s2)

Dò trong chuỗi *s1* theo chiều thuận và đếm ký tự cho đến khi gặp ký tự nào không có trong chuỗi *s2* thì dừng. Trả về chiều dài của chuỗi con đã đếm của chuỗi *s1*, hoàn toàn chứa các ký tự có trong chuỗi *s2*.

char * strstr (const char * s1, const char * s2)

Dò chuỗi *s1*, tìm chuỗi con giống như *s2* xuất hiện đầu tiên trong chuỗi *s1*. Trả về pointer chỉ đến vị trí tìm thấy, hoặc trả về NULL nếu không tìm thấy.

double strtod (const char * s, char ** endptr);

Đổi chuỗi *s* thành số double, với chuỗi *s* phải có dạng

[ktrắng] [+:-] [số] [.].[số] [e: E [+:-] số]

strtod() sẽ dừng lại khi gặp ký tự không lấy được theo dạng như trên. Trả về giá trị double đã đổi được, hoặc HUGE_VAL nếu tràn. Đồng thời, nếu *endptr* khác NULL, đổi tương mà nó chỉ tới nó sẽ được gán pointer chỉ đến vị trí kế tiếp của chuỗi *s* mà *strtod()* không đổi được nữa. Như vậy, lúc gọi hàm này, nếu ta gởi cho nó địa chỉ của pointer làm đối số thứ hai, thì sau khi gọi xong, pointer đó sẽ chỉ đến ký tự đầu tiên trong chuỗi *s* không đổi được thành số.

char * strtok (char * s1, const char * s2)

strtok() xem chuỗi *s1* như một loạt các chuỗi con, ngăn cách nhau bằng một hoặc nhiều ký tự có trong chuỗi *s2*. Và ta có thể lấy dần từng chuỗi con đó bằng cách gọi *strtok()* nhiều lần. Lần đầu tiên, phải có chuỗi *s1*, và *s2* đầy đủ. Các lần gọi sau, *s1* phải là NULL. Lần gọi *strtok()* đầu tiên, hàm sẽ trả về pointer chỉ đến chuỗi con đầu tiên trong *s1*, và viết ký tự nul vào cuối chuỗi con này (thay cho ký tự ngăn cách). Các lần gọi *strtok()* sau đó (với *s1 ==NULL*) sẽ cho các chuỗi con kế tiếp của chuỗi *s1*, cho đến khi hết các chuỗi con thì trả về NULL.

Ví dụ 18.7

```

char *p = "Sun-May/12/1997";
char n[30], *q;
strcpy ( n, strtok(p,"-/-") ) /* n sẽ được chép chuỗi Sun */;
printf("Chuỗi đầu tiên: %s\n", n); /* in ra chuỗi đầu tiên: Sun */
while ( ( q =strtok (NULL, "-/-") != NULL)
printf("Chuỗi kế tiếp: %s\n", q);

...
/* in ra:
   Chuỗi kế tiếp: May
   Chuỗi kế tiếp: 12
   Chuỗi kế tiếp: 1997
*/

```

long strtol (const char * s, char ** endptr, int radix)

Đổi chuỗi ký tự s thành một số long, với s là chuỗi ký tự có dạng:

[ktrắng] [+:-] [0] [x:X] [so]

strtol() sẽ dừng lại khi gặp một ký tự không thể đổi được nữa và cất pointer chỉ đến vị trí đó vào nội dung của pointer endptr để trả về, nếu như ta có gửi endptr khác NULL cho hàm.

Nếu radix từ 2-36, chuỗi ký tự s được hiểu là dạng viết ở cơ số radix.

Nếu radix là 0, các ký tự đầu tiên của chuỗi s sẽ quyết định cơ số dạng biểu diễn của s.

Ký tự	Ký tự	Chuỗi được hiểu ở dạng
0	1 - 7	bát phân
0	x hay X	thập lục phân
1 - 9		thập phân

Với radix khác, xem như không hợp lệ và xem như không có ký tự nào đổi được. Kết quả là 0. Nếu chuỗi được hiểu là đang ở một cơ số nào đó thì chỉ những ký tự số hoặc chữ biểu diễn cho cơ số đó là đổi được mà thôi. (Ví dụ, một chuỗi dạng cơ số 24 sẽ chỉ được hiểu là gồm có các ký tự 0 đến 9 và từ A đến N là được dùng để biểu diễn, các ký tự chữ khác, ngoài các ký tự này, đều hiểu là không đổi được). Trả về giá trị long đã đổi được.

unsigned long strtoul (const char * s, char ** endptr, in radix)

Hàm này làm việc tương tự như strtol(), nhưng đổi chuỗi s thành một số kiểu unsigned long và trả về giá trị đổi được.

void * memchr (const void * a, int c, size_t n)

Tìm ký tự c trong n byte của mảng a. Trả về pointer chỉ đến vị trí đầu tiên tìm thấy c hoặc trả về NULL nếu không thấy.

int memcmp (const void * a1, const void * a2, size_t n)

So sánh n byte đầu tiên của hai mảng unsigned char a1, và a2. Trả về giá trị:

< 0 nếu a1 nhỏ hơn a2

== 0 nếu a1 bằng với a2

> 0 nếu a1 lớn hơn a2

void * memcpy (void * dest, const void * src, size_t n)

Chép lại n byte từ mảng src vào mảng dest. Không xử lý trường hợp mảng src và dest có phần gối lên nhau. Trả về pointer chỉ đến dest.

void * memmove (void * dest, const void * src, size_t n)

Chép lại n byte từ mảng src vào mảng dest. Xử lý đúng ngay cả khi hai vùng nhớ gối lên nhau. Trả về pointer chỉ đến dest.

void * memset (const void * a, int c, size_t n)

Gán toàn bộ n byte đầu tiên của mảng a bằng giá trị c. Trả về giá trị chỉ đến mảng a.

Ngoài ra, còn nhiều hàm xử lý chuỗi khác không dùng trên chương trình viết theo chuẩn của ANSI C nên ta không nêu ra đây.

18.3.2 Các hàm cấp phát động

Cấp phát động bộ nhớ là một chức năng rất quan trọng đối với các chương trình C. Dùng cấp phát động chúng ta mới có thể xử lý được các kiểu dữ liệu phức tạp như xâu, cây, mảng pointer,...

Các hàm này là calloc(), malloc(), free() mà ta đã biết, còn một hàm nữa là

void * realloc (void * block, size_t size)

Đổi lại vùng nhớ đã cấp, chỉ bởi block, bằng một vùng nhớ có kích thước mới là size. Trả về pointer chỉ đến vùng nhớ mới, có thể trùng hoặc không trùng với vùng nhớ cũ.

18.3.3 Các hàm chuyển đổi

Chúng ta có thể cần thiết một hàm chuyển đổi dạng các chuỗi ký tự thành các dạng số để tính toán, hoặc đổi ký tự. Các hàm chuyển đổi này có prototype nằm trong file stdlib.h hoặc ctype.h.

Các hàm sau đây có prototype trong file stdlib.h:

double atof (const char * s)

đổi chuỗi ký tự s thành một số double. Chuỗi s phải có dạng:

[ktrắng] [+/-] [số] [.] [số] [e: E [+/-] số]

Trả về kết quả đổi được, hoặc 0 nếu không đổi được. Nếu kết quả là quá lớn, atof() sẽ trả về HUGE_VAL.

int atoi (const char * s)

Đổi chuỗi ký tự s thành số int. Chuỗi s phải có dạng:

[ktrắng] [+/-] [số]

Trả về kết quả đổi được, hoặc 0 nếu không đổi được.

int atol (const char * s)

đổi chuỗi ký tự s thành một số long. Chuỗi s phải có dạng:

[ktrắng] [+/-] [số]

Trả về kết quả đổi được, hoặc 0 nếu không đổi được.

Hai hàm sau có chức năng đổi ký tự, được khai báo trong ctype.h:

int tolower (int ch)

đổi ký tự ch thành chữ thường nếu ký tự đó là ký tự in hoặc giữ nguyên nếu không phải, và trả kết quả về.

int toupper (int ch)

Đổi ký tự *ch* thành chữ in nếu ký tự đó là ký tự thường hoặc giữ nguyên nếu không phải, và trả kết quả về.

Ngoài các hàm chuyển kiểu trên, còn có các macro được sử dụng để kiểm tra một ký tự *c* thuộc loại ký tự nào. Các macro này được định nghĩa trong tập tin chuẩn ctype.h.

int isalnum (int c)

Trả về giá trị khác 0 nếu *c* là ký tự chữ 'A' - 'Z' hay 'a' - 'z' hoặc ký tự số '0' - '9'

int isalpha (int c)

Trả về giá trị khác 0 nếu *c* là ký tự chữ 'A' - 'Z' hay 'a' - 'z'

int iscntrl (int c)

Trả về giá trị khác 0 nếu *c* là ký tự DELETE (0x7f) hay ký tự điều khiển (0x00-0x1f)

int isdigit (int c)

Trả về giá trị khác 0 nếu *c* là ký tự số ('0' - '9')

int isgraph (int c)

Trả về giá trị khác 0 nếu *c* là ký tự in được trừ khoảng trắng (0x21 - 0x7e)

int islower (int c)

Trả về giá trị khác 0 nếu *c* là ký tự thường 'a' - 'z'.

int isprint (int c)

Trả về giá trị khác 0 nếu *c* là ký tự in được (0x20 - 0x7E).

int ispunct (int c)

Trả về giá trị khác 0 nếu *c* là ký tự ngắt (0x00 - 0x20)

int isspace (int c)

Trả về giá trị khác 0 nếu c là ký tự trắng ('\n', '\t', '\r', '\v' hay 0x09 - 0x0D hay 0x20).

int isupper (int c)

Trả về giá trị khác 0 nếu c là ký tự hoa 'A' - 'Z'.

18.3.4 Các hàm tính toán

Tiêu chuẩn ANSI C cũng đưa ra một loạt các hàm tính toán được xem là chuẩn của mọi chương trình biên dịch C, và cũng yêu cầu các khai báo hàm, các định nghĩa kiểu đặc biệt, giá trị đặc biệt của chúng đặt trong tập tin chuẩn là math.h

int abs (int x)

Trả về trị tuyệt đối của x

double fabs (double x)

Trả về trị tuyệt đối của số double x

long int fabs (long int x)

Trả về trị tuyệt đối của số long x

long acos (double x)

Trả về arc cosine của giá trị x

$$-1 \leq x \leq 1$$

Kết quả: từ 0 đến π

double asin (double x)

Trả về arc sine của giá trị x

$$-1 \leq x \leq 1$$

Kết quả: từ $-\pi/2$ đến $\pi/2$

double atan (double x)

Trả về arc tangent của giá trị y/x

Kết quả: từ $-\pi/2$ đến $\pi/2$

double atan2 (double y, double x)

Trả về arc tangent của giá trị y/x

Kết quả: từ $-\pi/2$ đến $\pi/2$

double ceil (double x)

Trả về giá trị nguyên nhỏ nhất, lớn hơn hoặc bằng x (nhưng kiểu vẫn là double).

Ví dụ 18.8

`ceil (13.02)` sẽ là 14.00

`ceil (- 7.08)` sẽ là - 7.00

double floor (double x)

Trả về giá trị nguyên lớn nhất nhỏ hơn hoặc bằng x .

Kiểu trả về là double (làm tròn dưới của x)

double cos (double x)

Trả về cosine của x (x tính bằng radian)

double sin (double x)

Trả về sine của x (x tính bằng radian)

double tan (double x)

Trả về tangent của x (x tính bằng radian) với các giá trị x gán $-\pi/2$ hoặc $+\pi/2$, `tan()` sẽ cho giá trị 0, và bật biến báo lỗi `errno = ERANGE` để báo tràn.

double cosh (double x)

Trả về cosine hyperbolic của x . Nếu kết quả quá lớn, trả về `HUGE_VAL`, và bật biến báo lỗi `errno = ERANGE` để báo tràn.

double sinh (double x)

Trả về sine hyperbolic của x

div_t div (int numer, int denom)

Chia *numer* cho *denom* và trả về thương số và dư số trong cấu trúc *div_t* đã khai báo bằng *typedef* trong *math.h*:

```
typedef struct
{
    int quo; /* Thuong so */
    int rem; /* Du so */
} div_t;
```

ldiv_t ldiv (long int numer, long int denom)

Chia số long *numer* cho số long *denom* và trả về thương số và dư số trong cấu trúc *ldiv_t* đã khai báo bằng *typedef* trong *math.h*:

```
typedef struct
{
    long int quot; /* Thuong so */
    long int rem; /* Du so */
} ldiv_t;
```

double exp (double x)

Tính giá trị của e lũy thừa *x* và trả về. Nếu kết quả quá lớn, trả về *HUGE_VAL* và gán biến báo lỗi *errno = ERANGE* để báo tràn.

double fmod (double x, double y)

Tính phần dư của phép chia *x* cho *y*

double frexp (double x, int * exponent)

Hàm này tách số double *x* thành hai phần:

- Phần định trị *m* : $0,5 \leq m < 1$
- Phần lũy thừa *n* sao cho : $x = m * 2^n$

m được trả về làm kết quả của hàm, còn *n* được cất vào biến int chỉ bởi pointer *exponent*.

double ldexp (double x, int exp)

Tính giá trị $x * 2^{\text{exp}}$ và trả về.

double log (double x)

Tính logarithm tự nhiên của số x ($\ln(x)$)

Nếu $x \leq 0$, $\log()$ sẽ trả về `HUGE_VAL` âm, và gán biến báo lỗi `errno = EDOM` để báo sai vùng giá trị.

double log 10 (double x)

Tính logarithm cơ số 10 của số x ($\log_{10}(x)$)

Nếu $x \leq 0$, $\log_{10}()$ sẽ trả về `HUGE_VAL` âm, và gán biến báo lỗi `errno = EDOM` để báo sai vùng giá trị.

double modf (double x, double * ipar)

Tách số x thành hai phần: nguyên và thập phân. Phần thập phân được trả về. Phần nguyên được cất vào biến `double` chỉ bởi `pointer ipar`.

double pow (double x, double y)

Tính giá trị x lũy thừa y và trả về kết quả:

Nếu $x = 0, y = 0$, $\text{pow}()$ sẽ là 1

Nếu tính lũy thừa của số $x \leq 0$ mà y không nguyên, kết quả là `HUGE_VAL` âm, còn `errno = EDOM`

Nếu kết quả quá lớn thì hàm trả về `HUGE_VAL`, còn `errno = ERANGE`

double sqrt (double x)

Tính căn bậc hai dương của số x

Nếu x là số âm thì $\sqrt{x} = 0$, `errno = EDOM`

18.3.5 Các hàm điều khiển liên quan đến hệ điều hành

Chúng ta có một số các hàm chuẩn sau đây cho phép xử lý một số thao tác liên quan đến các thao tác của hệ điều hành (gọi là quá trình) (*process*). Các hàm này được khai báo trong `stdlib.h`

void abort (void)

In thông báo (*Abnormal program termination*) ra `stderr` và thoát khỏi chương trình với mã báo lỗi là 3.

int atexit (atexit_t func)

Hàm này ghi nhận một pointer chỉ đến hàm *func*, hàm chỉ bởi *func* sẽ được gọi đến khi chương trình thoát ra để trở về hệ điều hành. Kiểu dữ liệu *atexit_t* đã được định nghĩa trong stdlib.h bằng *typedef*. Ta có thể ghi nhiều hàm để gọi lúc ra, bằng nhiều lần gọi hàm *atexit()*. Hàm được ghi nhận sau cùng sẽ được gọi trước tiên theo kiểu stack (last-in first-out).

Hàm trả về 0 nếu ghi được, khác 0 nếu đã hết chỗ để ghi nhận hàm *func* vào.

void exit (int status)

Hàm *exit()* cho phép kết thúc chương trình đang thực hiện. Trước khi kết thúc, các tập tin sẽ được đóng lại đầy đủ, vùng đệm được làm sạch, và các hàm được ghi nhận. Giá trị *status* thường được dùng để trả về cho quá trình trước đó (chương trình gọi) để báo tình trạng. Chẳng hạn, 0 là thoát ra bình thường, 1 là thoát ra do có lỗi.

char * getenv (const char * name)

Hệ điều hành thường duy trì một vùng được gọi là môi-trường (*environment*) để lưu chuỗi có dạng

name = string

Ví dụ: như include = \BC31\BIN\INCLUDE Hàm *getenv()* cho phép chúng ta nhận được chuỗi *string* tương ứng với *name* mà chúng ta gửi đến cho hàm. Hàm trả về pointer chỉ đến chuỗi này. Chuỗi có thể là rỗng nếu *name* chưa có trong environment.

void longjmp (jmp_buf jmpb, int retval)

Hàm này cho phép một lệnh nhảy (như *goto*) nhưng có thể đến một chỗ xa trong chương trình và mang theo một giá trị *retval*. Vị trí nhảy đến được ghi nhận bằng hàm *setjmp()* và *jmpb* được dùng để nhận diện vị trí đó. Như vậy, *setjmp()* phải được gọi trước *longjmp()*.

Thường trình gọi *setjmp()* và tạo ra *jmpb* phải vẫn còn làm việc và không được trả về trước khi *longjmp()* được gọi đến. Giá trị *retval* của *longjmp()* phải khác 0, nếu là 0, *longjmp()* sẽ đổi thành 1. Hàm này được dùng để xử lý các lỗi trong chương trình. *jmp_buf* là kiểu dữ liệu được định nghĩa trong stdio.h

int setjmp (jmp_buf jmpb)

Tạo một vị trí để hàm longjmp () có thể nhảy đến sau đó. Hàm sẽ ghi nhận lại toàn bộ tình trạng của công việc cho đến lúc đó và *jmpb* trả về 0.

Hàm longjmp () sau đó nếu có được gọi với cùng *jmpb* sẽ nhảy đến chỗ của lệnh setjmp () này, nhận lại đầy đủ tình trạng đã ghi nhận, giống như lệnh setjmp () đã được gọi nhưng lại trả về *retval*. Hàm này sẽ về 0 khi gọi lần đầu tiên, và trả về *retval* của hàm longjmp () mỗi khi hàm longjmp () được gọi ở đâu sau đó.

Ví dụ 18.9

```
#include <stdio.h>
#include <stdlib.h>
void func (jmp_buf);
main ()
{
    int value;
    jmp_buf jumper;
    value = setjmp(jumper);
    if (value != 0)
    {
        printf("Longjmp with value %d\n", value);
        exit(value);
    }
    printf("About to call subroutine... \n");
    func (jumper);
}
void func(jmp_buf jumper)
{
    longjmp(jumper,1);
}
```

int system (const char * command)

Hàm này cho phép chúng ta gọi lệnh, thực hiện lệnh của hệ điều hành từ trong chương trình C. *command* là chuỗi chứa tên chương trình lệnh của hệ điều hành. Hàm trả về 0 nếu tốt, hoặc -1 nếu không được và các tham số dòng lệnh.

Ví dụ 18.10

`system(dir);`

Riêng hai hàm `raise()` và `signal()` được khai báo trong `signal.h`

int raise (int sign)

Hàm `raise()` sẽ gửi `sign` để yêu cầu thực hiện một việc nào đó. Việc này sẽ được thực hiện theo một quy định trước (*default*), hoặc thực hiện theo một hàm xử lý đã được cài đặt bằng hàm `signal()`. `sign` có thể có các giá trị đã được định nghĩa sau:

SIGABRT	Kết thúc chương trình bất thường.
SIGFPE	Lỗi do tính toán số thực. Kết thúc chương trình (<i>default</i>)
SIGILL	Lệnh không hợp lệ. Kết thúc chương trình (<i>default</i>)
SIGINT	Ngắt do Ctrl-Break. Gọi chức năng xử lý Ctrl-C của hệ điều hành (<i>default</i>). cuu.com
SIGSEGV	Truy xuất vùng nhớ không đúng đắn. Kết thúc chương trình (<i>default</i>).
SIGTERM	Yêu cầu kết thúc chương trình. Bỏ qua yêu cầu đó (<i>default</i>) Hàm trả về 0 nếu bình thường, khác 0 nếu có lỗi.

void (* signal (int sign, void (* func) (int sign)))(int)

Hàm `signal` cho phép xác định cách xử lý khi nhận tín hiệu `sign` sau đó. Chúng ta có thể dùng nó để cài đặt hàm xử lý riêng. Biến `sign` có thể có các giá trị sau: (định nghĩa trong `signal.h`)

SIGABRT	Kết thúc chương trình bất thường bằng cách gọi <code>exit(3)</code> .
SIGFPE	Lỗi do tính toán số thực. Kết thúc chương trình (<i>default</i>)
SIGILL	Lệnh không hợp lệ. Kết thúc chương trình (<i>default</i>)
SIGINT	Ngắt do Ctrl-Break. Gọi chức năng xử lý Ctrl-C của hệ điều hành (<i>default</i>)

SIGSEGV	Truy xuất vùng nhớ không đúng đắn Kết thúc chương trình (<i>default</i>)
SIGTERM	Yêu cầu kết thúc chương trình Bỏ qua yêu cầu đó (<i>default</i>).

func là một địa chỉ của một hàm hoặc là của các hằng đã định sẵn: SIG_DFL và SIG_IGN. Các địa chỉ này sẽ cho biết hàm sẽ dùng để xử lý.

Nếu func là	Sẽ xử lý
SIG_DFL	Kết thúc chương trình, đóng các tập tin, nhưng không xóa buffer
SIG_IGN	BỎ qua tín hiệu này. Không nên dùng cách này với SIGFPE vì không thể tiếp tục tính được trên số thực không xác định này.

Hàm của chương trình:

Hàm sẽ được gọi với đối số là *sign* nhận được, và func sẽ được đặt lại là SIG_DFL để tránh bị gọi lần nữa. Nếu muốn, chúng ta phải gọi lại signal () để cài đặt lại hàm func (). Hàm này có thể chấm dứt bằng lệnh return hay gọi đến abort (), exit () hay longjmp().

18.3.6 Các hàm xử lý ngày giờ

Có các hàm xử lý ngày giờ chuẩn, cho phép thực hiện các thao tác liên quan đến ngày giờ. Các prototype của các hàm này và các định nghĩa cấu trúc liên quan được đặt trong tập tin time.h. Chúng ta sẽ xét hàm dưới đây:

char * asctime (const struct tm * tblock)

Đổi cấu trúc ghi ngày giờ *tblock* thành một chuỗi ký tự có dạng:

Tue Jan 10 02:42:50 1980\n\0

Chuỗi này là một biến static của asctime (). Địa chỉ của nó sẽ được trả về. Muốn sử dụng chuỗi, ta phải chép nội dung của nó vào một mảng riêng, vì vùng này sẽ bị viết chồng lên khi gọi hàm này lần nữa.

clock_t clock (void)

Trả về khoảng thời gian đã trôi qua kể từ lúc chương trình bắt đầu chạy, cho đến lúc đó. Khoảng thời gian tính bằng số chu kỳ dao động của đồng hồ hệ thống.

char * ctime (const time_t * time)

Hàm này sẽ cho phép ta đổi cấu trúc ngày giờ *time_t* do hàm *time ()* tra về thành một chuỗi 26 ký tự có dạng:

```
Tue Jan 10 02:42:50 1980\n\0
```

Hàm trả về pointer chỉ đến chuỗi ký tự này, chuỗi này là mảng static của hàm *ctime ()*.

double difftime (time_t time2, time_t time1)

Hàm này sẽ tính khoảng thời gian cách biệt, theo second (giây) từ giờ *time1* đến giờ *time2*. Trả về kết quả đó ở kiểu double.

struct tm * gmtime (const time_t * timer)

Hàm này sẽ đổi cấu trúc ngày giờ *time_t* thành dạng cấu trúc *tm* tương ứng với ngày giờ tính theo GMT.

```
struct tm
{
    int tm_sec;
    int tm_min;
    int tm_hour; /* 24 giờ */
    int tm_mday; /* 1 đến 31 */
    int tm_mon;
    int tm_year; /* năm - 1900 */
    int tm_wday; /* 0 (Sunday) đến 6 */
    int tm_yday; /* 0 đến 365 */
    int tm_isdst; /* xét đến DST không */
};
```

Trả về pointer chỉ đến cấu trúc *tm*, cấu trúc này là static.

Nếu *tm_isdst > 0* có tính đến Daylight Saving Time (giờ từ 1 đến 12)

≈ 0 không tính

< 0 nếu không biết được

struct tm * localtime (const time_t * timer)

Hàm này sẽ đổi ngày giờ trong cấu trúc time_t (thường do hàm time() trả về) thành dạng biểu diễn theo cấu trúc tm, các giá trị trong tm sẽ được hiệu chỉnh theo giờ địa phương nếu có.

Kết quả trả về là pointer chỉ đến cấu trúc static tm đổi được.

time_t mktime (struct tm * timer)

Hàm mktime() đổi biến ngày giờ thành dạng cấu trúc tm thành thời gian time_t (tính bằng giây) như hàm time() trả về, đồng thời hiệu chỉnh lại các thành phần của struct tm, cho phù hợp nếu trước đó các thành phần của chúng không nằm trong các khoảng tính bình thường. Việc hiệu chỉnh sẽ được tính bằng các thành phần:

tm_mon, tm_year, tm_mday.

time_t time (time_t * timer)

Hàm này cho ta thời gian tính bằng giây, kể từ 00:00:00 GMT. Ngày 1 tháng 1 năm 1970, và cất vào nội dung của pointer timer, nếu pointer này khác NULL. Hàm cũng trả về giá trị này.

18.3.7 Các hàm phụ trợ

Bên cạnh các hàm trên, C còn cung cấp thêm một số hàm phụ trợ cho công việc xử lý của chúng ta như tạo số ngẫu nhiên, tìm kiếm và sắp xếp mảng các phần tử. Các hàm này cũng được khai báo trong stdlib.h.

void srand (unsigned seed)

Hàm khởi động bộ tạo số ngẫu nhiên. Bình thường, hàm này sẽ được gọi với *seed* = 1 bắt đầu để tạo các số ngẫu nhiên. Chúng ta có thể gọi nó với *seed* khác để khởi động lại việc tạo số, làm cho giá trị của các số ngẫu nhiên có vẻ ngẫu nhiên thật sự.

int rand (void)

Hàm phát sinh số ngẫu nhiên, bằng cách sử dụng cách tạo một dãy các số giả ngẫu nhiên, từ 0 đến 32767, và trả về số ngẫu nhiên đó.

void * bsearch (const void * key, const void * base, size_t nelem, size_t width, int (*fcmp) (const void *, const void *))

Hàm này cho phép chúng ta tìm kiếm theo kiểu nhị phân trên một mảng đã sắp xếp.

với:

- key* : pointer chỉ đến phần tử chứa *key* cần tìm
- base* : mảng cần tìm phần tử
- nelem* : số phần tử của mảng.
- width* : kích thước của mỗi phần tử trong mảng
(tính bằng byte)
- fcmp* : pointer chỉ đến hàm so sánh, dùng để so sánh
hai phần tử của mảng.

Hàm này phải được viết để nhận vào hai pointer *pt1* và *pt2* chỉ đến hai phần tử cần so sánh, và trả về kết quả:

< 0	nếu * <i>pt1</i> xếp trước * <i>pt2</i>
== 0	nếu * <i>pt1</i> == * <i>pt2</i>
> 0	nếu * <i>pt1</i> xếp sau * <i>pt2</i>

Hàm *bsearch()* sẽ dùng (** fcmp*) () để so sánh * *key* với một phần tử trong mảng để tìm. Hàm trả về địa chỉ đến phần tử đầu tiên của mảng phù hợp với phần tử chỉ bởi *key*, hoặc trả về NULL nếu không tìm thấy.

void qsort (void * base, size_t nelem, size_t width, int (*fcmp) (const void *, const void *))

Sắp xếp lại một mảng các phần tử, bằng giải thuật quicksort dạng "median of 3" bằng cách sử dụng hàm so sánh của người sử dụng.

với:

- base* : mảng cần xếp
- nelem* : số phần tử mảng.
- width* : kích thước của mỗi phần tử trong mảng (tính bằng byte)
- fcmp* : pointer chỉ đến hàm so sánh dùng so sánh hai phần tử của mảng.

Hàm này được viết để nhận vào hai pointer pt1 và pt2 chỉ đến hai phần tử cần so sánh, và trả về kết quả:

< 0	nếu *pt1 < *pt2
== 0	nếu *pt1 == *pt2
> 0	nếu *pt1 > *pt2

Nên nhớ rằng việc biểu diễn *pt1 > *pt2 có ý nghĩa rằng *pt1 sẽ được xếp trước *pt2.

void assert (int test)

Đây là một macro của một lệnh if, kiểm tra nếu *test* là 0, thì in thông báo ra stderr và thoát khỏi chương trình qua *abort ()*.

Prototype trong assert.h

Thông báo in ra có dạng:

Assertion on failat: <filename>,<line>

Hàm này nhằm mục đích giúp chúng ta có cách để tìm lỗi trong chương trình.

Chúng ta có thể hủy tác dụng của assert () (Coi như assert () không hề có) bằng cách đặt lệnh:

```
# define NDEBUG
```

vào đầu chương trình, trước các lệnh #include.

18.4 CÁC HÀM KHÔNG CHUẨN CỦA CÁC CHƯƠNG TRÌNH BIÊN DỊCH

Hiện nay do khuynh hướng sử dụng chuyên biệt, và nhu cầu về các hàm xử lý giao tiếp với người sử dụng ngày càng tăng, mỗi chương trình biên dịch C đều cung cấp thêm nhiều hàm khác, các hàm này mặc dù không có trong thư viện chuẩn, vẫn đang được các lập trình viên sử dụng phổ biến.

Độc giả có thể tìm thấy hướng dẫn của các hàm này trong các sách đi kèm theo các chương trình biên dịch của các nhà sản xuất. Khi sử dụng các hàm này, cần biết rõ rằng chúng không phải là các hàm chuẩn và do đó sẽ làm cho chương trình của chúng ta không dễ mang sang một chương trình biên dịch khác hoặc một hệ điều hành khác.

Chương 19

ĐỆ QUY

Trong chương “Hàm” chúng ta đã làm quen khái niệm đệ quy, tuy nhiên việc sử dụng đệ quy vào lập trình là một vấn đề khá khó khăn và phức tạp, đặc biệt là khi việc sử dụng đệ quy với các khái niệm con trỏ và kiểu dữ liệu tự định nghĩa struct. Chương này trình bày một số ứng dụng đệ quy để giải quyết các bài toán cụ thể như tháp Hà Nội, tìm kiếm nhị phân, cùng một số ví dụ khác. Ngoài ra cấu trúc dữ liệu cây cũng được nêu ra nhằm bổ sung khối kiến thức về cấu trúc dữ liệu cho môn học này.

19.1 ĐỆ QUY LÀ GÌ?

Như chúng ta đã biết, một hàm gọi chính nó là một hàm đệ quy, như hàm **RunningSum** trong ví dụ 19.1 sau.

Ví dụ 19.1 Tính tổng $\sum_1^n i$

```
int RunningSum(int n)
{
    if (n == 1)
        return 1;
    else
        return n + RunningSum(n-1);
}
```

Hàm này tính tổng tất cả các số nguyên từ 1 tới thông số nhập n . Ví dụ, **RunningSum(4)** sẽ tính $4+3+2+1$, tuy nhiên nó làm việc này một cách đệ quy. Có thể dễ dàng thấy rằng, tổng của 4 chính là 4 cộng với tổng của 3, tổng của 3 chính là 3 cộng với tổng của 2, và cứ thế. Định nghĩa đệ quy này chính là cơ sở cho giải thuật đệ quy tổng của n .

nhiều sau:

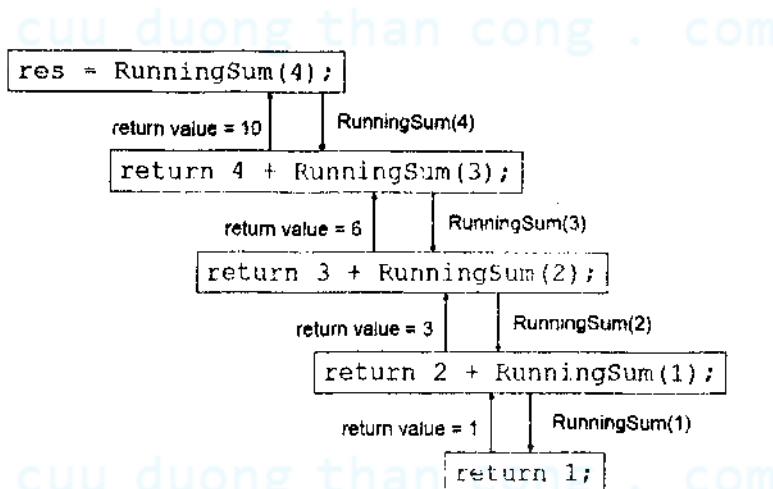
$$\text{RunningSum}(n) = n + \text{RunningSum}(n-1)$$

Về toán, chúng ta dùng các phương trình đệ quy để biểu diễn các hàm như vậy. Phương trình trên là một phương trình đệ quy của hàm **RunningSum**. Để hoàn chỉnh việc định trị của phương trình, chúng ta phải quy định trường hợp gốc, tức chúng ta cần có

$$\text{RunningSum}(1) = 1$$

trước khi tính trị, cụ thể để tính **RunningSum(4)** chúng ta có quá trình sau:

$$\begin{aligned}\text{RunningSum}(4) &= 4 + \text{RunningSum}(3) \\ &= 4 + 3 + \text{RunningSum}(2) \\ &= 4 + 3 + 2 + \text{RunningSum}(1) \\ &= 4 + 3 + 2 + 1\end{aligned}$$



Hình 19.1 Quá trình gọi hàm **RunningSum(4)**

Hàm **RunningSum** ở ví dụ 19.1 sẽ làm việc theo cùng cách với phương trình trên. Trong quá trình thực thi, việc gọi hàm **RunningSum(4)**, hàm **RunningSum** gọi hàm chính nó với đối số là 3 (nghĩa là **RunningSum(3)**). Tuy nhiên, trước khi hàm **RunningSum(3)** chấm dứt, nó gọi hàm **RunningSum(2)**. Và trước khi chấm dứt hàm **RunningSum(2)** làm cuộc gọi tới **RunningSum(1)**, tới

dây sẽ không có thêm các cuộc gọi nào nữa và hàm **RunningSum(1)** trả về trị 1 cho hàm **RunningSum(2)**, làm cho **RunningSum(2)** kết thúc, trả trị $2+1$ ngược về cho **RunningSum(3)**. Điều này lại làm **RunningSum(3)** kết thúc và trả trị $3+2+1$ cho **RunningSum(4)**. Và sau cùng ta có kết quả tổng từ 1 tới 4 qua việc gọi hàm **RunningSum(4)**. Hình 19.1 minh họa quá trình trên một cách trực quan.

19.2 ĐỆ QUY VÀ LẶP

Rõ ràng chúng ta đã có thể viết hàm **RunningSum** bằng vòng lặp **for**, và mã chương trình sẽ dễ dàng hơn là bản đệ quy của nó. Ở đây chúng tôi đã cung cấp bản đệ quy để minh họa việc sử dụng đệ quy khi viết chương trình.

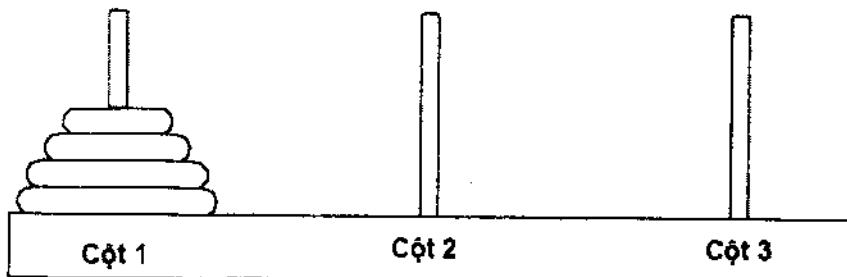
Có một sự song song giữa việc sử dụng đệ quy và vòng lặp (như **for** và **while**) trong lập trình. Tất cả các hàm đệ quy đều có thể được viết bằng vòng lặp. Tuy nhiên, trong một số bài toán lập trình việc sử dụng đệ quy sẽ dễ dàng và trong sáng hơn khi dùng vòng lặp. Các cách giải với các bài toán như vậy thông thường được biểu diễn ở dạng đệ quy. Chính vì lẽ đó mà đệ quy là một kỹ thuật lập trình không thể thiếu được. Một lập trình viên nhiều kinh nghiệm sẽ biết vấn đề nào yêu cầu đệ quy, khi nào cách giải đệ quy tốt hơn lặp, đó là một nghệ thuật của lập trình máy tính.

Chúng ta cũng phải trả giá cho sự hữu dụng của đệ quy. Như một thử nghiệm, các bạn hãy viết hàm **RunningSum** bằng vòng lặp, sau đó so sánh thời gian chạy với bản dùng đệ quy của nó với nhiều trị n khác nhau. Để làm điều này các bạn có thể dùng các hàm thư viện để tính thời gian như trong chương “Lớp lưu trữ của biến và sự chuyển kiểu” có đề cập. Chúng ta có thể thấy bản đệ quy tương đối chậm (miễn là bộ dịch không tối ưu hóa cho đệ quy) vì các hàm đệ quy chịu sự gọi hàm còn vòng lặp thì không.

19.3 THÁP HÀ NỘI

Một trong các bài toán minh họa cho thấy sự thuận lợi khi sử dụng thuật giải đệ quy là bài toán đố Tháp Hà Nội. Về hình thức, bài toán đố này gồm một nền có ba cột. Một trong ba cột có các đĩa gỗ sắp theo thứ tự đĩa nhỏ ở trên đĩa lớn ở dưới. Vấn đề của chúng ta là phải chuyển tất cả các đĩa từ cột hiện thời qua một trong hai cột kia theo

hai luật sau: mỗi lần chỉ được di chuyển một đĩa, và đĩa lớn không được đặt trên đĩa nhỏ trong suốt quá trình chuyển chỗ. Ví dụ hình 19.2 là một bài toán có bốn đĩa trên cột 1. Để giải nó, bốn đĩa này phải được chuyển qua một trong hai cột còn lại theo hai luật trên.



Hình 19.2 Bài toán Tháp Hà Nội

Cũng xin nói thêm là bài toán đố với luật chơi này xuất phát từ một truyền thuyết từ thời xưa, rằng khi thế giới vừa mới được tạo ra, thượng đế đã nói với các nhà sư là họ có nhiệm vụ chuyển 64 đĩa từ cột này sang cột kia theo luật mà chúng ta đã biết. Khi nào họ hoàn thành nhiệm vụ, thì lúc đó cũng là thời điểm tận thế. Và theo bạn thì sao? Với giải thuật được nêu ở đây và máy tính hiện nay, thời điểm tận thế của nhân loại là lúc nào? Các bạn hãy thử vậy.

Bây giờ nhiệm vụ của chúng ta là viết chương trình máy tính để giải bài toán này. Lấy ví dụ như hình 19.2 chúng ta có số đĩa $n = 4$, cột đầu tiên bên trái đang có các đĩa can chuyển chỗ, cột 2 là cột trung gian trong quá trình chuyển, và cột 3 bên phải sẽ là cột đích. Chúng ta có thể dễ dàng rút ra nhận xét: loạt chuyển chỗ sau cùng bao gồm việc chuyển đĩa lớn nhất từ cột 1 sang cột đích 3, và rồi chuyển các đĩa kia để lên đĩa lớn nhất này. Như vậy, vấn đề là chúng ta cần phải chuyển tất cả $n-1$ đĩa trên đĩa lớn nhất ra khỏi nó theo luật quy định qua cột trung gian (cột 2), và rồi chuyển đĩa lớn nhất từ cột của nó qua cột đích. Sau cùng, chúng ta chuyển tất cả $n-1$ đĩa từ cột trung gian sang cột đích. Và như vậy chúng ta đã xong! Tuy nhiên, có thể có độc giả bắt lỗi là việc di chuyển $n-1$ đĩa một lúc như vậy là không hợp lệ. Đúng vậy, nhưng thật ra việc di chuyển này chỉ là cách nói quy ước cho một thao tác gồm nhiều thao tác nhỏ mà tại một thời điểm cũng chỉ có một đĩa di chuyển mà thôi. Cụ thể, chúng ta đã bắt đầu bài toán theo cách mà chúng ta có thể giải nó nếu

chúng ta có thể giải hai vấn đề nhỏ hơn của nó. Một khi đĩa lớn nhất đã ở cột đích, chúng ta không cần quan tâm xử lý nó nữa. Bây giờ đĩa lớn thứ $n-1$ trở thành đĩa lớn nhất, và mục tiêu lúc này là di chuyển nó qua cột đích. Do đó chúng ta có thể áp dụng kỹ thuật tương tự nhưng ở cấp độ nhỏ hơn một đĩa. Và cứ như vậy cho các cấp độ nhỏ hơn, ít đĩa hơn. Đây chính là ý tưởng của đệ quy.

Như vậy, bây giờ chúng ta đã có một định nghĩa đệ quy của vấn đề: thao tác di chuyển n đĩa qua cột đích được chúng ta biểu diễn danh định là **Move(n, target)** mà đầu tiên sẽ chuyển $n-1$ đĩa qua cột trung gian- tức **Move(n-1, intermediate)**- rồi chuyển đĩa thứ n sang đích, và cuối cùng chuyển $n-1$ đĩa kia từ cột trung gian sang cột đích, đó chính là thao tác **Move(n-1, target)**. Như vậy thao tác **Move(n, target)** thực hiện hai cuộc gọi đệ quy tới chính nó để giải quyết hai vấn đề nhỏ hơn cho $n-1$ đĩa.

Như với các phương trình truy hồi trong toán, tất cả định nghĩa đệ quy đều cần phải có trường hợp gốc để kết thúc đệ quy. Theo cách mà chúng ta đã phân tích ở trên, trường hợp gốc sẽ là thao tác di chuyển đĩa nhỏ nhất (đĩa 1). Việc chuyển đĩa 1 không yêu cầu việc chuyển các đĩa khác vì nó luôn luôn ở trên đỉnh và có thể được di chuyển một cách trực tiếp từ cột này qua cột khác mà không cần phải thực hiện bất kỳ sự chuyển đĩa nào khác. Nếu không quy định trường hợp gốc, một hàm đệ quy có thể gặp đệ quy vô tận, tương tự như vòng lặp vô tận với các vòng lặp thông thường.

Việc dùng đệ quy cho mã C là khá dễ dàng. Hàm ví dụ sau đây là hàm đệ quy trong giải thuật của bài toán Tháp Hà Nội.

Ví dụ 19.2 Hàm đệ quy để giải bài toán Tháp Hà Nội

```

/* Dữ liệu nhập:
 - diskNumber là số hiệu của đĩa cần chuyển chỗ (đĩa 1 là đĩa nhỏ nhất)
 - startPost là cột mà hiện thời đĩa đang ở trên đó
 - endPost là cột mà chúng ta muốn đĩa chuyển đĩa tới
 - midPost là cột trung gian
 */
MoveDisk(diskNumber, startPost, endPost, midPost)
{
    if (diskNumber > 1)

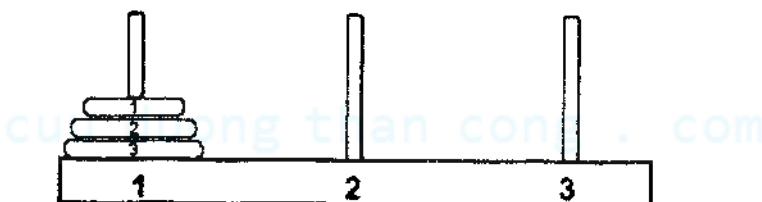
```

```

{
    /* Chuyển n-1 đĩa trên cùng qua cột trung gian */
    MoveDisk(diskNumber-1, startPost, midPost, endPost);
    printf("Move disk number %d from post %d to post %d.\n", diskNumber, startPost,
    endPost);
    /* Chuyển n-1 đĩa từ cột trung gian sang cột cuối */
    MoveDisk(diskNumber-1, midPost, endPost, startPost);
}
else
    printf("Move disk number 1 from post %d to post %d.\n", startPost, endPost);
}

```

Hãy quan sát xem cái gì xảy ra khi chúng ta có bài toán Tháp Hà Nội ba đĩa như trong hình 19.3.



Hình 19.3 Bài toán Tháp Hà Nội, trạng thái ban đầu

Đầu tiên, chúng ta gọi hàm MoveDisk để chuyển đĩa 3 (đĩa lớn nhất) từ cột 1 qua cột 3, dùng cột 2 là trung gian chuyển.

```

/* diskNumber 3; startPost 1; endPost 3; midPost 2 */
MoveDisk(3, 1, 3, 2)

```

Trong quá trình thực hiện, hàm này sẽ gọi tới chính nó với tham số khác để chuyển đĩa 1 và 2 khỏi đĩa 3 và đặt vào cột 2 với cột 3 là trung gian chuyển. Việc gọi này sẽ có dạng như sau:

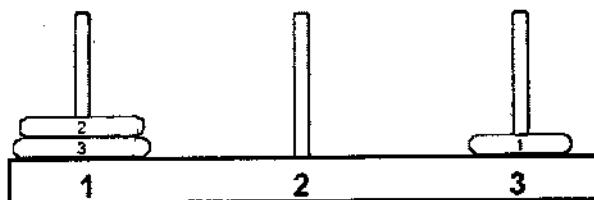
```

/* diskNumber 2; startPost 1; endPost 2; midPost 3 */
MoveDisk(2, 1, 2, 3)

```

Để chuyển đĩa 2 từ cột 1 tới cột 2, đầu tiên chúng ta phải chuyển đĩa 1 ra khỏi đĩa 2 và đặt vào cột trung gian 3 (hình 19.4), điều này tương ứng với việc gọi hàm MoveDisk

```
/* diskNumber 1; startPost 1; endPost 3; midPost 2 */
MoveDisk(1, 1, 3, 2)
```



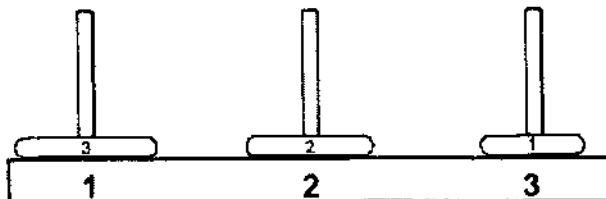
Hình 19.4 Tháp Hà Nội sau lần chuyển đầu tiên

Sau khi đĩa 1 được chuyển xong lệnh printf thứ hai được thực hiện và in ra

Move disk number 1 from post 1 to post 3

Vì đĩa 1 được chuyển xong, tức sau khi hàm Movedisk được gọi, điều khiển sẽ được trả về cho nơi gọi nó, tức hàm MoveDisk (2, 1, 2, 3). Cũng xin nhắc lại là chúng ta đang chờ tất cả các đĩa trên đĩa 2 đã được chuyển sang cột 3. Nên khi đã hoàn thành việc này, bây giờ chúng ta có thể chuyển đĩa 2 từ cột 1 sang cột 2. Hàm printf là phát biểu kế, báo hiệu đĩa khác đã được chuyển (hình 19.5).

Move disk number 2 from post 1 to post 2



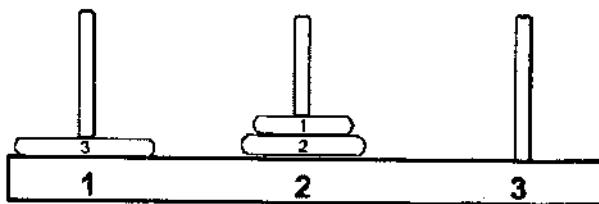
Hình 19.5 Tháp Hà Nội sau lần chuyển thứ hai

Kế đó, hàm MoveDisk lại được gọi để chuyển tất cả đĩa đã ở trên đĩa 2 trước đây (đang ở cột 3) đặt lên trên nó (để giải phóng cột 3)

```
/* diskNumber 1; startPost 3; endPost 2; midPost 1 */
MoveDisk(1, 3, 2, 1)
```

Vì đĩa 1 không có đĩa nào trên nó, nên sau khi thực hiện xong hàm trên ta có trạng thái hình 19.6 và nhận được thông báo sau

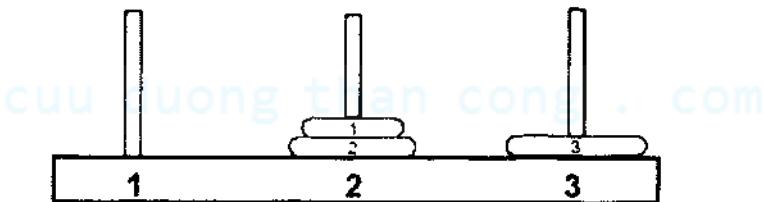
Move disk number 1 from post 3 to post 2



Hình 19.6 Tháp Hà Nội sau lần chuyển thứ ba

Lúc này điều khiển được chuyển ngược về nơi gọi hàm MoveDisk(2, 1, 2, 3), mà vừa hoàn thành việc chuyển đĩa 2 (và tất cả các đĩa trên nó) từ cột 1 qua cột 2. Nơi gọi hàm này chính là hàm MoveDisk(3, 1, 3, 2). Bây giờ tất cả các đĩa đã được chuyển khỏi đĩa 3 và đang được đặt ở cột 2, nên đĩa 3 có thể được chuyển một cách trực tiếp từ cột 1 sang cột 3 (hình 19.7). Hàm printf sẽ báo điều này

Move disk number 3 from post 1 to post 3



Hình 19.7 Tháp Hà Nội sau lần chuyển thứ tư

Nhiệm vụ kế tiếp là chuyển đĩa 2 (và tất cả các đĩa trên nó) từ cột 2 qua cột 3. Chúng ta có thể sử dụng cột 1 làm trung gian chuyển. Việc gọi hàm để làm việc này được thể hiện ở dòng lệnh

```
MoveDisk(diskNumber-1, midPost, endPost, startPost);
```

trong chương trình nguồn ở ví dụ 19.2, mà cụ thể là

```
/* diskNumber 2; startPost 2; endPost 3; midPost 1 */
MoveDisk(2, 2, 3, 1)
```

Để làm điều này, đầu tiên chúng ta phải chuyển đĩa 1 từ cột 2 qua cột 1, việc này tương ứng với việc gọi mã

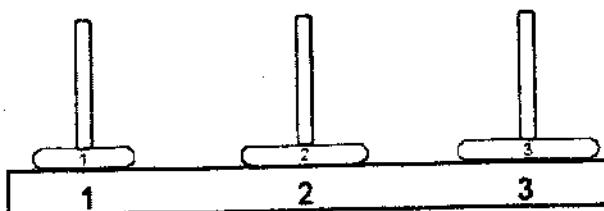
```
MoveDisk(diskNumber-1, startPost, midPost, endPost);
```

và cụ thể

```
/* diskNumber 1; startPost 2; endPost 1; midPost 3 */
MoveDisk(1, 2, 1, 3)
```

Hàm này được thực hiện ngay mà không cần phải có bất kỳ sự chuyển đĩa nào khác. Xem hình 19.8. ta cũng được thông báo

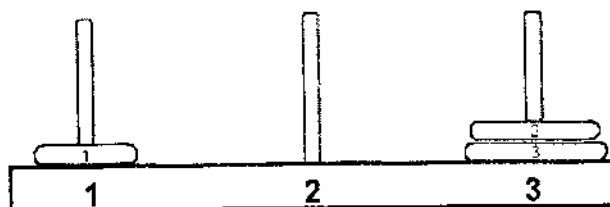
Move disk number 1 from post 2 to post 1



Hình 19.8 Tháp Hà Nội sau lần chuyển thứ năm

Sau đó điều khiển sẽ được trả về cho nơi gọi MoveDisk(2, 2, 3, 1) để đĩa 2 được chuyển sang cột 3 (hình 19.9) với thông báo

Move disk number 2 from post 2 to post 3



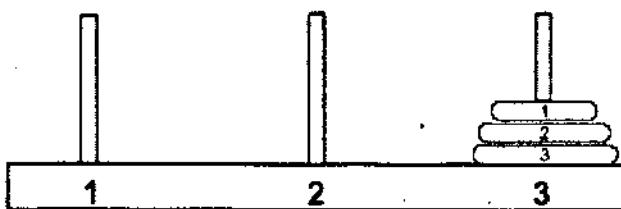
Hình 19.9 Tháp Hà Nội sau lần chuyển thứ sáu

Việc duy nhất còn lại là chúng ta phải chuyển tất cả đĩa đã từng ở trên đĩa 2 để ngược lên nó, tức

```
/* diskNumber 1; startPost 1; endPost 3; midPost 2 */
MoveDisk(1, 1, 3, 2)
```

Việc chuyển này được làm xong ngay lập tức, vì chúng ta chỉ còn đĩa 1 đang ở trên cột 1 mà thôi. Hình 19.10 cho thấy trạng hoàn thành của bài toán. Chúng ta cũng nhận được thông báo

Move disk number 1 from post 1 to post 3



Hình 19.10 Tháp Hà Nội đã được chuyển xong

Để dễ hình dung chúng ta hãy tóm tắt lại các thao tác đệ quy bằng việc xem lại chuỗi gọi hàm để giải bài toán Tháp Hà Nội 3 đĩa này

```

MoveDisk(3, 1, 3, 2) /* Gọi khởi động */
MoveDisk(2, 1, 2, 3)
MoveDisk(1, 1, 3, 2)
MoveDisk(1, 2, 3, 1)
MoveDisk(2, 2, 3, 1)
MoveDisk(1, 2, 1, 3)
MoveDisk(1, 1, 3, 2)

```

Việc hoàn chỉnh chương trình nguồn của bài toán này rất dễ dàng. Hơn nữa, nếu có điều kiện, xin mời độc giả viết lại chương trình giải bài toán Tháp Hà Nội này nhưng dùng dạng vòng lặp để có thể so sánh với bản dùng đệ quy ở trên. Và rồi hãy xem khi nào các vị sư có thể sắp xếp xong 64 đĩa, tức thời điểm tận thế đã điểm?

19.4 DÃY SỐ FIBONACCI

Các phương trình toán truy hồi sau tạo ra một chuỗi số được gọi là số Fibonacci mà nó có các tính chất toán, hình học và tự nhiên khá hấp dẫn.

$$f(n) = f(n - 1) + f(n - 2)$$

$$f(1) = 1$$

$$f(0) = 1$$

Nói cách khác, số Fibonacci thứ n là tổng của hai số trước đó. Chuỗi sẽ là 1, 1, 2, 3, 5, 8, 13, Chuỗi này được nhà toán học Ý Leonardo of Pisa tìm ra vào khoảng năm 1200, cha ông tên là Bonacci, nên ông thường gọi mình là Fibonacci, một cách gọi ngắn gọn của filius Bonacci, có nghĩa là con của Bonacci. Fibonacci đã tạo

ra chuỗi này như là một cách để khảo sát sự gia tăng nòi giống của thỏ, và từ đây chúng ta đã khám phá ra một số hiện tượng tự nhiên có cách thức phân bố theo mẫu của chuỗi này như cấu trúc xoắn tròn óc của vỏ sò hay mẫu hình dạng của cánh hoa.

Chúng ta có thể tạo ra một hàm đệ quy để tính số Fibonacci thứ n một cách trực tiếp từ các phương trình truy hồi trên. **Fibonacci(n)** được tính theo kiểu đệ quy từ **Fibonacci(n-1) + Fibonacci(n-2)**. Trường hợp gốc của đệ quy ở đây đơn giản là cả hai **Fibonacci(1)** và **Fibonacci(0)** đều bằng 1. Chương trình ví dụ 19.3 sau đây trình bày cách giải đệ quy để tính số Fibonacci thứ n .

Ví dụ 19.3 Chương trình đệ quy tính số Fibonacci thứ n .

```
#include <stdio.h>
int Fibonacci(int n);
int main()
{
    int in;
    int number;

    printf ("Which Fibonacci number? ");
    scanf ("%d", &in);

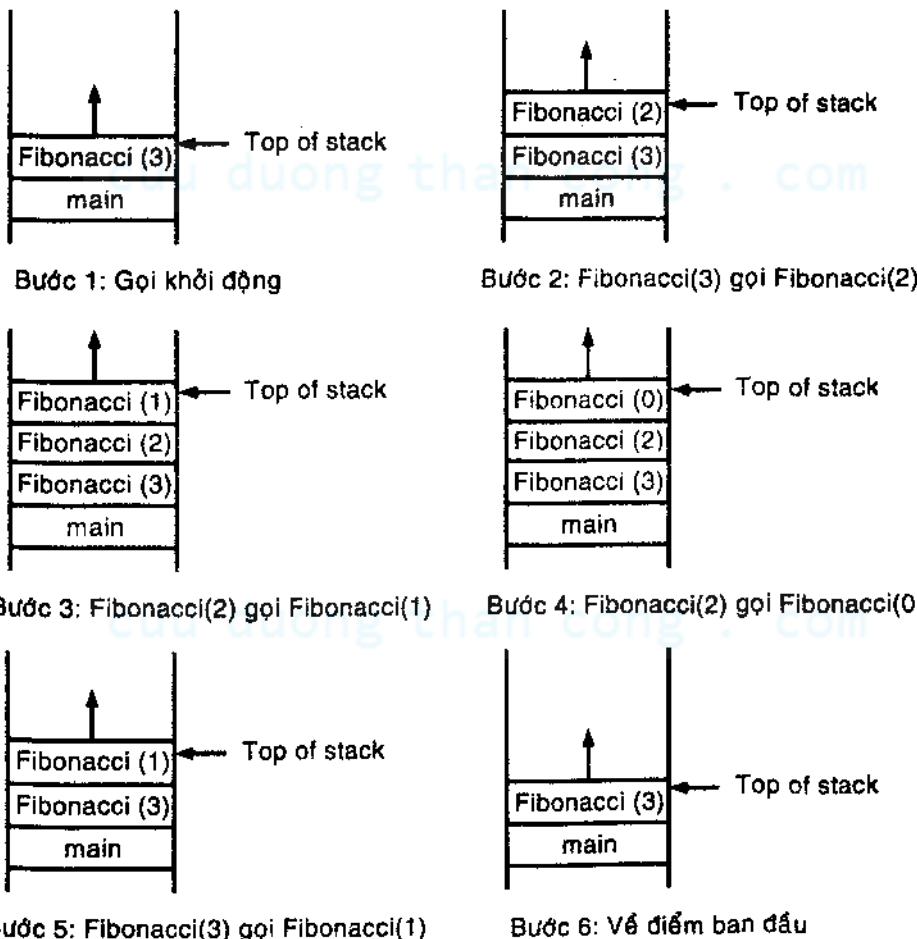
    number = Fibonacci(in);
    printf ("That Fibonacci number is %d\n", number);
}

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

Trong chương trình trên, hàm **Fibonacci** được thiết kế đệ quy. Chúng ta sẽ sử dụng ví dụ này để khảo sát xem đệ quy làm việc như thế nào theo dạng phối cảnh của các cấp thấp trong hệ thống máy tính. Đặc biệt, chúng ta sẽ xem xét cơ chế hoạt động của stack lúc chạy chương trình và cách thức mà stack đối phó với việc gọi đệ quy.

Bất cứ khi nào hàm được gọi, hoặc từ trong chính nó hay từ hàm khác, một bản sao mẫu tin kích hoạt mới của nó được lưu vào stack. Nghĩa là mỗi tham khảo hàm (gọi hàm) đều có một bản sao mới, riêng biệt với các tham số và biến cục bộ cho nó, bản sao này hoàn toàn phân biệt bản sao kia. Đây là một sự bắt buộc để đệ quy làm việc, và stack phải thực hiện. Nếu các biến của hàm được định vị tĩnh trong bộ nhớ, việc gọi đệ quy tới hàm **Fibonacci** có thể làm mất giá trị của các biến tĩnh của lần gọi trước.

Hãy xem cái gì xảy ra khi chúng ta gọi hàm **Fibonacci** với tham số là 3, tức **Fibonacci(3)**. Chúng ta hãy bắt đầu với mẫu tin kích hoạt cho **Fibonacci(3)** trên đỉnh stack. Hình 19.11 cho thấy tiến trình hoạt động của stack khi lần gọi hàm đầu tiên diễn ra.



Hình 19.11 Quá trình hoạt động của stack khi gọi hàm **Fibonacci(3)**

Lần gọi hàm **Fibonacci(3)** sẽ tính đầu tiên **Fibonacci(3-1)**, khi biểu thức **Fibonacci(n-1) + Fibonacci(n-2)** được định trị từ trái sang phải. Do đó, cuộc gọi đầu tiên được thực hiện là **Fibonacci(2)**, và một mẩu tin kích hoạt cho nó được đẩy vào stack (hình 19.11, bước 2).

Đối với hàm **Fibonacci(2)**, tham số n bằng 2 và chưa phải là điều kiện kết thúc đệ quy, và vì vậy cuộc gọi tiếp theo được thực hiện: **Fibonacci(1)** (hình 19.11, bước 3). Cuộc gọi này sẽ được thực hiện trong lúc định trị **Fibonacci(2-1) + Fibonacci(2-2)**.

Lần gọi **Fibonacci(1)** không còn gây ra lần gọi đệ quy nào nữa vì tham số $n=1$ là điều kiện kết thúc đệ quy. Trị 1 được trả về cho **Fibonacci(2)** để đến bây giờ có thể hoàn thành việc tính **Fibonacci(1) + Fibonacci(0)** bằng việc gọi **Fibonacci(0)** (xem hình 19.11, bước 4). Việc gọi hàm **Fibonacci(0)** ngay lập tức trả về trị 1.

Bây giờ, việc gọi hàm **Fibonacci(2)** có thể được hoàn thành và trả về trị cho 2 cho nơi gọi nó, tức **Fibonacci(3)**. Sau khi hoàn thành việc tính thành phần bên trái của biểu thức **Fibonacci(2) + Fibonacci(1)**, **Fibonacci(3)** gọi **Fibonacci(1)** (xem hình 18.11, bước 5) mà ngay lập tức trả về trị 1. Khi **Fibonacci(3)** được thực thi xong, nó cho kết quả là 3 (xem hình 19.11, bước 6).

Như vậy chúng ta có thể xem quá trình đệ quy khi gọi hàm **Fibonacci(3)** một cách đại số như sau:

$$\begin{aligned}\text{Fibonacci}(3) &= \text{Fibonacci}(2) + \text{Fibonacci}(1) \\ &= \text{Fibonacci}(1) + \text{Fibonacci}(0) + \text{Fibonacci}(1) \\ &= 1 + 1 + 1 = 3\end{aligned}$$

Chuỗi các hàm được gọi trong quá trình tính **Fibonacci(3)** như sau:

Fibonacci(3)
Fibonacci(2)
Fibonacci(1)
Fibonacci(0)
Fibonacci(1)

Tương tự, với việc thực thi **Fibonacci(4)** chúng ta sẽ thấy rằng chuỗi các cuộc gọi được tiến hành bởi **Fibonacci(3)** là một thành phần con của các cuộc gọi được **Fibonacci(4)** thực hiện vì

Fibonacci(4) = Fibonacci(3) + Fibonacci(2). Cũng như vậy, chuỗi các cuộc gọi được tiến hành bởi **Fibonacci(4)** là một thành phần con của các cuộc gọi được **Fibonacci(5)** thực hiện. Độc giả có thể tính xem có bao nhiêu cuộc gọi hàm khi ta tính **Fibonacci(n)**, đây có thể được xem là một bài tập.

19.5 TÌM KIẾM NHỊ PHÂN

Có nhiều giải thuật tìm kiếm phần tử trong một danh sách, nếu danh sách là ngẫu nhiên, giải thuật tìm kiếm tất yếu sẽ là tuần tự. Tuy nhiên, nếu danh sách với khóa tham khảo đã được sắp xếp theo một thứ tự từ nhỏ đến lớn (hay ngược lại), kỹ thuật tìm kiếm nhị phân (*binary search*) là cách rất nhanh chóng để tìm ra phần tử trong danh sách đó. Lúc này, chúng ta có thể sử dụng khái niệm hàm đệ quy trong C cho kỹ thuật tìm kiếm nhị phân.

Giả sử chúng ta muốn tìm một số nguyên trong một mảng số nguyên đã được sắp theo thứ tự tăng dần. Hàm cần thiết sẽ trả về chỉ số (vị trí) của phần tử, hoặc trị -1 nếu không có trị đó trong mảng này. Để làm điều này, chúng ta dùng kỹ thuật tìm kiếm nhị phân như sau: với một mảng đã cho, và một số nguyên cần tìm, chúng ta sẽ khảo sát điểm giữa của mảng và xác định xem số nguyên (1) bằng giá trị điểm giữa, (2) nhỏ hơn giá trị điểm giữa, hoặc (3) lớn hơn giá trị điểm giữa hay không. Nếu nó bằng, chúng ta đã xong việc. Nếu nó nhỏ hơn, chúng ta lại tiến hành tìm kiếm nhưng lần này chỉ với nửa đầu của mảng. Nếu nó lớn hơn, chúng ta tiến hành tìm kiếm tiếp chỉ với nửa sau của mảng. Chú ý rằng chúng ta có thể diễn tả các trường hợp (2) và (3) qua các cuộc gọi đệ quy. Nhưng điều gì sẽ xảy ra khi giá trị chúng ta đang tìm không tồn tại trong mảng? Có thể thấy với kỹ thuật đệ quy, việc tìm kiếm phần tử không tồn tại này sẽ được tiến hành trên các mảng ngày càng nhỏ, là các mảng con của mảng ban đầu, cuối cùng chúng ta thực hiện việc tìm kiếm trên mảng không có phần tử (nghĩa là, kích thước 0). Nếu gặp trường hợp này, chúng ta sẽ trả về trị -1. Đây chính là trường hợp gốc hay điều kiện kết thúc đệ quy.

Hàm trong ví dụ 19.4 dưới đây trình bày giải thuật tìm kiếm nhị phân trong C. Chú ý rằng để xác định kích thước của mảng ở mỗi bước, chúng ta phải truyền điểm bắt đầu và điểm kết thúc của mảng con cho mỗi cuộc gọi tới hàm Binarysearch. Mỗi cuộc gọi sẽ hiệu chỉnh các giá

trị start và end để quá trình tìm kiếm sẽ được tiến hành trên các mảng con ngày càng nhỏ dần của mảng gốc list ban đầu.

Ví dụ 19.4

```

/* Hàm này trả về vị trí của 'item' nếu nó tồn tại giữa list[start] và list[end], hay -1
nếu không tồn tại */

int BinarySearch(int item, int list[], int start, int end)
{
    *
    int middle = (end + start) / 2;

    /* Chúng ta đã không tìm thấy cái cần tìm? */
    if (end < start)
        return -1;

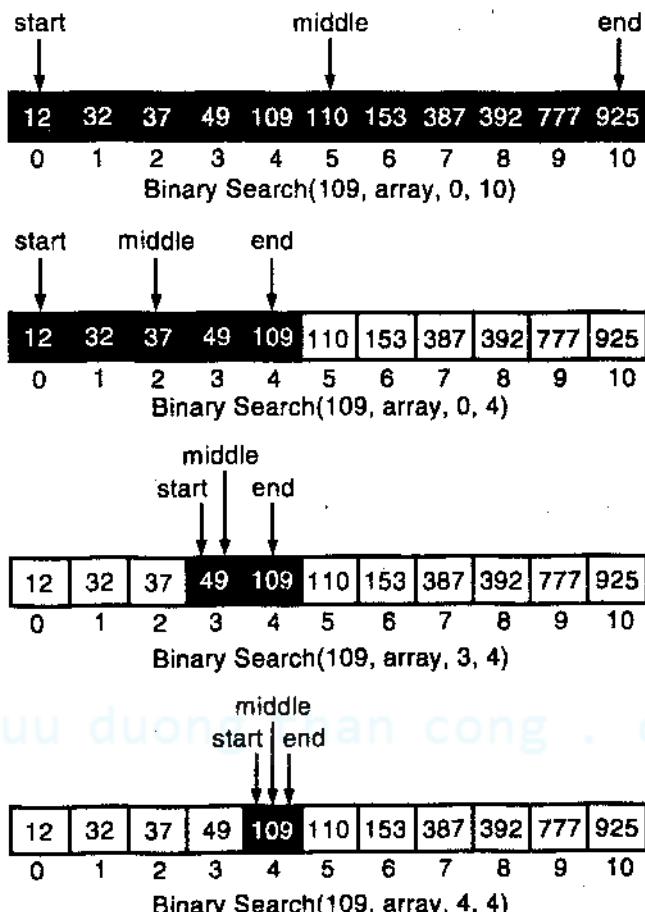
    /* Chúng ta đã tìm thấy */
    else if (list[middle] == item)
        return middle;

    /* Chúng ta nên tìm nửa đầu của mảng? */
    else if (item < list[middle])
        return BinarySearch(item, list, start, middle - 1);

    /* Hay là nên tìm nửa sau của mảng? */
    else
        return BinarySearch(item, list, middle + 1, end);
}

```

Hình 19.12 là một biểu diễn trực quan quá trình thực hiện đệ quy của ví dụ 19.4. Mảng list chứa 11 phần tử. Cuộc gọi ban đầu tới hàm Binarysearch cần được cung cấp giá trị chúng ta đang tìm (*item*) và mảng để tìm (tức địa chỉ của phần tử đầu tiên của mảng, hay địa chỉ nền của mảng). Cùng với mảng, chúng ta cũng cần cung cấp khu vực tìm kiếm trên mảng, tức điểm bắt đầu và điểm kết thúc của phần mảng mà trên đó ta đang tìm kiếm. Trong mỗi cuộc gọi đệ quy tiếp sau tới hàm Binarysearch, khu vực tìm kiếm này sẽ ngày càng nhỏ dần, cuối cùng tới được điểm mà ở đó tập con của mảng chúng ta đang tìm chỉ có duy nhất một phần tử hay không có phần tử nào cả. Đây là hai trường hợp gốc hay điều kiện kết thúc đệ quy.



Hình 19.12 Hàm *Binarysearch* được thực hiện trên mảng 11 phần tử để tìm phần tử 109

Đĩ nhiên, sau khi sắp xếp, chúng ta có thể tiến hành tìm kiếm phần tử trên mảng bằng phương pháp tuần tự, tức là chúng ta phải xét từng phần tử một, từ $list[0]$, rồi $list[1]$, $list[2]$, ..., và rồi cuối cùng hoặc tìm ra phần tử hoặc xác định nó không có trong mảng. Tuy nhiên, giải thuật tìm kiếm nhị phân đòi hỏi ít sự so sánh hơn và có thể thực thi thực sự nhanh hơn nếu mảng đủ lớn. Cụ thể, thời gian chạy của quá trình tìm kiếm nhị phân tỷ lệ thuận với $\log_2 n$, trong khi tìm kiếm tuần tự thì tỷ lệ với n .

19.6 CHUYỂN SỐ NGUYÊN SANG DÃY KÝ TỰ ASCII

Mục này trình bày một ví dụ khác của hàm đệ quy, đó là cách sử dụng hàm đệ quy để chuyển một giá trị nguyên bất kỳ sang chuỗi

ký tự ASCII. Chúng ta cũng nên biết rằng để biểu diễn một trị nguyên lên màn hình, mỗi ký số của trị này phải được trích ra một cách riêng lẻ, được chuyển sang mã ASCII và rồi được đưa tới thiết bị xuất liệu.

Chúng ta có thể làm điều này một cách đệ quy theo cách thức sau: nếu số được hiển thị chỉ có một chữ số, chúng ta chuyển nó sang ký số ASCII và hiển thị nó, thế là xong. Nhưng nếu số nguyên cần in có nhiều chữ số, chúng ta sẽ gọi đệ quy với đối số là số đã cho mà không có chữ số trọng số nhỏ nhất (bên phải), và khi hàm đệ quy trả về chúng ta hiển thị chữ số bên phải nhất này của số đó.

Ví dụ 19.5 trình bày hàm đệ quy trong C. Nó nhận vào một giá trị nguyên dương và chuyển mỗi chữ số của giá trị này sang mã ASCII, sau đó hiển thị các ký số kết quả.

Hàm đệ quy IntToAscii làm việc như sau: để in ra được một số, ví dụ 21669 (tức chúng ta cần gọi hàm IntToAscii(21669)), hàm sẽ chia nhỏ vấn đề ra làm hai phần. Đầu tiên số 2166 phải được in ra nhờ cuộc gọi đệ quy tới hàm IntToAscii, và một khi cuộc gọi đã xong, số 9 sẽ được in ra.

Hàm này sẽ bỏ chữ số trọng số nhỏ nhất của tham số *num*, tức dời nó qua phải một chữ số bằng thao tác chia cho 10. Với giá trị mới và nhỏ hơn này, chúng ta tiếp tục gọi đệ quy. Nếu giá trị nhập *num* chỉ có một chữ số, nó được chuyển ngay sang mã ASCII và được in ra màn hình, không có cuộc gọi đệ quy nào cần thiết trong trường hợp này.

Ví dụ 19.5

```
#include <stdio.h>
void IntToAscii(int i);
int main()
{
    int in;
    printf("Input number: ");
    scanf("%d", &in);
    IntToAscii(in);
    printf("\n");
}
```

```
}

void IntToAscii(int num)
{
    int prefix;
    int currDigit;

    if (num < 10) /* The terminal case */
        printf("%c", num + '0');
    else {
        prefix = num / 10; /* Convert the number */
        IntToAscii(prefix); /* without last digit */

        currDigit = num % 10; /* Then print last digit */
        printf("%c", currDigit + '0');
    }
}
```

Một khi điều khiển được trả về cho mỗi cuộc gọi, chữ số đã được bỏ đi được chuyển sang mã ASCII và được hiển thị. Để cho dễ hiểu, chúng ta biểu diễn chuỗi các cuộc gọi cho quá trình gọi `IntToAscii(12345)`:

IntToAscii(12345)

IntToAscii(1234)

IntToAscii(123)

IntToAscii(12)

IntToAscii(1)

printf('1')

printf('2')

printf('3')

printf('4')

printf('5')

19.7 CẤU TRÚC DỮ LIỆU CÂY - CÂY NHỊ PHÂN

Cây là sự tổng quát hóa của một danh sách liên kết (xin xem lại phần ứng dụng của chương con trỏ). Một nút trong danh sách liên kết có một nút đi ngay sau nó, được gọi là nút con hay nút hậu bối. Nút cuối cùng trong danh sách không có nút con. Chúng ta có thể mở rộng khái niệm này để cho phép một nút có thể có nhiều nút con.

Một chương trình khảo sát một cây như vậy có thể quyết định nút con mà nó sẽ đi theo dựa vào thông tin trong nút con đó, hoặc có thể chọn theo tất cả các nút con.

Một số khái niệm cơ bản về cây:

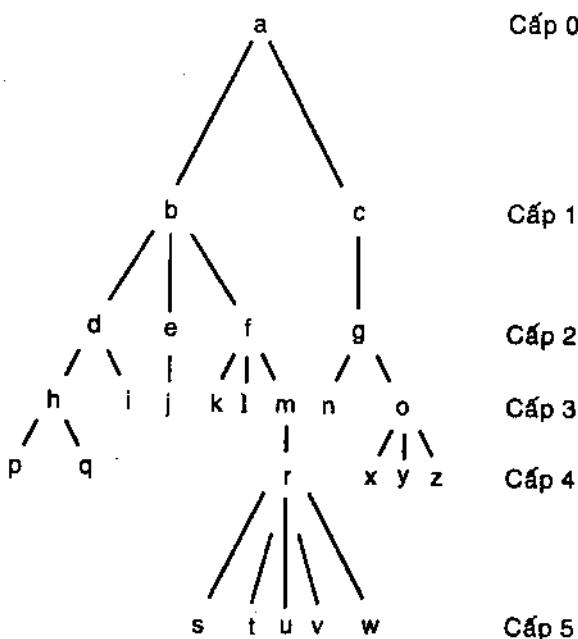
- Nút của cây mà không có con được gọi là nút lá.
- Nút của cây mà không có lá được gọi là nút trong.
- Số con mà một nút có được gọi là độ của nút đó.
- Nút trên cùng trong một cây, tức là nút dẫn tới tất cả nút khác, được gọi là nút gốc.
- Độ sâu của một nút x là số nút giữa nút gốc và x . Nút gốc được coi như có độ sâu bằng 0; các nút con của nút gốc đều ở độ sâu bằng 1, ... Độ cao của một cây là độ sâu lớn nhất của nút có được. Để thuận tiện, độ cao của một cây rỗng được định nghĩa bằng -1. Tất cả các nút cùng một độ sâu d thì ở trong một tập hợp mức d (còn gọi là cấp d).

Thông thường, chúng ta sử dụng các mối quan hệ gia đình (con, cháu, cha mẹ,...) để mô tả các quan hệ giữa các nút trong cây. Tương tự, các khái niệm cho cây cũng được sử dụng, như tập hợp nhiều cây là một khu rừng, cây với nhiều nhánh gọi là cây rậm rạp,...

Có hai cách để tạo ra cây:

- (1) tạo cây từ lá tới gốc, được gọi là bottom up (từ dưới lên)
- (2) tạo cây từ gốc tới lá, được gọi là top down (từ trên xuống).

Dưới đây là một hình ví dụ về cây

**Hình 19.13** Một ví dụ về cây

cuu duong than cong . com

19.7.1 Hiện thực hóa cây

Như các cấu trúc dữ liệu khác, các nút trong cây là để lưu trữ một loại dữ liệu nào đó, cụ thể dữ liệu này là một mẫu tin, ví dụ như là mẫu tin sinh viên với khóa là mã số sinh viên, MSSV. Để đơn giản chúng ta xét dữ liệu là một khóa, là một số nguyên.

Làm cách nào biểu diễn cây? Nếu chúng ta biết được độ tối đa (số con) của bất kỳ nút trong nào của cây, chúng ta có thể xử lý cây như với một danh sách liên kết, nhưng thay vì dùng một pointer chỉ tới phần tử kế thì ta cần một mảng các pointer chỉ tới các nút kế, khi đó ta có khai báo như sau:

Ví dụ 19.6

```

#define MAX_DEGREE 20 /* lên tới 20 con */

typedef struct _treenode
{
    int k; /* thông tin */
    struct _treenode *children[MAX_DEGREE];
} treenode;
  
```

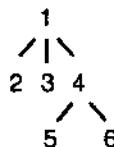
và chúng ta cũng sử dụng con trỏ NULL để chỉ liên kết rỗng, không có nút kế.

Nếu chúng ta không biết độ tối đa của nút, chúng ta sử dụng một danh sách liên kết các anh em mà mỗi cái trong chúng sẽ liên kết tới một danh sách liên kết các con, như khai báo sau:

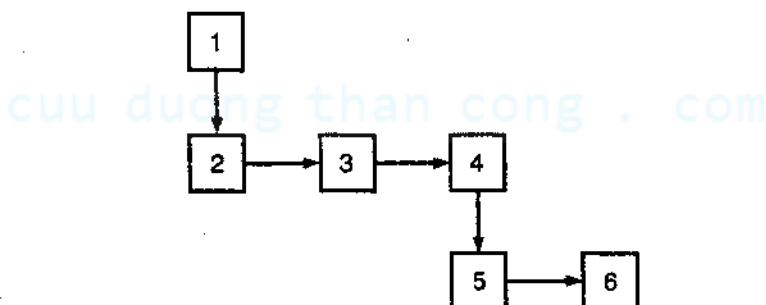
Ví dụ 19.7

```
typedef struct _treenode
{
    int k;                                /* thông tin */
    struct _treenode *children,             /* các con */
    *next_sibling;                         /* anh em */
} treenode;
```

Trong khai báo trên, vùng tin *next_sibling* chỉ tới nút kế trong danh sách các anh em ở cùng cấp, còn vùng *children* chỉ tới một danh sách các con, danh sách này được liên kết tới từ vùng tin *next_sibling*. Để cụ thể chúng ta xem cây sau



Theo định nghĩa trên, cây này được biểu diễn như hình sau:



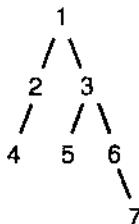
Hình 19.14 Cấu trúc một cây chưa biết độ của nút

Làm thế nào chúng ta chèn thêm phần tử vào một cây như vậy? Tất cả tùy thuộc vào bản chất trị của khóa và quy định sắp xếp của

cây. Trong thực tế lập trình, cây nhị phân tìm kiếm thường được sử dụng, nên mục sau trình bày vấn đề này.

19.7.2 Cây nhị phân

Cây nhị phân là cây mà trong đó độ lớn nhất của bất kỳ nút nào cũng là 2. Như vậy một nút có thể có 0, 1, hoặc 2 cây. Đây là một cây nhị phân

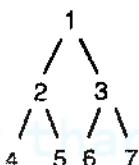


Với cây trên, độ cao của cây là bao nhiêu, nút nào là lá, nút nào là nút trong? Tất cả khái niệm về cây mà chúng ta đã biết đều được sử dụng cho cây nhị phân.

Một trường hợp đặc biệt của cây nhị phân là cây nhị phân hoàn chỉnh. Cây nhị phân hoàn chỉnh là cây có hai đặc điểm:

1. Mọi nút trong đều có độ chính xác là 2
2. Mọi nút lá đều ở cùng một cấp.

Ví dụ sau là một cây nhị phân hoàn chỉnh



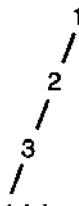
Cây nhị phân hoàn chỉnh có ý nghĩa sử dụng quan trọng vì nó cho phép ta khai thác đặc điểm của cây nhị phân trong một bối cảnh đơn giản.

Có bao nhiêu nút lá trong cây nhị phân hoàn chỉnh độ cao h ? Để dễ hình dung, ta gọi số nút lá ở độ cao h là $f(h)$, khi đó số nút lá ở độ cao h sẽ gấp đôi số nút lá ở độ cao $h-1$, tức $f(h) = 2 \times f(h-1)$. Điều này cũng có thể được mô tả qua hàm đệ quy $f(h)$ như sau:

$f(h) = 1$, khi $h = 0$ (tức khi cây chỉ có nút gốc)
 $2 \times f(h)$, khi h khác

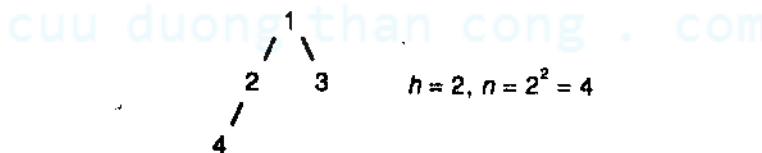
Đây chính là phép cộng dồn với cùng giá trị, nên chính là mũ, nên $f(h)$ sẽ là 2^h , từ đó chúng ta có thể dễ dàng thấy tổng số nút có trong một cây độ cao h là $2^{h+1} - 1$.

Độ cao lớn nhất của một cây nhị phân có n nút là bao nhiêu? Nếu cây được trải ra như một danh sách liên kết như hình sau

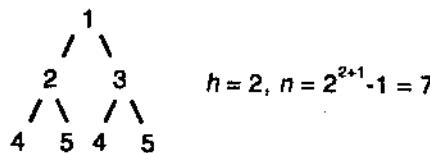


thì độ cao của cây là $n-1$. Dĩ nhiên là chúng ta không thích một cây phân bố như vậy. Một câu hỏi hay hơn nữa là độ cao nhỏ nhất của cây nhị phân có n nút là bao nhiêu? Để trả lời câu hỏi này, chúng ta cần xác định cấu trúc nút sẽ tạo ra độ cao nhỏ nhất. Một cây nhị phân gần như hoàn chỉnh, có nghĩa là một cây nhị phân hoàn chỉnh với hầu như chỉ có một vài nút lá bị thiếu, có thể có bất kỳ số nút lá nào, do đó ta sẽ xét loại cây mà không mất tính tổng quát.

Độ cao của cây nhị phân gần hoàn chỉnh độ cao h tương tự như độ cao của cây nhị phân hoàn chỉnh độ cao h , và hơn nữa cũng như cây nhị phân hoàn chỉnh độ cao $h-1$. Có bao nhiêu nút trong cây nhị phân hoàn chỉnh độ cao $h-1$? Chính xác là 2^{h-1} . Vì vậy một cây nhị phân gần hoàn chỉnh sẽ có số nút giữa 2^h , nghĩa là:



và $2^{h+1}-1$, tức:



Tức n phải là một giá trị giữa 2^h và $2^{h+1}-1$, nên ta có thể thấy bất kỳ cây nhị phân gần hoàn chỉnh nào có số nút nằm trong giới hạn này sẽ có độ cao bằng h . Nghĩa là một cây nhị phân có n nút sẽ có độ cao h là $h = \text{floor}(\log_2 n)$, với hàm $\text{floor}(n)$ cho trị gần nhất nhỏ hơn hay bằng n .

Việc biết được chiều cao của cây sẽ giúp chúng ta nhiều trong lập trình. Giả sử chúng ta cần tìm một thông tin quan trọng trong một cấu trúc dữ liệu như cây chẵng hạn, chúng ta biết là nó nằm dọc theo đường từ nút đầu tiên tới nút cuối cùng, là nút lá hay là nút cuối của danh sách. Nếu là một danh sách liên kết, việc đi từ đầu tới cuối danh sách sẽ mất khoảng thời gian $\Omega(n)$. Trong cây nhị phân, việc đi từ nút gốc tới nút lá chỉ mất khoảng thời gian $\Omega(\log_2 n)$. Vì $\log_2 n$ nhỏ hơn nhiều so với n , ví dụ $\log_2 1\ 000\ 000\ 000 \approx 30$, nên chúng ta có thể tận dụng đặc tính này để tạo ra các cấu trúc dữ liệu tìm kiếm hiệu quả.

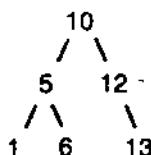
19.7.3 Cây nhị phân tìm kiếm

Một cấu trúc dữ liệu với việc bố trí dữ liệu thuận lợi cho việc tìm kiếm là cây nhị phân tìm kiếm. Các khóa trong các nút phục vụ việc tìm kiếm trong cây phải được lưu trữ theo một trật tự chung được quy định trước. Thí dụ, nếu khóa là một số nguyên hay thực thì trật tự này là từ nhỏ hơn hay bằng. Nếu khóa là một chuỗi, thì thứ tự này sẽ theo thứ tự từ điển, đó là trình tự abc không phân biệt mã ASCII hoa hay thường. Ta có thể dùng ký hiệu \leq để chỉ quy định này.

Như vậy cây nhị phân tìm kiếm là cây nhị phân thỏa:

- rỗng, hay
- có một nút lá, hay
- có một nút trong với giá trị khóa x sao cho tất cả các nút ở bên trái cây nhị phân tìm kiếm đều $\leq x$, và $x \leq$ tất cả các nút ở bên phải cây nhị phân tìm kiếm.

Hình dưới đây cho thấy một cây nhị phân tìm kiếm như thế.



Việc tìm kiếm dữ liệu (tức khóa) trong cây nhị phân tìm kiếm đơn giản là quá trình tìm kiếm nhị phân. Khi đó nếu dữ liệu cần tìm mà không nằm ở nút hiện thời và nó nhỏ hơn nút hiện thời thì việc tìm kiếm sẽ diễn ra ở bên trái cây, còn nếu nó lớn hơn trị hiện thời thì nó có khả năng nằm bên phải cây. Nếu chúng ta đi tới một điểm mà sau đó không còn nút để di tiếp thì dữ liệu cần tìm không có trong cây.

Chúng ta hãy xét việc hiện thực hóa cây nhị phân tìm kiếm với khóa là các số nguyên và trật tự quy định sắp xếp là từ nhỏ tới lớn từ trái qua phải cho các nút cùng cấp. Cấu trúc nút như vậy cần có không gian cho khóa, cho các pointer chỉ tới các cây con trái và phải giống như cây nhị phân bình thường, và nó được mô tả như sau:

Ví dụ 19.8

```
typedef struct _bstreenode
{
    int k; /* khóa */
    struct _bstreenode
    {
        *left, /* cây con trái */
        *right; /* cây con phải */
    } bstreenode, *bstree;
```

Một số thao tác với cây nhị phân tìm kiếm thường được sử dụng là: tìm kiếm, chèn nút, và duyệt cây.

1. Tìm kiếm trong cây nhị phân

Giả sử cây nhị phân tìm kiếm đã được khởi tạo và có kiểu bstree như khai báo trên. Hàm sau sẽ tìm khóa k trong cây bstree. Chú ý rằng, con trỏ NULL được sử dụng để báo việc hết cây con theo hướng đang xét. Như vậy, nếu việc tìm kiếm là thành công chúng ta cần phải trả về con trỏ chỉ tới nút lưu dữ liệu khóa cần tìm, còn nếu không tìm thấy, con trỏ NULL sẽ được trả về để báo việc này.

Ví dụ 19.9

```
int *search_bstree (bstree *t, int k)
{
    while (t) {
```

```

if (t->k == k) return &(t->k);      /* đã tìm ra */
if (k <= t->k)
    t = t->left;                  /* di về cây trái */
else
    t = t->right;                /* di về cây phải */
}
return NULL;
}

```

2. Chèn một nút mới vào cây

Để chèn một nút mới vào cây chúng ta cần chú ý tới trật tự quy định chung của dữ liệu khóa trong cây, sau khi chen nút mới vào trật tự này phải được bảo toàn. Do đó các thao tác lưu, chép để tạo các liên kết cần được chú ý. Hàm sau trình bày thao tác này.

Ví dụ 19.10

```

void insert_bstree (bstree *t, int k)
{
    bstree p;

    p = (bstree) malloc (sizeof (bstreenode));
    p->left = NULL;
    p->right = NULL;
    p->k = k;
    while (*t) {
        if (k <= (*t)->k)
            t = &(*t)->left;
        else
            t = &(*t)->right;
    }
    *t = p;
}

```

3. Duyệt cây

Nếu chúng ta muốn xem tất cả nút trong một cây nhị phân tìm kiếm, thì chúng ta cần thao tác duyệt cây. Do đã biết trước trật tự sắp xếp của cây, chúng ta có thể in ra các phần tử của cây theo thứ tự sau:

- Xem mọi thứ ở cây bên trái.
- Xem nút gốc.
- Xem mọi thứ ở cây bên phải.

Nếu chúng ta theo luật này cho mọi nút, thì trình tự các nút sẽ được in ra là: đầu tiên mọi nút bên trái sẽ được in ra, tới nút hiện thời, sau đó là mọi nút bên phải. Trình tự duyệt này được gọi là duyệt theo thứ tự trong (*inorder*). Thiết kế đệ quy có thể được sử dụng để viết hàm này. Hàm sau đây sẽ in ra tất cả các nút trong cây nhị phân:

Ví dụ 19.11

```
void traverse_tree (bstree t)
{
    if (!t) return;
    traverse_tree (t->left);
    printf ("%d\n", t->k);
    traverse_tree (t->right);
}
```

Thật là quá dễ dàng. Bây giờ chúng ta hãy xem kiểu viết không đệ quy của hàm duyệt này, lưu ý việc có sử dụng hiện thực cấu trúc stack cho các pointer chỉ tới nút của cây.

Ví dụ 19.12

```
void traverse_nr (bstree t)
{
    do {
        while (t)
            push (t);
        t = t->left;
    }
    if (!empty_stack())
    {
        t = pop ();
        printf ("%d\n", t->k);
        t = t->right;
    }
} while (t || !empty_stack());
}
```

Thao tác duyệt dựa vào stack có thể hiệu quả hơn vì nó thuận lợi cho phát hiện sai và sửa sai cũng như thấy rõ quá trình duyệt, nhưng thật khó để đọc và sửa chữa mã chương trình. Thường việc duyệt theo mã đệ quy được sử dụng nhiều.

Nếu chúng ta muốn xóa cây, dĩ nhiên ta cần phải xóa tất cả các nút trong cây, tức giải phóng bộ nhớ khi chúng ta dùng xong, chúng ta cũng dùng cách duyệt thứ tự trong như hàm sau đây:

Ví dụ 19.13

```
void delete_tree (bstree t)
{
    if (!t) return;
    traverse_tree (t->left);
    traverse_tree (t->right);
    free (t);
}
```

Với kiểu xóa cây như vậy chúng ta sẽ lần lượt tới cây con bên trái và xóa nó, sau đó tới cây con bên phải, và sau cùng là nút hiện thời. Kiểu duyệt này gọi là duyệt theo thứ tự sau (*postorder*). Ngoài ra, còn một cách duyệt cây nữa, đó là duyệt theo thứ tự trước (*preorder*). Khi đó nút hiện thời sẽ được thăm trước, sau đó tới cây con trái, và sau cùng là cây con phải. Độc giả có thể tự viết các hàm duyệt này, để mở rộng kiến thức.

19.7.4 Một cấu trúc dữ liệu tìm kiếm tổng quát

Cấu trúc dữ liệu cây nhị phân tìm kiếm rất tốt cho việc lưu trữ, tìm kiếm không những cho số nguyên, mà còn cho bất cứ loại dữ liệu mẫu tin nào, và ta không cần phải viết lại mã chương trình cho việc truy cập cây nhị phân tìm kiếm cho loại dữ liệu đó. Đây thật là một mục tiêu rất tốt khi viết mã. Chương trình sau đây hiện thực một dạng cây nhị phân tìm kiếm mục đích tổng quát và được sử dụng cho một giải thuật sắp xếp chuỗi. Chương trình này bao hàm việc truyền các tham số là các pointer chỉ tới hàm mà chúng ta cần lưu ý.

Ví dụ 19.14

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct _gbstreenode
{
    void *k; /* pointer chỉ tới kiểu dữ liệu chưa xác định,
    * người cần cung cấp khi sử dụng
    */
    struct _gbstreenode
        *left, /* cây con trái */
        *right; /* cây con phải */
} gbstreenode, *gbstree;

/* Tìm một cây tổng quát */
void *search_gbtree (gbstree t, void *k, int (*compar)(void *, void *))
{
    /* compar là một hàm có hai con trả a và b, trả về trị
     * < 0 nếu *a < *b, 0 nếu *a == *b, và > 0 nếu *a > *b
     */
    int c;

    while (t)
    {
        c = compar (t->k, k);
        if (c == 0) return &(t->k); /* found it */
        if (c < 0)
            t = t->left; /* go left */
        else
            t = t->right; /* go right */
    }
    return NULL;
}

/* chèn node */
```

```
void insert_gbmtree (gbstree *t, void *k, int (*compar)(void *, void *))
{
    gbstree p;
    int c;

    p = (gbstree) malloc (sizeof (gbstreenode));
    p->left = NULL;
    p->right = NULL;
    p->k = k;
    while (*t)
    {
        c = compar (k, (*t)->k);
        if (c < 0)
            t = &(*t)->left;
        else
            t = &(*t)->right;
    }
    *t = p;
}

/* duyệt cây */
void gbsinorder_traverse_tree (gbstree t, void (*visit)(void *))
{
    /* visit là một hàm nhận một con trỏ, làm một việc nào đó mà chúng ta
     không quan tâm */
    int c;

    if (!t) return;
    gbsinorder_traverse_tree (t->left, visit);
    visit (t->k);
    gbsinorder_traverse_tree (t->right, visit);
}

void gbspostorder_traverse_tree (gbstree t, void (*visit)(void *))
{
    /* visit là một hàm nhận một con trỏ, làm một việc nào đó mà chúng ta
     không quan tâm */
    int c;
    if (!t) return;
```

```

gbspostorder_traverse_tree (t->left, visit);
gbspostorder_traverse_tree (t->right, visit);
visit (t->k);
}

/* hàm 'visit' dùng cho cây các chuỗi, in chuỗi */
void print_string (char *s)
{
    fputs (s, stdout);
}

/* strcmp(), already in the standard C library, serves for compar */
/* chương trình này đọc các chuỗi và in chúng ra theo thứ tự đã được sắp xếp */
int main ()
{
    char s[100], *p;
    gbstree t;
    /* empty tree */
    t = NULL;

    /* lập cho tới cuối file */
    for (;;)
    {
        /* lấy một chuỗi */
        fgets (s, 100, stdin);
        if (feof (stdin)) break;

        /* strdup() dùng malloc để sao chuỗi, vì vậy nó sẽ có vùng nhớ riêng trong
        cây */
        p = strdup (s);
        /* chèn và gộp chuỗi */
        insert_gbtree (&t, p, (int (*)(void*,void*))strcmp);
    }

    /* duyệt cây và gộp */
    gbsinorder_traverse_tree (t, (void(*)(void*))print_string);
    /* duyệt theo thứ tự sau, giải phóng từng node */
    gbspostorder_traverse_tree (t, (void(*)(void*))free);
    exit (0);
}

```

Chương 20

GIỚI THIỆU LẬP TRÌNH C++

Chương này trình bày những đặc điểm của ngôn ngữ C++ dựa vào ngôn ngữ C đã được học, từ đó người học hiểu các chương trình C++ và vận dụng mà không thấy bỡ ngỡ như học một ngôn ngữ mới. Cái khác nhau chủ yếu giữa C++ và C là C++ đưa ra khái niệm đối tượng.

20.1 LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

Trong ngôn ngữ lập trình C, nếu muốn dấu mã nguồn chương trình thì chúng ta sẽ tạo nên một tập tin trong đó các hàm được khai báo tĩnh, *static*. Tuy nhiên, việc thiết kế chương trình sẽ hay hơn khi chúng ta có thể tạo ra các “module mã”, trong đó cách duy nhất các dữ liệu chương trình có thể thay đổi được là phải thông qua những hàm cụ thể, trong lập trình hướng đối tượng được gọi là phương thức, *method*.

C++ đưa ra khái niệm “module mã” bằng các lớp, *class*, mà trong lập trình hướng đối tượng gọi là đối tượng, *object*. Một class là một cấu trúc những thông tin thêm vào bao gồm sự dấu thông tin và tầm vực truy cập, các khai báo prototype hàm, và khai báo biến tĩnh. C++ cho phép lớp khai báo các phần thấy được, tức bên ngoài truy cập được, hay còn được gọi là toàn cục, và các phần riêng của lớp, còn được gọi là cục bộ.

Thấy được (toàn cục)	Riêng (cục bộ)
Sự tồn tại của các cấu trúc Các hàm giao tiếp	Các dữ liệu trong cấu trúc Các hàm trong đối tượng Các khởi tạo (constructor) Các bộ hủy (destructor)

Khi lập trình, mã của hàm đang hoạt động là một bản sao với đối số cụ thể của hàm đó. Chúng ta hãy xét cấu trúc *player_t* dưới

đây, nó mang các thông tin của một trong các game thủ trong một trò chơi mà chúng ta đang xây dựng.

```
struct player_t
{
    char * name;           /* tên đăng nhập về game thủ */
    char *password;        /* password */
    int num_played;        /* số lần chơi */
    int win_guesses[13];   /* số lần đoán trúng là thắng */
    double win_percent;    /* phần trăm của số lần thắng */
};
```

Trong C, chúng ta có thể khai báo một hàm mà sẽ được gọi khi game thủ thắng trò chơi. Hàm này sẽ cập nhật các biến **int num_played**, **win_percent**, và mảng **win_guesses** và được khai báo như sau:

```
void player_win (player_t * p, int num_guesses);
```

Hàm này sẽ được gọi như sau:

```
player_t player1; /* chấp nhận game thủ đã được khởi động rồi */
                    /* một vài đoạn mã ở đây */
                    /* game thủ player1 thắng trò chơi sau 5 lần đoán */
player_win (&player1, 5);
```

Bây giờ đoạn mã nguồn trên được viết lại bằng đối tượng lớp mã C++ như sau:

```
class player_t
{
public:
    void player_win (int num_guesses); /* đổi số player_t ở dạng ám chỉ */
private:
    char * name;           /* tên đăng nhập của game thủ */
    char *password;        /* password */
    int num_played;        /* số lần chơi */
    int win_guesses[13];   /* số lần đoán trúng là thắng */
    double win_percent;    /* phần trăm của số lần thắng */
};
```

Trong C++, đối số **player_t *** ở dạng ám chỉ khi chúng ta gọi hàm **player_win**. Hàm **player_win** được khai báo trong lớp như sau:

```
void player_win (int num_guesses);
```

Một hàm được khai báo như trên được gọi là một phương thức, method, có nghĩa là đối số đầu tiên của nó là ám chỉ, và kiểu của nó là pointer chỉ tới class đó. Một phương thức khi được gọi sẽ thực hiện các hành vi tương ứng cho đối tượng mà nó khai báo trong đó.

Để thực hiện việc gọi hàm trong C++, chấp nhận các biến đã được khởi động trị, ta thấy

```
player_t * p;           /* pointer tới class player */
/* một vài đoạn mã ở đây */
p->player_win (5);    /* game thủ chiến thắng sau 5 lần đoán */
```

Hàm **player_win** sẽ được định nghĩa như sau:

```
int player_t:: player_win (int num_guesses)
{
    this-> win_guesses[this->num_played] = num_guesses;
    this-> win_percent = ((this->num_percent * this->num_played ) + 1)
                           / (this->num_played +1);
    this->num_played++;
}
```

Danh hiệu lớp **player_t** theo sau là dấu:: và một phương thức, có nghĩa phương thức này là một thành viên của lớp **player_t**. Ký hiệu này được gọi là tầm vực, và chúng ta phải cung cấp đúng tầm vực khi viết các định nghĩa C++. Pointer “**this**” ở trên là một pointer ám chỉ tới đối tượng gọi phương thức, trong trường hợp này là **player_t ***. Ví dụ, nếu trong đoạn mã trên chúng ta có dòng lệnh **p->player_win(5)**; thì “**this**” sẽ đóng vai trò hoàn toàn tương tự như **p**. Tuy nhiên để tránh sự lặp đi lặp lại, C++ cho phép chúng ta không cần viết nó khi định nghĩa, và bộ dịch sẽ tự động thêm vào khi dịch. Như vậy, định nghĩa hàm **player_win** được viết lại như sau:

```

int player_t::player_win (int num_guesses)
{
    win_guesses[this->num_played] = num_guesses;
    win_percent = ((this->num_percent * this->num_played ) + 1)
                    / (this->num_played +1);
    num_played++;
}

```

20.2 CONSTRUCTOR VÀ DESTRUCTOR

C++ cho phép chúng ta khai báo các đoạn mã để bắt cứ khi nào muốn chúng ta đều có thể khởi tạo hay hủy một đối tượng. Đoạn mã như vậy được gọi là bộ tạo, *constructor*, dùng để khởi tạo đối tượng, và bộ hủy, *destructor*, để hủy đối tượng. Nếu chúng ta không khai báo cụ thể các *constructor* và *destructor* thì bộ dịch C++ sẽ tạo chúng một cách tự động. Ví dụ, khi muốn khởi tạo tất cả các giá trị trong cấu trúc của game thủ thì chúng ta sẽ thêm các khai báo constructor và destructor vào class như sau:

```

class player_t
{
public:
    player_t ();                                /* constructor */
    ~player_t ();                               /* destructor */
    void player_win (int num_guesses); /* đối số player_t* ở dạng ám chỉ */
private:
    char * name;                                /* tên đăng nhập của game thủ */
    char *password;                            /* password */
    int num_palyed;                            /* số lần chơi */
    int win_guesses[13];                         /* số lần đoán trúng là thắng */
    double win_percent;                         /* phần trăm của số lần thắng*/
};

/* Constructor

```

* Chúng ta khởi tạo trị và vùng nhớ động cho *name* và *password* bằng việc dùng toán tử * mới, xem mô tả sau

```

/*
player_t:: player_t ()
{
    num_played] = 0;
    for (int i = 0; i < 13; i++)
        win_guesses [i] = 0;
    win_percent = 0;
    name = new char [10]; /* bộ nhớ động cho 9 ký tự và NUL */
    password = new char [10];
}

/* Destructor

* Chúng ta cần hủy vùng nhớ động đã xin trong constructor
*/
player_t:: ~player_t ()
{
    delete [] name; /* xóa vùng nhớ động xin trong constructor */
    delete [] password;
}

```

Bây giờ, bất cứ khi nào chúng ta tạo một đối tượng kiểu `player_t`, constructor của đối tượng sẽ được bộ dịch gọi, và đối tượng sẽ được khởi tạo. Việc khởi tạo này sẽ gán trị 0 cho tất cả các vùng tin và vùng nhớ động `name` và `password`. Phương thức *destructor* được bộ dịch gọi khi đối tượng cần được hủy, nó sẽ giải phóng vùng nhớ động đã được khởi tạo trong constructor. Ví dụ, xét hàm sau đây:

```

void play_round ()
{
    player_t player1; /* mặc nhiên gọi constructor player_t */
    player_t player2; /* mặc nhiên gọi constructor player_t */
    /* đoạn mã chơi trò chơi */
}

```

Hàm `play_round` khai báo hai biến cục bộ: `player1` và `player2`, bộ dịch sẽ chèn đoạn mã gọi bộ tạo constructor `player_t` khi mỗi biến cục bộ này được tạo ra lúc khởi động hàm. Sau khi hàm này kết thúc, các biến cục bộ này không còn cần thiết nữa. Trước khi điều khiển

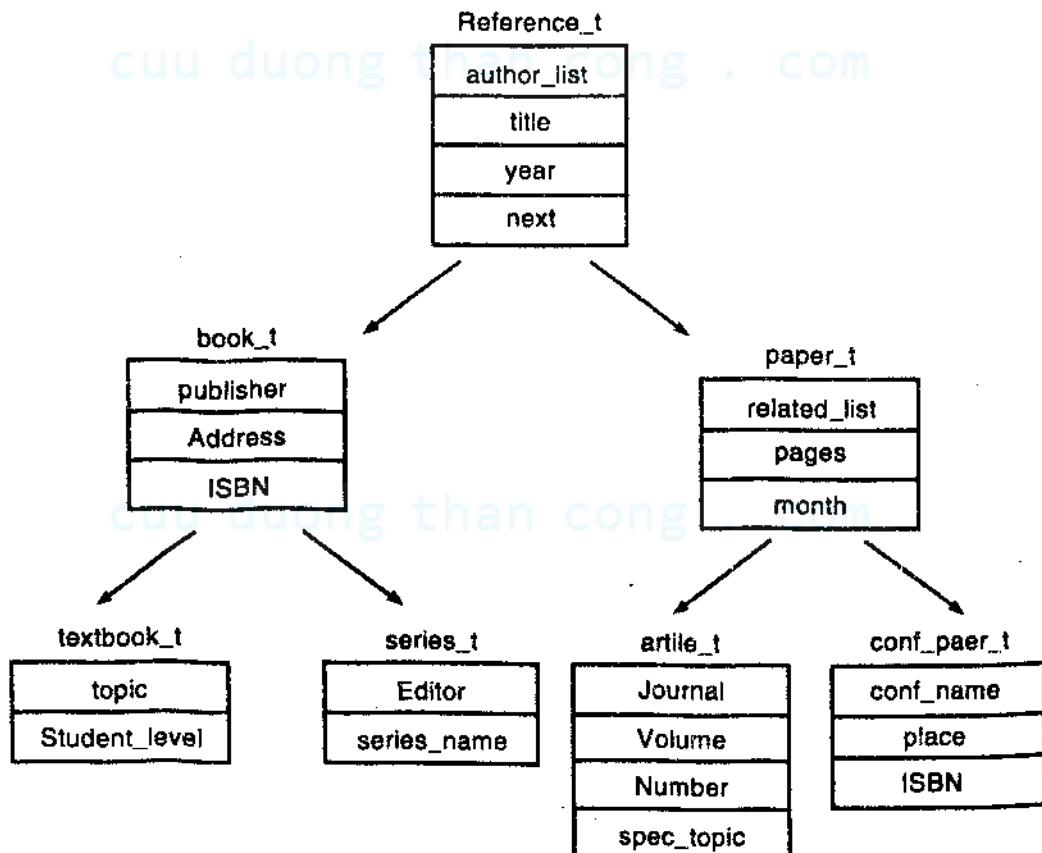
chương trình được trả về nơi gọi hàm, bộ dịch C++ sẽ chèn đoạn mã gọi bộ hủy destructor *player_t* cho cả hai đối tượng *player1* và *player2*. Việc gọi này được bộ dịch C++ thực hiện một cách tự động.

20.3 TOÁN TỬ NEW VÀ DELETE

Bên cạnh việc dùng **malloc** và **free**, chúng ta có thể định vị bộ nhớ động bằng việc dùng các toán tử **new** và **delete**. Các toán tử này cũng ủng hộ việc gọi các *constructor* và *destructor* của đối tượng. Toán tử **new** sẽ gọi hàm **malloc** khi bộ tạo được thực hiện, toán tử **delete** sẽ gọi hàm **free** khi bộ hủy được gọi.

20.4 SỰ THỪA KẾ DỮ LIỆU

C++ cung cấp khái niệm thừa kế dữ liệu. Việc thừa kế này rất hữu ích khi ta muốn tạo ra các đối tượng mà có thể chia sẻ thông tin với nhau.



Hình 20.1 Sơ thừa kế dữ liệu

Xét minh họa trong hình 20.1 dưới đây. Đầu tiên, trên đỉnh của cửa sổ hình 20.1, chúng ta có một danh sách *reference_t* lưu các chỉ mục tham khảo cần thiết để lưu trữ trong thư viện, bao gồm các thành phần *author_list*, *title*, *year*, và một *next* để chỉ tới *reference_t* kế tiếp. Tuy nhiên, đối với sách và báo, chúng ta cần nhiều thông tin hơn. Đối với sách, chúng ta cần thêm thông tin về nhà xuất bản, địa chỉ, và số ISBN. Hơn nữa, đối với các bài báo hàn lâm, chúng ta cần có thêm các thông tin khác nữa như danh sách các bài báo liên quan, số trang, thời điểm viết. Tương tự, với sách giáo khoa và tài liệu mới, chúng ta cần thêm các thông tin về nội dung sách, thông tin tham khảo.

Nếu mỗi kiểu đối tượng phải khai báo đi khai báo lại cho đủ chỉ mục cho từng loại thông tin thì quả là phí vì chúng có các thông tin trùng nhau. Làm thế nào chúng ta thực hiện việc này trong C++? Câu trả lời là lớp trong C++ có thể kế thừa từ các lớp khác. Kết quả là sự kế thừa lớp được tạo ra từ những lớp trên một lớp nào đó, và cho lớp dưới lớp đang xét.

Hình 20.2 Phân cấp *reference_t*

author_list
title
year
next

author_list
title
year
next
publisher
Address
ISBN
Editor
series_name

Hình 20.2
Phân cấp *reference_t*

Hình 20.3
Phân cấp *series_t*

Nếu chúng ta tạo khởi tạo lớp *reference_t* thì dữ liệu như trong hình 20.2 sẽ hình thành. Tuy nhiên, nếu chúng ta tạo một lớp *series_t*, nó sẽ được tạo riêng ra như hình 20.3.

Giả sử chúng ta tạo hàm **void print_cite()**; được định nghĩa trong lớp *reference_t*. Điều gì sẽ xảy ra nếu ta làm điều sau đây:

```
series_t * s;      /* chấp nhận bộ nhớ đã được định vị và khởi tạo */
s->print_cite();
```

Lệnh trên sẽ được thực hiện mặc dù chúng ta đang gọi một phương thức *reference_t* từ đối tượng *series_t*. C++ cho phép làm điều này vì sự gọi tới lớp trên là tự động trong C++. Ở đây ta cần chú ý tới cách mà *s* chỉ tới cùng một dữ liệu như con trỏ *r* tới đối tượng *reference_t*. Khi một đối tượng *series_t* gọi hàm **print_cite**, hàm **print_cite** sẽ đơn giản bỏ qua tất cả dữ liệu sau con trỏ *next*.

20.5 TỪ KHÓA STATIC

Từ khóa **static** khi được sử dụng trước một khai báo hàm trong đối tượng sẽ báo hàm đó không phải là phương thức của đối tượng, và vì vậy nó không đặc trưng cho một trường hợp xử lý đặc biệt của một lớp. Xét ví dụ sau đây:

```
class reference_t
{
public:
    static reference_t * find_author (const char * find);
private:
    author_list * author_list;
    char * title;
    int year;
    reference_t * next;
};
```

Khi gọi hàm **find_author** trên, chúng ta không muốn gọi nó như là một phương thức của đối tượng *reference_t* cụ thể, mà chúng ta muốn truy tìm danh sách các đối tượng *reference_t* cho một tác giả xác định.

Mặt khác, hàm **has_author** dưới đây nếu được khai báo trong đối tượng *reference_t*, nó sẽ là một phương thức vì nó cần hoạt động với một đối tượng xác định:

```
int has_author (const char * find);
r ->has_author ("Lumetta");
/* kiểm tra đối tượng để xem Lumetta có phải là tác giả không */
```

20.6 HÀM ẢO (SỰ THỪA KẾ THEO CHỨC NĂNG)

Nếu chúng ta đang trong quá trình tạo các lớp cấp dưới, mà chúng ta lại không biết có bao nhiêu lớp cấp dưới này cần tạo thì sao? Làm thế nào chúng ta có thể viết một hàm để in tất cả các thông tin về mỗi đối tượng của chúng ta? Câu trả lời là hàm ảo (*virtual function*).

Việc khai báo hàm ảo sẽ được thực hiện bằng từ khóa *virtual* như ví dụ sau đây:

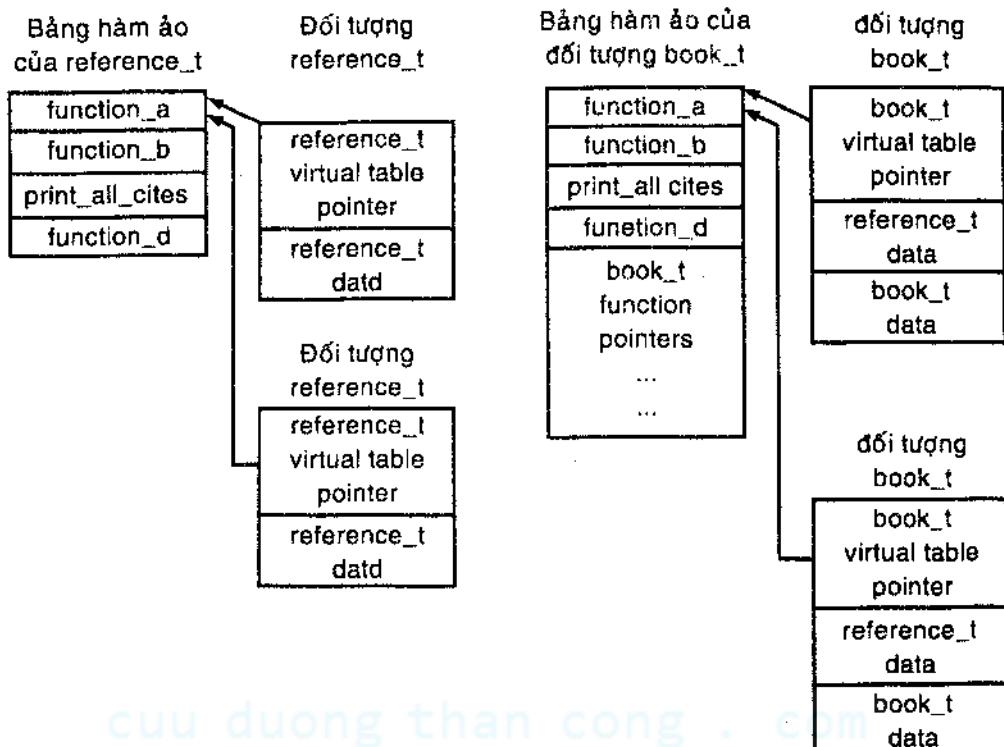
```
virtual void print_all_cites ();
reference_t * r;
r->print_all_cites ();
```

Xét ví dụ hình 20.4 dưới đây. Trong hình này, mỗi lớp đều có một bảng hàm ảo chứa các con trỏ tới các hàm. Một trong những hàm trong bảng hàm ảo của đối tượng *reference_t* là *print_all_cites*. Bảng hàm ảo *book_t* kế thừa những hàm ảo của lớp *reference_t*. Cái mà chúng ta cần làm là thay thế con trỏ hàm *print_all_cites* trong bảng hàm ảo của *book_t* bằng con trỏ chỉ tới hàm khác. Ví dụ, chúng ta có thể thay thế *reference_t::print_all_cites* trong bảng hàm ảo *book_t* ở nửa phải của hình 20.4 bằng một con trỏ hàm chỉ tới một hàm mới là *book_t::print_all_cites*.

Bây giờ nếu chúng ta viết

```
book_t * b;
b->print_all_cites ();
```

thì hàm *book_t::print_all_cites* sẽ được gọi thay vì hàm *reference_t::print_all_cites*. Chúng ta hãy thử xem việc này được thực hiện chính xác như thế nào. Có thể thấy từ hình 20.4, mỗi đối tượng đều có một vùng tin ám chỉ mà bộ dịch xen vào khi dịch để chỉ tới bảng hàm ảo của nó. Với những đối tượng *reference_t*, con trỏ chỉ tới bảng hàm ảo *reference_t*, trong khi đối với những đối tượng *book_t*, con trỏ chỉ tới bảng hàm ảo *book_t*. Khi đối tượng *book_t* gọi hàm *print_all_cites*, địa chỉ của hàm sẽ được tra từ vùng con trỏ của đối tượng *book_t* qua bảng hàm ảo của nó. Từ bảng hàm ảo, chúng ta có thể tìm thấy con trỏ tới hàm mà chúng ta muốn kích hoạt, đó là hàm *print_all_cites* trong trường hợp này.

**Hình 20.4** Các hàm ảo

Ví dụ dưới đây trình bày cách khai báo các lớp **reference_t** và **book_t**.

```
class reference_t
{
public:
    void print_reference_cites ()
    {
        printf ("Title: %s\n", title);
        printf ("Year: %d\n", year);
    }
    virtual void print_all_cites () /* (1) */
    {
        printf_reference_cites ();
    }
private:
    author_list * author_list;
```

```

        char * title;
        int year;
        reference_t * next;
    };
    class book_t: public reference_t
    {
    public:
        virtual void print_all_cites () /* (2) */
        {
            printf_reference_cites (); /* phương thức được thừa kế từ lớp reference_t */
            printf ("Publisher: %s\n", publisher);
            printf ("ISBN: %d\n", ISBN);
        }
    private:
        char * publisher;      /* tên nhà xuất bản */
        char * address;       /* địa chỉ nhà xuất bản */
        double ISBN;          /* số ISBN */
    };

```

Đoạn mã trên định nghĩa lớp **reference_t** với hàm ảo **print_all_cites**. Lớp **book_t** là một lớp cấp dưới của lớp **reference_t**, và được chỉ định bởi “**: public reference_t**”. Từ khóa **public** theo sau là **reference_t** cho biết tất cả các thành phần **public** của lớp cấp trên **reference_t** sẽ được giữ lại là **public** trong lớp cấp dưới **book_t**. Chúng ta cũng có thể dùng từ khóa **private** trong trường hợp trên, khi đó các thành phần **public** của lớp cấp trên **reference_t** sẽ trở nên **private** trong hàm cấp dưới **book_t**. Những định nghĩa lớp này nói với bộ dịch là bảng hàm ảo cho lớp **book_t** chứa con trỏ hàm tới hàm **book_t::print_all_cites** chứ không phải tới hàm **reference_t::print_all_cites**. Từ hình 20.4, chúng ta có thể thấy bảng hàm ảo **reference_t** ở bên trái có một con trỏ tới hàm được định nghĩa bởi (1) cho **print_all_cites**, và bảng hàm ảo **book_t** ở bên phải sẽ có một con trỏ tới hàm được định nghĩa bởi (2) cho **print_all_cites**.

Chúng ta hãy xem mỗi hàm sẽ in ra cái gì khi được gọi qua khai báo gọi hàm dưới đây.

```

reference_t * r;
r->print_all_cites ();

```

Lệnh gọi hàm trên sẽ gọi hàm được định nghĩa bởi (1) và in ra *title* và *year*.

Nhưng nếu ta có khai báo và gọi hàm dưới đây.

```
book_t * b;
b->print_all_cites();
```

Lệnh gọi hàm trên sẽ gọi hàm được định nghĩa bởi (2) và in ra *title*, *year*, *publisher*, và *ISBN*.

Chúng ta có thể tự hỏi tại sao phương thức **print_reference_cites** tồn tại được. Lý do của điều này là chỉ những thành phần **public** của một lớp thì mới được kế thừa, có nghĩa là lớp cấp dưới **book_t** không thể truy xuất trực tiếp các thành phần **author_list**, **title**, **year**, và **next**. Tuy nhiên, vì phương thức **print_reference_cites** là **public**, lớp **book_t** không có quyền truy cập phương thức này.

Ngoài ra, C++ còn cung cấp từ khóa **protected** xác định khả năng truy xuất, nó cũng có tác dụng tương tự như từ khóa **private**, ngoại trừ việc nó cho phép lớp cấp dưới có thể truy xuất các thành phần được khai báo sau từ khóa **protected**.

20.7 THAM KHẢO TRONG C++

Trong C++, một tham khảo là một con trỏ mặc nhiên ám chỉ. Ví dụ:

```
int val;
int & val_ref = val;    /* Khai báo một tham khảo */
val_ref = 10;           /* tương tự như *val_ptr = 10 */
```

Tại sao dạng tham khảo này được sử dụng? Nó thường được sử dụng trong khái niệm *quá tải của toán tử* (*operator overload*). Sự quá tải của toán tử làm chúng ta thay đổi cách một toán tử cụ thể (như + - *,...) đối xử với một lớp. Ví dụ, nếu chúng ta tạo ra một lớp mới để biểu diễn một số phức gọi là **complex_t**, chúng ta có thể muốn làm các lệnh như sau:

```
complex_t x, y, z;
z = x + y;
```

Nếu chúng ta làm quá tải thao tác cộng (+), chúng ta có thể thấy một cách chính xác cách đổi xử khi cộng hai đối tượng **complex_t**. Chú ý, một số phức là số mà có hai phần thực và ảo.

Chúng ta cũng có thể khai báo hàm quá tải như sau:

```
complex_t operator+(const complex_t & a, const complex_t & b);
```

Toán tử tham khảo "&" bắt bộ dịch chuyển qua dạng con trỏ để giá trị trong biến có thể được hàm quá tải hiệu chỉnh. Tuy nhiên, trong ví dụ trên chúng ta không cần phải thay đổi trị của hai đối số (*a* và *b*) để thực hiện thao tác cộng, vì vậy chúng ta dùng từ khóa **const** để tránh việc hàm có thể làm thay đổi trị của các đối số này. Đây là việc mà lập trình viên phải hết sức lưu ý, vì nếu không có thể dẫn tới việc trị của biến bị thay đổi gây sự khó hiểu cho chương trình. Một thuận lợi của đối số truyền có tham khảo **const** là đỡ mất thời gian truy xuất stack cho các đối tượng lớn, vì thời gian thêm vào cần để truy xuất gián tiếp giá trị thông qua con trỏ có thể lớn hơn việc chép các giá trị vào stack nếu gặp các đối tượng nhỏ. Mặt khác, kích thước thật của một đối tượng nhiều khi rất khó xác định, nên cách tốt nhất là ta truyền đối số bằng tham khảo **const**.

20.8 MỘT SỐ ĐIỂM KHÁC BIỆT CHÍNH GIỮA C VÀ C++

C++ phát triển dựa vào C, nó giữ lại rất nhiều khái niệm hàm mà C đã có. Tuy nhiên khi viết chương trình nguồn C++, ta thấy một số khác biệt cơ bản không tương thích hoàn toàn với C. Phần dưới đây sẽ trình bày một số điểm lưu ý dành cho lập trình viên C++ viết chương trình C, hoặc cho lập trình viên C viết chương trình dịch trên bộ dịch C++.

20.8.1 Phép gán ngầm từ con trỏ void*

Ta không thể gán ngầm một con trỏ thuộc kiểu **void*** trong C++ cho bất kỳ biến thuộc kiểu nào khác.

Chẳng hạn, ví dụ sau hoàn toàn hợp lệ trong C:

```
int *x = malloc(sizeof(int) * 10);
```

nhưng sẽ không hợp lệ trong C++ (bạn đọc chú ý prototype của hàm **malloc** trong chương 9) vì có sự không an toàn về kiểu khi gán. Giả sử việc gán trên được C++ chấp nhận, chúng ta sẽ có một con trỏ **void*** là địa chỉ của đối tượng không thuộc kiểu nào cả, tức khói nhớ xin bằng **malloc**, được gán vào một biến con trỏ thuộc kiểu khác, tức kiểu **int ***, và C++ sẽ không biết việc quản lý mặc nhiên là từng 4 byte (hoặc 2 byte) cho một **int** như C. Điều này sẽ gây ra lỗi khi sử dụng biến.

Xét ví dụ sau:

```
int an_int;
void *void_pointer = &an_int;
double *double_ptr = void_pointer;
*double_ptr = 5;
```

Khi ta gán ***double_ptr** giá trị 5, nó sẽ sử dụng 8 byte bộ nhớ để lưu trữ này, nhưng biến nguyên **an_int** chỉ chiếm 4 byte. Điều này có thể dẫn tới sai lầm mà lập trình viên cần chú ý.

20.8.2 Xin và giải phóng biến động

Trong C, chỉ có một hàm chính xin bộ nhớ động là **malloc** (hàm **calloc** cũng tương tự như **malloc**). Chúng ta dùng nó để định vị bộ nhớ cho cả biến đơn và mảng như ví dụ sau:

```
int *x = malloc( sizeof(int) );
int *x_array = malloc( sizeof(int) * 10 );
```

và ta luôn giải phóng bộ nhớ theo cách như sau:

```
free( x );
free( x_array );
```

Tuy nhiên, trong C++, việc định vị bộ nhớ động cho biến mảng thì hơi khác so với phần tử đơn; ta sử dụng toán tử **new[]** để định vị và dùng toán tử **delete[]** để xóa nó.

Ví dụ 20.1

```
int *x = new int;
int *x_array = new int[10];
delete x;
delete[] x_array;
```

Chú ý là khi ta dùng toán tử **new[]** để xin mảng hay phần tử đơn thì nó sẽ khởi động *constructor* định vị cho từng phần tử mảng, nên toán tử **delete[]** phải được sử dụng để xóa, khi đó nó sẽ gọi *destructor* thích hợp hủy từng phần tử của mảng.

Việc sử dụng hàm **malloc**, cũng như **calloc**, trong C++ là hoàn toàn có thể, tuy nhiên chúng không được khuyến khích do đặc tính không sử dụng khái niệm đối tượng của chúng.

20.8.3 Khai báo hàm trước khi sử dụng

Với C++, chúng ta phải khai báo hàm ở dạng prototype trước khi sử dụng. Với C, việc khai báo prototype không phải là bắt buộc, dù được khuyến khích sử dụng để tránh xảy ra những lỗi luận lý đáng tiếc. Ví dụ chương trình sau đối với C là hợp lệ, nhưng với C++ lại không được.

Ví dụ 20.2

```
#include <stdio.h>

int main()
{
    foo();
    return 0;
}

int foo()
{
    printf( "Hello world" );
}
```

Thay vì vậy, chúng ta phải thêm prototype của hàm trước khi sử dụng để bộ dịch C++ không báo lỗi, cụ thể như sau:

Ví dụ 20.3

```
#include <stdio.h>
int foo (void);
int main()
{
    foo();
    return 0;
}
int foo(void)
{
    printf( "Hello world" );
}
```

20.8.4 Struct và Enum

Khi khai báo kiểu struct, cả C và C++ đều sử dụng từ khoá **struct**, tuy nhiên khi khai báo biến với kiểu struct đã có thì C++ không cần dùng lại từ khoá **struct** như C. Như ví dụ sau:

Ví dụ 20.4

```
struct a_struct
{
    int x;
};
a_struct struct_instance;
```

Sau khai báo biến, ta có biến struct mới là *struct_instance*. Còn trong C, chúng ta phải sử dụng lại từ khoá struct để khai báo biến như sau:

```
struct a_struct struct_instance;
```

Tương tự như vậy với kiểu **enum**, trong C chúng ta phải bao gồm từ khoá **enum** khi khai báo kiểu và biến; còn trong C++, chúng ta chỉ dùng từ khoá **enum** cho khai báo kiểu, còn khi khai báo biến thì không cần.

Do đó để tạo sự đơn giản cho khai báo biến, người viết C thường khai báo kiểu kết hợp với từ khoá **typedef** để xác định tên cho kiểu **struct** hay **enum** cần làm việc.

Ví dụ 20.5

```
typedef struct struct_name
{
    /* variables */
} struct_name_t;
// Khai báo biến struct theo kiểu mới
struct_name_t struct_name_t_instance;
```

Tuy nhiên, khi viết C++ chúng ta vẫn phải dùng cú pháp “`struct struct_name`” để khai báo một thành phần của struct là một pointer chỉ tới struct đó.

Tương tự như trong C, C++ cũng sử dụng các danh hiệu đã xác định cho các khai báo như ví dụ sau:

Ví dụ 20.6

```
typedef struct struct_name
{
    struct struct_name instance;
    struct_name_t instance2; /* invalid! The typedef isn't defined yet */
} struct_name_t;
```

Trong ví dụ trên, khai báo kiểu kết hợp khai báo tên kiểu, tuy nhiên tên kiểu chưa được xác định, nên không thể được sử dụng để khai báo biến thành phần trong nó.

20.8.5 C++ có một thư viện lớn hơn C

Thư viện của C++ lớn hơn của C nhiều, nhiều thứ khi viết với C ta không liên kết được hay liên kết rất khó khăn với thư viện, thì với C++ ta có thể liên kết chúng với thư viện này rất dễ dàng. Chẳng hạn, nếu bạn đã từng dùng g++ cho các chương trình tính toán khi viết bằng C++, bạn sẽ thấy việc dịch liên kết thư viện khá đơn giản, trong khi với C, khi dùng gcc cho chương trình C, ta cần phải xác định rõ thư viện liên kết cho các hàm mà ta sử dụng như `sin` hay `sqrt`:

Với C++

```
% g++ foo.cc
```

hoặc với C

```
% gcc foo.c -lm
```

Hai dòng ví dụ trên diễn tả cách dịch chương trình có liên kết với thư viện C++ hay C trên hệ điều hành *nix (như Unix chẳng hạn) với các tập tin C++ (.cc) hoặc (.c).

Ngoài ra C++ cho phép sử dụng kiểu luận lý (*boolean type*) mà trong C không có, ta có thể mô phỏng trong C bằng từ khóa **typedef** để có kiểu luận lý bằng cách dùng kiểu enum như sau:

```
typedef enum {FALSE, TRUE} bool;
```

Trong C++, hàm **main()** mặc nhiên trả về trị 0, nên ta có thể không cần dòng lệnh 'return 0' cuối hàm **main()** mà vẫn có trị này được; với C, nếu muốn xác định hàm **main()** trả về một trị nào đó, như trị 0 chẳng hạn, ta phải để tường minh dòng 'return 0' cuối hàm **main()**, nếu không hàm **main()** sẽ trả về trị không xác định.

Ví dụ 20.7

Trong C++, ta viết

```
int main()
{
    printf( "Hello, World" );
}
```

Nhưng trong C, ta phải viết

```
int main()
{
    printf( "Hello, World" );
    return 0;
}
```

thì các hàm **main()** mới trả về trị 0, trị báo với đệ điều hành là chương trình kết thúc bình thường.

20.9 MỘT SỐ CHƯƠNG TRÌNH VÍ DỤ

Các chương trình sau đây đều được viết và dịch bằng bộ dịch Borland C++ 5.0x. Quan điểm giải thích các chương trình sau là độc giả đã phải trải qua quá trình học C, nên việc giải thích là ngắn gọn và không cần quá chi tiết.

20.9.1 Chương trình 1

Chương trình nhập số hiệu và tên sinh viên, sau đó in ra các thông tin này.

Các trị số xuất hiện đầu mỗi dòng là số hiệu dòng tương ứng với lệnh trên dòng đó. Nếu lệnh ở dòng 6 gây xuất liệu ra màn hình thì dòng xuất liệu trên màn hình tương ứng có số hiệu 6 ở phía trước, biểu thị lệnh dòng 6 gây ra hiệu ứng trên màn hình. Quy ước này được sử dụng cho cả các chương trình còn lại.

```

1 #include <iostream.h> // similar to stdio.h
2 void main() // no args, return no value
3 {
4     int sno;
5     char sname[80];
6     cout << "Please enter student number: "; // print prompt
7     cin >> sno; // input value from stdin
8     cout << "Please enter a student name: ";
9     cin >> sname;
10    cout << "Student number = " << sno
11        << "\nStudent name = " << sname << endl;
12 }
```

Chương trình cho xuất liệu ví dụ như sau:

```

6 Please enter student number: 1001
8 Please enter a student name: Vinh
10 Student number = 1001 Student name = Vinh
```

Chương trình trên cho phép nhập trị vào biến nguyên và mảng qua việc sử dụng đối tượng `cin` và toán tử `>>` theo sau là biến cần

nhập trị từ bàn phím (dòng nhập chuẩn) mà không cần cung cấp cụ thể địa chỉ biến. Tương tự, đối tượng **cout** là dòng xuất chuẩn, xuất dữ liệu ra màn hình có kết hợp với toán tử `<<`. Các toán tử `>>`, `<<` được gọi chung là các chuyển dòng xuất/nhập liệu. Các đối tượng **cin** và **cout** đều được khai báo trong **iostream.h**, nên cần bao hàm nó vào chương trình.

Chú ý chương trình trên sẽ không dừng màn hình để ta xem kết quả, muốn vậy ta cần phải thêm dòng **cin.get()**; trước khi kết thúc hàm **main()**. Tác dụng của nó hoàn toàn tương tự như **getchar()** trong C.

20.9.2 Chương trình 2

Chương trình sau sẽ kiểm tra xem tới weekend chưa, và còn bao nhiêu ngày nữa.

```

1 #include <iostream.h>
2 class week {
3     int day; // duong than cong : com
4     char *days[7];
5 public:
6     week(int); // class constructor: declaration only
7     void today() { cout << days[day] << endl; }
8     void weekend(); // function prototype
9 };
// class week
10 week::week(int d)
11 {
12     day = d; // init day to d
13     days[0] = "Monday"; // init array element
14     days[1] = "Tuesday";
15     days[2] = "Wednesday";
16     days[3] = "Thursday";
17     days[4] = "Friday";
18     days[5] = "Saturday";
19     days[6] = "Sunday";
20 } // class week constructor

```

```

21 void week::weekend()           // weekend yet?
22 {
23     if (day > 4)
24         cout << "Yes! Today is weekend.\n";
25     else
26         cout << "Not yet! Please wait " << 5-day << " days.\n";
27 }                                // function weekend
28 void main()
29 {
30     week day(1);                // declare and init
31     week *pday = new week(5);   // declare, allocate, and init
32     day.today();                // execute today() – what day is it?
33     day.weekend();              // execute weekend() – weekend yet?
34     pday->today();             // execute today() – pointer version
35     pday->weekend();            // execute weekend() – pointer version
36     delete pday;                // free space allocated for pday
37 }

```

Chương trình cho xuất liệu ví dụ như sau:

```

32 Tuesday
33 Not yet! Please wait 4 days.
34 Saturday
35 Yes! Today is weekend.

```

Dòng 30 trong hàm main() khai báo một đối tượng tên *day* kèm theo trị khởi động là 1 cho biến *day* trong nó, tức tương ứng với thứ ba (Tuesday). Dòng 31 khai báo một biến con trả *pday* có kiểu *week**, địa chỉ của lớp gán cho nó là một biến động xin bằng toán tử *new* có khởi động trị qua *constructor* của nó. Trị này sẽ được *constructor* tự động gán cho biến *day*.

Các dòng 32, 33 và 34, 35 thực thi các phương thức trong lớp theo cách gọi trực tiếp đối tượng *day*, hay gọi qua con trả tới đối tượng cùng kiểu.

20.9.3 Chương trình 3

Chương trình này sẽ nhập vào một số chuỗi ký tự, kết thúc việc nhập loạt chuỗi bằng ký tự kết thúc file (EOF), tức nhấn cặp phím Ctrl -Z.

```

1 #include <iostream.h>
2 int x;                                // global x
3 void hide()
4 {
5     int x;                            // local x hides global x
6     x = 10;                           // assign 10 to local x
7     {
8         int x;                      // hides first local x
9         x = 100;                     // assign 100 to second local x
10        cout << "x = " << x << endl;
11    }
12    cout << "x = " << x << endl;
13    cout << "x = " << ::x << endl; // global x
14 }                                // function hide
15 main()
16 {
17     x = 1;
18     hide();
19 }
```

Chương trình cho xuất liệu ví dụ là

```

10 x = 100
12 x = 10
13 x = 1
```

Trong chương trình trên, ba biến *x* được khai báo ở ba chỗ khác nhau, biến *x* toàn cục được sử dụng trong hàm *hide()* nhờ toán tử quy định tầm (*scope resolution operator*) “::”.

20.9.4 Chương trình 4

Chương trình sau đây nhập vào một số chuỗi, sắp xếp lại và in ra màn kết quả.

```

1 #include <iostream.h>
2 #include <string.h>
3 class string {
4     char *str;
5 public:
6     string() { // user-defined default constructor
7         str = new char[1]; // allocate one char space
8         *str = '\0'; // make it NULL
9     }
10    string(const char *); // user-defined constructor
11    void print() {
12        cout << str << endl; // print contents in str
13    }
14    friend int operator < (const string &s1, const string &s2) {
15        return strcmp(s1.str, s2.str) < 0;
16    }
17 }; // class string
18 string::string(const char *s)
19 {
20    if (s) { // if s is not NULL
21        str = new char[strlen(s) + 1]; // allocate space for str
22        strcpy(str, s); // copy content of s to str
23    } else {
24        str = new char[1];
25        *str = '\0';
26    }
27 } // user-defined constructor
28 void input(string *a, int limit, int &i) // a regular function
29 {
30    char buffer[100];
31    cout << "Enter strings (End with ^Z)\n";
32    for (i = 0; i < limit; i++)

```

```

33     if (!cin.eof()) {           // if not EOF
34         cin.getline(buffer, 100); // get one line of char's
35         a[i] = string(buffer);   // a[i] = (string)buffer
36     } else
37         break;
38 }
39 void output (string *a, int size) // regular function
40 {
41     cout << endl << "sorted list:\n";
42     for (int i = 0; i < size; i++)
43         a[i].print();
44     cout << endl;
45 }
46 void sort(string *a, int n)      // regular function
47 {
48     int changed;
49     do {
50         changed = 0;
51         for (int i = 0; i < n-1; i++)
52             if (a[i+1] < a[i]) {
53                 string temp = a[i];
54                 a[i] = a[i+1];
55                 a[i+1] = temp;
56                 changed = 1;
57             }
58     } while (changed);
59 }                                // bubble sort with flag
60 const int max = 10;               // named constant
61 static string list[max];        // file scope
62 static int size = 0;              // file scope
63 void main()
64 {
65     input(list, max, size);       // get input
66     sort(list, size);            // sort it
67     output(list, size);          // print it
68 }

```

Chương trình cho xuất liệu ví dụ như sau:

Department of Computer Science & Engineering

College of Engineering

Florida Atlantic University

777 Glades Road

Boca Raton, FL 33431^Z

sorted list:

777 Glades Road

Boca Raton, FL 33431

College of Engineering

Department of Computer Science & Engineering

Florida Atlantic University

Chương trình trên khai báo *class string* trong đó constructor mặc nhiên *string()* khởi động trị cho biến con trỏ *str* của class, constructor này sẽ được gọi tự động để khởi tạo trị cho mảng *list* ngoài hàm *main()*. Từ khoá **public** xác định mức độ truy xuất các thành phần sau nó. Phương thức *string(const char*)* là constructor xác định khả năng thay đổi trị của đối tượng *string*, nó khác với constructor mặc nhiên ở chỗ constructor mặc nhiên được gọi tự động khi chạy chương trình, nó thường dùng để khởi động trị cho biến, hay định vị bộ nhớ động khi biến chưa có trị xác định sau khai báo, hoặc biến con trỏ chưa chỉ tới bộ nhớ động cụ thể..., còn constructor do người dùng khởi tạo có thể được gọi tường minh khi chạy chương trình.

Dòng 14 khai báo toán tử “<” để so sánh hai chuỗi được cung cấp ở dạng đối số ngay sau. Từ khoá **friend** xác định hàm được khai báo có khả năng truy xuất các thành phần **private** và **protected** trong class. Chú ý, dòng 14 đã kết hợp sử dụng khai báo hàm và định nghĩa trong class, vì ngoài class này ra, hàm *operator < ()* không còn được sử dụng ở nơi nào khác trong chương trình.

20.9.5 Chương trình 5

Chương trình sau đây trình bày khái niệm về bộ tạo chép, copy constructor, việc khai báo và sử dụng constructor dạng này rất có lợi khi truyền đối số hay trả về trị từ hàm là đối tượng, hoặc khởi động trị cho đối tượng khác cùng kiểu.

```

1 #include <iostream.h>
2 #include <string.h>
3 class member {
4     char *str;
5 public:
6     member() { str=0; cout <<"Default constructor called\n"; }
7     member(const member& m) // copy constructor
8     {
9         str = new char[strlen(m.str)+1];
10        strcpy(str, m.str);
11        cout <<"\t\tCopy constructor called: str = "<< str<<endl;
12    }
13    member(char *s) // another constructor
14    {
15        str = new char[strlen(s)+1];
16        strcpy(str, s);
17        cout << "\tConstructor called: str = " << str << endl;
18    }
19 }; // class member
20 member pass(member m)
21 {
22     return m;
23 }
24 void main()
25 {
26     member nation("Clinton"), neighbour = nation, home;
27     home = pass(nation);
28 }

```

Chương trình cho xuất liệu ví dụ như sau:

```

26 Constructor called: str = Clinton
26     Copy constructor called: str = Clinton
26 Default constructor called
27     Copy constructor called: str = Clinton
27     Copy constructor called: str = Clinton

```

Cấu trúc tổng quát của một bộ tạo chép như sau:

```
class_name(const class_name &obj) {
    // body of constructor
}
```

Việc gọi bộ tạo chép có thể là tường minh qua lệnh (dòng 27 trong chương trình trên) hay tự động qua việc khởi động trị cho đối tượng mới (dòng 26, gán biến trong chương trình).

20.9.6 Chương trình 6

Chương trình khảo sát các bộ chép đối tượng, *copy constructor*, và hủy đối tượng, *destructor*. Việc sử dụng kết hợp các bộ chép và hủy cần phải thận trọng vì quy định của C++ khi hủy cho các đối tượng cùng tầm truy xuất.

```
1 #include <iostream.h>
2 #include <string.h>
3 class member {
4     char *str;
5 public:
6     member(): str(0) { cout <<"Default constructor called\n"; }
7     member(const member& m) // copy constructor
8     {
9         str = new char[strlen(m.str)+1];
10        strcpy(str, m.str);
11        cout <<"\nCopy constructor called: str = " << str << endl;
12    }
13    member(char *s) // another constructor
14    {
15        str = new char[strlen(s)+1];
16        strcpy(str, s);
17        cout << "\nConstructor called: str = " << str << endl;
18    }
19    ~member()
20    {
21        cout << Default destructor: str = " << (void *)str << endl;
22        delete [] str; // to delete an array
```

```

23 }
24 }; // class member
25 member pass(member m)
26 {
27   return m;
28 }
29 void main()
30 {
31   member nation("Clinton"), neighbour = nation, home;
32   home = pass(nation);
33   home = neighbour;
34 }

```

Chương trình cho xuất liệu ví dụ như sau:

```

31 Constructor called: str = Clinton
31 Copy constructor called: str = Clinton
31 Default constructor called
32 Copy constructor called: str = Clinton
32 Copy constructor called: str = Clinton
32 Default destructor: str = 0x00873034
32 Default destructor: str = 0x00873044
33 Default destructor: str = 0x00873024
34 Default destructor: str = 0x00873024
35 Default destructor: str = 0x00873014

```

Chương trình cho phép kết hợp bộ hủy và chép đối tượng. Chú ý dòng lệnh 32, tức việc gọi hàm và gán trị, tạo ra 4 dòng xuất liệu có ý nghĩa tuần tự như sau:

- gọi bộ chép đối tượng để gán trị cho đối tượng *m* cục bộ từ đối tượng *nation* là đối số hàm.
- trả về đối tượng *m* cho hàm qua một đối tượng “ẩn”.
- hủy đối tượng *m*
- hủy đối tượng “ẩn” sau khi gán nó vào cho *home* trong hàm *main()*.

Nếu ta bỏ phương thức chép đối tượng *member*, tức từ dòng 7 tới dòng 12 như dưới đây

```

7 // member(const member& m)
8 {
9 //   str = new char[strlen(m.str)+1];
10 //  strcpy(str, m.str);
11 //  cout <<"InitCopy constructor called: str = " <<str<<endl;
12 }

```

Thì chương trình sẽ cho xuất liệu như sau:

```

31 Constructor called: str = Clinton
31 Default constructor called
32 Default destructor: str = 0x00873020
32 Default destructor: str = 0x00873020
33 Default destructor: str = 0x00873020
33 Default destructor: str = 0x00873020
33 Default destructor: str = 0x00873020

```

Xin quý độc giả tự giải thích kết quả xuất liệu này.

20.9.7 Chương trình 7

Chương trình ví dụ sau đây minh họa cho việc sử dụng khái niệm thừa kế, *Inheritance*.

```

1 #include <iostream.h>
2 class base_class {
3 protected:
4     int num;
5     int val;
6 public:
7     base_class() { num = val = 1; }
8     void display() { cout << "num = " << num << endl; }
9 }; // class base_class
10 class derived_class: public base_class {
11 protected:
12     int num;
13 public:
14     derived_class() { num = val = 10; }

```

```

15 void display() { cout << "\tnum = " << num << endl;}
16 void show() { cout << "\tval = " << val << endl;}
17 }; // class derived_class
18 class further_derived: public derived_class {
19     int num;
20 public:
21     further_derived() { num = val = 100; }
22     void display() { cout << "\tnum = " << num << endl;}
23     void show() { cout << "\tval = " << val << endl;}
24 }; // class further_derived
25 main()
26 {
27     base_class bc;
28     derived_class dc;
29     further_derived fdc;
30     bc.display();
31     dc.display();
32     fdc.display();
33     dc.base_class::display();
34     fdc.derived_class::display();
35     fdc.base_class::display();
36     dc.show();
37     fdc.show();
38 }

```

Chương trình cho xuất liệu là:

```

30 num = 1
31      num = 10
32          num = 100
33      num = 1
34          num = 10
35      num = 1
36          val = 10
37          val = 100

```

Khái niệm thừa kế trong lập trình hướng đối tượng cũng tương tự như trong thực tế gia đình của chúng ta, tức con cái, các lớp được khai báo ở dòng 10, 18 sẽ thừa kế những dữ liệu hay hàm *không private* được khai báo trong lớp cha mẹ (đối với lớp ở dòng 10), ông bà (đối với ở dòng 18) ở dòng 2. Chú ý là dữ liệu, hàm *private* trong lớp được thừa kế, những *constructor*, *destructor*, cũng như các *friend function*, các thành phần *static*, và phép gán sẽ không được thừa kế, vì đó là những “quan hệ” riêng, hay những “của cải” riêng của lớp cấp trên đó, và như vậy các lớp cấp dưới chỉ được kế thừa “những cái” được cho phép.

20.9.8 Chương trình 8

Chương trình sau minh họa cho việc một lớp có thể thừa kế từ nhiều lớp cấp cao hơn, ta gọi là đa thừa kế.

```

1 #include <iostream.h>
2 #include <string.h>
3 class employee {
4     char *name;
5 protected:
6     int id;
7 public:
8     employee(char *, int);
9     ~employee() { delete [] name; }
10    void display() { cout << name << " " << id << endl; }
11    const char * get_name() { return name; }
12    int get_id() { return id; }
13 }; // class employee
14 class family_data {
15 protected:
16     char *spouse;
17     char *address;
18 public:
19     family_data(char *, char *);
20     ~family_data() { delete [] spouse; delete [] address; }
21     void display() {cout << spouse << " " << address << endl; }
```

```

22 const char *get_spouse() { return spouse; }
23 const char *get_address() { return address; }
24 }; // class family_data
25 employee::employee(char *nm, int ident)
26 {
27   id = ident;
28   name = new char[strlen(nm)+1];
29   strcpy(name, nm);
30 } // employee constructor
31 family_data::family_data(char *sp, char *ad)
32 {
33   spouse = new char[strlen(sp)+1];
34   strcpy(spouse, sp);
35   address = new char[strlen(ad)+1];
36   strcpy(address, ad);
37 } // family_data constructor
38 class record: public employee, public family_data {
39   char *occupation;
40 public:
41   record(char *, char *, char *, char *, int);
42   ~record() { delete [] occupation; }
43   void display() { cout << get_name() << " " << id << " "
44           << occupation << " " << spouse
45           << " " << address << endl; }
46   const char *get_occupation() { return occupation; }
47 }; // class record
48 record::record(char *nm, char *spouse, char *address,
49   char *oc, int ident): employee(nm, ident),
50   family_data(spouse, address)
51 {
52   occupation = new char[strlen(oc)+1];
53   strcpy(occupation, oc);
54 } // record constructor

```

```
51 main()
52 {
53     int id;
54     const char *name, *spouse, *address, *occupation;
55     employee fred("Fred Flinchstone", 9900);
56     family_data lisa("Lisa Minie", "Boca Raton");
57     record entry("Fred", "Lisa", "FAU", "Education", 40);
58     fred.display();
59     lisa.display();
60     entry.display();
61     id = fred.get_id();
62     cout << id << endl;
63     id = entry.get_id();
64     cout << id << endl;
65     spouse = lisa.get_spouse();
66     cout << spouse << endl;
67     spouse = entry.get_spouse();
68     cout << spouse << endl;
69     address = lisa.get_address();
70     cout << address << endl;
71     address = entry.get_address();
72     cout << address << endl;
73 }
```

Chương trình cho xuất liệu như sau:

```
58 Fred Flinchstone 9900
59 Lisa Minie Boca Raton
60 Fred 40 Education Lisa FAU
62 9900
64 40
66 Lisa Minie
68 Lisa
70 Boca Raton
72 FAU
```

Chú ý dòng 46 chỉ ra những dữ liệu trong sự thừa kế từ những lớp cấp trên. Chúng sẽ được sử dụng trong các phương thức thừa kế của lớp kế thừa những lớp cấp trên. Cụ thể các dòng 61, 63, hay 65, 67, hoặc 69, 71 trình bày rõ việc này.

20.9.9 Chương trình 9

Chương trình ví dụ này trình bày việc khai báo và sử dụng hàm ảo, virtual.

```

1 #include <iostream.h>
2 #include <string.h>
3 class drawing {
4 protected:
5     char *graphic;
6 public:
7     drawing(char *);
8     virtual ~drawing() { delete [] graphic; }
9     virtual void draw_graphic() { cout << "draw...\n" << graphic; }
10 }; // class drawing
11 drawing::drawing(char *shape)
12 {
13     graphic = new char[strlen(shape)+1];
14     strcpy(graphic, shape);
15 } // drawing constructor
16 class sphere: public drawing {
17     double radius;
18 public:
19     sphere(char *, double);
20     void draw_graphic();
21 }; // class sphere
22 sphere::sphere(char * object, double dimension)
23     : drawing(object)
24 {
25     radius = dimension;
26 } // sphere constructor
27 void sphere::draw_graphic()
28 {
29 }
```

```
28     cout << "sphere: radius = " << radius << endl;
29 } // sphere::draw_graphic
30 class square: public drawing {
31     double side;
32 public:
33     square(char *, double);
34     void draw_graphic();
35 }; // class square
36 square::square(char * object, double dimension)
    : drawing(object)
37 {
38     side = dimension;
39 } // square constructor
40 void square::draw_graphic()
41 {
42     cout << "square: side = " << side << endl;
43 } // sphere::draw_graphic
44 sphere earth("Earth", 23.34), moon("Moon", 12.1);
45 square room("Bedroom", 100), den("Den", 50);
46 main()
47 {
48     drawing exam("Drawing... !!!\n");
49     exam.draw_graphic();
50     earth.draw_graphic();
51     moon.draw_graphic();
52     room.draw_graphic();
53     den.draw_graphic();
54 }
```

Chương trình cho xuất liệu như sau:

```
49 draw...
49 Drawing... !!!
50 sphere: radius = 23.34
51 sphere: radius = 12.1
52 square: side = 100
53 square: side = 50
```

Trong chương trình trên, dòng 9 khai báo hàm ảo trong khai báo lớp, do đó *destructor* trong lớp đó bắt buộc phải được khai báo ảo. Hàm này có thể được định nghĩa trả lại trong các lớp con mà không quan tâm tới hàm có tên được kế thừa từ hàm cấp trên đã có. Cần lưu ý là khai báo ảo chỉ sử dụng cho hàm của các lớp có sự kế thừa, sự khác nhau của từng hàm ảo của các lớp kế thừa sẽ được ghi nhận và sử dụng cho từng lớp cấp dưới mà thôi. Hai chương trình sau đây độc giả tự giải thích để thấy rõ hơn vấn đề.

Chương trình 9.1:

```

1 #include <iostream.h>
2 class North {
3 public:
4     virtual void direction() { cout << "North\n"; }
5 };
6 class East: public North{
7 public:
8     virtual void direction() { cout << "East\n"; }
9 };
10 int main()
11 {
12     North north;
13     East east;
14     North *dir_ary[2];
15     dir_ary[0] = &north;
16     dir_ary[1] = &east;
17     for (int i = 0; i < 2; i++)
18         dir_ary[i]->direction();
19     return 0;
20 }
```

Chương trình cho xuất liệu là:

```

18 North
18 East
```

Chương trình 9.2:

```

1 #include <iostream.h>
2 class North {
3 public:
4     void direction() { cout << "North\n"; }
5 };
6 class East: public North{
7 public:
8     void direction() { cout << "East\n"; }
9 };
10 int main()
11 {
12     North north;
13     East east;
14     North *dir_ary[2];
15     dir_ary[0] = &north;
16     dir_ary[1] = &east;
17     for (int i = 0; i < 2; i++)
18         dir_ary[i]->direction();
19     return 0;
20 }
```

Chương trình cho xuất liệu là:

18 North

18 North

20.9.10 Chương trình 10

C++ cung cấp khái niệm quá khả năng, *overloading*, khái niệm này nhằm đưa ra việc chọn dùng hàm tùy hoàn cảnh các đối số cung cấp. Các hàm thường cùng tên, cùng số lượng đối số, cùng kiểu dữ liệu trả về. Tùy vào đối số khi gọi hàm, mà hàm nào sẽ được gọi. Khái niệm cùng tên gọi (hàm) cho từng loại dữ liệu tương tự như mảng con trả các hàm trong chương “Kiểu con trả” đã đề cập, tuy nhiên việc sử dụng rất phiền phức, nhưng với C++, vấn đề này trở nên chuẩn và được sử dụng rất dễ dàng.

```

1 #include <iostream.h>
2 #include <string.h>
3 void swap(int &a1, int &a2)
4 {
5     int temp = a1;
6     a1 = a2; a2 = temp; // 2 instr's on 1 line to save space
7 } // swap integers           // not good programming style
8 void swap(float &a1, float &a2) // function overloaded
9 {
10    float temp = a1;
11    a1 = a2; a2 = temp;
12 } // swap floats
13 void swap(char **a1, char **a2) // function overloaded
14 {
15    char *temp = *a1;
16    *a1 = *a2; *a2 = temp;
17 } // swap strings
18 main()
19 {
20    int i = 5, j = 10;
21    swap(i, j); cout << i << ":" << j << endl;
22    float a = 5, b = 10;
23    swap(a, b); cout << a << ":" << b << endl;
24    char *s = "5", *t= "10";
25    swap(&s, &t); cout << s << ":" << t << endl;
26 }

```

Chương trình cho xuất liệu như sau:

```

21 10:5
23 10:5
25 10:5

```

Trong chương trình trên, ba hàm **swap** cùng chức năng hoán đổi trị nhưng có tới ba kiểu dữ liệu khác nhau có thể được cung cấp trong đối số.

Phụ lục**Phụ lục A****KIẾN TRÚC TẬP LỆNH LC-3****Một số quy ước**

Ký hiệu	Ý nghĩa
xSố	Số trong hệ thập lục phân
#Số	Số trong hệ thập phân, có thể âm, dương hoặc bằng 0
A[i:r]	Vùng bit cần sử dụng từ bit[i] tới bit[r]
BaseR	Thanh ghi nền, một trong các thanh ghi R0..R7, kết hợp với sáu bit offset trong kiểu định vị nền
DR	Thanh ghi đích, một trong các thanh ghi R0..R7
imm5	Trị 5 bit tức thời ở dạng bù 2
LABEL	Nhãn để thành lập cấu trúc lệnh hợp ngữ
mem[address]	Nội dung của ô nhớ có địa chỉ address
offset6	Giá trị 6 bit, là các bit[5:0] trong một lệnh, ở dạng bù 2
PC	Thanh ghi đếm chương trình PC, dài 16 bit, chứa địa chỉ lệnh kế tiếp cần thực thi
PCoffset9	Giá trị 9 bit, là các bit[8:0] trong một lệnh, ở dạng bù 2, được dùng làm offset cho thanh ghi PC
PCoffset11	Giá trị 11 bit, là các bit[10:0] trong một lệnh, ở dạng bù 2, được dùng làm offset cho thanh ghi PC
PSR	Thanh ghi trạng thái của bộ xử lý, dài 16 bit, chứa thông tin của quá trình đang được xử lý trong PSR[2:0], PSR[2] = N, PSR[1] = Z, PSR[0] = P
setcc()	Đặt các mã điều kiện N, Z, P dựa vào kết quả được ghi vào DR
SEXT(A)	Sign-extend A. Bit trọng số lớn nhất của A được lặp lại nhiều lần để có trị A 16 bit
SP	Con trỏ stack hiện hành, là thanh ghi R6
SR,SR1,SR2	Thanh ghi nguồn, là một trong các thanh ghi R0..R7
SSP	Supervisor Stack Pointer
trapvect8	Giá trị 8 bit, là các bit[7:0] trong lệnh, được dùng trong lệnh TRAP để xác định địa chỉ bắt đầu của thủ tục phục vụ trap. Trị này được lấy ở dạng số nguyên không dấu và mở rộng zero để có 16 bit
USP	User Stack Pointer
ZEXT(A)	Zero-extend A. Bit 0 được lặp lại để có trị A 16 bit

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD*	0	0	0	1		DR		SR1		0	0	0		SR2		
ADD*	0	0	0	1		DR		SR1		1				imm5		
AND*	0	1	0	1		DR		SR1		0	0	0		SR2		
AND*	0	1	0	1		DR		SR1		1				imm5		
BR	0	0	0	0	n	z	p							PCoffset9		
JMP	1	1	0	0	0	0	0	BaseR		0	0	0	0	0	0	0
JSR	0	1	0	0	1		P	C	o	f	f	s	e	1	1	1
JSRR	0	1	0	0	0	0	BaseR		0	0	0	0	0	0	0	0
LD*	0	0	1	0		DR								PCoffset9		
LDI*	1	0	1	0		DR								PCoffset9		
LDR*	0	1	1	0		DR		BaseR						offset6		
LEA*	1	1	1	0		DR								PCoffset9		
NOT*	1	0	0	1		DR		SR		1	1	1	1	1	1	1
RET	1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0
RTI	1	0	0	0		0	0	0	0	0	0	0	0	0	0	0
ST	0	0	1	1		SR								PCoffset9		
STI	1	0	1	1		SR								PCoffset9		
STR	0	1	1	1		SR		BaseR						offset6		
TRAP	1	1	1	1		0	0	0	0					trapvect8		
mở rộng	1	1	0	1												

1- Lệnh cộng ADD*Dạng thức hợp ngữ*

ADD DR, SR1, SR2

ADD DR, SR1, imm5

Dạng nhị phân

15	12	11	9	8	6	5	4	3	2	0
0	0	0	1	DR	SR1	0	0	0	SR2	

15	12	11	9	8	6	5	4	0
0	0	0	1	DR	SR1	1	imm5	

Công dụng

If (bit[5] == 0)

DR = SR1+SR2

Else

DR = SR1+ SEXT (imm5);

Setcc();

*Ví dụ*ADD R2, R3, R4; R2 \leftarrow R3+R4ADD R2, R3, #7; R2 \leftarrow R3+7**2- Lệnh luận lý theo bit AND***Dạng thức hợp ngữ*

AND DR, SR1, SR2

AND DR, SR1, imm5

Dạng nhị phân

15	12	11	9	8	6	5	4	3	2	0
0	1	0	1	DR	SR1	0	0	0	SR2	

15	12	11	9	8	6	5	4	0
0	1	0	1	DR	SR1	1	imm5	

Công dụng

```

If (bit[5] == 0)
    DR = SR1 AND SR2
Else
    DR = SR1 + SEXT (imm5);
setcc();

```

Ví dụ

ADD R2, R3, R4; R2 \leftarrow R3 AND R4
 ADD R2, R3, #7; R2 \leftarrow R3 AND 7

3- Lệnh rẽ nhánh có điều kiện BR

Dạng thức hợp ngữ

BRn	LABEL	BRzp	LABEL
BRz	LABEL	BRnp	LABEL
BRp	LABEL	BRnz	LABEL
BR	LABEL	BRnzp	LABEL

(Chú ý: lệnh BR có công dụng hoàn toàn như lệnh BRnzp)

Dạng nhị phân

15	12	11	10	9	8	0
0	0	0	0	n	z	p

PCoffset9

Công dụng

```

if ((n AND N) OR (z AND Z) OR (p AND P))
    PC = PC + SEXT (PCoffset9);

```

Ví dụ

BRzp LOOP ; Rẽ nhánh tới LOOP nếu kết quả gần nhất là ≥ 0
 BR NEXT ; Rẽ nhánh không điều kiện tới NEXT

4- Lệnh nhảy JMP và lệnh trở về từ chương trình con RET

Dạng thức hợp ngữ

JMP BaseR
 RET

Dạng nhị phân

15	12	11	9	8	6	5	0
1	1	0	0	0	0	BaseR	0 0 0 0 0 0

15	12	11	9	8	6	5	0
1	1	0	0	0	0	1	1 1 0 0 0 0 0 0

Công dụng

PC = BaseR;

Ví dụ

JMP R2 ; PC ← R2

RET ; PC ← R7

5- Lệnh nhảy vào chương trình con JSR và JSRR*Dạng thức hợp ngữ thanh công . com*

JSR LABEL

JSRR BaseR

Dạng nhị phân

15	12	11	10	0
0	1	0	0	PCoffset11

15	12	11	10	9	8	6	5	0
0	1	0	0	0	0	BaseR	0 0 0 0 0 0	

Công dụng

R7 = PC ;

If (bit[11] == 0)

PC = BaseR ;

Else

PC = PC + SEXT (Pcoffset11);

Ví dụ

JSR QUEUE ; Đặt địa chỉ của lệnh ngay sau JSR vào R7
 ; Nhảy tới QUEUE.

JSRR R3 ; Đặt địa chỉ của lệnh ngay sau JSRR vào R7
 ; Nhảy tới địa
 ; chỉ được chứa trong R3.

6- Lệnh nạp LD**Dạng thức hợp ngữ**

LD DR, LABEL

Dạng nhị phân

15	12	11	9	8	0
0	0	1	0	DR	PCoffset9

Công dụng

DR = mem[PC + SEXT(Pcoffset9)];
 setcc();

Ví dụ

LD R4, VALUE ; R4 ← mem[VALUE]

7- Lệnh nạp gián tiếp LDI**Dạng thức hợp ngữ**

LDI DR, LABEL

Dạng nhị phân

15	12	11	9	8	0
1	0	1	0	DR	PCoffset9

Công dụng

DR = mem[mem[PC + SEXT (Pcoffset9)]];
 setcc()

Ví dụ

LDI R4, ONEMORE ; R4 \leftarrow mem[mem[ONEMORE]]

8- Lệnh nạp theo Base và offset LDR

Dạng thức hợp ngữ

LDR DR, BaseR, offset8

Dạng nhị phân

15	12	11 9	8	6	5	0
0	1	1 0	DR	BaseR	offset6	

Công dụng

```
DR = mem[BaseR + SEXT(offset6)];
setcc();
```

Ví dụ

LDR R4, R2, # -5 ; R4 \leftarrow mem[R2 - 5]

9- Lệnh nạp địa chỉ tác dụng LEA

Dạng thức hợp ngữ

LEA DR, LABEL

Dạng nhị phân

15	12	11	9	8	0
1	1	1 0	DR	PCoffset9	

Công dụng

```
DR = PC + SEXT (PCoffset9);
setcc();
```

Ví dụ

LEA R4, TARGET ; R4 \leftarrow địa chỉ của TARGET.

10- Lệnh bù bit NOT

Dạng thức hợp ngữ

NOT DR, SR

Dạng nhị phân

15	12	11	11	9	8	6	5	4	3	2	0
1	0	0	1	n	D	R	S	R	1	1	1

Công dụng

```
DR = NOT (SR) ;
setcc();
```

Ví dụ

NOT R4, R2; R4 ← NOT(R2)

11- Lệnh trả về từ chương trình con RET*Dạng thức hợp ngữ*

RET

Dạng nhị phân

15	12	11	9	8	6	4	0
1	1	0	0	0	0	R	1

Công dụng

PC = R7;

Ví dụ

RET; PC ← R7

12- Lệnh trả về từ ngắt RTI*Dạng thức hợp ngữ*

RTI

Dạng nhị phân

15	12	11	0
1	0	0	0

Công dụng

if (PSR[15] == 0)

PC = mem[R6]; R6 là SSP

R6 = R6 + 1;

```

TEMP = mem[R6];
R6 = R6+1;
PSR = TEMP
; Khôi phục lại privilege mode và các mã điều kiện của quá trình ngắt
else
    Khởi tạo mode không đặc quyền;

```

Ví dụ

RTI ; PC, PSR ← hai trị trên đỉnh stack được lấy ra

13- Lệnh lưu ST

Dạng thức hợp ngữ

ST SR, LABEL

Dạng nhị phân

15	12	11	9	8	0
0	0	1	1	SR	PCoffset9

Công dụng

mem[PC + SEXT (Pcoffset9)] = SR;

Ví dụ

ST R4, HERE ;mem[HERE] ← R4

14- Lệnh lưu gián tiếp STI

Dạng thức hợp ngữ

STI SR, LABEL

Dạng nhị phân

15	12	11	9	8	0
1	0	1	1	SR	PCoffset9

Công dụng

mem[mem[PC + SEXT(PCoffset9)]] = SR;

Ví dụ

STI R4, NOT_HERE ; mem[mem[NOT_HERE]] ← R4

15- Lệnh lưu theo Base và offset STR

Dạng thức hợp ngữ

STR SR, BaseR, offset6

Dạng nhị phân

15	12	11	9	8	6	8	0
0	1	1	1	SR	BaseR		offset6

Công dụng

$\text{mem}[\text{BaseR} + \text{SEXT}(\text{offset6})] = \text{SR}$

Ví dụ

STR R4, R2, #5; $\text{mem}[\text{R2} + 5] \leftarrow \text{R4}$

16- Lệnh gọi hệ thống TRAP

Dạng thức hợp ngữ

TRAP trapvector8

Dạng nhị phân

15	12	11	8	7	0
1	1	1	1	0 0 0 0	trapvect8

Công dụng

R7 = PC ;

PC = $\text{mem}[\text{ZEXT}(\text{trapvect8})]$;

17- Mã lệnh chưa sử dụng (để mở rộng)

Dạng nhị phân

15	12	11	0
1	1	0	1

Phụ lục B
BẢNG MÃ ASCII

Trị thập phân	➔	0	16	32	48	64	80	96	112
↓	Trị thập lục phân	0	1	2	3	4	5	6	7
0	0	NUL	►	BLANK (SPACE)	0	@	P	'	P
1	1	☺	◀	!	1	A	Q	A	Q
2	2	☻	↑	"	2	B	R	B	R
3	3	♥	!!	#	3	C	S	C	S
4	4	♦	¶	\$	4	D	T	D	T
5	5	♣	§	%	5	E	U	E	U
6	6	♠	-	&	6	F	V	F	V
7	7	•	↓	'	7	G	W	G	W
8	8	█	↑	(8	H	X	H	X
9	9	○	↓)	9	I	Y	I	Y
10	A	◎	→	*	:	J	Z	J	Z
11	B	♂	←	+	;	K	[K	{
12	C	♀	L	,	<	L	\	L	
13	D	♪	↔	-	=	M]	M	}
14	E	♫	▲	.	>	N	^	N	~
15	F	☀	▼	/	?	O	_	O	◊ (DEL)

Trị thập phân	→	128	144	160	176	192	208	224	240
↓	Trị thập lục phân	8	9	A	B	C	D	E	F
0	0	Ç	É	á	ß	L	LL	A	≡
1	1	ü	æ	í	ßß	⊥	⊤	ß	±
2	2	é	Æ	ó	ßßß	⊤	⊤	Γ	Σ
3	3	â	ô	ú		⊤	⊤	Π	Λ
4	4	ä	ö	ñ	-	-	L	Σ	∫
5	5	à	ò	Ñ	≡	+	F	Σ	ʃ
6	6	å	û	^a		F	Γ	μ	÷
7	7	ç	ù	°	⊤	⊤	⊤	T	≈
8	8	ê	ÿ	¿	⊤	L	⊤	Φ	°
9	9	ë	Ö	¬		F	⊤	Θ	•
10	A	è	Ü	¬		LL	Γ	Ω	•
11	B	ï	¢	½	⊤	⊤	█	Δ	√
12	C	î	£	¼	⊤	⊤	█	∞	n
13	D	ì	¥	:	⊤	=	█	Φ	²
14	E	Ä	Pts	«	⊤	⊤	█	E	█
15	F	Å	f	»	⊤	⊤	█	█	no-break space

Ý nghĩa của các ký tự trong bảng mã ASCII 8 bit
(Nguồn: <http://ascii-table.com/ascii.php>)

Dec	Hex	Ký tự	Ý nghĩa
0	00	NUL	Null
1	01	STX	<u>S</u> tart <u>o</u> f <u>H</u> ead <u>e</u> r
2	02	SOT	<u>S</u> tart <u>o</u> f <u>T</u> ext
3	03	ETX	<u>E</u> nd <u>o</u> f <u>Text</u>
4	04	EOT	<u>E</u> nd <u>o</u> f <u>Transmission</u>
5	05	ENQ	<u>E</u> nquiry
6	06	ACK	<u>A</u> cknowled <u>g</u> e
7	07	BEL	<u>B</u> ell
8	08	BS	<u>B</u> ack <u>S</u> pace
9	09	HT	<u>H</u> orizontal <u>Tabulation</u>
10	0A	LF	<u>L</u> ine <u>F</u> eed
11	0B	VT	<u>V</u> ertical <u>Tabulation</u>
12	0C	FF	<u>F</u> orm <u>F</u> eed
13	0D	CR	<u>C</u> arriage <u>R</u> eturn
14	0E	SO	<u>S</u> hift <u>O</u> t
15	0F	SI	<u>S</u> hift <u>I</u> n
16	10	DLE	<u>D</u> ata <u>L</u> ink <u>E</u> scape
17	11	DC1	<u>D</u> evice <u>C</u> ontrol <u>1</u> (XON)
18	12	DC2	<u>D</u> evice <u>C</u> ontrol <u>2</u>
19	13	DC3	<u>D</u> evice <u>C</u> ontrol <u>3</u> (XOFF)
20	14	DC4	<u>D</u> evice <u>C</u> ontrol <u>4</u>
21	15	NAK	<u>N</u> egative <u>a</u> cknowled <u>g</u> e
22	16	SYN	<u>S</u> ynchronous <u>I</u> dle
23	17	ETB	<u>E</u> nd <u>o</u> f <u>Transmission <u>B</u>lock</u>
24	18	CAN	<u>C</u> ancel
25	19	EM	<u>E</u> nd <u>o</u> f <u>M</u> edium
26	1A	SUB	<u>S</u> ub <u>s</u> titute
27	1B	ESC	<u>E</u> s <u>c</u> ape
28	1C	FS	<u>F</u> ile <u>S</u> eparator

29	ID	GS	<u>Group Separator</u>
30	1E	RS	<u>Record Separator</u>
31	1F	US	<u>Unit Separator</u>
32	20	[Space]	<u>Space</u>
33	21	!	Exclamation mark
34	22	"	Quotes
35	23	#	Hash
36	24	\$	Dollar
37	25	%	Percent
38	26	&	Ampersand
39	27	'	Apostrophe
40	28	(Open bracket
41	29)	Close bracket
42	2A	*	Asterisk
43	2B	+	Plus
44	2C	,	Comma
45	2D	-	Dash
46	2E	.	Full stop
47	2F	/	Slash
48	30	0	Zero
49	31	1	One
50	32	2	Two
51	33	3	Three
52	34	4	Four
53	35	5	Five
54	36	6	Six
55	37	7	Seven
56	38	8	Eight
57	39	9	Nine
58	3A	:	Colon
59	3B	;	Semi-colon
60	3C	<	Less than
61	3D	=	Equals

62	3E	>	Greater than
63	3F	?	Question mark
64	40	@	At
65	41	A	Uppercase A
66	42	B	Uppercase B
67	43	C	Uppercase C
68	44	D	Uppercase D
69	45	E	Uppercase E
70	46	F	Uppercase F
71	47	G	Uppercase G
72	48	H	Uppercase H
73	49	I	Uppercase I
74	4A	J	Uppercase J
75	4B	K	Uppercase K
76	4C	L	Uppercase L
77	4D	M	Uppercase M
78	4E	N	Uppercase N
79	4F	O	Uppercase O
80	50	P	Uppercase P
81	51	Q	Uppercase Q
82	52	R	Uppercase R
83	53	S	Uppercase S
84	54	T	Uppercase T
85	55	U	Uppercase U
86	56	V	Uppercase V
87	57	W	Uppercase W
88	58	X	Uppercase X
89	59	Y	Uppercase Y
90	5A	Z	Uppercase Z
91	5B	[Open square bracket
92	5C	\	Backslash
93	5D]	Close square bracket
94	5E	^	Caret / hat

95	5F	-	Underscore
96	60	'	Grave accent
97	61	A	Lowercase a
98	62	B	Lowercase b
99	63	C	Lowercase c
100	64	D	Lowercase d
101	65	E	Lowercase e
102	66	F	Lowercase f
103	67	G	Lowercase g
104	68	H	Lowercase h
105	69	I	Lowercase i
106	6A	J	Lowercase j
107	6B	K	Lowercase k
108	6C	L	Lowercase l
109	6D	M	Lowercase m
110	6E	N	Lowercase n
111	6F	O	Lowercase o
112	70	P	Lowercase p
113	71	Q	Lowercase q
114	72	R	Lowercase r
115	73	S	Lowercase s
116	74	T	Lowercase t
117	75	U	Lowercase u
118	76	V	Lowercase v
119	77	W	Lowercase w
120	78	X	Lowercase x
121	79	Y	Lowercase y
122	7A	Z	Lowercase z
123	7B	{	Open brace
124	7C		Pipe
125	7D	}	Close brace
126	7E	~	Tilde
127	7F	DEL	<u>Delete</u>

128	80	Ç	latin capital letter c with cedilla
129	81	ü	latin small letter u with diaeresis
130	82	é	latin small letter e with acute
131	83	â	latin small letter a with circumflex
132	84	ä	latin small letter a with diaeresis
133	85	à	latin small letter a with grave
134	86	å	latin small letter a with ring above
135	87	ç	latin small letter c with cedilla
136	88	ê	latin small letter e with circumflex
137	89	ë	latin small letter e with diaeresis
138	8A	è	latin small letter e with grave
139	8B	ï	latin small letter i with diaeresis
140	8C	í	latin small letter i with circumflex
141	8D	ì	latin small letter i with grave
142	8E	Ä	latin capital letter a with diaeresis
143	8F	Å	latin capital letter a with ring above
144	90	É	latin capital letter e with acute
145	91	æ	latin small ligature ae
146	92	Æ	latin capital ligature ae
147	93	ö	latin small letter o with circumflex
148	94	ö	latin small letter o with diaeresis
149	95	ò	latin small letter o with grave
150	96	û	latin small letter u with circumflex
151	97	ù	latin small letter u with grave
152	98	ÿ	latin small letter y with diaeresis
153	99	Ö	latin capital letter o with diaeresis
154	9A	Ü	latin capital letter u with diaeresis
155	9B	¢	cent sign
156	9C	£	pound sign
157	9D	¥	yen sign
158	9E	Pts	peseta sign
159	9F	f	latin small letter f with hook
160	A0	á	latin small letter a with acute

161	A1	í	latin small letter i with acute
162	A2	ó	latin small letter o with acute
163	A3	ú	latin small letter u with acute
164	A4	ñ	latin small letter n with tilde
165	A5	Ñ	latin capital letter n with tilde
166	A6	՚	feminine ordinal indicator
167	A7	՚	masculine ordinal indicator
168	A8	՞	inverted question mark
169	A9	՞	reversed not sign
170	AA	՞	not sign
171	AB	½	vulgar fraction one half
172	AC	¼	vulgar fraction one quarter
173	AD	፣	inverted exclamation mark
174	AE	«	left-pointing double angle quotation mark
175	AF	»	right-pointing double angle quotation mark
176	B0	❖	light shade
177	B1	❖	medium shade
178	B2	■	dark shade
179	B3		box drawings light vertical
180	B4	-	box drawings light vertical and left
181	B5	-	box drawings vertical single and left double
182	B6	--	box drawings vertical double and left single
183	B7	-	box drawings down double and left single
184	B8	-	box drawings down single and left double
185	B9	--	box drawings double vertical and left
186	BA		box drawings double vertical
187	BB	-	box drawings double down and left
188	BC	-	box drawings double up and left
189	BD		box drawings up double and left single
190	BE	-	box drawings up single and left double
191	BF	-	box drawings light down and left
192	C0	L	box drawings light up and right
193	C1	+	box drawings light up and horizontal

194	C2	⊤	box drawings light down and horizontal
195	C3	⊠	box drawings light vertical and right
196	C4	⊒	box drawings light horizontal
197	C5	⊢	box drawings light vertical and horizontal
198	C6	⊣	box drawings vertical single and right double
199	C7	⊤⊣	box drawings vertical double and right single
200	C8	⊤⊠	box drawings double up and right
201	C9	⊤⊠	box drawings double down and right
202	CA	⊤⊒	box drawings double up and horizontal
203	CB	⊤⊢	box drawings double down and horizontal
204	CC	⊤⊣	box drawings double vertical and right
205	CD	=	box drawings double horizontal
206	CE	⊤⊢	box drawings double vertical and horizontal
207	CF	⊒	box drawings up single and horizontal double
208	D0	⊒	box drawings up double and horizontal single
209	D1	⊤⊠	box drawings down single and horizontal double
210	D2	⊤⊠	box drawings down double and horizontal single
211	D3	⊤⊠	box drawings up double and right single
212	D4	⊠	box drawings up single and right double
213	D5	⊠	box drawings down single and right double
214	D6	⊠	box drawings down double and right single
215	D7	⊤⊠	box drawings vertical double and horizontal single
216	D8	⊤⊢	box drawings vertical single and horizontal double
217	D9	⊠	box drawings light up and left
218	DA	⊠	box drawings light down and right
219	DB	█	full block
220	DC	█	lower half block
221	DD	█	left half block
222	DE	█	right half block
223	DF	█	upper half block
224	E0	ⓐ	greek small letter alpha
225	E1	ⓑ	latin small letter sharp s
226	E2	ⓘ	greek capital letter gamma

227	E3	π	greek small letter pi
228	E4	Σ	greek capital letter sigma
229	E5	σ	greek small letter sigma
230	E6	μ	micro sign
231	E7	τ	greek small letter tau
232	E8	Φ	greek capital letter phi
233	E9	Θ	greek capital letter theta
234	EA	Ω	greek capital letter omega
235	EB	δ	greek small letter delta
236	EC	∞	Infinity
237	ED	ϕ	greek small letter phi
238	EE	ϵ	greek small letter epsilon
239	EF	\cap	Intersection
240	F0	\equiv	identical to
241	F1	\pm	plus-minus sign
242	F2	\geq	greater-than or equal to
243	F3	\leq	less-than or equal to
244	F4	\int	top half integral
245	F5	\int	bottom half integral
246	F6	\div	division sign
247	F7	\approx	almost equal to
248	F8	\circ	degree sign
249	F9	\cdot	bullet operator
250	FA	\cdot	middle dot
251	FB	$\sqrt{}$	square root
252	FC	n^{a}	superscript latin small letter n
253	FD	z^{2}	superscript two
254	FE	\blacksquare	black square
255	FF		no-break space

Phụ lục C

CÁC PHÍM MỞ RỘNG TRÊN BÀN PHÍM

Phím	Bình thường	Kết hợp với Shift	Kết hợp với Ctrl	Kết hợp với Alt
F1	0 59	0 84	0 94	0 104
F2	0 60	0 85	0 95	0 105
F3	0 61	0 86	0 96	0 106
F4	0 62	0 87	0 97	0 107
F5	0 63	0 88	0 98	0 108
F6	0 64	0 89	0 99	0 109
F7	0 65	0 90	0 100	0 110
F8	0 66	0 91	0 101	0 111
F9	0 67	0 92	0 102	0 112
F10	0 68	0 93	0 103	0 113
←	0 75	52	0 115	none
→	0 77	54	0 116	none
↑	0 72	56	none	none
↓	0 80	50	none	none
Home	0 71	55	0 119	none
End	0 79	49	0 117	none
PgUp	0 73	57	0 132	none
PgDn	0 81	51	0 118	none
Ins	0 82	48	none	none
Del	0 83	46	0 255	none
Esc	27	27	27	none
Backspace	8	8	127	none
Tab	9	0 15	none	none
Enter	13	13	10	none
A	97	65	1	0 30
B	98	66	2	0 48
C	99	67	3	0 46
D	100	68	4	0 32
E	101	69	5	0 18
F	102	70	6	0 33
G	103	71	7	0 34
H	104	72	8	0 35
I	105	73	9	0 23
J	106	74	10	0 36

Phím	Bình thường	Kết hợp với Shift	Kết hợp với Ctrl	Kết hợp với Alt
K	107	75	11	0 37
L	108	76	12	0 38
M	109	77	13	0 50
N	110	78	14	0 49
O	111	79	15	0 24
P	112	80	16	0 25
Q	113	81	17	9 16
R	114	82	18	0 19
S	115	83	19	0 31
T	116	84	20	0 20
U	117	85	21	0 22
V	118	86	22	0 47
W	119	87	23	0 17
X	120	88	24	0 45
Y	121	89	25	0 21
Z	122	90	26	0 44
[91	123	27	none
\	92	124	28	none
]	93	125	29	none
.	96	126	none	none
0	48	41	none	0 129
1	49	33	none	0 120
2	50	64	0 3	0 121
3	51	35	none	0 122
4	52	36	none	0 123
5	53	37	none	0 124
6	54	94	30	0 125
7	55	38	none	0 126
8	56	42	none	0 127
9	57	40	none	0 128
*	42	none	0 114	none
Keypad +	43	43	none	none
Keypad -	45	45	none	none
=	61	43	none	0 131
/	47	63	none	none
:	59	58	none	none
-	45	95	31	0 130

Phụ lục D

CÁCH TẠO PROJECT FILE TRONG C/C++

Một trong những điểm mạnh của C là cho phép tạo project file. Đây là một hình thức cho phép lập trình viên có thể tạo ra một file trong đó có thể liên kết nhiều file lại với nhau để file này có thể sử dụng các thông tin đã được khai báo trong các file khác, các file này tạm gọi là các file thư viện.

Các bước để tạo project file:

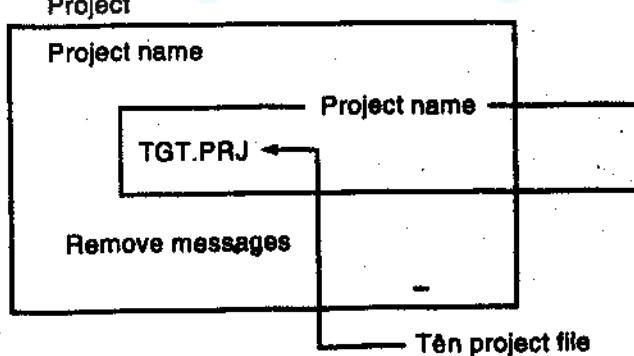
1- Bước 1. Tạo các file thư viện chứa các thông tin, hàm... mà chương trình chính cần sử dụng. Các file này thường có phần mở rộng là .C (hoặc nếu đã dịch xong thì có phần mở rộng là .OBJ) và file chứa chương trình chính.

2- Bước 2. Soạn thảo một file văn bản có phần mở rộng là .PRJ, tên file này sau khi dịch xong là file thực thi .EXE, nội dung file này liệt kê các file cần liên kết lại với nhau.

Ví dụ: Cần liên kết file func1.c và mymain.c, thì ta có thể đặt tên file .PRJ là TGT.PRJ với nội dung như sau:

```
func1.c
func2.c
cmymain.c
```

3- Bước 3. Lên menu Project/Project file khai báo tên file, ví dụ



4- Bước 4. Dịch file .PRJ:ấn Ctrl-F9 để dịch. Nếu có lỗi ta sẽ sửa lại các file thành phần và lại dịch trở lại file .PRJ.

Ví dụ: Ta có các file func1.c và func2.c với các nội dung như sau:

FUNC1.C

```
long tong (int n)
{
    int i;
    long s = 0;
    for (i = 1; i <= n; i++)
        s += i;
    return s;
}
```

FUNC2.C

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

File chương trình chính mymain.c như sau:

MYMAIN.C

```
/*
Chương trình tổng các giai thừa.
s = (1)! + (1+2)! + ... + (1+2+...+n)!
*/
```

```
#include <stdio.h>
```

```
#include <conio.h>
```

```
long tong (int n);
```

```
long tich (int n);
```

```
main()
```

```
{
```

```
    int n, i;
```

```
    long s = 0;
```

```

clrscr();
printf("chuong trinh tong cac giai thua\n");
printf(" s=(1)! + (1+2)! + ... + (1+2+...+n)!\n");
printf("Moi nhap mot so: ");
scanf("%d", &n);
for (i = 1; i <= n; i++)
    s += tich(tong(i));
printf("Tong cac giai thua la: %ld \n", s);
getch()
}

```

file thu.prj có nội dung như sau:

THU.PRJ

func1.c
func2.c
mymain.c

Chương trình cho xuất liệu:

Chương trình tổng các giai thừa
 $s = (1)! + (1+2)! + \dots + (1+2+\dots+n)!$
 Moi nhap mot so: 2
 Tong cac giai thua la: 7

Ta hoàn toàn có thể dịch từng modul func1.c, func2.c một cách riêng lẻ để sửa đổi cú pháp của từng modul chương trình, sau đó liên kết lại bằng project file, nhưng khi đó nội dung project file là các file .OBJ, như ví dụ như sau:

func1.obj
func2.obj
mymain.obj

Với môi trường IDE của BC++ version 5.0x, khái niệm project file không còn đơn giản như các version trước nữa. Lúc này các file có thuộc tính là .ide. Đọc giả có thể tham khảo các tài liệu về môi trường IDE của các bộ dịch để biết thêm chi tiết.

TÀI LIỆU THAM KHẢO CHÍNH

1. Robert Lafore, *C Programming Using Turbo C++*, SAMS, 1992.
2. Robert L. Kruse - Bruce P. Leung - Clovis L. Tondo, *Data structures and program design in C*, Prentice Hall.
3. E. M. Landesman, *Linear for mathematics, science and engineering*, Prentice Hall, 1989.
4. Robert C. Hutchison - Stephen B. Just, *Programming Using the C language*, Mc Graw Hill, 1986.
5. Nabajyoti Barkakati - The Waite Group, *Turbo C Bible*, SAMS, 1991.
6. F.S Hill, JR.: *Computer graphics*, Maxwell Mac Millan, 1990.
7. Mitchell Waite - Stephen Prata - The Waite Group, *New C Primer plus*, SAMS, 1991.
8. Đặng Thành Tín, *Tin học 1*, Đại học Bách khoa TP.HCM, 1994.
9. Yale N. Patt, và Sanjay J. Patel, *Introduction to computing systems*, Mc Graw Hill, 2005.
10. M. Morris Mano và Charles R. Kime, *Logic and Computer Design Fundamentals*, Prentice Hall, 2004.
11. Đặng Thành Tín, *Tin học 2*, Đại học Bách khoa TP.HCM, 2007.
12. Andrew S. Tanenbaum, *Structured computer organization (5th Edition)*, Prentice Hall, 2005.

HỆ THỐNG MÁY TÍNH VÀ NGÔN NGỮ C

Đặng Thành Tín

NHÀ XUẤT BẢN
ĐẠI HỌC QUỐC GIA TP HỒ CHÍ MINH
KP 6, P. Linh Trung, Q. Thủ Đức, TPHCM
Số 3 Công trường Quốc tế, Q.3, TPHCM
ĐT: 38239172, 38239170
Fax: 38239172; Email: vnuhp@vnuhcm.edu.vn

★ ★ ★

Chịu trách nhiệm xuất bản
TS HUỲNH BÁ LÂN

Tổ chức bản thảo và chịu trách nhiệm về tác quyền
TRƯỜNG ĐẠI HỌC BÁCH KHOA – ĐHQG TPHCM

Biên tập
NGUYỄN ĐỨC MAI LÂM

Sửa bản in
THÂN THỊ HỒNG

Trình bày bìa
VÕ THỊ HỒNG

In tái bản 500 cuốn, khổ 16 x 24 cm
Số đăng ký KHXB: 84-2011/CXB/418-04/ĐHQG-TPHCM
Quyết định xuất bản số: 581/QĐ-ĐHQG-TPHCM/TB
ngày 28/10/2011 của Nhà xuất bản ĐHQG TPHCM
In tại Xưởng in Đại học Bách khoa - ĐHQG TP.HCM
Nộp lưu chiểu tháng 11 năm 2011.