ỦY BAN NHÂN DÂN TỈNH THANH HÓA
TRƯỜNG ĐẠI HỌC VĂN HÓA, THỂ THAO VÀ DU LỊCH

TẬP BÀI GIẢNG

LẬP TRÌNH CĂN BẢN

(Dành cho sinh viên ngành Thông tin thư viện)

Giảng viên soạn	:	Hoàng Anh Công
Bộ môn	:	Thông tin thư viện
Khoa	:	Văn hóa thông tin
Mã học phần	:	ĐC012

THANH HÓA, NĂM 2018

MỤC LỤC

MỞ ĐẦU

1. Mục tiêu và yêu cầu của học phần

1.1. Mục tiêu tổng quát

Ngày nay, khoa học máy tính thâm nhập vào mọi lĩnh vực. Tự động hóa hiện đang là ngành chủ chốt điều hướng sự phát triển thế giới. Bất cứ ngành nghề nào cũng cần phải hiểu biết ít nhiều về Công nghệ Thông tin và lập trình nói chung. Cụ thể, C là một ngôn ngữ lập trình cấp cao mà mọi lập trình viên cần phải biết. Vì thế, trong giáo trình này, chúng ta sẽ nghiên cứu chi tiết **cấu trúc ngôn ngữ C**. Đầu tiên chúng ta tìm hiểu sự khác nhau của những khái niệm: Lệnh (Command), Chương trình (Program) và Phần mềm (Software).

C là ngôn ngữ lập trình cấp cao, được sử dụng rất phổ biến để lập trình hệ thống và phát triển các ứng dụng.

Vào những năm cuối thập kỷ 60 đầu thập kỷ 70 của thế kỷ XX, Dennish Ritchie (làm việc tại phòng thí nghiệm Bell) đã phát triển ngôn ngữ lập trình C dựa trên ngôn ngữ BCPL (do Martin Richards đưa ra vào năm 1967) và ngôn ngữ B (do Ken Thompson phát triển từ ngôn ngữ BCPL vào năm 1970 khi viết hệ điều hành UNIX đầu tiên trên máy PDP-7) và được cài đặt lần đầu tiên trên hệ điều hành UNIX của máy DEC PDP-11.

Năm 1978, Dennish Ritchie và B.W Kernighan đã cho xuất bản quyển "Ngôn ngữ lập trình C" và được phổ biến rộng rãi đến nay.

Lúc ban đầu, C được thiết kế nhằm lập trình trong môi trường của hệ điều hành Unix nhằm mục đích hỗ trợ cho các công việc lập trình phức tạp. Nhưng về sau, với những nhu cầu phát triển ngày một tăng của công việc lập trình, C đã vượt qua khuôn khổ của phòng thí nghiệm Bell và nhanh chóng hội nhập vào thế giới lập trình để rồi các công ty lập trình sử dụng một cách rộng rãi. Sau đó, các công ty sản xuất phần mềm lần lượt đưa ra các phiên bản hỗ trợ cho việc lập trình bằng ngôn ngữ C và chuẩn ANSI C cũng được khai sinh từ đó.

Ngôn ngữ lập trình C là một ngôn ngữ lập trình hệ thống rất mạnh và rất "mềm dẻo", có một thư viện gồm rất nhiều các hàm (function) đã được tạo sẵn. Người lập trình có thể tận dụng các hàm này để giải quyết các bài toán mà không cần phải tạo mới. Hơn thế nữa, ngôn ngữ C hỗ trợ rất nhiều phép toán nên phù hợp cho việc giải quyết các bài toán kỹ thuật có nhiều công thức phức tạp. Ngoài ra, C cũng cho phép người lập trình tự định nghĩa thêm các kiểu dữ liệu trừu tượng khác. Tuy nhiên, điều mà người mới vừa học lập trình C thường gặp "rắc rối" là "hơi khó hiểu" do sự "mềm dẻo" của C. Dù vậy, C được phổ biến khá rộng rãi và đã trở thành một công cụ lập trình khá mạnh, được sử dụng như là một ngôn ngữ lập trình chủ yếu trong việc xây dựng những phần mềm hiện nay.

Ngôn ngữ C có những đặc điểm cơ bản sau:

- *Tính cô đọng (compact)*: C chỉ có 32 từ khóa chuẩn và 40 toán tử chuẩn, nhưng hầu hết đều được biểu diễn bằng những chuỗi ký tự ngắn gọn.
- *Tính cấu trúc (structured)*: C có một tập hợp những chỉ thị của lập trình như cấu trúc lựa chọn, lặp… Từ đó các chương trình viết bằng C được tổ chức rõ ràng, dễ hiểu.
- *Tính tương thích (compatible)*: C có bộ tiền xử lý và một thư viện chuẩn vô cùng phong phú nên khi chuyển từ máy tính này sang máy tính khác các chương trình viết bằng C vẫn hoàn toàn tương thích.
- *Tính linh động (flexible)*: C là một ngôn ngữ rất uyển chuyển và cú pháp, chấp nhận nhiều cách thể hiện, có thể thu gọn kích thước của các mã lệnh làm chương trình chạy nhanh hơn.
- *Biên dịch (compile)*: C cho phép biên dịch nhiều tập tin chương trình riêng rẽ thành các tập tin đối tượng (object) và liên kết (link) các đối tượng đó lại với nhau thành một chương trình có thể thực thi được (executable) thống nhất.

1.2. Mục tiêu cụ thể

** Kiến thức*

Môn Lập Trình Căn Bản cung cấp cho sinh viên những kiến thức cơ bản về lập trình thông qua ngôn ngữ lập trình C. Môn học này là nền tảng để tiếp thu hầu hết các môn học khác trong chương trình đào tạo. Mặt khác, nắm vững ngôn ngữ C là cơ sở để phát triển các ứng dụng.

Học xong môn này, sinh viên ngành thông tin học nắm được các vấn đề sau:

- Khái niệm về ngôn ngữ lập trình.
- Khái niệm về kiểu dữ liệu
- Kiểu dữ liệu có cấu trúc (cấu trúc dữ liệu).
- Khái niệm về giải thuật
- Ngôn ngữ biểu diễn giải thuật.
- Ngôn ngữ sơ đồ (lưu đồ), sử dụng lưu đồ để biểu diễn các giải thuật.
- Tổng quan về Ngôn ngữ lập trình C.
- Các kiểu dữ liệu trong C.
- Các lệnh có cấu trúc.
- Cách thiết kế và sử dụng các hàm trong C.
- Một số cấu trúc dữ liệu trong C.

Các yêu cầu để học lập trình căn bản C:

- Kiến thức toán học.
- Kiến thức và kỹ năng thao tác trên máy tính.

2. Cấu trúc tổng quát học phần

TT	Nội dung cơ bản của bài	Tổng số tiết	Số tiết lên lớp Của GV	Số tiết SV học nhóm, làm bài tại lớp	Số tiết nghiên cứu ngoài xã hội
1	**Tín chỉ 1: Tổng quan về ngôn ngữ lập trình C**	**30**	**15**	**15**	
	<u>Bài 1:</u> Ngôn ngữ lập trình và phương pháp lập trình				
1.1	Ngôn ngữ lập trình Kỹ thuật lập trình Các bước lập trình	10	5	5	
	<u>Bài 2:</u> Các thành phần trong ngôn ngữ C Từ khóa				
1.2	Tên Kiểu dữ liệu Khai báo biến Ghi chú	10	5	5	

	<u>Bài 3</u>: Nhập xuất dữ liệu			
1.3	Ý nghĩa, cách sử dụng hàm printf, scanf Sử dụng khuôn dạng, ký tự đặc biệt, ký tự điều khiển trong printf, scanf.	10	5	5
2	**Tín chỉ 2: Cấu trúc, hàm, mảng và chuỗi.**	**30**	**15**	**15**
	<u>Bài 1:</u> Cấu trúc rẽ nhánh có điều kiện.			
2.1	Lệnh và khối lệnh Lệnh if Lệnh Switch	8	4	4
	<u>Bài 2:</u> Cấu trúc vòng lặp.			
2.2	Lệnh for Lệnh break Lệnh continue Lệnh while Lệnh do…while Vòng lặp lồng nhau So sánh sự khác nhau của các vòng lặp	8	4	4
	<u>Bài 3:</u> Hàm			
2.3	Tham số dạng tham biến và tham trị Sử dụng biến toàn cục Dùng dẫn hướng #define	8	4	4

	Bài 4: Mảng và chuỗi			
2.4	Mảng Chuỗi	6	3	3
3	**Tín chỉ 3: Con trỏ, tập tin và đệ quy**	**30**	**15**	**15**
	Bài 1: Con trỏ - Khái báo biến con trỏ - Truyền địa chỉ sang hàm - Con trỏ và mảng			
3.1	- Con trỏ trỏ đến mảng trong hàm - Con trỏ và chuỗi - Khởi tạo mảng con trỏ trỏ đến chuỗi - Xử lý con trỏ trỏ đến chuỗi - Con trỏ trỏ đến con trỏ	10	5	5
	Bài 2: Tập tin - Ghi, đọc mảng			
3.2	- Ghi, đọc structure - Các mode khác để mở tập tin - Một số hàm thao tác trên file khác	10	5	5
	Bài 3: Các kiểu dữ liệu tự tạo			
3.3	-Structure -Enum	10	5	5
	Tổng cộng	**90**	**45**	**45**

3. Nội dung tín chỉ 1

3.1. Danh mục tên bài

TT	Tên bài giảng	Tổng số tiết trên lớp của GV	Số tiết GV trình bầy	Số tiết GV hướng dẫn thảo luận	Bài tự luận/ nghiên cứu ngoài xã hội	Giảng viên thực hiện
1	**<u>Bài 1:</u> Ngôn ngữ lập trình và phương pháp lập trình** Ngôn ngữ lập trình Kỹ thuật lập trình Các bước lập trình	10	5	5	0	10
2	**<u>Bài 2:</u> Các thành phần trong ngôn ngữ C** Từ khóa Tên Kiểu dữ liệu Khai báo biến Ghi chú	10	5	5	0	10
3	**<u>Bài 3</u>: Nhập xuất dữ liệu** Ý nghĩa, cách sử dụng hàm printf, scanf Sử dụng khuôn dạng, ký tự đặc biệt, ký tự điều khiển trong printf, scanf.	10	5	5	0	10
	Tổng:	**30**	**15**	**15**	**0**	30

3.2 Nội dung bài giảng 1

3.2.1 Tên bài giảng: BÀI 1: TỔNG QUAN VỀ NGÔN NGỮ LẬP TRÌNH C

Số tiết lên lớp của GV 05 tiết ; Số tiết tự làm bài của SV: 05 tiết.

3.2.2. Phần mở đầu tiếp cận bài;

Ngày nay, khoa học máy tính thâm nhập vào mọi lĩnh vực. Tự động hóa hiện đang là ngành chủ chốt điều hướng sự phát triển thế giới. Bất cứ ngành nghề nào cũng cần phải hiểu biết ít nhiều về Công nghệ Thông tin và lập trình nói chung. Cụ thể, C là một ngôn ngữ lập trình cấp cao mà mọi lập trình viên cần phải biết. Vì thế, trong giáo trình này, chúng ta sẽ nghiên cứu chi tiết **cấu trúc ngôn ngữ C**. Đầu tiên chúng ta tìm hiểu sự khác nhau của những khái niệm: Lệnh (Command), Chương trình (Program) và Phần mềm (Software).

3.2.3. Phần kiến thức, kỹ thuật căn bản:

3.2.3.1 Phần kiến thức căn bản

Mục tiêu:

Kết thúc bài học này, bạn có thể:

- Phân biệt sự khác nhau giữa Câu lệnh, Chương trình và Phần mềm
- Biết được quá trình hình thành C
- Nên dùng C khi nào và tại sao
- Nắm được cấu trúc một chương trình C
- Hiểu rõ khái niệm giải thuật (algorithms)
- Vẽ lưu đồ (flowchart)
- Liệt kê các ký hiệu dùng trong lưu đồ

a) *Ra lệnh cho máy tính làm việc*

Khi một máy tính được khởi động, nó sẽ tự động thực thi một số tiến trình và xuất kết quả ra màn hình. Điều này diễn ra thế nào? Câu trả lời đơn giản là nhờ vào Hệ điều hành cài đặt bên trong máy tính. Hệ điều hành (operating system) được xem như phần mềm hệ thống. Phần mềm này khởi động máy tính và thiết lập các thông số ban đầu trước khi trao quyền cho người dùng. Để làm được điều này, hệ điều hành phải được cấu tạo từ một tập hợp các chương trình. Mọi chương trình đều cố gắng đưa ra lời giải cho một hay nhiều bài toán nào đóMọi chương trình cố gắng đưa ra giải pháp cho một hay nhiều vấn đề. Mỗi chương trình là tập hợp các câu lệnh giải quyết một bài toán cụ thể. Một nhóm lệnh tạo thành một chương trình và một nhóm các chương trình tạo thành một phần mềm.

Để rõ hơn, chúng ta hãy xem xét một thí dụ : Một người bạn đến nhà chúng ta chơi và được mời món sữa dâu. Anh ta thấy ngon miệng và muốn xin công thức làm. Chúng ta hướng dẫn cho anh ta làm như sau :

1. Lấy một ít sữa.

2. Đổ nước ép dâu vào.
3. Trộn hỗn hợp này và làm lạnh.

Bây giờ nếu bạn của chúng ta theo những chỉ dẫn này, họ cũng có thể tạo ra món sữa dâu tuyệt vời.

Chúng ta hãy phân tích chỉ thị (lệnh) ở trên

- Lệnh đầu tiên : Lệnh này hoàn chỉnh chưa ? Nó có trả lời được câu hỏi lấy sữa 'ở đâu' ?.
- Lệnh thứ hai : Một lần nữa, lệnh này không nói rõ nước ép dâu để 'ở đâu'.

May mắn là bạn của chúng ta đủ thông minh để hiểu được công thức pha chế nói trên, dù rằng còn nhiều điểm chưa rõ ràng. Do vậy nếu chúng ta muốn phổ biến cách làm, chúng ta cần bổ sung các bước như sau :

1. Rót một ly sữa vào máy trộn.
2. Đổ thêm vào một ít nước dâu ép.
3. Đóng nắp máy trộn
4. Mở điện và bắt đầu trộn
5. Dừng máy trộn lại
6. Nếu đã trộn đều thì tắt máy, ngược lại thì trộn tiếp.
7. Khi đã trộn xong, rót hỗn hợp vào tô và đặt vào tủ lạnh.
8. Để lạnh một lúc rồi lấy ra dùng.

So sánh hai cách hướng dẫn nêu trên, hướng dẫn thứ hai chắc chắn hoàn chỉnh, rõ ràng hơn, ai cũng có thể đọc và hiểu được.

Tương tự, máy tính cũng xử lý dữ liệu dựa vào tập lệnh mà nó nhận được. Đương nhiên các chỉ thị đưa cho máy vi tính cũng cần phải hoàn chỉnh và có ý nghĩa rõ ràng. Những chỉ thị này cần phải tuân thủ các quy tắc:

1. Tuần tự
2. Có giới hạn
3. Chính xác.

Mỗi chỉ thị trong tập chỉ thị được gọi là "câu lệnh" và tập các câu lệnh được gọi là "chương trình".

Chúng ta hãy xét trường hợp chương trình hướng dẫn máy tính cộng hai số.

Các lệnh trong chương trình có thể là :

1. Nhập số thứ nhất và nhớ nó.
2. Nhập số thứ hai và nhớ nó.
3. Thực hiện phép cộng giữa số thứ nhất và số thứ hai, nhớ kết quả phép cộng.
4. Hiển thị kết quả.

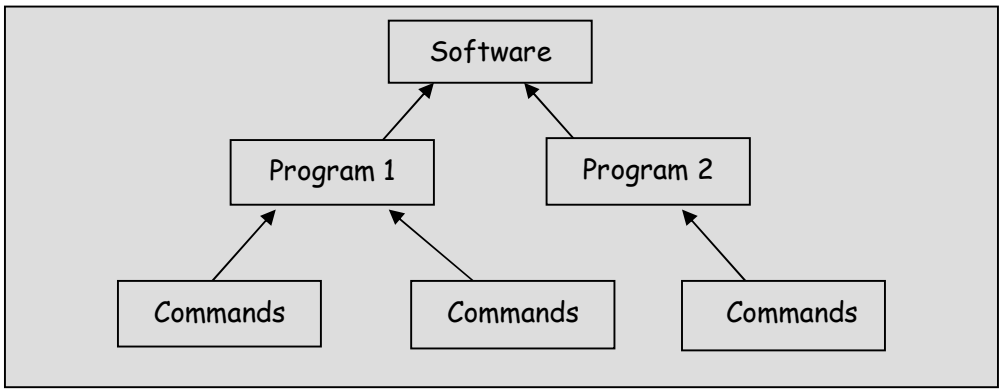
5. Kết thúc.

Tập lệnh trên tuân thủ tất cả các quy tắc đã đề cập. Vì vậy, tập lệnh này là một chương trình và nó sẽ thực hiện thành công việc cộng hai số trên máy tính.

Ghi chú: Khả năng nhớ của con người được biết đến như là trí nhớ, khả năng nhớ dữ liệu được đưa vào máy tính được gọi là "bộ nhớ". Máy tính nhận dữ liệu tại một thời điểm và làm việc với dữ liệu đó vào thời điểm khác, nghĩa là máy tính ghi dữ liệu vào trong bộ nhớ rồi sau đó đọc ra để truy xuất các giá trị dữ liệu và làm việc với chúng

Khi khối lượng công việc giao cho máy tính ngày càng nên nhiều và phức tạp thì tất cả các câu lệnh không thể được đưa vào một chương trình, chúng cần được chia ra thành một số chương trình nhỏ hơn. Tất cả các chương trình này cuối cùng được tích hợp lại để chúng có thể làm việc với nhau. Một tập hợp các chương trình như thế được gọi là phần mềm.

Mối quan hệ giữa ba khái niệm câu lệnh, chương trình và phần mềm có thể được biểu diễn bằng sơ đồ trong hình 1.1:



Hình 1.1: Phần mềm, chương trình và câu lệnh

b) Ngôn ngữ C

Vào đầu những năm 70 tại phòng thí nghiệm Bell, Dennis Ritchie đã phát triển ngôn ngữ C. C được sử dụng lần đầu trên một hệ thống cài đặt hệ điều hành UNIX. C có nguồn gốc từ ngôn ngữ BCPL do Martin Richards phát triển. BCPL sau đó đã được Ken Thompson phát triển thành ngôn ngữ B, đây là người khởi thủy ra C.

Trong khi BCPL và B không hỗ trợ kiểu dữ liệu, thì C đã có nhiều kiểu dữ liệu khác nhau. Những kiểu dữ liệu chính gồm : kiểu ký tự (character), kiểu số nguyên (interger) và kiểu số thực (float).

C liên kết chặt chẽ với hệ thống UNIX nhưng không bị trói buộc vào bất cứ một máy tính hay hệ điều hành nào. C rất hiệu quả để viết các chương trình thuộc nhiều những lĩnh vực khác nhau.

C cũng được dùng để lập trình hệ thống. Một chương trình hệ thống có ý nghĩa liên quan đến hệ điều hành của máy tính hay những tiện ích hỗ trợ nó. Hệ điều hành (OS), trình thông dịch (Interpreters), trình soạn thảo (Editors), chương trình Hợp Ngữ (Assembly) là các chương trình hệ thống. Hệ điều hành UNIX được phát triển dựa vào C. C đang được sử dụng rộng rãi bởi vì tính hiệu quả và linh hoạt. Trình biên dịch (compiler) C có sẵn cho hầu hết các máy tính. Mã lệnh viết bằng C trên máy này có thể được biên dịch và chạy trên máy khác chỉ cần thay đổi rất ít hoặc không thay đổi gì cả. Trình biên dịch C dịch nhanh và cho ra mã đối tượng không lỗi.

C khi thực thi cũng rất nhanh như hợp ngữ (Assembly). Lập trình viên có thể tạo ra và bảo trì thư viện hàm mà chúng sẽ được tái sử dụng cho chương trình khác. Do đó, những dự án lớn có thể được quản lý dễ dàng mà tốn rất ít công sức.

i) C – Ngôn ngữ bậc trung

C được hiểu là ngôn ngữ bậc trung bởi vì nó kết hợp những yếu tố của những ngôn ngữ cấp cao và những chức năng của hợp ngữ (ngôn ngữ cấp thấp). C cho phép thao tác trên những thành phần cơ bản của máy tính như bits, bytes, địa chỉ…. Hơn nữa, mã C rất dễ di chuyển nghĩa là phần mềm viết cho loại máy tính này có thể chạy trên một loại máy tính khác. Mặc dù C có năm kiểu dữ liệu cơ bản, nhưng nó không được xem ngang hàng với ngôn ngữ cao cấp về mặt kiểu dữ liệu. C cho phép chuyển kiểu dữ liệu. Nó cho phép thao tác trực tiếp trên bits, bytes, word và con trỏ (pointer). Vì vậy, nó được dùng cho lập trình mức hệ thống.

ii) C - Ngôn ngữ cấu trúc

Thuật ngữ ngôn ngữ cấu trúc khối (block-structured language) không áp dụng với C. Ngôn ngữ cấu trúc khối cho phép thủ tục (procedures) hay hàm (functions) được khai báo bên trong các thủ tục và hàm khác. C không cho phép việc tạo hàm trong hàm nên nó không phải là ngôn ngữ cấu trúc khối. Tuy nhiên, nó được xem là ngôn ngữ cấu trúc vì nó có nhiều điểm giống với ngôn ngữ cấu trúc ALGOL, Pascal và một số ngôn ngữ tương tự khác.

C cho phép có sự tổng hợp của mã lệnh và dữ liệu. Điều này là một đặc điểm riêng biệt của ngôn ngữ cấu trúc. Nó liên quan đến khả năng tập hợp cũng như ẩn dấu tất cả thông tin và các lệnh khỏi phần còn lại của chương trình để dùng cho những tác vụ riêng biệt. Điều này có thể thực hiện qua việc dùng các hàm hay các khối mã lệnh (Code Block). Các **hàm** được dùng để định nghĩa hay tách rời những tác vụ được yêu cầu trong chương trình. Điều này cho phép những chương trình hoạt động như trong một đơn vị thống nhất. **Khối mã lệnh** là một nhóm các câu lệnh chương trình được nối

kết với nhau theo một trật tự logic nào đó và cũng được xem như một đơn vị thống nhất. Một khối mã lệnh được tạo bởi một tập hợp nhiều câu lệnh tuần tự giữa dấu ngoặc mở và đóng xoắn như dưới đây:

```
do
{
    i = i + 1;
    .
    .
    } while (i < 40);
```

Ngôn ngữ cấu trúc hỗ trợ nhiều cấu trúc dùng cho vòng lặp (loop) như là **while, do-while,** và **for.** Những cấu trúc lặp này giúp lập trình viên điều khiển hướng thực thi trong chương trình.

c) Cấu trúc chương trình C

C có một số từ khóa, chính xác là 32. Những từ khóa này kết hợp với cú pháp của C hình thành ngôn ngữ C. Nhưng nhiều trình biên dịch cho C đã thêm vào những từ khóa dùng cho việc tổ chức bộ nhớ ở những giai đoạn tiền xử lý nhất định.

Vài quy tắc khi lập trình C như sau :

- Tất cả từ khóa là chữ thường (không in hoa)
- Đoạn mã trong chương trình C có phân biệt chữ thường và chữ hoa.
  Ví dụ : **do while** thì khác với **DO WHILE**
- Từ khóa không thể dùng cho các mục đích khác như đặt tên biến (variable name) hoặc tên hàm (function name)
- Hàm main() luôn là hàm đầu tiên được gọi đến khi một chương trình bắt đầu chạy (chúng ta sẽ xem xét kỹ hơn ở phần sau)

Xem xét đoạn mã chương trình:

```
main ()
{
    /* This is a sample program */
    int i = 0;
    i = i + 1;
    .
    .
    .
}
```

Ghi chú: Những khía cạnh khác nhau của chương trình C được xem xét qua đoạn mã trên. Đoạn mã này xem như là đoạn mã mẫu, nó sẽ được dùng lại trong suốt phần còn lại của giáo trình này.

i) Định nghĩa Hàm

Chương trình C được chia thành từng đơn vị gọi là hàm. Đoạn mã mẫu chỉ có duy nhất một hàm main(). Hệ điều hành luôn trao quyền điều khiển cho hàm main() khi một chương trình C được thực thi. Tên hàm luôn được theo sau là cặp dấu ngoặc đơn (). Trong dấu ngoặc đơn có thể có hay không có những tham số (parameters).

ii) Dấu phân cách (Delimiters)

Sau định nghĩa hàm sẽ là dấu ngoặc xoắn mở **{**. Nó thông báo điểm bắt đầu của hàm. Tương tự, dấu ngoặc xoắn đóng **}** sau câu lệnh cuối trong hàm chỉ ra điểm kết thúc của hàm. Dấu ngoặc xoắn mở đánh dấu điểm bắt đầu của một khối mã lệnh, dấu ngoặc xoắn đóng đánh dấu điểm kết thúc của khối mã lệnh đó. Trong đoạn mã mẫu có 2 câu lệnh giữa 2 dấu ngoặc xoắn.

Hơn nữa, đối với hàm, dấu ngoặc xoắn cũng dùng để phân định những đoạn mã trong trường hợp dùng cho cấu trúc vòng lặp và lệnh điều kiện..

iii) Dấu kết thúc câu lệnh (Terminator)

Dòng **int i = 0;** trong đoạn mã mẫu là một câu lệnh (statement). Một câu lệnh trong C thì được kết thúc bằng dấu chấm phẩy (**;**). C không hiểu việc xuống dòng dùng phím Enter, khoảng trắng dùng phím spacebar hay một khoảng cách do dùng phím tab. Có thể có nhiều hơn một câu lệnh trên cùng một hàng nhưng mỗi câu lệnh phải được kết thúc bằng dấu chấm phẩy. Một câu lệnh không được kết thúc bằng dấu chấm phẩy được xem như một câu lệnh sai.

iv) Dòng chú thích (Comment)

Những chú thích thường được viết để mô tả công việc của một lệnh đặc biệt, một hàm hay toàn bộ chương trình. Trình biên dịch sẽ không dịch chúng. Trong C, chú thích bắt đầu bằng ký hiệu /* và kết thúc bằng */. Trường hợp chú thích có nhiều dòng, ta phải chú ý ký hiệu kết thúc (*/), nếu thiếu ký hiệu này, toàn bộ chương trình sẽ bị coi như là một chú thích. Trong đoạn mã mẫu dòng chữ "This is a sample program" là dòng chú thích. Trong trường hợp chú thích chỉ trên một dòng ta có thể dùng //.

Ví dụ:

int a = 0; // Biến ‘a’ đã được khai báo như là một kiểu số nguyên (interger)

v) Thư viện C (Library)

Tất cả trình biên dịch C chứa một thư viện hàm chuẩn dùng cho những tác vụ chung. Một vài bộ cài đặt C đặt thư viện trong một tập tin (file) lớn trong khi đa số còn lại chứa nó trong nhiều tập tin nhỏ. Khi lập trình, những hàm được chứa trong thư viện có thể được dùng cho nhiều loại tác vụ khác nhau. Một hàm (được viết bởi một lập trình viên) có thể được đặt trong thư viện và được dùng bởi nhiều chương trình khi

được yêu cầu. Vài trình biên dịch cho phép hàm được thêm vào thư viện chuẩn trong khi số khác lại yêu cầu tạo một thư viện riêng.

d) Biên dịch và thực thi một chương trình

Những bước khác nhau của việc dịch một chương trình C từ mã nguồn thành mã thực thi được thực hiện như sau :

- **Soạn thảo/Xử lý từ**

Ta dùng một trình xử lý từ (word processor) hay trình soạn thảo (editor) để viết mã nguồn (source code). C chỉ chấp nhận loại mã nguồn viết dưới dạng tập tin văn bản chuẩn. Vài trình biên dịch (compiler) cung cấp môi trường lập trình (xem phụ lục) gồm trình soạn thảo.

- **Mã nguồn**

Đây là đoạn văn bản của chương trình mà người dùng có thể đọc. Nó là đầu vào của trình biên dịch C.

- **Bộ tiền xử lý C**

Từ mã nguồn, bước đầu tiên là chuyển nó qua bộ tiền xử lý của C. Bộ tiền xử lý này sẽ xem xét những câu lệnh bắt đầu bằng dấu #. Những câu lệnh này gọi là các chỉ thị tiền biên dịch (directives). Điều này sẽ được giải thích sau. Chỉ thị tiền biên dịch thường được đặt nơi bắt đầu chương trình mặc dù nó có thể được đặt bất cứ nơi nào khác. Chỉ thị tiền biên dịch là những tên ngắn gọn được gán cho một tập mã lệnh.

- **Mã nguồn mở rộng C**

Bộ tiền xử lý của C khai triển các chỉ thị tiền biên dịch và đưa ra kết quả. Đây gọi là mã nguồn C mở rộng, sau đó nó được chuyển cho trình biên dịch C.

- **Trình biên dịch C (Compiler)**

Trình biên dịch C dịch mã nguồn mở rộng thành ngôn ngữ máy để máy tính hiểu được.

Nếu chương trình quá lớn nó có thể được chia thành những tập tin riêng biệt và mỗi tập tin có thể được biên dịch riêng rẽ. Điều này giúp ích khi mà một tập tin bị thay đổi, toàn chương trình không phải biên dịch lại.

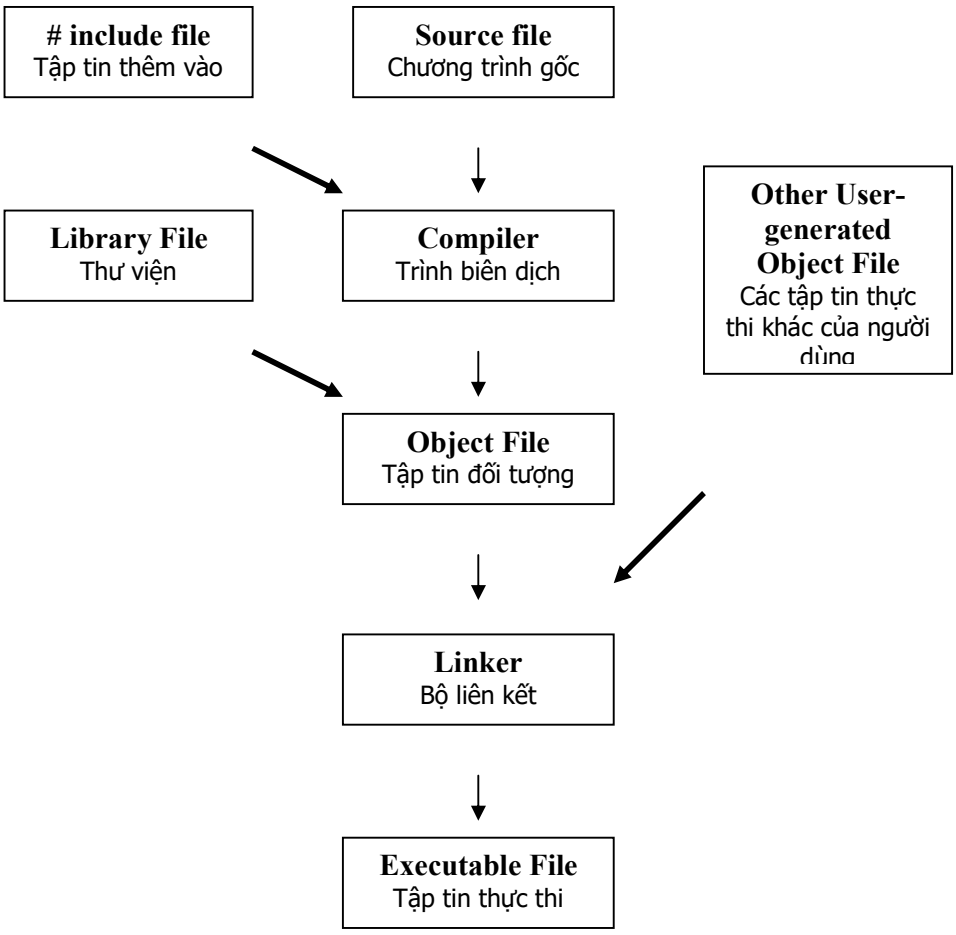
- **Bộ liên kết (Linker)**

Mã đối tượng cùng với những thủ tục hỗ trợ trong thư viện chuẩn và những hàm được dịch riêng lẻ khác kết nối lại bởi Bộ liên kết để cho ra mã có thể thực thi được.

- **Bộ nạp (Loader)**

Mã thực thi được thi hành bởi bộ nạp của hệ thống.

Tiến trình trên được mô tả qua lưu đồ 1.2 sau :



Hình 1.2: Biên dịch và thực thi một chương trình

e) Các bước lập trình giải quyết vấn đề

Chúng ta thường gặp phải những bài toán. Để giải quyết những bài toán đó, chúng ta cần hiểu chúng trước rồi sau đó mới hoạch định các bước cần làm .

Giả sử chúng ta muốn đi từ phòng học đến quán ăn tự phục vụ ở tầng hầm. Để thực hiện việc này chúng ta cần **hiểu** nó rồi tìm ra **các bước giải quyết** trước khi thực thi các bước đó:

BƯỚC 1 : Rời phòng

BƯỚC 2 : Đến cầu thang

BƯỚC 3 : Xuống tầng hầm

BƯỚC 4 : Đi tiếp đến quán ăn tự phục vụ

Thủ tục trên liệt kê tập hợp các bước thực hiện được xác định rõ ràng cho việc giải quyết vấn đề. Một tập hợp các bước như vậy gọi là giải thuật (Algorithm hay gọi vắn tắt là algo).

Một giải thuật (còn gọi là thuật toán) có thể được định nghĩa như là một thủ tục, công thức hay cách giải quyết vấn đề. Nó gồm một tập hợp các bước giúp đạt được lời giải.

Qua phần trên, chúng ta thấy rõ ràng để giải quyết được một bài toán, trước tiên ta phải hiểu bài toán đó, kế đến chúng ta cần tập hợp tất cả những thông tin liên quan tới nó. Bước kế sẽ là xử lý những mẩu thông tin đó. Cuối cùng, chúng ta cho ra lời giải của bài toán đó.

Giải thuật chúng ta có là một tập hợp các bước được liệt kê dưới dạng ngôn ngữ đơn giản. Rất có thể rằng các bước trên do hai người khác nhau viết vẫn tương tự nhau nhưng ngôn ngữ dùng diễn tả các bước có thể khác nhau. Do đó, cần thiết có những phương pháp chuẩn mực cho việc viết giải thuật để mọi người dễ dàng hiểu nó. Chính vì vậy , giải thuật được viết bằng cách dùng hai phương pháp chuẩn là mã giả (pseudo code) và lưu đồ (flowchart).

Cả hai phương pháp này đều dùng để xác định một tập hợp các bước cần được thi hành để có được lời giải. Liên hệ tới vấn đề đi đến quán ăn tự phục vụ trên, chúng ta đã vạch ra một kế hoạch (thuật toán) để đến đích. Tuy nhiên, để đến nơi, chúng ta phải cần thi hành những bước này thật sự. Tương tự, mã giả và lưu đồ chỉ đưa ra những bước cần làm. Lập trình viên phải viết mã cho việc thực thi những bước này qua việc dùng một ngôn ngữ nào đó.

Chi tiết về về mã giả và lưu đồ được trình bày dưới đây.

i) Mã giả (pseudo code)

Nhớ rằng mã giả không phải là mã thật. Mã giả sử dụng một tập hợp những từ tương tự như mã thật nhưng nó không thể được biên dịch và thực thi như mã thật.

Chúng ta hãy xem xét mã giả qua ví dụ sau:

Ví dụ này sẽ hiển thị câu 'Hello World!'.

Ví dụ 1:

```
BEGIN
DISPLAY 'Hello World!'
END
```

Qua ví dụ trên, mỗi đoạn mã giả phải bắt đầu với từ BEGIN hoặc START, và kết thúc với từ END hay STOP. Để hiển thị giá trị nào đó, từ DISPLAY hoặc WRITE được dùng. Khi giá trị được hiển thị là một giá trị hằng (không đổi), trong trường hợp này là (Hello World), nó được đặt bên trong dấu nháy. Tương tự, để nhận một giá trị của người dùng, từ INPUT hay READ được dùng.

Để hiểu điều này rõ hơn, chúng ta xem xét ví dụ 2, ở ví dụ này ta sẽ nhập hai số và máy sẽ hiển thị tổng của hai số.

Ví dụ 2:

```
BEGIN
INPUT A, B
DISPLAY A + B
END
```

Trong đoạn mã giả này, người dùng nhập vào hai giá trị, hai giá trị này được lưu trong bộ nhớ và có thể được truy xuất như là A và B theo thứ tự. Những vị trí được đặt tên như vậy trong bộ nhớ gọi là biến. Chi tiết về biến sẽ được giải thích trong phần sau của chương này. Bước kế tiếp trong đoạn mã giả sẽ hiển thị tổng của hai giá trị trong biến A và B.

Tuy nhiên, cũng đoạn mã trên, ta có thể bổ sung để lưu tổng của hai biến trong một biến thứ ba rồi hiển thị giá trị biến này như trong ví dụ 3 sau đây.

Ví dụ 3:

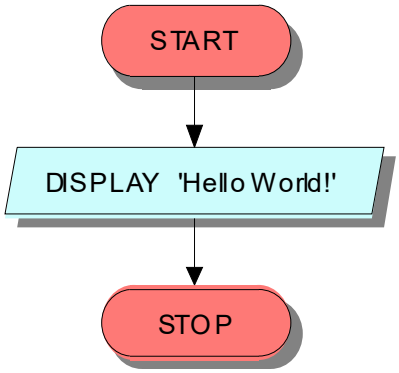
```
BEGIN
INPUT A, B
C = A + B
DISPLAY C
END
```

Một tập hợp những chỉ thị hay các bước trong mã giả thì được gọi chung là một cấu trúc. Có ba loại cấu trúc : tuần tự, chọn lựa và lặp lại. Trong đoạn mã giả ta viết ở trên,chúng ta dùng cấu trúc tuần tự. Chúng được gọi như vậy vì những chỉ thị được thi hành tuần tự, cái này sau cái khác và bắt đầu từ điểm đầu tiên. Hai loại cấu trúc còn lại sẽ được đề cập trong những chương sau.

ii) Lưu đồ (Flowcharts)

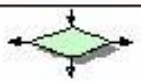
Một lưu đồ là một hình ảnh minh hoạ cho giải thuật. Nó vẽ ra biểu đồ của luồng chỉ thị hay những hoạt động trong một tiến trình. Mỗi hoạt động như vậy được biểu diễn qua những ký hiệu.

Để hiểu điều này rõ hơn, chúng ta xem lưu đồ trong hình 1.3 dùng hiển thị thông điệp truyền thống 'Hello World!'.



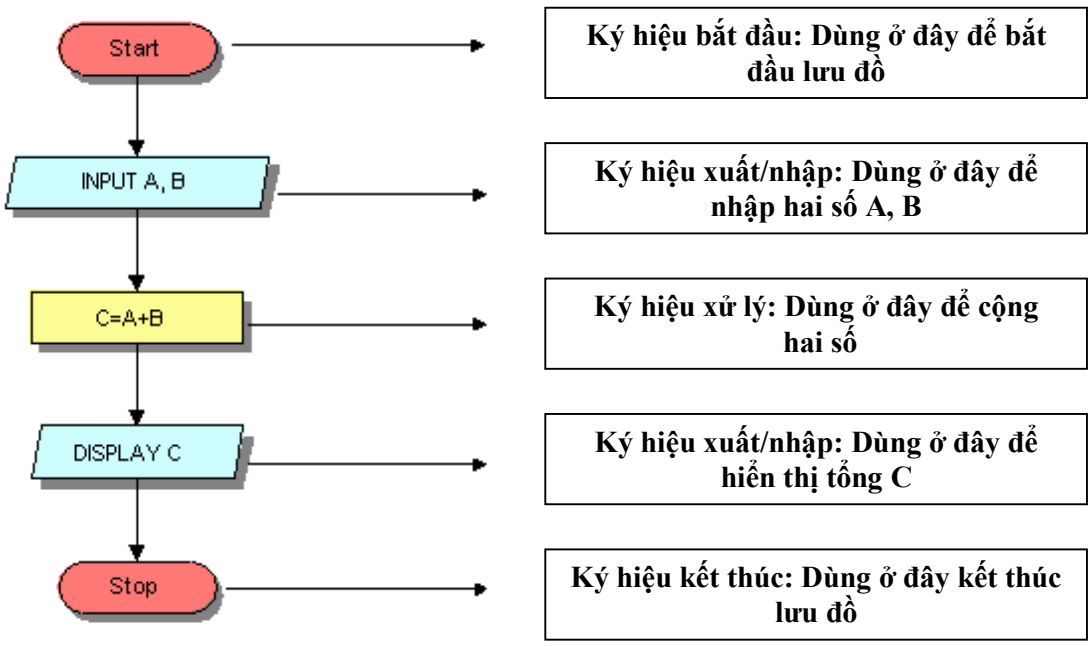
Hình 1.3: Lưu đồ

Lưu đồ giống với đoạn mã giả là cùng bắt đầu với từ BEGIN hoặc START, và kết thúc với từ END hay STOP. Tương tự, từ khóa DISPLAY được dùng để hiển thị giá trị nào đó đến người dùng. Tuy nhiên, ở đây, mọi từ khóa thì nằm trong những ký hiệu. Những ký hiệu khác nhau mang một ý nghĩa tương ứng được trình bày ở bảng trong Hình 1.4.

Biểu Tượng	Mô Tả
	Bắt đầu hay kết thúc chương trình
	Những bước tính toán
	Các lệnh xuất hay nhập
	Quyết định và rẽ nhánh
	Bộ nối hai phần trong chương trình (đầu nối)
	Dòng chảy

Hình 1.4: Ký hiệu trong lưu đồ

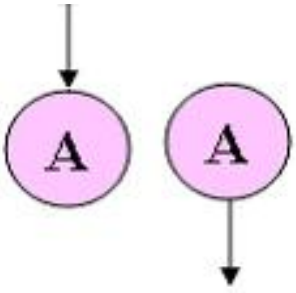
Ta hãy xét lưu đồ cho ví dụ 3 như ở Hình 1.5 dưới đây.



Hình 1.5: Lưu đồ cộng hai số

Tại bước mà giá trị của hai biến được cộng và gán cho biến thứ ba thì xem như là một xử lý và được trình bày bằng một hình chữ nhật.

Lưu đồ mà chúng ta xét ở đây là đơn giản.Thông thường, lưu đồ trải rộng trên nhiều trang giấy. Trong trường hợp như thế, biểu tượng bộ nối được dùng để chỉ điểm nối của hai phần trong một chương trình nằm ở hai trang kế tiếp nhau. Vòng tròn chỉ sự nối kết và phải chứa ký tự hoặc số như ở hình 1.6. Như thế, chúng ta có thể tạo liên kết giưa hai lưu đồ chưa hoàn chỉnh.



Hình 1.6: Bộ nối

Bởi vì lưu đồ được sử dụng để viết chương trình, chúng cần được trình bày sao cho mọi lập trình viên hiểu chúng dễ dàng. Nếu có ba lập trình viên dùng ba ngôn ngữ lập trình khác nhau để viết mã, bài toán họ cần giải quyết phải như nhau. Trong trường hợp này, mã giả đưa cho lập trình viên có thể giống nhau mặc dù ngôn ngữ lập trình họ dùng và tất nhiên là cú pháp có thể khác nhau. Nhưng kết quả cuối cùng là một. Do đó, cần thiết phải hiểu rõ bài toán và mã giả phải được viết cẩn thận. Chúng ta cũng kết luận rằng mã giả độc lập với ngôn ngữ lập trình.

Vài điểm cần thiết khác phải chú ý khi vẽ một lưu đồ :

➥ Lúc đầu chỉ tập trung vào khía cạnh logic của bài toán và vẽ các luồng xử lý chính của lưu đồ

➥ Một lưu đồ phải có duy nhất một điểm bắt đầu (START) và một điểm kết thúc (STOP).

➥ Không cần thiết phải mô tả từng bước của chương trình trong lưu đồ mà chỉ cần các bước chính và có ý nghĩa cần thiết.

Chúng ta tuân theo những cấu trúc tuần tự, mà trong đó luồng thực thi chương trình đi qua tất cả các chỉ thị bắt đầu từ chỉ thị đầu tiên. Chúng ta có thể bắt gặp các điều kiện trong chương trình, dựa trên các điều kiện này hướng thực thi của chương trình có thể rẽ nhánh. Những cấu trúc cho việc rẽ nhánh như là cấu trúc chọn lựa, cấu trúc điều kiện hay rẽ nhánh. Những cấu trúc này được đề cập chi tiết sau đây:

- **Cấu trúc IF (Nếu)**

Cấu trúc chọn lựa cơ bản là cấu trúc 'IF'. Để hiểu cấu trúc này chúng ta hãy xem xét ví dụ trong đó khách hàng được giảm giá nếu mua trên 100 đồng. Mỗi lần khách hàng trả tiền, một đoạn mã chương trình sẽ kiểm tra xem lượng tiền trả có quá 100 đồng không?. Nếu đúng thế thì sẽ giảm giá 10% của tổng số tiền trả, ngược lại thì không giảm giá.

Điều này được minh họa sơ lược qua mã giả như sau:

IF *khách hàng mua trên 100 thì giảm giá 10%*

Cấu trúc dùng ở đây là câu lệnh IF.

Hình thức chung cho câu lệnh IF (cấu trúc IF) như sau:

IF *Điều kiện*

Các câu lệnh ⟶ *Phần thân của cấu trúc* IF

END IF

Một cấu trúc 'IF' bắt đầu là IF theo sau là điều kiện. Nếu điều kiện là đúng (thỏa điều kiện) thì quyền điều khiển sẽ được chuyển đến các câu lệnh trong phần thân để thực thi. Nếu điều kiện sai (không thỏa điều kiện), những câu lệnh ở phần thân không được thực thi và chương trình nhảy đến câu lệnh sau END IF (chấm dứt cấu trúc IF). Cấu trúc IF phải được kết thúc bằng END IF.

Chúng ta xem ví dụ 4 cho cấu trúc IF.

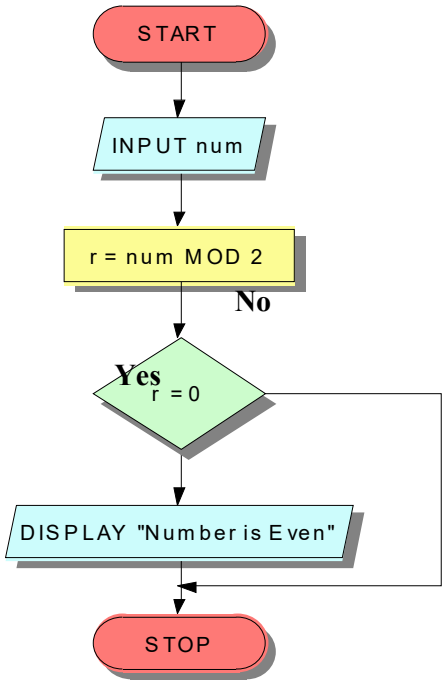
Ví dụ 4:

Yêu cầu: Kiểm xem một số là chẵn hay không và hiển thị thông điệp báo nếu đúng là số chẵn, ta xử lý như sau :

BEGIN
INPUT num
r = num MOD 2
IF r=0
Display "Number is even"
END IF
END

Đoạn mã trên nhập một số từ người dùng, thực hiện toán tử MOD (lấy phần dư) và kiểm tra xem phần dư có bằng 0 hay không. Nếu bằng 0 hiển thị thông điệp, ngược lại thoát ra.

Lưu đồ cho đoạn mã giả trên thể hiện qua hình 1.7.



Hình 1.7 : Kiểm tra số chẵn

Cú pháp của lệnh IF trong C như sau:

```
if (Điều kiện)
{
        Câu lệnh
}
```

Cấu trúc IF…ELSE

Trong ví dụ 4, sẽ hay hơn nếu ta cho ra thông điệp báo rằng số đó không là số chẵn tức là số lẻ thay vì chỉ thoát ra. Để làm điều này ta có thể thêm câu lệnh IF khác để kiểm tra xem trường hợp số đó không chia hết cho 2.

Ta xem ví dụ 5:

```
BEGIN
INPUT num
r = num MOD 2
IF r=0
DISPLAY “Even number”
END IF

IF r<>0
    DISPLAY “Odd number”
END IF
END
```

Ngôn ngữ lập trình cung cấp cho chúng ta cấu trúc **IF…ELSE**. Dùng cấu trúc này sẽ hiệu quả và tốt hơn để giải quyết vấn đề. Cấu trúc **IF …ELSE** giúp lập trình viên chỉ làm một phép so sánh và sau đó thực thi các bước tùy theo kết quả của phép so sánh là True (đúng) hay False (sai).

Cấu trúc chung của câu lệnh IF…ELSE như sau:

IF Đ*iều kiện*
    Câu lệnh 1
ELSE
    Câu lệnh 2
END IF

Cú pháp của cấu trúc if…else trong C như sau:

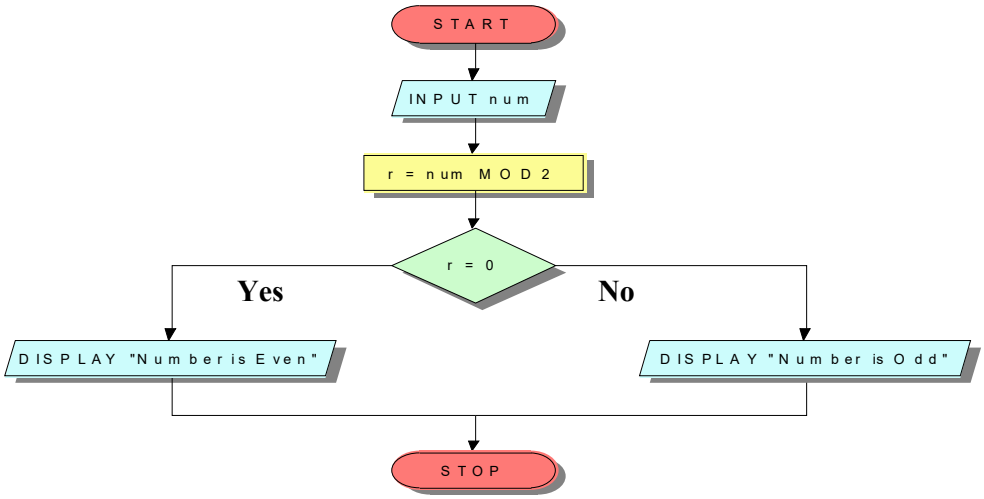
```
if(Điều kiện)
{
    Câu lệnh 1
}
else
{
    Câu lệnh 2
}
```

Nếu điều kiện thỏa (True), câu lệnh 1 được thực thi. Ngược lại, câu lệnh 2 được thực thi. Không bao giờ cả hai được thực thi cùng lúc. Vì vậy, đoạn mã tối ưu hơn cho ví dụ tìm số chẵn được viết ra như ví dụ 6.

Ví dụ 6:

```
BEGIN
INPUT num
r = num MOD 2
IF r = 0
   DISPLAY “Even Number”
ELSE
   DISPLAY “Odd Number”
END IF
END
```

Lưu đồ cho đoạn mã giả trên thể hiện qua Hình 1.8.



Hình 1.8: Số chẵn hay số lẻ

- **Đa điều kiện sử dụng AND/OR**

Cấu trúc IF…ELSE làm giảm độ phức tạp, gia tăng tính hữu hiệu. Ở một mức độ nào đó, nó cũng nâng cao tính dễ đọc của mã. Các thí dụ IF chúng ta đã đề cập đến thời điểm này thì khá đơn giản. Chúng chỉ có một điều kiện trong IF để đánh giá. Thỉnh thoảng chúng ta phải kiểm tra cho hơn một điều kiện, thí dụ: Để xem xét nhà cung cấp có đạt MVS (nhà cung cấp quan trọng nhất) không?, một công ty sẽ kiểm tra xem nhà cung cấp đó đã có làm việc với công ty ít nhất 10 năm không? và đã có tổng doanh thu ít nhất 5,000,000 không?. Hai điều kiện thỏa mãn thì nhà cung cấp được xem như là một MVS. Do đó toán tử AND có thể được dùng trong câu lệnh ‘IF’ như trong ví dụ 7 sau:

Ví dụ 7:

```
BEGIN
INPUT yearsWithUs
INPUT bizDone
IF yearsWithUs >= 10 AND bizDone >=5000000
  DISPLAY “Classified as an MVS”
ELSE
  DISPLAY “A little more effort required!”
END IF
END
```

Ví dụ 7 cũng khá đơn giản, vì nó chỉ có 2 điều kiện. Ở tình huống thực tế, chúng ta có thể có nhiều điều kiện cần được kiểm tra. Nhưng chúng ta có thể dễ dàng dùng toán tử AND để nối những điều kiện lại giống như ta đã làm ở trên.

Bây giờ, giả sử công ty trong ví dụ trên đổi quy định, họ quyết định đưa ra điều kiện dễ dàng hơn. Như là : Hoặc làm việc với công ty trên 10 năm hoặc có doanh số (giá trị thương mại,giao dịch) từ 5,000,000 trở lên. Vì vậy, ta thay thế toán tử AND bằng toán tử OR. Nhớ rằng toán tử OR cho ra giá trị True (đúng) nếu chỉ cần một điều kiện là True.

- **Cấu trúc IF lồng nhau**

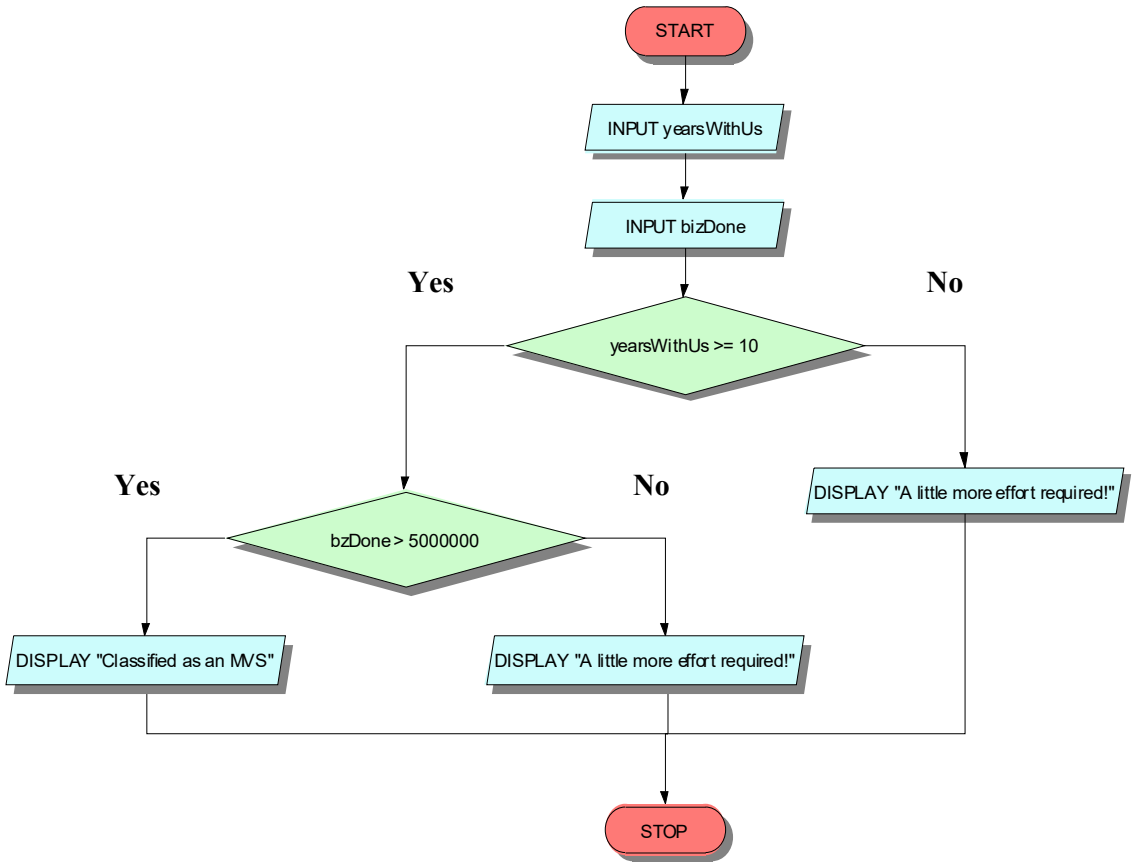
Một cách khác để thực hiện ví dụ 7 là sử dụng cấu trúc IF lồng nhau. Cấu trúc IF lồng nhau là câu lệnh IF này nằm trong trong câu lệnh IF khác. Chúng ta viết lại ví dụ 7 sử dụng cấu trúc IF lồng nhau ở ví dụ 8 như sau:

Ví dụ 8:

```
BEGIN
INPUT yearsWithUs
INPUT bizDone
IF yearsWithUs >= 10
    IF bizDone >=5000000
        DISPLAY “Classified as an MVS”
        ELSE
    DISPLAY “A little more effort required!”
    END IF
ELSE
            DISPLAY “A little more effort required!”
END IF
END
```

Đoạn mã trên thực hiện cùng nhiệm vụ nhưng không có 'AND'. Tuy nhiên, chúng ta có một lệnh IF (kiểm tra xem bizDone lớn hơn hoặc bằng 5,000,000 hay không?) bên trong lệnh IF khác (kiểm tra xem yearsWithUs lớn hơn hoặc bằng 10 hay không?). Câu lệnh IF đầu tiên kiểm tra điều kiện thời gian nhà cung cấp làm việc với công ty có lớn hơn 10 năm hay không. Nếu dưới 10 năm (kết quả trả về là False), nó sẽ không công nhận nhà cung cấp là một MVS; Nếu thỏa điều kiện nó xét câu lệnh IF thứ hai, nó sẽ kiểm tra tới điều kiện bizDone lớn hơn hoặc bằng 5,000,000 hay không. Nếu thỏa điều kiện (kết quả trả về là True) lúc đó nhà cung cấp được xem là một MVS, nếu không thì một thông điệp báo rằng đó không là một MVS.

Lưu đồ cho mã giả của ví dụ 8 được trình bày qua hình 1.9.



Hình 1.9: Câu lệnh IF lồng nhau

Mã giả trong trường hợp này của cấu trúc IF lồng nhau tại ví dụ 8 chưa hiệu quả. Câu lệnh thông báo không thỏa điều kiện MVS phải viết hai lần. Hơn nữa lập trình viên phải viết thêm mã nên trình biên dịch phải xét hai điều kiện của lệnh IF, do đó lãng phí thời gian. Ngược lại, nếu dùng toán tử AND chỉ xét tới điều kiện của câu

lệnh IF một lần. Điều này không có nghĩa là cấu trúc IF lồng nhau nói chung là không hiệu quả. Nó tùy theo tình huống cụ thể mà ta dùng nó. Có khi dùng toán tử AND hiệu quả hơn, có khi dùng cấu trúc IF lồng nhau hiệu quả hơn. Chúng ta sẽ xét một ví dụ mà dùng cấu trúc IF lồng nhau hiệu quả hơn dùng toán tử AND.

Một công ty định phần lương cơ bản cho công nhân dựa trên tiêu chuẩn như trong bảng 1.1.

Grade	**Experience**	**Salary**
E	2	2000
E	3	3000
M	2	3000
M	3	4000

Bảng 1.1: Lương cơ bản

Vì vậy, nếu một công nhân được xếp loại là E và có hai năm kinh nghiệm thì lương là 2000, nếu ba năm kinh nghiệm thì lương là 3000.
Mã giả dùng toán tử AND cho vấn đề trên như ví dụ 9:

Ví dụ 9:

```
BEGIN
INPUT grade
INPUT exp
IF grade =”E” AND exp =2
    salary=2000
ELSE
    IF grade = “E” AND exp=3
        salary=3000
    END IF
END IF
IF grade =”M” AND exp =2
salary=3000
ELSE
IF grade = “M” AND exp=3
        salary=4000
            END IF
END IF
END
```

Câu lệnh IF đầu tiên kiểm tra xếp loại và kinh nghiệm của công nhân. Nếu xếp loại là E và kinh nghiệm là 2 năm thì lương là 2000, ngoài ra nếu xếp loại E, nhưng có 3 năm kinh nghiệm thì lương là 3000.

Nếu cả 2 điều kiện không thỏa thì câu lệnh IF thứ hai cũng tương tự sẽ kiểm điều kiện xếp loại và kinh nghiệm cho công nhân để phân định lương.

Giả sử xếp loại của một công nhân là E và có hai năm kinh nghiệm. Lương người đó sẽ được tính theo mệnh đề IF đầu tiên. Phần còn lại của câu lệnh IF thứ nhất được bỏ qua. Tuy nhiên, điều kiện tại mệnh đề IF thứ hai sẽ được xét và tất nhiên là không thỏa, do đó nó kiểm tra mệnh đề ELSE của câu lệnh IF thứ 2 và kết quả cũng là False. Đây quả là những bước thừa mà chương trình đã xét qua. Trong ví dụ, ta chỉ có hai câu lệnh IF bởi vì ta chỉ xét có hai loại là E và M. Nếu có khoảng 15 loại thì sẽ tốn thời gian và tài nguyên máy tính cho việc tính toán thừa mặc dù lương đã xác định tại câu lệnh IF đầu tiên. Đây dứt khoát không phải là mã nguồn hiệu quả.

Bây giờ chúng ta xét mã giả dùng cấu trúc IF lồng nhau đã được sửa đổi trong ví dụ 10.

Ví dụ 10:

```
BEGIN
INPUT grade
INPUT exp
IF grade=”E”
        IF exp=2
                salary = 2000
        ELSE
                IF exp=3
                        salary=3000
                END IF
        END IF
ELSE
        IF grade=”M”
                IF exp=2
                        Salary=3000
                ELSE
                        IF exp=3
                                Salary=4000
                        END IF
                END IF
        END IF
```

END IF
END

Đoạn mã trên nhìn khó đọc. Tuy nhiên, nó đem lại hiệu suất cao hơn. Chúng ta xét cùng ví dụ như trên. Nếu công nhân được xếp loại là E và kinh nghiệm là 2 năm thì lương được tính là 2000 ngay trong bước đầu của câu lệnh IF. Sau đó, chương trình sẽ thoát ra vì không cần thực thi thêm bất cứ lệnh ELSE nào. Do đó, không có sự lãng phí và đoạn mã này mang lại hiệu suất cho chương trình và chương trình chạy nhanh hơn.

- **Vòng lặp**

Một chương trình máy tính là một tập các câu lệnh sẽ được thực hiện tuần tự. Nó có thể lặp lại một số bước với số lần lặp xác định theo yêu cầu của bài toán hoặc đến khi một số điều kiện nhất định được thỏa.

Chẳng hạn, ta muốn viết chương trình hiển thị tên của ta 5 lần. Ta xét mã giả dưới đây.

Ví dụ 11:

BEGIN
DISPLAY “Scooby”
DISPLAY “Scooby”
DISPLAY “Scooby”
DISPLAY “Scooby”
DISPLAY “Scooby”
END

Nếu để hiển thị tên ta 1000 lần, nếu ta viết DISPLAY “Scooby” 1000 lần thì rất tốn công sức. Ta có thể tinh giản vấn đề bằng cách viết câu lệnh DISPLAY chỉ một lần, sau đó đặt nó trong cấu trúc vòng lặp, và chỉ thị máy tính thực hiện lặp 1000 lần cho câu lệnh trên.

Ta xem mã giả của cấu trúc vòng lặp trong ví dụ 12 như sau:

Ví dụ 12:

Do loop 1000 times
    DISPLAY “Scooby”
End loop

Những câu lệnh nằm giữa Do loop và End loop (trong ví dụ trên là lệnh

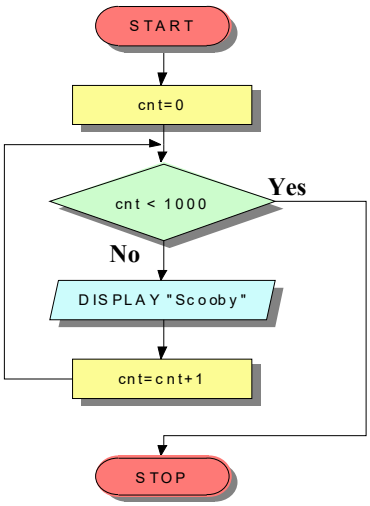
DISPLAY) được thực thi 1000 lần. Những câu lệnh này cùng với các lệnh **do loop** và **end loop** được gọi là cấu trúc vòng lặp. Cấu trúc vòng lặp giúp lập trình viên phát triển thành những chương trình lớn trong đó có thể yêu cầu thực thi hàng ngàn câu lệnh. **Do loop…end loop** là một dạng thức tổng quát của vòng lặp.

Ví dụ sau là cách viết khác nhưng cũng dùng cấu trúc vòng lặp.

Ví dụ 13:

```
BEGIN
cnt=0
WHILE (cnt < 1000)
DO
    DISPLAY “Scooby”
    cnt=cnt+1
END DO
END
```

Lưu đồ cho mã giả trong ví dụ 13 được vẽ trong Hình 1.10.



Hình 1.10: Cấu trúc vòng lặp

Chú ý rằng Hình 1.10 không có ký hiệu đặc biệt nào để biểu diễn cho vòng lặp. Chúng ta dùng ký hiệu phân nhánh để kiểm tra điều kiện và quản lý hướng đi của của chương trình bằng các dòng chảy (flow_lines).

3.2.4. Quy trình kỹ thuật trong bài học của GV và SV:

*** Quy trình thị phạm của GV**

- Bước 1: Chuẩn bị bài giảng, tài liệu và điều kiện trang thiết bị giảng dạy
- Bước 2: Chuẩn bị bài tập mẫu và đưa ra nội dung thực hành
- Bước 3: Phổ biến các quy trình, các bước thực hiện
- Bước 4: Hướng dẫn nội dung bài học, Hướng dẫn thị phạm mẫu
- Bước 5: Quản lý lớp học
- Bước 6: Giải đáp, tư vấn, đánh giá, nhận xét
- Bước 7: Chấm điểm và nộp các bài đánh giá cho bộ môn

*** Quy trình thực hiện bài của SV**

- Bước 1: SV chuẩn bị tài liệu, vở, bút, giấy nháp, tài liệu tra cứu
- Bước 2: Đọc tài liệu Lập trình căn bản C phần giới thiệu về ngôn ngữ C.
- Bước 3: Sinh viên trình bày bài thực hành, thảo luận và nhận xét góp ý.

3.2.5. Các mẫu hình sản phẩm cho SV tham khảo:

- SV xem video

3.2.6. Phần tự thực hành thao tác thường xuyên của SV trong bài học:

- Phần mềm là một tập hợp các chương trình.
- Một chương trình là một tập hợp các chỉ thị (lệnh).
- Những đoạn mã lệnh là cơ sở cho bất kỳ một chương trình C nào.
- Ngôn ngữ C có 32 từ khóa.
- Các bước cần thiết để giải quyết một bài toán là nghiên cứu chi tiết bài toán đó, thu thập thông tin thích hợp, xử lý thông tin và đi đến kết quả.
- Một giải thuật là một danh sách rút gọn và logic các bước để giải quyết vấn đề. Giải thuật được viết bằng mã giả hoặc lưu đồ.
- Mã giả là sự trình bày của giải thuật trong ngôn ngữ tương tự như mã thật
- Một lưu đồ là sự trình bày dưới dạng biểu đồ của một giải thuật.
- Lưu đồ có thể chia nhỏ thành nhiều phần và đầu nối dùng cho việc nối chúng lại tại nơi chúng bị chia cắt.
- Một chương trình có thể gặp một điều kiện dựa theo đó việc thực thi có thể được phân theo các nhánh rẽ khác nhau. Cấu trúc lệnh như vậy gọi là cấu trúc chọn lựa, điều kiện hay cấu trúc rẽ nhánh.

- Cấu trúc chọn cơ bản là cấu trúc "IF".
- Cấu trúc **IF …ELSE** giúp lập trình viên chỉ làm so sánh đơn và sau đó thực thi các bước tùy theo kết quả của phép so sánh là True (đúng) hay False (sai).
- Cấu trúc IF lồng nhau là câu lệnh IF này nằm trong câu lệnh IF khác.
- Thông thường ta cần lặp lại một số bước với số lần lặp xác định theo yêu cầu của bài toán hoặc đến khi một số điều kiện nhất định được thỏa. Những cấu trúc giúp làm việc này gọi là cấu trúc vòng lặp.

3.2.7. Sản phẩm thực hành:

+ Các ví dụ demo.

3.2.8. Điều kiện để GV- SV thực hiện bài học thực hành;

3.2.8.1. Điều kiện chuẩn bị trước bài học

+ Giảng viên:

- Đề cương chi tiết bài giảng
- Thiết kế bài giảng
- Học liệu tham khảo, mở rộng

+ Sinh viên:

- Đọc kỹ bài học trước khi giảng viên lên lớp
- Chuẩn bị vấn đề thảo luận, trao đổi với giảng viên
- Các tài liệu tham khảo, mở rộng do giảng viên yêu cầu

3.2.8.2. Kỹ thuật và phương tiện dạy học:

- Bảng, phấn viết
- Màn hình lớn
- Máy tính, máy chiếu

3.2.8.3. Phương pháp tổ chức dạy-học:

- Thuyết trình, diễn giảng, thực hành
- Phát vấn
- Trao đổi, thảo luận
- Cemina
- Vấn đáp

3.3 Nội dung bài giảng 2

3.3.1 Tên bài giảng: BÀI 2: CÁC THÀNH PHẦN NGÔN NGỮ LẬP TRÌNH C

Số tiết lên lớp của GV: 05 tiết; Số tiết tự làm bài của SV : 05 tiết

3.3.2. Phần mở đầu tiếp cận bài;

Bất cứ chương trình ứng dụng nào cần xử lý dữ liệu cũng cần có nơi để lưu trữ tạm thời dữ liệu ấy. Nơi mà dữ liệu được lưu trữ gọi là bộ nhớ. Những vị trí khác nhau trong bộ nhớ có thể được xác định bởi các địa chỉ duy nhất. Những ngôn ngữ lập trình trước đây yêu cầu lập trình viên quản lý mỗi vị trí ô nhớ thông qua địa chỉ, cũng như giá trị lưu trong nó. Các lập trình viên dùng những địa chỉ này để truy cập hoặc thay đổi nội dung của các ô nhớ. Khi ngôn ngữ lập trình phát triển, việc truy cập hay thay đổi giá trị ô nhớ đã được đơn giản hoá nhờ sự ra đời của khái niệm biến .

3.3.3. Phần kiến thức, kỹ thuật căn bản:

3.3.3.1 Phần kiến thức căn bản

- **Mục tiêu:**

  Kết thúc bài học này, bạn có thể:

  - Hiểu và sử dụng được biến (variables)
  - Phân biệt sự khác nhau giữa biến và hằng (constants)
  - Nắm vững và sử dụng các kiểu dữ liệu khác nhau trong chương trình C
  - Hiểu và sử dụng các toán tử số học.

a) Biến

Một chương trình ứng dụng có thể quản lý nhiều loại dữ liệu. Trong trường hợp này, chương trình phải chỉ định bộ nhớ cho mỗi đơn vị dữ liệu. Khi chỉ định bộ nhớ, có hai điểm cần lưu ý như sau :

1. Bao nhiêu bộ nhớ sẽ được gán
2. Mỗi đơn vị dữ liệu được lưu trữ ở đâu trong bộ nhớ.

Trước đây, các lập trình viên phải viết chương trình theo ngôn ngữ máy gồm các mã 1 và 0. Nếu muốn lưu trữ một giá trị tạm thời, vị trí chính xác nơi mà dữ liệu được lưu trữ trong bộ nhớ máy tính phải được chỉ định. Vị trí này là một con số cụ thể, gọi là địa chỉ bộ nhớ.

Các ngôn ngữ lập trình hiện đại cho phép chúng ta sử dụng các tên tượng trưng gọi là biến (variable), chỉ đến một vùng bộ nhớ nơi mà các giá trị cụ thể được lưu trữ.

Kiểu dữ liệu quyết định tổng số bộ nhớ được chỉ định. Những tên được gán cho biến giúp chúng ta sử dụng lại dữ liệu khi cần đến.

Chúng ta đã quen với cách sử dụng các ký tự đại diện trong một công thức. Ví dụ, diện tích hình chữ nhật được tính bởi :

Diện tích = A = chiều dài x chiều rộng = L x B

Cách tính lãi suất đơn giản được cho như sau:

Tiền lãi = I = Số tiền ban đầu x Thời gian x Tỷ lệ/100 = P x T x R /100

Các ký tự A, L, B, I, P, T, R là các biến và là các ký tự viết tắt đại diện cho các giá trị khác nhau.

Xem ví dụ sau đây :

Tính tổng điểm cho 5 sinh viên và hiển thị kết quả. Việc tính tổng được thực hiện theo hướng dẫn sau.

Hiển thị giá trị tổng của 24, 56, 72, 36 và 82

Khi giá trị tổng được hiển thị, giá trị này không còn được lưu trong bộ nhớ máy tính. Giả sử, nếu chúng ta muốn tính điểm trung bình, thì giá trị tổng đó phải được tính một lần nữa.

Tốt hơn là chúng ta sẽ lưu kết quả vào bộ nhớ máy tính, và sẽ lấy lại nó khi cần đến.

sum = 24 + 56 + 72 + 36 + 82

Ở đây, ***sum* là biến** được dùng để chứa tổng của 5 số. Khi cần tính điểm trung bình, có thể thực hiện như sau:

Avg = sum / 5

Trong C, tất cả biến cần phải được khai báo trước khi dùng chúng.

Chúng ta hãy xét ví dụ nhập hai số và hiển thị tổng của chúng trong ví dụ 1.

Ví dụ 1:

```
BEGIN
DISPLAY ‘Enter 2 numbers’
INPUT A, B
C = A + B
DISPLAY C
END
```

A, B và C trong đoạn mã trên là các biến. Tên biến giúp chúng ta tránh phải nhớ địa chỉ của vị trí bộ nhớ. Khi đoạn mã được viết và thực thi, hệ điều hành đảm nhiệm việc cấp không gian nhớ còn trống cho những biến này. Hệ điều hành ánh xạ một tên biến đến một vị trí xác định trong bộ nhớ (ô nhớ). Và để tham chiếu tới một giá trị riêng biệt trong bộ nhớ, chúng ta chỉ cần chỉ ra tên của biến. Trong ví dụ trên, giá trị của hai biến được nhập từ người dùng và chúng được lưu trữ nơi nào đó trong bộ nhớ. Những vị trí này có thể được truy cập thông qua các tên biến A và B. Trong bước kế tiếp, giá trị của hai biến được cộng và kết quả được lưu trong biến thứ 3 là biến C. Cuối cùng, giá trị biến C được hiển thị.

Trong khi một vài ngôn ngữ lập trình cho phép hệ điều hành xóa nội dung trong ô nhớ và cấp phát bộ nhớ này để dùng lại thì những ngôn ngữ khác như C yêu cầu lập

trình viên xóa vùng nhớ không sử dụng thông qua mã chương trình. Trong cả hai trường hợp, hệ điều hành đều lo việc cấp phát và thu hồi ô nhớ.

Hệ điều hành hoạt động như một giao diện giữa các ô nhớ và lập trình viên. Lập trình viên không cần lưu tâm về vị trí ô nhớ mà để cho hệ điều hành đảm nhiệm. Vậy việc điều khiển bộ nhớ (vị trí mà dữ liệu thích hợp lưu trữ) sẽ do hệ điều hành đảm trách, chứ không phải lập trình viên.

b) Hằng

Trong trường hợp ta dùng biến, giá trị được lưu sẽ thay đổi. Một biến tồn tại từ lúc khai báo đến khi thoát khỏi phạm vi dùng nó. Những câu lệnh trong phạm vi khối mã này có thể truy cập giá trị của biến, và thậm chí có thể thay đổi giá trị của biến. Trong thực tế, đôi khi cần sử dụng một vài khoản mục mà giá trị của chúng không bao giờ bị thay đổi.

Một hằng là một giá trị không bao giờ bị thay đổi. Ví dụ, *5* là một hằng, mà giá trị toán học luôn là 5 và không thể bị thay đổi bởi bất cứ ai. Tương tự, *'Black'* là một hằng, nó biểu thị cho màu đen. Khi đó, 5 được gọi là **hằng số** (numeric constant), 'Black' được gọi là **hằng chuỗi** (string constant).

c) Định danh

Tên của các biến (variables), các hàm (functions), các nhãn (labels) và các đối tượng khác nhau do người dùng định nghĩa gọi là định danh. Những định danh này có thể chứa một hay nhiều ký tự. Ký tự đầu tiên của định danh phải là một chữ cái hay một dấu gạch dưới (_). Các ký tự tiếp theo có thể là các chữ cái, các con số hay dấu gạch dưới.

Arena, s_count, marks40, và class_one là những định danh đúng. Các ví dụ về các định danh sai là 1sttest, oh!god, và start... end.

Các định danh có thể có chiều dài tuỳ ý, nhưng số ký tự trong một biến được nhận diện bởi trình biên dịch thì thay đổi theo trình biên dịch. Ví dụ, nếu một trình biên dịch nhận diện 31 con số có ý nghĩa đầu tiên cho một tên định danh thì các câu sau sẽ hiển thị cùng một kết quả:

Đây là biến testing.... testing

Đây là biến testing.... testing ... testing

Các định danh trong C có phân biệt chữ hoa và chữ thường, cụ thể, arena thì khác ARENA.

✓ Các nguyên tắc cho việc chỉ đặt tên

Các quy tắc đặt tên biến khác nhau tuỳ ngôn ngữ lập trình. Tuy nhiên, vài quy ước chuẩn được tuân theo như :

➢ Tên biến phải bắt đầu bằng một ký tự chữ cái.

- Các ký tự theo sau ký tự đầu bằng một chuỗi các chữ cái hoặc con số và cũng có thể bao gồm ký tự đặc biệt như dấu gạch dưới.

- Tránh dùng ký tự O tại những vị trí mà có thể gây lầm lẫn với số không (0) và tương tự chữ cái l (chữ thường của chữ hoa L) có thể lầm lẫn với số 1.
- Tên riêng nên tránh đặt tên cho biến.
- Theo tiêu chuẩn C các chữ cái thường và hoa thì xem như khác nhau ví dụ. biến ADD, add và Add là khác nhau.
- Việc phân biệt chữ hoa và chữ thường khác nhau tuỳ theo ngôn ngữ lập trình. Do đó, tốt nhất nên đặt tên cho biến theo cách thức chuẩn.
- Tên một biến nên có ý nghĩa, gợi tả và mô tả rõ kiểu dữ liệu của nó. Ví dụ, nếu tìm tổng của hai số thì tên biến lưu trữ tổng nên đặt là sum (tổng). Nếu đặt tên là *s* hay *ab12* thì không hay lắm.

d) Từ khóa (Keywords)

Tất cả các ngôn ngữ dành một số từ nhất định cho mục đích riêng. Những từ này có một ý nghĩa đặc biệt trong ngữ cảnh của từng ngôn ngữ, và được xem là "từ khóa". Khi đặt tên cho các biến, chúng ta cần bảo đảm rằng không dùng bất cứ từ khóa nào làm tên biến.

Tên kiểu dữ liệu tất cả được coi là từ khóa.

Do vậy, đặt tên cho một biến là **int** sẽ phát sinh một lỗi, nhưng đặt tên cho biến là **integer** thì không.

Vài ngôn ngữ lập trình yêu cầu lập trình viên chỉ ra tên của các biến cũng như kiểu dữ liệu của nó trước khi dùng biến đó thật sự. Bước này được gọi là khai báo biến. Ta sẽ nói rõ bước này trong phần tiếp theo khi thảo luận về các kiểu dữ liệu. Điều quan trọng cần nhớ bây giờ là bước này giúp hệ điều hành thật sự cấp phát một khoảng không gian vùng nhớ cho biến trước khi bắt đầu sử dụng nó.

e) Các kiểu dữ liệu

Các loại dữ liệu khác nhau được lưu trữ trong biến là :

- Số (Numbers)

- Các số nguyên.

Ví dụ : 10 hay 178993455.

- Các số thực.

Ví dụ : 15.22 hay 15463452.25.

- Các số dương.
- Các số âm.

- Tên.

Ví dụ : John.

- Giá trị luận lý.

Ví dụ : Y hay N.

Khi dữ liệu được lưu trữ trong các biến có kiểu dữ liệu khác nhau, nó yêu cầu dung lượng bộ nhớ sẽ khác nhau.

Dung lượng bộ nhớ được chỉ định cho một biến tùy thuộc vào kiểu dữ liệu của nó.

Để chỉ định bộ nhớ cho một đơn vị dữ liệu, chúng ta phải khai báo một biến với một kiểu dữ liệu cụ thể.

Khai báo một biến có nghĩa là một vùng nhớ nào đó đã được gán cho biến. Vùng bộ nhớ đó sau này sẽ được tham chiếu thông qua tên của biến. Dung lượng bộ nhớ được cấp cho biến bởi hệ điều hành phụ thuộc vào kiểu dữ liệu được lưu trữ trong biến. Vì vậy, một kiểu dữ liệu sẽ mô tả loại dữ liệu phù hợp với biến.

Dạng thức chung cho việc khai báo một biến:

Kiểu dữ liệu (Tên biến)

Kiểu dữ liệu thường được dùng trong các công cụ lập trình có thể được phân chia thành:

1 Kiểu dữ liệu số - lưu trữ giá trị số.

2 Kiểu dữ liệu ký tự – lưu trữ thông tin mô tả

Những kiểu dữ liệu này có thể có tên khác nhau trong các ngôn ngữ lập trình khác nhau. Ví dụ, một kiểu dữ liệu số được gọi trong C là **int** trong khi đó tại Visual Basic được gọi là **integer**. Tương tự, một kiểu dữ liệu ký tự được đặt tên là **char** trong C trong khi đó trong Visual Basic nó được đặt tên là **string**. Trong bất cứ trường hợp nào, các dữ liệu được lưu trữ luôn giống nhau. Điểm khác duy nhất là các biến được dùng trong một công cụ phải được khai báo theo tên của kiểu dữ liệu được hỗ trợ bởi chính công cụ đó.

C có 5 kiểu dữ liệu cơ bản. Tất cả những kiểu dữ liệu khác dựa vào một trong số những kiểu này. 5 kiểu dữ liệu đó là:

- **int** là một số nguyên, về cơ bản nó biểu thị kích cỡ tự nhiên của các số nguyên (**integers)**.
- **float** và **double** được dùng cho các số có dấu chấm động. Kiểu **float** (số thực) chiếm 4 byte và có thể có tới 6 con số phần sau dấu thập phân, trong khi **double** chiếm 8 bytes và có thể có tới 10 con số phần thập phân.
- **char** chiếm 1 byte và có khả năng lưu một ký tự đơn (**character**).
- **void** được dùng điển hình để khai báo một hàm không trả về giá trị. Điều này sẽ được nói rõ hơn trong phần hàm.

Dung lượng nhớ và phạm vi giá trị của những kiểu này thay đổi theo mỗi loại bộ xử lý và việc cài đặt các trình biên dịch C khác nhau.

Lưu ý: Các con số dấu chấm động được dùng để biểu thị các giá trị cần có độ chính xác ở phần thập phân.

- **Kiểu dữ liệu int**

Là kiểu dữ liệu lưu trữ dữ liệu số và là một trong những kiểu dữ liệu cơ bản trong bất cứ ngôn ngữ lập trình nào. Nó bao gồm một chuỗi của một hay nhiều con số.

Thí dụ trong C, để lưu trữ một giá trị số nguyên trong một biến tên là 'num', ta khai báo như sau:

```
int num;
```

Biến *num* không thể lưu trữ bất cứ kiểu dữ liệu nào như "Alan" hay "abc". Kiểu dữ liệu số này cho phép các số nguyên trong phạm vi -32768 tới 32767 được lưu trữ. Hệ điều hành cấp phát 16 bit (2 byte) cho một biến đã được khai báo kiểu int.

Ví dụ: 12322, 0, -232.

Nếu chúng ta gán giá trị 12322 cho *num* thì biến này là biến kiểu số nguyên và 12322 là hằng số nguyên.

- **Kiểu dữ liệu số thực (float)**

Một biến có kiểu dữ liệu số thực được dùng để lưu trữ các giá trị chứa phần thập phân. Trình biên dịch phân biệt các kiểu dữ liệu **float** và **int.**

Điểm khác nhau chính của chúng là kiểu dữ liệu **int** chỉ bao gồm các số nguyên, trong khi kiểu dữ liệu **float** có thể lưu giữ thêm cả các phân số.

Ví dụ, trong C, để lưu trữ một giá trị **float** trong một biến tên gọi là 'num', việc khai báo sẽ như sau :

```
float num;
```

Biến đã khai báo là kiểu dữ liệu float có thể lưu giá trị thập phân có độ chính xác tới 6 con số. Biến này được cấp phát 32 bit (4 byte) của bộ nhớ. Ví dụ: 23.05, 56.5, 32.

Nếu chúng ta gán giá trị 23.5 cho num, thì biến num là biến số thực và 23.5 là một hằng số thực.

- **Kiểu dữ liệu double**

Kiểu dữ liệu **double** được dùng khi giá trị được lưu trữ vượt quá giới hạn về dung lượng của kiểu dữ liệu **float**. Biến có kiểu dữ liệu là **double** có thể lưu trữ nhiều hơn khoảng hai lần số các chữ số của kiểu **float**.

Số các chữ số chính xác mà kiểu dữ liệu **float** hoặc **double** có thể lưu trữ tùy thuộc vào hệ điều hành cụ thể của máy tính.

Các con số được lưu trữ trong kiểu dữ liệu **float** hay **double** được xem như nhau trong hệ thống tính toán. Tuy nhiên, sử dụng kiểu dữ liệu **float** tiết kiệm bộ nhớ một nửa so với kiểu dữ liệu **double**.

Kiểu dữ liệu **double** cho phép độ chính xác cao hơn (tới 10 con số). Một biến khai báo kiểu dữ liệu **double** chiếm 64 bit (8 byte) trong bộ nhớ.

Thí dụ trong C, để lưu trữ một giá trị **double** cho một biến tên 'num', khai báo sẽ như sau:

```
double num;
```

Nếu chúng ta gán giá trị 23.34232324 cho num, thì biến num là biến kiểu double và 23.34232324 là một hằng kiểu double.

- **Kiểu dữ liệu char**

Kiểu dữ liệu **char** được dùng để lưu trữ một ký tự đơn.

Một kiểu dữ liệu **char** có thể lưu một ký tự đơn được bao đóng trong hai dấu nháy đơn (''). Thí dụ kiểu dữ liệu **char** như: 'a', 'm', '$' '%'.

Ta có thể lưu trữ những chữ số như những ký tự bằng cách bao chúng bên trong cặp dấu nháy đơn. Không nên nhầm lẫn chúng với những giá trị số. Ví dụ, '1', '5' và '9' sẽ không được nhầm lẫn với những số 1, 5 và 9.

Xem xét những câu lệnh của mã C dưới đây:

```
char gender;
gender='M';
```

Hàng đầu tiên khai báo biến gender của kiểu dữ liệu char. Hàng thứ hai lưu giữ một giá trị khởi tạo cho nó là 'M'. Biến gender là một biến ký tự và 'M' là một hằng ký tự. Biến này được cấp phát 8 bit (1 byte) trong bộ nhớ.

- **Kiểu dữ liệu void**

C có một kiểu dữ liệu đặc biệt gọi là **void**. Kiểu dữ liệu này chỉ cho trình biên dịch C biết rằng không có dữ liệu của bất cứ kiểu nào. Trong C, các hàm số thường trả về dữ liệu thuộc một kiểu nào đó. Tuy nhiên, khi một hàm không có gì để trả về, kiểu dữ liệu **void** được sử dụng để chỉ ra điều này.

✓ **Những kiểu dữ liệu cơ bản và dẫn xuất**

Bốn kiểu dữ liệu (char, int, float và double) mà chúng ta đã thảo luận ở trên được sử dụng cho việc trình bày dữ liệu thực sự trong bộ nhớ của máy tính. Những kiểu dữ liệu này có thể được sửa đổi sao cho phù hợp với những tình huống khác nhau một cách chính xác. Kết quả, chúng ta có được các kiểu dữ liệu dẫn xuất từ những kiểu cơ bản này.

Một bổ từ (modifier) được sử dụng để thay đổi kiểu dữ liệu cơ bản nhằm phù hợp với các tình huống đa dạng. Ngoại trừ kiểu **void**, tất cả các kiểu dữ liệu khác có thể cho phép những bổ từ đứng trước chúng. Bổ từ được sử dụng với C là **signed, unsigned, long** và **short**. Tất cả chúng có thể được áp dụng cho dữ liệu kiểu ký tự và kiểu số nguyên. Bổ từ **long** cũng có thể được áp dụng cho **double**.

Một vài bổ từ như :

1. unsigned
2. long
3. short

Để khai báo một biến kiểu dẫn xuất, chúng ta cần đặt trước khai báo biến thông thường một trong những từ khóa của bổ từ. Một giải thích chi tiết về các bổ từ này và cách thức sử dụng chúng được trình bày bên dưới.

- **Các kiểu có dấu (signed) và không dấu(unsigned)**

Khi khai báo một số nguyên, mặc định đó là một số nguyên có dấu. Tính quan trọng nhất của việc dùng **signed** là để bổ sung cho kiểu dữ liệu char, vì char là kiểu không dấu theo mặc định.

Kiểu **unsigned** chỉ rõ rằng một biến chỉ có thể có giá trị dương. Bổ từ này có thể được sử dụng với kiểu dữ liệu **int** và kiểu dữ liệu **float**. Kiểu **unsigned** có thể áp dụng cho kiểu dữ liệu float trong vài trường hợp nhưng điều này giảm bớt tính khả chuyển (portability) của mã lệnh.

Với việc thêm từ **unsigned** vào trước kiểu dữ liệu **int**, miền giá trị cho những số dương có thể được tăng lên gấp đôi.

Ta xem những câu lệnh của mã C cung cấp ở bên dưới, nó khai báo một biến theo kiểu **unsigned int** và khởi tạo biến này có giá trị 23123.

```
unsigned int varNum;
varNum = 23123;
```

Chú ý rằng không gian cấp phát cho kiểu biến này vẫn giữ nguyên. Nghĩa là, biến varNum được cấp phát 2 byte như khi nó dùng kiểu int. Tuy nhiên, những giá trị mà một kiểu **unsgned int** hỗ trợ sẽ nằm trong khoảng từ 0 đến 65535, thay vì là -32768 tới 32767 mà kiểu **int** hỗ trợ. Theo mặc định, int là một kiểu dữ liệu có dấu.

- **Các kiểu long và short**

Chúng được sử dụng khi một số nguyên có chiều dài ngắn hơn hoặc dài hơn chiều dài bình thường. Một bổ từ **short** được áp dụng cho kiểu dữ liệu khi chiều dài yêu cầu ngắn hơn chiều dài số nguyên bình thường và một bổ từ **long** được dùng khi chiều dài yêu cầu dài hơn chiều dài số nguyên bình thường.

Bổ từ **short** được sử dụng với kiểu dữ liệu **int**. Nó sửa đổi kiểu dữ liệu int theo hướng chiếm ít vị trí bộ nhớ hơn. Bởi vậy, trong khi một biến kiểu int chiếm giữ 16 bit (2 byte) thì một biến kiểu **short int** (hoặc chỉ là short), chiếm giữ 8 bit (1 byte) và cho phép những số có trong phạm vi từ -128 tới 127.

Bổ từ **long** được sử dụng tương ứng một miền giá trị rộng hơn. Nó có thể được sử dụng với **int** cũng như với kiểu dữ liệu **double**. Khi được sử dụng với kiểu dữ liệu **int**, biến chấp nhận những giá trị số trong khoảng từ -2,147,483,648 đến

2,147,483,647 và chiếm giữ 32 bit (4 byte). Tương tự, kiểu **long double** của một biến chiếm giữ 128 bit (16 byte).

Một biến **long int** được khai báo như sau:

```
long int varNum;
```

Nó cũng có thể được khai báo đơn giản như long varNum. Một số **long integer** có thể được khai báo như **long int** hay chỉ là **long**. Tương tự, ta có **short int** hay **short.**

Bảng dưới đây trình bày phạm vi giá trị cho các kiểu dữ liệu khác nhau và số bit nó chiếm giữ dựa theo tiêu chuẩn ANSI.

Kiểu	**Dung lượng xấp xỉ (đơn vị là bit)**	**Phạm vi**
char	8	-128 tới 127
unsigned	8	0 tới 255
signed char	8	-128 tới 127
Int	16	-32,768 tới 32,767
unsigned int	16	0 tới 65,535
signed int	16	Giống như kiểu int
short int	16	-128 tới 127
unsigned short int	16	0 tới 65, 535
signed short int	16	Giống như kiểu short int
long int	32	-2,147,483,648 tới 2,147,483,647
signed long int	32	Giống như kiểu long int
unsigned long int	32	0 tới 4,294,967,295
Float	32	6 con số thập phân

double	64	10 con số thập phân
long double	128	10 con số thập phân

Table 2.1: Các kiểu dữ liệu và phạm vi

Thí dụ sau trình bày cách khai báo những kiểu dữ liệu trên.

Ví dụ 2:

```
main()
{
        char abc;/*abc of type character */
        int xyz;  /*xyz of type integer */
        float length;     /*length of type float */
        double area; /* area of type double */
        long liteyrs;     /*liteyrs of type long int */
        short arm;   /*arm of type short integer*/
}
```

Chúng ta xem lại ví dụ cộng hai số và hiển thị tổng ở chương trước. Mã giả như sau :

Ví dụ 3:

```
BEGIN
INPUT A, B
C = A + B
DISPLAY C
END
```

Trong ví dụ này, các giá trị cho hai biến A và B được nhập. Các giá trị được cộng và tổng được lưu cho biến C bằng cách dùng câu lệnh C = A + B. Trong câu lệnh này, A và B là những biến và ký hiệu + gọi là toán tử. Chúng ta sẽ nói về toán tử số học của C ở phần sau đây. Tuy nhiên, có những loại toán tử khác trong C sẽ được bàn tới ở phần kế tiếp.

f) Các toán tử số học

Những toán tử số học được sử dụng để thực hiện những thao tác mang tính số học. Chúng được chia thành hai lớp : Toán tử số học một ngôi (**unary**) và toán tử số học hai ngôi (**binary**).

Bảng 2.2 liệt kê những toán tử số học và chức năng của chúng.

Các toán tử một ngôi	Chức năng	Các toán tử hai ngôi	Chức năng
-	Lấy đối số	+	Cộng
++	Tăng một giá trị	-	Trừ
--	Giảm một giá trị	*	Nhân
		%	Lấy phần dư
		/	Chia
		^	Lấy số mũ

Bảng 2.2: Các toán tử số học và chức năng

- **Các toán tử hai ngôi**

Trong C, các toán tử hai ngôi có chức năng giống như trong các ngôn ngữ khác. Những toán tử như +, -, * và / có thể được áp dụng cho hầu hết kiểu dữ liệu có sẵn trong C. Khi toán tử / được áp dụng cho một số nguyên hoặc ký tự, bất kỳ phần dư nào sẽ được cắt bỏ. Ví dụ, 5/2 sẽ bằng 2 trong phép chia số nguyên. Toán tử **%** sẽ cho ra kết quả là số dư của phép chia số nguyên. Ví dụ: 5%2 sẽ có kết quả là 1. Tuy nhiên, **%** không thể được sử dụng với những kiểu có dấu chấm động.

Chúng ta hãy xem xét một ví dụ của toán tử số mũ.

9^2

Ở đây 9 là cơ số và 2 là số mũ.

Số bên trái của '^' là cơ số và số bên phải '^' là số mũ.

Kết quả của 9^2 là 9*9 = 81.

Thêm ví dụ khác:

5 ^ 3

Có nghĩa là:

5 * 5 * 5

Do đó: 5 ^ 3 = 5 * 5 * 5 = 125.

Ghi chú: Những ngôn ngữ lập trình như Basic, hỗ trợ toán tử mũ. Tuy nhiên, ANSI C không hỗ trợ ký hiệu ^ cho phép tính lũy thừa. Ta có thể dùng cách khác tính lũy thừa trong C là dùng hàm pow() đã được định nghĩa trong **math.h**. Cú pháp của nó thể hiện qua ví dụ sau:

```
...
#include<math.h>
void main(void)
{
….
 /* the following function will calculate x to the power y. */
 z = pow(x, y);
 ….
}
```

Ví dụ sau trình bày tất cả toán tử hai ngôi được dùng trong C. Chú ý rằng ta chưa nói về hàm printf() và getchar(). Chúng ta sẽ bàn trong những phần sau.

Ví dụ 4:

```
#include<stdio.h>
main()
{
    int x,y;
    x = 5;
    y = 2;
printf("The integers are       : %d & %d\n", x, y);
printf("The addition gives     : %d\n", x + y);
printf("The subtraction gives  : %d\n", x - y);
printf("The multiplication gives : %d\n", x * y);
printf("The division gives     : %d\n", x / y);
printf("The modulus gives      : %d\n", x % y);
getchar();
}
```

Kết quả là:

```
The integers are     : 5 & 2
The addition gives   : 7
The subtraction gives :  3
The multiplication gives :    10
The division gives   : 2
The modulus gives    : 1
```

- **Các toán tử một ngôi (unary)**

Các toán tử một ngôi là toán tử trừ một ngôi '-', toán tử tăng '++' và toán tử giảm '--'

Toán tử trừ một ngôi

Ký hiệu giống như phép trừ hai ngôi. Lấy đối số để chỉ ra hay thay đổi dấu đại số của một giá trị.

Ví dụ:

a = -75;

b = -a;

Kết quả của việc gán trên là a được gán giá trị **-75** và b được gán cho giá trị **75** (-(- 75)). Dấu trừ được sử dụng như thế gọi là toán tử một ngôi vì nó chỉ có một toán hạng.

Nói một cách chính xác, không có toán tử một ngôi + trong C. Vì vậy, một lệnh gán như.

```
invld_pls = +50;
```

khi mà *invld_pls* là một biến số nguyên là không hợp lệ trong chuẩn của C. Tuy nhiên, nhiều trình biên dịch không phản đối cách dùng như vậy.

Các toán tử Tăng và Giảm

C bao chứa hai toán tử hữu ích mà ta không tìm thấy được trong những ngôn ngữ máy tính khác. Chúng là **++** và **--**. Toán tử ++ thêm vào toán hạng của nó một đơn vị, trong khi toán tử -- giảm đi toán hạng của nó một đơn vị.

Cụ thể:

x = x + 1;

có thể được viết là:

```
x++;
```

và:

```
x = x - 1;
```

có thể được viết là:

```
x--;
```

Cả hai toán tử này có thể đứng trước hoặc sau toán hạng, chẳng hạn:

```
x = x + 1;
```

có thể được viết lại là

```
x++ hay ++x;
```

Và cũng tương tự cho toán tử --.

Sự khác nhau giữa việc xử lý trước hay sau trong toán tử một ngôi thật sự có ích khi nó được dùng trong một biểu thức. Khi toán tử đứng trước toán hạng, C thực hiện việc tăng hoặc giảm giá trị trước khi sử dụng giá trị của toán hạng. Đây là tiền xử lý (pre-fixing). Nếu toán tử đi sau toán hạng, thì giá trị của toán hạng được sử dụng trước khi tăng hoặc giảm giá trị của nó. Đây là hậu xử lý (post-fixing). Xem xét ví dụ sau;

```
a = 10;
b = 5;
c = a * b++;
```

Trong biểu thức trên, giá trị hiện thời của b được sử dụng cho tính toán và sau đó giá trị của b sẽ tăng sau. Tức là, c được gán 50 và sau đó giá trị của b được tăng lên thành 6.

Tuy nhiên, nếu biểu thức trên là:

```
c = a * ++b;
```

thì giá trị của c sẽ là 60, và b sẽ là 6 bởi vì b được tăng 1 trước khi thực hiện phép nhân với a, sau đó giá trị được gán vào c.

Trong trường hợp mà tác động của việc tăng hay giảm là riêng lẻ thì toán tử có thể đứng trước hoặc sau toán hạng đều được.

Hầu hết trình biên dịch C sinh mã rất nhanh và hiệu quả đối với việc tăng và giảm giá trị. Mã này sẽ tốt hơn so với khi ta dùng toán tử gán. Vì vậy, các toán tử tăng và giảm nên được dùng bất cứ khi nào có thể.

3.3.4. Quy trình kỹ thuật trong bài học của GV và SV:

*** Quy trình thị phạm của GV**

- Bước 1: Chuẩn bị bài giảng, tài liệu và điều kiện trang thiết bị giảng dạy
- Bước 2: Chuẩn bị bài tập mẫu và đưa ra nội dung thực hành
- Bước 3: Phổ biến các quy trình, các bước thực hiện
- Bước 4: Hướng dẫn nội dung bài học, Hướng dẫn thị phạm mẫu
- Bước 5: Quản lý lớp học
- Bước 6: Giải đáp, tư vấn, đánh giá, nhận xét
- Bước 7: Chấm điểm và nộp các bài đánh giá cho bộ môn

*** Quy trình thực hiện bài của SV**

- Bước 1: SV chuẩn bị tài liệu, vở, bút, giấy nháp, tài liệu tra cứu
- Bước 2: Đọc tài liệu Lập trình căn bản C phần khai báo từ khóa, biến, hằng, kiểu dữ liệu.
- Bước 3: Sinh viên trình bày bài thực hành, thảo luận và nhận xét góp ý.

3.3.5. Các mẫu hình sản phẩm cho SV tham khảo:

- SV xem video

3.3.6. Phần tự thực hành thao tác thường xuyên của SV trong bài học:

- Thông thường, khi chương trình ứng dụng cần xử lý dữ liệu, nó cần có nơi nào đó để lưu trữ tạm thời dữ liệu này. Nơi mà dữ liệu được lưu trữ gọi là bộ nhớ.
- Các ngôn ngữ lập trình hiện đại ngày nay cho phép chúng ta sử dụng các tên tượng trưng gọi là biến (variable), dùng để chỉ đến một vùng trong bộ nhớ nơi mà các giá trị cụ thể được lưu trữ.
- Không có giới hạn về số vị trí bộ nhớ mà một chương trình có thể dùng.
- Một hằng (**constant)** là một giá trị không bao giờ bị thay đổi.
- Tên của các biến (variable), các hàm (function), các nhãn (label) và các đối tượng khác nhau do người dùng định nghĩa gọi là định danh.
- Tất cả ngôn ngữ dành một số từ nhất định cho mục đích riêng. Những từ này được gọi là là "từ khóa" (keywords).
- Các kiểu dữ liệu chính của C là character, integer, float, double và void.
- Một bổ từ được sử dụng để thay đổi kiểu dữ liệu cơ bản sao cho phù hợp với nhiều tình huống đa dạng. Các bổ từ được sử dụng trong C là **signed, unsigned, long** và **short**.

- C hỗ trợ hai loại toán tử số học: một ngôi và hai ngôi.
- Toán tử tăng '++' và toán tử giảm '--' là những toán tử một ngôi. Nó chỉ hoạt động trên biến kiểu số.
- Toán tử hai ngôi số học là +, -, *, /, %, nó chỉ tác động lên những hằng số, biến hay biểu thức.
- Toán tử phần dư '%' chỉ áp dụng trên các số nguyên và cho kết quả là phần dư của phép chia số nguyên.

3.3.7. Sản phẩm thực hành:

+ Các ví dụ demo.

3.3.8. Điều kiện để GV- SV thực hiện bài học thực hành;

3.3.8.1. Điều kiện chuẩn bị trước bài học

+ Giảng viên:

- Đề cương chi tiết bài giảng
- Thiết kế bài giảng
- Học liệu tham khảo, mở rộng

+ Sinh viên:

- Đọc kỹ bài học trước khi giảng viên lên lớp
- Chuẩn bị vấn đề thảo luận, trao đổi với giảng viên
- Các tài liệu tham khảo, mở rộng do giảng viên yêu cầu

3.3.8.2. Kỹ thuật và phương tiện dạy học:

- Bảng, phấn viết
- Màn hình lớn
- Máy tính, máy chiếu

3.3.8.3. Phương pháp tổ chức dạy-học:

- Thuyết trình, diễn giảng, thực hành
- Phát vấn
- Trao đổi, thảo luận
- Cemina
- Vấn đáp

3.4 Nội dung bài giảng 3

3.4.1 Tên bài giảng: BÀI 3: NHẬP XUẤT TRONG NGÔN NGỮ LẬP TRÌNH C

Số tiết lên lớp của GV: 05 tiết; Số tiết tự làm bài của SV : 05 tiết

3.4.2. Phần mở đầu tiếp cận bài;

Trong bất kỳ ngôn ngữ lập trình nào, việc nhập giá trị cho các biến và in chúng ra sau khi xử lý có thể được làm theo hai cách:

1. Thông qua phương tiện nhập/xuất chuẩn (I / O).
2. Thông qua những tập tin.

Trong phần này ta sẽ nói về chức năng nhập và xuất cơ bản. Nhập và xuất (I/O) luôn là các thành phần quan trọng của bất kỳ chương trình nào. Để tạo tính hữu ích, chương trình của bạn cần có khả năng nhập dữ liệu vào và hiển thị lại những kết quả của nó.

Trong C, thư viện chuẩn cung cấp những thủ tục cho việc nhập và xuất. Thư viện chuẩn có những hàm quản lý các thao tác nhập/xuất cũng như các thao tác trên ký tự và chuỗi. Trong bài học này, tất cả những hàm nhập dùng để đọc dữ liệu vào từ thiết bị nhập chuẩn và tất cả những hàm xuất dùng để viết kết quả ra thiết bị xuất chuẩn. Thiết bị nhập chuẩn thông thường là bàn phím. Thiết bị xuất chuẩn thông thường là màn hình (console). Nhập và xuất ra có thể được định hướng đến tập tin hay từ tập tin thay vì thiết bị chuẩn. Những tập tin có thể được lưu trên đĩa hay trên bất cứ thiết bị lưu trữ nào khác. Dữ liệu đầu ra cũng có thể được gửi đến máy in.

3.4.3. Phần kiến thức, kỹ thuật căn bản:

3.4.3.1 Phần kiến thức căn bản

- **Phần kiến thức**

Mục tiêu:

Kết thúc bài học này, bạn có thể:

- Hiểu các hàm nhập xuất có định dạng scanf() và printf()
- Sử dụng các hàm nhập xuất ký tự getchar() và putchar()

a. Tệp tin tiêu đề <studio.h>

Trong các ví dụ trước, ta đã từng viết dòng mã sau:

```
#include <stdio.h>
```

Đây là lệnh tiền xử lý (**preprocessor command)**. Trong C chuẩn, ký hiệu # nên đặt tại cột đầu tiên. ***stdio.h*** là một tập tin và được gọi là tập tin tiêu đề (**header).** Nó chứa các macro cho nhiều hàm nhập và xuất được dùng trong C. Hàm *printf(),*

scanf(), putchar() và *getchar()* được thiết kế theo cách gọi các macro trong tập tin ***stdio.h*** để thực thi các công việc tương ứng.

b. Nhập và xuất trong C

Thư viện chuẩn trong C cung cấp hai hàm để thực hiện các yêu cầu nhập và xuất có định dạng. Chúng là:

printf() – Hàm xuất có định dạng.

scanf() – Hàm nhập có định dạng.

Những hàm này gọi là những hàm được định dạng vì chúng có thể đọc và in dữ liệu ra theo các định dạng khác nhau được điều khiển bởi người dùng. Bộ định dạng qui định dạng thức mà theo đó giá trị của biến sẽ được nhập vào và in ra.

i. printf()

Chúng ta đã quen thuộc với hàm này qua các phần trước. Ở đây, chúng ta sẽ xem chúng chi tiết hơn. Hàm *printf()* được dùng để hiển thị dữ liệu trên thiết bị xuất chuẩn – console (màn hình). Dạng mẫu chung của hàm này như sau:

```
printf("control string", argument list);
```

Danh sách tham số (argument list) bao gồm các hằng, biến, biểu thức hay hàm và được phân cách bởi dấu phẩy. Cần phải có một lệnh định dạng nằm trong chuỗi điều khiển (**control string)** cho mỗi tham số trong danh sách. Những lệnh định dạng phải tương ứng với danh sách các tham số về số lượng, kiểu dữ liệu và thứ tự. Chuỗi điều khiển phải luôn được đặt bên trong cặp dấu nháy kép**“”**, đây là dấu phân cách (**delimiters)**. Chuỗi điều khiển chứa một hay nhiều hơn ba thành phần dưới đây :

- **Ký tự văn bản (Text characters)** – Bao gồm các ký tự có thể in ra được và sẽ được in giống như ta nhìn thấy. Các khoảng trắng thường được dùng trong việc phân chia các trường (field) được xuất ra.
- **Lệnh định dạng -** Định nghĩa cách thức các mục dữ liệu trong danh sách tham số sẽ được hiển thị. Một lệnh định dạng bắt đầu với một ký hiệu **%** và theo sau là một mã định dạng tương ứng cho mục dữ liệu. Dấu **%** được dùng trong hàm *printf()* để chỉ ra các đặc tả chuyển đổi. Các lệnh định dạng và các mục dữ liệu tương thích nhau theo thứ tự và kiểu từ trái sang phải. Một mã định dạng thì cần thiết cho mọi mục dữ liệu cần in ra.
- **Các ký tự không in được** – Bao gồm phím tab, dấu khoảng trắng và dấu xuống dòng.

Mỗi lệnh định dạng gồm một hay nhiều mã định dạng. Một mã định dạng bao gồm ký hiệu % và một bộ định kiểu. Bảng 6.1 liệt kê các mã định dạng khác nhau được hỗ trợ bởi câu lệnh *printf()*:

Định dạng	printf()	scanf()
Ký tự đơn (Single Character)	%c	%c
Chuỗi (String)	%s	%s
Số nguyên có dấu (Signed decimal integer)	%d	%d
Số thập phân có dấu chấm động (Floating point)	%f	%f hoặc %e
Số thập phân có dấu chấm động - Biểu diễn phần thập phân	%lf	%lf
Số thập phân có dấu chấm động - Biểu diễn dạng số mũ	%e	%f hoặc %e
Số thập phân có dấu chấm động (%f hoặc %e, con số nào ít hơn)	%g	
Số nguyên không dấu (Unsigned decimal integer)	%u	%u
Số thập lục phân không dấu (Dùng "ABCDEF") (Unsigned hexadecimal integer)	%x	%x
Số bát phân không dấu (Unsigned octal integer)	%o	%o

Bảng 6.1: Mã định dạng trong printf ()

Trong bảng trên, **c, d, f, lf, e, g, u, s, o** và **x** là bộ định kiểu.

Các quy ước in cho các mã định dạng khác nhau được tổng kết trong Bảng 6.2:

Mã định dạng	Quy ước in ấn
%d	Các con số trong số nguyên.
%f	Phần số nguyên của số sẽ được in nguyên dạng. Phần thập phân sẽ chứa 6 con số. Nếu phần thập phân của con số ít hơn 6 số, nó sẽ được thêm các số không (0) bên phải hay gọi là làm tròn phía bên phải.
%e	Một con số bên trái dấu chấm thập phân và 6 con số bên phải giống như %f.

Bảng 6.2: Quy ước in

Bởi vì các ký hiệu *%,*\ và *"* được dùng đặc biệt trong chuỗi điều khiển, nếu chúng ta cần in các ký hiệu này lên màn hình, chúng phải được dùng như trong Bảng 6.3:

\\	In ký tự \
\ "	In ký tự **"**
%%	In ký tự **%**

Bảng 6.3: Các ký tự đặc biệt trong chuỗi điều khiển

Bảng dưới đây đưa ra vài ví dụ sử dụng chuỗi điều khiển và mã định dạng khác nhau.

Số	Câu lệnh	Chuỗi điều khiển	Nội dung mà chuỗi điều khiển chứa đựng	Danh sách tham số	Giải thích danh sách tham số	Hiển thị trên màn hình
1.	printf("%d", 300);	%d	Chỉ chứa lệnh định dạng	300	Hằng số	300
2.	printf("%d", 10+5);	%d	Chỉ chứa lệnh định dạng	10 + 5	Biểu thức	15
3.	printf("Good Morning Mr. Lee.");	Good Morning Mr. Lee.	Chỉ là các ký tự văn bản	Không có (Nil)	Không có	Good Morning Mr. Lee.
4.	int count = 100; printf("%d", count);	%d	Chỉ chứa lệnh định dạng	Count	Biến	100
5.	printf("\nhello");	\nhello	Chỉ là các ký tự văn bản và ký tự không in được.	Không có	Không có	Hello (Trên dòng mới)
6.	#define str "Good Apple" …….. printf("%s", str);	%s	Chỉ chứa lệnh định dạng	Str	Hằng chuỗi	Good Apple
7.	…….. int count,stud_num; count = 0; stud_num = 100; printf("%d %d\n", count, stud_num);	%d %d	Chỉ chứa lệnh định dạng và trình tự thoát ra	count, stud_num	Hai biến	0, 100

Bảng : Chuỗi điều khiển và mã định dạng

Ví dụ 1.1 :

Đây là một chương trình đơn giản dùng minh họa cho một chuỗi có thể được in theo lệnh định dạng. Chương trình này cũng hiển thị một ký tự đơn, số nguyên và số thực (a single character, integer, và float).

```
#include <stdio.h>
void main()
{
      int a = 10;
      float b = 24.67892345;
      char ch = 'A';

      printf("\nInteger data = %d", a);
      printf("\nFloat Data = %f", b);
      printf("\nCharacter = %c", ch);
      printf("\nThis prints the string");
      printf("%s", "\nThis also prints a string");
}
```

Kết quả chương trình như sau:

```
Integer data = 10
Float Data = 24.678923
Character = A
This prints the string
This also prints a string
```

- **Bổ từ (Modifier) cho các lệnh định dạng trong printf()**

Các lệnh định dạng có thể có bổ từ (**modifier**), để thay đổi các đặc tả chuyển đổi gốc. Sau đây là các bổ từ được chấp nhận trong câu lệnh *printf()*. Nếu có nhiều bổ từ được dùng thì chúng tuân theo trình tự sau :

Bổ từ '-'

Dữ liệu sẽ được canh trái bên trong không gian dành cho nó, chúng sẽ được in bắt đầu từ vị trí ngoài cùng bên trái.

Bổ từ xác định độ rộng

Chúng có thể được dùng với kiểu: *float, double hay char array* (chuỗi-string). Bổ từ xác định độ rộng là một số nguyên xác định độ rộng nhỏ nhất của trường dữ liệu. Các dữ liệu có độ rộng nhỏ hơn sẽ cho kết quả canh phải trong trường dữ liệu. Các dữ liệu có kích thước lớn hơn sẽ được in bằng cách dùng thêm những vị trí cho đủ

yêu cầu.Ví dụ, ***%10f*** là lệnh định dạng cho các mục dữ liệu kiểu số thực với độ rộng trường dữ liệu thấp nhất là 10.

Bổ từ xác định độ chính xác

Chúng có thể được dùng với kiểu *float, double* hay *mảng ký tự (char array, string)*. Bổ từ xác định độ rộng chính xác được viết dưới dạng *.m* với *m* là một số nguyên. Nếu sử dụng với kiểu *float* và *double*, chuỗi số chỉ ra số con số *tối đa* có thể được in ra phía bên phải dấu chấm thập phân.

Nếu phần phân số của các mục dữ liệu kiểu *float hay double* vượt quá độ rộng con số chỉ trong bổ từ, thì số đó sẽ được **làm tròn**. Nếu chiều dài chuỗi vượt quá chiều dài chỉ định thì chuỗi sẽ được **cắt bỏ phần dư ra ở phía cuối**. Một vài số không (0) sẽ được thêm vào nếu số con số thực sự trong một mục dữ liệu ít hơn được chỉ định trong bổ từ. Tương tự, các khoảng trắng sẽ được thêm vào cho chuỗi ký tự. Ví dụ, %10.3f là lệnh định dạng cho mục dữ liệu kiểu *float*, với độ rộng tối thiểu cho trường dữ liệu là 10 và 3 vị trí sau phần thập phân.

Bổ từ '0'

Theo mặc định, việc thêm vào một trường được thực hiện với các khoảng trắng. Nếu người dùng muốn thêm vào trường với số không (0), bổ từ này phải được dùng.

Bổ từ 'l'

Bổ từ này có thể được dùng để hiển thị số nguyên như: *long int* hay một tham số kiểu *double*. Mã định dạng tương ứng cho nó là *%ld.*

Bổ từ 'h'

Bổ từ này được dùng để hiện thị kiểu *short* integer. Mã định dạng tương ứng cho nó là *%hd*.

Bổ từ '*'

Bổ từ này được dùng khi người dùng không muốn chỉ trước độ rộng của trường mà muốn chương trình xác định nó. Nhưng khi đi với bổ từ này, một tham số được yêu cầu phải chỉ ra độ rộng trường cụ thể.

Chúng ta hãy xem những bổ từ này hoạt động thế nào. Đầu tiên, chúng ta xem xét tác động của nó đối với những dữ liệu kiểu số nguyên.

Ví dụ 1.2:

```
/* Chương trình này trình bày cách dùng bổ từ trong
printf() */
#include <stdio.h>
void main()
{
     printf("The number 555 in various forms:\n");
```

```
    printf("Without any modifier: \n");
    printf("[%d]\n", 555);
    printf("With - modifier:\n");
    printf("[%-d]\n", 555);
    printf("With digit string 10 as modifier:\n");
    printf("[%10d]\n", 555);
    printf("With 0 as modifier: \n");
    printf("[%0d]\n", 555);
    printf("With 0 and digit string 10 as modifiers:\n");
    printf("[%010d]\n", 555);
    printf("With  -,  0  and  digit  string  10  as
modifiers:\n");
    printf("[%-010d]\n", 555);
}
```

Kết quả như dưới đây:

```
The number 555 in various forms:
Without any modifier:
[555]
With - modifier:
[555]
With digit string 10 as modifier:
[       555]
With 0 as modifier:
[555]
With 0 and digit string 10 as modifiers:
[0000000555]
With -, 0 and digit string 10 as modifiers:
[555       ]
```

Chúng ta đã dùng ký hiệu '**[**' và '**]**' để chỉ ra nơi trường bắt đầu và nơi kết thúc. Khi chúng ta dùng **%d** mà không có bổ từ, chúng ta thấy rằng nó dùng cho một trường có cùng độ rộng với số nguyên. Khi dùng **%10d** chúng ta thấy rằng nó dùng 10 khoảng trắng cho trường và số được canh lề phải theo mặc định. Nếu ta dùng bổ từ –, số sẽ được canh trái trong trường đó. Nếu dùng bổ từ 0, chúng ta thấy rằng số sẽ thêm vào 0 thay vì là khoảng trắng.

Bây giờ chúng ta hãy xem bổ từ dùng với số thực.

Ví dụ 1.3:

```
/* Chương trình này trình bày cách dùng bổ từ trong
printf() */
```

```
#include <stdio.h>
void main()
{
     printf(“The number 555.55 in various forms:\n”);
     printf(“In float form without modifiers:\n”);
     printf(“[%f]\n”, 555.55);
     printf(“In    exponential    form    without    any
modifier:\n”);
     printf(“[%e]\n”, 555.55);
     printf(“In float form with - modifier:\n”);
     printf(“[%-f]\n”, 555.55);
     printf(“In  float  form  with  digit  string  10.3  as
modifier\n”);
     printf(“[%10.3f]\n”, 555.55);
     printf(“In float form with 0 as modifier:\n”);
     printf(“[%0f]\n”, 555.55);
     printf(“In float form with 0 and digit string 10.3”);
     printf(“as modifiers:\n”);
     printf(“[%010.3f]\n”, 555.55);
     printf(“In float form with -, 0 ”);
     printf(“and digit string 10.3 as modifiers:\n”);
     printf(“[%-010.3f]\n”, 555.55);
     printf(“In exponential form with 0”);
     printf(“ and digit string 10.3 as modifiers:\n”);
     printf(“[%010.3e]\n”, 555.55);
     printf(“In exponential form with -, 0”);
     printf(“ and digit string 10.3 as modifiers:\n”);
     printf(“[%-010.3e]\n\n”, 555.55);
}
```

Kết quả như sau:

```
The number 555.55 in various forms:
In float form without modifiers:
[555.550000]
In exponential form without any modifier:
[5.555500e+02]
In float form with - modifier:
[555.550000]
In float form with digit string 10.3 as  modifier
```

```
[   555.550]
In float form with 0 as modifier:
[555.550000]
In float form with 0 and digit string 10.3 as modifiers:
[000555.550]
In float form with -, 0 and digit string 10.3 as
modifiers:
[555.550   ]
In exponential form with 0 and digit string 10.3 as
modifiers:
[05.555e+02]
In exponential form with -,0 and digit string 10.3 as
modifiers:
[5.555e+02]
```

Theo mặc định cho *%f*, chúng ta có thể thấy rằng có 6 con số cho phần thập phân và mặc định cho *%e* là một con số tại phần nguyên và 6 con số phần bên phải dấu chấm thập phân. Chú ý cách thể hiện 2 số cuối cùng trong ví dụ trên, số các con số bên phải dấu chấm thập phân là 3, dẫn đến kết quả không được làm tròn.

Bây giờ, chúng ta hãy xem bổ từ dùng với chuỗi số. Chú ý cách mở rộng trường để chứa toàn bộ chuỗi. Hơn nữa, chú ý cách đặc tả độ chính xác **.4** trong việc giới hạn số ký tự được in.

Ví dụ 1.4:

```
/* Chương trình trình bày cách dùng bổ từ với chuỗi*/
#include <stdio.h>
void main()
{
     printf("A string in various forms:\n");
     printf("Without any format command:\n");
     printf("Good day Mr. Lee. \n");
     printf("With format command but without any
modifier:\n");
     printf("[%s]\n", "Good day Mr. Lee.");
     printf("With digit string 4 as modifier:\n");
     printf("[%4s]\n", "Good day Mr. Lee.");
     printf("With digit string 19 as modifier: \n");
     printf("[%19s]\n", "Good day Mr. Lee.");
     printf("With digit string 23 as modifier: \n");
     printf("[%23s]\n", "Good day Mr. Lee.");
```

```
    printf("With digit string 25.4 as modifier: \n");
    printf("[%25.4s]\n", "Good day Mr.Lee.");
    printf("With - and digit string 25.4 as
modifiers:\n");
    printf("[%-25.4s]\n", "Good day Mr.shroff.");
}
```

Kết quả như sau:

```
A string in various forms:
Without any format command:
Good day Mr. Lee.
With format command but without any modifier:
[Good day Mr. Lee.]
With digit string 4 as modifier:
[Good day Mr. Lee.]
With digit string 19 as modifier:
[  Good day Mr. Lee.]
With digit string 23 as modifier:
[      Good day Mr. Lee.]
With digit string 25.4 as modifier:
[                     Good]
With - and digit string 25.4 as modifiers:
[Good                      ]
```

Những ký tự ta nhập tại bàn phím không được lưu ở dạng các ký tự. Thật sự chúng lưu theo dạng các số dưới dạng mã **ASCII (Bộ mã chuẩn Mỹ cho việc trao đổi thông tin - American Standard Code for Information Interchange)**. Các giá trị của một biến được thông dịch dưới dạng ký tự hay một số tùy vào kiểu của biến đó. Ví dụ sau mô tả điều này:

Ví dụ 1.5:

```
#include <stdio.h>
void main()
{
    int a = 80;
    char b= 'C';
    printf("\nThis is the number stored in 'a' %d",a);
    printf("\nThis is a character interpreted from 'a'
%c",a);
    printf("\nThis is also a character stored in 'b'
%c",b);
```

```
    printf("\nHey! The character of 'b' is printed as a number!  %d", b);
}
```

Kết quả như dưới đây:

```
This is the number stored in `a' 80
This is a character interpreted from `a' P
This is also a character stored in `b' C
Hey! The character of `b' is printed as a number!67
```

Kết quả này mô tả việc dùng các đặc tả định dạng và việc thông dịch của mã ASCII. Mặc dù các biến *a* và *b* đã được khai báo là các biến kiểu *int* và *char*, nhưng chúng đã được in như là ký tự và số nhờ vào việc dùng các bộ định dạng khác nhau. Đặc điểm này của C giúp việc xử lý dữ liệu được linh hoạt.

Khi dùng câu lệnh *printf()* để cho ra một chuỗi dài hơn 80 ký tự trên một dòng, khi xuống dòng ta phải ngắt mỗi dòng bởi ký hiệu \ như được trình bày trong ví dụ dưới đây:

Ví dụ 1.6:

```
/* Chương trình trình bày cách dùng một chuỗi dài các ký tự*/
#include <stdio.h>
void main()
{
printf("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
\aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
a\aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aa\aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaa\aaaaaaaaaaaaaaaaaaaaaaaaaaaaa");
}
```

Kết quả như sau:

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aa
aaaaaaaaaaaaa

Trong ví dụ trên, chuỗi trong câu lệnh *printf()* có 252 ký tự. Trong khi một dòng văn bản chứa 80 ký tự, do đó chuỗi được mở rộng thành 3 hàng trong kết quả như trên.

ii. scanf()

Hàm scanf() được sử dụng để nhập dữ liệu. Khuôn dạng chung của hàm scanf() như sau:

scanf(<Chuỗi các định dạng>, <Danh sách các tham số>);

Định dạng được sử dụng bên trong câu lệnh printf() cũng được sử dụng cùng cú pháp trong các câu lệnh scanf().

Những lệnh định dạng, bao gồm bổ từ và danh sách tham số được bàn luận cho printf() thì cũng hợp lệ cho scanf(), chúng tuân theo một số điểm khác biệt sau:

➢ **Sự khác nhau trong danh sách tham số giữa printf() và scanf()**

Hàm *printf()* dùng các tên biến, hằng số, hằng chuỗi và các biểu thức, nhưng scanf() sử dụng những con trỏ tới các biến. Một con trỏ tới một biến là một mục dữ liệu chứa đựng địa chỉ của nơi mà biến được cất giữ trong bộ nhớ. Những con trỏ sẽ đ-ược bàn luận chi tiết ở chương sau. Khi sử dụng scanf() cần tuân theo những quy tắc cho danh sách tham số:

• Nếu ta muốn nhập giá trị cho một biến có kiểu dữ liệu cơ bản, gõ vào tên biến cùng với ký hiệu **&** trước nó.

• Khi nhập giá trị cho một biến thuộc kiểu dữ liệu dẫn xuất (không phải thuộc bốn kiểu cơ bản char, int, float, double), không sử dụng **&** trước tên biến.

➢ **Sự khác nhau trong lệnh định dạng giữa printf() và scanf()**

a) Không có tùy chọn %g.

b) Mã định dạng %f và %e có cùng hiệu quả tác động. Cả hai nhận một ký hiệu tùy chọn, một chuỗi các con số có hay không có dấu chấm thập phân và một trường số mũ tùy chọn.

Cách thức hoạt động của scanf()

scanf() sử dụng những ký tự không được in như ký tự khoảng trắng, ký tự phân cách (tab), ký tự xuống dòng để quyết định khi nào một trường nhập kết thúc và bắt đầu. Có sự tương ứng giữa lệnh định dạng với những trường trong danh sách tham số theo một thứ tự xác định, bỏ qua những ký tự khoảng trắng bên trong. Do đó, đầu vào có thể được trải ra hơn một dòng, miễn là chúng ta có ít nhất một ký tự phân cách, khoảng trắng hay hàng mới giữa các trường nhập vào. Nó bỏ qua những khoảng trắng và ranh giới hàng để thu được dữ liệu.

Ví dụ 1.7

Chương trình sau mô tả việc dùng hàm *scanf()*

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

```
    float d;
    char ch, name[40];
    printf("Please enter the data\n");
    scanf("%d %f %c %s", &a, &d, &ch, name);
    printf("\nThe values accepted are: %d, %f, %c, %s", a,
d, ch, name);
}
```

Kết quả như sau:

```
Please enter the data
12 67.9 F MARK
The values accepted are:12, 67.900002, F, MARK
```

Dữ liệu đầu vào có thể là:

12 67.9

F MARK

hoặc như:

12

67.9

F

MARK

cũng được nhận vào các biến *a*, *d*, *ch*, và *name*.

Xem ví dụ khác:

Ví dụ 1.8

```
    #include <stdio.h>
    void main()
    {
          int i;
          float x;
          char c;
          .........
          scanf("%3d %5f %c", &i, &x, &c);
    }
```

Nếu dữ liệu nhập vào là:

21 10.345 F

Khi chương trình được thực thi, thì 21 sẽ gán tới i, 10.34 sẽ gán tới x và ký tự 5 sẽ được gán cho c. Còn lại là đặc tính F sẽ bị bỏ qua.

Khi ta chỉ rõ một chiều rộng trường bên trong scanf(), thí dụ %10s, rồi sau đó scanf() chỉ thu nhận tối đa 10 ký tự hoặc tới ký tự khoảng trắng đầu tiên (bất cứ ký tự nào đầu tiên). Điều này cũng áp dụng cho các kiểu *int*, *float và double*.

Ví dụ dưới đây mô tả việc sử dụng hàm scanf() để nhập vào một chuỗi gồm có những ký tự viết hoa và khoảng trắng. Chuỗi sẽ có chiều dài không xác định nhưng nó bị giới hạn trong 79 ký tự (thật ra, 80 ký tự bao gồm ký tự trống (null) được thêm vào nơi cuối chuỗi).

Ví dụ 1.9:

```
#include <stdio.h>
void main()
{
 char line[80]; /* line[80] là một mảng lưu 80 ký tự
*/
        ..........
        scanf("%[ ABCDEFGHIJKLMNOPQRSTUVWXYZ]", line);
        ..........
}
```

Mã khuôn dạng %[] có nghĩa những ký tự được định nghĩa bên trong [] có thể được chấp nhận như những ký tự chuỗi hợp lệ. Nếu chuỗi **BEIJING CITY** được nhập vào từ thiết bị nhập chuẩn, khi chương trình được thực thi, toàn bộ chuỗi sẽ được gán cho mảng một khi chuỗi chỉ toàn là ký tự viết hoa và khoảng trắng. Nếu chuỗi được viết là **Beijing city**, chỉ ký tự đơn B được gán cho mảng, khi đó thì ký tự viết thường đầu tiên (trong trường hợp này là 'e') được thông dịch như ký tự đầu tiên bên ngoài chuỗi.

Để chấp nhận bất kỳ ký tự nào đến khi gặp ký tự xuống dòng, chúng ta sử dụng mã định dạng %[^\n], điều này ngụ ý rằng chuỗi đó sẽ chấp nhận bất kỳ ký tự nào trừ "\n" (ký tự xuống dòng). Dấu mũ (^) ngụ ý rằng tất cả các ký tự trừ những ký tự nằm sau dấu mũ đó sẽ được chấp nhận như ký tự hợp lệ.

Ví dụ 1.10:

```
#include <stdio.h>
void main()
{
        char line[80];
        ……..
        scanf("%[^\n]", line);
        ……..
}
```

Khi hàm *scanf()* được thực thi, một chuỗi có chiều dài không xác định (nhưng không quá 79 ký tự) sẽ được nhập vào từ thiết bị nhập chuẩn và được gán cho mảng.

Sẽ không có giới hạn nào trên các ký tự của chuỗi, ngoại trừ tất cả chúng chỉ nằm trên một hàng. Ví dụ chuỗi sau:

```
All’s well that ends well!
```

Có thể được nhập vào từ bàn phím và được gán cho mảng.

Bổ từ * cho kết quả khác nhau trong *scanf()*. Dấu * được dùng để chỉ rằng một trường sẽ được bỏ qua luôn hay tạm bỏ qua.

Ví dụ xét chương trình:

```
#include <stdio.h>
void main()
{
      char item[20];
      int partno;
      float cost;

      .........
      scanf(“%s %*d %f”, item, &partno, &cost);
      .........
}
```

Nếu các mục dữ liệu tương ứng là:

```
battery 12345  0.05
```

thì **battery** sẽ được gán cho *item* và **0.05** sẽ được gán cho *cost* nhưng **12345** sẽ không được gán cho *partno* bởi vì dấu * ngăn chặn việc gán.

Bất cứ ký tự khác trong *scanf()* mà không là mã định dạng trong chuỗi điều khiển phải được nhập vào chính xác nếu không sẽ phát sinh lỗi. Đặc điểm này được dùng để chấp nhận dấu phân cách phẩy (,).

Ví dụ chuỗi dữ liệu

```
10, 15, 17
```

và lệnh nhập vào

```
scanf("%d, %f, %c", &intgr, &flt, &ch);
```

Chú ý rằng dấu phẩy trong chuỗi chuyển đổi tương ứng dấu phẩy trong chuỗi nhập và vì vậy nó sẽ có chức năng như dấu phân cách.

Ký tự khoảng trắng trong chuỗi điều khiển thường được bỏ qua mặc dù nó sẽ phát sinh trở ngại khi dùng với mã định dạng `%c`. Nếu chúng ta dùng bộ định dạng `%c` thì một khoảng trắng được xem như là một ký tự hợp lệ.

Xét đoạn mã sau:

```
int x, y;

char ch;

scanf("%2d %c %d",&x, &ch, &y);

printf("%d %d %d\n",x, ch, y);
```

ta nhập vào:

```
14 c  5
```

`14` sẽ được gán cho `x`, ký tự ch nhận ký tự khoảng trắng (số 32 trong hệ thập phân), do vậy `y` được gán giá trị của ký tự 'c' tức là số 99 trong hệ thập phân.

Xét đoạn mã sau:

```
#include <stdio.h>

void main()
{
     char c1, c2, c3;
```

```
        ……..
        scanf("%c%c%c",&c1, &c2, &c3);

        ……..
    }
```

Nếu dữ liệu nhập vào là:

```
a b c
```

(với khoảng trắng giữa các ký tự), thì kết quả của phép gán:

```
c1 = a,          c2 = <Khoảng trắng>,          c3 =
b
```

Ở đây chúng ta có thể thấy c2 chứa một khoảng trắng vì chuỗi nhập có chứa ký tự khoảng trắng. Để bỏ qua các ký tự khoảng trắng này và đọc ký tự tiếp theo không phải là ký tự khoảng trắng, ta nên dùng tập chuyển đổi `%1s`.

```
scanf("%c%1s%1s",&c1, &c2, &c3);
```

Khi đó kết quả sẽ khác đi với cùng dữ liệu nhập vào như trước và kết quả đúng như ý định của ta:

```
c1 = a,          c2 = b,          c3 = c
```

c. Bộ nhớ đệm Nhập và Xuất

Ngôn ngữ C bản thân nó không định nghĩa các thao tác nhập và xuất. Tất cả thao tác nhập và xuất được thực hiện bởi các hàm có sẵn trong thư viện hàm của C. Thư viện hàm C chứa một hệ thống hàm riêng mà nó điều khiển các thao tác này. Đó là:

- Bộ nhớ đệm Nhập và Xuất – được dùng để đọc và viết các ký tự ASCII

Một vùng đệm là nơi lưu trữ tạm thời, nằm trên bộ nhớ máy tính hoặc trên thẻ nhớ của bộ điều khiển thiết bị (controller card). Các ký tự nhập vào từ bàn phím được đưa vào bộ nhớ và đợi đến khi người dùng nhấn phím *return* hay *enter* thì chúng sẽ được thu nhận như một khối và cung cấp cho chương trình.

Bộ nhớ đệm nhập và xuất có thể được phân thành:

- Thiết bị nhập/xuất chuẩn (Console I/O)

- Tập tin đệm nhập/xuất (Buffered File I/O)

Thiết bị nhập/xuất chuẩn liên quan đến những hoạt động của bàn phím và màn hình của máy tính. **Tập tin đệm nhập/xuất** liên quan đến những hoạt động thực hiện đọc và viết dữ liệu vào tập tin. Chúng ta sẽ nói về **Thiết bị nhập/xuất**.

Trong C, **Thiết bị nhập/xuất chuẩn** là một thiết bị luồng. Các hàm trong Thiết bị nhập/xuất chuẩn hướng các thao tác đến thiết bị nhập và xuất chuẩn của hệ thống.

Các hàm đơn giản nhất của Thiết bị nhập/xuất chuẩn là:

- **getchar()** – Đọc một và chỉ một ký tự từ bàn phím.
- **putchar()** – Xuất một ký tự đơn ra màn hình.

i) getchar()

Hàm *getchar()* được dùng để đọc dữ liệu nhập vào, chỉ một ký tự tại một thời điểm từ bàn phím.Trong hầu hết việc thực thi của C, khi dùng *getchar(),* các ký tự nằm trong vùng đệm cho đến khi người dùng nhấn phím xuống dòng. Vì vậy nó sẽ đợi cho đến khi phím Enter được gõ. Hàm *getchar()* không có tham số, nhưng vẫn phải có cặp dấu ngoặc đơn. Nó đơn giản lấy về ký tự tiếp theo và sẵn sàng đưa ra cho chương trình. Chúng ta nói rằng hàm này trả về một giá trị có kiểu ký tự.

Chương trình sau trình bày cách dùng hàm *getchar()*.

Ví dụ 1.11:

```
/* Chương trình trình bày cách dùng getchar() */
#include <stdio.h>
void main()
{
    char letter;
    printf("\nPlease enter any character: ");
    letter = getchar();
    printf("\nThe character entered by you is %c. ",
letter);
}
```

Kết quả như sau:

```
Please enter any character: S
The character entered by you is S.
```

Trong chương trình trên ‘letter’ là một biến được khai báo là kiểu *char* do vậy nó sẽ nhận vào ký tự.

Một thông báo:

```
Please enter any character:
```

sẽ xuất hiện trên màn hình. Ta nhập vào một ký tự, trong ví dụ là S, qua bàn phím và nhấn Enter. Hàm *getchar()* nhận ký tự đó và gán cho biến có tên là *letter*. Sau đó nó được hiển thị trên màn hình và ta có được thông báo.

```
The character entered by you is S.
```

ii) putchar()

putchar() là hàm xuất ký tự trong C, nó sẽ xuất một ký tự lên màn hình tại vị trí con trỏ màn hình. Hàm này yêu cầu một tham số. Tham số của hàm *putchar()* có thể thuộc các loại sau:

- Hằng ký tự đơn
- Định dạng (Escape sequence)
- Một biến ký tự.

Nếu tham số là một hằng nó phải được bao đóng trong dấu nháy đơn. Bảng 6.5 trình bày vài tùy chọn cho *putchar()* và tác động của chúng.

Tham số	**Hàm**	**Tác dụng**
Biến ký tự	putchar(c)	Hiện thị nội dung của biến ký tự c
Hằng biến ký tự	putchar('A')	Hiển thị ký tự A
Hằng số	putchar('5')	Hiển thị con số 5
Định dạng (escape sequence)	putchar('\t')	Chèn một ký tự khoảng cách (tab) tại vị trí con trỏ màn hình
Định dạng (escape sequence)	putchar('\n')	Chèn một mã xuống dòng tại vị trí con trỏ màn hình

Bảng 6.5: Những tùy chọn cho putchar() và tác dụng của chúng

Chương trình sau trình bày về hàm putchar():**Ví dụ 1.12:**

```
/* Chương trình này trình bày việc sử dụng hằng và định
dạng trong hàm putchar() */
#include <stdio.h>
void main()
{
    putchar('H'); putchar('\n');
    putchar('\t');
    putchar('E'); putchar('\n');
    putchar('\t'); putchar('\t');
    putchar('L'); putchar('\n');
    putchar('\t'); putchar('\t'); putchar('\t');
    putchar('L'); putchar('\n');
    putchar('\t'); putchar('\t'); putchar('\t');
    putchar('\t');
    putchar('O');
}
```

Kết quả như sau:

```
H
     E
             L
                     L
                             O
```

Khác nhau giữa *getchar() và putchar()* là *putchar()* yêu cầu một tham số trong khi *getchar()* thì không.

Ví dụ 1.13:

```
/* Chương trình trình bày getchar() và putchar() */
#include <stdio.h>
void main()
{
    char letter;
    printf("You can enter a character now: ");
    letter = getchar();
    putchar(letter);
}
```

Kết quả như sau:

```
You can enter a character now: F
F
```

3.4.4. Quy trình kỹ thuật trong bài học của GV và SV:

*** Quy trình thị phạm của GV**

- Bước 1: Chuẩn bị bài giảng, tài liệu và điều kiện trang thiết bị giảng dạy
- Bước 2: Chuẩn bị bài tập mẫu và đưa ra nội dung thực hành
- Bước 3: Phổ biến các quy trình, các bước thực hiện
- Bước 4: Hướng dẫn nội dung bài học, Hướng dẫn thị phạm mẫu
- Bước 5: Quản lý lớp học
- Bước 6: Giải đáp, tư vấn, đánh giá, nhận xét
- Bước 7: Chấm điểm và nộp các bài đánh giá cho bộ môn

*** Quy trình thực hiện bài của SV**

- Bước 1: SV chuẩn bị tài liệu, vở, bút, giấy nháp, tài liệu tra cứu
- Bước 2: Đọc tài liệu Lập trình căn bản C nhập, xuất dữ liệu.
- Bước 3: Sinh viên trình bày bài thực hành, thảo luận và nhận xét góp ý.

3.4.5. Các mẫu hình sản phẩm cho SV tham khảo:

- SV xem video

3.4.6. Phần tự thực hành thao tác thường xuyên của SV trong bài học:

Câu 1. A. Hãy dùng câu lệnh *printf()* để :

a) Xuất ra giá trị của biến số nguyên *sum.*

b) Xuất ra chuỗi văn bản "Welcome", tiếp theo là một dòng mới.

c) Xuất ra biến ký tự *letter*.

d) Xuất ra biến số thực *discount*.

e) Xuất ra biến số thực *dump* có 2 vị trí phần thập phân.

Câu 1. B. Dùng câu lệnh *scanf()* và thực hiện:

a) Đọc giá trị thập phân từ bàn phím vào biến số nguyên *sum.*

b) Đọc một giá trị số thực vào biến *discount_rate*.

c) Xét chương trình sau:

```
#include <stdio.h>
void main()
{
 int breadth;
 float length, height;
 scanf("%d%f%6.2f", breadth, &length, height);
```

```
 printf("%d %f %e", &breadth, length, height);
}
```

Sửa lỗi chương trình trên.

d) Viết một chương trình nhập vào *name*, *basic*, *daper* (phần trăm của D.A), *bonper* (phần trăm lợi tức) và *loandet* (tiền vay bị khấu trừ) cho một nhân viên. Tính lương như sau:

```
salary = basic + basic * daper/100 + bonper * basic/100 - loandet
```

Bảng dữ liệu:

name	basic	Daper	bonper	loandet
MARK	2500	55	33.33	250.00

Tính salary và xuất ra kết quả dưới các đầu đề sau (Lương được in ra gần dấu đôla ($)):

Name Basic Salary

Viết một chương trình yêu cầu nhập vào tên, họ của bạn và sau đó xuất ra tên, họ theo dạng là họ, tên

3.4.7. Sản phẩm thực hành:

+ Các ví dụ demo.

3.4.8. Điều kiện để GV- SV thực hiện bài học thực hành;

3.4.8.1. Điều kiện chuẩn bị trước bài học

+ Giảng viên:

- Đề cương chi tiết bài giảng
- Thiết kế bài giảng
- Học liệu tham khảo, mở rộng

+ Sinh viên:

- Đọc kỹ bài học trước khi giảng viên lên lớp
- Chuẩn bị vấn đề thảo luận, trao đổi với giảng viên
- Các tài liệu tham khảo, mở rộng do giảng viên yêu cầu

3.4.8.2. Kỹ thuật và phương tiện dạy học:

- Bảng, phấn viết

- Màn hình lớn
- Máy tính, máy chiếu

3.4.8.3. Phương pháp tổ chức dạy-học:

- Thuyết trình, diễn giảng, thực hành
- Phát vấn
- Trao đổi, thảo luận
- Cemina
- Vấn đáp

4. Tín chỉ 2

4.1 Danh mục tên bài

TT	Tên bài giảng	Tổng số tiết trên lớp của GV	Số tiết GV trình bầy	Số tiết GV hướng dẫn thảo luận	Bài tự luận/ nghiên cứu ngoài xã hội	Giảng viên thực hiện
1	**Bài 1: Cấu trúc rẽ nhánh có điều kiện.** Lệnh và khối lệnh Lệnh if Lệnh Switch	8	4	4	0	8
2	**Bài 2: Cấu trúc vòng lặp**. Lệnh for Lệnh break Lệnh continue Lệnh while Lệnh do…while Vòng lặp lồng nhau So sánh sự khác nhau của các vòng lặp	8	4	4	0	8
3	**Bài 3: Hàm** Tham số dạng tham biến và tham trị Sử dụng biến toàn cục Dùng dẫn hướng #define	8	4	4	0	8
4	**Bài 4: Mảng và chuỗi** Mảng Chuỗi	6	3	3	0	6
	Tổng:	**30**	**15**	**15**	**0**	30

4.2 Nội dung bài giảng 1

4.2.1 Tên bài giảng: BÀI 1: CẤU TRÚC RẼ NHÁNH CÓ ĐIỀU KIỆN

Số tiết lên lớp của GV: 04 tiết; Số tiết tự làm bài của SV : 04 tiếT

4.2.2. Phần mở đầu tiếp cận bài;

Các vấn đề được đề cập từ đầu đến nay cho phép chúng ta viết nhiều chương trình. Tuy nhiên các chương trình đó có nhược điểm là bất cứ khi nào được chạy, chúng luôn thực hiện một chuỗi các thao tác giống nhau, theo cách thức giống nhau. Trong khi đó, chúng ta thường xuyên chỉ cho phép thực hiện các thao tác nhất định nếu nó thỏa mãn điều kiện đặt ra

4.2.3. Phần kiến thức, kỹ thuật căn bản:

4.2.3.1. Phần kiến thức căn bản:

- **Mục tiêu:**

Kết thúc bài học này, bạn có thể:

- Giải thích về Cấu trúc lựa chọn
  - Câu lệnh **if**
  - Câu lệnh **if – else**
  - Câu lệnh với nhiều lệnh **if**
  - Câu lệnh **if** lồng nhau
  - Câu lệnh **switch**.

a. Câu lệnh điều kiện là gì ?

Các câu lệnh điều kiện cho phép chúng ta thay đổI luồng chương trình. Dựa trên một điều kiện nào đó, một câu lệnh hay một chuỗI các câu lệnh có thể được thực hiện hoặc không.

Hầu hết các ngôn ngữ lập trình đều sử dụng lệnh if để đưa ra điều kiện. Nguyên tắc thực hiện như sau nếu điều kiện đưa ra là **đúng (true)**, chương trình sẽ thực hiện một công việc nào đó, nếu điều kiện đưa ra là **sai (false)**, chương trình sẽ thực hiện một công việc khác.

Ví dụ 1:

Để xác định một số là số chẵn hay số lẻ, ta thực hiện như sau:

1. Nhập vào một số.
2. Chia số đó cho 2 để xác định số dư.
3. Nếu số dư của phép chia là 0, đó là số “Chẵn”.

HOẶC

Nếu số dư của phép chia khác 0, đó là số “Lẻ”.

Bước 2 trong giảI thuật trên kiểm tra phần dư của số đó khi chia cho 2 có bằng 0 không? Nếu đúng, ta thực hiện việc hiển thị thông báo đó là số chẵn. Nếu số dư đó khác 0, ta thực hiện việc hiển thị thông báo đó là số lẻ.

Trong C một điều kiện được coi là đúng (true) khi nó có giá trị khác 0, là sai (false) khi nó có giá trị bằng 0.

b. Các câu lệnh lựa chọn:

C cung cấp hai dạng câu lệnh lựa chọn:

- Câu lệnh **if**
- Câu lệnh **switch**

Chúng ta hãy tìm hiểu hai câu lệnh lựa chọn này.

Câu lệnh '*if*':

Câu lệnh **if** cho phép ta đưa ra các quyết định dựa trên việc kiểm tra một điều kiện nào đó là **đúng (true)** hay **sai (false)**.

Các điều kiện gồm các toán tử so sánh và logic mà chúng ta đã thảo luận ở bài 4.

Dạng tổng quát của câu lệnh **if**:

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

Biểu thức phải luôn được đặt trong cặp dấu ngoặc (). Mệnh đề theo sau từ khoá **if** là một điều kiện (hoặc một biểu thức điều kiện) cần được kiểm tra. Tiếp đến là một lệnh hay một tập các lệnh sẽ được thực thi khi điều kiện (hoặc biểu thức điều kiện) có kết quả **true**.

Ví dụ 2:

```
#include <stdio.h>
void main()
{
     int x, y;
     char a = 'y';
     x = y = 0;
     if (a == 'y')
     {
          x += 5;
          printf("The numbers are %d and \t%d", x, y);
     }
}
```

Kết quả của chương trình như sau:

```
The numbers are 5 and 0
```

Có kết quả này là do biến *a* đã được gán giá trị '*y*'.

Chú ý rằng, khối lệnh sau lệnh **if** được đặt trong cặp ngoặc nhọn {}. Khi có nhiều lệnh cần được thực hiện, các câu lệnh đó được coi như một block (khốI lệnh) và phảI được đặt trong cặp dấu {}. Nếu trong ví dụ trên ta không đưa vào dấu ngoặc nhọn ở câu lệnh if, chỉ có câu lệnh đầu tiên (x += 5) được thực hiện khi điều kiện trong câu lệnh if là đúng.

Ví dụ dưới đây sẽ kiểm tra một năm có phải là năm nhuận hay không. Năm nhuận là năm chia hết cho 4 hoặc 400 nhưng không chia hết cho 100. Chúng ta sử dụng lệnh **if** để kiểm tra điều kiện.

Ví dụ 3:

```
/* To test for a leap year */

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

      printf("\nPlease enter a year:");
      scanf("%d", &year);

      if(year % 4 == 0 && year % 100 != 0 || year %
400 == 0)
            printf("\n%d is a leap year!", year);
}
```

Chương trình trên cho ra kết quả như sau:

```
Please enter a year: 1988
1988 is a leap year!
```

Điều kiện *year % 4 == 0 && year % 100 != 0 || year % 400 == 0* trả về giá trị 1 nếu năm đó là năm nhuận. Khi đó, chương trình hiển thị thông báo gồm biến year và dòng chữ **"is a leap year"**. Nếu điều kiện trên không thỏa mãn, chương trình không hiển thị thông báo nào.

Câu lệnh '*if … else*':

Ở trên chúng ta đã biết dạng đơn giản nhất của câu lệnh **if**, cho phép ta lựa chọn để thực hiện hay không một câu lệnh hoặc một chuỗI các lệnh. C cũng cho phép ta lựa chọn trong hai khốI lệnh để thực hiện bằng cách dùng cấu trúc **if – else**. Cú pháp như sau:

```
if (biểu thức)
        câu_lệnh – 1;
else
        câu_lệnh – 2;
```

Nếu biểu thức điều kiện trên là **đúng (khác 0)**, câu lệnh 1 được thực hiện. Nếu nó **sai (khác 0)** câu lệnh 2 được thực hiện. Câu lệnh sau **if** và **else** có thể là lệnh đơn hoặc lệnh phức. Các câu lệnh đó nên được lùi vào trong dòng mặc dù không bắt buộc. Cách viết đó giúp ta nhìn thấy ngay những lệnh nào sẽ được thực hiện tùy theo kết quả của biểu thức điều kiện.

Bây giờ chúng ta viết một chương trình kiểm tra một số là số chẵn hay số lẻ. Nếu đem chia số đó cho 2 được dư là 0 chương trình sẽ hiển thị dòng chữ "The number is Even", ngược lại sẽ hiển thị dòng chữ "The number is Odd".

Ví dụ 4:

```
#include <stdio.h>
void main()
{
      int num, res;

      printf("Enter a number: ");
      scanf("%d", &num);

      res = num % 2;

      if (res == 0)
           printf("The number is Even");
      else
           printf("The number is Odd");
}
```

Xem một ví dụ khác, đổi một ký tự hoa thành ký tự thường. Nếu ký tự không phải là một ký tự hoa, nó sẽ được in ra mà không cần thay đổi. Chương trình sử dụng cấu trúc **if-else** để kiểm tra xem một ký tự có phải là ký tự hoa không, rồI thực hiện các thao tác tương ứng.

Ví dụ 5:

```
/* Doi mot ky tu hoa thanh ky tu thuong */
#include <stdio.h>
```

```
void main()
{
      char c;
      printf("Please enter a character: ");
      scanf("%c", &c);

      if (c >= 'A' && c <= 'Z')
           printf("Lowercase character = %c", c + 'a' -
'A');
      else
           printf("Character Entered is = %c", c);
 }
```

Biểu thức *c >= 'A' && c <= 'Z'* kiểm tra ký tự nhập vào có là ký tự hoa không. Nếu biểu thức trả về **true**, ký tự đó sẽ được đổi thành ký tự thường bằng cách sử dụng biểu thức *c + 'a' – 'A'*, và được in ra màn hình qua hàm *printf()*. Nếu giá trị của biểu thức là **false**, câu lệnh sau **else** được chạy và chương trình hiển thị kí tự đó ra màn hình mà không cần thực hiện bất cứ sự thay đổI nào.

d. Nhiều lựa chọn – Các câu lệnh '*if … else*':

Câu lệnh **if** cho phép ta lựa chọn thực hiện một hành động nào đó hay không. Câu lệnh **if – else** cho phép ta lựa chọn thực hiện giữa hai hành động. C cho phép ta có thể đưa ra nhiều lựa chọn hơn. Chúng ta mở rộng cấu trúc **if – else** bằng cách thêm vào cấu trúc **else – if** để thực hiện điều đó. Nghĩa là mệnh đề **else** trong một câu lệnh **if – else** lạI chứa một câu lệnh **if – else** khác. Do đó nhiều điều kiện hơn được kiểm tra và tạo ra nhiều lựa chọn hơn.

Cú pháp tổng quát trong trường hợp này như sau:

```
if (biểu thức) câu_lệnh;
else
      if (biểu thức) câu_lệnh;
      ……
      else câu_lệnh;
```

Cấu trúc này gọi là **if–else–if ladder** hay **if-else-if staircase.**

Cách canh lề (lùi vào trong) như trên giúp ta nhìn chương trình một cách dễ dàng khi có một hoặc hai lệnh **if**. Tuy nhiên khi có nhiều lệnh **if** hơn cách viết đó dễ gây ra nhầm lẫn vì nhiều câu lệnh sẽ phải lùi vào quá sâu. Vì vậy, lệnh **if-else-if** thường được canh lề theo dạng:

```
      if (biểu thức)
           câu_lệnh;
      else if (biểu thức)
```

```
        câu_lệnh;
    else if (biểu thức)
        câu_lệnh;
    …….
    else
        câu_lệnh;
```

Các điều kiện được kiểm tra từ trên xuống dưới. Khi có một điều kiện nào đó là **true**, các câu lệnh gắn với nó sẽ được thực hiện và các lệnh còn lại sẽ được bỏ qua. Nếu không có điều kiện nào là **true**, các câu lệnh gắn với **else** cuối cùng sẽ được thực hiện. Nếu mệnh đề **else** đó không tồn tại, sẽ không có lệnh nào được thực hiện do tất cả các điều kiện đều **false**.

Ví dụ dưới đây nhận một số từ người dùng. Nếu số đó có giá trị từ 1 đến 3, chương trình sẽ in ra số đó, ngược lại chương trình in ra thông báo "Invalid choice".

Ví dụ 6:

```
#include <stdio.h>
    main()
{
    int x;
    x = 0;
    clrscr();
    printf("Enter Choice (1 - 3): ");
    scanf("%d", &x);
    if (x == 1)
        printf("\nChoice is 1");
    else if ( x == 2)
        printf("\nChoice is 2");
    else if ( x == 3)
        printf("\nChoice is 3");
    else
        printf("\nInvalid Choice: Invalid Choice");
}
```

Trong chương trình trên,

Nếu x = 1, hiển thị dòng chữ "**Choice is 1**".

Nếu x = 2, hiển thị dòng chữ "**Choice is 2**".

Nếu x = 3, hiển thị dòng chữ "**Choice is 3**" được hiển thị.

Nếu x là bất kỳ một số nào khác 1, 2, hoặc 3, "**Invalid Choice**" được hiển thị.

Nếu chúng ta muốn thực hiện nhiều hơn một lệnh sau mỗi câu lệnh **if** hay **else**, ta phải đặt các câu lệnh đó vào trong cặp dấu ngoặc nhọn {}. Các câu lệnh đó tạo thành một nhóm gọi là lệnh phức hay một khối lệnh.

```
if (result >= 45)
{
      printf("Passed\n");
       printf("Congratulations\n");
}
else
{
       printf("Failed\n");
       printf("Good luck next time\n");
}
```

e. Các cấu trúc *if* lồng nhau:

Một cấu trúc **if lồng nhau** là một lệnh **if** được đặt bên trong một lệnh **if** hoặc **else** khác. Trong C, lệnh **else** luôn gắn với lệnh **if** không có **else** gần nó nhất, và nằm trong cùng một khối lệnh với nó. Ví dụ:

```
if (biểu thức-1)
{
     if (biểu thức-2)
          câu_lệnh1;
     if (biểu thức-3)
          câu_lệnh2;
     else
          câu_lệnh3;          /* với if (biểu thức-3)
*/
}
else
     câu_lệnh4;          /* với if (biểu thức-1) */
```

Trong đoạn lệnh minh họa ở trên, nếu giá trị của biểu thức-1 là **true** thì lệnh if thứ hai sẽ được kiểm tra. Nếu biểu thức-2 là **true** thì lệnh câu_lệnh1 sẽ được thực hiện. Nếu biểu thứu-3 là **true**, câu_lệnh2 sẽ được thực hiện nếu không câu_lệnh3 được thực hiện. Nếu biểu thức-1 là **false** thì câu_lệnh4 được thực hiện.

Vì lệnh **else** trong cấu trúc **else-if** là không bắt buộc, nên có thể có một cấu trúc khác như dạng dưới đây:

```
if (điều kiện-1)
     if (điều kiện-2)
          câu_lệnh1;
```

```
        else
            câu_lệnh2;
    câu lệnh kế tiếp;
```

Trong đoạn mã trên, nếu điều kiện-1 là **true**, chương trình sẽ chuyển đến thực hiện lệnh if thứ hai và điều kiện-2 được kiểm tra. Nếu điều kiện đó là **true**, câu_lệnh1 được thực hiện, nếu không câu_lệnh2 được thực hiện, sau đó chương trình thực hiện những lệnh trong **câu lệnh kế tiếp.** Nếu điều kiện-1 là **false**, chương trình sẽ chuyển đến thực hiện những lệnh trong **câu lệnh kế tiếp.**

Ví dụ, marks1 và marks2 là điểm hai môn học của một sinh viên. Điểm marks2 sẽ được cộng thêm 5 điểm nếu nó nhỏ hơn 50 và marks1 lớn hơn 50. Nếu marks2 lớn hơn hoặc bằng 50 thì sinh viên đạt loại 'A'. Điều này có thể được biểu diễn bởi đoạn if có cấu trúc như sau:

```
if (marks1 > 50 && marks2 < 50)
     marks2 = marks2 + 5;
if (marks2 >= 50)
     grade = 'A';
```

Một số người đưa ra đoạn code như sau:

```
if (marks1 > 50)
     if (marks2 < 50)
          marks2 = marks2 + 5;
     else
          grade = 'A';
```

Trong đoạn lệnh này, 'A' được gán cho biến grace chỉ khi marks1 lớn hơn 50 và marks2 lớn hơn hoặc bằng 50. Nhưng theo như yêu cầu của bài toán, biến grace được gán giá trị 'A' sau khi thực hiện việc kiểm tra để cộng điểm và kiểm tra giá trị của marks2. Hơn nữa, giá trị của biến grace không phụ thuộc vào marks1.

Vì lệnh **else** trong cấu trúc if-else là không bắt buộc, nên khi có lệnh **else** nào đó không được đưa vào trong chuỗi cấu trúc **if** lồng nhau chương trình sẽ trông không rõ ràng. Một lệnh **else** luôn được gắn với lệnh **if** gần nó nhất mà lệnh **if** này chưa được kết hợp với một lệnh **else** nào.

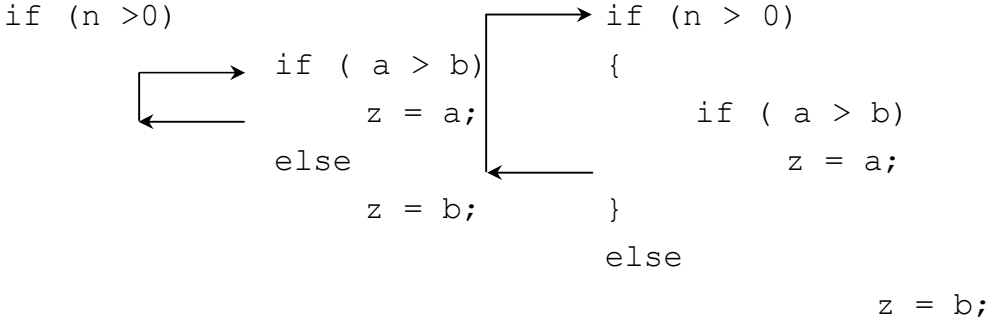
Ví dụ :

```
if (n >0)
        if ( a > b)
                z = a;
        else
                z = b;
```

Lệnh **else** đi với lệnh **if** bên trong. Việc viết lùi vào trong dòng là một cách thể hiện mối quan hệ đó. Tuy nhiên canh lề không có chức năng gắn else với lệnh if. Cặp dấu ngoặc nhọn {} giúp chúng ta thực hiện chức năng đó một cách chính xác.

```
if (n > 0)
{    if ( a > b)
          z = a;
}
else
     z = b;
```

Hình bên dưới biểu diễn sự kết hợp giữa **if** và **else** trong một chuỗi các lệnh **if** lồng nhau.



else kết hợp với **if** gần nhất

else kết hợp với **if** đầu tiên, bởi vì cặp dấu ngoặc nhọn đã đặt lệnh **if** bên trong.

Theo chuẩn ANSI, có thể lồng nhau đến 15 mức. Tuy nhiên, hầu hết trình biên dịch cho phép nhiều hơn thế.

Một ví dụ về if lồng nhau được cho bên dưới:

Ví dụ 7:

```
#include <stdio.h>
void main()
{
       int x, y;
       x = y = 0;
       clrscr();
       printf("Enter Choice (1 - 3): " );
       scanf("%d", &x);
       if(x == 1)
       {
          printf("\nEnter value for y (1 - 5): ");
```

```
            scanf ("%d", &y);
            if (y <= 5)
                printf("\nThe value for y is: %d", y);
            else
                printf("\nThe value of y exceeds 5");
        }
        else
            printf ("\nChoice entered was not 1");
    }
```

Trong chương trình trên, nếu giá trị của **x** được nhập là 1, người dùng được yêu cầu nhập tiếp giá trị của **y**. Ngược lại, dòng chữ "**Choice entered was not 1**" được hiển thị. Lệnh **if** đầu tiên có lồng một **if** trong đó để hiển thị giá trị của **y** nếu người dùng nhập vào một giá trị nhỏ hơn 5 cho y, hoặc ngược lại sẽ hiển thị dòng chữ "**The value of y exceeds 5**".

Chương trình dưới đây đưa ra cách sử dụng của **if lồng nhau.**

Ví dụ 8:

Một công ty sản xuất 3 loại sản phẩm có tên gọi: văn phòng phẩm cho máy tính (computer stationery), đĩa cứng (fixed disks) và máy tính (computer).

Sản phẩm	**Mã**
Computer Stationery	1
Fixed Disks	2
Computers	3

Công ty có chính sách giảm giá như sau:

Sản phẩm	**Giá trị đặt hàng**	**Tỷ lệ giảm giá**
Computer Stationery	$500/- hoặc hơn	12%
Computer Stationery	$300/- hoặc hơn	8%
Computer Stationery	dưới $300/-	2%
Fixed Disks	$2000/- hoặc hơn	10%
Fixed Disks	$1500/- hoặc hơn	5%
Computers	$5000/- hoặc hơn	10%
Computer	$2500/- hoặc hơn	5%

Dưới đây là chương trình tính giảm giá.

Ví dụ 9:

```
#include <stdio.h>

void main()
```

```
{
     int productcode;
     float orderamount, rate = 0.0;
     printf("\nPlease enter the product code: " );

     scanf("%d", &productcode);
     printf("Please enter the order amount: ");
     scanf("%f", &orderamount);

     if (productcode == 1)
     {
          if (orderamount >= 500)

               rate = 0.12;

          else if (orderamount >= 300)

               rate = 0.08;

          else

               rate = 0.02;
     }

     else if (productcode == 2)
     {

          if (orderamount >= 2000)

               rate = 0.10;

          else if (orderamount >= 1500)

               rate = 0.05;
     }

   else if (productcode == 3)
```

```
    {
        if (orderamount >= 5000)
            rate = 0.10;
        else if (orderamount >= 2500)
            rate = 0.05;
    }
    orderamount -= orderamount * rate;
    printf( "The net order amount is % .2f \n",
orderamount);
  }
```

Kết quả của chương trình được minh hoạ như sau:

```
Please enter the product code: 3
Please enter the order amount: 6000
The net order amount is 5400
```

Ở trên, **else** sau cùng trong chuỗi các **else-if** không cần kiểm tra bất kỳ điều kiện nào. Ví dụ, nếu mã sản phẩm được nhập vào là 1 và giá trị đặt hàng nhỏ hơn $300, thì không cần phải kiểm tra điều kiện, vì tất cả các khả năng đã được kiểm soát.

Kết quả thực thi chương trình với mã sản phẩm là 3 và giá trị đặt hàng là $6000 được trình bày ở trên.

Sửa đổi chương trình trên để chú ý đến trường hợp dữ liệu nhập là một mã sản phẩm không hợp lệ. Điều này có thể dễ dàng đạt được bằng cách thêm một lệnh **else** vào chuỗi lệnh **if** dùng kiểm tra mã sản phẩm. Nếu gặp một mã sản phẩm không hợp lệ, chương trình phải kết thúc mà không cần tính giá trị thực của đơn đặt hàng.

f. Câu lệnh 'sw*itch*':

Câu lệnh **switch** cho phép ta đưa ra quyết định có nhiều cách lựa chọn, nó kiểm tra giá trị của một biểu thức trên một danh sách các hằng số nguyên hoặc kí tự. Khi nó tìm thấy một giá trị trong danh sách trùng với giá trị của biểu thức điều kiện, các câu lệnh gắn với giá trị đó sẽ được thực hiện. Cú pháp tổng quát của lệnh **switch** như sau:

```
switch (biểu_thức)
     {    case hằng_1:
               chuỗi_câu_lệnh;
               break;
          case hằng_2:
               chuỗi_câu_lệnh;
               break;
          case hằng_3:
               chuỗi_câu_lệnh;
               break;
          default:
               chuỗi_câu_lệnh;
     }
```

Ở đó, **switch**, **case** và **default** là các từ khoá, chuỗi_câu_lệnh có thể là lệnh đơn hoặc lệnh ghép và không cần đặt trong cặp dấu ngoặc. Biểu_thức theo sau từ khóa **switch** phải được đặt trong dấu ngoặc **( )**, và toàn bộ phần thân của lệnh switch phải được đặt trong cặp ngoặc nhọn **{ }**. Kiểu dữ liệu kết quả của biểu_thức và kiểu dữ liệu của các hằng theo sau từ khoá case phải đồng nhất. Chú ý, hằng số sau **case** chỉ có thể là một **hằng** số nguyên hoặc **hằng** ký tự. Nó cũng có thể là các hằng biểu thức – những biểu thức không chứa bất kỳ một biến nào. Tất cả các giá trị của case phải khác nhau.

Trong câu lệnh **switch**, biểu thức được xác định giá trị, giá trị của nó được so sánh với từng giá trị gắn với từng case theo thứ tự đã chỉ ra. Nếu một giá trị trong một case trùng với giá trị của biểu thức, các lệnh gắn với case đó sẽ được thực hiện. Lệnh **break** (sẽ nói ở phần sau) cho phép thoát ra khỏi **switch**. Nếu không dùng lệnh **break**, các câu lệnh gắn với case bên dưới sẽ được thực hiện không kể giá trị của nó có trùng với giá trị của biểu thức điều kiện hay không. Chương trình cứ tiếp tục thực hiện như vậy cho đến khi gặp một lệnh **break**. Chính vì thế, lệnh **break** được coi là lệnh quan trọng nhất khi dùng **switch**.

Các câu lệnh gắn với **default** sẽ được thực hiện nếu không có **case** nào thỏa mãn. Lệnh **default** là tùy chọn. Nếu không có lệnh **default** và không có **case** nào thỏa mãn, không có hành động nào được thực hiện. Có thể thay đổi thứ tự của **case** và **default**.

Xét một ví dụ.

Ví dụ 10:

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

```
        char ch;
        clrscr ();

        printf("\nEnter a lower cased alphabet (a - z): ");
        scanf("%c", &ch);

        if (ch < 'a' || ch > 'z')
            printf("\nCharacter not a lower cased
alphabet");
        else
            switch (ch)
            {
            case 'a':
            case 'e':
            case 'i':
            case 'o':
            case 'u':
                printf("\nCharacter is a vowel");
                break;

            case 'z':
                printf ("\nLast Alphabet (z) was entered");
                break;

            default:
                printf("\nCharacter is a consonant");
                break;
            }
    }
```

Chương trình trên nhận vào một kí tự ở dạng chữ thường và hiển thị thông báo kí tự đó là nguyên âm, là chữ z hay là một phụ âm. Nếu nó không phải ba loại ở trên, chương trình hiển thị thông báo "**Character not a lower cased alphabet**".

Nên sử dụng lệnh **break** trong cả **case** cuối cùng hoặc **default** mặc dù về mặt logic là không cần thiết. Nhưng điều đó rất có ích nếu sau này chúng ta đưa thêm case vào cuối.

Dưới đây là một ví dụ, ở đó biểu thức của **switch** là một biến kiểu số nguyên và giá trị của mỗi case là một số nguyên.

Ví dụ 11:

```
/* Integer constants as case labels */
#include <stdio.h>
void main()
{
      int basic;

      printf("\n Please enter your basic: ");

      scanf("%d", &basic);
      switch (basic)
      {
           case 200:
                printf("\n Bonus is dollar %d\n", 50);
                break;

           case 300:
                printf("\n Bonus is dollar %d\n", 125);
                break;

           case 400:
                printf("\n Bonus is dollar %d\n", 140);
                break;

           case 500:
                printf("\n Bonus is dollar %d\n", 175);
                break;

           default:
                printf("\n Invalid entry");
                break;
      }
}
```

Từ ví dụ trên, lệnh **switch** rất thuận lợi khi chúng ta muốn kiểm tra một biểu thức dựa trên một danh sách giá trị riêng biệt. Nhưng nó không thể dùng để kiểm tra một giá trị có nằm trong một miền nào đó hay không. Ví dụ, không thể dùng switch để kiểm tra xem basic có nằm trong khoảng từ 200 đến 300 hay không, để từ đó xác định mức tiền thưởng. Trong những trường hợp như vậy, ta phải sử dụng **if-else**.

4.2.4. Quy trình kỹ thuật trong bài học của GV và SV:

*** Quy trình thị phạm của GV**

- Bước 1: Chuẩn bị bài giảng, tài liệu và điều kiện trang thiết bị giảng dạy
- Bước 2: Chuẩn bị bài tập mẫu và đưa ra nội dung thực hành
- Bước 3: Phổ biến các quy trình, các bước thực hiện
- Bước 4: Hướng dẫn nội dung bài học, Hướng dẫn thị phạm mẫu
- Bước 5: Quản lý lớp học
- Bước 6: Giải đáp, tư vấn, đánh giá, nhận xét
- Bước 7: Chấm điểm và nộp các bài đánh giá cho bộ môn

*** Quy trình thực hiện bài của SV**

- Bước 1: SV chuẩn bị tài liệu, vở, bút, giấy nháp, tài liệu tra cứu
- Bước 2: Đọc tài liệu Lập trình căn bản C phần cấu trúc rẽ nhánh có điều kiện.
- Bước 3: Sinh viên trình bày bài thực hành, thảo luận và nhận xét góp ý.

4.2.5. Các mẫu hình sản phẩm cho SV tham khảo:

- SV xem video

4.2.6. Phần tự thực hành thao tác thường xuyên của SV trong bài học:

Câu 1: Viết chương trình nhập vào hai số a và b, và kiểm tra xem a có chia hết cho b hay không.

Câu 2: Viết chương trình nhập vào hai số và kiểm tra xem tích của hai số này bằng hay lớn hơn 1000.

Câu 3: Viết chương trình nhập vào hai số. Tính hiệu của hai số này. Nếu hiệu số này bằng với một trong hai số đã nhập thì hiển thị thông tin:

Hiệu bằng giá tri <giá trị của số đã nhập vào>

Nếu hiệu không bằng với một trong hai giá trị đã nhập, hiển thị thông tin:

Hiệu không bằng bất kỳ giá trị nào đã được nhập

Câu 4: Công ty Montek đưa ra các mức trợ cấp cho nhân viên ứng với từng loại nhân viên như sau:

Loại nhân viên	Mức trợ cấp
A	300
B	250
Những loại khác	100

Tính lương cuối tháng của nhân viên (Mức lương và loại nhân viên được nhập từ người dùng).

Câu 5: Viết chương trình xếp loại sinh viên theo các qui luật dưới đây:

Nếu điểm => 75 - Loại A

Nếu 60 <= điểm < 75 - Loại B

Nếu 45 <= điểm < 60 - Loại C

Nếu 35 <= điểm < 45 - Loại D

Nếu điểm < 35 - Loại E

4.2.7. Sản phẩm thực hành:

+ Các ví dụ demo.

4.2.8. Điều kiện để GV- SV thực hiện bài học thực hành;

4.2.8.1. Điều kiện chuẩn bị trước bài học

+ Giảng viên:

- Đề cương chi tiết bài giảng
- Thiết kế bài giảng
- Học liệu tham khảo, mở rộng

+ Sinh viên:

- Đọc kỹ bài học trước khi giảng viên lên lớp
- Chuẩn bị vấn đề thảo luận, trao đổi với giảng viên
- Các tài liệu tham khảo, mở rộng do giảng viên yêu cầu

4.2.8.2. Kỹ thuật và phương tiện dạy học:

- Bảng, phấn viết
- Màn hình lớn
- Máy tính, máy chiếu

4.2.8.3. Phương pháp tổ chức dạy-học:

- Thuyết trình, diễn giảng, thực hành
- Phát vấn
- Trao đổi, thảo luận
- Cemina
- Vấn đáp

4.3 Nội dung bài giảng 2

4.3.1. Tên bài giảng: BÀI 2: CẤU TRÚC VÒNG LẶP

Số tiết lên lớp của GV: 05 tiết; Số tiết tự làm bài của SV : 05 tiết

4.3.2. Phần mở đầu tiếp cận bài;

Một trong những điểm mạnh lớn nhất của máy tính là khả năng thực hiện một chuỗi các lệnh lặp đi lặp lại. Điều đó có được là do sử dụng các cấu trúc lặp trong ngôn ngữ lập trình. Trong bài này bạn sẽ tìm hiểu các loại vòng lặp khác nhau trong C.

4.3.3. Phần kiến thức, kỹ thuật căn bản:

4.3.3.1 Phần kiến thức can bản

- **Mục tiêu:**

Kết thúc bài học này, bạn có thể:

- Hiểu được vòng lặp 'for' trong C
- Làm việc với toán tử 'phẩy'
- Hiểu các vòng lặp lồng nhau
- Hiểu vòng lặp 'while' và vòng lặp 'do-while'
- Làm việc với lệnh 'break' và lệnh 'continue'
- Hiểu hàm 'exit()'.

- **Vòng lặp:**

Vòng lặp là một đoạn mã lệnh trong chương trình được thực hiện lặp đi lặp lại cho đến khi thỏa mãn một điều kiện nào đó. Vòng lặp là một khái niệm cơ bản trong lập trình cấu trúc.

Trong C có các loại vòng lặp sau:

Vòng lặp **for**

Vòng lặp **while**

Vòng lặp **do…while**

Ta sử dụng các **toán tử quan hệ** và **toán tử logic** trong các biểu thức điều kiện để điều khiển sự thực hiện của vòng lặp.

- **Vòng lặp '*for*':**

Cú pháp tổng quát của vòng lặp **for** như sau:

```
for(khởi tạo giá trị cho biến điều khiển; biểu thức điều kiện;biểu thức thay đổi giá trị của biến điều khiển)
{
    Câu lệnh (các câu lệnh);
}
```

Khởi tạo giá trị cho biến điều khiển là một câu lệnh gán giá trị ban đầu cho biến điều khiển trước khi thực hiện vòng lặp. Lệnh này chỉ được thực hiện duy nhất một lần. *Biểu thức điều kiện* là một biểu thức quan hệ, xác định điều kiện thoát cho vòng lặp. *Biểu thức thay đổi giá trị của biến điều khiển* xác định biến điều khiển sẽ bị thay đổi như thế nào sau mỗi lần vòng lặp được lặp lại (thường là tăng hoặc giảm giá trị của biến điều khiển). Ba phần trên được phân cách bởi dấu chấm phẩy. Câu lệnh trong thân vòng lặp có thể là một lệnh duy nhất (lệnh đơn) hoặc lệnh phức (nhiều lệnh).

Vòng lặp for sẽ tiếp tục được thực hiện chừng nào mà biểu thức điều kiện còn **đúng (true)**. Khi biểu thức điều kiện là **sai (false)**, chương trình sẽ thoát ra khỏi vòng lặp **for**.

Xem ví dụ sau:

```
/* Đây là chương trình minh họa vòng lặp for trong chương trình C*/
#include <stdio.h>
main()
{
int count;
printf("\t This is a \n");
for (count = 1; count <= 6; count++)
     printf("\n \t \t nice");
printf("\n\t\t world. \n");
}
```

Kết quả của chương trình trên được minh họa như sau:

```
This is a
nice
nice
nice
nice
nice
nice
world.
```

Chúng ta sẽ xem xét kĩ đoạn vòng lặp **for** trong chương trình trên:

1. Khởi tạo giá trị cho biến điều khiển: c*ount = 1.*

Lệnh này được thực hiện duy nhất một lần khi vòng lặp bắt đầu được thực hiện, và biến count được đặt giá trị là 1.

2. Biểu thức điều kiện: *count < = 6.*

Chương trình kiểm tra xem giá trị hiện tại của biến count có nhỏ hơn hay bằng

6 hay không. Nếu đúng, các câu lệnh trong thân vòng lặp sẽ được thực hiện.

3. Thân của vòng lặp có duy nhất một lệnh

```
printf("\n \t \t nice");
```

Câu lệnh này có thể đặt trong cặp dấu ngoặc nhọn **{}** cho dễ nhìn.

4. Biểu thức thay đổi giá trị của biến điều khiển *count*++, tăng giá trị của biến *count* lên 1 cho lần lặp kế tiếp.

Các bước 2, 3, 4 được lặp lại cho đến khi biểu thức điều kiện là **sai**. Vòng lặp trên sẽ được thực hiện 6 lần với giá trị của *count* thay đổi từ 1 đến 6. Vì vậy, từ *nice* xuất hiện 6 lần trên màn hình. Sau đó, *count* tăng lên 7. Do giá trị này lớn hơn 6, vòng lặp kết thúc và câu lệnh sau vòng lặp được thực hiện.

Chương trình sau in ra các số chẵn từ 1 đến 25.

Ví dụ 2:

```
#include <stdio.h>
main()
{
int num;
     printf("The even numbers from 1 to 25 are:
\n\n");
     for (num2; num <= 25; num+=2)
          printf("%d\n", num);
}
```

Kết quả của chương trình trên như sau:

```
The even numbers from 1 to 25 are:
2
4
6
8
10
12
14
16
18
20
22
24
```

Vòng lặp **for** ở trên khởi tạo giá trị của biến nguyên *num* là 2 (để lấy một số chẵn) và tăng giá trị của nó lên 2 mỗi lẫn vòng lặp được lặp lại.

Trong các vòng lặp **for**, biểu thức điều kiện luôn được kiểm tra ngay khi bắt đầu vòng lặp. Do đó các lệnh trong thân vòng lập sẽ không được thực hiện nếu ngay từ ban đầu điều kiện đó là **sai**.

- **Toán tử *'phẩy (comma)'*:**

Phần biểu thức trong toán tử for có thể được mở rộng để thêm vào các lệnh khởi tạo hay các lệnh thay đổi giá trị của biến. Cú pháp như sau:

```
biểu_thức1 , biểu_thức2
```

Các biểu thức trên được phân cách bởi toán tử **'phẩy'** (,), và được thực hiện từ trái sang phải. Thứ tự của các biểu thức là quan trọng trong trường hợp giá trị của biểu thức thứ hai phụ thuộc vào giá trị của **biểu thức thứ nhất**. Toán tử này có độ ưu tiên thấp nhất trong các toán tử của C.

Ví dụ dưới đây in ra một bảng các phép cộng với kết quả không đổi để minh họa khái niệm về toán tử phẩy rõ ràng hơn.

Ví dụ 3:

```
#include <stdio.h>
main()
{
int  i, j, max;
     printf("Please enter the maxinum value \n");
     printf("for which a table can be printed: ");
     scanf("%d", &max);
     for (i = 0, j = max; i < = max; i++, j--)
          printf("\n%d + %d = %d", i, j, i + j);
}
```

Kết quả của chương trình trên được minh họa như sau:

```
Please enter the maxinum value
for which a table can be printed: 5
0     +     5     =     5
1     +     4     =     5
2     +     3     =     5
3     +     2     =     5
4     +     1     =     5
5     +     0     =     5
```

Chú ý trong vòng lặp for, phần khởi tạo giá trị là:

```
i = 0, j = max
```

Khi vòng lặp bắt đầu chạy, *i* được gán giá trị 0 và *j* được gán giá trị của **max**.

Phần thay đổi giá trị của biến điều khiển gồm hai biểu thức:

```
i++, j—
```

sau mỗi lần thực hiện thân vòng lặp, *i* được tăng lên 1 và *j* giảm đi 1. Tổng của hai biến đó luôn bằng **max** và được in ra màn hình:

- **Vòng lặp *'for lồng nhau'*:**

Một vòng lặp **for** được gọi là lồng nhau khi nó nằm bên trong một vòng lặp **for** khác. Nó sẽ có dạng tương tự như sau:

```
for (i = 1;  i < max1; i++)
{    ….
     ….
     for (j = 0; j < max2 ; j++)
     {
          …..
     }
     ….
}
```

Xem ví dụ sau:

Ví dụ 4:

```
#include <stdio.h>
main()
{
int  i, j, k;
i = 0;
printf("Enter no. of row: ");
scanf("%d", &i);
printf("\n");
for (j = 0; j < i; j++)
{
printf("\n");
     for (k = 0; k <= j; k++)    /*vòng  lặp  for  bên
trong*/
          printf("*");
}
}
```

Chương trình trên sẽ hiển thị ký tự '*' trên mỗi dòng và số ký tự '*' trên mỗi dòng sẽ tăng thêm 1. Chương trình sẽ nhận vào số dòng, từ đó ký tự '*' sẽ được in ra. Ví dụ, nếu nhập vào số 5, kết quả như sau

```
*
**
***
****
*****
```

➢ **Các trường hợp khác của vòng lặp *'for'*:**

Vòng lặp **for** có thể được sử dụng mà không cần phải có đầy đủ các thành phần của nó.

Ví dụ,

```
…
for (num = 0; num != 255;)
{    printf("Enter no. ");
     scanf("%d",&num);
     …
}
```

Đoạn mã trên sẽ yêu cầu nhập giá trị cho biến num cho đến khi nhập vào 255. Vòng lặp không có phần thay đổi giá trị của biến điều khiển. Vòng lặp sẽ kết thúc khi biến num có giá trị 255.

Tương tự, xét ví dụ sau:

```
.
.
printf("Enter value for checking :");
scanf("%d", &num);

for(; num < 100; )
{
    .
    .
}
```

Vòng lặp trên không có phần khởi tạo tham số và phần thay đổi giá trị của tham số.

Vòng lặp **for** khi không có bất kỳ thành phần nào sẽ là một vòng lặp vô tận

```
for ( ; ; )
printf("This loop will go on and on and on… \n");
```

Tuy nhiên, lệnh **break** bên trong vòng lặp sẽ cho phép thoát khỏi vòng lặp.

```
…
for ( ; ; )
```

```
{    printf("This will go on and on");
     i = getchar();
     if (i == 'X' || i == 'x');
          break;
}
…
```

Vòng lặp trên sẽ được thực hiện cho đến khi người dùng nhập vào *x* hoặc *X*.

Vòng lặp **for** (hay vòng lặp bất kì) có thể không có bất kì lệnh nào trong phần thân của nó. Kĩ thuật này giúp tăng tính hiệu quả trong một vài giải thuật và để tạo ra độ trễ về mặt thời gian.

```
for (i = 0; i < xyz_value; i++);
```

là một ví dụ để tạo ra độ trễ về thời gian.

- **Vòng lặp *'while'*:**

Cấu trúc lặp thứ hai trong C là vòng lặp **while**. Cú pháp tổng quát như sau:

```
while (điều_kiện là đúng)
     câu_lệnh;
```

Ở đó, câu_lệnh có thể là rỗng, hay một lệnh đơn, hay một khối lệnh. Nếu vòng lặp **while** chứa một tập các lệnh thì chúng phải được đặt trong cặp ngoặc xoắn **{}**. *điều_kiện* có thể là biểu thức bất kỳ. Vòng lặp sẽ được thực hiện lặp đi lặp lại khi điều kiện trên là **đúng (true)**. Chương trình sẽ chuyển đến thực hiện lệnh tiếp sau vòng lặp khi điều kiện trên là **sai (false)**.

Vòng lặp **for** có thể được sử dụng khi số lần thực hiện vòng lặp đã được xác định trước. Khi số lần lặp không biết trước, vòng lặp **while** có thể được sử dụng.

Ví dụ 5:

```
/* A simple program using the while loop*/
#include <stdio.h>
main()
{
int count = 1;
     while (count <= 10)
     {    printf("\n This is iteration %d\n", count);
          count++;
     }
     printf("\nThe loop is completed. \n");
}
```

Kết quả của chương trình trên được minh họa như sau:

```
This is iteration 1
This is iteration 2
```

```
This is iteration 3
This is iteration 4
This is iteration 5
This is iteration 6
This is iteration 7
This is iteration 8
This is iteration 9
This is iteration 10
The loop is completed.
```

Đầu tiên chương trình gán giá trị của *count* là 1 ngay trong câu lệnh khai báo nó. Sau đó chương trình chuyển đến thực hiện lệnh **while**. Phần biểu thức điều kiện được kiểm tra. Giá trị hiện tại của *count* là 1, nhỏ hơn 10. Kết quả kiểm tra điều kiện là **đúng (true)** nên các lệnh trong thân vòng lặp **while** được thực hiện. Các lệnh này được đặt trong cặp dấu ngoặc nhọn {}. Giá trị của biến *count* là 2 sau lần lặp đàu tiên. Sau đó biểu thức điều kiện lại được kiểm tra lần nữa. Quá trình này cứ lặp đi lặp lại cho đến khi giá trị của *count* lớn hơn 10. Khi vòng lặp kết thúc, lệnh *printf()* thứ hai được thực hiện.

Giống như vòng lặp for, vòng lặp while kiểm tra điều kiện ngay khi bắt đầu thực hiện vòng lặp. Do đó các lệnh trong thân vòng lặp sẽ không được thực hiện nếu ngay từ ban đầu điều kiện đó là sai

Biểu thức điều kiện trong vòng lặp có thể phức tạp tùy theo yêu cầu của bài toán. Các biến trong biểu thức điều kiện có thể bị thay đổi giá trị trong thân vòng lặp, nhưng cuối cùng điều kiện đó phải **sai (false)** nếu không vòng lặp sẽ không bao giờ kết thúc. Sau đây là ví dụ về một vòng lặp **while** vô hạn.

Ví dụ 6:

```
#include <stdio.h>
main()
{    int count = 0;
     while (count < 100)
     {
printf("This goes on forever, HELP!!!\n");
          count += 10;
          printf("\t%d", count);
          count -= 10;
          printf("\t%d", count);
          printf("\Ctrl - C will help");
     }
}
```

Ở trên, *count* luôn luôn bằng 0, nghĩa là luôn nhỏ hơn 100 và vì vậy biểu thức luôn luôn trả về giá trị **true**. Nên vòng lặp không bao giờ kết thúc.

Nếu có hơn một điều kiện được kiểm tra để kết thúc vòng lặp, vòng lặp sẽ kết thúc khi có ít nhất một điều kiện trong các điều kiện đó là **false**. Ví dụ sau sẽ minh họa điều này.

```
#include <stdio.h>
main()
{    int i, j;
     i = 0;
     j = 10;
     while (i < 100 && j > 5)
     {    ...
          i++;
     j -= 2;
}
...
}
```

Vòng lặp này sẽ thực hiện 3 lần, lần lặp thứ nhất *j* sẽ là 10, lần lặp kế tiếp *j* bằng 8 và lần lặp thứ ba *j* sẽ bằng 6. Khi đó *i* vẫn nhỏ hơn 100 (i bằng 3), *j* nhận giá trị 4 và điều kiện j > 5 trở thành **false**, vì vậy vòng lặp kết thúc.

Chúng ta hãy viết một chương trình nhận dữ liệu từ bàn phím và in ra màn hình. Chương trình kết thúc khi bạn nhấn phím **^Z (Ctrl + Z).**

Ví dụ 7:

```
/* ECHO PROGRAM */

/* A program to accept input data from the console
and print it on the screen */
/* End of input data is indicated by pressing '^Z'*/
#include <stdio.h>
main()
{
char ch;
while ((ch = getchar()) != EOF)
{
     putchar(ch)
}
}
```

Kết quả của chương trình trên được minh họa như sau:

Ví dụ một kết quả thực thi như sau:

```
Have
Have
a
a
good
good
day
day
^Z
```

Dữ liệu người dùng nhập vào được in đậm. Chương trình làm việc như thế nào ? Sau khi nhập vào một tập hợp các ký tự, nội dung của nó sẽ được in hai lần lên màn hình khi bạn nhấn **<Enter>**. Điều này là do các ký tự bạn nhập vào từ bàn phím được lưu trữ trong bộ đệm bàn phím. Và lệnh *putchar()* sẽ lấy nó từ bộ đệm sau khi bạn nhấn phím **<Enter>**. Chú ý cách thức kết thúc quá trình nhập dũe liệu bằng tổ hợp phím **^Z**, đây là kí tự kết thúc file tront DOS.

- **Vòng lặp *'do ... while'*:**

Vòng lặp **do ... while** còn được gọi là vòng lặp **do** trong C. Không giống như vòng lặp **for** và **while**, vòng lặp này kiểm tra điều kiện tại cuối vòng lặp. Điều này có nghĩa là vòng lặp **do ... while** sẽ được thực hiện ít nhất một lần, ngay cả khi điều kiện là sai **(false)** ở lần chạy đầu tiên.

Cú pháp tổng quát của vòng lặp **do ... while** như sau:

```
do{
     câu_lệnh;
} while (điều_kiện);
```

Cặp dấu ngoặc **{}** là không cần thiết khi chỉ có một câu lệnh hiện diện trong vòng lặp, nhưng việc sử dụng dấu ngoặc {} là một thói quen tốt. Vòng lặp **do ... while** lặp đến khi *điều_kiện* mang giá trị **false**. Trong vòng lặp **do ... while**, ***câu_lệnh*** (khối các câu lệnh) sẽ được thực thi trước, và sau đó *điều_kiện* được kiểm tra. Nếu điều kiện là true, chương trình sẽ quay lại thực hiện lệnh do. Nếu điều kiện là false, chương trình chuyển đến thực hiện lệnh nằm sau vòng lặp.

Xét chương trình sau:

Ví dụ 8:

```
/* accept only int value */
#include <stdio.h>
void main()
```

```
{    int num1, num2;
     num2 = 0;
     do{
          printf("\nEnter a number: ");
          scanf("%d",&num1);
          printf("No. is %d", num1);
          num2++;
     }while (num1 != 0);
     printf("\nThe total numbers entered were %d",--num2);
/* num2 is decremented before printing because count
for last integer (0) is not to be considered */
}
```

Kết quả của chương trình được minh họa như sau:

```
Enter a number: 10
No. is 10
Enter a number: 300
No. is 300
Enter a number: 45
No. is 45
Enter a number: 0
No. is 0
The total numbers entered were 3
```

Đoạn chương trình trên sẽ nhận các số nguyên và hiển thị chúng cho đến khi một số 0 được nhập vào. Và sau đó chương trình sẽ thoát khỏi vòng lặp **do ... while** và số lượng các số nguyên đã được nhập vào.

- **Các vòng lặp 'while lồng nhau' và 'do ... while'**

Cũng giống như vòng lặp **for**, các vòng lặp **while** và **do ... while** cũng có thể được lồng vào nhau. Hãy xem một ví dụ được đưa ra dưới đây.

Ví dụ 9:

```
#include <stdio.h>
void main()
{
int x;
     char i, ans;
     i = '';
     do{
          clrscr();
     x = 0;
```

```
            ans = 'y';
            printf("\nEnter sequence of character: ");
        do{
                i = getchar();
                x++;
            }while (i != '\n');
        i = '';
            printf("\nNumber of characters entered
is:%d", --x);
            printf("\nMore sequences (Y/N)?");
        ans = getch();
        }while (ans == 'Y' || ans == 'y');
    }
```

Kết quả của chương trình được minh họa như sau:

```
Enter sequence of character: Good Morning!
Number of character entered is: 14
More sequences (Y/N)? N
```

Chương trình trên yêu cầu người dùng nhập vào một chuỗi kí tự cho đến khi nhấn phím enter (**vòng lặp while bên trong**). Khi đó, chương trình thoát khỏi vòng lặp **do…while** bên trong. Sau đó chương trình hỏi người dùng có muốn nhập tiếp nữa hay thôi. Nếu người dùng nhấn phím 'y' hoặc 'Y', điều kiện cho vòng while bên ngoài là **true** và chương trình nhắc người dùng nhập vào chuỗi ký tự khác. Chương trình cứ tiếp tục cho đến khi người dùng nhấn bất kỳ một phím nào khác với phím 'y' hoặc 'Y'. Và chương trình kết thúc.

- **Các lệnh nhẩy:**

C có bốn câu lệnh thực hiện sự rẽ nhánh không điều kiện: **return**, **goto**, **break**, và **continue**. Sự rẽ nhánh không điều kiện nghĩa là sự chuyển điều khiển từ một điểm đến một lệnh xác định. Trong các lệnh chuyển điều khiển trên, **return** và **goto** có thể dùng bất kỳ vị trí nào trong chương trình, trong khi lệnh **break** và **continue** được sử dụng kết hợp với các câu lệnh vòng lặp.

- **Lệnh *'return'*:**

Lệnh **return** dùng để quay lại vị trí gọi hàm sau khi các lệnh trong hàm đó được thực thi xong. Trong lệnh **return** có thể có một giá trị gắn với nó, giá trị đó sẽ được trả về cho chương trình. Cú pháp tổng quát của câu lệnh **return** như sau:

```
return biểu_thức;
```

Biểu_thức là một tùy chọn (không bắt buộc). Có thể có hơn một lệnh **return** được sử dụng trong một hàm. Tuy nhiên, hàm sẽ quay trở về vị trí gọi hàm khi gặp lệnh **return** đầu tiên. Lệnh **return** sẽ được làm rõ hơn sau khi học về hàm.

- **Lệnh *'goto'*:**

C là một ngôn ngữ lập trình có cấu trúc, tuy vậy nó vẫn chứa một số câu lệnh làm phá vớ cấu trúc của chương trình:

- goto
- label

Lệnh **goto** cho phép chuyển quyền điều khiển tới một lệnh bất kì nằm trong cùng khối lệnh hay khác khối lệnh bên trong hàm đó. Vì vậy nó vi phạm các qui tắc của một ngôn ngữ lập trình có cấu trúc.

Cú pháp tổng quát của một câu lệnh **goto** là:

goto label;

Trong đó **label** là một định danh phải xuất hiện như là tiền tố (prefix) của một câu lệnh khác trong cùng một hàm. Dấu chấm phẩy (**;**) sau **label** đánh dấu sự kết thúc của lệnh **goto**. Các lệnh **goto** làm cho chương trình khó đọc. Chúng làm giảm độ tin cậy và làm cho chương trình khó bảo trì. Tuy nhiên, chúng vẫn được dùng vì chúng cung cấp các cách thức hữu dụng để thoát ra khỏi những vòng lặp lồng nhau quá nhiều mức. Xét đoạn mã sau:

```
for (...) {
     for(...) {
          for(...) {
               while(...) {
                    if (...) goto error1;
                         ...
               }
          }
     }
}
error1: printf("Error !!!");
```

Như đã thấy, **label** xuất hiện như là một tiền tố của một câu lệnh khác trong chương trình.

```
label: câu_lệnh
```

hoặc

```
label:
{
     Chuỗi các câu lệnh;
```

```
}
```

Ví dụ 10:

```
#include <stdio.h>
#include <conio.h>
void main()
{
int num
     clrscr();
     label1:
          printf("\nEnter a number (1): ");
     scanf("%d", &num);
          if (num == 1)
               goto Test;
     else
               goto label1;
Test:
     printf("All done...");
}
```

Kết quả của chương trình trên được minh họa như sau:

```
Enter a number: 4
Enter a number: 5
Enter a number: 1
All done...
```

- **Lệnh *'break'*:**

Câu lệnh **break** có hai cách dùng. Nó có thể được sử dụng để kết thúc một **case** trong câu lệnh **switch** hoặc để kết thúc ngay một vòng lặp, mà không cần kiểm tra điều kiện vòng lặp.

Khi chương trình gặp lệnh **break** trong một vòng lặp, ngay lập tức vòng lặp được kết thúc và quyền điều khiển chương trình được chuyển đến câu lệnh theo sau vòng lặp. Ví dụ,

Ví dụ 11:

```
#include <stdio.h>
void main()
{    int count1, count2;
     for (count1= 1,count2 = 0; count1 <= 100; count1++)
     {    printf("Enter %d Count2: " count1);
          scanf("%d",count2);
          if (count2==100) break;
```

```
        }
}
```

Kết quả của chương trình trên được minh họa như sau:

```
Enter 1 count2: 10
Enter 2 count2: 20
Enter 3 count2: 100
```

Trong đoạn mã lệnh trên, người dùng có thể nhập giá trị 100 cho *count2*. Tuy nhiên, nếu 100 được nhập vào, vòng lặp kết thúc và điều khiển được chuyển đến câu lệnh kế tiếp.

Một điểm khác cần lưu ý là việc sử dụng câu lênh **break** trong các lệnh lặp lồng nhau. Khi chương trình thực thi đến một lệnh **break** nằm trong một vòng lặp **for** lồng bên trong một vòng lặp **for** khác, quyền điều khiển được chuyển trở về vòng lặp **for** bên ngoài.

- **Lệnh *'continue'*:**

Lệnh **continue** kết thúc lần lặp hiện hành và bắt đầu lần lặp kế tiếp. Khi gặp lệnh này trong chương trình, các câu lệnh còn lại trong thân của vòng lặp được bỏ qua và quyền điều khiển được chuyển đến bước đầu của vòng lặp trong lần lặp kế tiếp.

Trong trường hợp vòng lặp **for**, **continue** thực hiện biểu thức thay đổi giá trị của biến điều khiển và sau đó kiểm tra biểu thức điều kiện. Trong trường hợp của lệnh **while** và **do…while**, quyền điều khiển chương trình được chuyển đến biểu thức kiểm tra điều kiện. Ví dụ:

Ví dụ 12:

```
#include <stdio.h>
void main()
{
int num;
     for (num = 1; num <= 100; num++)
     {
if (num % 9 == 0) countinue;
          printf("%d\t", num);
     }
}
```

Chương trình trên in ra tất cả các số từ 1 đến 100 không chia hết cho 9. Kết quả chương trình được trình bày như sau:

1	2	3	4	5	6	7	8	10	11	12	13
14	15	16	17	19	20	21	22	23	24	25	26
28	29	30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	46	47	48	49	50	51	52
53	55	56	57	58	59	60	61	62	63	64	65
66	67	68	69	70	71	73	74	75	76	77	78
79	80	82	83	84	85	86	87	88	89	91	92
93	94	95	96	97	98	100					

- **Hàm *'exit()'*:**

Hàm *exit()* là một hàm trong thư viện chuẩn của C. Nó làm việc tương tự như một lệnh chuyển quyền điều khiển, điểm khác nhau chính là các lệnh chuyển quyền điều khiển thường được sử dụng để thoát khỏi một vòng lặp, trong khi *exit()* được sử dụng để thoát khỏi chương trình. Hàm này sẽ ngay lập tức kết thúc chương trình và quyền điều khiển được trả về cho hệ điều hành. Hàm *exit()* thường được dùng để kiểm tra một điều kiện bắt buộc cho việc thực thi của một chương trình có được thoả mãn hay không. Cú pháp tổng quát của hàm *exit()* như sau:

```
exit (int mã_trả_về);
```

ở đó ***mã_trả_về*** là một tùy chọn. Số 0 thường được dùng như một ***mã_trả_về*** để xác định sự kết thúc chương trình một cách bình thường. Những giá trị khác xác định một vài loại lỗi.

4.3.4. Quy trình kỹ thuật trong bài học của GV và SV:

*** Quy trình thị phạm của GV**

- Bước 1: Chuẩn bị bài giảng, tài liệu và điều kiện trang thiết bị giảng dạy
- Bước 2: Chuẩn bị bài tập mẫu và đưa ra nội dung thực hành
- Bước 3: Phổ biến các quy trình, các bước thực hiện
- Bước 4: Hướng dẫn nội dung bài học, Hướng dẫn thị phạm mẫu
- Bước 5: Quản lý lớp học
- Bước 6: Giải đáp, tư vấn, đánh giá, nhận xét
- Bước 7: Chấm điểm và nộp các bài đánh giá cho bộ môn

*** Quy trình thực hiện bài của SV**

- Bước 1: SV chuẩn bị tài liệu, vở, bút, giấy nháp, tài liệu tra cứu
- Bước 2: Đọc tài liệu Lập trình căn bản C phần cấu trúc vòng lặp For
- Bước 3: Sinh viên trình bày bài thực hành, thảo luận và nhận xét góp ý.

4.3.5. Các mẫu hình sản phẩm cho SV tham khảo:

- SV xem video

4.3.6. Phần tự thực hành thao tác thường xuyên của SV trong bài học:

Câu 1: Viết chương trình in ra dãy số 100, 95, 90, 85, ….., 5.

Câu 2: Nhập vào hai số *num1* và *num2*. Tìm tổng của tất cả các số lẻ nằm giữa hai số đã được nhập.

Câu 3: Viết chương trình in ra chuỗi Fibonaci (1, 1, 2, 3, 5, 8, 13,…)

Câu 4: Viết chương trình để hiển thị theo mẫu dưới đây:

(a)	(b)
1	12345
12	1234
123	123
1234	12
12345	1

4.3.7. Sản phẩm thực hành:

+ Các ví dụ demo.

4.3.8. Điều kiện để GV- SV thực hiện bài học thực hành;

4.3.8.1. Điều kiện chuẩn bị trước bài học

+ Giảng viên:

- Đề cương chi tiết bài giảng
- Thiết kế bài giảng
- Học liệu tham khảo, mở rộng

+ Sinh viên:

- Đọc kỹ bài học trước khi giảng viên lên lớp
- Chuẩn bị vấn đề thảo luận, trao đổi với giảng viên
- Các tài liệu tham khảo, mở rộng do giảng viên yêu cầu

4.3.8.2. Kỹ thuật và phương tiện dạy học:

- Bảng, phấn viết
- Màn hình lớn
- Máy tính, máy chiếu

4.3.8.3. Phương pháp tổ chức dạy-học:

- Thuyết trình, diễn giảng, thực hành
- Phát vấn
- Trao đổi, thảo luận
- Cemina
- Vấn đáp

4.4 Nội dung bài giảng 3

4.4.1 Tên bài giảng: BÀI 3: HÀM

Số tiết lên lớp của GV: 04 tiết; Số tiết tự làm bài của SV : 04 tiết

4.4.2. Phần mở đầu tiếp cận bài;

Một hàm là một đoạn chương trình thực hiện một tác vụ được định nghĩa cụ thể. Chúng thực chất là những đoạn chương trình nhỏ giúp giải quyết một vấn đề lớn.

4.4.3. Phần kiến thức, kỹ thuật căn bản:

4.4.3.1. Phần kiến thức căn bản

- **Phần kiến thức**

Mục tiêu:

Kết thúc bài học này, bạn có thể:

- Tìm hiểu về cách sử dụng các hàm
- Tìm hiểu về cấu trúc của một hàm
- Khai báo hàm và các nguyên mẫu hàm
- Thảo luận các kiểu khác nhau của biến
- Tìm hiểu cách gọi các hàm:

• Gọi bằng giá trị

- Gọi bằng tham chiếu

Tìm hiểu về các qui tắc về phạm vi của hàm

- Tìm hiểu các hàm trong các chương trình có nhiều tập tin
- Tìm hiểu về các lớp lưu trữ
- Tìm hiểu về con trỏ hàm.

o **Sử dụng các hàm**

Nói chung, các hàm được sử dụng trong C để thực thi một chuỗi các lệnh liên tiếp. Tuy nhiên, cách sử dụng các hàm thì không giống với các vòng lặp. Các vòng lặp có thể lặp lại một chuỗi các chỉ thị với các lần lặp liên tiếp nhau. Nhưng việc gọi một hàm sẽ sinh ra một chuỗi các chỉ thị được thực thi tại vị trí bất kỳ trong chương trình. Các hàm có thể được gọi nhiều lần khi có yêu cầu. Giả sử một phần của mã lệnh trong một chương trình dùng để tính tỉ lệ phần trăm cho một vài con số. Nếu sau đó, trong cùng chương trình, việc tính toán như vậy cần phải thực hiện trên những con số khác, thay vì phải viết lại các chỉ thị giống như trên, một hàm có thể được viết ra để tính tỉ lệ phần trăm của bất kỳ các con số. Sau đó chương trình có thể nhảy đến hàm đó, để thực hiện việc tính toán (trong hàm) và trở về nơi nó đã được gọi. Điều này sẽ được giải thích rõ ràng hơn khi thảo luận về cách hoạt động của các hàm.

Một điểm quan trọng khác là các hàm thì dễ viết và dễ hiểu. Các hàm đơn giản có thể được viết để thực hiện các tác vụ xác định. Việc gỡ rối chương trình cũng dễ dàng hơn khi cấu trúc chương trình dễ đọc, nhờ vào sự đơn giản hóa hình thức của nó.

Mỗi hàm có thể được kiểm tra một cách độc lập với các dữ liệu đầu vào, với dữ liệu hợp lệ cũng như không hợp lệ. Các chương trình chứa các hàm cũng dễ bảo trì hơn, bởi vì những sửa đổi, nếu yêu cầu, có thể được giới hạn trong các hàm của chương trình. Một hàm không chỉ được gọi từ các vị trí bên trong chương trình, mà các hàm còn có thể đặt vào một thư viện và được sử dụng bởi nhiều chương trình khác, vì vậy tiết kiệm được thời gian viết chương trình.

- **Cấu trúc hàm**

Cú pháp tổng quát của một hàm trong C là:

```
type_specifier function_name (arguments)
{
    body of the function
    return statement
}
```

type_specifier xác định kiểu dữ liệu của giá trị sẽ được trả về bởi hàm. Nếu không có kiểu được đưa ra, hàm cho rằng trả về một kết quả số nguyên. Các đối số được phân cách bởi dấu phẩy. Một cặp dấu ngoặc rỗng () vẫn phải xuất hiện sau tên hàm ngay cả khi nếu hàm không chứa bất kỳ đối số nào. Các tham số xuất hiện trong cặp dấu ngoặc () được gọi là **tham số hình thức** hoặc **đối số hình thức.** Phần thân của hàm có thể chứa một hoặc nhiều câu lệnh. Một hàm nên trả về một giá trị và vì vậy ít nhất một lệnh return phải có trong hàm.

- **Các đối số của một hàm**

Trước khi thảo luận chi tiết về các đối số, xem ví dụ sau,

```
#include <stdio.h>
main()
{
    int i;
    for(i =1; i <=10; i++)
    printf("\nSquare of %d is %d ", i,squarer (i));
}
    squarer(int x)
/* int x; */
{
    int j;
    j = x * x;
    return(j);
}
```

Chương trình trên tính tính bình phương các số từ 1 đến 10. Điều này được thực hiện bằng việc gọi hàm **squarer**. Dữ liệu được truyền từ thủ tục gọi (trong trường hợp

trên là hàm main()) đến hàm được gọi **squarer** thông qua các đối số. Trong thủ tục gọi, các đối số được biết như là **các đối số thực** và trong định nghĩa của hàm được gọi (squarer()) các đối số được gọi là các **đối số hình thức**. Kiểu dữ liệu của các đối số thực phải cùng kiểu với các đối số hình thức. Hơn nữa, số lượng và thứ tự của các tham số thực phải giống như của các tham số hình thức.

Khi một hàm được gọi, quyền điều khiển sẽ được chuyển đến cho nó, ở đó các đối số hình thức được thay thế bởi các đối số thực. Sau đó hàm được thực thi và khi bắt gặp câu lệnh return, nó sẽ chuyển quyền điều khiển cho chương trình gọi nó.

Hàm squarer() được gọi bằng cách truyền số cần được tính bình phương. Đối số **x** có thể được khai báo theo một trong các cách sau khi định nghĩa hàm.

Phương pháp 1

```
squarer(int x)
/* x được định nghĩa cùng với kiểu dữ liệu trong cặp dấu ngoặc ()*/
```

Phương pháp 2

```
squarer(x)
int x;
/*x được đặt trong cặp dấu ngoặc (), và kiểu của nó được khai báo ngay sau tên hàm */
```

Chú ý, trong trường hợp sau, **x** phải được định nghĩa ngay sau tên hàm, trước khối lệnh. Điều này thật tiện lợi khi có nhiều tham số có cùng kiểu dữ liệu được truyền. Trong trường hợp như vậy, chỉ phải chỉ rõ kiểu đề một lần duy nhất tại điểm bắt đầu.

Khi các đối số được khai báo trong cặp dấu ngoặc (), mỗi đối số phải được định nghĩa riêng lẻ, cho dù chúng có cùng kiểu dữ liệu. Ví dụ, nếu x và y là hai đối số của một hàm abc(), thì `abc(char x, char y)` là một khai báo đúng và `abc(char x, y)` là sai.

- **Sự trả về từ hàm**

Lệnh **return** có hai mục đích:

- Ngay lập tức trả điều khiển từ hàm về chương trình gọi
- Bất kỳ cái gì bên trong cặp dấu ngoặc () theo sau **return** được trả về như là một giá trị cho chương trình gọi.

Trong hàm **squarer(),** một biến **j** kiểu int được định nghĩa để lưu giá trị bình phương của đối số truyền vào. Giá trị của biến này được trả về cho hàm gọi thông qua lệnh **return**. Một hàm có thể thực hiện một tác vụ xác định và trả quyền điều khiển về cho thủ tục gọi nó mà không cần trả về bất kỳ giá trị nào. Trong trường hợp như vậy, lệnh

return có thể được viết dạng **return(0)** hoặc **return**. Chú ý rằng, nếu một hàm cung cấp một giá trị trả về và nó không làm điều đó thì nó sẽ trả về giá trị không thích hợp.

Trong chương trình tính bình phương của các số, chương trình truyền dữ liệu tới hàm **squarer** thông qua các đối số. Có thể có các hàm được gọi mà không cần bất kỳ đối số nào. Ở đây, hàm thực hiện một chuỗi các lệnh và trả về giá trị, nếu được yêu cầu

Chú ý rằng, hàm **squarer()** cũng có thể được viết như sau

```
squarer(int x)
{
   return(x*x);
}
```

Ở đây một biểu thức hợp lệ được xem như một đối số trong câu lệnh return. Trong thực tế, lệnh return có thể được sử dụng theo một trong các cách sau đây:

```
return;
return(hằng);
return(biến);
return(biểu thức);
return(câu lệnh đánh giá);
```

ví dụ: `return(a>b?a:b);`

Tuy nhiên, giới hạn của lệnh **return** là nó chỉ có thể trả về một giá trị duy nhất.

- **Kiểu của một hàm**

type-specifier được sử dụng để xác định kiểu dữ liệu trả về của một hàm. Trong ví dụ trên, **type-specifier** không được viết bên cạnh hàm **squarer**(), vì **squarer**() trả về một giá trị kiểu **int**. type-specifier là không bắt buộc nếu một giá trị kiểu số nguyên được trả về hoặc nếu không có giá trị nào được trả về. . Tuy nhiên, tốt hơn nên chỉ ra kiểu dữ liệu trả về là int nếu một giá trị số nguyên được trả về và tương tự dùng void nếu hàm không trả về giá trị nào.

- **Gọi hàm**

Có thể gọi một hàm từ chương trình chính bằng cách sử dụng tên của hàm, theo sau là cặp dấu ngoặc (). Cặp dấu ngoặc là cần thiết để nói với trình biên dịch là đây là một lời gọi hàm. Khi một tên hàm được sử dụng trong chương trình gọi, tên hàm có thể là một phần của một một lệnh hoặc chính nó là một câu lệnh. Mà ta đã biết một câu lệnh luôn kết thúc với một dấu chấm phẩy (**;**). Tuy nhiên, khi định nghĩa hàm, không được dùng dấu chấm phẩy ở cuối phần định nghĩa. Sự vắng mặt của dấu chấm phẩy nói với trình biên dịch đây là phần định nghĩa của hàm và không được gọi hàm.

Một số điểm cần nhớ:

➢ Một dấu chấm phẩy được dùng ở cuối câu lệnh khi một hàm được gọi, nhưng nó không được dùng sau một sự định nghĩa hàm.

➢ Cặp dấu ngoặc () là bắt buộc theo sau tên hàm, cho dù hàm có đối số hay không.

➢ Hàm gọi đến một hàm khác được gọi là **hàm gọi** hay **thủ tục gọi**. Và hàm được gọi đến còn được gọi là **hàm được gọi** hay **thủ tục được gọi**.

➢ Các hàm không trả về một giá trị số nguyên cần phải xác định kiểu của giá trị được trả về.

➢ Chỉ một giá trị có thể được trả về bởi một hàm.

➢ Một chương trình có thể có một hoặc nhiều hàm.

o **Khai báo hàm**

Một hàm nên được khai báo trong hàm main() trước khi nó được định nghĩa hoặc sử dụng. Điều này phải được thực hiện trong trường hợp hàm được gọi trước khi nó được định nghĩa.

Xem ví dụ,

```
#include <stdio.h>
main()
{
    .
    .
    address();
    .
    .
}
address()
{
    .
    .
    .
}
```

Hàm **main**() gọi hàm **address**() và hàm **address**() được gọi trước khi nó được định nghĩa. Mặc dù, nó không được khai báo trong hàm **main**() thì điều này có thể thực hiện được trong một số trình biên dịch C, hàm **address**() được gọi mà không cần khai báo gì thêm cả. Đây là sự **khai báo không tường minh** của một hàm.

Các nguyên mẫu hàm

Một nguyên mẫu hàm là một khai báo hàm trong đó xác định rõ kiểu dữ liệu của các đối số và trị trả về. Thông thường, các hàm được khai báo bằng cách xác định kiểu của giá trị được trả về bởi hàm, và tên hàm. Tuy nhiên, chuẩn ANSI C cho phép

số lượng và kiểu dữ liệu của các đối số hàm được khai báo. Một hàm abc() có hai đối số kiểu int là x và y, và trả về một giá trị kiểu char, có thể được khai báo như sau:

```
char abc();
```

hoặc

```
char abc(int x, nt y);
```

Cách định nghĩa sau được gọi là **nguyên mẫu hàm**. Khi các nguyên mẫu được sử dụng, C có thể tìm và thông báo bất kỳ kiểu dữ liệu không hợp lệ khi chuyển đổi giữa các đối số được dùng để gọi một hàm với sự định nghĩa kiểu của các tham số. Một lỗi sẽ được thông báo ngay khi có sự khác nhau giữa số lượng các đối số được sử dụng để gọi hàm và số lượng các tham số khi định nghĩa hàm.

Cú pháp tổng quát của một nguyên mẫu hàm:

```
                    type                    function_name(type
parm_name1,type parm_name2,..type
                                             parm_nameN);
```

Khi hàm được khai báo không có các thông tin nguyên mẫu, trình biên dịch cho rằng không có thông tin về các tham số được đưa ra. Một hàm không có đối số có thể gây ra lỗi khi khai báo không có thông tin nguyên mẫu. Để tránh điều này, khi một hàm không có tham số, nguyên mẫu của nó sử dụng **void** trong cặp dấu ngoặc (). Như đã nói ở trên, **void** cũng được sử dụng để khai báo tường minh một hàm không có giá trị trả về.

Ví dụ, nếu một hàm noparam() trả về kiểu dữ liệu char và không có các tham số được gọi, có thể được khai báo như sau

```
char noparam(void);
```

Khai báo trên chỉ ra rằng hàm không có tham số, và bất kỳ lời gọi có truyền tham số đến hàm đó là không đúng.

Khi một hàm không nguyên mẫu được gọi tất cả các kiểu **char** được đổi thành kiểu **int** và tất cả kiểu **float** được đổi thành kiểu **double**. Tuy nhiên, nếu một hàm là nguyên mẫu, thì các kiểu đã đưa ra trong nguyên mẫu được giữ nguyên và không có sự tăng cấp kiểu xảy ra.

- **Các biến**

Như đã thảo luận, các biến là những vị trí được đặt tên trong bộ nhớ, được sử dụng để chứa giá trị có thể hoặc không thể được sửa đổi bởi một chương trình hoặc một hàm. Có ba loại biến cơ bản: **biến cục bộ, tham số hình thức,** và **biến toàn cục**.

1. **Biến cục bộ** là những biến được khai báo bên trong một hàm.
2. **Tham số hình thức** được khai báo trong một định nghĩa hàm như là các tham số.

3. **Biến toàn cục** được khai báo bên ngoài các hàm.

- **Biến cục bộ**

Biến cục bộ còn được gọi là **biến động**, từ khoá **auto** được sử dụng để khai báo chúng. Chúng chỉ được tham chiếu đến bởi các lệnh bên trong của khối lệnh mà biến được khai báo. Để rõ hơn, một biến cục bộ được tạo ra trong lúc vào một khối và bị huỷ trong lúc đi ra khỏi khối đó. Khối lệnh thông thường nhất mà trong đó một biến cục bộ được khai báo chính là hàm.

Xem đoạn mã lệnh sau:

```
   void  blkl(void)     /*  void  denotes  no  value
returned*/
       {
          char ch;
          ch = 'a';
          .
          .
       }
       void blk2(void)
       {
          char ch;
          ch = 'b';
          .
          .
       }
```

Biến **ch** được khai báo hai lần, trong **blk1()** và **blk2()**. **ch** trong **blk1()** không có liên quan đến **ch** trong **blk2()** bởi vì mỗi **ch** chỉ được biết đến trong khối lệnh mà nó được khai báo.

Vì các biến cục bộ được tạo ra và huỷ đi trong một khối mà chúng được khai báo, nên nội dung của chúng bị mất bên ngoài phạm vi của khối. Điều này có nghĩa là chúng không thể duy trì giá trị của chúng giữa các lần gọi hàm.

Từ khóa **auto** có thể được dùng để khai báo các biến cục bộ, nhưng thường nó không được dùng vì mặc nhiên các biến không toàn cục được xem như là biến cục bộ.

Các biến cục bộ được sử dụng bởi các hàm thường được khai báo ngay sau dấu ngoặc mở '{' của hàm và trước tất cả các câu lệnh. Tuy nhiên, các khai báo có thể ở bên trong một khối của một hàm. Ví dụ,

```
       void blk1(void)
       {
```

```
    int t;
    t = 1;
    if(t > 5)
    {
        char ch;
        .
        .
    }
    .
}
```

Trong ví dụ trên biến **ch** được tạo ra và chỉ hợp lệ bên trong khối mã lệnh **if'**. Nó không thể được tham chiếu đến trong một phần khác của hàm **blk1()**.

Một trong những thuận lợi của sự khai báo một biến theo cách này đó là bộ nhớ sẽ chỉ được cấp phát cho nó khi nếu điều kiện để đi vào khối lệnh **if** được thoả. Điều này là bởi vì các biến cục bộ chỉ được khai báo khi đi vào khối lệnh mà các biến được định nghĩa trong đó.

Chú ý: Điều quan trọng cần nhớ là tất cả các biến cục bộ phải được khai báo tại điểm bắt đầu của khối mà trong đó chúng được định nghĩa, và trước tất cả các câu lệnh thực thi.

Ví dụ sau có thể không làm việc với một số các trình biên dịch.

```
void blk1(void)
{
    int len;
    len = 1;
    char ch; /* This will cause an error */
    ch = 'a';
    .
    .
}
```

- **Tham số hình thức**

Một hàm sử dụng các đối số phải khai báo các biến để nhận các giá trị của các đối số. Các biến này được gọi là **tham số hình thức** của hàm và hoạt động giống như bất kỳ một biến cục bộ bên trong hàm.

Các biến này được khai báo bên trong cặp dấu ngoặc () theo sau tên hàm. Xem ví dụ sau:

```
blk1(char ch, int i)
{
    if(i > 5)
        ch = 'a';
    else
        i = i +1;
    return;
}
```

Hàm **blk1()** có hai tham số: **ch** và **i**.

Các tham số hình thức phải được khai báo cùng với kiểu của chúng. Như trong ví dụ trên, **ch** có kiểu **char** và **i** có kiểu **int**. Các biến này có thể được sử dụng bên trong hàm như các biến cục bộ bình thường. Chúng bị huỷ đi khi ra khỏi hàm. Cần chú ý là các tham số hình thức đã khai báo có cùng kiểu dữ liệu với các đối số được sử dụng khi gọi hàm. Trong trường hợp có sai, C có thể không hiển thị lỗi nhưng có thể đưa ra một kết quả không mong muốn. Điều này là vì, C vẫn đưa ra một vài kết quả trong các tình huống khác thường. Người lập trình phải đảm bảo rằng không có các lỗi về sai kiểu.

Cũng giống như với các biến cục bộ, các phép gán cũng có thể được thực hiên với tham số hình thức của hàm và chúng cũng có thể được sử dụng bất kỳ biểu thức nào mà C cho phép.

- **Biến toàn cục**

Các biến toàn cục là biến được thấy bởi toàn bộ chương trình, và có thể được sử dụng bởi một mã lệnh bất kỳ. Chúng được khai báo bên ngoài các hàm của chương trình và lưu giá trị của chúng trong suốt sự thực thi của chương trình. Các biến này có thể được khai báo bên ngoài **main**() hoặc khai báo bất kỳ nơi đâu trước lần sử dụng đầu tiên. Tuy nhiên, tốt nhất để khai báo các biến toàn cục là tại đầu chương trình, nghĩa là trước hàm **main**().

```
int ctr;          /* ctr is global */
void blk1(void);
void blk2(void);
void main(void)
{
    ctr = 10;
    blk1 ();
    .
    .
}
void blk1(void)
```

```
{
    int rtc;
    if (ctr > 8)
    {
        rtc = rtc + 1;
        blk2();
    }
}
void blk2(void)
{
    int ctr;
    ctr = 0;
}
```

Trong đoạn mã lệnh trên, **ctr** là một biến toàn cục và được khai báo bên ngoài hàm **main**() và **blk1**(), nó có thể được tham chiếu đến trong các hàm. Biến **ctr** trong **blk2**(), là một biến cục bộ và không có liên quan với biến toàn cục **ctr**. Nếu một biến toàn cục và cục bộ có cùng tên, tất cả các tham chiếu đến tên đó bên trong khối chứa định nghĩa biến cục bộ sẽ được kết hợp với biến cục bộ mà không phải là biến toàn cục.

Các biến toàn cục được lưu trữ trong các vùng cố định của bộ nhớ. Các biến toàn cục hữu dụng khi nhiều hàm trong chương trình sử dụng cùng dữ liệu. Tuy nhiên, nên tránh sử dụng biến toàn cục nếu không cần thiết, vì chúng giữ bộ nhớ trong suốt thời gian thực hiện chương trình. Vì vậy việc sử dụng một biến toàn cục ở nơi mà một biến cục bộ có khả năng đáp ứng cho hàm sử dụng là không hiệu quả. Ví dụ sau sẽ giúp làm rõ hơn điều này:

```
void addgen(int i, int j)
{
    return(i + j);
}
int i, j;
void addspe(void)
{
    return(i + j);
}
```

Cả hai hàm **addgen()** và **addspe()** đều trả về tổng của các biến **i** và **j**. Tuy nhiên, hàm **addgen()** được sử dụng để trả về tổng của hai số bất kỳ; trong khi hàm **addspe()** chỉ trả về tổng của các biến toàn cục **i** và **j**.

- **Lớp lưu trữ (Storage Class)**

Mỗi biến trong C có một đặc trưng được gọi là **lớp lưu trữ.** Lớp lưu trữ xác định hai khía cạnh của biến: **thời gian sống** của biến và **phạm vi** của biến. **Thời gian sống** của một biến là thời gian mà giá trị của biến tồn tại. **Sự thấy được** của một biến xác định các phần của một chương trình sẽ có thể nhận ra biến. Một biến có thể xuất hiện trong một khối, một hàm, một tập tin, một nhóm các tập tin, hoặc toàn bộ chương trình

Theo cách nhìn của trình biên dịch C, một tên biến xác định một vài vị trí vật lý bên trong máy tính, ở đó một chuỗi các bit biểu diễn giá trị được lưu trữ của biến. Có hai loại vị trí trong máy tính mà ở đó giá trị của biến có thể được lưu trữ: bộ nhớ hoặc thanh ghi CPU. Lớp lưu trữ của biến xác định vị trí biến được lưu trữ là trong bộ nhớ hay trong một thanh ghi. C có bốn lớp lưu trữ. Đó là:

- **auto**
- **extern**al
- **static**
- **register**

Đó là các từ khoá. Cú pháp tổng quát cho khai báo biến như sau:

```
storage_specifier type var_name;
```

- **Biến tự động**

Biến tự động thật ra là biến cục bộ mà chúng ta đã nói ở trên. Phạm vi của một biến tự động có thể nhỏ hơn hàm, nếu nó được khai báo bên trong một câu lệnh ghép: phạm vi của nó bị giới hạn trong câu lệnh ghép đó. Chúng có thể được khai báo bằng từ khóa **auto**, nhưng sự khai báo này là không cần thiết. Bất kỳ một biến được khai báo bên trong một hàm hoặc một khối lệnh thì mặc nhiên là thuộc lớp auto và hệ thống cung cấp vùng bộ nhớ được yêu cầu cho biến đó.

- **Biến ngoại**

Trong C, một chương trình lớn có thể được chia thành các module nhỏ hơn, các module này có thể được biên dịch riêng lẻ và được liên kết lại với nhau. Điều này được thực hiện nhằm tăng tốc độ quá trình biên dịch các chương trình lớn. Tuy nhiên, khi các module được liên kết, các tập tin phải được chương trình thông báo cho biết về các biến toàn cục được yêu cầu. Một biến toàn cục chỉ có thể được khai báo một lần. Nếu hai biến toàn cục có cùng tên được khai báo trong cùng một tập tin, một thông điệp lỗi '**duplicate variable name**' (tên biến trùng) có thể được hiển thị hoặc đơn giản trình biên dịch C chọn một biến khác. Một lỗi tương tự xảy ra nếu tất cả các biến toàn cục được yêu cầu bởi chương trình chứa trong mỗi tập tin. Mặc dù trình biên dịch không đưa ra bất kỳ một thông báo lỗi nào trong khi biên dịch, nhưng sự thật các bản

sao của cùng một biến đang được tạo ra. Tại thời điểm liên kết các tập tin, bộ liên kết sẽ hiển thị một thông báo lỗi như sau '**duplicate label**' (nhãn trùng nhau) vì nó không biết sử dụng biến nào. Lớp **extern** được dùng trong trường hợp này. Tất cả các biến toàn cục được khai báo trong một tập tin và các biến giống nhau được khai báo là ở ngoài trong tất cả các tập tin. Xem đoạn mã lệnh sau:

Filel **File2**

```
int i,j;                extern int i,j;
char a;                 extern char a;
main()                  xyz()
{                           {
    .                       i = j * 5
    .                       .
    .                       .
}                           }
abc()                       pqr()
{                           {
    i = 123;            j = 50;
    .                       .
    .                       .
}                           }
```

File2 có các biến toàn cục giống như **File1,** ngoại trừ một điểm là các biến này có từ khóa **extern** được thêm vào sự khai báo của chúng. Từ khóa này nói với trình biên dịch là tên và kiểu của biến toàn cục được sử dụng mà không cần phải tạo lại sự lưu trữ cho chúng. Khi hai module được liên kết, các tham chiếu đến các biến ngoại được giải quyết.

Nếu một biến không được khai báo trong một hàm, trình biên dịch sẽ kiểm tra nó có so khớp với bất kỳ biến toàn cục nào không. Nếu khớp với một biến toàn cục, thì trình biên dịch sẽ xem như một biến toàn cục đang được tham chiếu đến.

- **Biến tĩnh**

Các biến **tĩnh** là các biến cố định bên trong các hàm và các tập tin. Không giống như các biến toàn cục, chúng không được biết đến bên ngoài hàm hoặc tập tin của chúng, nhưng chúng giữ được giá trị của chúng giữa các lần gọi. Điều này có nghĩa là, nếu một hàm kết thúc và sau đó được gọi lại, các biến tĩnh đã định nghĩa trong hàm đó vẫn giữ được giá trị của chúng. Sự khai báo biến tĩnh được bắt đầu với từ khóa **static**.

Có thể định nghĩa các biến tĩnh có cùng tên như hướng dẫn với các biến ngoại. Các biến cục bộ (biến tĩnh cũng như biến động) có độ ưu tiên cao hơn các biến ngoại

và giá trị của các biến ngoại sẽ không ảnh hưởng bởi bất kỳ sự thay đổi nào các biến cục bộ. Các biến ngoại có cùng tên với các biến nội trong một hàm không thể được truy xuất trực tiếp bên trong hàm đó.

Các giá trị khởi tạo có thể được gán cho các biến trong sự khai báo các biến tĩnh, nhưng các giá trị này phải là các hằng hoặc các biểu thức. Trình biên dịch tự động gán một giá trị mặc nhiên 0 đến các biến tĩnh không được khởi tạo. Sự khởi tạo thực hiện ở đầu chương trình.

Xem hai chương trình sau. Sự khác nhau giữa biến cục bộ: tự động và tĩnh sẽ được làm rõ.

Ví dụ về biến tự động:

```
#include <stdio.h>
main()
{
    incre();
    incre();
    incre();
}

incre()
{
    char var = 65;/* var is automatic  variable*/
    printf("\nThe character stored in var is %c",
var++);
}
```

Kết quả của chương trình trên sẽ là:

```
The character stored in var is A
The character stored in var is A
The character stored in var is A
```

Ví dụ về biến tĩnh:

```
#include<stdio.h>
main()
{
    incre();
    incre():
    incre():
}
incre()
```

```
        {
            static char var = 65; /* var is static variable
*/
            printf("nThe character stored in var is %c",
var++);
        }
```

Kết quả của chương trình trên sẽ là:

```
The character stored in var is A
The character stored in var is B
The character stored in var is C
```

Cả hai chương trình gọi **incre**() ba lần. Trong chương trình thứ nhất, mỗi lần **incre**() được gọi, biến **var** với lớp lưu trữ **auto** (lớp lưu trữ mặc định) được khởi tạo lại là 65 (là mã ASCII tương ứng của ký tự A). Vì vậy khi kết thúc hàm, giá trị mới của **var** (66) bị mất đi (ASCII ứng với ký tự B).

Trong chương trình thứ hai, **var** là của lớp lưu trữ **static**. Ở đây **var** được khởi tạo là 65 chỉ một lần duy nhất khi biên dịch chương trình. Cuối lần gọi hàm đầu tiên, **var** có giá trị 66 (ASCII B) và tương tự ở lần gọi kế tiếp **var** có giá trị 67 (ASCII C). Sau lần gọi hàm cuối cùng, **var** được tăng giá trị theo sự thi hành của lệnh **printf**(). Giá trị này bị mất khi chương trình kết thúc.

- **Biến thanh ghi**

Các máy tính có các thanh ghi trong bộ số học logic - Arithmetic Logic Unit (ALU), các thanh ghi này được sử dụng để tạm thời lưu trữ dữ liệu được truy xuất thường xuyên. Kết quả tức thời của phép tính toán cũng được lưu vào các thanh ghi. Các thao tác thực hiện trên dữ liệu lưu trữ trong các thanh ghi thì nhanh hơn dữ liệu trong bộ nhớ. Trong ngôn ngữ assembly (hợp ngữ), người lập trình phải truy xuất đến các thanh ghi này và sử dụng chúng để giúp chương trình chạy nhanh hơn. Các ngôn ngữ lập trình bậc cao thường không truy xuất đến các thanh ghi của máy tính. Trong C, việc lựa chọn vị trí lưu trữ cho một giá trị tùy thuộc vào người lập trình. Nếu một giá trị đặc biệt được dùng thường xuyên (ví dụ giá trị điều khiển của một vòng lặp), lớp lưu trữ của nó có thể khai báo là **register**. Sau đó nếu trình biên dịch tìm thấy một thanh ghi còn trống, và các thanh ghi của máy tính đủ lớn để chứa biến, biến sẽ được đặt vào thanh ghi đó. Ngược lại, trình biên dịch sẽ xem các biến thanh ghi như các biến động khác, nghĩa là lưu trữ chúng trong bộ nhớ. Từ khóa **register** được dùng khi định nghĩa các biến thanh ghi.

Phạm vi và **sự khởi tạo** của các biến thanh ghi là giống như các biến **động**, ngoại trừ vị trí lưu trữ. Các biến thanh ghi là cục bộ trong một hàm. Nghĩa là, chúng

tồn tại khi hàm được gọi và giá trị bị mất đi một khi thoát khỏi hàm. Sự khởi tạo các biến này được thực hiện bởi người lập trình.

Vì số lượng các thanh ghi là có hạn, lập trình viên cần xác định các biến nào trong chương trình được sử dụng thường xuyên để khai báo chúng là các biến thanh ghi.

Sự hữu dụng của các biến thanh ghi thay đổi từ máy này đến một máy khác và từ một trình biên dịch C này đến một trình biên dịch khác. Đôi khi các biến thanh ghi không được hỗ trợ bởi tất cả – từ khóa **register** vẫn được chấp nhận nhưng được xem giống như là từ khóa **auto**. Trong các trường hợp khác, nếu biến thanh ghi được hỗ trợ và nếu lập trình viên sử dụng chúng một cách hợp lý, chương trình sẽ được thực thi nhanh hơn gấp đôi.

Các biến **thanh ghi** được khai báo như bên dưới:

```
register int x;
register char c;
```

Sự khai báo thanh ghi chỉ có thể gắn vào các biến động và tham số hình thức. Trong trường hợp sau, sự khai báo sẽ giống như sau:

```
f(c,n)
register int c, n;
{
    register int i;
    .
    .
    .
}
```

Xét một ví dụ, ở đó chương trình hiển thị tổng lập phương các số thành phần của một số bằng chính số đó. Ví dụ 370 là một số như vậy, vì:

$3^3 + 7^3 + 0^3 = 27 + 343 + 0 = 370$

Chương trình sau in ra các con số như vậy trong khoảng 1 đến 999.

```
#include <stdio.h>
main()
{
    register int i;
    int no, digit, sum;
    printf("\nThe numbers whose Sum of Cubes of
Digits is Equal to the number itself are:\n\n");
    for(i = 1; i < 999; i++)
    {
        sum = 0;
```

```
            no = i;
            while(no)
             {
                 digit = no%10;
                 no = no/10;
                 sum = sum + digit * digit * digit;
            }

            if (sum == i)
                 printf("t%d\n", i);
        }
    }
```

Kết quả của chương trình trên như sau:

```
    The numbers whose Sum of Cubes of Digits is Equal
to the
    number itself are:
        1
        153
        370
        371
        407
```

Trong chương trình trên, giá trị của **i** , thay đổi từ 1 đến 999. Với mỗi giá trị này, lập phương của từng con số riêng lẻ được cộng và kết quả tổng được so sánh với **i**. Nếu hai giá trị này là bằng nhau, **i** được hiển thị. Vì **i** được sử dụng để điều khiển sự lặp, (phần chính của chương trình), nó được khai báo là của lớp lưu trữ thanh ghi. Sự khai báo này làm tăng hiệu quả của chương trình.

- **Các qui luật về phạm vi của một hàm**

Qui luật về phạm vi là những qui luật quyết định một đoạn mã lệnh có thể truy xuất đến một đoạn mã lệnh khác hoặc dữ liệu hay không. Trong C, mỗi hàm của chương trình là các khối lệnh riêng lẻ. Mã lệnh bên trong một hàm là cục bộ với hàm đó và không thể được truy xuất bởi bất kỳ lệnh nào ở ngoài hàm, ngoại trừ lời gọi hàm. Mã lệnh bên trong một hàm là ẩn đối với phần còn lại của chương trình, và trừ khi nó sử dụng biến hoặc dữ liệu toàn cục, nó có thể tác động hoặc bị tác động bởi các phần khác của chương trình. Để rõ hơn, mã lệnh và dữ liệu được định nghĩa bên trong một hàm không thể tương tác với mã lệnh hay dữ liệu được định nghĩa trong hàm khác bởi vì hai hàm có phạm vi khác nhau.

Trong C, tất cả các hàm có cùng mức phạm vi. Nghĩa là, một hàm không thể được định nghĩa bên trong một hàm khác. Chính vì lý do này mà C không phải là một ngôn ngữ cấu trúc khối về mặt kỹ thuật.

- **Gọi hàm**

Một cách tổng quát, các hàm giao tiếp với nhau bằng cách truyền tham số. Các tham số được truyền theo một trong hai cách sau:

- Truyền bằng giá trị
- Truyền bằng tham chiếu.

- **Truyền bằng giá trị**

Mặc nhiên trong C, tất cả các đối số của hàm được truyền bằng giá trị. Điều này có nghĩa là, khi các đối số được truyền đến hàm được gọi, các giá trị được truyền thông qua các biến tạm. Mọi sự thao tác chỉ được thực hiện trên các biến tạm này. Hàm được gọi không thể thay đổi giá trị của chúng. Xem ví dụ sau,

```
#include <stdio.h>
main()
{
    int a, b, c;
    a = b = c = 0;
    printf("\nEnter 1st integer: ");
    scanf("%d", &a);
    printf("\nEnter 2nd integer: ");
    scanf("%d", &b);
    c = adder(a, b);
    printf("\n\na & b in main() are: %d, % d", a,
b);
    printf("\n\nc in main() is: %d", c);
    /* c gives the addition of a and b */
}
adder(int a, int b)
{
    int c;
    c = a + b;
    a *= a;
    b += 5;
    printf("\n\na & b within adder function
are: %d, %d ", a, b);
    printf("\nc within adder function is : %d",c);
    return(c);
```

```
}
```

Ví dụ về kết quả thực thi khi nhập vào 2 và 4:

```
a & b in main() are: 2, 4
c in main() is: 6
a & b within adder function are: 4, 9
c within adder function is : 6
```

Chương trình trên nhận hai số nguyên, hai số này được truyền đến hàm **adder**(). Hàm **adder**() thực hiện như sau: nó nhận hai số nguyên như là các đối số của nó, cộng chúng lại, tính bình phương cho số nguyên thứ nhất, và cộng 5 vào số nguyên thứ hai, in kết quả và trả về tổng của các đối số thực. Các biến được sử dụng trong hàm **main**() và **adder**() có cùng tên. Tuy nhiên, không có gì là chung giữa chúng. Chúng được lưu trữ trong các vị trí bộ nhớ khác nhau. Điều này được thấy rõ từ kết quả của chương trình trên. Các biến a và b trong hàm **adder**() được thay đổi từ 2 và 4 thành 4 và 9. Tuy nhiên, sự thay đổi này không ảnh hưởng đến các giá trị của a và b trong hàm **main**(). Các biến được lưu ở những vị trí bộ nhớ khác nhau. Biến **c** trong **main**() thì khác với biến **c** trong **adder**().

Vì vậy, các đối số được gọi là **truyền bằng giá trị** khi giá trị của các biến được truyền đến hàm được gọi và bất kỳ sự thay đổi trên giá trị này cũng không ảnh hưởng đến giá trị gốc của biến đã truyền.

- **Truyền bằng tham chiếu**

Khi các đối số được truyền bằng giá trị, các giá trị của đối số của hàm đang gọi không bị thay đổi. Tuy nhiên, có thể có trường hợp, ở đó giá trị của các đối số phải được thay đổi. Trong những trường hợp như vậy, **truyền bằng tham chiếu** được dùng. **Truyền bằng tham chiếu**, hàm được phép truy xuất đến vùng bộ nhớ thực của các đối số và vì vậy có thể thay đổi giá trị của các đối số của hàm gọi.

Ví dụ, xét một hàm, hàm này nhận hai đối số, hoán vị giá trị của chúng và trả về các giá trị của chúng. Nếu một chương trình giống như chương trình dưới đây được viết để giải quyết mục đích này, thì sẽ không bao giờ thực hiện được.

```
#include <stdio.h>
main()
{
    int x, y;
    x = 15; y = 20;
    printf("x = %d, y = %d\n", x, y);
    swap(x, y);
    printf("\nAfter interchanging x = %d, y =
%d\n", x, y);
```

```
}

swap(int u, int v)
{
    int temp;
    temp = u;
    u = v;
    v = temp;
    return;
}
```

Kết quả của chương trình trên như sau:

```
x = 15, y = 20
After interchanging x = 15, y = 20
```

Hàm **swap**() hoán vị các giá trị của u và v, nhưng các giá trị này không được truyền trở về hàm **main**(). Điều này là bởi vì các biến **u** và **v** trong **swap()** là khác với các biến **u** và **v** được dùng trong **main**(). Truyền bằng tham chiếu có thể được sử dụng trong trường hợp này để đạt được kết quả mong muốn, bởi vì nó sẽ thay đổi các giá trị của các đối số thực. Các con trỏ được dùng khi thực hiện truyền bằng tham chiếu.

Các con trỏ được truyền đến một hàm như là các đối số để cho phép hàm được gọi của chương trình truy xuất các biến mà phạm vi của nó không vượt ra khỏi hàm gọi. Khi một con trỏ được truyền đến một hàm, địa chỉ của dữ liệu được truyền đến hàm nên hàm có thể tự do truy xuất nội dung của địa chỉ đó. Các hàm gọi nhận ra bất kỳ thay đổi trong nội dung của địa chỉ. Theo cách này, đối số hàm cho phép dữ liệu được thay đổi trong hàm gọi, cho phép truyền dữ liệu hai chiều giữa hàm gọi và hàm được gọi. Khi các đối số của hàm là các con trỏ hoặc mảng, truyền bằng tham chiếu được tạo ra đối nghịch với cách truyền bằng giá trị.

Các đối số hình thức của một hàm là các con trỏ thì phải có một dấu * phía trước, giống như sự khai báo biến con trỏ, để xác định chúng là các con trỏ. Các đối số thực kiểu con trỏ trong lời gọi hàm có thể được khai báo là một biến con trỏ hoặc một biến được tham chiếu đến (&var).

Ví dụ, định nghĩa hàm

```
getstr(char *ptr_str, int *ptr_int)
```

đối số **ptr_str** trỏ đến kiểu **char** và **ptr_int** trỏ đến kiểu **int**. Hàm có thể được gọi bằng câu lệnh,

```
getstr(pstr, &var)
```

ở đó **pstr** được khai báo là một con trỏ và địa chỉ của biến **var** được truyền. Gán giá trị thông qua,

```
*ptr_int = var;
```

Hàm bây giờ có thể gán các giá trị đến biến **var** trong hàm gọi, cho phép truyền theo hai chiều đến và từ hàm.

```
char *pstr;
```

Quan sát ví dụ sau của hàm **swap**(). Bài toán này sẽ giải quyết được khi con trỏ được truyền thay vì dùng biến. Mã lệnh tương tự như sau:

```
#include <stdio.h>
void main()
{
    int x, y, *px, *py;

    /* Storing address of x in px */
    px = &x;

    /* Storing address of y in py */
    py = &y;

    x = 15; y = 20;
    printf("x = %d, y = %d \n", x, y);
    swap (px, py);

    /* Passing addresses of x and y */
    printf("\n After interchanging x = %d, y =
%d\n", x, y);
}

swap(int *u, int *v)
/* Accept the values of px and py into u and v */
{
    int temp;
    temp = *u;
    *u = *v;
    *v = temp;
    return;
}
```

Kết quả của chương trình trên như sau:

```
x = 15, y = 20
After interchanging x = 20, y = 15
```

Hai biến kiểu con trỏ **px** và **py** được khai báo, và địa chỉ của biến **x** và **y** được gán đến chúng. Sau đó các biến con trỏ được truyền đến hàm **swap()**, hàm này hoán vị các giá trị lưu trong x và y thông qua các con trỏ.

- **Sự lồng nhau của lời gọi hàm**

Lời gọi một hàm từ một hàm khác được gọi là **sự lồng nhau** của **lời gọi hàm**. Một chương trình kiểm tra một chuỗi có phải là chuỗi đọc xuôi - đọc ngược như nhau hay không, là một ví dụ cho các lời gọi hàm lồng nhau. Từ đọc xuôi - ngược giống nhau là một chuỗi các ký tự đối xứng. Xem đoạn mã lệnh theo sau đây:

```
main()
{
    .
    .
    palindrome();
    .
    .
}
palindrome()
{
    .
    .
    getstr();
    reverse();
    cmp();
    .
    .
}
```

Trong chương trình trên, hàm **main()** gọi hàm **palindrome()**. Hàm **palindrome()** gọi đến ba hàm khác **getstr()**, **reverse()** và **cmp()**. Hàm **getstr()** để nhận một chuỗi ký tự từ người dùng, hàm **reverse()** đảo ngược chuỗi và hàm **cmp()** so sánh chuỗi được nhập vào và chuỗi đã được đảo.

Vì **main**() gọi **palindrome**(), hàm **palindrome**() lần lượt gọi các hàm **getstr**(), **reverse**() và **cmp**(), các lời gọi hàm này được gọi là được lồng bên trong **palindrome**().

Sự lồng nhau của các lời gọi hàm như trên là được phép, trong khi định nghĩa một hàm bên trong một hàm khác là không được chấp nhận trong C.

- **Hàm trong chương trình nhiều tập tin**

Các chương trình có thể được tạo bởi nhiều tập tin. Những chương trình như vậy được tạo bởi các hàm lớn, ở đó mỗi hàm có thể chiếm một tập tin. Cũng như các biến trong các chương trình nhiều tập tin, các hàm cũng có thể được định nghĩa là **static** hoặc **external**. Phạm vi của hàm external có thể được sử dụng trong tất cả các tập tin của chương trình, và đó là cách lưu trữ mặc định cho các tập tin. Các hàm static chỉ được nhận biết bên trong tập tin chương trình và phạm vi của nó không vượt khỏi tập tin chương trình. Phần tiêu đề (header) của hàm như sau,

```
static fn _type fn_name (argument list)
```

hoặc

```
extern fn_type fn_name (argument list)
```

Từ khóa **extern** là một tuỳ chọn (không bắt buộc) vì nó là lớp lưu trữ mặc định.

- **Con trỏ đến hàm**

Một đặc tính mạnh mẽ của C vẫn chưa được đề cập, chính là **con trỏ hàm**. Dù rằng một hàm không phải là một biến, nhưng nó có địa chỉ vật lý trong bộ nhớ nơi có thể gán cho một con trỏ. Một địa chỉ hàm là điểm đi vào của hàm và con trỏ hàm có thể được sử dụng để gọi hàm.

Để hiểu các con trỏ hàm làm việc như thế nào, thật sự cần phải hiểu thật rõ một hàm được biên dịch và được gọi như thế nào trong C. Khi mỗi hàm được biên dịch, mã nguồn được chuyển thành mã đối tượng và một điểm đi vào của hàm được thiết lập. Khi một lời gọi được thực hiện đến một hàm, một lời gọi ngôn ngữ máy được thực hiện để chuyển điều khiển đến điểm đi vào của hàm. Vì vậy, nếu một con trỏ chứa địa chỉ của điểm đi vào của hàm, nó có thể được dùng để gọi hàm đó. Địa chỉ của một hàm có thể lấy được bằng cách sử dụng tên hàm không có dấu ngoặc () hay bất kỳ đối số nào. Chương trình sau sẽ minh họa khái niệm của con trỏ hàm.

```
#include <stdio.h>
#include <string.h>
void check(char *a, char *b, int (*cmp)());
main()
{
    char s1[80], s2[80];
    int (*p)();
    p = strcmp;
    gets(s1);
    gets(s2);
    check(s1, s2, p);
}
    void check(char *a, char *b, int (*cmp)())
{
printf("Testing for equality \n");
if(!(*cmp)(a, b))
    printf("Equal");
else
    printf("Not Equal");
}
```

Hàm **check()** được gọi bằng cách truyền hai con trỏ ký tự và một con trỏ hàm. Trong hàm **check()**, các đối số được khai báo là các con trỏ ký tự và một hàm con trỏ. Chú ý cách một hàm con trỏ được khai báo. Cú pháp tương tự được dùng khi khai báo các hàm con trỏ khác bất luận đó là kiểu trả về của hàm. Cặp dấu ngoặc () bao quanh ***cmp** là cần thiết để chương trình biên dịch hiểu câu lệnh này một cách rõ ràng.

4.4.4. Quy trình kỹ thuật trong bài học của GV và SV:

*** Quy trình thị phạm của GV**

- Bước 1: Chuẩn bị bài giảng, tài liệu và điều kiện trang thiết bị giảng dạy
- Bước 2: Chuẩn bị bài tập mẫu và đưa ra nội dung thực hành
- Bước 3: Phổ biến các quy trình, các bước thực hiện
- Bước 4: Hướng dẫn nội dung bài học, Hướng dẫn thị phạm mẫu
- Bước 5: Quản lý lớp học
- Bước 6: Giải đáp, tư vấn, đánh giá, nhận xét
- Bước 7: Chấm điểm và nộp các bài đánh giá cho bộ môn

*** Quy trình thực hiện bài của SV**

- Bước 1: SV chuẩn bị tài liệu, vở, bút, giấy nháp, tài liệu tra cứu
- Bước 2: Đọc tài liệu Lập trình căn bản C phần Hàm.
- Bước 3: Sinh viên trình bày bài thực hành, thảo luận và nhận xét góp ý.

4.4.5. Các mẫu hình sản phẩm cho SV tham khảo:

- SV xem video

4.4.6. Phần tự thực hành thao tác thường xuyên của SV trong bài học:

+ Hàm…

4.4.7. Sản phẩm thực hành:

+ Các ví dụ demo.

4.4.8. Điều kiện để GV- SV thực hiện bài học thực hành;

4.4.8.1. Điều kiện chuẩn bị trước bài học

+ Giảng viên:

- Đề cương chi tiết bài giảng
- Thiết kế bài giảng
- Học liệu tham khảo, mở rộng

+ Sinh viên:

- Đọc kỹ bài học trước khi giảng viên lên lớp
- Chuẩn bị vấn đề thảo luận, trao đổi với giảng viên
- Các tài liệu tham khảo, mở rộng do giảng viên yêu cầu

4.4.8.2. Kỹ thuật và phương tiện dạy học:

- Bảng, phấn viết
- Màn hình lớn

- Máy tính, máy chiếu

4.4.8.3. Phương pháp tổ chức dạy-học:

- Thuyết trình, diễn giảng, thực hành
- Phát vấn
- Trao đổi, thảo luận
- Cemina
- Vấn đáp

4.5 Nội dung bài giảng 4

4.5.1. Tên bài giảng: BÀI 4: MẢNG VÀ CHUỖI

Số tiết lên lớp của GV: 03 tiết; Số tiết tự làm bài của SV : 03 tiết

4.5.2. Phần mở đầu tiếp cận bài;

Có thể bạn sẽ gặp khó khăn khi lưu trữ một tập hợp các phần tử dữ liệu giống nhau trong các biến khác nhau. Ví dụ, điểm cho tất cả 11 cầu thủ của một đội bóng đá phải được ghi nhận trong một trận đấu. Sự lưu trữ điểm của mỗi cầu thủ trong các biến có tên khác nhau thì chắc chắn phiền hà hơn dùng một biến chung cho chúng. Với mảng mọi việc sẽ được thực hiện đơn giản hơn. Một **mảng** là một tập hợp các phần tử dữ liệu có cùng kiểu. Mỗi phần tử được lưu trữ ở các vị trí kế tiếp nhau trong bộ nhớ chính. Những phần tử này được gọi là **phần tử mảng**.

4.5.3. Phần kiến thức, kỹ thuật căn bản:

4.5.3.1 Phần kiến thức căn bản

- **Mục tiêu:**

Kết thúc bài học này, bạn có thể:

- Hiểu được các phần tử của mảng và các chỉ số mảng
- Khai báo một mảng
- Hiểu cách quản lý mảng trong C
- Hiểu một mảng được khởi tạo như thế nào
- Hiểu mảng chuỗi/ ký tự
- Hiểu mảng hai chiều
- Hiểu cách khởi tạo mảng nhiều chiều.

- **Các phần tử mảng và các chỉ mục:**

Mỗi phần tử của mảng được định danh bằng một **chỉ mục** hoặc **chỉ số** gán cho nó. **Chiều** của mảng được xác định bằng số chỉ số cần thiết để định danh duy nhất mỗi phần tử. Một chỉ số là một số nguyên dương được bao bằng dấu ngoặc vuông **[ ]** đặt ngay sau tên mảng, không có khoảng trắng ở giữa. Một **chỉ số** chứa các giá trị nguyên bắt đầu bằng 0. Vì vậy, một mảng **player** với 11 phần tử được biểu diễn như sau:

```
player[0], player[1], player[2], ... , player[10].
```

Như đã thấy, phần tử mảng bắt đầu với player[0], và vì vậy phần tử cuối cùng là player[10] không phải là player[11]. Điều này là do bởi trong C, chỉ số mảng bắt đầu từ 0; do đó trong mảng **N** phần tử, phần tử cuối cùng có chỉ số là **N-1**. Phạm vi cho phép của các giá trị chỉ số được gọi là **miền giới hạn** của chỉ số mảng, giới hạn **dưới** và giới hạn **trên**. Một chỉ số mảng hợp lệ phải có một giá trị nguyên nằm trong niềm giới hạn. Thuật ngữ **hợp lệ** được sử dụng cho một nguyên nhân rất đặc trưng. Trong C, nếu người dùng cố gắng truy xuất một phần tử nằm ngoài dãy chỉ số hợp lệ (như player[11] trong ví dụ trên của mảng), trình biên dịch C sẽ không phát sinh ra lỗi. Tuy

nhiên, có thể nó truy xuất một giá trị nào đó dẫn đến kết quả không đoán được. Cũng có nguy cơ viết chồng lên dữ liệu hoặc mã lệnh chương trình. Vì vậy, người lập trình phải đảm bảo rằng tất cả các chỉ số là nằm trong miền giới hạn hợp lệ.

- **Khai báo một mảng:**

Một mảng có một vài đặc tính riêng biệt và phải được khai báo khi sử dụng chúng. Những đặc tính này bao gồm:

- **Lớp lưu trữ**
- **Kiểu dữ liệu** của các phần tử mảng.
- **Tên mảng** – xác định vị trí phần tử đầu tiên của mảng.
- **Kích thước mảng** - một hằng số có giá trị nguyên dương.

Một mảng được khai báo giống như cách khai báo một biến, ngoại trừ tên mảng được theo sau bởi một hoặc nhiều biểu thức, được đặt trong dấu ngoặc vuông [] xác định chiều dài của mảng. Cú pháp tổng quát khai báo một mảng như sau:

```
lớp_lưu_trữ                                    kiểu_dữ_liệu
tên_mảng[biểu_thức_kích_thước]
```

Ở đây, **biểu_thức_kích_thước** là một biểu thức xác định số phần tử trong mảng và phải định ra một trị **nguyên dương**. Lớp_lưu_trữ là một tùy chọn. Mặc định lớp **automatic** được dùng cho mảng khai báo bên trong một hàm hoặc một khối lệnh, và lớp **external** được dùng cho mảng khai báo bên ngoài một hàm. Vì vậy mảng **player** được khai báo như sau:

```
int player[11];
```

Nên nhớ rằng, trong khi khai báo mảng, kích thước của mảng sẽ là 11, tuy nhiên các chỉ số của từng phần tử bên trong mảng sẽ là từ 0 đến 10.

Các qui tắc đặt tên mảng là giống với qui tắc đặt tên biến. Một **tên mảng** và một **tên biến** không được giống nhau, nó dẫn đến sự nhập nhằng. Nếu một sự khai báo như vậy xuất hiện trong chương trình, trình biên dịch sẽ hiển thị thông báo lỗi.

Một vài qui tắc với mảng:

- Tất cả các phần tử của một mảng có cùng kiểu. Điều này có nghĩa là, nếu một mảng được khai báo kiểu **int**, nó không thể chứa các phần tử có kiểu khác.
- Mỗi phần tử của mảng có thể được sử dụng bất cứ nơi nào mà một biến được cho phép hay được yêu cầu.
- Một phần tử của mảng có thể được tham chiếu đến bằng cách sử dụng một biến hoặc một biểu thức nguyên. Sau đây là các tham chiếu hợp lệ:

```
player[i]; /*Ở đó i là một biến, tuy nhiên cần phải
chú ý rằng i nằm trong miền giới hạn của chỉ số đã được
khai báo cho mảng player*/
```

```
player[3] = player[2] + 5;
player[0] += 2;
player[i / 2 + 1];
```

- Kiểu dữ liệu của mảng có thể là **int**, **char**, **float**, hoặc **double**.

o **Việc quản lý mảng trong C:**

Một mảng được "đối xử" khác với một biến trong C. Thậm chí hai mảng có cùng kiểu và kích thước cũng không thể tương đương nhau. Hơn nữa, không thể gán một mảng trực tiếp cho một mảng khác. Thay vì thế, mỗi phần tử mảng phải được gán riêng lẻ tương ứng với từng phần tử của mảng khác. Các giá trị không thể được gán cho toàn bộ một mảng, ngoại trừ tại thời điểm khởi tạo. Tuy nhiên, từng phần tử không chỉ có thể được gán trị mà còn có thể được so sánh.

```
int player1[11], player2[11];
for (i = 0; i < 11; i++)
     player1[i] = player2[i];
```

Tương tự, cũng có thể có kết quả như vậy bằng việc sử dụng các lệnh gán riêng lẻ như sau:

```
player1[0] = player2[0];
player1[1] = player2[1];
...
...
...
player1[10] = player2[10];
```

Cấu trúc **for** là cách lý tưởng để thao tác các mảng.

Ví dụ 1:

```
/* Program demonstrates a single dimensional array */
#include <stdio.h>
void main()
{    int num[5];
     int i;
     num[0] = 10;
     num[1] = 70;
     num[2] = 60;
     num[3] = 40;
     num[4] = 50;
     for (i = 0; i < 5; i++)
          pirntf("\n Number at [%d] is %d", i, num[i]);
}
```

Kết quả của chương trình được trình bày bên dưới:

```
Number at [0] is 10
Number at [1] is 70
Number at [2] is 60
Number at [3] is 40
Number at [4] is 50
```

Ví dụ bên dưới nhập các giá trị vào một mảng có kích thước 10 phần tử, hiển thị giá trị lớn nhất và giá trị trung bình.

Ví dụ 2:

```
/*Input values are accepted from the user into the array
ary[10]*/
#include <stdio.h>
void main()
{
     int ary[10];
     int i, total, high;
     for (i = 0;  i < 10; i++)
     {
     printf("\nEnter value: %d: ", i + 1);
          scanf("%d", &ary[i]);
     }

     /* Displays highest of the entered values */
     high = ary[0];
     for (i = 1; i < 10; i++)
     {
     if (ary[i] > high)
               high = ary[i];
     }
     printf("\n Highest value entered was %d", high);

     /* Prints average of value entered for ary[10] */
     for (i = 0, total = 0; i < 10; i++)
          total = total + ary[i];

     printf("\nThe average of the element of ary is %d",
total/i);
}
```

Một ví dụ về kết quả được trình bày dưới đây:

```
Enter value: 1: 10
Enter value: 2: 20
Enter value: 3: 30
Enter value: 4: 40
Enter value: 5: 50
Enter value: 6: 60
Enter value: 7: 70
Enter value: 8: 80
Enter value: 9: 90
Enter value: 10: 10
Highest value entered was 90
The average of the element of ary is 46
```

- **Việc khởi tạo mảng:**

Các mảng không được khởi tạo tự động, trừ khi mỗi phần tử mảng được gán một giá trị riêng lẻ. Không nên dùng các mảng trước khi có sự khởi tạo thích hợp. Điều này là bởi vì không gian lưu trữ của mảng không được khởi tạo tự động, do đó dễ gây ra kết quả không lường trước. Mỗi khi các phần tử của một mảng chưa khởi tạo được sử dụng trong các biểu thức toán học, các giá trị đã tồn tại sẵn trong ô nhớ sẽ được sử dụng, các giá trị này không đảm bảo rằng có cùng kiểu như khai báo của mảng, trừ khi các phần tử của mảng được khởi tạo một cách rõ ràng. Điều này đúng không chỉ cho các mảng mà còn cho các biến thông thường.

Trong đoạn mã lệnh sau, các phần tử của mảng được gán giá trị bằng các dùng vòng lặp **for**.

```
int ary[20], i;
for(i=0; i<20; i++)
     ary[i] = 0;
```

Khởi tạo một mảng sử dụng vòng lặp **for** có thể được thực hiện với một hằng giá trị, hoặc các giá trị được sinh ra từ một cấp số cộng.

Một vòng lặp **for** cũng có thể được sử dụng để khởi tạo một mảng các ký tự như sau:

Ví dụ 3:

```
#include <stdio.h>
void main()
{
    char alpha[26];
    int i, j;
```

```
        for(i = 65, j = 0; i < 91; i++, j++)
        {
            alpha[j] = i;
            printf("The character now assigned is %c\n",
alpha[j]);
        }
        getchar();
    }
```

Một phần kết quả của chương trình trên như sau:

```
The character now assigned is A
The character now assigned is B
The character now assigned is C
.
.
.
```

Chương trình trên gán các mã ký tự ASCII cho các phần tử của mảng **alpha**. Kết quả là khi in với định dạng **%c**, một chuỗi các ký tự được xuất ra màn hình. Các mảng cũng có thể được khởi tạo khi khai báo. Điều này được thực hiện bằng việc gán tên mảng với một danh sách các giá trị phân cách nhau bằng dấu phẩy (**,**) đặt trong cặp dấu ngoặc nhọn **{}**. Các giá trị trong cặp dấu ngoặc nhọn {} được gán cho các phần tử trong mảng theo đúng thứ tự xuất hiện.

Ví dụ:

```
int deci[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
static float rates[4] = {0.0, -2.5, 13.75, 18.0};
char company[5] = {'A', 'P', 'P', 'L', 'E'};
int marks[100] = {15, 13, 11, 9}
```

Các giá trị khởi tạo của mảng phải là các hằng, không thể là biến hoặc các biểu thức. Một vài phần tử đầu tiên của mảng sẽ được khởi tạo nếu số lượng giá trị khởi tạo là ít hơn số phần tử mảng được khai báo. Các phần tử còn lại sẽ được khởi tạo giá trị 0. Ví dụ, trong mảng **marks** sau khi có sự khởi tạo như trên, bốn phần tử đầu tiên (từ 0 đến 3) tương ứng được khởi tạo là 15, 13, 11 và 9. Các phần tử còn lại có giá trị 0. Không thể chỉ khởi tạo các phần tử từ 1 đến 4, hoặc từ 2 đến 4, hay từ 2 đến 5 khi sự

khởi tạo được thực hiện tại thời điểm khai báo. Trong C không có khả năng lặp lại sự khởi tạo giá trị.

Trong trường hợp sự khởi tạo là tường minh, lớp **extern** hoặc **static**, các phần tử của mảng được đảm bảo khởi tạo là 0 (không giống lớp **auto**).

Không cần thiết khai báo kích thước của mảng đang được khởi tạo. Nếu kích thước của mảng được bỏ qua khi khai báo, trình biên dịch sẽ xác định kích thước của mảng bằng cách đếm các giá trị đang được khởi tạo. Ví dụ, sự khai báo mảng **external** sau đây sẽ chỉ định kích thước của mảng **ary** là 5 vì có 5 giá trị khởi tạo.

```
int ary[] = {1, 2, 3, 4, 5};
```

- **Các mảng chuỗi/ký tự:**

Một **chuỗi** có thể được khai báo như là một mảng ký tự, và được kết thúc bởi một ký tự NULL. Mỗi ký tự của chuỗi chiếm 1 byte, và ký tự cuối cùng của chuỗi luôn luôn là ký tự '\0'. Ký tư '\0' được gọi là **ký tự null**. Nó là một mã thoát (escape sequence) tương tự như '\n', thay thế cho ký tự có giá trị 0. Vì '\0' luôn là ký tự cuối cùng của một chuỗi, nên các mảng ký tự phải có nhiều hơn một ký tự so với chiều dài tối đa mà chúng quản lý. Ví dụ, một mảng **ary** quản lý một chuỗi 10 ký tự phải được khai báo như sau:

```
char ary[11];
```

Vị trí thêm vào được sử dụng để lưu trữ ký tự null. Nên nhớ rằng ký tự kết thúc (ký tự null) là rất quan trọng.

Các giá trị chuỗi có thể được nhập vào bằng cách sử dụng hàm *scanf()***.** Với chuỗi **ary** được khai báo ở trên, mã lệnh nhập sẽ như sau:

```
scanf("%s", ary);
```

Trong lệnh trên, **ary** xác định vị trí nơi mà lần lượt các ký tự của mảng sẽ được lưu trữ.

Ví dụ 4:

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

```
    char ary[5];
    int i;
    printf("\n Enter string: ");
    scanf("%s", ary);
    printf("\n The string is %s \n\n", ary);

    for (i = 0; i < 5; i++)
        printf("\t%d", ary[i]);
}
```

Các kết quả thực thi chương trình với những dữ liệu nhập khác nhau như sau:

Nếu chuỗi được nhập là *appl*, kết quả sẽ là:

```
The string is appl
     97   112  112  108  0
```

Kết quả như trên là của 4 ký tự (appl) và ký tự thứ 5 là ký tự null. Điều này được thấy rõ với mã ASCII cho các ký tự được in ra ở dòng thứ hai. Ký tự thứ năm được in la 0, là giá trị của ký tự null.

Nếu chuỗi nhập vào là *apple*, kết quả sẽ là:

```
The string is apple
     97   112  112  108  101
```

Kết quả ở trên của là một dữ liệu đầu vào có 5 ký tự a, p, p, l và e. Nó không được xem là một chuỗi bởi vì ký tự thứ 5 của mảng không phải là \0. Một lần nữa, điều này được thấy rõ bằng dòng in ra mã ASCII của các ký tự a, p, p, l, e.

Nếu chuỗi được nhập vào là *ap*, thì kết quả sẽ là:

```
The string is ap
     97   112  0    6    100
```

Trong ví dụ trên, khi chỉ có hai ký tự được nhập, ký tự thứ ba sẽ là ký tự null. Điều này cho biết là chuỗi đã được kết thúc. Những ký tự còn lại là những ký tự không dự đoán được.

Trong trường hợp trên, tính quan trọng của ký tự null trở nên rõ ràng. Ký tự null xác định sự kết thúc của chuỗi và là cách duy nhất để các hàm làm việc với chuỗi sẽ biết đâu là điểm kết thúc của chuỗi.

Mặc dù C không có kiểu dữ liệu chuỗi, nhưng nó cho phép các hằng chuỗi. Một hằng chuỗi là một dãy các ký tự được đặt trong dấu nháy đôi (**“”**). Không giống như các hằng khác, nó không thể được sửa đổi trong chương trình. Ví dụ như:

```
"Hi Aptechite!"
```

Trình biên dịch C sẽ tự động thêm vào ký tự null cuối chuỗi.

C hỗ trợ nhiều hàm cho chuỗi, các hàm này nằm trong thư viện chuẩn ***string.h***. Một vài hàm được đưa ra trong bảng 11.1. Cách làm việc của các hàm này sẽ được thảo luận trong bài 17.

Tên hàm	**Chức năng**
strcpy(s1, s2)	Sao chép s2 vào s1
strcat(s1, s2)	Nối s2 vào cuối của s1
strlen(s1)	Trả về chiều dài của s1
strcmp(s1, s2)	Trả về 0 nếu s1 và s2 là giống nhau; nhỏ hơn 0 nếu s1<s2; lớn hơn 0 nếu s1> s2
strchr(s1, ch)	Trả về một con trỏ trỏ đến vị trí xuất hiện đầu tiên của ch trong s1
strstr(s1, s2)	Trả về một con trỏ trỏ đến vị trí xuất hiện đầu tiên của chuỗi s2 trong chuỗi s1

- **Mảng hai chiều:**

Chúng ta đã biết thế nào là **mảng một chiều**. Điều này có nghĩa là các mảng chỉ có một chỉ số. Các mảng có thể có nhiều hơn một chiều. Các mảng **đa chiều** giúp dễ dàng trình bày các đối tượng đa chiều, chẳng hạn một đồ thị với các dòng và cột hay tọa độ màn hình của máy tính. Các mảng đa chiều được khai báo giống như các mảng một chiều, ngoại trừ có thêm một cặp dấu ngoặc vuông [] trong trường hợp mảng hai chiều. Một mảng ba chiều sẽ cần ba cặp dấu ngoặc vuông,... Một cách tổng quát, một mảng đa chiều có thể được biểu diễn như sau:

```
storage_class data_type ary[exp1][exp2]....[expN];
```

Ở đó, **ary** là một mảng có lớp là **storage_class**, kiểu dữ liệu là **data_type**, và **exp1, exp2,..... , expN** là các biểu thức nguyên dương xác định số phần tử của mảng được kết hợp với mỗi chiều.

Dạng đơn giản nhất và thường được sử dụng nhất của các mảng đa chiều là **mảng hai chiều**. Một mảng hai chiều có thể xem như là một mảng của hai 'mảng một chiều'. Một mảng hai chiều đặc trưng như bảng lịch trình của máy bay, xe lửa. Để xác định thông tin, ta sẽ chỉ định dòng và cột cần thiết, và thông tin được đọc ra từ vị trí (dòng và cột) được tìm thấy. Tương tự như vậy, một mảng hai chiều là một khung lưới chứa các dòng và cột trong đó mỗi phần tử được xác định duy nhất bằng toạ độ dòng và cột của nó. Một mảng hai chiều **tmp** có kiểu **int** với 2 dòng và 3 cột có thể được khai báo như sau,

```
int tmp[2][3];
```

Mảng này sẽ chứa 2 x 3 (6) phần tử, và chúng có thể được biểu diễn như sau:

Dòng / Cột	**0**	**1**	**2**
0	e1	e2	e3
1	e4	e5	e6

Ở đó e1 – e6 biểu diễn cho các phần tử của mảng. Cả dòng và cột được đánh số từ 0. Phần tử e6 được xác định bằng dòng 1 và cột 2. Truy xuất đến phần tử này như sau:

```
tmp[1][2];
```

- **Khởi tạo mảng đa chiều:**

Khai báo mảng đa chiều có thể kết hợp với việc gán các giá trị khởi tạo. Cần phải cẩn thận lưu ý đến thứ tự các giá trị khởi tạo được gán cho các phần tử của mảng (chỉ có mảng **external** và **static** có thể được khởi tạo). Các phần tử trong dòng đầu tiên của mảng hai chiều sẽ được gán giá trị trước, sau đó đến các phần tử của dòng thứ hai, … Hãy xem sự khai báo mảng sau:

```
int ary[3][4] ={1,2,3,4,5,6,7,8,9,10,11,12};
```

Kết quả của phép khai báo trên sẽ như sau:

```
ary[0][0] = 1 ary[0][1] = 2 ary[0][2] = 3      ary[0][3]=
4
ary[1][0] = 5 ary[1][1] = 6 ary[1][2] = 7      ary[1][3]  =
8
ary[2][0] = 9 ary[2][1] = 10     ary[2][2]       =       11
    ary[2][3] = 12
```

Chú ý rằng chỉ số thứ 1 chạy từ 0 đến 2 và chỉ số thứ hai chạy tử 0 đến 3. Một điểm cần nhớ là các phần tử của mảng sẽ được lưu trữ ở những vị trí kế tiếp nhau trong bộ nhớ. Mảng **ary** ở trên có thể xem như là một mảng của 3 phần tử, mỗi phần tử là một mảng của 4 số nguyên, và sẽ xuất hiện như sau:

Dòng 0				**Dòng 1**				**Dòng 2**			
1	2	3	4	5	6	7	8	9	10	11	12

Thứ tự tự nhiên mà các giá trị khởi tạo được gán có thể thay đổi bằng hình thức nhóm các giá trị khởi tạo lại trong các dấu ngoặc nhọn {}. Quan sát sự khởi tạo sau:

```
int ary [3][4] ={
```

```
                    {1, 2, 3},
                    {4, 5, 6},
                    {7, 8, 9}
                };
```

Mảng sẽ được khởi tạo như sau:

```
ary[0][0]=1        ary[0][1]=2        ary[0][2]=3
     ary[0][3]=0
ary[1][0]=4        ary[1][1]=5        ary[1][2]=6
     ary[1][3]=0
ary[2][0]=7        ary[2][1]=8        ary[2][2]=9
     ary[2][3]=0
```

Một phần tử của mảng đa chiều có thể được sử dụng như một biến trong C bằng cách dùng các chỉ số để xác định phần tử của mảng.

Ví dụ 5:

```
/* Chương trình nhập các số vào một mảng hai chiều. */
#include <stdio.h>
void main()
{
     int arr[2][3];
     int row, col;

     for(row = 0; row < 2; row++)
     {
            for(col = 0; col < 3; col++)
            {
                printf("\nEnter a Number at [%d][%d]: ",
row, col);
                scanf("%d", &arr[row][col]);
              }
      }

     for(row = 0; row < 2; row++)
     {
          for(col = 0; col < 3; col++)
          {
```

```
      printf("\nThe Number at [%d][%d] is %d",
       row, col, arr[row][col]);
      }
      }
}
```

Một ví dụ về kết quả thực thi chương trình trên như sau:

```
Enter a Number at [0][0]: 10
Enter a Number at [0][1]: 100
Enter a Number at [0][2]: 45
Enter a Number at [1][0]: 67
Enter a Number at [1][1]: 45
Enter a Number at [1][2]: 230

The Number at [0][0] is 10
The Number at [0][1] is 100
The Number at [0][2] is 45
The Number at [1][0] is 67
The Number at [1][1] is 45
The Number at [1][2] is 230
```

- **Mảng hai chiều và chuỗi:**

Như chúng ta đã biết ở phần trước, một chuỗi có thể được biểu diễn bằng mảng một chiều, kiểu ký tự. Mỗi ký tự trong chuỗi được lưu trữ trong một phần tử của mảng. Mảng của chuỗi có thể được tạo bằng cách sử dụng mảng ký tự hai chiều. Chỉ số bên trái xác định số lượng chuỗi, và chỉ số bên phải xác định chiều dài tối đa của mỗi chuỗi. Ví dụ bên dưới khai báo một mảng chứa 25 chuỗi và mỗi chuỗi có độ dài tối đa 80 ký tự kể cả ký tự null.

```
char str_ary[25][80];
```

- **Ví dụ minh hoạ cách sử dụng của một mảng hai chiều:**

Ví dụ bên dưới minh hoạ cách dùng của mảng hai chiều như các chuỗi.

Xét bài toán tổ chức một danh sách tên theo thứ tự bảng chữ cái. Ví dụ sau đây nhập một danh sách các tên và sau đó sắp xếp chúng theo thứ tự bảng chữ cái.

Ví dụ 6

```
#include <stdio.h>
#include <string.h>
#include <conio.h>

void main()
{
      int i, n = 0;
      int item;
      char x[10][12];
      char temp[12];

      clrscr();
      printf("Enter each string on a separate line
\n\n");
      printf("Type 'END' when over \n\n");

      /* Read in the list of strings */
      do
      {
           printf("String %d: ", n + 1);
           scanf("%s", x[n]);
      } while (strcmp(x[n++], "END"));

      /*Reorder the list of strings */

      n = n - 1;
      for(item = 0; item < n - 1; ++item)
      {
           /* Find lowest of remaining strings */

           for(i = item + 1; i < n; ++i)
           {
                if(strcmp(x[item], x[i]) > 0)
                {
                     /*Interchange two strings*/
                     strcpy(temp, x[item]);
                     strcpy(x[item], x[i]);
```

```
                    strcpy(x[i], temp);
                }
            }
        }

        /* Display the arranged list of strings */

        printf("Recorded list of strings: \n");

        for(i = 0; i < n; ++i)
        {
            printf("\nString %d is %s", i + 1, x[i]);
        }
}
```

Chương trình trên nhập vào các chuỗi đến khi người dùng nhập vào từ "END". Khi END được nhập vào, chương trình sẽ sắp xếp danh sách các chuỗi và in ra theo thứ tự đã sắp xếp. Chương trình kiểm tra hai phần tử kế tiếp nhau. Nếu thứ tự của chúng không thích hợp, thì hai phần tử sẽ được đổi chỗ. Sự so sánh hai chuỗi được thực hiện với sự trợ giúp của hàm ***strcmp()*** và sự đổi chỗ được thực hiện với hàmg ***strcpy()***.

Một ví dụ về kết quả thực thi của chương trình như sau:

```
Enter each string on a separate line
Type 'END' when over
String 1:    has
String 2:    seen
String 3:    alice
String 4:    wonderland
String 5:    END
Record list of strings:
String 1 is alice
String 2 is has
String 3 is seen
String 4 is wonderland
```

Mục tiêu:

Kết thúc bài học này, bạn có thể:

- Giải thích các biến và hằng kiểu chuỗi
- Giải thích con trỏ trỏ đến chuỗi
- Thực hiện các thao tác nhập/xuất chuỗi
- Giải thích các hàm thao tác chuỗi
- Giải thích cách thức truyền mảng vào hàm như tham số
- Mô tả cách thức sử dụng chuỗi như các của hàm.

- **Các biến và hằng kiểu chuỗi**

Các biến chuỗi được sử dụng để lưu trữ một chuỗi các ký tự. Như các biến khác, các biến này phải được khai báo trước khi sử dụng.

```
char str[10];
```

str là một mảng các ký tự có thể lưu tối đa 10 ký tự. Giả sử str được gán một hằng chuỗi,

```
"WELL DONE"
```

Một **hằng chuỗi** là một dãy các ký tự nằm trong dấu nháy kép. Mỗi ký tự trong một chuỗi được lưu trữ như là một phần tử của mảng. Trong bộ nhớ, chuỗi được lưu trữ như sau:

‘	‘	‘	‘	‘	‘	‘	‘	‘	‘
W’	E’	L’	L’	’	D’	O’	N’	E’	\0’

- **Con trỏ trỏ đến chuỗi**

Chuỗi có thể được lưu và truy cập bằng cách sử dụng con trỏ kiểu ký tự. Một con trỏ kiểu ký tự trỏ đến một chuỗi được khai báo như sau:

```
char *pstr = "WELCOME";
```

Các thao tác nhập xuất chuỗi

Các thao tác nhập/xuất (I/O) chuỗi trong C được thực hiện bằng cách gọi các hàm. Các hàm này là một phần của thư viện nhập/xuất chuẩn tên **stdio.h**. Một chương trình muốn sử dụng các hàm nhập/xuất chuỗi phải có câu lệnh khai báo sau ở đầu chương trình:

```
#include <stdio.h>;
```

Khi chương trình có chứa câu lệnh này được biên dịch, thì nội dung của tập tin **stdio.h** sẽ trở thành một phần của chương trình.

- **Các thao tác nhập/xuất chuỗi đơn giản**

àm **gets()** là cách đơn giản nhất để nhập một chuỗi thông qua thiết bị nhập chuẩn. Các ký tự sẽ được nhập vào cho đến khi nhấn phím Enter. Hàm **gets()** thay thế ký tự kết thúc trở về đầu dòng '\n' bằng ký tự '\0'. Cú pháp hàm này như sau:

```
gets(str);
```

Trong đó **str** là một mảng ký tự đã được khai báo.

Tương tự, hàm **puts()** được sử dụng để hiển thị một chuỗi ra thiết bị xuất chuẩn. Ký tự xuống dòng sẽ kết thúc việc xuất chuỗi. Cú pháp hàm như sau:

```
puts(str);
```

Trong đó **str** là một mảng ký tự đã được khai báo và khởi tạo. Chương trình sau đây nhận vào một tên và hiển thị một .

Ví dụ 1:

```
#include <stdio.h>

void main()
{
 char name[20];
/*   name   is   declared   as   a   single   dimensional
character
      array */

 clrscr();              /* Clears the screen */
puts("Enter your name:");    /* Displays a message */

gets(name);             /* Accepts the input */

puts("Hi there: ");
puts(name);             /* Displays the input */

getch();
}
```

Nếu tên Lisa được nhập vào, chương trình trên cho ra kết quả:

```
Enter your name:
Lisa
Hi there:
Lisa
```

- **Các thao tác N/X chuỗi có định dạng**

Có thể sử đụng các hàm **scanf()** và **printf()** để nhập và hiển thị các giá trị chuỗi. Các hàm này được dùng để nhập và hiển thị các kiểu dữ liệu hỗn hợp trong một câu lệnh duy nhất. Cú pháp để nhập một chuỗi như sau:

```
scanf("%s", str);
```

Trong đó ký hiệu định dạng **%s** cho biết rằng một giá trị chuỗi sẽ được nhập vào. **str** là một mảng ký tự đã được khai báo. Tương tự, để hiển thị chuỗi, cú pháp sẽ là:

```
printf("%s", str);
```

trong đó ký hiệu định dạng **%s** cho biết rằng một giá trị chuỗi sẽ được hiển thị và **str** là một mảng ký tự đã được khai báo và khởi tạo.

Có thể sửa đổi chương trình bên trên để nhập vào và hiển thị một tên, sử dụng hàm scanf() và printf().

Ví dụ 2:

```
#include <stdio.h>

void main()
{
 char name[20];
 /* name is declared as a single dimensional
character
      array */

 clrscr();                    /* Clears the screen */
printf("Enter your name: "); /* Displays a message */

scanf("%s", name);           /* Accepts the input */

printf("Hi there: %s", name);     /* Displays the input
*/

getch();
}
```

Nếu nhập vào tên Brendan , chương trình trên cho ra kết quả:

```
Enter your name: Brendan
Hi there: Brendan
```

- **Các hàm về chuỗi**

C hỗ trợ rất nhiều hàm về chuỗi. Các hàm này có thể tìm thấy trong tập tin string.h. Một số thao tác mà các hàm này thực hiện là:

- Nối chuỗi
- So sánh chuỗi
- Định vị một ký tự trong chuỗi
- Sao chép một chuỗi sang chuỗi khác
- chiều dài chuỗi.

- **Hàm strcat()**

Hàm **strcat()** được sử dụng để nối hai chuỗi vào . Cú pháp hàm là:

```
strcat(str1, str2);
```

trong đó **str1** và **str2** là hai đã được khai báo và khởi tạo.

Ví dụ 3:

```
#include<stdio.h>
#include<string.h>
void  main()
{
char firstname[15];
char lastname[15];
clrscr();
printf("Enter your first name: ");
scanf("%s", firstname);
printf("Enter your last name:");
scanf("%s", lastname);
strcat(firstname, lastname);
/* Attaches the contents of lastname at the end of
firstname */
printf("%s", firstname);
getch();
}
```

```
Enter your first name: Carla
Enter your last name: Johnson
CarlaJohnson
```

- **Hàm strcmp()**

```
strcmp(str1, str2);
```

trong đó **str1** và **str2** là hai đã được khai báo và khởi tạo.Hàm trả về giá trị:

- Nhỏ hơn 0 nếu str1<str2
- 0 nếu str1 = str2
- Lớn hơn 0 nếu str1>str2

Ví dụ 4:

```
#include <stdio.h>
#include<string.h>
void  main()
{
char name1[15] = "Geena";
char name2[15] = "Dorothy";
char name3[15] = "Shania";
char name4[15] = "Geena";
int i;

clrscr();

i = strcmp(name1,name2);
printf("%s compared with %s returned %d\n", name1,
name2, i);

i=strcmp(name1, name3);
printf("%s compared with %s returned %d\n", name1,
name3, i);

i=strcmp(name1,name4);
printf("%s compared with %s returned %d\n", name1,
name4, i);

getch();
}
 Geena compared with Dorothy returned 3
 Geena compared with Shania returned -12
 Geena compared with Geena returned 0
```

- **Hàm strchr()**

Hàm **strchr()** xác định vị trí xuất hiện của một ký tự trong một chuỗi. Cú pháp hàm là:

```
strchr(str, chr);
```

trong đó str là một mảng ký tự hay chuỗi. chr là một biến ký tự chứa giá trị cần tìm. Hàm trả về con trỏ trỏ đến giá trị tìm được trong chuỗi, hoặc NULL nếu không tìm .

Chương trình sau đây xác định liệu ký tự 'a' có xuất hiện trong .

Ví dụ 5:

```
#include <stdio.h>
#include<string.h>
void  main()
{
char str1[15] = "New York";
char str2[15] = "Washington";
char chr = 'a', *loc;
clrscr();
 loc = strchr(str1, chr);
/* Checks for the occurrence of the character value held by chr in the first city name */
if(loc != NULL)
 printf("%c occurs in %s\n", chr, str1);
else
printf("%c does not occur in %s\n", chr, str1);

loc = strchr(str2, chr);
/* Checks for the occurrence of the character in the second city name */

if(loc != NULL)
printf("%c occurs in %s\n", chr, str2);
else
printf("%c does not occur in %s\n", chr, str2);

getch();
}
a does not occur in New York
a occurs in Washington
```

- **Hàm strcpy()**

Trong C không có toán tử nào xử lý một chuỗi như là một đơn vị duy nhất. Vì vậy, phép gán một giá trị chuỗi này cho một chuỗi khác đòi hỏi phải sử dụng hàm strcpy(). Cú pháp hàm là:

```
strcpy(str1, str2);
```

trong đó **str1** và **str2** là hai mảng ký tự đã được khai báo và khởi tạo. Hàm sao chép giá trị **str2** vào **str1** và trả về **str1**.

Chương trình sau đây minh họa việc sử dụng hàm strcpy(). Nó thay đổi tên của một khách sạn và hiển thị tên mới.

Ví dụ 6:

```
#include <stdio.h>
#include<string.h>

void  main()
{
char hotelname1[15] = "Sea View";
char hotelname2[15] = "Sea Breeze";

clrscr();

printf("The old name is %s\n", hotelname1);

strcpy(hotelname1, hotelname2);
/*Changes the hotel name*/

printf("The new name is %s\n", hotelname1);
/*Displays the new name*/

getch();
}
The old name is Sea View
The new name is Sea Breeze
```

- **Hàm strlen()**

```
strlen(str);
```

trong đó **str** là mảng ký tự đã được khai báo và khởi tạo. Hàm trả về của **str**.

Ví dụ 7:

```
#include<stdio.h>
#include<string.h>

void  main()
{
char compname[20] = "Microsoft";
int len, ctr;

clrscr();

len = strlen(compname);
/* Determines the length of the string */

for(ctr = 0; ctr < len; ctr++)
/*  Accesses  and  displays  each  character  of  the
string*/
printf("%c * ", compname[ctr]);

getch();
}
M * i * c * r * o * s * o * f * t *
```

- **Truyền mảng vào hàm**

Trong C, khi một mảng được truyền vào hàm như một số, thì chỉ có địa chỉ của mảng được truyền vào. Tên mảng không kèm theo chỉ số là địa chỉ của mảng.

```
void main()
{
    int ary[10];
    .
    .
    fn_ary(ary);
    .
    .
```

```
}
fn_ary (int ary [10])    /* sized array */
{
    :
}
```

hoặc

```
fn_arry (int ary [])     /*unsized array */
{
    :
}
```

Chương trình sau đây nhận các số vào một mảng số nguyên. Sau đó mảng này sẽ được truyền vào hàm sum_arr(). Hàm sẽ tính toán và trả về tổng của các số nguyên trong mảng.

Ví dụ 8:

```
#include <stdio.h>
void  main()
{
 int num[5], ctr, sum = 0;
 int  sum_arr(int  num_arr[]);  /*  Function  declaration
*/

 clrscr();

 for(ctr  =  0;  ctr  <  5;  ctr++)  /*Accepts  numbers  into
the array*/
 {
      printf("\nEnter number %d: ", ctr+1);
      scanf("%d", &num[ctr]);
 }

 sum = sum_arr(num); /* Invokes the function */

 printf("\nThe sum of the array is %d", sum);

 getch();
}

int sum_arr(int num_arr[]) /* Function definition */
```

```
{
 int i, total;

 for(i = 0, total = 0; i < 5; i++) /* Calculates the
sum */
      total += num_arr[i];

 return total; /* Returns the sum to main() */
}

Enter number 1: 5

Enter number 2: 10

Enter number 3: 13

Enter number 4: 26

Enter number 5: 21

The sum of the array is 75
```

- **Truyền chuỗi vào hàm**

Chuỗi, hay mảng ký tự, có thể được truyền vào hàm. Ví dụ, chương trình sau đây sẽ nhận vào các chuỗi và một mảng ký tự hai chiều. Sau đó, mảng này sẽ được truyền vào trong một hàm dùng để chuỗi dài nhất trong mảng.

Ví dụ 9:

```
#include <stdio.h>
void  main()
{
char lines[5][20];
int ctr, longctr = 0;

int longest(char lines_arr[][20]);
/* Function declaration */

 clrscr();

 for(ctr = 0; ctr < 5; ctr++)
```

```
     /* Accepts string values into the array */
     {
          printf("\nEnter string %d: ", ctr + 1);
          scanf("%s", lines[ctr]);
     }

     longctr = longest(lines);
     /* Passes the array to the function */

     printf("\nThe longest string is %s", lines[longctr]);

     getch();
    }

    int   longest(char   lines_arr[][20])   /*   Function
definition */
    {
     int i = 0, l_ctr = 0, prev_len, new_len;

     prev_len = strlen(lines_arr[i]);
     /* Determines the length of the first element */

     for(i++; i < 5; i++)
     {
          new_len = strlen(lines_arr[i]);
          /* Determines the length of the next element */

          if(new_len > prev_len)
               l_ctr = i;
    /* Stores the subscript of the longer string */

          prev_len = new_len;
     }
     return l_ctr;
     /* Returns the subscript of the longest string */
    }

    Enter string 1: The
```

```
Enter string 2: Sigma

Enter string 3: Protocol

Enter string 4: Robert

Enter string 5: Ludlum

The longest string is Protocol
```

4.5.4. Quy trình kỹ thuật trong bài học của GV và SV:

*** Quy trình thị phạm của GV**

- Bước 1: Chuẩn bị bài giảng, tài liệu và điều kiện trang thiết bị giảng dạy
- Bước 2: Chuẩn bị bài tập mẫu và đưa ra nội dung thực hành
- Bước 3: Phổ biến các quy trình, các bước thực hiện
- Bước 4: Hướng dẫn nội dung bài học, Hướng dẫn thị phạm mẫu
- Bước 5: Quản lý lớp học
- Bước 6: Giải đáp, tư vấn, đánh giá, nhận xét
- Bước 7: Chấm điểm và nộp các bài đánh giá cho bộ môn

*** Quy trình thực hiện bài của SV**

- Bước 1: SV chuẩn bị tài liệu, vở, bút, giấy nháp, tài liệu tra cứu
- Bước 2: Đọc tài liệu Lập trình căn bản C phần mảng và chuỗi
- Bước 3: Sinh viên trình bày bài thực hành, thảo luận và nhận xét góp ý.

4.5.5. Các mẫu hình sản phẩm cho SV tham khảo:

- SV xem video

4.5.6. Phần tự thực hành thao tác thường xuyên của SV trong bài học:

Câu 1: Viết một chương trình để sắp xếp các tên sau đây theo thứ tự abc.

```
George
Albert
Tina
Xavier
Roger
Tim
William
```

Câu 2: Viết một chương trình đếm số ký tự nguyên âm trong một dòng văn bản.

Câu 3: Viết một chương trình nhập các số sau đây vào một mảng và đảo ngược mảng

```
34
45
56
67
89
```

Câu 4: Viết một chương trình để nhập vào hai chuỗi. Chương trình sẽ xác định liệu chuỗi thứ nhất có xuất hiện ở cuối chuỗi thứ hai không.

Câu 5: Viết một chương trình nhập vào một mảng các số và hiển thị giá trị trung bình. Sử dụng hàm để tính giá trị trung bình.

4.5.7. Sản phẩm thực hành:

+ Các ví dụ demo.

4.5.8. Điều kiện để GV- SV thực hiện bài học thực hành;

4.5.8.1. Điều kiện chuẩn bị trước bài học

+ Giảng viên:

- Đề cương chi tiết bài giảng
- Thiết kế bài giảng
- Học liệu tham khảo, mở rộng

+ Sinh viên:

- Đọc kỹ bài học trước khi giảng viên lên lớp
- Chuẩn bị vấn đề thảo luận, trao đổi với giảng viên
- Các tài liệu tham khảo, mở rộng do giảng viên yêu cầu

4.5.8.2. Kỹ thuật và phương tiện dạy học:

- Bảng, phấn viết
- Màn hình lớn
- Máy tính, máy chiếu

4.5.8.3. Phương pháp tổ chức dạy-học:

- Thuyết trình, diễn giảng, thực hành
- Phát vấn
- Trao đổi, thảo luận
- Cemina
- Vấn đáp

5. Tín chỉ 3

5.1 Danh mục tên bài

TT	Tên bài giảng	Tổng số tiết trên lớp của GV	Số tiết GV trình bầy	Số tiết GV hướng dẫn thảo luận	Bài tự luận/ nghiên cứu ngoài xã hội	Giảng viên thực hiện
1	**Bài 1: Con trỏ** - Khái báo biến con trỏ - Truyền địa chỉ sang hàm - Con trỏ và mảng - Con trỏ trỏ đến mảng trong hàm - Con trỏ và chuỗi - Khởi tạo mảng con trỏ trỏ đến chuỗi - Xử lý con trỏ trỏ đến chuỗi - Con trỏ trỏ đến con trỏ	10	5	5	0	10
2	**Bài 2: Tập tin** - Ghi, đọc mảng - Ghi, đọc structure - Các mode khác để mở tập tin - Một số hàm thao tác trên file khác	10	5	5	0	10
3	**Bài 3: Các kiểu dữ liệu tự tạo** -Structure -Enum	10	5	5	0	10
	Tổng:	**30**	**15**	**15**	**0**	30

5.2 Nội dung bài giảng 1

5.2.1 Tên bài giảng 1: BÀI 01: CON TRỎ, TỆP TIN VÀ KIỂU DỮ LIỆU TỰ TẠO

Số tiết lên lớp của GV: 07 tiết; Số tiết tự làm bài của SV : 07 tiết

5.2.2. Phần mở đầu tiếp cận bài;

Con trỏ cung cấp một cách thức truy xuất biến mà không tham chiếu trực tiếp đến biến. Nó cung cấp cách thức sử dụng địa chỉ. Bài này sẽ đề cập đến các khái niệm về con trỏ và cách sử dụng chúng trong C.

5.2.3. Phần kiến thức, kỹ thuật căn bản:

5.2.3.1. Phần kiến thức căn bản

- **Phần kiến thức**

Mục tiêu:

Kết thúc bài học này, bạn có thể:

- Hiểu con trỏ là gì, và con trỏ được sử dụng ở đâu
- Biết cách sử dụng biến con trỏ và các toán tử con trỏ
- Gán giá trị cho con trỏ
- Hiểu các phép toán số học con trỏ
- Hiểu các phép toán so sánh con trỏ
- Biết cách truyền tham số con trỏ cho hàm
- Hiểu cách sử dụng con trỏ kết hợp với mảng một chiều
- Hiểu cách sử dụng con trỏ kết hợp với mảng đa chiều
- Hiểu cách cấp phát bộ nhớ được thực hiện như thế nào

- **Con trỏ là gì?**

Một **con trỏ** là một biến, nó chứa địa chỉ vùng nhớ của một biến khác, chứ không lưu trữ giá trị của biến đó. Nếu một biến chứa địa chỉ của một biến khác, thì biến này được gọi là **con trỏ** đến biến thứ hai kia. Một con trỏ cung cấp phương thức gián tiếp để truy xuất giá trị của các phần tử dữ liệu. Xét hai biến var1 và var2, var1 có giá trị 500 và được lưu tại địa chỉ 1000 trong bộ nhớ. Nếu var2 được khai báo như là một con trỏ tới biến var1,sự biểu diễn sẽ như sau:

Vị trí Bộ nhớ	**Giá trị lưu trữ**	**Tên biến**
1000	500	var1
1001		
1002		
.		
.		
1108	1000	var2

Ở đây, var2 chứa giá trị 1000, đó là địa chỉ của biến var1.

Các con trỏ có thể trỏ đến các biến của các kiểu dữ liệu cơ sở như **int**, **char**, hay **double** hoặc dữ liệu có cấu trúc như **mảng**.

Tại sao con trỏ được dùng?

Con trỏ có thể được sử dụng trong một số trường hợp sau:

- Để trả về nhiều hơn một giá trị từ một hàm
- Thuận tiện hơn trong việc truyền các mảng và chuỗi từ một hàm đến một hàm khác
- Sử dụng con trỏ để làm việc với các phần tử của mảng thay vì truy xuất trực tiếp vào các phần tử này
- Để cấp phát bộ nhớ động và truy xuất vào vùng nhớ được cấp phát này (dynamic memory allocation)

- **Các biến con trỏ**

Nếu một biến được sử dụng như một con trỏ, nó phải được khai báo trước. Câu lệnh khai báo con trỏ bao gồm một kiểu dữ liệu cơ bản, một dấu *, và một tên biến. Cú pháp tổng quát để khai báo một biến con trỏ như sau: `type *name;`

Ở đó **type** là một kiểu dữ liệu bất kỳ, và **name** là tên của biến con trỏ. Câu lệnh khai báo trên nói với trình biên dịch là **name** được sử dụng để lưu địa chỉ của một biến có kiểu dữ liệu **type**. Trong câu lệnh khai báo, * xác định rằng một biến con trỏ đang được khai báo.

Trong ví dụ của **var1** và **var2** ở trên, vì **var2** là một con trỏ giữ địa chỉ của biến **var1** có kiểu **int**, nó sẽ được khai báo như sau:

```
int *var2;
```

Bây giờ, **var2** có thể được sử dụng trong một chương trình để trực tiếp truy xuất giá trị của **var1**. Nhớ rằng, **var2** không phải có kiểu dữ liệu **int** nhưng nó là một con trỏ trỏ đến một biến có kiểu dữ liệu int.

Kiểu dữ liệu cơ sở của con trỏ xác định kiểu của biến mà con trỏ trỏ đến. Về mặt kỹ thuật, một con trỏ có kiểu bất kỳ có thể trỏ đến bất kỳ vị trí nào trong bộ nhớ. Tuy nhiên, tất cả các phép toán số học trên con trỏ đều có liên quan đến kiểu cơ sở của nó, vì vậy khai báo kiểu dữ liệu của con trỏ một cách rõ ràng là điều rất quan trọng.

- **Các toán tử con trỏ**

Có hai toán tử đặc biệt được dùng với con trỏ: * và **&**. Toán tử **&** là một toán tử một ngôi và nó **trả về địa chỉ** của toán hạng. Ví dụ,

```
var2 = &var1;
```

lấy địa chỉ vùng nhớ của biến var1 gán cho var2. Địa chỉ này là vị trí ô nhớ bên trong máy tính của biến var1 và nó không làm gì với giá trị của var1. **Toán tử & có thể hiểu là trả về "địa chỉ của".** Vì vậy, phép gán trên có nghĩa là "var2 nhận địa chỉ

của var1". Trở lại, giá trị của var1 là 500 và nó dùng vùng nhớ **1000** để lưu giá trị này. Sau phép gán trên, **var2** sẽ có giá trị **1000**.

Toán tử thứ hai được dùng với con trỏ là phần của toán tử **&**. Nó là một toán tử một ngôi và **trả về giá trị** chứa trong vùng nhớ được trỏ bởi giá trị của biến con trỏ.

Xem ví dụ trước, ở đó **var1** có giá trị 500 và được lưu trong vùng nhớ 1000, sau câu lệnh

```
var2 = &var1;
```

var chứa giá trị 1000, và sau lệnh gán

```
temp = *var2;
```

temp sẽ chứa 500 không phải là 1000. Toán tử * có thể được hiu là "tại địa chỉ".

Cả hai toán tử * và **&** có độ ưu tiên cao hơn tất cả các toán tử toán học ngoại trừ toán tử lấy giá trị âm. Chúng có cùng độ ưu tiên với toán tử lấy giá trị âm (-).

Chương trình dưới đây in ra giá trị của một biến kiểu số nguyên, địa chỉ của nó được lưu trong một biến con trỏ, và chương trình cũng in ra địa chỉ của biến con trỏ.

```
#include <stdio.h>
void main()
{
    int var = 500, *ptr_var;

    /* var is declared as an integer and ptr_var as a
pointer
       pointing to an integer */

    ptr_var = &var; /*stores address of var in
ptr_var*/
    /* Prints value of variable (var) and address where
var is
                        stored */

    printf("The value %d is stored at address %u:",
var, &var);

    /* Prints value stored in ptr variable (ptr_var)
and address
      where ptr_var is stored */

    printf("\nThe value %u is stored at address: %u",
```

```
                                                    ptr_var,
&ptr_var);
       /* Prints value of variable (var) and address
where
          var is stored, using pointer to variable */
       printf("\nThe value %d is stored at address:%u",
*ptr_var,          ptr_var);
    }

       The value 500 is stored at address: 65500
       The value 65500 is stored at address: 65502
       The value 500 is stored at address: 65500
```

Trong ví dụ trên, **ptr_var** chứa địa chỉ 65500, là địa chỉ vùng nhớ lưu trữ giá trị của **var**. Nội dung ô nhớ 65500 này có thể lấy được bằng cách sử dụng toán tử *, như *ptr_var. Lúc này *ptr_var giá trị 500, là giá trị của **var**. Bởi vì **ptr_var** cũng là một biến, nên địa chỉ của nó có thể được in ra bằng toán tử **&**. Trong ví dụ trên, ptr_var được lưu tại địa chỉ 65502. Mã quy cách %u chỉ định cách in giá trị các tham số theo kiểu số nguyên không dấu (unsigned int).

Nhớ lại là, một biến kiểu số nguyên chiếm 2 bytes bộ nhớ. Vì vậy, giá trị của var được lưu trữ tại địa chỉ 65500 và trình biên dịch cấp phát ô nhớ kế tiếp 65502 cho ptr_var. Tương tự, một số thập phân kiểu float yêu cầu 4 bytes và kiểu double yêu cầu 8 bytes. Các biến con trỏ lưu trữ một giá trị nguyên. Với hầu hết các chương trình sử dụng con trỏ, kiểu con trỏ có thể xem như một giá trị 16-bit – chiếm 2 bytes bộ nhớ.

Chú ý rằng hai câu lệnh sau cho ra cùng một kết quả.

```
printf("The value is %d", var);
printf("The value is %d", *(&var));
```

- **Gán giá trị cho con trỏ**

Các giá trị có thể được gán cho biến con trỏ thông qua toán tử **&**. Câu lệnh gán sẽ là:

```
ptr_var = &var;
```

Lúc này địa chỉ của **var** được lưu trong biến **ptr_var**. Cũng có thể gán giá trị cho con trỏ thông qua một biến con trỏ khác trỏ đến một phần tử dữ liệu có cùng kiểu.

```
ptr_var = &var;
ptr_var2 = ptr_var;
```

Giá trị NULL cũng có thể được gán đến một con trỏ bằng số 0 như sau:

```
ptr_var = 0;
```

Các biến cũng có thể được gán giá trị thông qua con trỏ của chúng.

```
*ptr_var  =  10;
```

sẽ gán **10** cho biến **var** nếu **ptr_var** trỏ đến **var**.

Nói chung, các biểu thức có chứa con trỏ cũng theo cùng qui luật như các biểu thức khác trong C. Điều quan trọng cần chú ý phải gán giá trị cho biến con trỏ trước khi sử dụng chúng; nếu không chúng có thể trỏ đến một giá trị không xác định nào đó.

- **Phép toán số học con trỏ**

Chỉ phép cộng và trừ là các toán tử có thể thực hiện trên các con trỏ. Ví dụ sau minh họa điều này:

```
int var, *ptr_var;
ptr_var = &var;
var = 500;
```

Trong ví dụ trên, chúng ta giả sử rằng **var** được lưu tại địa chỉ **1000**. Sau đó, giá trị **1000** sẽ được lưu vào **ptr_var**. Vì kiểu số nguyên chiếm 2 bytes, nên sau biểu thức:

```
ptr_var++ ;
```

ptr_var sẽ chứa **1002** mà KHÔNG phải là **1001**. Điều này có nghĩa là ptr_var bây giờ trỏ đến một số nguyên được lưu tại địa chỉ 1002. Mỗi khi ptr_var được tăng lên, nó sẽ trỏ đến số nguyên kế tiếp và bởi vì các số nguyên là 2 bytes, ptr_var sẽ được tăng trị là 2. Điều này cũng tương tự với phép toán giảm trị.

Đây là một vài ví dụ.

++ptr_var or ptr_var++	Trỏ đến **số nguyên** kế tiếp đứng sau var
--ptr_var or ptr_var--	Trỏ đến **số nguyên** đứng trước var
ptr_var + i	Trỏ đến số nguyên thứ i sau var
ptr_var - i	Trỏ đến số nguyên thứ i trước var
++*ptr_var or (*ptr_var)++	Sẽ tăng trị **var** bởi 1
*ptr_var++	Sẽ tác động đến giá trị của số nguyên kế tiếp sau var

Mỗi khi một con trỏ được tăng trị, nó sẽ trỏ đến ô nhớ của phần tử kế tiếp. Mỗi khi nó được giảm trị, nó sẽ trỏ đến vị trí của phần tử đứng trước nó. Với những con trỏ trỏ tới các ký tự, nó xuất hiện bình thường, bởi vì mỗi ký tự chiếm 1 byte. Tuy nhiên, tất cả những con trỏ khác sẽ tăng hoặc giảm trị tuỳ thuộc vào độ dài kiểu dữ liệu mà chúng trỏ tới.

Như đã thấy trong các ví dụ trên, ngoài các toán tử tăng trị và giảm trị, các số nguyên cũng có thể được cộng vào và trừ ra với con trỏ. Ngoài phép cộng và trừ một con trỏ với một số nguyên, không có một phép toán nào khác có thể thực hiện được trên các con trỏ. Nói rõ hơn, các con trỏ không thể được nhân hoặc chia. Cũng như kiểu float và double không thể được cộng hoặc trừ với con trỏ.

- **So sánh con trỏ.**

Hai con trỏ có thể được so sánh trong một biểu thức quan hệ. Tuy nhiên, điều này chỉ có thể nếu cả hai biến này đều trỏ đến các biến có cùng kiểu dữ liệu. **ptr_a** và **ptr_b** là hai biến con trỏ trỏ đến các phần tử dữ liệu **a** và **b**. Trong trường hợp này, các phép so sánh sau đây là có thể thực hiện:

ptr_a < ptr_b	Trả về giá trị true nếu **a** được lưu trữ ở vị trí trước **b**
ptr_a > ptr_b	Trả về giá trị true nếu **a** được lưu trữ ở vị trí sau **b**
ptr_a <= ptr_b	Trả về giá trị true nếu **a** được lưu trữ ở vị trí trước **b** hoặc ptr_a và ptr_b trỏ đến cùng một vị trí
ptr_a >= ptr_b	Trả về giá trị true nếu **a** được lưu trữ ở vị trí sau **b** hoặc ptr_a và ptr_b trỏ đến cùng một vị trí
ptr_a == ptr_b	Trả về giá trị true nếu cả hai con trỏ ptr_a và ptr_b trỏ đến cùng một phần tử dữ liệu.
ptr_a != ptr_b	Trả về giá trị true nếu cả hai con trỏ ptr_a và ptr_b trỏ đến các phần tử dữ liệu khác nhau nhưng có cùng kiểu dữ liệu.
ptr_a == NULL	Trả về giá trị true nếu ptr_a được gán giá trị NULL (0)

Tương tự, nếu **ptr_begin** và **ptr_end** trỏ đến các phần tử của cùng một mảng thì,

```
ptr_end - ptr_begin
```

sẽ trả về số bytes cách biệt giữ hai vị trí mà chúng trỏ đến.

- **Con trỏ và mảng một chiều**

Tên của một mảng thật ra là một con trỏ trỏ đến phần tử đầu tiên của mảng đó. Vì vậy, nếu **ary** là một mảng một chiều, thì địa chỉ của phần tử đầu tiên trong mảng có thể được biểu diễn là **&ary[0]** hoặc đơn giản chỉ là **ary**. Tương tự, địa chỉ của phần tử mảng thứ hai có thể được viết như **&ary[1]** hoặc **ary+1**,... Tổng quát, địa chỉ của phần tử mảng thứ (i + 1) có thể được biểu diễn là **&ary[i]** hay **(ary+i)**. Như vậy, địa chỉ của một phần tử mảng bất kỳ có thể được biểu diễn theo hai cách:

- Sử dụng ký hiệu & trước một phần tử mảng
- Sử dụng một biểu thức trong đó chỉ số được cộng vào tên của mảng.

Ghi nhớ rằng trong biểu thức **(ary + i), ary** tượng trưng cho một địa chỉ, trong khi **i** biểu diễn số nguyên. Hơn thế nữa, **ary** là tên của một mảng mà các phần tử có thể có kiểu số nguyên, ký tự, số thập phân,… (dĩ nhiên, tất cả các phần tử của mảng phải có cùng kiểu dữ liệu). Vì vậy, biểu thức ở trên không chỉ là một phép cộng; nó thật ra là xác định một địa chỉ, một số xác định của các ô nhớ . Biểu thức **(ary + i)** là một sự trình bày cho một địa chỉ chứ không phải là một biểu thức toán học.

Như đã nói ở trước, số lượng ô nhớ được kết hợp với một mảng sẽ tùy thuộc vào kiểu dữ liệu của mảng cũng như là kiến trúc của máy tính. Tuy nhiên, người lập trình chỉ có thể xác định địa chỉ của phần tử mảng đầu tiên, đó là tên của mảng (trong trường hợp này là **ary**) và số các phần tử tiếp sau phần tử đầu tiên, đó là, một giá trị chỉ số. Giá trị của **i** đôi khi được xem như là một **độ dời** khi được dùng theo cách này.

Các biểu thức **&ary[i]** và **(ary+i)** biểu diễn địa chỉ phần tử thứ **i** của **ary**, và như vậy một cách logic là cả **ary[i]** và ***(ary + i)** đều biểu diễn nội dung của địa chỉ đó, nghĩa là, giá trị của phần tử thứ **i** trong mảng **ary**. Cả hai cách có thể thay thế cho nhau và được sử dụng trong bất kỳ ứng dụng nào khi người lập trình mong muốn.

Chương trình sau đây biểu diễn mối quan hệ giữa các phần tử mảng và địa chỉ của chúng.

```
#include<stdio.h>

void main()
{
    static int ary[10] = {1, 2, 3, 4, 5, 6, 7, 8,
9, 10};
    int i;
    for (i = 0; i < 10; i ++)
    {
        printf("\n i = %d , ary[i] = %d , *(ary+i)=
%d ", i,
                    ary[i], *(ary + i));
    printf("&ary[i] = %X , ary + i = %X", &ary[i],
ary + i);
        /* %X gives unsigned hexadecimal */
    }
}
```

Chương trình trên định nghĩa mảng một chiều **ary**, có 10 phần tử kiểu số nguyên, các phần tử mảng được gán giá trị tương ứng là 1, 2, ..10. Vòng lặp **for** được dùng để hiển thị giá trị và địa chỉ tương ứng của mỗi phần tử mảng. Chú ý rằng, giá trị của mỗi

phần tử được xác định theo hai cách khác nhau, ary[i] và *(ary + i), nhằm minh họa sự tương đương của chúng. Tương tự, địa chỉ của mỗi phần tử mảng cũng được hiển thị theo hai cách. Kết quả thực thi của chương trình như sau:

```
    i=0       ary[i]=1   *(ary+i)=1  &ary[i]=194    ary+i  =
194
    i=1       ary[i]=2   *(ary+i)=2  &ary[i]=196    ary+i  =
196
    i=2       ary[i]=3   *(ary+i)=3  &ary[i]=198    ary+i  =
198
    i=3       ary[i]=4   *(ary+i)=4  &ary[i]=19A    ary+i  =
19A
    i=4       ary[i]=5   *(ary+i)=5  &ary[i]=19C    ary+i  =
19C
    i=5       ary[i]=6   *(ary+i)=6  &ary[i]=19E    ary+i  =
19E
    i=6       ary[i]=7   *(ary+i)=7  &ary[i]=1A0    ary+i  =
1A0
    i=7       ary[i]=8   *(ary+i)=8  &ary[i]=1A2    ary+i  =
1A2
    i=8       ary[i]=9   *(ary+i)=9  &ary[i]=1A4    ary+i  =
1A4
    i=9       ary[i]=10  *(ary+i)=10 &ary[i]=1A6    ary+i  =
1A6
```

Kết quả này trình bày rõ ràng sự khác nhau giữa **ary[i]** - biểu diễn giá trị của phần tử thứ **i** trong mảng, và **&ary[i]** - biểu diễn địa chỉ của nó.

Khi gán một giá trị cho một phần tử mảng như **ary[i]**, vế trái của lệnh gán có thể được viết là **ary[i]** hoặc ***(ary + i)**. Vì vậy, một giá trị có thể được gán trực tiếp đến một phần tử mảng hoặc nó có thể được gán đến vùng nhớ mà địa chỉ của nó là phần tử mảng. Đôi khi cần thiết phải gán một địa chỉ đến một định danh. Trong những trường hợp như vậy, một con trỏ phải xuất hiện trong vế trái của câu lệnh gán. Không thể gán một địa chỉ tùy ý cho một tên mảng hoặc một phần tử của mảng. Vì vậy, các biểu thức như **ary**, **(ary + i)** và **&ary[i]** không thể xuất hiện trong vế trái của một câu lệnh gán. Hơn thế nữa, địa chỉ của một mảng không thể thay đổi một cách tùy ý, vì thế các biểu thức như **ary++** là không được phép. Lý do là vì: **ary** là địa chỉ của mảng **ary.** Khi mảng được khai báo, bộ liên kết đã quyết định mảng được bắt đầu ở đâu, ví dụ, bắt đầu

ở địa chỉ 1002. Một khi địa chỉ này được đưa ra, mảng sẽ ở đó. Việc cố gắng tăng địa chỉ này lên là điều vô nghĩa, giống như khi nói

```
x = 5++;
```

Bởi vì hằng không thể được tăng trị, trình biên dịch sẽ đưa ra thông báo lỗi.

Trong trường hợp mảng ary, **ary** cũng được xem như là một **hằng con trỏ**. Nhớ rằng, (ary + 1) không di chuyển mảng **ary** đến vị trí (ary + 1), nó chỉ trỏ đến vị trí đó, trong khi ary++ cố găng dời **ary** sang 1 vị trí.

Địa chỉ của một phần tử không thể được gán cho một phần tử mảng khác, mặc dù giá trị của một phần tử mảng có thể được gán cho một phần tử khác thông qua con trỏ.

```
&ary[2]     =    &ary[3]; /* không cho phép*/
ary[2]=     ary[3];  /* cho phép*/
```

Nhớ lại rằng trong hàm scanf(), tên các tham biến kiểu dữ liệu cơ bản phải đặt sau dấu (&), trong khi tên tham biến mảng là ngoại lệ. Điều này cũng dễ hiểu. Vì scanf() đòi hỏi địa chỉ bộ nhớ của từng biến dữ liệu trong danh sách tham số, trong khi toán tử & trả về địa chỉ bộ nhớ của biến, do đó trước tên biến phải có dấu &. Tuy nhiên dấu & không được yêu cầu đối với tên mảng, bởi vì tên mảng tự biểu diễn địa chỉ của nó.Tuy nhiên, nếu một phần tử trong mảng được đọc, dấu & cần phải sử dụng.

```
scanf("%d", *ary)/*  đối với phần tử đầu tiên */
                  scanf("%d", &ary[2]) /*   đối      với
phần tử bất kỳ */
```

- **Con trỏ và mảng nhiều chiều**

Một mảng nhiều chiều cũng có thể được biểu diễn dưới dạng con trỏ của mảng một chiều (tên của mảng) và một độ dời (chỉ số). Thực hiện được điều này là bởi vì một mảng nhiều chiều là một tập hợp của các mảng một chiều.Ví dụ, một mảng hai chiều có thể được định nghĩa như là một con trỏ đến một nhóm các mảng một chiều kế tiếp nhau. Cú pháp báo mảng hai chiều có thể viết như sau:

```
data_type (*ptr_var)[expr 2];
```

thay vì

```
data_type array[expr 1][expr 2];
```

Khái niệm này có thể được tổng quát hóa cho các mảng nhiều chiều, đó là,

```
data_type (*ptr_var)[exp 2] .... [exp N];
```

thay vì

```
data_type array[exp 1][exp 2] ... [exp N];
```

Trong các khai báo trên, `data_type` là kiểu dữ liệu của mảng, `ptr_var` là tên của biến con trỏ, `array` là tên mảng, và `exp 1`, `exp 2`, `exp 3`, ... `exp N` là các giá trị nguyên dương xác định số lượng tối đa các phần tử mảng được kết hợp với mỗi chỉ số.

Chú ý dấu ngoặc () bao quanh tên mảng và dấu * phía trước tên mảng trong cách khai báo theo dạng con trỏ. Cặp dấu ngoặc () là không thể thiếu, ngược lại cú pháp khai báo sẽ khai báo một mảng của các con trỏ chứ không phải một con trỏ của một nhóm các mảng.

Ví dụ, nếu **ary** là một mảng hai chiều có 10 dòng và 20 cột, nó có thể được khai báo như sau:

```
int (*ary)[20];
```

thay vì

```
int ary[10][20];
```

Trong sự khai báo thứ nhất, **ary** được định nghĩa là một con trỏ trỏ tới một nhóm các mảng một chiều liên tiếp nhau, mỗi mảng có 20 phần tử kiểu số nguyên. Vì vậy, **ary** trỏ đến phần tử đầu tiên của mảng, đó là dòng đầu tiên (dòng 0) của mảng hai chiều. Tương tự, (ary + 1) trỏ đến dòng thứ hai của mảng hai chiều, ...

Một mảng thập phân ba chiều **fl_ary** có thể được khai báo như:

```
float (*fl_ary)[20][30];
```

thay vì

```
float fl_ary[10][20][30];
```

Trong khai báo đầu, `fl_ary` được định nghĩa như là một nhóm các mảng thập phân hai chiều có kích thước 20 x 30 liên tiếp nhau. Vì vậy, `fl_ary` trỏ đến mảng 20 x 30 đầu tiên, (fl_ary + 1) trỏ đến mảng 20 x 30 thứ hai,...

Trong mảng hai chiều **ary**, phần tử tại dòng 4 và cột 9 có thể được truy xuất sử dụng câu lệnh:

```
ary[3][8];
```

hoặc

```
*(*(ary + 3) + 8);
```

Cách thứ nhất là cách thường được dùng. Trong cách thứ hai, (ary + 3) là một con trỏ trỏ đến dòng thứ 4. Vì vậy, đối tượng của con trỏ này, *(ary + 3), tham chiếu đến toàn bộ dòng. Vì dòng 3 là một mảng một chiều, *(ary + 3) là một con trỏ trỏ đến phần tử đầu tiên trong dòng 3, sau đó 8 được cộng vào con trỏ. Vì vậy, *(*(ary + 3) + 8) là một con trỏ trỏ đến phần tử 8 (phần tử thứ 9) trong dòng thứ 4. Vì vậy đối tượng của con trỏ này, *(*(ary + 3) + 8), tham chiếu đến tham chiếu đến phần tử trong cột thứ 9 của dòng thứ 4, đó là ary [3][8].

Có nhiều cách thức để định nghĩa mảng, và có nhiều cách để xử lý các phần tử mảng. Lựa chọn cách thức nào tùy thuộc vào người dùng. Tuy nhiên, trong các ứng dụng có các mảng dạng số, định nghĩa mảng theo cách thông thường sẽ dễ dàng hơn.

✓ **Con trỏ và chuỗi**

Chuỗi đơn giản chỉ là một mảng một chiều có kiểu ký tự. Mảng và con trỏ có mối liên hệ mật thiết, và như vậy, một cách tự nhiên chuỗi cũng sẽ có mối liên hệ mật thiết với con trỏ. Xem trường hợp hàm **strchr().** Hàm này nhận các tham số là một chuỗi và một ký tự để tìm kiếm ký tự đó trong mảng, nghĩa là,

```
ptr_str = strchr(strl, 'a');
```

biến con trỏ `ptr_str` sẽ được gán địa chỉ của ký tự 'a' đầu tiên xuất hiện trong chuỗi `str`. Đây không phải là vị trí trong chuỗi, từ 0 đến cuối chuỗi, mà là địa chỉ, từ địa chỉ bắt đầu chuỗi đến địa chỉ kết thúc của chuỗi.

Chương trình sau sử dụng hàm strchr(), đây là chương trình cho phép người dùng nhập vào một chuỗi và một ký tự để tìm kiếm. Chương trình in ra địa chỉ bắt đầu của chuỗi, địa chỉ của ký tự, và vị trí tương đối của ký tự trong chuỗi (0 là vị trí của ký tự đầu tiên, 1 là vị trí của ký tự thứ hai,...). Vị trí tương đối này là hiệu số giữa hai địa chỉ, địa chỉ bắt đầu của chuỗi và địa chỉ nơi mà ký tự cần tìm đầu tiên xuất hiện.

```
#include <stdio.h>
```

```
#include <string.h>

void main ()
{
    char a, str[81], *ptr;
    printf("\nEnter a sentence:");
    gets(str);
    printf("\nEnter character to search for:");
    a = getche();
    ptr = strchr(str, a);

    /* return pointer to char*/
    printf("\nString starts at address: %u", str);
    printf("\nFirst occurrence of the character is at
address: %u", ptr);
    printf("\nPosition of first occurrence (starting
from 0)is: %d", ptr-str);
}
Enter a sentence: We all live in a yellow submarine
Enter character to search for: Y
String starts at address: 65420.
First occurrence of the character is at address:
65437.
Position of first occurrence (starting from 0) is: 17
```

Trong câu lệnh khai báo, biến con trỏ **ptr** được thiết đặt để chứa địa chỉ trả về từ hàm **strchr()**, vì vậy đây là một địa chỉ của một ký tự (**ptr** có kiểu **char**).

Hàm strchr() không cần thiết phải khai báo nếu thư viện **string.h** được khai báo.

✓ **Cấp phát bộ nhớ**

Cho đến thời điểm này thì chúng ta đã biết là tên của một mảng thật ra là một con trỏ trỏ tới phần tử đầu tiên của mảng. Hơn nữa, ngoài cách định nghĩa một mảng thông thường có thể định nghĩa một mảng như là một biến con trỏ. Tuy nhiên, nếu một mảng được khai báo một cách bình thường, kết quả là một khối bộ nhớ cố định được dành sẵn tại thời điểm bắt đầu thực thi chương trình, trong khi điều này không xảy ra nếu mảng được khai báo như là một biến con trỏ. Sử dụng một biến con trỏ để biểu diễn một mảng đòi hỏi việc gán một vài ô nhớ khởi tạo trước khi các phần tử mảng được xử lý. Sự cấp phát bộ nhớ như vậy thông thường được thực hiện bằng cách sử dụng hàm thư viện **malloc()**.

Xem ví dụ sau. Một mảng số nguyên một chiều **ary** có 20 phần tử có thể được khai báo như sau:

```
int *ary;
```

thay vì

```
int ary[20];
```

Tuy nhiên, **ary** sẽ không được tự động gán một khối bộ nhớ khi nó được khai báo như là một biến con trỏ, trong khi một khối ô nhớ đủ để chứa 10 số nguyên sẽ được dành sẵn nếu **ary** được khai báo như là một mảng. Nếu **ary** được khai báo như là một con trỏ, số lượng bộ nhớ có thể được gán như sau:

```
 ary = malloc(20 *sizeof(int));
```

Sẽ dành một khối bộ nhớ có kích thước (tính theo bytes) tương đương với kích thước của một số nguyên. Ở đây, một khối bộ nhớ cho 20 số nguyên được cấp phát. 20 con số gán với 20 bytes (một byte cho một số nguyên) và được nhân với **sizeof(int)**, sizeof(int) sẽ trả về kết quả 2, nếu máy tính dùng 2 bytes để lưu trữ một số nguyên. Nếu một máy tính sử dụng 1 byte để lưu một số nguyên, hàm sizeof() không đòi hỏi ở đây. Tuy nhiên, sử dụng nó sẽ tạo khả năng uyển chuyển cho mã lệnh. Hàm malloc() trả về một con trỏ chứa địa chỉ vị trí bắt đầu của vùng nhớ được cấp phát. Nếu không gian bộ nhớ yêu cầu không có, malloc() trả về giá trị NULL. Sự cấp phát bộ nhớ theo cách này, nghĩa là, **khi được yêu cầu trong một chương trình** được gọi là **Cấp phát bộ nhớ động**.

Trước khi tiếp tục xa hơn, chúng ta hãy thảo luận về khái niêm **Cấp phát bộ nhớ** động. Một chương trình C có thể lưu trữ các thông tin trong bộ nhớ của máy tính theo hai cách chính. Phương pháp thứ nhất bao gồm các biến toàn cục và cục bộ – bao gồm các mảng. Trong trường hợp các biến toàn cục và biến tĩnh, sự lưu trữ là cố định suốt thời gian thực thi chương trình. Các biến này đòi hỏi người lập trình phải biết trước tổng số dung lượng bộ nhớ cần thiết cho mỗi trường hợp. Phương pháp thứ hai, thông tin có thể được lưu trữ thông qua **Hệ thống cấp phát động** của C**.** Trong phương pháp này, sự lưu trữ thông tin được cấp phát từ vùng nhớ còn tự do và khi cần thiết.

Hàm **malloc()** là một trong các hàm thường được dùng nhất, nó cho phép thực hiện việc cấp phát bộ nhớ từ vùng nhớ còn tự do. Tham số cho **malloc()** là một số nguyên xác định số bytes cần thiết.

Một ví dụ khác, xét mảng ký tự hai chiều **ch_ary** có 10 dòng và 20 cột. Sự khai báo và cấp phát bộ nhớ trong trường hợp này phải như sau:

```
char (*ch_ary)[20];
ch_ary = (char*)malloc(10*20*sizeof(char));
```

Như đã nói ở trên, **malloc()** trả về một con trỏ trỏ đến kiểu rỗng (void). Tuy nhiên, vì **ch_ary** là một con trỏ kiểu char, sự chuyển đổi kiểu là cần thiết. Trong câu lệnh trên, (char*) đổi kiểu trả về của **malloc()** thành một con trỏ trỏ đến kiểu char.

Tuy nhiên, nếu sự khai báo của mảng phải chứa phép gán các giá trị khởi tạo thì mảng phải được khai báo theo cách bình thường, không thể dùng một biến con trỏ:

```
int ary[10] = {1,2,3,4,5,6,7,8,9,10};
```

hoặc

```
int ary[] = {1,2,3,4,5,6,7,8,9,10};
```

Ví dụ sau đây tạo một mảng một chiều và sắp xếp mảng theo thứ tự tăng dần. Chương trình sử dụng con trỏ và hàm malloc() để gán bộ nhớ.

```
#include<stdio.h>
#include<malloc.h>
void main()
{
int *p, n, i, j, temp;
printf("\n Enter number of elements in the array: ");
scanf("%d", &n);
p = (int*) malloc(n * sizeof(int));
for(i = 0; i < n; ++i)
{
printf("\nEnter element no. %d:", i + 1);
scanf("%d", p + i);
}
for(i = 0; i < n - 1; ++i)
for(j = i + 1; j < n; ++j)
      if(*(p + i) > *(p + j))
{
           temp = *(p + i);
           *(p + i) = *(p + j);
           *(p + j) = temp;
      }
for(i = 0; i < n; ++i)
      printf("%d\n", *(p + i));
}
```

Chú ý lệnh **malloc()**:

```
p = (int*)malloc(n*sizeof(int));
```

Ở đây, p được khai báo như một con trỏ trỏ đến một mảng và được gán bộ nhớ sử dụng malloc().

Dữ liệu được đọc sử dụng scanf().

```
scanf("%d",p+i);
```

Trong scanf(), biến con trỏ được sử dụng để lưu dữ liệu vào trong mảng.

Các phần tử mảng đã lưu trữ được hiển thị bằng printf().

```
printf("%d\n", *(p + i));
```

Chú ý dấu * trong trường hợp này, vì giá trị lưu trong vị trí đó phải được hiển thị. Không có dấu *, printf() sẽ hiển thị địa chỉ.

- **free()**

Hàm này có thể được sử dụng để giải phóng bộ nhớ khi nó không còn cần thiết.

Dạng tổng quát của hàm free():

```
void  free(    void *ptr );
```

Hàm free() giải phóng không gian được trỏ bởi *ptr*, không gian được giải phóng này có thể sử dụng trong tương lai. *ptr* đã sử dụng trước đó bằng cách gọi đến malloc(), calloc(), hoặc realloc(), calloc() và realloc() (sẽ được thảo luận sau).

Ví dụ bên dưới sẽ hỏi bạn có bao nhiêu số nguyên sẽ được bạn lưu vào trong một mảng. Sau đó sẽ cấp phát bộ nhớ động bằng cách sử dụng **malloc** và lưu số lượng số nguyên, in chúng ra, và sau đó xóa bộ nhớ cấp phát bằng cách sử dụng **free**.

```
#include <stdio.h>
#include <stdlib.h> /* required for the malloc and
free functions */
int main()
{
   int number;
   int *ptr;
   int i;
   printf("How many ints would you like store? ");
   scanf("%d", &number);
   ptr = (int *) malloc (number * sizeof(int));
/*allocate memory*/
   if(ptr != NULL)
```

```
        {
                    for(i = 0 ; i < number ; i++)
                    {
                                *(ptr+i) = i;
                    }
                    for(i=number ; i>0 ; i--)
                    {
                                printf("%d\n",      *(ptr+(i-
1))); /*print out in reverse order*/
                    }
                    free(ptr); /* free allocated memory */
                    return 0;
        }
        else
        {
                    printf("\nMemory allocation failed - not
enough memory.\n");
                    return 1;
        }
    }
```

Kết quả như sau nếu giá trị được nhập vào **3**:

```
How many ints would you like store? 3
2
1
0
```

- **calloc()**

calloc tương tự như **malloc**, nhưng khác biệt chính là mặc nhiên các giá trị được lưu trong không gian bộ nhớ đã cấp phát là 0. Với **malloc**, cấp phát bộ nhớ có thể có giá trị bất kỳ.

calloc đòi hỏi hai đối số. Đối số thứ nhất là số các biến mà bạn muốn cấp phát bộ nhớ cho. Đối số thứ hai là kích thước của mỗi biến.

```
void *calloc( size_t num, size_t size );
```

Giống như **malloc, calloc** sẽ trả về một con trỏ rỗng (void) nếu sự cấp phát bộ nhớ là thành công, ngược lại nó sẽ trả về một con trỏ NULL.

Ví dụ bên dưới chỉ ra cho bạn gọi hàm **calloc** như thế nào và tham chiếu đến ô nhớ đã cấp phát sử dụng một chỉ số mảng. Giá trị khởi tạo của vùng nhớ đã cấp phát được in ra trong vòng lặp for.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  float *calloc1, *calloc2;
  int i;
  calloc1 = (float *) calloc(3, sizeof(float));
  calloc2 = (float *) calloc(3, sizeof(float));
  if(calloc1 != NULL && calloc2 != NULL)
  {
     for(i = 0; i < 3; i++)
     {
                    printf("\ncalloc1[%d] holds %05.5f
", i, calloc1[i]);
                    printf("\ncalloc2[%d]        holds
%05.5f", i, *(calloc2 + i));
     }

     free(calloc1);
     free(calloc2);
     return 0;
  }
  else
  {
     printf("Not enough memory\n");
     return 1;
  }
}
```

Kết quả:

```
calloc1[0] holds 0.00000
calloc2[0] holds 0.00000
calloc1[1] holds 0.00000
calloc2[1] holds 0.00000
calloc1[2] holds 0.00000
calloc2[2] holds 0.00000
```

Trong tất cả các máy, các mảng calloc1 và calloc2 phải chứa các giá trị 0. **calloc** đặc biệt hữu dụng khi bạn đang sử dụng mảng đa chiều. Đây là một ví dụ khác minh họa cách dùng của hàm calloc().

```
/* This program gets the number of elements, allocates
   spaces for the elements, gets a value for each
   element, sum the values of the elements, and print
   the number of the elements and the sum.
 */
#include <stdio.h>
#include <stdlib.h>
main()
{
 int *a, i, n, sum = 0;
 printf("\n%s%s", "An array will be created
dynamically. \n\n",
             "Input an array size n followed by integers:
");
 scanf( "%d", &n); /* get the number of elements */
 a = (int *) calloc (n, sizeof(int)); /* allocate
space */
 /* get a value for each element */
 for( i = 0; i < n; i++ )
 {
          printf("Enter %d values: ", n);
               scanf("%d", a + i);
 }
 /* sum the values */
 for(i = 0; i < n; i++ )
 sum += a[i];
 free(a); /* free the space */
 /* print the number and the sum */
 printf("\n%s%7d\n%s%7d\n\n", "Number of elements: ",
n,
                              "Sum of the elements: ",
sum);
}
```

- **realloc()**

Giả sử chúng ta đã cấp phát một số bytes cho một mảng nhưng sau đó nhận ra là bạn muốn thêm các giá trị. Bạn có thể sao chép mọi thứ vào một mảng lớn hơn, cách này không hiệu quả. Hoặc bạn có thể cấp phát thêm các bytes sử dụng bằng cách gọi hàm **realloc**, mà dữ liệu của bạn không bị mất đi.

realloc() nhận hai đối số. Đối số thứ nhất là một con trỏ tham chiếu đến bộ nhớ. Đối số thứ hai là tổng số bytes bạn muốn cấp phát thêm.

```
void *realloc( void *ptr, size_t size );
```

Truyền 0 như là đối số thứ hai thì tương đương với việc gọi hàm **free**.

Một lần, **realloc** trả về một con trỏ rỗng (void) nếu thành công, ngược lại một con trỏ NULL được trả về.

Ví dụ này sử dụng **calloc** để cấp phát đủ bộ nhớ cho một mảng int có năm phần tử. Sau đó **realloc** được gọi để mở rộng mảng để có thể chứa bảy phần tử.

```
#include<stdio.h>
#include <stdlib.h>
int main()
{
  int *ptr;
  int i;
  ptr = (int *)calloc(5, sizeof(int *));
  if(ptr!=NULL)
  {
        *ptr = 1;
        *(ptr + 1) = 2;
     ptr[2] = 4;
     ptr[3] = 8;
     ptr[4] = 16;
     /* ptr[5] = 32; wouldn't assign anything */
     ptr = (int *)realloc(ptr, 7 * sizeof(int));
     if(ptr!=NULL)
     {
        printf("Now allocating more memory... \n");
        ptr[5] = 32; /* now it's legal! */
        ptr[6] = 64;
        for(i = 0;i < 7; i++)
        {
```

```
            printf("ptr[%d] holds %d\n", i, ptr[i]);
            }
            realloc(ptr, 0); /* same as free(ptr); - just
fancier! */
            return 0;
         }
         else
         {
            printf("Not enough memory - realloc
failed.\n");
            return 1;
         }
      }
      else
      {
         printf("Not enough memory - calloc failed.\n");
         return 1;
      }
   }
```

Kết quả:

```
   Now          allocating          more          memory...
ptr[0]                        holds                        1
ptr[1]                        holds                        2
ptr[2]                        holds                        4
ptr[3]                        holds                        8
ptr[4]                       holds                        16
ptr[5]                       holds                        32
ptr[6] holds 64
```

Chú ý hai cách khác nhau được sử dụng khi khởi tạo mảng: ptr[2] = 4 là tương đương với *(ptr + 2) = 4 (chỉ dễ đọc hơn!).

Trước khi sử dụng realloc, việc gán một giá trị đến phần tử ptr[5] không gây ra lỗi cho trình biên dịch. Chương trình vẫn thực thi, nhưng ptr[5] không chứa giá trị mà bạn đã gán.

5.2.4. Quy trình kỹ thuật trong bài học của GV và SV:

*** Quy trình thị phạm của GV**

- Bước 1: Chuẩn bị bài giảng, tài liệu và điều kiện trang thiết bị giảng dạy
- Bước 2: Chuẩn bị bài tập mẫu và đưa ra nội dung thực hành
- Bước 3: Phổ biến các quy trình, các bước thực hiện
- Bước 4: Hướng dẫn nội dung bài học, Hướng dẫn thị phạm mẫu
- Bước 5: Quản lý lớp học
- Bước 6: Giải đáp, tư vấn, đánh giá, nhận xét
- Bước 7: Chấm điểm và nộp các bài đánh giá cho bộ môn

*** Quy trình thực hiện bài của SV**

- Bước 1: SV chuẩn bị tài liệu, vở, bút, giấy nháp, tài liệu tra cứu
- Bước 2: Đọc tài liệu Lập trình căn bản C phần con trỏ
- Bước 3: Sinh viên trình bày bài thực hành, thảo luận và nhận xét góp ý.

5.2.5. Các mẫu hình sản phẩm cho SV tham khảo:

- SV xem video

5.2.6. Phần tự thực hành thao tác thường xuyên của SV trong bài học:

Câu 1: Viết một chương trình để nhận vào một chuỗi và in ra nó nếu đó là chuỗi đọc xuôi – ngược đều giống nhau.

Câu 2: Viết một chương trình sử dụng con trỏ trỏ đến các chuỗi để nhận tên của một con thú và một con chim và trả về các tên theo dạng số nhiều.

5.2.7. Sản phẩm thực hành:

+ Các ví dụ demo.

5.2.8. Điều kiện để GV- SV thực hiện bài học thực hành;

5.2.8.1. Điều kiện chuẩn bị trước bài học

+ Giảng viên:

- Đề cương chi tiết bài giảng
- Thiết kế bài giảng
- Học liệu tham khảo, mở rộng

+ Sinh viên:

- Đọc kỹ bài học trước khi giảng viên lên lớp
- Chuẩn bị vấn đề thảo luận, trao đổi với giảng viên
- Các tài liệu tham khảo, mở rộng do giảng viên yêu cầu

5.2.8.2. Kỹ thuật và phương tiện dạy học:

- Bảng, phấn viết
- Màn hình lớn
- Máy tính, máy chiếu

5.2.8.3. Phương pháp tổ chức dạy-học:

- Thuyết trình, diễn giảng, thực hành
- Phát vấn
- Trao đổi, thảo luận
- Cemina
- Vấn đáp

5.3 Nội dung bài giảng 2

5.3.1 Tên bài giảng: BÀI 2: QUẢN LÝ TỆP TIN

Số tiết lên lớp của GV: 08 tiết; Số tiết tự làm bài của SV : 08 tiết

5.3.2. Phần mở đầu tiếp cận bài;

Hầu hết các chương trình đều yêu cầu đọc và ghi dữ liệu vào các hệ thống lưu trữ trên đĩa. Các chương trình xử lý văn bản cần lưu các tập tin văn bản, chương trình xử lý bảng tính cần lưu nội dung của các ô, chương trình cơ sỡ dữ liệu cần lưu các mẫu tin. Bài này sẽ khám phá các tiện ích trong C dành cho các thao tác nhập/xuất (I/O) đĩa hệ thống.

Ngôn ngữ C không chứa bất kỳ câu lệnh nhập/xuất nào một cách tường minh. Tất cả các thao tác nhập/xuất đều thực hiện thông qua các hàm thư viện chuẩn của C. Tiếp cận này làm cho hệ thống quản lý tập tin của C rất mạnh và uyển chuyển. Nhập/xuất trong C là tuyệt vời vì dữ liệu có thể truyền ở dạng nhị phân hay ở dạng văn bản mà con người có thể đọc được. Điều này làm cho việc tạo tập tin để đáp ứng mọi nhu cầu một cách dễ dàng.

Việc hiểu rõ sự khác biệt giữa stream và tập tin là rất quan trọng. Hệ thống nhập/xuất của C cung cấp cho người dùng một giao diện độc lập với thiết bị thật sự đang truy cập. Giao diện này không phải là một tập tin thật sự mà là một sự biểu diễn trừu tượng của thiết bị. Giao diện trừu tượng này được gọi là một stream và thiết bị thật sự được gọi là **tập tin**.

5.3.3. Phần kiến thức, kỹ thuật căn bản:

5.3.3.1. Phần kiến thức căn bản

- **Phần kiến thức**

Mục tiêu:

Kết thúc bài học này, bạn có thể:

- Giải thích khái niệm luồng (streams) và tập tin (files)
- Thảo luận các luồng văn bản và các luồng nhị phân
- Giải thích các hàm xử lý tập tin
- Giải thích con trỏ tập tin
- Thảo luận con trỏ kích hoạt hiện hành
- Giải thích các đối số từ dòng nhắc lệnh (command-line).

- **File Streams**

Hệ thống tập tin của C làm việc được với rất nhiều thiết bị khác nhau bao gồm máy in, ổ đĩa, ổ băng từ và các thiết bị đầu cuối. Mặc dù tất cả các thiết bị đều khác nhau, nhưng hệ thống tập tin có vùng đệm sẽ chuyển mỗi thiết bị về một thiết bị logic gọi là một stream. Vì mọi **streams** hoạt động tương tự, nên việc quản lý các thiết bị là rất dễ dàng. Có hai loại streams – **văn bản** (text) và **nhị phân** (binary).

✓ **Streams văn bản**

Một **streams văn bản** là một chuỗi các ký tự. Các streams văn bản có thể được tổ chức thành các dòng, mỗi dòng kết thúc bằng một ký tự sang dòng mới. Tuy nhiên, ký tự sang dòng mới là tùy chọn trong dòng cuối và được quyết định khi cài đặt. Hầu hết các trình biên dịch C không kết thúc stream văn bản với ký tự sang dòng mới. Trong một stream văn bản, có thể xảy ra một vài sự chuyển đổi ký tự khi môi trường yêu cầu. Chẳng hạn như, ký tự sang dòng mới có thể được chuyển thành một cặp ký tự về đầu dòng/nhảy đến dòng kế. Vì vậy, mối quan hệ giữa các ký tự được ghi (hay đọc) và những ký tự ở thiết bị ngoại vi có thể không phải là mối quan hệ một-một. Và cũng vì sự chuyển đổi có thể xảy ra này, số lượng ký tự được ghi (hay đọc) có thể không giống như số lượng ký tự nhìn thấy ở thiết bị ngoại vi.

✓ **Streams nhị phân**

Một **streams nhị phân** là một chuỗi các byte với sự tương ứng một-một với thiết bị ngoại vi, nghĩa là, không có sự chuyển đổi ký tự. Cũng vì vậy, số lượng byte đọc (hay ghi) cũng sẽ giống như số lượng byte ở thiết bị ngoại vi. Các stream nhị phân là các chuỗi byte thuần túy, mà không có bất kỳ ký hiệu nào được dùng để chỉ ra điểm kết thúc của tập tin hay kết thúc của record. Kết thúc của tập tin được xác định bằng độ lớn của tập tin.

o **Các hàm về tập tin và structure FILE**

Một tập tin có thể tham chiếu đến bất cứ cái gì: từ một tập tin trên đĩa đến một thiết bị đầu cuối hay một máy in. Tuy nhiên, tất cả các tập tin đều không có cùng khả năng. Ví dụ như, một tập tin trên đĩa có thể hỗ trợ truy cập ngẫu nhiên trong khi một bàn phím thì không. Một tập tin sẽ kết hợp với một stream bằng cách thực hiện thao tác mở. Tương tự, nó sẽ thôi kết hợp với một stream bằng thao tác đóng. Khi một chương trình kết thúc bình thường, tất cả các tập tin đều tự động đóng. Tuy nhiên, khi một chương trình bị treo hoặc kết thúc bất thường, các tập tin vẫn còn mở.

✓ **Các hàm cơ bản về tập tin**

Một hệ thống quản lý tập tin theo chuẩn ANSI bao gồm một số hàm liên quan với nhau. Các hàm thông dụng nhất được liệt kê trong bảng 21.1.

Name	**Function**
fopen()	Mở một tập tin
fclose()	Đóng một tập tin
fpu	Ghi một ký tự vào một tập tin

tc()	
fgetc()	Đọc một ký tự từ một tập tin
fread()	Đọc từ một tập tin vào một vùng đệm
fwrite()	Ghi từ một vùng đệm vào tập tin
fseek()	Tìm một vị trí nào đó trong tập tin
fprintf()	Hoạt động giống như printf(), nhưng trên một tập tin
fscanf()	Hoạt động giống như scanf(), nhưng trên một tập tin
feof()	Trả về true nếu đã đến cuối tập tin (end-of-file)
ferror()	Trả về true nếu xảy ra một lỗi
rewind()	Đặt lại con trỏ định vị trí (position locator) bên trong tập tin về đầu tập tin
remove()	Xóa một tập tin
fflush()	Ghi dữ liệu từ một vùng đệm bên trong vào một tập tin xác định

Các hàm trên chứa trong tập tin header **stdio.h**. Tập tin header này phải được bao gồm vào chương trình có sử dụng các hàm này. Hầu hết các hàm này tương tự như các hàm nhập/xuất từ thiết bị nhập xuất chuẩn. Tập tin header **stdio.h** còn định nghĩa một số macro sử dụng trong quá trình xử lý tập tin. Ví dụ như, macro EOF được định nghĩa là -1, chứa giá trị trả về khi một hàm cố đọc tiếp khi đã đến cuối tập tin.

✓ **Con trỏ tập tin**

Một con trỏ tập tin (file pointer) rất cần thiết cho việc đọc và ghi các tập tin. Nó là một con trỏ đến một structure chứa thông tin về tập tin. Thông tin bao gồm: tên tập tin, vị trí hiện tại của tập tin, tập tin đang được đọc hay ghi, có bất kỳ lỗi nào xuất hiện hay đã đến cuối tập tin. Người dùng không cần thiết phải biết chi tiết, vì các định nghĩa lấy từ studio.h có bao gồm một khai báo structure tên là **FILE**. Câu lệnh khai báo duy nhất cần thiết cho một con trỏ tập tin là:

```
FILE *fp;
```

Khai báo này cho biết **fp** là một con trỏ trỏ đến một FILE.

- **Các tập tin văn bản**

Có nhiều hàm khác nhau để quản lý tập tin văn bản. Chúng ta sẽ thảo luận trong các đoạn bên dưới:

✓ **Mở một tập tin văn bản**

Hàm **fopen()** mở một stream để sử dụng và liên kết một tập tin với stream đó. Con trỏ kết hợp với tập tin được trả về từ hàm **fopen()**. Trong hầu hết các trường hợp, tập tin đang mở là một tập tin trên đĩa. Nguyên mẫu của hàm **fopen()** là:

```
FILE *fopen(const char *filename, const char  *mode);
```

trong đó **filename** là một con trỏ trỏ đến chuỗi ký tự chứa một tên tập tin hợp lệ và cũng có thể chứa cả phần mô tả đường dẫn. Chuỗi được trỏ đến bởi con trỏ **mode** xác định cách thức tập tin được mở. Bảng 21.2 liệt kê các chế độ hợp lệ mà một tập tin có thể mở.

Chế độ	**Ý nghĩa**
r	Mở một tập tin văn bản để đọc
w	Tạo một tập tin văn bản để ghi
a	Nối vào một tập tin văn bản
r+	Mở một tập tin văn bản để đọc/ghi
w+	Tạo một tập tin văn bản để đọc/ghi
a+f	Nối hoặc tạo một tập tin văn bản để đọc/ghi

Bảng 21.2 cho thấy các tập tin có thể được mở ở nhiều chế độ khác nhau. Một con trỏ null được trả về nếu xảy ra lỗi khi hàm fopen() mở tập tin. Lưu ý rằng các chuỗi như "a+f" có thể được biểu diễn như "af+".

Nếu phải mở một tập tin **xyz** để ghi, câu lệnh sẽ là:

```
FILE *fp;
fp =  fopen ("xyz", "w");
```

Tuy nhiên, một tập tin nói chung được mở bằng cách sử dụng một tập hợp các câu lệnh tương tự như sau:

```
FILE *fp;
if ((fp = fopen ("xyz", "w")) == NULL)
     {
     printf("Cannot open file");
     exit (1);
}
```

Macro **NULL** được định nghĩa trong stdio.h là '**\0**'. Nếu sử dụng phương pháp trên để mở một tập tin, thì hàm **fopen()** sẽ phát hiện ra lỗi nếu có, chẳng hạn như đĩa đang ở chế độ cấm ghi (write-protected) hay đĩa đầy, trước khi bắt đầu ghi đĩa.

Nếu một tập tin được mở để ghi, bất kỳ một tập tin nào có cùng tên và đang mở sẽ bị viết chồng lên. Vì khi một tập tin được mở ở chế độ ghi, thì một tập tin mới được tạo ra. Nếu muốn nối thêm các mẫu tin vào tập tin đã có, thì nó phải được mở với chế độ "a". Nếu một tập tin được mở ở chế độ đọc và nó không tồn tại, hàm sẽ trả về lỗi. Nếu một tập tin được mở để đọc/ghi, nó sẽ không bị xóa nếu đã tồn tại. Tuy nhiên, nếu nó không tồn tại, thì nó sẽ được tạo ra.

Theo chuẩn ANSI, tám tập tin có thể được mở tại một thời điểm. Tuy vậy, hầu hết các trình biên dịch C và môi trường đều cho phép mở nhiều hơn tám tập tin.

✓ **Đóng một tập tin văn bản**

Vì số lượng tập tin có thể mở tại một thời điểm bị giới hạn, việc đóng một tập tin khi không còn sử dụng là một điều quan trọng. Thao tác này sẽ giải phóng tài nguyên và làm giảm nguy cơ vượt quá giới hạn đã định. Đóng một stream cũng sẽ làm sạch và chép vùng đệm kết hợp của nó ra ngoài (một thao tác quan trọng để tránh mất dữ liệu) khi ghi ra đĩa. Hàm **fclose()** đóng một stream đã được mở bằng hàm **fopen()**. Nó ghi bất kỳ dữ liệu nào còn lại trong vùng đệm của đĩa vào tập tin. Nguyên mẫu của hàm **fclose()** là:

```
int  fclose(FILE *fp);
```

trong đó fp là một con trỏ tập tin. Hàm **fclose()** trả về **0** nếu đóng thành công. Bất kỳ giá trị trả về nào khác 0 đều cho thấy có lỗi xảy ra. Hàm **fclose()** sẽ thất bại nếu đĩa đã sớm được gỡ ra khỏi ổ đĩa hoặc đĩa bị đầy.

Một hàm khác dùng để đóng stream là hàm **fcloseall()**. Hàm này hữu dụng khi phải đóng cùng một lúc nhiều stream đang mở. Nó sẽ đóng tất cả các stream và trả về số stream đã đóng hoặc EOF nếu có phát hiện lỗi. Nó có thể được sử dụng theo cách như sau:

```
fcl = fcloseall();
if (fcl == EOF)
     printf("Error closing files");
else
     printf("%d file(s) closed", fcl);
```

✓ **Ghi một ký tự**

Streams có thể được ghi vào tập tin theo từng ký tự một hoặc theo từng chuỗi. Trước hết chúng ta hãy thảo luận về cách ghi các ký tự vào tập tin. Hàm **fputc()** được sử dụng để ghi các ký tự vào tập tin đã được mở trước đó bằng hàm **fopen()**. Nguyên mẫu của hàm này như sau:

```
int fputc(int ch, FILE *fp);
```

trong đó **fp** là một con trỏ tập tin trả về bởi hàm **fopen()** và **ch** là ký tự cần ghi. Mặc dù **ch** được khai báo là kiểu **int**, nhưng nó được hàm **fputc()** chuyển đổi thành kiểu **unsigned char**. Hàm **fputc()** ghi một ký tự vào stream đã định tại vị trí hiện hành của con trỏ định vị trí bên trong tập tin và sau đó tăng con trỏ này lên. Nếu **fputc()** thành công, nó trả về ký tự đã ghi, ngược lại nó trả về EOF.

Đọc một ký tự

Hàm **fgetc()** được dùng để đọc các ký tự từ một tập tin đã được mở ở chế độ đọc, sử dụng hàm **fopen()**. Nguyên mẫu của hàm là:

```
int fgetc (FILE *fp);
```

trong đó **fp** là một con trỏ tập tin kiểu FILE trả về bởi hàm **fopen()**. Hàm **fgetc()** trả về ký tự kế tiếp của vị trí hiện hành trong stream input, và tăng con trỏ định vị trí bên trong tập tin lên. Ký tự đọc được là một ký tự kiểu **unsigned char** và được chuyển thành kiểu **int**. Nếu đã đến cuối tập tin, **fgetc()** trả về EOF.

Để đọc một tập tin văn bản từ đầu cho đến cuối, câu lệnh sẽ là:

```
do
{
     ch = fgetc(fp);
}  while (ch != EOF);
```

Chương trình sau đây nhận các ký tự từ bàn phím và ghi chúng vào một tập tin cho đến khi người dùng nhập ký tự '@'. Sau khi người dùng nhập thông tin vào, chương trình sẽ hiển thị nội dung ra màn hình.

Ví dụ 1:

```
#include <stdio.h>
main()
{
 FILE *fp;
     char ch= ' ';

 /* Writing to file JAK  */
     if ((fp=fopen("jak", "w"))==NULL)
 {
      printf("Cannot open file \n\n");
      exit(1);
 }
 clrscr();
 printf("Enter characters (type @ to terminate): \n");
 ch = getche();
 while (ch !='@')
 {
      fputc(ch, fp) ;
      ch = getche();
 }
fclose(fp);

/* Reading from file JAK  */
 printf("\n\nDisplaying contents of file JAK\n\n");

 if((fp=fopen("jak", "r"))==NULL)
 {
      printf("Cannot open file\n\n");
      exit(1);
 }

 do
 {
```

```
        ch = fgetc (fp);
        putchar(ch) ;
    }  while (ch!=EOF);

    getch();
    fclose(fp);
}
```

Một mẫu chạy cho chương trình trên là:

```
Enter Characters (type @ to terminate):
This is the first input to the File JAK@

Displaying Contents of File JAK

This is the first input to the File JAK
```

✓ **Nhập xuất chuỗi**

Ngoài fgetc() và fputc(), C còn hổ trợ các hàm fputs() và fgets() để ghi vào và đọc ra các chuỗi ký tự từ tập tin trên đĩa.

Nguyên mẫu cho hai hàm này như sau:

```
int fputs(const char *str, FILE *fp);
char *fgets(char *str, int length, FILE *fp);
```

Hàm **fputs()** làm việc giống như hàm fputc(), ngoại trừ là nó viết toàn bộ chuỗi vào stream. Nó trả về EOF nếu xảy ra lỗi.

Hàm **fgets()** đọc một chuỗi từ stream đã cho cho đến khi đọc được một ký tự sang dòng mới hoặc sau khi đã đọc được **length-1** ký tự. Nếu đọc được một ký tự sang dòng mới, ký tự này được xem như là một phần của chuỗi (không giống như hàm gets()). Chuỗi kết quả sẽ kết thúc bằng ký tự null. Hàm trả về một con trỏ trỏ đến chuỗi nếu thành công và null nếu xảy ra lỗi.

o **Các tập tin nhị phân**

Các hàm dùng để xử lý các tập tin nhị phân cũng giống như các hàm sử dụng để quản lý tập tin văn bản. Tuy nhiên, chế độ mở tập tin của hàm fopen() thì khác đi trong trường hợp các tập tin nhị phân.

✓ **Mở một tập tin nhị phân**

Bảng sau đây liệt kê các chế độ khác nhau của hàm fopen() trong trường hợp mở tập tin nhị phân.

Chế độ	Ý nghĩa
rb	Mở một tập tin nhị phân để đọc
wb	Tạo một tập tin nhị phân để ghi
ab	Nối vào một tập tin nhị phân
r+b	Mở một tập tin nhị phân để đọc/ghi
w+b	Tạo một tập tin nhị phân để đọc/ghi
a+b	Nối vào một tập tin nhị phân để đọc/ghi

Nếu một tập tin **xyz** được mở để ghi, câu lệnh sẽ là:

```
FILE *fp;
fp =  fopen ("xyz", "wb");
```

✓ **Đóng một tập tin nhị phân**

Ngoài tập tin văn bản, hàm fclose() cũng có thể được dùng để đóng một tập tin nhị phân. Nguyên mẫu của fclose như sau:

```
int fclose(FILE *fp);
```

trong đó fp là một con trỏ tập tin trỏ đến một tập tin đang mở.

✓ **Ghi một tập tin nhị phân**

Một số ứng dụng liên quan đến việc sử dụng các tập tin dữ liệu để lưu trữ các khối dữ liệu, trong đó mỗi khối bao gồm các byte liên tục. Mỗi khối nói chung sẽ biểu diễn một cấu trúc dữ liệu phức tạp hoặc một mảng.

Chẳng hạn như, một tập tin dữ liệu có thể bao gồm nhiều cấu trúc có cùng thành phần cấu tạo, hoặc nó có thể chứa nhiều mảng có cùng kiểu và kích thước. Và với những ứng dụng như vậy thường đòi hỏi đọc toàn bộ khối dữ liệu từ tập tin dữ liệu hoặc ghi toàn bộ khối vào tập tin dữ liệu hơn là đọc hay ghi các thành phần độc lập

(nghĩa là các thành viên của cấu trúc hay các phần tử của mảng) trong mỗi khối riêng biệt.

Hàm **fwrite()** được dùng để ghi dữ liệu vào tập tin dữ liệu trong những tình huống như vậy. Hàm này có thể dùng để ghi bất kỳ kiểu dữ liệu nào. Nguyên mẫu của fwrite() là:

```
    size_t fwrite(const void *buffer, size_t num_bytes,
size_t
                count, FILE *fp);
```

Kiểu dữ liệu **size_t** được thêm vào C chuẩn để tăng tính tương thích của chương trình với nhiều hệ thống. Nó được định nghĩa trước như là một kiểu số nguyên đủ lớn để lưu giữ kết quả của hàm **sizeof()**. Đối với hầu hết các hệ thống, nó có thể được dùng như một số nguyên dương..

Buffer là một con trỏ trỏ đến thông tin sẽ được ghi vào tập tin. Số byte phải đọc hoặc ghi được cho bởi **num_bytes**. Đối số **count** xác định có bao nhiêu mục (mỗi mục dài **num_bytes**) được đọc hoặc ghi. Cuối cùng, **fp** là một con trỏ tập tin trỏ đến một stream đã được mở trước đó. Các tập tin mở cho những thao tác này phải mở ở chế độ nhị phân.

Hàm này trả về số lượng các đối tượng đã ghi vào tập tin nếu thao tác ghi thành công. Nếu giá trị này nhỏ hơn *count* thì đã xảy ra lỗi. Hàm ferror() (sẽ được thảo luận trong phần tới) có thể được dùng để xác định lỗi.

✓ **Đọc một tập tin nhị phân**

Hàm **fread()** có thể được dùng để đọc bất kỳ kiểu dữ liệu nào. Nguyên mẫu của hàm là:

```
    size_t fread(void *buffer, size_t num_bytes, size_t
count
                FILE *fp);
```

buffer là một con trỏ trỏ đến vùng nhớ sẽ nhận dữ liệu từ tập tin. Số byte phải đọc hoặc ghi được cho bởi **num_bytes**. Đối số count xác định có bao nhiêu mục (mỗi mục dài **num_bytes**) được đọc hoặc ghi. Cuối cùng, **fp** là một con trỏ tập tin trỏ đến một stream đã được mở trước đó. Các tập tin đã mở cho những thao tác này phải mở ở chế độ nhị phân.

Hàm này trả về số lượng các đối tượng đã đọc nếu thao tác đọc thành công. Nó trả về 0 nếu đọc đến cuối tập tin hoặc xảy ra lỗi. Hàm feof() và hàm ferror() (sẽ được thảo luận trong phần tới) có thể được dùng để xác định nguyên nhân.

Các hàm fread() và fwrite() thường được gọi là các hàm **đọc hoặc ghi không định dạng**.

Miễn là tập tin được mở cho các thao tác nhị phân, hàm **fread()** và **fwrite()** có thể đọc và ghi bất kỳ kiểu thông tin nào. Ví dụ, chương trình sau đây ghi vào và sau đó đọc ngược ra một số kiểu double, một số kiểu int và một số kiểu long từ tập tin trên đĩa. Lưu ý rằng nó sử dụng hàm **sizeof()** để xác định độ dài của mỗi kiểu dữ liệu.

Ví dụ 2:

```
#include <stdio.h>
main ()
{
FILE *fp;
double d = 23.31 ;
int  i = 13;
long li = 1234567L;

clrscr();
if ((fp = fopen ("jak", "wb+")) == NULL )
{
printf("Cannot open file ");
exit(1);
}
fwrite (&d, sizeof(double), 1, fp);
fwrite (&i, sizeof(int), 1, fp);
fwrite (&li, sizeof(long), 1,fp);
fclose (fp);
if ((fp = fopen ("jak", "rb+")) == NULL )
{
printf("Cannot open file");
exit(1);
}
fread (&d, sizeof(double), 1, fp);
fread(&i, sizeof(int), 1, fp);
fread (&li, sizeof(long), 1, fp);

printf ("%f %d %ld", d, i, li);
fclose (fp);
}
```

Như chương trình này minh họa, có thể đọc buffer và thường nó chỉ là một vùng nhớ để giữ một biến. Trong chương trình đơn giản trên, giá trị trả về của hàm **fread()** và **fwrite()** được bỏ qua. Tuy nhiên, để lập trình hiệu quả, các giá trị đó nên được kiểm tra xem đã có lỗi xảy ra không.

Một trong những ứng dụng hữu dụng nhất của **fread()** và **fwrite()** liên quan đến việc đọc và ghi các kiểu dữ liệu do người dùng định nghĩa, đặc biệt là các cấu trúc. Ví dụ ta có cấu trúc sau:

```
 struct struct_type
{
      float balance;
 char name[80];
} cust;
```

Câu lệnh sau đây ghi nội dung của **cust** vào tập tin đang được trỏ đến bởi **fp**.

```
fwrite(&cust, sizeof(struct struct_type), 1, fp);
```

- **Các hàm xử lý tập tin**

Các hàm xử lý tập tin khác được thảo luận trong phần này.

- ✓ **Hàm feof()**

Khi một tập tin được mở để đọc ở dạng nhị phân, một số nguyên có giá trị tương đương với EOF có thể được đọc. Trong trường hợp này, quá trình đọc sẽ cho rằng đã đến cuối tập tin, mặc dù chưa đến cuối tập tin thực sự. Một hàm **feof()** có thể được dùng những trong trường hợp này. Nguyên mẫu của hàm là:

```
 int feof(FILE *fp );
```

Nó trả về true nếu đã đến cuối tập tin, nếu không nó trả về false (0). Hàm này được dùng trong khi đọc dữ liệu nhị phân.

Đoạn lệnh sau đây đọc một tập tin nhị phân cho đến cuối tập tin.

```
.
.
while (!feof(fp) )
 ch = fgetc(fp);
.
.
```

✓ **Hàm rewind()**

Hàm **rewind()** đặt lại con trỏ định vị trí bên trong tập tin về đầu tập tin. Nó lấy con trỏ tập tin làm đối số. Cú pháp của **rewind()** là:

```
rewind(fp);
```

Chương trình sau mở một tập tin ở chế độ đọc/ghi, sử dụng hàm fputs() với đầu vào là các chuỗi, đưa con trỏ quay về đầu tập tin và sau đó hiển thị các chuỗi giống như vậy bằng hàm fgets().

Ví dụ 3:

```
#include <stdio.h>
main()
{
 FILE *fp;
 char str [80];

 /* Writing to File JAK */

 if ((fp = fopen("jak", "w+")) == NULL)
 {
      printf ("Cannot open file \n\n");
      exit(1);
 }
 clrscr ();
 do
 {
      printf ("Enter a string (CR to quit): \n");
      gets (str);
      if(*str  != '\n')
      {    strcat (str, "\n"); /* add a new line */
           fputs (str, fp);
      }
      } while (*str  != '\n');

 /*Reading from File JAK */

 printf ("\n\n Displaying Contents of File JAK\n\n");
```

```
 rewind (fp);

 while (!feof(fp))
 {
      fgets (str, 81, fp);
      printf ("\n%s", str);
 }
 fclose(fp);
}
```

Một mẫu chạy chương trình trên như sau:

```
Enter a string (CR to quit):
This is input line 1

Enter a string (CR to quit) :
This is input line 2

Enter a string (CR to quit):
This is input line 3

Enter a string (CR to quit):

Displaying Contents of File JAK

This is input line 1

This is input line 2

This is input line 3
```

✓ **Hàm ferror()**

Hàm **ferror()** xác định liệu một thao tác trên tập tin có sinh ra lỗi hay không. Nguyên mẫu của hàm là:

```
 int ferror(FILE * fp) ;
```

trong đó **fp** là một con trỏ tập tin hợp lệ. Nó trả về true nếu có xảy ra một lỗi trong thao tác cuối cùng trên tập tin ; ngược lại, nó trả về false. Vì mỗi thao tác thiết

lập lại tình trạng lỗi, nên hàm **ferror()** phải được gọi ngay sau mỗi thao tác; nếu không, lỗi sẽ bị mất.

Chương trình trước có thể được sửa đổi để kiểm tra và cảnh báo về bất kỳ lỗi nào trong khi ghi như sau:

```
.
.
do
{
      printf(" Enter a string (CR to quit): \n");
      gets(str);
 if(*str  != '\n')
 {    strcat (str, "\n"); /* add a new line */
      fputs (str, fp);
 }
      if(ferror(fp))
           printf("\nERROR in writing\n");
  } while(*str!='\n');
  .
  .
```

✓ **Xóa tập tin**

Hàm **remove()** xóa một tập tin đã định. Nguyên mẫu của hàm là:

```
int remove (char *filename);
```

Nó trả về **0** nếu thành công ngược lại trả về một giá trị khác 0.

Ví dụ, xét đoạn mã lệnh sau đây:

```
.
.
     printf ("\nErase file %s (Y/N) ? ", file1);
     ans  =  getchar ();
.
.
 if(remove(file1))
 {
```

```
        printf ("\nFile cannot be erased");
        exit(1);
    }
```

✓ **Làm sạch các stream**

Thông thường, các tập tin xuất chuẩn được trang bị vùng đệm. Điều này có nghĩa là kết xuất cho tập tin được thu thập trong bộ nhớ và không thật sự hiển thị cho đến khi vùng đệm đầy. Nếu một chương trình bị treo hay kết thúc bất thường, một số ký tự vẫn còn nằm trong vùng đệm. Kết quả là chương trình có vẻ như kết thúc sớm hơn là nó thật sự đã làm. Hàm **fflush()** sẽ giải quyết vấn đề này. Như tên gọi của nó, nó sẽ làm sạch vùng đệm và chép những gì có trong vùng đệm ra ngoài. Hành động làm sạch tùy theo kiểu tập tin. Một tập tin được mở để đọc sẽ có vùng đệm nhập trống, trong khi một tập tin được mở để ghi thì vùng đệm xuất của nó sẽ được ghi vào tập tin.

Nguyên mẫu của hàm này là:

```
int fflush(FILE  * fp);
```

Hàm fflush() sẽ ghi nội dung của bất kỳ vùng đệm dữ liệu nào vào tập tin kết hợp với **fp**. Hàm fflush(), không có đối số, sẽ làm sạch tất cả các tập tin đang mở để xuất. Nó trả về 0 nếu thành công, ngược lại, nó trả về EOF.

✓ **Các stream chuẩn**

Mỗi khi một chương trình C bắt đầu thực thi dưới DOS, hệ điều hành sẽ tự động mở 5 stream đặc biệt. 5 stream này là:

- Nhập chuẩn (stdin)
- Xuất chuẩn (stdout)
- Lỗi chuẩn (stderr)
- Máy in chuẩn (stdprn)
- Thiết bị hỗ trợ chuẩn (stdaux)

Trong đó, **stdin**, **stdout** và **stderr** được gán mặc định cho các thiết bị nhập/xuất chuẩn của hệ thống trong khi **stdprn** được gán cho cổng in song song đầu tiên và **stdaux** được gán cho cổng nối tiếp đầu tiên. Chúng được định nghĩa như là các con trỏ cố định kiểu FILE, vì vậy chúng có thể được sử dụng ở bất kỳ nơi nào mà việc sử dụng con trỏ FILE là hợp lệ. Chúng cũng có thể được chuyển một cách hiệu quả cho các stream hay thiết bị khác mỗi khi cần định hướng lại.

Chương trình sau đây in nội dung của tập tin vào máy in.

Ví dụ 4:

```
#include <stdio.h>
main()
{
FILE *in;
 char buff[81], fname[13];
 clrscr();
 printf("Enter the Source File Name:");
 gets(fname);

 if((in=fopen(fname, "r"))==NULL)
 {
      fputs("\nFile not found", stderr);
      /*  display  error  message  on  standard  error
rather
 than standard output */

      exit(1);
 }
 while(!feof(in))
 {
      if(fgets(buff, 81, in))
      {
      fputs(buff, stdprn);
      /* Send line to printer */
      }
 }
 fclose(in);
}
```

Lưu ý cách sử dụng của stream **stderr** với hàm **fputs()** trong chương trình trên. Nó được sử dụng thay cho hàm printf vì kết xuất của hàm printf là ở stdout, nơi mà có thể định hướng lại. Nếu kết xuất của một chương trình được định hướng lại và một lỗi xảy ra trong quá trình thực thi, thì tất cả các thông báo lỗi đưa ra cho stream stdout cũng phải được định hướng lại. Để tránh điều này, stream **stderr** được dùng để hiển thị thông báo lỗi lên màn hình vì kết xuất của **stderr** cũng là thiết bị xuất chuẩn, nhưng stream **stderr** không thể định hướng lại. Nó luôn luôn hiển thị thông báo lên màn hình.

✓ **Con trỏ kích hoạt hiện hành**

Để lần theo vị trí nơi mà các thao tác nhập/xuất đang diễn ra, một con trỏ được duy trì trong cấu trúc FILE. Mỗi khi một ký tự được đọc ra hay ghi vào một stream, con trỏ kích hoạt hiện hành (current active pointer) (gọi là **curp**) được tăng lên. Hầu hết các hàm nhập xuất đều tham chiếu đến **curp**, và cập nhật nó sau các thủ tục nhập hoặc xuất trên stream. Vị trí hiện hành của con trỏ này có thể được tìm thấy bằng sự trợ giúp của hàm **ftell()**. Hàm **ftell()** trả về một giá trị kiểu **long int** biểu diễn vị trí của **curp** tính từ đầu tập tin trong stream đã cho. Nguyên mẫu của hàm **ftell()** là:

```
long int ftell(FILE *fp);
```

Câu lệnh trích từ một chương trình sẽ hiển thị vị trí của con trỏ hiện hành trong stream **fp**.

```
printf("The current location of the file pointer is :
%ld ",
ftell (fp));
```

➢ **Đặt lại vị trí hiện hành**

Ngay sau khi mở stream, con trỏ kích hoạt hiện hành được đặt là 0 và trỏ đến byte đầu tiên của stream. Như đã thấy trước đây, mỗi khi có một ký tự được đọc hay ghi vào stream, con trỏ kích hoạt hiện hành sẽ tăng lên. Bên trong một chương trình, con trỏ có thể được đặt đến một vị trí bất kỳ khác với vị trí hiện hành vào bất kỳ lúc nào. Hàm **rewind()** đặt vị trí con trỏ này về đầu. Một hàm khác được sử dụng để đặt lại vị trí con trỏ này là **fseek()**.

Hàm **fseek()** định lại vị trí của **curp** dời đi một số byte tính từ đầu, từ vị trí hiện hành hay từ cuối stream là tùy vào vị trí được qui định khi gọi hàm **fseek()**. Nguyên mẫu của hàm **fseek()** là:

```
int fseek(FILE *fp, long int offset, int origin);
```

trong đó **offset** là số byte cần di chuyển vượt qua vị trí tập tin được cho bởi tham số **origin**. Tham số origin chỉ định vị trí bắt đầu tìm kiếm và phải có giá trị là 0, 1 hoặc 2, biểu diễn cho 3 hằng ký hiệu (được định nghĩa trong stdio.h) như trong bảng 21.4:

Origin	Vị trí tập tin
SEEK_SET or 0	Đầu tập tin
SEEK_CUR or 1	Vị trí con trỏ của tập tin hiện hành
SEEK_END or 2	Cuối tập tin

Hàm **fseek()** trả về giá trị 0 nếu đã thành công và giá trị khác 0 nếu thất bại.

Đoạn lệnh sau tìm mẫu tin thứ 6 trong tập tin:

```
struct addr
{
      char name[40];
      char street[40];
      char city[40];
      char state[3];
 char pin[7];
}
FILE *fp;
.
.
.
fseek(fp, 5L*sizeof(struct addr), SEEK_SET);
```

Hàm sizeof() được dùng để tìm độ dài của mỗi mẩu tin theo đơn vị byte. Giá trị trả về được dùng để xác định số byte cần thiết để nhảy qua 5 mẩu tin đầu tiên.

✓ **Hàm fprintf() và fscanf()**

Ngoài các hàm nhập xuất đã được thảo luận, hệ thống nhập/xuất có vùng đệm còn bao gồm các hàm **fprintf()** và **fscanf()**. Các hàm này tương tự như hàm **printf()** và **scanf()** ngoại trừ rằng chúng thao tác trên tập tin. Nguyên mẫu của hàm **fprintf()** và **fscanf()** là:

```
int fprintf(FILE * fp, const char *control_string,..);
int fscanf(FILE *fp, const char *control_string,...);
```

trong đó **fp** là con trỏ tập tin trả về bởi lời gọi hàm **fopen()**. Hàm **fprintf()** và **fscanf()** định hướng các thao tác nhập xuất của chúng đến tập tin được trỏ bởi **fp**.

Đoạn chương trình sau đây đọc một chuỗi và một số nguyên từ bàn phím, ghi chúng vào một tập tin trên đĩa, và sau đó đọc thông tin và hiển thị trên màn hình.

```
.
 .
printf("Enter a string and a number: ");
fscanf(stdin, "%s %d", str, &no);
/* read from the keyboard */
fprintf(fp, "%s %d", str, no);
/* write to the file*/
fclose (fp);
.
.
fscanf(fp, "%s %d", str, &no)
/* read from file  */
fprintf(stdout, "%s %d", str, no)
/* print on screen */
.
.
```

Nên nhớ rằng, mặc dù **fprintf()** và **fscanf()** thường là cách dễ nhất để ghi vào và đọc dữ liệu hỗn hợp ra các tập tin trên đĩa, nhưng chúng không phải luôn luôn là hiệu quả nhất. Nguyên nhân là mỗi lời gọi phải mất thêm một khoảng thời gian, vì dữ liệu được ghi theo dạng ASCII có định dạng (như nó sẽ xuất hiện trên màn hình) chứ không phải theo định dạng nhị phân. Vì vậy, nếu tốc độ và độ lớn của tập tin là đáng ngại, **fread()** và **fwrite()** sẽ là lựa chọn tốt hơn.

5.3.4. Quy trình kỹ thuật trong bài học của GV và SV:

*** Quy trình thị phạm của GV**

- Bước 1: Chuẩn bị bài giảng, tài liệu và điều kiện trang thiết bị giảng dạy
- Bước 2: Chuẩn bị bài tập mẫu và đưa ra nội dung thực hành
- Bước 3: Phổ biến các quy trình, các bước thực hiện
- Bước 4: Hướng dẫn nội dung bài học, Hướng dẫn thị phạm mẫu
- Bước 5: Quản lý lớp học
- Bước 6: Giải đáp, tư vấn, đánh giá, nhận xét
- Bước 7: Chấm điểm và nộp các bài đánh giá cho bộ môn

*** Quy trình thực hiện bài của SV**

- Bước 1: SV chuẩn bị tài liệu, vở, bút, giấy nháp, tài liệu tra cứu
- Bước 2: Đọc tài liệu Lập trình căn bản C phần quản lý tệp tin
- Bước 3: Sinh viên trình bày bài thực hành, thảo luận và nhận xét góp ý.

5.3.5. Các mẫu hình sản phẩm cho SV tham khảo:

- SV xem video

5.3.6. Phần tự thực hành thao tác thường xuyên của SV trong bài học:

Câu 1: Viết một chương trình để nhập dữ liệu vào một tập tin và in nó theo thứ tự ngược lại.

Câu 2: Viết một chương trình để truyền dữ liệu từ một tập tin này sang một tập tin khác, loại bỏ tất cả các nguyên âm (a, e, i, o, u). Loại bỏ các nguyên âm ở dạng chữ hoa lẫn chữ thường. Hiển thị nội dung của tập tin mới.

5.3.7. Sản phẩm thực hành:

+ Các ví dụ demo.

5.3.8. Điều kiện để GV- SV thực hiện bài học thực hành;

5.3.8.1. Điều kiện chuẩn bị trước bài học

+ Giảng viên:

- Đề cương chi tiết bài giảng
- Thiết kế bài giảng
- Học liệu tham khảo, mở rộng

+ Sinh viên:

- Đọc kỹ bài học trước khi giảng viên lên lớp
- Chuẩn bị vấn đề thảo luận, trao đổi với giảng viên

- Các tài liệu tham khảo, mở rộng do giảng viên yêu cầu

5.3.8.2. Kỹ thuật và phương tiện dạy học:

- Bảng, phấn viết
- Màn hình lớn
- Máy tính, máy chiếu

2.8.3. Phương pháp tổ chức dạy-học:

- Thuyết trình, diễn giảng, thực hành
- Phát vấn
- Trao đổi, thảo luận
- Cemina
- Vấn đáp

TÀI LIỆU THAM KHẢO

1. **Nguyễn xuân Huy**, *Thuật toán*, Nhà xuất bản Khoa Học và Kỹ Thuật.

2. **Hoàng Kiếm**, *Giải một bài toán trên máy tính như thế nào*, Nhà xuất bản Giáo dục.

3. **Nguyễn Thanh Thủy** (chủ biên), *Nhập môn lập trình Ngôn ngữ C*, Nhà xuất bản Khoa học và Kỹ thuật.

4. **Trần Văn Lăng**, *Lập trình hướng đối tượng sử dụng C++*, Nhà xuất bản Thống Kê.

5. **GS. Phạm Văn Ất**, *C++ và lập trình hướng đối tượng*, Nhà xuất bản Khoa học và Kỹ thuật.

6. **Nguyễn Thanh Thủy**, *Lập trình hướng đối tượng*, Nhà xuất bản Khoa học và Kỹ thuật.

7. **Tô Oai Hùng**, *Giáo trình Cơ sở lập trình sử dụng ngôn ngữ C++*, Tài liệu lưu hành nội bộ (Trường Đại Học Mở TP. HCM)

8. **Joel Adams & Larry Nyhoff**, *C++ An Introduction to Computing*, Prentice Hall 2002, Third Edition.

9. **H.M. Deiteil & P.J. Deitel**, *C++ How to Program*, Prentice Hall, New Jersey, 2003, Fourth Edition.

10. **Bjarne Stroustrup**, *The C++ Programming Language*, Addition Wesley Longman, 1997, Third Edition.

KHOA VHTT	BỘ MÔN TTH 2	BIÊN SOẠN
Hà Đình Hùng	**Phùng Thị Thúy Phương**	**Hoàng Anh Công**