TRẦN ĐAN THƯ - NGUYỄN THANH PHƯƠNG
ĐINH BÁ TIẾN - TRẦN MINH TRIẾT

Nhập môn lập trình

NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ THUẬT 2011

Lời nói đầu

Đa số các lập trình viên đều có những khó khăn trong những ngày đầu học lập trình của mình. Làm sao để có một tài liệu học lập trình dễ hiểu cho người học, vừa huấn luyện được những kiến thức cơ bản, lại vừa tạo nền tảng để phát triển nghề nghiệp sau này? Đó là câu hỏi mà ban biên soạn tìm cách giải đáp thông qua quyển sách này.

Trong khuôn khổ hạn hẹp của của giáo trình, các tác giả cố gắng trình bày được một phần nào các kiến thức hàn lâm về lập trình thông qua các ví dụ cụ thể. Chúng tôi cũng không quên chú trọng vào những khía cạnh kỹ thuật và công nghệ trong lập trình mà có liên quan đến từng chủ đề được thảo luận trong mỗi chương. Một số ví dụ tiêu biểu trong lập trình cũng được sưu tập và chọn lọc để đưa vào những đề mục phù hợp. Mục tiêu cuối cùng mà giáo trình muốn hướng đến là tạo cho người học những kỹ năng khởi đầu để viết và tổ chức các chương trình sao cho dễ bảo trì, có thể mở rộng và tái sử dụng mã nguồn trong các dự án phần mềm.

Ban biên soạn đã chọn cách tiếp cận sử dụng một ngôn ngữ lập trình cụ thể, chủ yếu là ngôn ngữ lập trình *C* có kết hợp với *C++*, làm ngôn ngữ chính để diễn đạt và minh họa về các khái niệm cơ bản, các thủ thuật và phương pháp lập trình.

Giáo trình hướng đến người đọc là các giảng viên và sinh viên ở các trường đại học, cao đẳng, trung tâm đào tạo chuyên nghiệp. Đây cũng là tài liệu phù hợp cho những lập trình viên mới chuyển từ các ngành nghề khác sang làm phần mềm và cũng phù hợp để các kỹ sư phần mềm ôn luyện, hay dùng trong công tác huấn luyện.

Chúng tôi xin trân trọng gửi lời cảm ơn đến Ban giám hiệu *Trường Đại học Khoa học Tự nhiên* (*Đại học Quốc gia Tp.HCM*), các đồng nghiệp đang công tác tại *Khoa Công nghệ thông tin* và các trường bạn, *Nhà xuất bản Khoa học và Kỹ thuật* đã đóng góp ý kiến và giúp đỡ chúng tôi trong quá trình biên soạn.

Mặc dù chúng tôi đã rất nỗ lực trong quá trình biên soạn, tuy nhiên giáo trình không tránh khỏi những thiếu sót. Chúng tôi rất mong

nhận được những góp ý xây dựng từ quý độc giả để nâng cao chất lượng giáo trình trong những lần tái bản sau. Mọi góp ý xin gửi bằng thư hay email đến địa chỉ:

Khoa Công Nghệ Thông Tin,
Trường Đại học Khoa học tự nhiên, ĐHQG-TPHCM,
227 Nguyễn Văn Cừ, Quận 5, Thành phố Hồ Chí Minh,
Điện thoại: (08) 38354266
Email: fitbooks@fit.hcmus.edu.vn

Xin vui lòng ghi chủ đề là "*Góp Ý cho Giáo trình Nhập môn lập trình*".

Tp. Hồ Chí Minh, ngày 15 tháng 9 năm 2011,

Thay mặt nhóm biên soạn,

PGS. TS. Trần Đan Thư.

Mục lục

Chương 1
TỔNG QUAN VỀ LẬP TRÌNH MÁY TÍNH

I. KHÁI NIỆM VỀ CHƯƠNG TRÌNH

Một chương trình (*program*) là một dãy các chỉ thị (*instruction*) điều khiển sự hoạt động của máy tính nhằm để giải quyết một công việc nào đó. Người viết chương trình (còn gọi là: lập trình viên, thảo chương viên – *programmer*) là những người tạo lập ra những chương trình điều khiển máy tính.

Để hiểu rõ hơn về khái niệm chương trình, chúng ta cần phải tìm hiểu qua những khái niệm cơ bản nhất liên quan đến công nghệ viết chương trình cho máy tính (hay còn gọi là "lập trình", tiếng Anh là *programming*).

I.1 Chương trình mã máy

CPU của máy tính được thiết kế để có thể thực hiện được các **chương trình mã máy** (*machine code program*) đã được hệ điều hành nạp vào RAM của máy tính. Mỗi chương trình mã máy thường phải tương thích với từng họ máy cụ thể, bao gồm một tập hợp các chỉ thị được viết bằng các lệnh CPU của họ máy đó, được lưu trên đĩa dưới dạng một **tập tin mã thực thi** (*executable program file*) của một hệ điều hành cụ thể. Chẳng hạn với họ hệ điều hành *Windows* thì đó là các tập tin dạng ***.EXE** hay ***.DLL**.

Qui trình thực hiện một chương trình mã máy đã có sẵn trên đĩa dưới dạng các tập tin mã thực thi gồm các bước theo thứ tự như sau:

- **Người sử dụng** (*người dùng cuối – end user*) ra lệnh thực hiện chương trình, tức là chạy hay khởi động chương trình thông qua lệnh gọi chương trình theo cách thức qui định của hệ điều hành;
- **Hệ điều hành** nhận được lệnh, tìm và nạp *tập tin mã thực thi* của chương trình (đang nằm trên đĩa) vào RAM máy tính, bộ đếm lệnh của CPU (*CPU program counter*) được trỏ đến lệnh đầu tiên của chương trình (cũng còn gọi là "**ngõ vào chương trình**", *program entry point*);
- *CPU* bắt đầu quá trình thực hiện từng chỉ thị một trong RAM cho đến khi gặp lệnh kết thúc:
    - Chép lệnh mã máy hiện hành vào thanh ghi lệnh;
    - Tăng bộ đếm lệnh (để trỏ đến lệnh kế tiếp);
    - Thi hành lệnh mã máy.
- Kết thúc thực hiện chương trình, hệ điều hành chờ nhận lệnh mới.

Qua mô tả như trên, chúng ta có thể ghi nhận các đặc điểm sau đây về các chương trình mã máy:

- Mỗi chỉ thị của chương trình là một lệnh mã máy, tức là một dãy các byte chỉ phù hợp với qui ước tập lệnh của một loại CPU nào đó mà thôi;
- Mỗi chương trình mã máy cũng phải được cấu trúc hóa theo qui ước của hệ điều hành (thì hệ điều hành mới có thể nạp nó vào RAM và cho thi hành);
- Chương trình mã máy của một họ CPU và một hệ điều hành cụ thể có thể không chạy được trên họ CPU khác hay trên một hệ điều hành khác;
- Nội dung của một chương trình mã máy rất khó hiểu đối với người dùng máy tính, chỉ có CPU thích hợp mới hiểu rõ và thi hành được chương trình đó.

Từ những đặc điểm vừa nêu, chúng ta thấy rằng *khó có thể sản xuất ra phần mềm bằng cách viết trực tiếp các chương trình mã máy*. Hơn nữa dù có làm được thì giá cả sẽ rất đắt (do quá khó, tốn quá nhiều

thời gian và công sức) mà khả năng dùng lại rất giới hạn (không thể đem bán cho người dùng trên họ máy tính khác hay người dùng sử dụng hệ điều hành khác).

I.2 Chương trình hợp ngữ và trình hợp dịch

Việc viết các chương trình mã máy rất cực và kém hiệu quả ngay cả đối với các lập trình viên chuyên nghiệp. Một trong những giải pháp khởi đầu để cải tiến năng suất lập trình cho máy tính sử dụng hợp ngữ (*assembly language*) thay cho mã máy.

Hợp ngữ là loại ngôn ngữ lập trình trừu tượng hơn so với ngôn ngữ máy, mỗi lệnh hợp ngữ là một từ viết ngắn thay cho một mã số lệnh (ví dụ: lệnh cộng hai số được ký hiệu bởi một từ tiếng Anh là ADD), dữ liệu của lệnh được thay bằng các tên mô tả thay cho các địa chỉ vật lý của ô nhớ. Vì vậy một chương trình viết bằng hợp ngữ dễ viết, dễ đọc và dễ sửa hơn chương trình ngôn ngữ máy.

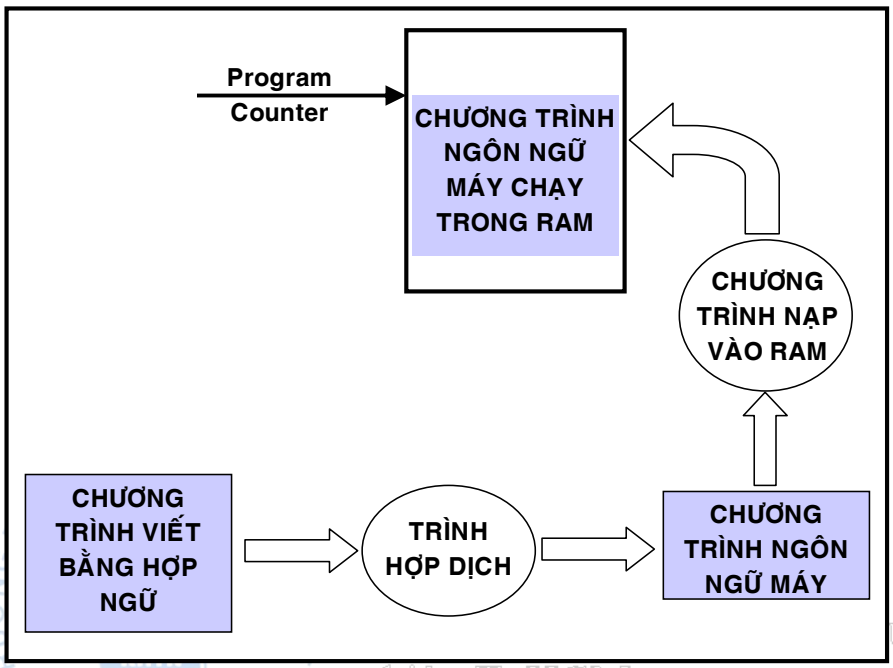
Thật ra, CPU máy tính không thể hiểu một chương trình được bằng hợp ngữ, do đó trước khi một chương trình hợp ngữ được thực hiện, người ta phải dịch thành chương trình ngôn ngữ máy. Quá trình dịch chương trình hợp ngữ thành mã máy được tiến hành nhờ một chương trình đặc biệt được gọi là "*trình hợp dịch*" (*assembler*): hình 1.1 minh họa cho quá trình này. Mỗi chương trình hợp dịch chỉ có thể dịch hợp ngữ của một họ máy tính cụ thể. Mỗi lệnh hợp ngữ tương ứng với đúng một lệnh mã máy, vì vậy quá trình dịch chương trình hợp ngữ thành chương trình mã máy gần như là tương ứng một-một.

Mặc dù chương trình hợp ngữ dễ dàng đối với người lập trình hơn là chương trình mã máy, chương trình viết ra chỉ dùng được cho một họ máy cụ thể, không tương thích với nhiều họ máy tính. Do đó hợp ngữ cũng hạn chế sử dụng trong quá trình sản xuất phần mềm vì tính không tương thích của nó.

I.3 Các ngôn ngữ lập trình

Ngôn ngữ lập trình (*programming language*) là ngôn ngữ được lập trình viên sử dụng để viết chương trình cho máy tính. Khi một chương trình được viết bằng một ngôn ngữ lập trình nào đó thì các chỉ thị, câu

lệnh trong chương trình phải tuân theo các qui tắc, các luật do ngôn ngữ lập trình đó qui định.

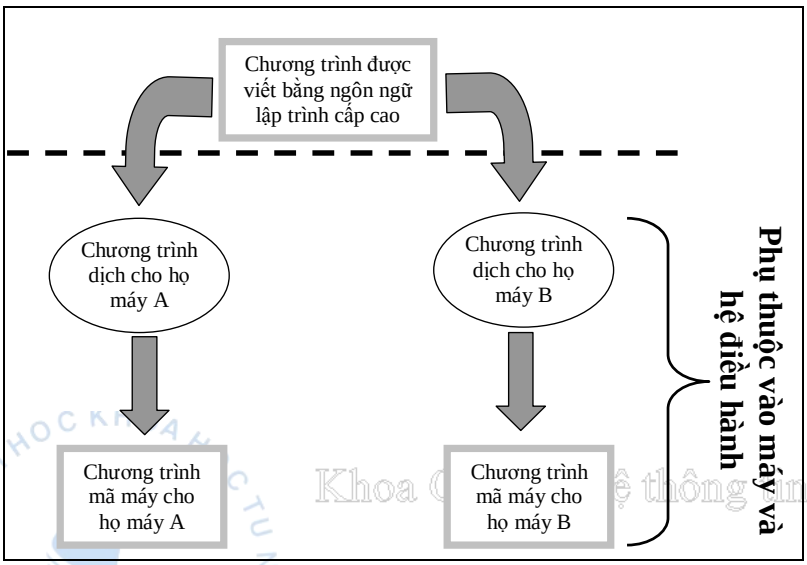


Hình 1.1: Sơ đồ dịch và chạy một chương trình viết bằng hợp ngữ

Hợp ngữ cũng là một dạng ngôn ngữ lập trình và được gọi là ngôn ngữ lập trình cấp thấp (*low-level programming language*). Ngôn ngữ lập trình cấp thấp phụ thuộc vào từng họ máy cụ thể, vì vậy không có tính tương thích, tức là một chương trình ngôn ngữ cấp thấp của họ máy này không thể chạy được trên họ máy khác. Ngôn ngữ cấp thấp có ưu điểm là tận dụng được tính năng của mỗi họ máy cụ thể, nhờ vậy chương trình có thể chạy nhanh hơn, có thể khai thác và tận dụng được các ưu điểm của từng họ máy cụ thể.

Do sự hạn chế của ngôn ngữ lập trình cấp thấp, các ngôn ngữ lập trình cấp cao (*high-level programming language*) được đề xuất với mục đích nâng cao tính tương thích của các chương trình được viết bằng ngôn ngữ lập trình cấp cao. Ý tưởng này được minh họa trong hình 1.2: cùng một chương trình được viết bằng ngôn ngữ cấp cao có thể được dịch ra thành các chương trình chạy được trên nhiều họ máy

khác nhau. Khi có thêm họ máy mới, người ta sẽ bổ sung thêm chương trình dịch mới, trong khi chương trình viết bằng ngôn ngữ vẫn được giữ nguyên.



Hình 1.2: Tính tương thích của ngôn ngữ lập trình cấp cao

Người lập trình thường viết chương trình bằng các ngôn ngữ ngữ lập trình cấp cao bởi vì nó dễ dùng, có thể diễn đạt được các ý tưởng trừu tượng, và có tính tương thích cao (nghĩa là khi thay đổi dạng máy tính thì chỉ cần sửa chương trình rất ít hay thậm chí khỏi cần sửa mà vẫn bảo đảm chương trình chạy đúng).

Có rất nhiều loại ngôn ngữ lập trình cấp cao, chẳng hạn như: *Ada*, *BASIC*, *C/C++*, *COBOL*, *FORTRAN*, *Lisp*, *Pascal*, *Java*, *C#*, *Visual Basic* (*VB*), *Perl*, *Ruby*,… Mỗi ngôn ngữ được thiết kế để phù hợp với một số mục đích cụ thể.

- Ngôn ngữ *BASIC* (viết tắt của "*Beginners All-purpose Symbolic Instructional Code*"): đây là ngôn ngữ lập trình rất nổi tiếng trong quá khứ vì rất dễ sử dụng, ngay cả cho những đối tượng không chuyên tin học. Hiện nay, ngôn ngữ *VB (Visual Basic)* và *VB.NET* của *Microsoft* cũng là các ngôn ngữ lập trình khá mạnh về các chức năng lập trình nhưng lại rất dễ sử dụng.

- Ngôn ngữ *C* được dùng cho các lập trình viên chuyên nghiệp để viết các hệ điều hành, cài đặt các hệ quản lý cơ sở dữ liệu, các tính toán số trong các lĩnh vực khoa học khác.
- Ngôn ngữ *C++* là ngôn ngữ lập trình hướng đối tượng được cải tiến từ ngôn ngữ *C*. Đây là một ngôn ngữ lập trình rất mạnh và đa dụng, được dùng cho những người chuyên nghiệp. Hiện nay, số lượng các phần mềm được viết bằng ngôn ngữ *C++* chiếm tỉ lệ rất lớn.
- Ngôn ngữ *COBOL* (*Common Business Oriented Language*) dùng để viết các chương trình xử lý dữ liệu thương mại của các hệ thống thông tin cũ.
- Ngôn ngữ *FORTRAN* (*Formula Translation*) dành cho các chương trình tính toán trong khoa học kỹ thuật.
- Các ngôn ngữ *Java* và *C#* là đại diện cho thế hệ ngôn ngữ lập trình hiện đại và mang tính chuyên nghiệp, sẽ được dùng chủ yếu để phát triển các hệ thống phần mềm lớn trong tương lai gần.
- Các ngôn ngữ *PHP*, *Ruby*, *Perl* thường được dùng trong các ứng dụng Web, lập trình quản trị và mạng.

II. CÁC CHƯƠNG TRÌNH DỊCH

II.1 Vai trò của các chương trình dịch

Giống như các *trình hợp dịch* cho những chương trình hợp ngữ, các chương trình dịch ngôn ngữ cấp cao có nhiệm vụ dịch các chương trình ngôn ngữ cấp cao thành các chương trình mã máy. Mỗi chương trình dịch ngôn ngữ cấp cao thường được viết để chạy trên một hệ điều hành cụ thể, chuyển các *tập tin chương trình cấp cao* thành *mã thực thi* chạy được trên một hệ điều hành cụ thể. Chương trình được viết bằng ngôn ngữ cấp cao cũng được gọi là "*chương trình nguồn*" (*source code program*) hay "*mã nguồn*" (*source code*).

Nói chung, có hai loại trình dịch cấp cao là "*trình biên dịch*" (*compiler*) và "*trình thông dịch*" (*interpreter*).

- ***Các chương trình thông dịch***: Lần lượt từng chỉ thị trong chương trình nguồn được kiểm tra xem đúng cú pháp hay không, nếu lệnh đúng thì sẽ được dịch thành mã máy và nạp vào *RAM* (bộ nhớ trong) máy tính để thực hiện ngay lập tức.
- ***Các chương trình biên dịch***: Toàn bộ mã nguồn chương trình được kiểm tra cú pháp, kiểm tra tính chặt chẽ của mã nguồn tùy theo qui định của ngôn ngữ, sau đó được dịch hết thành mã thực thi (*bao gồm cả mã máy lẫn thông tin quản lý của hệ điều hành* cần thiết cho quá trình nạp vào bộ nhớ) và ghi lên đĩa dưới dạng một tập tin thực thi.

Như vậy đối với trường hợp trình biên dịch thì *tập tin chương trình thực thi* được lưu dưới dạng tập tin trên đĩa của một hệ điều hành cụ thể, có khả năng chứa những thông tin phụ thuộc vào hệ điều hành.

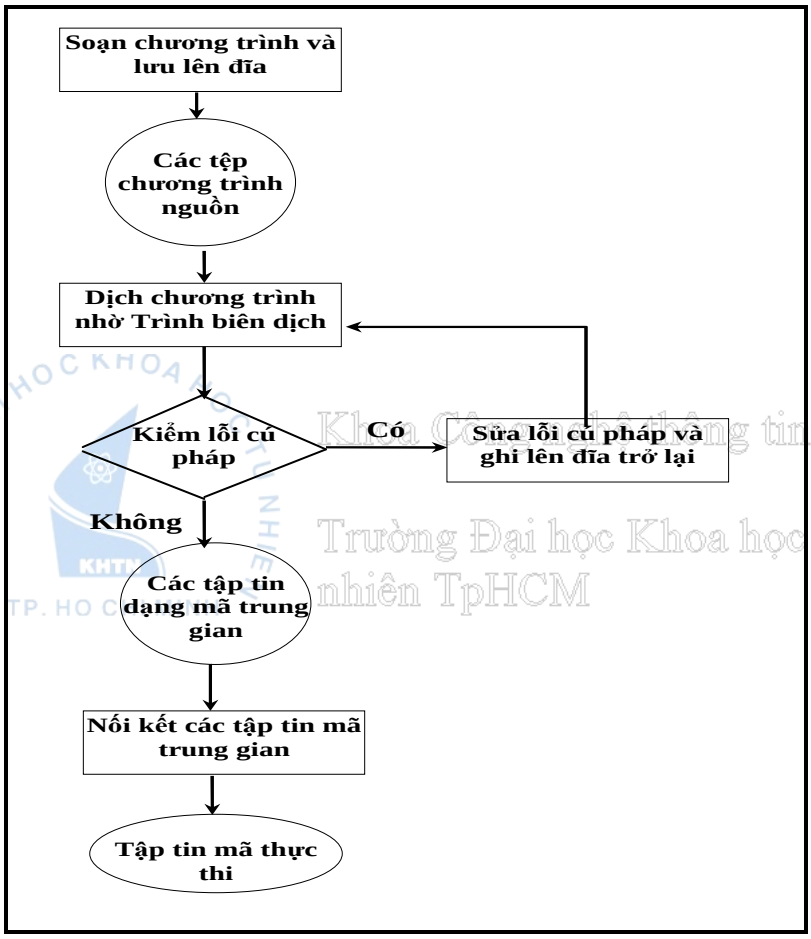
II.2 Qui trình viết và biên dịch chương trình

Đối với các ngôn ngữ cấp cao truyền thống (trước thế hệ của *Java* và *C#*), quá trình viết, biên dịch và chạy một chương trình được minh họa trong hình 1.3, gồm các công đoạn chính như sau:

- Người lập trình sử dụng một ***trình soạn thảo*** văn bản (*text editor*) để soạn và ghi chương trình vào đĩa thành một hay nhiều tập tin chương trình nguồn. Thông thường, trình soạn thảo văn bản được chạy tại trạm cuối của một máy tính lớn hay chạy trên một máy tính cá nhân.
- Chương trình nguồn được dịch thành ngôn ngữ máy nhờ vào ***trình biên dịch***. Nếu chương trình nguồn không có lỗi cú pháp (*syntax error*) thì sẽ được dịch thành dạng mã đối tượng (*object code*), một loại mã trung gian chưa phải là mã máy thật sự nên không thể được nạp vào bộ nhớ để chạy.
- Các tập tin mã trung gian sinh ra từ bước trên được nối kết lại (liên kết, ***link***) để thành một chương trình ngôn ngữ máy hoàn chỉnh có thể chạy được. Chương trình dùng để nối kết các tập tin mã trung gian được gọi là chương trình liên kết mã (*linker*, *link program*).

- Chạy chương trình ngôn ngữ máy hoàn chỉnh, kiểm tra kết quả.

Đối với các ngôn ngữ lập trình cấp cao hiện đại như *Java* và *C#*, công nghệ biên dịch và thực thi có nhiều cải tiến: điều này sẽ được giới thiệu trong phần sau.



Hình 1.3: Qui trình dịch chương trình ngôn ngữ cấp cao nhờ dùng một trình biên dịch

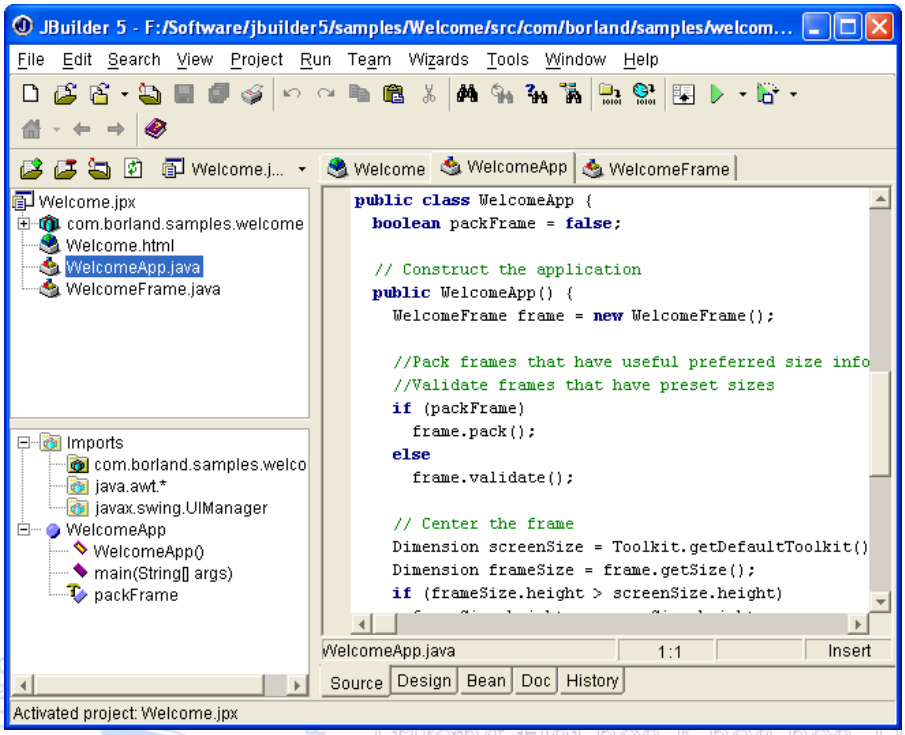
III. MÔI TRƯỜNG LẬP TRÌNH

Hiện nay người lập trình thực hiện toàn bộ qui trình biên dịch một cách dễ dàng và thuận tiện nhờ vào các công cụ do các hãng phần mềm cung cấp cho mỗi loại ngôn ngữ lập trình cấp cao.

Các công cụ này được gọi là môi trường phát triển chương trình tích hợp (viết tắt là *IDE* của cụm từ tiếng Anh "*Integrated Development Environment*"). Mỗi IDE có thể tích hợp nhiều chức năng như:

- ***Soạn thảo chương trình***: Cung cấp các tiện ích trong quá trình soạn thảo mã nguồn chẳng hạn như đổi màu từ khóa, loại trừ bớt các sơ sót trong khi soạn mã nguồn, hỗ trợ giúp đỡ lập trình viên trong lúc soạn mã nguồn.
- ***Quản lý hệ thống tập tin mã nguồn***: Đối với các dự án phần mềm lớn thì việc hệ thống tập tin mã nguồn rất nhiều, có khi hàng trăm tập tin, vì vậy cần phải có sự hỗ trợ thích hợp đối với từng ngôn ngữ lập trình (các trình quản lý tập tin thông thường, do quá tổng quát, không thích hợp để quản lý hệ thống tập tin mã nguồn của một ngôn ngữ lập trình cụ thể).
- ***Quản lý hệ thống các phiên bản của mã nguồn***: Một tập tin mã nguồn có thể có lịch sử thay đổi theo sự phát triển của hệ thống phần mềm, có thể sự chỉnh sửa theo yêu cầu của khách hàng. Vì vậy, đây là chức năng hỗ trợ lập trình viên theo dõi vết của quá trình lịch sử thay đổi mỗi tập tin mã nguồn.
- ***Chức năng chính***: Kiểm tra lỗi cú pháp, biên dịch và liên kết chương trình.
- ***Một số chức năng tiện nghi khác như***: chạy thử chương trình, chạy chương trình theo từng dòng lệnh (*debug*) để tìm lỗi.

Các IDE thông dụng hiện nay có sẵn trong các sản phẩm phần mềm hỗ trợ lập trình các ngôn ngữ lập trình khác nhau như: *Eclipse* (cho nhiều ngôn ngữ), *C++ Visual Studio* (ngôn ngữ C++), *C# Visual Studio* (ngôn ngữ C#), *Visual Café* (ngôn ngữ *Java*), *J Buider* (ngôn ngữ *Java*).



Hình 1.4: Phần mềm JBuilder là một IDE cho ngôn ngữ lập trình Java

Hình 1.4 minh họa *Jbuilder* (phiên bản 1.0) một ***IDE*** cho ngôn ngữ lập trình *Java*. Trong cửa sổ giao diện của *Jbuilder*, ta thấy có nhiều cửa sổ con để hiển thị mã nguồn *Java* cũng như các thông tin khác về chương trình *Java* đang được viết.

IV. MỘT VÍ DỤ ĐƠN GIẢN VỀ LẬP TRÌNH

Phần này trình bày một ví dụ về một chương trình đơn giản: chỉ in ra một dòng chữ có nội dung là "*Hello everybody !*". Bảng 1.1 trình bày chương trình này được viết bằng ngôn ngữ *Java* và *C*. Ý nghĩa các dòng trong chương trình như sau:

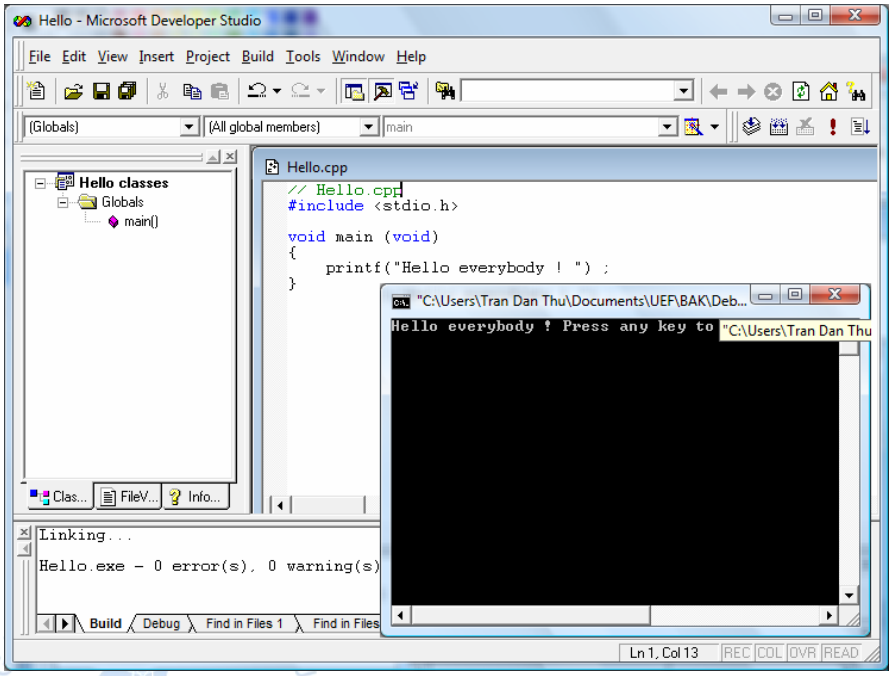
- ***Dòng 1***: Mỗi dòng ghi chú được bắt đầu bởi // , dòng này không có tác dụng khi chương hoạt động, các chú thích này có thể ghi tùy ý bằng tiếng Việt hay tiếng Anh.

- ***Dòng 2***: Khai báo để sử dụng các hàm thư viện có sẵn của ngôn ngữ, đối với *Java* là thư viện **java.util.***, đối với ngôn ngữ *C* là trong **stdio.h**, lệnh dùng ở dòng 6 là hàm được định nghĩa trong thư viện đã khai báo ở dòng này.
- ***Dòng 3***: Bắt đầu khai báo lớp đối với trường hợp *Java*.
- ***Dòng 4***: Đầu vào (*entry point*) của chương trình chính bắt đầu bằng một hàm đặc biệt: tên hàm này là **main**, chương trình sẽ bắt đầu chạy tại chỗ vào này.
- ***Dòng 5***: Chương trình bắt đầu bằng dấu móc mở **{**
- **Dòng 6**: Lệnh in được gọi để in dữ kiện ra thiết bị xuất chuẩn, thiết bị xuất mặc nhiên là màn hình máy tính.
- **Dòng 7**: Chương trình kết thúc bằng dấu móc đóng **}**
- **Dòng 8**: Kết thúc khai báo lớp đối với trường hợp **Java**.

	Chương trình Java	Chương trình C
1	`// Hello.java`	`// Hello.c`
2	`import java.util.* ;`	`#include <stdio.h>`
3	`public class Hello {`	
4	`public static void main( String argv[ ] )`	`void main (void)`
5	`{`	`{`
6	`System.out.print ("Hello everybody ! ") ;`	`printf("Hello everybody ! ") ;`
7	`}`	`}`
8	`}`	

Bảng 1.1 Chương trình in ra một dòng chữ

Hình 1.5 minh họa việc dùng *C++ Visual Studio* để dịch và chạy thử chương trình *Hello.c*. Trên *Windows* thì chương trình này được dịch thành *Hello.exe* có thể chạy độc lập trên hệ điều hành mà không cần đến chương trình dịch. Tuy nhiên nếu mang *Hello.exe* qua hệ điều hành *Linux* thì chương trình này không thể chạy được: phải mang chương trình nguồn *Hello.c* qua *Linux* để dịch lại và chạy.



Hình 1.5: Dịch và chạy chương trình Hello.cpp trong IDE của Visual C++

V. CÔNG NGHỆ LẬP TRÌNH HIỆN ĐẠI

Một hạn chế đối với các chương trình cấp cao truyền thống là chương trình dịch của chúng phát sinh trực tiếp mã thực thi phụ thuộc vào mã máy tính của một họ máy tính và hệ điều hành cụ thể. Vì vậy các tập tin mã thực thi phụ thuộc hệ điều hành, không thể mang đi sử dụng ở những hệ điều hành khác. Như trường hợp chương trình *Hello.cpp* nói trên: phải mang tập tin mã nguồn đến hệ điều hành khác và dịch lại để chạy.

Đối với các ngôn ngữ lập trình hiện đại như *Java* hay *C#* thì trình biên dịch không dịch trực tiếp mã nguồn thành mã thực thi. Các ngôn ngữ này được thiết kế để có thể dịch thành mã thực thi trừu tượng (*abstract executable code*) độc lập máy và hệ điều hành. Tuy nhiên vì máy tính thật không thể hiểu được mã trừu tượng, những chương trình dạng mã thực thi trừu tượng chỉ chạy được khi có sẵn máy ảo hỗ trợ cho việc thi hành loại mã thực thi đó.

Ví dụ, đối với trường hợp *Java*, các chương trình nguồn *Java* (ghi trên đĩa dưới dạng các tập tin ***.java**) được dịch thành mã thực thi không phụ thuộc máy tính (ghi thành các tập tin ***.class**) và có thể chạy được trên bất kỳ máy tính nào đã cài đặt máy ảo *Java* (*Java Virtual Machine - JVM*)**.**

Trong những năm gần đây, các ứng dụng chạy trên *web* phát triển rất mạnh. Đây là các ứng dụng được chạy trên Internet thông qua một trình duyệt Web. Các ứng dụng này được viết bằng các ngôn ngữ như **PHP**, **ASP.NET**, **JSP**, **Java Script**, **VB Script…** có tính tương thích cao, hoạt động trên bất kỳ máy tính nào có *Internet*. Vì vậy những ngôn ngữ phát triển phần mềm *web* sẽ những ngôn ngữ lập trình có triển vọng trong tương lai.

VI. KẾT CHƯƠNG

Chương này trình bày tổng quan về các khái niệm cơ bản về lập trình cho máy tính và công nghệ lập trình.

- Lập trình viên sử dụng các ngôn ngữ lập trình, đa phần là ngôn ngữ lập trình cấp cao, để viết ra các chương trình hay các phần mềm máy tính.
- Các chương trình viết bằng ngôn ngữ lập trình cấp cao được dịch thành mã thực thi để có thể thi hành được. Đối với các ngôn ngữ hiện đại như *Java* hay *C#* thì mã thực thi độc lập với máy và hệ điều hành, chạy trên bất kỳ hệ điều hành nào đã thiết lập sẵn máy ảo cho ngôn ngữ.
- Ngôn ngữ *C++* hiện đang dùng rất nhiều trong giới công nghiệp phần mềm. Các ngôn ngữ *Java* hay *C#* là các ngôn ngữ để phát triển những hệ thống phần mềm trong tương lai.
- Các ngôn ngữ phát triển phần mềm *web* được dùng để phát triển phần mềm dùng rộng khắp trên *Internet*, tương thích cao, không cần cài đặt trên từng máy cục bộ.

VII. THUẬT NGỮ TIẾNG ANH

Sau đây là những thuật ngữ tiếng Anh đã dùng trong chương này.

abstract executable code: mã trừu tượng

assembler: trình hợp dịch

assembly language: hợp ngữ

compiler: trình biên dịch

data type: kiểu dữ liệu

debug: chạy chương trình theo từng dòng lệnh để tìm lỗi

executable program file: một tập tin mã thực thi

end user(s): người sử dụng, người dùng cuối

IDE: viết tắt của "*Integrated Development Environment*", môi trường phát triển chương trình tích hợp

instruction: chỉ thị

interpreter: trình thông dịch

link: nối kết các mã trung gian

linker (hay **link program**): chương trình liên kết mã trung gian

machine code program: chương trình mã máy

object code: mã đối tượng, một loại mã trung gian chưa phải là mã máy thật sự

program entry point: ngõ vào chương trình

program: chương trình

programmer: người viết chương trình, lập trình viên, từ cũ: "*thảo chương viên*"

programming language: ngôn ngữ lập trình

→ **low-level programming language**: ngôn ngữ lập trình cấp thấp

→ **high-level programming language**: ngôn ngữ lập trình cấp cao

programming: lập trình

source code program: chương trình nguồn
source code: mã nguồn
syntax error: lỗi cú pháp
text editor: trình soạn thảo văn bản (có thể dùng để soạn mã nguồn)

Chương 2

CHƯƠNG TRÌNH, DỮ LIỆU CƠ SỞ và PHÉP TOÁN

Kể từ chương này, chúng tôi sẽ mượn *ngôn ngữ lập trình C*, đôi khi cũng sử dụng *ngôn ngữ C++* khi cần thiết, để minh họa hay diễn giải cho phương pháp lập trình ở mức độ nhập môn. Độc giả có thể chọn một trình biên dịch và *IDE* nào đó để dịch và chạy thử các chương trình. Mặc dù sử dụng ngôn ngữ lập trình cụ thể để trình bày, các khái niệm được trình bày trong giáo trình sẽ là những khái niệm nền tảng chung về lập trình và rất có ích cho mọi lập trình viên.

I. CHƯƠNG TRÌNH ĐƠN GIẢN

Dòng	Chương trình C	Chương trình C++
1	`/* Hello.c */`	`// Hello.cpp`
2	`#include <stdio.h>`	`#include <iostream>`
3		`using namespace std;`
4	`void main()`	`void main()`
5	`{`	`{`
6	`  printf("Hello every body!") ;`	`  cout << "Hello every body!" ;`
7	`}`	`}`

Bảng 2.1 Chương trình in ra dòng chữ "Hello every body!"

Bảng 2.1 trình bày chương trình in ra dòng chữ "*Hello every body!*", viết bằng ngôn ngữ lập trình *C* (ở cột trái) và *C++* (ở cột phải), được lưu trên máy tính thành tập tin tên là *Hello.c* hay *Hello.cpp*. Chương

trình có thể được dịch thành chương trình mã thực thi *Hello.exe* (trên *Windows*) hay *Hello.a* (trên các hệ điều hành họ *Unix*) để chạy được.

Ý nghĩa các dòng lệnh trong chương trình nói trên như sau:

- ***Dòng 1***: Mỗi dòng ghi chú được bắt đầu bởi // đối với *C++* hay đặt bên trong /* và */ đối với ngôn ngữ *C*, dòng này không có tác dụng gì khi chương trình chạy.
- ***Dòng 2***: Lệnh khai báo sử dụng các hàm hay đối tượng có sẵn của ngôn ngữ lập trình, trường hợp này là bộ nhập xuất chuẩn (**<stdio.h>** đối với *C* và **<iostream>** đối với *C++*. Lệnh dùng ở dòng 6 đã được định nghĩa sẵn trong bộ nhập xuất này.
- ***Dòng 3***: Đối với *C++*, ta cần khai báo không gian tên ngầm định là **std** để viết ngắn gọn dòng lệnh ở dòng 6, nếu không ta phải thay **cout** ở dòng 6 bởi **std::cout**.
- ***Dòng 4***: Đầu vào (*entry point*) của chương trình chính bắt đầu bằng một hàm đặc biệt: tên hàm này là **main**, chương trình sẽ bắt đầu chạy tại chỗ vào này.
- ***Dòng 5***: Chương trình bắt đầu bằng dấu móc mở **{**
- **Dòng 6**: Lệnh in được gọi để in dữ kiện ra thiết bị xuất chuẩn, thiết bị xuất mặc nhiên là màn hình máy tính.
- **Dòng 7**: Chương trình kết thúc bằng dấu móc đóng **}**.

Bảng 2.2 trình bày chương trình tương tự như trên, nhưng xuất ra hai dòng chữ, chi tiết về chương trình này như sau:

- Đối với chương trình *C*, phía trên của bảng, trong hàm *printf* ta thấy có chèn thêm ký tự xuống dòng ‘\n’ (có nghĩa là *newline*) vào cuối của chuỗi được xuất ra (xem dòng 6 và dòng 7), kết quả là lần in kế tiếp thì sẽ bắt đầu trên một dòng mới.
- Đối với chương trình *C++*, phía dưới của bảng, ký hiệu **endl** (trong các chỉ thị ở dòng 6 và dòng 7) có nghĩa là “*end line*”, kết thúc dòng để in dòng mới. Cũng có thể thay thế **endl** bởi ‘\n’.

Dòng	Chương trình C
1	/* Hello.c */
2	#include <stdio.h>
3	
4	void main()
5	{
6	printf ("Hello every body!\n") ;
7	printf ("Let's begin.\n") ;
8	}

Dòng	Chương trình C++
1	// Hello.cpp
2	#include <iostream>
3	using namespace std;
4	void main()
5	{
6	cout << "Hello every body!" << endl ;
7	cout << "Let's begin." << endl ;
8	}

Bảng 2.2 Chương trình in ra 2 dòng chữ

I.1 Kiểu dữ liệu, hằng và biến trong chương trình

Bảng 2.3 là chương trình tính diện tích của một hình tròn bán kính đã biết trước. Chương trình này có bổ sung thêm các chỉ thị tính toán trên số thực và in kết quả tính toán. Ý nghĩa các lệnh như sau:

- **Dòng 6**: Định nghĩa một hằng số thực tên là ***Pi*** (lưu giá trị gần đúng của số pi, thường được dùng khi tính diện tích hay chu vi hình tròn). Chỉ thị ***#define*** (hay ***const float*** đối với *C++*) được dùng để khai báo hằng số thực, đặt ngay phía trước hằng số cần khai báo; hằng số được gán bằng với giá trị cố định mà người lập trình mong muốn.
- **Dòng 7**: Khai báo biến số thực (loại ***float***) tên là ***R*** nhận giá trị khởi đầu là 1.25, sau này khi chương trình đang chạy, tùy theo tình huống ***R*** có thể thay đổi thành giá trị khác.

- **Dòng 8**: Khai báo biến thực (loại ***float***) tên là ***Dientich***, chưa xác định giá trị. Khi biến được khai báo dạng này thì người lập trình sẽ viết các chỉ thị thực hiện các tính toán cần thiết để lưu kết quả vào biến đó (như trong dòng lệnh số 10).

Dòng	Chương trình C
1	/* Circle.c */
2	#include <stdio.h >
3	
4	void main()
5	{
6	#define Pi 3.14159
7	float R=1.25 ;
8	float Dientich ;
9	
10	Dientich=Pi*R*R ;
11	printf("Hinh tron, ban kinh = %f \n ", R) ;
12	printf("Dien tich = %f ", Dientich) ;
13	}
Dòng	**Chương trình C++**
1	// Circle.cpp
2	#include <iostream>
3	using namespace std;
4	void main()
5	{
6	const float Pi=3.14159 ;
7	float R=1.25 ;
8	float Dientich ;
9	
10	Dientich=Pi*R*R ;
11	cout << "Hinh tron, ban kinh = " << R << endl ;
12	cout << "Dien tich = " << Dientich ;
13	}

Bảng 2.3 Chương trình tính diện tích hình tròn bán kính R=1.25

- **Dòng 10**: Biến ***Dientich*** được tính theo các giá trị ***R*** và ***Pi*** (hai biến này đã được xác định giá trị như trong chương trình).

- **Dòng 11**: Xuất bán kính của hình tròn và xuống dòng, biến ***R*** trong lệnh này không đặt trong dấu ngoặc kép bởi vì chúng ta muốn in ra giá trị đang được lưu trong ***R*** (tức là số 1.25).
- **Dòng 12**: Xuất diện tích đã được tính, biến ***Dientich*** trong lệnh này cũng không đặt trong dấu ngoặc kép nhằm mục đích xuất ra diện tích của hình tròn đã được tính và lưu vào biến ***Dientich*** ở dòng 10.

Đối với chương trình *C* thì định dạng in ***%f*** trong lệnh ***printf*** ở dòng 11 và dòng 12 nhằm mục đích in ra các biến có kiểu ***float***.

Các hằng và biến ***Pi***, ***R***, ***Dientich*** được nói là có kiểu dữ liệu dạng ***float***, đây là kiểu số thực chính xác đơn (khá chính xác) được cung cấp sẵn bởi hầu hết các ngôn ngữ lập trình hiện nay. Khi chương trình chạy, các hằng và biến ***Pi***, ***R***, ***Dientich*** sẽ được hệ điều hành cấp phát tương ứng với các ô nhớ của bộ nhớ trong (*RAM*) của máy tính.

I.2 Qui ước đặt tên

Về mặt qui định của ngôn ngữ lập trình, các biến, hằng và các đối tượng khác phải đặt tên bằng cách dùng các chữ cái từ A đến Z, các số từ 0 đến 9, dấu gạch _ (gạch dưới, không phải dấu trừ). Tên phải luôn bắt đầu bằng chữ cái. Ví dụ: ***X1***, ***X2*** hay ***zValue***, ***Delta_46*** là các tên hợp lệ.

Để chương trình dễ hiểu, người lập trình cũng nên tôn trọng các qui ước thông thường hay cần phải tuân theo các qui định đặt tên cụ thể.

- Tên phải gợi nhớ và có liên quan về mặt ngữ nghĩa với đối tượng được đặt tên. Chẳng hạn nếu muốn sử dụng một biến số để lưu diện tích của một hình nào đó thì có thể đặt tên là ***Dien_tich***, ***mArea***. Trường hợp phải đặt tên viết tắt thì cũng nên theo cách mọi người hay dùng: ***S*** (do các công thức toán học thường ký hiệu ***S*** cho diện tích) hay ***DT*** (lấy hai ký tự đầu của từ ***diện tích***).
- Trong trường có những thỏa thuận cụ thể về bàn giao mã nguồn cho đối tác (trong các hợp đồng làm phần mềm), tên đối tượng có thể được đặt theo qui ước riêng của một số tổ chức, công ty sản xuất phần mềm.

I.3 Kích thước lưu trữ của biến

Chương trình C

```c
/* varSize.c */
#include <stdio.h>

void main( )
{
   short Delta=9;
   printf(" Kich thuoc bien Delta = %d\n ",  sizeof(Delta)) ;
   printf(" Kich thuoc kieu int = %d\n ", sizeof(int)) ;
   printf(" Kich thuoc kieu long = %d\n ", sizeof(long)) ;
   printf(" Kich thuoc kieu float = %d\n ",  sizeof(float)) ;
   printf(" Kich thuoc kieu double = %d\n ", sizeof(double)) ;
   printf(" Kich thuoc kieu char = %d\n ",  sizeof(char)) ;
}
```

Chương trình C++

```cpp
// varSize.cpp
#include <iostream>
using namespace std;
void main( )
{
   short Delta=9;
   cout << " Kich thuoc bien Delta=" << sizeof(Delta) << '\n';
   cout << " Kich thuoc kieu int=" << sizeof(int) << '\n';
   cout << " Kich thuoc kieu long=" << sizeof(long) << '\n';
   cout << " Kich thuoc kieu float=" << sizeof(float) << '\n';
   cout << " Kich thuoc kieu double=" << sizeof(double) << '\n';
   cout << " Kich thuoc kieu char=" << sizeof(char) << '\n';
}
```

Bảng 2.4 Chương trình in ra kích thước của các kiểu dữ liệu cơ bản

Khi chương trình chạy, mỗi biến hay hằng của chương trình sẽ được kết buộc với một ô nhớ bên trong bộ nhớ của máy tính. Tùy theo kiểu dữ liệu, kích thước (hay độ dài) của ô nhớ này (cũng gọi là kích thước của biến hay hằng tương ứng) sẽ chiếm một số byte nhất định trong bộ nhớ. Chẳng hạn mỗi biến có kiểu số thực ***double*** (chính xác kép) sẽ

cần 8 byte để lưu trong bộ nhớ máy tính, trong khi kiểu số thực ***float*** chỉ cần 4 byte để lưu trữ mỗi biến.

Để biết kích thước của một biến hay một kiểu dữ liệu, người lập trình có thể gọi hàm ***sizeof***(): ví dụ như đối với chương trình trong bảng 2.3, ***sizeof(****R****)*** hay ***sizeof(float)*** đều có giá trị là 4 byte, tức là số byte cần thiết để lưu số thực có độ chính xác đơn trên máy tính.

Bảng 2.4 trình bày chương trình in ra kích thước của biến ***Delta*** (có kiểu ***short***) và cũng in ra kích thước của các kiểu dữ liệu ***int***, ***long***, ***float***, ***double***, ***char***. Định dạng ***%d*** trong các lệnh in ***printf*** (tại các dòng 7 – 12) dùng để in các giá trị nguyên thông thường.

II. KHÁI NIỆM VỀ NHẬP, XUẤT, TÍNH TOÁN

Đa số các chương trình máy tính đều thực hiện ba nhóm thao tác chính như sau:

- Nhận dữ liệu từ người sử dụng (thông qua một thiết bị nhập nào đó như bàn phím, chuột…) hay từ chương trình khác: thao tác này được gọi là *nhập dữ liệu*;
- Tính toán hay xử lý dữ liệu nhập một cách thích hợp để ra được kết quả cần thiết tùy theo từng bài toán cụ thể;
- Gửi kết quả tính toán ra các thiết bị xuất (máy in, màn hình…): thao tác này được gọi là *xuất dữ liệu*.

Các chương trình trong các bảng 2.1 – 2.4 sử dụng lệnh ***printf*** (chương trình *C*) hay đối tượng ***cout*** (chương trình *C++*) để xuất dữ liệu ra thiết bị xuất chuẩn (mặc định là màn hình). Để nhập dữ liệu từ thiết bị nhập chuẩn (mặc định là bàn phím):

- Người lập trình *C* có thể dùng hàm ***scanf***. Ví dụ để nhập bán kính hình tròn và lưu vào biến ***R*** có kiểu ***float***, ta có thể gọi hàm ***scanf("%f", &R)***; cũng giống như trong bảng 2.3 định dạng ***%f*** được dùng cho số thực kiểu ***float***.
- Người lập trình *C++* có thể dùng đối tượng ***cin***. Ví dụ để nhập bán kính hình tròn và lưu vào biến ***R*** có kiểu ***float***, ta có thể ghi lệnh ***cin*** >> ***R*** trong chương trình đang viết.

	Chương trình C	
1	/* Sum.c */	// Sum.cpp
2	#include <stdio.h>	#include <iostream>
3		using namespace std;
4	void main()	void main()
5	{	{
6	int A, B;	int A, B;
7	int sum;	int sum;
8	printf("Gia tri cua A=");	cout << "Gia tri cua A=";
9	scanf("%d", &A);	cin >> A;
10	printf("Gia tri cua B=");	cout << "Gia tri cua B=";
11	scanf("%d", &B);	cin >> B;
12	sum = A + B;	sum = A + B;
13	printf("Tong so = %d\n", sum);	cout << "Tong so = "
14	}	<< sum << endl ;
		}

Bảng 2.5 Chương trình nhập và tính tổng hai số

Bảng 2.5 trình bày chương trình nhập hai số nguyên lưu vào trong các biến ***A*** và ***B*** (kiểu dữ liệu ***int***) tính tổng của hai số này, lưu vào biến ***sum*** (cũng có kiểu là ***int***), sau cùng in ra kết quả. Ý nghĩa các dòng trong chương trình này như sau:

- Các dòng 1- 5 có nghĩa tương tự như các chương trình trước ;
- Dòng 6: Kkhai báo hai biến ***A***, ***B*** có kiểu ***int*** ;
- Dòng 7: Khai báo biến ***sum*** có kiểu ***int*** ;
- Dòng 8: In ra dòng chữ "*Gia tri của A=*" ;
- Dòng 9: Đợi người sử dụng chương trình nhập vào một số nguyên và lưu giá trị số đó vào biến ***A*** ;
- Dòng10: In ra dòng chữ "*Gia tri của B=*" ;
- Dòng 11: Đợi người sử dụng chương trình nhập vào một số nguyên và lưu giá trị số đó vào biến ***B*** ;
- Dòng 12: Tính tổng hai số lưu trong các biến ***A, B*** và lưu kết quả vào biến ***sum*** ;
- Dòng 13: In ra giá trị của biến ***sum***.

Để in ra một dòng chữ thì dòng chữ đó phải được đặt trong dấu nháy kép. Để in ra giá trị của một biến thì biến đó không được đặt trong dấu nháy kép. Ngoài ra, khi sử dụng lệnh ***printf*** của ngôn ngữ *C* thì phải qui định định dạng ***%d***, ***%f*** tùy theo biến có kiểu là ***int***, ***float***.

III. CÁC KIỂU DỮ LIỆU CƠ SỞ VÀ PHÉP TOÁN

Mỗi ngôn ngữ lập trình đều có một hệ thống các kiểu dữ liệu cơ sở cùng với các phép toán để người lập trình có thể thực hiện các tính toán và dựa vào kiểu dữ liệu cơ sở để xây dựng những kiểu dữ liệu phức tạp hơn trong quá trình viết chương trình.

III.1 Kiểu số nguyên

Ngôn ngữ lập trình *C/C++* cung cấp các kiểu số nguyên (với nhiều kích thước lưu trữ khác nhau) cùng với những phép toán: phép tính cộng (dấu +), phép tính trừ (dấu -), phép nhân (dấu *), và phép chia (dấu / và dấu %). Trong đó, phép chia có thể tính thương số (với phép toán /) và tính dư số (với phép toán %).

Sau đây là các kiểu số nguyên trong *C/C++*, tùy theo tình huống mà người lập trình có thể chọn kiểu nguyên phù hợp nhất cho chương trình của mình. Riêng hai kiểu ***char*** và ***unsigned char*** cũng là kiểu ký tự, sẽ được giải thích rõ trong phần kế tiếp.

char

Kiểu số nguyên có dấu 1-byte hay 8-bit (lưu được các số nguyên có phạm vi từ –128= -2^7 đến 127 = $2^7 - 1$).

unsigned char

Kiểu số nguyên không dấu 1-byte (lưu được các số nguyên không âm phạm vi từ 0 đến 255 = $2^8 - 1$).

int

Số nguyên có dấu, thông thường có độ dài 16 bit hay 32 bit tùy thuộc vào hệ điều hành. Lưu các số có phạm vi từ -2^{n-1} đến $2^{n-1} - 1$ với n=16 hay 32. Hiện nay n thường là 32 đối với *C++*.

unsigned int

Số nguyên không dấu, thông thường có độ dài 16 bit hay 32 bit tùy thuộc vào hệ điều hành. Lưu được các số có phạm vi từ 0 đến $2^n - 1$ với *n*=16 hay 32. Hiện nay *n* thường là 32 đối với *C++*.

short

Số nguyên có dấu, độ dài 16-bit, mỗi biến lưu được các số có phạm vi từ -32768 = -2^{15} đến 32767 = $2^{15} - 1$.

unsigned short

Số nguyên không dấu, độ dài 16-bit, mỗi biến lưu được các số nguyên không âm có phạm vi từ 0 đến 65535 = $2^{16} - 1$.

long

Số nguyên dài có dấu 32-bit, mỗi biến lưu được các số có phạm vi từ –2.147.483.648 = -2^{31} đến 2.147.483.647 = $-2^{31} - 1$ (giá trị tuyệt đối khoảng 2,1 tỷ).

unsigned long

Số nguyên dài không dấu 32-bit, lưu được các số nguyên không âm có phạm vi từ 0 đến 4.294.967.295 = $2^{32} - 1$ (khoảng 4,3 tỷ).

Hằng số nguyên trong *C/C++* có thể ghi bằng một trong ba dạng: bát phân (bắt đầu bằng số 0), thập lục phân (bắt đầu bằng 0x), thập phân (ghi bắt đầu bằng số từ 1 đến 9). Ví dụ,

- Số *0125* nghĩa là $1 \times 8^2 + 2 \times 8^1 + 5 = 85$;
- Số *0x2A* nghĩa là $2 \times 16^1 + 10 = 42$;
- Số *028* không hợp lệ vì số hệ bát phân chỉ dùng các chữ số từ *0* đến *7*.

Ngoài bốn phép toán số học nói trên, *C/C++* cũng cho phép thực hiện các phép toán trên từng *bit* của các biến thuộc kiểu nguyên không dấu (như ***unsigned long*** hay ***unsigned short***…). Có những phép toán *bit*

như sau: & (phép *and bit*), | (phép *or bit*), ^ (phép *xor bit*), ~ (phép *not bit*), << (phép dịch trái, *left shift*), >> (phép dịch phải, *right shift*). Đối với những phép toán *bit*, mỗi số nguyên không dấu được đưa về biểu diễn nhị phân, sau đó các tính toán được thực hiện trên từng *bit* của số tham gia vào phép toán đang thực hiện.

- Phép & (phép *and bit*): để tính ***A*** & ***B***, từng cặp *bit* trong biểu diễn nhị phân của ***A*** và ***B*** từ phải sang trái được thực hiện phép *and* (chỉ có trường hợp cả hai *bit* đều là 1 thì kết quả mới là 1, ngược lại thì kết quả là 0).
- Phép | (phép *or bit*): để tính ***A*** | ***B***, từng cặp bit trong biểu diễn nhị phân của ***A*** và ***B*** từ phải sang trái được thực hiện phép *or* (chỉ có trường hợp cả hai bit đều là 0 thì kết quả mới là 0, ngược lại thì kết quả là 1).
- Phép ^ (phép *xor bit*): để tính ***A*** ^ ***B***, từng cặp bit trong biểu diễn nhị phân của ***A*** và ***B*** từ phải sang trái được thực hiện phép *xor* (hoặc loại trừ, chỉ có trường hợp hai bit khác nhau thì kết quả mới là 1, ngược lại kết quả là 0).
- Phép ~ (phép *not bit*): để tính ~***A***, các bit 0 của ***A*** được đổi thành 1, trong khi các bit 1 được đổi thành 0.
- Phép << (phép dịch trái): ***A*** << ***n*** nghĩa là dịch từng bit của ***A*** sang trái ***n*** bit, các bit dịch ra ngoài phạm vi lưu trữ xem như bị mất. Biểu thức ***A*** << ***n*** tương tự như $\boldsymbol{A*2^n}$.
- Phép >> (phép dịch phải): ***A*** >> ***n*** nghĩa là dịch từng bit của ***A*** sang phải ***n*** *bit*, các *bit* dịch ra ngoài phạm vi lưu trữ xem như bị mất. Biểu thức ***A*** >> ***n*** tương tự như $\boldsymbol{A/2^n}$ (phép chia nguyên, chỉ lấy thương số, bỏ đi phần dư).

Bảng 2.6 trình bày ví dụ về việc thực hiện các phép toán trên *bit* cho các hai biến ***A*** và ***B***, kiểu ***unsigned char*** có độ dài 8 *bit*. Các phép toán trên bit ***không cho kết quả giống với các phép toán cộng, trừ, nhân, chia thông thường*** (nếu có giống nhau thì chỉ là tình cờ trong các trường hợp đặc biệt). Những phép toán này sẽ được áp dụng khi các lập trình viên muốn lập trình thao tác trên các *bit* của dữ liệu hay

muốn tăng tốc độ xử lý của chương trình trong một vài tình huống nhất định.

Biểu thức	Giá trị	Biểu diễn nhị phân (dạng 8 bit)
A (kiểu **unsigned char**, 8 bit)	45	00101101
B (kiểu **unsigned char**, 8 bit)	58	00111010
A & B	40	00101000
A | B	63	00111111
A ^ B	23	00010111
~A	210	11010010
A << 4	208	11010000
A >> 4	2	00000010

Bảng 2.6 Ví dụ về phép toán trên bit với toán hạng độ dài 8 bit

Bảng 2.7 là chương trình minh họa việc tính toán các phép toán *bit* cho trường hợp các biến ***A***, ***B***, ***result*** có độ dài 8 *bit*. Biến ***result*** (khai báo ở dòng 5) lần lượt lưu kết quả tính toán (các dòng 6, 8, 10, 12, 14, 16) và in kết quả ra thiết bị xuất mặc định. Đối với chương trình *C++*, do cơ chế kiểm tra kiểu chặt chẽ hơn *C* nên biến ***result*** có kiểu ***unsigned char*** (ký tự) phải chuyển sang số nguyên không dấu 8 *bit* bằng cách thay ***result*** bởi biểu thức ***int (result)***. Kết quả xuất của chương trình này giống như các giá trị (dạng thập phân) trong cột 2 của bảng 2.6.

Dòng	Chương trình C
1	#**include** <stdio.h>
2	
3	**void** main()
4	{ **unsigned char** A=45, B=58;
5	**unsigned char** result;
6	result = A & B ;
7	printf("Ket qua A & B: %d\n", result) ;
8	result = A | B ;
9	printf("Ket qua A | B: %d\n", result) ;
10	result = A ^ B ;
11	printf("Ket qua A ^ B: %d\n", result) ;
12	result = ~A ;
13	printf("Ket qua ~A: %d\n", result) ;
14	result = A<<4 ;
15	printf("Ket qua A<<4: %d\n", result) ;
16	result = A>>4 ;
17	printf("Ket qua A>>4: %d\n", result) ;
18	}

Dòng	Chương trình C++
1	#**include** <iostream>
2	**using namespace** std;
3	**void** main()
4	{ **unsigned char** A=45, B=58;
5	**unsigned char** result;
6	result = A & B ;
7	**cout** << "Ket qua A & B:" << **int**(result) << **endl**;
8	result = A | B ;
9	**cout** << "Ket qua A | B:" << **int**(result) << **endl**;
10	result = A ^ B ;
11	**cout** << "Ket qua A ^ B:" << **int**(result) << **endl**;
12	result = ~A ;
13	**cout** << "Ket qua ~A:" << **int**(result) << **endl**;
14	result = A<<4 ;
15	**cout** << "Ket qua A<<4:" << **int**(result) << **endl**;
16	result = A>>4 ;
17	**cout** << "Ket qua A>>4:" << **int**(result) << **endl**;
18	}

Bảng 2.7 Chương trình minh họa phép toán bit

III.2 Kiểu số thực

Kiểu số thực cho phép thực hiện những phép toán: cộng (dấu +), trừ (dấu -), nhân (dấu *), và chia (dấu /). Mặc dù sử dụng cùng một ký hiệu phép toán chia là dấu / giống như phép tính thương số của kiểu số nguyên, phép chia số thực cho ra giá trị thực kể cả phần lẻ nếu không chia hết. Chẳng hạn 3/4 sẽ ra kết quả là 0 (vì là phép chia số nguyên), trong khi đó 3.0/4 sẽ ra kết quả là 0.75 (vì là phép chia số thực). Ngôn ngữ *C/C++* có sẵn các kiểu số thực với độ chính xác và độ lớn khác nhau như sau:

float. Kiểu thực 32-bit (4 byte) có độ chính xác đơn, số trị tuyệt đối nhỏ nhất xấp xỉ 1,401298 x 10^{-45} (đây cũng là độ chính xác, không thể chia nhỏ hơn), số lớn nhất cỡ khoảng 3,40282 x 10^{38} ;

double. Kiểu thực 64-bit (8 byte) có độ chính xác kép, số trị tuyệt đối nhỏ nhất là 4,94066x10^{-324} (đây cũng là độ chính xác, không thể chia nhỏ hơn), số lớn nhất cỡ khoảng 1,79769x10^{308}.

Trong các cài đặt tương lai của ngôn ngữ *C/C++*, các giá trị của thể của độ chính xác và độ lớn có thể thay đổi khác hơn so với các giá trị như trên.

Ngoài ra kiểu ***long double*** được thiết kế để lưu trữ số thực lớn 80-bit (10 byte) và có độ chính xác rất cao, số lớn nhất cỡ khoảng 3,4x10^{4932}. Tuy nhiên hiện nay đa số môi trường lập trình thường đồng nhất kiểu này với kiểu ***double***, vì vậy kiểu này ít được sử dụng trong lập trình ứng dụng. Cấu trúc lưu trữ bên trong (cách bố trí và ý nghĩa của các bit) của số thực được thiết kế theo chuẩn số chấm động (*floating point*) của IEEE. *Phụ lục A* sẽ trình bày chi tiết về chuẩn lưu trữ này.

Khi ghi hằng số thực trong chương trình *C/C++* thì mặc định được hiểu là số kiểu ***double***, nếu muốn ghi hằng số kiểu ***float*** thì qui ước viết tận cùng bằng chữ **F**. Chẳng hạn, ghi ***321.45*** thì hiểu là số ***double*** (trình dịch sẽ dành ô nhớ 8 byte để lưu trữ hằng số này), còn nếu ghi là ***321.45F*** thì hiểu là số ***float*** (trình dịch sẽ dành ô nhớ 4 byte để lưu trữ hằng số này). Việc nhập xuất trong ngôn ngữ *C* được thực hiện nhờ dùng các hàm ***printf*** và ***scanf*** với định dạng ***%f***, ***%e*** (sẽ được trình bày chi tiết trong phần định dạng nhập xuất ở cuối chương này).

Ngoài bốn phép tính số học, việc tính toán trên số thực có thể thực hiện nhờ vào các hàm toán học như: *căn số*, *lũy thừa*, *logarit* và các hàm lượng giác. Phần **V.2** sẽ trình bày chi tiết về cách sử dụng các hàm này trong lập trình.

III.3 Kiểu luận lý

Kiểu luận lý (*logic*) trong ngôn ngữ lập trình C/C++ là kiểu nguyên đặc biệt, mỗi biểu thức luận lý khác 0 nghĩa là *đúng* (**true**), bằng 0 nghĩa là *sai* (**false**). Ngoài ra, việc lượng giá của một biểu thức luận lý bất kỳ (thực hiện bởi ngôn ngữ lập trình *C/C++*) ***luôn cho một trong hai kết quả là 0 hay 1***. Người lập trình có thể thực hiện các phép toán sau đây trong biểu thức luận lý:

- Phép *logic and* (và): ký hiệu phép toán là &&, biểu thức **A && B** chỉ đúng khi cả **A** và **B** đều đúng.
- Phép *logic or* (hay): ký hiệu phép toán là ||, biểu thức **A || B** chỉ sai khi cả A và B đều sai.
- Phép *logic not* (phủ định): ký hiệu phép toán là !, biểu thức **!A** chỉ đúng khi **A** sai.

Trong trường hợp cần phải lưu trữ giá trị của một biểu thức luận lý, người lập trình có thể khai báo biến kiểu ***bool*** (đối với *C++* chuẩn) hoặc dùng bất kỳ một kiểu dữ liệu nguyên khác (như ***int***, ***long*** hay ***short***). Bảng 2.8 tóm tắt qui định về cách lượng giá các biểu thức luận lý trong ngôn ngữ lập trình *C/C++*.

A	B	A && B	A || B	!A
khác 0	khác 0	1	1	0
khác 0	0	0	1	0
0	khác 0	0	1	1
0	0	0	0	1

Bảng 2.8 Lượng giá các phép toán luận lý

Chú ý: Lưu ý phân biệt ký hiệu phép toán luận lý với phép toán trên bit, chẳng hạn như phân biệt giữa phép && (*logic and*) và & (*bit and*): kết quả tính toán biểu thức và ý nghĩa hoàn toàn khác nhau. Nói chung, trình biên dịch không thể phát hiện được khi người lập trình nhầm lẫn *phép toán logic* với *phép toán bit*, vì vậy sự sai sót thuộc loại này sẽ dẫn đến chương trình chạy sai và có thể rất khó phát hiện ra lỗi.

Các biểu thức thuộc mọi kiểu dữ liệu cơ bản (*số nguyên*, *số thực*, *ký tự*, *luận lý*…) đều thể so sánh với nhau nhờ các phép so sánh:

- Phép so sánh xem có bằng nhau hay không, ký hiệu phép toán là == (hai dấu bằng viết liền nhau).
- Phép so sánh xem có khác nhau hay không, ký hiệu phép toán là != (dấu chấm thang viết ngay phía trước dấu bằng).
- Phép so sánh lớn hơn, ký hiệu phép toán là >**,** biểu thức *A*>*B* đúng (có giá trị 1) nếu *A* lớn hơn *B*.
- Phép so sánh lớn hơn hay bằng, ký hiệu phép toán là >= (dấu lớn hơn kế đến là dấu bằng viết liền nhau) **,** biểu thức *A*>=*B* đúng (có giá trị 1) nếu *A* lớn hơn hay bằng *B*.
- Phép so sánh nhỏ hơn, ký hiệu phép toán là <**,** biểu thức *A*<*B* đúng (có giá trị 1) nếu *A* nhỏ hơn *B*.
- Phép so sánh nhỏ hơn hay bằng, ký hiệu phép toán là <= (dấu nhỏ hơn kế đến là dấu bằng viết liền nhau)**,** biểu thức *A*<=*B* đúng (có giá trị 1) nếu *A* nhỏ hơn hay bằng *B*.

Nếu *A* và *B* đều là hai biểu thức luận lý thì *A!=B* cũng có nghĩa là phép "*hoặc loại trừ*"(xor, *logically exclusive or*) của *A* và *B* có chân trị đúng chỉ khi nào *A* và *B* có chân trị khác nhau.

Bảng 2.9 minh họa chương trình có thực hiện các phép tính luận lý. Biến *bVal* (có kiểu **bool**) lần lượt lưu giá trị của các biểu thức luận lý khác nhau và được in ra ở các dòng 8, 10, 12. Một vài điểm cần lưu ý trong chương trình này như sau:

- Ở dòng 7: biến ***bVal*** nhận giá trị của biểu thức (***x == y***) trường hợp này sẽ có giá trị 0 (**false**) bởi vì ***x*** khác ***y***;

- Ở dòng 9: biến ***bVal*** nhận giá trị của biểu thức (***x*** < ***y***) trường hợp này sẽ có giá trị 1 (**true**) bởi vì ***x*** nhỏ hơn ***y***;
- Ở dòng 11: biến ***bVal*** nhận giá trị của biểu thức (*2*x* > ***y***) trường hợp này sẽ có giá trị 1 (**true**) bởi vì ***2*x*** là **93.4** lớn hơn **y=93**;

Riêng dòng 13, trường hợp này biểu thức bên phải của ***z*** là biểu thức số thực (không phải là biểu thức luận lý). Biểu thức luận lý (***x*** > ***y***) được nhân với ***x***, ra kết quả là một số thực và cộng với biểu thức ***(x<=y)*y*** cũng là số thực. Về mặt ngữ nghĩa thì biểu thức bên phải của ***z*** luôn là số lớn hơn trong hai số ***x*** và ***y*** (tức là ***z = max(x, y)***) nên ***z*** sẽ nhận giá trị là **93**.

	Chương trình C	Chương trình C++
1	#**include** <stdio.h>	#**include** <iostream>
2		**using namespace** std;
3	**void** main()	**void** main()
4	**{**	**{**
5	**bool** bVal;	**bool** bVal;
6	**double** x=46.7, y=93, z;	**double** x=46.7, y=93, z;
7	bVal = (x==y);	bVal = (x==y);
8	printf("%d\n", bVal) ;	**cout** << bVal << **endl**;
9	bVal = (x<y);	bVal = (x<y);
10	printf("%d\n", bVal) ;	**cout** << bVal << **endl**;
11	bVal = (2*x>y);	bVal = (2*x>y);
12	printf("%d\n", bVal) ;	**cout** << bVal << **endl**;
13	z = (x>y)*x + (x<=y)*y;	z = (x>y)*x + (x<=y)*y;
14	printf("%f\n", z) ;	**cout** << "z = " << z ;
15	**}**	**}**

Bảng 2.9 Chương trình minh họa phép toán luận lý

III.4 Kiểu ký tự

Ngôn ngữ *C/C++* cung cấp hai kiểu cho việc lưu trữ các ký tự: kiểu ***char*** hay ***unsigned char*** dùng để lưu các ký tự 1 byte (8 bit) và kiểu ***wchar_t*** dùng để lưu các ký tự 2 byte (16 bit).

- Ký tự 8 bit dựa trên bảng mã **ASCII** (mỗi ký tự hay ký hiệu có một mã số từ 0 cho đến 255), các biến ký tự kiểu ***char*** (hay

unsigned char) lưu mã **ASCII** của các ký tự. Chẳng hạn, hai lệnh gán ***ch*=65** và ***ch*='A'** là như nhau vì mã **ASCII** của ký tự **'A'** là **65**.

- Ký tự *Unicode* dạng 16 *bit* được lưu trữ dựa trên bảng mã quốc tế **UTF-16** (một dạng mã *Unicode*) có khả năng biểu diễn khoảng 65000 ký hiệu khác nhau. Để lưu một ký tự **UTF-16** cần phải dùng biến kiểu ***wchar_t***, chiếm 2 byte bộ nhớ máy tính.

Đối với trường hợp ký tự 8 bit, việc thao tác hay tính toán trên các biến ký tự khá đơn giản vì sự đồng nhất mỗi ký tự với mã **ASCII** của nó. Chẳng hạn, nếu ***ch*** là một biến ký tự 8 bit (kiểu ***char***) thì có thể chuyển ***ch*** (nếu ***ch*** là một ký tự thường) thành ký tự hoa tương ứng bằng công thức ***ch* = *ch* – ('a' – 'A')**. Do đó ta có công thức chuyển đổi:

$$ch \text{ (mới)} = \begin{cases} ch - ('a' - 'A') & if \quad 'a' \le ch \le 'z' \\ ch & otherwise \end{cases}$$

Trong mọi trường hợp thì vế phải tương đương với công thức:

***ch* - ('a' - 'A')*(*ch*>='a' && *ch*<='z')**.

Hoàn toàn tương tự ta cũng có công thức để chuyển ký tự lưu trong biến ***ch*** thành ký tự thường (nếu nó đang là ký tự hoa) như sau:

***ch* + ('a' - 'A')*(*ch*>='A' && *ch*<='Z')**.

Bảng 2.10 minh họa chương trình đơn giản về ký tự 8 bit. Dòng 6 và dòng 8 của chương trình gán trị biến ký tự ***ch*** bằng hai cách khác nhau: sử dụng mã **ASCII** và dùng trực tiếp ký tự (đặt trong dấu nháy đơn). Tuy nhiên kết quả in ra thiết bị xuất (các lệnh ở dòng 7 và dòng 9) luôn là như nhau. Đối với chương trình *C*, định dạng ***%c*** trong các hàm ***printf*** và ***scanf*** dùng để xuất và nhập các biến kiểu ký tự.

Dòng 11 chờ người dùng nhập vào ký tự mới và mã **ASCII** của được in ra ở dòng 12. Đối với chương trình *C*, ta dùng định dạng ***%d*** (dạng số nguyên) để in mã **ASCII**; trong khi đối với chương trình *C++*: ta ghi ***int(ch)*** để in ra mã **ASCII** của ký tự thay vì in ra ký tự. Dòng 13 chuyển ký tự ***ch*** sang dạng chữ hoa (nếu ***ch*** đang là chữ thường) theo công thức chuyển đổi đã thiết lập ở trên.

Dòng	Chương trình C
1	#**include** <stdio.h>
2	
3	**void** main()
4	**{**
5	**char** ch;
6	ch=65;
7	printf("ch = **%c\n**", ch) ;
8	ch='A';
9	printf("ch = **%c\n**", ch) ;
10	printf("ch = ") ;
11	scanf("**%c**", &ch) ; // nhập ký tự trong chương trình *C*
12	printf("ASCII code = **%d\n**", ch) ;
13	ch -= ('a' - 'A')*(ch>='a' && ch<='z') ;
14	printf("Upper case: **%c\n**", ch) ;
15	**}**

Dòng	Chương trình C++
1	#**include** <iostream>
2	**using namespace** std;
3	**void** main()
4	**{**
5	**char** ch;
6	ch=65;
7	**cout** << "ch = " << ch << **endl**;
8	ch='A';
9	**cout** << "ch = " << ch << **endl**;
10	**cout** << "ch = ";
11	**cin** >> ch; // nhập ký tự trong chương trình *C++*
12	**cout** << "ASCII code = " << **int**(ch) << **endl**;
13	ch -= ('a' - 'A')*(ch>='a' && ch<='z');
14	**cout** << "Upper case:" << ch << **endl**;
15	**}**

Bảng 2.10 Chương trình minh họa ký tự 8 bit

Mặc dù trong bảng mã **UTF-16** mỗi ký tự chiếm 2 byte, mã **UTF-16** của mỗi ký tự thông thường (như '0', '1' đến '9', 'a' đến 'z', 'A' đến 'Z'…) trùng với mã **ASCII** tương ứng. Do đó, nếu biến ***ch*** (kiểu ***char***) đã lưu một ký tự theo mã **ASCII** và biến ***wch*** có kiểu ***wchar_t***, thì có thể chuyển ***ch*** sang ***wch*** bằng một trong ba lệnh như sau:

- Cách 1: wch = ***wchar_t (****ch****)*** ;
- Cách 2: wch = ***(wchar_t)*** *ch* ; // cách làm theo *C++*
- Cách 3: wch = ch.

Hằng ký tự **UTF-16** được đặt trước bằng chữ ***L***, ví dụ ***L'B'***. Về mặt giá trị thì ***'B'*** và ***L'B'*** như nhau (cùng là 66) nhưng kích thước lưu trữ trong bộ nhớ thì khác nhau: ***sizeof*****('B')** luôn là 1 (trong *C++*) trong khi ***sizeof*****(L'B')** lại có giá trị là 2.

III.5 Phép gán và lệnh viết ngắn

Việc tính toán trong chương trình được thực hiện bằng cách tính toán và chép kết quả tính toán vào một biến nằm bên trái của phép gán. Ví dụ:

- Lệnh ***sum = A + B***: tính tổng các giá trị đang lưu trữ trong các biến ***A***, ***B*** và chép kết quả vào biến ***sum***;
- Lệnh ***sum = sum + 2***: lấy giá trị đang có trong biến ***sum*** cộng với 2 và ghi kết quả mới trở lại vào biến ***sum***;
- Lệnh ***sum = sum + n***: lấy giá trị đang có trong biến ***sum*** và biến ***n***, cộng với nhau và ghi kết quả mới trở lại vào biến ***sum***.

Ngôn ngữ lập trình *C/C++* cho phép viết các lệnh ngắn, nhờ đó người lập trình:

- Có thể viết ***sum++*** (hay ***++sum***) thay cho ***sum*** = ***sum****+1*;
- Hay viết ***sum += 2*** thay cho lệnh gán ***sum*** = ***sum****+2*;
- Hay viết ***sum += n*** thay cho lệnh gán ***sum*** = ***sum+n***.

Đối với những phép toán khác (ngoại trừ phép toán luận lý) cũng có thể viết một cách tương tự, chẳng hạn như:

- Viết ***n--*** hay ***--n*** thay cho ***n = n – 1*** ;
- Viết ***p -= x*** thay cho ***p = p - x*** ;
- Viết ***n*** **<<= 2** thay cho ***n = n*** **<< 2**.

Ngoài ra, lập trình viên cũng có thể viết cô đọng hơn như sau:

- Lệnh gán ***n*** = ***m***++ sẽ tương đương với hai lệnh gán, ***n***=***m*** được thực hiện trước, sau đó tăng ***m*** lên 1 (tức là gán ***m*** = ***m***+**1**) ;
- Lệnh gán ***n*** = ++***m*** tương đương với tăng ***m*** lên 1 trước (tức là gán ***m*** = ***m***+**1**), rồi sau đó gán ***n*** = ***m*** (nghĩa là ***n*** và ***m*** bằng nhau, bằng với giá trị cũ của ***m*** tăng lên 1).

Lệnh cô đọng ***n*** = ***m***-- (***n*** bằng giá trị cũ của ***m*** còn ***m*** thì giảm đi 1) và ***n*** = --***m*** (cả ***n*** và ***m*** đều bằng giá trị cũ của ***m*** giảm đi 1) cũng có ý nghĩa tương tự các lệnh vừa nói trên.

Tuy nhiên, việc viết các lệnh cô đọng có thể làm cho chương trình khó đọc, khó bắt lỗi trong quá trình xây dựng các chương trình có kích thước mã nguồn lớn. Đương nhiên đối với những người mới học lập trình thì lại càng không nên viết các lệnh cô đọng.

III.6 Độ lớn, độ chính xác, vấn đề tràn số (overflow)

Thông thường, dữ liệu số lưu trữ trên máy tính luôn có sự giới hạn về cả hai chiều vĩ mô và vi mô. Trong các phần trình bày bên trên chúng ta đã thấy phạm vi giới hạn của mỗi kiểu dữ liệu: cụ thể đối với kiểu nguyên thì có số nhỏ nhất, số lớn nhất; còn đối với kiểu thực thì có giới hạn về số có trị tuyệt đối lớn nhất (*độ lớn*) và số dương nhỏ nhất (không thể chia nhỏ hơn, cũng có nghĩa là *độ chính xác* của số thực).

Hiện tượng tràn số (*overflow*) trong tính toán xảy ra trong trường hợp kết quả của phép tính quá lớn hay quá nhỏ (nói chung là trị tuyệt đối quá lớn), hoặc là trị tuyệt đối của số thực quá nhỏ, nhỏ hơn cả độ chính xác. Đây là trường hợp xảy ra đối với cả số nguyên và số thực.

Khi xảy ra hiện tượng tràn số thì kết quả của phép tính không còn đúng với kết quả mong đợi như trong trường hợp thông thường. Điều này đưa đến chương trình chạy sai, đôi khi rất kỳ quặc. Đây là lỗi sai về mặt kỹ thuật mặc dù các công thức tính toán đều chính xác. Vì vậy một trong những việc rất quan trọng của người lập trình là ước định các kết quả tính toán và chọn kiểu dữ liệu thích hợp để tránh được sự tràn số.

Đối với số thực, khi trị tuyệt đối của phép tính là số dương quá nhỏ, nhỏ hơn số thực dương nhỏ nhất có thể lưu trữ được thì mặc dù không tràn số nhưng các phép tính sẽ không đúng theo các tính chất thông

thường (chẳng hạn như chia đôi số nhỏ nhất sẽ có kết quả là 0, nhân cho 0,9 nhưng vẫn bằng với số cũ...).

Bảng 2.11 (chương trình *C*) và 2.11B (chương trình *C++*) là các chương trình minh họa cho trường hợp xảy ra các tính toán tràn số, kết quả chạy của chương trình này được trình bày trong bảng 2.12. Đối với các số thực, tùy theo định dạng kết xuất (số chữ số sau dấu chấm thập phân) mà các giá trị in ra sẽ được làm tròn theo qui tắc thông thường. Hai biến kiểu ***short*** là ***x*** và ***y*** được khai báo ở dòng 5 với giá trị tương ứng là 400 và 500, biến ***z*** nhận giá trị tích ***x**y*** (ở dòng 6), kết quả đúng phải là 200000 vượt quá phạm vi lưu trữ của số ***short*** nên giá trị của ***z*** là 3392 (như trong bảng 2.12), không đúng như mong đợi.

Xem dòng 9 và 10 của chương trình: các biến ***floatMin*** và ***floatMax*** nhận tương các giá trị $1{,}401298 \times 10^{-45}$ (số thực dương nhỏ nhất) và $3{,}40282 \times 10^{38}$ (số thực lớn nhất). Trong cách viết **1.401298E-45F** thì **E-45** là ký hiệu cho 10^{-45}, còn chữ **F** tận cùng ký hiệu cho số có kiểu ***float*** (nếu không ghi thì hiểu mặc định là số ***double***, nhưng chúng ta đang cần khởi gán giá trị cho biến ***floatMin*** có kiểu ***float***). Dòng lệnh 14 và 16 thay đổi giá trị của các biến: biến ***floatMin*** được nhân với **0,9** (sau đó được in ra nhờ dòng lệnh 15), biến ***floatMax*** được cộng thêm **10** (sau đó được in ra nhờ dòng lệnh 17). Tuy nhiên giá trị của các biến này vẫn giữ nguyên như cũ, không giống như các qui tắc tính toán thông thường.

Giá trị lưu trong biến ***floatMin*** khi chia cho 2 (dòng lệnh 19) thì trở thành 0. Nói chung, khi giảm giá trị này xuống ít hơn hay bằng 0,5 lần giá trị cũ thì sẽ giảm thành 0; còn nếu giảm xuống bằng **t** lần giá trị cũ với $0{,}5 < \mathbf{t} < 1$ thì biến ***floatMin*** vẫn bị trơ ra tại giá trị cũ, không hề thay đổi. Ở dòng lệnh 21, khi nhân biến ***floatMax*** lên hai lần thì sẽ tràn số và nhận giá trị vô cực (ký hiệu **1.#INF**). Các ví dụ tính toán cho các biến ***doubleMin*** và ***doubleMax*** trong trường hợp tràn số và vượt quá giới hạn độ chính xác của kiểu số thực ***double*** cũng tương tự như đối với trường hợp hợp kiểu số thực ***float***.

Ghi chú. So sánh hai chương trình *C* và *C++* tương ứng ở các bảng 2.11 và 2.11B ta thấy có điểm khác nhau về việc khai báo biến: các biến trong chương trình *C* phải được khai báo ở đầu khối chương trình

(dòng 5) và sau đó mới được gán giá trị ở các dòng 9, 10, 24, 25; trong khi đó thì chương trình *C++* có thể khai báo và khởi gán ngay ở các dòng 9, 10, 24, 25 mà không cần bắt buộc phải khai báo biến ở đầu khối chương trình.

	Chương trình C
1	#include <stdio.h>
2	
3	void main()
4	{
5	short x=400, y=500, z; float floatMin, floatMax; double doubleMin, doubleMax;
6	z=x*y;
7	printf("z = %d\n", z) ;
8	printf("----------\n") ;
9	floatMin=1.401298E-45F;
10	floatMax=3.40282E+38F;
11	printf("floatMin = %e\n", floatMin) ;
12	printf("floatMax = %e\n", floatMax) ;
13	printf("----------\n") ;
14	floatMin *= 0.9F; printf("\n");
15	printf("floatMin * 0.9 = %e\n", floatMin) ;
16	floatMax += 10;
17	printf("floatMax + 10 = %e\n", floatMax) ;
18	printf("----------\n") ;
19	floatMin /= 2;
20	printf("floatMin/2 = %e\n", floatMin) ;
21	floatMax *= 2;
22	printf("floatMax*2 = %e\n", floatMax);
23	printf("----------\n");
24	doubleMin=4.94066E-324;
25	doubleMax=1.79769E+308;
26	printf("doubleMin = %e\n", doubleMin) ;
27	printf("doubleMax = %e\n", doubleMax) ;
28	printf("----------\n") ;
29	doubleMin *= 0.9;
30	printf("doubleMin * 0.9 = %e\n", doubleMin) ;
31	doubleMax += 10;
32	printf("doubleMax + 10 = %e\n", doubleMax) ;
33	printf("----------\n") ;
34	doubleMin /= 2;
35	printf("doubleMin/2 = %e\n", doubleMin) ;
36	doubleMax *= 2;
37	printf("doubleMax*2 = %e\n", doubleMax) ;
38	}

Bảng 2.11 Chương trình có các phép tính bị tràn số

	Chương trình C++
1	#**include** <iostream>
2	**using namespace** std;
3	**void** main()
4	**{**
5	**short** x=400, y=500, z;
6	z=x*y;
7	**cout** << "z = " << z << **endl**;
8	**cout** << "----------" << **endl**;
9	**float** floatMin=1.401298E-45F;
10	**float** floatMax=3.40282E+38F;
11	**cout** << "floatMin = " << floatMin << **endl**;
12	**cout** << "floatMax = " << floatMax << **endl**;
13	**cout** << "----------" << **endl**;
14	floatMin *= 0.9F; **cout** << **endl**;
15	**cout** << "floatMin * 0.9 = " << floatMin << **endl**;
16	floatMax += 10;
17	**cout** << "floatMax + 10 = " << floatMax << **endl**;
18	**cout** << "----------" << **endl**;
19	floatMin /= 2;
20	**cout** << "floatMin/2 = " << floatMin << **endl**;
21	floatMax *= 2;
22	**cout** << "floatMax*2 = " << floatMax << **endl**;
23	**cout** << "----------" << **endl**;
24	**double** doubleMin=4.94066E-324;
25	**double** doubleMax=1.79769E+308;
26	**cout** << "doubleMin = " << doubleMin << **endl**;
27	**cout** << "doubleMax = " << doubleMax << **endl**;
28	**cout** << "----------" << **endl**;
29	doubleMin *= 0.9;
30	**cout** << "doubleMin * 0.9 = " << doubleMin << **endl**;
31	doubleMax += 10;
32	**cout** << "doubleMax + 10 = " << doubleMax << **endl**;
33	**cout** << "----------" << **endl**;
34	doubleMin /= 2;
35	**cout** << "doubleMin/2 = " << doubleMin << **endl**;
36	doubleMax *= 2;
37	**cout** << "doubleMax*2 = " << doubleMax << **endl**;
38	**}**

Bảng 2.11B Chương trình có các phép tính bị tràn số

```
z  = 3392
----------------------------
floatMin = 1.401298e-045
floatMax = 3.40282e+038
----------------------------
floatMin * 0.9 = 1.401298e-045
floatMax + 10 = 3.40282e+038
----------------------------
floatMin/2 = 0
floatMax*2 = 1.#INF
----------------------------
doubleMin = 4.94066e-324
doubleMax = 1.79769e+308
----------------------------
doubleMin * 0.9 = 4.94066e-324
doubleMax + 10 = 1.79769e+308
----------------------------
doubleMin/2 = 0
doubleMax*2 = 1.#INF
```

Bảng 2.12 Kết quả chạy của chương trình ở bảng 2.11

Một vấn đề của số thực là sự không chính xác do biểu diễn nhị phân xấp xỉ. Các số có phần lẻ thập phân khi đưa vào bộ nhớ máy tính sẽ được xấp xỉ thành các lũy thừa của 2 theo hệ nhị phân. Vì vậy một số

phép tính có kết quả rất hiển nhiên đối với tính nhẩm thông thường nhưng lại phải xấp xỉ gần đúng khi viết chương trình.

	Chương trình C
1	#**include** <stdio.h>
2	
3	**void** main()
4	{
5	**double** x=1.123, y=1.456;
6	**double** z=2.579; **double** a, b, c;
7	
8	**if** (x+y == z) // xét trường hợp x+y = z
9	printf(" x + y = z \n") ;
10	**else** { // trường hợp ngược lại, tức là x+y khác z
11	printf(" x + y != z \n") ;
12	printf(" x + y - z = %e\n", x + y - z) ;
13	}
14	
15	a=1.125; b=1.250;
16	c=2.375;
17	**if** (a+b == c) // xét trường hợp a+b = c
18	printf(" a + b = c \n") ;
19	**else** // trường hợp ngược lại, tức là a+b khác c
20	printf(" a + b != c \n") ;
21	}

Kết quả chạy chương trình:

```
x + y != z
x + y - z = -4.440892e-016
a + b = c
```

Bảng 2.13 Chương trình minh họa sự không chính xác khi tính toán số thực

Các bảng 2.13 (chương trình *C*) và 2.13B (chương trình *C++*) là chương trình minh họa về việc xấp xỉ số thực (chỉ thị ***if else*** sẽ được trình bày trong chương sau, bạn đọc có thể hiểu một cách tự nhiên dựa vào các chú giải trong chương trình). Hai biến ***x*** và ***y*** (khai báo ở dòng 5) có giá trị tương ứng là 1,123 và 1,456; biến ***z*** (khai báo ở dòng 6) có giá trị là 2,579 chính là tổng số (về mặt toán học) của ***x*** và ***y*** (tức là

x+y = z). Tuy nhiên khi ta chạy chương trình thì ***x+y ≠ z*** và hiệu số ***(x+y) – z*** khác 0, có giá trị là -4,44089x10^{-16} (xấp xỉ gần bằng 0). Đối với trường hợp ở dòng lệnh 15 và 16: biến ***a***=1,125 (có biểu diễn $1+\frac{1}{2^3}$ bằng đúng với 1,001 trong cơ số 2) và ***b***=1,250 (có biểu diễn $1+\frac{1}{2^2}$ bằng đúng với 1,01 trong cơ số 2); biến ***c*** (có biểu diễn $1+\frac{1}{2^2}+\frac{1}{2^3}$ bằng đúng với 10,011 trong cơ số 2). Việc lưu trữ các giá trị này không cần phải xấp xỉ nhị phân, vì vậy ***a+b = c*** luôn đúng cho cả trường hợp tính toán thông thường và trường hợp lưu trữ trong máy tính.

	Chương trình C++
1	#**include** <iostream>
2	**using namespace** std;
3	**void** main()
4	{
5	**double** x=1.123, y=1.456;
6	**double** z=2.579;
7	
8	**if** (x+y == z) // xét trường hợp x+y = z
9	**cout** << " x + y = z " << **endl**;
10	**else** { // trường hợp ngược lại, tức là x+y khác z
11	**cout** << " x + y != z " << **endl**;
12	**cout** << " x + y - z = " << x + y - z << **endl**;
13	}
14	
15	**double** a=1.125, b=1.250;
16	**double** c=2.375;
17	**if** (a+b == c) // xét trường hợp a+b = c
18	**cout** << " a + b = c " << **endl**;
19	**else** // trường hợp ngược lại, tức là a+b khác c
20	**cout** << " a + b != c " << **endl**;
21	}

Bảng 2.13B Sự không chính xác khi tính toán số thực

Trong các chương trình ứng dụng quan trọng, các lập trình viên thường rất cẩn thận khi so sánh các số thực: biểu thức ***a != b*** (***a*** ≠ ***b***, nghĩa là ***a*** khác ***b***) luôn được thay thế bởi ***fabs(a – b) < epsilon*** (tức là │***a – b***│< ***epsilon***, nghĩa là trị tuyệt đối của hiệu ***a*** và ***b*** khá nhỏ). Trong đó ***epsilon*** là một hằng số thực khá nhỏ, chẳng hạn một phần vạn hay một phần triệu, có giá trị thay đổi tùy theo từng tình huống.

IV. THƯ VIỆN HÀM CÓ SẴN

Để tiết kiệm công sức**,** người lập trình có thể sử dụng các hàm có sẵn trong quá trình viết chương trình. Tập hợp các hàm được xây dựng sẵn của ngôn ngữ lập trình thường được gọi là ***thư viện hàm***. Hệ thống hàm trong thư viện hàm thường rất đa dạng, cách sử dụng và ý nghĩa có thể khác nhau, thông thường người lập trình cần phải tra cứu thêm tài liệu tham khảo hoặc hệ thống giúp đỡ của phần mềm hỗ trợ lập trình.

Trong phần này, chúng ta làm quen với các hàm đơn giản, dễ sử dụng, thao tác trên các biến thuộc kiểu dữ liệu cơ sở. Một số hàm khác (thao tác trên những loại dữ liệu có cấu trúc và liên quan đến những khái niệm nâng cao hơn trong ngôn ngữ lập trình...) sẽ được trình bày trong những chương sau khi đã trình bày các kiến thức thích hợp.

IV.1 Khái niệm về hàm trong lập trình

Nhu cầu sử dụng hàm trong quá trình lập trình hay làm phần mềm là rất cần thiết và không thể né tránh. Một số tính toán rất tự nhiên đối với yêu cầu bình thường của khách hàng nhưng đôi khi có thể quá khó hoặc quá tốn thời gian nếu như người lập trình viết chương trình mà không sử dụng hàm.

Xét một ví dụ đơn giản: để tính biểu thức ***F(x, y)*** = $x+\sqrt{1+y^2}$, với ***x*** và ***y*** là những số thực, chúng ta cần phải tính căn bậc hai của một số thực. Hiếm có một lập trình viên nào có thể viết trực tiếp một đoạn chương trình tính căn bậc hai bởi vì cần phải có những kiến thức nhất định về toán học. Tuy nhiên, hầu hết các lập trình viên lại biết rằng có thể dùng hàm ***sqrt( )*** một cách rất dễ dàng để tính căn bậc hai của số thực.

Bảng 2.14 trình bày chương trình minh họa việc sử dụng hàm ***sqrt( )*** để tính căn hai của số thực. Đây là hàm có sẵn trong thư viện ***<math.h>*** đối với ngôn ngữ *C* hay ***<cmath>*** đối với ngôn ngữ *C++*. Thư viện hàm toán học được khai báo sử dụng ở dòng 1 của chương trình. Ở dòng 11 của chương trình, việc tính giá trị $\sqrt{1+y^2}$ được thực hiện rất đơn giản bằng việc viết biểu thức ***sqrt(1 + y*y)*** giống y hệt như một công thức toán học: chỉ cần thay ký hiệu căn bậc hai bằng ***sqrt*** và sử dụng dấu ngoặc một cách thích hợp. Việc sử dụng hàm ***sqrt( )*** như ở dòng 11 được nói là "*gọi hàm*".

	Chương trình C	Chương trình C++
1	`#include <math.h>`	`#include <cmath>`
2	`#include <stdio.h >`	`#include <iostream>`
3		`using namespace std;`
4	`void main()`	`void main()`
5	`{`	`{`
6	`  double x, y, Fxy;`	`  double x, y, Fxy;`
7	`  printf("x = ");`	`  cout << "x = ";`
8	`  scanf("%lf", &x);`	`  cin >> x;`
9	`  printf("y = ");`	`  cout << "y = ";`
10	`  scanf("%lf", &y);`	`  cin >> y;`
11	`  Fxy = x + sqrt(1 + y*y);`	`  Fxy = x + sqrt(1 + y*y);`
12	`  printf("F(x, y) = %lf", Fxy);`	`  cout << "F(x, y) = " << Fxy ;`
13	`}`	`}`

Bảng 2.14 Chương trình minh họa sử dụng hàm sqrt() tính căn bậc hai

IV.2 Các hàm toán học

Các hàm toán học đều có tham số kiểu ***double,*** giá trị nhập vào và kết quả tính toán đều có kiểu ***double***. Để sử dụng các hàm toán học, người lập trình cần ghi thêm chỉ thị ***#include <math.h>*** (đối với ngôn ngữ

C) hay ***#include <cmath>*** (đối với *C++* chuẩn[1]) vào đầu chương trình. Các hàm toán học thông dụng thường dùng khi viết chương trình được liệt kê trong bảng 2.15.

Hàm	Ý nghĩa
double sqrt**(double** x**) ;**	Tính căn bậc hai của số thực $x \geq 0$.
double pow**(double** x**, double** y**) ;**	Tính x lũy thừa y, x^y với $x>0$.
double exp**(double** x**) ;**	Tính e^x với e là cơ số logarit tự nhiên (với $e \approx 2{,}71828$).
double log**(double** x**) ;**	Tính logarit tự nhiên (logarit cơ số e) của x (với $e \approx 2{,}71828$).
double log10**(double** x**) ;**	Tính logarit cơ số 10 của x, $log_{10}(x)$.
int abs**(int** x**) ;** **long** labs**(long** x**) ;** **double** fabs**(double** x**) ;**	Tính trị tuyệt đối |x| với x nguyên (**int**), nguyên dài (**long**), số thực (**double**).
double cos**(double** x**) ;** **double** sin**(double** x**) ;** **double** tan**(double** x**) ;**	Các hàm lượng giác: tính *cos(x)*, *sin(x)*, *tg(x)* của góc x cho bằng đơn vị radian.
double acos**(double** x**) ;** **double** asin**(double** x**) ;** **double** atan**(double** x**) ;**	Các hàm lượng giác ngược: tính góc *arccos(x)*, *arcsin(x)*, arctg(x) khi đã biết trước giá trị lượng giác x.
double floor**(double** x**) ;** **double** ceil**(double** x**) ;**	Tính số nguyên lớn nhất không vượt quá x (phần nguyên x, tức là $[x]$), số nguyên nhỏ nhất lớn hơn hay bằng x, ký kiệu toán học là $\lceil x \rceil$.

Bảng 2.15 Các hàm toán học trong thư viện chuẩn của C/C++

Ta xét một ví dụ áp dụng các hàm toán học vào việc giải một phương trình bậc ba đặc biệt. Phương trình $y^3+3p^2y+2q = 0$ (với p và q cho trước) luôn có một nghiệm duy nhất là:

[1] Với ngôn ngữ C++ không chuẩn thì vẫn dùng chỉ thị **#include <math.h>**

$$y = \sqrt[3]{\sqrt{p^6 + q^2} - q} - \sqrt[3]{\sqrt{p^6 + q^2} + q}\,.$$

Dòng	Chương trình C
1	#**include** <math.h>
2	#**include** <stdio.h>
3	
4	**void** main()
5	**{**
6	**double** p, q, y, delta; **double** test;
7	printf("p = ");
8	scanf("%lf", &p);
9	printf("q = ");
10	scanf("%lf", &q);
11	delta = pow(p, 6.0) + pow(q, 2.0);
12	delta = sqrt (delta);
13	y = pow(delta-q, 1.0/3) - pow(delta+q, 1.0/3);
14	printf("Nghiem y = %lf\n", y);
15	test=y*y*y + 3*p*p*y + 2*q;
16	printf("y^3 + 3p^2 y + 2q = %lf\n", test) ;
17	}

Dòng	Chương trình C++
1	#**include** <cmath>
2	#**include** <iostream>
3	**using namespace** std;
4	**void** main()
5	**{**
6	**double** p, q, y, delta;
7	**cout** << "p = ";
8	**cin** >> p;
9	**cout** << "q = ";
10	**cin** >> q;
11	delta = pow(p, 6.0) + pow(q, 2.0);
12	delta = sqrt (delta);
13	y = pow(delta-q, 1.0/3) - pow(delta+q, 1.0/3);
14	**cout** << "Nghiem y = " << y << **endl**;
15	**double** test=y*y*y + 3*p*p*y + 2*q;
16	**cout** << "y^3 + 3p^2 y + 2q = " << test << **endl**;
17	}

Bảng 2.16 Chương trình sử dụng hàm toán học

Chương trình trong bảng 2.16 minh họa việc tính nghiệm của phương trình nói trên. Chương trình này sử dụng hai hàm ***pow( )*** và ***sqrt( )***. Trong đó, hàm ***pow( )*** được gọi ở dòng 11 để tính lũy thừa 6 (giá trị

của p^6) và bình phương (giá trị của q^2). Hàm ***sqrt( )*** ở dòng 12 được gọi để tính căn bậc hai (giá trị của $\sqrt{p^6+q^2}$). Sau đó ở dòng 13, hàm ***pow( )*** lại được sử dụng hai lần để tính căn bậc ba nhờ việc xem căn bậc ba như là lũy thừa $\frac{1}{3}$ (ta viết là 1.0/3, vì nếu viết 1/3 thì sẽ là 0 do phép chia số nguyên). Sau khi tính ra nghiệm y (nhờ dòng lệnh 13) và xuất ra (nhờ dòng lệnh 14), biến ***test*** (dòng lệnh 15) tính giá trị vế trái của phương trình và in ra (dòng lệnh 16) để thử lại xem nghiệm tính có chính xác không. Nếu nghiệm đúng thì giá trị in ra của ***test*** sẽ gần bằng 0.

IV.3 Các hàm ký tự

Người lập trình có thể gọi sử dụng các hàm ký tự liệt kê trong danh sách sau đây bằng cách dùng thư viện nhờ chỉ thị ***#include <ctype.h>***.

bool isupper(**char** ch);	Kiểm tra xem ký tự ***ch*** có là ký tự hoa hay không
char toupper(**char** ch);	Trả về ký tự hoa tương ứng với ***ch*** nếu ***ch*** là ký tự thường.
bool islower(**char** ch);	Kiểm tra xem ký tự ***ch*** có là ký tự thường hay không
char tolower(**char** ch);	Trả về ký tự thường tương ứng với ***ch*** nếu ***ch*** là ký tự hoa.
Các hàm *iswupper( ), towupper( ), iswlower( ), towlower( )* có tác dụng tương tự nhưng được dùng cho ký tự 16 bit, tức là kiểu *wchar_t*.	

V. ĐỊNH DẠNG DỮ LIỆU NHẬP XUẤT

V.1 Định dạng dữ liệu nhập xuất đối với ngôn ngữ *C*

Các định dạng nói chung:

- Ký tự đặc biệt trong chuỗi xuất: \\ (dấu \) và **%%** (dấu **%**).
- Ký tự *tab* và ký tự xuống dòng: ***\t***, ***\n***, ***\r***.
- Nhập xuất số nguyên dạng thập phân có dấu: ***%d*** và ***%i***.
- Nhập xuất số nguyên dạng thập phân không dấu: ***%u***.
- Nhập xuất số nguyên dạng thập lục phân (cơ số 16) không dấu: ***%x*** (xuất ra chữ thường) và ***%X*** (xuất ra chữ hoa).
- Nhập xuất số nguyên dạng bát phân (cơ số 8, *octal*) không dấu: **%o** (dấu % và kế đến là chữ **o** viết thường).
- Nhập xuất số thực chấm động dạng viết thập phân: ***%f*** (nhập xuất số ***float*** và xuất số ***double***).
- Nhập xuất số thực chấm động dạng viết số mũ (chữ ***e*** hay ***E*** thay cho cơ số 10, chẳng hạn 1.2E-8 nghĩa là 1,2 x 10^{-8}): ***%e*** và ***%E*** (nhập xuất số ***float*** và xuất số ***double***).
- Nhập xuất ký tự: ***%c***.
- Nhập xuất chuỗi ký tự: ***%s***.

Trường hợp nhập số ***long*** và số ***double***:

- Dùng ***%lf, %le, %lE*** cho số ***double***, thay chữ ***l*** bởi ***L*** cho số kiểu ***long double***.
- Dùng ***%ld, %li, %lu, %lx*** cho các số nguyên dài.

Trường hợp nhập xuất số nguyên ***short*** (16-bit): dùng ***%hd, %hi, %ho, %hu, %hx, %hX***.

Về việc qui định độ rộng và độ chính xác (nếu là số thực) cho dữ liệu xuất được ghi ngay sau dấu **%** với dạng ***wid.pre***, ví dụ ***%9.2f*** nghĩa là độ rộng ít nhất chín ký tự (thêm khoảng trống vào nếu thiếu) và nhiều nhất là hai ký tự cho phần lẻ sau dấu chấm thập phân.

V.2 Định dạng dữ liệu nhập xuất đối với ngôn ngữ C++

Việc nhập xuất dữ liệu bằng thiết bị nhập xuất chuẩn được thực hiện bởi các đối tượng ***cin*** (nhập dữ liệu) và ***cout*** (xuất dữ liệu) đã định nghĩa sẵn trong **<*iostream*>**. Toán tử >> (được gọi là *extraction*

operator) được dùng với ***cin*** khi nhập dữ liệu; toán tử << (được gọi là *insertion operator*) được dùng với ***cout*** để xuất kết quả tính toán.

Ngôn ngữ *C++* cung cấp một hệ thống định dạng dữ liệu nhập xuất cho thiết bị nhập xuất chuẩn. Hơn nữa, kỹ thuật định dạng dữ liệu này có thể dùng cho những trường hợp mở rộng hơn sau này như là: định dạng dữ liệu khi xuất lên một tập tin văn bản hay là đọc dữ liệu từ một tập tin văn bản.

Để định dạng dữ liệu, ta thêm chỉ thị ***#include <iomanip>*** vào đầu chương trình. Việc định dạng dữ liệu được thực hiện bằng các toán tử định dạng (được gọi là các *manipulator*). Ví dụ: ***endl*** là một toán tử định dạng có tác dụng xuống dòng mới khi chuyển ***endl*** đến một thiết bị dạng *stream* bằng cách dùng toán tử <<. Sau đây là một số toán tử định dạng đơn giản, nhưng thông dụng.

- Toán tử ***setw(n)*** có thể dùng để định độ rộng của dữ liệu xuất. Khi dùng ***setw(n)*** thì các khoảng trống (trường hợp mặc định) được thêm vào bên trái hay bên phải (để tổng số là ***n*** ký tự) để kết quả in ra được canh đều lề phải hay trái.
- Toán tử ***left*** và ***right*** được dùng chung với ***setw(n)*** để canh lề trái hay canh lề phải.
- Toán tử ***setfill(ch)*** dùng chung với ***set(w)*** để qui định ký tự ***ch*** được thêm vào thay vì dùng khoảng trống mặc định. Ví dụ nếu dùng ***setfill('.')*** thì dấu chấm sẽ dùng thay cho khoảng trống.
- Các toán tử ***dec*** (thập phân), ***oct*** (bát phân), ***hex*** (thập lục phân) được dùng để qui định số nguyên (khi nhập xuất) được ghi theo dạng thập phân, bát phân, hay thập lục phân.
- Toán tử ***setprecision(n)*** dùng để qui định độ chính xác khi in số thực: ***n*** là tổng số các chữ số.

Bảng 2.18 là chương trình và kết quả chạy minh họa cho việc sử dụng toán tử ***setw(n)*** để canh lề cho dữ liệu in ra. Trường hợp này mặc định là canh lề phải, các khoảng trống được thêm vào bên trái.

Bảng 2.19 minh họa chương trình sử dụng hai toán tử ***hex*** và ***oct*** để qui định việc nhập theo dạng thập lục phân (hệ đếm cơ số 16) và xuất theo dạng bát phân (hệ đếm cơ số 8). Dòng lệnh số 8 dùng toán tử ***hex*** với đối tượng ***cin*** để chờ người dùng cuối nhập vào một số theo dạng

biểu diễn thập lục phân và lưu trong biến ***n***. Dòng lệnh số 9 in trở lại giá trị đã nhập, nhưng dùng toán tử ***oct*** để in ra dưới dạng biểu diễn bát phân.

Dòng	Chương trình C++
1	#**include** <iostream>
2	#**include** <iomanip>
3	**using namespace** std;
4	**void** main()
5	**{**
6	**int** Area=970, Height=10, Volume=9700;
7	**cout** << **setw(8)** << "Area" << **setw(10)** << Area << **endl**;
8	**cout** << **setw(8)**<< "H" << **setw(10)** << Height << **endl**;
9	**cout** << **setw(8)**<< "Volume" << **setw(10)** << Volume<< **endl**;
10	**}**

Kết quả chạy chương trình:

```
Area        970
   H         10
Volume      9700
```

Bảng 2.18 Chương trình minh họa định dạng độ rộng của số và chuỗi

Dòng	Chương trình C++
1	#**include** <iostream>
2	#**include** <iomanip>
3	**using namespace** std;
4	**void** main()
5	**{**
6	**long** n;
7	**cout** << "n (hexadecimal) = ";
8	**cin** >> **hex** >> n;
9	**cout** << "Octal representation: " << **oct** << n << **endl**;
10	**cin** >> n;
11	**}**

Bảng 2.19 Chương trình định dạng số bằng các toán tử hex và oct

VI. KẾT CHƯƠNG

Chương này trình bày về cách viết những chương trình đơn giản nhất bằng cách sử dụng một ngôn ngữ lập trình, đồng thời cũng giới thiệu những kiểu dữ liệu cơ sở cùng với những phép toán và thao tác nhập xuất; giới thiệu về thư viện hàm có sẵn và cách gọi hàm. Một số điểm quan trọng cần lưu ý như sau.

- Việc đặt tên các biến trong chương trình ảnh hưởng tính dễ đọc của chương trình. Các biến trong chương trình được đặt xúc tích, gợi nhớ, gắn liền với ngữ nghĩa của đối tượng trong thế giới thực sẽ tạo điều kiện dễ dàng bảo trì và phát triển chương trình trong tương lai.
- Hầu hết những kiểu dữ liệu số đều có phạm vi lưu trữ giới hạn: điều này ảnh hưởng đến độ lớn và độ chính xác của số được lưu trữ. Việc tính toán trên các biến dữ liệu kiểu số có khả năng bị tràn số. Các lập trình viên cần hiểu rõ cơ chế này để kiểm soát được tính đúng đắn của chương trình.
- Vì lý do lịch sử, ký tự có nhiều dạng lưu trữ. Các biến ký tự *ASCII* lưu các ký tự độ dài 1 byte, các biến ký tự *UTF-16* lưu các ký tự có độ dài 2 byte. Ngoài ra cũng có dạng ký tự nhiều byte, mỗi ký tự chiếm từ 1 đến 4 byte.
- Hệ thống hàm có sẵn đóng vai trò rất quan trọng trong quá trình lập trình. Hầu hết các chương trình ứng dụng trong thực tế đều phải sử dụng hàm trong quá trình xây dựng nó. Vì vậy các lập trình viên cần phải trau dồi kỹ thuật sử dụng các hàm có sẵn, tận dụng tối đa hàm có sẵn khi viết chương trình.

VII. THUẬT NGỮ TIẾNG ANH

Sau đây là những thuật ngữ tiếng Anh liên quan đến những khái niệm đã dùng trong chương này.

ASCII code: mã ký tự theo chuẩn 1 byte. Bảng mã *ASCII* (*American Standard Code for Information Interchange*) có 256 ký tự (gồm cả ký tự thông thường và ký tự đặc biệt).

Character: ký tự nói chung;
wide character: ký tự 16 bit;
wide string: chuỗi ký tự gồm các ký tự 16 bit.

Constant: hằng số.

Data type: kiểu dữ liệu.

Floating-point, real data type: số thực dấu chấm động, kiểu dữ liệu số thực.

Function library: thư viện hàm.

Fundamental data type: kiểu dữ liệu cơ bản, cơ sở.

Input: nhập;
input data: dữ liệu nhập.

Integral data type, integer: kiểu dữ liệu nguyên;
long integer: kiểu nguyên dài (32 bit).

Operator: toán tử, phép toán ;
bit mask: mặt nạ bit ;
bit operator: phép toán trên bit ;
logical operator, boolean operator: phép toán luận lý.

Output: xuất;
output data: dữ liệu xuất.

Overflow: tràn số

Unicode: nói chung về ký tự unicode.

Variable: biến (dùng trong lập trình)
variable declaration: khai báo biến.

Chương 3
CÁC CẤU TRÚC ĐIỀU KHIỂN TRONG LẬP TRÌNH

I. GIỚI THIỆU

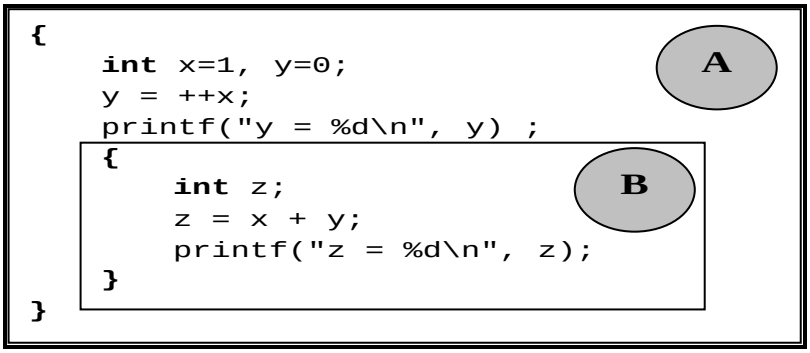
Những chương trình trong các ví dụ ở chương trước bao gồm các chỉ thị được thực hiện một cách tuần tự. Trong thực tế, để viết một chương trình hoàn chỉnh, lập trình viên cần phải sử dụng nhiều cấu trúc bổ sung thêm cho các chỉ thị tuần tự. Chẳng hạn như lặp lại một chỉ thị nào đó nhiều lần hay bỏ qua không thực hiện một vài chỉ thị không cần thiết nào đó.

Chương này trình bày và thảo luận cách sử dụng các cấu trúc điều khiển trong quá trình viết chương trình. Tất cả các ngôn ngữ lập trình đều hỗ trợ các cấu trúc điều khiển tương tự nhau. Việc hiểu rõ bản chất của mỗi cấu trúc điều khiển trong từng ngôn ngữ lập trình cụ thể là rất quan trọng đối với người lập trình.

Để khởi đầu cho việc làm quen với các cấu trúc điều khiển lập trình trong *ngôn ngữ lập trình C/C++*, trước hết chúng ta cần tiếp cận với các khái niệm: khối lệnh, phạm vi sử dụng biến, và khái niệm *namespace*.

I.1 Khối lệnh

Một khối lệnh (*block*) là một tập hợp các lệnh nằm trong một cặp dấu ngoặc nhọn: bắt đầu bởi dấu { và kết thúc bởi dấu }. Các khối lệnh có thể lồng nhau: một khối lệnh có thể nằm bên trong một khối lệnh khác. Hình 3.1 minh họa hai khối lệnh được chú giải là **A** và **B**; khối lệnh **B** nằm lồng bên trong khối lệnh **A**.



Hình 3.1: Các khối lệnh lồng nhau

Bên trong mỗi khối lệnh có thể khai báo các biến: những biến này được gọi là *biến cục bộ*. Chẳng hạn như *x*, *y* được khai báo, khởi động giá trị và sử dụng trong khối lệnh **A**; còn *z* được khai báo và sử dụng trong khối lệnh **B**. Trong ví dụ này thì cả ba biến x, y, z được gọi là biến cục bộ. Những biến không nằm trong bất kỳ một khối lệnh nào được gọi là *biến toàn cục*.

I.2 Phạm vi sử dụng của biến

Các biến có thể được khai báo một cách cục bộ bên trong mỗi khối lệnh. Biến trong một khối lệnh không thể sử dụng được ở phía bên ngoài của khối lệnh đó. Cần lưu ý thêm:

- Hai biến trong cùng một khối không thể trùng tên;
- Biến trong một khối lệnh **A** có thể được hiểu trong các khối lệnh nằm lồng bên trong **A**;
- Trong trường hợp trùng tên biến giữa biến bên trong một khối lệnh và biến nằm ở khối lệnh lồng bên ngoài: mặc định sẽ hiểu biến nằm trong phạm vi của khối lệnh gần nhất bao bọc vị trí sử dụng biến.

Bảng 3.1 trình bày đoạn chương trình *C* có sự trùng tên giữa các biến trong các khối lệnh lồng nhau; bảng 3.1B là chương trình *C++* tương ứng. Phía bên dưới là kết quả chạy chương trình. Biến ***a*** chỉ được khai báo một lần ở dòng 6 trong khối lệnh của hàm ***main( )***. Khối lệnh con (nằm từ dòng 10 đến dòng 15) không có bất kỳ biến nào tên là ***a***: vì

vậy lệnh gán ở dòng 12 sẽ hiểu là tác động lên biến ***a*** của khối lệnh chính; giá trị lưu trong ***a*** sẽ đổi thành 2001 (thay cho giá trị cũ 1984). Đối với trường hợp biến ***b***, có hai chỗ khai báo biến: dòng 6 là biến ***b*** của khối lệnh chính được khởi tạo giá trị 1988; trong khi dòng 11 là biến ***b*** của khối lệnh con được khởi tạo giá trị 1996. Bên trong của khối lệnh con (dòng 10 – 15) sẽ hiểu là biến ***b*** được khai báo ở dòng 11 (có giá trị 1996); bên trong của khối lệnh chính nhưng nằm ngoài khối lệnh con (tức là các dòng 5 – 9 và 16 – 18) thì hiểu là biến ***b*** khai báo ở dòng 6 (có giá trị 1988).

Dòng	Chương trình C
1	#**include** <stdio.h>
2	
3	**void** main()
4	**{**
5	**/* main block */**
6	**int** a=1984, b=1988;
7	printf(" a (main block)= %d\n", a) ;
8	printf(" b (main block)= %d\n", b) ;
9	**/* sub block */**
10	**{**
11	**int** b=1996;
12	a = 2001;
13	printf(" a (of main block is changed) = %d\n", a) ;
14	printf(" b (sub block) = %d\n", b) ;
15	**}**
16	printf("Now in main block:\n") ;
17	printf(" a (changed) = %d\n", a) ;
18	printf(" b (unchanged) = %d\n", b) ;
19	**}**

Kết quả chạy chương trình:

```
    a (main block)= 1984
    b (main block)= 1988
    a (of main block is changed) = 2001
    b (sub block) = 1996
Now in main block:
    a (changed) = 2001
    b (unchanged) = 1988
```

Bảng 3.1 Chương trình minh họa phạm vi sử dụng của biến

Dòng	Chương trình C++
1	#**include** <iostream>
2	**using namespace** std;
3	**void** main()
4	**{**
5	// main block
6	**int** a=1984, b=1988;
7	**cout** << " a (main block)= " << a << **endl** ;
8	**cout** << " b (main block)= " << b << **endl** ;
9	// sub block
10	**{**
11	**int** b=1996;
12	a = 2001;
13	**cout** << " a (of main block is changed) = " << a << **endl** ;
14	**cout** << " b (sub block) = " << b << **endl** ;
15	**}**
16	**cout** << "Now in main block:" << **endl** ;
17	**cout** << " a (changed) = " << a << **endl** ;
18	**cout** << " b (unchanged) = " << b << **endl** ;
19	**}**

Bảng 3.1B Chương trình C++ minh họa phạm vi sử dụng của biến

I.3 Khái niệm về *namespace*

Khái niệm *namespace* (tạm gọi là "*không gian tên*") của *C++* được sử dụng để đặt tên cho các khối lệnh nhằm để tổ chức các lớp, hàm, biến, đối tượng một cách có hệ thống và truy xuất đến chúng thông qua một tên được viết tường minh thay vì ngầm hiểu theo mặc định.

Chẳng hạn như các đối tượng ***cin*** và ***cout*** của ***<iostream>*** được đặt trong phạm vi của một *namespace* có tên là ***std***. Vì vậy khi sử dụng các đối tượng này có hai cách như sau:

- Phải dùng tường minh bằng tên của *namespace* như ***std::cin*** hay ***std::cout*** trong các lệnh nhập xuất. Tức là phải viết các lệnh ví dụ như ***std::cout*** << "*Hello everybody!*" << ***endl***;
- Thực hiện một khai báo là ***using namespace std;*** ở đầu chương trình để qui định tên của *namespace* mặc định là ***std***. Nhờ vậy khi viết ***cout*** thì trình biên dịch sẽ hiểu mặc định là ***std::cout***.

Việc áp dụng *namespace* trước hết dựa theo khai báo nội dung:

```
namespace < Tên >
{
     // các thực thể trong phạm vi namespace
     // các chỉ thị thích hợp
}
```

và sau đó là khai sử dụng: "***using namespace*** **< Tên > ;**". Để truy xuất trực tiếp đến một thực thể bên trong *namespace*, ta sử dụng *tên của namespace* ghép với toán tử **::** và ghép với *tên của thực thể*. Nếu đã khai báo sử dụng (bằng chỉ thị ***using namespace***) thì sau đó không cần dùng tên của *namespace*, chỉ cần dùng tên thực thể.

Dòng	Chương trình C++
1	#**include** <iostream>
2	**using namespace** std;
3	**namespace** Data_2D
4	**{**
5	**int** dX=3, dY=4; **// Kích thước**
6	**float** Area; **// Diện tích**
7	**}**
8	**namespace** Data_3D
9	**{**
10	**float** dX=5, dY=6, dZ=7; **// Kích thước**
11	**float** Volume; **// Thể tích**
12	**namespace** Base
13	**{**
14	**float** Area, h;
15	**}**
16	**}**
17	**void** main()
18	**{**
19	**using namespace** Data_2D;
20	Area = dX*dY ;
21	**cout** << "Data_2D::Area = " << Area << **endl** ;
22	**// --**
23	Data_3D::Base::Area = Data_3D::dX * Data_3D::dY ;
24	Data_3D::Volume = Data_3D::Base::Area * Data_3D::dZ ;
25	Data_3D::Base::h = (Data_3D::dX + Data_3D::dY)/2 ;
26	**cout** << "Data 3D" << **endl** ;
27	**cout** << " Volume = " << Data_3D::Volume << **endl** ;
28	**cout** << " h = " << Data_3D::Base::h << **endl** ;
29	**}**

Bảng 3.2 Chương trình minh họa việc sử dụng namespace

Bảng 3.2 minh họa việc khai báo ba *namespace* tên lần lượt là: ***Data_2D*** (dòng 3 – 7), ***Data_3D*** (dòng 8 – 16), ***Base*** (dòng 12 – 15). Trong đó ***Base*** được lồng bên trong của ***Data_3D***. Dòng 19 khai báo *namespace* mặc định là ***Data_2D***: vì vậy các biến ***Area***, ***dX***, và **dY** (được dùng ở dòng 20 và 21) cũng chính là ***Data_2D::Area***, ***Data_2D::dX***, ***Data_2D::dY*** (nhưng được viết ngắn gọn hơn). Để truy xuất các biến của ***Data_3D*** (các dòng 23 – 28) ta phải dùng ***Data_3D::*** ghép với tên biến, chẳng hạn như là ***Data_3D::Volume***. Đối với *namespace* tên là ***Base***, vì nó được lồng bên trong ***Data_3D*** nên để truy xuất các biến của nó phải dùng ***Data_3D::Base*** ghép với tên biến. Ví dụ như ở dòng 23 và 25: các biến ***Data_3D::Base::Area*** và ***Data_3D::Base::h*** chính là các biến của ***Base***.

I.4 Biến toàn cục và nguyên tắc sử dụng

Biến toàn cục được khai báo trong tập tin chương trình và nằm bên ngoài tất cả các khối lệnh, có các tính chất sau đây:

- Có thể được truy xuất từ mọi nơi trong chương trình, ngoại trừ trường hợp những chỗ có biến trùng tên;
- Trong trường hợp có biến trùng tên, nếu viết bằng *C++*, cũng có thể truy xuất biến toàn cục nhờ dùng toán tử **::** ghép với tên biến.

Bảng 3.3 là chương trình *C++* có khai báo biến toàn cục ***dX*** (dòng lệnh 3). Trong khối lệnh chính của hàm ***main( )*** không có khai báo biến nên dùng được biến ***dX*** toàn cục. Trong khối lệnh con (từ dòng 8 đến dòng 13) có khai báo biến cục bộ bên trong khối cũng tên là ***dX*** (dòng 9, trùng tên với biến toàn cục ***dX***). Trong khối lệnh con biến toàn cục ***dX*** được dùng nhờ toán tử **::** bằng cách ghi là ***::dX*** (ở các dòng lệnh 11 và 12).

Nói chung, lập trình viên nên hạn chế tối đa việc sử dụng biến toàn cục để tránh những lỗi logic của chương trình do hiệu ứng dùng chung biến. Trong mọi trường hợp, đều có giải pháp và kỹ thuật thích hợp để khỏi phải dùng biến toàn cục. Trong một nhóm lập trình viên cùng phát triển phần mềm, những biến toàn cục (nếu buộc phải có) sẽ do trưởng nhóm quyết định cụ thể là gồm những biến nào và đề ra những qui ước ràng buộc nhất định về việc truy xuất những biến toàn cục đó.

Dòng	Chương trình C++
1	#**include** <iostream>
2	**using namespace** std ;
3	**int** dX=7 ;
4	**void** main()
5	**{**
6	**cout** << " dX (global) = " << dX << **endl** ;
7	**// sub block**
8	**{**
9	**int** dX=9 ;
10	**cout** << " dX (sub block) = " << dX << **endl** ;
11	**cout** << " dX (global) in sub block = " << ::dX << **endl** ;
12	::dX = 12;
13	**}**
14	**cout** << " dX (global) is changed = " << dX << **endl** ;
15	**}**

Bảng 3.3 Chương trình C++ minh họa biến toàn cục

Bảng 3.4 là chương trình *C++* có khai báo biến toàn cục ***dX*** (dòng lệnh 3). Trong khối lệnh chính của hàm ***main( )*** không có khai báo biến nên dùng được biến ***dX*** toàn cục. Trong khối lệnh con (từ dòng 8 đến dòng 11) có khai báo biến cục bộ ở dòng 9, trùng tên với biến toàn cục ***dX***. Mặc dù lệnh khác báo ở dòng 9 khởi gán ***dX*** với giá trị là 9, nhưng điều này không ảnh hưởng đến biến toàn cục ***dX***. Vì vậy chỉ thị in ở dòng 12 vẫn xuất ra giá trị của ***dX*** (toàn cục) là 7.

Dòng	Chương trình C
1	#**include** <stdio.h>
2	
3	**int** dX=7 ;
4	**void** main()
5	**{**
6	printf(" dX (global) = %d\n", dX) ;
7	**/* sub block */**
8	**{**
9	**int** dX=9 ;
10	printf(" dX (sub block) = %d\n", dX) ;
11	**}**
12	printf(" dX (global) is unchanged = %d\n", dX) ;
13	**}**

Bảng 3.4 Chương trình minh họa biến toàn cục

I.5 Tránh nhập nhằng khó hiểu trong mã nguồn

Mặc dù ngôn ngữ lập trình *C++* cung cấp khái niệm *namespace* nhằm để phân giải các tên trùng nhau, người lập trình cần phải vận dụng chức năng này một cách có tổ chức nhằm để hướng đến mã nguồn chương trình rõ ràng và dễ bảo trì. Các lập trình viên không nên lạm dụng chức năng này để khai báo nhiều đối tượng trùng tên nhau đưa đến việc nhập nhằng và khó hiểu trong mã nguồn chương trình. Mục đích của khái niệm này là hỗ trợ việc cấu trúc hóa mã nguồn nhờ sự phân lớp một cách có hệ thống các thực thể lập trình (*lớp, hàm, biến, đối tượng*) để có thể khai thác chúng một cách hiệu quả.

Việc sử dụng *namespace* đặc biệt có ý nghĩa quan trọng trong việc tổ chức và đóng gói thư viện hàm hay thư viện lớp, chuyển giao mã nguồn giữa các đối tác làm phần mềm. Giải quyết được sự trùng lặp tên thực thể giữa những nhóm phát triển phần mềm khác nhau mà cần phải chia sẻ, sử dụng chung những gói mã nguồn quan trọng và mang tính chiến lược.

II. CẤU TRÚC RẼ NHÁNH

Các cấu trúc lập trình rẽ nhánh được sử dụng trong trường hợp việc tính toán trong chương trình phụ thuộc vào một điều kiện luận lý nào đó, được viết dưới dạng một biểu thức luận lý. Khi biểu thức luận lý đúng thì một số lệnh tương ứng được thực hiện; nếu ngược lại (nghĩa là biểu thức luận lý sai) thì một số câu lệnh khác được thi hành. Ngôn ngữ lập trình *C/C+* có sẵn hai cấu trúc rẽ nhánh: ***if else*** và ***switch***.

II.1 Cấu trúc rẽ nhánh *if else*

Hình 3.2 trình bày cấu trúc rẽ nhánh ***if else*** dạng đơn giản (không có phần ***else***). Trong cấu trúc này, một số lệnh trong khối lệnh sẽ được thực hiện nếu một biểu thức luận lí có giá trị đúng (cũng còn gọi là biểu thức điều kiện). Điều này có nghĩa là việc thực hiện của chương trình được rẽ nhánh sang một số lệnh cần thiết nếu biểu thức điều kiện có giá trị đúng. Trường hợp đặc biệt khi khối lệnh chỉ có duy nhất một lệnh thì có thể không cần dùng dấu mở khối { và dấu đóng khối }.

Chẳng hạn để tìm số lớn nhất và số nhỏ nhất trong hai số ***a*** và ***b***, ta có thể thực hiện các bước như sau: