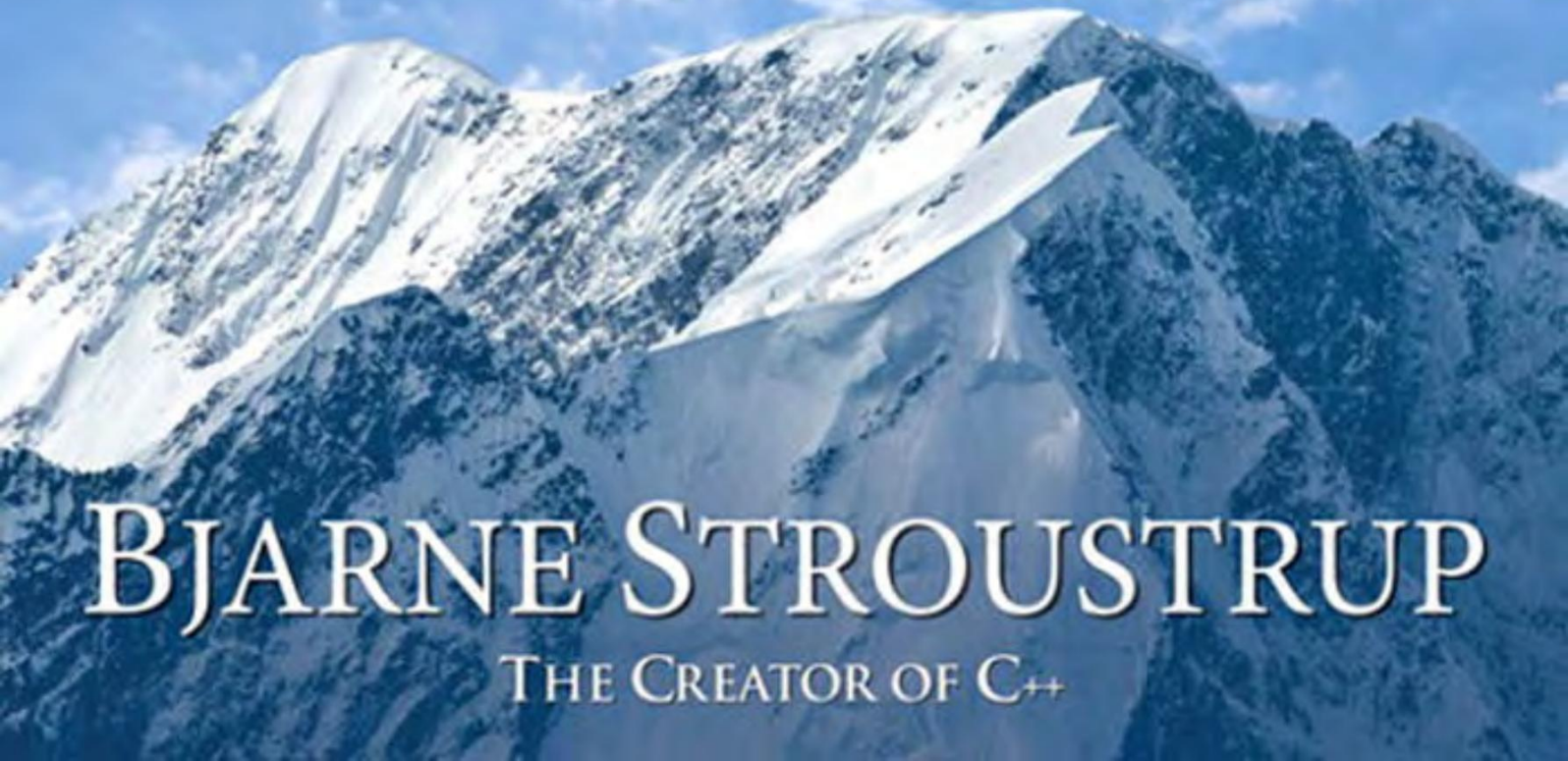
Machine Translated by Google

C++11

THE C++ PROGRAMMING LANGUAGE

FOURTH EDITION



BJARNE STROUSTRUP

THE CREATOR OF C++

Machine Translated by Google

Các C++ lập trình Ngôn ngữ

Phiên bản thứ tư

Bjarne Stroustrup

Addison-Wesley

Thư ợng lư u sông Saddle, NJ • Boston • Indianapolis • San Francisco
New York • Totonto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Thành phố Mexico

Machine Translated by Google

Nhiều ký hiệu đư ợc nhà sản xuất và ngư ời bán sử dụng để phân biệt sản phẩm của họ đư ợc tuyên bố là nhãn hiệu. Khi những ký hiệu đó xuất hiện trong cuốn sách này và nhà xuất bản đã biết về khiếu nại nhãn hiệu, thì các ký hiệu đó đã đư ợc in bằng các chữ cái đầu viết hoa hoặc viết hoa toàn bộ.

Tác giả và nhà xuất bản đã quan tâm đến việc chuẩn bị cuốn sách này, như ng không bảo đảm rõ ràng hay ngụ ý dư ới bất kỳ hình thức nào và không chịu trách nhiệm về những sai sót hoặc thiếu sót. Không có trách nhiệm pháp lý nào đư ợc thừa nhận đối với các thiệt hại ngẫu nhiên hoặc do hậu quả liên quan đến hoặc phát sinh từ việc sử dụng thông tin hoặc chư ơ ng trình có trong tài liệu này.

Nhà xuất bản cung cấp giảm giá tuyệt vời cho cuốn sách này khi đặt hàng với số lư ợng lớn để mua số lư ợng lớn hoặc bán hàng đặc biệt, có thể bao gồm các phiên bản điện tử và/hoặc bìa tùy chỉnh và nội dung dành riêng cho doanh nghiệp, mục tiêu đào tạo, trọng tâm tiếp thị và sở thích xây dựng thư ơ ng hiệu của bạn. Vui lòng liên lạc để biết thêm thông tin:

Bán hàng cho Doanh nghiệp và Chính phủ Hoa

Kỳ (800) 382-3419

corpsales@pearsontechgroup.com Để bán

hàng bên ngoài Hoa Kỳ, vui lòng liên hệ:

Doanh số bán hàng quốc tế

quốc tế@pearsoned.com

Ghé thăm chúng tôi trên trang Web: informit.com/aw

Dữ liệu Biên mục của Thư viện Quốc hội

Stroustrup, Bjarne.

Ngôn ngữ lập trình C++ / Bjarne Stroustrup.—Ấn bản lần thứ tư .

trang cm

Bao gồm tài liệu tham khảo và chỉ mục.

ISBN 978-0-321-56384-2 (pbk. : giấy kiềm)—ISBN 0-321-56384-0 (pbk. : giấy kiềm)

1. C++ (Ngôn ngữ lập trình máy tính) I. Tên bài.

QA76.73.C153 S77 2013 005.13'3

—dc23 2013002159

Bản quyền © 2013 của Pearson Education, Inc.

Đã đăng ký Bản quyền. In tại Hoa Kỳ. Ấn phẩm này đư ợc bảo vệ bởi bản quyền và phải đư ợc nhà xuất bản cho phép trư ớc khi thực hiện bất kỳ hành vi sao chép, lư u trữ nào bị cấm trong hệ thống truy xuất hoặc truyền tải dư ới bất kỳ hình thức nào hoặc bằng bất kỳ phư ơ ng tiện nào, điện tử, cơ khí, sao chụp, ghi âm hoặc tư ơ ng tự. Để đư ợc phép sử dụng tài liệu từ tác phẩm này, vui lòng gửi yêu cầu bằng văn bản tới Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, hoặc bạn có thể gửi fax yêu cầu của mình tới (201) 236-3290 .

Cuốn sách này đã đư ợc tác giả sắp chữ trên tờ Times và Helvetica.

ISBN-13: 978-0-321-56384-2

ISBN-10: 0-321-56384-0

Te xt đư ợc in ở Hoa Kỳ trên giấy tái chế tại Edwards Brothers Malloy ở Ann Arbor, Michigan.

In lần thứ hai, tháng 6 năm 2013

Machine Translated by Google

nội dung

nội dung iii

lời nói đầu TRONG

Lời tựa cho lần xuất bản thứ tư ....
v Lời tựa cho lần xuất bản thứ ba .... ix Lời
nói đầu cho lần xuất bản thứ hai .... xi Lời
tựa cho lần xuất bản đầu tiên xii

Phần I: Tài liệu giới thiệu

1. Ghi chú cho ngư ời đọc 3
2. Tìm hiểu về C++: Cơ bản .... 37 3. Tìm hiểu về
C++: Các cơ chế trừu tư ợng ..... 59 4. Giới thiệu về C++: Bộ chứa và thuật
toán .. .. 87 5. Tìm hiểu về C++: Đồng thời và Tiện
ích 111

Phần II: Cơ sở vật chất cơ bản 133

6. Các loại và khai báo .... 135 7. Con
trỏ, Mảng và Tham chiếu .... 171 8. Cấu trúc, Liên kết và
Liệt kê .... 201 9. Các câu
lệnh .... 225 10. Biểu
thức 241

Machine Translated by Google

11. Chọn Thao tác ....
273 12. Chức năng ....305 ....

13. Xử lý ngoại lệ .... 343 14. Không

gian tên .... 389 15.

Tệp nguồn và chư ơ ng trình 419

Phần III: Cơ chế trừu tư ợng hóa 447

16. Lớp học ....

449 17. Xây dựng, Dọn dẹp, Sao chép, và Di chuyển ..... 481 18. Nạp

chồng .... 527 19.

Ngư ời vận hành đặc biệt ....
549 20. Các lớp dẫn xuất ....

577 21. Hệ thống phân cấp lớp ....613.....

22. Thông tin về loại thời gian chạy .... 641 23.

Mẫu .... 665 24.

Lập trình chung .... 699 25.

Chuyên ngành 721
26. Khởi tạo 741

27. Mẫu và cấu trúc phân cấp .... 759 28. Lập

trình siêu dữ liệu .... 779 29. Thiết

kế ma trận 827

Phần IV: Thư viện tiêu chuẩn 857

30. Tóm tắt thư viện tiêu chuẩn .... 859 31.
Bộ chứa STL .... 885 32. Các

thuật toán STL... .... 927 33.
Trình lặp STL ....

953 34. Bộ nhớ và tài nguyên .... 973 35.
Tiện ích ....

1009 36. Chuỗi .... 1033.....

37. Biểu thức chính quy .... 1051 38.
Luồng I/O 1073

39. Địa phư ơ ng ....

1109 40. Số .... .1159

41. Tư ơ ng tranh 1191
42. Chủ đề và Nhiệm vụ 1209

43. Thư viện tiêu chuẩn C .... 1253 44. Khả

năng tư ơ ng thích 1267

Machine Translated by Google

lời nói đầu

Tất cả các vấn đề trong khoa học máy
tính có thể đư ợc giải quyết bằng một mức độ
gián tiếp khác, ngoại trừ vấn đề có quá nhiều lớp gián tiếp.
– David J. Wheeler

C ++ giống như một ngôn ngữ mới. Tức là, tôi có thể diễn đạt ý tư ởng của mình rõ ràng hơ n, đơ n giản hơ n và trực tiếp hơ n trong C++11 so với C++98. Hơ n nữa, các chư ơ ng trình kết quả đư ợc trình biên dịch kiểm tra tốt hơ n và chạy nhanh hơ n.

Trong cuốn sách này, tôi hư ớng đến sự trọn vẹn. Tôi mô tả mọi tính năng ngôn ngữ và thư viện tiêu chuẩn thành phần mà một lập trình viên chuyên nghiệp có thể cần. Đối với mỗi, tôi cung cấp:

- Cơ sở lý luận: Những loại vấn đề nào nó đư ợc thiết kế để giúp giải quyết? Nguyên tắc nào làm cơ sở thiết kế? Những hạn chế cơ bản là gì?
- Đặc tả: Định nghĩa của nó là gì? Mức độ chi tiết đư ợc chọn cho chư ơ ng trình chuyên gia; luật sư ngôn ngữ đầy tham vọng có thể làm theo nhiều tài liệu tham khảo về tiêu chuẩn ISO. • Ví dụ:

Làm thế nào để nó có thể đư ợc sử dụng tốt một mình và kết hợp với các tính năng khác? Các kỹ thuật và thành ngữ chính là gì? Ý nghĩa của khả năng bảo trì và hiệu suất là gì?

Việc sử dụng C++ đã thay đổi đáng kể qua nhiều năm và bản thân ngôn ngữ này cũng vậy. Từ quan điểm của một lập trình viên, hầu hết các thay đổi đều là cải tiến. Tiêu chuẩn ISO C++ hiện tại (ISO/IEC 14882-2011, thư ờng đư ợc gọi là C++11) đơ n giản là một công cụ tốt hơ n nhiều để viết phần mềm chất lư ợng so với các phiên bản trư ớc. Làm thế nào nó là một công cụ tốt hơ n? C ++ hiện đại hỗ trợ những loại phong cách và kỹ thuật lập trình nào? Những tính năng của ngôn ngữ và thư viện tiêu chuẩn nào hỗ trợ các kỹ thuật đó? Các khối xây dựng cơ bản của mã C++ tao nhã, chính xác, có thể bảo trì và hiệu quả là gì? Đó là những câu hỏi chính đư ợc trả lời bởi cuốn sách này. Nhiều câu trả lời không giống như bạn tìm thấy với C ++ cổ điển 1985, 1995 hoặc 2005: tiến trình xảy ra.

C++ là ngôn ngữ lập trình có mục đích chung, nhấn mạnh vào thiết kế và sử dụng các trừu tư ợng nhẹ, giàu kiểu. Nó đặc biệt phù hợp với các ứng dụng hạn chế về tài nguyên, chẳng hạn như các ứng dụng đư ợc tìm thấy trong cơ sở hạ tầng phần mềm. C ++ thư ởng cho lập trình viên dành thời gian để thành thạo

Machine Translated by Google

kỹ thuật để viết mã chất lượng. C++ là ngôn ngữ dành cho những người thực hiện nhiệm vụ kết hợp chương trình một cách nghiêm túc. Nền văn minh của chúng ta phụ thuộc rất nhiều vào phần mềm; nó tốt hơn nên là phần mềm chất lượng.

Có hàng tỷ dòng C++ được triển khai. Điều này đặt ưu tiên cho tính ổn định, vì vậy mã C++ 1985 và 1995 vẫn hoạt động và sẽ tiếp tục hoạt động trong nhiều thập kỷ. Tuy nhiên, đối với tất cả các ứng dụng, bạn có thể làm tốt hơn với C++ hiện đại; nếu bạn sử dụng các kiểu cũ hơn, bạn sẽ viết mã có chất lượng thấp hơn và hoạt động kém hơn. Việc nhấn mạnh vào tính ổn định cũng ngụ ý rằng mã phù hợp với tiêu chuẩn mà bạn viết hôm nay sẽ vẫn hoạt động trong vài thập kỷ kể từ bây giờ. Tất cả mã trong cuốn sách này tuân theo tiêu chuẩn ISO C++ 2011.

Cuốn sách này dành cho ba đối tượng: • Các

lập trình viên C++ muốn biết tiêu chuẩn ISO C++ mới nhất cung cấp những gì, • Các lập trình viên C thắc mắc C++ cung cấp những gì ngoài C, và • Những người có nền tảng về ngôn ngữ

ứng dụng, chẳng hạn như Java, C#, Python và Ruby, đang tìm kiếm thứ gì đó ''gần với máy hơn'' – thứ gì đó linh hoạt hơn, thứ gì đó cung cấp khả năng kiểm tra thời gian biên dịch tốt hơn hoặc thứ gì đó mang lại hiệu suất tốt hơn.

Dương nhiên, ba nhóm này không tách rời nhau – một nhà phát triển phần mềm chuyên nghiệp không chỉ thành thạo một ngôn ngữ lập trình.

Cuốn sách này giả định rằng độc giả của nó là các lập trình viên. Nếu bạn hỏi ''Vòng lặp for là gì?'' hay ''Trình biên dịch là gì?'' thì cuốn sách này (chưa) dành cho bạn; thay vào đó, tôi khuyên bạn nên Lập trình: Nguyên tắc và Thực hành Sử dụng C++ để bắt đầu với lập trình và C++. Hơn nữa, tôi cho rằng độc giả đã trưởng thành nhất định với tư cách là nhà phát triển phần mềm. Nếu bạn hỏi ''Tại sao phải thử nghiệm?'' hoặc nói, ''Tất cả các ngôn ngữ về cơ bản đều giống nhau; chỉ cho tôi cú pháp'' hoặc tự tin rằng có một ngôn ngữ duy nhất lý tưởng cho mọi tác vụ, thì đây không phải là cuốn sách dành cho bạn.

Những tính năng nào mà C++11 cung cấp hơn và trên C++98? Một mẫu máy phù hợp với hiện đại

máy tính với nhiều đồng thời. Ngôn ngữ và cơ sở thư viện tiêu chuẩn để thực hiện lập trình đồng thời cấp hệ thống (ví dụ: sử dụng đa lõi). Xử lý biểu thức chính quy, con trỏ quản lý tài nguyên, số ngẫu nhiên, vùng chứa cải tiến (bao gồm, bảng băm), v.v.

Khởi tạo chung và thống nhất, câu lệnh for đơn giản hơn , ngữ nghĩa di chuyển, hỗ trợ Unicode cơ bản, lambdas, biểu thức hằng chung, kiểm soát giá trị mặc định của lớp, mẫu biến thể, chữ do người dùng xác định, v.v. Hãy nhớ rằng các thư viện và tính năng ngôn ngữ đó tồn tại để hỗ trợ các kỹ thuật lập trình lập trình để phát triển phần mềm chất lượng. Chúng có nghĩa là được sử dụng kết hợp – như những viên gạch trong một bộ tòa nhà – thay vì được sử dụng riêng lẻ trong sự cô lập tương đối để giải quyết một vấn đề cụ thể. Máy tính là một cỗ máy vạn năng và C++ phục vụ nó với khả năng đó. Đặc biệt, thiết kế của C++ nhằm mục đích đủ linh hoạt và tổng quát để đối phó với các vấn đề trong tương lai mà các nhà thiết kế của nó không ngờ tới.

Machine Translated by Google

Lời cảm ơ n Ngoài những
ngư ời đư ợc đề cập trong phần lời cảm ơ n của các ấn bản trư ớc, tôi muốn cảm ơ n Pete Becker, Hans-J. Boehm, Marshall Clow, Jonathan Coe, Lawrence Crowl, Walter Daugherty, J. Daniel Garcia, Robert Harle, Greg Hickman, Howard Hinnant, Brian Kernighan, Daniel Krügler, Nevin Liber, Michel Michaud, Gary Powell, Jan Christiaan van Winkel và Leor Zolman. Nếu không có sự giúp đỡ của họ, cuốn sách này sẽ nghèo nàn hơ n nhiều.

Cảm ơ n Howard Hinnant đã trả lời nhiều câu hỏi về thư viện tiêu chuẩn.

Andrew Sutton là tác giả của thư viện Origin, nơ i thử nghiệm phần lớn cuộc thảo luận về các khái niệm mô phỏng trong các chư ơ ng mẫu và của thư viện ma trận là chủ đề của Chư ơ ng 29. Thư viện Origin là nguồn mở và có thể đư ợc đư ợc tìm thấy bằng cách tìm kiếm ''Origin'' và ''Andrew Sutton'' trên Web.

Cảm ơ n lớp thiết kế sau đại học của tôi vì đã tìm ra nhiều vấn đề hơ n với ''các chư ơ ng tham quan'' hơ n bất kỳ ai khác.

Nếu tôi có thể làm theo mọi lời khuyên của những ngư ời đánh giá của mình, cuốn sách chắc chắn sẽ đư ợc cải thiện nhiều, như ng nó cũng sẽ dài hơ n hàng trăm trang. Mọi chuyên gia đánh giá đều đề xuất thêm các chi tiết kỹ thuật, ví dụ nâng cao và nhiều quy ư ớc phát triển hữu ích; mọi ngư ời mới đánh giá (hoặc nhà giáo dục) đề xuất thêm các ví dụ; và hầu hết những ngư ời đánh giá nhận xét (chính xác) rằng cuốn sách có thể quá dài.

Cảm ơ n Khoa Khoa học Máy tính của Đại học Princeton và đặc biệt là Giáo sư Brian Kernighan, vì đã tiếp đón tôi trong một phần của kỳ nghỉ phép giúp tôi có thời gian viết cuốn sách này.

Xin cảm ơ n Phòng thí nghiệm Máy tính của Đại học Cambridge, và đặc biệt là Giáo sư Andy Hopper, đã tiếp đón tôi trong một phần kỳ nghỉ phép để tôi có thời gian viết cuốn sách này.

Cảm ơ n biên tập viên của tôi, Peter Gordon, và nhóm sản xuất của anh ấy tại Addison-Wesley vì sự giúp đỡ và kiên nhẫn của họ.

Trạm đại học, Texas Bjarne Stroustrup

Machine Translated by Google

Trang này cố ý để trống

Machine Translated by Google

Lời tựa cho lần xuất bản thứ ba

Lập trình là sự hiểu biết.
-Kristen Nygaard

Tôi thấy việc sử dụng C++ thú vị hơ n bao giờ hết. Sự hỗ trợ của C++ dành cho thiết kế và lập trình đã đư ợc cải thiện đáng kể trong những năm qua và rất nhiều kỹ thuật hữu ích mới đã đư ợc phát triển để sử dụng nó. Tuy nhiên, C++ không chỉ thú vị. Các lập trình viên thực tế thông thư ờng đã đạt đư ợc những cải tiến đáng kể về năng suất, khả năng bảo trì, tính linh hoạt và chất lư ợng trong các dự án thuộc mọi loại và quy mô. Đến bây giờ, C++ đã hoàn thành hầu hết những kỳ vọng ban đầu của tôi dành cho nó, và cũng thành công ở những nhiệm vụ mà tôi thậm chí còn chư a từng mơ tới.

Cuốn sách này giới thiệu C++† tiêu chuẩn và các kỹ thuật thiết kế và lập trình chính đư ợc hỗ trợ bởi C++. C++ tiêu chuẩn là một ngôn ngữ mạnh mẽ và bóng bẩy hơ n nhiều so với phiên bản C++ đư ợc giới thiệu trong ấn bản đầu tiên của cuốn sách này. Các tính năng ngôn ngữ mới như không gian tên, ngoại lệ, mẫu và nhận dạng loại thời gian chạy cho phép nhiều kỹ thuật đư ợc áp dụng trực tiếp hơ n so với trư ớc đây và thư viện chuẩn cho phép lập trình viên bắt đầu từ cấp độ cao hơ n nhiều so với ngôn ngữ thông thư ờng.

Khoảng một phần ba thông tin trong ấn bản thứ hai của cuốn sách này đến từ ấn bản đầu tiên. Lần xuất bản thứ ba này là kết quả của việc viết lại với quy mô thậm chí còn lớn hơ n. Nó cung cấp một cái gì đó cho ngay cả những lập trình viên C ++ có kinh nghiệm nhất; đồng thời, cuốn sách này dễ tiếp cận hơ n đối với ngư ời mới bắt đầu so với những cuốn trư ớc đó. Sự bùng nổ của việc sử dụng C++ và lư ợng kinh nghiệm khổng lồ đư ợc tích lũy khiến điều này trở nên khả thi.

Định nghĩa của một thư viện chuẩn mở rộng tạo ra sự khác biệt đối với cách trình bày các khái niệm C++. Như trư ớc đây, cuốn sách này trình bày C++ độc lập với bất kỳ triển khai cụ thể nào, và cũng như trư ớc đó, các chư ơ ng hư ớng dẫn trình bày các cấu trúc và khái niệm ngôn ngữ theo thứ tự ''từ dư ới lên'' để một cấu trúc chỉ đư ợc sử dụng sau khi nó đã đư ợc xác định. Tuy nhiên, việc sử dụng một thư viện đư ợc thiết kế tốt sẽ dễ dàng hơ n nhiều so với việc hiểu chi tiết về cách triển khai nó. Do đó, thư viện tiêu chuẩn có thể đư ợc sử dụng để cung cấp các ví dụ thực tế và thú vị trư ớc khi ngư ời đọc có thể hiểu đư ợc hoạt động bên trong của nó. Bản thân thư viện chuẩn cũng là một nguồn dồi dào các ví dụ lập trình và kỹ thuật thiết kế.

Cuốn sách này trình bày mọi tính năng chính của ngôn ngữ C++ và thư viện chuẩn. Nó đư ợc tổ chức xung quanh các cơ sở ngôn ngữ và thư viện. Tuy nhiên, các tính năng đư ợc trình bày trong bối cảnh sử dụng của chúng.

† ISO/IEC 14882, Tiêu chuẩn cho ngôn ngữ lập trình C++.

Machine Translated by Google

Nghĩa là, trọng tâm là ngôn ngữ như một công cụ thiết kế và lập trình hơ n là bản thân ngôn ngữ. Cuốn sách này trình bày các kỹ thuật chính giúp C++ trở nên hiệu quả và dạy các khái niệm cơ bản cần thiết để thành thạo. Trừ khi minh họa các tính năng kỹ thuật, các ví dụ đư ợc lấy từ miền phần mềm hệ thống. Một ngư ời bạn đồng hành, The Annotated C++ Language Stan dard, trình bày định nghĩa ngôn ngữ hoàn chỉnh cùng với các chú thích để làm cho nó dễ hiểu hơ n.

Mục đích chính của cuốn sách này là giúp ngư ời đọc hiểu cách thức các phư ơ ng tiện do C++ cung cấp hỗ trợ các kỹ thuật lập trình chính. Mục đích là đư a ngư ời đọc vư ợt xa điểm mà họ chạy mã chủ yếu bằng cách sao chép các ví dụ và mô phỏng các phong cách lập trình từ các ngôn ngữ khác. Chỉ có sự hiểu biết tốt về các ý tư ởng đằng sau các cơ sở ngôn ngữ mới dẫn đến sự thông thạo. Đư ợc bổ sung bằng tài liệu triển khai, thông tin đư ợc cung cấp là đủ để hoàn thành các dự án quan trọng trong thế giới thực. Hy vọng rằng cuốn sách này sẽ giúp ngư ời đọc có đư ợc những hiểu biết mới và trở thành một lập trình viên và nhà thiết kế giỏi hơ n.

Lời cảm ơ n Ngoài
những ngư ời đư ợc đề cập trong phần lời cảm ơ n của ấn bản thứ nhất và thứ hai, tôi xin cảm ơ n Matt Austern, Hans Boehm, Don Caldwell, Lawrence Crowl, Alan Feuer, Andrew Forrest, David Gay, Tim Griffin, Peter Juhl , Brian Kernighan, Andrew Koenig, Mike Mowbray, Rob Murray, Lee Nackman, Joseph Newcomer, Alex Stepanov, David Vandevoorde, Peter Weinberger và Chris Van Wyk vì đã nhận xét về các chư ơ ng dự thảo của ấn bản thứ ba này. Nếu không có sự giúp đỡ và góp ý của họ, cuốn sách này sẽ khó hiểu hơ n, chứa nhiều lỗi hơ n, kém hoàn chỉnh hơ n một chút và có lẽ sẽ ngắn hơ n một chút.

Tôi cũng xin cảm ơ n các tình nguyện viên trong ủy ban tiêu chuẩn C++ đã làm rất nhiều việc.
số lư ợng công việc mang tính xây dựng để tạo ra C++ như ngày nay. Sẽ hơ i không công bằng khi chỉ ra từng cá nhân, như ng sẽ càng không công bằng hơ n nếu không đề cập đến bất kỳ ai, vì vậy tôi muốn đặc biệt nhắc đến Mike Ball, Dag Br¨uck, Sean Corfield, Ted Goldstein, Kim Knuttila, Andrew Koenig, Dmitry Lenkov, Nathan Myers, Martin O'Riordan, Tom Plum, Jonathan Shopiro, John Spicer, Jerry Schwarz, Alex Stepanov và Mike Vilot, với tư cách là những ngư ời từng hợp tác trực tiếp với tôi về một số phần của C++ và thư viện tiêu chuẩn của nó.

Sau lần in đầu tiên của cuốn sách này, hàng chục ngư ời đã gửi thư cho tôi những chỉnh sửa và gợi ý để cải thiện. Tôi đã có thể đáp ứng nhiều đề xuất của họ trong khuôn khổ cuốn sách để những lần in sau này đư ợc hư ởng lợi đáng kể. Các dịch giả của cuốn sách này sang nhiều ngôn ngữ cũng đã cung cấp nhiều lời giải thích rõ ràng. Để đáp ứng yêu cầu của độc giả, tôi đã thêm vào các phụ lục D và E. Nhân cơ hội này, tôi xin cảm ơ n một số ngư ời đã giúp đỡ: Dave Abrahams, Matt Austern, Jan Bielawski, Janina Mincer Daszkiewicz, Andrew Koenig, Diet mar K¨uhl , Nicolai Josuttis, Nathan Myers, Paul E. Sevin ¸c, Andy Tenne-Sens, Shoichi Uchida, Ping-Fai (Mike) Yang, và Dennis Yelle.

Đồi Murray, Áo mới Bjarne Stroustrup

Machine Translated by Google

Lời nói đầu cho lần xuất bản thứ hai

Con đường đi mãi mãi.
- Đau bụng

Như đã hứa trong ấn bản đầu tiên của cuốn sách này, C++ đã và đang phát triển để đáp ứng nhu cầu của người dùng. Sự phát triển này đã được hướng dẫn bởi kinh nghiệm của người dùng có nhiều nền tảng khác nhau làm việc trong nhiều lĩnh vực ứng dụng. Cộng đồng người dùng C++ đã tăng gấp trăm lần trong sáu năm kể từ lần xuất bản đầu tiên của cuốn sách này; nhiều bài học đã được học và nhiều kỹ thuật đã được khám phá và/hoặc xác nhận bằng kinh nghiệm. Một số trong những kinh nghiệm này được phản ánh ở đây.

Mục đích chính của các phần mở rộng ngôn ngữ được thực hiện trong sáu năm qua là nâng cao C++ như một ngôn ngữ để trừu tượng hóa dữ liệu và lập trình hướng đối tượng nói chung và nâng cao nó như một công cụ để viết các thư viện chất lượng cao về các loại do người dùng định nghĩa trong cụ thể. ''Thư viện chất lượng cao'' là thư viện cung cấp khái niệm cho người dùng dưới dạng một hoặc nhiều lớp sử dụng thuận tiện, an toàn và hiệu quả. Trong ngữ cảnh này, an toàn có nghĩa là một lớp cung cấp giao diện an toàn kiểu cụ thể giữa người dùng thư viện và nhà cung cấp của nó; hiệu quả có nghĩa là việc sử dụng lớp không áp đặt các chi phí đáng kể trong thời gian chạy hoặc không gian đối với người dùng so với mã C viết tay.

Cuốn sách này trình bày toàn bộ ngôn ngữ C++. Các chương từ 1 đến 10 giới thiệu hướng dẫn; Các chương từ 11 đến 13 thảo luận về các vấn đề thiết kế và phát triển phần mềm; và cuối cùng, hướng dẫn tham khảo C++ đầy đủ được bao gồm. Đương nhiên, các tính năng được thêm vào và các giải pháp được tạo ra từ phiên bản gốc là những phần không thể thiếu của bản trình bày. Chúng bao gồm độ phân giải quá tải được tinh chỉnh, phương tiện quản lý bộ nhớ và cơ chế kiểm soát truy cập, liên kết an toàn kiểu, hàm thành viên const và tĩnh , lớp trừu tượng, đa kế thừa, mẫu và xử lý ngoại lệ.

C++ là ngôn ngữ lập trình đa năng; miền ứng dụng cốt lõi của nó là lập trình hệ thống theo nghĩa rộng nhất. Ngoài ra, C++ được sử dụng thành công trong nhiều lĩnh vực ứng dụng không nằm trong nhãn này. Việc triển khai C ++ tồn tại từ một số máy vi tính khiêm tốn nhất đến siêu máy tính lớn nhất và cho hầu hết mọi hệ điều hành. Do đó, cuốn sách này mô tả chính ngôn ngữ C++ mà không cố gắng giải thích cách triển khai, môi trường lập trình hoặc thư viện cụ thể.

Cuốn sách này trình bày nhiều ví dụ về các lớp học, mặc dù hữu ích, nên được phân loại là “đồ chơi”. Phong cách trình bày này cho phép các nguyên tắc chung và kỹ thuật hữu ích nổi bật rõ ràng hơn so với trong một chương trình được soạn thảo đầy đủ, nơi chúng sẽ được chôn vùi trong chi tiết. Hầu hết

Machine Translated by Google

trong số các lớp hữu ích được trình bày ở đây, chẳng hạn như danh sách được liên kết, mảng, chuỗi ký tự, ma trận, lớp đồ họa, mảng kết hợp, v.v., đều có sẵn trong các phiên bản ''chống đạn'' và/hoặc ''mạ vàng'' từ nhiều loại nguồn thương mại và phi thương mại. Nhiều lớp học và thư viện ''sức mạnh công nghiệp'' này thực sự là hậu duệ trực tiếp và gián tiếp của các phiên bản đồ chơi được tìm thấy ở đây.

Ấn bản này nhấn mạnh nhiều hơn vào các khía cạnh hướng dẫn so với ấn bản đầu tiên của cuốn sách này. Tuy nhiên, bài thuyết trình vẫn nhắm trực tiếp vào các lập trình viên có kinh nghiệm và cố gắng không xúc phạm trí thông minh hoặc kinh nghiệm của họ. Thảo luận về các vấn đề thiết kế đã được mở rộng rất nhiều để phản ánh nhu cầu về thông tin ngoài việc mô tả các tính năng của ngôn ngữ và việc sử dụng chúng ngay lập tức. Chi tiết kỹ thuật và độ chính xác cũng đã được tăng lên. Đặc biệt, sách hướng dẫn tham khảo thể hiện nhiều năm làm việc theo hướng này. Mục đích là cung cấp một cuốn sách có chiều sâu đủ để hầu hết các lập trình viên có thể đọc nhiều hơn một lần. Nói cách khác, cuốn sách này trình bày ngôn ngữ C++, các nguyên tắc cơ bản của nó và các kỹ thuật chính cần thiết để áp dụng nó. Thưởng thức!

Lời cảm ơn Ngoài
những người được đề cập trong phần lời cảm ơn trong lời nói đầu của ấn bản đầu tiên, tôi xin cảm ơn Al Aho, Steve Buroff, Jim Coplien, Ted Goldstein, Tony Hansen, Lorraine Juhl, Peter Juhl, Brian Kernighan, Andrew Koenig, Bill Leggett, Warren Montgomery, Mike Mowbray, Rob Murray, Jonathan Shopiro, Mike Vilot và Peter Weinberger vì đã nhận xét về các chương dự thảo của ấn bản thứ hai này. Nhiều người đã ảnh hưởng đến sự phát triển của C++ từ năm 1985 đến năm 1991. Tôi chỉ có thể nhắc đến một vài người: Andrew Koenig, Brian Kernighan, Doug McIlroy và Jonathan Shopiro. Cũng xin cảm ơn nhiều người tham gia ''đánh giá bên ngoài'' các bản thảo hướng dẫn sử dụng tham khảo và cảm ơn những người đã chịu đựng trong năm đầu tiên của X3J16.

Đồi Murray, Áo mới Bjarne Stroustrup

Machine Translated by Google

Lời nói đầu cho lần xuất bản đầu tiên

Ngôn ngữ định hình cách chúng ta suy nghĩ
và xác định những gì chúng ta có thể nghĩ về.

– BLWhorf

C++ là ngôn ngữ lập trình có mục đích chung đư ợc thiết kế để làm cho việc lập trình trở nên thú vị hơ n đối với những lập trình viên nghiêm túc. Ngoại trừ các chi tiết nhỏ, C ++ là siêu ngôn ngữ lập trình C. Ngoài các tiện ích do C cung cấp, C++ cung cấp các tiện ích linh hoạt và hiệu quả để định nghĩa các kiểu mới. Một lập trình viên có thể phân vùng một ứng dụng thành các phần có thể quản lý đư ợc bằng cách xác định các loại mới phù hợp chặt chẽ với các khái niệm của ứng dụng. Kỹ thuật xây dựng chư ơ ng trình này thư ờng đư ợc gọi là trừu tư ợng hóa dữ liệu. Các đối tư ợng của một số loại do ngư ời dùng định nghĩa chứa thông tin về loại.
Các đối tư ợng như vậy có thể đư ợc sử dụng một cách thuận tiện và an toàn trong các ngữ cảnh mà loại của chúng không thể đư ợc xác định tại thời điểm biên dịch. Các chư ơ ng trình sử dụng các đối tư ợng thuộc loại này thư ờng đư ợc gọi là dựa trên đối tư ợng. Khi đư ợc sử dụng tốt, những kỹ thuật này dẫn đến các chư ơ ng trình ngắn hơ n, dễ hiểu hơ n và dễ bảo trì hơ n.

Khái niệm chính trong C++ là lớp. Một lớp là một loại do ngư ời dùng định nghĩa. Các lớp cung cấp khả năng ẩn dữ liệu, khởi tạo dữ liệu đư ợc đảm bảo, chuyển đổi kiểu ẩn cho các kiểu do ngư ời dùng định nghĩa, kiểu gõ động, quản lý bộ nhớ do ngư ời dùng kiểm soát và cơ chế cho các toán tử quá tải. C++ cung cấp các phư ơ ng tiện tốt hơ n nhiều để kiểm tra kiểu và thể hiện tính mô đun hơ n C. Nó cũng chứa các cải tiến không liên quan trực tiếp đến các lớp, bao gồm hằng ký hiệu, thay thế nội tuyến của hàm, đối số hàm mặc định, tên hàm quá tải, toán tử quản lý cửa hàng miễn phí và loại tham chiếu. C ++ giữ lại khả năng của C để xử lý hiệu quả các đối tư ợng cơ bản của phần cứng (bit, byte, từ, địa chỉ, v.v.). Điều này cho phép các loại do ngư ời dùng xác định đư ợc triển khai với mức độ hiệu quả dễ chịu.

C ++ và các thư viện tiêu chuẩn của nó đư ợc thiết kế cho tính di động. Việc triển khai hiện tại sẽ chạy trên hầu hết các hệ thống hỗ trợ C. Thư viện C có thể đư ợc sử dụng từ chư ơ ng trình C++ và hầu hết các công cụ hỗ trợ lập trình trong C đều có thể đư ợc sử dụng với C++.

Cuốn sách này chủ yếu nhằm giúp các lập trình viên nghiêm túc học ngôn ngữ này và sử dụng nó cho các dự án không tầm thư ờng. Nó cung cấp một mô tả đầy đủ về C++, nhiều ví dụ hoàn chỉnh và nhiều đoạn chư ơ ng trình khác.

Machine Translated by Google

Lời cảm ơn C++ có thể
không bao giờ trưởng thành nếu không có sự sử dụng liên tục, góp ý và phê bình mang tính xây dựng của nhiều bạn bè và đồng nghiệp. Đặc biệt, Tom Cargill, Jim Coplien, Stu Feldman, Sandy Fraser, Steve Johnson, Brian Kernighan, Bart Locanthi, Doug McIlroy, Dennis Ritchie, Larry Rosler, Jerry Schwarz và Jon Shopiro đã đưa ra những ý tưởng quan trọng để phát triển ngôn ngữ này. Dave Pre sotto đã viết triển khai hiện tại của thư viện luồng I/O.

Ngoài ra, hàng trăm người đã đóng góp cho sự phát triển của C++ và trình biên dịch của nó bằng cách gửi cho tôi các đề xuất cải tiến, mô tả các vấn đề họ gặp phải và lỗi trình biên dịch. Tôi chỉ có thể nhắc đến một vài người: Gary Bishop, Andrew Hume, Tom Karzes, Victor Milenkovic, Rob Murray, Leonie Rose, Brian Schmult và Gary Walker.

Nhiều người cũng đã giúp cho việc xuất bản cuốn sách này, đặc biệt là Jon Bentley, Laura Eaves, Brian Kernighan, Ted Kow alski, Steve Mahaney, Jon Shopiro, và những người tham gia khóa học C++ được tổ chức tại Bell Labs, Columbus, Ohio, Ngày 26-27/6/1985.

Đồi Murray, Áo mới Bjarne Stroustrup

Machine Translated by Google

phần tôi

Giới thiệu

Phần giới thiệu này cung cấp một cái nhìn tổng quan về các khái niệm và tính năng chính của ngôn ngữ ngữ pháp chuyên nghiệp C++ và thư viện chuẩn của nó. Nó cũng cung cấp một cái nhìn tổng quan về cuốn sách này và giải thích cách tiếp cận đư ợc thực hiện để mô tả các cơ sở ngôn ngữ và việc sử dụng chúng. Ngoài ra, các chư ơ ng giới thiệu trình bày một số thông tin cơ bản về C++, thiết kế của C++ và việc sử dụng C++.

chư ơ ng

1 Ghi chú cho Ngư ời đọc
2 Chuyến tham quan C++: Khái niệm cơ bản
3 Tham quan C++: Cơ chế trừu tư ợng
4 Giới thiệu về C++: Bộ chứa và thuật toán
5 Tham quan C++: Đồng thời và Tiện ích

Machine Translated by Google

''... và anh, Marcus, anh đã cho tôi nhiều thứ; bây giờ tôi sẽ cho bạn lời khuyên tốt này. Đư ợc nhiều ngư ời. Hãy từ bỏ trò chơ i luôn là Marcus Cocoza. Bạn đã lo lắng quá nhiều về Marcus Cocoza, đến nỗi bạn thực sự là nô lệ và tù nhân của anh ta. Bạn đã không làm bất cứ điều gì mà không cân nhắc trư ớc xem điều đó sẽ ảnh hư ởng đến hạnh phúc và uy tín của Marcus Cocoza như thế nào. Bạn luôn lo lắng rằng Marcus có thể làm điều gì đó ngu ngốc, hoặc cảm thấy buồn chán. Điều gì sẽ thực sự quan trọng? Trên khắp thế giới, mọi ngư ời đang làm những điều ngu xuẩn...

Anh muốn em nhẹ nhàng, trái tim nhỏ bé của em nhẹ nhàng trở lại. Từ bây giờ, bạn phải trở thành nhiều hơ n một, nhiều ngư ời, bao nhiêu ngư ời mà bạn có thể nghĩ ra...''

- Karen Blixen,

Những kẻ mộng mơ trong Bảy câu chuyện Gothic (1934)

Machine Translated by Google

1

Ghi chú cho ngư ời đọc

Vội vàng Chậm
rãi (festina lente).
- Octavius, Caesar Augustus

• Cấu trúc của cuốn sách này
Giới thiệu; Cơ sở vật chất; Cơ chế trừu tư ợng; Thư viện tiêu chuẩn; Ví dụ và Tài liệu tham khảo
• Thiết kế của
C++
Phong cách lập trình; Kiểm tra loại; Khả năng tư ơ ng thích C; Ngôn ngữ, Thư viện và Hệ thống
• Học C++
Lập trình bằng C++; Gợi ý cho Lập trình viên C++; Gợi ý cho Lập trình viên C; Gợi ý cho Lập trình viên Java • Lịch sử Dòng thời gian;
Những năm
đầu; Tiêu chuẩn 1998; Tiêu chuẩn 2011; C ++ đư ợc sử dụng để làm gì? • Lời khuyên • Tài liệu tham khảo

1.1 Cấu trúc của cuốn sách này

Một hư ớng dẫn thuần túy sắp xếp các chủ đề của nó để không có khái niệm nào đư ợc sử dụng trư ớc khi nó đư ợc giới thiệu; nó phải đư ợc đọc tuyến tính bắt đầu từ trang một. Ngư ợc lại, một hư ớng dẫn tham khảo thuần túy có thể đư ợc truy cập bắt đầu từ bất kỳ thời điểm nào; nó mô tả từng chủ đề một cách ngắn gọn với các tham chiếu (tiến và lùi) đến các chủ đề liên quan. Về nguyên tắc, một hư ớng dẫn thuần túy có thể đư ợc đọc mà không cần điều kiện tiên quyết – nó mô tả cẩn thận tất cả. Một tài liệu tham khảo thuần túy chỉ có thể đư ợc sử dụng bởi những ngư ời quen thuộc với tất cả các khái niệm và kỹ thuật cơ bản. Cuốn sách này kết hợp các khía cạnh của cả hai. Nếu bạn biết hầu hết các khái niệm và kỹ thuật, bạn có thể truy cập nó theo từng chư ơ ng hoặc thậm chí theo từng phần. Nếu không, bạn có thể bắt đầu từ đầu, như ng cố gắng đừng sa lầy vào chi tiết. Sử dụng chỉ mục và các tham chiếu chéo.

Machine Translated by Google

Việc làm cho các phần của cuốn sách trở nên tư ơ ng đối khép kín ngụ ý một số sự lặp lại, như ng sự lặp lại cũng đóng vai trò đánh giá đối với những ngư ời đọc cuốn sách một cách tuyến tính. Cuốn sách đư ợc tham chiếu chéo nhiều đến cả chính nó và tiêu chuẩn ISO C++. Các lập trình viên có kinh nghiệm có thể đọc "chuyến tham quan" C ++ nhanh (tư ơ ng đối) để có đư ợc cái nhìn tổng quan cần thiết để sử dụng cuốn sách làm tài liệu tham khảo. Cuốn sách này bao gồm bốn

phần: Phần I Giới thiệu: Chư ơ ng 1 (chư ơ ng này) là hư ớng dẫn của cuốn sách này và cung cấp một chút kiến thức cơ bản về C++. Chư ơ ng 2-5 giới thiệu nhanh về ngôn ngữ C++ và thư viện chuẩn của nó.

Phần II Cơ sở vật chất cơ bản: Chư ơ ng 6-15 mô tả các kiểu dựng sẵn của C++ và các cơ sở vật chất cơ bản để xây dựng chư ơ ng trình từ chúng.

Phần III Các cơ chế trừu tư ợng: Các chư ơ ng 16-29 mô tả các cơ chế trừu tư ợng hóa của C++ và việc sử dụng chúng cho lập trình hư ớng đối tư ợng và lập trình chung.

Phần IV Các chư ơ ng 30-44 cung cấp tổng quan về thư viện chuẩn và thảo luận về các vấn đề tư ơ ng thích.

1.1.1 Giới thiệu

Chư ơ ng này, Chư ơ ng 1, cung cấp tổng quan về cuốn sách này, một số gợi ý về cách sử dụng nó và một số thông tin cơ bản về C++ và cách sử dụng nó. Bạn đư ợc khuyến khích đọc lư ớt qua nó, đọc những gì thấy thú vị và quay lại với nó sau khi đọc các phần khác của cuốn sách. Xin đừng cảm thấy bắt buộc phải đọc tất cả cẩn thận trư ớc khi tiếp tục.

Các chư ơ ng sau cung cấp tổng quan về các khái niệm và tính năng chính của C++ pro ngôn ngữ ngữ pháp và thư viện tiêu chuẩn của nó:

Chư ơ ng 2 Giới thiệu về C++: Khái niệm cơ bản mô tả mô hình bộ nhớ, tính toán và xử lý lỗi.

Chư ơ ng 3 Tham quan C++: Cơ chế trừu tư ợng trình bày các tính năng ngôn ngữ hỗ trợ trừu tư ợng hóa dữ liệu, lập trình hư ớng đối tư ợng và lập trình chung.

Chư ơ ng 4 Giới thiệu về C++: Bộ chứa và Thuật toán giới thiệu các chuỗi, I/O đơ n giản, con hàm và các thuật toán như đư ợc cung cấp bởi thư viện tiêu chuẩn.

Chư ơ ng 5 Giới thiệu C++: Đồng thời và Tiện ích phác thảo các tiện ích thư viện tiêu chuẩn liên quan đến quản lý tài nguyên, tư ơ ng tranh, tính toán toán học, biểu thức chính quy, v.v.

Chuyến tham quan vòng quanh cơ sở vật chất của C++ này nhằm cung cấp cho ngư ời đọc cảm nhận về những gì C++ cung cấp. Đặc biệt, nó sẽ thuyết phục độc giả rằng C++ đã đi một chặng đư ờng dài kể từ lần xuất bản thứ nhất, thứ hai và thứ ba của cuốn sách này.

1.1.2 Cơ sở vật chất cơ bản

Phần II tập trung vào tập hợp con của C++ hỗ trợ các phong cách lập trình truyền thống đư ợc thực hiện bằng C và các ngôn ngữ tư ơ ng tự. Nó giới thiệu các khái niệm về loại, đối tư ợng, phạm vi và lư u trữ. Nó trình bày các nguyên tắc cơ bản của tính toán: biểu thức, câu lệnh và hàm. Tính mô đun – như đư ợc hỗ trợ bởi không gian tên, tệp nguồn và xử lý ngoại lệ – cũng đư ợc thảo luận: Các loại và Tuyên bố: Các loại cơ bản, đặt tên,

Chư ơ ng 6 phạm vi, khởi tạo, suy diễn loại đơ n giản, thời gian tồn tại của đối tư ợng và bí danh loại

Machine Translated by Google

Chương 7 Con trỏ, Mảng và Tham chiếu Chương 8 Cấu
trúc, Liên kết và Phép liệt kê Chương 9 Câu lệnh: Các câu
lệnh dưới dạng khai báo, câu lệnh lựa chọn (if và switch), câu lệnh lặp (for, while và do), goto và nhận xét
Chương 10 Biểu thức: Ví dụ về máy tính bàn, khảo sát toán tử,
biểu thức hằng số và chuyển đổi kiểu ẩn.

Chương 11 Chọn phép toán: Toán tử logic, biểu thức điều kiện, tăng và giảm, lưu trữ miễn phí (mới và xóa), danh sách {}, biểu thức lambda và chuyển đổi kiểu rõ ràng (static_cast và const_cast)

Chương 12 Hàm: Định nghĩa và khai báo hàm, hàm nội tuyến , hàm constexpr , truyền đối số, hàm quá tải, điều kiện trước và sau, con trỏ tới hàm và macro Chương 13 Xử lý ngoại lệ: Các kiểu xử lý lỗi, đảm bảo ngoại lệ, quản lý tài nguyên, thực thi
bất biến , ném và bắt, triển khai véc tơ Chương 14 Không gian tên: không gian tên, mô đun hóa và giao diện, thành phần sử dụng tên

không gian

Chương 15 Tệp nguồn và chương trình: Biên dịch riêng biệt, liên kết, sử dụng tệp tiêu đề, bắt đầu và kết thúc chương trình

Tôi cho rằng bạn đã quen thuộc với hầu hết các khái niệm lập trình được sử dụng trong Phần I. Ví dụ, tôi giải thích các phương tiện C++ để biểu diễn đệ quy và phép lặp, nhưng tôi không đi vào chi tiết kỹ thuật hoặc dành nhiều thời gian để giải thích các khái niệm này hữu ích như thế nào.

Ngoại lệ cho quy tắc này là ngoại lệ. Nhiều lập trình viên thiếu kinh nghiệm với các ngoại lệ hoặc có kinh nghiệm từ các ngôn ngữ (chẳng hạn như Java), nơi quản lý tài nguyên và xử lý ngoại lệ không được tích hợp. Do đó, chương về xử lý ngoại lệ (Chương 13) trình bày triết lý cơ bản về xử lý ngoại lệ và quản lý tài nguyên trong C++. Nó đi sâu vào một số chi tiết về chiến lược, tập trung vào kỹ thuật ''Mua lại tài nguyên là khởi tạo'' (RAII).

1.1.3 Cơ chế trừu tượng hóa

Phần III mô tả các cơ sở C++ hỗ trợ các hình thức trừu tượng khác nhau, bao gồm lập trình hướng đối tượng và lập trình chung. Các chương được chia thành ba loại thô: lớp, chies thứ bậc lớp và mẫu.

Bốn chương đầu tiên tập trung vào bản thân các lớp: Chương 16 Các
lớp: Khái niệm về một loại do người dùng định nghĩa, một lớp, là nền tảng của tất cả C++
các cơ chế trừu tượng hóa.

Chương 17 Xây dựng, Dọn dẹp, Sao chép và Di chuyển chỉ ra cách một lập trình viên có thể định nghĩa ý nghĩa của việc tạo và khởi tạo các đối tượng của một lớp. Hơn nữa, ý nghĩa của sao chép, di chuyển và hủy có thể được chỉ định.

Chương 18 Nạp chồng toán tử trình bày các quy tắc để đưa ra ý nghĩa cho các toán tử đối với các loại do người dùng định nghĩa, nhấn mạnh vào các toán tử số học và logic thông thường, chẳng hạn như +, và &.

Chương 19 Các toán tử đặc biệt thảo luận về việc sử dụng toán tử do người dùng định nghĩa cho các mục đích phi số học, chẳng hạn như [] cho chỉ số con, () cho các đối tượng hàm và > cho ''con trỏ thông minh.''

Machine Translated by Google

Các lớp có thể đư ợc tổ chức thành các thứ bậc:

Chư ơ ng 20 Các lớp dẫn xuất trình bày các cơ sở ngôn ngữ cơ bản để xây dựng hệ thống phân cấp bên ngoài các lớp và các cách cơ bản để sử dụng chúng. Chúng tôi có thể cung cấp sự tách biệt hoàn toàn giữa một giao diện (một lớp trừu tư ợng) và các triển khai của nó (các lớp dẫn xuất); kết nối giữa chúng đư ợc cung cấp bởi các chức năng ảo.

Mô hình C++ để kiểm soát truy cập (công khai, đư ợc bảo vệ và riêng tư) đư ợc trình bày.

Chư ơ ng 21 Hệ thống phân cấp lớp thảo luận về các cách sử dụng hệ thống phân cấp lớp một cách hiệu quả. Nó cũng trình bày khái niệm đa kế thừa, nghĩa là một lớp có nhiều hơ n một lớp cơ sở trực tiếp.

Chư ơ ng 22 Thông tin về loại thời gian chạy trình bày các cách để điều hư ớng hệ thống phân cấp lớp bằng cách sử dụng dữ liệu đư ợc lư u trữ trong các đối tư ợng. Chúng ta có thể sử dụng dynamic_cast để hỏi xem một đối tư ợng của lớp cơ sở có đư ợc định nghĩa là đối tư ợng của lớp dẫn xuất hay không và sử dụng typeid để lấy thông tin tối thiểu từ một đối tư ợng (chẳng hạn như tên của lớp đối tư ợng).

Nhiều phần trừu tư ợng linh hoạt, hiệu quả và hữu ích nhất liên quan đến việc tham số hóa các loại (lớp) và thuật toán (hàm) với các loại và thuật toán khác: Chư ơ ng 23 Mẫu trình bày các nguyên tắc cơ bản đằng sau các mẫu và cách sử dụng chúng. Mẫu lớp, mẫu chức năng và bí danh mẫu đư ợc trình bày.

Chư ơ ng 24 Lập trình chung giới thiệu các kỹ thuật cơ bản để thiết kế chư ơ ng trình chung. Kỹ thuật nâng một thuật toán trừu tư ợng từ một số ví dụ mã cụ thể là trung tâm, cũng như khái niệm về các khái niệm xác định các yêu cầu của thuật toán chung đối với các đối số của nó.

Chư ơ ng 25 Chuyên biệt hóa mô tả cách các mẫu đư ợc sử dụng để tạo các lớp và chức năng, chuyên biệt hóa, cung cấp một tập hợp các đối số mẫu.

Chư ơ ng 26 Khởi tạo tập trung vào các quy tắc ràng buộc tên.

Chư ơ ng 27 Mẫu và phân cấp giải thích cách các mẫu và phân cấp lớp có thể đư ợc đư ợc sử dụng kết hợp.

Chư ơ ng 28 Siêu lập trình khám phá cách các mẫu có thể đư ợc sử dụng để tạo chư ơ ng trình. Các mẫu cung cấp một cơ chế hoàn chỉnh Turing để tạo mã.

Chư ơ ng 29 Thiết kế ma trận đư a ra một ví dụ dài để chỉ ra cách sử dụng kết hợp các tính năng ngôn ngữ để giải quyết vấn đề thiết kế phức tạp: thiết kế ma trận N chiều với các loại phần tử gần như tùy ý.

Các tính năng ngôn ngữ hỗ trợ các kỹ thuật trừu tư ợng hóa đư ợc mô tả trong ngữ cảnh của các kỹ thuật đó. Kỹ thuật trình bày ở Phần III khác với ở Phần II ở chỗ tôi không cho rằng ngư ời đọc biết các kỹ thuật đư ợc mô tả.

1.1.4 Thư viện tiêu chuẩn

Các chư ơ ng thư viện ít hư ớng dẫn hơ n các chư ơ ng ngôn ngữ. Cụ thể, chúng đư ợc dùng để đọc theo bất kỳ thứ tự nào và có thể đư ợc sử dụng làm hư ớng dẫn cấp độ ngư ời dùng cho các thành phần thư viện:

Chư ơ ng 30 Tổng quan về thư viện chuẩn cung cấp tổng quan về thư viện chuẩn, liệt kê các tiêu đề của thư viện chuẩn và trình bày hỗ trợ ngôn ngữ và hỗ trợ chẩn đoán, chẳng hạn như ngoại lệ và lỗi hệ thống.

Chư ơ ng 31 Bộ chứa STL trình bày các bộ chứa từ các bộ lặp, bộ chứa và khuôn khổ thuật toán (đư ợc gọi là STL), bao gồm vectơ , bản đồ và unordered_set.

Machine Translated by Google

Chương 32 Thuật toán STL trình bày các thuật toán từ STL, bao gồm tìm(), sắp xếp() và hợp nhất().

Chương 33 STL Iterators trình bày các iterator và các tiện ích khác từ STL, bao gồm reverse_iterator, move_iterator và hàm.

Chương 34 Bộ nhớ và Tài nguyên trình bày các thành phần tiện ích liên quan đến quản lý tài nguyên và bộ nhớ, chẳng hạn như mảng, bitset, cặp, bộ dữ liệu, unique_ptr, shared_ptr , bộ cấp phát và giao diện bộ thu gom rác.

Chương 35 Tiện ích trình bày các thành phần tiện ích nhỏ, chẳng hạn như tiện ích thời gian, đặc điểm loại và chức năng loại khác nhau.

Chương 36 Chuỗi ghi lại thư viện chuỗi , bao gồm các đặc điểm của ký tự là cơ sở cho việc sử dụng các bộ ký tự khác nhau.

Chương 37 Biểu thức chính quy mô tả cú pháp biểu thức chính quy và nhiều cách sử dụng nó để so khớp chuỗi, bao gồm regex_match() để so khớp một chuỗi hoàn chỉnh, regex_search() để tìm một mẫu trong chuỗi, regex_replace() để thay thế đơn giản và regex_iterator để duyệt chung một dòng ký tự.

Chương 38 Luồng I/O ghi lại thư viện luồng I/O. Nó mô tả định dạng và unfor đầu vào và đầu ra mờ, xử lý lỗi và đệm.

Chương 39 Ngôn ngữ mô tả ngôn ngữ lớp và các khía cạnh khác nhau của nó cung cấp hỗ trợ cho việc xử lý sự khác biệt về văn hóa trong bộ ký tự, định dạng giá trị số, định dạng ngày và giờ, v.v.

Chương 40 Số mô tả các cơ sở để tính toán số (chẳng hạn như số phức, valarray, số ngẫu nhiên và thuật toán số tổng quát).

Chương 41 Đồng thời trình bày mô hình bộ nhớ cơ bản của C++ và các phương tiện được cung cấp cho lập trình đồng thời không có khóa.

Chương 42 Luồng và Tác vụ trình bày các lớp cung cấp lập trình cho thuê đồng thời kiểu luồng và khóa (chẳng hạn như luồng, timed_mutex, lock_guard và try_lock()) và hỗ trợ đồng thời dựa trên tác vụ (chẳng hạn như tương lai và không đồng bộ()) .

Chương 43 Thư viện chuẩn C ghi lại thư viện chuẩn C (bao gồm printf() và clock()) như được tích hợp vào thư viện chuẩn C++.

Chương 44 Khả năng tương thích thảo luận về mối quan hệ giữa C và C++ và giữa Standard C++ (còn được gọi là ISO C++) và các phiên bản C++ đi trước nó.

1.1.5 Ví dụ và Tài liệu tham khảo

Cuốn sách này nhấn mạnh đến việc tổ chức chương trình hơn là thiết kế các thuật toán. Do đó, tôi tránh các thuật toán thông minh hoặc khó hiểu hơn. Một thuật toán tầm thường thường phù hợp hơn để minh họa một khía cạnh của định nghĩa ngôn ngữ hoặc một điểm về cấu trúc chương trình. Ví dụ: tôi sử dụng sắp xếp Shell trong đó, trong mã thực, sắp xếp nhanh sẽ tốt hơn. Thông thường, việc triển khai lại bằng một thuật toán phù hợp hơn là một bài tập. Trong mã thực, lệnh gọi hàm thư viện thường phù hợp hơn mã được sử dụng ở đây để minh họa các tính năng ngôn ngữ.

Các ví dụ về xtbook nhất thiết phải đưa ra một cái nhìn sai lệch về phát triển phần mềm. Bằng cách làm rõ và đơn giản hóa các ví dụ, sự phức tạp phát sinh từ quy mô sẽ biến mất. Tôi thấy không có sự thay thế nào cho việc viết các chương trình có kích thước thực tế để có ấn tượng về chương trình và

Machine Translated by Google

ngôn ngữ lập trình thực sự như thế nào. Cuốn sách này tập trung vào các đặc điểm ngôn ngữ và cơ sở vật chất tiêu chuẩn của thư viện. Đây là những kỹ thuật cơ bản mà từ đó mọi chư ơ ng trình đư ợc soạn thảo.
Các quy tắc và kỹ thuật cho bố cục như vậy đư ợc nhấn mạnh.

Việc lựa chọn các ví dụ phản ánh nền tảng của tôi về trình biên dịch, thư viện nền tảng và mô phỏng. Sự nhấn mạnh phản ánh sự quan tâm của tôi đối với lập trình hệ thống. Các ví dụ là các phiên bản đơ n giản hóa của những gì đư ợc tìm thấy trong mã thực. Việc đơ n giản hóa là cần thiết để giữ cho ngôn ngữ lập trình và các điểm thiết kế không bị mất chi tiết. Lý tư ởng của tôi là ví dụ ngắn gọn và rõ ràng nhất minh họa nguyên tắc thiết kế, kỹ thuật lập trình, cấu trúc ngôn ngữ hoặc tính năng thư viện.
Không có ví dụ ''dễ thư ơ ng'' nào mà không có đối tác trong mã thực. Đối với các ví dụ thuần túy về ngôn ngữ-kỹ thuật, tôi sử dụng các biến có tên là x và y, các loại đư ợc gọi là A và B, và các hàm có tên là f() và g().

Nếu có thể, các tính năng thư viện và ngôn ngữ C++ đư ợc trình bày trong ngữ cảnh sử dụng của chúng thay vì theo cách khô khan của sách hư ớng dẫn. Các tính năng ngôn ngữ đư ợc trình bày và chi tiết mà chúng đư ợc mô tả đại khái phản ánh quan điểm của tôi về những gì cần thiết để sử dụng hiệu quả C++. Mục đích là cung cấp cho bạn ý tư ởng về cách sử dụng một tính năng, thư ờng là kết hợp với các tính năng khác. Sự hiểu biết về mọi chi tiết ngôn ngữ-kỹ thuật của một tính năng ngôn ngữ hoặc thành phần thư viện là không cần thiết và cũng không đủ để viết các chư ơ ng trình tốt. Trên thực tế, nỗi ám ảnh về việc hiểu từng chi tiết nhỏ là đơ n thuốc cho mã khủng khiếp – quá phức tạp và quá thông minh. Điều cần thiết là sự hiểu biết về các kỹ thuật thiết kế và lập trình cùng với sự đánh giá cao về các lĩnh vực ứng dụng.

Tôi cho rằng bạn có quyền truy cập vào các nguồn thông tin trực tuyến. Trọng tài cuối cùng của ngôn ngữ và quy tắc thư viện chuẩn là tiêu chuẩn ISO C++ [C++,2011].

Tài liệu tham khảo cho các phần của cuốn sách này có dạng §2.3.4 (Chư ơ ng 2, mục 3, tiểu mục 4) và §iso.5.3.1 (tiêu chuẩn ISO C++, §5.3.1). Chữ in nghiêng ít đư ợc sử dụng để nhấn mạnh (ví dụ: ''một chuỗi ký tự không đư ợc chấp nhận''), cho các lần xuất hiện đầu tiên của các khái niệm quan trọng (ví dụ: tính đa hình) và cho các nhận xét trong các ví dụ về mã.

Để tiết kiệm một vài cây và để đơ n giản hóa việc bổ sung, hàng trăm bài tập cho cuốn sách này đã đư ợc chuyển lên Web. Hãy tìm chúng tại www.stroustrup.com.

Ngôn ngữ và thư viện đư ợc sử dụng trong cuốn sách này là ''C++ thuần túy'' theo định nghĩa của tiêu chuẩn C++ [C++,2011]. Do đó, các ví dụ sẽ chạy trên mọi triển khai C++ cập nhật. Các đoạn chư ơ ng trình chính trong cuốn sách này đã đư ợc thử sử dụng một số triển khai C++. Các ví dụ sử dụng các tính năng chỉ mới đư ợc áp dụng vào C++ gần đây đã không đư ợc biên dịch trên mỗi lần triển khai. Tuy nhiên, tôi không thấy có ích gì khi đề cập đến việc triển khai nào không biên dịch đư ợc ví dụ nào. Những thông tin như vậy sẽ sớm lỗi thời vì những ngư ời triển khai đang nỗ lực làm việc để đảm bảo rằng việc triển khai của họ chấp nhận chính xác mọi tính năng của C++. Xem Chư ơ ng 44 để biết các gợi ý về cách đối phó với các trình biên dịch C++ cũ hơ n và với mã đư ợc viết cho các trình biên dịch C.

Tôi sử dụng các tính năng của C++11 một cách tự do ở bất cứ nơ i nào tôi thấy phù hợp nhất. Ví dụ: tôi thích trình khởi tạo kiểu {} và sử dụng bí danh cho loại. Ở những nơ i, việc sử dụng đó có thể khiến ''những ngư ời tính giờ cũ'' phải giật mình. Dù thế nào đi nữa, giật mình thư ờng là một cách tốt để bắt đầu xem lại tài liệu. Mặt khác, tôi không sử dụng các tính năng mới chỉ vì chúng mới; lý tư ởng của tôi là cách thể hiện tao nhã nhất của các ý tư ởng cơ bản - và điều đó rất có thể đang sử dụng thứ gì đó đã có trong C++ hoặc thậm chí trong C từ lâu.

Rõ ràng, nếu bạn phải sử dụng trình biên dịch trư ớc C++ 11 (giả sử vì một số khách hàng của bạn chư a nâng cấp lên tiêu chuẩn hiện tại), bạn phải hạn chế sử dụng các tính năng mới. Tuy nhiên, xin đừng cho rằng ''cách cũ'' tốt hơ n hoặc đơ n giản hơ n chỉ vì chúng đã cũ và quen thuộc. §44.2 tóm tắt sự khác biệt giữa C++98 và C++11.

Machine Translated by Google

1.2 Thiết kế của C++

Mục đích của ngôn ngữ lập trình là giúp thể hiện ý tư ởng trong mã. Trong đó, một ngôn ngữ lập trình thực hiện hai nhiệm vụ liên quan: nó cung cấp phư ơng tiện để lập trình viên chỉ định các hành động sẽ đư ợc máy thực hiện và nó cung cấp một tập hợp các khái niệm để lập trình viên sử dụng khi nghĩ về những gì có thể đư ợc thực hiện. Mục đích đầu tiên lý tư ởng nhất là yêu cầu một ngôn ngữ ''gần với máy'' để tất cả các khía cạnh quan trọng của máy đư ợc xử lý một cách đơ n giản và hiệu quả theo cách dễ thấy đối với ngư ời lập trình. Ngôn ngữ C đư ợc thiết kế chủ yếu với ý nghĩ này. Mục đích thứ hai lý tư ởng nhất là yêu cầu một ngôn ngữ ''gần với vấn đề cần giải quyết'' để các khái niệm về giải pháp có thể đư ợc diễn đạt một cách trực tiếp và ngắn gọn. Các phư ơ ng tiện đư ợc thêm vào C để tạo C++, chẳng hạn như kiểm tra đối số hàm, const, các lớp, hàm tạo và hàm hủy, ngoại lệ và mẫu, đư ợc thiết kế chủ yếu với mục đích này. Do đó, C++ dựa trên ý tư ởng cung cấp cả hai

- ánh xạ trực tiếp các hoạt động và loại tích hợp vào phần cứng để cung cấp việc sử dụng bộ nhớ hiệu quả và các hoạt động cấp thấp hiệu quả, đồng thời • các cơ

chế trừu tư ợng hóa linh hoạt và giá cả phải chăng để cung cấp các loại do ngư ời dùng xác định với cùng hỗ trợ ký hiệu, phạm vi sử dụng và hiệu suất như các loại tích hợp.

Điều này ban đầu đạt đư ợc bằng cách áp dụng các ý tư ởng từ Simula cho C. Qua nhiều năm, việc áp dụng thêm các ý tư ởng đơ n giản này đã dẫn đến một tập hợp các cơ sở tổng quát, hiệu quả và linh hoạt hơ n nhiều. Kết quả hỗ trợ tổng hợp các phong cách lập trình có thể đồng thời hiệu quả và thanh lịch.

Thiết kế của C++ đã tập trung vào các kỹ thuật lập trình xử lý các khái niệm cơ bản như bộ nhớ, khả năng thay đổi, tính trừu tư ợng, quản lý tài nguyên, biểu thức thuật toán, xử lý lỗi và tính mô đun. Đó là những mối quan tâm quan trọng nhất của một lập trình viên hệ thống và nói chung là của các lập trình viên của các hệ thống hiệu suất cao và bị hạn chế về tài nguyên.

Bằng cách định nghĩa các thư viện lớp, phân cấp lớp và mẫu, bạn có thể viết các chư ơ ng trình C++ ở mức cao hơ n nhiều so với mức đư ợc trình bày trong cuốn sách này. Ví dụ, C++ đư ợc sử dụng rộng rãi trong các hệ thống tài chính, phát triển trò chơ i và tính toán khoa học (§1.4.5). Để việc lập trình ứng dụng cấp cao trở nên hiệu quả và thuận tiện, chúng ta cần có các thư viện. Việc chỉ sử dụng các tính năng ngôn ngữ cơ bản khiến hầu hết tất cả các chư ơ ng trình trở nên khá khó khăn. Điều đó đúng với mọi ngôn ngữ có mục đích chung. Ngư ợc lại, cung cấp cho các thư viện phù hợp bất kỳ tác vụ lập trình nào cũng có thể thú vị.

Phần giới thiệu tiêu chuẩn về C++ của tôi đư ợc sử

dụng để bắt đầu: • C++ là ngôn ngữ lập trình đa năng thiên về lập trình hệ thống.

Điều này vẫn đúng. Điều đã thay đổi trong những năm qua là sự gia tăng tầm quan trọng, sức mạnh và tính linh hoạt của các cơ chế trừu tư ợng của C++:

- C++ là ngôn ngữ lập trình đa năng cung cấp một mô hình trực tiếp và hiệu quả của phần cứng kết hợp với các phư ơ ng tiện để xác định trừu tư ợng nhẹ.

Hoặc ngắn gọn hơ n:

- C++ là một ngôn ngữ để phát triển và sử dụng các khái niệm trừu tư ợng thanh lịch và hiệu quả.

Theo ngôn ngữ lập trình đa năng , ý tôi là ngôn ngữ đư ợc thiết kế để hỗ trợ nhiều mục đích sử dụng khác nhau. C++ thực sự đã đư ợc sử dụng cho nhiều mục đích sử dụng đáng kinh ngạc (từ vi điều khiển đến các ứng dụng thư ơ ng mại phân tán khổng lồ), như ng điểm mấu chốt là C++ không đư ợc chuyên biệt hóa một cách có chủ ý cho bất kỳ lĩnh vực ứng dụng nào. Không có ngôn ngữ nào là lý tư ởng cho mọi ứng dụng và mọi lập trình viên, như ng lý tư ởng cho C++ là hỗ trợ tốt phạm vi ứng dụng rộng nhất có thể.

Machine Translated by Google

Theo lập trình hệ thống, ý tôi là viết mã sử dụng trực tiếp tài nguyên phần cứng, có các hạn chế nghiêm trọng về tài nguyên hoặc tư ơ ng tác chặt chẽ với mã có sử dụng. Cụ thể, việc triển khai cơ sở hạ tầng phần mềm (ví dụ: trình điều khiển thiết bị, ngăn xếp truyền thông, máy ảo, hệ điều hành, hệ điều hành, môi trư ờng lập trình và thư viện nền tảng) hầu hết là lập trình hệ thống. Tầm quan trọng của tiêu chuẩn ''thiên về lập trình hệ thống'' trong đặc điểm lâu đời của tôi về C++ là C++ đã không đư ợc đơ n giản hóa (thỏa hiệp) bằng cách loại bỏ các phư ơ ng tiện nhằm mục đích sử dụng tài nguyên phần cứng và hệ thống ở cấp độ chuyên gia với hy vọng làm cho nó phù hợp hơ n cho các lĩnh vực ứng dụng khác.

Tất nhiên, bạn cũng có thể lập trình theo cách ẩn hoàn toàn phần cứng, sử dụng các trừu tư ợng tốn kém (ví dụ: mọi đối tư ợng trên cửa hàng miễn phí và mọi hoạt động là một chức năng ảo), sử dụng các kiểu không trang nhã (ví dụ: trừu tư ợng hóa quá mức) hoặc về cơ bản không sử dụng trừu tư ợng nào (''mã lắp ráp đư ợc tôn vinh''). Tuy nhiên, nhiều ngôn ngữ có thể làm điều đó, vì vậy đó không phải là đặc điểm phân biệt của C++.

Cuốn sách Design and Evolution of C++ [Stroustrup,1994] (đư ợc gọi là D&E) phác thảo các ý tư ởng và mục đích thiết kế của C++ một cách chi tiết hơ n, như ng cần lư u ý hai nguyên tắc: • Không có chỗ

cho ngôn ngữ cấp thấp hơ n C++ (ngoại trừ cho mã lắp ráp trong trư ờng hợp hiếm hoi). Nếu bạn có thể viết mã hiệu quả hơ n bằng ngôn ngữ cấp thấp hơ n thì ngôn ngữ đó rất có thể sẽ trở thành ngôn ngữ lập trình hệ thống đư ợc lựa chọn.

• Những gì bạn không sử dụng, bạn không phải trả tiền. Nếu các lập trình viên có thể viết tay mã hợp lý để mô phỏng một tính năng ngôn ngữ hoặc một khái niệm trừu tư ợng cơ bản và cung cấp hiệu suất thậm chí còn tốt hơ n một chút, thì sẽ có ngư ời làm như vậy và nhiều ngư ời sẽ bắt chư ớc. Do đó, một tính năng ngôn ngữ và một sự trừu tư ợng hóa cơ bản phải đư ợc thiết kế để không lãng phí một byte đơ n lẻ hoặc một chu kỳ bộ xử lý đơ n lẻ so với các lựa chọn thay thế tư ơ ng đư ơ ng. Điều này đư ợc gọi là nguyên tắc chi phí bằng không.

Đây là những nguyên tắc hà khắc, như ng cần thiết trong một số (như ng rõ ràng không phải tất cả) bối cảnh. Đặc biệt, nguyên tắc chi phí bằng không liên tục dẫn C++ đến các cơ sở đơ n giản hơ n, tao nhã hơ n và mạnh mẽ hơ n so với hình dung ban đầu. STL là một ví dụ (§4.1.1, §4.4, §4.5, Chư ơ ng 31, Chư ơ ng 32, Chư ơ ng 33). Những nguyên tắc này rất cần thiết trong nỗ lực nâng cao trình độ lập trình.

1.2.1 Phong cách lập trình

Các tính năng ngôn ngữ tồn tại để cung cấp hỗ trợ cho các phong cách lập trình. Vui lòng không xem một tính năng ngôn ngữ riêng lẻ như một giải pháp, mà hãy xem một viên gạch xây dựng từ một tập hợp đa dạng có thể đư ợc kết hợp để thể hiện các giải pháp.

Những lý tư ởng chung cho thiết kế và lập trình có thể đư ợc diễn đạt một cách đơ n

giản: • Diễn đạt ý tư ởng trực tiếp

bằng mã. • Thể hiện độc lập các ý tư ởng độc lập trong code.

• Thể hiện trực tiếp mối quan hệ giữa các ý tư ởng trong mã. • Tự

do kết hợp các ý tư ởng đư ợc thể hiện trong mã – ở đâu và chỉ ở đâu mà sự kết hợp có ý nghĩa. • Diễn đạt ý đơ n giản một cách đơ n giản.

Đây là những lý tư ởng đư ợc nhiều ngư ời chia sẻ, như ng các ngôn ngữ đư ợc thiết kế để hỗ trợ chúng có thể khác nhau đáng kể. Lý do cơ bản cho điều đó là một ngôn ngữ là hiện thân của một tập hợp các cân bằng kỹ thuật phản ánh các nhu cầu, thị hiếu và lịch sử khác nhau của các cá nhân và cộng đồng khác nhau. Câu trả lời của C++ đối với những thách thức thiết kế chung đư ợc định hình bởi nguồn gốc của nó trong lập trình hệ thống (quay trở lại C và BCPL [Richards, 1980]), mục đích của nó là giải quyết các vấn đề về độ phức tạp của chư ơ ng trình thông qua trừu tư ợng hóa (quay trở lại Simula) và lịch sử của nó.

Machine Translated by Google

Các tính năng của ngôn ngữ C++ hỗ trợ trực tiếp nhất bốn phong cách lập trình: • Lập trình thủ tục • Trừu tượng hóa dữ liệu • Lập trình hướng đối tượng • Lập trình chung

Tuy nhiên, điểm nhấn là sự hỗ trợ của sự kết hợp hiệu quả của những điều đó. Giải pháp tốt nhất (chính nhất, dễ đọc nhất, nhỏ nhất, nhanh nhất, v.v.) cho hầu hết các vấn đề không cần thiết có xu hướng là giải pháp kết hợp các khía cạnh của các phong cách này.

Như thường lệ với các thuật ngữ quan trọng trong thế giới máy tính, rất nhiều định nghĩa về các thuật ngữ này phổ biến ở nhiều bộ phận khác nhau của ngành công nghiệp máy tính và giới học thuật. Ví dụ: cái mà tôi gọi là ''kiểu lập trình'', những người khác gọi là ''kỹ thuật lập trình'' hoặc ''mô hình''. Tôi thích sử dụng ''kỹ thuật lập trình'' hơn cho thứ gì đó hạn chế hơn và ngôn ngữ -cụ thể. Tôi cảm thấy không thoải mái với từ ''khuôn mẫu'' như là sự tự phụ và (từ định nghĩa ban đầu của Kuhn) có ngụ ý tuyên bố về tính độc quyền.

Lý tưởng của tôi là các phương tiện ngôn ngữ có thể được sử dụng kết hợp một cách tao nhã để hỗ trợ sự liên tục của các phong cách lập trình và nhiều kỹ thuật lập trình khác nhau. • Lập trình thủ tục: Đây là lập trình tập trung vào xử lý và thiết kế các cấu trúc dữ liệu phù hợp. Đó là những gì C được thiết kế để hỗ trợ (và Algol, Fortran, cũng như nhiều ngôn ngữ khác). Sự hỗ trợ của C++ xuất hiện dưới dạng các kiểu dựng sẵn, toán tử, câu lệnh, hàm, cấu trúc , hiệp hội, v.v. Với những ngoại lệ nhỏ, C là một tập hợp con của C++. So với C, C++ hỗ trợ thêm cho lập trình thủ tục dưới dạng nhiều cấu trúc ngôn ngữ bổ sung và một hệ thống kiểu iv e hỗ trợ chặt chẽ hơn, linh hoạt hơn và nhiều hơn.

• Trừu tượng hóa dữ liệu: Đây là lập trình tập trung vào việc thiết kế các giao diện, ẩn các chi tiết triển khai nói chung và các biểu diễn nói riêng. C++ hỗ trợ các lớp cụ thể và trừu tượng. Các phương tiện để định nghĩa các lớp với các chi tiết triển khai riêng, hàm tạo và hàm hủy và các hoạt động liên quan trực tiếp hỗ trợ điều này. Khái niệm về một lớp trừu tượng cung cấp hỗ trợ trực tiếp cho việc ẩn dữ liệu hoàn chỉnh.

• Lập trình hướng đối tượng: Đây là lập trình tập trung vào thiết kế, triển khai và sử dụng hệ thống phân cấp lớp. Ngoài việc cho phép định nghĩa các mạng lớp, C++ cung cấp nhiều tính năng để điều hướng các mạng lớp và để đơn giản hóa việc định nghĩa một lớp từ những cái hiện có. Hệ thống phân cấp lớp cung cấp tính đa hình thời gian chạy (§20.3.2, §21.2) và đóng gói (§20.4, §20.5). • Lập trình chung: Đây là lập trình tập trung vào việc thiết kế, triển khai và sử dụng các thuật toán chung. Ở đây, ''chung'' có nghĩa là một thuật toán có thể được thiết kế để chấp nhận nhiều loại khác nhau miễn là chúng đáp ứng các yêu cầu của thuật toán trên các đối số của nó.

Mẫu là hỗ trợ chính của C++ cho lập trình chung. Các mẫu cung cấp tính đa hình tham số (thời gian biên dịch).

Bất cứ điều gì làm tăng tính linh hoạt hoặc hiệu quả của các lớp đều cải thiện sự hỗ trợ của tất cả các kiểu đó. Do đó, C++ có thể (và đã) được gọi là hướng lớp.

Mỗi phong cách thiết kế và lập trình này đã góp phần tạo nên sự tổng hợp đó là C++.

Tập trung hoàn toàn vào một trong những phong cách này là một sai lầm: ngoại trừ các ví dụ về đồ chơi, làm như vậy dẫn đến lãng phí nỗ lực phát triển và mã dưới mức tối ưu (không linh hoạt, dài dòng, hoạt động kém, không thể bảo trì, v.v.).

Machine Translated by Google

Tôi nhăn mặt khi ai đó chỉ mô tả đặc điểm của C++ thông qua một trong các phong cách này (ví dụ: ''C++ là ngôn ngữ hư ớng đối tư ợng'') hoặc sử dụng một thuật ngữ (ví dụ: ''lai'' hoặc ''mô hình hỗn hợp'') để ngụ ý rằng một ngôn ngữ hạn chế hơ n sẽ đư ợc ư a thích hơ n. Cái trư ớc bỏ qua thực tế là tất cả các phong cách đư ợc đề cập đã đóng góp một điều gì đó quan trọng cho sự tổng hợp; cái sau phủ nhận tính hợp lệ của tổng hợp. Các kiểu đư ợc đề cập không phải là các lựa chọn thay thế riêng biệt: mỗi kiểu đóng góp các kỹ thuật cho một kiểu lập trình hiệu quả và biểu cảm hơ n và C++ cung cấp hỗ trợ ngôn ngữ trực tiếp để sử dụng kết hợp chúng.

Ngay từ khi mới thành lập, thiết kế của C++ hư ớng đến sự tổng hợp giữa các phong cách lập trình và thiết kế. Ngay cả tài khoản đư ợc xuất bản sớm nhất về C++ [Stroustrup, 1982] cũng trình bày các ví dụ sử dụng kết hợp các kiểu khác nhau này và trình bày các tính năng ngôn ngữ nhằm hỗ trợ các kết hợp đó:

- Các lớp hỗ trợ tất cả các kiểu đư ợc đề cập ; tất cả đều dựa vào ngư ời dùng đại diện cho các ý tư ởng dư ới dạng các loại hoặc đối tư ợng do ngư ời dùng xác định.
- Kiểm soát truy cập công khai/riêng tư hỗ trợ trừu tư ợng hóa dữ liệu và lập trình hư ớng đối tư ợng bằng cách phân biệt rõ ràng giữa giao diện và triển khai.
- Các hàm thành viên, hàm tạo, hàm hủy và phép gán do ngư ời dùng định nghĩa cung cấp một giao diện chức năng rõ ràng cho các đối tư ợng khi cần bằng cách trừu tư ợng hóa dữ liệu và kết hợp chư ơ ng trình hư ớng đối tư ợng. Họ cũng cung cấp một ký hiệu thống nhất khi cần thiết cho lập trình chung. Quá tải chung hơ n phải đợi đến năm 1984 và khởi tạo thống nhất cho đến năm 2010.
- Các khai báo hàm cung cấp các giao diện đư ợc kiểm tra tĩnh cụ thể cho các hàm thành viên cũng như các hàm độc lập, vì vậy chúng hỗ trợ tất cả các kiểu đư ợc đề cập. Chúng cần thiết cho quá tải. Vào thời điểm đó, C thiếu ''nguyên mẫu hàm'' như ng Simula có các khai báo hàm cũng như các hàm thành viên.
- Các hàm chung và các kiểu đư ợc tham số hóa (đư ợc tạo từ các hàm và lớp bằng cách sử dụng macro) hỗ trợ lập trình chung. Các mẫu phải đợi đến năm 1988.
- Lớp cơ sở và lớp dẫn xuất cung cấp nền tảng cho lập trình hư ớng đối tư ợng và một số các hình thức trừu tư ợng hóa dữ liệu. Các chức năng ảo phải đợi đến năm 1983.
- Nội tuyến làm cho việc sử dụng các phư ơ ng tiện này có giá cả phải chăng trong lập trình hệ thống và xây dựng thư viện thời gian chạy và không gian hiệu quả.

Các tính năng ban đầu này là các cơ chế trừu tư ợng chung, thay vì hỗ trợ cho các kiểu kết hợp chư ơ ng trình rời rạc. C++ ngày nay cung cấp hỗ trợ tốt hơ n nhiều cho thiết kế và lập trình dựa trên sự trừu tư ợng hóa nhẹ, như ng mục đích của mã thanh lịch và hiệu quả đã có ngay từ đầu.

Những phát triển kể từ năm 1981 cung cấp hỗ trợ tốt hơ n nhiều cho việc tổng hợp các phong cách lập trình (''mô hình'') đư ợc xem xét ban đầu và cải thiện đáng kể sự tích hợp của chúng.

Đối tư ợng cơ bản trong C++ có danh tính; nghĩa là, nó nằm ở một vị trí cụ thể trong bộ nhớ và có thể đư ợc phân biệt với các đối tư ợng khác có cùng giá trị (có khả năng) bằng cách so sánh các địa chỉ. Biểu thức biểu thị các đối tư ợng như vậy đư ợc gọi là giá trị (§6.4). Tuy nhiên, ngay từ những ngày đầu tiên của tổ tiên C++ [Barron, 1963] cũng đã có những đối tư ợng không có danh tính (những đối tư ợng mà địa chỉ không thể đư ợc lư u trữ an toàn để sử dụng sau này). Trong C++11, khái niệm về giá trị này đã đư ợc phát triển thành một khái niệm về giá trị có thể đư ợc di chuyển với giá rẻ (§3.3.2, §6.4.1, §7.7.2). Các đối tư ợng như vậy là cơ sở của các kỹ thuật giống với những gì đư ợc tìm thấy trong lập trình chức năng (trong đó khái niệm về các đối tư ợng có danh tính đư ợc coi là kinh dị). Điều này bổ sung một cách độc đáo cho các kỹ thuật và tính năng ngôn ngữ (ví dụ: biểu thức lambda) đư ợc phát triển chủ yếu cho lập trình chung. Nó cũng giải quyết các vấn đề cổ điển liên quan đến ''các kiểu dữ liệu trừu tư ợng đơ n giản'', chẳng hạn như cách trả về một ma trận lớn một cách tinh tế và hiệu quả từ một phép toán (ví dụ: ma trận +).

Machine Translated by Google

Ngay từ những ngày đầu tiên, các chương trình C++ và bản thân thiết kế của C++ đã được quan tâm về quản lý tài nguyên. Lý tưởng là (và là) để quản lý tài nguyên được

- đơn giản (đối với người triển khai và đặc biệt là đối với người dùng),
- chung (tài nguyên là bất kỳ thứ gì phải được lấy từ một nơi nào đó và sau đó được giải phóng),
- hiệu quả (tuân thủ nguyên tắc không chi phí; §1.2),
- hoàn hảo (không có rò rỉ chấp nhận được) và
- an toàn kiểu tĩnh.

Nhiều lớp C++ quan trọng, chẳng hạn như vectơ , chuỗi , luồng, mutex, unique_ptr, fstream và regex của thư viện chuẩn , là các trình điều khiển tài nguyên. Các thư viện nền tảng và ứng dụng ngoài tiêu chuẩn đã cung cấp nhiều ví dụ khác, chẳng hạn như Ma trận và Widget. Bước đầu tiên trong việc hỗ trợ khái niệm xử lý tài nguyên được thực hiện với việc cung cấp các hàm tạo và hàm hủy trong bản nháp ''C with Classes'' đầu tiên. Điều này đã sớm được hỗ trợ với khả năng kiểm soát bản sao bằng cách xác định phép gán cũng như các hàm tạo bản sao. Việc giới thiệu các hàm tạo di chuyển và các phần gán di chuyển (§3.3) trong C++11 hoàn thiện dòng suy nghĩ này bằng cách cho phép di chuyển rẻ các đối tượng có tiềm năng lớn từ phạm vi này sang phạm vi khác (§3.3.2) và đơn giản kiểm soát thời gian tồn tại của đa hình hoặc các đối tượng được chia sẻ (§5.2.1).

Các cơ sở hỗ trợ quản lý tài nguyên cũng mang lại lợi ích trừu tượng không phải là tài nguyên tay cầm. Bất kỳ lớp nào thiết lập và duy trì một bất biến đều dựa trên một tập hợp con của các tính năng đó.

1.2.2 Kiểm tra loại

Mối liên hệ giữa ngôn ngữ mà chúng ta nghĩ/lập trình với các vấn đề và giải pháp mà chúng ta có thể tưởng tượng là rất gần. Vì lý do này, việc hạn chế các tính năng ngôn ngữ với mục đích loại bỏ các lỗi của lập trình viên là nguy hiểm nhất. Một ngôn ngữ cung cấp cho người lập trình một bộ công cụ khái niệm; nếu những thứ này không đủ cho một nhiệm vụ, chúng sẽ bị bỏ qua. Thiết kế tốt và không có lỗi không thể được đảm bảo chỉ bằng sự hiện diện hay vắng mặt của các tính năng ngôn ngữ cụ thể. Tuy nhiên, các tính năng ngôn ngữ và hệ thống loại được cung cấp cho người lập trình để thể hiện chính xác và ngắn gọn một thiết kế trong mã.

Khái niệm về các kiểu tĩnh và kiểm tra kiểu thời gian biên dịch là trung tâm để sử dụng hiệu quả C++. Việc sử dụng các loại tĩnh là chìa khóa cho tính biểu cảm, khả năng bảo trì và hiệu suất. Theo Simula, thiết kế của các loại do người dùng định nghĩa với các giao diện được kiểm tra tại thời điểm biên dịch là chìa khóa cho tính biểu cảm của C++. Hệ thống loại C ++ có thể mở rộng theo những cách không cần thiết (Chương 3, Chương 16, Chương 18, Chương 19, Chương 21, Chương 23, Chương 28, Chương 29), nhằm mục đích hỗ trợ bình đẳng cho các loại tích hợp và loại do người dùng xác định .

Các tính năng kiểm tra kiểu và ẩn dữ liệu của C++ dựa trên phân tích thời gian biên dịch của các chương trình để ngăn chặn việc vô tình làm hỏng dữ liệu. Chúng không cung cấp bí mật hoặc bảo vệ chống lại ai đó đang cố tình vi phạm các quy tắc: C++ bảo vệ chống lại tai nạn, không chống gian lận. Dù thế nào đi nữa, chúng có thể được sử dụng tự do mà không phải chịu chi phí về thời gian chạy hoặc không gian. Ý tưởng là để trở nên hữu ích, một tính năng ngôn ngữ không chỉ phải thanh lịch mà còn phải phù hợp với bối cảnh của một chương trình trong thế giới thực.

Hệ thống kiểu tĩnh của C++ rất linh hoạt và việc sử dụng các kiểu đơn giản do người dùng định nghĩa hàm ý rất ít, nếu có bất kỳ chi phí nào. Mục đích là để hỗ trợ một phong cách lập trình đại diện cho các ý tưởng riêng biệt dưới dạng các loại khác biệt, thay vì chỉ sử dụng các khái quát hóa, chẳng hạn như số nguyên, số dấu phẩy động, chuỗi, ''bộ nhớ thô'' và ''đối tượng'' ở mọi nơi . Phong cách lập trình phong phú về loại làm cho mã nhiều hơn

Machine Translated by Google

có thể đọc đư ợc, có thể bảo trì và có thể phân tích đư ợc. Một hệ thống loại tầm thư ờng chỉ cho phép phân tích tầm thư ờng, trong khi phong cách lập trình giàu loại mở ra cơ hội phát hiện và tối ư u hóa lỗi không cần thiết. Trình biên dịch C++ và các công cụ phát triển hỗ trợ phân tích dựa trên kiểu như vậy [Stroustrup,2012].

Việc duy trì hầu hết C dư ới dạng tập hợp con và duy trì ánh xạ trực tiếp tới phần cứng cần thiết cho các tác vụ lập trình hệ thống cấp thấp đòi hỏi khắt khe nhất ngụ ý khả năng phá vỡ hệ thống kiểu tĩnh. Tuy nhiên, lý tư ởng của tôi là (và luôn luôn là) loại an toàn hoàn chỉnh. Về điểm này, tôi đồng ý với Dennis Ritchie, ngư ời đã nói, ''C là một ngôn ngữ đư ợc gõ mạnh, đư ợc kiểm tra yếu''. Lư u ý rằng Simula vừa an toàn về kiểu vừa linh hoạt. Trên thực tế, lý tư ởng của tôi khi tôi bắt đầu sử dụng C++ là ''Algol68 với các lớp'' hơ n là ''C với các lớp.'' Tuy nhiên, danh sách các lý do vững chắc chống lại việc dựa trên công việc của tôi trên Algol68 an toàn kiểu [Woodward,1974] đã lâu và đau đớn. Vì vậy, an toàn kiểu hoàn hảo là một lý tư ởng mà C++ với tư cách là ngôn ngữ chỉ có thể gần đúng. Như ng đó là một lý tư ởng mà các lập trình viên C++ (đặc biệt là những ngư ời xây dựng thư viện) có thể phấn đấu đạt đư ợc. Trong những năm qua, tập hợp các tính năng ngôn ngữ, các thành phần thư viện tiêu chuẩn và các kỹ thuật hỗ trợ lý tư ởng đó đã phát triển. Bên ngoài các phần mã cấp thấp (hy vọng đư ợc phân tách bằng các giao diện an toàn kiểu), mã giao diện với mã tuân theo các quy ư ớc ngôn ngữ khác nhau (ví dụ: giao diện cuộc gọi hệ điều hành) và triển khai các trừu tư ợng cơ bản (ví dụ: chuỗi và vectơ), hiện tại không cần mã loại không an toàn nữa.

1.2.3 Khả năng tư ơ ng thích C

C++ đư ợc phát triển từ ngôn ngữ lập trình C và, với một vài ngoại lệ, giữ lại C như một tập hợp con. Những lý do chính để dựa vào C là để xây dựng trên một tập hợp các cơ sở ngôn ngữ cấp thấp đã đư ợc chứng minh và là một phần của cộng đồng kỹ thuật. Tầm quan trọng lớn đư ợc gắn liền với việc duy trì mức độ tư ơ ng thích cao với C [Koenig,1989] [Stroustrup,1994] (Chư ơ ng 44); điều này (thật không may) đã ngăn cản việc dọn dẹp cú pháp C. Sự phát triển liên tục, ít nhiều song song của C và C++ là một nguồn quan tâm thư ờng xuyên và đòi hỏi sự chú ý liên tục [Stroustrup,2002].

Có hai ủy ban chuyên lo giữ cho hai ngôn ngữ đư ợc sử dụng rộng rãi “càng tư ơ ng thích càng tốt” không phải là một cách đặc biệt tốt để tổ chức công việc. Cụ thể, có sự khác biệt về quan điểm đối với giá trị của tính tư ơ ng thích, sự khác biệt về quan điểm về điều gì tạo nên một chư ơ ng trình tốt và sự khác biệt về quan điểm về những hỗ trợ cần thiết để lập trình tốt. Chỉ duy trì liên lạc giữa các ủy ban là một khối lư ợng lớn công việc.

Khả năng tư ơ ng thích 100% C/C++ không bao giờ là mục tiêu của C++ vì điều đó sẽ hứa hẹn sự an toàn của kiểu và sự tích hợp trơ n tru của các kiểu do ngư ời dùng xác định và tích hợp sẵn. Tuy nhiên, định nghĩa của C++ đã đư ợc xem xét lại nhiều lần để loại bỏ những điểm không tư ơ ng thích vô cớ; C ++ hiện tư ơ ng thích với C hơ n so với ban đầu. C++98 đã áp dụng nhiều chi tiết từ C89 (§44.3.1).

Sau đó, khi C phát triển từ C89 [C,1990] thành C99 [C,1999], C++ đã áp dụng gần như tất cả các tính năng mới, loại bỏ VLA (mảng độ dài thay đổi) là một tính năng sai và đư ợc chỉ định các trình khởi tạo là dư thừa. Các cơ sở của C cho các tác vụ lập trình hệ thống cấp thấp đư ợc giữ lại và nâng cao; ví dụ: xem inlining (§3.2.1.1, §12.1.5, §16.2.8) và constexpr (§2.2.3, §10.4, §12.1.6).

Ngư ợc lại, C hiện đại đã đư ợc áp dụng (với mức độ trung thực và hiệu quả khác nhau) nhiều tính năng từ C++ (ví dụ: const, nguyên mẫu hàm và nội tuyến; xem [Stroustrup,2002]).

Định nghĩa của C++ đã đư ợc sửa đổi để đảm bảo rằng cấu trúc vừa là C hợp pháp vừa là C++ hợp lệ có cùng ý nghĩa trong cả hai ngôn ngữ (§44.3).

Một trong những mục tiêu ban đầu của C là thay thế mã hóa hợp ngữ cho các nhiệm vụ lập trình hệ thống đòi hỏi khắt khe nhất. Khi C++ đư ợc thiết kế, ngư ời ta đã cẩn thận để không ảnh hư ởng đến những lợi ích trong việc này.

Machine Translated by Google

khu vực. Sự khác biệt giữa C và C++ chủ yếu ở mức độ nhấn mạnh vào kiểu và cấu trúc. C là biểu cảm và cho phép. Thông qua việc sử dụng rộng rãi hệ thống kiểu, C ++ thậm chí còn biểu cảm hơ n mà không làm giảm hiệu suất.

Biết C không phải là điều kiện tiên quyết để học C++. Lập trình bằng C khuyến khích nhiều kỹ thuật và thủ thuật không cần thiết bởi các tính năng của ngôn ngữ C++. Ví dụ: chuyển đổi kiểu rõ ràng (ép kiểu) ít cần thiết hơ n trong C++ so với trong C (§1.3.3). Tuy nhiên, các chư ơ ng trình C tốt có xu hư ớng là các chư ơ ng trình C++. Ví dụ, mọi chư ơ ng trình trong Kernighan và Ritchie, The C Pro gramming Language, Second Edition [Kernighan,1988], là một chư ơ ng trình C++. Kinh nghiệm với bất kỳ ngôn ngữ gõ tĩnh nào sẽ giúp ích khi học C++.

1.2.4 Ngôn ngữ, Thư viện và Hệ thống

Các loại, toán tử và câu lệnh cơ bản (tích hợp sẵn) của C++ là những loại mà phần cứng máy tính xử lý trực tiếp: số, ký tự và địa chỉ. C++ không có các kiểu dữ liệu cấp cao tích hợp sẵn và không có các phép toán nguyên thủy cấp cao. Ví dụ, ngôn ngữ C++ không cung cấp kiểu ma trận với toán tử đảo ngư ợc hoặc kiểu chuỗi với toán tử nối. Nếu ngư ời dùng muốn một loại như vậy, nó có thể đư ợc định nghĩa bằng chính ngôn ngữ đó. Trên thực tế, việc xác định một loại mục đích chung hoặc loại dành riêng cho ứng dụng mới là hoạt động lập trình cơ bản nhất trong C++. Loại do ngư ời dùng xác định đư ợc thiết kế tốt chỉ khác với loại tích hợp sẵn ở cách nó đư ợc xác định chứ không phải ở cách nó đư ợc sử dụng. Thư viện chuẩn C++ (Chư ơ ng 4, Chư ơ ng 5, Chư ơ ng 30, Chư ơ ng 31, v.v.) cung cấp nhiều ví dụ về các kiểu như vậy và cách sử dụng chúng. Từ quan điểm của ngư ời dùng, có rất ít sự khác biệt giữa kiểu dựng sẵn và kiểu do thư viện chuẩn cung cấp. Ngoại trừ một vài sự cố không may và không quan trọng, thư viện chuẩn C++ đư ợc viết bằng C++. Viết thư viện chuẩn C++ bằng C++ là một bài kiểm tra quan trọng đối với hệ thống kiểu C++ và các cơ chế trừu tư ợng: chúng phải (và đang) đủ mạnh (biểu cảm) và hiệu quả (giá cả phải chăng) cho các tác vụ lập trình ngữ pháp hệ thống đòi hỏi khắt khe nhất. Điều này đảm bảo rằng chúng có thể đư ợc sử dụng trong các hệ thống lớn thư ờng bao gồm lớp trên lớp trừu tư ợng.

Các tính năng có thể làm phát sinh thời gian chạy hoặc chi phí bộ nhớ ngay cả khi không đư ợc sử dụng đã đư ợc loại bỏ. Ví dụ: các cấu trúc cần lư u trữ ''thông tin vệ sinh'' trong mọi đối tư ợng đã bị từ chối, vì vậy nếu ngư ời dùng khai báo một cấu trúc bao gồm hai đại lư ợng 16 bit, thì cấu trúc đó sẽ phù hợp với một thanh ghi 32 bit. Ngoại trừ các toán tử new, delete, typeid, dynamic_cast và throw cũng như khối thử, các biểu thức và câu lệnh C++ riêng lẻ không cần hỗ trợ trong thời gian chạy. Điều này có thể cần thiết cho các ứng dụng nhúng và hiệu suất cao. Cụ thể, điều này ngụ ý rằng các cơ chế trừu tư ợng hóa của C++ có thể sử dụng đư ợc cho các ứng dụng nhúng, hiệu năng cao, độ tin cậy cao và thời gian thực. Vì vậy, các lập trình viên của các ứng dụng như vậy không cần phải làm việc với một tập hợp các tính năng ngôn ngữ cấp thấp (dễ bị lỗi, nghèo nàn và không hiệu quả).

C++ đư ợc thiết kế để sử dụng trong môi trư ờng thời gian chạy và biên dịch truyền thống: môi trư ờng lập trình lập trình C trên hệ thống UNIX [UNIX,1985]. May thay, C++ không bao giờ bị giới hạn trong UNIX; nó chỉ đơ n giản sử dụng UNIX và C làm mô hình cho các mối quan hệ giữa ngôn ngữ, thư viện, trình biên dịch, trình liên kết, môi trư ờng thực thi, v.v. Mô hình tối thiểu đó đã giúp C++ thành công trên mọi nền tảng máy tính. Tuy nhiên, có những lý do chính đáng để sử dụng C++ trong các môi trư ờng cung cấp hỗ trợ thời gian chạy nhiều hơ n đáng kể. Các tiện ích như tải động, biên dịch gia tăng và cơ sở dữ liệu định nghĩa kiểu có thể đư ợc sử dụng tốt mà không ảnh hư ởng đến ngôn ngữ.

Machine Translated by Google

Không phải mọi đoạn mã đều có thể đư ợc cấu trúc tốt, độc lập với phần cứng, dễ đọc, v.v. C++ sở hữu các tính năng nhằm mục đích thao tác với các cơ sở phần cứng theo cách trực tiếp và hiệu quả mà không cần quan tâm đến tính an toàn hoặc tính dễ hiểu. Nó cũng sở hữu các phư ơng tiện để ẩn mã như vậy đằng sau các giao diện thanh lịch và an toàn.

Đư ơng nhiên, việc sử dụng C++ cho các chư ơng trình lớn hơ n dẫn đến việc sử dụng C++ bởi các nhóm ngư ời lập trình. C ++ nhấn mạnh vào tính mô đun, giao diện đư ợc gõ mạnh và tính linh hoạt đư ợc đền đáp ở đây. Bao giờ hết, khi các chư ơng trình trở nên lớn hơ n, các vấn đề liên quan đến sự phát triển và bảo trì của chúng chuyển từ vấn đề ngôn ngữ sang vấn đề toàn cầu hơ n về công cụ và quản lý.

Cuốn sách này nhấn mạnh các kỹ thuật để cung cấp các phư ơng tiện có mục đích chung, các loại hữu ích nói chung, các thư viện, v.v. Những kỹ thuật này sẽ phục vụ các lập trình viên của các chư ơng trình nhỏ cũng như các lập trình viên lập trình của các chư ơng trình lớn. Hơ n nữa, bởi vì tất cả các chư ơng trình không cần thiết bao gồm nhiều phần độc lập bán độc lập, các kỹ thuật để viết các phần như vậy phục vụ cho các lập trình viên của tất cả các ứng dụng.

Tôi sử dụng việc triển khai và sử dụng các thành phần thư viện chuẩn, chẳng hạn như vectơ , làm ví dụ. Phần này giới thiệu các thành phần thư viện và các khái niệm thiết kế cơ bản và kỹ thuật triển khai của chúng. Những ví dụ như vậy cho thấy cách các lập trình viên có thể thiết kế và triển khai các thư viện của riêng họ. Tuy nhiên, nếu thư viện tiêu chuẩn cung cấp một thành phần giải quyết vấn đề, thì việc sử dụng thành phần đó hầu như luôn luôn tốt hơ n là xây dựng thành phần của riêng bạn. Ngay cả khi thành phần tiêu chuẩn đư ợc cho là kém hơ n một chút so với thành phần tự chế cho một vấn đề cụ thể, thì thành phần tiêu chuẩn có thể đư ợc áp dụng rộng rãi hơ n, phổ biến rộng rãi hơ n và đư ợc biết đến rộng rãi hơ n. Về lâu dài, thành phần tiêu chuẩn (có thể đư ợc truy cập thông qua giao diện tùy chỉnh thuận tiện) có khả năng giảm chi phí bảo trì, chuyển, điều chỉnh và giáo dục dài hạn.

Bạn có thể nghi ngờ rằng việc chỉ định chư ơng trình bằng cách sử dụng cấu trúc kiểu chi tiết hơ n sẽ làm tăng kích thư ớc của văn bản nguồn chư ơng trình (hoặc thậm chí kích thư ớc của mã đư ợc tạo). Với C++, điều này không phải như vậy. Một chư ơng trình C++ khai báo các kiểu đối số hàm, sử dụng các lớp, v.v., thư ờng ngắn hơ n một chút so với chư ơng trình C tư ơng đư ơng không sử dụng các phư ơng tiện này. Khi các thư viện đư ợc sử dụng, một chư ơng trình C++ sẽ xuất hiện ngắn hơ n nhiều so với chư ơng trình C tư ơng đư ơng của nó, tất nhiên, giả sử rằng một chư ơng trình C tư ơng đư ơng đang hoạt động có thể đã đư ợc xây dựng.

C++ hỗ trợ lập trình hệ thống. Điều này ngụ ý rằng mã C++ có khả năng tư ơng tác hiệu quả với phần mềm đư ợc viết bằng các ngôn ngữ khác trên hệ thống. Ý tư ởng viết tất cả phần mềm bằng một ngôn ngữ duy nhất là điều viển vông. Ngay từ đầu, C++ đã đư ợc thiết kế để tư ơng tác đơ n giản và hiệu quả với C, trình dịch hợp ngữ và Fortran. Bằng cách đó, ý tôi là hàm C++, C, trình biên dịch mã chư ơng trình hoặc hàm Fortran có thể gọi các hàm trong các ngôn ngữ khác mà không cần thêm chi phí hoặc chuyển đổi cấu trúc dữ liệu đư ợc truyền giữa chúng.

C++ đư ợc thiết kế để hoạt động trong một không gian địa chỉ duy nhất. Việc sử dụng nhiều quy trình và nhiều không gian địa chỉ dựa trên sự hỗ trợ của hệ điều hành (ngoài ngôn ngữ). Đặc biệt, tôi giả định rằng một lập trình viên C++ sẽ có sẵn ngôn ngữ lệnh của hệ điều hành để soạn các quy trình thành một hệ thống. Ban đầu, tôi dựa vào UNIX Shell để làm việc đó, như ng hầu như bất kỳ ''ngôn ngữ kịch bản'' nào cũng đư ợc. Do đó, C++ không cung cấp hỗ trợ cho nhiều không gian địa chỉ và không hỗ trợ cho nhiều quy trình, như ng nó đã đư ợc sử dụng cho các hệ thống dựa trên các tính năng đó từ những ngày đầu tiên. C++ đư ợc thiết kế để trở thành một phần của các hệ thống đa ngôn ngữ lớn, đồng thời.

Machine Translated by Google

1.3 Học C++

Không có ngôn ngữ lập trình nào là hoàn hảo. May mắn thay, một ngôn ngữ lập trình không cần phải hoàn hảo để trở thành một công cụ tốt để xây dựng các hệ thống tuyệt vời. Trên thực tế, một ngôn ngữ lập trình đa năng không thể hoàn hảo cho tất cả các tác vụ mà nó được giao. Những gì hoàn hảo cho một nhiệm vụ thường là thiếu sót nghiêm trọng cho một nhiệm vụ khác bởi vì sự hoàn hảo trong một lĩnh vực ngụ ý chuyên môn hóa. Vì vậy, C++ được thiết kế để trở thành một công cụ tốt cho việc xây dựng nhiều hệ thống khác nhau và cho phép nhiều ý tưởng khác nhau được diễn đạt trực tiếp.

Không phải mọi thứ đều có thể được diễn đạt trực tiếp bằng các tính năng có sẵn của một ngôn ngữ. Trên thực tế, đó thậm chí không phải là lý tưởng. Các tính năng ngôn ngữ tồn tại để hỗ trợ nhiều phong cách lập trình và kỹ thuật. Do đó, nhiệm vụ học một ngôn ngữ nên tập trung vào việc nắm vững phong cách bản địa và tự nhiên của ngôn ngữ đó – chứ không phải hiểu từng chi tiết nhỏ của mọi đặc điểm ngôn ngữ. Viết chương trình là điều cần thiết; hiểu một ngôn ngữ lập trình không chỉ là một bài tập trí tuệ. Ứng dụng thực tế của ý tưởng là cần thiết.

Trong lập trình thực tế, có rất ít lợi thế khi biết các tính năng ngôn ngữ tối nghĩa nhất hoặc sử dụng số lượng tính năng lớn nhất. Một tính năng ngôn ngữ riêng lẻ ít được quan tâm Chỉ trong ngữ cảnh được cung cấp bởi các kỹ thuật và bởi các tính năng khác, tính năng đó mới có được ý nghĩa và sự quan tâm. Vì vậy, khi đọc các chương tiếp theo, hãy nhớ rằng mục đích thực sự của việc kiểm tra các chi tiết của C++ là để có thể sử dụng đồng bộ các tính năng ngôn ngữ và cơ sở thư viện để hỗ trợ các phong cách lập trình tốt trong bối cảnh thiết kế âm thanh.

Không có hệ thống quan trọng nào được xây dựng độc quyền về các tính năng ngôn ngữ. Chúng tôi xây dựng và sử dụng các thư viện để đơn giản hóa nhiệm vụ lập trình và nâng cao chất lượng hệ thống của chúng tôi. Chúng tôi sử dụng các thư viện để cải thiện khả năng bảo trì, tính di động và hiệu suất. Các khái niệm ứng dụng cơ bản được biểu diễn dưới dạng trừu tượng (ví dụ: lớp, mẫu và phân cấp lớp) trong thư viện. Nhiều khái niệm lập trình cơ bản nhất được trình bày trong thư viện chuẩn. Vì vậy, học thư viện chuẩn là một phần không thể thiếu trong việc học C++. Thư viện chuẩn là kho lưu trữ nhiều kiến thức khó kiếm được về cách sử dụng tốt C++.

C++ được sử dụng rộng rãi trong giảng dạy và nghiên cứu. Điều này đã gây ngạc nhiên cho một số người – chính xác – chỉ ra rằng C++ không phải là ngôn ngữ nhỏ nhất hoặc sạch nhất từng được thiết kế. Tuy nhiên, đó là:

- Đủ rõ ràng để dạy thành công các khái niệm lập trình và thiết kế cơ bản
- Đủ toàn diện để trở thành phương tiện dạy các khái niệm và kỹ thuật nâng cao
- Đủ thực tế, hiệu quả và linh hoạt cho các dự án đòi hỏi khắt khe
- Đủ thương mại để trở thành phương tiện đưa những gì đã học vào sử dụng phi học thuật
- Có đầy đủ cho các tổ chức và cộng tác dựa trên môi trường thực thi và phát triển đa dạng

C++ là ngôn ngữ mà bạn có thể phát triển cùng.

Điều quan trọng nhất cần làm khi học C++ là tập trung vào các khái niệm cơ bản (chẳng hạn như an toàn kiểu, quản lý tài nguyên và bất biến) và các kỹ thuật lập trình (chẳng hạn như quản lý tài nguyên bằng cách sử dụng các đối tượng có phạm vi và sử dụng trình lặp trong thuật toán) và không bị lạc trong ngôn ngữ-kỹ thuật chi tiết. Mục đích của việc học một ngôn ngữ lập trình là để trở thành một lập trình viên giỏi hơn, nghĩa là trở nên hiệu quả hơn trong việc thiết kế và triển khai các hệ thống mới cũng như duy trì các hệ thống cũ. Đối với điều này, việc đánh giá cao các kỹ thuật lập trình và thiết kế quan trọng hơn nhiều so với việc hiểu tất cả các chi tiết. Sự hiểu biết về các chi tiết kỹ thuật đi kèm với thời gian và thực hành.

Machine Translated by Google

Lập trình C++ dựa trên kiểm tra kiểu tĩnh mạnh mẽ và hầu hết các kỹ thuật nhằm đạt đư ợc mức độ trừu tư ợng cao và biểu diễn trực tiếp các ý tư ởng của lập trình viên. Điều này thư ờng có thể đư ợc thực hiện mà không ảnh hư ởng đến thời gian chạy và hiệu quả không gian so với các kỹ thuật công nghệ cấp thấp hơ n. Để đạt đư ợc những lợi ích của C++, các lập trình viên đến với nó từ một ngôn ngữ khác phải học và tiếp thu phong cách và kỹ thuật lập trình C++ đặc trư ng. Điều tư ơ ng tự cũng áp dụng cho các nhà ngữ pháp chuyên nghiệp đã quen với các phiên bản C++ cũ hơ n và kém biểu cảm hơ n.

Việc áp dụng thiếu suy nghĩ các kỹ thuật hiệu quả trong ngôn ngữ này sang ngôn ngữ khác thư ờng dẫn đến mã không ổn định, hoạt động kém và khó bảo trì. Viết mã như vậy cũng gây khó chịu nhất vì mọi dòng mã và mọi thông báo lỗi của trình biên dịch đều nhắc nhở lập trình viên rằng ngôn ngữ đư ợc sử dụng khác với ''ngôn ngữ cũ''. Bạn có thể viết theo phong cách của Fortran, C, Lisp, Java , v.v., bằng bất kỳ ngôn ngữ nào, như ng làm như vậy không dễ chịu và cũng không tiết kiệm bằng một ngôn ngữ có triết lý khác. Mọi ngôn ngữ đều có thể là nguồn ý tư ởng phong phú về cách viết chư ơ ng trình C++.
Tuy nhiên, các ý tư ởng phải đư ợc chuyển đổi thành một cái gì đó phù hợp với cấu trúc chung và hệ thống loại của C++ để có hiệu quả trong C++. Trên hệ thống loại cơ bản của một ngôn ngữ, chỉ có thể có đư ợc những chiến thắng của ngư ời Pyrrhic.

Trong cuộc tranh luận đang diễn ra về việc liệu một ngư ời có cần học C trư ớc C++ hay không, tôi tin chắc rằng tốt nhất là học trực tiếp với C++. C++ an toàn hơ n và biểu cảm hơ n, đồng thời giảm nhu cầu tập trung vào các kỹ thuật cấp thấp. Bạn sẽ dễ dàng học các phần phức tạp hơ n của C cần thiết để bù đắp cho việc thiếu các phư ơ ng tiện cấp cao hơ n sau khi bạn đã tiếp xúc với tập hợp con chung của C và C++ và một số kỹ thuật cấp cao hơ n đư ợc hỗ trợ trực tiếp trong C++. Chư ơ ng 44 là hư ớng dẫn dành cho các lập trình viên chuyển từ C++ sang C, chẳng hạn, để xử lý mã kế thừa. Ý kiến của tôi về cách dạy C++ cho ngư ời mới đư ợc đại diện bởi [Stroustrup,2008].

Có một số triển khai C++ đư ợc phát triển độc lập. Chúng đư ợc hỗ trợ bởi vô số công cụ, thư viện và môi trư ờng phát triển phần mềm. Để giúp nắm vững tất cả những điều này, bạn có thể tìm thấy sách giáo khoa, sách hư ớng dẫn và vô số tài nguyên trực tuyến gây hoang mang. Nếu bạn dự định sử dụng C++ một cách nghiêm túc, tôi thực sự khuyên bạn nên có quyền truy cập vào một số nguồn như vậy. Mỗi cái đều có điểm nhấn và thành kiến riêng, vì vậy hãy sử dụng ít nhất hai.

1.3.1 Lập trình bằng C++

Câu hỏi ''Làm thế nào để một ngư ời viết chư ơ ng trình tốt bằng C++?'' rất giống với câu hỏi ''Làm thế nào để viết văn xuôi tiếng Anh tốt?'' Có hai câu trả lời: ''Biết điều bạn muốn nói'' và ' 'Luyện tập. Bắt chư ớc cách viết hay.'' Cả hai dư ờng như đều phù hợp với C++ cũng như đối với tiếng Anh – và khó theo dõi.

Lý tư ởng chính cho lập trình C++ – cũng như lập trình trong hầu hết các ngôn ngữ cấp cao hơ n – là thể hiện các khái niệm (ý tư ởng, khái niệm, v.v.) từ một thiết kế trực tiếp bằng mã. Chúng tôi cố gắng đảm bảo rằng các khái niệm mà chúng tôi nói đến, thể hiện bằng các hộp và mũi tên trên bảng trắng của chúng tôi và tìm thấy trong sách giáo khoa (không lập trình) của chúng tôi có các đối tác trực tiếp và rõ ràng trong các chư ơ ng trình của chúng tôi:

[1] Trình bày ý tư ởng trực tiếp trong mã.

[2] Thể hiện mối quan hệ giữa các ý tư ởng trực tiếp trong mã (ví dụ: mối quan hệ thứ bậc, tham số và quyền sở hữu).

[3] Trình bày các ý tư ởng độc lập một cách độc lập trong mã.

[4] Giữ những điều đơ n giản trở nên đơ n giản (không biến những điều phức tạp thành không thể).

Machine Translated by Google

Cụ thể hơ n: [5] Ư u

tiên các giải pháp kiểm tra kiểu tĩnh (nếu có).

[6] Giữ thông tin cục bộ (ví dụ: tránh các biến toàn cục, giảm thiểu việc sử dụng con trỏ).

[7] Đừng trừu tư ợng hóa quá mức (nghĩa là không khái quát hóa, giới thiệu hệ thống phân cấp giai cấp hoặc tham số hóa vư ợt quá nhu cầu và kinh nghiệm rõ ràng).

Các đề xuất cụ thể hơ n đư ợc liệt kê trong §1.3.2.

1.3.2 Gợi ý cho Lập trình viên C++

Đến nay, nhiều ngư ời đã sử dụng C++ đư ợc một hoặc hai thập kỷ. Nhiều ngư ời khác đang sử dụng C++ trong một môi trư ờng duy nhất và đã học cách chung sống với những hạn chế do các trình biên dịch đời đầu và các thư viện thế hệ đầu tiên áp đặt. Thông thư ờng, điều mà một lập trình viên C ++ có kinh nghiệm đã không nhận thấy trong nhiều năm không phải là việc giới thiệu các tính năng mới như vậy, mà là những thay đổi trong mối quan hệ giữa các tính năng làm cho các kỹ thuật lập trình mới cơ bản trở nên khả thi. Nói cách khác, những gì bạn không nghĩ đến khi lần đầu tiên học C++ hoặc thấy không thực tế có thể là một cách tiếp cận ư u việt ngày nay. Bạn chỉ tìm ra bằng cách xem xét lại những điều cơ bản.

Đọc qua các chư ơ ng theo thứ tự. Nếu bạn đã biết nội dung của một chư ơ ng, bạn có thể hoàn thành trong vài phút. Nếu bạn chư a biết nội dung, bạn sẽ học đư ợc điều gì đó bất ngờ. Tôi đã học đư ợc kha khá khi viết cuốn sách này, và tôi nghi ngờ rằng khó có lập trình viên C++ nào biết mọi tính năng và kỹ thuật đư ợc trình bày. Hơ n nữa, để sử dụng tốt ngôn ngữ, bạn cần có một quan điểm mang lại trật tự cho tập hợp các tính năng và kỹ thuật. Thông qua cách tổ chức và các ví dụ, cuốn sách này đư a ra một quan điểm như vậy.

Tận dụng cơ hội đư ợc cung cấp bởi các cơ sở C ++ 11 mới để hiện đại hóa thiết kế và chuyên nghiệp của bạn kỹ thuật ngữ pháp: [1] Sử

dụng hàm tạo để thiết lập bất biến (§2.4.3.2, §13.4, §17.2.1).

[2] Sử dụng các cặp hàm tạo/hàm hủy để đơ n giản hóa việc quản lý tài nguyên (RAII; §5.2, §13.3).

[3] Tránh ''trần'' mới và xóa (§3.2.1.2, §11.2.1).

[4] Sử dụng bộ chứa và thuật toán thay vì mảng tích hợp sẵn và mã đặc biệt (§4.4, §4.5, §7.4, Chư ơ ng 32).

[5] Ư u tiên các cơ sở thư viện tiêu chuẩn hơ n mã đư ợc phát triển tại địa phư ơ ng (§1.2.4).

[6] Sử dụng ngoại lệ, thay vì mã lỗi, để báo cáo các lỗi không thể xử lý cục bộ (§2.4.3, §13.1).

[7] Sử dụng ngữ nghĩa di chuyển để tránh sao chép các đối tư ợng lớn (§3.3.2, §17.5.2).

[8] Sử dụng unique_ptr để tham chiếu các đối tư ợng thuộc loại đa hình (§5.2.1).

[9] Sử dụng shared_ptr để tham chiếu các đối tư ợng đư ợc chia sẻ, nghĩa là các đối tư ợng không có chủ sở hữu duy nhất chịu trách nhiệm về việc phá hủy chúng (§5.2.1).

[10] Sử dụng mẫu để duy trì sự an toàn của loại tĩnh (loại bỏ phôi) và tránh sử dụng không cần thiết của hệ thống phân cấp lớp (§27.2).

Bạn cũng nên xem lại lời khuyên dành cho lập trình viên C và Java (§1.3.3, §1.3.4).

1.3.3 Gợi ý cho Lập trình viên C

Ngư ời càng biết C càng giỏi thì càng khó tránh viết C++ theo phong cách C, do đó làm mất đi nhiều lợi ích tiềm năng của C++. Vui lòng xem Chư ơ ng 44, trong đó mô tả sự khác biệt giữa C và C++.

Machine Translated by Google

[1] Đừng nghĩ C++ là C với một vài tính năng đư ợc thêm vào. C++ có thể đư ợc sử dụng theo cách đó, như ng chỉ dư ới mức tối ư u. Để có đư ợc những lợi thế thực sự lớn từ C++ so với C, bạn cần áp dụng các phong cách thiết kế và triển khai khác nhau.

[2] Không viết C bằng C++; điều đó thư ờng không tối ư u nghiêm trọng cho cả bảo trì và hiệu suất cây chùy.

[3] Sử dụng thư viện chuẩn C++ như một giáo viên về các kỹ thuật và phong cách lập trình mới. Lư u ý sự khác biệt so với thư viện chuẩn C (ví dụ: = thay vì strcpy() để sao chép và == thay vì strcmp() để so sánh).

[4] Thay thế macro hầu như không cần thiết trong C++. Sử dụng const (§7.5), constexpr (§2.2.3, §10.4), enum hoặc enum class (§8.4) để xác định các hằng số kê khai, inline (§12.1.5) để tránh phí gọi hàm, mẫu (§3.4, Chư ơ ng 23) để chỉ định các họ hàm và loại cũng như không gian tên (§2.4.2, §14.3.1) để tránh xung đột tên.

[5] Đừng khai báo một biến trư ớc khi bạn cần đến nó, mà hãy khởi tạo nó ngay lập tức. Một tuyên bố có thể xảy ra ở bất cứ đâu mà một câu lệnh có thể (§9.3), trong bộ khởi tạo câu lệnh (§9.5) và trong các điều kiện (§9.4.3).

[6] Không sử dụng malloc(). Toán tử mới (§11.2) thực hiện công việc tư ơ ng tự tốt hơ n và thay vì realloc(), hãy thử một vectơ (§3.4.2). Đừng chỉ thay thế malloc() và free() bằng ''naked'' new và delete (§3.2.1.2, §11.2.1).

[7] Tránh void , union và cast, ngoại trừ sâu bên trong việc triển khai một số chức năng hoặc lớp. Việc sử dụng chúng giới hạn sự hỗ trợ mà bạn có thể nhận đư ợc từ hệ thống loại và có thể gây hại cho hình thức. Trong hầu hết các trư ờng hợp, dàn diễn viên là dấu hiệu của lỗi thiết kế. Nếu bạn phải sử dụng chuyển đổi kiểu rõ ràng, hãy thử sử dụng một trong các kiểu chuyển đổi đư ợc đặt tên (ví dụ: static_cast; §11.5.2) để có tuyên bố chính xác hơ n về những gì bạn đang cố gắng thực hiện.

[8] Giảm thiểu việc sử dụng mảng và chuỗi kiểu C. Chuỗi thư viện chuẩn C++ (§4.2), mảng (§8.2.4) và vectơ (§4.4.1) thư ờng có thể đư ợc sử dụng để viết mã đơ n giản hơ n và dễ bảo trì hơ n so với kiểu C truyền thống. Nói chung, cố gắng không tự xây dựng những gì đã đư ợc cung cấp bởi thư viện tiêu chuẩn.

[9] Tránh dùng con trỏ số học ngoại trừ trong mã rất chuyên dụng (chẳng hạn như trình quản lý bộ nhớ) và để duyệt mảng đơ n giản (ví dụ: ++p).

[10] Đừng cho rằng thứ gì đó đư ợc viết một cách tỉ mỉ theo phong cách C (tránh các tính năng của C++ như lớp, mẫu và ngoại lệ) sẽ hiệu quả hơ n một cách thay thế ngắn hơ n (ví dụ: sử dụng các cơ sở thư viện tiêu chuẩn). Thông thư ờng (như ng tất nhiên không phải luôn luôn), điều ngư ợc lại mới đúng.

Để tuân theo các quy ư ớc về liên kết C, hàm C++ phải đư ợc khai báo là có liên kết C (§15.2.5).

1.3.4 Gợi ý cho Lập trình viên Jav a

C ++ và Java là những ngôn ngữ khá khác nhau với cú pháp tư ơ ng tự nhau. Mục tiêu của chúng khác nhau đáng kể và nhiều lĩnh vực ứng dụng của chúng cũng vậy. Java không phải là sự kế thừa trực tiếp của C++ theo nghĩa là một ngôn ngữ có thể hoạt động giống như ngư ời tiền nhiệm của nó, như ng tốt hơ n và hơ n thế nữa. Để sử dụng tốt C++, bạn cần áp dụng các kỹ thuật lập trình và thiết kế phù hợp với C++, thay vì cố gắng viết Java bằng C++. Nó không chỉ là vấn đề ghi nhớ xóa các đối tư ợng mà bạn tạo mới bởi vì bạn không thể dựa vào sự hiện diện của trình thu gom rác:

[1] Đừng chỉ bắt chư ớc phong cách Java trong C++; điều đó thư ờng không tối ư u nghiêm trọng cho cả hai chính khả năng tồn tại và hiệu suất.

Machine Translated by Google

[2] Sử dụng các cơ chế trừu tư ợng hóa C++ (ví dụ: các lớp và mẫu): đừng quay lại phong cách lập trình C vì cảm giác quen thuộc sai lầm.

[3] Sử dụng thư viện chuẩn C++ như một giáo viên về các kỹ thuật và phong cách lập trình mới.

[4] Đừng ngay lập tức phát minh ra một cơ sở duy nhất cho tất cả các lớp của bạn (một lớp Đối tư ợng). Về cơ bản, bạn có thể làm tốt hơ n nếu không có nó đối với nhiều/hầu hết các lớp.

[5] Giảm thiểu việc sử dụng biến tham chiếu và biến con trỏ: sử dụng biến cục bộ và biến thành viên (§3.2.1.2, §5.2, §16.3.4, §17.1).

[6] Hãy nhớ rằng: một biến không bao giờ là một tham chiếu mặc nhiên.

[7] Coi con trỏ là tư ơ ng đư ơ ng với tham chiếu Java của C++ (tham chiếu C++ hạn chế hơ n đã nói; không có việc nối lại các tham chiếu C++).

[8] Một chức năng không phải là ảo theo mặc định. Không phải mọi lớp đều có nghĩa là thừa kế.

[9] Sử dụng các lớp trừu tư ợng làm giao diện cho hệ thống phân cấp lớp; tránh ''các lớp cơ sở giòn'', nghĩa là, các lớp cơ sở với các thành viên dữ liệu.

[10] Sử dụng quản lý tài nguyên theo phạm vi (''Resource Acquisition Is Initialization''; RAII) khi ev là có thể.

[11] Sử dụng hàm tạo để thiết lập bất biến lớp (và ném ngoại lệ nếu không thể).

[12] Nếu cần thực hiện hành động dọn dẹp khi một đối tư ợng bị xóa (ví dụ: nằm ngoài phạm vi), hãy sử dụng hàm hủy cho việc đó. Cuối cùng, đừng bắt chư ớc (làm như vậy là đặc biệt hơ n và về lâu dài sẽ hiệu quả hơ n nhiều so với việc dựa vào các hàm hủy).

[13] Tránh ''trần trụi'' mới và xóa; thay vào đó, hãy sử dụng các vùng chứa (ví dụ: vectơ , chuỗi và bản đồ) và các lớp xử lý (ví dụ: khóa và unique_ptr).

[14] Sử dụng các hàm độc lập (hàm không phải thành viên) để giảm thiểu khớp nối (ví dụ: xem các thuật toán tiêu chuẩn) và sử dụng không gian tên (§2.4.2, Chư ơ ng 14) để giới hạn phạm vi của các hàm tự do.

[15] Không sử dụng đặc tả ngoại lệ (ngoại trừ không ngoại lệ; §13.5.1.1).

[16] Một lớp lồng trong C++ không có quyền truy cập vào một đối tư ợng của lớp kèm theo.

[17] C++ chỉ cung cấp phản ánh thời gian chạy tối thiểu nhất: dynamic_cast và typeid (Chư ơ ng 22). Dựa nhiều hơ n vào các cơ sở thời gian biên dịch (ví dụ: tính đa hình thời gian biên dịch; Chư ơ ng 27, Chư ơ ng 28).

Hầu hết lời khuyên này đều áp dụng cho các lập trình viên C#.

1.4 Lịch sử

Tôi đã phát minh ra C++, viết các định nghĩa ban đầu của nó, và thực hiện lần thực hiện đầu tiên của nó. Tôi đã chọn và mô phỏng các tiêu chí thiết kế cho C++, thiết kế các tính năng ngôn ngữ chính của nó, phát triển hoặc giúp phát triển nhiều thư viện ban đầu và chịu trách nhiệm xử lý các đề xuất mở rộng trong ủy ban tiêu chuẩn C++.

C++ đư ợc thiết kế để cung cấp các phư ơ ng tiện của Simula cho việc tổ chức chư ơ ng trình [Dahl,1970] [Dahl,1972] cùng với tính hiệu quả và linh hoạt của C cho lập trình hệ thống [Kernighan,1978] [Kernighan,1988]. Simula là nguồn ban đầu của các cơ chế trừu tư ợng hóa của C++. Khái niệm lớp (với các lớp dẫn xuất và hàm ảo) đư ợc vay mư ợn từ nó. Tuy nhiên, các mẫu và ngoại lệ đã đến với C++ sau này với các nguồn cảm hứng khác nhau.

Machine Translated by Google

Sự phát triển của C++ luôn nằm trong ngữ cảnh sử dụng của nó. Tôi đã dành nhiều thời gian để lắng nghe ngư ời dùng và tìm kiếm ý kiến của các lập trình viên có kinh nghiệm. Đặc biệt, các đồng nghiệp của tôi tại Phòng thí nghiệm AT&T Bell rất cần thiết cho sự phát triển của C++ trong thập kỷ đầu tiên của nó.

Phần này là một tổng quan ngắn gọn; nó không cố gắng đề cập đến mọi tính năng ngôn ngữ và thành phần thư viện. Hơ n nữa, nó không đi vào chi tiết. Để biết thêm thông tin, và đặc biệt để biết thêm tên của những ngư ời đã đóng góp, xem [Stroustrup,1993], [Stroustrup,2007], và [Stroustrup,1994]. Hai bài báo của tôi từ hội thảo Lịch sử ngôn ngữ lập trình ACM và cuốn sách Thiết kế và phát triển C++ của tôi (đư ợc gọi là ''D&E'') mô tả chi tiết thiết kế và sự phát triển của C++ và tài liệu ảnh hư ởng từ các ngôn ngữ lập trình khác.

Hầu hết các tài liệu đư ợc tạo ra như một phần của nỗ lực tiêu chuẩn ISO C++ đều có sẵn trực tuyến [WG21]. Trong Câu hỏi thư ờng gặp của mình, tôi cố gắng duy trì mối liên hệ giữa các cơ sở tiêu chuẩn và những ngư ời đã đề xuất và cải tiến các cơ sở đó [Stroustrup,2010]. C++ không phải là công việc của một ủy ban vô danh, vô danh hay của một ''nhà độc tài suốt đời'' đư ợc cho là toàn năng; đó là công việc của nhiều cá nhân tận tụy, có kinh nghiệm, làm việc chăm chỉ.

1.4.1 Dòng thời gian

Công việc dẫn đến C++ bắt đầu vào mùa thu năm 1979 dư ới tên ''C with Classes.'' Đây là một mốc thời gian đư ợc đơ n giản hóa:

1979 Công việc về ''C with Classes'' bắt đầu. Bộ tính năng ban đầu bao gồm các lớp và dẫn xuất các lớp, kiểm soát truy cập công khai/riêng tư , hàm tạo và hàm hủy, và khai báo hàm với kiểm tra đối số. Thư viện đầu tiên hỗ trợ các tác vụ đồng thời không ư u tiên và trình tạo số ngẫu nhiên.

1984 ''C with Classes'' đư ợc đổi tên thành C++. Đến lúc đó, C++ đã có đư ợc các hàm ảo, nạp chồng hàm và toán tử, các tham chiếu, luồng I/O và các thư viện số phức.

1985 Phiên bản thư ơ ng mại đầu tiên của C++ (14 tháng 10). Thư viện bao gồm các luồng I/O, com số plex và nhiệm vụ (lập lịch không ư u tiên).

1985 Ngôn ngữ lập trình C++ (''TC++PL,'' ngày 14 tháng 10) [Stroustrup,1986].

1989 Hư ớng dẫn tham khảo C++ đư ợc chú thích (''the ARM'').

1991 Ngôn ngữ lập trình C++, Ấn bản thứ hai [Stroustrup, 1991], trình bày cách lập trình chung sử dụng các mẫu và xử lý lỗi dựa trên các ngoại lệ (bao gồm thành ngữ quản lý tài nguyên chung ''Chuyển đổi tài nguyên là khởi tạo'').

1997 Ngôn ngữ lập trình C++, Phiên bản thứ ba [Stroustrup,1997] giới thiệu ISO C++, bao gồm các không gian tên, dynamic_cast, và nhiều cải tiến của các mẫu. Thư viện tiêu chuẩn đã thêm khung STL của các thuật toán và vùng chứa chung.

Tiêu chuẩn ISO C++ 1998 .

2002 Bắt đầu làm việc trên một tiêu chuẩn sửa đổi, có tên thông tục là C++0x.

2003 Bản sửa đổi ''sửa lỗi'' của tiêu chuẩn ISO C++ đư ợc ban hành. Báo cáo kỹ thuật C++ đã giới thiệu các thành phần thư viện chuẩn mới, chẳng hạn như biểu thức chính quy, bộ chứa không có thứ tự (bảng băm) và con trỏ quản lý tài nguyên, sau này trở thành một phần của C++0x.

2006 Báo cáo kỹ thuật ISO C++ về hiệu suất đã đư ợc phát hành để trả lời các câu hỏi về chi phí, khả năng dự đoán và kỹ thuật, chủ yếu liên quan đến lập trình hệ thống nhúng.

Machine Translated by Google

2009 C++0x đã hoàn thành tính năng. Nó cung cấp khởi tạo thống nhất, di chuyển ngữ nghĩa, đối số mẫu adic khác nhau, biểu thức lambda, bí danh loại, mô hình bộ nhớ phù hợp cho đồng thời, v.v. Thư viện tiêu chuẩn đã thêm một số thành phần, bao gồm luồng, khóa và hầu hết các thành phần từ Báo cáo kỹ thuật năm 2003.

2011 Tiêu chuẩn ISO C++ 11 đã chính thức được phê duyệt.

2012 Triển khai C++ 11 hoàn chỉnh đầu tiên xuất hiện.

2012 Bắt đầu làm việc với các tiêu chuẩn ISO C++ trong tương lai (gọi tắt là C++14 và C++17).

2013 Ngôn ngữ lập trình C++, Phiên bản thứ tư đã giới thiệu C++11.

Trong quá trình phát triển, C++11 được gọi là C++0x. Như thường lệ trong các dự án lớn, chúng tôi đã quá lạc quan về ngày hoàn thành.

1.4.2 Những năm đầu đời

Ban đầu, tôi thiết kế và triển khai ngôn ngữ này vì tôi muốn phân phối các dịch vụ của nhân UNIX trên các bộ đa xử lý và mạng cục bộ (hiện được gọi là đa lõi và cụm). Để làm được điều đó, tôi cần một số mô phỏng hướng sự kiện mà Simula sẽ là lựa chọn lý tưởng, ngoại trừ những cân nhắc về hiệu suất. Tôi cũng cần xử lý trực tiếp phần cứng và cung cấp các cơ chế lập trình đồng thời hiệu suất cao mà C sẽ là lý tưởng, ngoại trừ khả năng hỗ trợ yếu của nó đối với tính mô đun và kiểm tra kiểu. Kết quả của việc thêm các lớp kiểu Simula vào C, ''C với các lớp'', đã được sử dụng cho các dự án lớn trong đó các phương tiện để viết các chương trình sử dụng thời gian và không gian tối thiểu đã được thử nghiệm nghiêm ngặt. Nó thiếu quá tải toán tử, tham chiếu, hàm ảo, khuôn mẫu, ngoại lệ và nhiều, rất nhiều chi tiết [Stroustrup, 1982]. Việc sử dụng C++ đầu tiên bên ngoài một tổ chức nghiên cứu bắt đầu vào tháng 7 năm 1983.

Cái tên C++ (phát âm là ''see plus plus'') do Rick Mascitti đặt ra vào mùa hè năm 1983 và được tôi chọn làm tên thay thế cho ''C with Classes''. Cái tên biểu thị bản chất tiến hóa của những thay đổi từ C; ''++'' là toán tử gia tăng C. Tên ngắn hơn một chút ''C+'' là một lỗi cú pháp; nó cũng đã được sử dụng làm tên của một ngôn ngữ không liên quan. Những người sành sỏi về ngữ nghĩa C thấy C++ kém hơn ++C. Ngôn ngữ này không được gọi là D, bởi vì nó là một phần mở rộng của C, bởi vì nó không cố gắng khắc phục các vấn đề bằng cách loại bỏ các tính năng, và bởi vì đã tồn tại một số người kế nhiệm C có thể có tên là D. Đối với một cách hiểu khác về tên C++ , xem phần phụ lục của [Orwell,1949].

C++ được thiết kế chủ yếu để bạn bè của tôi và tôi không phải lập trình bằng trình biên dịch mã chương trình, C, hoặc các ngôn ngữ cấp cao đang thịnh hành lúc bấy giờ. Mục đích chính của nó là làm cho việc viết các chương trình tốt trở nên dễ dàng và thú vị hơn đối với từng lập trình viên. Trong những năm đầu, không có thiết kế giấy C++; thiết kế, tài liệu và triển khai diễn ra đồng thời. Cũng không có ''dự án C++'', hay ''ủy ban thiết kế C++''. Xuyên suốt, C++ đã phát triển để đối phó với các vấn đề mà người dùng gặp phải và là kết quả của các cuộc thảo luận giữa bạn bè, đồng nghiệp và tôi.

1.4.2.1 Đặc điểm ngôn ngữ và cơ sở vật chất thư viện

Thiết kế đầu tiên của C++ (khi đó được gọi là ''C with Classes'') bao gồm các khai báo hàm với kiểm tra kiểu đối số và chuyển đổi ẩn, các lớp có sự phân biệt công khai/riêng tư giữa giao diện và triển khai, các lớp dẫn xuất, hàm tạo và hàm hủy. Tôi đã sử dụng macro để cung cấp tham số nguyên thủy. Điều này đã được sử dụng vào giữa năm 1980. Cuối năm đó, tôi đã

Machine Translated by Google

có thể trình bày một tập hợp các cơ sở ngôn ngữ hỗ trợ một tập hợp các phong cách lập trình mạch lạc; xem §1.2.1. Nhìn lại, tôi coi việc giới thiệu hàm tạo và hàm hủy là quan trọng nhất.
Theo thuật ngữ thời bấy giờ, ''hàm tạo tạo môi trư ờng thực thi cho các hàm thành viên và hàm hủy đảo ngư ợc điều đó.'' Đây là gốc rễ của các chiến lư ợc quản lý tài nguyên của C++ (gây ra nhu cầu về ngoại lệ) và là chìa khóa của nhiều các kỹ thuật để làm cho mã ngư ời dùng ngắn và rõ ràng. Nếu có các ngôn ngữ khác vào thời điểm đó hỗ trợ nhiều hàm tạo có khả năng thực thi mã chung, thì tôi đã không (và không) biết về chúng. Các hàm hủy là mới trong C++.

C++ đư ợc phát hành thư ơ ng mại vào tháng 10 năm 1985. Đến lúc đó, tôi đã thêm nội tuyến (§12.1.5, §16.2.8), hằng số (§2.2.3, §7.5, §16.2.9), nạp chồng hàm (§12.3), tham chiếu (§7.7), nạp chồng toán tử (§3.2.1.1, Chư ơ ng 18, Chư ơ ng 19) và hàm ảo (§3.2.3, §20.3.2). Trong số các tính năng này, hỗ trợ cho tính đa hình trong thời gian chạy ở dạng hàm ảo cho đến nay là điều gây tranh cãi nhất. Tôi biết giá trị của nó từ Simula như ng không thể thuyết phục hầu hết mọi ngư ời trong thế giới lập trình hệ thống về giá trị của nó. Các lập trình viên hệ thống có xu hư ớng nghi ngờ các lệnh gọi hàm gián tiếp, và những ngư ời quen thuộc với các ngôn ngữ khác hỗ trợ lập trình hư ớng đối tư ợng khó có thể tin rằng các hàm ảo có thể đủ nhanh để hữu ích trong mã hệ thống. Ngư ợc lại, nhiều lập trình viên có nền tảng hư ớng đối tư ợng đã (và nhiều ngư ời vẫn còn) gặp khó khăn trong việc làm quen với ý tư ởng rằng bạn chỉ sử dụng các lệnh gọi hàm ảo để thể hiện lựa chọn phải đư ợc thực hiện trong thời gian chạy. Việc chống lại các chức năng ảo có thể liên quan đến việc chống lại ý tư ởng rằng bạn có thể có đư ợc các hệ thống tốt hơ n thông qua cấu trúc mã thông thư ờng hơ n đư ợc hỗ trợ bởi một ngôn ngữ lập trình chuyên nghiệp. Nhiều lập trình viên C dư ờng như bị thuyết phục rằng điều thực sự quan trọng là tính linh hoạt hoàn toàn và việc chế tạo cẩn thận từng chi tiết của chư ơ ng trình. Quan điểm của tôi là (và là) rằng chúng tôi cần mọi sự trợ giúp mà chúng tôi có thể nhận đư ợc từ các ngôn ngữ và công cụ: sự phức tạp vốn có của các hệ thống mà chúng tôi đang cố gắng xây dựng luôn nằm ngoài giới hạn của những gì chúng tôi có thể diễn đạt.

Phần lớn thiết kế của C++ đư ợc thực hiện trên bảng đen của các đồng nghiệp của tôi. Trong những năm đầu, phản hồi từ Stu Feldman, Alexander Fraser, Steve Johnson, Brian Kernighan, Doug McIlroy và Dennis Ritchie là vô giá.

Vào nửa cuối những năm 1980, tôi tiếp tục bổ sung các tính năng ngôn ngữ để phản hồi các nhận xét của ngư ời dùng. Điều quan trọng nhất trong số đó là các mẫu [Stroustrup,1988] và xử lý ngoại lệ [Koenig,1990], đư ợc coi là thử nghiệm tại thời điểm nỗ lực tiêu chuẩn bắt đầu. Khi thiết kế các mẫu, tôi buộc phải quyết định giữa tính linh hoạt, hiệu quả và kiểm tra kiểu sớm.
Vào thời điểm đó, không ai biết cách đồng thời có đư ợc cả ba và để cạnh tranh với mã kiểu C cho các ứng dụng hệ thống đòi hỏi khắt khe, tôi cảm thấy rằng mình phải chọn hai thuộc tính đầu tiên. Nhìn lại, tôi nghĩ rằng sự lựa chọn là đúng, và việc tìm kiếm các mẫu kiểm tra kiểu tốt hơ n vẫn tiếp tục [Gregor,2006] [Sutton,2011] [Stroustrup,2012a]. Thiết kế của các ngoại lệ tập trung vào việc lan truyền các ngoại lệ theo nhiều cấp độ, truyền thông tin tùy ý tới một trình xử lý lỗi và tích hợp giữa các ngoại lệ và quản lý tài nguyên bằng cách sử dụng các đối tư ợng cục bộ có hàm hủy để biểu diễn và giải phóng tài nguyên (cái mà tôi gọi một cách vụng về là '' Thu nhận tài nguyên Là khởi tạo''; §13.3).

Tôi đã tổng quát hóa cơ chế kế thừa của C++ để hỗ trợ nhiều lớp cơ sở [Strous trup,1987a]. Điều này đư ợc gọi là đa thừa kế và đư ợc coi là khó khăn và gây tranh cãi. Tôi coi nó ít quan trọng hơ n nhiều so với các mẫu hoặc ngoại lệ. Tính đa thừa kế của các lớp trừu tư ợng (thư ờng đư ợc gọi là giao diện) hiện phổ biến trong các ngôn ngữ hỗ trợ kiểm tra kiểu tĩnh và lập trình hư ớng đối tư ợng.

Machine Translated by Google

Ngôn ngữ C++ đã phát triển cùng với một số cơ sở thư viện chính được trình bày trong cuốn sách này. Ví dụ, tôi đã thiết kế các lớp phức hợp [Stroustrup,1984], vector, stack, và (I/O) stream [Stroustrup,1985] cùng với các cơ chế nạp chồng toán tử. Các lớp chuỗi và danh sách đầu tiên được phát triển bởi Jonathan Shopiro và tôi như một phần của nỗ lực tương tự. Các lớp chuỗi và danh sách của Jonathan là những lớp đầu tiên được sử dụng rộng rãi như một phần của thư viện. Lớp chuỗi từ thư viện C++ tiêu chuẩn bắt nguồn từ những nỗ lực ban đầu này. Thư viện nhiệm vụ được mô tả trong [Strous trup,1987b] là một phần của chương trình ''C with Classes'' đầu tiên từng được viết vào năm 1980. Tôi đã viết nó và các lớp liên quan của nó để hỗ trợ mô phỏng kiểu Simula. Thật không may, chúng tôi đã phải đợi đến năm 2011 (30 năm!) để nhận được hỗ trợ đồng thời được tiêu chuẩn hóa và có sẵn trên toàn cầu (§1.4.4.2, §5.3, Chương 41). Sự phát triển của cơ sở mẫu bị ảnh hưởng bởi nhiều mẫu vectơ , bản đồ, danh sách và sắp xếp do Andrew Koenig, Alex Stepanov, tôi và những người khác nghĩ ra.

C++ lớn lên trong một môi trường có vô số ngôn ngữ lập trình đã được thiết lập và thử nghiệm (ví dụ: Ada [Ichbiah,1979], Algol 68 [Woodward,1974] và ML [Paulson,1996]). Vào thời điểm đó, tôi cảm thấy thoải mái với khoảng 25 ngôn ngữ và ảnh hưởng của chúng đối với C++ được ghi lại trong [Stroustrup,1994] và [Stroustrup,2007]. Tuy nhiên, những ảnh hưởng quyết định luôn đến từ các ứng dụng mà tôi gặp phải. Đó là một chính sách có chủ ý để có sự phát triển của C++ ''theo hướng vấn đề'' hơn là bắt chước.

1.4.3 Tiêu chuẩn 1998

Sự phát triển bùng nổ của việc sử dụng C++ đã gây ra một số thay đổi. Vào khoảng năm 1987, rõ ràng là việc tiêu chuẩn hóa chính thức của C++ là không thể tránh khỏi và chúng tôi cần bắt đầu chuẩn bị nền tảng cho nỗ lực tiêu chuẩn hóa [Stroustrup, 1994]. Kết quả là một nỗ lực có ý thức để duy trì liên lạc giữa những người triển khai trình biên dịch C++ và những người dùng chính. Điều này đã được thực hiện thông qua thư điện tử và giấy và thông qua các cuộc gặp mặt trực tiếp tại các hội nghị C++ và các nơi khác.

AT&T Bell Labs đã đóng góp lớn cho C++ và cộng đồng rộng lớn hơn của nó bằng cách cho phép tôi chia sẻ bản nháp của các phiên bản sửa đổi của sổ tay tham khảo C++ với những người triển khai và người dùng. Bởi vì nhiều người trong số họ đã làm việc cho các công ty có thể được coi là cạnh tranh với AT&T, nên không nên đánh giá thấp tầm quan trọng của sự đóng góp này. Một công ty kém sáng suốt hơn có thể đã gây ra những vấn đề lớn về sự phân mảnh ngôn ngữ chỉ bằng cách không làm gì cả. Khi nó xảy ra, khoảng một trăm cá nhân từ hàng chục tổ chức đã đọc và nhận xét về những gì đã trở thành hướng dẫn tham khảo được chấp nhận rộng rãi và tài liệu cơ sở cho nỗ lực tiêu chuẩn hóa tiêu chuẩn ANSI C++. Tên của chúng có thể được tìm thấy trong The Annotated C++ Reference Manual (''the ARM'') [Ellis,1989]. Ủy ban X3J16 của ANSI được triệu tập vào tháng 12 năm 1989 theo sáng kiến của Hewlett-Packard. Vào tháng 6 năm 1991, tiêu chuẩn hóa ANSI (quốc gia Mỹ) này của C++ đã trở thành một phần của nỗ lực tiêu chuẩn hóa ISO (quốc tế) cho C++ và được đặt tên là WG21. Từ năm 1990, các ủy ban chung về tiêu chuẩn C++ này đã trở thành diễn đàn chính cho sự phát triển của C++ và sự hoàn thiện định nghĩa của nó. Tôi đã phục vụ trong suốt các ủy ban này. Đặc biệt, với tư cách là chủ tịch của nhóm làm việc cho các tiện ích mở rộng (sau này được gọi là nhóm tiến hóa), tôi chịu trách nhiệm trực tiếp xử lý các đề xuất về những thay đổi lớn đối với C++ và bổ sung các tính năng ngôn ngữ mới. Một tiêu chuẩn dự thảo ban đầu để công chúng xem xét được đưa ra vào tháng 4 năm 1995. Tiêu chuẩn ISO C++ đầu tiên (ISO/IEC 14882-1998) [C++,1998] đã được phê chuẩn bởi một cuộc bỏ phiếu toàn quốc với tỉ lệ 22-0 vào năm 1998. Một ''bản phát hành sửa lỗi'' ' của tiêu chuẩn này đã được ban hành vào năm 2003, vì vậy đôi khi bạn nghe mọi người nhắc đến C++03, nhưng đó thực chất là ngôn ngữ giống như C++98.

Machine Translated by Google

1.4.3.1 Đặc điểm ngôn ngữ

Vào thời điểm các nỗ lực tiêu chuẩn ANSI và ISO bắt đầu, hầu hết các tính năng ngôn ngữ chính đã có sẵn và đư ợc ghi lại trong ARM [Ellis, 1989]. Do đó, hầu hết công việc liên quan đến việc sàng lọc các tính năng và thông số kỹ thuật của chúng. Đặc biệt, các cơ chế mẫu đư ợc hư ởng lợi từ nhiều công việc chi tiết. Không gian tên đã đư ợc giới thiệu để đối phó với kích thư ớc tăng lên của các chư ơ ng trình C++ và số lư ợng thư viện tăng lên. Theo sáng kiến của Dmitry Lenkov từ Hewett-Packard, các phư ơ ng tiện tối thiểu để sử dụng thông tin loại thời gian chạy (RTTI; Chư ơ ng 22) đã đư ợc giới thiệu. Tôi đã bỏ những cơ sở như vậy ra khỏi C++ vì tôi thấy chúng bị lạm dụng nghiêm trọng trong Simula. Tôi đã cố gắng chấp nhận một cơ sở để thu gom rác bảo thủ tùy chọn, như ng không thành công. Chúng tôi đã phải đợi đến tiêu chuẩn năm 2011 cho điều đó.

Rõ ràng, ngôn ngữ 1998 vư ợt trội hơ n nhiều về tính năng và đặc biệt là về chi tiết đặc tả so với ngôn ngữ 1989. Tuy nhiên, không phải tất cả các thay đổi đều là cải tiến. Ngoài những lỗi nhỏ không thể tránh khỏi, hai tính năng chính đã đư ợc thêm vào mà khi nhìn lại lẽ ra không nên có:

- Thông số kỹ thuật ngoại lệ cung cấp khả năng thực thi trong thời gian chạy đối với những ngoại lệ mà một chức năng đư ợc phép đư a ra. Chúng đư ợc thêm vào theo sáng kiến tràn đầy năng lư ợng của những ngư ời từ Sun Microsys tems. Các thông số kỹ thuật ngoại lệ hóa ra còn tệ hơ n là vô dụng để cải thiện khả năng đọc, độ tin cậy và hiệu suất. Chúng không đư ợc dùng nữa (đư ợc lên lịch để loại bỏ trong tư ơ ng lai) trong tiêu chuẩn 2011. Tiêu chuẩn năm 2011 đã giới thiệu noexcept (§13.5.1.1) như một giải pháp đơ n giản hơ n cho nhiều vấn đề mà các thông số kỹ thuật ngoại lệ đư ợc

cho là sẽ giải quyết. • Rõ ràng là việc biên soạn riêng biệt các mẫu và cách sử dụng chúng sẽ là lý tư ởng [Stroustrup,1994]. Làm thế nào để đạt đư ợc điều đó dư ới những ràng buộc từ việc sử dụng các tấm tem trong thế giới thực hoàn toàn không rõ ràng. Sau một cuộc tranh luận dài trong ủy ban, một thỏa hiệp đã đạt đư ợc và một cái gì đó gọi là các mẫu xuất khẩu đã đư ợc chỉ định như một phần của tiêu chuẩn n
Đó không phải là một giải pháp hay cho vấn đề, chỉ có một nhà cung cấp triển khai expor t (Nhóm thiết kế Edi son) và tính năng này đã bị xóa khỏi tiêu chuẩn năm 2011. Chúng tôi vẫn đang tìm kiếm một giải pháp. Ý kiến của tôi là vấn đề cơ bản không phải là bản thân quá trình biên dịch riêng biệt, mà là sự khác biệt giữa giao diện và việc triển khai mẫu không đư ợc chỉ định rõ. Vì vậy, export t đã giải quyết vấn đề sai. Trong tư ơ ng lai, hỗ trợ ngôn ngữ cho ''khái niệm'' (§24.3) có thể trợ giúp bằng cách cung cấp thông số kỹ thuật chính xác của các yêu cầu mẫu.
Đây là một lĩnh vực nghiên cứu và thiết kế tích cực [Sutton,2011] [Stroustrup,2012a].

1.4.3.2 Thư viện tiêu chuẩn

Cải tiến lớn nhất và quan trọng nhất trong tiêu chuẩn năm 1998 là việc đư a STL, một khung thuật toán và bộ chứa, vào thư viện tiêu chuẩn (§4.4, §4.5, Chư ơ ng 31, Chư ơ ng 32, Chư ơ ng 33). Đó là công việc của Alex Stepanov (cùng với Dave Musser, Meng Le và những ngư ời khác) dựa trên công trình hơ n một thập kỷ về lập trình chung. Andrew Koenig, Beman Dawes và tôi đã làm nhiều việc để giúp STL đư ợc chấp nhận [Stroustrup,2007]. STL đã có ảnh hư ởng lớn trong cộng đồng C++ và hơ n thế nữa.

Ngoại trừ STL, thư viện tiêu chuẩn là một tập hợp các thành phần, chứ không phải là một thiết kế thống nhất. Tôi đã thất bại trong việc cung cấp một thư viện nền tảng đủ lớn với Phiên bản 1.0 của C++ [Stroustrup,1993], và một ngư ời quản lý AT&T (không nghiên cứu) vô ích đã ngăn tôi và các đồng nghiệp sửa lỗi đó cho Phiên bản 2.0. Điều đó có nghĩa là mọi tổ chức lớn (chẳng hạn như Borland, IBM, Microsoft và Texas Instruments) đều có thư viện nền tảng của riêng mình bởi

Machine Translated by Google

thời gian các tiêu chuẩn làm việc bắt đầu. Do đó, ủy ban bị giới hạn trong phạm vi chắp vá các thành phần dựa trên những gì luôn có sẵn (ví dụ: thư viện phức hợp), những gì có thể đư ợc thêm vào mà không can thiệp vào các thư viện của nhà cung cấp chính và những gì cần thiết để đảm bảo sự hợp tác giữa các thư viện phi tiêu chuẩn khác nhau.

Chuỗi thư viện tiêu chuẩn (§4.2, Chư ơ ng 36) có nguồn gốc từ công việc ban đầu của Jonathan Shopiro và tôi tại Bell Labs như ng đã đư ợc sửa đổi và mở rộng bởi một số cá nhân và nhóm khác nhau trong quá trình tiêu chuẩn hóa. Thư viện valarray để tính toán số (§40.5) chủ yếu là tác phẩm của Kent Budge. Jerry Schwarz đã chuyển đổi thư viện luồng của tôi (§1.4.2.1) thành thư viện iostreams (§4.3, Chư ơ ng 38) bằng cách sử dụng kỹ thuật thao tác của Andrew Koenig (§38.4.5.2) và các ý tư ởng khác. Thư viện iostreams đã đư ợc hoàn thiện hơ n nữa trong quá trình tiêu chuẩn hóa, trong đó phần lớn công việc đư ợc thực hiện bởi Jerry Schwarz, Nathan Myers và Norihiro Kumagai.

Theo tiêu chuẩn thư ơ ng mại, thư viện chuẩn C++ 98 rất nhỏ. Ví dụ: không có GUI tiêu chuẩn, thư viện truy cập cơ sở dữ liệu hoặc thư viện ứng dụng Web. Những thư viện như vậy có sẵn rộng rãi như ng không phải là một phần của tiêu chuẩn ISO. Lý do cho điều đó là thực tế và thư ơ ng mại, hơ n là kỹ thuật. Tuy nhiên, thư viện chuẩn C đã (và là) thư ớc đo của nhiều ngư ời có ảnh hư ởng về thư viện chuẩn và so với thư viện chuẩn C++ thì rất lớn.

1.4.4 Tiêu chuẩn 2011

C++ hiện tại, C++11, đư ợc biết đến trong nhiều năm với tên C++0x, là công việc của các thành viên của WG21. Ủy ban làm việc theo các quy trình và thủ tục tự áp đặt ngày càng nặng nề. Những quy trình này có thể dẫn đến một đặc điểm kỹ thuật tốt hơ n (và khắt khe hơ n), như ng chúng cũng hạn chế sự đổi mới [Stroustrup, 2007]. Một tiêu chuẩn dự thảo ban đầu để xem xét công khai đã đư ợc đư a ra vào năm 2009. Tiêu chuẩn ISO C++ thứ hai (ISO/IEC 14882-2011) [C++,2011] đã đư ợc phê chuẩn bởi một cuộc bỏ phiếu quốc gia với tỷ lệ 21-0 vào tháng 8 năm 2011.

Một lý do cho khoảng cách dài giữa hai tiêu chuẩn là hầu hết các thành viên của ủy ban (bao gồm cả tôi) đều có ấn tư ợng sai lầm rằng các quy tắc của ISO yêu cầu một ''thời gian chờ đợi'' sau khi một tiêu chuẩn đư ợc ban hành trư ớc khi bắt đầu làm việc với các tính năng mới. Do đó, công việc nghiêm túc về các tính năng ngôn ngữ mới đã không bắt đầu cho đến năm 2002. Các lý do khác bao gồm kích thư ớc ngày càng tăng của các ngôn ngữ hiện đại và thư viện nền tảng của chúng. Xét về số trang văn bản tiêu chuẩn, ngôn ngữ tăng khoảng 30% và thư viện tiêu chuẩn tăng khoảng 100%. Phần lớn sự gia tăng là do đặc điểm kỹ thuật chi tiết hơ n, chứ không phải chức năng mới. Ngoài ra, công việc trên một tiêu chuẩn C++ mới rõ ràng phải hết sức cẩn thận để không ảnh hư ởng đến mã cũ hơ n thông qua các thay đổi không tư ơ ng thích. Có hàng tỷ dòng mã C++ đang đư ợc sử dụng mà ủy ban không đư ợc vi phạm.

Mục tiêu chung của nỗ lực C++11 là: • Làm cho C++

trở thành ngôn ngữ tốt hơ n cho lập trình hệ thống và xây dựng thư viện. • Giúp cho việc dạy và học C++ trở nên dễ dàng hơ n.

Các mục tiêu đư ợc ghi lại và trình bày chi tiết trong [Stroustrup,2007].

Một nỗ lực lớn đã đư ợc thực hiện để làm cho việc lập trình hệ thống đồng thời trở nên an toàn và di động. Điều này liên quan đến một mô hình bộ nhớ (§41.2) và một tập hợp các phư ơ ng tiện để lập trình không khóa (§41.3), chủ yếu là công việc của Hans Boehm, Brian McKnight và những ngư ời khác. Trên hết, chúng tôi đã thêm thư viện chủ đề . Pete Becker, Peter Dimov, How ard Hinnant, William Kempf, Anthony Williams và những ngư ời khác đã làm rất nhiều công việc về điều đó. Để cung cấp một ví dụ về những gì có thể đạt đư ợc trên các tiện ích đồng thời cơ bản, tôi đã đề xuất nghiên cứu về ''một cách trao đổi

Machine Translated by Google

thông tin giữa các tác vụ mà không sử dụng rõ ràng khóa,'' đã trở thành tư ơ ng lai và không đồng bộ () (§5.3.5); Lawrence Crowl và Detlef Vollmann đã làm hầu hết công việc đó. Đồng thời là một lĩnh vực liệt kê đầy đủ và chi tiết ai đã làm gì và tại sao sẽ cần một bài viết rất dài. Ở đây, tôi thậm chí không thể thử.

1.4.4.1 Tính năng ngôn ngữ

Danh sách các tính năng ngôn ngữ và cơ sở thư viện tiêu chuẩn đư ợc thêm vào C++98 để có đư ợc C++11 đư ợc gửi trư ớc trong §44.2. Ngoại trừ hỗ trợ đồng thời, mọi bổ sung cho ngôn ngữ có thể đư ợc coi là "thứ yếu", như ng làm như vậy sẽ bỏ sót điểm chính: các tính năng ngôn ngữ đư ợc dùng kết hợp để viết các chư ơ ng trình tốt hơ n. Khi nói ''tốt hơ n'', ý tôi là dễ đọc hơ n, dễ viết hơ n, trang nhã hơ n, ít lỗi hơ n, dễ bảo trì hơ n, chạy nhanh hơ n, tiêu tốn ít tài nguyên hơ n, v.v.

Dư ới đây là những gì tôi coi là ''viên gạch xây dựng'' mới hữu ích nhất ảnh hư ởng đến phong cách của Mã C++11 có tham chiếu đến văn bản và tác giả chính của chúng:

- Kiểm soát giá trị mặc định: =delete và =default: §3.3.4, §17.6.1, §17.6.4; Lawrence Crowl và Bjarne Stroustrup.
- Suy ra loại đối tư ợng từ trình khởi tạo của nó, auto: §2.2.2, §6.3.6.1; Bjarne Stroustrup. Lần đầu tiên tôi thiết kế và triển khai auto vào năm 1983 như ng đã phải gỡ bỏ nó vì các vấn đề về tính tư ơ ng thích của C.
- Đánh giá biểu thức hằng tổng quát (bao gồm các kiểu chữ), constexpr: §2.2.3, §10.4, §12.1.6; Gabriel Dos Reis và Bjarne Stroustrup [DosReis,2010].
- Trình khởi tạo thành viên trong lớp: §17.4.4; Michael Spertus và Bill Seymour.
- Kế thừa hàm tạo: §20.3.5.1; Bjarne Stroustrup, Michael Wong và Michel Michaud.
- Biểu thức Lambda, một cách xác định ngầm định các đối tư ợng hàm tại điểm sử dụng chúng trong một biểu thức: §3.4.3, §11.4; Jaakko Jarvi.
- Di chuyển ngữ nghĩa, một cách truyền thông tin mà không cần sao chép: §3.3.2, §17.5.2; Howard Hinnan.
- Một cách để nói rằng một chức năng có thể không đư a ra ngoại lệ noexcept: §13.5.1.1; David Abra giăm bông, Rani Sharoni và Doug Gregor.
- Tên thích hợp cho con trỏ null, §7.2.2; Herb Sutter và Bjarne Stroustrup.
- Câu lệnh phạm vi : §2.2.5, §9.5.1; Thorsten Ottosen và Bjarne Stroustrup.
- Điều khiển ghi đè: cuối cùng và ghi đè: §20.3.4. Alisdair Meredith, Chris Uzdavinis, và Ville Voutilainen.
- Nhập bí danh, cơ chế cung cấp bí danh cho một loại hoặc một mẫu. Cụ thể, cách xác định mẫu bằng cách ràng buộc một số đối số của mẫu khác: §3.4.5, §23.6; Bjarne Stroustrup và Gabriel Dos Reis.
- Kiểu liệt kê đã nhập và phạm vi: lớp enum: §8.4.1; David E. Miller, Herb Sutter, và Bjarne Stroustrup.
- Khởi tạo phổ quát và thống nhất (bao gồm danh sách khởi tạo có độ dài tùy ý và bảo vệ chống thu hẹp): §2.2.2, §3.2.1.3, §6.3.5, §17.3.1, §17.3.4; Bjarne Stroustrup và Gabriel Dos Reis.
- Các mẫu biến thể, một cơ chế để truyền số lư ợng đối số tùy ý của tùy ý nhập vào một mẫu: §3.4.4, §28.6; Doug Gregor và Jaakko Jarvi.

Machine Translated by Google

Nhiều người hơn có thể được liệt kê ở đây xứng đáng được đề cập. Các báo cáo kỹ thuật cho ủy ban [WG21] và Câu hỏi thường gặp về C++11 của tôi [Stroustrup,2010a] đưa ra nhiều cái tên. Biên bản của các tổ công tác của ủy ban còn đề cập nhiều hơn thế. Lý do tên tôi xuất hiện thường xuyên như vậy (tôi hy vọng) không phải là phù phiếm, mà đơn giản là tôi đã chọn làm việc với những gì tôi cho là quan trọng. Đây là những tính năng sẽ phổ biến trong mã tốt. Vai trò chính của chúng là bổ sung tính năng C++ được thiết lập để hỗ trợ tốt hơn các kiểu lập trình (§1.2.1). Chúng là nền tảng của sự tổng hợp đó là C++11.

Nhiều công việc đã được đưa vào một đề xuất không được đưa vào tiêu chuẩn. ''Khái niệm'' là một cơ sở để xác định và kiểm tra các yêu cầu đối với các đối số mẫu [Gregor,2006] dựa trên nghiên cứu trước đó (ví dụ: [Stroustrup,1994] [Siek,2000] [DosReis,2006]) và công việc mở rộng trong Ủy ban. Nó đã được thiết kế, chỉ định, triển khai và thử nghiệm, nhưng theo đa số, ủy ban đã quyết định rằng đề xuất vẫn chưa sẵn sàng. Nếu chúng tôi có thể tinh chỉnh ''khái niệm'', thì đó sẽ là tính năng đơn lẻ quan trọng nhất trong C++11 (đối thủ cạnh tranh duy nhất của nó cho tiêu đề đó là hỗ trợ đồng thời). Tuy nhiên, ủy ban đã quyết định chống lại ''các khái niệm'' trên cơ sở phức tạp, khó sử dụng và hiệu suất thời gian biên dịch [Stroustrup, 2010b]. Tôi nghĩ rằng chúng tôi (ủy ban) đã làm đúng với ''khái niệm'' cho C++11, nhưng tính năng này thực sự là ''cái đã bỏ đi''. Đây hiện là một lĩnh vực đang được nghiên cứu và thiết kế tích cực [Sutton ,2011] [Stroustrup,2012a].

1.4.4.2 Thư viện tiêu chuẩn

Công việc về cái đã trở thành thư viện chuẩn C++11 bắt đầu với một báo cáo kỹ thuật của ủy ban tiêu chuẩn (''TR1''). Ban đầu, Matt Austern là người đứng đầu Nhóm Công tác Thư viện, sau đó Howard Hinnant tiếp quản cho đến khi chúng tôi gửi bản thảo tiêu chuẩn cuối cùng vào năm 2011.

Đối với các đặc điểm ngôn ngữ, tôi sẽ chỉ liệt kê một vài thành phần thư viện tiêu chuẩn có liên quan đến văn bản và tên của các cá nhân có liên quan chặt chẽ nhất với chúng. Để biết danh sách chi tiết hơn, hãy xem §44.2.2. Một số thành phần, chẳng hạn như unordered_map (bảng băm), là những thành phần đơn giản là chúng tôi đã không quản lý để hoàn thành kịp thời cho tiêu chuẩn C++98. Nhiều thứ khác, chẳng hạn như unique_ptr và chức năng là một phần của báo cáo kỹ thuật (TR1) dựa trên các thư viện Boost. Boost là một tổ chức tình nguyện được thành lập để cung cấp các thành phần thư viện hữu ích dựa trên STL [Boost].

- Các vùng chứa đã băm, chẳng hạn như unordered_map: §31.4.3; Matt Austern.
- Các thành phần thư viện đồng thời cơ bản, chẳng hạn như luồng, mutex và khóa: §5.3, §42.2; Pete Becker, Peter Dimov, Howard Hinnant, William Kempf, Anthony Williams, v.v.
- Khởi chạy tính toán không đồng bộ và trả về kết quả, tương lai, lời hứa và không đồng bộ(): §5.3.5, §42.4.6; Detlef Vollmann, Lawrence Crowl, Bjarne Stroustrup, và Herb Sutter.
- Giao diện thu gom rác: §34.5; Michael Spertus và Hans Boehm.
- Thư viện biểu thức chính quy, biểu thức chính quy: §5.5, Chương 37; John Maddock.
- Một thư viện số ngẫu nhiên: §5.6.3, §40.7; Jens Maurer và Walter Brown. Đã đến lúc. Tôi đã vận chuyển thư viện số ngẫu nhiên đầu tiên với ''C with Classes'' vào năm 1980.

Một số thành phần tiện ích đã được dùng thử trong Boost:

- Một con trỏ để truyền tài nguyên một cách đơn giản và hiệu quả, unique_ptr: §5.2.1, §34.3.1; Howard E. Hinnant. Điều này ban đầu được gọi là move_ptr và đáng lẽ auto_ptr phải là như vậy nếu chúng ta biết cách làm như vậy đối với C++98.
- Một con trỏ biểu diễn quyền sở hữu chung, shared_ptr: §5.2.1, §34.3.2; Peter Dimov. MỘT kế thừa cho đề xuất Count_ptr C++98 từ Greg Colvin.

Machine Translated by Google

• Thư viện tuple : §5.4.3, §28.5, §34.2.4.2; Jaakko Jarvi và Gary Powell. Họ ghi nhận một danh sách dài những ngư ời đóng góp, bao gồm Doug Gregor, David Abrahams và Jeremy Siek. • Liên kết chung (): §33.5.1; Peter Dimov. Lời cảm ơ n của anh ấy liệt kê một ngư ời thực sự là ai của Boost (bao gồm Doug Gregor, John Maddock, Dave Abrahams và Jaakko Jarvi). • Loại chức năng để giữ các đối tư ợng có thể gọi: §33.5.3; Doug Gregor. Ông tin tư ởng William Kempf và những ngư ời khác có đóng góp.

1.4.5 C++ dùng để làm gì?

Đến bây giờ (2013), C++ đư ợc sử dụng gần như ở khắp mọi nơ i: nó có trong máy tính, điện thoại, ô tô của bạn, thậm chí có thể là trong máy ảnh của bạn. Bạn không thư ờng nhìn thấy nó. C++ là một ngôn ngữ lập trình hệ thống và những ứng dụng phổ biến nhất của nó nằm sâu trong cơ sở hạ tầng mà chúng ta, với tư cách là ngư ời dùng, không bao giờ để ý đến.

C++ đư ợc sử dụng bởi hàng triệu lập trình viên trong mọi lĩnh vực ứng dụng. Hàng tỷ (hàng nghìn triệu) dòng C++ hiện đang đư ợc triển khai. Việc sử dụng rộng rãi này đư ợc hỗ trợ bởi nửa tá triển khai độc lập, hàng nghìn thư viện, hàng trăm sách giáo khoa và hàng chục trang web. Đào tạo và giáo dục ở nhiều cấp độ khác nhau đư ợc phổ biến rộng rãi.

Các ứng dụng ban đầu có xu hư ớng mang hư ơ ng vị lập trình hệ thống mạnh mẽ. Ví dụ, một số hệ điều hành ban đầu đã đư ợc viết bằng C++: [Campbell,1987] (học thuật), [Rozier,1988] (thời gian thực), [Berg,1995] (I/O thông lư ợng cao). Nhiều hệ điều hành hiện tại (ví dụ: Windows, HĐH của Apple, Linux và hầu hết các HĐH dành cho thiết bị di động) có các phần chính đư ợc thực hiện bằng C++. Điện thoại di động và bộ định tuyến Internet của bạn rất có thể đư ợc viết bằng C++. Tôi coi hiệu quả ở mức độ thấp không thỏa hiệp là điều cần thiết cho C++. Điều này cho phép chúng tôi sử dụng C++ để viết trình điều khiển thiết bị và phần mềm khác dựa trên thao tác trực tiếp với phần cứng dư ới các ràng buộc thời gian thực. Trong mã như vậy, khả năng dự đoán hiệu suất ít nhất cũng quan trọng như tốc độ thô. Thông thư ờng, tính nhỏ gọn của hệ thống kết quả cũng vậy. C++ đư ợc thiết kế sao cho mọi tính năng của ngôn ngữ đều có thể sử dụng đư ợc trong mã dư ới những hạn chế nghiêm trọng về thời gian và không gian (§1.2.4) [Stroustrup,1994,§4.5].

Một số hệ thống đư ợc sử dụng rộng rãi và dễ thấy nhất hiện nay có các phần quan trọng đư ợc viết bằng C++. Ví dụ như Amadeus (bán vé máy bay), Amazon (Thư ơ ng mại trên web), Bloomberg (thông tin tài chính), Google (tìm kiếm trên web) và Facebook (truyền thông xã hội). Nhiều ngôn ngữ và công nghệ lập trình khác phụ thuộc rất nhiều vào hiệu suất và độ tin cậy của C++ trong quá trình triển khai của chúng.
Các ví dụ bao gồm Máy ảo Java đư ợc sử dụng rộng rãi nhất (ví dụ: HotSpot của Oracle), trình thông dịch JavaScript (ví dụ: V8 của Google), trình duyệt (ví dụ: Internet Explorer của Microsoft, Firefox của Mozilla, Safari của Apple và Chrome của Google) và các khung ứng dụng (ví dụ: Khung dịch vụ Web .NET của Microsoft). Tôi coi C++ có thế mạnh riêng trong lĩnh vực phần mềm cơ sở hạ tầng [Stroustrup,2012a].

Hầu hết các ứng dụng đều có các phần mã rất quan trọng để có hiệu suất chấp nhận đư ợc. Tuy nhiên, lư ợng mã lớn nhất không nằm trong các phần như vậy. Đối với hầu hết các mã, khả năng bảo trì, dễ mở rộng và dễ kiểm tra là chìa khóa. Sự hỗ trợ của C++ đối với những mối quan tâm này đã dẫn đến việc nó đư ợc sử dụng rộng rãi trong các lĩnh vực mà độ tin cậy là bắt buộc và khi các yêu cầu thay đổi đáng kể theo thời gian. Ví dụ như các hệ thống tài chính, viễn thông, điều khiển thiết bị và các ứng dụng quân sự. Trong nhiều thập kỷ, việc điều khiển trung tâm hệ thống điện thoại đư ờng dài của Hoa Kỳ đã dựa vào C++, và cứ 800 cuộc gọi (nghĩa là cuộc gọi do bên đư ợc gọi trả tiền) đã đư ợc định tuyến bởi một chư ơ ng trình C++ [Kamath, 1993]. Nhiều ứng dụng như vậy là lớn và tồn tại lâu dài. Kết quả là tính ổn định, tính tư ơ ng thích và khả năng mở rộng luôn là mối quan tâm hàng đầu trong quá trình phát triển của C++. Các chư ơ ng trình C++ nhiều triệu dòng là phổ biến.

Machine Translated by Google

Trò chơ i là một lĩnh vực khác mà nhiều ngôn ngữ và công cụ cần phải cùng tồn tại với một ngôn ngữ mang lại hiệu quả vư ợt trội (thư ờng trên phần cứng "không bình thư ờng"). Do đó, trò chơ i là một lĩnh vực ứng dụng chính khác của C++.

Cái từng đư ợc gọi là lập trình hệ thống đư ợc tìm thấy rộng rãi trong các hệ thống nhúng, vì vậy không có gì ngạc nhiên khi tìm thấy việc sử dụng rộng rãi C++ trong các dự án hệ thống nhúng đòi hỏi khắt khe, bao gồm chụp cắt lớp máy tính (máy quét CAT), phần mềm điều khiển chuyến bay (ví dụ: Lockheed-Martin) , điều khiển tên lửa, động cơ tàu (ví dụ: điều khiển động cơ diesel hàng hải lớn nhất thế giới của MAN), phần mềm mật ô tô (ví dụ: BMW) và điều khiển tuabin gió (ví dụ: Vesta).

C ++ không đư ợc thiết kế đặc biệt với tính toán số. Tuy nhiên, nhiều tính toán số, khoa học và kỹ thuật đư ợc thực hiện trong C++. Lý do chính cho điều này là công việc tính toán truyền thống thư ờng phải đư ợc kết hợp với đồ họa và tính toán dựa trên cấu trúc dữ liệu không phù hợp với khuôn mẫu Fortran truyền thống (ví dụ: [Root,1995]). Tôi đặc biệt hài lòng khi thấy C++ đư ợc sử dụng trong các nỗ lực khoa học quan trọng, chẳng hạn như Dự án Hệ gen Ngư ời, Tàu thám hiểm Sao Hỏa của NASA, cuộc tìm kiếm các nguyên tắc cơ bản của vũ trụ của CERN và nhiều dự án khác.

Khả năng của C++ đư ợc sử dụng hiệu quả cho các ứng dụng đòi hỏi phải làm việc trong nhiều lĩnh vực ứng dụng khác nhau là một thế mạnh quan trọng. Các ứng dụng liên quan đến kết nối mạng cục bộ và diện rộng, số học, đồ họa, tư ơ ng tác ngư ời dùng và truy cập cơ sở dữ liệu là phổ biến. Theo truyền thống, các lĩnh vực ứng dụng như vậy đư ợc coi là khác biệt và đư ợc phục vụ bởi các cộng đồng kỹ thuật riêng biệt sử dụng nhiều ngôn ngữ lập trình. Tuy nhiên, C++ đư ợc sử dụng rộng rãi trong tất cả các lĩnh vực đó và hơ n thế nữa. Nó đư ợc thiết kế sao cho mã C++ có thể cùng tồn tại với mã đư ợc viết bằng các ngôn ngữ khác. Ở đây, một lần nữa, sự ổn định của C++ qua nhiều thập kỷ là rất quan trọng. Hơ n nữa, không có hệ thống thực sự chính nào đư ợc viết 100% bằng một ngôn ngữ duy nhất. Do đó, mục đích thiết kế ban đầu của C++ về khả năng tư ơ ng tác trở nên quan trọng.

Các ứng dụng chính không đư ợc viết bằng ngôn ngữ thô. C++ đư ợc hỗ trợ bởi nhiều thư viện (ngoài thư viện chuẩn ISO C++) và các bộ công cụ, chẳng hạn như Boost [Boost] (thư viện nền tảng di động), POCO (Phát triển web), QT (phát triển ứng dụng đa nền tảng), wxWidgets (thư viện GUI đa nền tảng), WebKit (thư viện công cụ bố cục cho trình duyệt Web), CGAL (hình học tính toán), QuickFix (Thông tin tài chính eXchange), OpenCV (xử lý ảnh thời gian thực) và Root [Root,1995] (Cao -Vật lý năng lư ợng). Có hàng nghìn thư viện C++ nên việc theo kịp tất cả chúng là điều không thể.

1.5 Lời khuyên

Mỗi chư ơ ng có phần ''Lời khuyên'' với một tập hợp các khuyến nghị cụ thể liên quan đến nội dung của nó. Những lời khuyên như vậy bao gồm các quy tắc thô sơ , không phải là quy luật bất biến. Một lời khuyên chỉ nên đư ợc áp dụng khi hợp lý. Không gì có thể thay thế đư ợc trí thông minh, kinh nghiệm, ý thức chung và khiếu thẩm mỹ.

Tôi thấy các quy tắc dạng ''không bao giờ làm điều này'' là không hữu ích. Do đó, hầu hết các lời khuyên đều đư ợc diễn đạt dư ới dạng gợi ý cho những việc cần làm. Những đề xuất tiêu cực có xu hư ớng không đư ợc coi là những điều cấm tuyệt đối và tôi cố gắng đề xuất những giải pháp thay thế. Tôi biết không có tính năng chính nào của C++ mà tôi chư a thấy đư ợc sử dụng tốt. Phần ''Lời khuyên'' không có phần giải thích. Thay vào đó, mỗi lời khuyên đều đư ợc đi kèm với phần tham chiếu đến một phần thích hợp của cuốn sách.

Đối với những ngư ời mới bắt đầu, đây là một số khuyến nghị cấp cao bắt nguồn từ các phần về thiết kế, học tập và lịch sử của C++:

Machine Translated by Google

[1] Biểu diễn các ý tư ởng (khái niệm) trực tiếp trong mã, chẳng hạn như dư ới dạng một hàm, một lớp hoặc một enu sự hợp nhất; §1.2.

[2] Đặt mục tiêu mã của bạn vừa thanh lịch vừa hiệu quả; §1.2.

[3] Đừng trừu tư ợng hóa quá mức; §1.2.

[4] Thiết kế tập trung vào việc cung cấp các trừu tư ợng thanh lịch và hiệu quả, có thể đư ợc trình bày dư ới dạng thư viện; §1.2.

[5] Thể hiện mối quan hệ giữa các ý tư ởng trực tiếp trong mã, ví dụ, thông qua tham số hóa tion hoặc một hệ thống phân cấp lớp; §1.2.1.

[6] Trình bày các ý tư ởng độc lập một cách riêng biệt trong mã, ví dụ, tránh sự phụ thuộc lẫn nhau giữa các lớp; §1.2.1.

[7] C++ không chỉ hư ớng đối tư ợng; §1.2.1.

[8] C++ không chỉ dành cho lập trình chung; §1.2.1.

[9] Thích các giải pháp có thể đư ợc kiểm tra tĩnh; §1.2.1.

[10] Làm rõ các tài nguyên (biểu diễn chúng dư ới dạng các đối tư ợng lớp); §1.2.1, §1.4.2.1.

[11] Diễn đạt những ý kiến đơ n giản một cách đơ n giản; §1.2.1.

[12] Sử dụng các thư viện, đặc biệt là thư viện tiêu chuẩn, thay vì cố gắng xây dựng mọi thứ từ cào; §1.2.1.

[13] Sử dụng kiểu lập trình phong phú; §1.2.2.

[14] Mã cấp thấp không nhất thiết phải hiệu quả; không tránh các lớp, mẫu và các thành phần thư viện tiêu chuẩn vì sợ các vấn đề về hiệu suất; §1.2.4, §1.3.3.

[15] Nếu dữ liệu có tính bất biến, hãy đóng gói nó; §1.3.2.

[16] C++ không chỉ là C với một vài phần mở rộng; §1.3.3.

Nói chung: Để viết một chư ơ ng trình tốt cần có trí thông minh, óc thẩm mỹ và sự kiên nhẫn. Bạn sẽ không hiểu ngay lần đầu tiên. Cuộc thí nghiệm!

1.6 Tài liệu tham khảo

[Hàu, 2003] Matt Austern và cộng sự: Gỡ rối sự cân bằng và tìm kiếm của cây tìm kiếm nhị phân cân bằng. Phần mềm – Thực hành & Trải nghiệm. Tập 33, Số 13. tháng 11 năm 2003.

[Barron,1963] DW Barron và cộng sự: Các tính năng chính của CPL. Tạp chí Máy tính. 6(2): 134. (1963). comjnl.oxfordjournals.org/content/6/2/134.full.pdf+html.

[Barton,1994] JJ Barton và LR Nackman: Khoa học và Kỹ thuật C++: Phần giới thiệu với các ví dụ và kỹ thuật nâng cao. Addison-Wesley. Reading, Massachusetts. 1994. ISBN 0-201-53393-6.

[Berg,1995] William Berg, Marshall Cline, và Mike Girou: Bài học rút ra từ Dự án OS/400 OO. CACM. tập 38, số 10. Tháng 10 năm 1995.

[Boehm,2008] Hans-J. Boehm và Sarita V. Adve: Nền tảng của mô hình bộ nhớ tư ơ ng tranh C++. ACM PLDI'08.

[Tăng] Bộ sư u tập thư viện Boost. www.boost.org.

[Budge,1992] Kent Budge, JS Perry và AC Robinson: Tính toán khoa học hiệu suất cao sử dụng C++. Proc. Hội nghị USENIX C++. Portland, Oregon. Tháng 8 năm 1992.

Machine Translated by Google

[C,1990] Thư ký X3: Tiêu chuẩn – Ngôn ngữ C. X3J11/90-013. ISO Tiêu chuẩn ISO/IEC 9899-1990. Hiệp hội các nhà sản xuất thiết bị máy tính và kinh doanh. Washington DC.

[C,1999] ISO/IEC 9899. Tiêu chuẩn – Ngôn ngữ C. X3J11/90-013-1999.

[C,2011] ISO/IEC 9899. Tiêu chuẩn – Ngôn ngữ C. X3J11/90-013-2011.

[C++,1998] ISO/IEC JTC1/SC22/WG21: Tiêu chuẩn quốc tế – Ngôn ngữ C++. ISO/IEC 14882:1998.

[C++Math,2010] International Standard – Extensions to the C++ Library to Support Mathe chức năng đặc biệt matical. ISO/IEC 29124:2010.

[C++,2011] ISO/IEC JTC1/SC22/WG21: Tiêu chuẩn quốc tế – Ngôn ngữ C++. ISO/IEC 14882:2011.

[Campbell,1987] Roy Campbell và cộng sự: Thiết kế hệ điều hành đa bộ xử lý. Proc. Hội nghị USENIX C++. Santa Fe, New Mexico. Tháng 11 năm 1987.

[Coplian, 1995] James O. Coplien: Các mẫu mẫu lặp lại một cách kỳ lạ. Báo cáo C++. Tháng 2 năm 1995.

[Cox,2007] Russ Cox: So khớp biểu thức chính quy có thể đơn giản và nhanh chóng. Tháng 1 năm 2007. swtch.com/~rsc/regexp/regexp1.html.

[Czarnecki,2000] K. Czarnecki và U. Eisenecker: Lập trình sáng tạo: Phương pháp, Công cụ và Ứng dụng. Addison-Wesley. Đọc, Massachusetts. 2000. ISBN 0-201-30977-7.

[Dahl,1970] OJ. Dahl, B. Myrhaug, và K. Nygaard: MÔ PHỎNG Ngôn ngữ cơ sở chung. Trung tâm Điện toán Na Uy S-22. Oslo, Na Uy. 1970.

[Dahl,1972] OJ. Dahl và CAR Hoare: Xây dựng chương trình phân cấp trong lập trình có cấu trúc. Báo chí học thuật. Newyork. 1972.

[Trường khoa,2004] J. Dean và S. Ghemawat: MapReduce: Xử lý dữ liệu đơn giản hóa trên các cụm lớn. OSDI'04: Hội nghị chuyên đề lần thứ sáu về thiết kế và triển khai hệ điều hành. 2004.

[Dechev, 2010] D. Dechev, P. Pirkelbauer và B. Stroustrup: Hiểu và ngăn chặn hiệu quả vấn đề ABA trong các thiết kế không có khóa dựa trên bộ mô tả. Hội nghị chuyên đề ISORC 2010 của Hiệp hội máy tính IEEE lần thứ 13. tháng 5 năm 2010.

[DosRice,2006] Gabriel Dos Reis và Bjarne Stroustrup: Chỉ định các khái niệm C++. POPL06. Tháng 1 năm 2006.

[DosRice,2010] Gabriel Dos Reis và Bjarne Stroustrup: Biểu thức hằng chung cho ngôn ngữ lập trình hệ thống. SAC-2010. Hội nghị chuyên đề ACM lần thứ 25 về điện toán ứng dụng. Tháng 3 năm 2010.

[DosRice,2011] Gabriel Dos Reis và Bjarne Stroustrup: Một đại diện nguyên tắc, đầy đủ và hiệu quả của C++. Tạp chí Toán học trong Khoa học Máy tính. tập 5, Số 3. 2011.

[Ellis,1989] Margaret A. Ellis và Bjarne Stroustrup: Cẩm nang tham khảo C++ được chú thích. Addison-Wesley. Reading, Mass. 1990. ISBN 0-201-51459-1.

[Freeman,1992] Len Freeman và Chris Phillips: Thuật toán số song song. Hội trường Prentice. Vách đá Englewood, New Jersey. 1992. ISBN 0-13-651597-5.

[Friedl, 1997]: Jeffrey EF Friedl: Làm chủ các biểu thức chính quy. Truyền thông O'Reilly. Sevastopol, California. 1997. ISBN 978-1565922570.

Machine Translated by Google

[Gamma,1995] Erich Gamma và cộng sự: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. Đọc, Massachusetts. 1994. ISBN 0-201-63361-2.

[Gregor,2006] Douglas Gregor và cộng sự: Các khái niệm: Hỗ trợ ngôn ngữ cho việc kết hợp chư ơ ng trình chung trong C++. OOPSLA'06.

[Hennessy,2011] John L. Hennessy và David A. Patterson: Kiến trúc máy tính, Phiên bản thứ năm: Phư ơ ng pháp tiếp cận định lư ợng. Morgan Kaufmann. Sanfrancisco, California. 2011. ISBN 978-0123838728.

[Ichbiah,1979] Jean D. Ichbiah và cộng sự: Cơ sở lý luận cho việc thiết kế ngôn ngữ ngữ pháp chuyên nghiệp của ADA. Thông báo SIGPLAN. tập 14, Số 6. Tháng 6 năm 1979.

[Kamath,1993] Yogeesh H. Kamath, Ruth E. Smilan và Jean G. Smith: Gặt hái lợi ích với Công nghệ hư ớng đối tư ợng. Tạp chí Kỹ thuật AT&T. tập 72, số 5. Tháng 9/10 năm 1993.

[Kernighan,1978] Brian W. Kernighan và Dennis M. Ritchie: Ngôn ngữ lập trình C. Hội trư ờng Prentice. Vách đá Englewood, New Jersey. 1978.

[Kernighan,1988] Brian W. Kernighan và Dennis M. Ritchie: Ngôn ngữ lập trình C, Ấn bản thứ hai. Prentice-Hall. Vách đá Englewood, New Jersey. 1988. ISBN 0-13-110362-8.

[Knuth,1968] Donald E. Knuth: Nghệ thuật lập trình máy tính. Addison-Wesley. Đọc, Massachusetts. 1968.

[Koenig,1989] Andrew Koenig và Bjarne Stroustrup: C++: Càng gần C càng tốt – như ng không gần hơ n. Báo cáo C++. tập 1, số 7. Tháng 7 năm 1989.

[Koenig,1990] AR Koenig và B. Stroustrup: Xử lý ngoại lệ cho C++ (đã sửa đổi). Hội nghị Proc USENIX C++. Tháng 4 năm 1990.

[Kolecki, 2002] Joseph C. Kolecki: Giới thiệu về Tenxơ cho Sinh viên Vật lý và Kỹ thuật. NASA/TM-2002-211716.

[Langer,2000] Angelika Langer và Klaus Kreft: C++ IOStreams và Locales tiêu chuẩn: Hư ớng dẫn và tham khảo dành cho lập trình viên nâng cao. Addison-Wesley. 2000. ISBN 978-0201183955.

[McKenney] Paul E. McKenney: Lập trình song song có khó không, và nếu vậy, bạn có thể làm gì với nó? hạt nhân.org. Corvallis, Oregon. 2012. http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html.

[Maddock,2009] John Maddock: Boost.Regex. www.boost.org. 2009.

[Orwell,1949] George Orwell: 1984. Secker và Warburg. London. 1949.

[Paulson,1996] Larry C. Paulson: Học máy cho Lập trình viên đang làm việc. Nhà xuất bản Đại học Cambridge. Cambridge. 1996. ISBN 0-521-56543-X.

[Pirkelbauer,2009] P. Pirkelbauer, Y. Solodkyy, và B. Stroustrup: Thiết kế và đánh giá các phư ơ ng thức đa phư ơ ng thức mở trong C++. Khoa học lập trình máy tính. Tạp chí Elsevier. Tháng 6 năm 2009. doi:10.1016/j.scico.2009.06.002.

[Richards,1980] Martin Richards và Colin Whitby-Strevens: BCPL – Ngôn ngữ và Trình biên dịch của nó. Nhà xuất bản Đại học Cambridge. Cambridge. 1980. ISBN 0-521-21965-5.

[Gốc,1995] ROOT: Khung phân tích dữ liệu. root.cern.ch. Có vẻ thích hợp để đại diện cho một công cụ từ CERN, nơ i ra đời của World Wide Web, bởi một

Machine Translated by Google

Địa chỉ web.

[Rozier,1988] M. Rozier và cộng sự: Hệ điều hành phân tán CHORUS. Hệ thống máy tính tập 1, số 4. Mùa thu 1988.

[Bệnh, 2000] Jeremy G. Siek và Andrew Lumsdaine: Kiểm tra khái niệm: Ràng buộc tính đa hình hệ mét trong C++. Proc. Hội thảo đầu tiên về ngữ pháp C++ Template Pro. Erfurt, Đức. 2000.

[Solodkyy, 2012] Y. Solodkyy, G. Dos Reis, và B. Stroustrup: Chuyển đổi kiểu mở và hiệu quả cho C++. Proc. OOPSLA'12.

[Stepanov, 1994] Alexander Stepanov và Meng Lee: Thư viện mẫu chuẩn. Báo cáo Kỹ thuật Phòng thí nghiệm HP HPL-94-34 (R. 1). 1994.

[Stewart,1998] GW Stewart: Thuật toán ma trận, Tập I. Sự phân tách cơ bản. SIAM. Philadelphia, Pennsylvania. 1998.

[Stroustrup,1982] B. Stroustrup: Các lớp: Cơ sở kiểu dữ liệu trừu tượng cho ngôn ngữ C. Thông báo Sigplan. Tháng 1 năm 1982. Mô tả công khai đầu tiên về ''C với các lớp''

[Stroustrup,1984] B. Stroustrup: Nạp chồng toán tử trong C++. Proc. Hội nghị IFIP WG2.4 về Ngôn ngữ Triển khai Hệ thống: Kinh nghiệm & Đánh giá. Tháng 9 năm 1984.

[Stroustrup,1985] B. Stroustrup: Cơ sở I/O có thể mở rộng cho C++. Proc. Mùa hè 1985 Hội nghị USENIX.

[Stroustrup,1986] B. Stroustrup: Ngôn ngữ lập trình C++. Addison-Wesley. Đọc ing, Massachusetts. 1986. ISBN 0-201-12078-X.

[Stroustrup,1987] B. Stroustrup: Đa kế thừa cho C++. Proc. Hội nghị mùa xuân EUUG ence. Tháng 5 năm 1987.

[Stroustrup,1987b] B. Stroustrup và J. Shopiro: Tập hợp các lớp C cho ngữ pháp chuyên nghiệp phong cách đồng quy. Proc. Hội nghị USENIX C++. Santa Fe, New Mexico. Tháng 11 năm 1987.

[Stroustrup,1988] B. Stroustrup: Các kiểu được tham số hóa cho C++. Proc. Hội nghị USENIX C++ thôi nào, Denver. 1988.

[Stroustrup,1991] B. Stroustrup: Ngôn ngữ lập trình C++ (Ấn bản thứ hai). Con trai Addi-Wesley. Đọc, Massachusetts. 1991. ISBN 0-201-53992-6.

[Stroustrup,1993] B. Stroustrup: Lịch sử của C++: 1979-1991. Proc. Hội nghị Lịch sử ngôn ngữ lập pháp ACM (HOPL-2). Thông báo Sigplan của ACM. Tập 28, số 3. 1993.

[Stroustrup,1994] B. Stroustrup: Thiết kế và sự phát triển của C++. Addison-Wesley. Đọc ing, Mass. 1994. ISBN 0-201-54330-3.

[Stroustrup,1997] B. Stroustrup: Ngôn ngữ lập trình C++, Phiên bản thứ ba. Addison Wesley. Đọc, Massachusetts. 1997. ISBN 0-201-88954-4. Phiên bản bìa cứng (''Đặc biệt''). 2000. ISBN 0-201-70073-5.

[Stroustrup,2002] B. Stroustrup: C and C++: Siblings, C and C++: A Case for Compatibility, and C and C++: Case Studies in Compatibility. Tạp chí Người dùng C/C++. Tháng 7-tháng 9 năm 2002. www.stroustrup.com/papers.html.

[Stroustrup,2007] B. Stroustrup: Phát triển một ngôn ngữ trong và cho thế giới thực: C++ 1991-2006. ACM HOPL-III. tháng 6 năm 2007.

Machine Translated by Google

[Stroustrup,2008] B. Stroustrup: Lập trình – Nguyên tắc và Thực hành Sử dụng C++. Addi con trai-Wesley. 2009. ISBN 0-321-54372-6.

[Stroustrup,2010a] B. Stroustrup: Câu hỏi thư ờng gặp về C++11. www.stroustrup.com/C++11FAQ.html.

[Stroustrup,2010b] B. Stroustrup: Quyết định ''Xóa bỏ các khái niệm'' của C++0x. Nhật ký của Tiến sĩ Dobb. Tháng 7 năm 2009.

[Stroustrup,2012a] B. Stroustrup và A. Sutton: Thiết kế ý tư ởng cho STL. kỹ thuật WG21 cal Báo cáo N3351==12-0041. Tháng 1 năm 2012.

[Stroustrup,2012b] B. Stroustrup: Phát triển phần mềm cho cơ sở hạ tầng. Máy tính. Tháng Một 2012. doi:10.1109/MC.2011.353.

[Sutton, 2011] A. Sutton và B. Stroustrup: Thiết kế Thư viện Khái niệm cho C++. Proc. SLE 2011 (Hội nghị quốc tế về Kỹ thuật ngôn ngữ phần mềm). Tháng 7 năm 2011.

[Tanenbaum,2007] Andrew S. Tanenbaum: Hệ điều hành hiện đại, Phiên bản thứ ba. Hội trư ờng Prentice. Thư ợng nguồn sông Saddle, New Jersey. 2007. ISBN 0-13-600663-9.

[Tsafrir, 2009] Dan Tsafrir và cộng sự: Giảm thiểu sự phụ thuộc trong các lớp chung cho các chư ơ ng trình nhanh hơ n và nhỏ hơ n. ACM OOPSLA'09. Tháng 10 năm 2009.

[Unicode,1996] Hiệp hội Unicode: Tiêu chuẩn Unicode, Phiên bản 2.0. Addison Wesley. Đọc, Massachusetts. 1996. ISBN 0-201-48345-9.

[UNIX,1985] Hệ thống chia sẻ thời gian UNIX: Sổ tay lập trình viên. Phiên bản nghiên cứu, Phiên bản thứ mư ời. Phòng thí nghiệm AT&T Bell, Murray Hill, New Jersey. Tháng 2 năm 1985.

[Vandevoorde,2002] David Vandevoorde và Nicolai M. Josuttis: C++ Templates: The Complete Guide. Addison Wesley. 2002. ISBN 0-201-73484-2.

[Veldhuizen,1995] Todd Veldhuizen: Các mẫu biểu thức. Báo cáo C++. Tháng 6 năm 1995.

[Veldhuizen,2003] Todd L. Veldhuizen: Mẫu C++ đã hoàn thành Turing. đại học Indiana site Báo cáo Kỹ thuật Khoa học Máy tính. 2003.

[Vitter, 1985] Jefferey Scott Vitter: Lấy mẫu ngẫu nhiên với hồ chứa. Giao dịch ACM trên Phần mềm Toán học, Tập. 11, số 1. 1985.

[WG21] ISO SC22/WG21 Ủy ban tiêu chuẩn ngôn ngữ lập trình C++: Lư u trữ tài liệu. www.open-std.org/jtc1/sc22/wg21.

[Williams,2012] Anthony Williams: Hoạt động đồng thời của C++ – Đa luồng thực tế. Công ty xuất bản Manning ISBN 978-1933988771.

[Wilson,1996] Gregory V. Wilson và Paul Lu (biên tập): Lập trình song song bằng C++. Báo chí MIT. Cambridge, Mass. 1996. ISBN 0-262-73118-5.

[Gỗ,1999] Alistair Wood: Giới thiệu về Phân tích Số. Addison-Wesley. Reading, Massachusetts. 1999. ISBN 0-201-34291-X.

[Woodward,1974] PM Woodward và SG Bond: Hư ớng dẫn sử dụng Algol 68-R. của nữ hoàng Văn Phòng Phẩm Văn Phòng Phẩm. London. 1974.

Machine Translated by Google

2

Tham quan C ++: Khái niệm cơ bản

Điều đầu tiên chúng tôi làm,
hãy giết tất cả các luật sư ngôn ngữ.
– Henry VI, Phần II

• Giới thiệu • Khái niệm cơ bản
Hello, World!; Các loại, biến và số học; hằng số; Kiểm tra và Vòng lặp; Con trỏ, Mảng và Vòng lặp • Cấu trúc kiểu do người dùng định nghĩa; Các lớp học;
Enumerations • Modularity Riêng Biên dịch; Không gian
tên; Xử lý lỗi • Postscript • Lời khuyên

2.1 Giới thiệu

Mục đích của chương này và ba chương tiếp theo là cung cấp cho bạn ý tưởng về C++ là gì mà không đi sâu vào chi tiết. Chương này trình bày một cách không chính thức ký hiệu của C++, mô hình bộ nhớ và tính toán của C++, và các cơ chế cơ bản để tổ chức mã thành một chương trình. Đây là những cơ sở ngôn ngữ hỗ trợ các phong cách thường thấy nhất trong C và đôi khi được gọi là lập trình thủ tục. Chương 3 tiếp theo bằng cách trình bày các cơ chế trừu tượng hóa của C++. Chương 4 và Chương 5 đưa ra các ví dụ về cơ sở vật chất tiêu chuẩn của thư viện.

Giả định là bạn đã lập trình trước đó. Nếu không, vui lòng cân nhắc việc đọc sách giáo khoa, chẳng hạn như Lập trình: Nguyên tắc và Thực hành Sử dụng C++ [Stroustrup,2009], trước khi tiếp tục ở đây. Ngay cả khi bạn đã lập trình trước đó, ngôn ngữ bạn sử dụng hoặc các ứng dụng bạn viết có thể rất khác so với phong cách của C++ được trình bày ở đây. Nếu bạn cảm thấy khó hiểu về ''chuyến du hành chớp nhoáng'' này, hãy bỏ qua phần trình bày có hệ thống hơn bắt đầu từ Chương 6.

Machine Translated by Google

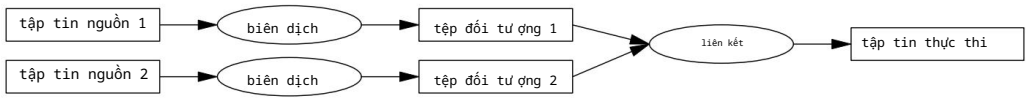
Chuyến tham quan C++ này giúp chúng ta không phải trình bày nghiêm ngặt từ dư ới lên về ngôn ngữ và các cơ sở thư viện bằng cách cho phép sử dụng một tập hợp các cơ sở phong phú ngay cả trong các chư ơ ng đầu tiên. Ví dụ, các vòng lặp không đư ợc thảo luận chi tiết cho đến Chư ơ ng 10, như ng chúng sẽ đư ợc sử dụng theo những cách rõ ràng trư ớc đó rất lâu. Tư ơ ng tự, mô tả chi tiết về các lớp, mẫu, sử dụng kho lư u trữ miễn phí và thư viện chuẩn đư ợc trải rộng trong nhiều chư ơ ng, như ng các loại thư viện chuẩn, chẳng hạn như vectơ , chuỗi, phức hợp, bản đồ, unique_ptr và ostream, đư ợc sử dụng tự do nơ i cần thiết để cải thiện các ví dụ về mã.

Tư ơ ng tự, hãy nghĩ về một chuyến tham quan ngắn ngày đến một thành phố, chẳng hạn như Copenhagen hoặc New York. Chỉ trong vài giờ, bạn sẽ đư ợc xem nhanh các điểm tham quan chính, đư ợc kể một số câu chuyện cơ bản và thư ờng đư ợc đư a ra một số gợi ý về những gì sẽ xem tiếp theo. Bạn không biết thành phố sau một tour du lịch như vậy. Bạn không hiểu tất cả những gì bạn đã thấy và nghe. Để thực sự biết một thành phố, bạn phải sống ở đó, thư ờng là trong nhiều năm. Tuy nhiên, với một chút may mắn, bạn sẽ có đư ợc một chút cái nhìn tổng quan, khái niệm về những điểm đặc biệt của thành phố và những ý tư ởng về những gì bạn có thể quan tâm. Sau chuyến tham quan, cuộc khám phá thực sự có thể bắt đầu.

Chuyến tham quan này giới thiệu C++ như một tổng thể tích hợp, chứ không phải là một chiếc bánh nhiều lớp. Do đó, nó không xác định các tính năng ngôn ngữ như hiện có trong C, một phần của C++98 hoặc mới trong C++11. Thông tin lịch sử như vậy có thể đư ợc tìm thấy trong §1.4 và Chư ơ ng 44.

2.2 Khái niệm cơ bản

C ++ là một ngôn ngữ đư ợc biên dịch. Để một chư ơ ng trình có thể chạy, văn bản nguồn của nó phải đư ợc xử lý bởi một trình xếp chồng com, tạo ra các tệp đối tư ợng, đư ợc kết hợp bởi một trình liên kết tạo ra một chư ơ ng trình thực thi. Một chư ơ ng trình C++ thư ờng bao gồm nhiều tệp mã nguồn (thư ờng đư ợc gọi đơ n giản là tệp nguồn).



Một chư ơ ng trình thực thi đư ợc tạo cho sự kết hợp phần cứng/hệ thống cụ thể; nó không thể di động, chẳng hạn như từ máy Mac sang PC Windows. Khi chúng ta nói về tính khả chuyển của các chư ơ ng trình C++, chúng tôi thư ờng muốn nói đến tính khả chuyển của mã nguồn; nghĩa là mã nguồn có thể đư ợc biên dịch thành công và chạy trên nhiều hệ thống khác nhau.

Tiêu chuẩn ISO C++ định nghĩa hai loại thực thể: • Các tính năng ngôn

ngữ cốt lõi, chẳng hạn như các kiểu dựng sẵn (ví dụ: char và int) và các vòng lặp (ví dụ: các câu lệnh for và câu lệnh while) • Các thành phần thư viện chuẩn, chẳng

hạn như vùng chứa (ví dụ: vectơ và bản đồ) và thao tác I/O (ví dụ: << và getline())

Các thành phần thư viện chuẩn là mã C++ hoàn toàn bình thư ờng đư ợc cung cấp bởi mọi bản cài đặt C++. Nghĩa là, thư viện chuẩn C++ có thể đư ợc triển khai trong chính C++ (và với việc sử dụng rất ít mã máy cho những thứ như chuyển ngữ cảnh luồng). Điều này ngụ ý rằng C++ đủ biểu đạt và hiệu quả cho các nhiệm vụ lập trình hệ thống đòi hỏi khắt khe nhất.

C ++ là một ngôn ngữ đư ợc gõ tĩnh. Nghĩa là, loại của mọi thực thể (ví dụ: đối tư ợng, giá trị, tên và biểu thức) phải đư ợc trình biên dịch biết tại thời điểm sử dụng. Loại đối tư ợng xác định tập hợp các hoạt động áp dụng cho nó.

Machine Translated by Google

2.2.1 Xin chào, Thế giới!

Chư ơ ng trình C++ tối thiểu là

```
int main() { }          // tối thiểu    chư ơ ng trình C++
```

Điều này xác định một chức năng đư ợc gọi là chính, không có đối số và không làm gì (§15.4).

Dấu ngoặc nhọn, { }, nhóm nhanh trong C++. Ở đây, chúng chỉ ra điểm bắt đầu và kết thúc của thân hàm. Dấu gạch chéo kép, //, bắt đầu một chú thích kéo dài đến cuối dòng. Một bình luận là dành cho ngư ời đọc con ngư ời; trình biên dịch bỏ qua nhận xét.

Mọi chư ơ ng trình C++ phải có chính xác một hàm toàn cục tên là main(). Chư ơ ng trình bắt đầu bằng cách thực hiện chức năng đó. Giá trị int đư ợc trả về bởi hàm main(), nếu có, là giá trị trả về của chư ơ ng trình cho ''hệ thống.'' Nếu không có giá trị nào đư ợc trả về, hệ thống sẽ nhận đư ợc một giá trị cho biết hoàn thành thành công. Giá trị khác không từ hàm main() biểu thị lỗi. Không phải mọi hệ điều hành và môi trư ờng thực thi đều sử dụng giá trị trả về đó: Các môi trư ờng dựa trên Linux/Unix thư ờng sử dụng, như ng các môi trư ờng dựa trên Win dows hiếm khi sử dụng.

Thông thư ờng, một chư ơ ng trình tạo ra một số đầu ra. Đây là chư ơ ng trình viết Hello, World!:

```
#include <iostream>

int main()
{
      std::cout << "Xin chào, Thế giới!\n";
}
```

Dòng #include <iostream> hư ớng dẫn trình biên dịch bao gồm các khai báo của cơ sở I/O luồng tiêu chuẩn như đư ợc tìm thấy trong iostream. Không có những khai báo này, biểu thức

```
std::cout << "Xin chào, Thế giới!\n"
```

sẽ không có ý nghĩa gì. Toán tử << (''put to'') viết đối số thứ hai của nó lên đối số thứ nhất. Trong trư ờng hợp này, chuỗi ký tự "Xin chào, Thế giới!\n" đư ợc ghi vào luồng đầu ra tiêu chuẩn std::cout. Chuỗi ký tự là một chuỗi các ký tự đư ợc bao quanh bởi dấu ngoặc kép. Trong một chuỗi ký tự, ký tự gạch chéo ngư ợc \ theo sau bởi một ký tự khác biểu thị một ''ký tự đặc biệt''. Trong trư ờng hợp này, \n là ký tự xuống dòng, do đó các ký tự đư ợc viết là Xin chào, Thế giới! theo sau là một dòng mới.

std :: chỉ định rằng tên cout sẽ đư ợc tìm thấy trong không gian tên thư viện chuẩn (§2.4.2, Chư ơ ng 14). Tôi thư ờng bỏ qua std:: khi thảo luận về các tính năng tiêu chuẩn; §2.4.2 chỉ ra cách hiển thị tên từ một không gian tên mà không cần xác định rõ ràng.

Về cơ bản, tất cả mã thực thi đư ợc đặt trong các hàm và đư ợc gọi trực tiếp hoặc gián tiếp từ hàm main(). Ví dụ:

```
#include <iostream> sử
dụng không gian tên std;          // đặt tên    từ tiêu chuẩn hiển thị mà không có tiêu chuẩn:: (§2.4.2)

hình vuông đôi (đôi x) {          // vuông a    số dấu phẩy động chính xác gấp đôi hình

      trả về x  x;
}
```

Machine Translated by Google


```
void print_square(double x) {

      cout << "bình phư ơ ng của "            " là " << vuông(x) << "\n"; <<x<<
}

int main()
{
      print_square(1.234);          // in ra: của 1,234 là 1,52276 hình vuông
}
```

Một khoảng trống ''kiểu trả về'' chỉ ra rằng một hàm không trả về một giá trị.

2.2.2 Kiểu, Biến và Số học Mỗi tên và mỗi biểu thức đều

có một kiểu xác định các thao tác có thể đư ợc thực hiện trên nó. Ví dụ, tuyên bố

```
int inch;
```

xác định rằng inch là kiểu int; nghĩa là, inch là một biến số nguyên.

Một khai báo là một câu lệnh giới thiệu một tên vào chư ơ ng trình. Nó chỉ định một loại cho thực thể đư ợc đặt tên: • Một loại

xác định một tập hợp các giá trị có thể và một tập hợp các hoạt động (đối với một đối

tư ợng). • Một đối tư ợng là một số bộ nhớ chứa một giá trị của một số

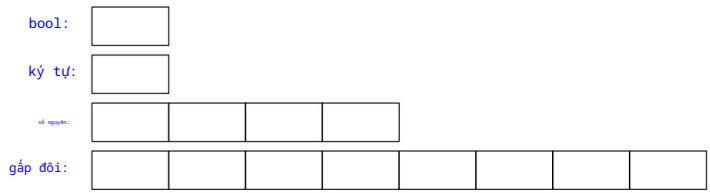
loại. • Một giá trị là một tập hợp các bit diễn giải theo một

loại. • Một biến là một đối tư ợng đư ợc đặt tên.

C++ cung cấp nhiều loại cơ bản. Ví dụ:

```
bool        // Boolean, ký tự giá trị có    là  đúng và sai
char          thể có, ví dụ: số nguyên 'a', 'z' và '9',
int         // ví dụ: 1, 42 và số dấu phẩy động có độ
gấp đôi //    chính xác kép, ví dụ: 3.14 và 299793.0
```

Mỗi loại cơ bản tư ơ ng ứng trực tiếp với các thiết bị phần cứng và có kích thư ớc cố định để xác định phạm vi giá trị có thể đư ợc lư u trữ trong đó:



Biến char có kích thư ớc tự nhiên để chứa một ký tự trên một máy nhất định (thư ờng là byte 8 bit) và kích thư ớc của các loại khác đư ợc trích dẫn theo bội số của kích thư ớc của một ký tự . Kích thư ớc của một loại đư ợc xác định theo triển khai (nghĩa là nó có thể khác nhau giữa các máy khác nhau) và có thể đư ợc lấy bởi toán tử sizeof ; ví dụ: sizeof(char) bằng 1 và sizeof(int) thư ờng bằng 4.

Các toán tử số học có thể đư ợc sử dụng cho các kết hợp thích hợp của các loại này:

Machine Translated by Google


```
x+y          // thêm
+x             unar và cộng
x  y         // trừ
  x            đơn trừ
x  yx/       // nguyên y
yx%y           nhân
             // chia phần dư (mô đun) cho số nguyên
```

Các toán tử so sánh cũng vậy:

```
x==yx!       // bằng
=y           // không bằng
x<y          // nhỏ hơn
x>y          // lớn hơn
x<=y         // ít hơn hoặc bằng
x>=y         // lớn hơn hoặc bình đẳng
```

Trong các phép gán và phép tính số học, C++ thực hiện tất cả các chuyển đổi có ý nghĩa (§10.5.3) giữa các kiểu cơ bản để chúng có thể được trộn lẫn một cách tự do:

```
void some_function() // {          chức năng không trả về một giá trị

      gấp đôi d = 2,2;        // khởi tạo số dấu phẩy động khởi
      int tôi =                 tạo số nguyên d
      7; d =                  // gán thành tổng
      d+i; i=d  i;            // gán sản phẩm cho i (cắt đôi d*i thành int) một
}
```

Lưu ý rằng = là toán tử gán và == kiểm tra đẳng thức.

C++ cung cấp nhiều ký hiệu khác nhau để thể hiện khởi tạo, chẳng hạn như dấu = được sử dụng ở trên và dấu dạng phổ quát dựa trên danh sách trình khởi tạo được phân tách bằng dấu ngoặc nhọn:

```
gấp đôi d1 = 2,3; đôi
d2 {2,3};

phức tạp<gấp đôi> z = 1;                    //  số phức với số vô hướng dấu chấm động chính xác kép a
phức<đôi> z2 {d1,d2}; phức hợp <đôi>
z3 = {1,2};                                 // các = là tùy chọn với {  ···}

vectơ <int> v {1,2,3,4,5,6};                // Một véc tơ số nguyên
```

Biểu mẫu = là truyền thống và có từ C, nhưng nếu nghi ngờ, hãy sử dụng biểu mẫu danh sách {} chung (§6.3.5.2). Nếu không có gì khác, nó sẽ cứu bạn khỏi các chuyển đổi bị mất thông tin (thu hẹp chuyển đổi; §10.5):

```
int i1 = 7,2;          // i1 trở thành 7
int i2 {7.2};            lỗi : chuyển đổi dấu phẩy động sang số nguyên
int i3 = {7.2};        // lỗi : dấu phẩy động để chuyển đổi số nguyên (các = là dư thừa)
```

Không thể bỏ khởi tạo một hằng số (§2.2.3) và chỉ nên để một biến ở trạng thái chưa khởi tạo trong những trường hợp cực kỳ hiếm. Đừng giới thiệu một cái tên cho đến khi bạn có một giá trị phù hợp cho nó. Các loại do người dùng xác định (chẳng hạn như chuỗi, vectơ , Ma trận, Motor_controller và Orc_warrior) có thể được xác định để được khởi tạo ngầm (§3.2.1.1).

Machine Translated by Google

Khi xác định một biến, bạn thực sự không cần phải nêu rõ loại của nó khi nó có thể đư ợc suy ra từ trình khởi tạo:

```
tự động b = đúng;          // Một bool
tự động ch = 'x';              ký
tự động tôi =              // Một int
123; tự động d =               double a
1,2; tự động z = sqrt(y);  // có kiểu là anything t(y) retur sqr ns
```

Với ô tô, chúng tôi sử dụng cú pháp = vì không có chuyển đổi loại liên quan có thể gây ra vấn đề (§6.3.6.2).

Chúng tôi sử dụng tự động khi chúng tôi không có lý do cụ thể để đề cập đến loại một cách rõ ràng. ''Các lý do cụ thể'' bao gồm: •

Định nghĩa ở phạm vi rộng mà chúng tôi muốn ngư ời đọc có thể nhìn thấy loại rõ ràng mã của chúng tôi.

• Chúng tôi muốn rõ ràng về phạm vi hoặc độ chính xác của một biến (ví dụ: double thay vì float).

Sử dụng tự động, chúng tôi tránh đư ợc sự dư thừa và viết tên loại dài. Điều này đặc biệt quan trọng trong lập trình chung khi ngư ời lập trình khó có thể biết đư ợc loại chính xác của đối tư ợng và tên loại có thể khá dài (§4.5.1).

Ngoài các toán tử số học và logic thông thư ờng (§10.3), C++ cung cấp nhiều hoạt động cific để sửa đổi một biến:

```
x+=y +        // x = tăng
+x            // x+y: x          = x+1
x  =y         // x = xy
  x           // giảm: x         = x-1
x  =yx/       // chia tỷ    x =  x*y
              // lệ: chia   x =  x/y
=yx%=y //       x = tỷ lệ: x%y
```

Các toán tử này ngắn gọn, thuận tiện và đư ợc sử dụng rất thư ờng xuyên.

2.2.3 Hằng số

C++ hỗ trợ hai khái niệm về tính bất biến (§7.5):

- const: nghĩa đại khái là ''Tôi hứa sẽ không thay đổi giá trị này'' (§7.5). Điều này đư ợc sử dụng chủ yếu để xác định các giao diện, do đó dữ liệu có thể đư ợc chuyển đến các chức năng mà không sợ nó bị sửa đổi. Trình biên dịch thực thi lời hứa của const.
- constexpr: nghĩa là ''đư ợc đánh giá tại thời điểm biên dịch'' (§10.4). Điều này đư ợc sử dụng chủ yếu để chỉ định các hằng số, để cho phép đặt dữ liệu vào bộ nhớ mà nó không có khả năng bị hỏng và để thực hiện.

Ví dụ:

```
const int dmv = 17; int                                   dmv là hằng số đư ợc đặt tên
var = 17;                                              // Ad tăng 1a không phải là hằng số
constexpr double max1 = 1,4  square(dmv); constexpr double   OK nếu bình phư ơ ng (17) là một biểu thức không đổi không phải
max2 = 1,4  square(var); const double max3 = 1.4  square(var);  // lỗi: biến là một biểu thức hằng số
                                                       // OK, đư ợc đánh giá tại thời điểm có thể chạy
```

Machine Translated by Google


```
tổng gấp đôi (véc tơ const <kép> &); vectơ <kép> v
{1.2, 3.4, 4.5}; const double s1 = sum(v);
constexpr double s2 = sum(v);
```

// Tổng sẽ không sửa đổi đối số của nó (§2.2.5) không
OK phải là hằng số và
// đánh giá khi chạy thời gian
// sum(v) không phải lỗi biểu thức hằng số :

Để một hàm có thể sử dụng được trong một biểu thức hằng số, nghĩa là trong một biểu thức sẽ được trình biên dịch đánh giá, nó phải được xác định constexpr. Ví dụ:

```
constexpr double square(double x) { return x  x; }
```

Để trở thành constexpr, một hàm phải khá đơn giản: chỉ cần một câu lệnh trả về tính toán một giá trị. Hàm constexpr có thể được sử dụng cho các đối số không cố định, nhưng khi điều đó được thực hiện thì kết quả không phải là một biểu thức hằng. Chúng ta cho phép một hàm constexpr được gọi với các đối số biểu thức không phải là biểu thức hằng trong ngữ cảnh không yêu cầu biểu thức hằng, do đó chúng ta không cần phải định nghĩa về cơ bản cùng một hàm hai lần: một lần cho biểu thức hằng và một lần cho các biến .

Ở một số nơi, các biểu thức hằng được yêu cầu bởi các quy tắc ngôn ngữ (ví dụ: giới hạn mảng (§2.2.5, §7.3), nhãn chữ hoa chữ thường (§2.2.4, §9.4.2), một số đối số mẫu (§25.2) và hằng được khai báo bằng cách sử dụng constexpr). Trong các trường hợp khác, đánh giá thời gian biên dịch là quan trọng đối với hiệu suất. Độc lập với các vấn đề về hiệu suất, khái niệm về tính bất biến (của một đối tượng có trạng thái không thể thay đổi) là một mối quan tâm thiết kế quan trọng (§10.4).

2.2.4 Kiểm tra và Vòng lặp

C++ cung cấp một tập hợp các câu lệnh thông thường để thể hiện lựa chọn và vòng lặp. Ví dụ, đây là một hàm đơn giản nhắc người dùng và trả về một Boolean cho biết phản hồi:

```
bool chấp nhận() {

        cout << "Bạn muốn tiếp tục (y hay n)?\n";                    // Viết câu hỏi

        câu trả lời char = 0;
        cin >> trả lời;                                              // đọc câu trả lời

        nếu (câu trả lời == 'y') trả về đúng;
        trả về sai;
}
```

Để khớp với << toán tử đầu ra (''put to''), toán tử >> (''get from'') được sử dụng cho đầu vào; cin là luồng đầu vào tiêu chuẩn. Loại toán hạng bên phải của >> xác định đầu vào nào được chấp nhận và toán hạng bên phải của nó là mục tiêu của thao tác nhập. Ký tự \n ở cuối chuỗi đầu ra đại diện cho một dòng mới (§2.2.1).

Ví dụ này có thể được cải thiện bằng cách tính đến câu trả lời n (cho ''không''):

```
bool accept2() {

        cout << "Bạn muốn tiếp tục (y hay n)?\n";                    // Viết câu hỏi

        câu trả lời char = 0;
        cin >> trả lời;                                              // đọc câu trả lời
```

Machine Translated by Google


```
        switch (câu trả lời)
        { case
              'y': trả về
        true;
              trư ờng hợp
        'n': trả
              về sai; default: cout << "I'll got it
              for a no.\n"; trả về sai;
        }
    }
```

Câu lệnh switch kiểm tra một giá trị dựa trên một tập hợp các hằng số. Các hằng số trư ờng hợp phải khác biệt và nếu giá trị đư ợc kiểm tra không khớp với bất kỳ trư ờng hợp nào trong số chúng, thì giá trị mặc định sẽ đư ợc chọn. Nếu không có giá trị mặc định nào đư ợc cung cấp, thì sẽ không có hành động nào đư ợc thực hiện nếu giá trị không khớp với bất kỳ hằng số nào.

Rất ít chư ơng trình đư ợc viết mà không có vòng lặp. Ví dụ: chúng tôi có thể muốn cung cấp cho ngư ời dùng một vài lần thử để tạo đầu vào có thể chấp nhận đư ợc:

```
    bool accept3() {

        int thử = 1;
        while (tries<4)
              { cout << "Bạn muốn tiếp tục (y hay n)?\n"; câu trả      // Viết câu hỏi
              lời char = 0; cin
              >> trả lời;                                              // đọc câu trả lời

              switch (câu trả lời)
              { case 'y':
                    trả về true;
              trư ờng hợp 'n':
                    trả về sai;
              mặc định:
                    cout << "Xin lỗi, tôi không hiểu.\n"; ++ cố
                    gắng; //      tăng
              }

        } cout << "Tôi sẽ chấp nhận điều đó.
        \n"; trả về sai;
    }
```

Câu lệnh while thực thi cho đến khi điều kiện của nó trở thành sai.

2.2.5 Con trỏ, Mảng và Vòng lặp

Một mảng các phần tử kiểu char có thể đư ợc khai báo như sau:

```
    ký tự v[6];            //       mảng 6 ký tự
```

Tư ơng tự, một con trỏ có thể đư ợc khai báo như sau:

```
    char   p;              // con trỏ tới ký tự
```

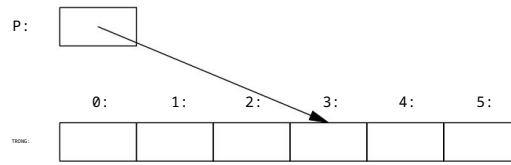
Trong các khai báo, [] có nghĩa là ''mảng'' và có nghĩa là ''con trỏ tới.'' Tất cả các mảng đều có giá trị thấp hơ n là 0

Machine Translated by Google

bị ràng buộc, vì vậy v có sáu phần tử, v[0] đến v[5]. Kích thư ớc của một mảng phải là một biểu thức không đổi (§2.2.3). Một biến con trỏ có thể chứa địa chỉ của một đối tư ợng thuộc loại thích hợp:

```
char   p = &v[3]; ký          //   trỏ đến phần tử thứ tư  của v p *p
tự x =   p;                   // là đối tư ợng trỏ đến p
```

Trong một biểu thức, tiền tố đơ n hạng có nghĩa là ''nội dung của'' và tiền tố đơ n hạng & có nghĩa là ''địa chỉ của'' Chúng ta có thể biểu diễn kết quả của định nghĩa đư ợc khởi tạo đó bằng đồ thị:



Cân nhắc sao chép mư ời phần tử từ mảng này sang mảng khác:

```
khoảng trống copy_fct()
{
        int v1[10] = {0,1,2,3,4,5,6,7,8,9}; int
        v2[10];                          // để trở thành một  sao v1  của bản

        for (auto i=0; i!=10; ++i) //         chép  yếu tố sao
                v2[i]=v1[i];
        // ...
}
```

Câu lệnh for này có thể đư ợc đọc là '' set i to zero; trong khi tôi không phải là 10, hãy sao chép phần tử thứ i và tăng i.'' Khi đư ợc áp dụng cho một biến số nguyên, toán tử tăng, ++, chỉ cần thêm 1. C++ cũng cung cấp một câu lệnh for đơ n giản hơ n, đư ợc gọi là câu lệnh phạm vi cho , đối với các vòng lặp đi qua một dãy theo cách đơ n giản nhất:

```
khoảng trống in()
{
        int v[] = {0,1,2,3,4,5,6,7,8,9};

        for (auto x : v)                 // cho mỗi trong v x
                cout << x << '\n';

        cho (tự động x : {10,21,32,43,54,65})
                cout << x << '\n';
        // ...
}
```

Phạm vi cho câu lệnh đầu tiên có thể đư ợc đọc là ''đối với mọi phần tử của v, từ phần tử đầu tiên đến phần tử cuối cùng, hãy đặt một bản sao vào x và in nó.'' Lư u ý rằng chúng ta không cần phải chỉ định một mảng bị ràng buộc khi chúng ta khởi tạo nó với một danh sách. Phạm vi cho câu lệnh có thể đư ợc sử dụng cho bất kỳ chuỗi phần tử nào (§3.4.1).

Nếu chúng ta không muốn sao chép các giá trị từ v vào biến x, mà chỉ muốn x tham chiếu đến một phần tử, chúng ta có thể viết:

Machine Translated by Google


```
gia tăng khoảng trống ()
{
        int v[] = {0,1,2,3,4,5,6,7,8,9};

        cho (tự động& x : v)
                ++x;
        // ...
}
```

Trong một khai báo, hậu tố đơ n nguyên & có nghĩa là ''tham chiếu đến.'' Tham chiếu tư ơ ng tự như con trỏ, ngoại trừ việc bạn không cần sử dụng tiền tố để truy cập giá trị đư ợc tham chiếu tới. Ngoài ra, một tham chiếu không thể đư ợc thực hiện để chỉ một đối tư ợng khác sau khi khởi tạo. Khi đư ợc sử dụng trong phần khai báo, các toán tử (chẳng hạn như &, và []) đư ợc gọi là toán tử khai báo:

```
T a[n];      // T[n]: cửa n mảng Ts (§7.3) T*:
T   p;       // trỏ tới T (§7.2)
T&r;         // T&: tham chiếu đến đối số T
Tf(A);       // T(A): chức năng lấy MỘT(§7.7) của loại A trả về kết quả loại T (§2.2.1) a
```

Chúng tôi cố gắng đảm bảo rằng một con trỏ luôn trỏ đến một đối tư ợng, để việc hủy bỏ hội nghị là hợp lệ. Khi chúng ta không có đối tư ợng để trỏ tới hoặc nếu chúng ta cần biểu diễn khái niệm ''không có đối tư ợng nào'' (ví dụ: đối với phần cuối của danh sách), chúng ta sẽ cung cấp cho con trỏ giá trị nullptr (' ' the con trỏ rỗng''). Chỉ có một nullptr đư ợc chia sẻ bởi tất cả các loại con trỏ:

```
double   pd = nullptr;
Liên kết<Bản ghi>   lst = nullptr; // int con trỏ đến một Liên kết đến một Ghi
x = nullptr; //                     lỗi : nullptr là một con trỏ không phải là số nguyên
```

Sẽ là khôn ngoan khi kiểm tra xem một đối số con trỏ đư ợc cho là trỏ đến một cái gì đó, thực sự trỏ đến một cái gì đó:

```
int đếm_x(char   p, char x) // //
        đếm số lần xuất hiện của x trong p[] của mảng char zero-ter minated
        a đư ợc giả định là trỏ tới p (hoặc không có gì)
{
        nếu (p==nullptr) trả về 0; số
        int = 0; for (;
         p!=0; ++p) if (  p==x)
              ++count; đếm
                     trở lại;

}
```

Lư u ý cách chúng ta có thể di chuyển một con trỏ để trỏ đến phần tử tiếp theo của một mảng bằng cách sử dụng ++ và chúng ta có thể bỏ phần khởi tạo trong câu lệnh for nếu chúng ta không cần.

Định nghĩa của count_x() giả định rằng char là một chuỗi kiểu C, tức là con trỏ trỏ đến một mảng char không kết thúc .

Trong mã cũ hơ n, 0 hoặc NULL thư ờng đư ợc sử dụng thay vì nullptr (§7.2.2). Tuy nhiên, sử dụng nullptr loại bỏ khả năng nhầm lẫn giữa số nguyên (chẳng hạn như 0 hoặc NULL) và con trỏ (chẳng hạn như nullptr).

Machine Translated by Google

2.3 Các loại do ngư ời dùng định nghĩa

Chúng tôi gọi các loại có thể đư ợc xây dựng từ các loại cơ bản (§2.2.2), công cụ sửa đổi const (§2.2.3) và các loại tích hợp sẵn của toán tử khai báo (§2.2.5) . Tập hợp các loại và hoạt động tích hợp sẵn của C ++ rất phong phú, như ng ở mức độ thấp có chủ ý. Chúng phản ánh trực tiếp và hiệu quả khả năng của phần cứng máy tính thông thư ờng. Tuy nhiên, chúng không cung cấp cho lập trình viên các phư ơ ng tiện cấp cao để thuận tiện viết các ứng dụng nâng cao. Thay vào đó, C ++ tăng cư ờng các loại và hoạt động tích hợp sẵn bằng một bộ cơ chế trừu tư ợng tinh vi mà từ đó các lập trình viên có thể xây dựng các cơ sở cấp cao như vậy. Các cơ chế trừu tư ợng hóa của C++ chủ yếu đư ợc thiết kế để cho phép các lập trình viên thiết kế và triển khai các kiểu của riêng họ, với các biểu diễn và hoạt động phù hợp, và để các lập trình viên sử dụng các kiểu đó một cách đơ n giản và trang nhã. Các kiểu đư ợc xây dựng từ các kiểu có sẵn sử dụng cơ chế trừu tư ợng hóa của C++ đư ợc gọi là các kiểu do ngư ời dùng định nghĩa. Chúng đư ợc gọi là các lớp và kiểu liệt kê. Phần lớn cuốn sách này dành cho việc thiết kế, triển khai và sử dụng các kiểu do ngư ời dùng định nghĩa. Phần còn lại của chư ơ ng này trình bày những cơ sở đơ n giản và cơ bản nhất cho điều đó. Chư ơ ng 3 mô tả đầy đủ hơ n về các cơ chế trừu tư ợng hóa và các kiểu lập trình mà chúng hỗ trợ.

Chư ơ ng 4 và Chư ơ ng 5 trình bày tổng quan về thư viện chuẩn và vì thư viện chuẩn chủ yếu bao gồm các loại do ngư ời dùng định nghĩa nên chúng cung cấp các ví dụ về những gì có thể đư ợc xây dựng bằng cách sử dụng các phư ơ ng tiện ngôn ngữ và kỹ thuật lập trình đư ợc trình bày trong Chư ơ ng 2 và Chư ơ ng 3.

2.3.1 Kết cấu

Bư ớc đầu tiên trong việc xây dựng một kiểu dữ liệu mới thư ờng là tổ chức các phần tử mà nó cần thành một cấu trúc dữ liệu, một cấu trúc:

```
cấu trúc vectơ
      { int sz; // double  số phần tử con trỏ
      phần tử; //          tới phần tử
};
```

Phiên bản đầu tiên này của Vector bao gồm một int và double .

Một biến kiểu Vector có thể đư ợc định nghĩa như sau:

```
Vectơ v;
```

Tuy nhiên, bản thân nó không đư ợc sử dụng nhiều vì con trỏ elem của v không trỏ đến bất cứ thứ gì. Để trở nên hữu ích, chúng ta phải cung cấp cho v một số phần tử để trỏ tới. Ví dụ, chúng ta có thể xây dựng một Vector như thế này:

```
void vector_init(Vector& v, int s) {

      v.elem = new double[s]; //    phân bổ một    mảng  of s nhân đôi
      v.sz = s;
}
```

Nghĩa là, phần tử elem của v nhận đư ợc một con trỏ do toán tử new tạo ra và phần tử size của v nhận đư ợc số phần tử. & trong Vector& chỉ ra rằng chúng ta chuyển v bằng tham chiếu không phải const (§2.2.5, §7.7); theo cách đó, vector_init() có thể sửa đổi vectơ đư ợc truyền cho nó.

Toán tử mới phân bổ bộ nhớ từ một khu vực đư ợc gọi là lư u trữ miễn phí (còn đư ợc gọi là động bộ nhớ và đống; §11.2).

Machine Translated by Google

Một cách sử dụng đơ n giản của Vector trông như thế này:

```
gấp đôi read_and_sum(int s) //
        đọc số nguyên từ cin và trả về giả định là tổng s dư ơ ng; S
{
      Vectơ v;
      vector_init(v,s);                    // phân bổ s    các yếu tố cho v
      for (int i=0; i!=s; ++i)
            cin>>v.elem[i];                // đọc thành phần tử

      tổng gấp đôi = 0;
      for (int i=0; i!=s; ++i)
            sum+=v.elem[i];                // lấy tổng các phần tử
      trả lại tổng;
}
```

Còn một chặng đư ờng dài nữa trư ớc khi Vector của chúng ta thanh lịch và linh hoạt như vector thư viện tiêu chuẩn. Cụ thể, ngư ời dùng Vector phải biết mọi chi tiết về biểu diễn của Vector . Phần còn lại của chư ơ ng này và chư ơ ng tiếp theo dần dần cải thiện Vector như một ví dụ về các tính năng và kỹ thuật của ngôn ngữ. Chư ơ ng 4 trình bày vectơ thư viện chuẩn , trong đó có nhiều cải tiến tốt đẹp, và Chư ơ ng 31 trình bày vectơ hoàn chỉnh trong bối cảnh các cơ sở thư viện chuẩn khác.

Tôi sử dụng vectơ và các thành phần thư viện tiêu chuẩn khác làm ví dụ • để minh họa các tính năng ngôn ngữ và kỹ thuật thiết kế, đồng thời • để giúp bạn tìm hiểu và sử dụng các thành phần thư viện tiêu chuẩn.

Đừng phát minh lại các thành phần thư viện tiêu chuẩn, chẳng hạn như vectơ và chuỗi; sử dụng chúng.

Chúng tôi sử dụng . (dấu chấm) để truy cập các thành viên cấu trúc thông qua tên (và thông qua một tham chiếu) và > để truy cập các thành viên cấu trúc thông qua một con trỏ. Ví dụ:

```
void f(Vector v, Vector& rv, Vector  pv) {

      int i1 = v.sz;          // truy cập thông qua tên
      int i2 = rv.sz;         // truy cập thông qua tham chiếu
      int i4 = pv  >sz;       // truy cập thông qua con trỏ
}
```

2.3.2 Các lớp

Việc có dữ liệu đư ợc chỉ định riêng biệt với các thao tác trên đó có lợi thế, chẳng hạn như khả năng sử dụng dữ liệu theo các cách tùy ý. Tuy nhiên, một kết nối chặt chẽ hơ n giữa biểu diễn và các hoạt động là cần thiết để một loại do ngư ời dùng xác định có tất cả các thuộc tính như một ''kiểu thực''. Đặc biệt, chúng ta thư ờng muốn giữ cho ngư ời dùng không thể truy cập biểu diễn, vì vậy để dễ sử dụng, đảm bảo việc sử dụng dữ liệu nhất quán và cho phép chúng tôi cải thiện cách trình bày sau này. Để làm điều đó, chúng ta phải phân biệt giữa giao diện với một loại (đư ợc sử dụng bởi tất cả) và việc triển khai nó (có quyền truy cập vào dữ liệu không thể truy cập đư ợc). Cơ chế ngôn ngữ cho điều đó đư ợc gọi là một lớp. Một lớp đư ợc định nghĩa là có một tập hợp các thành viên, có thể là thành viên dữ liệu, hàm hoặc kiểu. Giao diện đư ợc xác định bởi các thành viên chung của một lớp và các thành viên riêng tư chỉ có thể truy cập đư ợc thông qua giao diện đó. Ví dụ:

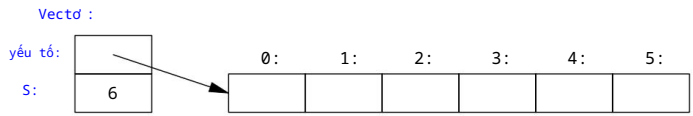
Machine Translated by Google


```
lớp Vectơ { công
khai:
      Vector(int s) :elem{new double[s]}, sz{s} { } // double& xây dựng một véc tơ
      operator[](int i) { return elem[i]; } // int size()      truy cập phần tử: đăng ký
      { return sz; } private:
double
      elem; // int sz; // con trỏ tới phần tử số
                          lư ợng phần tử
};
```

Do đó, chúng ta có thể định nghĩa một biến của loại Vector mới:

```
Vectơ v(6);      //  Vectơ có 6 phần tử a
```

Chúng ta có thể minh họa một đối tư ợng Vector bằng đồ họa:



Về cơ bản, đối tư ợng Vector là một ''tay cầm'' chứa một con trỏ tới các phần tử (elem) cộng với số phần tử (sz). Số lư ợng phần tử (6 trong ví dụ) có thể thay đổi từ đối tư ợng Vector sang đối tư ợng Vector và một đối tư ợng Vector có thể có số lư ợng phần tử khác nhau tại các thời điểm khác nhau (§3.2.1.3). Tuy nhiên, bản thân đối tư ợng Vector luôn có cùng kích thư ớc. Đây là kỹ thuật cơ bản để xử lý các lư ợng thông tin khác nhau trong C++: một bộ điều khiển có kích thư ớc cố định đề cập đến một lư ợng dữ liệu thay đổi ''ở nơ i khác'' (ví dụ: trên kho lư u trữ miễn phí đư ợc phân bổ bởi new; §11.2) . Làm thế nào để thiết kế và sử dụng các đối tư ợng như vậy là chủ đề chính của Chư ơ ng 3.

Ở đây, biểu diễn của Vector (các thành viên elem và sz) chỉ có thể truy cập thông qua giao diện đư ợc cung cấp bởi các thành viên công khai : Vector(), operator[]() và siz e(). Ví dụ read_and_sum () từ §2.3.1 đơ n giản hóa thành:

```
gấp đôi read_and_sum(int s) {

      Vectơ v(s);                          // biến vectơ của các phần tử s thành a
      for (int i=0; i!=v.siz e(); +
            +i) cin>>v[i];                 // đọc thành phần tử

      tổng gấp đôi = 0;
      for (int i=0; i!=v.siz e(); +
            +i)                            // lấy tổng          của các yếu tố
      sum+=v[i]; trả lại tổng;
}
```

Một ''hàm'' có cùng tên với lớp của nó đư ợc gọi là hàm tạo, tức là hàm đư ợc sử dụng để xây dựng các đối tư ợng cấu trúc của một lớp. Vì vậy, hàm tạo, Vector(), thay thế vector_init() từ §2.3.1. Không giống như một hàm thông thư ờng, một hàm tạo đư ợc đảm bảo đư ợc sử dụng để khởi tạo các đối tư ợng của lớp của nó. Do đó, việc định nghĩa một hàm tạo sẽ loại bỏ vấn đề về các biến chư a đư ợc khởi tạo cho một lớp.

Machine Translated by Google

Vector(int) xác định cách các đối tượng kiểu Vector được xây dựng. Cụ thể, nó nói rằng nó cần một số nguyên để làm điều đó. Số nguyên đó được dùng làm số phần tử. Hàm tạo khởi tạo các thành viên Vector bằng danh sách khởi tạo thành viên:

```
:elem{new double[s]}, sz{s}
```

Đó là, đầu tiên chúng ta khởi tạo elem với một con trỏ tới s phần tử kiểu double lấy được từ cửa hàng miễn phí. Sau đó, chúng tôi khởi tạo sz thành s.

Quyền truy cập vào các phần tử được cung cấp bởi một hàm chỉ số, được gọi là toán tử []. Nó trả về một tham chiếu đến phần tử thích hợp (kép&).

Hàm size() được cung cấp để cung cấp cho người dùng số lượng phần tử.

Rõ ràng, việc xử lý lỗi hoàn toàn không có, nhưng chúng ta sẽ quay lại vấn đề đó trong §2.4.3. Tương tự như vậy, chúng tôi đã không cung cấp cơ chế ''trả lại'' mảng nhân đôi mà người mới có được ; §3.2.1.2 chỉ ra cách sử dụng hàm hủy để thực hiện điều đó một cách tao nhã.

2.3.3 Phép liệt kê

Ngoài các lớp, C++ hỗ trợ một dạng đơn giản của kiểu do người dùng định nghĩa mà chúng ta có thể liệt kê các giá trị:

```
lớp enum Màu {đỏ, xanh lam, xanh lục}; enum class
Traffic_light { xanh, vàng, đỏ };

Màu col = Màu::đỏ;
Traffic_light light = Traffic_light::red;
```

Lưu ý rằng các điều tra viên (ví dụ: màu đỏ) nằm trong phạm vi của lớp enum của chúng, do đó chúng có thể được sử dụng nhiều lần trong các lớp enum khác nhau mà không bị nhầm lẫn. Ví dụ: Color::red là màu đỏ của Color khác với Traffic_light::red.

Phép liệt kê được sử dụng để biểu diễn các tập hợp nhỏ các giá trị số nguyên. Chúng được sử dụng để làm cho mã dễ đọc hơn và ít bị lỗi hơn so với khi không sử dụng các tên bộ đếm tương trưng (và dễ nhớ).

Lớp sau enum xác định rằng một phép liệt kê được gõ mạnh và các điều tra viên của nó nằm trong phạm vi . Là các loại riêng biệt, các lớp enum giúp ngăn ngừa việc vô tình sử dụng sai các hằng số. Đặc biệt, chúng ta không thể trộn lẫn các giá trị Traffic_light và Color :

```
Màu x = đỏ;                    // lỗi : màu đỏ nào
màu sắc và = Traffic_light::red;   lỗi : màu đỏ đó không phải một Tô màu
Màu z = Màu::đỏ;               // được rồi
```

Tương tự, chúng ta không thể trộn lẫn hoàn toàn các giá trị Màu và số nguyên:

```
int i = Màu::đỏ;                   lỗi : Màu ::đỏ không phải là số nguyên màu 2 không
Màu c = 2;                         // lỗi : là số Màu sắc
```

Nếu bạn không muốn xác định rõ ràng tên điều tra viên và muốn giá trị điều tra viên là số nguyên (không cần chuyển đổi rõ ràng), bạn có thể xóa lớp khỏi lớp enum để nhận '' enum đơn giản '' (§8.4.2).

Theo mặc định, một lớp enum chỉ có phép gán, khởi tạo và so sánh (ví dụ: == và <; §2.2.2) xác định. Tuy nhiên, kiểu liệt kê là kiểu do người dùng định nghĩa nên chúng ta có thể định nghĩa các toán tử cho nó:

Machine Translated by Google


```
Traffic_light& operator++(Traffic_light& t)
        // tăng tiền tố: ++
{
        switch (t)
        { case Traffic_light::green: case        trả về t=Traffic_light::yellow; trả về
        Traffic_light::yellow: case              t=Traffic_light::red; trả về
        Traffic_light::red: }                    t=Traffic_light::green;

}

Traffic_light next = ++light;                    // tiếp theo trở thành Traffic_light::green
```

C ++ cũng cung cấp một enum '' đơ n giản '' đư ợc gõ ít mạnh hơ n (§8.4.2).

2.4 Tính mô đun

Một chư ơ ng trình C++ bao gồm nhiều phần đư ợc phát triển riêng biệt, chẳng hạn như hàm (§2.2.1, Chư ơ ng 12), kiểu do ngư ời dùng định nghĩa (§2.3, §3.2, Chư ơ ng 16), phân cấp lớp (§3.2.4, Chư ơ ng 20), và tem nhãn (§3.4, Chư ơ ng 23). Chìa khóa để quản lý điều này là xác định rõ ràng các tư ơ ng tác giữa các bộ phận đó. Bư ớc đầu tiên và quan trọng nhất là phân biệt giữa giao diện của một phần và phần thực hiện của nó. Ở cấp độ ngôn ngữ, C++ thể hiện các giao diện bằng cách khai báo. Một khai báo chỉ rõ tất cả những gì cần thiết để sử dụng một hàm hoặc một kiểu. Ví dụ:

```
đôi sqrt(gấp đôi);                // hàm gốc nhận gấp đôi và trả về bình phư ơ ng gấp đôi a a

lớp Vectơ { công
khai:
        Vectơ (int s);
        double& toán tử[](int i); kích
        thư ớc
int();
        riêng tư : double      yếu tố trỏ đến một          của mảng đôi sz
        elem; // int sz;
};
```

Điểm mấu chốt ở đây là thân hàm, các định nghĩa hàm, nằm ''ở chỗ khác''. Đối với ví dụ này, chúng ta có thể muốn biểu diễn của Vector cũng ''ở chỗ khác'', như ng chúng ta sẽ giải quyết vấn đề đó sau (kiểu trừu tư ợng; §3.2.2). Định nghĩa của sqrt() sẽ như thế này:

```
đôi sqrt(đôi d) {                    // định nghĩa của sqrt()

        // ... thuật toán tìm thấy trong sách giáo khoa toán ... as
}
```

Đối với Vector, chúng ta cần xác định cả ba hàm thành viên:

```
Vectơ ::Vector(int                                  định nghĩa của hàm tạo khởi
        s) :elem{new double[s]}, sz{s}              // tạo thành viên

{ }
```

Machine Translated by Google


```
double& Vector::operator[](int i) {                    // định nghĩa của đăng ký

        trả lại elem[i];
}

int Vector::size e() {                                 // định nghĩa kích thư ớc()

        trả lại sz;
}
```

Chúng ta phải định nghĩa các hàm của Vector , chứ không phải hàm sqrt() vì nó là một phần của thư viện chuẩn. Tuy nhiên, điều đó không tạo ra sự khác biệt thực sự: thư viện chỉ đơ n giản là một số ''mã khác mà chúng tôi tình cờ sử dụng'' đư ợc viết bằng cùng phư ơ ng tiện ngôn ngữ mà chúng tôi sử dụng.

2.4.1 Biên dịch riêng biệt

C++ hỗ trợ khái niệm biên dịch riêng biệt trong đó mã ngư ời dùng chỉ nhìn thấy các khai báo về kiểu và hàm đư ợc sử dụng. Các định nghĩa về các loại và chức năng đó nằm trong các tệp nguồn riêng biệt và com đư ợc xếp chồng lên nhau. Điều này có thể đư ợc sử dụng để tổ chức một chư ơ ng trình thành một tập hợp các đoạn mã bán độc lập. Sự tách biệt như vậy có thể đư ợc sử dụng để giảm thiểu thời gian biên dịch và thực thi chặt chẽ việc tách biệt các phần riêng biệt về mặt logic của chư ơ ng trình (do đó giảm thiểu khả năng xảy ra lỗi). Một thư viện thư ờng là một đoạn mã đư ợc biên dịch riêng biệt (ví dụ: các hàm).

Thông thư ờng, chúng tôi đặt các khai báo chỉ định giao diện cho một mô-đun trong một tệp có tên cho biết mục đích sử dụng của nó. Ví dụ:

```
// Vectơ .h:

lớp Vectơ { công
khai:
        Vectơ (int s);
        double& toán tử[](int i); kích
        thư ớc
int();
        riêng tư : double          // yếu tố trỏ đến một của mảng sz nhân đôi
        elem; int sz;
};
```

Phần khai báo này sẽ đư ợc đặt trong một tệp Vector.h và ngư ời dùng sẽ đư a tệp đó, đư ợc gọi là tệp tiêu đề, để truy cập vào giao diện đó. Ví dụ:

```
// ngư ời dùng.cpp:

#include "Vector.h"              //     Giao diện của
#include <cmath> // sử dụng không   Vector get giao diện hàm toán học thư viện tiêu chuẩn bao gồm sqrt() y
gian tên std; //                   làm cho các thành viên std hiển thị (§2.4.2)
```

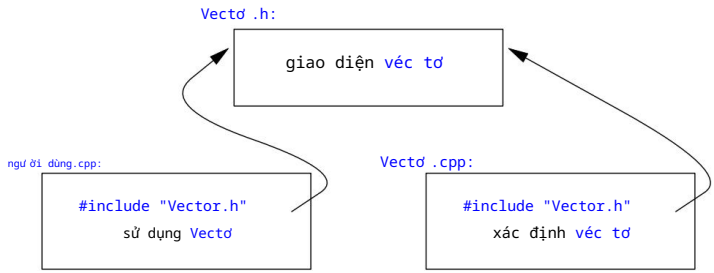
Machine Translated by Google


```
gấp đôi sqrt_sum(Vector& v) {

      tổng gấp đôi = 0;
      for (int i=0; i!=v.siz e(); +
            +i) sum+=sqrt(v[i]);                    // Tổng   căn bậc hai
      trả lại tổng;
}
```

Để giúp trình biên dịch đảm bảo tính nhất quán, tệp .cpp cung cấp việc triển khai Vector cũng sẽ bao gồm tệp .h cung cấp giao diện của nó:

```
// Vectơ .cpp:

#include "Vector.h" //   lấy giao diện

Vectơ ::Vector(int
      s) :elem{new double[s]}, sz{s}

{ }

double& Vector::operator[](int i) {

      trả lại elem[i];
}

int Vector::size e()
{
      trả lại sz;
}
```

Mã trong user.cpp và Vector.cpp chia sẻ thông tin giao diện Vector đư ợc trình bày trong Vector.h, như ng hai tệp này độc lập và có thể đư ợc biên dịch riêng. Về mặt đồ họa, các đoạn chư ơ ng trình có thể đư ợc biểu diễn như sau:



Nói một cách chính xác, sử dụng trình biên dịch riêng biệt không phải là vấn đề về ngôn ngữ; đó là một vấn đề về cách tốt nhất để tận dụng lợi thế của việc triển khai ngôn ngữ cụ thể. Tuy nhiên, nó có tầm quan trọng thực tiễn rất lớn. Cách tiếp cận tốt nhất là tối đa hóa tính mô-đun, thể hiện tính mô-đun đó một cách logic thông qua các tính năng ngôn ngữ, sau đó khai thác tính mô-đun một cách vật lý thông qua các tệp để biên dịch riêng hiệu quả (Chư ơ ng 14, Chư ơ ng 15).

Machine Translated by Google

2.4.2 Không gian tên

Ngoài hàm (§2.2.1, Chư ơ ng 12), lớp (Chư ơ ng 16) và kiểu liệt kê (§2.3.3, §8.4), C++ cung cấp không gian tên (Chư ơ ng 14) như một cơ chế để thể hiện rằng một số khai báo thuộc về nhau và rằng tên của họ không đư ợc xung đột với các tên khác. Ví dụ: tôi có thể muốn thử nghiệm với loại số phức của riêng mình (§3.2.1.1, §18.3, §40.4):

```
không gian tên My_code
      { phức hợp lớp { /* */ };
      phức sqr t(phức tạp); // int
        ...
      main();
}

int My_code::main() {

      phức z{1,2}; tự
      động z2 = sqrt(z);
      std::cout << '{' << z2.real() << ',' << z2.imag() << "}\n"; //
        ...
};

int main()
{
      trả về My_code::main();
}
```

Bằng cách đặt mã của tôi vào không gian tên My_code, tôi đảm bảo rằng tên của tôi không xung đột với tên thư viện chuẩn trong không gian tên std (§4.1.2). Biện pháp phòng ngừa là khôn ngoan, bởi vì thư viện tiêu chuẩn cung cấp hỗ trợ cho số học phức tạp (§3.2.1.1, §40.4).

Cách đơ n giản nhất để truy cập một tên trong không gian tên khác là xác định tên đó bằng tên không gian tên (ví dụ: std::cout và My_code::main). ''real main()'' đư ợc xác định trong không gian tên chung, nghĩa là không cục bộ đối với một không gian tên, lớp hoặc hàm đã xác định. Để có quyền truy cập vào các tên trong không gian tên stan dard-library, chúng ta có thể sử dụng chỉ thị sử dụng (§14.2.3):

```
sử dụng không gian tên std;
```

Không gian tên chủ yếu đư ợc sử dụng để tổ chức các thành phần chư ơ ng trình lớn hơ n, chẳng hạn như thư viện. Chúng đơ n giản hóa thành phần của một chư ơ ng trình từ các phần đư ợc phát triển riêng biệt.

2.4.3 Xử lý lỗi

Xử lý lỗi là một chủ đề lớn và phức tạp với các mối quan tâm và sự phân nhánh vư ợt xa các cơ sở ngôn ngữ sang các kỹ thuật và công cụ lập trình. Tuy nhiên, C++ cung cấp một vài tính năng để trợ giúp. Công cụ chính là chính hệ thống loại. Thay vì xây dựng các ứng dụng của chúng tôi một cách tỉ mỉ từ các kiểu dựng sẵn (ví dụ: char, int và double) và các câu lệnh (ví dụ: if, while và for), chúng tôi xây dựng nhiều kiểu phù hợp hơ n cho các ứng dụng của mình (ví dụ: chuỗi , map và regex) và các thuật toán (ví dụ: sort(), find_if() và draw_all()). Các cấu trúc cấp cao hơ n như vậy đơ n giản hóa việc lập trình của chúng ta, hạn chế khả năng mắc lỗi của chúng ta (ví dụ: bạn không thể cố gắng áp dụng duyệt cây cho hộp thoại),

Machine Translated by Google

và tăng khả năng bắt lỗi như vậy của trình biên dịch. Phần lớn các cấu trúc C++ được dành riêng cho việc thiết kế và triển khai các khái niệm trừu tượng thanh lịch và hiệu quả (ví dụ: các loại và thuật toán do người dùng định nghĩa sử dụng chúng). Một ảnh hưởng của tính mô đun và tính trừu tượng này (cụ thể là việc sử dụng các thư viện) là điểm có thể phát hiện ra lỗi thời gian chạy được tách biệt khỏi điểm có thể xử lý lỗi. Khi các chương trình phát triển và đặc biệt là khi các thư viện được sử dụng rộng rãi, các tiêu chuẩn để xử lý lỗi trở nên quan trọng.

2.4.3.1 Ngoại lệ

Xem xét lại ví dụ Vector . Phải làm gì khi chúng ta cố gắng truy cập một phần tử nằm ngoài phạm vi của vectơ từ §2.3.2?

- Người viết Vector không biết người dùng muốn làm gì trong trường hợp này (người viết Vector thường thậm chí không biết vector sẽ chạy trong chương trình nào). • Người dùng Vector không thể phát hiện vấn đề một cách nhất quán (nếu người dùng có thể, việc truy cập ngoài phạm vi sẽ không xảy ra ngay từ đầu).

Giải pháp dành cho trình triển khai Vector là phát hiện nỗ lực truy cập ngoài phạm vi và sau đó thông báo cho người dùng về điều đó. Sau đó, người dùng có thể thực hiện hành động thích hợp. Ví dụ: Vector::operator[]() có thể phát hiện một nỗ lực truy cập ngoài phạm vi và đưa ra một ngoại lệ out_of_range :

```
double& Vector::operator[](int i) {

        if (i<0 || size()<=i) throw out_of_rang e{"Vector::operator[]"}; trả lại elem[i];

}
```

Lệnh ném chuyển điều khiển tới một trình xử lý đối với các trường hợp ngoại lệ thuộc loại out_of_range trong một số chức năng được gọi trực tiếp hoặc gián tiếp là Vector::operator[]() . Để làm điều đó, việc triển khai sẽ giải phóng ngăn xếp lệnh gọi hàm khi cần để quay lại ngữ cảnh của người gọi đó (§13.5.1). Ví dụ:

```
void f(Vectơ & v) {

        // ...
        thử { // ngoại lệ ở đây là          được xử lý bởi trình xử lý được xác định bên dưới

                v[v.không có e()] = 7; // cố gắng truy cập sau khi kết thúc v

        } bắt (out_of_rang e) { // //        tiếc:   lỗi out_of_range rất
                 ... xử lý lỗi phạm vi ...

        } //...
}
```

Chúng tôi đặt mã mà chúng tôi quan tâm đến việc xử lý các ngoại lệ vào một khối thử. Việc gán cho v[v.size()] đó sẽ không thành công. Do đó, mệnh đề khai báo cung cấp trình xử lý cho out_of_range sẽ được nhập. Loại out_of_range được định nghĩa trong thư viện chuẩn và trên thực tế được sử dụng bởi một số hàm truy cập vùng chứa thư viện chuẩn.

Việc sử dụng các cơ chế xử lý ngoại lệ có thể làm cho việc xử lý lỗi đơn giản hơn, có hệ thống hơn, và dễ đọc hơn. Xem Chương 13 để biết thêm thảo luận, chi tiết và ví dụ.

Machine Translated by Google

2.4.3.2 Bất biến

Việc sử dụng các ngoại lệ để báo hiệu truy cập ngoài phạm vi là một ví dụ về một chức năng kiểm tra đối số của nó và từ chối hành động vì một giả định cơ bản, một điều kiện tiên quyết , không đúng. Nếu chúng tôi chính thức chỉ định toán tử chỉ số của Vector , chúng tôi sẽ nói điều gì đó như ''chỉ mục phải nằm trong phạm vi [0:size()) ,'' và đó thực tế là những gì chúng tôi đã thử nghiệm trong toán tử[]() của mình. Bất cứ khi nào chúng ta định nghĩa một hàm, chúng ta nên xem xét các điều kiện tiên quyết của nó là gì và nếu khả thi, hãy kiểm tra chúng (xem §12.4, §13.4).

Tuy nhiên, toán tử []() hoạt động trên các đối tư ợng thuộc loại Vector và không có ý nghĩa gì trừ khi các thành viên của Vector có các giá trị ''hợp lý''. Đặc biệt, chúng tôi đã nói ''elem chỉ vào một mảng sz double'' như ng chúng tôi chỉ nói điều đó trong một nhận xét. Một tuyên bố như vậy về những gì đư ợc giả định là đúng đối với một lớp đư ợc gọi là bất biến lớp, hay đơ n giản là một bất biến. Công việc của một hàm tạo là thiết lập bất biến cho lớp của nó (để các hàm thành viên có thể dựa vào nó) và đối với các hàm thành viên là đảm bảo rằng bất biến đư ợc giữ nguyên khi chúng thoát. Thật không may, trình tạo Vector của chúng tôi chỉ thực hiện một phần công việc của nó. Nó đã khởi tạo đúng các thành viên Vector , như ng không thể kiểm tra xem các đối số đư ợc truyền cho nó có hợp lý hay không. Coi như :

```
Vectơ v( 27);
```

Điều này có khả năng gây ra sự hỗn loạn.

Đây là một định nghĩa phù hợp hơ n:

```
Vectơ ::Vectơ (int s) {

      if (s<0) throw length_error{};
      elem = double[s] mới;
      sz = s;
}
```

Tôi sử dụng trư ờng hợp ngoại lệ thư viện tiêu chuẩn length_error để báo cáo số phần tử không dư ơ ng vì một số hoạt động của thư viện tiêu chuẩn sử dụng ngoại lệ đó để báo cáo các sự cố thuộc loại này. Nếu toán tử mới không thể tìm thấy bộ nhớ để phân bổ, nó sẽ đư a ra một lỗi std::bad_alloc. Bây giờ chúng ta có thể viết:

```
kiểm tra khoảng trống
() {
      thử {
            Vectơ v( 27);

      } bắt (std::length_error) { //
              xử lý kích thư ớc âm

      } bắt (std::bad_alloc) { //
              xử lý cạn kiệt bộ nhớ
      }
}
```

Bạn có thể định nghĩa các lớp của riêng mình sẽ đư ợc sử dụng làm ngoại lệ và yêu cầu chúng mang thông tin tùy ý từ điểm phát hiện lỗi đến điểm có thể xử lý lỗi (§13.5).

Thông thư ờng, một chức năng không có cách nào hoàn thành nhiệm vụ đư ợc giao sau khi một ngoại lệ đư ợc đư a ra. Sau đó, ''xử lý'' một ngoại lệ chỉ đơ n giản có nghĩa là thực hiện một số thao tác dọn dẹp cục bộ tối thiểu và sắp xếp lại ngoại lệ đó (§13.5.2.1).

Machine Translated by Google

Khái niệm bất biến là trung tâm của việc thiết kế các lớp và các điều kiện tiên quyết đóng vai trò tư ơ ng tự trong việc thiết kế các hàm. Bất biến •

giúp chúng ta hiểu chính xác những gì chúng ta muốn • buộc

chúng ta phải cụ thể; điều đó mang lại cho chúng tôi cơ hội tốt hơ n để nhận đư ợc mã chính xác (sau gỡ lỗi và thử nghiệm).

Khái niệm bất biến làm cơ sở cho khái niệm quản lý tài nguyên của C++ đư ợc hỗ trợ bởi hàm tạo (§2.3.2) và hàm hủy (§3.2.1.2, §5.2). Xem thêm §13.4, §16.3.1 và §17.2.

2.4.3.3 Xác nhận tĩnh

Lỗi báo cáo ngoại lệ đư ợc tìm thấy trong thời gian chạy. Nếu một lỗi có thể đư ợc tìm thấy tại thời điểm biên dịch, thì tốt hơ n là nên làm như vậy. Đó là phần lớn của hệ thống loại và phư ơ ng tiện để chỉ định các mặt giao tiếp cho các loại do ngư ời dùng xác định. Tuy nhiên, chúng ta cũng có thể thực hiện kiểm tra đơ n giản trên các ràng buộc thích hợp khác đư ợc biết tại thời điểm biên dịch và báo cáo lỗi dư ới dạng thông báo lỗi trình biên dịch. Ví dụ:

```
static_assert(4<=sizeof(int), "số nguyên quá nhỏ"); //
```

kiểm tra kích thư ớc số nguyên

Điều này sẽ ghi số nguyên quá nhỏ nếu 4<=sizeof(int) không giữ, nghĩa là, nếu một int trên hệ thống này không có ít nhất 4 byte. Chúng tôi gọi những tuyên bố về kỳ vọng như vậy là những khẳng định.

Cơ chế static_assert có thể đư ợc sử dụng cho bất kỳ thứ gì có thể đư ợc biểu thị dư ới dạng hằng số biểu thức (§2.2.3, §10.4). Ví dụ:

```
constexpr double C = 299792,458;                    // km/s

void f(tốc độ gấp đôi) {

        const double local_max = 160.0/(60  60);    // 160 km/h == 160,0/(60*60) km/s

        static_assert(speed<C,"không thể nhanh như vậy");
        static_assert(local_max<C,"không thể nhanh như vậy"); //   // lỗi : tốc độ phải là một không thay đổi
                                                                      đư ợc nói

        // ...
}
```

Nói chung, static_assert(A,S) in S dư ới dạng thông báo lỗi trình biên dịch nếu A không đúng.

Việc sử dụng static_assert quan trọng nhất xuất hiện khi chúng ta xác nhận về các loại đư ợc sử dụng làm tham số trong lập trình chung (§5.4.2, §24.3).

Đối với các xác nhận đư ợc kiểm tra thời gian chạy, hãy xem §13.4.

2.5 Tái bút

Các chủ đề trong chư ơ ng này gần tư ơ ng ứng với nội dung của Phần II (Chư ơ ng 6–15).
Đó là những phần của C++ làm nền tảng cho tất cả các kỹ thuật và phong cách lập trình đư ợc C++ hỗ trợ.
Các lập trình viên C và C++ có kinh nghiệm, xin lư u ý rằng nền tảng này không tư ơ ng ứng chặt chẽ với các tập hợp con C hoặc C++98 của C++ (nghĩa là C++11).

Machine Translated by Google

2.6 Lời khuyên

[1] Đừng hoảng sợ! Tất cả sẽ trở nên rõ ràng trong thời gian; §2.1.

[2] Bạn không cần phải biết mọi chi tiết của C++ để viết các chư ơ ng trình tốt; §1.3.1.

[3] Tập trung vào kỹ thuật lập trình, không phải tính năng ngôn ngữ; §2.1.

Machine Translated by Google

3

Tham quan C ++: Cơ chế trừu tượng

Không hoảng loạn!

- Douglas Adams

• Giới thiệu • Lớp học

Các Loại Bê Tông; Các loại trừu tượng; Hàm ảo; Hệ thống phân cấp lớp

• Sao chép và Di chuyển

Sao chép Container; Chuyển Container; Quản lý nguồn tài nguyên; Ngăn chặn hoạt động • Mẫu

Loại tham số hóa; Mẫu chức năng; Đối tượng chức năng; Mẫu đa dạng; Bí danh • Lời khuyên

3.1 Giới thiệu

Chương này nhằm cung cấp cho bạn ý tưởng về sự hỗ trợ của C++ đối với sự trừu tượng hóa và quản lý tài nguyên mà không đi sâu vào chi tiết. Nó trình bày một cách không chính thức các cách định nghĩa và sử dụng các kiểu mới (kiểu do người dùng định nghĩa). Cụ thể, nó trình bày các thuộc tính cơ bản, kỹ thuật triển khai và cơ sở ngôn ngữ được sử dụng cho các lớp cụ thể, lớp trừu tượng và phân cấp lớp. Các mẫu được giới thiệu như một cơ chế để tham số hóa các loại và thuật toán với các loại và thuật toán (khác). Các tính toán trên các loại tích hợp và do người dùng xác định được biểu diễn dưới dạng các hàm, đôi khi được khái quát hóa thành các hàm mẫu và các đối tượng hàm. Đây là những cơ sở ngôn ngữ hỗ trợ các phong cách lập trình được gọi là lập trình hướng đối tượng và lập trình chung. Hai chương tiếp theo sẽ trình bày các ví dụ về cơ sở vật chất của thư viện tiêu chuẩn và việc sử dụng chúng.

Giả định là bạn đã lập trình trước đó. Nếu không, vui lòng cân nhắc việc đọc sách giáo khoa, chẳng hạn như Lập trình: Nguyên tắc và Thực hành Sử dụng C++ [Stroustrup,2009], trước khi tiếp tục ở đây. Ngay cả khi bạn đã lập trình trước đó, ngôn ngữ bạn sử dụng hoặc các ứng dụng bạn viết có thể rất khác so với phong cách của C++ được trình bày ở đây. Nếu bạn cảm thấy khó hiểu về ''chuyến du hành chớp nhoáng'' này, hãy bỏ qua phần trình bày có hệ thống hơn bắt đầu từ Chương 6.

Machine Translated by Google

Như trong Chương 2, chuyến tham quan này trình bày C++ như một tổng thể tích hợp, chứ không phải là một chiếc bánh nhiều lớp. Do đó, nó không xác định các tính năng ngôn ngữ như hiện có trong C, một phần của C++98 hoặc mới trong C++11. Thông tin lịch sử như vậy có thể được tìm thấy trong §1.4 và Chương 44.

3.2 Lớp học

Tính năng ngôn ngữ trung tâm của C++ là lớp. Một lớp là một loại do người dùng định nghĩa được cung cấp để diễn đạt lại một khái niệm đã gửi trong mã chương trình. Bất cứ khi nào thiết kế của chúng tôi cho một chương trình có một khái niệm, ý tưởng, thực thể hữu ích, v.v., chúng tôi cố gắng biểu diễn nó dưới dạng một lớp trong chương trình để ý tưởng đó có trong mã, thay vì chỉ trong đầu chúng tôi, trong một tài liệu thiết kế , hoặc trong một số bình luận. Một chương trình được xây dựng từ một tập hợp các lớp được lựa chọn tốt sẽ dễ hiểu và hiểu đúng hơn nhiều so với một chương trình xây dựng mọi thứ trực tiếp theo các loại tích hợp sẵn. Đặc biệt, các lớp học thường là những gì thư viện cung cấp.

Về cơ bản, tất cả các phương tiện ngôn ngữ ngoài các kiểu, toán tử và câu lệnh cơ bản đều tồn tại để giúp định nghĩa các lớp tốt hơn hoặc sử dụng chúng thuận tiện hơn. Khi nói ''tốt hơn'', ý tôi là chính xác hơn, dễ bảo trì hơn, hiệu quả hơn, trang nhã hơn, dễ sử dụng hơn, dễ đọc hơn và dễ lập luận hơn. Hầu hết các kỹ thuật lập trình đều dựa trên việc thiết kế và triển khai các loại lớp cụ thể. Nhu cầu và thị hiếu của các lập trình viên rất khác nhau. Do đó, sự hỗ trợ cho các lớp học là rộng rãi. Ở đây, chúng ta sẽ chỉ xem xét hỗ trợ cơ bản cho ba loại lớp quan trọng: • Lớp cụ thể (§3.2.1) • Lớp trừu tượng (§3.2.2) • Lớp trong phân cấp lớp (§3.2.4)

Một số lượng đáng kinh ngạc của các lớp hữu ích hóa ra lại thuộc ba loại này. Thậm chí nhiều lớp hơn có thể được coi là các biến thể đơn giản của các loại này hoặc được triển khai bằng cách sử dụng kết hợp các kỹ thuật được sử dụng cho các loại này.

3.2.1 Loại bê tông

Ý tưởng cơ bản của các lớp cụ thể là chúng hành xử ''giống như các kiểu dựng sẵn.'' Ví dụ, một kiểu số phức và một số nguyên có độ chính xác vô hạn giống như kiểu int dựng sẵn, tất nhiên là ngoại trừ việc chúng có ngữ nghĩa riêng và tập hợp các hoạt động. Tương tự, một vectơ và một chuỗi giống như các mảng dựng sẵn, ngoại trừ việc chúng hoạt động tốt hơn (§4.2, §4.3.2, §4.4.1).

Đặc điểm xác định của một loại cụ thể là biểu diễn của nó là một phần của định nghĩa. Trong nhiều trường hợp quan trọng, chẳng hạn như một vectơ , biểu diễn đó chỉ là một hoặc nhiều con trỏ tới nhiều dữ liệu được lưu trữ ở nơi khác, nhưng nó hiện diện trong mỗi đối tượng của một lớp cụ thể. Điều đó cho phép việc triển khai đạt hiệu quả tối ưu về thời gian và không gian. Đặc biệt, nó cho phép chúng ta • đặt các đối tượng thuộc loại cụ thể vào ngăn xếp, trong bộ nhớ được

cấp phát tĩnh và trong các bộ nhớ khác.

đối tượng (§6.4.2);

• tham chiếu trực tiếp đến các đối tượng (chứ không chỉ thông qua con trỏ hoặc tham chiếu); •

khởi tạo đối tượng ngay lập tức và đầy đủ (ví dụ: sử dụng hàm tạo; §2.3.2); và • sao chép đối tượng (§3.3).

Biểu diễn có thể là riêng tư (như đối với Vector; §2.3.2) và chỉ có thể truy cập thông qua các hàm thành viên, nhưng nó hiện diện. Do đó, nếu biểu diễn thay đổi theo bất kỳ cách quan trọng nào, người dùng phải biên dịch lại. Đây là cái giá phải trả để có các loại bê tông hoạt động chính xác như tích hợp sẵn

Machine Translated by Google

các loại. Đối với các loại không thay đổi thư ờng xuyên và khi các biến cục bộ mang lại sự rõ ràng và hiệu quả rất cần thiết, điều này có thể chấp nhận đư ợc và thư ờng là lý tư ởng. Để tăng tính linh hoạt, một loại cụ thể có thể giữ các phần chính của biểu diễn của nó trên kho lư u trữ miễn phí (bộ nhớ động, heap) và truy cập chúng thông qua phần đư ợc lư u trữ trong chính đối tư ợng lớp. Đó là cách thực hiện vectơ và chuỗi ; chúng có thể đư ợc coi là xử lý tài nguyên với các giao diện đư ợc chế tạo cẩn thận.

3.2.1.1 Loại số học

''Kiểu số học cổ điển do ngư ời dùng định nghĩa'' rất phức tạp:

```
phức hợp lớp {
      đôi tái, im; //    đại diện: hai đôi
public:
      complex(double r, double i) :re{r}, im{i} {}         xây dựng phức hợp từ hai vô hư ớng
      complex(double r) :re{r}, im{0} {}                 // xây dựng phức tạp từ một phức vô hư ớng
      complex() :re{0}, tôi{0} {}                        // hợp mặc định: {0,0}

      double real() const { return re; }
      void real(double d) { re=d; }
      double image() const { return im; }
      void image(double d) { im=d; }

      phức& toán tử+=(phức z) { re+=z.re , im+=z.im; trả lại   này; } // //          thêm vào lại  và tôi là
                                                                                      và trả về kết quả
      phức& toán tử  =(phức z) { re  =z.re , im  =z.im; trả lại   này; }

      phức& toán tử  =(phức); // phức& toán tử/ đư ợc xác định ngoài lớp ở đâu đó
      =(phức); //                              đư ợc xác định ngoài lớp ở đâu đó
};
```

Đây là phiên bản đơ n giản hóa một chút của tổ hợp thư viện chuẩn (§40.4). Bản thân định nghĩa lớp chỉ chứa các thao tác yêu cầu quyền truy cập vào biểu diễn. Các đại diện là đơ n giản và thông thư ờng. Vì những lý do thực tế, nó phải tư ơ ng thích với những gì Fortran đã cung cấp cách đây 50 năm và chúng tôi cần một nhóm ngư ời vận hành thông thư ờng. Ngoài các yêu cầu hợp lý, phức tạp phải hiệu quả nếu không nó sẽ không đư ợc sử dụng. Điều này ngụ ý rằng các hoạt động đơ n giản phải đư ợc nội tuyến.
Nghĩa là, các thao tác đơ n giản (chẳng hạn như hàm tạo, += và hình ảnh ()) phải đư ợc triển khai mà không có lệnh gọi hàm trong mã máy đư ợc tạo. Các chức năng đư ợc định nghĩa trong một lớp đư ợc mặc định nội tuyến. Một tổ hợp sức mạnh công nghiệp (như thư viện tiêu chuẩn) đư ợc triển khai cẩn thận để thực hiện nội tuyến phù hợp.

Hàm tạo có thể đư ợc gọi mà không có đối số đư ợc gọi là hàm tạo mặc định. Do đó, complex() là phư ơ ng thức khởi tạo mặc định của phức hợp . Bằng cách xác định một hàm tạo mặc định, bạn loại bỏ khả năng của các biến chư a đư ợc khởi tạo của loại đó.

Các chỉ định const trên các hàm trả về phần thực và phần ảo chỉ ra rằng các phần này các chức năng không sửa đổi đối tư ợng mà chúng đư ợc gọi.

Nhiều hoạt động hữu ích không yêu cầu truy cập trực tiếp vào biểu diễn phức tạp, vì vậy chúng có thể đư ợc định nghĩa riêng biệt với định nghĩa lớp:

Machine Translated by Google


```
toán tử phức +(phức a, phức b) { return a+=b; } toán tử phức
(phức a, phức b) { return a =b; } toán tử phức(phức a)
{ return { a.real(), a.imag()}; } toán tử phức (phức a, phức      // vui mừng trừ và
b) { return a =b; } toán tử phức/(phức a, phức b) { return
a/=b; }
```

Ở đây, tôi sử dụng thực tế là một đối số được truyền theo giá trị được sao chép để tôi có thể sửa đổi một đối số mà không ảnh hưởng đến bản sao của người gọi và sử dụng kết quả làm giá trị trả về.

Các định nghĩa của == và != rất đơn giản:

```
toán tử bool==(phức a, phức b) {                    // bình đẳng

     trả lại a.real()==b.real() && a.imag()==b.imag();
}

toán tử bool!=(phức a, phức b) {                    // không công bằng

     trả về !(a==b);
}

phức sqrt(phức tạp);

// ...
```

Lớp phức tạp có thể được sử dụng như thế này:

```
void f(phức z) {

     phức a{2,3};                // xây dựng {2.3,0.0} từ 2.3
     phức hợp b {1/a};
     phức c {a+z phức{1,2.3}}; // if
       ...
     (c != b)
          c = (b/a)+2 b;
}
```

Trình biên dịch chuyển đổi các toán tử liên quan đến số phức thành các lời gọi hàm thích hợp. Ví dụ: c!=b có nghĩa là toán tử!=(c,b) và 1/a có nghĩa là toán tử/(phức hợp{1},a).

Các toán tử do người dùng định nghĩa (''toán tử quá tải'') nên được sử dụng một cách thận trọng và theo quy ước. Cú pháp được cố định bởi ngôn ngữ, vì vậy bạn không thể xác định /. Ngoài ra, không thể thay đổi ý nghĩa của toán tử đối với các loại tích hợp, vì vậy bạn không thể xác định lại + để trừ các số nguyên.

3.2.1.2 Vùng chứa

Vùng chứa là một đối tượng chứa một tập hợp các phần tử, vì vậy chúng tôi gọi Vector là vùng chứa vì nó là loại đối tượng là vùng chứa. Như được định nghĩa trong §2.3.2, Vector không phải là một vùng chứa nhân đôi không hợp lý: nó rất dễ hiểu, thiết lập một bất biến hữu ích (§2.4.3.2), cung cấp quyền truy cập được kiểm tra phạm vi (§2.4.3.1) và cung cấp kích thước ( ) để cho phép chúng tôi lặp lại các phần tử của nó. Tuy nhiên, nó có một lỗ hổng chết người: nó phân bổ các phần tử bằng cách sử dụng new nhưng không bao giờ phân bổ chúng. Đó không phải là một ý tưởng hay bởi vì mặc dù C++ định nghĩa một giao diện cho bộ thu gom rác (§34.5), nhưng nó không phải là

Machine Translated by Google

đảm bảo rằng có sẵn một bộ nhớ chư a sử dụng cho các đối tư ợng mới. Trong một số môi trư ờng, bạn không thể sử dụng trình thu thập và đôi khi bạn thích kiểm soát hủy diệt chính xác hơ n (§13.6.4) vì lý do logic hoặc hiệu suất. Chúng ta cần một cơ chế để đảm bảo rằng bộ nhớ đư ợc cấp phát bởi hàm tạo đư ợc giải phóng; cơ chế đó là một hàm hủy:

```
lớp Vector
{ private:
        double  elem; int          // yếu tố trỏ đến một      của mảng sz nhân đôi
        sz; công
cộng:
        Vectơ (int s) :elem{new double[s]}, sz{s} {                    // nhà xây dựng: có đư ợc tài nguyên

                for (int i=0; i!=s; ++i) elem[i]=0;                    // khởi tạo các phần tử
        }

        ~Vector() { xóa[] phần tử; }                                   // hàm hủy: giải phóng tài nguyên

        double& toán tử[](int i); kích
        thư ớc int() const;
};
```

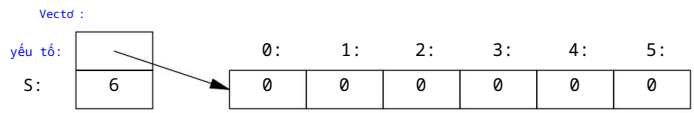
Tên của hàm hủy là toán tử bù, ~, theo sau là tên của lớp; nó là phần bù của một hàm tạo. Hàm tạo của Vector phân bổ một số bộ nhớ trên kho lư u trữ miễn phí (còn đư ợc gọi là heap hoặc kho lư u trữ động) bằng cách sử dụng toán tử mới . Hàm hủy sẽ dọn sạch bằng cách giải phóng bộ nhớ đó bằng toán tử xóa . Tất cả điều này đư ợc thực hiện mà không cần sự can thiệp của ngư ời dùng Vector. Ngư ời dùng chỉ cần tạo và sử dụng Vectơ giống như các biến của các loại tích hợp sẵn. Ví dụ:

```
khoảng trống fct(int n)
{
        Vectơ  v(n);

        // ... sử dụng v  ...

        {
                Vectơ  v2(2  n); //
                   ... sử dụng v và v2     ...
        } // v2 bị phá hủy ở đây

        // ... sử dụng v  ..

} //   bị phá hủy ở đây v
```

Vector tuân theo các quy tắc giống nhau về đặt tên, phạm vi, phân bổ, thời gian tồn tại, v.v., cũng như kiểu dựng sẵn, chẳng hạn như int và char. Để biết chi tiết về cách kiểm soát thời gian tồn tại của một đối tư ợng, hãy xem §6.4. Vector này đã đư ợc đơ n giản hóa bằng cách loại bỏ việc xử lý lỗi; xem §2.4.3.

Sự kết hợp hàm tạo/hàm hủy là cơ sở của nhiều kỹ thuật tao nhã. Đặc biệt, nó là cơ sở cho hầu hết các kỹ thuật quản lý tài nguyên chung của C++ (§5.2, §13.3). Hãy xem xét một minh họa đồ họa của một Vector:

Machine Translated by Google



Hàm tạo phân bổ các phần tử và khởi tạo các phần tử Vector một cách thích hợp. De structor giải phóng các phần tử. Mô hình xử lý dữ liệu này thường được sử dụng để quản lý dữ liệu có thể thay đổi kích thước trong suốt vòng đời của một đối tượng. Kỹ thuật thu thập tài nguyên trong một hàm tạo và giải phóng chúng trong một hàm hủy, được gọi là Khởi tạo thu thập tài nguyên hoặc RAII, cho phép chúng tôi loại bỏ '' các hoạt động mới trần trụi '', nghĩa là tránh phân bổ trong mã chung và giữ chúng bên trong việc thực hiện các trừu tượng hoạt động tốt. Tương tự như vậy, nên tránh '' thao tác xóa trần trụi ''. Việc tránh xóa mới và xóa hoàn toàn giúp mã ít bị lỗi hơn và dễ dàng hơn rất nhiều để tránh rò rỉ tài nguyên (§5.2).

3.2.1.3 Khởi tạo vùng chứa

Một thùng chứa tồn tại để chứa các phần tử, vì vậy rõ ràng chúng ta cần những cách thuận tiện để đưa các phần tử vào một thùng chứa. Chúng ta có thể xử lý điều đó bằng cách tạo một Vector với số lượng phần tử thích hợp và sau đó gán cho chúng, nhưng thông thường các cách khác sẽ thanh lịch hơn. Ở đây, tôi chỉ đề cập đến hai yêu thích:

- Hàm tạo Initializer-list: Khởi tạo với một danh sách các phần tử.
- push_back(): Thêm một phần tử mới vào cuối (ở phía sau) chuỗi.

Chúng có thể được khai báo như thế này:

```
lớp Vectơ { công
khai:
        Vectơ (std::initializ er_list<double>); // void          // khởi tạo với một danh sách
            ...
        push_back(double); //                                    // thêm phần tử vào cuối tăng kích thước bằng một
            ...
};
```

Push_back () hữu ích cho việc nhập số lượng phần tử tùy ý. Ví dụ:

```
Đọc vectơ (istream& is) {

        Vectơ v; for
        (double d; is>>d;)                    // đọc các giá trị dấu phẩy động vào d
                v.push_back(d); trở lại       // thêm d vào v
        v;
}
```

Vòng lặp đầu vào bị chấm dứt do lỗi kết thúc tệp hoặc lỗi định dạng. Cho đến khi điều đó xảy ra, mỗi số đọc được thêm vào Vector để cuối cùng, kích thước của v là số phần tử được đọc. Tôi đã sử dụng câu lệnh for thay vì câu lệnh while thông thường hơn để giữ phạm vi của d giới hạn trong vòng lặp. Việc triển khai push_back() được thảo luận trong §13.6.4.3. Cách cung cấp cho Vector một hàm tạo di chuyển, sao cho việc trả về một lượng dữ liệu khổng lồ có thể có từ read() là rẻ, được giải thích trong §3.3.2.

Machine Translated by Google

std ::initializer_list được sử dụng để xác định hàm khởi tạo danh sách khởi tạo là một loại thư viện chuẩn mà trình biên dịch biết: khi chúng ta sử dụng một {}-list, chẳng hạn như {1,2,3,4}, trình biên dịch sẽ tạo một đối tượng kiểu khởi tạo_list để cung cấp cho chương trình. Vì vậy, chúng ta có thể viết:

```
Vectơ v1 = {1,2,3,4,5}; Vectơ          // v1 có 5 phần tử
v2 = {1,23, 3,45, 6,7, 8};             // v2 có 4 yếu tố
```

Hàm tạo danh sách khởi tạo của Vector có thể được định nghĩa như sau:

```
Vectơ ::Vector(std::initializer_list<double> lst) :elem{new        // khởi tạo với một danh sách
        double[lst.size()]}, sz{lst.size()}
{
        sao chép (lst.begin(),lst.end(),elem);          //      từ lst vào bản sao elem
}
```

3.2.2 Các kiểu trừu tượng

Các loại như phức tạp và Vector được gọi là các loại cụ thể vì biểu diễn của chúng là một phần trong định nghĩa của chúng. Trong đó, chúng giống với các loại tích hợp sẵn. Ngược lại, một loại trừu tượng là một loại cách ly hoàn toàn người dùng khỏi các chi tiết triển khai. Để làm điều đó, chúng tôi tách giao diện khỏi biểu diễn và từ bỏ các biến cục bộ chính hãng. Vì chúng ta không biết gì về biểu diễn của một kiểu trừu tượng (thậm chí không biết kích thước của nó), nên chúng ta phải phân bổ các đối tượng trên kho lưu trữ miễn phí (§3.2.1.2, §11.2) và truy cập chúng thông qua các tham chiếu hoặc con trỏ (§2.2.5 , §7.2, §7.7).

Đầu tiên, chúng ta định nghĩa giao diện của một lớp Container mà chúng ta sẽ thiết kế như một phiên bản trừu tượng hơn của Vector của chúng ta:

```
lớp Container { public:
virtual
        double& operator[](int) = 0; kích thước int          // hàm  hàm ảo const
        ảo() const = 0; ảo ~Container() {}                     thành viên thuần túy (§3.2.1.1)
                                                             // hàm hủy (§3.2.1.2)
};
```

Lớp này là một giao diện thuần túy cho các thùng chứa cụ thể được định nghĩa sau. Từ ảo có nghĩa là ''có thể được định nghĩa lại sau này trong một lớp bắt nguồn từ lớp này.'' Không có gì ngạc nhiên khi một hàm được khai báo là ảo được gọi là một hàm ảo. Một lớp bắt nguồn từ Container cung cấp triển khai cho giao diện Con tainer . Cú pháp =0 tò mò cho biết chức năng này là ảo thuần túy; nghĩa là một số lớp dẫn xuất từ Container phải định nghĩa hàm. Như vậy, không thể định nghĩa một đối tượng chỉ là Container; một Vùng chứa chỉ có thể đóng vai trò là giao diện cho một lớp thực hiện các hàm operator[]() và size() của nó . Một lớp có chức năng thuần ảo được gọi là lớp trừu tượng.

Container này có thể được sử dụng như thế này:

```
vô hiệu sử dụng (Container & c)
{
        const int sz = c.size();

        for (int i=0; i!=sz; ++i)
                cout << c[i] << '\n';
}
```

Machine Translated by Google

Lưu ý cách sử dụng () sử dụng giao diện Vùng chứa mà hoàn toàn không biết chi tiết triển khai. Nó sử dụng size() và [] mà không biết chính xác loại nào cung cấp cách triển khai của chúng. Một lớp cung cấp giao diện cho nhiều lớp khác thường được gọi là loại đa hình (§20.3.2).

Như thường thấy đối với các lớp trừu tượng, Container không có hàm tạo. Rốt cuộc, nó không có bất kỳ dữ liệu nào để khởi tạo. Mặt khác, Container có một hàm hủy và hàm hủy đó là ảo. Một lần nữa, điều đó phổ biến đối với các lớp trừu tượng vì chúng có xu hướng bị thao túng thông qua các tham chiếu hoặc con trỏ và ai đó phá hủy Vùng chứa thông qua một con trỏ không biết tài nguyên nào được sở hữu bởi việc triển khai nó; xem thêm §3.2.4.

Một bộ chứa thực hiện các chức năng theo yêu cầu của giao diện được xác định bởi lớp trừu tượng Container có thể sử dụng lớp cụ thể Vector:

```
class Vector_container : public Container { //                    Vector_container thực hiện Container
        Vector v;
công cộng:
        Vector_container(int s) : v(s) { }              // Vector của các phần tử s
        ~Vector_container() {}

        double& operator[](int i) { return v[i]; } int
        size() const { return v.size(); }
};
```

:public có thể được đọc là ''có nguồn gốc từ'' hoặc ''là một kiểu con của.'' Lớp Vector_container được cho là có nguồn gốc từ lớp Container và lớp Container được cho là cơ sở của lớp Vector_container.
Một thuật ngữ khác gọi lớp con và lớp cha của Vector_container và Container tương ứng. Lớp dẫn xuất được cho là kế thừa các thành viên từ lớp cơ sở của nó, vì vậy việc sử dụng lớp cơ sở và lớp dẫn xuất thường được gọi là kế thừa.

Các thành viên operator[]() và size() được cho là ghi đè lên các thành viên tương ứng trong lớp cơ sở Container (§20.3.2). Hàm hủy (~Vector_container()) ghi đè hàm hủy của lớp cơ sở (~Container()). Lưu ý rằng hàm hủy thành viên (~Vector()) được gọi ngầm bởi hàm hủy của lớp (~Vector_container()).

Đối với một chức năng như use(Container&) để sử dụng Container mà hoàn toàn không biết về việc triển khai chi tiết, một số chức năng khác sẽ phải tạo một đối tượng mà nó có thể hoạt động. Ví dụ:

```
khoảng trống
g() {
        Vector_container vc {10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0}; sử dụng(vc);

}
```

Vì use() không biết về Vector_containers mà chỉ biết giao diện Vùng chứa , nên nó sẽ hoạt động tốt đối với cách triển khai Vùng chứa khác . Ví dụ:

```
class List_container : public Container { //              List_container triển khai danh sách nhân
        std::list<double> ld; // công cộng:  đôi Bộ chứa (thư viện tiêu chuẩn y) (§4.4.2)

        Danh_sách() { }                    // sách trốngDanh
        List_container(initializer_list<double> il) : ld{il} { }
        ~List_container() {}
```

Machine Translated by Google


```
        double& toán tử[](int i); int
        size() const { return ld.size(); }

};

double& List_container::operator[](int i) {

        for (auto& x : ld)
                { if (i==0) return
                x;   i;

        } throw out_of_rang e("Danh sách chứa");
}
```

Ở đây, biểu diễn là danh sách thư viện chuẩn <double>. Thông thường, tôi sẽ không triển khai bộ chứa với thao tác chỉ số con bằng cách sử dụng danh sách, bởi vì hiệu suất của chỉ số con danh sách rất kém so với chỉ số con vector . Tuy nhiên, ở đây tôi chỉ muốn trình bày một cách triển khai hoàn toàn khác với cách triển khai thông thường.

Một chức năng có thể tạo List_container và sử dụng () sử dụng nó:

```
khoảng trống
h() {
        List_container lc = { 1, 2, 3, 4, 5, 6, 7, 8, 9 }; sử
        dụng (lc);
}
```

Vấn đề là use(Container&) không biết liệu đối số của nó là Vector_container, List_container hay một số loại vùng chứa khác; nó không cần biết. Nó có thể sử dụng bất kỳ loại Container nào. Nó chỉ biết giao diện được định nghĩa bởi Container. Do đó, không cần biên dịch lại use(Container&) nếu việc triển khai List_container thay đổi hoặc một lớp hoàn toàn mới dẫn xuất từ Container được sử dụng.

Mặt trái của tính linh hoạt này là các đối tượng phải được thao tác thông qua con trỏ hoặc tham chiếu (§3.3, §20.4).

3.2.3 Hàm ảo

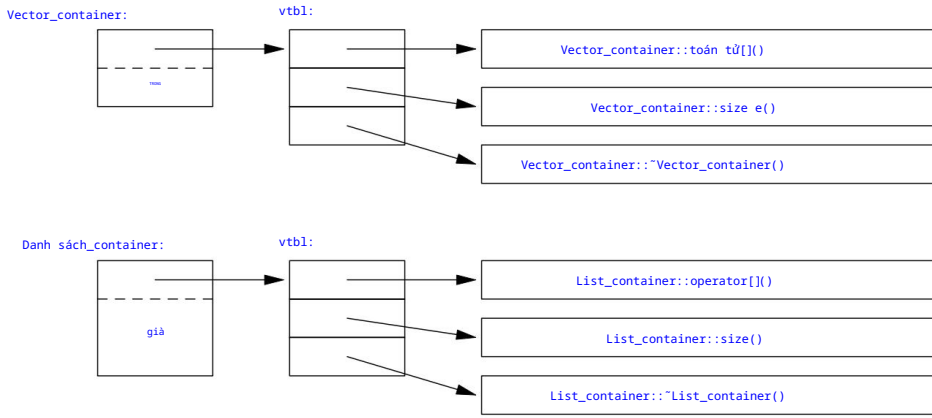
Xem xét lại việc sử dụng Container:

```
vô hiệu sử dụng (Container & c)
{
        const int sz = c.size();

        for (int i=0; i!=sz; ++i)
                cout << c[i] << '\n';
}
```

Cuộc gọi c[i] đang sử dụng() được giải quyết như thế nào đối với toán tử bên phải []()? Khi h() gọi use(), phải gọi toán tử List_container []() . Khi g() gọi use(), phải gọi toán tử của Vector_container []() . Để đạt được độ phân giải này, một đối tượng Vùng chứa phải chứa thông tin để cho phép nó chọn đúng chức năng để gọi trong thời gian chạy. Kỹ thuật thực hiện thông thường là để trình biên dịch chuyển đổi tên của một hàm ảo thành một chỉ mục thành một bảng các con trỏ tới các hàm. Cái bàn đó thường

Machine Translated by Google

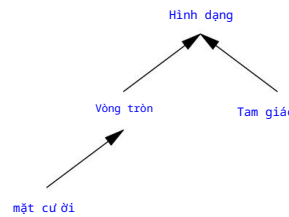
gọi là bảng hàm ảo hay đơn giản là vtbl. Mỗi lớp có chức năng ảo có vtbl riêng xác định các chức năng ảo của nó. Điều này có thể được biểu diễn bằng đồ họa như thế này:



Các chức năng trong vtbl cho phép đối tượng được sử dụng chính xác ngay cả khi kích thước của đối tượng và cách bố trí dữ liệu của nó không được người gọi biết. Việc triển khai người gọi chỉ cần biết vị trí của con trỏ tới vtbl trong Vùng chứa và chỉ mục được sử dụng cho từng chức năng ảo. Cơ chế gọi ảo này có thể được thực hiện gần như hiệu quả như cơ chế ''gọi chức năng thông thường'' (trong vòng 25%). Chi phí không gian của nó là một con trỏ trong mỗi đối tượng của một lớp với các chức năng ảo cộng với một vtbl cho mỗi lớp như vậy.

3.2.4 Phân cấp lớp

Ví dụ Container là một ví dụ rất đơn giản về phân cấp lớp. Hệ thống phân cấp lớp là một tập hợp các lớp được sắp xếp theo một mạng được tạo bởi dẫn xuất (ví dụ : công khai). Chúng tôi sử dụng hệ thống thứ bậc của lớp để đại diện cho các khái niệm có mối quan hệ thứ bậc, chẳng hạn như ''Xe cứu hỏa là một loại xe tải là một loại phương tiện'' và ''Mặt cười là một loại hình tròn là một loại phương tiện giao thông''. một loại hình dạng.'' Hệ thống phân cấp khổng lồ, với hàng trăm lớp, vừa sâu vừa rộng là phổ biến. Là một ví dụ cổ điển bán thực tế, hãy xem xét các hình dạng trên màn hình:



Các mũi tên đại diện cho các mối quan hệ thừa kế. Ví dụ, lớp Circle được dẫn xuất từ lớp Shape. Để biểu diễn sơ đồ đơn giản đó trong mã, trước tiên chúng ta phải chỉ định một lớp xác định các thuộc tính chung của tất cả các hình dạng:

Machine Translated by Google


```
lớp Shape { public:
virtual
        Point center() const = 0; di chuyển khoảng          // khiết  ảo tính
        trống ảo (Trỏ tới) = 0;

        vẽ khoảng trống ảo () const = 0; xoay               // vẽ "Canvas" hiện tại trên
        khoảng trống ảo (góc int) = 0;

        ảo ~Shape() {} //                                   // kẻ hủy diệt
           ...
};
```

Đư ơ ng nhiên, giao diện này là một lớp trừu tư ợng: khi có liên quan đến biểu diễn, không có gì (ngoại trừ vị trí của con trỏ tới vtbl) là chung cho mọi Hình dạng. Với định nghĩa này, chúng ta có thể viết các hàm tổng quát thao tác vectơ của con trỏ tới hình:

```
void rotate_all(vector<Shape  >& v, int angle) // {                xoay các phần tử của v  theo góc độ

        for (auto p : v)
                p  >xoay(góc);
}
```

Để xác định một hình dạng cụ thể, chúng ta phải nói rằng đó là một Hình dạng và chỉ định các thuộc tính cụ thể của nó (bao gồm cả các chức năng ảo của nó):

```
lớp Vòng tròn: hình dạng công khai { công
khai:
        Đư ờng tròn(Điểm p, int rr);                // ngư ời xây dựng

        Tâm điểm() const { return x; } void move(Trỏ
        tới) { x=to; }

        vẽ vô hiệu () const;
        khoảng trống xoay (int)                     // thuật  toán đơ n giản tốt đẹp
{} riêng tư :
        Điểm x; // int    bán kính
        r; //             trung tâm
};
```

Cho đến nay, ví dụ Hình dạng và Hình tròn không cung cấp gì mới so với Vùng chứa và Ví dụ về Vector_container , như ng chúng ta có thể xây dựng thêm:

```
class Smiley : public Circle { // public:      sử dụng vòng tròn như  cơ  sở cho một  khuôn mặt

        Mặt cư ời(Điểm p, int r): Vòng tròn{p,r}, miệng{nullptr} { }

        ~Smiley() {

                xóa miệng; cho (tự
                động p: mắt) xóa p;
        }
```

Machine Translated by Google


```
        di chuyển void(Trỏ tới);

        vẽ vô hiệu () const; vô
        hiệu xoay (int);

        void add_eye(Shape s) { eyes.push_back(s); } void
        set_mouth(Shape s); nháy mắt void
        ảo (int i);                          // nháy mắt số tôi mắt

        // ...

    riêng tư :
        vector<Shape > mắt;                  // thường có hai mắt
        Hình miệng;
    };
```

Hàm thành viên push_back() thêm đối số của nó vào vectơ (ở đây là mắt), tăng kích thước của vectơ đó lên một đơn vị.

```
    Bây giờ chúng ta có thể định nghĩa Smiley::draw() bằng cách sử dụng các lệnh gọi tới base của Smiley và các draw() thành viên:

    void Smiley::draw() {

        Hình tròn::vẽ();
        for (auto p : eyes)
            p >draw();
        miệng >vẽ();
    }
```

Lưu ý cách mà Smiley giữ mắt của nó trong một vectơ thư viện chuẩn và xóa chúng trong hàm hủy của nó. Hàm hủy của Shape là ảo và hàm hủy của Smiley ghi đè lên nó. Một hàm hủy ảo là cần thiết cho một lớp trừu tượng bởi vì một đối tượng của một lớp dẫn xuất thường được thao tác thông qua giao diện được cung cấp bởi lớp cơ sở trừu tượng của nó. Đặc biệt, nó có thể bị xóa thông qua một con trỏ tới một lớp cơ sở. Sau đó, cơ chế gọi hàm ảo đảm bảo rằng hàm hủy thích hợp được gọi.

Sau đó, hàm hủy đó gọi ngầm các hàm hủy của các cơ sở và thành viên của nó.

Trong ví dụ đơn giản này, nhiệm vụ của lập trình viên là đặt mắt và miệng phù hợp khéo léo trong vòng tròn đại diện cho khuôn mặt.

Chúng tôi có thể thêm các thành viên dữ liệu, hoạt động hoặc cả hai khi chúng tôi xác định một lớp mới bằng cách dẫn xuất. Điều này mang lại sự linh hoạt tuyệt vời với các cơ hội tương ứng cho sự nhầm lẫn và thiết kế kém. Xem Chương 21. Một hệ thống phân cấp lớp mang lại hai loại lợi ích:

- Kế thừa giao diện: Một đối tượng của lớp dẫn xuất có thể được sử dụng ở bất cứ nơi nào cần một đối tượng của lớp cơ sở. Nghĩa là lớp cơ sở đóng vai trò giao diện cho lớp dẫn xuất. Các lớp Con tainer và Shape là những ví dụ. Các lớp như vậy thường là các lớp trừu tượng.
- Kế thừa triển khai: Một lớp cơ sở cung cấp các chức năng hoặc dữ liệu giúp đơn giản hóa việc triển khai các lớp dẫn xuất. Smiley sử dụng hàm tạo của Circle và của Circle::draw() là những ví dụ. Các lớp cơ sở như vậy thường có các thành viên dữ liệu và hàm tạo.

Các lớp cụ thể – đặc biệt là các lớp có biểu diễn nhỏ – rất giống với các kiểu dựng sẵn: chúng ta định nghĩa chúng như các biến cục bộ, truy cập chúng bằng cách sử dụng tên của chúng, sao chép chúng, v.v. Các lớp trong hệ thống phân cấp lớp là khác nhau: chúng ta có xu hướng phân bổ chúng trên cửa hàng miễn phí bằng cách sử dụng mới và chúng tôi truy cập

Machine Translated by Google

chúng thông qua con trỏ hoặc tham chiếu. Ví dụ: hãy xem xét một hàm đọc dữ liệu mô tả các hình dạng từ luồng đầu vào và xây dựng các đối tượng Hình dạng thích hợp :

```
lớp enum Loại {hình tròn, hình tam giác, mặt cười};

Shape  read_shape(istream& is) {                    // đọc mô tả hình dạng từ luồng đầu vào là

        // ... đọc tiêu đề hình dạng từ is và tìm nó              con k    ...

        switch (k)
        { trường hợp Loại::circle:
                // đọc dữ liệu vòng kết nối  {Điểm,int} vào và pr
                trả về Vòng kết nối mới{p,r};
        case Kind::triangle: //
                  đọc dữ liệu tam giác    {Điểm,Điểm,Điểm} vào p1, p2 và p3
                return new Triangle{p1,p2,p3}; trường hợp
        Loại::smiley: //
                  đọc dữ liệu mặt cười  {Điểm,int,Hình,Hình,Hình} thành e1 ,e2 và p, r, m
                Mặt cười   ps = Mặt cười mới{p,r};
                ps  >add_eye(e1);
                ps  >add_eye(e2);
                ps  >set_mouth(m); trở lại
                ps;
        }
}
```

Một chương trình có thể sử dụng trình đọc hình đó như thế này:

```
người dùng vô hiệu ()
{
        std::vector<Shape  > v; trong
        khi (cin)
                v.push_back(read_shape(cin)); draw_all(v);
        rotate_all(v,45);                       // gọi draw() cho mỗi phần tử gọi
        for (auto p :v) xóa                     // rotate(45) cho mỗi phần tử
        p;                                      // nhớ xóa các phần tử
}
```

Rõ ràng, ví dụ này được đơn giản hóa – đặc biệt là đối với việc xử lý lỗi – nhưng nó minh họa một cách sinh động rằng user() hoàn toàn không biết nó thao tác với loại hình dạng nào. Mã user () có thể được biên dịch một lần và sau đó được sử dụng cho các Hình dạng mới được thêm vào chương trình. Lưu ý rằng không có con trỏ tới các hình bên ngoài user(), vì vậy user() chịu trách nhiệm giải phóng chúng. Điều này được thực hiện với toán tử xóa và chủ yếu dựa vào hàm hủy ảo của Shape . Bởi vì hàm hủy đó là ảo, xóa gọi hàm hủy cho lớp dẫn xuất nhất. Điều này rất quan trọng vì một lớp dẫn xuất có thể đã có được tất cả các loại tài nguyên (chẳng hạn như xử lý tệp, khóa và luồng đầu ra) cần được giải phóng. Trong trường hợp này, Smiley xóa các đối tượng mắt và miệng của nó .

Các lập trình viên có kinh nghiệm sẽ nhận thấy rằng tôi đã bỏ ngỏ hai cơ hội rõ ràng để mắc lỗi: • Người dùng có thể không xóa được con trỏ được trả về bởi read_shape(). • Người sở hữu vùng chứa các con trỏ Hình dạng có thể không xóa các đối tượng được trỏ tới.

Theo nghĩa đó, các hàm trả về một con trỏ tới một đối tượng được phân bổ trên cửa hàng miễn phí là nguy hiểm.

Machine Translated by Google

Một giải pháp cho cả hai vấn đề là trả về thư viện chuẩn unique_ptr (§5.2.1) thay vì ''con trỏ trần'' và lưu trữ unique_ptrs trong vùng chứa:

```
unique_ptr<Shape> read_shape(istream& is) // đọc mô tả hình dạng từ luồng đầu vào là {

        // đọc tiêu đề hình dạng từ is và tìm nó con k

        switch (k)
        { trường hợp Loại::circle:
                // đọc dữ liệu vòng kết nối {Điểm,int} vào và pr
                return unique_ptr<Shape>{Vòng kết nối mới{p,r}};                    // §5.2.1
        // ...
}

người dùng vô hiệu ()
{
        vector<unique_ptr<Hình>v; trong khi (cin)

        v.push_back(read_shape(cin)); draw_all(v);
        rotate_all(v,45);                    // gọi draw() cho mỗi phần tử gọi
                                                // rotate(45) cho mỗi phần tử
} // tất cả Hình dạng hoàn toàn bị phá hủy
```

Bây giờ đối tượng được sở hữu bởi unique_ptr sẽ xóa đối tượng khi nó không còn cần thiết nữa, tức là khi unique_ptr của nó vượt quá phạm vi.

Để phiên bản unique_ptr của user() hoạt động, chúng ta cần các phiên bản draw_all() và rotate_all() chấp nhận vector<unique_ptr<Shape>>s. Viết nhiều hàm _all() như vậy có thể trở nên tẻ nhạt, vì vậy §3.4.3 đưa ra một giải pháp thay thế.

3.3 Sao chép và Di chuyển

Theo mặc định, các đối tượng có thể được sao chép. Điều này đúng với các đối tượng thuộc kiểu do người dùng định nghĩa cũng như đối với các kiểu dựng sẵn. Ý nghĩa mặc định của copy là memberwise copy: sao chép từng thành viên. Ví dụ: sử dụng phức hợp từ §3.2.1.1:

```
kiểm tra void (phức tạp z1) {

        phức z2{z1}; phức hợp        // khởi tạo sao chép
        z3; z3 = z2; //
                                     //        sao chép bài tập
          ...
}
```

Bây giờ z1, z2 và z3 có cùng giá trị vì cả phép gán và khởi tạo đều sao chép cả hai thành viên.

Khi chúng ta thiết kế một lớp, chúng ta phải luôn xem xét liệu một đối tượng có thể được sao chép hay không và bằng cách nào. Đối với các loại cụ thể đơn giản, sao chép thành viên thường chính xác là ngữ nghĩa phù hợp cho sao chép. Đối với một số loại cụ thể phức tạp, chẳng hạn như Vector, sao chép theo thành viên không phải là ngữ nghĩa phù hợp để sao chép và đối với các loại trừu tượng thì hầu như không bao giờ.

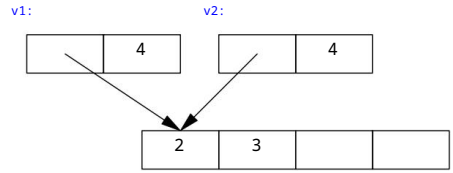
Machine Translated by Google

3.3.1 Sao chép vùng chứa

Khi một lớp là một bộ điều khiển tài nguyên, nghĩa là nó chịu trách nhiệm đối với một đối tư ợng đư ợc truy cập thông qua một con trỏ, thì bản sao theo thành viên mặc định thư ờng là một thảm họa. Sao chép theo chiều thành viên sẽ vi phạm bất biến của trình xử lý tài nguyên (§2.4.3.2). Ví dụ: bản sao mặc định sẽ để lại một bản sao của vòng tham chiếu Vector có cùng thành phần như bản gốc:

```
void bad_copy(Vector v1) {

        Vectơ v2 = v1;          // v2[0] biểu diễn của v1 thành v2 copy
        v1[0] = 2;                cũng là 2!lbây giờ v1[1]
        v2[1] = 3;              // bây giờ        cũng 3!
}
```

Giả sử rằng v1 có bốn phần tử, kết quả có thể đư ợc biểu diễn bằng đồ thị như sau:



May mắn thay, thực tế là Vector có hàm hủy là một gợi ý mạnh mẽ rằng ngữ nghĩa sao chép mặc định (theo thành viên) là sai và trình biên dịch ít nhất nên cảnh báo đối với ví dụ này (§17.6). Chúng ta cần xác định ngữ nghĩa sao chép tốt hơ n.

Việc sao chép một đối tư ợng của một lớp đư ợc xác định bởi hai thành viên: một hàm tạo sao chép và một phép gán sao chép:

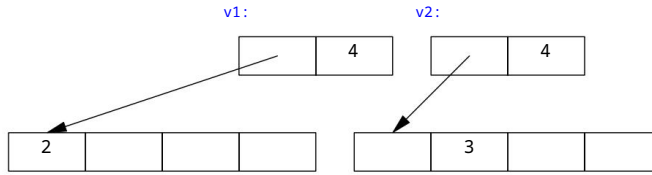
```
lớp Vector
{ private:
        double  elem; // int  elem trỏ đến nhân đôi một mảng sz
        sz;
công cộng:
        Vectơ  (int                                  // hàm tạo: thiết lập bất biến, lấy tài nguyên hàm
        s); ~Vector() { xóa[] phần tử; }             // hủy: giải phóng tài nguyên

        Vectơ (const Vectơ &a);                      // sao    constr uctor
        Vector& toán tử=(const Vector& a);           // chépchép bài tập sao

        double& toán tử[](int i); const
        double& toán tử[](int i) const;

        kích thư ớc int() const;
};
```

Một định nghĩa phù hợp về hàm tạo bản sao cho Vector phân bổ không gian cho số lư ợng phần tử cần thiết và sau đó sao chép các phần tử vào đó, sao cho sau một bản sao, mỗi Vector có bản sao các phần tử của riêng nó:

Machine Translated by Google


```
Vectơ ::Vector(const Vector&          // uctor bản sao constr
      a) :elem{new double[sz]},       // phân bổ không gian cho các phần tử
       sz{a.sz}
{
      for (int i=0; i!=sz; +          // chép yếu tố sao
            +i) elem[i] = a.elem[i];
}
```

Kết quả của ví dụ v2=v1 bây giờ có thể được trình bày dưới dạng:



Tất nhiên, chúng ta cần gán bản sao ngoài hàm tạo bản sao:

```
Vector& Vector::toán tử=(const Vector& a) {          // tập sao chép bài

      double  p = new double[a.sz]; for
      (int i=0; i!=a.sz; ++i)
            p[i] = a.elem[i];
      xóa [] phần tử; // elem  xóa các yếu tố cũ
      = p; sz =
      a.sz; trả
      lại  này;
}
```

Tên này được xác định trước trong hàm thành viên và trỏ đến đối tượng mà hàm thành viên được gọi.

Một hàm tạo bản sao và một phép gán bản sao cho lớp X thường được khai báo để nhận một đối số đề cập đến loại const X&.

3.3.2 Di chuyển Container

Chúng ta có thể kiểm soát việc sao chép bằng cách xác định hàm tạo bản sao và gán bản sao, nhưng việc sao chép có thể tốn kém đối với các vùng chứa lớn. Coi như :

```
Toán tử vectơ +(const Vector& a, const Vector& b) {

      if (a.size()!=b.siz
            e()) ném Vector_siz e_mismatch{};

      Độ phân giải véc-
      tơ (a.size()); for (int i=0; i!
            =a.size(); ++i)
      res[i]=a[i]+b[i]; trả lại độ phân giải;
}
```

Machine Translated by Google

Trả về từ + liên quan đến việc sao chép kết quả ra khỏi biến cục bộ res và vào một nơ i nào đó mà ngư ời gọi có thể truy cập. Chúng ta có thể sử dụng cái này + như thế này:

```
void f(const Vector& x, const Vector& y, const Vector& z) {

        Vectơ
        r; .//
        r = x+y+z; //
          ...
}
```

Đó sẽ là sao chép một Vector ít nhất hai lần (một lần cho mỗi lần sử dụng toán tử +). Nếu một Vector lớn, chẳng hạn, 10.000 lần nhân đôi, điều đó có thể gây lúng túng. Phần đáng xấu hổ nhất là res trong toán tử+() không bao giờ đư ợc sử dụng lại sau khi sao chép. Chúng tôi không thực sự muốn có một bản sao; chúng tôi chỉ muốn lấy kết quả từ một hàm: chúng tôi muốn di chuyển một Vector hơ n là sao chép nó. May mắn thay, chúng ta có thể nói rõ ý định đó:

```
lớp Vectơ { //
        ...

        Vectơ (const Vectơ & a);                    // trúc bản saocấu
        Vector& toán tử=(const Vector& a);                // tập sao chép bài

        Vectơ (Vectơ && a);                         // di chuyển cấu trúc
        Vector& toán tử=(Vectơ && a);                     // chuyển như ợng
};
```

Với định nghĩa đó, trình biên dịch sẽ chọn hàm tạo di chuyển để thực hiện việc chuyển giá trị trả về ra khỏi hàm. Điều này có nghĩa là r=x+y+z sẽ không liên quan đến việc sao chép các Vectơ .
Thay vào đó, Vectors chỉ đư ợc di chuyển.

Thông thư ờng, hàm tạo di chuyển của Vector rất đơ n giản để xác định:

```
Vectơ ::Vector(Vector&&
        a) :elem{a.elem}, //       "lấy các phần tử" từ một
         sz{a.sz}
{
        a.elem = nullptr;              // là không có các yếu tố bây giờ
        a.sz = 0;
}
```

&& có nghĩa là ''tham chiếu giá trị'' và là một tham chiếu mà chúng ta có thể liên kết một giá trị (§6.4.1) . Từ ''giá trị'' nhằm bổ sung cho từ ''lvalue'', có nghĩa đại khái là ''thứ gì đó có thể xuất hiện ở phía bên trái của một phép gán.'' Vì vậy, giá trị là – theo phép tính gần đúng đầu tiên – một giá trị mà bạn không thể gán cho, chẳng hạn như một số nguyên đư ợc trả về bởi lệnh gọi hàm và tham chiếu giá trị là tham chiếu đến thứ mà không ai khác có thể gán cho. Biến cục bộ res trong toán tử+() cho Vec tors là một ví dụ.

Hàm tạo di chuyển không nhận đối số const : xét cho cùng, hàm tạo di chuyển đư ợc cho là loại bỏ giá trị khỏi đối số của nó. Một chuyển như ợng di chuyển đư ợc xác định tư ơ ng tự.

Thao tác di chuyển đư ợc áp dụng khi một tham chiếu giá trị đư ợc sử dụng làm trình khởi tạo hoặc làm phía bên phải của phép gán.

Machine Translated by Google

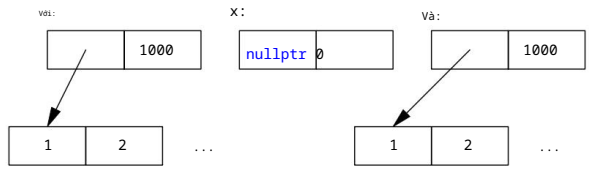
Sau khi di chuyển, một đối tượng được di chuyển từ phải ở trạng thái cho phép chạy hàm hủy. đánh máy cụ thể, chúng ta cũng nên cho phép gán cho một đối tượng được chuyển từ (§17.5, §17.6.2).

Trường hợp lập trình viên biết rằng một giá trị sẽ không được sử dụng lại, nhưng trình biên dịch không thể dự kiến sẽ đủ thông minh để tìm ra điều đó, lập trình viên có thể cụ thể:

```
Vector f() {
        Vector x(1000);
        Vector y(1000);
        Vector z(1000); //
           ...
        z = x;                          // chúng tôi có Một sao chép
           = std::move(x); // y //        chúng tôi được di chuyển
           ...
        trả lại z;                      // chúng tôi được di chuyển
};
```

Hàm thư viện chuẩn move() trả về một tham chiếu giá trị cho đối số của nó.

Ngay trước khi trở lại, chúng tôi có:



Khi z bị hủy, nó cũng đã được di chuyển từ (bằng cách trả về) sao cho, giống như x, nó trống (không chứa phần tử nào).

3.3.3 Quản lý tài nguyên

Bằng cách xác định các hàm tạo, thao tác sao chép, thao tác di chuyển và hàm hủy, lập trình viên có thể cung cấp khả năng kiểm soát hoàn toàn thời gian tồn tại của tài nguyên được chứa (chẳng hạn như các phần tử của bộ chứa). Hơn nữa, một hàm tạo di chuyển cho phép một đối tượng di chuyển đơn giản và rẻ tiền từ phạm vi này sang phạm vi khác. Bằng cách đó, các đối tượng mà chúng ta không thể hoặc không muốn sao chép ra khỏi một phạm vi có thể được di chuyển ra ngoài một cách đơn giản và rẻ tiền. Hãy xem xét một chuỗi thư viện tiêu chuẩn đại diện cho một hoạt động cho thuê đồng thời (§5.3.1) và một Vector triệu nhân đôi. Chúng tôi không thể sao chép cái trước và không muốn sao chép cái sau.

```
std::vector<thread> my_threads;

Vector init(int n) {
        sợi t{nhịp tim};                          // chạy nghe tbeat đồng thời (riêng t vào my_threads di chủ đề)
        my_threads.push_back(move(t)); // //            chuyển
           ... khởi tạo thêm ...
```

Machine Translated by Google


```
        Vectơ vec(n); for
        (int i=0; i<vec.size(); ++i) vec[i] = 777; trả vec; //
                                                    di chuyển độ phân giải ra khỏi init()
    }

    tự động v = init(); //  sao t nghe tbeat và khởi tạo v
```

Điều này làm cho các bộ điều khiển tài nguyên, chẳng hạn như Vectơ và luồng, trở thành một giải pháp thay thế cho việc sử dụng con trỏ trong nhiều trư ờng hợp. Trên thực tế, ''con trỏ thông minh'' của thư viện chuẩn, chẳng hạn như unique_ptr, chính là các trình điều khiển tài nguyên (§5.2.1).

Tôi đã sử dụng vectơ thư viện chuẩn để giữ các luồng vì chúng tôi không tham số hóa Vectơ với một loại phần tử cho đến §3.4.1.

Rất giống với cách tạo mới và xóa biến mất khỏi mã ứng dụng, chúng ta có thể làm cho con trỏ biến mất trong phần xử lý tài nguyên. Trong cả hai trư ờng hợp, kết quả là mã đơ n giản hơ n và dễ bảo trì hơ n mà không cần thêm chi phí. Đặc biệt, chúng ta có thể đạt đư ợc sự an toàn tài nguyên mạnh mẽ; nghĩa là chúng ta có thể loại bỏ rò rỉ tài nguyên đối với một khái niệm chung về tài nguyên. Ví dụ là các vectơ giữ bộ nhớ, các luồng chứa các luồng hệ thống và các fstream giữ các phần xử lý tệp.

3.3.4 Loại bỏ các hoạt động Sử dụng bản sao

mặc định hoặc di chuyển cho một lớp trong cấu trúc phân cấp thư ờng là một thảm họa: chỉ đư ợc cung cấp một con trỏ tới một cơ sở, chúng ta đơ n giản là không biết lớp dẫn xuất có những thành viên nào (§3.2.2), vì vậy chúng tôi không thể biết làm thế nào để sao chép chúng. Vì vậy, điều tốt nhất nên làm thư ờng là xóa các thao tác sao chép và di chuyển mặc định, nghĩa là loại bỏ các định nghĩa mặc định của hai thao tác đó:

```
    lớp Hình dạng { công
    khai:
        Hình dạng (const Hình dạng &) = xóa;                    // KHÔNG chép hoạt động sao
        Shape& operator=(const Shape&) =delete;

        Hình dạng(Hình dạng&&) =xóa;                            // KHÔNG di chuyển hoạt động
        Hình dạng& toán tử=(Hình dạng&&) =xóa;

        hình dạng();
        // ...
    };
```

Bây giờ, một nỗ lực sao chép Hình dạng sẽ bị trình biên dịch bắt gặp. Nếu bạn cần sao chép một đối tư ợng trong hệ thống phân cấp lớp, hãy viết một số loại hàm sao chép (§22.2.4).

Trong trư ờng hợp cụ thể này, nếu bạn quên xóa một thao tác sao chép hoặc di chuyển, thì cũng không có hại gì. Thao tác di chuyển không đư ợc tạo ngầm định cho một lớp mà ngư ời dùng đã khai báo rõ ràng một cấu trúc hủy. Hơ n nữa, việc tạo các thao tác sao chép không đư ợc dùng trong trư ờng hợp này (§44.2.3). Đây có thể là một lý do chính đáng để xác định rõ ràng một hàm hủy ngay cả khi trình biên dịch sẽ ngụ ý cung cấp một hàm hủy (§17.2.3).

Lớp cơ sở trong hệ thống phân cấp lớp chỉ là một ví dụ về đối tư ợng mà chúng ta không muốn sao chép. MỘT xử lý tài nguyên nói chung không thể đư ợc sao chép chỉ bằng cách sao chép các thành viên của nó (§5.2, §17.2.2).

Cơ chế =delete là chung, nghĩa là nó có thể đư ợc sử dụng để chặn bất kỳ thao tác nào (§17.6.4).

Machine Translated by Google

3.4 Mẫu

Một người nào đó muốn một véctơ không phải lúc nào cũng muốn một véctơ nhân đôi. Vectơ là một khái niệm chung, độc lập với khái niệm số dấu phẩy động. Do đó, loại phần tử của vectơ phải được biểu diễn độc lập. Mẫu là một lớp hoặc một hàm mà chúng ta tham số hóa bằng một tập hợp các loại hoặc giá trị . Chúng tôi sử dụng các mẫu để biểu diễn các khái niệm được hiểu tốt nhất là một cái gì đó rất chung chung mà từ đó chúng tôi có thể tạo các loại và chức năng cụ thể bằng cách chỉ định các đối số, chẳng hạn như loại phần tử double.

3.4.1 Các loại tham số hóa

Chúng ta có thể khái quát hóa loại vectơ nhân đôi thành loại vectơ bất kỳ thứ gì bằng cách đặt nó làm mẫu và thay thế loại cụ thể double bằng một tham số. Ví dụ:

```
mẫu <typename T> lớp
Vector { private:

      phần tử T  ; // elem trỏ đến các phần tử của kiểu T
                   mảng sz
      int no;
công cộng:
      Vectơ (int                    // hàm tạo: thiết lập bất biến, lấy tài nguyên hàm hủy:
      s); ~Vector() { xóa[] phần tử; }  // giải phóng tài nguyên

      // ··· copy và các thao tác di chuyển...

      Toán tử T&[](int i);
      const T& toán tử[](int i) const;
      int size() const { return sz; }
};
```

Tiền tố template <typename T> làm cho T trở thành một tham số của phần khai báo mà nó đặt trước. Nó là phiên bản toán học ''cho mọi T'' của C++ hay chính xác hơn là ''cho mọi kiểu T.'' Các hàm thành viên có thể được định nghĩa tương tự:

```
mẫu <tên chữ T>
Vectơ <T>::Vector(int s) {

      if (s<0) throw Negative_siz e{};
      elem = T[s] mới;
      sz = s;
}

mẫu<typename T> const T&
Vector<T>::operator[](int i) const {

      if (i<0 ||
            size()<=i) throw out_of_rang e{"Vector::operator[]"};
      trả lại elem[i];
}
```

Machine Translated by Google

Đưa ra các định nghĩa này, chúng ta có thể định nghĩa các vectơ như thế này:

```
Vectơ <char> vc(200);                // véc tơ 200 ký tự
Vectơ <chuỗi> vs(17);                  vectơ 17 chuỗi vectơ
Vectơ <danh sách<int>> vli(45);      // 45 danh sách số nguyên
```

>> trong Vector<list<int>> chấm dứt các đối số mẫu lồng nhau ; nó không phải là một toán tử đầu vào bị đặt sai chỗ. Không cần thiết (như trong C++ 98) để đặt khoảng trắng giữa hai > s.

Chúng ta có thể sử dụng các Vector như thế này:

```
void write(const Vector<string>& vs) {                // Vectơ của một số dây

      for (int i = 0; i!=vs.size(); ++i)
            cout << vs[i] << '\n';
}
```

Để hỗ trợ vòng lặp phạm vi cho Vector của chúng ta, chúng ta phải xác định các hàm begin() và end() phù hợp :

```
mẫu <tên chữ T>
T   bắt đầu(Vector<T>& x) {

      trả về &x[0];           // con trỏ để yếu tố đầu tiên
}

mẫu <tên chữ T>
T   end(Vectơ <T>& x) {

      trả về x.begin()+x.size(); //   con trỏ tới phần tử one-past-last
}
```

Với những điều đó, chúng ta có thể viết:

```
void f2(const Vector<string>& vs) // {      Vector của một số dây

      for (auto& s : vs)
            cout << s << '\n';
}
```

Tương tự, chúng ta có thể định nghĩa danh sách, vectơ , bản đồ (nghĩa là mảng kết hợp), v.v., dưới dạng mẫu (§4.4, §23.2, Chương 31).

Các mẫu là một cơ chế thời gian biên dịch, vì vậy việc sử dụng chúng không phát sinh chi phí thời gian chạy so với ''mã viết tay'' (§23.2.2).

3.4.2 Mẫu chức năng

Các mẫu có nhiều cách sử dụng hơn là chỉ tham số hóa một vùng chứa với một loại phần tử. Đặc biệt, chúng được sử dụng rộng rãi để tham số hóa cả hai loại và thuật toán trong thư viện tiêu chuẩn (§4.4.5, §4.5.5). Ví dụ: chúng ta có thể viết một hàm tính tổng các giá trị phần tử của bất kỳ vùng chứa nào như sau:

Machine Translated by Google


```
mẫu <typename Container, typename Giá trị>
Tổng giá trị(const Container& c, Giá trị v) {

        for (auto x : c)
                v+=x;
        trở lại v;
}
```

Đối số mẫu Giá trị và đối số hàm v ở đó để cho phép người gọi chỉ định loại và giá trị ban đầu của bộ tích lũy (biến để tích lũy tổng):

```
void user(Vector<int>& vi, std::list<double>& ld, std::vector<complex<double>>& vc) {

        int x = tổng(vi,0);                          // tổng của một vectơ các số nguyên (thêm các số nguyên) của mục
        double d = sum(vi,0.0); đôi                  // Tổng vectơ các số nguyên (thêm các số nguyên)
        dd = tổng (ld, 0,0); auto z =                // danh sách tổng gấp đôi
        sum(vc,complex<double>{}); //                  Tổng của một vectơ phức tạp <double> giá
                                                     // trị ban đầu là {0.0,0.0}
}
```

Mục đích của việc thêm số nguyên vào một số kép sẽ là xử lý một cách duyên dáng một số lớn hơn số nguyên lớn nhất. Lưu ý cách các loại đối số mẫu cho sum<T,V> được suy ra từ các đối số hàm. May mắn thay, chúng ta không cần chỉ định rõ ràng các loại đó.

Tổng () này là một phiên bản đơn giản hóa của thư viện tiêu chuẩn tích lũy () (§40.6).

3.4.3 Đối tượng chức năng

Một loại khuôn mẫu đặc biệt hữu ích là đối tượng hàm (đôi khi được gọi là functor), được sử dụng để xác định các đối tượng có thể được gọi giống như các hàm. Ví dụ:

```
mẫu <typename T> lớp Ít hơn
{ const T val; công
        cộng:                   // có giá trị đối với so sánh với

        Less_than(const T& v) :val(v) { } toán tử
        bool()(const T& x) const { return x<val; } //   nhà điều hành cuộc gọi
};
```

Hàm được gọi là operator() thực hiện ''gọi hàm,'' ''gọi,'' hoặc ''ứng dụng'' toán tử ().

Chúng ta có thể định nghĩa các biến được đặt tên thuộc loại Less_than cho một số loại đối số:

```
Ít hơn<int> lti {42}; //                lti(i) will ilts(s) bằng cách sử dụng so sánh < (tôi<42)
Less_than<string> lts {"Backus"}; //    so sánh s ĐẾN "Backus" sử dụng < (s<"Backus")
```

Chúng ta có thể gọi một đối tượng như vậy, giống như chúng ta gọi một hàm:

```
void fct(int n, const string & s) {

        bool b1 = lti(n);          // nếu đúng n<42
        bool b2 = lts(s); //       // true nếu s<"Backus"
          ...
}
```

Machine Translated by Google

Các đối tượng hàm như vậy được sử dụng rộng rãi làm đối số cho các thuật toán. Ví dụ: chúng ta có thể đếm số lần xuất hiện của các giá trị mà vị từ trả về giá trị đúng:

```
mẫu<typename C, typename P> int count(const
C& c, P pred) {

      int cnt =
      0; for (const auto& x :
            c) if

      (pred(x)) ++cnt; trở lại cnt;
}
```

Vị ngữ là thứ mà chúng ta có thể gọi để trả về giá trị đúng hoặc sai. Ví dụ:

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s) {

      cout << "số lượng giá trị nhỏ hơn " << x
            << ": " << đếm(vec,Less_than<int>{x}) << '\n';
            cout <<
      "số lượng giá trị nhỏ hơn " << s
            << ": " << đếm(lst,Less_than<string>{s}) << '\n';

}
```

Ở đây, Less_than<int>{x} xây dựng một đối tượng mà toán tử cuộc gọi so sánh với int được gọi là x; Less_than<string>{s} xây dựng một đối tượng so sánh với chuỗi gọi là s. Vẻ đẹp của các đối tượng chức năng này là chúng mang giá trị được so sánh với chúng. Chúng ta không cần phải viết một hàm riêng biệt cho từng giá trị (và từng loại) và chúng ta không cần phải giới thiệu các biến toàn cục khó chịu để giữ các giá trị. Ngoài ra, đối với một đối tượng hàm đơn giản như nội tuyến Less_than rất đơn giản, do đó, lệnh gọi Less_than hiệu quả hơn nhiều so với lệnh gọi hàm gián tiếp. Khả năng mang dữ liệu cộng với hiệu quả của chúng làm cho các đối tượng hàm đặc biệt hữu ích làm đối số cho các thuật toán.

Các đối tượng hàm được sử dụng để xác định ý nghĩa của các thao tác chính của thuật toán tổng quát (chẳng hạn như Less_than cho count()) thường được gọi là đối tượng chính sách.

Chúng ta phải định nghĩa riêng Less_than khỏi việc sử dụng nó. Điều đó có thể được coi là bất tiện. Do đó, có một ký hiệu cho các đối tượng hàm sinh ngầm định:

```
void f(const Vector<int>& vec, const list<string>& lst, int x, const string& s) {

      cout << "số lượng giá trị nhỏ hơn " << x
            << ": " << đếm(vec,[&](int a){ return a<x; }) <<
            '\n';
      cout << "số lượng giá trị nhỏ hơn " << s
            << ": " << đếm(lst,[&](const string& a){ return a<s; }) << '\n';

}
```

Ký hiệu [&](int a){ return a<x; } được gọi là biểu thức lambda (§11.4). Nó tạo ra một đối tượng hàm chính xác như Less_than<int>{x}. [&] là danh sách chụp xác định rằng các tên cục bộ được sử dụng (chẳng hạn như x) sẽ được chuyển qua tham chiếu. Nếu chúng ta chỉ muốn ''chụp'' x, chúng ta có thể nói

Machine Translated by Google

vì vậy: [&x]. Nếu chúng ta muốn cung cấp cho đối tượng được tạo một bản sao của x, chúng ta có thể nói như vậy: [=x]. Ghi lại không có gì là [], ghi lại tất cả các tên cục bộ được sử dụng theo tham chiếu là [&] và ghi lại tất cả các tên cục bộ được sử dụng bởi giá trị là [=].

Sử dụng lambdas có thể thuận tiện và ngắn gọn, nhưng cũng khó hiểu. Đối với các hành động không cần thiết (chẳng hạn, hơn cả một biểu thức đơn giản), tôi muốn đặt tên cho hoạt động đó để nêu rõ hơn mục đích của nó và để nó có sẵn để sử dụng ở một số nơi trong chương trình.

Trong §3.2.4, chúng tôi nhận thấy sự khó chịu khi phải viết nhiều hàm để thực hiện các thao tác trên các phần tử của vectơ con trỏ và unique_ptrs, chẳng hạn như draw_all() và rotate_all(). Các đối tượng chức năng (cụ thể là lambdas) có thể trợ giúp bằng cách cho phép chúng ta tách quá trình truyền tải của vùng chứa khỏi đặc điểm kỹ thuật của những gì sẽ được thực hiện với từng phần tử.

Đầu tiên, chúng ta cần một hàm áp dụng một thao tác cho từng đối tượng được chỉ ra bởi các phần tử của một vùng chứa con trỏ:

```
mẫu<lớp C, lớp Toán tử> void for_all(C&
c, Toán tử op) {                    // giả sử cái đó C là một thùng chứa các con trỏ

        for (auto& x : c)
                op( x);             // vượt qua bật () Một tham chiếu đến từng phần tử được trỏ đến
}
```

Bây giờ, chúng ta có thể viết một phiên bản user() từ §3.2.4 mà không cần viết một tập hợp các hàm _all :

```
người dùng vô hiệu ()
{
        vector<unique_ptr<Hình>v; trong khi (cin)

        v.push_back(read_shape(cin));
        for_all(v,[](Shape& s){ s.draw(); }); for_all(v,        // draw_all()
        [](Shape& s){ s.rotate(45); });                         // rotate_all(45)
}
```

Tôi chuyển một tham chiếu tới Shape tới lambda để lambda không cần quan tâm chính xác cách các đối tượng được lưu trữ trong vùng chứa. Đặc biệt, những lệnh gọi for_all() đó vẫn hoạt động nếu tôi thay đổi v thành vector<Shape >.

3.4.4 Mẫu đa dạng

Một mẫu có thể được xác định để chấp nhận một số lượng đối số tùy ý thuộc các loại tùy ý. Mẫu như vậy được gọi là mẫu matrixdic. Ví dụ:

```
template<typename T, typename ... Tail> void f(T head,
Tail... tail) {

        g(đầu); //      làm điều gì đó    tới hư ớng
        f(đuôi...); //  thử lại với đuôi
}

khoảng trống f() { }        // không làm gì cả
```

Chìa khóa để thực hiện một mẫu variadic là lưu ý rằng khi bạn chuyển một danh sách các đối số cho nó,

Machine Translated by Google

bạn có thể tách đối số đầu tiên khỏi phần còn lại. Ở đây, chúng ta thực hiện điều gì đó với đối số đầu tiên (phần đầu) và sau đó gọi f() theo cách đệ quy với các đối số còn lại (phần đuôi). Dấu chấm lửng, ..., được dùng để biểu thị ''phần còn lại'' của danh sách. Cuối cùng, tất nhiên, đuôi sẽ trở nên trống rỗng và chúng ta cần một chức năng riêng để giải quyết vấn đề đó.

Chúng ta có thể gọi đây là f() như thế này:

```
int main()
{
      cout << "\n1: ";
      f(1,2.2,"xin chào");

      cout << "\ngiây: "
      f(0,2,'c',"yuck!",0,1,2);
      cout<<"\n";
}
```

Cái này sẽ gọi f(1,2.2,"hello"), cái này sẽ gọi f(2.2,"hello"), cái này sẽ gọi f("hello"), cái này sẽ gọi f(). Cuộc gọi g(head) có thể làm gì? Rõ ràng, trong một chương trình thực tế, nó sẽ làm bất cứ điều gì chúng ta muốn làm cho mỗi đối số. Ví dụ: chúng ta có thể làm cho nó viết đối số của nó (ở đây, phần đầu) thành đầu ra:

```
mẫu<tên chữ T> void g(T x)
{

      cout<<x<<" ";
}
```

Cho rằng, đầu ra sẽ là:

```
đầu tiên: 1 2.2 xin chào
thứ hai: 0,2 c yuck! 0 1 2
```

Có vẻ như f() là một biến thể đơn giản của printf() in các danh sách hoặc giá trị tùy ý – được triển khai trong ba dòng mã cộng với các khai báo xung quanh chúng.

Điểm mạnh của mẫu biến đổi (đôi khi chỉ được gọi là mẫu biến đổi) là chúng có thể chấp nhận bất kỳ đối số nào mà bạn muốn cung cấp cho chúng. Điểm yếu là việc kiểm tra kiểu của giao diện có thể là một chương trình mẫu phức tạp. Để biết chi tiết, xem §28.6. Ví dụ, hãy xem §34.2.4.2 (N-bộ dữ liệu) và Chương 29 (Ma trận N chiều).

3.4.5 Bí danh

Đáng ngạc nhiên là thường rất hữu ích khi giới thiệu một từ đồng nghĩa cho một loại hoặc một mẫu (§6.5). Ví dụ: tiêu đề chuẩn <cstddef> chứa định nghĩa về bí danh size_t, có thể là:

```
sử dụng size_t = int không dấu;
```

Loại thực tế có tên size_t phụ thuộc vào quá trình triển khai, do đó, trong một triển khai khác, size_t có thể là một loại dài không dấu. Có bí danh size_t cho phép lập trình viên viết mã di động.

Loại được tham số hóa cung cấp bí danh cho các loại liên quan đến mẫu của chúng là điều rất phổ biến tranh luận. Ví dụ:

Machine Translated by Google


```
mẫu <typename T> class Vector
{ public: using
value_type
      = T; //
        ...
};
```

Trên thực tế, mọi bộ chứa thư viện tiêu chuẩn đều cung cấp value_type làm tên của loại giá trị của nó (§31.3.1). Điều này cho phép chúng tôi viết mã sẽ hoạt động cho mọi vùng chứa tuân theo quy ước này. Ví dụ:

```
mẫu<typename C> sử dụng
Element_type = typename C::value_type;

template<typename Container> void
algo(Container& c) {

      Vector<Element_type<Container>> vec; //                giữ kết quả ở đây
      // ...
}
```

Cơ chế ràng cưa có thể được sử dụng để xác định một mẫu mới bằng cách ràng buộc một số hoặc tất cả các đối số mẫu. Ví dụ:

```
template<typename Key, typename Value> class Map { //

        ...
};

mẫu<typename Value> sử dụng
String_map = Map<string,Value>;

String_map<int> m; //         tôi là một Bản đồ<string,int>
```

Xem §23.6.

3.5 Lời khuyên

[1] Diễn đạt ý tưởng trực tiếp bằng mã; §3.2.

[2] Định nghĩa các lớp để biểu diễn các khái niệm ứng dụng trực tiếp trong mã; §3.2.

[3] Sử dụng các lớp cụ thể để biểu diễn các khái niệm đơn giản và các thành phần quan trọng về hiệu năng; §3.2.1.

[4] Tránh thao tác tạo mới và xóa ''trần trụi'' ; §3.2.1.2.

[5] Sử dụng tài nguyên xử lý và RAII để quản lý tài nguyên; §3.2.1.2.

[6] Sử dụng các lớp trừu tượng làm giao diện khi cần tách hoàn toàn giao diện và triển khai; §3.2.2.

[7] Sử dụng hệ thống thứ bậc của lớp để biểu diễn các khái niệm với cấu trúc thứ bậc cố hữu; §3.2.4.

Machine Translated by Google

[8] Khi thiết kế hệ thống phân cấp lớp, hãy phân biệt giữa kế thừa triển khai và kế thừa liên mặt; §3.2.4.

[9] Kiểm soát việc xây dựng, sao chép, di chuyển và hủy đối tượng; §3.3.

[10] Trả lại thùng chứa theo giá trị (dựa vào việc di chuyển để đạt hiệu quả); §3.3.2.

[11] Cung cấp sự an toàn tài nguyên mạnh mẽ; nghĩa là, đừng bao giờ làm rò rỉ bất cứ thứ gì mà bạn cho là tài nguyên; §3.3.3.

[12] Sử dụng các thùng chứa, được định nghĩa là các mẫu xử lý tài nguyên, để chứa các tập hợp giá trị cùng loại; §3.4.1.

[13] Sử dụng mẫu hàm để biểu diễn thuật toán chung; §3.4.2.

[14] Sử dụng các đối tượng chức năng, bao gồm cả lambda, để thể hiện các chính sách và hành động; §3.4.3.

[15] Sử dụng bí danh loại và mẫu để cung cấp ký hiệu thống nhất cho các loại có thể khác nhau giữa các các loại tương tự hoặc giữa các triển khai; §3.4.5.

Machine Translated by Google

Trang này cố ý để trống

Machine Translated by Google

4

Tham quan C ++: Bộ chứa và thuật toán

Tại sao lãng phí thời gian
học tập khi sự thiếu hiểu biết là tức thời?
– Hobbes

• Thư viện

Tiêu chuẩn-Tổng quan về thư viện; Tiêu đề thư viện chuẩn và không gian tên • Chuỗi • Luồng I/O

Đầu ra; Đầu vào; I/O của các loại do người dùng định nghĩa •

Vectơ bộ chứa; danh sách; bản đồ; unordered_map; Tổng quan về vùng chứa • Thuật toán

Sử dụng Trình vòng lặp; Các loại trình lặp; Trình lặp luồng; Vị ngữ; Tổng quan thuật toán; Thuật toán

Container • Lời khuyên

4.1 Thư viện

Không có chương trình quan trọng nào được viết chỉ bằng một ngôn ngữ lập trình đơn thuần. Đầu tiên, một bộ thư viện được phát triển. Những điều này sau đó tạo thành cơ sở cho công việc tiếp theo. Hầu hết các chương trình đều rất tẻ nhạt khi viết bằng ngôn ngữ thuần túy, trong khi bất kỳ tác vụ nào cũng có thể được thực hiện đơn giản bằng cách sử dụng các thư viện tốt.

Tiếp tục từ Chương 2 và 3, chương này và chương tiếp theo giới thiệu nhanh về các cơ sở thư viện tiêu chuẩn chính. Tôi cho rằng bạn đã lập trình trước đó. Nếu không, hãy xem xét việc đọc sách giáo khoa, chẳng hạn như Lập trình: Nguyên tắc và Thực hành Sử dụng C++ [Stroustrup,2009], trước khi tiếp tục. Ngay cả khi bạn đã lập trình trước đó, các thư viện bạn đã sử dụng hoặc các ứng dụng bạn đã viết có thể rất khác so với phong cách của C++ được trình bày ở đây. Nếu bạn thấy "chuyến du lịch chớp nhoáng" này khó hiểu, bạn có thể bỏ qua phần trình bày ngôn ngữ từ dưới lên và có hệ thống hơn bắt đầu từ Chương 6. Tương tự, phần mô tả có hệ thống hơn về thư viện tiêu chuẩn bắt đầu từ Chương 30.

Machine Translated by Google

Tôi trình bày rất ngắn gọn các loại thư viện chuẩn hữu ích, chẳng hạn như string, ostream, vector, map (chư ơ ng này), unique_ptr, thread, regex và complex (Chư ơ ng 5), cũng như những cách phổ biến nhất để sử dụng chúng. Làm điều này cho phép tôi đư a ra các ví dụ tốt hơ n trong các chư ơ ng sau. Như trong Chư ơ ng 2 và Chư ơ ng 3, bạn đư ợc khuyến khích mạnh mẽ là không nên bị phân tâm hoặc nản lòng bởi việc hiểu chư a đầy đủ các chi tiết. Mục đích của chư ơ ng này là cho bạn biết những gì sắp xảy ra và truyền đạt những hiểu biết cơ bản về các phư ơ ng tiện thư viện hữu ích nhất.

Đặc tả của thư viện chuẩn gần bằng 2/3 tiêu chuẩn ISO C++. Khám phá nó và thích nó hơ n các lựa chọn thay thế làm tại nhà. Mặc dù phần lớn đã đư ợc đư a vào thiết kế của nó, vẫn còn nhiều hơ n nữa trong quá trình triển khai và nhiều nỗ lực sẽ dành cho việc bảo trì và mở rộng.

Các cơ sở thư viện tiêu chuẩn đư ợc mô tả trong cuốn sách này là một phần của mọi chư ơ ng trình triển khai C++ hoàn chỉnh. Ngoài các thành phần thư viện tiêu chuẩn, hầu hết các triển khai đều cung cấp các hệ thống ''giao diện ngư ời dùng đồ họa'' (GUIs), giao diện Web, giao diện cơ sở dữ liệu, v.v. Tư ơ ng tự, hầu hết các môi trư ờng phát triển ứng dụng đều cung cấp ''thư viện nền tảng'' cho doanh nghiệp hoặc công nghiệp môi trư ờng phát triển và/hoặc thực thi 'chuẩn''. Ở đây, tôi không mô tả các hệ thống và thư viện như vậy. Mục đích là để cung cấp một mô tả độc lập về C++ như đư ợc định nghĩa theo tiêu chuẩn và để giữ cho các ví dụ có thể mang theo đư ợc, trừ khi đư ợc ghi chú cụ thể. Đư ơ ng nhiên, một lập trình viên đư ợc khuyến khích khám phá các tiện ích mở rộng hơ n có sẵn trên hầu hết các hệ thống.

4.1.1 Tổng quan về thư viện tiêu chuẩn

Các phư ơ ng tiện đư ợc cung cấp bởi thư viện tiêu chuẩn có thể đư ợc phân loại như sau: • Hỗ trợ ngôn ngữ thời gian chạy (ví dụ: đối với thông tin về loại thời gian chạy và phân bổ); xem §30.3. • Thư viện chuẩn C (với những sửa đổi rất nhỏ để giảm thiểu vi phạm kiểu hệ thống); xem Chư ơ ng 43.

- Chuỗi và luồng I/O (có hỗ trợ bộ ký tự quốc tế và bản địa hóa); xem Chư ơ ng 36, Chư ơ ng 38 và Chư ơ ng 39. Luồng I/O là một khung mở rộng mà ngư ời dùng có thể thêm luồng, chiến lư ợc đệm và bộ ký tự của riêng họ.
- Khung chứa (chẳng hạn như vectơ và bản đồ) và các thuật toán (chẳng hạn như tìm(), sắp xếp() và hợp nhất()); xem §4.4, §4.5, Chư ơ ng 31-33. Khung này, thư ờng đư ợc gọi là STL [Stepanov,1994], có thể mở rộng để ngư ời dùng có thể thêm các bộ chứa và thuật toán của riêng họ. • Hỗ trợ tính toán số (chẳng hạn như các hàm toán học tiêu chuẩn, số phức, vectơ với các phép toán số học và trình tạo số ngẫu nhiên); xem §3.2.1.1 và Chư ơ ng 40.

• Hỗ trợ đối sánh biểu thức chính quy; xem §5.5 và Chư ơ ng 37. • Hỗ trợ lập trình đồng thời, bao gồm luồng và khóa; xem §5.3 và Chư ơ ng 41.

Hỗ trợ đồng thời là nền tảng để ngư ời dùng có thể thêm hỗ trợ cho các mô hình đồng thời mới dư ới dạng thư viện.

- Các tiện ích hỗ trợ lập trình siêu dữ liệu mẫu (ví dụ: đặc điểm loại; §5.4.2, §28.2.4, §35.4), lập trình chung kiểu STL (ví dụ: cặp; §5.4.3, §34.2.4.1) và lập trình chung (ví dụ: đồng hồ; §5.4.1, §35.2). • ''Con trỏ thông minh'' để quản lý tài nguyên (ví dụ: unique_ptr và shared_ptr; §5.2.1, §34.3) và giao diện cho bộ thu gom rác (§34.5).
- Bộ chứa mục đích đặc biệt, chẳng hạn như mảng (§34.2.1), bitset (§34.2.2) và bộ (§34.2.4.2).

Machine Translated by Google

Các tiêu chí chính để đưa một lớp vào thư viện là: • nó có thể hữu ích cho hầu hết mọi lập trình viên C++ (cả người mới và chuyên gia), • nó có thể được cung cấp ở dạng chung mà không thêm chi phí đáng kể so với phiên bản đơn giản hơn của cùng một cơ sở, và • rằng việc sử dụng đơn giản nên dễ học (liên quan đến mức độ phức tạp vốn có của nhiệm vụ của chúng).

Về cơ bản, thư viện chuẩn C++ cung cấp các cấu trúc dữ liệu cơ bản phổ biến nhất cùng với các thuật toán cơ bản được sử dụng trên chúng.

4.1.2 Tiêu đề thư viện tiêu chuẩn và không gian tên

Mỗi cơ sở thư viện tiêu chuẩn được cung cấp thông qua một số tiêu đề tiêu chuẩn. Ví dụ:

```
#include<chuỗi>
#include<danh sách>
```

Điều này làm cho chuỗi và danh sách tiêu chuẩn có sẵn.

Thư viện tiêu chuẩn được xác định trong một không gian tên (§2.4.2, §14.3.1) được gọi là std. Để sử dụng tiêu chuẩn cơ sở thư viện, tiền tố std:: có thể được sử dụng:

```
std::string s {"Bốn chân Tốt; hai chân Baaad!"};
std::list<std::string> slogans {"Chiến tranh là hòa bình", "Tự do là nô lệ", "Sự ngu dốt là sức mạnh"};
```

Để đơn giản, tôi sẽ hiếm khi sử dụng tiền tố std:: một cách rõ ràng trong các ví dụ. Tôi cũng sẽ không bao giờ #bao gồm các tiêu đề cần thiết một cách rõ ràng. Để biên dịch và chạy các đoạn chương trình tại đây, bạn phải #gộp các tiêu đề thích hợp (như được liệt kê trong §4.4.5, §4.5.5 và §30.2) và làm cho các tên mà chúng khai báo có thể truy cập được. Ví dụ:

```
#include<string> sử dụng không gian tên std;   // làm cho các cơ sở chuỗi tiêu chuẩn có thể truy cập được làm cho tên std có sẵn mà không có tiền tố std::

string s {"C++ là ngôn ngữ lập trình đa năng"}; // OK: chuỗi là std::string
```

Nói chung, việc đổ mọi tên từ một không gian tên vào không gian tên chung là không hợp lý. Tuy nhiên, trong cuốn sách này, tôi hầu như chỉ sử dụng thư viện tiêu chuẩn và thật tốt khi biết nó cung cấp những gì. Vì vậy, tôi không đặt tiền tố cho mỗi lần sử dụng tên thư viện chuẩn với std::. Tôi cũng không bao gồm các tiêu đề thích hợp trong mọi ví dụ. Giả sử rằng đã xong.

Đây là một lựa chọn các tiêu đề thư viện tiêu chuẩn, tất cả đều cung cấp các khai báo trong không gian tên std:

Tiêu đề thư viện chuẩn được chọn (tiếp theo)		§32.2 §iso.25
<thuật toán>	sao chép(), tìm(), sắp xếp()	§34.2.1 §iso.23.3.2
<mảng>	thời	§35.2 §iso.20.11.2 §40.3
<chrono>	lượng mảng, time_point	§iso.26.8 §40.4 §iso.26.8
<cmath>	sqrt(), pow()	§38.2.1 §iso. 27.9.1
<phức tạp>	phức hợp, sqrt(), pow()	§5.3.5 §iso.30.6 §38.1
<dòng chảy>	fstream, ifstream, ofstream future,	§iso.27.4
<tương lai>	hứa hẹn istream,	
<iostream>	ostream, cin, cout	

Machine Translated by Google

Tiêu đề thư viện tiêu chuẩn đư ợc chọn (tiếp theo)		
<bản đồ>	multimap §5.2.1	§31.4.3 §iso.23.4.4 map,
<bộ nhớ>	normal_distribution §iso.20.7 §iso.26.5 unique_ptr, shared_ptr, allocator default_random_engine,	
<ngẫu nhiên>	Chư ơ ng 37 §iso.28.8 regex, smatch Chư ơ ng 36 §iso.21.3 string, basic_string	
<regex>	multiset §38.2.2	§31.4.3 §iso.23.4.6 set,
<chuỗi>	§iso.23.5.4 §35.5 §iso.27.8 istrstream, ostrstream §5.3.1 §iso.30.3 §31.4.3.2	
<bộ>	§iso. 20.1 §31.4 §iso.23.3.6	
<sstream>		
<chủ đề>	chủ đề	
<unordered_map>	unordered_map, unordered_multimap	<tiện ích>
	di chuyển (), hoán đổi (), ghép nối	
<vectơ >	véc tơ	

Danh sách này còn lâu mới hoàn thành; xem §30.2 để biết thêm thông tin.

4.2 Chuỗi

Thư viện chuẩn cung cấp một kiểu chuỗi để bổ sung cho các chuỗi ký tự. Kiểu chuỗi cung cấp nhiều thao tác chuỗi hữu ích, chẳng hạn như phép nối. Ví dụ:

```
chuỗi soạn thảo (chuỗi const & tên, chuỗi const & tên miền) {

      tên trả về + '@' + tên miền;
}

auto addr = Compose("dmr","bell labs.com");
```

Tại đây, addr đư ợc khởi tạo thành chuỗi ký tự dmr@bell labs.com. ''Bổ sung'' của chuỗi có nghĩa là nối. Bạn có thể nối một chuỗi, một chuỗi ký tự, một chuỗi kiểu C hoặc một ký tự thành một chuỗi. Chuỗi tiêu chuẩn có một hàm tạo di chuyển để trả về các chuỗi thậm chí dài theo giá trị là hiệu quả (§3.3.2).

Trong nhiều ứng dụng, hình thức nối phổ biến nhất là thêm một cái gì đó vào cuối của một chuỗi. Điều này đư ợc hỗ trợ trực tiếp bởi thao tác += . Ví dụ:

```
void m2(chuỗi&s1, chuỗi&s2) {

      s1 = s1 + '\n'; //   nối thêm dòng mới
      s2 += '\n'; //       nối thêm dòng mới
}
```

Hai cách thêm vào cuối chuỗi là tư ơ ng đư ơ ng về mặt ngữ nghĩa, như ng tôi thích cách sau hơ n vì nó rõ ràng hơ n về những gì nó thực hiện, ngắn gọn hơ n và có thể hiệu quả hơ n.

Một chuỗi có thể thay đổi. Ngoài = và +=, các phép toán đăng ký (sử dụng []) và chuỗi con đư ợc hỗ trợ. Chuỗi thư viện chuẩn đư ợc mô tả trong Chư ơ ng 36. Trong số các tính năng hữu ích khác, nó cung cấp khả năng thao tác với các chuỗi con. Ví dụ:

Machine Translated by Google


```
tên chuỗi = "Niels Stroustrup";

khoảng trống m3()
{
        chuỗi s = tên.substr(6,10);                    // S = "Hãy mạnh mẽ lên"
        name.replace(0,5,"nicholas"); tên[0] =            tên trở thành "Nicholas Stroustrup" tên trở
        topper(tên[0]);                                // thành "Nicholas Stroustrup"
}
```

Hoạt động substr() trả về một chuỗi là bản sao của chuỗi con được chỉ định bởi các đối số của nó. Đối số đầu tiên là một chỉ mục trong chuỗi (một vị trí) và đối số thứ hai là độ dài của chuỗi con mong muốn. Vì việc lập chỉ mục bắt đầu từ 0, s nhận giá trị Stroustrup.

Thao tác replace() thay thế một chuỗi con bằng một giá trị. Trong trường hợp này, chuỗi con bắt đầu từ 0 với độ dài 5 là Niels; nó được thay thế bởi nicholas. Cuối cùng, tôi thay thế ký tự ban đầu bằng chữ hoa tương đương. Do đó, giá trị cuối cùng của tên là Nicholas Stroustrup. Lưu ý rằng chuỗi thay thế không cần phải có cùng kích thước với chuỗi con mà nó đang thay thế.

Đương nhiên, các chuỗi có thể được so sánh với nhau và với các chuỗi ký tự. Ví dụ:

```
câu thần chú chuỗi;

void reply(const string& answer) {

        if (câu trả lời == câu thần chú) {
                // đục lỗ tôi ảo thuật

        } khác nếu (trả lời == "có") { //
                ...

        } //...
}
```

Thư viện chuỗi được mô tả trong Chương 36. Các kỹ thuật phổ biến nhất để triển khai chuỗi được trình bày trong ví dụ về Chuỗi (§19.3).

4.3 Luồng I/O

Thư viện chuẩn cung cấp đầu vào và đầu ra ký tự được định dạng thông qua thư viện iostream . Các hoạt động đầu vào được nhập và mở rộng để xử lý các loại do người dùng xác định. Phần này là phần giới thiệu rất ngắn gọn về việc sử dụng iostream; Chương 38 là một mô tả khá đầy đủ về cơ sở vật chất của thư viện iostream .

Các hình thức tương tác người dùng khác, chẳng hạn như I/O đồ họa, được xử lý thông qua các thư viện không phải là một phần của tiêu chuẩn ISO và do đó không được mô tả ở đây.

4.3.1 Đầu ra

Thư viện luồng I/O xác định đầu ra cho mọi loại tích hợp sẵn. Hơn nữa, thật dễ dàng để xác định đầu ra của loại do người dùng xác định (§4.3.3). Toán tử << (''put to'') được sử dụng làm toán tử xuất trên các đối tượng của

Machine Translated by Google

gõ ostream; cout là luồng đầu ra tiêu chuẩn và cerr là luồng tiêu chuẩn để báo cáo lỗi.
Theo mặc định, các giá trị ghi vào cout đư ợc chuyển đổi thành một chuỗi ký tự. Ví dụ, để xuất số thập phân 10, chúng ta có thể viết:

```
khoảng trống
f() {
        cout<<10;
}
```

Điều này đặt ký tự 1 theo sau là ký tự 0 trên luồng đầu ra tiêu chuẩn.
Tư ơ ng tự, chúng ta có thể viết:

```
khoảng trống
g() {
        int tôi
        {10}; cout<<i;
}
```

Đầu ra của các loại khác nhau có thể đư ợc kết hợp theo cách rõ ràng:

```
khoảng trống h(int
i) {
        cout << "Gia tri cua i la ";
        cout<<i;
        cout << '\n';
}
```

Đối với h(10), đầu ra sẽ là:

```
giá trị của tôi là 10
```

Mọi ngư ời sẽ sớm mệt mỏi với việc lặp lại tên của luồng đầu ra khi xuất một số mục liên quan.
May mắn thay, bản thân kết quả của một biểu thức đầu ra có thể đư ợc sử dụng cho đầu ra tiếp theo. Ví dụ:

```
khoảng trống h2(int i)
{
        cout << "Gia tri cua i la " << i << '\n';
}
```

H2() này tạo ra kết quả giống như h().
Hằng ký tự là một ký tự đư ợc đặt trong dấu nháy đơ n. Lư u ý rằng một ký tự đư ợc xuất ra dư ới dạng một ký tự chứ không phải là một giá trị số. Ví dụ:

```
khoảng trống
k() {
        int b = 'b';        // lư u ý: char đư ợc chuyển đổi hoàn toàn thành int
        ký tự c = 'c';
        cout << 'a' << b << c;
}
```

Giá trị số nguyên của ký tự 'b' là 98 (trong mã hóa ASCII đư ợc sử dụng trong triển khai C++ mà tôi đã sử dụng), vì vậy, giá trị này sẽ xuất ra a98c.

Machine Translated by Google

4.3.2 Đầu vào

Thư viện tiêu chuẩn cung cấp istream cho đầu vào. Giống như ostream, istream xử lý các biểu diễn chuỗi ký tự của các kiểu dựng sẵn và có thể dễ dàng mở rộng để đối phó với các kiểu do người dùng định nghĩa.

Toán tử >> (''get from'') được sử dụng làm toán tử nhập liệu; cin là luồng đầu vào tiêu chuẩn. Loại toán hạng bên phải của >> xác định đầu vào nào được chấp nhận và mục tiêu của hoạt động đầu vào là gì. Ví dụ:

```
khoảng trống
f() {
        int
        tôi; cin >> tôi;        // đọc số nguyên vào i an

        gấp đôi d;
        cin >> d;               // đọc một số dấu phẩy động có độ chính xác kép thành d
}
```

Thao tác này đọc một số, chẳng hạn như 1234, từ đầu vào tiêu chuẩn vào biến số nguyên i và một số dấu phẩy động, chẳng hạn như 12.34e5, vào biến dấu phẩy động có độ chính xác kép d.

Thông thường, chúng tôi muốn đọc một chuỗi các ký tự. Một cách thuận tiện để làm điều đó là đọc vào một chuỗi. Ví dụ:

```
vô hiệu xin chào ()
{
        cout << "Vui lòng nhập tên\n"; chuỗi str; cin >>
        str; cout <<
        "Xin chao, "
        << str << "!\n";
}
```

Nếu bạn gõ Eric, câu trả lời là:

```
Chào Eric!
```

Theo mặc định, một ký tự khoảng trắng (§7.3.2), chẳng hạn như khoảng trắng, sẽ kết thúc quá trình đọc, vì vậy nếu bạn nhập Eric Bloodaxe giả vờ là vị vua xấu số của York, câu trả lời vẫn là:

```
Chào Eric!
```

Bạn có thể đọc cả một dòng (bao gồm cả ký tự xuống dòng kết thúc) bằng cách sử dụng hàm getline() . Ví dụ:

```
khoảng trống hello_line()
{
        cout << "Vui lòng nhập tên\n"; chuỗi str;

        getline(cin,str); <<
        "Xin chao, "        str << "!\n"; cout <<
}
```

Với chương trình này, đầu vào Eric Bloodaxe mang lại đầu ra mong muốn:

```
Xin chào, Eric Bloodaxe!
```

Machine Translated by Google

Dòng mới kết thúc dòng bị loại bỏ, vì vậy cin đã sẵn sàng cho dòng đầu vào tiếp theo.

Các chuỗi tiêu chuẩn có đặc tính tốt là mở rộng để chứa những gì bạn đặt vào chúng; bạn không cần phải tính toán trư ớc kích thư ớc tối đa. Vì vậy, nếu bạn nhập một vài megabyte dấu chấm phẩy, chư ơ ng trình sẽ lặp lại các trang có dấu chấm phẩy cho bạn.

4.3.3 I/O của các loại do ngư ời dùng xác định

Ngoài I/O của các loại tích hợp sẵn và các chuỗi tiêu chuẩn, thư viện iostream cho phép các lập trình viên xác định I/O cho các loại của riêng họ. Ví dụ: hãy xem xét một Mục nhập loại đơ n giản mà chúng tôi có thể sử dụng để biểu thị các mục nhập trong danh bạ điện thoại:

```
struct Mục
      nhập { tên
      chuỗi; số nguyên;
};
```

Chúng ta có thể định nghĩa một toán tử đầu ra đơ n giản để viết một Mục nhập bằng cách sử dụng định dạng {"name",number} tư ơ ng tự như định dạng mà chúng ta sử dụng để khởi tạo trong mã:

```
toán tử ostream&<<(ostream& os, const Entry& e) {

      return os << "{\"" << e.name << "\", " << e.number << "}";
}
```

Toán tử đầu ra do ngư ời dùng định nghĩa lấy luồng đầu ra của nó (theo tham chiếu) làm đối số đầu tiên và trả về nó làm kết quả. Xem §38.4.2 để biết chi tiết.

Toán tử đầu vào tư ơ ng ứng phức tạp hơ n vì nó phải kiểm tra tính chính xác của phép đối chiếu và xử lý các lỗi:

```
istream& toán tử>>(istream& is, Entry& e)
      // đọc {"tên" ,số } cặp.  Lư u ý: đối với mở với { " " ,Và }
{
      ký tự c,
      c2; if (is>>c && c=='{' && is>>c2 && c2=='"')   sao t với "
            { // tên                    // { giá trị mặc định của một chuỗi là chuỗi: để trống ""
            chuỗi; while (is.get(c) &&  // thứ trư ớc " là một phần của tên
                  c!='"') tên+= c;

            if (is>>c && c==',')
                  { int number =
                  0; if (is>>number>>c && c=='}')   đọc số và một } gán
                        { // e = {tên ,số}; // trả về  nhập mục
                        là;
                  }
            }

      } is.setf(ios_base::failbit);   // đăng ký lỗi trong luồng
      trở lại là;
}
```

Một hoạt động đầu vào trả về một tham chiếu đến istream của nó có thể đư ợc sử dụng để kiểm tra xem hoạt động đó có

Machine Translated by Google

đã thành công. Ví dụ, khi được sử dụng như một điều kiện, is>>c có nghĩa là ''Chúng ta đã đọc thành công từ is thành c chưa?''

Theo mặc định, is >>c bỏ qua khoảng trắng, nhưng is.get(c) thì không, do đó, toán tử đầu vào Entry này bỏ qua (bỏ qua) khoảng trắng bên ngoài chuỗi tên, nhưng không phải bên trong nó. Ví dụ:

```
{ "John Marwood Cleese"      ,123456        }
{"Michael Edward Palin",987654}
```

Chúng ta có thể đọc một cặp giá trị như vậy từ đầu vào vào một Entry như sau:

```
for (Entry ee; cin>>ee; ) // đọc từ cin thành ee
      cout << ee << '\n';    //ee ite to cout wr
```

Đầu ra là:

```
{"John Marwood Cleese", 123456}
{"Michael Edward Palin", 987654}
```

Xem §38.4.1 để biết thêm chi tiết kỹ thuật và kỹ thuật viết toán tử đầu vào cho các loại do người dùng xác định. Xem §5.5 và Chương 37 để biết kỹ thuật có hệ thống hơn để nhận dạng các mẫu trong luồng ký tự (khớp biểu thức chính quy).

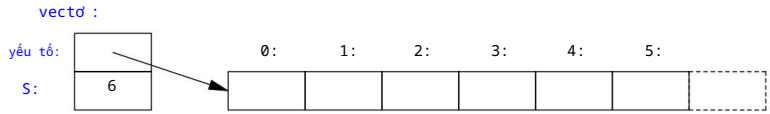
4.4 Vật chứa

Hầu hết các máy tính liên quan đến việc tạo ra các tập hợp giá trị và sau đó thao tác với các tập hợp đó. Đọc các ký tự thành một chuỗi và in ra chuỗi là một ví dụ đơn giản. Một lớp với mục đích chính là chứa các đối tượng thường được gọi là vùng chứa. Cung cấp các thùng chứa phù hợp cho một nhiệm vụ nhất định và hỗ trợ chúng bằng các hoạt động cơ bản hữu ích là các bước quan trọng trong quá trình xây dựng bất kỳ chương trình nào.

Để minh họa các bộ chứa thư viện tiêu chuẩn, hãy xem xét một chương trình đơn giản để lưu giữ tên và số điện thoại. Đây là loại chương trình mà các cách tiếp cận khác nhau có vẻ ''đơn giản và rõ ràng'' đối với những người có hoàn cảnh khác nhau. Lớp Entry từ §4.3.3 có thể được sử dụng để chứa một mục nhập danh bạ điện thoại đơn giản. Ở đây, chúng tôi cố tình bỏ qua nhiều phức tạp trong thế giới thực, chẳng hạn như thực tế là nhiều số điện thoại không có biểu diễn đơn giản dưới dạng int 32 bit .

4.4.1 vectơ

Bộ chứa thư viện tiêu chuẩn hữu ích nhất là vector. Một vectơ là một chuỗi các phần tử của một loại nhất định. Các phần tử được lưu trữ liên tục trong bộ nhớ:



Các ví dụ về vectơ trong §3.2.2 và §3.4 đưa ra ý tưởng về việc triển khai vectơ và §13.6 và §31.4 cung cấp một cuộc thảo luận thấu đáo.

Machine Translated by Google

Chúng ta có thể khởi tạo một vectơ với một tập hợp các giá trị của loại phần tử của nó:

```
vector<Entry> phone_book = {
        {"David Hume",123456},
        {"Karl Popper",234567},
        {"Bertrand Ar thur William Russell",345678}
};
```

Các yếu tố có thể đư ợc truy cập thông qua đăng ký:

```
void print_book(const vector<Entry>& book) {

        for (int i = 0; i!=book.size(); ++i) cout
                << book[i] << '\n';
}
```

Như thư ờng lệ, việc lập chỉ mục bắt đầu từ 0 để cuốn sách[0] giữ mục nhập cho David Hume. Hàm thành viên vectơ size() đư a ra số phần tử.

Các phần tử của vectơ tạo thành một phạm vi, vì vậy chúng ta có thể sử dụng vòng lặp phạm vi (§2.2.5):

```
void print_book(const vector<Entry>& book) {

        for (const auto& x : book) cout          // cho "tự động" §2.2.2 xem
                << x << '\n';
}
```

Khi chúng tôi xác định một vectơ , chúng tôi cung cấp cho nó một kích thư ớc ban đầu (số phần tử ban đầu):

```
vectơ <int> v1 = {1, 2, 3, 4};                  // kích thư ớc là 4
vectơ <chuỗi> v2;                                  kích thư ớc
vector<Hình  > v3(23);                          // Đầ 0 kích thư ớc là 23; giá trị phần tử ban đầu: kích thư ớc
vectơ <đôi> v4(32,9,9);                         // nullptr là 32; giá trị phần tử ban đầu: 9,9
```

Ví dụ: kích thư ớc rõ ràng đư ợc đặt trong dấu ngoặc đơ n thông thư ờng, (23) và theo mặc định, các phần tử đư ợc khởi tạo thành giá trị mặc định của loại phần tử (ví dụ: nullptr cho con trỏ và 0 cho số). Nếu bạn không muốn giá trị mặc định, bạn có thể chỉ định một giá trị làm đối số thứ hai (ví dụ: 9,9 cho 32 phần tử của v4).

Kích thư ớc ban đầu có thể đư ợc thay đổi. Một trong những thao tác hữu ích nhất trên vectơ là push_back(), trong đó thêm một phần tử mới vào cuối vectơ , tăng kích thư ớc của nó lên một. Ví dụ:

```
đầu vào vô hiệu ()
{
        for (Entr ye; cin>>e;)
                phone_book.push_back(e);
}
```

Thao tác này đọc Các mục nhập từ đầu vào tiêu chuẩn vào phone_book cho đến khi đạt đến điểm cuối của đầu vào (ví dụ: cuối tệp) hoặc thao tác nhập gặp phải lỗi định dạng. Vectơ thư viện chuẩn đư ợc triển khai sao cho việc phát triển vectơ bằng cách lặp lại các push_back() là hiệu quả.

Một vectơ có thể đư ợc sao chép trong các phép gán và khởi tạo. Ví dụ:

```
vector<Entry> book2 = phone_book;
```

Machine Translated by Google

Việc sao chép và di chuyển các vectơ được thực hiện bởi các hàm tạo và toán tử gán như được mô tả trong §3.3. Việc gán một vectơ liên quan đến việc sao chép các phần tử của nó. Do đó, sau khi khởi tạo book2, book2 và phone_book giữ các bản sao riêng biệt của mọi Mục nhập trong danh bạ điện thoại. Khi một véc-tơ chứa nhiều phần tử, các phép gán và khởi tạo có vẻ ngây thơ như vậy có thể rất tốn kém. Khi việc sao chép là không mong muốn, nên sử dụng các tham chiếu hoặc con trỏ (§7.2, §7.7) hoặc thao tác di chuyển (§3.3.2, §17.5.2).

4.4.1.1 Các phần tử

Giống như tất cả các bộ chứa thư viện tiêu chuẩn, vectơ là một bộ chứa các phần tử thuộc loại T nào đó, nghĩa là một vectơ <T>. Gần như bất kỳ loại nào đủ điều kiện là một loại phần tử: các loại số dựng sẵn (chẳng hạn như char, int và double), các loại do người dùng xác định (chẳng hạn như chuỗi, Mục nhập, danh sách<int> và Ma trận<double ,2>) và con trỏ (chẳng hạn như const char , Shape và double). Khi bạn chèn một phần tử mới, giá trị của phần tử đó sẽ được sao chép vào vùng chứa. Ví dụ: khi bạn đặt một số nguyên có giá trị 7 vào một vùng chứa, phần tử kết quả thực sự có giá trị 7. Phần tử này không phải là một tham chiếu hoặc một con trỏ tới một số đối tượng chứa 7. Điều này tạo ra các vùng chứa nhỏ gọn đẹp mắt với khả năng truy cập nhanh. Đối với những người quan tâm đến kích thước bộ nhớ và hiệu suất thời gian chạy, điều này rất quan trọng.

4.4.1.2 Kiểm tra phạm vi

Vectơ thư viện chuẩn không đảm bảo kiểm tra phạm vi (§31.2.2). Ví dụ:

```
void người ngắn(vector<Entry>& cuốn sách) {
        int i = book[ph.size()].number; //                    // book.size() nằm ngoài phạm vi
          ...
}
```

Việc khởi tạo đó có khả năng đặt một số giá trị ngẫu nhiên vào i thay vì đưa ra lỗi. Đây là điều không mong muốn và lỗi ngoài phạm vi là một vấn đề phổ biến. Do đó, tôi thường sử dụng một điều chỉnh kiểm tra phạm vi đơn giản của vectơ :

```
mẫu<typename T> class Vec :
public std::vector<T> { public: using
vector<T>::vector; //                    sử dụng các hàm tạo từ vector (dưới tên                    Một vật); nhìn thấy §20.3.5.1
        T& toán tử[](int i) { véc                    // kiểm tra phạm vi
                tơ trả về<T>::at(i); }
        const T& toán tử[](int i) const { véc tơ                    // phạm vi kiểm tra các đối tượng const; §3.2.1.1
                trả về<T>::at(i); }
};
```

Vec kế thừa mọi thứ từ vector ngoại trừ các hoạt động chỉ số con mà nó xác định lại để thực hiện kiểm tra phạm vi. Thao tác at() là một thao tác chỉ số dưới của vectơ đưa ra một ngoại lệ loại out_of_range nếu đối số của nó nằm ngoài phạm vi của vectơ (§2.4.3.1, §31.2.2).

Machine Translated by Google

Đối với Vec, quyền truy cập ngoài phạm vi sẽ đư a ra một ngoại lệ mà ngư ời dùng có thể bắt gặp. Ví dụ:

```
void check(Vec<Entr y>& book) {

      thử {
            sách[book.size()] = {"Joe",999999}; //          // sẽ ném ngoại lệ một
              ...

      } catch (out_of_rang e)
            { cout << "lỗi phạm vi\n";
      }
}
```

Ngoại lệ sẽ đư ợc ném và sau đó bị bắt (§2.4.3.1, Chư ơ ng 13). Nếu ngư ời dùng không bắt đư ợc ngoại lệ, chư ơ ng trình sẽ kết thúc theo cách đư ợc xác định rõ thay vì tiếp tục hoặc thất bại theo cách không xác định. Một cách để giảm thiểu những bất ngờ từ các ngoại lệ chư a đư ợc phát hiện là sử dụng hàm main() với khối y làm phần thân của nó. Ví dụ:

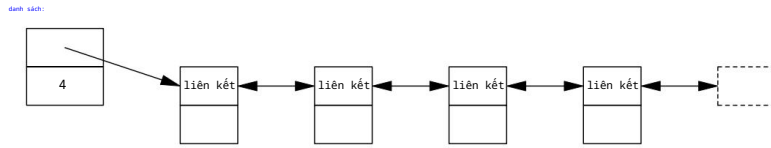
```
int main()
thử {
      // mã của bạn

} catch (out_of_rang e)
      { cerr << "lỗi phạm vi\n";

} catch
      (...) { cerr << "ném ngoại lệ không xác định\n";
}
```

Điều này cung cấp các trình xử lý ngoại lệ mặc định để nếu chúng tôi không bắt đư ợc một số ngoại lệ, thông báo lỗi sẽ đư ợc in trên cerr luồng đầu ra chẩn đoán lỗi tiêu chuẩn (§38.1).

Một số triển khai giúp bạn không phải bận tâm khi xác định Vec (hoặc tư ơ ng đư ơ ng) bằng cách cung cấp phiên bản vectơ đư ợc kiểm tra phạm vi (ví dụ: dư ới dạng tùy chọn trình biên dịch).

4.4.2 danh sách

Thư viện tiêu chuẩn cung cấp một danh sách liên kết kép đư ợc gọi là danh sách:



Chúng tôi sử dụng một danh sách cho các chuỗi mà chúng tôi muốn chèn và xóa các phần tử mà không cần di chuyển các phần tử khác. Chèn và xóa các mục trong danh bạ điện thoại có thể phổ biến, do đó, một danh sách có thể phù hợp để đại diện cho một danh bạ điện thoại đơ n giản. Ví dụ:

```
list<Entr y> phone_book = {
      {"David Hume",123456},
```

Machine Translated by Google


```
        {"Karl Popper",234567},
        {"Bertrand Ar thur William Russell",345678}
    };
```

Khi chúng ta sử dụng một danh sách đư ợc liên kết, chúng ta có xu hư ớng không truy cập các phần tử bằng cách sử dụng chỉ số con theo cách mà chúng ta thư ờng làm đối với các vectơ . Thay vào đó, chúng ta có thể tìm kiếm danh sách để tìm một phần tử có giá trị nhất định. Để làm điều này, chúng ta tận dụng lợi thế của một danh sách là một dãy như đư ợc mô tả trong §4.5:

```
    int get_number(const string& s) {

        for (const auto& x : phone_book) if
            (x.name==s)
                return x.number; trả
        về 0; //      sử dụng 0 để đại diện cho "số không tìm thấy"
    }
```

Quá trình tìm kiếm s bắt đầu ở đầu danh sách và tiếp tục cho đến khi tìm thấy s hoặc đến cuối phone_book .

Đôi khi, chúng ta cần xác định một phần tử trong danh sách. Ví dụ: chúng tôi có thể muốn xóa nó hoặc chèn một mục mới trư ớc nó. Để làm điều đó, chúng tôi sử dụng một trình vòng lặp: một trình lặp danh sách xác định một phần tử của danh sách và có thể đư ợc sử dụng để lặp qua một danh sách (do đó có tên như vậy). Mỗi bộ chứa thư viện tiêu chuẩn cung cấp các hàm bắt đầu() và kết thúc(), trả về một trình vòng lặp tư ơ ng ứng cho phần tử đầu tiên và phần tử cuối cùng (§4.5, §33.1.1). Sử dụng các trình vòng lặp một cách rõ ràng, chúng ta có thể – ít tao nhã hơ n – viết hàm get_number() như sau:

```
    int get_number(const string& s) {

        for (auto p = phone_book.begin(); p!=phone_book.end(); ++p)
            nếu (p >tên==s)
                trả về p >số; trả về
        0; //      sử dụng 0 đến đại diện cho "số không tìm thấy"
    }
```

Trên thực tế, đây gần như là cách trình biên dịch triển khai vòng lặp phạm vi ngắn hơ n và ít bị lỗi hơ n. Cho en một iterator p, p là phần tử mà nó tham chiếu tới, ++p tiến p để tham chiếu tới phần tử tiếp theo, và khi p tham chiếu tới một lớp có phần tử m, thì p >m tư ơ ng đư ơ ng với (buổi chiều.

Việc thêm các phần tử vào danh sách và xóa các phần tử khỏi danh sách thật dễ dàng:

```
    void f(const Entry& ee, list<Entr y>::iterator p, list<Entry>::iterator q) {

        phone_book.insert(p,ee);        // thêm ee trư ớc phần tử đư ợc gọi bởi p
        phone_book.erase(q);            //          phần tử đư ợc gọi bởi q loại bỏ
    }
```

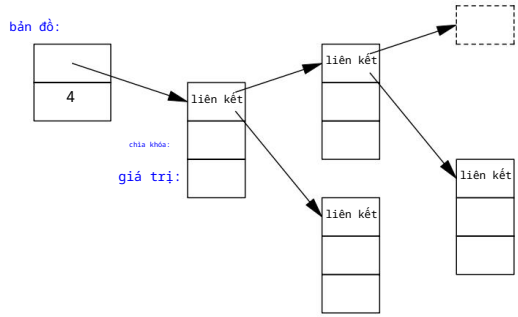
Để biết mô tả đầy đủ hơ n về thao tác chèn() và xóa(), hãy xem §31.3.7.

Các ví dụ về danh sách này có thể đư ợc viết giống hệt nhau bằng cách sử dụng vectơ và (thật đáng ngạc nhiên, trừ khi bạn hiểu kiến trúc máy) hoạt động tốt hơ n với một vectơ nhỏ so với một danh sách nhỏ. Khi tất cả những gì chúng ta muốn là một chuỗi các phần tử, chúng ta có thể lựa chọn giữa việc sử dụng một vectơ và một danh sách. Trừ khi bạn có lý do để không, hãy sử dụng một vectơ . Vectơ hoạt động tốt hơ n khi truyền tải (ví dụ: find() và count()) cũng như để sắp xếp và tìm kiếm (ví dụ: sort () và binary_search()).

Machine Translated by Google

4.4.3 bản đồ

Viết mã để tra cứu tên trong danh sách các cặp (tên, số) khá tẻ nhạt. Ngoài ra, tìm kiếm theo dòng không hiệu quả đối với tất cả trừ các danh sách ngắn nhất. Thư viện tiêu chuẩn cung cấp một cây tìm kiếm (cây đỏ đen) đư ợc gọi là bản đồ:



Trong các ngữ cảnh khác, bản đồ đư ợc gọi là mảng kết hợp hoặc từ điển. Nó đư ợc thực hiện như một cây nhị phân cân bằng.

Bản đồ thư viện chuẩn (§31.4.3) là nơ i chứa các cặp giá trị đư ợc tối ư u hóa để tra cứu. Chúng tôi có thể sử dụng cùng một trình khởi tạo như đối với vectơ và danh sách (§4.4.1, §4.4.2):

```
map<string,int> phone_book {
      {"David Hume",123456},
      {"Karl Popper",234567},
      {"Ber trand Ar thur William Russell",345678}
};
```

Khi đư ợc lập chỉ mục bởi một giá trị của loại đầu tiên (đư ợc gọi là khóa), bản đồ sẽ trả về giá trị tư ơ ng ứng của loại thứ hai (đư ợc gọi là giá trị hoặc loại đư ợc ánh xạ). Ví dụ:

```
int get_number(const string& s) {

      trả lại phone_book[s];
}
```

Nói cách khác, đăng ký bản đồ về cơ bản là tra cứu mà chúng tôi gọi là get_number(). Nếu không tìm thấy khóa , nó sẽ đư ợc nhập vào bản đồ với giá trị mặc định cho giá trị của nó. Giá trị mặc định cho kiểu số nguyên là 0; giá trị tôi vừa chọn đại diện cho một số điện thoại không hợp lệ.

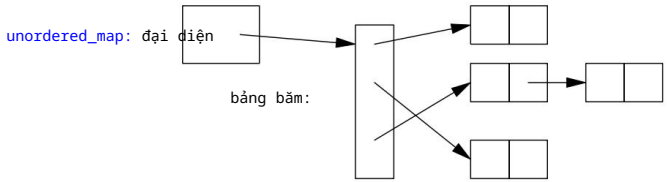
Nếu chúng tôi muốn tránh nhập số không hợp lệ vào danh bạ điện thoại của mình, chúng tôi có thể sử dụng find() và chèn t() thay vì [] (§31.4.3.1).

4.4.4 bản đồ không có thứ tự

Chi phí tra cứu bản đồ là O(log(n)) trong đó n là số phần tử trong bản đồ. Đó là khá tốt. Ví dụ, với một bản đồ có 1.000.000 phần tử, chúng ta chỉ thực hiện khoảng 20 phép so sánh và chỉ dẫn để tìm ra một phần tử. Tuy nhiên, trong nhiều trư ờng hợp, chúng ta có thể làm tốt hơ n bằng cách sử dụng tra cứu đư ợc băm thay vì so sánh bằng hàm sắp xếp, chẳng hạn như <. Thư viện tiêu chuẩn băm

Machine Translated by Google

các thùng chứa đư ợc gọi là ''không có thứ tự'' vì chúng không yêu cầu chức năng đặt hàng:



Ví dụ: chúng ta có thể sử dụng unordered_map từ <unordered_map> cho danh bạ điện thoại của mình:

```
unordered_map<string,int> phone_book {
      {"David Hume",123456},
      {"Karl Popper",234567},
      {"Bertrand Ar thur William Russell",345678}
};
```

Đối với bản đồ, chúng ta có thể đăng ký một unordered_map:

```
int get_number(const string& s) {

      trả lại phone_book[s];
}
```

Thư viện chuẩn unordered_map cung cấp hàm băm mặc định cho các chuỗi. Nếu cần, bạn có thể cung cấp (§31.4.3.4) của riêng mình.

4.4.5 Tổng quan về vùng chứa

Thư viện tiêu chuẩn cung cấp một số loại vùng chứa chung và hữu ích nhất để cho phép ngư ời lập trình chuyên nghiệp chọn vùng chứa phục vụ tốt nhất cho nhu cầu của ứng dụng:

Tóm tắt vùng chứa tiêu chuẩn	
vectơ <T>	Một vectơ có kích thư ớc thay đổi (§31.4)
danh sách<T>	Danh sách liên kết kép (§31.4.2)
forward_list<T>	Danh sách liên kết đơ n (§31.4.2)
deque<T>	Hàng đợi hai đầu (§31.2)
đặt<T>	Một bộ (§31.4.3)
nhiều bộ<T>	Một tập hợp trong đó một giá trị có thể xuất hiện nhiều lần (§31.4.3)
map<K,V>	Mảng kết hợp (§31.4.3)
multimap<K,V>	Bản đồ trong đó một khóa có thể xuất hiện nhiều lần (§31.4.3)
unordered_map<K,V>	Bản đồ sử dụng tra cứu đư ợc băm (§31.4.3.2)
unordered_multimap<K,V>	Multimap sử dụng tra cứu băm (§31.4.3.2)
unordered_set<T>	Một bộ sử dụng tra cứu băm (§31.4.3.2)
Một bộ nhiều sử dụng tra cứu băm (§31.4.3.2)	unordered_multiset<T>

Các vùng chứa không có thứ tự đư ợc tối ư u hóa để tra cứu bằng một khóa (thư ờng là một chuỗi); nói cách khác, chúng đư ợc thực hiện bằng cách sử dụng bảng băm.

Machine Translated by Google

Các thùng chứa tiêu chuẩn đư ợc mô tả trong §31.4. Các vùng chứa đư ợc xác định trong không gian tên std và đư ợc trình bày trong các tiêu đề <vector>, <list>, <map>, v.v. (§4.1.2, §30.2). Ngoài ra, thư viện chuẩn cung cấp bộ điều hợp bộ chứa queue<T> (§31.5.2), stack<T> (§31.5.1), deque<T> (§31.4) và priority_queue<T> (§31.5.3). Thư viện tiêu chuẩn cũng cung cấp các loại giống như vùng chứa chuyên biệt hơ n, chẳng hạn như mảng có kích thư ớc cố định array <T,N> (§34.2.1) và bitset<N> (§34.2.2).

Các thùng chứa tiêu chuẩn và các hoạt động cơ bản của chúng đư ợc thiết kế giống nhau từ quan điểm ký hiệu. Hơ n nữa, ý nghĩa của các hoạt động là tư ơng đư ơng với các bộ chứa khác nhau. Các hoạt động cơ bản áp dụng cho mọi loại vùng chứa mà chúng có ý nghĩa và có thể đư ợc triển khai hiệu quả. Ví dụ: • begin() và end() lần lư ợt cung cấp các trình vòng lặp cho phần tử đầu tiên và phần tử cuối cùng. • push_back() có thể đư ợc

sử dụng (hiệu quả) để thêm các phần tử vào cuối vectơ , danh sách chuyển tiếp, danh sách và các vùng chứa khác.

• size() trả về số phần tử.

Tính đồng nhất về ký hiệu và ngữ nghĩa này cho phép các lập trình viên cung cấp các loại vùng chứa mới có thể đư ợc sử dụng theo cách rất giống với các loại tiêu chuẩn. Vectơ đư ợc kiểm tra phạm vi, Vector (§2.3.2, §2.4.3.1), là một ví dụ về điều đó. Tính đồng nhất của các giao diện vùng chứa cũng cho phép chúng tôi chỉ định các thuật toán độc lập với các loại vùng chứa riêng lẻ. Tuy nhiên, mỗi cái đều có điểm mạnh và điểm yếu. Ví dụ: đăng ký và duyệt một vectơ là rẻ và dễ dàng. Mặt khác, các phần tử vectơ đư ợc di chuyển khi chúng ta chèn hoặc xóa các phần tử; list có các thuộc tính hoàn toàn ngư ợc lại. Xin lư u ý rằng một vectơ thư ờng hiệu quả hơ n một danh sách cho các chuỗi ngắn gồm các phần tử nhỏ (ngay cả đối với thao tác chèn() và xóa()). Tôi đề xuất vectơ thư viện chuẩn làm loại mặc định cho chuỗi phần tử: bạn cần có lý do để chọn loại khác.

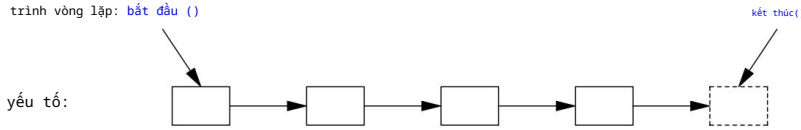
4.5 Thuật toán

Một cấu trúc dữ liệu, chẳng hạn như một danh sách hoặc một vectơ , không hữu ích cho lắm. Để sử dụng một, chúng ta cần các thao tác để truy cập cơ bản như thêm và xóa các phần tử (như đư ợc cung cấp cho danh sách và vectơ). Hơ n nữa, chúng tôi hiếm khi chỉ lư u trữ các đối tư ợng trong một thùng chứa. Chúng tôi sắp xếp chúng, in chúng, trích xuất các tập hợp con, xóa phần tử, tìm kiếm đối tư ợng, v.v. Do đó, thư viện chuẩn cung cấp các thuật toán phổ biến nhất cho vùng chứa ngoài việc cung cấp các loại vùng chứa phổ biến nhất. Ví dụ: phần sau đây sắp xếp một vectơ và đặt một bản sao của từng phần tử vectơ duy nhất vào một danh sách:

```
toán tử bool<(const Entry& x, const Entry& y) {                    // ít hơ n

        trả về x.name<y.name;                    // đặt hàng Bài viết theo tên của họ
}

void f(vector<Entry>& vec, list<Entry>& lst) {

        sắp xếp(vec.begin(),vec.end());                                  sử dụng để đặt hàng
        unique_copy(vec.begin(),vec.end(),lst.begin());                  // 19 không sao chép các phần tử bằng nhau liền kề
}
```

Các thuật toán tiêu chuẩn đư ợc mô tả trong Chư ơng 32. Chúng đư ợc thể hiện dư ới dạng trình tự các phần tử. Một chuỗi đư ợc biểu diễn bằng một cặp trình vòng lặp chỉ định phần tử đầu tiên và phần tử tiếp theo:

Machine Translated by Google



Trong ví dụ này, sor t() sắp xếp trình tự đư ợc xác định bởi cặp vòng lặp vec.begin() và vec.end() – đây chỉ là tất cả các phần tử của một vectơ . Để viết (đầu ra), bạn chỉ cần chỉ định phần tử đầu tiên đư ợc viết. Nếu nhiều hơ n một phần tử đư ợc viết, các phần tử theo sau phần tử ban đầu đó sẽ bị ghi đè. Do đó, để tránh lỗi, lst phải có ít nhất nhiều phần tử bằng số giá trị duy nhất trong vec.

Nếu chúng ta muốn đặt các phần tử duy nhất vào một vùng chứa mới, chúng ta có thể viết:

```
list<Entr y> f(vector<Entr y>& vec) {

      danh sách<Entr y>
      res; sor t(vec.begin(),vec.end());
      unique_copy(vec.begin(),vec.end(),back_inser ter(res)); // trả lại    thêm vào res
      độ phân giải;
}
```

Back_inser ter() thêm các phần tử vào cuối vùng chứa, mở rộng vùng chứa để như ờng chỗ cho chúng (§33.2.2). Do đó, các bộ chứa tiêu chuẩn cộng với back_inser ter()s loại bỏ nhu cầu sử dụng quản lý bộ nhớ kiểu C rõ ràng, dễ bị lỗi bằng cách sử dụng realloc() (§31.5.1). Danh sách thư viện chuẩn có một hàm tạo di chuyển (§3.3.2, §17.5.2) giúp trả về độ phân giải theo giá trị hiệu quả (ngay cả đối với danh sách hàng nghìn phần tử).

Nếu bạn thấy kiểu mã cặp vòng lặp, chẳng hạn như sor t(vec.begin(),vec.end()), tẻ nhạt, bạn có thể xác định các phiên bản vùng chứa của thuật toán và viết sor t(vec) (§ 4.5.6).

4.5.1 Sử dụng Trình vòng lặp

Khi bạn gặp một vùng chứa lần đầu tiên, bạn có thể nhận đư ợc một vài trình vòng lặp đề cập đến các phần tử hữu ích; begin() và end() là những ví dụ tốt nhất về điều này. Ngoài ra, nhiều thuật toán trả về các trình vòng lặp. Ví dụ: thuật toán find tiêu chuẩn tìm kiếm một giá trị trong một chuỗi và trả về một trình vòng lặp cho phần tử đư ợc tìm thấy:

```
bool has_c(const string& s, char c) {        // không   chứa ký tự c?

      auto p = find(s.begin(),s.end(),c);
      if (p!=s.end())
            trả về true;
      khác
            trả về sai;
}
```

Giống như nhiều thuật toán tìm kiếm trong thư viện tiêu chuẩn, find trả về end() để biểu thị ''không tìm thấy.'' Một định nghĩa tư ơ ng đư ơ ng, ngắn hơ n, của has_c() là:

Machine Translated by Google


```
bool has_c(const string& s, char c) {              // không   chứa ký tự c?

        trả về find(s.begin(),s.end(),c)!=s.end();
}
```

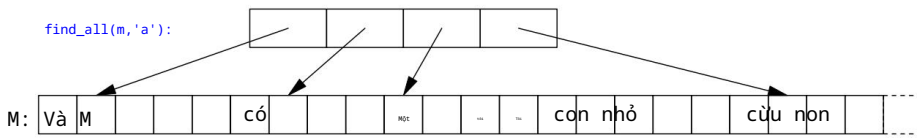
Một bài tập thú vị hơ n là tìm vị trí của tất cả các lần xuất hiện của một ký tự trong một chuỗi. Chúng ta có thể trả về tập hợp các lần xuất hiện dư ới dạng vectơ của các trình lặp chuỗi . Trả về một véc-tơ hiệu quả vì véc-tơ cung cấp ngữ nghĩa di chuyển (§3.3.1). Giả sử rằng chúng tôi muốn sửa đổi các vị trí đư ợc tìm thấy, chúng tôi chuyển một chuỗi không phải const :

```
vector<string::iterator> find_all(string& s, char c) {                // tìm tất cả các lần xuất hiện  của c trong s

        vectơ <chuỗi::iterator> độ phân
        giải; for (auto p = s.begin(); p!=s.end(); ++p)
                if (  p==c)
                        res.push_back(p); trả
        lại độ phân giải;
}
```

Chúng tôi lặp qua chuỗi bằng cách sử dụng một vòng lặp thông thư ờng, di chuyển trình lặp p về phía trư ớc một phần tử tại một thời điểm bằng cách sử dụng ++ và xem xét các phần tử bằng cách sử dụng toán tử quy ư ớc . Chúng ta có thể kiểm tra find_all() như thế này:

```
kiểm tra khoảng trống

() {
        string m {"Mary had a little lamb"}; for
        (auto p : find_all(m,'a')) if
                (  p!='a')
                        cerr << "a bug!\n";
}
```

Cuộc gọi find_all() đó có thể đư ợc biểu diễn bằng đồ họa như sau:



Trình vòng lặp và thuật toán tiêu chuẩn hoạt động như nhau trên mọi vùng chứa tiêu chuẩn mà việc sử dụng chúng có ý nghĩa. Do đó, chúng ta có thể khái quát hóa find_all():

```
template<typename C, typename V>
vector<typename C::iterator> find_all(C& c, V v) {                // tìm tất cả các lần xuất hiện  của v trong c

        véc-tơ <tên kiểu chữ C::iterator> res; for
        (auto p = c.begin(); p!=c.end(); ++p) if (  p==v)

                res.push_back(p); trả lại độ
        phân giải;
}
```

Machine Translated by Google

Tên kiểu là cần thiết để thông báo cho trình biên dịch rằng trình vòng lặp của C đư ợc coi là một kiểu chứ không phải là một giá trị của kiểu nào đó, ví dụ, số nguyên 7. Chúng ta có thể ẩn chi tiết triển khai này bằng cách giới thiệu một bí danh kiểu (§3.4.5) cho Iterator :

```
mẫu<typename T> sử dụng
Iterator<T> = typename T::iterator;

template<typename C, typename V> vector<Iterator<C>>
find_all(C& c, V v) {                                   // tìm tất cả các lần xuất hiện của v trong c

        vector<Iterator<C>> res; for
        (auto p = c.begin(); p!=c.end(); ++p) if (  p==v)

                res.push_back(p); trả lại độ
        phân giải;
}
```

Bây giờ chúng ta có thể viết:

```
kiểm tra khoảng trống
() {
        string m {"Mary had a little lamb"}; for (auto
        p : find_all(m,'a')) if (  p!='a')              // p là một str ing::iterator
                cerr <<
                        "chuỗi lỗi!\n";

        danh sách<kép> ld {1.1, 2.2, 3.3, 1.1}; for
        (auto p : find_all(ld,1.1)) if (  p!
                =1.1) cerr <<
                        "list bug!\n";

        vector<string> vs { "đỏ", "lam", "xanh", "xanh", "cam", "xanh" }; for (auto p :
        find_all(vs,"green")) if (  p!="green") cerr
                << "lỗi vector!\n";


        for (auto p : find_all(vs,"green"))   p =
                "ver t";
}
```

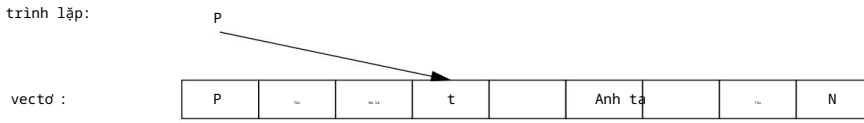
Trình vòng lặp đư ợc sử dụng để phân tách thuật toán và vùng chứa. Một thuật toán hoạt động trên dữ liệu của nó thông qua các trình vòng lặp và không biết gì về vùng chứa chứa các phần tử. Ngư ợc lại, một vùng chứa không biết gì về các thuật toán hoạt động trên các phần tử của nó; tất cả những gì nó làm là cung cấp các trình vòng lặp theo yêu cầu (ví dụ: bắt đầu() và kết thúc()). Mô hình tách biệt giữa lư u trữ dữ liệu và thuật toán này mang lại phần mềm rất tổng quát và linh hoạt.

4.5.2 Các loại trình vòng lặp

Iterator thực sự là gì? Bất kỳ trình vòng lặp cụ thể nào cũng là một đối tư ợng thuộc một số loại. Tuy nhiên, có nhiều loại trình vòng lặp khác nhau, bởi vì một trình vòng lặp cần chứa thông tin cần thiết để thực hiện

Machine Translated by Google

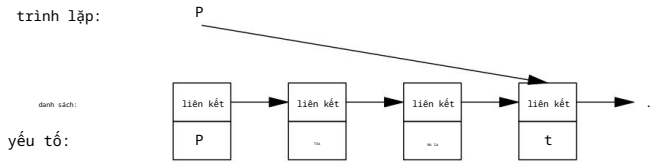
công việc của nó cho một loại container cụ thể. Các loại trình vòng lặp này có thể khác nhau tùy theo vùng chứa và các nhu cầu chuyên biệt mà chúng phục vụ. Ví dụ: trình vòng lặp của vectơ có thể là một con trỏ thông thư ờng, bởi vì con trỏ là một cách khá hợp lý để tham chiếu đến một phần tử của vectơ :



Ngoài ra, một trình lặp vectơ có thể đư ợc triển khai dư ới dạng một con trỏ tới vectơ cộng với một chỉ mục:

Sử dụng trình vòng lặp như vậy sẽ cho phép kiểm tra phạm vi.

Trình lặp danh sách phải phức tạp hơ n một con trỏ đơ n giản tới một phần tử vì phần tử của danh sách nói chung không biết phần tử tiếp theo của danh sách đó ở đâu. Do đó, một trình lặp danh sách có thể là một con trỏ tới một liên kết:



Điều chung cho tất cả các trình vòng lặp là ngữ nghĩa của chúng và cách đặt tên cho các hoạt động của chúng. Ví dụ: áp dụng ++ cho bất kỳ trình vòng lặp nào sẽ tạo ra một trình vòng lặp tham chiếu đến phần tử tiếp theo. Tư ơng tự, mang lại phần tử mà trình vòng lặp đề cập đến. Trên thực tế, bất kỳ đối tư ợng nào tuân theo một vài quy tắc đơ n giản như thế này đều là một trình vòng lặp (§33.1.4). Hơ n nữa, ngư ời dùng hiếm khi cần biết loại trình vòng lặp cụ thể; mỗi vùng chứa ''biết'' các loại trình vòng lặp của nó và cung cấp chúng dư ới các tên quy ư ớc itera tor và const_iterator. Ví dụ: list<Entr y>::iterator là loại trình vòng lặp chung cho list<Entr y>.
Chúng tôi hiếm khi phải lo lắng về các chi tiết về cách xác định loại đó.

4.5.3 Trình lặp luồng

Trình vòng lặp là một khái niệm chung và hữu ích để xử lý các chuỗi phần tử trong vùng chứa.
Tuy nhiên, các thùng chứa không phải là nơ i duy nhất chúng ta tìm thấy các chuỗi phần tử. Ví dụ: một luồng đầu vào tạo ra một chuỗi giá trị và chúng tôi ghi một chuỗi giá trị vào luồng đầu ra.
Do đó, khái niệm về trình vòng lặp có thể đư ợc áp dụng hữu ích cho đầu vào và đầu ra.

Để tạo một ostream_iterator, chúng ta cần chỉ định luồng nào sẽ đư ợc sử dụng và loại các đối tư ợng đư ợc ghi vào nó. Ví dụ:

Machine Translated by Google


```
ostream_iterator<string> oo {cout};                    // viết chuỗi vào cout
```

Tác dụng của việc gán cho oo là ghi giá trị đư ợc gán vào cout. Ví dụ:

```
int main() {

        có = "Xin chào,        // nghĩa là cout<<"Xin chào,"
      "; +
      +có;   yes = "thế giới!\n"; // nghĩa là cout<<"wor ld!\n"
}
```

Đây là một cách khác để viết thông báo chính tắc thành đầu ra tiêu chuẩn. ++ oo đư ợc thực hiện để bắt chư ớc việc ghi vào một mảng thông qua một con trỏ.

Tư ơ ng tự, istream_iterator là thứ cho phép chúng ta coi luồng đầu vào là luồng chỉ đọc thùng đựng hàng. Một lần nữa, chúng ta phải chỉ định luồng sẽ đư ợc sử dụng và loại giá trị dự kiến:

```
istream_iterator<string> ii {cin};
```

Trình lặp đầu vào đư ợc sử dụng theo cặp đại diện cho một chuỗi, vì vậy chúng tôi phải cung cấp istream_iterator để chỉ ra phần cuối của đầu vào. Đây là istream_iterator mặc định:

```
istream_iterator<string> eos {};
```

Thông thư ờng, istream_iterators và ostream_iterators không đư ợc sử dụng trực tiếp. Thay vào đó, chúng đư ợc cung cấp dư ới dạng đối số cho các thuật toán. Ví dụ: chúng ta có thể viết một chư ơ ng trình đơ n giản để đọc một tệp, sắp xếp các từ đã đọc, loại bỏ các từ trùng lặp và ghi kết quả vào một tệp khác:

```
int main() {

      chuỗi từ, đến; cin
      >> từ >> đến;                          // lấy nguồn đích cin và tên tập

      ifstream là {từ};                        luồng đầu vào cho tệp "từ" cho
      istream_iterator<string> ii {is};      // luồng đầu vào của trình lặp
      istream_iterator<string> eos {};       // đầu vào

      ofstream os{to};                       // luồng đầu ra tệp "đến" cho trình
      ostream_iterator<string> oo {os,"\n"}; //   lặp đầu ra cho luồng

      vectơ <chuỗi> b {ii,eos}; sắp          // vào đư ợc khởi tạo theo vectơ from [ii:eos) b là đầu
      xếp(b.begin(),b.end());                // bia bộ đệm

      unique_copy(b.begin(),b.end(),oo);     // chép bộ đệm để xuất, loại bỏ các giá trị đư ợc sao

      trả về !is.eof() || !os;               // trả lại trạng thái lỗi (§2.2.1, §38.3)
}
```

Ifstream là istream có thể đư ợc đính kèm vào tệp và ofstream là ostream có thể đư ợc đính kèm vào tệp. Đối số thứ hai của ostream_iterator đư ợc sử dụng để phân định các giá trị đầu ra.

Trên thực tế, chư ơ ng trình này dài hơ n mức cần thiết. Chúng tôi đọc các chuỗi thành một vectơ , sau đó chúng tôi sắp xếp () chúng, sau đó chúng tôi viết chúng ra, loại bỏ các bản sao. Một giải pháp tao nhã hơ n là không

Machine Translated by Google

lư u trữ trùng lặp ở tất cả. Điều này có thể đư ợc thực hiện bằng cách giữ các chuỗi trong một tập hợp, không giữ các bản sao trùng lặp và giữ các phần tử của nó theo thứ tự (§31.4.3). Bằng cách đó, chúng ta có thể thay thế hai dòng sử dụng vectơ bằng một dòng sử dụng tập hợp và thay thế unique_copy() bằng bản sao() đơ n giản hơ n:

```
set<string> b {ii,eos}; sao                    // thu thập các chuỗi từ bộ đệm đầu
chép(b.begin(),b.end(),oo);                    // ra vào để sao chép đầu
```

Chúng tôi chỉ sử dụng tên ii, eos và oo một lần, vì vậy chúng tôi có thể tiếp tục giảm kích thư ớc của chư ơ ng trình:

```
int main() {

    chuỗi từ, đến; cin
    >> từ >> đến;                          // lấy đích và tên nguồn tệp

    ifstream là {từ};                      // luồng đầu vào cho tệp "từ" luồng
    ofstream os {đến};                     // đầu ra cho tệp "đến"

    set<string> b {istream_iterator<string>{is},istream_iterator<string>{}}; // đọc đầu vào
    copy(b.begin(),b.end(),ostream_iterator<string>{os,"\n"});               // đầu ra để sao chép

    trả về !is.eof() || !os;               // trả về trạng thái lỗi (§2.2.1, §38.3)
}
```

Việc đơ n giản hóa cuối cùng này có cải thiện khả năng đọc hay không là vấn đề của sở thích và kinh nghiệm.

4.5.4 Vị từ

Trong các ví dụ trên, các thuật toán chỉ đơ n giản là ''đư ợc tích hợp sẵn'' hành động đư ợc thực hiện cho mỗi phần tử của một chuỗi. Tuy nhiên, chúng tôi thư ờng muốn biến hành động đó thành một tham số cho thuật toán. Ví dụ: thuật toán tìm kiếm (§32.4) cung cấp một cách thuận tiện để tìm kiếm một giá trị cụ thể. Một biến thể tổng quát hơ n tìm kiếm một phần tử đáp ứng một yêu cầu cụ thể, một vị từ (§3.4.2).
Ví dụ: chúng tôi có thể muốn tìm kiếm bản đồ cho giá trị đầu tiên lớn hơ n 42. Bản đồ cho phép chúng tôi truy cập các phần tử của nó dư ới dạng một chuỗi các cặp (khóa, giá trị) , vì vậy chúng tôi có thể tìm kiếm bản đồ<string,int>'s chuỗi cho một cặp<const string,int> trong đó int lớn hơ n 42:

```
void f(map<string,int>& m) {

    auto p = find_if(m.begin(),m.end(),Greater_than{42}); //
      ...
}
```

Ở đây, Greater_than là một đối tư ợng hàm (§3.4.3) giữ giá trị (42) để so sánh với:

```
cấu trúc lớn hơ n { int val;

    Greater_than(int v) : val{v} { } toán
    tử bool()(const pair<string,int>& r) { return r.second>val; }
};
```

Ngoài ra, chúng ta có thể sử dụng biểu thức lambda (§3.4.3):

```
int cxx = count_if(m.begin(), m.end(), [](const pair<string,int>& r) { return r.second>42; });
```

Machine Translated by Google

4.5.5 Tổng quan về thuật toán

Một định nghĩa chung về thuật toán là ''một tập hợp hữu hạn các quy tắc đư a ra một chuỗi các thao tác để giải quyết một tập hợp cụ thể các vấn đề [và] có năm đặc điểm quan trọng: Tính hữu hạn ... Tính xác định ... Đầu vào... Đầu ra... Hiệu quả'' [Knuth,1968,§1.1]. Trong ngữ cảnh của thư viện chuẩn C++, một thuật toán là một mẫu hàm hoạt động trên các chuỗi phần tử.

Thư viện tiêu chuẩn cung cấp hàng chục thuật toán. Các thuật toán đư ợc xác định trong không gian tên std và đư ợc trình bày trong tiêu đề <algorithm> . Tất cả các thuật toán thư viện chuẩn này đều lấy các chuỗi làm đầu vào (§4.5). Chuỗi nửa mở từ b đến e đư ợc gọi là [b:e). Dư ới đây là một vài điều tôi thấy đặc biệt hữu ích:

Các thuật toán tiêu chuẩn đã chọn	
p=find(b,e,x)	p là p đầu tiên trong [b:e) sao cho
p=find_if(b,e,f)	p==xp là p đầu tiên trong [b:e) sao cho
n=count(b,e,x)	f(p)==true n là số phần tử q trong [b: e) sao cho q==xn
n=count_if(b,e,f)	là số phần tử q trong [b:e) sao cho f(q,x)
replace(b,e,v,v2)	Thay các phần tử q trong [b:e) sao cho q==v bởi v2
replace_if(b,e,f,v2)	Thay các phần tử q trong [b:e) sao cho f(q) bởi v2
p=copy(b,e ,out)	Sao chép [b:e) sang [ra:p)
p=copy_if(b,e ,out,f)	Sao chép các phần tử q từ [b:e) sao cho f(q) sang [out:p)
p=unique_copy(b,e ,out)	Sao chép [b:e) sang [ra:p); không sao chép các bản sao liền kề
sort(b,e)	Sắp xếp các phần tử của [b:e) sử dụng < làm tiêu chí sắp xếp
sort(b,e,f)	Sắp xếp các phần tử của [b:e) sử dụng f làm tiêu chí sắp xếp
(p1,p2)=equal_range(b,e,v)	[p1:p2) là dãy con của dãy [b:e) đã sắp xếp có giá trị v; về cơ bản là tìm kiếm nhị phân cho v
p=hợp nhất(b,e ,b2,e2,ra)	Hợp nhất hai chuỗi đã sắp xếp [b:e) và [b2:e2) thành [out:p)

Các thuật toán này và nhiều thuật toán khác (xem Chư ơ ng 32), có thể đư ợc áp dụng cho các phần tử của vùng chứa, chuỗi và mảng dựng sẵn.

4.5.6 Thuật toán vùng chứa

Một trình tự đư ợc xác định bởi một cặp vòng lặp [bắt đầu: kết thúc). Điều này là chung và linh hoạt, như ng thông thư ờng nhất, chúng tôi áp dụng thuật toán cho một chuỗi là nội dung của vùng chứa. Ví dụ:

```
sắp xếp(v.begin(),v.end());
```

Tại sao chúng ta không chỉ nói sắp xếp (v)? Chúng tôi có thể dễ dàng cung cấp tốc ký đó:

```
không gian tên Estd
      { sử dụng không gian tên std;

      mẫu<lớp C> sắp xếp
      void(C& c) {

            sắp xếp(c.begin(),c.end());
      }
```

Machine Translated by Google


```
        template<class C, class Pred> void
        sort(C& c, Pred p) {

              sắp xếp(c.begin(),c.end(),p);
        }

        // ...
    }
```

Tôi đặt các phiên bản vùng chứa của sort() (và các thuật toán khác) vào không gian tên Estd (''extended std'') của riêng chúng để tránh can thiệp vào việc sử dụng không gian tên std của các lập trình viên khác.

4.6 Lời khuyên

[1] Đừng phát minh lại bánh xe; sử dụng thư viện; §4.1.

[2] Khi bạn có lựa chọn, hãy ư u tiên thư viện tiêu chuẩn hơ n các thư viện khác; §4.1.

[3] Đừng nghĩ rằng thư viện tiêu chuẩn là lý tư ởng cho mọi thứ; §4.1.

[4] Hãy nhớ #gộp các tiêu đề cho các cơ sở bạn sử dụng; §4.1.2.

[5] Hãy nhớ rằng các cơ sở thư viện tiêu chuẩn đư ợc định nghĩa trong namespace std; §4.1.2.

[6] Thích chuỗi hơ n chuỗi kiểu C (a char ; §2.2.5); §4.2, §4.3.2. [7] iostream là loại nhạy cảm, loại an toàn và có thể mở rộng; §4.3.

[8] Thích vector<T>, map<K,T> và unordered_map<K,T> hơ n T[]; §4.4.

[9] Biết các thùng chứa tiêu chuẩn của bạn và sự đánh đổi của chúng; §4.4.

[10] Sử dụng vectơ làm vùng chứa mặc định của bạn; §4.4.1.

[11] Ư u tiên cấu trúc dữ liệu nhỏ gọn; §4.4.1.1.

[12] Nếu còn nghi ngờ, hãy sử dụng vectơ đư ợc kiểm tra phạm vi (chẳng hạn như Vec); §4.4.1.2.

[13] Sử dụng push_back() hoặc back_inser ter() để thêm phần tử vào vùng chứa; §4.4.1, §4.5.

[14] Sử dụng push_back() trên vectơ thay vì realloc() trên mảng; §4.5.

[15] Nắm bắt các ngoại lệ phổ biến trong hàm main(); §4.4.1.2.

[16] Biết các thuật toán tiêu chuẩn của bạn và thích chúng hơ n các vòng lặp viết tay; §4.5.5.

[17] Nếu việc sử dụng trình vòng lặp trở nên tẻ nhạt, hãy xác định thuật toán vùng chứa; §4.5.6.

Machine Translated by Google

5

Tham quan C ++: Đồng thời và Tiện ích

Khi bạn muốn hư ớng dẫn, hãy
ngắn gọn.
– Cicero

• Giới thiệu •
Quản lý tài nguyên unique_ptr
và shared_ptr • Đồng thời

Nhiệm vụ và chủ đề; Vư ợt qua đối số; Trả Kết quả; Chia Sẻ Dữ Liệu; giao tiếp
Nhiệm
vụ • Các thành phần tiện ích nhỏ
Thời gian; Loại chức năng; cặp và bộ •
Biểu thức chính quy • Toán
học
Các hàm và thuật toán toán học; Số phức; Số ngẫu nhiên; véc tơ
Môn số học; Giới hạn số • Lời
khuyên

5.1 Giới thiệu

Từ quan điểm của ngư ời dùng cuối, thư viện tiêu chuẩn lý tư ởng sẽ cung cấp các thành phần hỗ trợ trực tiếp cho mọi nhu cầu về cơ bản. Đối với một miền ứng dụng give en, một thư viện thư ơ ng mại khổng lồ có thể đạt đến mức lý tư ởng đó. Tuy nhiên, đó không phải là điều mà thư viện chuẩn C++ đang cố gắng thực hiện. Một thư viện dễ quản lý, có sẵn trên toàn cầu không thể là tất cả đối với mọi ngư ời. Thay vào đó, thư viện chuẩn C++ nhằm mục đích cung cấp các thành phần hữu ích cho hầu hết mọi ngư ời trong hầu hết các lĩnh vực ứng dụng. Đó là, nó nhằm mục đích phục vụ giao điểm của tất cả các nhu cầu hơ n là sự kết hợp của chúng. Ngoài ra, hỗ trợ cho một số lĩnh vực ứng dụng quan trọng rộng rãi, chẳng hạn như tính toán toán học và thao tác văn bản, đã xuất hiện.

Machine Translated by Google

5.2 Quản lý tài nguyên

Một trong những nhiệm vụ chính của bất kỳ chư ơ ng trình không tầm thư ờng nào là quản lý tài nguyên. Tài nguyên là thứ phải đư ợc mua và sau đó đư ợc giải phóng (rõ ràng hoặc ngầm định). Ví dụ là bộ nhớ, khóa, ổ cắm, xử lý luồng và xử lý tệp. Đối với một chư ơ ng trình chạy trong thời gian dài, việc không giải phóng tài nguyên kịp thời (''rò rỉ'') có thể gây ra sự suy giảm hiệu suất nghiêm trọng và thậm chí có thể gây ra sự cố nghiêm trọng. Ngay cả đối với các chư ơ ng trình ngắn, rò rỉ có thể trở thành một sự bối rối, chẳng hạn như do thiếu tài nguyên làm tăng thời gian chạy theo các mức độ lớn.

Các thành phần thư viện tiêu chuẩn đư ợc thiết kế để không rò rỉ tài nguyên. Để làm điều này, chúng dựa vào hỗ trợ ngôn ngữ cơ bản để quản lý tài nguyên bằng cách sử dụng các cặp hàm tạo/hàm hủy để đảm bảo rằng tài nguyên không tồn tại lâu hơ n một đối tư ợng chịu trách nhiệm về nó. Việc sử dụng cặp hàm tạo/hàm hủy trong Vector để quản lý thời gian tồn tại của các phần tử của nó là một ví dụ (§3.2.1.2) và tất cả các bộ chứa thư viện tiêu chuẩn đều đư ợc triển khai theo những cách tư ơ ng tự. Điều quan trọng là cách tiếp cận này tư ơ ng tác chính xác với việc xử lý lỗi bằng các ngoại lệ. Ví dụ: kỹ thuật này đư ợc sử dụng cho các lớp khóa thư viện tiêu chuẩn:

```
câm m; // //      đư ợc sử dụng để bảo vệ quyền truy cập vào dữ liệu đư ợc chia sẻ
   ...
void f() {

        unique_lock<mutex> lck {m}; // //           có đư ợc mutex m
          ... thao tác dữ liệu đư ợc chia sẻ ···
}
```

Một luồng sẽ không tiếp tục cho đến khi hàm tạo của lck có đư ợc mutex của nó, m (§5.3.4). Trình phá hủy tư ơ ng ứng giải phóng tài nguyên. Vì vậy, trong ví dụ này, hàm hủy của unique_lock giải phóng mutex khi chuỗi điều khiển rời f() (thông qua một lư ợt trả về, bằng cách ''rơ i khỏi phần cuối của hàm'' hoặc thông qua một lần ném ngoại lệ).

Đây là một ứng dụng của kỹ thuật ''Mua lại tài nguyên là khởi tạo'' (RAII; §3.2.1.2, §13.3). Kỹ thuật này là nền tảng cho việc xử lý tài nguyên thành ngữ trong C++. Các bộ chứa (chẳng hạn như vectơ và bản đồ), chuỗi và iostream quản lý tài nguyên của chúng (chẳng hạn như bộ xử lý tệp và bộ đệm) theo cách tư ơ ng tự.

5.2.1 unique_ptr và shared_ptr

Các ví dụ cho đến nay xử lý các đối tư ợng đư ợc xác định trong một phạm vi, giải phóng các tài nguyên mà chúng có đư ợc khi thoát khỏi phạm vi, như ng còn các đối tư ợng đư ợc phân bổ trên cửa hàng miễn phí thì sao? Trong <memory>, thư viện stan dard cung cấp hai ''con trỏ thông minh'' để giúp quản lý các đối tư ợng trên kho lư u trữ miễn phí:

[1] unique_ptr để thể hiện quyền sở hữu duy nhất (§34.3.1) [2]

shared_ptr để thể hiện quyền sở hữu đư ợc chia sẻ (§34.3.2)

Công dụng cơ bản nhất của những ''con trỏ thông minh'' này là để ngăn rò rỉ bộ nhớ do lập trình lập trình bất cẩn. Ví dụ:

```
void f(int i, int j) {      // X* so với unique_ptr<X>

        X   p = X mới;                          // chỉ định   mới      X
        unique_ptr<X> sp {new X}; //               phân bổ mới         X và đư a con trỏ của nó tới unique_ptr
        // ...
```

Machine Translated by Google


```
        if (i<99) ném Z{}; nếu                  // thể một ném có      ngoại lệ
        (j<77) quay lại;                           có thể trả về "đầu ném ly"
        p  >do_something();                     // thể một ném có ngoại lệ ngoại
        sp  >do_something(); //                 //       lệ có thể là một
           ...

        xóa p;                                  // tiêu diệt *p
    }
```

Ở đây, chúng ta ''quên'' xóa p nếu i<99 hoặc nếu j<77. Mặt khác, unique_ptr đảm bảo rằng đối tư ợng của nó bị hủy đúng cách cho dù chúng ta thoát khỏi f() theo cách nào (bằng cách đư a ra một ngoại lệ, bằng cách thực hiện trả về hoặc bằng cách ''rơ i khỏi phần cuối''). Trớ trêu thay, chúng ta có thể giải quyết vấn đề đơ n giản bằng cách không sử dụng con trỏ và không sử dụng new:

```
    void f(int i, int j)        // dụng một var địa phư ơ ng tôi có thể sử
    {
          Xx; //
             ...
    }
```

Thật không may, việc lạm dụng new (và của con trỏ và tham chiếu) dư ờng như là một vấn đề ngày càng gia tăng.

Tuy nhiên, khi bạn thực sự cần ngữ nghĩa của con trỏ, thì unique_ptr là một cơ chế rất nhẹ, không tốn nhiều không gian hoặc thời gian so với việc sử dụng đúng con trỏ tích hợp sẵn. Các ứng dụng khác của nó bao gồm chuyển các đối tư ợng đư ợc phân bổ lư u trữ miễn phí vào và ra khỏi các chức năng:

```
    unique_ptr<X> make_X(int i) //
             làm một    X và ngay lập tức đư a nó cho một    unique_ptr
    {
          // ... kiểm tra tôi, v.v.  ...
          trả về unique_ptr<X>{new X{i}};
    }
```

Một unique_ptr là một bộ điều khiển cho một đối tư ợng riêng lẻ (hoặc một mảng) theo cách giống như một vectơ là một bộ điều khiển cho một chuỗi các đối tư ợng. Cả hai đều kiểm soát thời gian tồn tại của các đối tư ợng khác (sử dụng RAII) và cả hai đều dựa vào ngữ nghĩa di chuyển để trả về đơ n giản và hiệu quả.

shared_ptr tư ơ ng tự như unique_ptr ngoại trừ shared_ptrs đư ợc sao chép thay vì di chuyển. shared_ptrs cho một đối tư ợng chia sẻ quyền sở hữu của một đối tư ợng và đối tư ợng đó bị hủy khi shared_ptrs cuối cùng của nó bị hủy. Ví dụ:

```
    void f(shared_ptr<fstream>); void
    g(shared_ptr<fstream>);

    void user(const string& name, ios_base::openmode mode) {

          shared_ptr<fstream> fp {fstream mới(tên ,chế độ)}; if (! fp) ném
          No_file{}; //                      bảo đảm      tập tin là     mở đúng cách

          f(fp);
          g(fp); //
             ...
    }
```

Machine Translated by Google

Bây giờ, tệp đư ợc mở bởi hàm tạo của fp sẽ bị đóng bởi hàm cuối cùng (rõ ràng hoặc ngụ ý) hủy một bản sao của fp. Lư u ý rằng f() hoặc g() có thể sinh ra một tác vụ đang giữ một bản sao của fp hoặc theo một cách nào đó lư u trữ một bản sao tồn tại lâu hơ n user(). Do đó, shared_ptr cung cấp một dạng bộ sư u tập rác tôn trọng việc quản lý tài nguyên dựa trên hàm hủy của các đối tư ợng đư ợc quản lý bộ nhớ. Điều này không miễn phí cũng không quá đắt, như ng làm cho thời gian tồn tại của đối tư ợng đư ợc chia sẻ khó dự đoán. Chỉ sử dụng shared_ptr nếu bạn thực sự cần chia sẻ quyền sở hữu.

Với unique_ptr và shared_ptr, chúng ta có thể triển khai chính sách ''no naked new'' hoàn chỉnh (§3.2.1.2) cho nhiều chư ơ ng trình. Tuy nhiên, những ''con trỏ thông minh'' này vẫn là các con trỏ về mặt khái niệm và do đó chỉ là lựa chọn thứ hai của tôi để quản lý tài nguyên – sau các vùng chứa và các loại khác quản lý tài nguyên của chúng ở mức khái niệm cao hơ n. Cụ thể, bản thân shared_ptrs không cung cấp bất kỳ quy tắc nào để chủ sở hữu của chúng có thể đọc và/hoặc ghi đối tư ợng đư ợc chia sẻ. Các cuộc chạy đua dữ liệu (§41.2.4) và các dạng nhầm lẫn khác không đư ợc giải quyết đơ n giản bằng cách loại bỏ các vấn đề về quản lý tài nguyên.

Chúng ta sử dụng ''con trỏ thông minh'' (chẳng hạn như unique_ptr) ở đâu thay vì các bộ điều khiển tài nguyên với các thao tác đư ợc thiết kế riêng cho tài nguyên (chẳng hạn như vectơ hoặc luồng)? Không có gì ngạc nhiên khi câu trả lời là ''khi chúng ta cần ngữ nghĩa con trỏ.''

- Khi chia sẻ một đối tư ợng, chúng ta cần các con trỏ (hoặc tham chiếu) để tham chiếu đến đối tư ợng đư ợc chia sẻ, vì vậy shared_ptr trở thành lựa chọn hiển nhiên (trừ khi có một lựa chọn hiển nhiên ngư ời sở hữu).
- Khi chúng ta tham chiếu đến một đối tư ợng đa hình, chúng ta cần một con trỏ (hoặc một tham chiếu) bởi vì chúng ta không biết chính xác kiểu của đối tư ợng đư ợc tham chiếu hoặc thậm chí kích thư ớc của nó), do đó, một unique_ptr trở thành sự lựa chọn hiển nhiên .
- Một đối tư ợng đa hình đư ợc chia sẻ thư ờng yêu cầu shared_ptrs.

Chúng ta không cần sử dụng một con trỏ để trả về một tập hợp các đối tư ợng từ một hàm; vùng chứa là nơ i xử lý tài nguyên sẽ thực hiện việc đó một cách đơ n giản và hiệu quả (§3.3.2).

5.3 Đồng thời

Đồng thời – việc thực thi đồng thời một số tác vụ – đư ợc sử dụng rộng rãi để cải thiện thông qua đặt (bằng cách sử dụng một số bộ xử lý cho một lần tính toán) hoặc để cải thiện khả năng phản hồi (bằng cách cho phép một phần của chư ơ ng trình tiến triển trong khi phần khác đang chờ phản hồi). Tất cả các ngôn ngữ ngữ pháp chuyên nghiệp hiện đại đều cung cấp hỗ trợ cho việc này. Sự hỗ trợ đư ợc cung cấp bởi thư viện chuẩn C++ là một biến thể di động và an toàn kiểu của những gì đã đư ợc sử dụng trong C++ trong hơ n 20 năm và hầu như đư ợc phần cứng hiện đại hỗ trợ phổ biến. Hỗ trợ thư viện tiêu chuẩn chủ yếu nhằm mục đích hỗ trợ đồng thời cấp hệ thống thay vì trực tiếp cung cấp các mô hình đồng thời cấp cao phức tạp hơ n; chúng có thể đư ợc cung cấp dư ới dạng các thư viện đư ợc xây dựng bằng các cơ sở thư viện tiêu chuẩn.

Thư viện chuẩn hỗ trợ trực tiếp việc thực thi đồng thời nhiều luồng trong một không gian địa chỉ. Để cho phép điều đó, C++ cung cấp một mô hình bộ nhớ phù hợp (§41.2) và một tập hợp các hoạt động nguyên tử (§41.3). Tuy nhiên, hầu hết ngư ời dùng sẽ chỉ thấy đồng thời về mặt thư viện chuẩn và các thư viện đư ợc xây dựng trên đó. Phần này đư a ra các ví dụ ngắn gọn về các phư ơ ng tiện hỗ trợ đồng thời trong thư viện tiêu chuẩn chính: luồng, mutexes, lock() operation, packages_tasks và futures. Các tính năng này đư ợc xây dựng trực tiếp dựa trên những gì hệ điều hành cung cấp và không phải chịu các hình phạt về hiệu suất so với các tính năng đó.

Machine Translated by Google

5.3.1 Nhiệm vụ và luồng

Chúng tôi gọi một phép tính có khả năng đư ợc thực hiện đồng thời với các phép tính khác là một tác vụ. Một luồng là biểu diễn cấp hệ thống của một tác vụ trong một chư ơ ng trình. Một tác vụ đư ợc thực thi đồng thời với các tác vụ khác đư ợc khởi chạy bằng cách xây dựng một std::thread (đư ợc tìm thấy trong <thread>) với tác vụ làm đối số của nó. Một nhiệm vụ là một chức năng hoặc một đối tư ợng chức năng:

```
khoảng trống f();                    // chức năng

cấu trúc F                           // toán tử gọi đối
        { toán tử void()();          // tư ợng hàm F (§3.4.3)
};

ngư ời dùng vô hiệu ()
{
        đề t1{f}; luồng                f() thực thi trong luồng riêng biệt
        t2 {F()};                    // F()() thực thi trong luồng riêng biệt

        t1.join();                   // đợi t1
        t2.join();                   // đợi t2
}
```

Các tham gia () đảm bảo rằng chúng tôi không thoát khỏi ngư ời dùng () cho đến khi các luồng hoàn thành. ''tham gia'' có nghĩa là ''chờ chủ đề kết thúc.''

Chủ đề của một chư ơ ng trình chia sẻ một không gian địa chỉ duy nhất. Ở điểm này, các luồng khác với các quy trình, thư ờng không chia sẻ dữ liệu trực tiếp. Vì các luồng chia sẻ một không gian địa chỉ nên chúng có thể giao tiếp thông qua các đối tư ợng đư ợc chia sẻ (§5.3.4). Giao tiếp như vậy thư ờng đư ợc kiểm soát bởi các khóa hoặc các cơ chế khác để ngăn chặn các cuộc đua dữ liệu (truy cập đồng thời không đư ợc kiểm soát vào một biến).

Lập trình các tác vụ đồng thời có thể rất phức tạp. Xem xét các triển khai có thể có của nhiệm vụ f (một chức năng) và F (một đối tư ợng chức năng):

```
void f() { cout << "Xin chao "; }

struct F
        { toán tử void()() { cout << "Thế giới song song!\n"; }
};
```

Đây là một ví dụ về một lỗi nghiêm trọng: Ở đây, f và F() đều sử dụng đối tư ợng cout mà không có bất kỳ hình thức đồng bộ hóa thời gian nào. Đầu ra kết quả sẽ không thể đoán trư ớc và có thể khác nhau giữa các lần thực hiện khác nhau của chư ơ ng trình vì thứ tự thực hiện các thao tác riêng lẻ trong hai tác vụ không đư ợc xác định. Chư ơ ng trình có thể tạo đầu ra ''lẻ'', chẳng hạn như

```
PaHerallllel o Thế giới!
```

Khi xác định các nhiệm vụ của một chư ơ ng trình đồng thời, mục tiêu của chúng tôi là giữ cho các nhiệm vụ hoàn toàn tách biệt trừ khi chúng giao tiếp theo những cách đơ n giản và rõ ràng. Cách đơ n giản nhất để nghĩ về một nhiệm vụ cho thuê đồng thời là một chức năng xảy ra để chạy đồng thời với ngư ời gọi nó. Để điều đó hoạt động, chúng tôi chỉ cần chuyển các đối số, nhận lại kết quả và đảm bảo rằng không sử dụng dữ liệu đư ợc chia sẻ ở giữa (không có cuộc đua dữ liệu).

Machine Translated by Google

5.3.2 Thông qua đối số

Thông thư ờng, một tác vụ cần có dữ liệu để thực hiện. Chúng ta có thể dễ dàng chuyển dữ liệu (hoặc con trỏ hoặc tham chiếu đến dữ liệu) dư ới dạng đối số. Coi như :

```
void f(vectơ <kép>& v);                    // chức năng làm gì đó với v

cấu trúc F                                 // đối tư ợng chức năng: làm gì đó với v
        { véc tơ  <gấp đôi>& v;
        F(vector<double>& vv) :v{vv} { } toán tử
        void()();                          // toán tử ứng dụng §3.4.3 ;
};

int main() {

        vector<double> some_vec {1,2,3,4,5,6,7,8,9}; vectơ <đôi>
        vec2 {10,11,12,13,14};

        chủ đề t1 {f,some_vec}; // thread    f(some_vec) thực thi trong một chuỗi riêng biệt
        t2 {F{vec2}}; //                     F(vec2)() thực thi trong một luồng riêng biệt

        t1.join();
        t2.join();
}
```

Rõ ràng, `F{vec2}` lư u một tham chiếu đến vectơ đối số trong F. Bây giờ `F` có thể sử dụng mảng đó và hy vọng không có tác vụ nào khác truy cập `vec2` trong khi `F` đang thực thi. Chuyển `vec2` theo giá trị sẽ loại bỏ rủi ro đó.

Quá trình khởi tạo với `{f,some_vec}` sử dụng hàm tạo mẫu biến đổi theo `luồng` có thể chấp nhận một chuỗi đối số tùy ý (§28.6). Trình biên dịch kiểm tra xem đối số đầu tiên có thể đư ợc gọi với các đối số sau hay không và xây dựng đối tư ợng hàm cần thiết để chuyển đến luồng. Do đó, nếu `F::operator()()` và `f()` thực hiện cùng một thuật toán, thì việc xử lý hai tác vụ gần như tư ơ ng đư ơ ng nhau: trong cả hai trư ờng hợp, một đối tư ợng hàm đư ợc tạo để luồng `thực` thi.

5.3.3 Trả kết quả

Trong ví dụ ở §5.3.2, tôi chuyển các đối số bằng tham chiếu `không phải const` . Tôi chỉ làm điều đó nếu tôi muốn tác vụ sửa đổi giá trị của dữ liệu đư ợc đề cập (§7.7). Đó là một cách hơ i lén lút, như ng không phải là hiếm, để trả lại kết quả. Một kỹ thuật ít khó hiểu hơ n là chuyển dữ liệu đầu vào bằng tham chiếu `const` và chuyển vị trí của một địa điểm để gửi kết quả dư ới dạng đối số riêng biệt:

```
void f(const vector<double>& v, double   res);//          lấy đầu vào từ vị trí kết quả trong *res v;

lớp F
{ công khai:
        F(const vector<double>& vv, double   p) :v{vv}, res{p} { } toán tử void()();
                                                  // đặt kết quả vào *res
```

Machine Translated by Google


```
riêng tư :
        const vector<double>& v; gấp đôi                // tiêu      của mục
        độ phân giải;                            // nguồn đầu vào cho đầu ra
};

int main() {

        vector<đôi> some_vec; vecto<đôi>
        vec2; //
           ...

        gấp đôi độ phân giải1;
        gấp đôi độ phân giải2;

        chủ đề t1 {f,some_vec,&res1}; // thread t2  f(some_vec,&res1) thực thi trong một chuỗi riêng biệt
        {F{vec2,&res2}};                                // F{vec2,&res2}() thực thi trong một chuỗi riêng biệt

        t1.join();
        t2.join();

        cout << res1 <<          ''
                                    << res2 << '\n';
}
```

Tôi không coi kết quả trả về thông qua các đối số là đặc biệt tao nhã, vì vậy tôi quay lại chủ đề này trong §5.3.5.1.

5.3.4 Chia sẻ dữ liệu

Đôi khi các tác vụ cần chia sẻ dữ liệu. Trong trư ờng hợp đó, quyền truy cập phải đư ợc đồng bộ hóa để tối đa một tác vụ tại một thời điểm có quyền truy cập. Các lập trình viên có kinh nghiệm sẽ nhận ra đây là một sự đơ n giản hóa (ví dụ: không có vấn đề gì với nhiều tác vụ đồng thời đọc dữ liệu bất biến), như ng hãy cân nhắc cách đảm bảo rằng tại một thời điểm, nhiều nhất một tác vụ có quyền truy cập vào một tập đối tư ợng nhất định.

Yếu tố cơ bản của giải pháp là một mutex, một ''đối tư ợng loại trừ lẫn nhau''. có đư ợc một mutex bằng thao tác lock() :

```
câm m; // int   kiểm soát dữ liệu
sh; //           chia sẻ mutex


khoảng trống
f() {
        unique_lock<mutex> lck {m}; // sh +=        có đư ợc mutex
        7; // //                                   thao tác dữ liệu đư ợc chia sẻ
}         giải phóng mutex hoàn toàn
```

Hàm tạo của unique_lock thu đư ợc mutex (thông qua lệnh gọi m.lock()). Nếu một luồng khác đã có đư ợc mutex, thì luồng đó sẽ đợi (''khối'') cho đến khi luồng kia hoàn tất quyền truy cập của nó. Khi một luồng đã hoàn thành quyền truy cập vào dữ liệu đư ợc chia sẻ, unique_lock sẽ giải phóng mutex (với lệnh gọi m.unlock()). Các phư ơ ng tiện khóa và loại trừ lẫn nhau đư ợc tìm thấy trong <mutex>.

Machine Translated by Google

Sự tư ơ ng ứng giữa dữ liệu đư ợc chia sẻ và một mutex là quy ư ớc: lập trình viên chỉ cần biết mutex nào đư ợc cho là tư ơ ng ứng với dữ liệu nào. Rõ ràng, điều này dễ gây ra lỗi, và rõ ràng là chúng tôi cố gắng làm cho sự tư ơ ng ứng rõ ràng thông qua các phư ơ ng tiện ngôn ngữ khác nhau.
Ví dụ:

```
Bản ghi lớp { công
khai:
      mutex rm; //
          ...
};
```

Không cần phải là một thiên tài để đoán rằng đối với Bản ghi có tên là rec, rec.rm là một mutex mà bạn phải có đư ợc trư ớc khi truy cập dữ liệu khác của rec, mặc dù một nhận xét hoặc một cái tên hay hơ n có thể đã giúp ích cho ngư ời đọc.

Không có gì lạ khi cần truy cập đồng thời một số tài nguyên để thực hiện một số hành động. Điều này có thể dẫn đến bế tắc. Ví dụ: nếu thread1 nhận đư ợc mutex1 và sau đó cố gắng nhận đư ợc mutex2 trong khi thread2 nhận đư ợc mutex2 và sau đó cố gắng nhận đư ợc mutex1, thì sẽ không có tác vụ nào đư ợc tiếp tục. Thư viện tiêu chuẩn cung cấp trợ giúp dư ới dạng thao tác để có đư ợc nhiều khóa cùng lúc:

```
khoảng trống
f() {
        // ...
        unique_lock<mutex> lck1 {m1,defer_lock}; // unique_lock<mutex> lck2      defer_lock: chư a thử lấy mutex
        {m2,defer_lock}; unique_lock<mutex> lck3 {m3,defer_lock};

        // ...
        khóa(lck1,lck2,lck3);                                          // có đư ợc cả ba ổ khóa
        // ... thao tác dữ liệu đư ợc chia sẻ     ...
} // hoàn toàn giải phóng tất cả các mutexes
```

Lock() này sẽ chỉ tiến hành sau khi có đư ợc tất cả các đối số mutex của nó và sẽ không bao giờ chặn (''đi ngủ'') trong khi giữ một mutex. Các bộ hủy cho các khóa duy nhất riêng lẻ đảm bảo rằng các bộ chuyển đổi đư ợc giải phóng khi một chuỗi rời khỏi phạm vi.

Giao tiếp thông qua dữ liệu đư ợc chia sẻ ở mức khá thấp. Đặc biệt, lập trình viên phải nghĩ ra các cách để biết công việc nào đã đư ợc thực hiện và chư a đư ợc thực hiện bởi các tác vụ khác nhau. Về vấn đề đó, việc sử dụng dữ liệu đư ợc chia sẻ kém hơ n so với khái niệm gọi và quay lại. Mặt khác, một số ngư ời tin chắc rằng việc chia sẻ phải hiệu quả hơ n việc sao chép các đối số và trả về. Điều đó thực sự có thể xảy ra khi có liên quan đến một lư ợng lớn dữ liệu, như ng việc khóa và mở khóa là những hoạt động tư ơ ng đối tốn kém. Mặt khác, máy móc hiện đại sao chép dữ liệu rất tốt, đặc biệt là dữ liệu nhỏ gọn, chẳng hạn như các phần tử vectơ . Vì vậy, đừng chọn dữ liệu đư ợc chia sẻ để liên lạc vì ''hiệu quả'' mà không cần suy nghĩ và tốt nhất là không nên đo lư ờng.

5.3.4.1 Chờ sự kiện

Đôi khi, một luồng cần đợi một số loại sự kiện bên ngoài, chẳng hạn như một luồng khác hoàn thành một tác vụ hoặc một khoảng thời gian nhất định đã trôi qua. "Sự kiện" đơ n giản nhất chỉ đơ n giản là thời gian trôi qua. Coi như :

Machine Translated by Google


```
sử dụng không gian tên std::chrono;                // min thấy §35.2

tự động t0 = high_resolution_clock::now();
this_thread::sleep_for(mili giây{20}); auto t1 =
high_resolution_clock::now(); cout << duration_cast<nano
giây>(t1  t0).count() << " nano giây đã qua\n";
```

Lư u ý rằng tôi thậm chí không phải khởi chạy một chủ đề; theo mặc định, this_thread đề cập đến một và chỉ một luồng (§42.2.6).

Tôi đã sử dụng duration_cast để điều chỉnh đơ n vị của đồng hồ thành nano giây mà tôi muốn. Xem §5.4.1 và §35.2 trư ớc khi thử bất cứ điều gì phức tạp hơ n điều này theo thời gian. Các cơ sở thời gian đư ợc tìm thấy trong <thời gian>.

Hỗ trợ cơ bản để giao tiếp bằng các sự kiện bên ngoài đư ợc cung cấp bởi biến_điều kiện có trong <biến_điều_kiện> (§42.3.4). Biến_điều_kiện là một cơ chế cho phép một luồng chờ một luồng khác . Đặc biệt, nó cho phép một luồng đợi một số điều kiện (thư ờng đư ợc gọi là một sự kiện) xảy ra do kết quả của công việc đư ợc thực hiện bởi các luồng khác.

Hãy xem xét ví dụ cổ điển về hai luồng giao tiếp bằng cách truyền tin nhắn qua hàng đợi. Để đơ n giản, tôi tuyên bố hàng đợi và cơ chế tránh điều kiện chủng tộc trên hàng đợi đó trên toàn cầu cho nhà sản xuất và ngư ời tiêu dùng:

```
tin nhắn lớp {              // đối tư ợng đư ợc giao tiếp
      // ...
};

hàng đợi <Tin nhắn> mqueue; điều                // tin nhắn của hàng đợi của sự kiện giao tiếp
kiện_biến mcond; đột biến mmutex;               // khả thi var cơ chế khóa
                                                //
```

Các loại queue, condition_variable và mutex đư ợc cung cấp bởi thư viện chuẩn.

Consumer () đọc và xử lý Messages:

```
void ngư ời tiêu dùng()
{
      while(true)
            { unique_lock<mutex> lck{mutex}; trong khi                lấy lck phát hành
            (mcond.wait(lck)) /*                không làm gì cả //    mmutex và đợi; lấy lại lck
                                                                      Wakeup khi nhận đư ợc tin nhắn
            auto m = mqueue.front();                            */; // // //
            mqueue.pop();
            lck.unlock(); //                                      // phát hành lck
              ... quá trình m   ...
      }
}
```

Ở đây, tôi bảo vệ rõ ràng các thao tác trên hàng đợi và trên biến_điều_kiện với một_khóa_d_nhất trên mutex. Việc chờ đợi trên biến_điều_kiện sẽ giải phóng đối số khóa của nó cho đến khi thời gian chờ đợi kết thúc (để hàng đợi không trống) rồi yêu cầu lại nó.

Nhà sản xuất tư ơ ng ứng trông như thế này:

Machine Translated by Google


```
void nhà sản xuất()
{
        trong khi (đúng) {
                Nhắn m; //
                  ...điền tin nhắn...
                unique_lock<mutex> lck {mmutex}; mqueue.push(m);      // bảo vệ hoạt động
                mcond.notify_one();
                                                                      thông
        }                                                             // báo khóa phát hành (ở cuối phạm vi)
}
```

Việc sử dụng biến_điều_kiện hỗ trợ nhiều hình thức chia sẻ hiệu quả và tao nhã, như ng có thể khá phức tạp (§42.3.4).

5.3.5 Các nhiệm vụ giao tiếp Thư viện

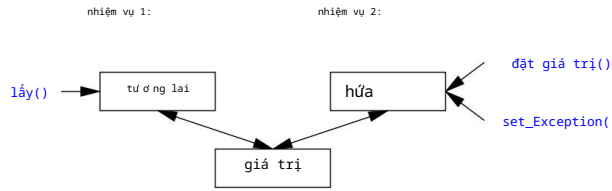
chuẩn cung cấp một vài tiện ích cho phép các lập trình viên vận hành ở cấp độ khái niệm của các nhiệm vụ (công việc có khả năng đư ợc thực hiện đồng thời) thay vì trực tiếp ở cấp độ thấp hơ n của các luồng và khóa: [1] tư ơ ng lai và lời

hứa cho trả về một giá trị từ một tác vụ đư ợc sinh ra trên một luồng riêng biệt [2] packet_task để giúp khởi chạy các tác vụ và kết nối các cơ chế trả về kết quả [3] async() để khởi chạy một tác vụ theo cách rất giống với cách gọi một hàm.

Những cơ sở này đư ợc tìm thấy trong <tư ơ ng lai>.

5.3.5.1 tư ơ ng lai và hứa hẹn

Điểm quan trọng về tư ơ ng lai và lời hứa là chúng cho phép chuyển một giá trị giữa hai tác vụ mà không cần sử dụng khóa một cách rõ ràng; ''hệ thống'' thực hiện việc chuyển giao một cách hiệu quả. Ý tư ởng cơ bản rất đơ n giản: Khi một tác vụ muốn chuyển một giá trị cho một tác vụ khác, nó sẽ đặt giá trị đó vào một lời hứa. Bằng cách nào đó, việc triển khai làm cho giá trị đó xuất hiện trong tư ơ ng lai tư ơ ng ứng, từ đó nó có thể đư ợc đọc (thư ờng bởi trình khởi chạy tác vụ). Chúng ta có thể biểu diễn điều này bằng đồ họa:



Nếu chúng ta có một future<X> đư ợc gọi là fx, chúng ta có thể lấy() một giá trị loại X từ nó:

```
X v = fx.g et(); //    nếu đợi giá trị đư ợc tính toán cần thiết,
```

Nếu giá trị chư a có, luồng của chúng tôi sẽ bị chặn cho đến khi nó đến. Nếu giá trị không thể đư ợc tính toán, get() có thể đư a ra một ngoại lệ (từ hệ thống hoặc đư ợc truyền từ tác vụ mà chúng tôi đang cố lấy() giá trị).

Machine Translated by Google

Mục đích chính của một lời hứa là cung cấp các hoạt động ''đặt'' đơ n giản (đư ợc gọi là set_value() và set_Exception()) để khớp với get() của tư ơ ng lai . Những cái tên ''tư ơ ng lai'' và ''hứa hẹn'' là lịch sử; xin đừng đổ lỗi cho tôi. Chúng là một nguồn chơ i chữ phong phú khác.

Nếu bạn có một lời hứa và cần gửi kết quả loại X đến một tư ơ ng lai, bạn có thể thực hiện một trong hai điều sau: chuyển một giá trị hoặc chuyển một ngoại lệ. Ví dụ:

```
void f(lời hứa<X>& px) // { Một nhiệm vụ: đặt kết quả trong px

        // ...
        thử {X
              độ phân
                  ... tính toán giá trị cho một độ phân ...
              giải; // px.set_value(res);

        } bắt (...) { // // tiếc: không thể tính toán rất độ phân giải
                    ngoại lệ đối với chủ đề của tư ơ ng lai: vư ợt qua
              px.set_Exception(current_Exception());
        }
}
```

current_Exception () đề cập đến ngoại lệ bị bắt (§30.4.1.2).

Để đối phó với một ngoại lệ đư ợc truyền qua một tư ơ ng lai, ngư ời gọi get() phải sẵn sàng bắt nó ở đâu đó. Ví dụ:

```
void g(tư ơ ng lai<X>& fx) {  // nhiệm vụ: lấy kết quả từ fx a

        // ...
        thử
            { X v = fx.g et(); // // nếu đợi giá trị đư ợc tính toán cần thiết,
                ... sử dụng v ...

        } bắt (...) {  // ai đó không thể tính toán rất tiếc: trong
              // ... xử lý lỗi ...
        }
}
```

5.3.5.2 đóng gói_tác vụ

Làm cách nào để chúng ta đư a tư ơ ng lai vào nhiệm vụ cần kết quả và lời hứa tư ơ ng ứng vào luồng sẽ tạo ra kết quả đó? Loại đóng gói_task đư ợc cung cấp để đơ n giản hóa việc thiết lập các tác vụ đư ợc kết nối với tư ơ ng lai và hứa hẹn sẽ chạy trên các luồng. Một đóng gói_task cung cấp mã trình bao bọc để đư a giá trị trả về hoặc ngoại lệ từ tác vụ vào một lời hứa (như mã đư ợc hiển thị trong §5.3.5.1). Nếu bạn hỏi nó bằng cách gọi get_future, thì một packages_task sẽ cung cấp cho bạn tư ơ ng lai tư ơ ng ứng với lời hứa của nó. Ví dụ: chúng ta có thể thiết lập hai tác vụ để mỗi tác vụ thêm một nửa phần tử của vectơ <double> bằng cách sử dụng hàm tích lũy thư viện chuẩn () (§3.4.2, §40.6):

Machine Translated by Google


```
kiêm đôi(double  beg, double   end, double init)
      // tính toán [beg:end) bắt đầu với giá trị ban đầu init sum
{
      trả về tích lũy (xin, kết thúc, khởi tạo);
}

double comp2(vector<double>& v) {

      sử dụng Task_type = double(double  ,double  ,double);                    // loại nhiệm vụ

      packet_task<Task_type> pt0 {accum}; packet_task<Task_type>               // đóng gói nhiệm vụ (tức là tích lũy)
      pt1 {accum};

      tư ơ ng lai<đôi> f0 {pt0.get_future()}; tư ơ ng                          // nắm giữ tư ơ nglai của pt0
      lai<đôi> f1 {pt1.get_future()};                                          // nắm giữ tư ơ nglai của pt1

      double  first = &v[0]; chủ
      đề t1 {move(pt0),first,first+v.siz e()/2,0}; chủ đề t2                   chuỗisao ta cho pt0 chuỗi
      {move(pt1),first+v.siz e()/2,first+v.siz e(),0};                         // sao ta cho pt1

      // ...

      trả về f0.get()+f1.g et();                                          // nhận kết quả
}
```

Mẫu packages_task lấy loại nhiệm vụ làm đối số mẫu của nó (ở đây Task_type, bí danh cho double(double ,double ,double)) và nhiệm vụ làm đối số hàm tạo của nó (ở đây, accum). Thao tác move () là cần thiết vì không thể sao chép một packages_task .

Xin lư u ý rằng mã này không đề cập rõ ràng đến khóa: chúng tôi có thể tập trung vào các nhiệm vụ phải hoàn thành hơ n là vào các cơ chế đư ợc sử dụng để quản lý giao tiếp của chúng. Hai tác vụ sẽ đư ợc chạy trên các luồng riêng biệt và do đó có khả năng song song.

5.3.5.3 không đồng bộ()

Lối suy nghĩ mà tôi theo đuổi trong chư ơ ng này là lối suy nghĩ mà tôi cho là đơ n giản nhất như ng vẫn mạnh mẽ nhất: Coi một nhiệm vụ như một chức năng có thể xảy ra để chạy đồng thời với các nhiệm vụ khác. Nó không phải là mô hình duy nhất đư ợc thư viện chuẩn C++ hỗ trợ, như ng nó phục vụ tốt cho nhiều nhu cầu khác nhau. Các mô hình phức tạp và phức tạp hơ n, ví dụ, kiểu lập trình dựa trên bộ nhớ dùng chung, có thể đư ợc sử dụng khi cần thiết.

Để khởi chạy các tác vụ có khả năng chạy không đồng bộ, chúng ta có thể sử dụng async():

```
nhân đôi comp4(vectơ <đôi>& v)
      // nhiều            nhiệm vụ nếu v đủ lớn sinh ra
{
      if (v.siz e()<10000) return accum(v.begin(),v.end(),0.0);

      tự động v0 = &v[0];
      auto sz = v.siz e();
```

Machine Translated by Google


```
        auto f0 = async(accum,v0,v0+sz/4,0.0); auto          // quý 1 quý 2
        f1 = async(accum,v0+sz/4,v0+sz/2,0.0); auto f2 =       quý 3 quý 4 quý
        async(accum,v0+sz/2,v0+sz  3/4,0.0); auto f3 =       // 3
        async(accum,v0+sz  3/4,v0+sz,0.0);                   //

        return f0.get()+f1.g et()+f2.get()+f3.get(); //  thu thập và kết hợp các kết quả
    }
```

Về cơ bản, async() tách biệt ''phần cuộc gọi'' của lệnh gọi hàm với phần ''lấy kết quả'' và tách biệt cả hai khỏi quá trình thực thi tác vụ thực tế. Sử dụng async(), bạn không cần phải suy nghĩ về chủ đề và khóa. Thay vào đó, bạn chỉ nghĩ về các tác vụ có khả năng tính toán kết quả của chúng một cách không đồng bộ. Có một hạn chế rõ ràng: Thậm chí đừng nghĩ đến việc sử dụng async() cho các tác vụ chia sẻ tài nguyên cần khóa – với async() bạn thậm chí không biết có bao nhiêu luồng sẽ được sử dụng vì tùy thuộc vào async() để quyết định dựa trên dựa trên những gì nó biết về tài nguyên hệ thống có sẵn tại thời điểm cuộc gọi. Ví dụ: async() có thể kiểm tra xem có bất kỳ lõi (bộ xử lý) nhàn rỗi nào có sẵn hay không trước khi quyết định sử dụng bao nhiêu luồng .

Xin lưu ý rằng async() không chỉ là một cơ chế dành riêng cho tính toán song song để tăng hiệu suất. Ví dụ: nó cũng có thể được sử dụng để sinh ra tác vụ lấy thông tin từ người dùng, khiến ''chương trình chính'' hoạt động với một thứ khác (§42.4.6).

5.4 Các thành phần tiện ích nhỏ

Không phải tất cả các thành phần thư viện tiêu chuẩn đều là một phần của các phương tiện được dán nhãn rõ ràng, chẳng hạn như ''container'' hoặc ''I/O''. Phần này đưa ra một số ví dụ về các thành phần nhỏ, hữu ích rộng rãi:

- đồng hồ và khoảng thời gian để đo thời gian.
- Nhập các hàm, chẳng hạn như iterator_traits và is_arithmetic, để lấy thông tin về các loại.
- cặp và bộ để biểu diễn các tập giá trị nhỏ có khả năng không đồng nhất.

Vấn đề ở đây là một chức năng hoặc một loại không cần phải phức tạp hoặc liên kết chặt chẽ với hàng loạt các chức năng và loại khác để trở nên hữu ích. Các thành phần thư viện như vậy chủ yếu hoạt động như các khối xây dựng cho các cơ sở thư viện mạnh mẽ hơn, bao gồm các thành phần khác của thư viện tiêu chuẩn.

5.4.1 Thời gian

Thư viện tiêu chuẩn cung cấp các phương tiện để xử lý thời gian. Ví dụ, đây là cách cơ bản để tính thời gian cho một thứ gì đó:

```
    sử dụng không gian tên std::chrono;      // nhìn thấy §35.2

    tự động t0 = high_resolution_clock::now();
    làm việc();
    auto t1 = high_resolution_clock::now(); cout
    << duration_cast<mili giây>(t1  t0).count() << "msec\n";
```

Đồng hồ trả về time_point (một thời điểm). Trừ hai time_points sẽ cho một khoảng thời gian (một khoảng thời gian). Các đồng hồ khác nhau cho kết quả theo các đơn vị thời gian khác nhau (đồng hồ tôi sử dụng đo nano giây), do đó, thường nên chuyển đổi thời lượng thành một đơn vị đã biết. Đó là điều mà duration_cast làm.

Machine Translated by Google

Các cơ sở thư viện tiêu chuẩn để xử lý thời gian đư ợc tìm thấy trong không gian tên con std::chrono trong <chrono> (§35.2).

Đừng đư a ra tuyên bố về ''hiệu quả'' của mã mà không thực hiện các phép đo thời gian trư ớc. Đoán về hiệu suất là không đáng tin cậy nhất.

5.4.2 Loại chức năng

Hàm loại là một hàm đư ợc đánh giá tại thời điểm biên dịch đư ợc cung cấp một loại làm đối số của nó hoặc trả về một loại. Thư viện chuẩn cung cấp nhiều loại hàm để giúp ngư ời cố vấn triển khai thư viện và lập trình viên nói chung viết mã tận dụng các khía cạnh của ngôn ngữ, thư viện chuẩn và mã nói chung.

Đối với các loại số, numeric_limits từ <limits> trình bày nhiều thông tin hữu ích (§5.6.5). Ví dụ:

```
constexpr float min = numeric_limits<float>::min();          // số float dư ơ ng nhỏ nhất (§40.2)
```

Tư ơ ng tự, kích thư ớc đối tư ợng có thể đư ợc tìm thấy bằng toán tử sizeof tích hợp sẵn (§2.2.2). Ví dụ:

```
constexpr int size = sizeof(int); //  số byte trong int an
```

Các hàm kiểu như vậy là một phần của cơ chế tính toán thời gian biên dịch của C++, cho phép kiểm tra kiểu chặt chẽ hơ n và hiệu suất tốt hơ n so với khả năng khác. Việc sử dụng các tính năng như vậy thư ờng đư ợc gọi là siêu lập trình hoặc (khi có liên quan đến các mẫu) kết hợp siêu lập trình mẫu (Chư ơ ng 28). Ở đây, tôi chỉ trình bày hai phư ơ ng tiện do thư viện chuẩn cung cấp: iterator_traits (§5.4.2.1) và vị từ kiểu (§5.4.2.2).

5.4.2.1 iterator_traits

Sắp xếp thư viện chuẩn () lấy một cặp trình vòng lặp đư ợc cho là xác định một chuỗi (§4.5). Hơ n nữa, các trình lặp đó phải cung cấp quyền truy cập ngẫu nhiên vào trình tự đó, nghĩa là chúng phải là các trình lặp truy cập ngẫu nhiên. Một số vùng chứa, chẳng hạn như forward_list, không cung cấp điều đó. Cụ thể, for ward_list là một danh sách liên kết đơ n nên việc đăng ký sẽ tốn kém và không có cách nào hợp lý để tham chiếu lại phần tử trư ớc đó. Tuy nhiên, giống như hầu hết các vùng chứa, forward_list cung cấp các trình vòng lặp chuyển tiếp có thể đư ợc sử dụng để duyệt qua chuỗi theo thuật toán và câu lệnh for (§33.1.1).

Thư viện chuẩn cung cấp một cơ chế, iterator_traits cho phép chúng tôi kiểm tra loại trình vòng lặp nào đư ợc hỗ trợ. Do đó, chúng ta có thể cải thiện sắp xếp phạm vi () từ §4.5.6 để chấp nhận một vectơ hoặc một danh sách chuyển tiếp. Ví dụ:

```
kiểm tra void(vector<string>& v, forward_list<int>& lst) {

        sắp xếp(v); //      t vector sor
        sắp xếp(lst); //danh sách liên kết đơ n t
}
```

Các kỹ thuật cần thiết để thực hiện công việc đó nói chung là hữu ích.

Đầu tiên, tôi viết hai hàm trợ giúp nhận thêm một đối số cho biết liệu chúng sẽ đư ợc sử dụng cho trình lặp truy cập ngẫu nhiên hay trình lặp chuyển tiếp. Phiên bản lấy các đối số trình lặp truy cập ngẫu nhiên là tầm thư ờng:

Machine Translated by Google


```
mẫu<typename Ran> void                                          // cho trình vòng lặp truy cập ngẫu nhiên
sort_helper(Ran beg, Ran end, random_access_iterator_tag) {     // có thể chỉ số vào [beg:end) chúng ta

        sắp xếp (cầu xin, kết thúc);        // chỉ cần sắp xếp nó

}
```

Phiên bản cho các trình vòng lặp chuyển tiếp gần như đơn giản; chỉ cần sao chép danh sách vào một vectơ , sắp xếp và sao chép lại:

```
mẫu<typename For> void                                          // cho các trình vòng lặp chuyển tiếp
sort_helper(For beg, For end, forward_iterator_tag) {           // chúng ta có thể đi ngang [xin: kết thúc)

        vectơ <decltype( beg)> v {xin, kết thúc}; sắp      // khởi tạo vector từ [beg:end) a
        xếp(v.begin(),v.end()); sao
        chép(v.begin(),v.end(),beg);                       //      các yếu tố sao chép lại
}
```

decltype () là một hàm kiểu tích hợp trả về kiểu đối số đã khai báo của nó (§6.3.6.3).
Do đó, v là một vectơ <X> trong đó X là loại phần tử của chuỗi đầu vào.

''Loại ma thuật'' thực sự nằm trong việc lựa chọn các chức năng của trình trợ giúp:

```
template<typname C> void
sort(C& c) {

        sử dụng Iter = Iterator_type<C>;
        sort_helper(c.begin(),c.end(),Iterator_category<Iter>{});
}
```

Ở đây, tôi sử dụng hai hàm loại: Iterator_type<C> trả về loại trình lặp của C (nghĩa là C::iterator) và sau đó Iterator_category<Iter>{} tạo một giá trị ''tag'' cho biết loại trình lặp được cung cấp:

- std::random_access_iterator_tag nếu bộ lặp của C hỗ trợ truy cập ngẫu nhiên.
- std::forward_iterator_tag nếu bộ lặp của C hỗ trợ phép lặp xuôi.

Do đó, chúng ta có thể chọn giữa hai thuật toán sắp xếp tại thời điểm biên dịch. Kỹ thuật này, được gọi là công văn thẻ , là một trong nhiều kỹ thuật được sử dụng trong thư viện tiêu chuẩn và những nơi khác để cải thiện tính linh hoạt và hiệu suất.

Hỗ trợ thư viện tiêu chuẩn cho các kỹ thuật sử dụng trình vòng lặp, chẳng hạn như gửi thẻ, xuất hiện ở dạng mẫu lớp đơn giản iterator_traits từ <iterator> (§33.1.3). Điều này cho phép các định nghĩa đơn giản của các hàm kiểu được sử dụng trong sort():

```
mẫu<typename C> sử dụng
        Iterator_type = typename C::iterator; //                kiểu lặp của C

mẫu<typename Iter> sử dụng
        Iterator_category = typename std::iterator_traits<Iter>::iterator_category; //                Thể loại của Iter và
```

Nếu bạn không muốn biết loại ''phép thuật kiểu thời gian biên dịch'' nào được sử dụng để cung cấp các tính năng thư viện tiêu chuẩn, bạn có thể bỏ qua các tiện ích như iterator_traits . Nhưng sau đó bạn không thể sử dụng các kỹ thuật mà họ hỗ trợ để cải thiện mã của riêng bạn.

Machine Translated by Google

5.4.2.2 Vị ngữ loại

Vị ngữ kiểu thư viện chuẩn là một hàm kiểu đơ n giản trả lời câu hỏi cơ bản về kiểu. Ví dụ:

```
bool b1 = Is_arithmetic<int>();          // là có, một int kiểu số học
bool b2 = Is_arithmetic<chuỗi>(); //      không, một std::str ing không phải là kiểu số học
```

Các vị từ này đư ợc tìm thấy trong <type_traits> và đư ợc mô tả trong §35.4.1. Các ví dụ khác là is_class, is_pod, is_literal_type, has_virtual_desturationor và is_base_of. Chúng hữu ích nhất khi chúng ta viết các mẫu. Ví dụ:

```
mẫu <typename Scalar> lớp phức
tạp {
     vô hư ớng re,
im;
    public: static_assert(Is_arithmetic<Scalar>(),"Xin lỗi, tôi chỉ hỗ trợ tổ hợp các loại số học");
      // ...
};
```

Để cải thiện khả năng đọc so với việc sử dụng trực tiếp thư viện chuẩn, tôi đã định nghĩa một hàm kiểu:

```
mẫu<typename T> constexpr
bool Is_arithmetic() {

     trả về std::is_arithmetic<T>::value ;
}
```

Các chư ơ ng trình cũ hơ n sử dụng ::value trực tiếp thay vì (), như ng tôi cho rằng điều đó khá xấu và nó để lộ các chi tiết đề cập thực tế.

5.4.3 cặp và bộ

Thông thư ờng, chúng ta cần một số dữ liệu chỉ là dữ liệu; nghĩa là, một tập hợp các giá trị, chứ không phải là một đối tư ợng của một lớp có ngữ nghĩa đư ợc xác định rõ và bất biến đối với giá trị của nó (§2.4.3.2, §13.4). Trong những trư ờng hợp như vậy, chúng ta có thể định nghĩa một cấu trúc đơ n giản với một tập hợp các thành viên đư ợc đặt tên thích hợp. Ngoài ra, chúng ta có thể để thư viện chuẩn viết định nghĩa cho chúng ta. Ví dụ: thuật toán thư viện chuẩn equal_range (§32.6.1) trả về một cặp trình vòng lặp chỉ định một chuỗi con đáp ứng một vị từ:

```
mẫu<tên kiểu Forward_iterator, tên kiểu T, kiểu so sánh> cặp
      <Forward_iterator,Forward_iterator>
      equal_range(Forward_iterator trư ớc, Forward_iterator cuối cùng, const T& val, So sánh cmp);
```

Cho trư ớc một dãy đư ợc sắp xếp [đầu:cuối), equal_range() sẽ trả về cặp đại diện cho dãy con khớp với vị từ cmp. Chúng ta có thể sử dụng điều đó để tìm kiếm trong một chuỗi các Bản ghi đư ợc sắp xếp:

```
auto rec_eq = [](const Record& r1, const Record& r2) { return r1.name<r2.name;};//            so sánh tên

void f(const vector<Record>& v) {              // cho rằng    đó v   đư ợc sắp xếp trên  trư ờng "tên" của nó

      auto er = equal_range(v.begin(),v.end(),Record{"Reg"},rec_eq);
```

Machine Translated by Google


```
        for (auto p = er.first; p!=er.second; ++p) cout          // in tất cả các bản ghi
              <<   p;                                            // nhau giả sử rằng     được xác định cho  Ghi
    }
```

Thành viên đầu tiên của một cặp được gọi là thứ nhất và thành viên thứ hai được gọi là thứ hai. Cách đặt tên này không đặc biệt sáng tạo và thoạt nhìn có thể hơi kỳ lạ, nhưng cách đặt tên nhất quán như vậy là một lợi ích khi chúng ta muốn viết mã chung.

Cặp thư viện chuẩn (từ <utility>) được sử dụng khá thường xuyên trong thư viện chuẩn và các nơi khác. Một cặp cung cấp các toán tử, chẳng hạn như =, == và <, nếu các phần tử của nó cung cấp. Hàm make_pair() giúp dễ dàng tạo một cặp mà không cần đề cập rõ ràng về loại của nó (§34.2.4.1). Ví dụ:

```
    void f(vectơ <chuỗi>& v) {

        auto pp = make_pair(v.begin(),2); // //       pp  là một cặp<vector<string>::iterator,int>
          ...
    }
```

Nếu bạn cần nhiều hơn hai phần tử (hoặc ít hơn), bạn có thể sử dụng tuple (từ <utility>; §34.2.4.2). Một bộ là một chuỗi các phần tử không đồng nhất; Ví dụ:

```
    tuple<string,int,double> t2("Sild",123, 3.14); //          loại được chỉ định rõ ràng

    auto t = make_tuple(string("Herring"),10, 1.23);                loại được suy ra t là
                                                              // một tuple<string,int,double>

    chuỗi s = get<0>(t); //     get  phần tử đầu tiên của tuple
    int x = get<1>(t);
    double d = get<2>(t);
```

Các phần tử của một bộ được đánh số (bắt đầu bằng 0), thay vì được đặt tên theo cách các phần tử của các cặp (thứ nhất và thứ hai). Để có được lựa chọn phần tử thời gian biên dịch, thật không may, tôi phải sử dụng get<1>(t) xấu xí, thay vì get(t,1) hoặc t[1] (§28.5.2).

Giống như các cặp, các bộ có thể được chỉ định và so sánh nếu các phần tử của chúng có thể được.

Một cặp phổ biến trong các giao diện vì chúng ta thường muốn trả về nhiều hơn một giá trị, chẳng hạn như một kết quả và một chỉ báo về chất lượng của kết quả đó. Ít phổ biến hơn khi cần ba phần trở lên cho một kết quả, do đó, các bộ dữ liệu thường được tìm thấy nhiều hơn trong quá trình triển khai các thuật toán chung.

5.5 Biểu thức chính quy

Biểu thức chính quy là một công cụ mạnh mẽ để xử lý văn bản. Chúng cung cấp một cách để mô tả các mẫu trong văn bản một cách đơn giản và ngắn gọn (ví dụ: mã ZIP của Hoa Kỳ, chẳng hạn như TX 77845 hoặc ngày theo kiểu ISO, chẳng hạn như 2009 06 07) và để tìm các mẫu như vậy trong văn bản một cách hiệu quả. Trong <regex>, thư viện chuẩn cung cấp hỗ trợ cho các biểu thức chính quy dưới dạng lớp std::regex và các chức năng hỗ trợ của nó. Để biết phong cách của thư viện regex , chúng ta hãy xác định và in một mẫu:

```
    regex pat (R"(\w{2}\s  \d{5}(\d{4})?)"); // cout <<  Mẫu mã ZIP: XXddddd-dddd và iants var
    "mẫu: " << pat << '\n';
```

Machine Translated by Google

Những ngư ời đã sử dụng cụm từ thông dụng trong bất kỳ ngôn ngữ nào sẽ thấy \w{2}\s \d{5}(\d{4})? thân thuộc. Nó chỉ định một mẫu bắt đầu bằng hai chữ cái \w{2}, tùy chọn, theo sau là khoảng trắng \s , theo sau là năm chữ số \d{5} và tùy chọn, theo sau là dấu gạch ngang và bốn chữ số \d{4}. Nếu bạn không quen với các biểu thức chính quy, đây có thể là thời điểm tốt để tìm hiểu về chúng ([Stroustrup,2009], [Maddock,2009], [Friedl,1997]). Biểu thức chính quy đư ợc tóm tắt trong §37.1.1.

Để thể hiện mẫu, tôi sử dụng một chuỗi ký tự thô (§7.3.2.1) bắt đầu bằng R"(và kết thúc bằng)". Điều này cho phép dấu gạch chéo ngư ợc và dấu ngoặc kép đư ợc sử dụng trực tiếp trong chuỗi.

Cách đơ n giản nhất để sử dụng một mẫu là tìm kiếm nó trong luồng:

```
int lineno = 0;
for (dòng chuỗi; getline(cin,line);) { +                // đọc vào bộ đệm dòng
      +lineno;
      các trận đấu đối                                   các chuỗi phù hợp ở đây hãy
      kháng; if (regex_search(line ,matches,pat))        // kiểm pat in line
            cout << lineno << ": " << Match[0] << '\n';
}
```

Regex_search (line ,matches,pat) tìm kiếm bất kỳ dòng nào khớp với biểu thức chính quy đư ợc lư u trữ trong pat và nếu nó tìm thấy bất kỳ kết quả khớp nào, nó sẽ lư u chúng trong các kết quả khớp. Nếu không tìm thấy kết quả phù hợp, regex_search(line ,matches,pat) trả về false. Biến match có kiểu smatch. ''s'' là viết tắt của ''sub'' và một smatch là một véc-tơ của các kết quả phụ. Phần tử đầu tiên, ở đây khớp với [0], là khớp hoàn chỉnh.

Để có mô tả đầy đủ hơ n, xem Chư ơ ng 37.

5.6 Toán

C ++ không đư ợc thiết kế chủ yếu với tính toán số. Tuy nhiên, C++ đư ợc sử dụng nhiều cho tính toán số và thư viện chuẩn phản ánh điều đó.

5.6.1 Các hàm và thuật toán toán học

Trong <cmath>, chúng tôi tìm thấy ''các hàm toán học thông thư ờng'', chẳng hạn như sqrt(), log() và sin() cho các đối số kiểu float, double và long double (§40.3). Phiên bản số phức của các hàm này đư ợc tìm thấy trong <complex> (§40.4).

Trong <numeric>, chúng tôi tìm thấy một tập hợp nhỏ các thuật toán số tổng quát, chẳng hạn như tích lũy(). Ví dụ:

```
khoảng trống
f() {
      danh sách<đôi> lst {1, 2, 3, 4, 5, 9999.99999};
      auto s = tích lũy(lst.begin(),lst.end(),0.0); // cout <<   tính tổng
      s << '\n';                                          // in 10014.9999
}
```

Các thuật toán này hoạt động cho mọi chuỗi thư viện chuẩn và có thể có các hoạt động đư ợc cung cấp dư ới dạng đối số (§40.6).

Machine Translated by Google

5.6.2 Số phức

Thư viện chuẩn hỗ trợ một họ các loại số phức dọc theo các dòng của lớp phức được mô tả trong §2.3. Để hỗ trợ các số phức trong đó các số vô hướng là các số dấu phẩy động có độ chính xác đơn (số float), số dấu phẩy động có độ chính xác kép (nhân đôi), v.v., tổ hợp thư viện tiêu chuẩn là một mẫu:

```
mẫu <typename Vô hướng> lớp phức tạp
{ public: phức tạp
(const
        Scalar& re ={}, const Scalar& im ={}); //
           ...
};
```

Các phép tính số học thông thường và các hàm toán học phổ biến nhất được hỗ trợ cho các số phức. Ví dụ:

```
void f(phức<float> fl, phức<double> db) {

        phức hợp<dài gấp đôi> ld {fl+sqrt(db)}; db += fl  3; fl
        = pow(1/fl,2);

        // ...
}
```

Các hàm sqrt() và pow() (lũy thừa) nằm trong số các hàm toán học thông thường được định nghĩa trong <complex>. Để biết thêm chi tiết, xem §40.4.

5.6.3 Số ngẫu nhiên

Số ngẫu nhiên hữu ích trong nhiều ngữ cảnh, chẳng hạn như thử nghiệm, trò chơi, mô phỏng và bảo mật. Sự đa dạng của các lĩnh vực ứng dụng được phản ánh trong nhiều lựa chọn trình tạo số ngẫu nhiên được cung cấp bởi thư viện tiêu chuẩn trong <random>. Một bộ tạo số ngẫu nhiên bao gồm hai phần: [1] một bộ máy tạo ra một chuỗi các giá trị ngẫu nhiên hoặc giả ngẫu nhiên. [2] một phân phối ánh xạ các giá trị đó thành một phân phối toán học trong một phạm vi. Ví dụ về phân phối là thống nhất_int_phân phối (trong đó tất cả các số nguyên được tạo ra đều có khả năng như nhau), phân phối_bình thường (''đường cong hình chuông'') và phân phối theo cấp số nhân (tăng trưởng theo cấp số nhân); mỗi cho một số phạm vi được chỉ định. Ví dụ:

```
sử dụng my_engine = default_random_engine; sử dụng my_distribution
= uniform_int_distribution<>;

my_engine lại {};
my_distribution one_to_six {1,6}; auto die =
bind(one_to_six,re);

int x = die();
```

```
// loại   loại động cơ
// phân phối

   phân phối công cụ mặc
// định mà ints 1..6 cho bản đồ làm cho trình tạo trở thành
//

// tung xúc xắc :x   trở thành giá trị trong [1:6] a
```

Hàm thư viện chuẩn bind() tạo đối tượng hàm sẽ gọi đối số đầu tiên của nó (ở đây, one_to_six) đưa đối số thứ hai (ở đây, re) làm đối số (§33.5.1). Do đó, một cuộc gọi die() tương đương với một cuộc gọi one_to_six(re).

Machine Translated by Google

Nhờ sự chú ý không khoan như ợng của nó đối với tính tổng quát và hiệu suất, một chuyên gia đã coi thành phần số ngẫu nhiên của thư viện tiêu chuẩn là ''điều mà mọi thư viện số ngẫu nhiên muốn trở thành khi nó lớn lên''. Tuy nhiên, nó khó có thể đư ợc coi là ''thân thiện với ngư ời mới bắt đầu''. '' Các câu lệnh sử dụng làm cho những gì đang đư ợc thực hiện rõ ràng hơ n một chút. Thay vào đó, tôi chỉ có thể viết:

```
auto die = bind(uniform_int_distribution<>{1,6}, default_random_engine{});
```

Phiên bản nào dễ đọc hơ n hoàn toàn phụ thuộc vào ngữ cảnh và ngư ời đọc.

Đối với ngư ời mới (thuộc bất kỳ nền tảng nào), giao diện hoàn toàn chung với thư viện số ngẫu nhiên có thể là một trở ngại nghiêm trọng. Một trình tạo số ngẫu nhiên thống nhất đơ n giản thư ờng là đủ để bắt đầu. Ví dụ:

```
Rand_int rnd{1,10}; int x =                    // làm một    trình tạo số ngẫu nhiên cho [1:10] là một số
rnd();                              // X trong [1:10]
```

Vì vậy, làm thế nào chúng ta có thể có đư ợc điều đó? Chúng ta phải lấy một cái gì đó như die() bên trong một lớp Rand_int:

```
lớp Rand_int { công
khai:
        Rand_int(int thấp, int cao) :dist{low,high} { } int operator()
        () { return dist(re); } riêng tư :                    // vẽ int một

default_random_engine re;
        uniform_int_distribution<> dist;
};
```

Định nghĩa đó vẫn là ''cấp độ chuyên gia'', như ng việc sử dụng Rand_int () có thể quản lý đư ợc trong tuần đầu tiên của khóa học C++ dành cho ngư ời mới. Ví dụ:

```
int main() {

        Rand_int rnd{0,4};                // làm một    đồng phục trình tạo số ngẫu nhiên

        vector<int> biểu đồ(5); for             // tạo véc tơ cỡ 5 a
        (int i=0; i!=200; ++i) +
              +histogram[rnd()];                // điền vào biểu đồ với tần số của các số [0:4]

        for (int i = 0; i!=mn.size(); ++i) { cout         // viết ra một   thanh biểu đồ
              << i << '\t'; for
              (int j=0; j!=mn[i]; ++j) cout << '  '; cout<<endl;

        }
}
```

Đầu ra là một phân phối thống nhất (chắc chắn là nhàm chán) (với sự thay đổi thống kê hợp lý):

```
0
1
2
3
4
```

Machine Translated by Google

Không có thư viện đồ họa tiêu chuẩn cho C++, vì vậy tôi sử dụng ''đồ họa ASCII.'' Rõ ràng, có rất nhiều thư viện GUI và đồ họa mã nguồn mở và thư ơ ng mại cho C++, như ng trong cuốn sách này, tôi sẽ giới hạn bản thân mình trong các cơ sở tiêu chuẩn ISO.

Để biết thêm thông tin về số ngẫu nhiên, hãy xem §40.7.

5.6.4 Số học vectơ

Vectơ đư ợc mô tả trong §4.4.1 đư ợc thiết kế để trở thành một cơ chế chung để giữ giá trị, linh hoạt và phù hợp với kiến trúc của bộ chứa, bộ lặp và thuật toán . Tuy nhiên, nó không hỗ trợ các phép toán vectơ . Việc thêm các phép toán như vậy vào véc-tơ sẽ dễ dàng, như ng tính tổng quát và tính linh hoạt của nó ngăn cản các tối ư u hóa thư ờng đư ợc coi là cần thiết cho công việc số nghiêm túc. Do đó, thư viện tiêu chuẩn cung cấp (trong <valarray>) một khuôn mẫu giống như vectơ , đư ợc gọi là valarray, ít chung chung hơ n và dễ dàng hơ n để tối ư u hóa cho tính toán số:

```
mẫu<typename T> lớp
valarray {
      // ...
};
```

Các phép toán số học thông thư ờng và các hàm toán học phổ biến nhất đư ợc hỗ trợ cho valarrays. Ví dụ:

```
void f(valarray<double>& a1, valarray<double>& a2) {

      valarray<double> a = a1  3.14+a2/a1; a2            // con số  toán tử ic *, +, / và mảng      =
      += a1  3,14; a
      = abs(a);
      gấp đôi d =
      a2[7]; //
}
```

Để biết thêm chi tiết, xem §40.5. Đặc biệt, valarray cung cấp quyền truy cập sải chân để giúp thực hiện các phép tính đa chiều.

5.6.5 Giới hạn số

Trong <limits>, thư viện chuẩn cung cấp các lớp mô tả các thuộc tính của các kiểu dựng sẵn – chẳng hạn như số mũ tối đa của một số thực hoặc số byte trong một số nguyên; xem §40.2. Ví dụ: chúng ta có thể khẳng định rằng một char đã đư ợc ký:

```
static_assert(numeric_limits<char>::is_signed,"ký tự không dấu!");
static_assert(100000<numeric_limits<int>::max(),"số nguyên nhỏ!");
```

Lư u ý rằng xác nhận thứ hai (chỉ) hoạt động vì numeric_limits<int>::max() là một hàm constexpr (§2.2.3, §10.4).

Machine Translated by Google

5.7 Lời khuyên

[1] Sử dụng tay cầm tài nguyên để quản lý tài nguyên (RAII); §5.2.

[2] Sử dụng unique_ptr để chỉ các đối tư ợng thuộc loại đa hình; §5.2.1.

[3] Sử dụng shared_ptr để chỉ các đối tư ợng đư ợc chia sẻ; §5.2.1.

[4] Sử dụng cơ chế an toàn loại cho đồng thời; §5.3.

[5] Giảm thiểu việc sử dụng dữ liệu đư ợc chia sẻ; §5.3.4.

[6] Đừng chọn dữ liệu đư ợc chia sẻ để liên lạc vì ''hiệu quả'' mà không cần suy nghĩ và tốt nhất là không nên đo lư ờng; §5.3.4.

[7] Suy nghĩ về các nhiệm vụ đồng thời, thay vì theo chủ đề; §5.3.5.

[8] Một thư viện không nhất thiết phải lớn hoặc phức tạp để trở nên hữu ích; §5.4.

[9] Hẹn giờ chư ơ ng trình của bạn trư ớc khi đư a ra tuyên bố về hiệu quả; §5.4.1.

[10] Bạn có thể viết mã để phụ thuộc rõ ràng vào thuộc tính của các loại; §5.4.2.

[11] Sử dụng biểu thức chính quy để khớp mẫu đơ n giản; §5.5.

[12] Đừng cố gắng tính toán các con số một cách nghiêm túc mà chỉ sử dụng ngôn ngữ; sử dụng thư viện; §5.6.

[13] Có thể truy cập thuộc tính của các loại số thông qua numeric_limits; §5.6.5.

Machine Translated by Google

Phần II

Cơ sở vật chất

Phần này mô tả các kiểu dựng sẵn của C++ và các tiện ích cơ bản để xây dựng các chương trình từ chúng. Tập hợp con C của C++ được trình bày cùng với sự hỗ trợ bổ sung của C++ cho các phong cách lập trình truyền thống. Nó cũng thảo luận về các phương tiện cơ bản để soạn một chương trình C++ từ các phần logic và vật lý.

chương

6 Loại và Tuyên bố
7 Con trỏ, Mảng và Tham chiếu
8 Cấu trúc, Công đoàn và Phép liệt kê
9 Tuyên bố
10 biểu thức
11 Chọn Thao tác
12 chức năng
13 Xử lý ngoại lệ
14 Không gian tên
15 tệp nguồn và chương trình

Machine Translated by Google

''... Từ lâu, tôi đã nghi ngờ các quyết định của các nhà triết học về mọi chủ đề, và nhận thấy bản thân có khuynh hướng tranh cãi hơn là đồng ý với kết luận của họ. Có một sai lầm mà họ dường như phải chịu trách nhiệm, hầu như không có ngoại lệ; họ hạn chế quá nhiều các nguyên tắc của mình và không tính đến sự đa dạng rộng lớn đó, thứ mà thiên nhiên đã ảnh hưởng rất nhiều đến mọi hoạt động của cô ấy. Khi một triết gia đã từng nắm giữ một nguyên tắc yêu thích, nguyên tắc có lẽ giải thích cho nhiều tác động tự nhiên, thì ông ta mở rộng nguyên tắc đó cho toàn bộ sự sáng tạo, và quy về nó mọi hiện tượng, mặc dù bằng cách lập luận bạo lực và phi lý nhất. ...''

– David Hume,

Tiểu luận, Đạo đức, Chính trị và Văn học. PHẦN I. (1752)

Machine Translated by Google

6

Các loại và Tuyên bố

Sự hoàn hảo chỉ đạt
đư ợc ở điểm sụp đổ.
– CN Parkinson

• Tiêu chuẩn ISO C++
Triển khai; Bộ ký tự nguồn cơ bản • Các loại

Các loại cơ bản; Booleans; Các loại nhân vật; Các kiểu số nguyên; Các kiểu dấu phẩy động; Tiền sửa lỗi và hậu tố; vô hiệu; Kích cỡ; Căn chỉnh • Tuyên bố
Cấu trúc của Tuyên bố; Khai Báo Nhiều Tên; Tên; Phạm vi; Khởi tạo;
Suy diễn Loại: auto và decltype() • Đối tư ợng và Giá trị
Lvalues và Rvalues; Vòng đời của các đối tư ợng • Nhập bí danh • Lời khuyên

6.1 Tiêu chuẩn ISO C++

Ngôn ngữ C++ và thư viện chuẩn đư ợc xác định theo tiêu chuẩn ISO của chúng: ISO/IEC 14882:2011. Trong cuốn sách này, các tham chiếu đến tiêu chuẩn có dạng §iso.23.3.6.1. Trong trư ờng hợp văn bản của cuốn sách này đư ợc coi là không chính xác, không đầy đủ hoặc có thể sai, hãy tham khảo tiêu chuẩn. Như ng đừng mong đợi tiêu chuẩn là một hư ớng dẫn hoặc có thể dễ dàng truy cập bởi những ngư ời không phải là chuyên gia.

Bản thân việc tuân thủ nghiêm ngặt tiêu chuẩn thư viện và ngôn ngữ C++ không đảm bảo mã tốt hoặc thậm chí mã di động. Tiêu chuẩn không nói liệu một đoạn mã là tốt hay xấu; nó chỉ đơ n giản nói những gì một lập trình viên có thể và không thể dựa vào từ việc triển khai. Thật dễ dàng để viết các chư ơ ng trình phù hợp với tiêu chuẩn hoàn toàn khủng khiếp và hầu hết các chư ơ ng trình trong thế giới thực đều dựa vào các tính năng mà tiêu chuẩn không đảm bảo là có thể mang theo đư ợc. Họ làm như vậy để truy cập giao diện hệ thống và

Machine Translated by Google

các tính năng phần cứng không thể thể hiện trực tiếp trong C++ hoặc yêu cầu phụ thuộc vào các chi tiết triển khai cụ thể.

Nhiều điều quan trọng đư ợc coi là thực hiện xác định theo tiêu chuẩn. Điều này có nghĩa là mỗi triển khai phải cung cấp một hành vi cụ thể, đư ợc xác định rõ cho một cấu trúc và hành vi đó phải đư ợc ghi lại. Ví dụ:

```
ký tự không dấu c1 = 64; ký tự // đư ợc xác định rõ ràng: a  char có ít nhất 8 bit và có thể  luôn giữ 64 do triển
không dấu c2 = 1256;             // khai xác định: cắt bớt nếu một ký tự chỉ có 8 bit
```

Việc khởi tạo c1 đư ợc xác định rõ vì một char phải có ít nhất 8 bit. Tuy nhiên, hành vi khởi tạo c2 đư ợc xác định theo triển khai vì số lư ợng bit trong một ký tự đư ợc xác định theo triển khai. Nếu char chỉ có 8 bit, giá trị 1256 sẽ bị cắt bớt thành 232 (§10.5.2.1).

Hầu hết các tính năng do triển khai xác định liên quan đến sự khác biệt trong phần cứng đư ợc sử dụng để chạy chư ơ ng trình.

Các hành vi khác không xác định; nghĩa là, một loạt các hành vi có thể đư ợc chấp nhận, như ng ngư ời thực hiện không bắt buộc phải chỉ định hành vi nào thực sự xảy ra. Thông thư ờng, lý do để cho rằng một số thứ không xác định là hành vi chính xác không thể đoán trư ớc vì những lý do cơ bản. Ví dụ: giá trị chính xác đư ợc trả về bởi new không đư ợc chỉ định. Giá trị của một biến đư ợc gán cho từ hai luồng cũng vậy trừ khi một số cơ chế đồng bộ hóa đã đư ợc sử dụng để ngăn chặn cuộc chạy đua dữ liệu (§41.2).

Khi viết các chư ơ ng trình trong thế giới thực, thư ờng cần phải dựa vào hành vi do triển khai xác định. Hành vi như vậy là cái giá chúng ta phải trả cho khả năng hoạt động hiệu quả trên nhiều hệ thống. Ví dụ, C++ sẽ đơ n giản hơ n nhiều nếu tất cả các ký tự là 8 bit và tất cả các con trỏ là 32 bit. Tuy nhiên, bộ ký tự 16 bit và 32 bit không phải là hiếm và các máy có con trỏ 16 bit và 64 bit đang đư ợc sử dụng rộng rãi.

Để tối đa hóa tính di động, điều khôn ngoan là phải rõ ràng về những tính năng do triển khai xác định mà chúng tôi dựa vào và tách biệt các ví dụ tinh vi hơ n trong các phần đư ợc đánh dấu rõ ràng của chư ơ ng trình. Một ví dụ điển hình của thực tiễn này là trình bày tất cả các phụ thuộc vào kích thư ớc phần cứng dư ới dạng các hằng số và định nghĩa kiểu trong một số tệp tiêu đề. Để hỗ trợ các kỹ thuật như vậy, thư viện chuẩn cung cấp các giới hạn số (§40.2). Có thể kiểm tra nhiều giả định về các tính năng do triển khai xác định bằng cách nêu chúng dư ới dạng xác nhận tĩnh (§2.4.3.3). Ví dụ:

```
static_assert(4<=sizeof(int),"kích thư ớc eof(int) quá nhỏ");
```

Hành vi không xác định là khó chịu hơ n. Một cấu trúc đư ợc coi là không đư ợc xác định theo tiêu chuẩn nếu không có hành vi hợp lý nào đư ợc yêu cầu bởi việc triển khai. Thông thư ờng, một số kỹ thuật triển khai rõ ràng sẽ khiến chư ơ ng trình sử dụng tính năng không xác định hoạt động rất tệ. Ví dụ:

```
kích thư ớc const int = 4  1024;
trang char[siz e];

khoảng trống
f() {
        trang[bạn e+size] = 7; //        không xác định
}
```

Kết quả hợp lý của đoạn mã này bao gồm ghi đè dữ liệu không liên quan và gây ra lỗi/ngoại lệ phần cứng. Việc triển khai không bắt buộc phải lựa chọn trong số các kết quả hợp lý.

Khi sử dụng các trình tối ư u hóa mạnh mẽ, các tác động thực tế của hành vi không xác định có thể trở nên khá khó lư ờng. Nếu tồn tại một tập hợp các lựa chọn thay thế hợp lý và dễ thực hiện, một tính năng đư ợc coi là

Machine Translated by Google

không xác định hoặc xác định triển khai thay vì không xác định.

Đáng để dành thời gian và nỗ lực đáng kể để đảm bảo rằng một chương trình không sử dụng một số thứ được coi là không xác định hoặc không được xác định theo tiêu chuẩn. Trong nhiều trường hợp, các công cụ tồn tại để giúp thực hiện việc này.

6.1.1 Triển khai

Việc triển khai C++ có thể được lưu trữ hoặc độc lập (§iso.17.6.1.3). Một triển khai được lưu trữ bao gồm tất cả các cơ sở thư viện tiêu chuẩn như được mô tả trong tiêu chuẩn (§30.2) và trong cuốn sách này. Việc triển khai độc lập có thể cung cấp ít cơ sở thư viện tiêu chuẩn hơn, miễn là những điều sau đây được cung cấp:

Tiêu đề triển khai độc lập		
các loại	<cstddef>	§10.3.1
thuộc tính thực hiện	<cfloat> <giới hạn> <climits>	§40.2
các kiểu số nguyên	<cstdint>	§43.7
Bắt đầu và kết thúc	<cstdlib>	§43.7
Quản lý bộ nhớ động <mới>		§11.2.3
nhận dạng loại	<typeinfo>	§22.5
xử lý ngoại lệ	§30.4.1.1 <ngoại lệ > §30.3.1	
danh sách trình khởi tạo	<initializer_list> <cstdalign> <cstdarg>	
Hỗ trợ thời gian chạy khác	§12.2.4, §44.3. 4 §35.4.1 <type_traits> §41.3	
Loại đặc điểm nguyên tử	<nguyên tử>	

Các triển khai độc lập dành cho mã chạy chỉ với sự hỗ trợ tối thiểu nhất của hệ điều hành. Nhiều triển khai cũng cung cấp tùy chọn (không chuẩn) để không sử dụng ngoại lệ cho các chương trình thực sự tối thiểu, gần với phần cứng.

6.1.2 Bộ ký tự nguồn cơ bản

Tiêu chuẩn C++ và các ví dụ trong cuốn sách này được viết bằng cách sử dụng bộ ký tự nguồn cơ bản bao gồm các chữ cái, chữ số, ký tự đồ họa và ký tự khoảng trắng từ biến thể Hoa Kỳ của bộ ký tự 7 bit quốc tế ISO 646-1983 được gọi là ASCII (ANSI3.4-1968). Điều này có thể gây ra sự cố cho những người sử dụng C++ trong môi trường có bộ ký tự khác:

- ASCII chứa các ký tự chấm câu và ký hiệu toán tử (chẳng hạn như], { và !) không phải là có sẵn trong một số bộ ký tự.
- Chúng tôi cần một ký hiệu cho các ký tự không có biểu diễn ký tự thuận tiện (chẳng hạn như dòng mới và ''ký tự có giá trị 17'').
- ASCII không chứa các ký tự (chẳng hạn như ñ, Þ và Æ) được sử dụng để viết ngôn ngữ khác với tiếng Anh.

Để sử dụng bộ ký tự mở rộng cho mã nguồn, môi trường lập trình có thể ánh xạ bộ ký tự mở rộng vào bộ ký tự nguồn cơ bản theo một số cách, chẳng hạn như bằng cách sử dụng tên ký tự chung (§6.2.3.2).

Machine Translated by Google

6.2 Các loại

Coi như :

```
x = y+f(2);
```

Để điều này có ý nghĩa trong chư ơ ng trình C++, các tên x, y và f phải đư ợc khai báo phù hợp. Nghĩa là, lập trình viên phải xác định rằng các thực thể có tên x, y và f tồn tại và chúng thuộc loại mà = (gán), + (cộng) và () (gọi hàm), tư ơ ng ứng, có ý nghĩa.

Mỗi tên (mã định danh) trong chư ơ ng trình C++ đều có một loại đư ợc liên kết với nó. Loại này xác định những thao tác nào có thể đư ợc áp dụng cho tên (tức là cho thực thể đư ợc gọi bằng tên) và cách các thao tác đó đư ợc diễn giải. Ví dụ:

```
nổi x;          // là một số nguyên x thi khả
int y = 7;      // dấuphẩy động  ds tổng biable với giá trị ban đầu là 7
phao f(int);    // f là một chức năng lấy MỘT đối số kiểu int và trả về Một số điểm nổi
```

Những tuyên bố này sẽ làm cho ví dụ có ý nghĩa. Vì y đư ợc khai báo là một số nguyên nên nó có thể đư ợc gán cho, đư ợc sử dụng làm toán hạng cho +, v.v. Mặt khác, f đư ợc khai báo là một hàm nhận một số nguyên là đối số của nó, vì vậy nó có thể đư ợc gọi là đã cho bộ đàm 2.

Chư ơ ng này trình bày các kiểu cơ bản (§6.2.1) và các khai báo (§6.3). Các ví dụ của nó chỉ thể hiện các đặc điểm ngôn ngữ; họ không có ý định làm bất cứ điều gì hữu ích. Các ví dụ mở rộng và thực tế hơ n đư ợc lư u lại cho các chư ơ ng sau. Chư ơ ng này chỉ cung cấp những thành phần cơ bản nhất mà từ đó các chư ơ ng trình C++ đư ợc xây dựng. Bạn phải biết những yếu tố này, cộng với thuật ngữ và cú pháp đơ n giản đi kèm với chúng, để hoàn thành một dự án thực sự trong C++ và đặc biệt để đọc mã do ngư ời khác viết. Tuy nhiên, việc hiểu thấu đáo mọi chi tiết đư ợc đề cập trong chư ơ ng này không phải là điều kiện bắt buộc để hiểu các chư ơ ng sau. Do đó, bạn có thể muốn đọc lư ớt qua chư ơ ng này, quan sát các khái niệm chính và quay lại sau khi cần hiểu thêm chi tiết.

6.2.1 Các loại cơ bản

C++ có một tập hợp các kiểu cơ bản tư ơ ng ứng với các đơ n vị lư u trữ cơ bản phổ biến nhất của máy tính và các cách phổ biến nhất sử dụng chúng để lư u giữ dữ liệu: §6.2.2 Một kiểu

Boolean (bool) §6.2.3 Các kiểu ký

tự (chẳng hạn như char và wchar_t) §6.2.4 Kiểu số

nguyên (chẳng hạn như int và long long) §6.2.5 Kiểu

dấu phẩy động (chẳng hạn như double và long double) §6.2.7 Một

loại, void, đư ợc sử dụng để biểu thị sự vắng mặt của thông tin Từ các loại này, chúng ta có thể xây dựng các loại khác bằng cách sử dụng các toán tử khai

báo: §7.2 Các loại con trỏ (chẳng hạn

như int) §7.3 Các loại mảng (chẳng

hạn như char[]) §7.7 Các loại tham chiếu (chẳng hạn như double& và vector<int>&&)

Ngoài ra, ngư ời dùng có thể xác định các loại bổ sung:

§8.2 Cấu trúc dữ liệu và lớp (Chư ơ ng 16) §8.4 Kiểu

liệt kê để biểu diễn các tập giá trị cụ thể (enum và lớp enum)

Machine Translated by Google

Các kiểu Boolean, ký tự và số nguyên được gọi chung là các kiểu tích phân. Các kiểu tích phân và dấu phẩy động được gọi chung là các kiểu số học. Kiểu liệt kê và lớp (Chương 16) được gọi là kiểu do người dùng định nghĩa vì chúng phải được người dùng định nghĩa thay vì có sẵn để sử dụng mà không cần khai báo trước, theo cách của các kiểu cơ bản. Ngược lại, các kiểu cơ bản, con trỏ và tham chiếu được gọi chung là các kiểu dựng sẵn. Thư viện chuẩn cung cấp nhiều kiểu do người dùng định nghĩa (Chương 4, Chương 5).

Các loại tích phân và dấu phẩy động được cung cấp ở nhiều kích cỡ khác nhau để người lập trình có thể lựa chọn dung lượng lưu trữ đã sử dụng, độ chính xác và phạm vi có sẵn để tính toán (§6.2.8). Giả định là một máy tính cung cấp các byte để giữ các ký tự, các từ để giữ và tính toán các giá trị số nguyên, một số thực thể phù hợp nhất cho tính toán dấu phẩy động và các địa chỉ để tham chiếu đến các thực thể đó. Các kiểu cơ bản của C++ cùng với con trỏ và mảng trình bày các khái niệm ở cấp độ máy này cho người lập trình theo cách độc lập hợp lý với việc triển khai.

Đối với hầu hết các ứng dụng, chúng ta có thể sử dụng bool cho các giá trị logic, char cho các ký tự, int cho các giá trị số nguyên và double cho các giá trị dấu phẩy động. Các loại cơ bản còn lại là các biến thể để tối ưu hóa, nhu cầu đặc biệt và khả năng tương thích tốt nhất nên bỏ qua cho đến khi các nhu cầu đó phát sinh.

6.2.2 Boolean

Boolean, bool, có thể có một trong hai giá trị đúng hoặc sai. Boolean được sử dụng để biểu thị kết quả của các hoạt động logic. Ví dụ:

```
void f(int a, int b) {

        bool b1 {a==b}; //
          ...
}
```

Nếu a và b có cùng giá trị, b1 trở thành true; mặt khác, b1 trở thành sai.

Một cách sử dụng phổ biến của bool là loại kết quả của một hàm kiểm tra một số điều kiện (một thuộc tính dự đoán). Ví dụ:

```
bool is_open(Tệp );

bool lớn hơn(int a, int b) { return a>b; }
```

Theo định nghĩa, true có giá trị 1 khi được chuyển đổi thành số nguyên và false có giá trị 0. Ngược lại, số nguyên có thể được chuyển đổi hoàn toàn thành giá trị bool : số nguyên khác 0 chuyển thành true và 0 chuyển thành false. Ví dụ:

```
bool b1 = 7;          7!=0, b trở thành true nên
bool b2{7};         // lỗi :   thu hẹp (§2.2.2, §10.5)

int i1 = đúng;      // i1 trở thành 1
int i2 {true}; //     i2 trở thành 1
```

Nếu bạn muốn sử dụng cú pháp {}-initializer để ngăn thu hẹp, nhưng vẫn muốn chuyển đổi một int thành một bool, bạn có thể nói rõ:

Machine Translated by Google


```
khoảng trống f(int tôi)
{
        bool b {i!=0}; //
          ...
};
```

Trong các biểu thức số học và logic, bool đư ợc chuyển thành int; phép toán số học và logic số nguyên đư ợc thực hiện trên các giá trị đư ợc chuyển đổi. Nếu kết quả cần đư ợc chuyển đổi lại thành bool, thì 0 đư ợc chuyển đổi thành false và giá trị khác 0 đư ợc chuyển đổi thành true. Ví dụ:

```
bool a = đúng; bool
b = đúng;

bool x = a+b; // bool y   a+b là 2, a|| vậy x  trở thành sự thật
= a|b; // bool z =       b là 1, ab  vì thế  trở thành đúng ("||" y trở có nghĩa  "hoặc")
a  b; //                 là 0,       vậy z thành sai
```

Một con trỏ có thể đư ợc chuyển đổi hoàn toàn thành bool (§10.5.2.5). Một con trỏ khác null chuyển thành true; con trỏ có giá trị nullptr chuyển thành false. Ví dụ:

```
khoảng trống g(int  p)
{
        bool b = p; bool                    // hẹp thực sự để đặt thu      SAI
        b2 {p!=nullptr};                    // kiểm tra rõ ràng đối với nullptr

        Nếu p)          // tư ơ ng đư ơ ng với p!=nullptr
              { //...
        }
}
```

Tôi thích if (p) hơ n if (p!=nullptr) vì nó diễn đạt trực tiếp hơ n khái niệm ''nếu p hợp lệ'' và cũng vì nó ngắn hơ n. Hình thức ngắn hơ n để lại ít cơ hội mắc lỗi hơ n.

6.2.3 Các loại ký tự

Có nhiều bộ ký tự và mã hóa bộ ký tự đư ợc sử dụng. C++ cung cấp nhiều loại ký tự phản ánh điều đó – thư ờng gây bối rối – đa dạng: • char: Loại ký tự mặc định,

đư ợc sử dụng cho văn bản chư ơ ng trình. Một ký tự đư ợc sử dụng cho bộ ký tự của trình triển khai và thư ờng là 8 bit. • signed char: Giống

như char, như ng đảm bảo là có dấu, nghĩa là có khả năng chứa cả giá trị âm và dư ơ ng. • unsigned char: Giống như

char, như ng đảm bảo là unsigned. • wchar_t: Đư ợc cung cấp để

giữ các ký tự của bộ ký tự lớn hơ n như Unicode (xem §7.3.2.2).

Kích thư ớc của wchar_t đư ợc triển khai xác định và đủ lớn để chứa bộ ký tự lớn nhất đư ợc ngôn ngữ triển khai hỗ trợ (Chư ơ ng 39). • char16_t: Một loại để

giữ bộ ký tự 16-bit, chẳng hạn như UTF-16. • char32_t: Một loại để giữ

bộ ký tự 32-bit, chẳng hạn như UTF-32.

Đây là sáu loại riêng biệt (mặc dù thực tế là hậu tố _t thư ờng đư ợc sử dụng để biểu thị bí danh; §6.5). Trên mỗi lần triển khai, loại char sẽ giống hệt với loại char đã ký hoặc chư a ký

Machine Translated by Google

char, như ng ba tên này vẫn đư ợc coi là các loại riêng biệt.

Một biến char có thể chứa một ký tự của bộ ký tự triển khai. Ví dụ:

```
char ch = 'a';
```

Hầu như phổ biến, một char có 8 bit để nó có thể chứa một trong 256 giá trị khác nhau. Thông thư ờng, bộ ký tự là một biến thể của ISO-646, chẳng hạn như ASCII, do đó cung cấp các ký tự xuất hiện trên bàn phím của bạn. Nhiều vấn đề phát sinh từ thực tế là bộ ký tự này chỉ đư ợc chuẩn hóa một phần.

Các biến thể nghiêm trọng xảy ra giữa các bộ ký tự hỗ trợ các ngôn ngữ tự nhiên khác nhau và giữa các bộ ký tự hỗ trợ cùng một ngôn ngữ tự nhiên theo những cách khác nhau. Ở đây, chúng ta chỉ quan tâm đến việc những khác biệt đó ảnh hư ởng như thế nào đến các quy tắc của C++. Vấn đề lớn hơ n và thú vị hơ n về cách lập trình trong môi trư ờng đa ngôn ngữ, nhiều ký tự nằm ngoài phạm vi của cuốn sách này, mặc dù nó đư ợc ám chỉ ở một số chỗ (§6.2.3, §36.2.1, Chư ơ ng 39) .

Sẽ an toàn khi giả định rằng bộ ký tự triển khai bao gồm các chữ số thập phân, 26 ký tự chữ cái tiếng Anh và một số ký tự dấu câu cơ bản. Sẽ không an toàn nếu cho rằng: • Không có nhiều hơ n 127 ký tự trong bộ

ký tự 8 bit (ví dụ: một số bộ cung cấp 255
nhân vật).

• Không có nhiều ký tự chữ cái hơ n tiếng Anh cung cấp (hầu hết các ngôn ngữ châu Âu
cung cấp thêm, ví dụ: æ, þ và ß).

• Các ký tự chữ cái liền kề nhau (EBCDIC để khoảng trống giữa 'i' và 'j'). • Mọi ký tự
dùng để viết C++ đều có sẵn (ví dụ: một số bộ ký tự quốc gia không cung cấp {, }, [,], | và \). •
Một char phù hợp với

1 byte. Có những bộ xử lý nhúng không có phần cứng truy cập byte mà char là 4 byte. Ngoài ra, ngư ời
ta có thể sử dụng hợp lý mã hóa Unicode 16 bit cho các ký tự cơ bản.

Bất cứ khi nào có thể, chúng ta nên tránh đư a ra các giả định về việc biểu diễn các đối tư ợng. Quy tắc chung này áp dụng ngay cả cho các ký tự.

Mỗi ký tự có một giá trị số nguyên trong bộ ký tự đư ợc triển khai sử dụng. Ví dụ: giá trị của 'b' là 98 trong bộ ký tự ASCII. Đây là một vòng lặp xuất giá trị số nguyên của bất kỳ ký tự nào bạn muốn nhập:

```
khoảng trống giá trị ()
{
      for (char c; cin >> c; )
            cout << "giá trị của '" "' là " << int{c} << '\n'; <<c<<
}
```

Ký hiệu int{c} cung cấp giá trị số nguyên cho một ký tự c (''số int mà chúng ta có thể tạo từ c''). Khả năng chuyển đổi một ký tự thành một số nguyên đặt ra câu hỏi: một ký tự ký tự hay không ký? 256 giá trị đư ợc biểu thị bằng một byte 8 bit có thể đư ợc hiểu là các giá trị từ 0 đến 255 hoặc các giá trị từ 127 đến 127. Không, không phải 128 đến 127 như ngư ời ta có thể mong đợi: tiêu chuẩn C++ để ngỏ khả năng của phần cứng bổ sung của một ngư ời và điều đó loại bỏ một giá trị; do đó, việc sử dụng 128 là không khả dụng. Thật không may, sự lựa chọn đã ký hoặc chư a ký cho một char đơ n giản đư ợc xác định triển khai. C++ cung cấp hai loại mà câu trả lời là xác định: sign char, có thể chứa ít nhất các giá trị 127 đến 127 và unsigned char, có thể chứa ít nhất các giá trị từ 0 đến 255.

Machine Translated by Google

May mắn thay, sự khác biệt chỉ quan trọng đối với các giá trị nằm ngoài phạm vi từ 0 đến 127 và các ký tự phổ biến nhất nằm trong phạm vi đó.

Các giá trị bên ngoài phạm vi đó đư ợc lư u trữ trong một char đơ n giản có thể dẫn đến các vấn đề về tính di động tinh vi. Nhìn thấy §6.2.3.1 nếu bạn cần sử dụng nhiều loại char hoặc nếu bạn lư u số nguyên trong biến char .

Lư u ý rằng các loại ký tự là các loại tích phân (§6.2.1) để số học và logic bit áp dụng các hoạt động (§10.3). Ví dụ:

```
chữ số trống()
{
      for (int i=0; i!=10; ++i)
            cout << static_cast<char>('0'+i);
}
```

Đây là một cách viết mư ời chữ số vào cout. Ký tự chữ '0' đư ợc chuyển đổi thành giá trị nguyên của nó và i đư ợc thêm vào. Kết quả int sau đó đư ợc chuyển đổi thành char và đư ợc ghi vào cout. Đồng bằng '0'+i là một int, vì vậy nếu tôi bỏ qua static_cast<char>, đầu ra sẽ giống như 48, 49, v.v., thay vì 0, 1, v.v.

6.2.3.1 Ký tự có dấu và không dấu

Nó đư ợc xác định theo triển khai cho dù một char đơ n giản đư ợc coi là đã ký hay chư a ký. Điều này mở ra khả năng xảy ra một số bất ngờ khó chịu và phụ thuộc vào việc triển khai. Ví dụ:

```
ký tự c = 255; //    255 là ''tất cả những cái một,'' thập lục phân 0xFF
int i = c;
```

Giá trị của i sẽ là bao nhiêu? Thật không may, câu trả lời là không xác định. Khi triển khai với byte 8 bit, câu trả lời phụ thuộc vào ý nghĩa của mẫu bit char ''tất cả'' khi đư ợc mở rộng thành int. Trên máy có ký tự không ký, câu trả lời là 255. Trên máy có ký tự ký tự, câu trả lời là -1. Trong trư ờng hợp này, trình biên dịch có thể cảnh báo về việc chuyển đổi chữ 255 thành giá trị char 1. Tuy nhiên, C++ không cung cấp một cơ chế chung để phát hiện loại vấn đề này. Một giải pháp là tránh char đơ n giản và chỉ sử dụng các loại char cụ thể . Thật không may, một số chức năng thư viện tiêu chuẩn, chẳng hạn như strcmp(), chỉ nhận ký tự đơ n giản (§43.4).

Một char phải hoạt động giống hệt với char đã ký hoặc char chư a ký. Tuy nhiên, ba các kiểu ký tự là khác nhau, vì vậy bạn không thể kết hợp các con trỏ với các kiểu ký tự khác nhau . Ví dụ:

```
void f(char c, char sc đã ký, char chư a ký uc) {

      char   pc = &uc;                    // lỗi: không chuyển đổi con trỏ
      char   psc = pc; char không         trỏ lỗi chuyển đổi con
      dấu   puc = pc; psc = puc;          // lỗi: không : chuyển đổi con trỏ
                                          // trỏ không :lỗi chuyển đổi con
}
```

Các biến của ba loại ký tự có thể đư ợc gán tự do cho nhau. Tuy nhiên, việc gán một giá trị quá lớn cho một char đã ký (§10.5.2.1) vẫn chư a đư ợc xác định. Ví dụ:

```
void g(char c, char sc đã ký, char chư a ký uc) {

      c = 255; //    đư ợc xác định theo triển khai nếu các ký tự đơ n giản đư ợc ký và có 8 bit là
```

Machine Translated by Google


```
        c = sc;        // được rồi
        c = úc; // sc   được xác định theo triển khai nếu ký tự đơn giản được ký và nếu giá trị của uc quá lớn thì
        = uc; // uc =   triển khai được xác định nếu giá trị của uc quá lớn
        sc; // //       OK: chuyển đổi thành triển khai
        sc = c;         không dấu được xác định nếu ký tự đơn giản là   không dấu và nếu giá trị của c quá lớn
        úc = c;       // OK: chuyển đổi thành không dấu
    }
```

Để cụ thể, giả sử rằng một `char` là 8 bit:

```
    ký char sc =  160; ký tự không
    dấu uc = sc; // cout << uc;          úc == 116 (vì 256-160==116)
                                      // in 't'

    số lượng ký tự [256];             // giả định Ký tự 8 bit
    ++count[sc]; +                    // thảm họa có thể xảy ra: truy cập ngoài phạm vi
    +đếm[uc];                         // được rồi
```

Không có vấn đề và sự nhầm lẫn tiềm ẩn nào xảy ra nếu bạn sử dụng `char` đơn giản xuyên suốt và tránh các giá trị ký tự âm.

6.2.3.2 Ký tự chữ

Một ký tự chữ là một ký tự đơn nằm trong dấu nháy đơn, ví dụ: `'a'` và `'0'`. Loại ký tự chữ là `char`. Một ký tự chữ có thể được chuyển đổi hoàn toàn thành giá trị số nguyên của nó trong bộ ký tự của máy chạy chương trình C++. Ví dụ: nếu bạn đang chạy ning trên một máy sử dụng bộ ký tự ASCII, giá trị của `'0'` là `48`. Việc sử dụng ký tự bằng chữ thay vì ký hiệu thập phân giúp chương trình dễ mang theo hơn.

Một vài ký tự có tên tiêu chuẩn sử dụng dấu gạch chéo ngược, \ , như một ký tự thoát:

Tên	Tên ASCII	Tên C++
Dòng mới	NL (LF)	\N
Tab ngang	HT	\t
tab dọc	VT	\TRONG
xóa lùi	BS	\b
Vận chuyển trở lại	CR	\ r
Thức ăn dạng	FF	\f
Báo động	BÍ MẬT	\Một
dấu gạch chéo ngược		\\
Dấu chấm hỏi	\ ?	\?
trích dẫn đơn	'	\'
báo giá kép	"	\"
số bát phân	ooo	\ooo
Số thập lục phân hhh		\xhhhh...

Mặc dù xuất hiện, đây là những nhân vật đơn lẻ.

Chúng ta có thể biểu diễn một ký tự từ bộ ký tự triển khai dưới dạng số bát phân một, hai hoặc ba chữ số (\ theo sau là chữ số bát phân) hoặc dưới dạng số thập lục phân (\x theo sau bởi

Machine Translated by Google

chữ số thập lục phân). Không có giới hạn về số lư ợng chữ số thập lục phân trong chuỗi. Một chuỗi các chữ số bát phân hoặc thập lục phân đư ợc kết thúc bằng ký tự đầu tiên không phải là chữ số bát phân hoặc chữ số thập lục phân tư ơ ng ứng. Ví dụ:

Bát phân	Thập lục phân	Thập lục phân	ASCII
'\6'	'\x6'	6	ACK
'\60'	'\x30'	48	
'\137'	'\x05f'	95	'0' '_'

Điều này giúp có thể biểu diễn mọi ký tự trong bộ ký tự của máy và đặc biệt là nhúng các ký tự đó vào chuỗi ký tự (xem §7.3.2). Việc sử dụng bất kỳ ký hiệu số nào cho các ký tự khiến chư ơ ng trình không thể chuyển đổi đư ợc trên các máy có bộ ký tự khác nhau.

Có thể đặt nhiều hơ n một ký tự trong một ký tự chữ, ví dụ: 'ab'. Việc sử dụng như vậy là cổ xư a, phụ thuộc vào việc triển khai và tốt nhất nên tránh. Loại của một chữ nhiều ký tự như vậy là int.

Khi nhúng một hằng số vào một chuỗi bằng cách sử dụng ký hiệu bát phân, bạn nên luôn sử dụng ba chữ số cho số đó. Ký hiệu đủ khó để đọc mà không phải lo lắng về việc ký tự sau hằng có phải là chữ số hay không. Đối với hằng số thập lục phân, hãy sử dụng hai chữ số.
Hãy xem xét các ví dụ sau:

```
char v1[] = "a\xah\129"; char v2[] = "a\xah\127"; char v3[] = "a\xad\127"; char v4[] = "a\xad\0127";
```

```
// 6 ký tự: 'a' '\xa' 'h' '\12' '9' '\0'
   5 ký tự: 'a' '\xa' 'h' '\127' '\0' 4 ký
// tự: 'a' '\xad' '\127' '\0'
// 5 ký tự: 'a' '\xad' '\012' '7' '\0'
```

Các ký tự rộng có dạng L'ab' và thuộc loại wchar_t. Số lư ợng ký tự giữa các dấu ngoặc kép và ý nghĩa của chúng đư ợc xác định theo cách triển khai.

Một chư ơ ng trình C++ có thể thao tác các bộ ký tự phong phú hơ n nhiều so với bộ ASCII 127 ký tự, chẳng hạn như Unicode. Chữ của các bộ ký tự lớn hơ n như vậy đư ợc trình bày dư ới dạng chuỗi bốn hoặc tám chữ số thập lục phân đứng trư ớc chữ U hoặc chữ u. Ví dụ:

```
U'\UFADEBEEF'
u'\uDEAD'
u'\xDEAD'
```

Ký hiệu ngắn hơ n u'\uXXXX' tư ơ ng đư ơ ng với U'\U0000XXXX' cho bất kỳ chữ số thập lục phân X nào. Một số chữ số thập lục phân khác với bốn hoặc tám là một lỗi từ vựng. Ý nghĩa của số thập phân hexa đư ợc xác định theo tiêu chuẩn ISO/IEC 10646 và các giá trị như vậy đư ợc gọi là tên ký tự chung. Trong tiêu chuẩn C++, tên ký tự chung đư ợc mô tả trong §iso.2.2, §iso.2.3, §iso.2.14.3, §iso.2.14.5 và §iso.E.

6.2.4 Các kiểu số nguyên

Giống như char, mỗi kiểu số nguyên có ba dạng: ''plain'' int, sign int và unsigned int. Ngoài ra, số nguyên có bốn kích cỡ: short int, ''plain'' int, long int và long long int. Một int dài có thể đư ợc gọi là dài đơ n giản và một int dài dài có thể đư ợc gọi là dài dài đơ n giản. Tư ơ ng tự, short là một từ đồng nghĩa với short int, unsigned cho unsigned int, và sign cho signed int. Không, không có long short int tư ơ ng đư ơ ng với int.

Machine Translated by Google

Các loại số nguyên không dấu là lý tư ởng cho việc sử dụng coi lư u trữ dư ới dạng một mảng bit. Sử dụng unsigned thay vì int để có thêm một bit để biểu diễn các số nguyên dư ơ ng hầu như không bao giờ là một ý tư ởng hay. Nỗ lực đảm bảo rằng một số giá trị là dư ơ ng bằng cách khai báo các biến không dấu thư ờng sẽ bị các quy tắc chuyển đổi ngầm định (§10.5.1, §10.5.2.1) đánh bại.

Không giống như ký tự đơ n giản , int đơ n giản luôn đư ợc ký. Các loại int đã ký chỉ đơ n giản là các từ đồng nghĩa rõ ràng hơ n cho các đối tác int đơ n giản của chúng , thay vì các loại khác nhau.

Nếu bạn cần kiểm soát chi tiết hơ n đối với kích thư ớc số nguyên, bạn có thể sử dụng bí danh từ <cstdint> (§43.7), chẳng hạn như int64_t (số nguyên có dấu chính xác 64 bit), uint_fast16_t (số nguyên không dấu có chính xác 8 bit, đư ợc cho là nhanh nhất như vậy số nguyên) và int_least32_t (số nguyên đã ký có ít nhất 32 bit, giống như int đơ n giản). Các loại số nguyên đơ n giản có kích thư ớc tối thiểu đư ợc xác định rõ (§6.2.8), do đó, <cstdint> đôi khi dư thừa và có thể bị lạm dụng.

Ngoài các kiểu số nguyên tiêu chuẩn, việc triển khai có thể cung cấp các kiểu số nguyên mở rộng (có dấu và không dấu). Các loại này phải hoạt động giống như số nguyên và đư ợc coi là loại số nguyên khi xem xét chuyển đổi và giá trị số nguyên, như ng chúng thư ờng có phạm vi lớn hơ n (chiếm nhiều không gian hơ n).

6.2.4.1 Số nguyên

Chữ số nguyên có ba dạng: thập phân, bát phân và thập lục phân. Chữ thập phân đư ợc sử dụng phổ biến nhất và trông giống như bạn mong đợi:

```
7 1234 976 12345678901234567890
```

Trình biên dịch phải cảnh báo về các ký tự quá dài để biểu diễn, như ng lỗi chỉ đư ợc đảm bảo cho các trình khởi tạo {} (§6.3.5).

Một chữ bắt đầu bằng số 0 theo sau là x hoặc X (0x hoặc 0X) là một số thập lục phân (cơ số 16). MỘT nghĩa đen bắt đầu bằng 0 như ng không theo sau bởi x hoặc X là một số bát phân (cơ số 8). Ví dụ:

Thập lục bát phân	Thập lục phân	
	0	0x0
2	02	0x2
63	077	0x3f
83	0123	0x63

Các chữ cái a, b, c, d, e và f, hoặc chữ hoa tư ơ ng đư ơ ng của chúng, đư ợc sử dụng để đại diện cho 10, 11, 12, 13, 14 và 15 tư ơ ng ứng. Các ký hiệu bát phân và thập lục phân hữu ích nhất để thể hiện các mẫu bit. Sử dụng các ký hiệu này để thể hiện các số chính hãng có thể dẫn đến những điều bất ngờ. Ví dụ: trên một máy có int đư ợc biểu diễn dư ới dạng số nguyên 16 bit bù hai, 0xffff là số thập phân âm 1. Nếu có nhiều bit hơ n đư ợc sử dụng để biểu diễn một số nguyên, thì nó sẽ là số thập phân dư ơ ng 65535.

Hậu tố U có thể đư ợc sử dụng để viết các chữ không dấu một cách rõ ràng . Tư ơ ng tự, hậu tố L có thể đư ợc sử dụng để viết chữ dài rõ ràng . Ví dụ: 3 là int, 3U là unsigned int và 3L là long int.

Sự kết hợp của các hậu tố đư ợc cho phép. Ví dụ:

```
cout << 0xF0UL << '' << 0LU << '\n';
```

Machine Translated by Google

Nếu không có hậu tố nào đư ợc cung cấp, trình biên dịch sẽ cung cấp cho một số nguyên một loại phù hợp dựa trên giá trị của nó và kích thư ớc số nguyên của triển khai (§6.2.4.2).

Bạn nên giới hạn việc sử dụng các hằng số không rõ ràng đối với một số bộ khởi tạo const (§7.5), constexpr (§10.4) và bộ đếm (§8.4) đư ợc nhận xét tốt.

6.2.4.2 Các loại số nguyên

Nói chung, kiểu của một số nguyên phụ thuộc vào hình thức, giá trị và hậu tố của nó:

- Nếu là số thập phân và không có hậu tố, thì nó có kiểu đầu tiên trong số các kiểu này, trong đó giá trị của nó có thể là rep bực bội: int, long int, long long int.
- Nếu nó là bát phân hoặc thập lục phân và không có hậu tố, thì nó có kiểu đầu tiên trong số những kiểu này mà giá trị của nó có thể đư ợc biểu diễn: int, unsigned int, long int, unsigned long int, long long int, unsigned long long int.
- Nếu nó có hậu tố là u hoặc U, thì loại của nó là loại đầu tiên trong số các loại này mà giá trị của nó có thể đư ợc thay thế đã gửi: unsigned int, unsigned long int, unsigned long long int.
- Nếu nó là số thập phân và có hậu tố là l hoặc L, thì loại của nó là loại đầu tiên trong số các loại này mà giá trị của nó có thể đư ợc biểu diễn: long int, long long int.
- Nếu nó là bát phân hoặc thập lục phân và có hậu tố là l hoặc L, thì kiểu của nó là kiểu đầu tiên trong số những kiểu này mà giá trị của nó có thể đư ợc biểu diễn: long int, unsigned long int, long long int, unsigned long long int. • Nếu nó có hậu tố là ul, lu, uL, Lu, Ul, lU, UL hoặc LU, thì kiểu của nó là kiểu đầu tiên trong số những kiểu này mà giá trị của nó có thể đư ợc biểu diễn: unsigned long int, unsigned long long int.
- Nếu là số thập phân và có hậu tố là ll hoặc LL, kiểu của nó là long long int. • Nếu nó là hệ bát phân hoặc thập lục phân và có hậu tố là ll hoặc LL, thì kiểu của nó là kiểu đầu tiên trong số các kiểu này trong mà giá trị của nó có thể đư ợc biểu diễn: long long int, unsigned long long int. • Nếu nó có hậu tố là llu, llU, ull, Ull, LLu, LLU, uLL, hoặc ULL, kiểu của nó là unsigned long long int.

Ví dụ: 100000 thuộc loại int trên máy có số nguyên 32 bit như ng thuộc loại long int trên máy có số nguyên 16 bit và độ dài 32 bit . Tư ơ ng tự, 0XA000 thuộc loại int trên máy có số nguyên 32 bit như ng thuộc loại unsigned int trên máy có số nguyên 16 bit . Có thể tránh các phụ thuộc triển khai này bằng cách sử dụng các hậu tố: 100000L thuộc loại long int trên tất cả các máy và 0XA000U thuộc loại unsigned int trên tất cả các máy.

6.2.5 Các kiểu dấu phẩy động

Các loại dấu phẩy động đại diện cho các số dấu phẩy động. Số dấu phẩy động là một phép tính gần đúng của một số thực đư ợc biểu diễn trong một lư ợng bộ nhớ cố định. Có ba loại dấu phẩy động: float (độ chính xác đơ n), double (độ chính xác kép) và long double (độ chính xác mở rộng).

Ý nghĩa chính xác của độ chính xác đơ n, kép và mở rộng đư ợc xác định theo triển khai. Việc chọn độ chính xác phù hợp cho một vấn đề trong đó sự lựa chọn quan trọng đòi hỏi sự hiểu biết đáng kể về tính toán dấu phẩy động. Nếu bạn không có sự hiểu biết đó, hãy nhận lời khuyên, dành thời gian để tìm hiểu hoặc sử dụng gấp đôi và hy vọng điều tốt nhất.

6.2.5.1 Chữ dấu phẩy động

Theo mặc định, một chữ số dấu phẩy động có kiểu gấp đôi. Một lần nữa, trình biên dịch phải cảnh báo về các ký tự dấu phẩy động quá lớn để đư ợc biểu diễn. Dư ới đây là một số nghĩa đen dấu phẩy động:

Machine Translated by Google


```
1,23 .23 0,23 1. 1,0 1,2e10 1,23e 15
```

Lư u ý rằng khoảng trắng không thể xuất hiện ở giữa một ký tự dấu chấm động. Ví dụ: 65,43 e 21 không phải là một ký tự dấu chấm động mà là bốn mã thông báo từ vựng riêng biệt (gây ra lỗi cú pháp):

```
65,43 và 21
```

Nếu bạn muốn một ký tự dấu phẩy động có kiểu float, bạn có thể định nghĩa một kiểu bằng cách sử dụng hậu tố f hoặc F:

```
3.14159265f 2.0f 2.997925F 2.9e 3f
```

Nếu bạn muốn một ký tự dấu phẩy động thuộc loại long double, bạn có thể xác định một ký tự bằng cách sử dụng hậu tố l hoặc L:

```
3.14159265L 2.0L 2.997925L 2.9e 3L
```

6.2.6 Tiền tố và hậu tố

Có một vư ờn thú nhỏ các hậu tố biểu thị các loại chữ và cũng có một vài tiền tố:

Các tiền tố và hậu tố theo nghĩa đen số học fix

ký hiệu		Nghĩa	Ví dụ Tham khảo Tiền tố ISO tiền		
0		tố bát phân	0776	§6.2.4.1	§iso.2.14.2
0x	0X	hậu tố thập lục phân	0xff	§6.2.4.1	§iso.2.14.2
TRONG	TRONG	hậu tố không dấu	10U	§6.2.4.1	§iso.2.14.2
-	L	hậu tố dài hậu	20000L	§6.2.4.1	§iso.2.14.2
sẽ	LL	tố dài hậu tố dài	20000LL	§6.2.4.1	§iso.2.14.2
f	F	trung trời nổi	10f	§6.2.5.1	§iso.2.14.4
ba 10	VÀ	tố tiền tố dấu	10e 4	§6.2.5.1	§iso.2.14.4
.		chấm động dấu	12.3	§6.2.5.1	§iso.2.14.4
'		phẩy động tiền	'c'	§6.2.3.2	§iso .2.14.3
TRONG'		tố char tiền tố	u'c'	§6.2.3.2	§iso.2.14.3
TRONG'		char16_t tiền tố	u'c'	§6.2.3.2	§iso.2.14.3
L'		char32_t tiền tố	cái c'	§6.2.3.2	§iso.2.14.3
"		wchar_t tiền tố	"sự lộn xộn"	§7.3	§iso.2.14.5
R"		chuỗi tiền tố chuỗi	R"(\b)"	.2	§iso.2.14.5
u8"	thô u8R"	tiền tố chuỗi UTF-8	u8"foo"	§7.3.2.1	§iso.2.14.5
TRONG"		tiền tố chuỗi UTF-16	bạn "foo"	§7.3.2.2	§iso.2.14.5
TRONG"	bạn"	chuỗi uR" tiền tố chuỗi UTF-32	Bạn "foo"	§7.3.2.2	§iso.2.14
L"	LR"	tiền tố chuỗi wchar_t	"foo"	§7.3.2.2 §7.3.2.2.5	§iso.2.14.5

Lư u ý rằng ''chuỗi'' ở đây có nghĩa là ''chuỗi ký tự'' (§7.3.2) chứ không phải ''của loại std::string.''

Rõ ràng, chúng ta cũng có thể xem xét . và e là trung tố và R" và u8" là phần đầu tiên của tập hợp các dấu phân cách. Tuy nhiên, tôi coi danh pháp ít quan trọng hơ n là đư a ra một cái nhìn tổng quan về sự đa dạng khó hiểu của nghĩa đen.

Các hậu tố l và L có thể đư ợc kết hợp với các hậu tố u và U để thể hiện các loại dài không dấu . Ví dụ:

Machine Translated by Google


```
1LU                 // không dấu dài
2UL                    không dấu dài
3ULL                // không dấu dài dài không
LẦN THỨ 4              dấu dài dài
THỨ 5               // lỗi
```

Các hậu tố l và L có thể đư ợc sử dụng cho các ký tự dấu phẩy động để diễn tả long double. Ví dụ:

```
1L                  // dài int
1.0L                // dài gấp đôi
```

Cho phép kết hợp các tiền tố R, L và u , ví dụ: uR" (foo\(bar)) ". Lư u ý sự khác biệt đáng kể về ý nghĩa của tiền tố U cho một ký tự (không dấu) và cho một chuỗi mã hóa UTF-32 (§7.3.2.2).

Ngoài ra, ngư ời dùng có thể xác định các hậu tố mới cho các loại do ngư ời dùng xác định. Ví dụ, bằng cách xác định một toán tử chữ do ngư ời dùng định nghĩa (§19.2.6), chúng ta có thể lấy

```
"thanh foo"         // Một chữ của loại chuỗi std::string
123_km              // một của loại Khoảng cách
```

Hậu tố không bắt đầu bằng _ đư ợc dành riêng cho thư viện tiêu chuẩn.

6.2.7 vô hiệu

Loại void về mặt cú pháp là một loại cơ bản. Tuy nhiên, nó chỉ có thể đư ợc sử dụng như một phần của loại phức tạp hơ n; không có đối tư ợng kiểu void. Nó đư ợc sử dụng để xác định rằng một hàm không trả về một giá trị hoặc là kiểu cơ sở cho các con trỏ tới các đối tư ợng có kiểu không xác định. Ví dụ:

```
khoảng trống        // lỗi : không có các đối tư ợng
x; vô hiệu&            lỗi: void không có tham chiếu đến void
r; khoảng           // hàm f không trả về a giá trị (§12.1.4)
trống f(); void  pv; // con trỏ tới đối tư ợng thuộc loại không xác định (§7.2.1)
```

Khi khai báo một hàm, bạn phải chỉ định loại giá trị đư ợc trả về. Về mặt logic, bạn sẽ mong đợi có thể chỉ ra rằng một hàm không trả về giá trị bằng cách bỏ qua kiểu trả về. Dù sao đi nữa, điều đó sẽ làm cho ngữ pháp trở nên lộn xộn (§iso.A). Do đó, void đư ợc sử dụng làm ''kiểu trả về giả'' để chỉ ra rằng một hàm không trả về giá trị.

6.2.8 Kích thư ớc

Một số khía cạnh của các loại cơ bản của C++, chẳng hạn như kích thư ớc của một int, đư ợc thực hiện xác định (§6.1). Tôi chỉ ra những yếu tố phụ thuộc này và thư ờng khuyên bạn nên tránh chúng hoặc thực hiện các bư ớc để giảm thiểu tác động của chúng. Tại sao bạn phải bận tâm? Những ngư ời lập trình trên nhiều hệ thống hoặc sử dụng nhiều trình biên dịch khác nhau rất quan tâm vì nếu không, họ buộc phải lãng phí thời gian để tìm và sửa các lỗi khó hiểu. Những ngư ời tuyên bố rằng họ không quan tâm đến tính di động thư ờng làm như vậy bởi vì họ chỉ sử dụng một hệ thống duy nhất và cảm thấy họ có thể chấp nhận thái độ rằng ''ngôn ngữ là thứ mà trình biên dịch của tôi triển khai''. Đây là một quan điểm hạn hẹp và thiển cận. Nếu chư ơ ng trình của bạn thành công, nó sẽ đư ợc chuyển, vì vậy ai đó sẽ phải tìm và khắc phục sự cố liên quan đến các tính năng phụ thuộc vào việc triển khai. Ngoài ra, các chư ơ ng trình thư ờng cần đư ợc biên dịch với các trình biên dịch khác cho cùng một hệ thống và ngay cả bản phát hành trong tư ơ ng lai của trình biên dịch yêu thích của bạn cũng có thể thực hiện một số điều khác với hiện tại.

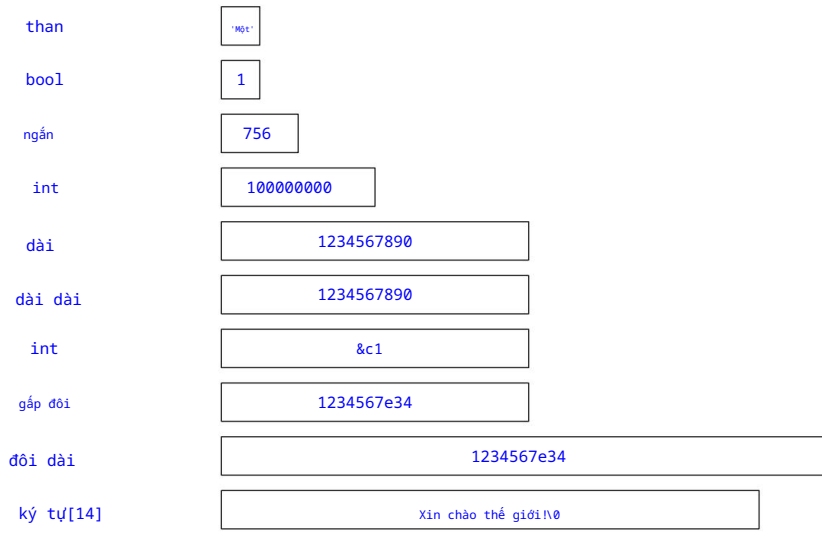
Machine Translated by Google

một. Việc biết và hạn chế tác động của các phụ thuộc triển khai khi một chương trình được viết sẽ dễ dàng hơn nhiều so với việc cố gắng gỡ rối mớ hỗn độn sau đó.

Việc hạn chế tác động của các tính năng ngôn ngữ phụ thuộc vào việc triển khai là tương đối dễ dàng. Hạn chế tác động của các cơ sở thư viện phụ thuộc vào hệ thống khó hơn nhiều. Sử dụng các cơ sở thư viện tiêu chuẩn bất cứ nơi nào khả thi là một cách tiếp cận.

Lý do cung cấp nhiều hơn một loại số nguyên, nhiều hơn một loại không dấu và nhiều hơn một loại dấu phẩy động là để cho phép lập trình viên tận dụng các đặc tính của phần cứng. Trên nhiều máy, có sự khác biệt đáng kể về yêu cầu bộ nhớ, thời gian truy cập bộ nhớ và tốc độ tính toán giữa các loại cơ bản khác nhau. Nếu bạn biết về máy, bạn thường dễ dàng chọn kiểu số nguyên thích hợp cho một biến cụ thể chẳng hạn. Viết mã cấp thấp di động thực sự khó hơn.

Đây là một biểu diễn đồ họa của một tập hợp hợp lý các loại cơ bản và một chuỗi ký tự mẫu (§7.3.2):



Trên cùng một tỷ lệ (0,2 inch trên một byte), một megabyte bộ nhớ sẽ kéo dài khoảng 3 dặm (5 km) về bên phải.

Kích thước của các đối tượng C++ được thể hiện dưới dạng bội số của kích thước của một ký tự, do đó, theo định nghĩa, kích thước của một ký tự là 1. Kích thước của một đối tượng hoặc loại có thể được lấy bằng cách sử dụng toán tử sizeof (§10.3). Đây là những gì được đảm bảo về kích thước của các loại cơ bản:

- 1 ≡ sizeof(char) ≤ sizeof(ngắn) ≤ sizeof(int) ≤ sizeof(dài) ≤ sizeof(dài dài) 1 ≤
- sizeof(bool) ≤ sizeof(dài) •

sizeof(char) ≤ sizeof(wchar_t) ≤ sizeof (dài) •

sizeof(float) ≤ sizeof(double) ≤ sizeof(long double) •

sizeof(N) ≡ sizeof(N có dấu) ≡ sizeof(N không có dấu)

Machine Translated by Google

Trong dòng cuối cùng đó, N có thể là char, short, int, long hoặc long long. Ngoài ra, đảm bảo rằng một ký tự có ít nhất 8 bit, ngắn ít nhất 16 bit và dài ít nhất 32 bit. Một ký tự có thể chứa một ký tự trong bộ ký tự của máy. Loại char đư ợc cho là do quá trình triển khai chọn là loại phù hợp nhất để giữ và thao tác các ký tự trên một máy tính nhất định; nó thư ờng là một byte 8 bit. Tư ơ ng tự, kiểu int đư ợc cho là đư ợc chọn là kiểu phù hợp nhất để giữ và thao tác với các số nguyên trên một máy tính nhất định; nó thư ờng là một từ 4 byte (32 bit). Thật không khôn ngoan khi giả định nhiều hơ n. Ví dụ: có những máy có ký tự 32 bit. Sẽ cực kỳ không khôn ngoan khi cho rằng kích thư ớc của một int giống như kích thư ớc của một con trỏ; nhiều máy (''kiến trúc 64-bit'') có con trỏ lớn hơ n số nguyên. Lư u ý rằng không đảm bảo rằng sizeof(long)<sizeof(long long) hoặc sizeof(double)<siz eof(long double).

Một số khía cạnh do triển khai xác định của các loại cơ bản có thể đư ợc tìm thấy bằng cách sử dụng sizeof đơ n giản và có thể tìm thấy nhiều khía cạnh khác trong <limits>. Ví dụ:

```
#include <giới hạn> //  §40.2
#include <iostream>

int main()
{
      cout << "kích thư ớc của dài " << sizeof(1L) << '\n';
      cout << "kích thư ớc của long long " << sizeof(1LL) << '\n';

                                  "
      cout << "phao to nhat ==      << std::numeric_limits<float>::max() << '\n';
                                    "
      cout << "char duoc ban ==       << std::numeric_limits<char>::is_signed << '\n';
}
```

Các hàm trong <limits> (§40.2) là constexpr (§10.4) để có thể sử dụng chúng mà không cần tốn phí thời gian chạy và trong các ngữ cảnh yêu cầu biểu thức hằng số.

Các loại cơ bản có thể đư ợc trộn lẫn tự do trong các bài tập và biểu thức. Bất cứ khi nào có thể, các giá trị đư ợc chuyển đổi để không làm mất thông tin (§10.5).

Nếu một giá trị v có thể đư ợc biểu diễn chính xác trong một biến loại T, thì việc chuyển đổi v thành T là giá trị bảo quản. Tốt nhất nên tránh chuyển đổi không bảo toàn giá trị (§2.2.2, §10.5.2.6).

Nếu bạn cần một kích thư ớc cụ thể của số nguyên, chẳng hạn như số nguyên 16 bit, bạn có thể #include tiêu đề chuẩn <cstdint> xác định nhiều loại khác nhau (hay đúng hơ n là nhập bí danh; §6.5). Ví dụ:

```
int16_t x {0xaabb};                          // 2 byte
int64_t xxxx {0xaaaabbbbccccdddd}; //           8 byte
int_least16_t y; // int_least32_t yy int_fast32_t ít nhất 2 byte (giống int) ít
z;                                           // nhất 4 byte (giống long) kiểu
                                             // int nhanh nhất ít nhất 4 byte
```

Tiêu đề chuẩn <cstddef> xác định một bí danh đư ợc sử dụng rất rộng rãi trong cả mã khai báo thư viện chuẩn và mã ngư ời dùng: size_t là một loại số nguyên không dấu do triển khai xác định có thể chứa kích thư ớc theo byte của mọi đối tư ợng. Do đó, nó đư ợc sử dụng khi chúng ta cần giữ kích thư ớc đối tư ợng. Ví dụ:

```
void   phân bổ (size_t n); // lấy N byte
```

Tư ơ ng tự, <cstddef> định nghĩa kiểu số nguyên có dấu ptrdiff_t để giữ kết quả trừ hai con trỏ để lấy một số phần tử.

Machine Translated by Google

6.2.9 Căn chỉnh

Một đối tư ợng không chỉ cần đủ bộ nhớ để chứa biểu diễn của nó. Ngoài ra, trên một số kiến trúc máy, các byte đư ợc sử dụng để giữ nó phải đư ợc căn chỉnh phù hợp để phần cứng truy cập hiệu quả (hoặc trong trư ờng hợp cực đoan là không thể truy cập đư ợc). Ví dụ: một int 4 byte thư ờng phải đư ợc căn chỉnh trên một ranh giới Word (4 byte) và đôi khi một ký tự kép 8 byte phải đư ợc căn chỉnh trên một ranh giới Word (8 byte). Tất nhiên, đây là tất cả những cách triển khai rất cụ thể và đối với hầu hết các lập trình viên, nó hoàn toàn ẩn. Bạn có thể viết mã C++ tốt trong nhiều thập kỷ mà không cần phải rõ ràng về việc căn chỉnh. Vị trí căn chỉnh thư ờng hiển thị nhất là trong bố cục đối tư ợng: đôi khi cấu trúc chứa ''lỗ'' để cải thiện căn chỉnh (§8.2.1).

Toán tử alignof () trả về sự căn chỉnh của biểu thức đối số của nó. Ví dụ:

```
auto ac = alignof('c'); tự        // sự liên kết của một char sự
động ai = alignof(1);               liên kết của int sự liên kết
quảng cáo tự động = alignof(2.0);  // một double


int a[20];
tự động aa = alignof(a);          // sự liên kết của int một
```

Đôi khi, chúng ta phải sử dụng căn chỉnh trong một khai báo, trong đó một biểu thức, chẳng hạn như alignof(x+y) không đư ợc phép. Thay vào đó, chúng ta có thể sử dụng bộ xác định loại alignas: alignas(T) có nghĩa là ''căn chỉnh giống như chữ T''. Ví dụ: chúng ta có thể dành bộ nhớ chư a khởi tạo cho một số loại X như sau:

```
void user(const vector<X>& vx) {

        constexpr int bufmax = 1024; bộ đệm
        alignas(X)[bufmax];                    // chư a khởi tạo

        const int max = min(vx.size(),bufmax/siz eof(X));
        uninitialized_copy(vx.begin(),vx.begin()+max,buffer);
        // ...
}
```

6.3 Tuyên bố

Trư ớc khi một tên (số nhận dạng) có thể đư ợc sử dụng trong chư ơ ng trình C++, nó phải đư ợc khai báo. Nghĩa là, loại của nó phải đư ợc chỉ định để thông báo cho trình biên dịch loại thực thể mà tên đó đề cập đến. Ví dụ:

```
ký tự;
dây; tự
động đếm = 1; const
double pi {3.1415926535897}; lỗi int bên ngoài_số;



const char   name = "Njal"; const
char   season[] = { "xuân", "hạ", "thu", "đông" }; vectơ <chuỗi> ngư ời {tên,
"Skarphedin", "Gunnar" };
```

Machine Translated by Google


```
ngày cấu trúc { int d, m, y; };
int day(Date   p) { return p  >d; } đôi
sqrt(gấp đôi); mẫu<lớp T>
T abs(T a) { return a<0?   a : a; }

constexpr int fac(int n) { return (n<2)?1:n  fac(n  1); } constexpr        // đánh giá thời gian biên dịch có thể (§2.2.3) khởi
double zz { ii  fac(7) };                                                  // tạo thời gian biên dịch

sử dụng Cmplx = std::complex<double>; cấu trúc                            // loại bí danh (§3.4.5, §6.5)
Ngư ời dùng;                                                               // loại tên
loại bia { Carlsberg, Tuborg, Thor }; không gian tên NS
{ int a; }
```

Như có thể thấy từ các ví dụ này, một khai báo có thể làm đư ợc nhiều việc hơ n là chỉ kết hợp một loại với một tên. Hầu hết các khai báo này cũng là định nghĩa. Một định nghĩa là một khai báo cung cấp tất cả những gì cần thiết trong một chư ơ ng trình để sử dụng một thực thể. Đặc biệt, nếu cần bộ nhớ để đại diện cho một cái gì đó, thì bộ nhớ đó đư ợc đặt sang một bên theo định nghĩa của nó. Một thuật ngữ khác coi phần khai báo là một phần của giao diện và phần định nghĩa của phần triển khai. Khi thực hiện chế độ xem đó, chúng tôi cố gắng tổng hợp các giao diện từ các khai báo có thể đư ợc sao chép trong các tệp riêng biệt (§15.2.2); các định nghĩa đặt bộ nhớ sang một bên không thuộc về các giao diện.

Giả sử rằng các khai báo này thuộc phạm vi toàn cầu (§6.3.4), chúng ta có:

```
ký tự; tự                                   // dành riêng cho bộ nhớ a   char và khởi tạo nó thành 0
động đếm = 1; const                         // int khởi tạo thành 1; bộ nhớ dành riêng cho bộ nhớ
char   name = "Njal";                       // nhớ khởi tạo con trỏ với địa chỉ của chuỗi con trỏ tới ký tự
                                            // ký tự đó                  chuỗi chữ        "Móng tay"
                                            //

ngày cấu trúc { int d, m, y; };             // Ngày là một cấu trúc có ba thành viên
int day(Date   p) { return p  >d; }         // day là một chức năng thực thi mã đư ợc chỉ định

sử dụng Point = std::complex<shor t>;//            Điểm là một cho tên std::complex<shor t>
```

Trong số các khai báo ở trên, chỉ có ba khai báo không phải là định nghĩa:

```
đôi sqrt(gấp đôi); lỗi int                  khai báo hàm khai báo
bên ngoài_số; cấu trúc Ngư ời               // biến
dùng;                                       // khai báo tên loại
```

Nghĩa là, nếu đư ợc sử dụng, thực thể mà chúng đề cập đến phải đư ợc xác định ở nơ i khác. Ví dụ:

```
double sqrt(double d) { /* int     ··· */ }
error_number = 1; cấu trúc
ngư ời dùng { /* */ };···
```

Phải luôn có chính xác một định nghĩa cho mỗi tên trong chư ơ ng trình C++ (để biết tác dụng của #include, xem §15.2.3). Tuy nhiên, có thể có nhiều khai báo.

Tất cả các khai báo của một thực thể phải thống nhất về kiểu của nó. Vì vậy, đoạn này có hai lỗi:

```
số nguyên;
số nguyên;          // lỗi :   định nghĩa lại
```

Machine Translated by Google


```
lỗi int bên ngoài_số;
error_number ngắn bên ngoài;            // khớp : gõ lỗi không
```

Điều này không có lỗi (đối với việc sử dụng bên ngoài, xem §15.2):

```
lỗi int bên ngoài_số; lỗi
int bên ngoài_số; //          OK: khai báo lại
```

Một số định nghĩa chỉ rõ ''giá trị'' cho các thực thể mà chúng định nghĩa. Ví dụ:

```
ngày cấu trúc { int d, m,
y; }; sử dụng Point = std::complex<shor        // Điểm là một cái tên cho std::complex<shor t>
t>; int day(Date  p) { return
p  >d; } const double pi {3.1415926535897};
```

Đối với các loại, bí danh, mẫu, hàm và hằng số, ''giá trị'' là vĩnh viễn. Đối với các loại dữ liệu không phải là const , giá trị ban đầu có thể được thay đổi sau đó. Ví dụ:

```
khoảng trống
f() {
      số int {1}; // const char  name      khởi tạo đếm thành
      {"Bjarne"}; // tính = 2; // name =   tên   cũng là var 1 iable trỏ đến một  hằng số (§7.5)
      "Marian";                            chỉ định 2 để đếm

}
```

Trong số các định nghĩa, chỉ có hai định nghĩa không chỉ định giá trị:

```
ký tự;
dây;
```

Xem §6.3.5 và §17.3.3 để biết giải thích về cách thức và thời điểm một biến được gán giá trị mặc định.
Bất kỳ tuyên bố nào chỉ định một giá trị là một định nghĩa.

6.3.1 Cấu trúc của Tuyên bố

Cấu trúc của một khai báo được định nghĩa bởi ngữ pháp C++ (§iso.A). Ngữ pháp này đã phát triển hơn bốn thập kỷ, bắt đầu với ngữ pháp C đầu tiên và khá phức tạp. Tuy nhiên, không cần quá nhiều sự đơn giản hóa triệt để, chúng ta có thể coi một tuyên bố có năm phần (theo thứ tự):

- Bộ chỉ định tiền tố tùy chọn (ví dụ: tĩnh hoặc ảo) • Loại cơ sở (ví dụ: vector<double> hoặc const int) • Bộ khai báo tùy chọn bao gồm tên (ví dụ: p[7], n hoặc ( )[]) • Các bộ định danh chức năng có hậu tố tùy chọn (ví dụ: const hoặc noexcept) • Bộ khởi tạo hoặc thân hàm tùy chọn (ví dụ: ={7,5,3} hoặc {return x;})

Ngoại trừ các định nghĩa hàm và không gian tên, một khai báo được kết thúc bằng dấu chấm phẩy. Xem xét định nghĩa của một mảng các chuỗi kiểu C:

```
const char  các vị vua[] = { "Antigonus", "Seleukos", "Ptolemy" };
```

Ở đây, kiểu cơ sở là const char, bộ khai báo là kings[] và bộ khởi tạo là = theo sau là danh sách {}.

Trình xác định là một từ khóa ban đầu, chẳng hạn như ảo (§3.2.3, §20.3.2), extern (§15.2) hoặc constexpr (§2.2.3), chỉ định một số thuộc tính không phải kiểu của nội dung đang được khai báo.

Machine Translated by Google

Một bộ khai báo bao gồm một tên và tùy chọn một số toán tử khai báo. com nhiều nhất toán tử khai báo mon là:

Ngư ời khai báo Toán tử		
tiền tố		con trỏ
tiền tố	const	hằng con trỏ biến
tiền tố	dễ bay hơ i	động con trỏ
tiền tố	&	lvalue tham chiếu (§7.7.1)
tiền tố tiền tố	&&	tham chiếu rvalue (§7.7.2)
tiền tố tiền tố	tự	hàm (dùng kiểu trả về hậu tố) hàm
động hậu	[]	mảng
tố hậu tố	()	trả về
hậu tố	>	từ hàm

Việc sử dụng chúng sẽ đơ n giản nếu chúng đều là tiền tố hoặc hậu tố. Tuy nhiên, , [] và () đư ợc thiết kế để phản ánh việc sử dụng chúng trong các biểu thức (§10.3). Do đó, là tiền tố và [] và () là hậu tố. Các toán tử khai báo hậu tố liên kết chặt chẽ hơ n các toán tử tiền tố. Do đó, char kings[] là một mảng các con trỏ tới char, trong khi char(kings)[] là một con trỏ tới một mảng char. Chúng ta phải sử dụng dấu ngoặc đơ n để diễn đạt các kiểu như ''con trỏ tới mảng'' và ''con trỏ tới hàm''; xem các ví dụ trong §7.2.

Lư u ý rằng loại không thể bị loại khỏi khai báo. Ví dụ:

```
hằng c = 7;        // lỗi: không    kiểu

gt(int a, int b) // {  lỗi: không    kiểu trả về

      trả về (a>b)? một : b;
}

ui không dấu; li        OK: ''unsigned''có nghĩa là ''unsigned int''
dài;                // OK: ''dài'' ''đó nghĩa'' đôi int''
```

Về điều này, C++ tiêu chuẩn khác với các phiên bản đầu tiên của C và C++ cho phép hai ví dụ đầu tiên bằng cách coi int là loại khi không có loại nào đư ợc chỉ định (§44.3). Quy tắc '' int ngầm '' này là nguồn gốc của các lỗi tinh vi và nhiều nhầm lẫn.

Một số loại có tên bao gồm nhiều từ khóa, chẳng hạn như long long và int dễ bay hơ i. Một số tên kiểu thậm chí trông không giống tên lắm, chẳng hạn như decltype(f(x)) (kiểu trả về của lệnh gọi f(x); §6.3.6.3).

Trình xác định dễ bay hơ i đư ợc mô tả trong §41.4.

Trình xác định alignas() đư ợc mô tả trong §6.2.9.

6.3.2 Khai báo nhiều tên

Có thể khai báo nhiều tên trong một lần khai báo. Tuyên bố chỉ chứa một danh sách các khai báo đư ợc phân tách bằng dấu phẩy. Ví dụ, chúng ta có thể khai báo hai số nguyên như sau:

```
int x, y;          // intx; int y;
```

Machine Translated by Google

Các toán tử chỉ áp dụng cho các tên riêng lẻ – và không áp dụng cho bất kỳ tên nào tiếp theo trong cùng một khai báo. Ví dụ:

```
int   p, y;          // int* int y; p;      KHÔNG int* y;
int x,   q; // int      intx; int* q;
v[10],   pv; //         int v[10]; int * pv;
```

Các khai báo như vậy với nhiều tên và các bộ khai báo không cần thiết làm cho chương trình khó đọc hơn và nên tránh.

6.3.3 Tên

Một tên (mã định danh) bao gồm một chuỗi các chữ cái và chữ số. Ký tự đầu tiên phải là một chữ cái. Ký tự gạch dưới, _, được coi là một chữ cái. C++ không áp đặt giới hạn về số ký tự trong tên. Tuy nhiên, một số phần của việc triển khai không nằm dưới sự kiểm soát của người viết trình biên dịch (cụ thể là trình liên kết) và thật không may, những phần đó đôi khi áp đặt các giới hạn. Một số môi trường thời gian chạy cũng cần phải mở rộng hoặc hạn chế bộ ký tự được chấp nhận trong mã định danh. Các tiện ích mở rộng (ví dụ: cho phép ký tự $ trong tên) mang lại các chương trình không thể chuyển đổi được. Không thể sử dụng từ khóa C++ (§6.3.3.1), chẳng hạn như new hoặc int làm tên của thực thể do người dùng xác định. Ví dụ về tên là:

```
Xin chào            this_is_a_most_unnormal_long_identifier_that_is_better_avoided foO
ĐỊNH NGHĨA                      quán ba          u_name          Cảm giác ngựa
ở đâu0              biến1       LỚP HỌC          _lớp học        ___
```

Ví dụ về các chuỗi ký tự không thể được sử dụng làm số nhận dạng là:

```
012                 một quán            $sys            lớp nếu         3var
thanh toán.đến hạn  bar ngu ngốc        .tên
```

Các tên không cục bộ bắt đầu bằng dấu gạch dưới được dành riêng cho các phương tiện đặc biệt trong quá trình triển khai và môi trường thời gian chạy, vì vậy các tên đó không nên được sử dụng trong các chương trình ứng dụng. Tương tự, các tên bắt đầu bằng dấu gạch dưới kép (__) hoặc dấu gạch dưới theo sau là chữ cái viết hoa (ví dụ: _Foo) được bảo lưu (§iso.17.6.4.3).

Khi đọc một chương trình, trình biên dịch luôn tìm kiếm chuỗi ký tự dài nhất có thể tạo thành tên. Do đó, var10 là một tên duy nhất, không phải tên var theo sau số 10. Ngoài ra, elseif là một tên duy nhất, không phải từ khóa else theo sau từ khóa if.

Chữ hoa và chữ thường khác nhau, vì vậy Đếm và đếm là các tên khác nhau, nhưng thường không khôn ngoan khi chọn các tên chỉ khác nhau bằng cách viết hoa. Nói chung, tốt nhất là tránh những cái tên chỉ khác nhau theo những cách tinh tế. Ví dụ: trong một số phông chữ, chữ hoa ''o'' (O) và số không (0) có thể khó phân biệt, cũng như chữ thường ''L'' (l), chữ hoa ''i'' (I), và một (1). Do đó, l0, lO, l1, ll và I1l là những lựa chọn tồi cho các tên định danh. Không phải tất cả các phông chữ đều có vấn đề giống nhau, nhưng hầu hết đều có một số.

Tên từ một phạm vi lớn phải có tên tương đối dài và hợp lý rõ ràng, chẳng hạn như vector, Window_with_border và Department_number. Tuy nhiên, mã sẽ rõ ràng hơn nếu các tên chỉ được sử dụng trong một phạm vi nhỏ có các tên ngắn gọn, thông thường như x, i và p. Hàm (Chương 12), lớp (Chương 16) và không gian tên (§14.3.1) có thể được sử dụng để giữ cho phạm vi nhỏ. Sẽ rất hữu ích nếu giữ các tên được sử dụng thường xuyên tương đối ngắn và dành các tên thực sự dài cho các thực thể ít được sử dụng.

Machine Translated by Google

Chọn tên để phản ánh ý nghĩa của một thực thể hơ n là việc thực hiện nó. Ví dụ: phone_book tốt hơ n number_vector ev en nếu các số điện thoại tình cờ đư ợc lư u trữ trong một vectơ (§4.4). Không mã hóa thông tin loại trong một tên (ví dụ: pcname cho tên là char hoặc icount cho số đếm là int) như đôi khi đư ợc thực hiện trong các ngôn ngữ có hệ thống loại động hoặc yếu:

- Kiểu mã hóa trong tên làm giảm mức độ trừu tư ợng của chư ơ ng trình; đặc biệt, nó không cho phép lập trình chung chung (dựa trên một cái tên có thể tham chiếu đến các thực thể thuộc các loại thực thể khác nhau).
- Trình biên dịch theo dõi các loại tốt hơ n bạn. • Nếu bạn muốn thay đổi loại tên (ví dụ: sử dụng std::string để giữ tên), bạn sẽ phải thay đổi mỗi lần sử dụng tên (hoặc mã hóa loại trở thành một lời nói dối).
- Bất kỳ hệ thống viết tắt loại nào bạn có thể nghĩ ra sẽ trở nên quá phức tạp và khó hiểu khi số lư ợng các loại bạn sử dụng tăng lên.

Chọn tên hay là một nghệ thuật.

Cố gắng duy trì một phong cách đặt tên nhất quán. Ví dụ: viết hoa tên của các loại do ngư ời dùng xác định và bắt đầu tên của các thực thể không phải loại bằng một chữ cái viết thư ờng (ví dụ: Shape và current_token). Ngoài ra, hãy sử dụng tất cả các chữ hoa cho macro (nếu bạn phải sử dụng macro (§12.6); ví dụ: HACK) và không bao giờ viết hoa cho các macro không phải (ngay cả đối với các hằng số không phải macro). Sử dụng dấu gạch dư ới để phân tách các từ trong một mã định danh; number_of_elements dễ đọc hơ n numberOfElements. Tuy nhiên, khó đạt đư ợc tính nhất quán vì các chư ơ ng trình thư ờng bao gồm các đoạn từ các nguồn khác nhau và một số kiểu hợp lý khác nhau đang đư ợc sử dụng. Hãy nhất quán trong việc sử dụng các chữ viết tắt và từ viết tắt của bạn.

Lư u ý rằng ngôn ngữ và thư viện chuẩn sử dụng chữ thư ờng cho các loại; điều này có thể đư ợc coi là một gợi ý rằng chúng là một phần của tiêu chuẩn.

6.3.3.1 Từ khóa

Các từ khóa C++ là:

Từ khóa C++					
thẳng hàng	thẳng hàng	Và	and_eq	asm	tự đồ ng
bitand	đang tới	bool	phá vỡ	trư ờng hợp	nắm lấy
than	char16_t char32_t const_cast		lớp học	tuân thủ	hằng số
constexpr	tiếp tục		khai báo	mặc định	xóa bỏ
LÀM	gấp đôi	dynamic_cast	khác	liệt kê	goto rõ
bên ngoài	SAI	trôi nổi	vì	ngư ời bạn	ràng
~~	nội tuyến	int	long	có thể thay đổi	không gian tên
mới	không ngoại lệ	không	not_eq	nullptr	đăng ký
hoặc	or_eq	riêng tư	đư ợc bảo vệ	công khai	điều hành
reinterpret_cast trả về		ngắn	đã ký	kích thư ớc của	tĩnh
static_assert static_cast tên kiểu		cấu trúc	công tắc	mẫu typedef	cái này
thread_local	ném	ĐÚNG VẬY	thử		đánh máy
	liên hiệp	chư a ký	sử dụng	cho tual	khoảng trống
bay hơ i	wchar_t	trong khi	miễn phí	xor_eq	

Ngoài ra, từ export t đư ợc dành riêng để sử dụng trong tư ơ ng lai.

Machine Translated by Google

6.3.4 Phạm vi

Một khai báo giới thiệu một tên vào một phạm vi; nghĩa là, một tên chỉ có thể đư ợc sử dụng trong một phần cụ thể của văn bản

chư ơ ng trình. • Phạm vi cục bộ: Tên đư ợc khai báo trong hàm (Chư ơ ng 12) hoặc lambda (§11.4) đư ợc gọi là tên cục bộ. Phạm vi của nó mở rộng từ điểm khai báo của nó đến cuối khối mà tại đó việc khai báo của nó xảy ra. Một khối là một phần mã đư ợc giới hạn bởi một cặp {} . Tên tham số hàm và lambda đư ợc coi là tên cục bộ trong khối ngoài cùng của hàm hoặc lambda của chúng.

• Phạm vi lớp: Một tên đư ợc gọi là tên thành viên (hoặc tên thành viên lớp) nếu nó đư ợc định nghĩa trong một lớp bên ngoài bất kỳ hàm, lớp (Chư ơ ng 16), lớp enum (§8.4.1) hoặc không gian tên nào khác. Phạm vi của nó kéo dài từ phần mở đầu { của khai báo lớp đến phần cuối của khai báo lớp. • Phạm vi không gian tên: Một tên đư ợc gọi là tên thành viên không gian tên nếu nó đư ợc định nghĩa trong một không gian tên (§14.3.1) bên ngoài bất kỳ hàm nào, lambda (§11.4), lớp (Chư ơ ng 16), lớp enum (§8.4.1) , hoặc không gian tên khác. Phạm vi của nó kéo dài từ điểm khai báo đến hết không gian tên của nó. Tên không gian tên cũng có thể truy cập đư ợc từ các đơ n vị dịch thuật khác (§15.2).

• Phạm vi toàn cầu: Một tên đư ợc gọi là tên toàn cầu nếu nó đư ợc định nghĩa bên ngoài bất kỳ hàm, lớp (Chư ơ ng 16), lớp enum (§8.4.1) hoặc không gian tên (§14.3.1). Phạm vi của một tên toàn cầu mở rộng từ điểm khai báo đến cuối tệp trong đó khai báo của nó xảy ra. Tên chung cũng có thể truy cập đư ợc từ các đơ n vị dịch thuật khác (§15.2). Về mặt kỹ thuật, không gian tên chung đư ợc coi là một không gian tên, vì vậy tên toàn cầu là một ví dụ về tên thành viên không gian tên. • Phạm vi câu lệnh: Một tên nằm trong phạm

vi câu lệnh nếu nó đư ợc xác định trong phần () của câu lệnh for-, while-, if- hoặc switch. Phạm vi của nó kéo dài từ điểm khai báo đến cuối câu lệnh của nó. Tất cả các tên trong phạm vi tuyên bố là tên địa phư ơ ng. • Phạm vi chức năng: Một nhãn (§9.6) nằm trong phạm

vi từ thời điểm khai báo cho đến khi kết thúc chức năng.

Việc khai báo một tên trong một khối có thể che giấu một khai báo trong một khối kèm theo hoặc một tên chung. Nghĩa là, một tên có thể đư ợc xác định lại để chỉ một thực thể khác trong một khối. Sau khi thoát khỏi khối, tên tiếp tục ý nghĩa trư ớc đó của nó. Ví dụ:

```
intx;                   // toàn cầu x

khoảng trống
f() {
        intx;           // địa phư ơ ng x ẩn gán toàn x
        x = 1;          // cục cho x cục bộ
        {
                intx;     ẩn x cục bộ đầu tiên
                x = 2;  // gán cho x cục bộ thứ hai

        } x = 3;        // gán cho địa phư ơ ng đầu tiên x
}

int  p = &x;            // lấy địa chỉ của toàn cầu x
```

Machine Translated by Google

Ẩn tên là không thể tránh khỏi khi viết các chư ơ ng trình lớn. Tuy nhiên, ngư ời đọc có thể dễ dàng không nhận thấy rằng một tên đã bị ẩn (còn đư ợc gọi là ẩn). Vì những lỗi như vậy tư ơ ng đối hiếm nên chúng có thể rất khó tìm. Do đó, việc ẩn tên nên đư ợc giảm thiểu.
Sử dụng các tên như i và x cho các biến toàn cục hoặc cho các biến cục bộ trong một hàm lớn sẽ gặp rắc rối.

Một tên toàn cục ẩn có thể đư ợc tham chiếu bằng cách sử dụng toán tử phân giải phạm vi, ::. Ví dụ:

```
intx;

khoảng trống f2()
{
        int x = 1; // ::x ẩn toàn cầu x
        = 2; // x =     gán cho x toàn cầu
        2; // //        gán cho x cục bộ
           ...
}
```

Không có cách nào để sử dụng một tên địa phư ơ ng ẩn.

Phạm vi của một tên không phải là thành viên lớp bắt đầu tại điểm khai báo của nó, nghĩa là sau bộ khai báo hoàn chỉnh và trư ớc bộ khởi tạo. Điều này ngụ ý rằng một tên thậm chí có thể đư ợc sử dụng để chỉ định giá trị ban đầu của chính nó. Ví dụ:

```
int x = 97;

khoảng trống f3()
{
        int x = x;          // chính xác câu thơ : khởi tạo với mỗi x của giá trị (chư a khởi tạo)
}
```

Một trình biên dịch tốt sẽ cảnh báo nếu một biến đư ợc sử dụng trư ớc khi nó đư ợc khởi tạo.

Có thể sử dụng một tên duy nhất để chỉ hai đối tư ợng khác nhau trong một khối mà không cần sử dụng :: nhà điều hành. Ví dụ:

```
int x = 11;

khoảng trống f4()                // mỗi câu thơ : sử dụng của các đối tư ợng khác nhau đều đư ợc gọi là hai x trong một phạm vi duy nhất
{
        int y = x;          // sử dụng toàn cầu x: Và = 11
        int x = 22;
        Và = x;             // sử dụng địa phư ơ ng x: Và = 22
}
```

Một lần nữa, tốt nhất nên tránh những sự tế nhị như vậy.

Tên của các đối số hàm đư ợc coi là đã khai báo trong khối ngoài cùng của hàm.
Ví dụ:

```
khoảng trống f5(int
x) {
        intx;          // lỗi
}
```

Machine Translated by Google

Đây là một lỗi vì x được xác định hai lần trong cùng một phạm vi.

Các tên được giới thiệu trong câu lệnh for là cục bộ của câu lệnh đó (trong phạm vi câu lệnh). Cái này cho phép chúng ta sử dụng tên quy ước cho các biến vòng lặp nhiều lần trong một hàm. Ví dụ:

```
void f(vector<string>& v, list<int>& lst) {

        for (const auto& x : v) cout << x << '\n'; for
        (auto x : lst) cout << x << '\n'; for
        (int i = 0, i!=v.siz e(), ++i) cout << v[i] << '\n'; for
        (auto i : {1, 2, 3, 4, 5, 6, 7}) cout << i << '\n';
}
```

Điều này không chứa xung đột tên.

Không được phép khai báo dưới dạng câu lệnh duy nhất trên nhánh của câu lệnh if (§9.4.1).

6.3.5 Khởi tạo

Nếu một trình khởi tạo được chỉ định cho một đối tượng, thì trình khởi tạo đó sẽ xác định giá trị ban đầu của một đối tượng. Trình khởi tạo có thể sử dụng một trong bốn kiểu cú pháp:

```
X a1{v};
X a2 = {v};
X a3 = v;
Xa4(v);
```

Trong số này, chỉ cái đầu tiên có thể được sử dụng trong mọi ngữ cảnh và tôi thực sự khuyên bạn nên sử dụng nó. Nó rõ ràng hơn và ít bị lỗi hơn so với các lựa chọn thay thế. Tuy nhiên, biểu mẫu đầu tiên (được sử dụng cho a1) là mới trong C++11, vì vậy ba biểu mẫu còn lại là những gì bạn tìm thấy trong mã cũ hơn. Hai hình thức sử dụng = là những gì bạn sử dụng trong C. Thói quen cũ khó có thể thay đổi, vì vậy đôi khi tôi (không nhất quán) sử dụng = khi khởi tạo một biến đơn giản với một giá trị đơn giản. Ví dụ:

```
int x1 = 0;
ký tự c1 = 'z';
```

Tuy nhiên, bất cứ điều gì phức tạp hơn điều đó được thực hiện tốt hơn bằng cách sử dụng {}. Khởi tạo sử dụng {}, khởi tạo danh sách, không cho phép thu hẹp (§iso.8.5.4). Đó là:

- Một số nguyên không thể chuyển đổi thành một số nguyên khác không thể giữ nguyên giá trị của nó. Ví dụ: cho phép char chuyển sang int , nhưng int sang char thì không.
- Một giá trị dấu phẩy động không thể được chuyển đổi thành một kiểu dấu phẩy động khác không thể giữ giá trị của nó. Ví dụ, cho phép float để tăng gấp đôi , nhưng không tăng gấp đôi để chuyển động.
- Một giá trị dấu phẩy động không thể chuyển đổi thành kiểu số nguyên.
- Một giá trị số nguyên không thể được chuyển đổi thành kiểu dấu phẩy động.

Ví dụ:

```
void f(val đôi, int val2) {

        int x2 = val;           // nếu val==7.9, x2 trở thành 7
        ký tự c2 = val2;        // nếu val2==1025, c2 trở thành 1
```

Machine Translated by Google


```
    int x3 {val};          // lỗi : có thể cắt ngắn lỗi thu
    char c3 {val2};        //       hẹp có thể:

    ký tự c4 {24}; ký      // Đư ợc rồi: 24 lon đư ợc biểu thị chính xác lỗi như một than
    tự c5 {264};           // (giả sử ký tự 8 bit): 264 không thể đư ợc biểu diễn char dư ới dạng

    int x4 {2.0};          // lỗi: không chuyển đổi giá trị gấp đôi thành int

    // ...
}
```

Xem §10.5 để biết các quy tắc chuyển đổi cho các loại tích hợp sẵn.

Không có lợi thế nào khi sử dụng khởi tạo {} và một cái bẫy khi sử dụng tự động để lấy loại đư ợc xác định bởi trình khởi tạo. Cái bẫy là nếu trình khởi tạo là một {}-list, chúng tôi có thể không muốn loại của nó đư ợc suy ra (§6.3.6.2). Ví dụ:

```
xe z1{99}; // auto z2 = z1 là một initializer_list<int> z2
99; //                 không phải là một
```

Vì vậy, ư u tiên = khi sử dụng tự động.

Có thể định nghĩa một lớp sao cho một đối tư ợng có thể đư ợc khởi tạo bởi một danh sách các giá trị và đư ợc xây dựng một cách luân phiên với một vài đối số không chỉ đơ n giản là các giá trị đư ợc lư u trữ. Ví dụ cổ điển là một vectơ số nguyên:

```
vectơ <int> v1 {99};          v1 là một vectơ gồm 1 phần tử có giá trị 99 v2 là một
vectơ <int> v2(99);           // trong 99 phần tử, mỗi phần tử có giá trị mặc định là 0 vectơ
```

Tôi sử dụng lời gọi rõ ràng của hàm tạo, (99), để hiểu nghĩa thứ hai. Hầu hết các loại không cung cấp các lựa chọn thay thế khó hiểu như vậy – thậm chí hầu hết các vectơ cũng không; Ví dụ:

```
vector<string> v1{"xin chào!"};          v1 là một vectơ gồm 1 phần tử có giá trị "xin chào!" hàm tạo
vector<string> v2("Xin chao!");          // lỗi: không vector mất một chuỗi chữ
```

Vì vậy, hãy ư u tiên khởi tạo {} hơ n các lựa chọn thay thế trừ khi bạn có lý do chính đáng để không làm như vậy.

Danh sách trình khởi tạo trống, {}, đư ợc sử dụng để chỉ ra rằng giá trị mặc định là mong muốn. Ví dụ:

```
int x4 {};                // x4 trở thành 0
gấp đôi d4 {};            d4 trở thành 0,0
char  p {};               // trở thành nullptr p
vectơ <int> v4{}; chuỗi   v4 trở thành vectơ trống
s4 {};                    // s4 trở thành ""
```

Hầu hết các loại có một giá trị mặc định. Đối với các loại tích phân, giá trị mặc định là một đại diện phù hợp của số không. Đối với con trỏ, giá trị mặc định là nullptr (§7.2.2). Đối với các loại do ngư ời dùng xác định, giá trị mặc định (nếu có) đư ợc xác định bởi hàm tạo của loại (§17.3.3).

Đối với các loại do ngư ời dùng định nghĩa, có thể có sự khác biệt giữa khởi tạo trực tiếp (trong đó chuyển đổi đư ợc cho phép) và sao chép khởi tạo (nếu không); xem §16.2.6.

Việc khởi tạo các loại đối tư ợng cụ thể đư ợc thảo luận khi thích hợp: • Con trỏ:

§7.2.2, §7.3.2, §7.4 • Tham chiếu:

§7.7.1 (giá trị), §7.7.2 (giá trị)

Machine Translated by Google

• Mảng: §7.3.1, §7.3.2 •
Hằng số: §10.4 • Lớp:
§17.3.1 (không sử dụng hàm tạo), §17.3.2 (sử dụng hàm tạo), §17.3.3 (mặc định),
§17.4 (thành viên và cơ sở), §17.5 (sao chép và di chuyển)
• Bộ chứa do người dùng xác định: §17.3.4

6.3.5.1 Thiếu bộ khởi tạo

Đối với nhiều loại, bao gồm tất cả các loại tích hợp sẵn, có thể loại bỏ trình khởi tạo. Nếu bạn làm điều đó - và điều đó không may là phổ biến - thì tình hình sẽ phức tạp hơn. Nếu bạn không thích sự phức tạp, chỉ cần khởi tạo một cách nhất quán. Trường hợp thực sự tốt duy nhất cho một biến chưa được khởi tạo là một bộ đệm đầu vào lớn. Ví dụ:

```
constexpr int max = 1024  1024; char buf[tối
đa];
some_stream.get(buf,max); //                    đọc tối đa ký tự tối đa vào buf
```

Chúng ta có thể dễ dàng khởi tạo buf:

```
char buf[max] {};                          // khởi tạo char thành 0 mỗi
```

Bằng cách khởi tạo dư thừa, chúng tôi sẽ phải chịu một cú hích về hiệu suất mà có thể rất đáng kể. Tránh sử dụng bộ đệm ở mức độ thấp như vậy nếu bạn có thể và không để những bộ đệm đó không được khởi tạo trừ khi bạn biết (ví dụ: từ phép đo) rằng việc tối ưu hóa so với việc sử dụng mảng được khởi tạo là rất quan trọng.

Nếu không có trình khởi tạo nào được chỉ định, một không gian tên toàn cục (§6.3.4), không gian tên (§14.3.1), tĩnh cục bộ (§12.1.8) hoặc thành viên tĩnh (§16.2.12) (được gọi chung là các đối tượng tĩnh) được khởi tạo thành {} của loại thích hợp. Ví dụ:

```
int a;                    có nghĩa là "int a{0};" trở thành 0 có nghĩa là a
gấp đôi d;            // //                          vì thế mà d trở thành 0,0
```

Các biến cục bộ và đối tượng được tạo trên kho lưu trữ miễn phí (đôi khi được gọi là đối tượng động hoặc đối tượng heap; §11.2) không được khởi tạo theo mặc định trừ khi chúng thuộc loại do người dùng xác định với hàm tạo mặc định (§17.3.3). Ví dụ:

```
khoảng trống
f() {
        intx;                                   x  không có giá trị được xác định rõ, buf[i] không
        char buf[1024];                       // có giá trị                giá trị được xác định rõ

        int   p {new int};                      *p không có giá trị được xác định rõ aq[i] không
        char   q {char mới[1024]};            // có a                      giá trị được xác định rõ

        dây; véc                                s=="" vì hàm tạo mặc định của chuỗi v=={} vì hàm tạo mặc định
        tơ <ký tự> v;                         // của vectơ

                                                        ""
        string   ps {chuỗi mới}; //           // *ps là     vì hàm tạo mặc định của chuỗi
           ...
}
```

Machine Translated by Google

Nếu bạn muốn khởi tạo các biến cục bộ của kiểu dựng sẵn hoặc các đối tư ợng của kiểu dựng sẵn đư ợc tạo bằng cái mới, hãy sử dụng {}. Ví dụ:

```
khoảng trống ff()
{
        int x {};                                   // X trở thành 0
        char buf[1024]{};                           // buf[i] trở thành 0 với tất cả i

        int   p {int mới{10}}; char                 // *p trở thành 10 q[i]
        q {char mới[1024]{}}; //                      trở thành 0 với mọi i

        // ...
}
```

Một thành viên của một mảng hoặc một lớp đư ợc khởi tạo mặc định nếu mảng hoặc cấu trúc đó.

6.3.5.2 Danh sách khởi tạo

Cho đến nay, chúng ta đã xem xét các trư ờng hợp không có bộ khởi tạo và một giá trị của bộ khởi tạo. Các đối tư ợng phức tạp hơ n có thể yêu cầu nhiều hơ n một giá trị làm bộ khởi tạo. Điều này chủ yếu đư ợc xử lý bởi danh sách trình khởi tạo đư ợc phân tách bằng { và }. Ví dụ:

```
int a[] = { 1, 2 };                                 //         mảng khởi tạo
cấu trúc S { int x, chuỗi s }; S s =
{ 1, "Helios" }; phức tạp                           // trình khởi tạo cấu trúc
<đôi> z = {0, pi}; véc tơ  <kép> v = {0.0,          // sử dụng ngư ời xây dựng
1.1, 2.2, 3.3}; //                                     sách  sử dụng hàm tạo danh
```

Để khởi tạo mảng kiểu C, xem §7.3.1. Đối với cấu trúc kiểu C, xem §8.2. Đối với các loại do ngư ời dùng xác định có hàm tạo, hãy xem §2.3.2 hoặc §16.2.5. Để biết các hàm tạo danh sách khởi tạo, hãy xem §17.3.4.

Trong các trư ờng hợp trên, dấu = là dư thừa. Tuy nhiên, một số thích thêm nó để nhấn mạnh rằng một tập hợp các giá trị đư ợc sử dụng để khởi tạo một tập hợp các biến thành viên.

Trong một số trư ờng hợp, danh sách đối số kiểu hàm cũng có thể đư ợc sử dụng (§2.3, §16.2.5). Ví dụ:

```
phức<đôi> z(0,pi); vectơ <đôi> v(10,3.3);      tạo  sử dụng hàm
                                            // tạo :sử dụng hàm          10 phần tử đư ợc khởi tạo 3.3 đư ợc tới
```

Trong một khai báo, một cặp dấu ngoặc đơ n trống, (), luôn có nghĩa là ''hàm'' (§12.1). Vì vậy, nếu bạn muốn rõ ràng về ''sử dụng khởi tạo mặc định'', bạn cần có {}. Ví dụ:

```
phức<đôi> z1(1,2); phức hợp<đôi> f1();          khởi tạo kiểu hàm (khởi tạo bởi constr uctor) khai báo hàm
                                            // //

phức<đôi> z2 {1,2}; phức tạp <kép> f2 {};       khởi tạo bởi constr uctor thành{1,2} khởi tạo bởi
                                            // /constr uctor thành giá trị mặc định {0,0}
```

Lư u ý rằng việc khởi tạo bằng ký hiệu {} không thu hẹp (§6.3.5).

Khi sử dụng tự động, {} -list có loại đư ợc suy ra thành std::initializer_list<T>. Ví dụ:

```
autox1{1,2,3,4}; tự động                // x1 là một   initializer_list<int>
x2 {1.0, 2.25, 3.5}; // autox3{1.0,2}; // x2 là một   initializer_list of<double> of
                                           suy ra lỗi loại:                  {1.0,2} (§6.3.6.2) không thể
```

Machine Translated by Google

6.3.6 Suy diễn Loại: auto và decltype()

Ngôn ngữ này cung cấp hai cơ chế để suy ra một loại từ một biểu thức:

- auto để suy ra một loại đối tượng từ trình khởi tạo của nó; loại có thể là loại của biến có thể, const hoặc constexpr.
- decltype(expr) để suy ra loại thứ gì đó không phải là trình khởi tạo đơn giản, chẳng hạn như kiểu trả về cho một hàm hoặc kiểu của một thành viên lớp.

Việc khấu trừ được thực hiện ở đây rất đơn giản: auto và decltype() chỉ báo cáo kiểu của một biểu thức đã được trình biên dịch biết.

6.3.6.1 Trình xác định loại tự động

Khi khai báo một biến có một bộ khởi tạo, chúng ta không cần chỉ định rõ ràng một loại.
Thay vào đó, chúng ta có thể để biến có kiểu khởi tạo của nó. Coi như :

```
int a1 = 123;
ký tự a2 = 123;
tự động a3 = 123; //  loại a3 là ''int''
```

Kiểu của số nguyên 123 là int, vì vậy a3 là một int. Nghĩa là, auto là trình giữ chỗ cho loại trình khởi tạo.

Không có nhiều lợi thế khi sử dụng auto thay vì int cho một biểu thức đơn giản như 123. Loại càng khó viết và loại càng khó biết thì auto càng trở nên hữu ích. Ví dụ:

```
mẫu<lớp T> void f1(vector<T>& arg) {

        for (vector<T>::iterator p = arg.begin(); p!=arg.end(); ++p)
                p = 7;

        for (auto p = arg.begin(); p!=arg.end(); ++p)   p = 7;

}
```

Vòng lặp sử dụng auto sẽ thuận tiện hơn để viết và dễ đọc hơn. Ngoài ra, nó linh hoạt hơn với các thay đổi mã. Ví dụ: nếu tôi thay đổi arg thành một danh sách, thì vòng lặp sử dụng auto vẫn hoạt động bình thường trong khi vòng lặp đầu tiên sẽ cần phải được viết lại. Vì vậy, trừ khi có lý do chính đáng để không sử dụng auto trong phạm vi nhỏ.

Nếu một phạm vi lớn, việc đề cập đến một loại rõ ràng có thể giúp bản địa hóa các lỗi. Tức là so với sử dụng một loại cụ thể, việc sử dụng tự động có thể trì hoãn việc phát hiện lỗi loại. Ví dụ:

```
khoảng trống f(gấp đôi d)
{
        constexpr auto max = d+7; int a[tối
        đa]; // //                    lỗi : không mảng một tràng buộc số nguyên
          ...
}
```

Nếu tính năng tự động gây bất ngờ, cách khắc phục tốt nhất thường là làm cho các chức năng nhỏ hơn, dù sao thì đây thường là một ý tưởng hay (§12.1).

Machine Translated by Google

Chúng ta có thể trang trí một loại suy luận với các từ xác định và từ bổ nghĩa (§6.3.1), chẳng hạn như const và & (tham khảo; §7.7). Ví dụ:

```
void f(vector<int>& v) {

        for (const auto& x : v) { // //       x là một const int&
                ...
        }
}
```

Ở đây, auto đư ợc xác định bởi kiểu phần tử của v, tức là int.

Lư u ý rằng loại biểu thức không bao giờ là tham chiếu vì tham chiếu hoàn toàn bị hủy bỏ phát sinh trong các biểu thức (§7.7). Ví dụ:

```
khoảng trống g(int& v)
{
        tự động x = v;        // x là một int (không MỘT int&)
        xe & và = TRONG; // là bạn phải int&
}
```

6.3.6.2 danh sách tự động và {}

Khi chúng ta đề cập rõ ràng đến kiểu của một đối tư ợng mà chúng ta đang khởi tạo, chúng ta có hai kiểu cần xem xét: kiểu của đối tư ợng và kiểu của trình khởi tạo. Ví dụ:

```
ký tự v1 = 12345; // int v2 =  12345 là một    int
'c'; //                        'c' là một ký tự
T v3 = f();
```

Bằng cách sử dụng cú pháp {}-initializer cho các định nghĩa như vậy, chúng tôi giảm thiểu khả năng xảy ra các phiên bản lừa đảo đáng tiếc:

```
ký tự v1 {12345}; int          lỗi :  thu hẹp
v2 {'c'};                   // tốt: ẩn char->int chuyển đổi
T v3 {f()};                 // hoạt động khi và chỉ khi loại f() đư ợc chuyển đổi hoàn toàn thành T
```

Khi chúng ta sử dụng auto, chỉ có một loại liên quan, loại bộ khởi tạo và chúng ta có thể sử dụng cú pháp = một cách an toàn :

```
tự động v1 = 12345; // // auto v1 là một  int
v2 = 'c'; tự động               v2 là một ký tự
v3 = f();                   // v3 là một số loại thích hợp
```

Trên thực tế, việc sử dụng cú pháp = với auto có thể là một lợi thế , bởi vì cú pháp {}-list có thể khiến ai đó bất ngờ:

```
tự động v1 {12345};          // v1 là danh sách int
tự động v2 {'c'};               v2 là danh sách char
tự động v3 {f()};            // v3 là danh sách của một số loại thích hợp
```

Điều này là hợp lý. Coi như :

Machine Translated by Google


```
tự động x0                    // lỗi : không thể suy ra một nhập
{}; tự động x1                danh sách int có phần tử một danh
{1}; tự động x2               // int có hai phần tử
{1,2}; tự động x3 {1,2,3};    // danh sách int có ba phần tử
```

Loại danh sách đồng nhất của các phần tử loại T được coi là loại initializer_list<T> (§3.2.1.3, §11.3.3). Đặc biệt, loại x1 không được suy ra là int. Nếu có, các loại x2 và x3 sẽ như thế nào?

Do đó, tôi khuyên bạn nên sử dụng = thay vì {} cho các đối tượng được chỉ định tự động bất cứ khi nào chúng tôi không có ý định ''danh sách''.

6.3.6.3 Công cụ xác định decltype()

Chúng ta có thể sử dụng auto khi chúng ta có một trình khởi tạo phù hợp. Nhưng đôi khi, chúng ta muốn suy ra một kiểu mà không cần xác định một biến khởi tạo. Sau đó, chúng ta có thể sử dụng một công cụ xác định kiểu khai báo: decltype(expr) là kiểu khai báo của expr. Điều này chủ yếu hữu ích trong lập trình chung. Cân nhắc việc viết một hàm cộng hai ma trận với các loại phần tử có khả năng khác nhau. Loại kết quả của việc bổ sung là gì? Tất nhiên là một ma trận, nhưng loại phần tử của nó có thể là gì? Câu trả lời rõ ràng là loại phần tử của tổng là loại tổng của các phần tử. Vì vậy, tôi có thể tuyên bố:

```
mẫu<lớp T, lớp U> auto operator+
(const Matrix<T>& a, const Matrix<U>& b)   > Matrix<decltype(T{}+U{})>;
```

Tôi sử dụng cú pháp kiểu trả về hậu tố (§12.1) để có thể diễn đạt kiểu trả về theo các đối số: Matrix<decltype(T{}+U{})>. Nghĩa là, kết quả là một Ma trận với loại phần tử là những gì bạn nhận được từ việc thêm một cặp phần tử từ đối số Ma trận : T{}+U{}.

Trong định nghĩa, tôi lại cần decltype() để thể hiện kiểu phần tử của Matrix :

```
mẫu<lớp T, lớp U> auto operator+
(const Matrix<T>& a, const Matrix<U>& b)   > Matrix<decltype(T{}+U{})> {

        Ma trận<decltype(T{}+U{})> res; for (int
        i=0; i!=a.rows(); ++i)
                for (int j=0; j!=a.cols(); ++j)
                        res(i,j) += a(i,j) + b(i,j);
        trả lại độ phân giải;
}
```

6.4 Đối tượng và Giá trị

Chúng ta có thể phân bổ và sử dụng các đối tượng không có tên (ví dụ: được tạo bằng new) và có thể gán cho các biểu thức trông lạ (ví dụ: p[a+10]=7). Do đó, chúng ta cần một cái tên cho ''thứ gì đó trong bộ nhớ''. Đây là khái niệm cơ bản và đơn giản nhất về một đối tượng. Nghĩa là, một đối tượng là một vùng lưu trữ liền kề; một giá trị là một biểu thức đề cập đến một đối tượng. Từ ''lvalue'' ban đầu được đặt ra để có nghĩa là ''thứ gì đó có thể ở phía bên trái của một nhiệm vụ.'' Tuy nhiên, không phải mọi giá trị đều có thể được sử dụng ở phía bên trái của một nhiệm vụ; MỘT

Machine Translated by Google

lvalue có thể tham chiếu đến một hằng số (§7.7). Một giá trị chư a đư ợc khai báo const thư ờng đư ợc gọi là giá trị có thể sửa đổi. Không nên nhầm lẫn khái niệm cấp thấp và đơ n giản này của một đối tư ợng với các khái niệm về đối tư ợng lớp và đối tư ợng kiểu đa hình (§3.2.2, §20.3.2).

6.4.1 Lvalues và Rvalues

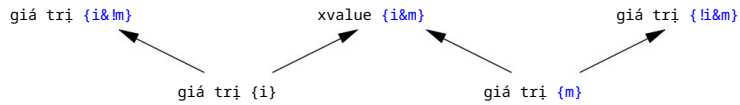
Để bổ sung cho khái niệm về một giá trị, chúng ta có khái niệm về một giá trị. Đại khái, rvalue có nghĩa là ''một giá trị không phải là một giá trị'', chẳng hạn như một giá trị tạm thời (ví dụ: giá trị đư ợc trả về bởi một hàm).

Nếu bạn cần kỹ thuật hơ n (ví dụ, vì bạn muốn đọc tiêu chuẩn ISO C++), bạn cần có cái nhìn tinh tế hơ n về lvalue và rvalue. Có hai thuộc tính quan trọng đối với một đối tư ợng khi đề cập đến việc định địa chỉ, sao chép và di chuyển:

- Có nhận dạng: Chư ơ ng trình có tên, con trỏ tới hoặc tham chiếu tới đối tư ợng để có thể xác định xem hai đối tư ợng có phải là đối tư ợng không. giống nhau, liệu giá trị của đối tư ợng có thay đổi hay không, v.v.

- Có thể di chuyển: Đối tư ợng có thể đư ợc di chuyển từ (tức là chúng ta đư ợc phép di chuyển giá trị của nó đến vị trí khác và để đối tư ợng ở trạng thái hợp lệ như ng không xác định, chứ không phải sao chép; §17.5).

Nó chỉ ra rằng ba trong số bốn kết hợp có thể có của hai thuộc tính đó là cần thiết để mô tả chính xác các quy tắc ngôn ngữ C++ (chúng ta không cần các đối tư ợng không có danh tính và không thể di chuyển). Sử dụng ''m cho di động'' và ''i cho có danh tính'', chúng ta có thể biểu thị sự phân loại các biểu thức này bằng đồ thị:



Vì vậy, một giá trị cổ điển là thứ có bản sắc và không thể di chuyển (vì chúng ta có thể kiểm tra nó sau khi di chuyển) và giá trị cổ điển là bất kỳ thứ gì mà chúng ta đư ợc phép di chuyển từ đó. Các lựa chọn thay thế khác là prvalue (''giá trị thuần túy''), glvalue (''giá trị tổng quát'') và xvalue (''x'' cho ''ex traordinary'' hoặc ''chỉ dành cho chuyên gia''; các đề xuất cho ý nghĩa của chữ ''x'' này hoàn toàn có nguồn gốc từ trí tư ởng tư ợng). Ví dụ:

```
void f(vector<string>& vs) {

        vector<string>& v2 = std::move(vs);                    // di chuyển so với đến v2
        // ...
}
```

Ở đây, std::move(vs) là một xvalue: nó rõ ràng có danh tính (chúng ta có thể gọi nó là vs), như ng rõ ràng chúng ta đã cho phép nó đư ợc di chuyển từ đó bằng cách gọi std::move() (§ 3.3.2, §35.5.1).

Đối với lập trình thực tế, suy nghĩ về giá trị và giá trị thư ờng là đủ. Ghi chú rằng mọi biểu thức là một giá trị hoặc một giá trị, như ng không phải cả hai.

6.4.2 Thời gian tồn tại của các đối tư ợng

Thời gian tồn tại của một đối tư ợng bắt đầu khi hàm tạo của nó hoàn thành và kết thúc khi hàm hủy của nó bắt đầu thực thi. Các đối tư ợng thuộc loại không có hàm tạo đư ợc khai báo, chẳng hạn như int, có thể đư ợc coi là có hàm tạo và hàm hủy mặc định không làm gì cả.

Machine Translated by Google

Chúng ta có thể phân loại các đối tượng dựa trên thời gian tồn tại của chúng:

- Tự động: Trừ khi người lập trình chỉ định khác (§12.1.8, §16.2.12), một đối tượng được khai báo trong một hàm được tạo khi gặp định nghĩa của nó và bị hủy khi tên của nó không còn nữa. phạm vi. Những đối tượng như vậy đôi khi được gọi là đối tượng tự động . Trong một triển khai điển hình, các đối tượng tự động được phân bổ trên ngăn xếp; mỗi lần gọi hàm có khung ngăn xếp riêng để giữ các đối tượng tự động của nó.
- Tĩnh: Các đối tượng được khai báo trong phạm vi toàn cục hoặc không gian tên (§6.3.4) và các tĩnh được khai báo trong hàm (§12.1.8) hoặc lớp (§16.2.12) được tạo và khởi tạo một lần (chỉ) và ''sống'' cho đến khi chương trình kết thúc (§15.4.3). Những đối tượng như vậy được gọi là đối tượng tĩnh . Một đối tượng tĩnh có cùng một địa chỉ trong suốt thời gian thực hiện chương trình. Các đối tượng tĩnh có thể gây ra sự cố nghiêm trọng trong chương trình đa luồng vì chúng được chia sẻ giữa tất cả các luồng và thường yêu cầu khóa để tránh chạy đua dữ liệu (§5.3.1, §42.3).
- Free store: Sử dụng toán tử new và delete , chúng ta có thể tạo các đối tượng có thời gian tồn tại được kiểm soát trực tiếp (§11.2).
- Các đối tượng tạm thời (ví dụ: các kết quả trung gian trong một phép tính hoặc một đối tượng được sử dụng để giữ giá trị cho một tham chiếu đến đối số const): thời gian tồn tại của chúng được xác định bởi việc sử dụng chúng. Nếu chúng bị ràng buộc với một tham chiếu, thì thời gian tồn tại của chúng là của tham chiếu đó; mặt khác, chúng ''sống'' cho đến khi kết thúc biểu hiện đầy đủ mà chúng là một phần. Một biểu thức đầy đủ là một biểu thức không phải là một phần của biểu thức khác. Thông thường, các đối tượng tạm thời là tự động.
- Đối tượng luồng cục bộ ; nghĩa là, các đối tượng được khai báo thread_local (§42.2.8): các đối tượng như vậy được tạo khi luồng của chúng tồn tại và bị hủy khi luồng của chúng tồn tại.

Tĩnh và tự động thường được gọi là các lớp lưu trữ.

Các phần tử mảng và các thành viên lớp không tĩnh có thời gian tồn tại của chúng được xác định bởi đối tượng mà chúng là một phần.

6.5 Loại bí danh

Đôi khi, chúng ta cần một tên mới cho một loại. Những lý do có thể bao gồm:

- Tên ban đầu quá dài, phức tạp hoặc xấu xí (trong mắt một số lập trình viên).
- Một kỹ thuật lập trình yêu cầu các loại khác nhau có cùng tên trong một ngữ cảnh.
- Một loại cụ thể chỉ được đề cập ở một nơi để đơn giản hóa việc bảo trì.

Ví dụ:

```
sử dụng Pchar = char  ; sử          // con trỏ tới ký tự con
dụng PF = int(  )(double);           // trỏ tới hàm lấy gấp đôi và trả về int a an
```

Các loại tương tự có thể xác định cùng tên với bí danh thành viên:

```
mẫu <lớp T> vectơ  lớp
{ sử dụng
      value_type = T;                    // mỗi mộtcontainer có      value_type
      // ...
};
```

Machine Translated by Google


```
mẫu <lớp T> danh sách
lớp { sử dụng
      value_type = T; //                    // mỗi một container có      value_type
        ...
};
```

Đối với tốt và xấu, bí danh loại là từ đồng nghĩa với các loại khác chứ không phải là loại riêng biệt. Nghĩa là, một bí danh đề cập đến loại mà nó là một bí danh. Ví dụ:

```
Pchar p1 = nullptr;          // p1 là một ký tự
char   p3 = p1;              // * tốt
```

Những ngư ời muốn có các loại riêng biệt với ngữ nghĩa giống hệt nhau hoặc biểu diễn giống hệt nhau nên xem các kiểu liệt kê (§8.4) và các lớp (Chư ơ ng 16).

Một cú pháp cũ hơ n sử dụng từ khóa typedef và đặt tên đư ợc khai báo ở nơ i nó sẽ have been in a khai báo của một biến có thể đư ợc sử dụng tư ơ ng đư ơ ng trong nhiều ngữ cảnh. Ví dụ:

```
typedef int int32_t; gõ              // tư ơ ng đư ơ ngĐẾN '' sử dụng int32_t   = int;''
ngắn int16_t; typedef                  với tư ơ ng đư ơ ng'' sử dụng int16_t   = ngắn;''
void(  PtoF)(int);                   // //          ĐẾN '' sử dụngPtoF   = khoảng trống(*)(int);''
```

Bí danh đư ợc sử dụng khi chúng tôi muốn cách ly mã của mình khỏi các chi tiết của máy bên dư ới. Tên int32_t chỉ ra rằng chúng tôi muốn nó đại diện cho một số nguyên 32 bit. Sau khi viết mã của chúng ta dư ới dạng int32_t, thay vì '' int đơ n giản'', chúng ta có thể chuyển mã của mình sang một máy có sizeof(int)==2 bằng cách xác định lại lần xuất hiện duy nhất của int32_t trong mã của chúng ta để sử dụng một số nguyên dài hơ n:

```
sử dụng int32_t = dài;
```

Hậu tố _t là quy ư ớc cho các bí danh (''typedefs''). Bạn có thể tìm thấy int16_t , int32_t và các bí danh khác như vậy trong <stdint> (§43.7). Lư u ý rằng việc đặt tên một loại theo biểu diễn của nó chứ không phải theo mục đích của nó không nhất thiết là một ý tư ởng hay (§6.3.3).

Từ khóa sử dụng cũng có thể đư ợc sử dụng để giới thiệu bí danh mẫu (§23.6). Ví dụ:

```
mẫu<typename T> sử dụng Vector
      = std::vector<T, My_allocator<T>>;
```

Chúng tôi không thể áp dụng các trình xác định loại, chẳng hạn như không dấu, cho một bí danh. Ví dụ:

```
sử dụng Char = char; sử
dụng Uchar = unsigned Char; sử dụng          // lỗi
Uchar = unsigned char;                       // đư ợc rồi
```

6.6 Lời khuyên

[1] Để biết ý kiến cuối cùng về vấn đề định nghĩa ngôn ngữ, hãy xem tiêu chuẩn ISO C++; §6.1.

[2] Tránh hành vi không xác định và không xác định; §6.1.

Cô lập mã phải phụ thuộc vào hành vi do triển khai xác định; §6.1. [3]

[4] Tránh những giả định không cần thiết về giá trị số của ký tự; §6.2.3.2, §10.5.2.1.

[5] Hãy nhớ rằng số nguyên bắt đầu bằng số 0 là số bát phân; §6.2.4.1.

Machine Translated by Google

[6] Tránh ''hằng số ma thuật''; §6.2.4.1.

[7] Tránh các giả định không cần thiết về kích thư ớc của số nguyên; §6.2.8.

[8] Tránh các giả định không cần thiết về phạm vi và độ chính xác của các loại dấu phẩy động; §6.2.8.

[9] Thích ký tự đơ n giản hơ n ký tự ký và ký tự không ký; §6.2.3.1.

[10] Coi chừng chuyển đổi giữa các loại có dấu và không dấu; §6.2.3.1.

[11] Khai một tên (chỉ) một tờ khai; §6.3.2.

[12] Giữ các tên phổ biến và địa phư ơ ng ngắn, đồng thời giữ các tên không phổ biến và phi địa phư ơ ng dài hơ n; §6.3.3.

[13] Tránh những cái tên na ná nhau; §6.3.3.

[14] Đặt tên cho một đối tư ợng để phản ánh ý nghĩa của nó hơ n là loại của nó; §6.3.3.

[15] Duy trì cách đặt tên nhất quán; §6.3.3.

[16] Tránh tên ALL_CAPS ; §6.3.3.

[17] Giữ phạm vi nhỏ; §6.3.4.

[18] Không sử dụng cùng một tên trong cả phạm vi và phạm vi kèm theo; §6.3.4.

[19] Thích cú pháp {}-initializer cho các khai báo có loại đư ợc đặt tên; §6.3.5.

[20] Thích cú pháp = để khởi tạo trong các khai báo sử dụng auto; §6.3.5.

[21] Tránh các biến chư a khởi tạo; §6.3.5.1.

[22] Sử dụng bí danh để xác định tên có ý nghĩa cho loại tích hợp sẵn trong trư ờng hợp loại tích hợp dùng để biểu thị giá trị có thể thay đổi; §6.5.

[23] Sử dụng bí danh để xác định từ đồng nghĩa cho các loại; sử dụng kiểu liệt kê và lớp để xác định kiểu mới; §6.5.

Machine Translated by Google

Trang này cố ý để trống

Machine Translated by Google

7

Con trỏ, Mảng và Tham chiếu

Điều siêu phàm và lố bịch thư ờng
gần giống nhau đến mức khó có thể
phân loại chúng một cách riêng biệt.
– Thomas Paine

• Giới thiệu •
Con trỏ
khoảng trống ;
nullptr • Mảng
Bộ khởi tạo mảng; Chuỗi ký tự • Con
trỏ vào mảng
Điều hư ớng mảng; Mảng nhiều chiều; Truyền mảng • Con trỏ và const
• Con trỏ và quyền sở
hữu • Tham chiếu

Lvalue Tài liệu tham khảo; Giá trị tài liệu tham khảo; Tài liệu tham khảo Tài liệu tham khảo; Con trỏ và Tham chiếu • Lời khuyên

7.1 Giới thiệu

Chư ơ ng này đề cập đến các cơ chế ngôn ngữ cơ bản để đề cập đến bộ nhớ. Rõ ràng, chúng ta có thể tham chiếu đến một đối tư ợng theo tên, như ng trong C++ (hầu hết) các đối tư ợng ''có danh tính''. Nghĩa là, chúng cư trú tại một địa chỉ cụ thể trong bộ nhớ và một đối tư ợng có thể đư ợc truy cập nếu bạn biết địa chỉ và địa chỉ của nó. kiểu. Cấu trúc ngôn ngữ để giữ và sử dụng địa chỉ là con trỏ và tham chiếu.

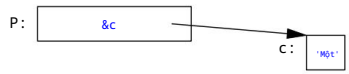
Machine Translated by Google

7.2 Con trỏ

Đối với kiểu T, T là kiểu ''con trỏ tới T.'' Nghĩa là, một biến kiểu T có thể chứa địa chỉ của một đối tư ợng kiểu T. Ví dụ:

```
ký tự c = 'a';
char  p = &c;          //  giữ địa chỉ của c; & là địa chỉ của toán tử p
```

hoặc đồ họa:



Thao tác cơ bản trên một con trỏ là hủy tham chiếu, nghĩa là tham chiếu đến đối tư ợng đư ợc trỏ tới bởi con trỏ. Hoạt động này còn đư ợc gọi là gián tiếp. Toán tử hủy hội nghị là (tiền tố) đơ n nguyên . Ví dụ:

```
ký tự c = 'a';
char  p = &c; // char   giữ địa chỉ của c; & là địa chỉ của toán tử p c2
c2 =  p; //              ==  'Một'; * là toán tử quy ư ớc
```

Đối tư ợng đư ợc trỏ tới bởi p là c và giá trị đư ợc lư u trong c là 'a', vì vậy giá trị của p đư ợc gán cho c2 là 'a'.

Có thể thực hiện một số phép tính số học trên con trỏ tới phần tử mảng (§7.4).

Việc triển khai con trỏ nhằm mục đích ánh xạ trực tiếp tới các cơ chế định địa chỉ của máy mà chư ơ ng trình chạy trên đó. Hầu hết các máy có thể giải quyết một byte. Những thứ không thể có phần cứng để trích xuất byte từ các từ. Mặt khác, một số máy có thể xử lý trực tiếp một bit riêng lẻ. Do đó, đối tư ợng nhỏ nhất có thể đư ợc phân bổ độc lập và trỏ đến bằng cách sử dụng loại con trỏ dựng sẵn là một ký tự. Lư u ý rằng một bool chiếm ít không gian nhất bằng một char (§6.2.8). Để lư u trữ các giá trị nhỏ hơ n một cách gọn gàng hơ n, bạn có thể sử dụng các phép toán logic theo bit (§11.1.1), trư ờng bit trong cấu trúc (§8.2.7) hoặc tập hợp bit (§34.2.2).

, nghĩa là ''con trỏ tới'', đư ợc sử dụng làm hậu tố cho tên loại. Thật không may, con trỏ đến mảng và con trỏ tới hàm cần một ký hiệu phức tạp hơ n:

```
int                  // con trỏ tới int
pi; char               con trỏ tới con trỏ tới char
ppc; int             // mảng  trong số 15 con trỏ tới
ap[15]; int (  fp)     int con trỏ tới hàm lấy đối số char*; trả về int một hàm lấy
(char  ); int  f(char  );  // //  Một đối số char*; lợi nhuận  Một con trỏ tới int
```

Xem §6.3.1 để biết giải thích về cú pháp khai báo và §iso.A để biết ngữ pháp hoàn chỉnh.

Con trỏ tới các chức năng có thể hữu ích; chúng đư ợc thảo luận trong §12.5. Con trỏ tới các thành viên lớp đư ợc trình bày trong §20.6.

7.2.1 khoảng trống

Trong mã cấp thấp, đôi khi chúng ta cần lư u trữ hoặc chuyển địa chỉ của một vị trí bộ nhớ mà không thực sự biết loại đối tư ợng nào đư ợc lư u trữ ở đó. Một void đư ợc sử dụng cho điều đó. Bạn có thể đọc void dư ới dạng ''con trỏ tới một đối tư ợng thuộc loại không xác định.''

Machine Translated by Google

Một con trỏ tới bất kỳ loại đối tượng nào cũng có thể được gán cho một biến kiểu void , nhưng một con trỏ tới hàm (§12.5) hoặc một con trỏ tới thành viên (§20.6) thì không thể. Ngoài ra, một void có thể được gán cho một void khác, các void có thể được so sánh về đẳng thức và bất bình đẳng, và một void có thể được chuyển đổi rõ ràng thành một loại khác. Các hoạt động khác sẽ không an toàn vì trình biên dịch không thể biết loại đối tượng nào thực sự được trỏ tới. Do đó, các hoạt động khác dẫn đến lỗi thời gian biên dịch. Để sử dụng một void , chúng ta phải chuyển đổi nó một cách rõ ràng thành một con trỏ tới một kiểu cụ thể. Ví dụ:

```
khoảng trống f(int pi)
{
        void pv = pi; // // ok: chuyển đổi ngầm định của int* thành void*
                           không can't dereference lỗi void* :
          pv; // ++pv;      lỗi : thể tăng void* (kích thước của đối tượng trỏ tới không xác định)

        int pi2 = static_cast<int >(pv);                    // chuyển đổi rõ ràng trở lại int*

        double pd1 = pv; gấp                                  lỗi
        đôi pd2 = pi; double                                // lỗi
        pd3 = static_cast<double >(pv);                     // không an toàn (§11.5.2)
}
```

Nói chung, sẽ không an toàn khi sử dụng con trỏ đã được chuyển đổi (''cast'') thành loại khác với loại đối tượng được trỏ tới. Ví dụ, một máy có thể giả định rằng mỗi double được phân bổ trên một ranh giới 8 byte. Nếu vậy, hành vi lạ có thể phát sinh nếu pi chỉ vào một int không được phân bổ theo cách đó. Hình thức chuyển đổi loại rõ ràng này vốn không an toàn và xấu xí. Do đó, ký hiệu được sử dụng, static_cast (§11.5.2), được thiết kế để trở nên xấu xí và dễ tìm thấy trong mã.

Công dụng chính của void là để chuyển các con trỏ tới các hàm không được phép đưa ra các giả định về loại đối tượng và để trả về các đối tượng chưa được nhập từ các hàm. Để sử dụng một đối tượng như vậy, chúng ta phải sử dụng chuyển đổi kiểu tường minh.

Các chức năng sử dụng con trỏ void thường tồn tại ở mức thấp nhất của hệ thống, nơi các tài nguyên phần cứng thực được thao tác. Ví dụ:

```
void my_alloc(size_t n);          // phân bổ n byte từ đống đặc biệt của tôi
```

Sự xuất hiện của khoảng trống ở các cấp độ cao hơn của hệ thống nên được xem xét với sự nghi ngờ lớn vì chúng có khả năng là dấu hiệu của lỗi thiết kế. Khi được sử dụng để tối ưu hóa, void có thể được ẩn sau giao diện an toàn loại (§27.3.1).

Không thể gán con trỏ tới hàm (§12.5) và con trỏ tới thành viên (§20.6) cho void s.

7.2.2 nullptr

Chữ nullptr đại diện cho con trỏ null, nghĩa là một con trỏ không trỏ đến một đối tượng. Nó có thể được gán cho bất kỳ loại con trỏ nào, nhưng không được gán cho các loại tích hợp sẵn khác:

```
int pi = nullptr;
double pd = nullptr; int
tôi = nullptr;                // lỗi : tôi không phải là con trỏ a
```

Chỉ có một nullptr, có thể được sử dụng cho mọi loại con trỏ, thay vì một con trỏ null cho từng loại con trỏ.

Machine Translated by Google

Trư ớc khi nullptr đư ợc giới thiệu, số không (0) đã đư ợc sử dụng làm ký hiệu cho con trỏ null. Ví dụ:

```
int  x = 0; //  X nhận giá trị nullptr
```

Không có đối tư ợng nào đư ợc phân bổ với địa chỉ 0 và 0 (mẫu bit toàn số không) là cách diễn đạt phổ biến nhất của nullptr. Không (0) là một số nguyên. Tuy nhiên, các chuyển đổi tiêu chuẩn (§10.5.2.3) cho phép 0 đư ợc sử dụng làm hằng số của con trỏ hoặc loại con trỏ thành thành viên.

Việc xác định macro NULL để biểu diễn con trỏ null đã trở nên phổ biến. Ví dụ:

```
int  p = NULL; //      sử dụng macro NULL
```

Tuy nhiên, có sự khác biệt trong định nghĩa NULL trong các triển khai khác nhau; ví dụ, NULL có thể là 0 hoặc 0L. Trong C, NULL thư ờng là (void)0, khiến nó trở thành bất hợp pháp trong C++ (§7.2.1):

```
int  p = NULL; //      lỗi :  không thể gán void* cho int* a an
```

Việc sử dụng nullptr làm cho mã dễ đọc hơ n so với các lựa chọn thay thế và tránh sự nhầm lẫn tiềm ẩn khi một hàm bị quá tải để chấp nhận một con trỏ hoặc một số nguyên (§12.3.1).

7.3 Mảng

Đối với loại T, T[size] là loại ''mảng các phần tử kích thư ớc của loại T.'' Các phần tử đư ợc lập chỉ mục từ 0 đến size 1. Ví dụ:

```
float v[3];          // MỘT gồm 32 của ba số float: v[0], v[1], v[2] mảng
char  a[32];         //          con trỏ tới char: a[0] một mảng      ··một[31]
```

Bạn có thể truy cập một mảng bằng cách sử dụng toán tử chỉ số [] hoặc thông qua một con trỏ (sử dụng toán tử hoặc toán tử []; §7.4). Ví dụ:

```
khoảng trống
f() {
        int aa[10];
        aa[6] = 9;           // gán cho hành vi không xác định phần
        int x = aa[99]; //      tử thứ 7 của aa
}
```

Truy cập ra khỏi phạm vi của một mảng là không xác định và thư ờng là thảm họa. Cụ thể, kiểm tra phạm vi thời gian chạy không đư ợc đảm bảo và cũng không phổ biến.

Số phần tử của mảng, mảng bị ràng buộc, phải là một biểu thức không đổi (§10.4). Nếu bạn cần giới hạn biến, hãy sử dụng vectơ (§4.4.1, §31.4). Ví dụ:

```
khoảng trống f(int
n) {
        int v1[n]; // vecto<int>       hằng số: mảng a kích thư ớc không phải là lỗi biểu thức
        v2(n); //                      OK: vectơ với n        phần tử int
}
```

Mảng nhiều chiều đư ợc biểu diễn dư ới dạng mảng của mảng (§7.4.2).

Mảng là cách cơ bản của C++ để biểu diễn một chuỗi các đối tư ợng trong bộ nhớ. Nếu những gì bạn muốn là một chuỗi các đối tư ợng có độ dài cố định đơ n giản thuộc một loại nhất định trong bộ nhớ, thì một mảng là giải pháp lý tư ởng. Đối với mọi nhu cầu khác, một mảng có vấn đề nghiêm trọng.

Machine Translated by Google

Một mảng có thể được phân bổ tĩnh, trên ngăn xếp và trên kho lưu trữ miễn phí (§6.4.2). Ví dụ:

```
int a1[10];                    // 10 int trong bộ lưu trữ tĩnh

khoảng trống
f() {
        int a2[20];            // 20 int trên ngăn xếp
        int p = new int[40]; // 40 int cửa hàng miễn phí trên
          ...
}
```

Mảng dựng sẵn trong C++ là một cơ sở cấp thấp vốn có nên được sử dụng chủ yếu bên trong việc triển khai các cấu trúc dữ liệu cấp cao hơn, hoạt động tốt hơn, chẳng hạn như vectơ hoặc mảng thư viện chuẩn . Không có phép gán mảng và tên của một mảng chuyển đổi hoàn toàn thành một con trỏ tới phần tử đầu tiên của nó khi có sự khiêu khích nhỏ nhất (§7.4). Cụ thể, tránh sử dụng mảng trong giao diện (ví dụ: làm đối số hàm; §7.4.3, §12.2.2) vì việc chuyển đổi ngầm định thành con trỏ là nguyên nhân gốc rễ của nhiều lỗi phổ biến trong mã C và mã C++ kiểu C. Nếu bạn phân bổ một mảng trên cửa hàng miễn phí, hãy nhớ xóa [] con trỏ của nó một lần và chỉ sau lần sử dụng cuối cùng (§11.2.2). Điều đó được thực hiện dễ dàng và đáng tin cậy nhất bằng cách kiểm soát vòng đời của mảng lưu trữ miễn phí bởi một trình điều khiển tài nguyên (ví dụ: chuỗi (§19.3, §36.3), vector (§13.6, §34.2) hoặc unique_ptr (§34.3.1)). Nếu bạn phân bổ một mảng tĩnh hoặc trên ngăn xếp, hãy đảm bảo không bao giờ xóa [] mảng đó. Rõ ràng, các lập trình viên C không thể làm theo những lời khuyên này vì C thiếu khả năng đóng gói các mảng, nhưng điều đó không làm cho lời khuyên trở nên tồi tệ trong ngữ cảnh của C++.

Một trong những loại mảng được sử dụng rộng rãi nhất là mảng char có đầu cuối bằng 0. Đó là cách C lưu trữ các chuỗi, do đó, một mảng char không kết thúc thường được gọi là chuỗi kiểu C. Các ký tự chuỗi ký tự C++ tuân theo quy ước đó (§7.3.2) và một số hàm thư viện chuẩn (ví dụ: strcpy() và strcmp(); §43.4) dựa vào nó. Thông thường, một char hoặc một const char được coi là trỏ đến một chuỗi ký tự không có đầu cuối.

7.3.1 Trình khởi tạo mảng

Một mảng có thể được khởi tạo bởi một danh sách các giá trị. Ví dụ:

```
int v1[] = { 1, 2, 3, 4 }; char
v2[] = { 'a', 'b', 'c', 0 };
```

Khi một mảng được khai báo mà không có kích thước cụ thể nhưng có danh sách bộ khởi tạo, thì kích thước được tính bằng cách đếm các phần tử của danh sách bộ khởi tạo. Do đó, v1 và v2 lần lượt là kiểu int[4] và char[4] . Nếu kích thước được chỉ định rõ ràng, sẽ có lỗi khi đưa các phần tử dư thừa vào danh sách trình khởi tạo. Ví dụ:

```
char v3[2] = { 'a', 'b', 0 }; char        lỗi : quá nhiều trình khởi tạo
v4[3] = { 'a', 'b', 0 };                  // được rồi
```

Nếu trình khởi tạo cung cấp quá ít phần tử cho một mảng, thì 0 được sử dụng cho phần còn lại. Ví dụ:

```
int v5[8] = { 1, 2, 3, 4 };
```

tương đương với

```
int v5[] = { 1, 2, 3, 4, 0, 0, 0, 0};
```

Machine Translated by Google

Không có thao tác sao chép tích hợp sẵn cho mảng. Bạn không thể khởi tạo một mảng với một mảng khác (không phải ev en cùng loại), và không có phép gán mảng:

```
int v6[8] = v5; // v6 lỗi : không thể sao (không thể gán mảng MỘT int* thành một mảng)
= v5; // không có mảng chép lỗi gán:
```

Tư ơ ng tự, bạn không thể truyền mảng theo giá trị. Xem thêm §7.4.

Khi bạn cần gán cho một tập hợp các đối tư ợng, hãy sử dụng một vectơ (§4.4.1, §13.6, §34.2), một thay vào đó là mảng (§8.2.4) hoặc valarray (§40.5).

Một mảng các ký tự có thể đư ợc khởi tạo thuận tiện bằng một chuỗi ký tự (§7.3.2).

7.3.2 Chuỗi ký tự

Một chuỗi ký tự là một chuỗi ký tự đư ợc đặt trong dấu ngoặc kép:

```
"đây là một chuỗi"
```

Một chuỗi ký tự chứa nhiều hơ n một ký tự mà nó có vẻ có; nó đư ợc kết thúc bởi ký tự null, '\0', với giá trị 0. Ví dụ:

```
sizeof("Bohr")==5
```

Loại của một chuỗi ký tự là ''mảng có số ký tự const thích hợp '', vì vậy "Bohr" thuộc loại const char[5].

Trong C và trong mã C++ cũ hơ n, bạn có thể gán một chuỗi ký tự cho một ký tự không phải hằng :

```
khoảng trống
f() {
        char  p = "Plato"; // p[4]  11,  như ng đư ợc chấp nhận trong lỗi mã chuẩn trư ớc C++
        = 'và'; //                 lỗi gán cho const
}
```

Rõ ràng là sẽ không an toàn nếu nhận nhiệm vụ đó. Nó đã (và đang) là nguồn gốc của các lỗi tinh vi, vì vậy xin đừng phàn nàn quá nhiều nếu một số mã cũ không biên dịch đư ợc vì lý do này. Việc có các ký tự chuỗi ký tự bất biến không chỉ rõ ràng mà còn cho phép việc triển khai thực hiện các tối ư u hóa quan trọng theo cách lư u trữ và truy cập các ký tự chuỗi ký tự.

Nếu chúng tôi muốn một chuỗi mà chúng tôi đư ợc đảm bảo có thể sửa đổi, chúng tôi phải đặt các ký tự trong một mảng không const :

```
khoảng trống
f() {
        char p[] = "Zeno"; // // p[0]  là 5 char p một mảng
        'R';                       =  đư ợc mở
}
```

Một chuỗi ký tự đư ợc phân bổ tĩnh để có thể an toàn khi trả về một chuỗi ký tự từ một hàm. Ví dụ:

```
const char  error_message(int i) {

        // ...
        trả về "lỗi phạm vi";
}
```

Machine Translated by Google

Bộ nhớ chứa "lỗi phạm vi" sẽ không biến mất sau lệnh gọi error_message().

Việc hai chuỗi ký tự giống hệt nhau được phân bổ thành một mảng hay thành hai là việc triển khai được xác định (§6.1). Ví dụ:

```
const char  p = "Heraclitus";
const char  q = "Heraclitus";

khoảng trống
g() {
      if (p == q) cout << "một!\n"; //        // kết quả là xác định thực hiện
        ...
}
```

Lưu ý rằng == so sánh địa chỉ (giá trị con trỏ) khi được áp dụng cho con trỏ chứ không phải giá trị được trỏ tới.

Chuỗi trống được viết dưới dạng một cặp dấu ngoặc kép liền kề, "" và có kiểu const char[1]. Một ký tự của chuỗi rỗng là ký tự kết thúc '\0'.

Quy ước dấu gạch chéo ngược để biểu thị các ký tự không phải đồ họa (§6.2.3.2) cũng có thể được sử dụng trong một chuỗi. Điều này giúp có thể biểu thị dấu nháy kép (") và dấu gạch chéo ngược của ký tự thoát (\) trong một chuỗi. Ký tự như vậy phổ biến nhất cho đến nay là ký tự xuống dòng, '\n'.
Ví dụ:

```
cout<<"Bep khi tin nhắn kết thúc\a\n";
```

Ký tự thoát, '\a', là ký tự ASCII BEL (còn được gọi là cảnh báo), khiến âm thanh phát ra.

Không thể có một dòng mới ''thực'' trong một chuỗi ký tự (không thô):

```
"đây không phải là một
chuỗi mà là một lỗi cú pháp"
```

Các chuỗi dài có thể được ngắt bằng khoảng trắng để làm cho văn bản chương trình gọn gàng hơn. Ví dụ:

```
char alpha[] = "abcdefghijklmnopqrstuvwxyz"
                 "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

Trình biên dịch sẽ nối các chuỗi liền kề, do đó, alpha có thể được khởi tạo tương đương bằng một chuỗi

```
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

Có thể có ký tự null trong một chuỗi, nhưng hầu hết các chương trình sẽ không nghi ngờ rằng có các ký tự sau nó. Ví dụ: chuỗi "Jens\000Munk" sẽ được coi là "Jens" bởi các hàm thư viện tiêu chuẩn như strcpy() và strlen(); xem §43.4.

7.3.2.1 Chuỗi ký tự thô

Để thể hiện dấu gạch chéo ngược (\) hoặc dấu ngoặc kép (") trong một chuỗi ký tự, chúng ta phải đặt trước nó bằng dấu gạch chéo ngược. Điều đó hợp lý và trong hầu hết các trường hợp khá đơn giản. Tuy nhiên, nếu chúng ta cần nhiều dấu gạch chéo ngược và nhiều trong các ký tự chuỗi, kỹ thuật đơn giản này trở nên khó quản lý. Đặc biệt, trong các biểu thức chính quy, dấu gạch chéo ngược được sử dụng như một ký tự thoát và để giới thiệu các ký tự

Machine Translated by Google

đại diện cho các lớp ký tự (§37.1.1). Đây là một quy ư ớc đư ợc chia sẻ bởi nhiều ngôn ngữ lập trình, vì vậy chúng tôi không thể thay đổi nó. Do đó, khi bạn viết các biểu thức chính quy để sử dụng với thư viện biểu thức chính quy chuẩn (Chư ơ ng 37), thực tế là dấu gạch chéo ngư ợc là một ký tự thoát sẽ trở thành một nguồn lỗi đáng chú ý. Xem xét cách viết mẫu đại diện cho hai từ đư ợc phân tách bằng dấu gạch chéo ngư ợc (\):

```
chuỗi s = "\\w\\\\w";                // tôi hi vọng tôi hiểu đúng rồi
```

Để tránh sự thất vọng và lỗi gây ra bởi xung đột quy ư ớc này, C++ cung cấp các ký tự chuỗi thô. Một chuỗi ký tự thô là một chuỗi ký tự trong đó dấu gạch chéo ngư ợc chỉ là dấu gạch chéo ngư ợc (và dấu ngoặc kép chỉ là dấu ngoặc kép) để ví dụ của chúng ta trở thành:

```
chuỗi s = R"(\w\\w)";                // Tôi xinh đẹp Chắc chắn tôi hiểu đúng rồi
```

Chuỗi ký tự thô sử dụng ký hiệu R"(ccc)" cho một chuỗi ký tự ccc. R ban đầu ở đó để phân biệt chuỗi ký tự thô với chuỗi ký tự thông thư ờng. Dấu ngoặc đơ n ở đó để cho phép dấu ngoặc kép (''không thoát''). Ví dụ:

```
R"("chuỗi trích dẫn")"               // chuỗi là "chuỗi trích dẫn"
```

Vì vậy, làm cách nào để chuyển chuỗi ký tự)" thành một chuỗi ký tự thô? May mắn thay, đó là một vấn đề hiếm gặp, như ng "(và)" chỉ là cặp dấu phân cách mặc định. Chúng ta có thể thêm các dấu phân cách trư ớc (và sau) trong " (...)". Ví dụ:

```
R"     ("chuỗi trích dẫn chứa dấu kết thúc thông thư ờng (""))")     "
       // "chuỗi trích dẫn chứa dấu kết thúc thông thư ờng (""))"
```

Chuỗi ký tự sau dấu) phải giống hệt chuỗi ký tự trư ớc dấu (. Bằng cách này, chúng ta có thể đối phó với (gần như) các mẫu phức tạp tùy ý.

Trừ khi bạn làm việc với các biểu thức chính quy, các chuỗi ký tự thô có thể chỉ là một sự tò mò (và một điều nữa cần tìm hiểu), như ng các biểu thức chính quy rất hữu ích và đư ợc sử dụng rộng rãi. Hãy xem xét một ví dụ thực tế:

```
"('(?:[^\\\\']|\\\.)  '|\"(?:[^\\\\\"]|\\\.)  \")| "          // Là năm dấu gạch chéo ngư ợc chính xác hay    không?
```

Với các ví dụ như vậy, ngay cả các chuyên gia cũng dễ dàng bị nhầm lẫn và các chuỗi ký tự thô cung cấp một dịch vụ quan trọng.

Ngư ợc lại với các chuỗi ký tự không thô, một chuỗi ký tự thô có thể chứa một dòng mới. Ví dụ:

```
chuỗi đếm {R"(1 22 333)"};
```

tư ơ ng đư ơ ng với

```
chuỗi x {"1\n22\n333"};
```

7.3.2.2 Bộ ký tự lớn hơ n

Một chuỗi có tiền tố L, chẳng hạn như L"angst", là một chuỗi ký tự rộng (§6.2.3). Loại của nó là const wchar_t[]. Tư ơ ng tự, một chuỗi có tiền tố LR, chẳng hạn như LR"(angst)", là một chuỗi thô (§7.3.2.1) gồm các ký tự rộng thuộc loại const wchar_t[]. Một chuỗi như vậy đư ợc kết thúc bằng ký tự L'\0' .

Machine Translated by Google

Có sáu loại chữ ký tự hỗ trợ Unicode (chữ Unicode). Điều này nghe có vẻ thừa, như ng có ba bảng mã chính của Unicode: UTF-8, UTF-16 và UTF-32. Đối với mỗi trong số ba lựa chọn thay thế này, cả hai chuỗi thô và ''thông thư ờng'' đều đư ợc hỗ trợ. Cả ba mã hóa UTF đều hỗ trợ tất cả các ký tự Unicode, do đó bạn sử dụng ký tự nào tùy thuộc vào hệ thống mà bạn cần để phù hợp.
Về cơ bản, tất cả các ứng dụng Internet (ví dụ: trình duyệt và email) đều dựa trên một hoặc nhiều mã hóa này.

UTF-8 là mã hóa có độ rộng thay đổi: các ký tự phổ biến vừa với 1 byte, các ký tự ít đư ợc sử dụng hơ n (theo một số ư ớc tính sử dụng) thành 2 byte và các ký tự hiếm hơ n thành 3 hoặc 4 byte. Đặc biệt, các ký tự ASCII khớp với 1 byte có cùng mã hóa (giá trị nguyên) trong UTF-8 như trong ASCII. Các bảng chữ cái Latinh khác nhau, tiếng Hy Lạp, Cyrillic, tiếng Do Thái, tiếng Ả Rập, v.v. phù hợp với 2 byte.

Chuỗi UTF-8 đư ợc kết thúc bởi '\0', chuỗi UTF-16 bởi u'\0' và chuỗi UTF-32 bởi U'\0'.

Chúng ta có thể biểu diễn một chuỗi ký tự tiếng Anh thông thư ờng theo nhiều cách khác nhau. Hãy xem xét một tập tin tên sử dụng dấu gạch chéo ngư ợc làm dấu phân cách:

```
"thư mục"                  // chuỗi ký tự triển khai chuỗi ký tự
R"(thư mục\tệp)"             triển khai thô
u8"thư mục\\tệp"           // chuỗi UTF-8
u8R"(thư mục\tệp)" // u"thư  Chuỗi UTF-8 thô
mục\\tệp" //                 chuỗi UTF-16
uR"(thư mục\tệp)"                // Chuỗi UTF-16 thô
U"thư mục\\tệp"            // chuỗi UTF-32
UR"(thư mục\tệp)"                // UTF-32 str ing thô
```

Nếu đư ợc in ra, tất cả các chuỗi này sẽ trông giống nhau, như ng ngoại trừ các chuỗi ''plain'' và UTF-8, các biểu diễn bên trong của chúng có thể khác nhau.

Rõ ràng, mục đích thực sự của chuỗi Unicode là để có thể đặt các ký tự Unicode vào chúng.
Ví dụ:

```
u8"Các nguyên âm chính thức trong tiếng Đan Mạch là: a, e, i, o, u, \u00E6, \u00F8, \u00E5 và y."
```

In chuỗi đó một cách thích hợp mang lại cho bạn

```
Các nguyên âm chính thức trong tiếng Đan Mạch là: a, e, i, o, u, æ, ø, å và y.
```

Số thập lục phân sau \u là một điểm mã Unicode (§iso.2.14.3) [Unicode,1996]. Một điểm mã như vậy độc lập với mã hóa đư ợc sử dụng và trên thực tế sẽ có các biểu diễn khác nhau (dư ới dạng bit tính bằng byte) trong các mã hóa khác nhau. Ví dụ: u'0430' (chữ cái viết thư ờng Cyrillic ''a'') là giá trị thập lục phân 2 byte D0B0 trong UTF-8, giá trị thập lục phân 2 byte 0403 trong UTF-16 và giá trị thập lục phân 4 byte 00000403 trong UTF-32. Các giá trị thập lục phân này đư ợc gọi là tên ký tự chung.

Thứ tự của chúng tôi và R và trư ờng hợp của chúng là quan trọng: RU và Ur không phải là tiền tố chuỗi hợp lệ.

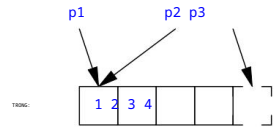
7.4 Con trỏ vào mảng

Trong C++, con trỏ và mảng có quan hệ mật thiết với nhau. Tên của một mảng có thể đư ợc sử dụng như một con trỏ tới phần tử ban đầu của nó. Ví dụ:

Machine Translated by Google


```
int v[] = { 1, 2, 3, 4 };
int   p1 = // v; int   p2 =  con trỏ tới phần tử ban đầu (chuyển đổi ngầm định) con
&v[0]; // int   p3 = v+4;    trỏ tới phần tử ban đầu con
                        // trỏ tới phần tử ngoài một phần tử cuối cùng
```

hoặc đồ họa:



Lấy một con trỏ tới phần tử nằm ngoài phần cuối của một mảng đư ợc đảm bảo hoạt động. Điều này rất quan trọng đối với nhiều thuật toán (§4.5, §33.1). Tuy nhiên, vì một con trỏ như vậy không thực sự trỏ đến một phần tử của mảng, nên nó có thể không đư ợc sử dụng để đọc hoặc ghi. Kết quả của việc lấy địa chỉ của phần tử trư ớc phần tử ban đầu hoặc sau phần tử một phần tử cuối cùng là không xác định và nên tránh. Ví dụ:

```
int   p4 = v  1; //  trư ớc      bắt đầu, không xác định:     đừng làm thế
int   p5 = v+7; //   ngoài kết thúc, không xác định: không làm điều đó
```

Việc chuyển đổi ngầm định một tên mảng thành một con trỏ tới phần tử ban đầu của mảng đư ợc sử dụng rộng rãi trong các lệnh gọi hàm trong mã kiểu C. Ví dụ:

```
bên ngoài "C" int strlen(const char  );          // từ <string.h>


khoảng trống
f() {
        char v[] = "Annemarie";
        char   p = v;      // chuyển đổi ngầm định char[] thành char*
        strlen(p);
        strlen(v);           chuyển đổi ngầm định char[] thành char*
        v = p;             // lỗi :   không thể gán cho mảng
}
```

Cùng một giá trị đư ợc chuyển đến hàm thư viện chuẩn `strlen()` trong cả hai lệnh gọi. Điều khó khăn là không thể tránh chuyển đổi ngầm định. Nói cách khác, không có cách nào để khai báo một hàm sao cho mảng `v` đư ợc sao chép khi hàm đư ợc gọi. May mắn thay, không có chuyển đổi rõ ràng hoặc ẩn từ một con trỏ thành một mảng.

Việc chuyển đổi ngầm định đối số mảng thành một con trỏ có nghĩa là kích thư ớc của mảng bị mất đối với hàm đư ợc gọi. Tuy nhiên, chức năng đư ợc gọi phải bằng cách nào đó xác định kích thư ớc để thực hiện một hoạt động có ý nghĩa. Giống như các hàm thư viện chuẩn C khác lấy con trỏ tới các ký tự, `strlen()` dựa vào số `0` để chỉ ra phần cuối của chuỗi; `strlen(p)` trả về số lư ợng ký tự tối đa và không bao gồm số `0` ở cuối. Tất cả đều ở mức khá thấp. `Vectơ` thư viện chuẩn (§4.4.1, §13.6, §31.4), `mảng` (§8.2.4, §34.2.1) và `chuỗi` (§4.2) không gặp sự cố này. Các loại thư viện này cung cấp số lư ợng phần tử dư ới dạng `size()` của chúng mà không cần phải đếm phần tử mỗi lần.

Machine Translated by Google

7.4.1 Điều hư ớng mảng

Truy cập mảng (và các cấu trúc dữ liệu tư ơ ng tự) hiệu quả và tinh tế là chìa khóa cho nhiều thuật toán (xem §4.5, Chư ơ ng 32). Có thể truy cập thông qua một con trỏ tới một mảng cộng với một chỉ mục hoặc thông qua một con trỏ tới một phần tử. Ví dụ:

```
khoảng trống fi(char v[])
{
      for (int i = 0; v[i]!=0; ++i)
            use(v[i]);
}

void fp(char v[]) {

      for (char  p = v;  p!=0; ++p)
            use(  p);
}
```

Toán tử tiền tố hủy tham chiếu một con trỏ sao cho p là ký tự đư ợc trỏ tới bởi p, và ++ tăng con trỏ để nó tham chiếu tới phần tử tiếp theo của mảng.

Không có lý do cố hữu tại sao một phiên bản phải nhanh hơ n phiên bản kia. Với các trình xếp chồng com hiện đại, mã giống hệt nhau phải đư ợc tạo (và thư ờng là) cho cả hai ví dụ. Các lập trình viên có thể lựa chọn giữa các phiên bản trên cơ sở hợp lý và thẩm mỹ.

Việc đăng ký một mảng dựng sẵn đư ợc xác định theo các phép toán con trỏ + và . Đối với mọi mảng dựng sẵn a và số nguyên j trong phạm vi của a, chúng ta có e:

```
a[j] ==  (&a[0]+j) ==  (a+j) ==  (j+a) == j[a]
```

Mọi ngư ời thư ờng ngạc nhiên khi thấy rằng a[j]==j[a]. Ví dụ: 3["Texas"]=="Texas"[3]=='a'. Sự thông minh như vậy không có chỗ trong mã sản xuất. Các giá trị tư ơ ng đư ơ ng này ở mức khá thấp và không phù hợp với các bộ chứa thư viện chuẩn, chẳng hạn như mảng và vectơ .

Kết quả của việc áp dụng các toán tử số học +, , ++ hoặc cho con trỏ phụ thuộc vào kiểu của đối tư ợng đư ợc trỏ tới. Khi một toán tử số học đư ợc áp dụng cho một con trỏ p kiểu T , p đư ợc coi là trỏ đến một phần tử của một mảng các đối tư ợng kiểu T; p+1 trỏ tới phần tử tiếp theo của mảng đó và p 1 trỏ tới phần tử trư ớc đó. Điều này ngụ ý rằng giá trị nguyên của p+1 sẽ lớn hơ n giá trị nguyên của p. Ví dụ:

```
mẫu<typename T> int
byte_diff(T  p, T  q) {

      return reinterpret_cast<char >(q) reinterpret_cast<char >(p);
}

void diff_test() {

      intvi[10];
      so với ngắn [10];
```

Machine Translated by Google


```
        cout << ta << '' << &vi[1] << '' << &bạn[1] &bạn[0] << '' << byte_diff(&vi[0],&vi[1]) << '\n';
        cout << vs << '' << &vs[1] << '' << &vs[1] &vs[0] << '' << byte_diff(&vs[0],&vs[1]) << '\n';
    }
```

Điều này tạo ra:

```
    0x7fffaef0 0x7fffaef4 1 4
    0x7fffaedc 0x7fffaede 1 2
```

Các giá trị con trỏ đư ợc in bằng cách sử dụng ký hiệu thập lục phân mặc định. Điều này cho thấy rằng khi tôi triển khai, `sizeof(short)` là `2` và `sizeof(int)` là 4.

Phép trừ con trỏ chỉ đư ợc xác định khi cả hai con trỏ trỏ đến các phần tử của cùng một mảng (mặc dù ngôn ngữ không có cách nhanh chóng để đảm bảo trư ờng hợp đó xảy ra). Khi trừ một con trỏ `p` khỏi một con trỏ q khác, `q p,` kết quả là số phần tử mảng trong dãy [p:q) (một số nguyên). Ngư ời ta có thể thêm một số nguyên vào một con trỏ hoặc trừ một số nguyên khỏi một con trỏ; trong cả hai trư ờng hợp, kết quả là một giá trị con trỏ. Nếu giá trị đó không trỏ đến một phần tử của cùng một mảng với con trỏ ban đầu hoặc một phần tử khác, thì kết quả của việc sử dụng giá trị đó là không xác định. Ví dụ:

```
    khoảng trống
    f() {
        int v1[10];
        int v2[10];

        int i1 = &v1[5] &v1[3]; // int   i1 = 2
        i2 = &v1[5] &v2[3]; //           kết quả không xác định

        int  p1 = v2+2;              // p1 = &v2[2]
        int  p2 = v2 2;              // *p2 không xác định
    }
```

Số học con trỏ phức tạp thư ờng không cần thiết và tốt nhất nên tránh. Việc bổ sung các con trỏ là vô nghĩa và không đư ợc phép.

Mảng không tự mô tả vì số phần tử của mảng không đảm bảo đư ợc lư u trữ cùng với mảng. Điều này ngụ ý rằng để duyệt qua một mảng không chứa dấu kết thúc theo cách mà các chuỗi kiểu C thực hiện, bằng cách nào đó chúng ta phải cung cấp số lư ợng phần tử. Ví dụ:

```
    void fp(char v[], int size) {

        for (int i=0; i!=size; ++i)
            use(v[i]);               // hy vọng rằng v có ít nhất các yếu tố kích thư ớc
        cho (int x : v)
            sử dụng(x);              // lỗi : phạm vi không hoạt động k cho con trỏ

        const int N = 7;
        ký tự v2[N];
        for (int i=0; i!=N; ++i)
            use(v2[i]);
        cho (int x : v2)
            sử dụng (x);             // phạm vi cho công việc ks cho mảng kích thư ớc đã biết
    }
```

Machine Translated by Google

Khái niệm mảng này vốn là cấp thấp. Hầu hết các ư u điểm của mảng tích hợp và một số ư u điểm của dis có thể đạt đư ợc thông qua việc sử dụng mảng bộ chứa thư viện tiêu chuẩn (§8.2.4, §34.2.1).
Một số triển khai C++ cung cấp kiểm tra phạm vi tùy chọn cho mảng. Tuy nhiên, việc kiểm tra như vậy có thể khá tốn kém, vì vậy nó thư ờng chỉ đư ợc sử dụng như một công cụ hỗ trợ phát triển (hơ n là đư ợc đư a vào mã sản xuất). Nếu bạn không sử dụng kiểm tra phạm vi cho các truy cập riêng lẻ, hãy cố gắng duy trì chính sách nhất quán là chỉ truy cập các phần tử trong các phạm vi đư ợc xác định rõ. Điều đó đư ợc thực hiện tốt nhất khi các mảng đư ợc thao tác thông qua giao diện của loại vùng chứa cấp cao hơ n, chẳng hạn như vectơ , nơ i khó bị nhầm lẫn hơ n về phạm vi của các phần tử hợp lệ.

7.4.2 Mảng nhiều chiều

Mảng nhiều chiều đư ợc biểu diễn dư ới dạng mảng của mảng; một mảng 3x5 đư ợc khai báo như sau:

```
int ma[3][5]; //      3 với 5 int mỗi mảng
```

Chúng ta có thể khởi tạo ma như thế này:

```
khoảng trống init_ma()
{
      for (int i = 0; i!=3; i++)
            for (int j = 0; j!=5; j++)
                  ma[i][j] = 10  i +j;
}
```

hoặc đồ họa:

Và:

00	01	02	03	04	10	11	12	13	14	20	21	22	23	24	

Mảng ma chỉ đơ n giản là 15 số nguyên mà chúng ta truy cập như thể nó là 3 mảng 5 số nguyên. Đặc biệt, không có đối tư ợng đơ n lẻ nào trong bộ nhớ là ma trận ma – chỉ có các phần tử đư ợc lư u trữ. Kích thư ớc 3 và 5 chỉ tồn tại trong nguồn trình biên dịch. Khi chúng tôi viết mã, công việc của chúng tôi là ghi nhớ chúng một số cách và cung cấp các kích thư ớc khi cần. Ví dụ: chúng tôi có thể in ma như thế này:

```
khoảng trống print_ma()
{
      for (int i = 0; i!=3; i++)
            { for (int j = 0; j!=5; j+
                  +) cout << ma[i][j] <<
            '\t'; cout << '\n';
      }
}
```

Ký hiệu dấu phẩy đư ợc sử dụng cho giới hạn mảng trong một số ngôn ngữ không thể đư ợc sử dụng trong C++ vì dấu phẩy (,) là toán tử sắp xếp thứ tự (§10.3.2). May mắn thay, hầu hết các sai lầm đều đư ợc phát hiện bởi com piler. Ví dụ:

```
int xấu[3,5];              // đổi: dấu phẩy    không đư ợc phép trong lỗi biểu thức không
int tốt[3][5];             // 3 với 5 int mỗi mảng int
int ouch = tốt[1,4]; int   //        khởi tạo bởi int* (good[1,4] có nghĩa là good[4], đó là lỗi : an     int*)
đẹp = tốt[1][4];
```

Machine Translated by Google

7.4.3 Truyền mảng

Mảng không thể đư ợc truyền trực tiếp theo giá trị. Thay vào đó, một mảng đư ợc truyền dư ới dạng một con trỏ tới phần tử đầu tiên của nó. Ví dụ:

```
void comp(double arg[10]) {                // arg là một double*

      for (int i=0; i!=10; ++i)
            arg[i]+=99;
}
```

```
khoảng trống
f() {
      gấp đôi a1[10];
      gấp đôi a2[5];
      gấp đôi a3[100];

      tính toán (a1);
      tính toán(a2);        // thảm họa!
      tính toán(a3);        // dụng chỉ 10 yếu tố đầu tiên sử
};
```

Mã này có vẻ lành mạnh, như ng không phải vậy. Mã biên dịch, như ng lệnh gọi comp(a2) sẽ ghi ngoài giới hạn của a2. Ngoài ra, bất kỳ ai đoán rằng mảng đư ợc truyền theo giá trị sẽ thất vọng: việc ghi vào arg[i] đư ợc ghi trực tiếp vào các phần tử của đối số comp() , thay vì vào một bản sao.
Hàm này có thể đư ợc viết tư ơ ng đư ơ ng như

```
void comp(double  arg) {

      for (int i=0; i!=10; ++i)
            arg[i]+=99;
}
```

Bây giờ sự điên rồ là (hy vọng) rõ ràng. Khi đư ợc sử dụng làm đối số hàm, kích thư ớc đầu tiên của một mảng đư ợc coi đơ n giản là một con trỏ. Bất kỳ mảng nào đư ợc chỉ định chỉ đơ n giản là bị bỏ qua. Điều này ngụ ý rằng nếu bạn muốn truyền một chuỗi các phần tử mà không làm mất thông tin về kích thư ớc, thì bạn không nên truyền một mảng dựng sẵn. Thay vào đó, bạn có thể đặt mảng bên trong một lớp với tư cách là thành viên (như đư ợc thực hiện đối với std::array) hoặc định nghĩa một lớp hoạt động như một bộ điều khiển (như đư ợc thực hiện đối với std::string và std::vector).

Nếu bạn khăng khăng sử dụng mảng trực tiếp, bạn sẽ phải giải quyết các lỗi và sự nhầm lẫn mà không nhận lại đư ợc những lợi ích đáng chú ý. Xem xét việc xác định một hàm để thao tác với ma trận hai chiều. Nếu kích thư ớc đư ợc biết tại thời điểm biên dịch, không có vấn đề gì:

```
void print_m35(int m[3][5]) {

      for (int i = 0; i!=3; i++)
            { for (int j = 0; j!=5; j+
                  +) cout << m[i][j] <<
            '\t'; cout << '\n';
      }
}
```

Machine Translated by Google

Ma trận được biểu diễn dưới dạng mảng nhiều chiều được truyền dưới dạng con trỏ (chứ không phải sao chép; §7.4). Kích thước đầu tiên của một mảng không liên quan đến việc tìm vị trí của một phần tử; nó chỉ đơn giản cho biết có bao nhiêu phần tử (ở đây, 3) thuộc loại thích hợp (ở đây, int[5]) . Ví dụ, nhìn vào bố cục của ma ở trên và lưu ý rằng chỉ cần biết chiều thứ hai là 5, chúng ta có thể định vị ma[i][5] cho bất kỳ i nào. Do đó, thứ nguyên đầu tiên có thể được chuyển thành đối số:

```
void print_mi5(int m[][5], int dim1) {

        for (int i = 0; i!=dim1; i++)
                { for (int j = 0; j!=5; j+
                        +) cout << m[i][j] <<
                '\t'; cout << '\n';
        }
}
```

Khi cả hai chiều cần được thông qua, '' giải pháp rõ ràng '' không hoạt động:

```
void print_me(int m[][], int dim1, int dim2) // {          không cư  xử như     hầu hết mọi người sẽ nghĩ

        for (int i = 0; i!=dim1; i++) {
                for (int j = 0; j!=dim2; j++)
                        cout << m[i][j] << '\t';        // TRÊN phích cắm!
                cout << '\n';
        }
}
```

May mắn thay, khai báo đối số m[][] là không hợp lệ vì chiều thứ hai của mảng đa chiều phải được biết để tìm vị trí của một phần tử. Tuy nhiên, biểu thức m[i][j] được diễn giải (chính xác) là ((m+i)+j), mặc dù đó không chắc là ý định của người lập trình. Một giải pháp đúng là:

```
void print_me(int   m, int dim1, int dim2) {

        for (int i = 0; i!=dim1; i++) {
                for (int j = 0; j!=dim2; j++)
                        cout << m[i  dim2+j] << '\t'; //   mơ  hồ
                cout << '\n';
        }
}
```

Biểu thức được sử dụng để truy cập các thành viên trong print_mij() tương đương với biểu thức mà trình biên dịch tạo ra khi biết kích thước cuối cùng.

Để gọi hàm này, chúng ta chuyển một ma trận dưới dạng một con trỏ bình thường:

```
kiểm tra int()
{
        int v[3][5] = {
                {0,1,2,3,4}, {10,11,12,13,14}, {20,21,22,23,24}
        };
```