Hướng dẫn Lập trình C của Beej

Brian “Beej Jorgensen” Hall (bản dịch tiếng Việt của Duc-Tam Nguyen)

Contents

Chapter 1

Lời nói đầu

> *C không phải là một ngôn ngữ lớn, và nó không hợp với một cuốn sách lớn.*
>
> –Brian W. Kernighan, Dennis M. Ritchie

Không có lý do gì để phí lời ở đây nữa, các bạn, ta nhảy thẳng vào code C luôn:

```
E((ck?main((z?(stat(M,&t)?P+=a+'{'?0:3:
execv(M,k),a=G,i=P,y=G&255,
sprintf(Q,y/'@'-3?A(*L(V(%d+%d)+%d,0)
```

Và họ sống hạnh phúc mãi mãi về sau. Hết.

Hử? Bạn bảo vẫn còn điều gì đó chưa rõ về cái ngôn ngữ lập trình C này?

Ừ thì, nói thật, chính tôi cũng không biết đoạn code trên làm gì. Nó là một mẩu trích từ một bài dự thi năm 2001 của International Obfuscated C Code Contest[1], một cuộc thi tuyệt vời mà người dự thi cố viết code C khó đọc nhất có thể, thường cho ra kết quả gây ngạc nhiên.

Tin xấu là nếu bạn mới bắt đầu với thứ này, mọi đoạn code C bạn thấy trông có lẽ đều giống như bị làm rối tung lên! Tin tốt là, cảm giác đó sẽ không kéo dài lâu đâu.

Điều tôi sẽ cố làm trong suốt hướng dẫn này là dẫn bạn từ trạng thái hoang mang toàn tập đến kiểu hạnh phúc tỉnh ngộ chỉ có thể đạt được qua lập trình C thuần túy. Cứ thế nhé.

Ngày xưa, C là một ngôn ngữ đơn giản hơn. Rất nhiều tính năng trong cuốn sách này cùng *một đống* tính năng trong tập Library Reference còn chưa tồn tại khi K&R viết ấn bản thứ hai nổi tiếng vào năm 1988. Dù vậy, phần lõi của ngôn ngữ vẫn nhỏ, và tôi hy vọng mình đã trình bày ở đây theo cách bắt đầu từ cái lõi đơn giản đó rồi mở rộng dần ra.

Và đó là lý do tôi bào chữa cho việc viết một cuốn sách to đến buồn cười về một ngôn ngữ nhỏ gọn và cô đọng như vậy.

1.1 Đối tượng đọc

Hướng dẫn này giả định rằng bạn đã có sẵn một chút kiến thức lập trình từ một ngôn ngữ khác, kiểu như Python[2], JavaScript[3], Java[4], Rust[5], Go[6], Swift[7], v.v. (Dân Objective-C[8] sẽ cực kỳ dễ thở!)

Chúng ta sẽ giả định là bạn biết biến là gì, vòng lặp làm gì, hàm hoạt động ra sao, và đại loại thế.

[1] https://www.ioccc.org/
[2] https://en.wikipedia.org/wiki/Python_(programming_language)
[3] https://en.wikipedia.org/wiki/JavaScript
[4] https://en.wikipedia.org/wiki/Java_(programming_language)
[5] https://en.wikipedia.org/wiki/Rust_(programming_language)
[6] https://en.wikipedia.org/wiki/Go_(programming_language)
[7] https://en.wikipedia.org/wiki/Swift_(programming_language)
[8] https://en.wikipedia.org/wiki/Objective-C

Nếu điều đó không đúng với bạn vì lý do nào đi nữa, thì điều tốt nhất tôi có thể hy vọng cung cấp là một chút giải trí chân thành cho niềm vui đọc sách của bạn. Điều duy nhất tôi có thể hứa một cách hợp lý là hướng dẫn này sẽ không kết thúc ở một nút thắt hồi hộp... hay *là* sẽ kết thúc như thế?

1.2 Cách đọc cuốn sách này

Hướng dẫn chia làm hai tập, và đây là tập đầu: tập hướng dẫn!

Tập thứ hai là library reference[9], và nó mang tính tham khảo hơn là hướng dẫn nhiều.

Nếu bạn là người mới, hãy đi qua phần hướng dẫn theo thứ tự, nói chung là vậy. Càng lên cao trong các chương thì thứ tự càng bớt quan trọng.

Và dù trình độ của bạn đến đâu, phần tham khảo luôn sẵn ở đó với các ví dụ đầy đủ về những hàm trong thư viện chuẩn, giúp bạn làm mới trí nhớ bất cứ khi nào cần. Hợp để đọc khi đang ăn một tô ngũ cốc hoặc trong những lúc rảnh khác.

Cuối cùng, liếc qua phần mục lục (nếu bạn đang đọc bản in), các mục thuộc phần tham khảo được in nghiêng.

1.3 Nền tảng và trình biên dịch

Tôi sẽ cố bám vào C chuẩn ISO kiểu cũ[10]. Ờ, phần lớn thôi. Đôi khi tôi có thể nổi hứng mà nói về POSIX[11] hay gì đó, nhưng để xem đã.

Người dùng **Unix** (ví dụ Linux, BSD, v.v.) thử chạy `cc` hoặc `gcc` từ dòng lệnh, biết đâu bạn đã có sẵn một trình biên dịch cài rồi. Nếu chưa, tìm trong bản phân phối của bạn cách cài `gcc` hoặc `clang`.

Người dùng **Windows** nên xem qua Visual Studio Community[12]. Hoặc, nếu bạn muốn trải nghiệm kiểu Unix hơn (rất khuyến khích!), cài WSL[13] và `gcc`.

Người dùng **Mac** sẽ muốn cài XCode[14], và đặc biệt là bộ command line tools.

Có cả tá trình biên dịch ngoài kia, và hầu như tất cả đều dùng được cho cuốn sách này. Một trình biên dịch C++ cũng sẽ biên dịch được phần lớn (nhưng không phải tất cả!) code C. Tốt nhất là dùng một trình biên dịch C đúng nghĩa nếu được.

1.4 Trang chủ chính thức

Vị trí chính thức của tài liệu này là https://beej.us/guide/bgc/[15]. Có thể điều này sẽ thay đổi trong tương lai, nhưng khả năng cao hơn là mọi hướng dẫn khác sẽ được dời khỏi máy tính ở Chico State.

1.5 Chính sách email

Tôi thường có mặt để giúp trả lời các câu hỏi qua email, nên cứ viết cho tôi, nhưng tôi không thể bảo đảm sẽ trả lời. Tôi có một cuộc sống khá bận rộn và có những lúc đơn giản là không thể trả lời câu hỏi của bạn. Khi đó, thường là tôi xóa tin nhắn đi luôn. Không có gì cá nhân cả; chỉ là tôi sẽ không bao giờ có đủ thời gian để đưa ra câu trả lời chi tiết mà bạn cần.

Theo nguyên tắc chung, câu hỏi càng phức tạp thì khả năng tôi trả lời càng thấp. Nếu bạn thu hẹp được câu hỏi trước khi gửi và nhớ đính kèm mọi thông tin liên quan (như nền tảng, trình biên dịch, thông báo lỗi bạn đang nhận được, và bất cứ thứ gì bạn nghĩ có thể giúp tôi tìm ra vấn đề), khả năng có hồi âm sẽ cao hơn nhiều.

[9] https://beej.us/guide/bgclr/
[10] https://en.wikipedia.org/wiki/ANSI_C
[11] https://en.wikipedia.org/wiki/POSIX
[12] https://visualstudio.microsoft.com/vs/community/
[13] https://docs.microsoft.com/en-us/windows/wsl/install-win10
[14] https://developer.apple.com/xcode/
[15] https://beej.us/guide/bgc/

Nếu bạn không nhận được hồi âm, cứ tiếp tục mò mẫm, cố tự tìm ra câu trả lời, và nếu vẫn không ra, viết lại cho tôi với thông tin đã tìm được, hy vọng khi đó sẽ đủ để tôi giúp đỡ.

Giờ mà tôi đã cằn nhằn xong về chuyện viết hay không viết email cho tôi, chỉ xin nói thêm rằng tôi *thực sự* trân trọng mọi lời khen mà cuốn hướng dẫn này đã nhận được suốt những năm qua. Nó là một liều tinh thần thật sự, và tôi vui khi biết nó đang được dùng vào việc tốt! `:-)` Cảm ơn bạn!

1.6 Sao lưu (mirror)

Bạn hoàn toàn được hoan nghênh sao lưu trang này, dù là công khai hay riêng tư. Nếu bạn mirror công khai và muốn tôi liên kết tới bản của bạn từ trang chính, cứ gửi cho tôi một dòng ở `beej@beej.us`.

1.7 Ghi chú cho người dịch

Nếu bạn muốn dịch hướng dẫn này sang một ngôn ngữ khác, hãy viết cho tôi tại `beej@beej.us` và tôi sẽ liên kết tới bản dịch của bạn từ trang chính. Cứ thoải mái thêm tên và thông tin liên hệ của bạn vào bản dịch.

Xin lưu ý các điều khoản giấy phép ở mục Bản quyền và Phân phối bên dưới.

1.8 Bản quyền và Phân phối

1.9 Lời tri ân

Những điều khó nhất khi viết các hướng dẫn này là:

- Học tài liệu đủ kỹ để có thể giảng lại
- Tìm ra cách giải thích rõ ràng nhất, một quá trình lặp đi lặp lại tưởng như không có hồi kết
- Tự đặt mình vào vai kẻ được gọi là *người có thẩm quyền*, trong khi thật ra tôi chỉ là một người bình thường đang cố hiểu mọi thứ, giống như mọi người khác thôi
- Kiên trì khi có biết bao thứ khác kéo sự chú ý của tôi đi chỗ khác

Rất nhiều người đã giúp tôi đi qua quá trình này, và tôi muốn ghi nhận những người đã khiến cuốn sách này thành sự thật.

- Mọi người trên Internet đã quyết định chia sẻ kiến thức của mình dưới hình thức này hay hình thức khác. Chính việc tự do chia sẻ các thông tin mang tính hướng dẫn đã khiến Internet trở thành nơi tuyệt vời như hiện nay.
- Những tình nguyện viên ở cppreference.com[16], những người bắc chiếc cầu nối từ bản spec sang thế giới thực.

[16] https://en.cppreference.com/

- Các cao nhân thân thiện trên comp.lang.c[17] và r/C_Programming[18], những người đã kéo tôi qua những phần khó nhằn của ngôn ngữ.
- Tất cả những ai đã gửi sửa lỗi và pull request cho mọi thứ, từ hướng dẫn gây hiểu lầm cho đến lỗi chính tả.

Cảm ơn bạn! ♥

[17] https://groups.google.com/g/comp.lang.c
[18] https://www.reddit.com/r/C_Programming/

Chapter 2

Hello, World!

2.1 Kỳ vọng gì từ C

> *"Mấy cái cầu thang này dẫn đi đâu?"*
> *"Nó dẫn đi lên."*
>
> —Ray Stantz và Peter Venkman, Ghostbusters

C là một ngôn ngữ cấp thấp.

Nó đâu từng như vậy. Thời xa xưa khi người ta còn đục thẻ bìa đục lỗ từ đá hoa cương, C là một cách tuyệt vời để thoát khỏi cực hình của các ngôn ngữ cấp thấp hơn như assembly[1].

Nhưng ở thời hiện đại này, các ngôn ngữ thế hệ mới cung cấp đủ thứ tính năng không tồn tại vào năm 1972 khi C được phát minh. Điều đó có nghĩa là C là một ngôn ngữ khá cơ bản với không nhiều tính năng. Nó có thể làm *mọi thứ*, nhưng sẽ bắt bạn đổ mồ hôi cho chúng.

Vậy tại sao ta vẫn còn dùng C đến bây giờ?

- Như một công cụ học tập: C không chỉ là một mảnh lịch sử đáng kính của ngành máy tính, mà còn kết nối với phần cứng thô[2] (bare metal) theo cách mà các ngôn ngữ hiện thời không có. Khi học C, bạn học về việc phần mềm tương tác với bộ nhớ máy tính ở cấp độ thấp như thế nào. Không có dây an toàn. Bạn sẽ viết ra các phần mềm bị crash, tôi bảo đảm với bạn. Và đó là một phần của cuộc vui!
- Như một công cụ hữu ích: C vẫn còn được dùng cho một số ứng dụng nhất định, chẳng hạn như xây dựng hệ điều hành[3] hay trong hệ thống nhúng[4]. (Mặc dù ngôn ngữ Rust[5] đang dòm ngó cả hai lĩnh vực đó!)

Nếu bạn đã quen với một ngôn ngữ khác, nhiều thứ trong C sẽ dễ. C đã truyền cảm hứng cho rất nhiều ngôn ngữ khác, và bạn sẽ thấy mảng mảng của nó trong Go, Rust, Swift, Python, JavaScript, Java, và đủ loại ngôn ngữ khác. Những phần đó sẽ quen thuộc.

Thứ duy nhất trong C làm người ta khựng lại là *con trỏ* (pointers). Gần như mọi thứ khác đều quen thuộc, nhưng con trỏ là đứa con lạ. Khái niệm đằng sau con trỏ có lẽ bạn đã biết rồi, nhưng C buộc bạn phải tường minh về nó, bằng các toán tử mà có thể bạn chưa từng thấy bao giờ.

Điều đặc biệt khó chịu là một khi bạn đã *nắm được*[6] con trỏ, nó bỗng nhiên trở nên dễ. Còn trước thời điểm đó, chúng cứ trơn tuột như lươn.

Mọi thứ khác trong C chỉ đơn giản là ghi nhớ một cách khác (đôi khi chính là *cùng* một cách!) để làm một việc bạn đã làm rồi. Con trỏ là phần lạ lẫm. Và, nếu xét kỹ, ngay cả con trỏ cũng chỉ là biến tấu

[1] https://en.wikipedia.org/wiki/Assembly_language
[2] https://en.wikipedia.org/wiki/Bare_machine
[3] https://en.wikipedia.org/wiki/Operating_system
[4] https://en.wikipedia.org/wiki/Embedded_system
[5] https://en.wikipedia.org/wiki/Rust_(programming_language)
[6] https://en.wikipedia.org/wiki/Grok

trên một chủ đề có lẽ bạn đã quen.

Vậy chuẩn bị tinh thần cho một chuyến phiêu lưu náo nhiệt gần nhất với lõi của máy tính mà bạn có thể đến được mà không cần đụng đến assembly, bằng ngôn ngữ có ảnh hưởng nhất mọi thời đại[7]. Giữ chặt!

2.2 Hello, World!

Đây là ví dụ chuẩn mực của một chương trình C. Ai cũng dùng nó. (Lưu ý là các con số ở bên trái chỉ để người đọc tham khảo, chúng không phải là một phần của mã nguồn.)

```
/* Hello world program */

#include <stdio.h>

int main(void)
{
    printf("Hello, World!\n");  // Actually do the work here
}
```

Chúng ta sẽ đeo găng tay cao su loại dày tay áo dài, vớ lấy con dao mổ, và rạch thẳng vào thứ này để xem cái gì làm nó hoạt động. Nào, rửa tay đi, vì ta bắt đầu. Cắt nhẹ thôi...

Ta giải quyết cái dễ trước: mọi thứ nằm giữa hai cặp ký hiệu `/*` và `*/` là chú thích (comment) và sẽ bị trình biên dịch bỏ qua hoàn toàn. Mọi thứ nằm trên một dòng sau `//` cũng vậy. Nó cho phép bạn để lại thông điệp cho chính mình và cho người khác, để khi bạn quay lại đọc code của mình ở tương lai xa, bạn biết cái quái gì mình đang định làm. Tin tôi đi, bạn sẽ quên; chuyện đó xảy ra.

Giờ, cái `#include` này là gì? KINH QUÁ! Được rồi, nó báo cho C Preprocessor kéo nội dung của một file khác và chèn vào code ngay *chỗ đó*.

Khoan, C Preprocessor là cái gì? Câu hỏi hay. Việc biên dịch có hai giai đoạn[8]: preprocessor và compiler. Bất cứ thứ gì bắt đầu bằng dấu pound, dấu thăng, hay "octothorpe", (`#`) là thứ mà preprocessor xử lý trước khi compiler thậm chí còn bắt đầu. Các *chỉ thị preprocessor* (preprocessor directives) thường gặp, như người ta hay gọi, là `#include` và `#define`. Bàn thêm về mấy cái đó sau.

Trước khi đi tiếp, tại sao tôi lại dày công chỉ ra rằng dấu pound được gọi là octothorpe? Câu trả lời đơn giản: tôi thấy từ octothorpe nó buồn cười xuất sắc, nên phải rải bừa cái tên đó ra mọi khi có dịp. Octothorpe. Octothorpe, octothorpe, octothorpe.

Nên *dù sao đi nữa*. Sau khi C preprocessor xử lý xong mọi thứ, kết quả được trao cho compiler để nó sản xuất ra mã assembly[9], mã máy[10], hay bất cứ cái gì nó định làm. Mã máy là "ngôn ngữ" mà CPU hiểu, và nó có thể hiểu *rất nhanh*. Đây là một trong những lý do chương trình C thường chạy nhanh.

Đừng lo về các chi tiết kỹ thuật của quá trình biên dịch lúc này; cứ biết rằng code của bạn chạy qua preprocessor, rồi output của nó chạy qua compiler, rồi cái đó tạo ra một file thực thi để bạn chạy.

Còn phần còn lại của dòng thì sao? Cái `<stdio.h>` là gì? Đó là thứ người ta gọi là *header file*. Chính cái chấm-h ở cuối đã tiết lộ điều đó. Thực ra nó là header file "Standard I/O" (`stdio`) mà bạn sẽ dần dần quen và yêu mến. Nó cho ta quyền truy cập vào một loạt chức năng I/O[11]. Với chương trình demo của ta, ta đang xuất chuỗi "Hello, World!", nên cụ thể là ta cần truy cập đến hàm `printf()` để làm việc đó. File `<stdio.h>` cho ta quyền truy cập đó. Nói cơ bản, nếu ta cố dùng `printf()` mà không có `#include <stdio.h>`, compiler sẽ rền rĩ phàn nàn với ta về chuyện đó.

[7] Tôi biết sẽ có người cãi tôi về điểm này, nhưng chắc ít nhất cũng phải nằm trong top ba, đúng không?

[8] Ờ thì, về mặt kỹ thuật thì nhiều hơn hai, nhưng thôi, ta cứ giả vờ là có hai, càng ít biết càng vui, nhỉ?

[9] https://en.wikipedia.org/wiki/Assembly_language

[10] https://en.wikipedia.org/wiki/Machine_code

[11] Về mặt kỹ thuật, nó chứa các chỉ thị preprocessor và nguyên mẫu hàm (function prototypes, bàn thêm sau) cho các nhu cầu input/output thông dụng.

Sao tôi biết phải `#include <stdio.h>` cho `printf()`? Câu trả lời: nó nằm trong tài liệu. Nếu bạn đang trên hệ Unix, gõ `man 3 printf` và nó sẽ cho bạn biết ngay ở đầu trang man cần những file header nào. Hoặc xem phần tham khảo trong cuốn sách này. `:-)`

Trời đất ơi. Ngần ấy chỉ để cover dòng đầu tiên! Nhưng, nói thẳng ra, nó đã bị mổ xẻ hoàn toàn. Không còn bí ẩn nào sót lại!

Vậy thở một hơi đi... nhìn lại mã mẫu. Chỉ còn vài dòng dễ nữa thôi.

Chào mừng quay lại sau kỳ nghỉ! Tôi biết bạn chẳng nghỉ thực sự đâu; tôi chỉ chiều lòng bạn thôi.

Dòng tiếp theo là `main()`. Đây là định nghĩa của hàm `main()`; mọi thứ giữa cặp dấu ngoặc nhọn ngoằn ngoèo (`{` và `}`) là một phần của định nghĩa hàm.

(Vậy thì *gọi* một hàm khác như thế nào nhỉ? Câu trả lời nằm ở dòng `printf()`, ta sẽ đến đó trong một phút nữa.)

Giờ, hàm main là đặc biệt theo nhiều nghĩa, nhưng có một nghĩa nổi bật hơn cả: nó là hàm sẽ được gọi tự động khi chương trình của bạn bắt đầu chạy. Không có gì của bạn được gọi trước `main()`. Trong ví dụ của ta, điều này ổn vì tất cả những gì ta muốn làm là in một dòng rồi thoát.

À, còn chuyện này: một khi chương trình chạy qua khỏi cuối `main()`, chỗ dấu ngoặc nhọn đóng ở dưới đó, chương trình sẽ thoát, và bạn sẽ trở lại với dấu nhắc dòng lệnh.

Vậy giờ ta biết rằng chương trình đó đã kéo vào một header file, `stdio.h`, và khai báo một hàm `main()` sẽ chạy khi chương trình được khởi động. Bên trong `main()` có những món ngon gì?

Tôi rất vui là bạn đã hỏi. Thật đấy! Ta chỉ có đúng một món ngon thôi: lời gọi đến hàm `printf()`. Bạn có thể nhận ra đây là một lời gọi hàm chứ không phải định nghĩa hàm qua nhiều cách, nhưng một dấu hiệu là không có cặp dấu ngoặc nhọn ngoằn ngoèo đi sau nó. Và bạn kết thúc lời gọi hàm bằng một dấu chấm phẩy để compiler biết đây là điểm kết của biểu thức. Bạn sẽ đặt dấu chấm phẩy sau gần như mọi thứ, bạn sẽ thấy.

Bạn đang truyền một đối số (argument) cho hàm `printf()`: một chuỗi sẽ được in ra khi bạn gọi nó. À đúng rồi, ta đang *gọi* một hàm! Ta giỏi thế! Khoan, khoan, đừng vội tự mãn. Cái `\n` khùng khùng ở cuối chuỗi là gì? Ồ, phần lớn các ký tự trong chuỗi sẽ được in ra đúng như cách chúng được lưu. Nhưng có một số ký tự không thể in trên màn hình một cách đẹp đẽ nên được nhúng dưới dạng mã hai ký tự bắt đầu bằng dấu chéo ngược. Một trong những cái phổ biến nhất là `\n` (đọc là "backslash-N" hoặc đơn giản "newline") tương ứng với ký tự *xuống dòng*. Đây là ký tự làm cho việc in tiếp theo bắt đầu ở đầu dòng tiếp chứ không ở dòng hiện tại. Giống như bạn nhấn return ở cuối dòng.

Vậy chép đoạn code đó vào một file tên là `hello.c` và build nó. Trên nền tảng kiểu Unix (ví dụ Linux, BSD, Mac, hay WSL), từ dòng lệnh bạn sẽ build bằng lệnh kiểu thế này:

```
gcc -o hello hello.c
```

(Có nghĩa là "biên dịch `hello.c`, và xuất ra file thực thi tên là `hello`".)

Sau khi xong, bạn sẽ có một file tên là `hello` mà bạn có thể chạy bằng lệnh này:

```
./hello
```

(Phần `./` ở đầu bảo shell "chạy file từ thư mục hiện tại".)

Và xem thử nó ra cái gì:

```
Hello, World!
```

Xong và đã test! Ship it!

2.3 Chi tiết về biên dịch

Nói thêm một chút về cách build chương trình C, và chuyện gì xảy ra hậu trường.

Giống các ngôn ngữ khác, C có *mã nguồn* (source code). Nhưng, tùy vào ngôn ngữ bạn đến từ đâu, có thể bạn chưa bao giờ phải *biên dịch* mã nguồn của mình thành một *file thực thi* (executable).

Biên dịch là quá trình lấy mã nguồn C của bạn và biến nó thành một chương trình mà hệ điều hành có thể thực thi.

Dân JavaScript và Python không hề quen với một bước biên dịch tách biệt, dù rằng hậu trường nó vẫn đang diễn ra! Python biên dịch mã nguồn của bạn thành thứ gọi là *bytecode* mà máy ảo Python có thể chạy. Dân Java thì quen với việc biên dịch, nhưng cái đó sinh ra bytecode cho Java Virtual Machine.

Khi biên dịch C, *mã máy* được sinh ra. Đây là các số 1 và 0 mà CPU có thể chạy trực tiếp và nhanh chóng.

> Các ngôn ngữ thường không biên dịch được gọi là ngôn ngữ *thông dịch* (interpreted). Nhưng như ta đã nói với Java và Python, chúng cũng có một bước biên dịch. Và không có luật nào nói C không thể được thông dịch. (Ngoài kia có cả interpreter cho C đấy!) Nói ngắn gọn, nó là một mớ ranh giới mờ. Biên dịch nói chung chỉ là việc lấy mã nguồn và biến nó thành một dạng khác, dễ thực thi hơn.

Trình biên dịch C (C compiler) là chương trình làm việc biên dịch đó.

Như đã nói, `gcc` là một trình biên dịch được cài sẵn trên rất nhiều hệ điều hành kiểu Unix[12]. Và thường được chạy từ dòng lệnh trong terminal, nhưng không phải luôn luôn. Bạn cũng có thể chạy nó từ IDE.

Vậy ta build từ dòng lệnh kiểu gì?

2.4 Build với `gcc`

Nếu bạn có một file nguồn tên là `hello.c` trong thư mục hiện tại, bạn có thể build nó thành một chương trình tên là `hello` bằng lệnh gõ trong terminal sau:

```
gcc -o hello hello.c
```

Cờ `-o` có nghĩa là "xuất ra file này"[13]. Và ở cuối là `hello.c`, tên của file ta muốn biên dịch.

Nếu mã nguồn được tách làm nhiều file, bạn có thể biên dịch tất cả cùng nhau (gần như thể chúng là một file, dù các quy tắc thực sự có phần phức tạp hơn) bằng cách đưa tất cả các file `.c` lên dòng lệnh:

```
gcc -o awesomegame ui.c characters.c npc.c items.c
```

và tất cả chúng sẽ được build cùng nhau thành một file thực thi to.

Thế là đủ để bắt đầu, sau này ta sẽ bàn chi tiết về nhiều file nguồn, object files, và đủ thứ vui khác.

2.5 Build với `clang`

Trên máy Mac, trình biên dịch mặc định không phải `gcc`, mà là `clang`. Nhưng cũng có một wrapper được cài sẵn để bạn vẫn chạy `gcc` được.

Bạn cũng có thể cài đúng trình `gcc` qua Homebrew[14] hay cách khác.

[12] https://en.wikipedia.org/wiki/Unix

[13] Nếu bạn không chỉ định tên file xuất, nó sẽ xuất ra một file tên là `a.out` theo mặc định, cái tên này có gốc rễ sâu trong lịch sử Unix.

[14] https://formulae.brew.sh/formula/gcc

2.6 Build từ IDE

Nếu bạn đang dùng *Môi trường phát triển tích hợp* (Integrated Development Environment, IDE), có lẽ bạn không phải build từ dòng lệnh.

Với Visual Studio, `CTRL-F7` sẽ build, và `CTRL-F5` sẽ chạy.

Với VS Code, bạn có thể nhấn `F5` để chạy qua debugger. (Bạn sẽ phải cài C/C++ Extension.)

Với XCode, bạn có thể build bằng `COMMAND-B` và chạy bằng `COMMAND-R`. Để có bộ command line tools, Google "XCode command line tools" và bạn sẽ tìm được hướng dẫn cài đặt.

Để bắt đầu, tôi khuyến khích bạn cũng thử build từ dòng lệnh, nó là lịch sử mà!

2.7 Các phiên bản C

C đã đi một chặng đường dài qua nhiều năm, và nó có nhiều số hiệu phiên bản được đặt tên để chỉ ra phương ngữ của ngôn ngữ mà bạn đang dùng.

Chúng thường tham chiếu đến năm của bản đặc tả.

Nổi tiếng nhất là C89, C99, C11, và C23. Ta sẽ tập trung vào cái cuối cùng trong cuốn sách này.

Nhưng đây là một bảng đầy đủ hơn:

Phiên bản	Mô tả
K&R C	1978, bản gốc. Đặt tên theo Brian Kernighan và Dennis Ritchie. Ritchie thiết kế và viết ngôn ngữ, còn Kernighan đồng tác giả cuốn sách về nó. Ngày nay ít khi thấy code K&R gốc. Nếu có thấy, nó sẽ trông lạ, giống như tiếng Anh Trung cổ trông lạ với người đọc tiếng Anh hiện đại.
C89, ANSI C, C90	Năm 1989, Viện Tiêu chuẩn Quốc gia Hoa Kỳ (ANSI) cho ra một bản đặc tả ngôn ngữ C đặt nền tảng cho C kéo dài đến tận hôm nay. Một năm sau, dây cương được trao cho Tổ chức Tiêu chuẩn hóa Quốc tế (ISO), cho ra C90 giống hệt.
C95	Một bản bổ sung ít được nhắc tới cho C89 có thêm hỗ trợ ký tự rộng.
C99	Đợt đại tu lớn đầu tiên với rất nhiều bổ sung về ngôn ngữ. Thứ mà hầu như ai cũng nhớ là thêm kiểu chú thích `//`. Đây là phiên bản C phổ biến nhất còn được dùng tính đến thời điểm viết cuốn sách này.
C11	Bản cập nhật lớn này gồm hỗ trợ Unicode và đa luồng. Lưu ý rằng nếu bạn bắt đầu dùng các tính năng ngôn ngữ này, có thể bạn đang đánh đổi tính dễ chuyển với những nơi còn mắc kẹt ở C99. Nhưng, nói thật, 1999 cũng đã khá lâu rồi.
C17, C18	Bản cập nhật sửa lỗi cho C11. C17 có vẻ là tên chính thức, nhưng việc xuất bản bị hoãn đến 2018. Theo tôi thấy, hai tên này có thể thay nhau, C17 được ưa chuộng hơn.
C23	Bản đặc tả mới nhất.

Bạn có thể ép GCC dùng một trong các chuẩn này bằng tham số dòng lệnh `-std=`. Nếu muốn nó soi kỹ chuẩn, thêm `-pedantic`.

Ví dụ:

```
gcc -std=c11 -pedantic foo.c
```

Với cuốn sách này, tôi biên dịch chương trình cho C23 với toàn bộ cảnh báo bật lên:

```
gcc -Wall -Wextra -std=c23 -pedantic foo.c
```

Chapter 3

Biến và câu lệnh

> *"It takes all kinds to make a world, does it not, Padre?"*
> *"So it does, my son, so it does."*
>
> —Pirate Captain Thomas Bartholomew Red to the Padre, Pirates

Một chương trình C có thể chứa đủ thứ trên đời.

Ừ đấy.

Và vì nhiều lý do, sẽ dễ cho tất cả chúng ta nếu phân loại một vài thứ hay gặp trong chương trình, để ai nấy đều rõ chúng ta đang nói về cái gì.

3.1 Biến

Người ta hay nói "biến giữ giá trị". Nhưng một cách nghĩ khác là: biến là một cái tên dễ đọc đối với con người, dùng để tham chiếu tới một mẩu dữ liệu nào đó trong bộ nhớ.

Chúng ta sẽ dừng lại một nhịp để he hé nhìn xuống cái hang thỏ mang tên pointer (con trỏ). Đừng lo lắng gì cả.

Bạn có thể hình dung bộ nhớ như một mảng khổng lồ gồm các byte[1]. Dữ liệu được lưu trong "mảng" này[2]. Nếu một số lớn hơn một byte, nó được lưu trong nhiều byte. Vì bộ nhớ giống như một mảng, mỗi byte có thể được tham chiếu qua chỉ số của nó. Chỉ số này vào bộ nhớ còn được gọi là *address* (địa chỉ), *location* (vị trí), hay *pointer* (con trỏ).

Khi bạn có một biến trong C, giá trị của biến đó nằm trong bộ nhớ ở *đâu đó*, tại một địa chỉ nào đó. Dĩ nhiên. Chứ nó còn ở chỗ nào được nữa? Nhưng nhắc tới một giá trị bằng địa chỉ số thì thật khổ sở, nên ta đặt cho nó cái tên, và cái tên đó chính là biến.

Lý do tôi nói hết đống này có hai:

1. Nó sẽ giúp bạn dễ hiểu biến con trỏ sau này, chúng là biến chứa địa chỉ của các biến khác!
2. Và nó cũng giúp bạn dễ hiểu con trỏ sau này.

Tóm lại, biến là một cái tên cho mẩu dữ liệu được lưu trong bộ nhớ ở một địa chỉ nào đó.

3.1.1 Tên biến

Bạn có thể dùng các ký tự trong khoảng 0-9, A-Z, a-z, và dấu gạch dưới cho tên biến, với các luật sau:

- Không được bắt đầu tên biến bằng chữ số 0-9.
- Không được bắt đầu tên biến bằng hai dấu gạch dưới.

[1] Một "byte" thường là một số nhị phân 8 bit. Cứ coi như một số nguyên chỉ có thể chứa giá trị từ 0 đến 255. Về mặt kỹ thuật, C cho phép byte có số bit bất kỳ, và nếu muốn chỉ rõ ràng một số 8 bit, bạn nên dùng từ *octet*. Nhưng lập trình viên mặc nhiên hiểu "byte" là 8 bit trừ khi bạn nói rõ khác đi.

[2] Tôi đang đơn giản hoá cực độ cách bộ nhớ hiện đại hoạt động. Nhưng mô hình tưởng tượng này vẫn dùng được, nên xin thứ lỗi.

- Không được bắt đầu tên biến bằng một dấu gạch dưới rồi tới chữ hoa A-Z.

Với Unicode thì cứ thử xem. Trong spec §D.2 có vài luật nói về dải codepoint Unicode nào được phép ở phần nào của định danh, nhưng viết hết ra thì dài quá, và có lẽ đời bạn cũng chẳng cần nghĩ tới.

3.1.2 Kiểu biến

Tuỳ vào ngôn ngữ bạn đã biết, có thể bạn đã quen với khái niệm type (kiểu), có thể chưa. Nhưng C hơi khó tính ở chỗ này, nên ta nên ôn lại một chút.

Vài kiểu ví dụ, thuộc loại cơ bản nhất:

Kiểu	Ví dụ	Kiểu C
Số nguyên	`3490`	`int`
Số dấu phẩy động	`3.14159`	`float` [3]
Ký tự (đơn)	`'c'`	`char`
Chuỗi	`"Hello, world!"`	`char *` [4]

C cố gắng tự động chuyển đổi giữa hầu hết các kiểu số khi bạn yêu cầu. Ngoài chuyện đó, mọi phép chuyển đổi đều phải làm bằng tay, đặc biệt là giữa chuỗi và số.

Gần như mọi kiểu trong C đều là biến thể của những kiểu trên.

Trước khi dùng một biến, bạn phải *khai báo* (declare) biến đó và cho C biết biến chứa kiểu gì. Một khi đã khai báo, kiểu của biến không thể đổi sau này lúc chạy chương trình. Đặt là gì thì nó là thế cho đến khi rơi ra khỏi scope (phạm vi) và bị vũ trụ hấp thụ lại.

Hãy lấy code "Hello, world" trước đó và thêm vài biến vào:

```
#include <stdio.h>

int main(void)
{
    int i;    // Holds signed integers, e.g. -3, -2, 0, 1, 10
    float f;  // Holds signed floating point numbers, e.g. -3.1416

    printf("Hello, World!\n");  // Ah, blessed familiarity
}
```

Đó! Ta đã khai báo vài biến. Chưa dùng đến, và cả hai đều chưa được khởi tạo. Một biến giữ số nguyên, biến kia giữ số dấu phẩy động (về cơ bản là số thực, nếu bạn có nền toán).

Biến chưa được khởi tạo có giá trị không xác định[5]. Chúng phải được khởi tạo, nếu không bạn phải giả định chúng chứa một số nhảm nào đó.

> Đây là một trong những chỗ C có thể "cắn" bạn. Theo kinh nghiệm của tôi, phần lớn trường hợp cái giá trị không xác định đó là số không... nhưng nó có thể khác nhau giữa các lần chạy! Đừng bao giờ giả định giá trị sẽ là 0, kể cả khi bạn thấy đúng là 0. *Luôn luôn* khởi tạo biến một cách rõ ràng trước khi dùng[a].
>
> [a] Điều này không hoàn toàn đúng 100%. Khi tới phần static storage duration (thời gian lưu trữ tĩnh), bạn sẽ thấy một số biến được tự động khởi tạo về 0. Nhưng cách an toàn là luôn tự khởi tạo.

Khoan, bạn muốn lưu số vào mấy biến đó à? Điên rồ!

[3] Tôi đang nói dối một chút ở đây. Về kỹ thuật `3.14159` là kiểu `double`, nhưng ta chưa tới chỗ đó và tôi muốn bạn gắn `float` với "Floating Point", và C sẽ vui vẻ ép kiểu đó thành `float`. Tóm lại, đừng bận tâm tới khi gặp lại sau.

[4] Đọc là "pointer to a char" hoặc "char pointer" (con trỏ tới char). "Char" là viết tắt của character (ký tự). Dù tôi không tìm thấy nghiên cứu nào, theo quan sát thì đa số đọc là "char", một thiểu số đọc "car", và lác đác đọc "care". Ta sẽ nói kỹ hơn về con trỏ sau.

[5] Nói chung ta bảo chúng có giá trị "ngẫu nhiên", nhưng thật ra không phải số ngẫu nhiên thật, thậm chí cũng chẳng phải ngẫu nhiên giả lập.

Thì cứ làm đi:

```
int main(void)
{
    int i;

    i = 2; // Assign the value 2 into the variable i

    printf("Hello, World!\n");
}
```

Đỉnh. Ta vừa lưu một giá trị. Giờ in nó ra nào.

Ta sẽ in bằng cách truyền *hai* đối số tuyệt vời cho hàm `printf()`. Đối số thứ nhất là một chuỗi mô tả cần in gì và in như thế nào (gọi là *format string*), và đối số thứ hai là giá trị cần in, cụ thể là thứ đang nằm trong biến `i`.

`printf()` quét chuỗi format tìm các chuỗi đặc biệt bắt đầu bằng dấu phần trăm (`%`) để biết phải in gì. Ví dụ, khi gặp `%d`, nó nhìn vào tham số kế tiếp và in ra dưới dạng số nguyên. Gặp `%f` thì in dưới dạng float. Gặp `%s` thì in chuỗi.

Nhờ vậy, ta có thể in ra giá trị của nhiều kiểu khác nhau như này:

```
#include <stdio.h>

int main(void)
{
    int i = 2;
    float f = 3.14;
    char *s = "Hello, world!";  // char * ("char pointer") is the string type

    printf("%s  i = %d and f = %f!\n", s, i, f);
}
```

Và output sẽ là:

```
Hello, world!  i = 2 and f = 3.14!
```

Kiểu này, `printf()` có thể giống với các loại chuỗi format hay chuỗi tham số hoá trong những ngôn ngữ khác mà bạn đã quen.

LƯU Ý: Trong các mục tiếp theo, tôi mặc định bạn đã khai báo các biến từ trước. Nếu ví dụ dùng số nguyên `i` và `j`, cứ giả định đâu đó phía trên ví dụ tôi đã có:

```
int i, j;
```

3.1.3 Kiểu Boolean

C có kiểu Boolean, true hay false?

`1`!

Về lịch sử, C không có kiểu Boolean, và vài người có thể tranh cãi là bây giờ vẫn chưa có.

Trong C, `0` nghĩa là "false", và khác không nghĩa là "true".

Nên `1` là true. `-37` cũng true. Và `0` là false.

Bạn có thể khai báo Boolean như `int`:

```
int x = 1;

if (x) {
    printf("x is true!\n");
}
```

Trong C23, bạn có `bool`, `true`, và `false` thực sự. Trước đó, nếu bạn có phiên bản C đủ mới, có thể `#include <stdbool.h>` để có thứ tương tự.

```
#include <stdio.h>
#include <stdbool.h>  // not needed in C23

int main(void) {
    bool x = true;

    if (x) {
        printf("x is true!\n");
    }
}
```

Về lý thuyết bạn nên gán biến `bool` là `true`, `false`, hoặc kết quả của biểu thức trả ra true/false, nhưng thật ra bạn có thể ép đủ thứ thành `bool`. Có vài luật cụ thể, nhưng đại khái thứ nào giống-số-không thường thành `false`, còn thứ khác-không thì thành true.

Nhưng cẩn thận nếu bạn trộn lẫn, vì giá trị số của `true` là `1`, gần như chắc chắn[6], và nếu bạn trông cậy vào một giá trị dương khác mang nghĩa true, bạn có thể bị lệch. Ví dụ:

```
printf("%d\n", true == 12);  // Prints "0", false!
```

3.2 Toán tử và biểu thức

Các toán tử trong C chắc đã quen thuộc với bạn từ ngôn ngữ khác. Ta lướt nhanh qua một số ở đây.

(Có cả đống chi tiết hơn thế này, nhưng trong mục này ta sẽ làm đủ để bắt đầu thôi.)

3.2.1 Số học

Hy vọng mấy cái này quen thuộc:

```
i = i + 3;  // Addition (+) and assignment (=) operators, add 3 to i
i = i - 8;  // Subtraction, subtract 8 from i
i = i * 9;  // Multiplication
i = i / 2;  // Division
i = i % 5;  // Modulo (division remainder)
```

Có các biến thể viết tắt cho tất cả mấy dòng trên. Mỗi dòng có thể viết ngắn gọn hơn như sau:

```
i += 3;  // Same as "i = i + 3", add 3 to i
i -= 8;  // Same as "i = i - 8"
i *= 9;  // Same as "i = i * 9"
i /= 2;  // Same as "i = i / 2"
i %= 5;  // Same as "i = i % 5"
```

[6] Về kỹ thuật chỉ một bit của `char` được dùng để biểu diễn `bool`, nên nó chỉ có thể là 0 hoặc 1. Chỉ là các bit còn lại (padding) của `char` thì không được quy định rõ. Với `false`, chắc chắn phải toàn 0. Nhưng với `true`, tôi không chắc là nó phải toàn 0 hay không.

Không có toán tử luỹ thừa. Bạn sẽ phải dùng một trong các biến thể của hàm `pow()` trong `math.h`.

Giờ thì nhảy vào mấy thứ lạ hơn mà có thể ngôn ngữ khác của bạn không có!

3.2.2 Toán tử ba ngôi

C cũng có *toán tử ba ngôi* (ternary operator). Đây là một biểu thức mà giá trị của nó phụ thuộc vào kết quả của một điều kiện được nhúng trong biểu thức.

```
// If x > 10, add 17 to y. Otherwise add 37 to y.

y += x > 10? 17: 37;
```

Rối thật! Đọc nhiều sẽ quen. Để giúp một chút, tôi viết lại biểu thức trên bằng câu lệnh `if`:

```
// This expression:

y += x > 10? 17: 37;

// is equivalent to this non-expression:

if (x > 10)
    y += 17;
else
    y += 37;
```

So sánh hai đoạn cho tới khi bạn nhận ra từng thành phần của toán tử ba ngôi.

Hoặc một ví dụ khác, in ra xem số trong `x` là chẵn hay lẻ:

```
printf("The number %d is %s.\n", x, x % 2 == 0? "even": "odd");
```

Format specifier `%s` trong `printf()` nghĩa là in một chuỗi. Nếu biểu thức `x % 2` cho ra `0`, giá trị của toàn bộ biểu thức ba ngôi là chuỗi `"even"`. Ngược lại là chuỗi `"odd"`. Khá ngầu!

Cần lưu ý rằng toán tử ba ngôi không phải flow control (điều khiển luồng) như câu lệnh `if`. Nó chỉ là một biểu thức cho ra một giá trị.

3.2.3 Tăng giảm tiền tố và hậu tố

Giờ nghịch tiếp một thứ mà có lẽ bạn chưa thấy.

Đây là cặp toán tử huyền thoại post-increment và post-decrement:

```
i++;        // Add one to i (post-increment)
i--;        // Subtract one from i (post-decrement)
```

Thường thì chúng được dùng như phiên bản ngắn của:

```
i += 1;        // Add one to i
i -= 1;        // Subtract one from i
```

nhưng tinh ý hơn thì chúng khác một chút, mấy anh bạn ranh mãnh này.

Xem luôn biến thể pre-increment và pre-decrement:

```
++i;        // Add one to i (pre-increment)
--i;        // Subtract one from i (pre-decrement)
```

Với pre-increment và pre-decrement, giá trị của biến được tăng hoặc giảm *trước* khi biểu thức được tính. Sau đó biểu thức được tính với giá trị mới.

Với post-increment và post-decrement, giá trị của biểu thức được tính trước bằng giá trị hiện tại, *rồi sau đó* giá trị mới được tăng hay giảm sau khi giá trị của biểu thức đã được xác định.

Bạn có thể nhúng chúng vào biểu thức như sau:

```
i = 10;
j = 5 + i++;  // Compute 5 + i, _then_ increment i

printf("%d, %d\n", i, j);  // Prints 11, 15
```

So sánh với toán tử pre-increment:

```
i = 10;
j = 5 + ++i;  // Increment i, _then_ compute 5 + i

printf("%d, %d\n", i, j);  // Prints 11, 16
```

Kỹ thuật này được dùng rất thường xuyên khi truy cập và thao tác mảng và con trỏ. Nó cho bạn cách dùng giá trị trong một biến, đồng thời tăng hoặc giảm giá trị đó trước hoặc sau khi dùng.

Nhưng chỗ bạn hay thấy nhất là trong vòng lặp `for`:

```
for (i = 0; i < 10; i++)
    printf("i is %d\n", i);
```

Để sau nói tiếp.

3.2.4 Toán tử dấu phẩy

Đây là một cách ít dùng để ngăn các biểu thức sẽ được chạy từ trái sang phải:

```
x = 10, y = 20;  // First assign 10 to x, then 20 to y
```

Nghe hơi vô nghĩa, vì bạn có thể thay dấu phẩy bằng dấu chấm phẩy đúng không?

```
x = 10; y = 20;  // First assign 10 to x, then 20 to y
```

Nhưng hai cái hơi khác nhau đấy. Cái sau là hai biểu thức riêng biệt, còn cái trước là một biểu thức duy nhất!

Với toán tử dấu phẩy, giá trị của biểu thức dấu phẩy là giá trị của biểu thức ngoài cùng bên phải:

```
x = (1, 2, 3);

printf("x is %d\n", x);  // Prints 3, because 3 is rightmost in the comma list
```

Nhưng ngay cả thế cũng khá gượng gạo. Một chỗ phổ biến hay dùng toán tử dấu phẩy là trong vòng lặp `for` để làm nhiều việc trong từng phần của câu lệnh:

```
for (i = 0, j = 10; i < 100; i++, j++)
    printf("%d, %d\n", i, j);
```

Ta sẽ quay lại phần này sau.

3.2.5 Toán tử điều kiện

Với giá trị Boolean, ta có cả loạt toán tử chuẩn:

```
a == b;  // True if a is equivalent to b
a != b;  // True if a is not equivalent to b
a < b;   // True if a is less than b
a > b;   // True if a is greater than b
a <= b;  // True if a is less than or equal to b
a >= b;  // True if a is greater than or equal to b
```

Đừng lẫn phép gán (`=`) với phép so sánh (`==`)! Hai dấu bằng là so sánh, một dấu bằng là gán.

Ta có thể dùng biểu thức so sánh với câu lệnh `if` :

```
if (a <= 10)
    printf("Success!\n");
```

3.2.6 Toán tử Boolean

Ta có thể nối hoặc biến đổi các biểu thức điều kiện bằng toán tử Boolean cho *and*, *or*, và *not*.

Toán tử	Nghĩa Boolean
`&&`	and
`||`	or
`!`	not

Ví dụ Boolean "and":

```
// Do something if x less than 10 and y greater than 20:

if (x < 10 && y > 20)
    printf("Doing something!\n");
```

Ví dụ Boolean "not":

```
if (!(x < 12))
    printf("x is not less than 12\n");
```

`!` có độ ưu tiên cao hơn các toán tử Boolean khác, nên trong trường hợp này ta phải dùng dấu ngoặc.

Dĩ nhiên, thế thì cũng chỉ tương đương:

```
if (x >= 12)
    printf("x is not less than 12\n");
```

nhưng tôi cần ví dụ mà!

3.2.7 Toán tử `sizeof`

Toán tử này cho bạn biết kích thước (tính bằng byte) mà một biến hoặc một kiểu dữ liệu cụ thể chiếm trong bộ nhớ.

Chính xác hơn, nó cho biết kích thước (tính bằng byte) mà *kiểu của một biểu thức cụ thể* (có thể chỉ là một biến đơn) chiếm trong bộ nhớ.

Con số này có thể khác nhau trên các hệ thống khác nhau, trừ `char` và các biến thể của nó (luôn là 1 byte).

Và có thể hiện tại trông nó chưa hữu ích lắm, nhưng ta sẽ nhắc tới đây đó, nên đáng nói qua.

Vì nó tính số byte cần để lưu một kiểu, bạn có thể nghĩ nó sẽ trả về một `int`. Hoặc... vì kích thước không thể âm, có lẽ trả về `unsigned`?

Hoá ra C có một kiểu đặc biệt cho giá trị trả về từ `sizeof`. Đó là `size_t`, đọc là "*size tee*"[7]. Tất cả những gì ta biết là nó là kiểu số nguyên unsigned có thể chứa kích thước tính bằng byte của bất cứ thứ gì bạn đưa vào `sizeof`.

`size_t` xuất hiện ở rất nhiều nơi khi ta truyền hoặc trả về đếm số lượng. Cứ coi nó như một giá trị đại diện cho một phép đếm.

Bạn có thể lấy `sizeof` của một biến hoặc biểu thức:

```
int a = 999;

// %zu is the format specifier for type size_t
// If your compiler balks at the "z" part, leave it off

printf("%zu\n", sizeof a);      // Prints 4 on my system
printf("%zu\n", sizeof(2 + 7)); // Prints 4 on my system
printf("%zu\n", sizeof 3.14);   // Prints 8 on my system

// If you need to print out negative size_t values, use %zd
```

Nhớ nhé: đó là kích thước tính bằng byte của *kiểu* của biểu thức, chứ không phải kích thước của chính biểu thức. Đó là lý do kích thước của `2+7` bằng với kích thước của `a`, cả hai đều kiểu `int`. Ta sẽ gặp lại con số `4` ở khối code kế tiếp...

...Ở đó bạn sẽ thấy có thể lấy `sizeof` của một kiểu (lưu ý cần dấu ngoặc quanh tên kiểu, khác với biểu thức):

```
printf("%zu\n", sizeof(int));   // Prints 4 on my system
printf("%zu\n", sizeof(char));  // Prints 1 on all systems
```

Một điều quan trọng cần nhớ: `sizeof` là phép toán *thời điểm biên dịch* (compile-time)[8]. Kết quả của biểu thức được xác định toàn bộ lúc biên dịch, chứ không phải lúc chạy.

Ta sẽ tận dụng điều này sau.

3.3 Điều khiển luồng

Boolean thì tốt, nhưng chẳng đi tới đâu nếu ta không điều khiển được luồng chương trình. Hãy nhìn qua một số cấu trúc: `if`, `for`, `while`, và `do-while`.

Trước hết, một ghi chú chung hướng về phía trước, về câu lệnh và khối câu lệnh, gửi tới bạn bởi lập trình viên C thân thiện ở khu phố của bạn:

Sau một thứ như `if` hay `while`, bạn có thể đặt một câu lệnh duy nhất để thực thi, hoặc một khối các câu lệnh thực thi lần lượt theo thứ tự.

Bắt đầu với một câu lệnh đơn:

```
if (x == 10) printf("x is 10\n");
```

Cái này cũng đôi khi được viết trên một dòng riêng. (Whitespace trong C phần lớn không có ý nghĩa, không như Python.)

[7] Chữ `_t` là viết tắt của `type`.

[8] Trừ trường hợp variable length array, nhưng chuyện đó để dịp khác.

```
if (x == 10)
    printf("x is 10\n");
```

Nhưng nếu bạn muốn nhiều thứ xảy ra do điều kiện thì sao? Bạn có thể dùng dấu ngoặc nhọn ngoằn ngoèo để đánh dấu một *block* hay *compound statement* (khối hay câu lệnh ghép).

```
if (x == 10) {
    printf("x is 10\n");
    printf("And also this happens when x is 10\n");
}
```

Có một phong cách khá phổ biến là *luôn luôn* dùng ngoặc nhọn ngay cả khi không cần thiết:

```
if (x == 10) {
    printf("x is 10\n");
}
```

Một số dev thấy code dễ đọc hơn và tránh được lỗi kiểu như ví dụ sau, nhìn qua thì có vẻ cả hai dòng đều nằm trong khối `if`, nhưng thực ra không phải:

```
// BAD ERROR EXAMPLE

if (x == 10)
    printf("This happens if x is 10\n");
    printf("This happens ALWAYS\n");  // Surprise!! Unconditional!
```

`while` và `for` và các cấu trúc lặp khác hoạt động giống như các ví dụ trên. Nếu bạn muốn làm nhiều việc trong vòng lặp hoặc sau `if`, cứ bọc chúng trong ngoặc nhọn.

Nói cách khác, `if` sẽ chạy đúng một thứ ngay sau nó. Và "một thứ" đó có thể là một câu lệnh đơn hoặc một khối các câu lệnh.

3.3.1 Câu lệnh `if` - `else`

Ta đã dùng `if` trong nhiều ví dụ rồi, vì có lẽ bạn đã thấy nó trong ngôn ngữ nào đó, nhưng đây là thêm một ví dụ nữa:

```
int i = 10;

if (i > 10) {
    printf("Yes, i is greater than 10.\n");
    printf("And this will also print if i is greater than 10.\n");
}

if (i <= 10) printf("i is less than or equal to 10.\n");
```

Trong code ví dụ, thông báo sẽ được in nếu `i` lớn hơn 10, còn không thì chạy tiếp xuống dòng kế. Để ý các dấu ngoặc nhọn sau câu lệnh `if`. Nếu điều kiện đúng, hoặc là câu lệnh/biểu thức đầu tiên ngay sau `if` sẽ chạy, hoặc là khối code trong dấu ngoặc nhọn sau `if` sẽ chạy. Hành vi *khối code* (code block) này đúng với mọi câu lệnh.

Dĩ nhiên, vì C cũng vui theo kiểu này, bạn có thể làm gì đó khi điều kiện sai bằng mệnh đề `else`:

```
int i = 99;

if (i == 10)
    printf("i is 10!\n");
```

```
else {
    printf("i is decidedly not 10.\n");
    printf("Which irritates me a little, frankly.\n");
}
```

Và bạn thậm chí có thể xâu chuỗi để kiểm tra nhiều điều kiện khác nhau, như này:

```
int i = 99;

if (i == 10)
    printf("i is 10!\n");

else if (i == 20)
    printf("i is 20!\n");

else if (i == 99) {
    printf("i is 99! My favorite\n");
    printf("I can't tell you how happy I am.\n");
    printf("Really.\n");
}

else
    printf("i is some crazy number I've never heard of.\n");
```

Dù nếu đi hướng đó, nhớ xem câu lệnh `switch` để có giải pháp có khả năng tốt hơn. Cái gượng là `switch` chỉ làm việc với so sánh bằng với hằng số. Cascade `if` - `else` ở trên có thể so sánh bất đẳng, khoảng, biến, hay bất cứ thứ gì bạn dựng được trong biểu thức điều kiện.

3.3.2 Câu lệnh `while`

`while` là cấu trúc lặp bình dân. Làm một việc trong khi biểu thức điều kiện còn đúng.

Làm một cái nào!

```
// Print the following output:
//
//   i is now 0!
//   i is now 1!
//   [ more of the same between 2 and 7 ]
//   i is now 8!
//   i is now 9!

int i = 0;

while (i < 10) {
    printf("i is now %d!\n", i);
    i++;
}

printf("All done!\n");
```

Thế là bạn có một vòng lặp cơ bản. C cũng có `for` có lẽ sẽ gọn hơn cho ví dụ đó.

Một kiểu không hiếm gặp khi dùng `while` là lặp vô hạn, lặp khi điều kiện luôn đúng:

```
while (1) {
    printf("1 is always true, so this repeats forever.\n");
```

```
}
```

3.3.3 Câu lệnh `do-while`

Giờ đã thuần được `while`, hãy ngó qua ông anh họ gần của nó, `do-while`.

Về cơ bản hai thằng giống nhau, chỉ khác là nếu điều kiện sai ngay lần đầu, `do-while` vẫn chạy một lần, còn `while` không chạy lần nào. Nói cách khác, phép kiểm tra xem có thực thi khối hay không xảy ra ở *cuối* khối với `do-while`. Còn với `while` là ở *đầu* khối.

Xem ví dụ:

```
// Using a while statement:

i = 10;

// this is not executed because i is not less than 10:
while(i < 10) {
    printf("while: i is %d\n", i);
    i++;
}

// Using a do-while statement:

i = 10;

// this is executed once, because the loop condition is not checked until
// after the body of the loop runs:

do {
    printf("do-while: i is %d\n", i);
    i++;
} while (i < 10);

printf("All done!\n");
```

Để ý rằng trong cả hai trường hợp, điều kiện lặp là sai ngay từ đầu. Thế nên với `while`, vòng lặp thất bại và khối code sau đó không bao giờ chạy. Còn với `do-while`, điều kiện được kiểm tra *sau khi* khối code chạy, nên nó luôn chạy ít nhất một lần. Ở đây, nó in thông báo, tăng `i`, rồi kiểm tra thấy điều kiện sai, và chạy tiếp tới dòng "All done!".

Bài học rút ra là: nếu bạn muốn vòng lặp chạy ít nhất một lần, bất kể điều kiện lặp thế nào, hãy dùng `do-while`.

Tất cả ví dụ trên có lẽ đã tốt hơn nếu dùng `for`. Làm một cái đỡ tiền định hơn: lặp cho đến khi một số ngẫu nhiên nhất định xuất hiện!

```
#include <stdio.h>   // For printf
#include <stdlib.h>  // For rand

int main(void)
{
    int r;

    do {
        r = rand() % 100; // Get a random number between 0 and 99
        printf("%d\n", r);
```

```
    } while (r != 37);    // Repeat until 37 comes up
}
```

Ghi chú bên lề: bạn có chạy cái đó nhiều lần không? Nếu có, bạn có để ý chính dãy số đó lại xuất hiện. Lại. Và lại? Đó là vì `rand()` là một bộ sinh số giả ngẫu nhiên, nó cần được *seed* (gieo mầm) bằng một số khác nhau để sinh dãy khác nhau. Xem hàm `srand()` [9] để biết chi tiết.

3.3.4 Câu lệnh `for`

Chào mừng đến với một trong những vòng lặp phổ biến nhất thế giới! Vòng lặp `for`!

Đây là vòng lặp tuyệt vời nếu bạn biết trước số lần cần lặp.

Bạn có thể làm việc tương tự chỉ với `while`, nhưng `for` giúp code sạch hơn.

Đây là hai đoạn code tương đương, để ý `for` chỉ là cách biểu diễn gọn hơn:

```
// Print numbers between 0 and 9, inclusive...

// Using a while statement:

i = 0;
while (i < 10) {
    printf("i is %d\n", i);
    i++;
}

// Do the exact same thing with a for-loop:

for (i = 0; i < 10; i++) {
    printf("i is %d\n", i);
}
```

Đúng vậy, thưa các bạn, hai đoạn làm chính xác cùng một việc. Nhưng bạn có thể thấy `for` gọn hơn và dễ nhìn hơn. (Dân JavaScript lúc này sẽ thấy rõ nguồn gốc C của nó.)

Nó được chia thành ba phần, ngăn bởi dấu chấm phẩy. Phần đầu là khởi tạo, phần hai là điều kiện lặp, và phần ba là thứ xảy ra ở cuối khối nếu điều kiện lặp còn đúng. Cả ba phần đều tuỳ chọn.

```
for (initialize things; loop if this is true; do this after each loop)
```

Lưu ý rằng vòng lặp sẽ không chạy dù chỉ một lần nếu điều kiện lặp sai ngay từ đầu.

> **`for`-loop fun fact!**
>
> Bạn có thể dùng toán tử dấu phẩy để làm nhiều việc trong từng vế của `for`!
>
> ```
> for (i = 0, j = 999; i < 10; i++, j--) {
>     printf("%d, %d\n", i, j);
> }
> ```

Một `for` rỗng sẽ chạy mãi mãi:

```
for(;;) {  // "forever"
    printf("I will print this again and again and again\n" );
    printf("for all eternity until the heat-death of the universe.\n");
```

[9] https://beej.us/guide/bgclr/html/split/stdlib.html#man-srand

```
    printf("Or until you hit CTRL-C.\n");
}
```

3.3.5 Câu lệnh `switch`

Tuỳ ngôn ngữ bạn đang dùng, có thể bạn quen hoặc chưa quen với `switch`, hoặc phiên bản C của nó có thể chặt chẽ hơn bạn tưởng. Đây là câu lệnh cho phép bạn thực hiện nhiều hành động khác nhau tuỳ thuộc vào giá trị của một biểu thức số nguyên.

Cơ bản là nó tính biểu thức ra một giá trị số nguyên, rồi nhảy đến `case` tương ứng với giá trị đó. Thực thi tiếp tục từ điểm ấy. Nếu gặp câu lệnh `break`, thực thi nhảy ra khỏi `switch`.

Đây là ví dụ, với một số dê cho trước, ta in ra cảm nhận về số dê đó.

```
#include <stdio.h>

int main(void)
{
    int goat_count = 2;

    switch (goat_count) {
        case 0:
            printf("You have no goats.\n");
            break;

        case 1:
            printf("You have a singular goat.\n");
            break;

        case 2:
            printf("You have a brace of goats.\n");
            break;

        default:
            printf("You have a bona fide plethora of goats!\n");
            break;
    }
}
```

Trong ví dụ đó, `switch` nhảy tới `case 2` và chạy từ đó. Khi (nếu) gặp `break`, nó nhảy ra khỏi `switch`.

Bạn cũng có thể thấy nhãn `default` ở dưới cùng. Cái này chạy khi không `case` nào khớp.

Mọi `case`, kể cả `default`, đều tuỳ chọn. Và chúng có thể xuất hiện theo bất kỳ thứ tự nào, nhưng thông thường `default`, nếu có, được đặt cuối cùng.

Cho nên toàn bộ hoạt động như một cascade `if` - `else`:

```
if (goat_count == 0)
    printf("You have no goats.\n");
else if (goat_count == 1)
    printf("You have a singular goat.\n");
else if (goat_count == 2)
    printf("You have a brace of goats.\n");
else
    printf("You have a bona fide plethora of goats!\n");
```

Với vài điểm khác biệt chủ chốt:

- `switch` thường nhảy tới đoạn code đúng nhanh hơn (dù spec không bảo đảm điều đó).
- `if` - `else` có thể làm các so sánh quan hệ như `<` và `>=`, cộng số dấu phẩy động và các kiểu khác, còn `switch` thì không.

Còn một thứ khá thú vị về switch đôi khi bạn sẽ thấy: *fall through* (rơi xuyên qua).

Nhớ `break` làm ta nhảy ra khỏi switch chứ?

Thế, chuyện gì xảy ra nếu ta *không* `break`?

Hoá ra ta cứ tiếp tục chạy xuống `case` kế tiếp! Demo!

```
switch (x) {
    case 1:
        printf("1\n");
        // Fall through!
    case 2:
        printf("2\n");
        break;
    case 3:
        printf("3\n");
        break;
}
```

Nếu `x == 1`, `switch` này trước hết trúng `case 1`, nó in `1`, nhưng rồi cứ chạy tiếp xuống dòng code kế... in ra `2`!

Rồi, cuối cùng, ta gặp `break` nên nhảy ra khỏi `switch`.

nếu `x == 2`, thì ta chỉ trúng `case 2`, in `2`, và `break` như thường.

Không có `break` được gọi là *fall through*.

ProTip: *LUÔN LUÔN* đặt comment trong code chỗ bạn chủ ý fall through, như tôi đã làm phía trên. Nó sẽ cứu lập trình viên khác khỏi thắc mắc bạn có cố ý làm thế không.

Thực tế, đây là một trong những chỗ phổ biến phát sinh bug trong chương trình C: quên đặt `break` trong `case`. Bạn phải đặt nó nếu không muốn rơi tiếp xuống case kế[10].

Ở trên tôi đã nói `switch` làm việc với kiểu số nguyên, cứ giữ như thế. Đừng dùng số dấu phẩy động hay kiểu chuỗi trong đó. Một kẽ hở ở đây là bạn có thể dùng kiểu ký tự vì ký tự bí mật là số nguyên. Nên đoạn này hoàn toàn chấp nhận được:

```
char c = 'b';

switch (c) {
    case 'a':
        printf("It's 'a'!\n");
        break;

    case 'b':
        printf("It's 'b'!\n");
        break;

    case 'c':
        printf("It's 'c'!\n");
        break;
}
```

[10]Chuyện này được xem là nguy hiểm đến mức những người thiết kế ngôn ngữ Go đã đặt `break` làm mặc định; bạn phải dùng rõ ràng câu lệnh `fallthrough` của Go nếu muốn rơi sang case kế.

Cuối cùng, bạn có thể dùng `enum` trong `switch` vì chúng cũng là kiểu số nguyên. Nhưng để sau, ở chương `enum`.

Chapter 4

Hàm

> *"Sir, not in an environment such as this. That's why I've also been programmed for over thirty secondary functions that—"*
>
> —C3PO, before being rudely interrupted, reporting a now-unimpressive number of additional functions, *Star Wars* script

Rất giống với các ngôn ngữ khác bạn đã quen, C có khái niệm *function* (hàm).

Hàm có thể nhận nhiều loại *argument* (đối số) và trả về một giá trị. Có một điều quan trọng: kiểu của đối số và giá trị trả về phải được khai báo trước, vì C thích thế!

Hãy nhìn một hàm. Đây là hàm nhận một `int` làm đối số, và trả về một `int`.

```
#include <stdio.h>

int plus_one(int n)  // The "definition"
{
    return n + 1;
}
```

Chữ `int` trước `plus_one` chỉ kiểu trả về.

`int n` chỉ rằng hàm nhận một đối số `int`, được lưu trong *parameter* (tham số) `n`. Parameter là một loại biến cục bộ đặc biệt mà đối số được sao chép vào.

Tôi sẽ nhấn mạnh rằng đối số được *sao chép* vào parameter. Nhiều thứ trong C dễ hiểu hơn nếu bạn biết parameter là một *bản sao* của đối số, chứ không phải bản thân đối số. Nói thêm sau một chút.

Đi tiếp xuống `main()`, ta có thể thấy lời gọi hàm, nơi ta gán giá trị trả về vào biến cục bộ `j`:

```
int main(void)
{
    int i = 10, j;

    j = plus_one(i);  // The "call"

    printf("i + 1 is %d\n", j);
}
```

> Trước khi quên, để ý rằng tôi đã định nghĩa hàm trước khi dùng nó. Nếu không làm vậy, trình biên dịch sẽ chưa biết gì về hàm khi biên dịch `main()` và sẽ văng lỗi gọi hàm không xác định. Có cách chuẩn chỉnh hơn để viết đoạn code trên bằng *function prototype* (nguyên mẫu hàm), nhưng sẽ nói sau.

Để ý luôn rằng `main()` cũng là một hàm!

Nó trả về `int`.

Nhưng cái `void` này là gì? Đây là một keyword dùng để nói rằng hàm không nhận đối số nào.

Bạn cũng có thể trả về `void` để nói rằng bạn không trả về giá trị nào:

```
#include <stdio.h>

// This function takes no arguments and returns no value:

void hello(void)
{
    printf("Hello, world!\n");
}

int main(void)
{
    hello();  // Prints "Hello, world!"
}
```

4.1 Truyền theo giá trị

Tôi đã nói trước đó rằng khi bạn truyền một đối số cho hàm, một bản sao của đối số đó được tạo ra và lưu vào parameter tương ứng.

Nếu đối số là một biến, một bản sao của giá trị biến đó được tạo ra và lưu vào parameter.

Tổng quát hơn, toàn bộ biểu thức đối số được tính ra giá trị. Giá trị đó được sao chép vào parameter.

Trong mọi trường hợp, giá trị trong parameter là của riêng nó. Nó độc lập với bất kỳ giá trị hay biến nào bạn dùng làm đối số khi gọi hàm.

Hãy xem một ví dụ. Nghiên cứu và thử đoán output trước khi chạy:

```
#include <stdio.h>

void increment(int a)
{
    a++;
}

int main(void)
{
    int i = 10;

    increment(i);

    printf("i == %d\n", i);  // What does this print?
}
```

Thoạt nhìn, có vẻ `i` là `10`, và ta truyền nó vào hàm `increment()`. Ở đó giá trị được tăng, nên khi in ra phải là `11` đúng không?

> *"Get used to disappointment."*
>
> —Dread Pirate Roberts, *The Princess Bride*

Nhưng không phải `11`, nó in ra `10`! Sao thế?

Mọi chuyện nằm ở việc biểu thức bạn truyền vào hàm được *sao chép* vào parameter tương ứng. Parameter là bản sao, không phải bản gốc.

Vậy `i` là `10` ngoài `main()`. Và ta truyền nó vào `increment()`. Parameter tương ứng có tên là `a` trong hàm đó.

Và phép sao chép xảy ra, như thể là một phép gán. Đại khái, `a = i`. Nên tại thời điểm đó, `a` là `10`. Và ngoài `main()`, `i` cũng là `10`.

Rồi ta tăng `a` lên `11`. Nhưng ta không chạm vào `i` chút nào! Nó vẫn là `10`.

Cuối cùng, hàm kết thúc. Tất cả biến cục bộ bị bỏ đi (chào nhé, `a`!) và ta quay lại `main()`, nơi `i` vẫn là `10`.

Rồi ta in ra, được `10`, và xong.

Đây là lý do trong ví dụ trước với hàm `plus_one()`, ta đã `return` giá trị đã bị sửa cục bộ để có thể thấy nó lại trong `main()`.

Nghe hạn chế nhỉ? Kiểu như bạn chỉ lấy về được một mẩu dữ liệu từ hàm, bạn đang nghĩ vậy đấy. Tuy nhiên, còn một cách khác để lấy dữ liệu về; dân C gọi cách đó là *passing by reference* (truyền theo tham chiếu) và đó là câu chuyện để dành dịp khác.

Nhưng đừng để cái tên hoa lá cành đó đánh lừa bạn khỏi sự thật rằng *MỌI THỨ* bạn truyền vào hàm, *KHÔNG NGOẠI LỆ*, đều được sao chép vào parameter tương ứng, và hàm thao tác trên bản sao cục bộ đó, *BẤT KỂ THẾ NÀO*. Nhớ lấy, ngay cả khi ta đang nói về cái gọi là truyền theo tham chiếu.

4.2 Function Prototype

Nếu bạn còn nhớ từ thời kỳ băng hà vài mục trước, tôi có nói rằng bạn phải định nghĩa hàm trước khi dùng nó, không thì trình biên dịch sẽ chưa biết gì về hàm, và sẽ văng lỗi.

Điều này không hoàn toàn nghiêm ngặt đúng. Bạn có thể báo trước cho trình biên dịch rằng bạn sẽ dùng một hàm có kiểu nhất định với danh sách parameter nhất định. Như thế, hàm có thể được định nghĩa ở đâu cũng được (kể cả ở file khác), miễn là *function prototype* đã được khai báo trước khi bạn gọi hàm đó.

May thay, function prototype thực ra khá dễ. Nó chỉ là bản sao của dòng đầu tiên trong định nghĩa hàm, kèm thêm dấu chấm phẩy ở cuối cho chắc ăn. Ví dụ, đoạn code này gọi một hàm được định nghĩa ở phía sau, vì prototype đã được khai báo trước:

```
#include <stdio.h>

int foo(void);  // This is the prototype!

int main(void)
{
    int i;

    // We can call foo() here before it's definition because the
    // prototype has already been declared, above!

    i = foo();

    printf("%d\n", i);  // 3490
}

int foo(void)  // This is the definition, just like the prototype!
{
    return 3490;
}
```

Nếu bạn không khai báo hàm trước khi dùng (bằng prototype hoặc bằng định nghĩa), bạn đang làm một thứ gọi là *implicit declaration* (khai báo ngầm). Chuyện này được cho phép trong chuẩn C đầu tiên (C89), và chuẩn đó có quy định cho nó, nhưng ngày nay không còn được phép nữa. Và không có lý do chính đáng nào để trông cậy vào nó trong code mới.

Bạn có thể để ý một điều về các đoạn code mẫu ta đã dùng... Đó là ta đã dùng hàm `printf()` cũ kỹ mà tốt lành mà không định nghĩa cũng không khai báo prototype! Làm sao ta thoát được sự vô luật pháp này? Thật ra ta không thoát đâu. Có prototype; nó nằm trong file header `stdio.h` mà ta đã kèm vào bằng `#include`, nhớ không? Nên ta vẫn hợp pháp đó, thưa ông cảnh sát!

4.3 Danh sách parameter rỗng

Bạn có thể thấy cái này đây đó trong code cũ, nhưng không bao giờ nên viết nó trong code mới. Luôn dùng `void` để chỉ rằng hàm không nhận parameter nào. Không bao giờ[1] có lý do để bỏ qua chuyện này trong code hiện đại.

Nếu bạn giỏi nhớ việc bỏ `void` vào cho danh sách parameter rỗng trong hàm và prototype, bạn có thể bỏ qua phần còn lại của mục này.

Có hai bối cảnh cho chuyện này:

- Bỏ hết parameter khi định nghĩa hàm
- Bỏ hết parameter trong prototype

Trước tiên xem định nghĩa hàm tiềm tàng:

```
void foo()  // Should really have a `void` in there
{
    printf("Hello, world!\n");
}
```

Dù spec có nói hành vi trong trường hợp này *như thể* bạn đã ghi `void` (C11 §6.7.6.3¶14), kiểu `void` ở đó có lý do. Hãy dùng nó.

Nhưng trong trường hợp function prototype, có một khác biệt *đáng kể* giữa dùng `void` và không:

```
void foo();
void foo(void);  // Not the same!
```

Bỏ `void` khỏi prototype báo cho trình biên dịch rằng không có thông tin thêm về các parameter của hàm. Nó hiệu quả tắt hết mọi kiểm tra kiểu.

Với prototype, **chắc chắn** dùng `void` khi bạn có danh sách parameter rỗng.

[1] Đừng bao giờ nói "không bao giờ".

Chapter 5

Con trỏ, khép nép mà run!

> *"How do you get to Carnegie Hall?"*
> *"Practice!"*
>
> —20th-century joke of unknown origin

Pointer (con trỏ) là một trong những thứ bị sợ nhất trong ngôn ngữ C. Thật ra, đó chính là thứ khiến ngôn ngữ này có chút thử thách. Nhưng vì sao?

Vì nó, nói thật, có thể tạo ra dòng điện chạy ngược từ bàn phím lên rồi *hàn* tay bạn dính vĩnh viễn tại chỗ, đày bạn cả đời trước bàn phím với ngôn ngữ từ những năm 70!

Thật à? Ừ, không thật đâu. Tôi chỉ đang dựng sẵn bối cảnh để bạn thành công.

Tuỳ ngôn ngữ bạn đến từ đâu, có thể bạn đã hiểu khái niệm *reference* (tham chiếu), nơi một biến tham chiếu tới một đối tượng nào đó.

Chuyện trong C cũng rất giống, chỉ là ta phải nói rõ ràng với C hơn về việc đang nói tới tham chiếu hay nói tới thứ được tham chiếu.

5.1 Bộ nhớ và biến

Bộ nhớ máy tính chứa đủ loại dữ liệu, đúng không? Nó chứa `float`, `int`, và đủ thứ khác. Để bộ nhớ dễ xử lý, mỗi byte trong bộ nhớ được gán một số nguyên để nhận dạng. Các số này tăng dần khi bạn đi lên trong bộ nhớ[1]. Bạn có thể hình dung như một đống hộp được đánh số, mỗi hộp chứa một byte[2] dữ liệu. Hoặc như một mảng lớn mà mỗi phần tử chứa một byte, nếu bạn đến từ ngôn ngữ có mảng. Con số đại diện cho mỗi hộp được gọi là *address* (địa chỉ).

Và không phải kiểu dữ liệu nào cũng chỉ dùng một byte. Ví dụ, `int` thường bốn byte, `float` cũng vậy, nhưng thật ra tuỳ hệ thống. Bạn có thể dùng toán tử `sizeof` để xem một kiểu dùng bao nhiêu byte bộ nhớ.

```
// %zu is the format specifier for type size_t

printf("an int uses %zu bytes of memory\n", sizeof(int));

// That prints "4" for me, but can vary by system.
```

> **Sự thật vui về bộ nhớ**: Khi bạn có một kiểu dữ liệu (như `int` điển hình) dùng nhiều hơn một byte, các byte tạo nên dữ liệu luôn nằm liền kề nhau trong bộ nhớ. Đôi khi chúng theo thứ tự bạn nghĩ, đôi khi không[a]. Dù C không bảo đảm thứ tự bộ nhớ cụ thể (tuỳ nền tảng), vẫn hoàn

[1] Thông thường. Tôi chắc chắn có ngoại lệ đâu đó trong những hành lang tối tăm của lịch sử điện toán.

[2] Byte là một số gồm không quá 8 chữ số nhị phân, gọi tắt là *bit*. Nghĩa là tính theo chữ số thập phân, giống thứ bà của bạn từng dùng, nó có thể chứa một số không dấu từ 0 đến 255, bao gồm cả hai đầu.

> toàn có thể viết code theo hướng không phụ thuộc nền tảng, nơi bạn thậm chí không phải nghĩ tới cái trật tự byte phiền phức đó.
>
> [a]Thứ tự các byte sắp xếp được gọi là *endianness* của số. Các ứng cử viên quen thuộc là *big-endian* (byte quan trọng nhất ở đầu) và *little-endian* (byte quan trọng nhất ở cuối), hoặc, hiếm gặp hơn bây giờ, *mixed-endian* (byte quan trọng nhất ở đâu đó).

Vậy *dù sao thì*, nếu ta có thể đi tiếp và làm một hồi trống cùng chút nhạc dồn dập cho định nghĩa của con trỏ, *con trỏ là một biến chứa địa chỉ.* Hãy tưởng tượng bản nhạc kinh điển của 2001: A Space Odyssey ngay lúc này. Ba bum ba bum ba bum BAAAAH!

Được rồi, có lẽ hơi lên gân nhỉ? Con trỏ không có nhiều bí hiểm lắm đâu. Nó là địa chỉ của dữ liệu. Cũng như biến `int` có thể chứa giá trị `12`, biến con trỏ có thể chứa địa chỉ của dữ liệu.

Nghĩa là các thứ sau đây cùng mang một nghĩa, tức là một con số biểu diễn một điểm trong bộ nhớ:

- Chỉ số vào bộ nhớ (nếu bạn nghĩ bộ nhớ như một mảng lớn)
- Địa chỉ (Address)
- Vị trí (Location)

Tôi sẽ dùng lẫn lộn. Và đúng, tôi đã quăng *location* vào đó, vì không bao giờ là đủ từ đồng nghĩa cả.

Và biến con trỏ giữ con số địa chỉ đó. Cũng như biến `float` có thể chứa `3.14159`.

Hãy tưởng tượng bạn có một xấp giấy nhớ Post-it® được đánh số thứ tự theo địa chỉ. (Cái đầu tiên ở chỉ số `0`, cái kế ở chỉ số `1`, và cứ thế.)

Ngoài con số biểu diễn vị trí, bạn cũng có thể viết lên mỗi tờ một số khác tuỳ thích. Có thể là số chó bạn có. Hoặc số mặt trăng quanh sao Hoả...

...Hoặc, *đó có thể là chỉ số của một tờ Post-it khác!*

Nếu bạn đã viết số chó bạn có, đó chỉ là một biến bình thường. Nhưng nếu bạn viết chỉ số của một tờ Post-it khác, *đó là một con trỏ.* Nó trỏ tới tờ giấy kia!

Một phép tương tự khác có thể là với địa chỉ nhà. Bạn có thể có một căn nhà với những đặc tính nhất định, sân vườn, mái kim loại, tấm pin mặt trời, v.v. Hoặc bạn có thể có địa chỉ của căn nhà đó. Địa chỉ không phải là căn nhà. Một bên là nguyên căn nhà, bên kia chỉ là vài dòng chữ. Nhưng địa chỉ của căn nhà là một *con trỏ* tới căn nhà đó. Nó không phải căn nhà, nhưng nó nói cho bạn biết tìm căn nhà ở đâu.

Và ta có thể làm điều tương tự trong máy tính với dữ liệu. Bạn có thể có một biến dữ liệu chứa giá trị nào đó. Và giá trị đó nằm trong bộ nhớ ở một địa chỉ nào đó. Và bạn có thể có một *biến con trỏ* khác chứa địa chỉ của biến dữ liệu đó.

Nó không phải chính biến dữ liệu, nhưng, giống như địa chỉ nhà, nó nói cho ta biết tìm biến đó ở đâu.

Khi có được điều đó, ta nói ta có một "con trỏ tới" mẩu dữ liệu đó. Và ta có thể đi theo con trỏ để truy cập đến bản thân dữ liệu.

(Tuy đến giờ trông chưa có vẻ đặc biệt hữu ích, tất cả sẽ trở nên không thể thiếu khi dùng với lời gọi hàm. Chịu khó với tôi đến khi tới chỗ đó.)

Giờ nếu ta có một `int`, và ta muốn một con trỏ tới nó, thứ ta muốn là cách nào đó để lấy địa chỉ của `int` đó, đúng không? Xét cho cùng, con trỏ chỉ giữ *địa chỉ của* dữ liệu. Bạn đoán xem ta dùng toán tử nào để tìm *địa chỉ của* `int`?

Ồ, với một bất ngờ kinh hoàng mà chắc hẳn gây sốc tới bạn, người đọc dịu dàng, ta dùng toán tử `address-of` (hoá ra là dấu và: "`&`") để tìm địa chỉ của dữ liệu. Ampersand (dấu và).

Ví dụ nhanh, ta sẽ giới thiệu một *format specifier* mới cho `printf()` để bạn có thể in một con trỏ. Bạn đã biết `%d` in số nguyên thập phân đúng không? Thì `%p` in một con trỏ. Giờ, con trỏ này sẽ trông như một con số rác (và có thể được in ở dạng hexadecimal[3] thay vì thập phân), nhưng nó chỉ là chỉ số vào bộ nhớ nơi dữ liệu được lưu. (Hoặc chỉ số vào bộ nhớ nơi byte đầu tiên của dữ liệu được lưu, nếu dữ liệu gồm nhiều byte.) Trong hầu như mọi tình huống, kể cả trường hợp này, giá trị thực tế của con số được in là không quan trọng với bạn, và tôi đưa ra đây chỉ để minh hoạ toán tử `address-of`.

[3]Tức cơ số 16 với các chữ số 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, và F.

```
#include <stdio.h>

int main(void)
{
    int i = 10;

    printf("The value of i is %d\n", i);
    printf("And its address is %p\n", (void *)&i);
}
```

> **Đoạn code trên có một *cast*** (ép kiểu) nơi ta cưỡng ép kiểu của biểu thức `&i` thành kiểu `void*`. Đây là để ngăn trình biên dịch văng ra cảnh báo. Tất cả thứ này đều là chuyện ta chưa nói tới, nên cứ kệ `(void*)` trong đoạn code trên và giả vờ nó không có ở đó.

Trên máy tính của tôi, đoạn này in ra:

```
The value of i is 10
And its address is 0x7ffddf7072a4
```

Nếu bạn tò mò, con số hexadecimal đó là 140.727.326.896.068 trong thập phân (cơ số 10 y như bà của bạn từng dùng). Đó là chỉ số vào bộ nhớ nơi dữ liệu của biến `i` được lưu. Đó là địa chỉ của `i`. Đó là vị trí của `i`. Đó là con trỏ tới `i`.

> **Khoan, bạn có 140 terabyte RAM à?** Có! Bạn không có à? Nhưng tôi đang bốc phét thôi; dĩ nhiên tôi không có (khoảng 2024). Máy tính hiện đại dùng một công nghệ kỳ diệu gọi là virtual memory[a] (bộ nhớ ảo) khiến các process nghĩ rằng nó có toàn bộ không gian bộ nhớ của máy tính cho riêng nó, bất kể bao nhiêu RAM vật lý thực sự hậu thuẫn. Nên dù địa chỉ là con số khổng lồ đó, nó được hệ thống bộ nhớ ảo của CPU ánh xạ về một địa chỉ bộ nhớ vật lý thấp hơn. Máy tính cụ thể này có 16 GB RAM (lại nữa, khoảng 2024, nhưng tôi đang chạy Linux, nên vậy là dư dả). Terabyte RAM à? Tôi là giáo viên, không phải tỷ phú dot-com. Chẳng có gì trong mấy chuyện này mà ai trong chúng ta cần lo cả, trừ phần tôi không phải giàu đến khủng khiếp.
>
> [a] https://en.wikipedia.org/wiki/Virtual_memory

Nó là con trỏ vì nó cho bạn biết `i` ở đâu trong bộ nhớ. Như địa chỉ nhà viết trên mẩu giấy cho bạn biết có thể tìm căn nhà nào đó ở đâu, con số này chỉ cho ta biết chỗ nào trong bộ nhớ ta có thể tìm thấy giá trị của `i`. Nó trỏ tới `i`.

Lại nữa, thường thì ta không thực sự quan tâm con số địa chỉ chính xác là gì. Ta chỉ quan tâm nó là con trỏ tới `i`.

5.2 Kiểu con trỏ

Được rồi... tất cả thế này ổn thôi. Giờ bạn có thể thành công lấy địa chỉ của biến và in ra màn hình. Có một chút gì đó cho bản CV đấy nhỉ? Đây là lúc bạn tóm cổ tôi và lịch sự hỏi con trỏ rốt cuộc có ích cái quái gì.

Câu hỏi tuyệt vời, và ta sẽ đến đó ngay sau các thông điệp từ nhà tài trợ.

```
ACME ROBOTIC HOUSING UNIT CLEANING SERVICES. YOUR HOMESTEAD WILL BE
DRAMATICALLY IMPROVED OR YOU WILL BE TERMINATED. MESSAGE ENDS.
```

Chào mừng trở lại với một kỳ nữa của Beej's Guide. Lần gặp trước ta đang nói về cách tận dụng con trỏ. Tốt, thứ ta sẽ làm là lưu một con trỏ vào biến để có thể dùng sau. Bạn có thể nhận ra *kiểu con trỏ* vì có dấu sao (`*`) đứng trước tên biến và sau kiểu của nó:

```
int main(void)
{
```

```
    int i;  // i's type is "int"
    int *p; // p's type is "pointer to an int", or "int-pointer"
}
```

Này, vậy ta có một biến kiểu con trỏ, và nó có thể trỏ tới các `int` khác. Tức là, nó có thể chứa địa chỉ của các `int` khác. Ta biết nó trỏ tới `int`, vì kiểu của nó là `int*` (đọc là "int-pointer").

Khi bạn gán vào một biến con trỏ, kiểu của vế phải phép gán phải trùng với kiểu biến con trỏ. May thay, khi bạn lấy `address-of` một biến, kiểu kết quả là con trỏ tới kiểu biến đó, nên các phép gán kiểu sau đây là hoàn hảo:

```
int i;
int *p;  // p is a pointer, but is uninitialized and points to garbage

p = &i;  // p is assigned the address of i--p now "points to" i
```

Bên trái phép gán, ta có một biến kiểu pointer-to-`int` (`int*`), và bên phải là biểu thức kiểu pointer-to-`int` vì `i` là `int` (bởi address-of một `int` cho bạn một con trỏ tới `int`). Địa chỉ của một thứ có thể được lưu trong một con trỏ tới thứ đó.

Hiểu chứ? Tôi biết vẫn chưa hợp lý lắm vì bạn chưa thấy công dụng thực tế của biến con trỏ, nhưng ta đang đi từng bước nhỏ để không ai lạc. Giờ ta sẽ giới thiệu toán tử nghịch-address-of. Nó hơi giống address-of trong Thế giới Bizarro vậy.

5.3 Dereference

Biến con trỏ có thể được xem là *refer* (nhắc tới) một biến khác bằng cách trỏ tới nó. Hiếm khi bạn nghe dân C nói về "refer" hay "references" đâu, nhưng tôi nhắc tới để cái tên của toán tử này có chút ý nghĩa hơn.

Khi bạn có con trỏ tới một biến (đại khái "một reference tới một biến"), bạn có thể dùng biến gốc thông qua con trỏ bằng cách *dereference* con trỏ. (Có thể nghĩ như "de-pointering" con trỏ, nhưng không ai từng nói "de-pointering" cả.)

Trở lại phép so sánh, việc này hơi giống nhìn địa chỉ nhà rồi đi đến căn nhà đó.

Giờ, "truy cập đến biến gốc" nghĩa là gì? Nếu bạn có biến tên `i`, và có con trỏ tới `i` tên `p`, bạn có thể dùng con trỏ `p` đã được dereference *y hệt như chính biến `i` gốc*!

Bạn gần như có đủ kiến thức để xem ví dụ. Mẩu cuối cùng bạn cần biết thật ra là: toán tử dereference là gì? Nó thật ra được gọi là *indirection operator* (toán tử gián tiếp), vì bạn đang truy cập giá trị gián tiếp qua con trỏ. Và nó là dấu sao, lần nữa: `*`. Giờ, đừng nhầm lẫn nó với dấu sao bạn đã dùng lúc khai báo con trỏ, ở trước. Cùng là một ký tự, nhưng có nghĩa khác nhau ở các bối cảnh khác nhau[4].

Đây là một ví dụ đầy đủ:

```
#include <stdio.h>

int main(void)
{
    int i;
    int *p;  // this is NOT a dereference--this is a type "int*"

    p = &i;  // p now points to i, p holds address of i

    i = 10;  // i is now 10
```

[4]Chưa hết! Nó còn dùng trong `/*comments*/` và trong phép nhân, và trong function prototype với variable length array! Tất cả cùng là `*`, nhưng bối cảnh cho nó nghĩa khác nhau.

```
    *p = 20; // the thing p points to (namely i!) is now 20!!

    printf("i is %d\n", i);   // prints "20"
    printf("i is %d\n", *p);  // "20"! dereference-p is the same as i!
}
```

Nhớ rằng `p` giữ địa chỉ của `i`, như bạn thấy ở chỗ ta gán cho `p` ở dòng 8. Cái toán tử indirection làm là bảo máy tính *dùng đối tượng mà con trỏ trỏ tới* thay vì dùng chính con trỏ. Bằng cách này, ta đã biến `*p` thành một dạng bí danh cho `i`.

Tuyệt, nhưng *sao chứ*? Làm tất cả thứ này để làm gì?

5.4 Truyền con trỏ làm đối số

Đến giờ, bạn đang nghĩ là mình có khá nhiều kiến thức về con trỏ nhưng không một chút ứng dụng, đúng chứ? Kiểu, `*p` có ích gì nếu bạn cứ việc viết `i`?

Bạn của tôi ơi, sức mạnh thực sự của con trỏ xuất hiện khi bạn bắt đầu truyền chúng cho hàm. Sao lại quan trọng? Bạn có thể nhớ từ trước rằng có thể truyền đủ loại đối số cho hàm, chúng sẽ được ngoan ngoãn sao chép vào parameter, và rồi bạn có thể thao tác các bản sao cục bộ của biến bên trong hàm, và rồi có thể trả về một giá trị duy nhất.

Lỡ bạn muốn đem về nhiều hơn một mẩu dữ liệu từ hàm thì sao? Ý là bạn chỉ có thể trả về một thứ duy nhất, đúng không? Lỡ tôi trả lời câu hỏi đó bằng một câu hỏi khác? ...Ồ, hai câu hỏi?

Chuyện gì xảy ra khi bạn truyền một con trỏ làm đối số cho hàm? Một bản sao của con trỏ có được đặt vào parameter tương ứng không? *Tất nhiên có luôn.* Nhớ hồi trước tôi huyên thuyên về chuyện *MỌI ĐỐI SỐ* được sao chép vào parameter và hàm dùng một bản sao của đối số không? Cũng như vậy ở đây. Hàm sẽ nhận một bản sao của con trỏ.

Nhưng, và đây là phần thông minh: ta đã dựng sẵn con trỏ để trỏ tới một biến... và rồi hàm có thể dereference bản sao con trỏ của nó để quay lại biến gốc! Hàm không thấy được chính biến đó, nhưng chắc chắn nó có thể dereference một con trỏ tới biến đó!

Việc này tương tự như viết một địa chỉ nhà lên mảnh giấy, rồi chép nó sang một mảnh giấy khác. Giờ bạn có *hai* con trỏ tới căn nhà đó, và cả hai đều dẫn tốt như nhau tới chính căn nhà.

Trong trường hợp lời gọi hàm, một trong các bản sao được lưu trong một biến con trỏ ở scope gọi hàm, bản kia được lưu trong một biến con trỏ là parameter của hàm.

Ví dụ nào! Hãy quay lại hàm `increment()` cũ của ta, nhưng lần này làm sao cho nó thực sự tăng giá trị ở phía người gọi.

```
#include <stdio.h>

void increment(int *p)  // note that it accepts a pointer to an int
{
    *p = *p + 1;        // add one to the thing p points to
}

int main(void)
{
    int i = 10;
    int *j = &i;  // note the address-of; turns it into a pointer to i

    printf("i is %d\n", i);        // prints "10"
    printf("i is also %d\n", *j);  // prints "10"

    increment(j);                  // j is an int*--to i
```

```
    printf("i is %d\n", i);        // prints "11"!
}
```

Được rồi! Có vài thứ cần để ý ở đây... không kém phần quan trọng là hàm `increment()` nhận `int*` làm đối số. Ta truyền cho nó một `int*` trong lời gọi bằng cách đổi biến `int` `i` thành `int*` qua toán tử `address-of`. (Nhớ nhé, con trỏ giữ địa chỉ, nên ta tạo con trỏ tới biến bằng cách cho chúng chạy qua toán tử `address-of`.)

Hàm `increment()` nhận một bản sao của con trỏ. Cả con trỏ gốc `j` (trong `main()`) lẫn bản sao của nó `p` (parameter trong `increment()`) đều trỏ tới cùng một địa chỉ, cụ thể là địa chỉ chứa giá trị `i`. (Lại nữa, theo phép so sánh, như hai mảnh giấy cùng ghi một địa chỉ nhà.) Dereference bất kỳ cái nào cũng cho phép bạn sửa biến gốc `i`! Hàm có thể sửa một biến ở scope khác! Tuyệt vời!

Ví dụ trên thường được viết gọn hơn trong lời gọi bằng cách dùng address-of ngay trong danh sách đối số:

```
printf("i is %d\n", i);  // prints "10"
increment(&i);
printf("i is %d\n", i);  // prints "11"!
```

Quy tắc chung, nếu bạn muốn hàm sửa thứ bạn đang truyền vào sao cho bạn thấy kết quả, bạn sẽ phải truyền một con trỏ tới thứ đó.

5.5 Con trỏ `NULL`

Bất kỳ biến con trỏ kiểu nào cũng có thể được gán một giá trị đặc biệt là `NULL`. Điều này cho biết con trỏ không trỏ tới thứ gì.

```
int *p;

p = NULL;
```

Vì nó không trỏ tới giá trị nào, dereference nó là hành vi không xác định, và có lẽ sẽ dẫn đến crash:

```
int *p = NULL;

*p = 12;  // CRASH or SOMETHING PROBABLY BAD. BEST AVOIDED.
```

Dù bị gọi là lỗi lầm tỷ đô, bởi chính người tạo ra nó[5], con trỏ `NULL` là một sentinel value[6] tốt và là chỉ dấu chung cho thấy một con trỏ chưa được khởi tạo.

(Dĩ nhiên, cũng giống các biến khác, con trỏ trỏ tới rác trừ khi bạn gán rõ ràng cho nó trỏ tới một địa chỉ hoặc `NULL`.)

5.6 Một ghi chú về khai báo con trỏ

Cú pháp khai báo con trỏ có thể hơi kỳ quặc. Hãy xem ví dụ này:

```
int a;
int b;
```

Ta có thể gộp thành một dòng chứ?

[5] https://en.wikipedia.org/wiki/Null_pointer#History
[6] https://en.wikipedia.org/wiki/Sentinel_value

```
int a, b;  // Same thing
```

Nên `a` và `b` đều là `int`. Không vấn đề.

Nhưng còn cái này?

```
int a;
int *p;
```

Có gộp được thành một dòng không? Được. Nhưng `*` đặt ở đâu?

Quy tắc là `*` đứng trước bất kỳ biến nào là kiểu con trỏ. Tức là, `*` *không phải* là một phần của `int` trong ví dụ này, nó là một phần của biến `p`.

Với điều đó trong đầu, ta có thể viết:

```
int a, *p;  // Same thing
```

Quan trọng cần lưu ý rằng dòng sau đây *không* khai báo hai con trỏ:

```
int *p, q;  // p is a pointer to an int; q is just an int.
```

Trông càng gian trá hơn nếu lập trình viên viết dòng (hợp lệ) sau, có chức năng giống hệt dòng trên.

```
int* p, q;  // p is a pointer to an int; q is just an int.
```

Giờ xem cái này và xác định biến nào là con trỏ, biến nào không:

```
int *a, b, c, *d, e, *f, g, h, *i;
```

Tôi sẽ để đáp án ở footnote[7].

5.7 `sizeof` và con trỏ

Chỉ một chút cú pháp ở đây có thể gây bối rối, đôi khi bạn sẽ gặp.

Nhớ rằng `sizeof` hoạt động trên *kiểu* của biểu thức.

```
int *p;

// Prints size of an 'int'
printf("%zu\n", sizeof(int));

// p is type 'int *', so prints size of 'int*'
printf("%zu\n", sizeof p);

// *p is type 'int', so prints size of 'int'
printf("%zu\n", sizeof *p);
```

Bạn có thể gặp code ngoài đời với `sizeof` cuối cùng đó. Chỉ cần nhớ `sizeof` nói về kiểu của biểu thức, chứ không phải về bản thân các biến trong biểu thức.

[7] Các biến kiểu con trỏ là `a`, `d`, `f`, và `i`, vì đó là những biến có `*` đứng trước.

Chapter 6

Mảng

> *"Should array indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration."*
>
> —Stan Kelly-Bootle, computer scientist

May thay, C có mảng. Ý tôi là, tôi biết nó được xem là ngôn ngữ cấp thấp[1] nhưng ít nhất nó có khái niệm mảng được tích hợp sẵn. Và vì khá nhiều ngôn ngữ lấy cảm hứng cú pháp từ C, có lẽ bạn đã quen với việc dùng `[` và `]` để khai báo và dùng mảng.

Nhưng C chỉ *vừa đủ* có mảng thôi! Như ta sẽ thấy sau, mảng chỉ là đường cú pháp (syntactic sugar) trong C, sâu thẳm bên trong chúng là con trỏ và đủ thứ. *Hoảng lên đi!* Nhưng bây giờ, cứ dùng chúng như mảng đã. *Phù.*

6.1 Ví dụ dễ

Xắn tay làm ví dụ luôn:

```
#include <stdio.h>

int main(void)
{
    int i;
    float f[4];  // Declare an array of 4 floats

    f[0] = 3.14159;  // Indexing starts at 0, of course.
    f[1] = 1.41421;
    f[2] = 1.61803;
    f[3] = 2.71828;

    // Print them all out:

    for (i = 0; i < 4; i++) {
        printf("%f\n", f[i]);
    }
}
```

Khi khai báo một mảng, bạn phải cho nó kích thước. Và kích thước phải cố định[2].

Trong ví dụ trên, ta tạo một mảng 4 `float`. Giá trị trong dấu ngoặc vuông ở khai báo cho ta biết điều đó.

[1] Ít ra là dạo này.

[2] Lại nữa, không hẳn, nhưng variable-length array, thứ mà tôi không khoái lắm, là chuyện để dành dịp khác.

Ở các dòng sau, ta truy cập các giá trị trong mảng, gán hoặc đọc, lại bằng dấu ngoặc vuông.

Hy vọng cái này quen thuộc từ các ngôn ngữ bạn đã biết!

6.2 Lấy chiều dài của mảng

Bạn không thể... ở mức độ nào đó. C không ghi lại thông tin này[3]. Bạn phải quản lý riêng trong một biến khác.

Khi tôi nói "không thể", thật ra có một số hoàn cảnh bạn *có thể*. Có một mẹo để lấy số phần tử của mảng trong scope mà mảng được khai báo. Nhưng, nói chung, nó sẽ không hoạt động như bạn mong muốn nếu bạn truyền mảng cho hàm[4].

Xem cái mẹo này. Ý tưởng cơ bản là bạn lấy `sizeof` mảng, rồi chia cho kích thước của mỗi phần tử để ra chiều dài. Ví dụ, nếu một `int` là 4 byte, và mảng dài 32 byte, vậy chắc chắn có chỗ cho $\frac{32}{4}$ hay 8 `int` trong đó.

```
int x[12];  // 12 ints

printf("%zu\n", sizeof x);     // 48 total bytes
printf("%zu\n", sizeof(int));  // 4 bytes per int

printf("%zu\n", sizeof x / sizeof(int));  // 48/4 = 12 ints!
```

Nếu là mảng `char`, thì `sizeof` mảng *chính là* số phần tử, vì `sizeof(char)` được định nghĩa là 1. Với bất cứ thứ gì khác, bạn phải chia cho kích thước của mỗi phần tử.

Nhưng mẹo này chỉ hoạt động trong scope mà mảng được định nghĩa. Nếu bạn truyền mảng cho hàm, nó không hoạt động. Ngay cả khi bạn làm cho nó "to" trong signature của hàm:

```
void foo(int x[12])
{
    printf("%zu\n", sizeof x);     // 8?! What happened to 48?
    printf("%zu\n", sizeof(int));  // 4 bytes per int

    printf("%zu\n", sizeof x / sizeof(int));  // 8/4 = 2 ints?? WRONG.
}
```

Đó là vì khi bạn "truyền" mảng cho hàm, bạn chỉ truyền một con trỏ tới phần tử đầu tiên, và đó là thứ `sizeof` đo. Sẽ nói thêm ở mục Passing Single Dimensional Arrays to Functions phía dưới.

Một thứ nữa bạn có thể làm với `sizeof` và mảng là lấy kích thước của một mảng có số phần tử cố định mà không cần khai báo mảng. Giống như cách bạn lấy kích thước của `int` bằng `sizeof(int)`.

Ví dụ, để xem cần bao nhiêu byte cho một mảng 48 `double`, bạn có thể làm:

```
sizeof(double [48]);
```

6.3 Khởi tạo mảng

Bạn có thể khởi tạo mảng bằng hằng số từ trước:

```
#include <stdio.h>

int main(void)
```

[3] Vì mảng, sâu bên dưới, chỉ là con trỏ tới phần tử đầu tiên, không có thông tin bổ sung nào ghi lại chiều dài.

[4] Vì khi bạn truyền mảng cho hàm, thật ra bạn chỉ đang truyền một con trỏ tới phần tử đầu tiên của mảng, chứ không phải "toàn bộ" mảng.

```
{
    int i;
    int a[5] = {22, 37, 3490, 18, 95};  // Initialize with these values

    for (i = 0; i < 5; i++) {
        printf("%d\n", a[i]);
    }
}
```

Đừng bao giờ để nhiều phần tử trong initializer hơn mức mảng chứa được, không thì trình biên dịch sẽ khó chịu:

```
foo.c: In function ‘main’:
foo.c:6:39: warning: excess elements in array initializer
    6 |     int a[5] = {22, 37, 3490, 18, 95, 999};
      |                                       ^~~
foo.c:6:39: note: (near initialization for ‘a’)
```

Nhưng (vui đây!) bạn có thể để *ít* phần tử trong initializer hơn mức mảng có. Các phần tử còn lại trong mảng sẽ được tự động khởi tạo bằng zero. Điều này đúng với tất cả các dạng khởi tạo mảng: nếu bạn có initializer, bất cứ thứ gì không được gán giá trị rõ ràng sẽ được đặt thành zero.

```
int a[5] = {22, 37, 3490};

// is the same as:

int a[5] = {22, 37, 3490, 0, 0};
```

Có một mẹo tắt phổ biến bạn hay thấy trong initializer khi muốn đặt toàn bộ mảng về zero:

```
int a[100] = {0};
```

Nghĩa là, “Đặt phần tử đầu tiên thành zero, rồi tự động đặt phần còn lại cũng thành zero.”

Bạn cũng có thể đặt phần tử cụ thể của mảng trong initializer, bằng cách chỉ định chỉ số cho giá trị! Khi làm thế, C sẽ vui vẻ tiếp tục khởi tạo các giá trị kế sau cho bạn đến khi initializer cạn, lấp phần còn lại bằng `0`.

Để làm vậy, đặt chỉ số trong ngoặc vuông với `=` theo sau, rồi đặt giá trị.

Đây là ví dụ ta dựng một mảng:

```
int a[10] = {0, 11, 22, [5]=55, 66, 77};
```

Vì ta liệt kê chỉ số 5 là điểm bắt đầu cho `55`, dữ liệu kết quả trong mảng là:

```
0 11 22 0 0 55 66 77 0 0
```

Bạn cũng có thể đặt biểu thức hằng số đơn giản vào đó.

```
#define COUNT 5

int a[COUNT] = {[COUNT-3]=3, 2, 1};
```

cho ra:

```
0 0 3 2 1
```

Cuối cùng, bạn cũng có thể để C tự tính kích thước mảng từ initializer, bằng cách bỏ trống kích thước:

```
int a[3] = {22, 37, 3490};

// is the same as:

int a[] = {22, 37, 3490};  // Left the size off!
```

6.4 Vượt biên!

C không ngăn bạn truy cập mảng vượt biên. Có khi còn không cảnh báo luôn.

Chôm ví dụ phía trên và cứ thế in vượt qua cuối mảng. Nó chỉ có 5 phần tử, nhưng cứ thử in 10 xem chuyện gì xảy ra:

```
#include <stdio.h>

int main(void)
{
    int i;
    int a[5] = {22, 37, 3490, 18, 95};

    for (i = 0; i < 10; i++) {  // BAD NEWS: printing too many elements!
        printf("%d\n", a[i]);
    }
}
```

Chạy trên máy tính của tôi, nó in:

```
22
37
3490
18
95
32765
1847052032
1780534144
-56487472
21890
```

Hú hồn! Cái gì thế? Hoá ra in vượt qua cuối mảng dẫn đến thứ mà dân C gọi là *undefined behavior* (hành vi không xác định). Ta sẽ nói thêm về con quái này sau, nhưng hiện giờ nó nghĩa là, "Bạn đã làm điều gì đó xấu, và bất cứ điều gì cũng có thể xảy ra trong lúc chương trình chạy."

Và "bất cứ điều gì", thường là như tìm thấy zero, tìm thấy số rác, hay crash. Nhưng thật ra spec C nói trong trường hợp này trình biên dịch được phép sinh code làm *bất cứ thứ gì*[5].

Phiên bản ngắn: đừng làm bất cứ thứ gì gây ra undefined behavior. Bao giờ[6].

6.5 Mảng nhiều chiều

Bạn có thể thêm bao nhiêu chiều tuỳ thích cho mảng.

[5]Trong những ngày xưa tốt đẹp của MS-DOS trước khi memory protection ra đời, tôi đang viết một số code C đặc biệt bạo lực, cố tình đâm đầu vào đủ loại undefined behavior. Nhưng tôi biết mình đang làm gì, và mọi thứ vẫn chạy khá tốt. Cho tới khi tôi lỡ bước làm treo máy và, sau khi reboot, phát hiện toàn bộ cài đặt BIOS đã bị xoá sạch. Vui đáo để. (Gửi lời tới @man vì những lúc vui đó.)

[6]Có rất nhiều thứ gây undefined behavior, không chỉ truy cập mảng vượt biên. Đây chính là thứ làm ngôn ngữ C trở nên *sôi động*.

```
int a[10];
int b[2][7];
int c[4][5][6];
```

Chúng được lưu trong bộ nhớ theo thứ tự row-major order[7]. Nghĩa là với mảng 2D, chỉ số đầu tiên được liệt kê chỉ hàng, chỉ số thứ hai chỉ cột.

Bạn cũng có thể dùng initializer trên mảng nhiều chiều bằng cách lồng chúng:

```
#include <stdio.h>

int main(void)
{
    int row, col;

    int a[2][5] = {      // Initialize a 2D array
        {0, 1, 2, 3, 4},
        {5, 6, 7, 8, 9}
    };

    for (row = 0; row < 2; row++) {
        for (col = 0; col < 5; col++) {
            printf("(%d,%d) = %d\n", row, col, a[row][col]);
        }
    }
}
```

Cho output:

```
(0,0) = 0
(0,1) = 1
(0,2) = 2
(0,3) = 3
(0,4) = 4
(1,0) = 5
(1,1) = 6
(1,2) = 7
(1,3) = 8
(1,4) = 9
```

Và bạn có thể khởi tạo với chỉ số rõ ràng:

```
// Make a 3x3 identity matrix

int a[3][3] = {[0][0]=1, [1][1]=1, [2][2]=1};
```

sẽ dựng một mảng 2D như này:

```
1 0 0
0 1 0
0 0 1
```

6.6 Mảng và con trỏ

[*Thản nhiên*] Thì... tôi có lẽ đã đề cập phía trên rằng sâu thẳm bên trong mảng là con trỏ nhỉ? Giờ ta

[7] https://en.wikipedia.org/wiki/Row-_and_column-major_order

nên làm một cú lặn cạn vào chuyện đó, để mọi thứ không bị hoàn toàn rối rắm. Sau này, ta sẽ nhìn kỹ mối quan hệ thực sự giữa mảng và con trỏ, nhưng bây giờ tôi chỉ muốn xem chuyện truyền mảng cho hàm.

6.6.1 Lấy con trỏ tới một mảng

Tôi muốn kể bạn nghe một bí mật. Nói chung, khi một lập trình viên C nói về con trỏ tới một mảng, họ đang nói về con trỏ *tới phần tử đầu tiên* của mảng[8].

Nào, lấy một con trỏ tới phần tử đầu tiên của mảng.

```
#include <stdio.h>

int main(void)
{
    int a[5] = {11, 22, 33, 44, 55};
    int *p;

    p = &a[0];  // p points to the array
                // Well, to the first element, actually

    printf("%d\n", *p);  // Prints "11"
}
```

Chuyện này quá phổ biến trong C, đến nỗi ngôn ngữ cho phép ta một cách viết tắt:

```
p = &a[0];  // p points to the array

// is the same as:

p = a;      // p points to the array, but much nicer-looking!
```

Chỉ cần nhắc đến tên mảng đứng một mình là tương đương với lấy một con trỏ tới phần tử đầu tiên của mảng! Ta sẽ dùng điều này rộng rãi trong các ví dụ sắp tới.

Nhưng khoan đã, `p` chẳng phải là `int*` sao? Và `*p` cho ta `11`, cùng giống `a[0]`? Đúúúúng. Bạn đang bắt đầu thấy thoáng qua mối quan hệ giữa mảng và con trỏ trong C. (Ta sẽ nói nhiều hơn về chuyện này trong chương Pointers II, ở mục Array/Pointer Equivalence.)

6.6.2 Truyền mảng một chiều cho hàm

Làm một ví dụ với mảng một chiều. Tôi sẽ viết vài hàm mà ta có thể truyền mảng vào để làm việc khác nhau.

Chuẩn bị cho vài signature hàm làm đầu óc bùng nổ!

```
#include <stdio.h>

// Passing as a pointer to the first element
void times2(int *a, int len)
{
    for (int i = 0; i < len; i++)
        printf("%d\n", a[i] * 2);
}

// Same thing, but using array notation
void times3(int a[], int len)
```

[8] Về kỹ thuật thì không chính xác, vì con trỏ tới một mảng và con trỏ tới phần tử đầu tiên của mảng có kiểu khác nhau. Nhưng ta sẽ đốt cầu khi đến đó.

```
{
    for (int i = 0; i < len; i++)
        printf("%d\n", a[i] * 3);
}

// Same thing, but using array notation with size
void times4(int a[5], int len)
{
    for (int i = 0; i < len; i++)
        printf("%d\n", a[i] * 4);
}

int main(void)
{
    int x[5] = {11, 22, 33, 44, 55};

    times2(x, 5);
    times3(x, 5);
    times4(x, 5);
}
```

Tất cả các cách liệt kê mảng làm parameter trong hàm đó đều giống hệt nhau.

```
void times2(int *a, int len)
void times3(int a[], int len)
void times4(int a[5], int len)
```

Trong cách dùng của dân C quen tay, cách đầu tiên phổ biến nhất, bỏ xa các cách còn lại.

Và, thật ra, ở tình huống cuối, trình biên dịch thậm chí không quan tâm bạn truyền số nào (miễn lớn hơn không[9]). Nó không ép buộc gì cả.

Giờ sau khi đã nói vậy, kích thước mảng trong khai báo hàm thật ra *có* ý nghĩa khi bạn truyền mảng nhiều chiều vào hàm, nhưng hãy quay lại chuyện đó sau.

6.6.3 Thay đổi mảng trong hàm

Ta đã nói mảng chỉ là con trỏ trá hình. Nghĩa là nếu bạn truyền mảng cho hàm, nhiều khả năng bạn đang truyền con trỏ tới phần tử đầu tiên của mảng.

Nhưng nếu hàm có con trỏ tới dữ liệu, nó có thể thao tác trên dữ liệu đó! Vậy nên những thay đổi mà hàm làm với mảng sẽ thấy được ở phía người gọi.

Đây là ví dụ ta truyền một con trỏ tới mảng cho hàm, hàm thao tác trên các giá trị trong mảng đó, và các thay đổi đó thấy được ở phía người gọi.

```
#include <stdio.h>

void double_array(int *a, int len)
{
    // Multiply each element by 2
    //
    // This doubles the values in x in main() since x and a both point
    // to the same array in memory!
```

[9] C11 §6.7.6.2¶1 yêu cầu nó phải lớn hơn không. Nhưng bạn có thể thấy code ngoài đời với mảng chiều dài bằng không ở cuối các `struct`, và GCC đặc biệt dễ dãi chuyện đó trừ khi bạn biên dịch với `-pedantic`. Mảng chiều dài bằng không là một cơ chế kiểu hack để tạo struct có chiều dài biến đổi. Không may, về kỹ thuật truy cập mảng như vậy là undefined behavior, dù nó gần như chạy được ở mọi nơi. C99 chuẩn hoá một phương án thay thế được định nghĩa rõ gọi là *flexible array members*, ta sẽ tán gẫu chuyện đó sau.

```
    for (int i = 0; i < len; i++)
        a[i] *= 2;
}

int main(void)
{
    int x[5] = {1, 2, 3, 4, 5};

    double_array(x, 5);

    for (int i = 0; i < 5; i++)
        printf("%d\n", x[i]);  // 2, 4, 6, 8, 10!
}
```

Mặc dù ta truyền mảng vào parameter `a` kiểu `int*`, nhìn cách ta truy cập nó bằng ký pháp mảng `a[i]`! Cáááái gì. Chuyện này hoàn toàn được cho phép.

Sau này khi nói về sự tương đương giữa mảng và con trỏ, ta sẽ thấy chuyện này hợp lý hơn nhiều. Giờ, đủ để biết rằng hàm có thể thay đổi mảng mà thấy được ở phía người gọi.

6.6.4 Truyền mảng nhiều chiều cho hàm

Câu chuyện thay đổi một chút khi ta nói về mảng nhiều chiều. C cần biết tất cả các chiều (trừ chiều đầu tiên) để có đủ thông tin biết tìm giá trị ở đâu trong bộ nhớ.

Đây là ví dụ ta nói rõ tất cả các chiều:

```
#include <stdio.h>

void print_2D_array(int a[2][3])
{
    for (int row = 0; row < 2; row++) {
        for (int col = 0; col < 3; col++)
            printf("%d ", a[row][col]);
        printf("\n");
    }
}

int main(void)
{
    int x[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    };

    print_2D_array(x);
}
```

Nhưng trong trường hợp này, hai cách này[10] là tương đương:

```
void print_2D_array(int a[2][3])
void print_2D_array(int a[][3])
```

Trình biên dịch thật ra chỉ cần chiều thứ hai để biết cần nhảy bao xa trong bộ nhớ cho mỗi lần tăng chiều đầu tiên. Nói chung, nó cần biết tất cả các chiều trừ chiều đầu tiên.

[10] Cách này cũng tương đương: `void print_2D_array(int (*a)[3])`, nhưng đó là chuyện tôi không muốn đi sâu ngay lúc này.

Ngoài ra, nhớ rằng trình biên dịch chỉ làm kiểm tra biên tối thiểu lúc biên dịch (nếu bạn may mắn), và C không kiểm tra biên gì hết lúc chạy. Không dây an toàn! Đừng crash bằng cách truy cập phần tử mảng vượt biên!

Chapter 7

Chuỗi

Cuối cùng! Chuỗi! Có gì đơn giản hơn được nữa?

Nhưng hoá ra chuỗi thật ra không phải chuỗi trong C. Đúng vậy đấy! Chúng là con trỏ! Dĩ nhiên rồi!

Giống như mảng, chuỗi trong C *vừa đủ để tồn tại.*

Nhưng cứ xem nào, không phải chuyện ghê gớm lắm đâu.

7.1 String Literal

Trước khi bắt đầu, hãy nói về string literal trong C. Đây là các chuỗi ký tự nằm trong dấu nháy *kép* (`"`). (Nháy đơn bọc quanh ký tự, và đó là con thú hoàn toàn khác.)

Ví dụ:

```
"Hello, world!\n"
"This is a test."
"When asked if this string had quotes in it, she replied, \"It does.\""
```

Cái đầu tiên có newline ở cuối, khá phổ biến.

Cái cuối có dấu nháy kép nhúng bên trong, nhưng bạn thấy mỗi cái được đặt trước (ta nói "được escape bởi") một dấu gạch chéo ngược (`\`) báo cho biết một dấu nháy kép chữ thuộc về chuỗi tại điểm đó. Đây là cách trình biên dịch C phân biệt giữa in dấu nháy kép và dấu nháy kép ở cuối chuỗi.

7.2 Biến chuỗi

Giờ ta đã biết cách làm một string literal, hãy gán nó vào biến để làm gì đó với nó.

```
char *s = "Hello, world!";
```

Để ý kiểu: pointer tới `char` . Biến chuỗi `s` thật ra là con trỏ tới ký tự đầu tiên trong chuỗi đó, chính là `H` .

Và ta có thể in nó bằng format specifier `%s` (viết tắt cho "string"):

```
char *s = "Hello, world!";

printf("%s\n", s);  // "Hello, world!"
```

7.3 Biến chuỗi dưới dạng mảng

Một lựa chọn khác, gần như tương đương với cách dùng `char*` phía trên:

```
char s[14] = "Hello, world!";

// or, if we were properly lazy and have the compiler
// figure the length for us:

char s[] = "Hello, world!";
```

Nghĩa là bạn có thể dùng ký pháp mảng để truy cập các ký tự trong chuỗi. Làm đúng thế để in tất cả ký tự trong chuỗi trên cùng một dòng:

```
#include <stdio.h>

int main(void)
{
    char s[] = "Hello, world!";

    for (int i = 0; i < 13; i++)
        printf("%c", s[i]);
    printf("\n");
}
```

Để ý ta dùng format specifier `%c` để in một ký tự đơn.

Và, xem cái này. Chương trình vẫn chạy tốt nếu ta đổi định nghĩa `s` thành kiểu `char*`:

```
#include <stdio.h>

int main(void)
{
    char *s = "Hello, world!";   // char* here

    for (int i = 0; i < 13; i++)
        printf("%c", s[i]);    // But still use arrays here...?
    printf("\n");
}
```

Và ta vẫn có thể dùng ký pháp mảng để in ra! Bất ngờ nhỉ, nhưng chỉ vì ta chưa nói về sự tương đương giữa mảng và con trỏ. Nhưng đây là thêm một gợi ý nữa rằng sâu thẳm bên trong, mảng và con trỏ là cùng một thứ.

7.4 Khởi tạo chuỗi

Ta đã thấy vài ví dụ khởi tạo biến chuỗi bằng string literal:

```
char *s = "Hello, world!";
char t[] = "Hello, again!";
```

Nhưng hai cách khởi tạo này khác nhau một chút tinh tế. Một string literal, tương tự một integer literal, có bộ nhớ được trình biên dịch tự động quản lý giúp bạn! Với số nguyên, tức là một mẩu dữ liệu kích thước cố định, trình biên dịch khá dễ quản lý. Nhưng chuỗi là con thú số byte thay đổi, được trình biên dịch thuần hoá bằng cách quăng vào một mẩu bộ nhớ và đưa cho bạn một con trỏ tới đó.

Cách viết này trỏ tới bất cứ nơi đâu chuỗi được đặt. Thường thì nơi đó nằm trong vùng đất xa xôi so với phần còn lại của bộ nhớ chương trình, bộ nhớ chỉ đọc, vì lý do liên quan đến hiệu năng và an toàn.

```
char *s = "Hello, world!";
```

Nên nếu bạn thử đột biến chuỗi bằng:

```
char *s = "Hello, world!";

s[0] = 'z';  // BAD NEWS: tried to mutate a string literal!
```

Hành vi là không xác định. Có lẽ, tuỳ hệ thống, sẽ dẫn đến crash.

Nhưng khai báo nó dưới dạng mảng thì khác. Trình biên dịch không cất các byte đó ở phần khác của thành phố, chúng ở ngay cuối đường. Cái này là một *bản sao* có thể đột biến của chuỗi, cái ta có thể đổi tuỳ thích:

```
char t[] = "Hello, again!";  // t is an array copy of the string
t[0] = 'z'; //  No problem

printf("%s\n", t);  // "zello, again!"
```

Nên nhớ nhé: nếu bạn có con trỏ tới một string literal, đừng cố đổi nó! Và nếu bạn dùng chuỗi trong dấu nháy kép để khởi tạo mảng, đó thật ra không phải là string literal.

7.5 Lấy chiều dài chuỗi

Bạn không thể, vì C không theo dõi hộ bạn. Và khi tôi nói "không thể", tôi thật ra muốn nói "có thể"[1]. Có một hàm trong `<string.h>` tên là `strlen()` có thể được dùng để tính chiều dài của bất kỳ chuỗi nào tính bằng byte[2].

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

int main(void)
{
    char *s = "Hello, world!";

    printf("The string is %zu bytes long.\n", strlen(s));
}
```

Hàm `strlen()` trả về kiểu `size_t`, là một kiểu số nguyên nên bạn có thể dùng nó cho phép toán số nguyên. Ta in `size_t` bằng `%zu`.

Chương trình trên in:

```
The string is 13 bytes long.
```

Tuyệt! Vậy là *có thể* lấy chiều dài chuỗi!

Nhưng... nếu C không theo dõi chiều dài chuỗi ở đâu, làm sao nó biết chuỗi dài bao nhiêu?

7.6 Kết thúc chuỗi

C xử lý chuỗi hơi khác nhiều ngôn ngữ lập trình, và thật ra khác với gần như mọi ngôn ngữ lập trình hiện đại.

[1] Dù đúng là C không theo dõi chiều dài chuỗi.

[2] Nếu bạn dùng bộ ký tự cơ bản hoặc một bộ ký tự 8 bit, bạn quen với chuyện một ký tự là một byte. Nhưng điều này không đúng với mọi bộ mã ký tự.

Khi bạn làm một ngôn ngữ mới, về cơ bản bạn có hai lựa chọn để lưu chuỗi trong bộ nhớ:

1. Lưu các byte của chuỗi cùng với một con số chỉ chiều dài chuỗi.
2. Lưu các byte của chuỗi, và đánh dấu điểm kết thúc chuỗi bằng một byte đặc biệt gọi là *terminator* (kết thúc).

Nếu bạn muốn chuỗi dài hơn 255 ký tự, phương án 1 cần ít nhất hai byte để lưu chiều dài. Còn phương án 2 chỉ cần một byte để kết thúc chuỗi. Vậy là tiết kiệm được chút ít.

Dĩ nhiên, ngày nay nghe có vẻ lố bịch khi lo tiết kiệm một byte (hay 3, nhiều ngôn ngữ vui vẻ cho phép bạn có chuỗi dài 4 gigabyte). Nhưng ngày xưa, đó là chuyện lớn hơn.

Nên C chọn phương án #2. Trong C, một "chuỗi" được định nghĩa bởi hai đặc tính cơ bản:

- Một con trỏ tới ký tự đầu tiên trong chuỗi.
- Một byte có giá trị bằng không (hay ký tự `NUL`[3]) nằm đâu đó trong bộ nhớ sau con trỏ, báo hiệu kết thúc chuỗi.

Ký tự `NUL` có thể được viết trong code C là `\0`, dù bạn không hay phải làm thế.

Khi bạn đặt chuỗi trong dấu nháy kép trong code, ký tự `NUL` được tự động, ngầm định, đưa vào.

```
char *s = "Hello!";  // Actually "Hello!\0" behind the scenes
```

Với chuyện này trong đầu, hãy viết hàm `strlen()` của riêng ta, đếm các `char` trong chuỗi cho tới khi gặp `NUL`.

Quy trình là quét dọc chuỗi tìm một ký tự `NUL` duy nhất, đếm khi đi qua[4]:

```
int my_strlen(char *s)
{
    int count = 0;

    while (s[count] != '\0')  // Single quotes for single char
        count++;

    return count;
}
```

Và đó về cơ bản là cách `strlen()` có sẵn làm chuyện này.

7.7 Sao chép một chuỗi

Bạn không thể sao chép chuỗi qua toán tử gán (`=`). Tất cả những gì nó làm là tạo một bản sao của con trỏ tới ký tự đầu tiên... nên bạn kết thúc với hai con trỏ cùng trỏ tới một chuỗi:

```
#include <stdio.h>

int main(void)
{
    char s[] = "Hello, world!";
    char *t;

    // This makes a copy of the pointer, not a copy of the string!
    t = s;

    // We modify t
```

[3] Cái này khác với con trỏ `NULL`, và tôi sẽ viết tắt nó thành `NUL` khi nói về ký tự so với `NULL` cho con trỏ.
[4] Sau này ta sẽ học cách làm gọn hơn với pointer arithmetic.

```
    t[0] = 'z';

    // But printing s shows the modification!
    // Because t and s point to the same string!

    printf("%s\n", s);  // "zello, world!"
}
```

Nếu bạn muốn tạo một bản sao của chuỗi, bạn phải chép từng byte, nghĩa là bạn sẽ lấy từng byte của chuỗi từ một chỗ trong bộ nhớ và nhân đôi chúng ở chỗ khác trong bộ nhớ. Chuyện này được làm dễ hơn nhờ hàm `strcpy()`.[5]

Trước khi sao chép chuỗi, đảm bảo bạn có chỗ để chép vào, tức mảng đích sẽ chứa các ký tự phải dài ít nhất bằng chuỗi bạn đang chép.

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

int main(void)
{
    char s[] = "Hello, world!";
    char t[100];  // Each char is one byte, so plenty of room

    // This makes a copy of the string!
    strcpy(t, s);

    // We modify t
    t[0] = 'z';

    // And s remains unaffected because it's a different string
    printf("%s\n", s);  // "Hello, world!"

    // But t has been changed
    printf("%s\n", t);  // "zello, world!"
}
```

Để ý với `strcpy()`, con trỏ đích là đối số đầu tiên, con trỏ nguồn là đối số thứ hai. Một mẹo nhớ tôi dùng là đó là thứ tự mà bạn sẽ đặt `t` và `s` nếu phép gán `=` chạy được cho chuỗi, với nguồn ở bên phải và đích ở bên trái.

[5] Có một hàm khác tên `strncpy()` hạn chế số byte được sao chép. Có người nói bạn nên luôn dùng `strncpy()` vì bảo vệ chống tràn bộ đệm. Người khác nói bạn không bao giờ nên dùng `strncpy()` vì nó không nhất thiết kết thúc chuỗi của bạn, một cái bẫy tự bắn chân cực kỳ kinh dị khác. Nếu bạn thật sự muốn an toàn, có thể viết phiên bản `strncpy()` của riêng mình luôn luôn kết thúc chuỗi.

Chapter 8

Struct

Trong C, có một thứ gọi là `struct`, một kiểu do người dùng định nghĩa được, giữ nhiều mẩu dữ liệu, có thể thuộc các kiểu khác nhau.

Đây là cách tiện lợi để gói nhiều biến vào làm một. Việc này có ích khi truyền biến cho hàm (chỉ cần truyền một thay vì nhiều), và hữu dụng để tổ chức dữ liệu và làm code dễ đọc hơn.

Nếu bạn đến từ ngôn ngữ khác, có thể bạn đã quen với khái niệm *class* và *object*. Những thứ này không tồn tại sẵn trong C[1]. Bạn có thể nghĩ về `struct` như class chỉ có các trường dữ liệu, không có method.

8.1 Khai báo một struct

Bạn có thể khai báo một `struct` trong code của mình như này:

```
struct car {
    char *name;
    float price;
    int speed;
};
```

Chuyện này thường được làm ở scope toàn cục bên ngoài các hàm để `struct` khả dụng toàn cục.

Khi làm vậy, bạn đang tạo một *kiểu* (type) mới. Tên kiểu đầy đủ là `struct car`. (Không phải chỉ `car`, viết thế không hoạt động.)

Chưa có biến nào kiểu đó, nhưng ta có thể khai báo vài biến:

```
struct car saturn;  // Variable "saturn" of type "struct car"
```

Giờ ta có biến chưa khởi tạo `saturn`[2] kiểu `struct car`.

Ta nên khởi tạo nó! Nhưng làm sao đặt giá trị cho từng trường riêng lẻ?

Giống như nhiều ngôn ngữ khác lấy lại từ C, ta sẽ dùng toán tử chấm (`.`) để truy cập từng trường.

```
saturn.name = "Saturn SL/2";
saturn.price = 15999.99;
saturn.speed = 175;

printf("Name:           %s\n", saturn.name);
```

[1] Mặc dù trong C các mẩu riêng lẻ trong bộ nhớ như `int` được gọi là "object", chúng không phải object theo nghĩa lập trình hướng đối tượng.

[2] Saturn là một hãng xe phổ thông khá được ưa chuộng ở Hoa Kỳ cho tới khi bị đóng cửa vì khủng hoảng 2008, buồn thay với những fan như chúng ta.

```
printf("Price (USD):    %f\n", saturn.price);
printf("Top Speed (km): %d\n", saturn.speed);
```

Ở các dòng đầu, ta đặt giá trị vào `struct car`, rồi ở đoạn sau, in các giá trị đó ra.

8.2 Khởi tạo Struct

Ví dụ ở mục trước hơi cồng kềnh. Chắc phải có cách tốt hơn để khởi tạo biến `struct`!

Bạn có thể làm với initializer bằng cách đặt giá trị cho các trường *theo thứ tự chúng xuất hiện trong* `struct` khi bạn định nghĩa biến. (Cách này không chạy sau khi biến đã được định nghĩa, phải xảy ra ngay lúc định nghĩa).

```
struct car {
    char *name;
    float price;
    int speed;
};

// Now with an initializer! Same field order as in the struct declaration:
struct car saturn = {"Saturn SL/2", 16000.99, 175};

printf("Name:      %s\n", saturn.name);
printf("Price:     %f\n", saturn.price);
printf("Top Speed: %d km\n", saturn.speed);
```

Việc các trường trong initializer phải cùng thứ tự nghe hơi rùng mình. Nếu ai đó đổi thứ tự trong `struct car`, có thể làm hỏng hết code khác!

Ta có thể cụ thể hơn với initializer:

```
struct car saturn = {.speed=175, .name="Saturn SL/2"};
```

Giờ nó độc lập với thứ tự trong khai báo `struct`. Code an toàn hơn hẳn.

Tương tự như initializer của mảng, bất kỳ field designator nào bị bỏ sót đều được khởi tạo về không (trong trường hợp này, đó là `.price`, tôi đã bỏ qua).

8.3 Truyền Struct cho hàm

Bạn có vài cách để truyền `struct` cho hàm.

1. Truyền bản thân `struct`.
2. Truyền con trỏ tới `struct`.

Nhớ rằng khi bạn truyền thứ gì đó cho hàm, một *bản sao* của thứ đó được tạo ra cho hàm thao tác, bất kể đó là bản sao của con trỏ, của `int`, của `struct`, hay bất cứ thứ gì.

Về cơ bản có hai trường hợp bạn muốn truyền con trỏ tới `struct`:

1. Bạn cần hàm có thể thay đổi `struct` được truyền vào, và để những thay đổi đó hiện ra ở phía người gọi.
2. `struct` hơi lớn và chép nó lên stack tốn hơn là chép một con trỏ[3].

Vì hai lý do đó, truyền con trỏ tới `struct` cho hàm phổ biến hơn nhiều, dù truyền cả `struct` thì không hề phạm luật.

Thử truyền con trỏ, làm một hàm cho phép bạn đặt trường `.price` của `struct car`:

[3] Một con trỏ có lẽ 8 byte trên hệ thống 64 bit.

```
#include <stdio.h>

struct car {
    char *name;
    float price;
    int speed;
};

int main(void)
{
    struct car saturn = {.speed=175, .name="Saturn SL/2"};

    // Pass a pointer to this struct car, along with a new,
    // more realistic, price:
    set_price(&saturn, 799.99);

    printf("Price: %f\n", saturn.price);
}
```

Bạn nên có thể nghĩ ra signature của hàm `set_price()` chỉ bằng cách nhìn kiểu của các đối số ta có ở đó.

`saturn` là `struct car`, nên `&saturn` phải là địa chỉ của `struct car`, tức là con trỏ tới `struct car`, cụ thể là `struct car*`.

Và `799.99` là `float`.

Nên khai báo hàm phải như này:

```
void set_price(struct car *c, float new_price)
```

Ta chỉ cần viết phần thân. Lần thử đầu tiên có thể là:

```
void set_price(struct car *c, float new_price) {
    c.price = new_price;  // ERROR!!
}
```

Cách đó không chạy vì toán tử chấm chỉ chạy trên `struct`... nó không chạy trên *con trỏ* tới `struct`.

Được rồi, ta có thể dereference biến `c` để "de-pointer" nó để tới bản thân `struct`. Dereference một `struct car*` cho ra `struct car` mà con trỏ trỏ tới, ta nên có thể dùng toán tử chấm lên đó:

```
void set_price(struct car *c, float new_price) {
    (*c).price = new_price;  // Works, but is ugly and non-idiomatic :(
}
```

Và chạy! Nhưng hơi lôi thôi khi gõ hết đám dấu ngoặc với dấu sao. C có một thứ đường cú pháp gọi là *toán tử mũi tên* (arrow operator) giúp chuyện đó.

8.4 Toán tử mũi tên

Toán tử mũi tên giúp tham chiếu tới các trường trong con trỏ tới `struct`.

```
void set_price(struct car *c, float new_price) {
    // (*c).price = new_price;  // Works, but non-idiomatic :(
    //
    // The line above is 100% equivalent to the one below:
```

```
    c->price = new_price;  // That's the one!
}
```

Vậy khi truy cập các trường, khi nào dùng chấm và khi nào dùng mũi tên?

- Nếu bạn có `struct`, dùng chấm (`.`).
- Nếu bạn có con trỏ tới `struct`, dùng mũi tên (`->`).

8.5 Sao chép và trả về `struct`

Đây là phần dễ cho bạn!

Chỉ cần gán từ cái này sang cái kia!

```
struct car a, b;

b = a;  // Copy the struct
```

Và trả về một `struct` (thay vì trả về một con trỏ tới nó) từ hàm cũng tạo một bản sao tương tự vào biến nhận.

Đây không phải "deep copy"[4]. Tất cả các trường được chép y nguyên, kể cả con trỏ tới các thứ.

8.6 So sánh `struct`

Chỉ có một cách an toàn duy nhất: so sánh từng trường một.

Bạn có thể nghĩ có thể dùng `memcmp()`[5], nhưng nó không xử lý được trường hợp có thể có padding bytes nằm lẫn trong đó.

Nếu bạn xoá `struct` về không trước bằng `memset()`[6], thì nó *có thể* chạy, dù vẫn có thể có những phần tử lạ không so sánh như bạn mong[7].

[4] Một *deep copy* đi theo các con trỏ trong `struct` và chép cả dữ liệu chúng trỏ tới. Một *shallow copy* chỉ chép các con trỏ, chứ không chép thứ chúng trỏ tới. C không có sẵn chức năng deep copy tích hợp nào.

[5] https://beej.us/guide/bgclr/html/split/stringref.html#man-strcmp

[6] https://beej.us/guide/bgclr/html/split/stringref.html#man-memset

[7] https://stackoverflow.com/questions/141720/how-do-you-compare-structs-for-equality-in-c

Chapter 9

File Input/Output

Ta đã thấy vài ví dụ về I/O với `printf()` để làm I/O ở console.

Nhưng ở chương này ta sẽ đẩy các khái niệm đó đi xa hơn một chút.

9.1 Kiểu dữ liệu `FILE*`

Khi làm bất kỳ dạng I/O nào trong C, ta làm thông qua một mẩu dữ liệu dưới dạng kiểu `FILE*`. `FILE*` này giữ mọi thông tin cần để giao tiếp với hệ thống I/O về file bạn đang mở, vị trí hiện tại trong file, v.v.

Đặc tả gọi những thứ này là *stream*, tức một dòng dữ liệu chảy ra từ file hoặc từ bất kỳ nguồn nào. Tôi sẽ dùng lẫn "file" với "stream", nhưng thực sự bạn nên coi "file" là một trường hợp đặc biệt của "stream". Có những cách khác để đẩy dữ liệu vào chương trình ngoài chuyện đọc từ file.

Chút nữa ta sẽ xem cách đi từ một tên file tới một `FILE*` đã mở, nhưng trước hết tôi muốn nhắc đến ba stream đã được mở sẵn và có thể dùng ngay.

Tên `FILE*`	Mô tả
`stdin`	Standard Input (đầu vào chuẩn), mặc định thường là bàn phím
`stdout`	Standard Output (đầu ra chuẩn), mặc định thường là màn hình
`stderr`	Standard Error (lỗi chuẩn), mặc định cũng thường là màn hình

Hoá ra ta đã dùng chúng ngầm suốt rồi. Chẳng hạn, hai lời gọi này là một:

```
printf("Hello, world!\n");
fprintf(stdout, "Hello, world!\n");  // printf to a file
```

Nhưng chuyện đó để sau.

Bạn cũng sẽ nhận ra rằng cả `stdout` và `stderr` đều ra màn hình. Nhìn qua tưởng như sơ suất hay trùng lặp, nhưng thực ra không phải. Các hệ điều hành điển hình cho phép bạn *redirect* (chuyển hướng) đầu ra của bất kỳ cái nào trong hai vào các file khác nhau, và việc tách thông báo lỗi khỏi đầu ra thường có thể rất tiện.

Ví dụ, trong shell POSIX (như sh, ksh, bash, zsh, v.v.) trên hệ thống kiểu Unix, ta có thể chạy chương trình và đẩy đầu ra không-lỗi (`stdout`) vào một file, còn đầu ra lỗi (`stderr`) vào file khác.

```
./foo > output.txt 2> errors.txt   # This command is Unix-specific
```

Vì lý do đó, bạn nên gửi các thông báo lỗi nghiêm trọng ra `stderr` chứ đừng ra `stdout`.

Cách làm sẽ nói sau.

9.2 Đọc file văn bản

Stream được phân loại đại khái theo hai cách: *text* (văn bản) và *binary* (nhị phân).

Stream văn bản được phép dịch dữ liệu khá nhiều, đáng chú ý nhất là dịch các newline sang các biểu diễn khác nhau[1]. File văn bản về mặt logic là một chuỗi *dòng* được ngăn bởi newline. Để portable, dữ liệu đầu vào nên luôn kết thúc bằng một newline.

Nhưng quy tắc chung là nếu bạn có thể chỉnh sửa file đó trong một trình soạn thảo văn bản thông thường thì đó là file văn bản. Ngược lại là nhị phân. Nhị phân thì để sau.

Vậy, vào việc, làm sao mở một file để đọc và kéo dữ liệu ra?

Tạo một file `hello.txt` chỉ chứa mỗi:

```
Hello, world!
```

Rồi viết một chương trình mở file, đọc một ký tự ra, rồi đóng file khi xong. Kế hoạch thế!

```
#include <stdio.h>

int main(void)
{
    FILE *fp;                   // Variable to represent open file

    fp = fopen("hello.txt", "r");  // Open file for reading

    int c = fgetc(fp);          // Read a single character
    printf("%c\n", c);          // Print char to stdout

    fclose(fp);                 // Close the file when done
}
```

Thấy đấy, khi mở file với `fopen()`, nó trả `FILE*` về cho ta để dùng sau.

(Tôi bỏ qua để gọn, nhưng `fopen()` sẽ trả về `NULL` nếu có gì trục trặc, như không tìm thấy file, nên bạn thật sự phải check lỗi cho nó!)

Cũng chú ý chuỗi `"r"` ta truyền vào, nghĩa là "mở một stream văn bản để đọc". (Có nhiều chuỗi khác nhau có thể truyền cho `fopen()` với ý nghĩa khác, như ghi, append, v.v.)

Sau đó ta dùng hàm `fgetc()` để lấy một ký tự từ stream. Có thể bạn thắc mắc sao tôi khai báo `c` là `int` chứ không phải `char`, giữ câu hỏi đó lại nhé!

Cuối cùng ta đóng stream khi xong. Mọi stream đều được đóng tự động khi chương trình thoát, nhưng đóng file thủ công khi không còn dùng nữa là phong cách tốt và cẩn thận.

`FILE*` theo dõi vị trí của ta trong file. Nên các lời gọi `fgetc()` tiếp theo sẽ lấy ký tự kế, rồi ký tự kế nữa, cho tới hết.

Nhưng nghe khổ sở quá. Xem có cách nào dễ hơn không.

9.3 Hết file: `EOF`

Có một ký tự đặc biệt được định nghĩa dạng macro: `EOF`. Đây là thứ `fgetc()` sẽ trả về khi đã tới cuối file và bạn cố đọc thêm một ký tự nữa.

Giờ kể một Fun Fact™. Hoá ra `EOF` là lý do `fgetc()` và các hàm kiểu nó trả về `int` thay vì `char`. `EOF` không phải ký tự đúng nghĩa, và giá trị của nó nhiều khả năng nằm ngoài miền của `char`. Vì

[1] Trước kia ta có ba loại newline dùng rộng rãi: Carriage Return (CR, dùng trên Mac đời cũ), Linefeed (LF, dùng trên hệ Unix), và Carriage Return/Linefeed (CRLF, dùng trên hệ Windows). May mắn là sự xuất hiện của OS X, vốn dựa trên Unix, đã rút con số xuống còn hai.

`fgetc()` phải có khả năng trả về bất kỳ byte nào **và** `EOF`, nó cần một kiểu rộng hơn để chứa nhiều giá trị hơn. Nên là `int`. Nhưng trừ khi bạn đang so sánh giá trị trả về với `EOF`, trong thâm tâm bạn có thể yên tâm rằng đó là một `char`.

Ổn! Quay lại thực tế! Ta có thể dùng cái này để đọc cả file trong một vòng lặp.

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    int c;

    fp = fopen("hello.txt", "r");

    while ((c = fgetc(fp)) != EOF)
        printf("%c", c);

    fclose(fp);
}
```

(Nếu dòng 10 lạ quá, cứ mổ nó ra bắt đầu từ ngoặc lồng trong cùng. Việc đầu tiên là gán kết quả của `fgetc()` vào `c`, *rồi* mới so sánh *cái đó* với `EOF`. Ta vừa nhét tất cả vào một dòng. Đọc có vẻ khó, nhưng nghiền ngẫm đi, đây là C idiomatic đấy.)

Chạy thử, ta thấy:

```
Hello, world!
```

Nhưng vẫn đang xử lý từng ký tự một, mà nhiều file văn bản hợp lý hơn khi nhìn theo dòng. Chuyển sang cái đó.

9.3.1 Đọc từng dòng một

Vậy làm sao lấy nguyên một dòng cùng lúc? `fgets()` giải cứu! Tham số gồm một con trỏ tới buffer `char` để chứa byte, số byte tối đa được đọc, và một `FILE*` để đọc từ đó. Nó trả về `NULL` khi hết file hoặc lỗi. `fgets()` còn tử tế đến mức NUL-terminate chuỗi khi xong[2].

Làm một vòng lặp tương tự trước, nhưng lần này với file nhiều dòng và đọc từng dòng một.

Đây là file `quote.txt`:

```
A wise man can learn more from
a foolish question than a fool
can learn from a wise answer.
                  --Bruce Lee
```

Và đây là đoạn code đọc file đó từng dòng một và in số dòng trước mỗi dòng:

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    char s[1024];  // Big enough for any line this program will encounter
    int linecount = 0;
```

[2]Nếu buffer không đủ lớn để đọc hết một dòng, nó sẽ dừng giữa dòng, và lời gọi `fgets()` kế tiếp sẽ tiếp tục đọc phần còn lại của dòng đó.

```
    fp = fopen("quote.txt", "r");

    while (fgets(s, sizeof s, fp) != NULL)
        printf("%d: %s", ++linecount, s);

    fclose(fp);
}
```

Cho ra:

```
1: A wise man can learn more from
2: a foolish question than a fool
3: can learn from a wise answer.
4:                   --Bruce Lee
```

9.4 Đầu vào có định dạng

Bạn biết cách lấy đầu ra có định dạng bằng `printf()` chứ (và cũng vậy với `fprintf()` ta sẽ thấy ở dưới)?

Bạn có thể làm y như thế với `fscanf()`.

> Trước khi bắt đầu, xin lưu ý rằng dùng các hàm kiểu `scanf()` có thể nguy hiểm với đầu vào không đáng tin. Nếu không chỉ định độ rộng trường cho `%s`, bạn có thể tràn buffer. Tệ hơn, chuyển đổi số không hợp lệ sẽ dẫn tới undefined behavior. Cách an toàn với đầu vào không đáng tin là dùng `%s` kèm độ rộng trường, rồi dùng các hàm như `strtol()` hay `strtod()` để chuyển đổi.

Ta có một file với một dãy bản ghi dữ liệu. Trường hợp này là cá voi, với tên, chiều dài tính bằng mét, và cân nặng tính bằng tấn. `whales.txt`:

```
blue 29.9 173
right 20.7 135
gray 14.9 41
humpback 16.0 30
```

Phải, ta có thể đọc bằng `fgets()` rồi parse chuỗi bằng `sscanf()` (và cách đó chống đỡ tốt hơn với file hỏng), nhưng lần này cứ dùng `fscanf()` kéo thẳng vào.

Hàm `fscanf()` bỏ qua whitespace ở đầu khi đọc, và trả về `EOF` khi hết file hoặc lỗi.

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    char name[1024];  // Big enough for any line this program will encounter
    float length;
    int mass;

    fp = fopen("whales.txt", "r");

    while (fscanf(fp, "%s %f %d", name, &length, &mass) != EOF)
        printf("%s whale, %d tonnes, %.1f meters\n", name, mass, length);

    fclose(fp);
```

```
}
```

Cho ra:

```
blue whale, 173 tonnes, 29.9 meters
right whale, 135 tonnes, 20.7 meters
gray whale, 41 tonnes, 14.9 meters
humpback whale, 30 tonnes, 16.0 meters
```

9.5 Ghi file văn bản

Giống hệt như cách ta dùng `fgetc()`, `fgets()`, và `fscanf()` để đọc stream văn bản, ta có thể dùng `fputc()`, `fputs()`, và `fprintf()` để ghi stream văn bản.

Để làm vậy, ta phải `fopen()` file ở chế độ ghi bằng cách truyền `"w"` làm đối số thứ hai. Mở một file đang tồn tại ở chế độ `"w"` sẽ lập tức cắt file đó về 0 byte để ghi đè hoàn toàn.

Ta sẽ ghép một chương trình đơn giản xuất ra file `output.txt` dùng vài hàm xuất khác nhau.

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    int x = 32;

    fp = fopen("output.txt", "w");

    fputc('B', fp);
    fputc('\n', fp);   // newline
    fprintf(fp, "x = %d\n", x);
    fputs("Hello, world!\n", fp);

    fclose(fp);
}
```

Chương trình này tạo ra file `output.txt` với nội dung:

```
B
x = 32
Hello, world!
```

Fun fact: vì `stdout` là một file, bạn có thể thay dòng 8 bằng:

```
fp = stdout;
```

và chương trình sẽ xuất ra console thay vì ra file. Thử xem!

9.6 I/O file nhị phân

Tới giờ ta mới nói file văn bản. Nhưng còn con thú còn lại nhắc hồi đầu gọi là file *nhị phân* (binary), hay binary stream.

Chúng hoạt động khá giống file văn bản, chỉ khác ở chỗ hệ thống I/O không dịch dữ liệu như có thể sẽ làm với file văn bản. Với file nhị phân, bạn có một dòng byte thô, thế thôi.

Khác biệt lớn khi mở file là phải thêm `"b"` vào mode. Tức là, để đọc file nhị phân, mở ở mode `"rb"`. Để ghi file, mở ở mode `"wb"`.

Vì là dòng byte, và dòng byte có thể chứa ký tự NUL, mà ký tự NUL là dấu kết chuỗi trong C, hiếm khi người ta dùng `fprintf()` và đồng bọn để thao tác file nhị phân.

Thay vào đó hai hàm phổ biến nhất là `fread()` và `fwrite()`. Các hàm này đọc và ghi một số byte chỉ định vào stream.

Demo cho biết, ta sẽ viết mấy chương trình. Một chương trình sẽ ghi một dãy giá trị byte ra đĩa cùng lúc. Chương trình thứ hai sẽ đọc từng byte một và in ra[3].

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    unsigned char bytes[6] = {5, 37, 0, 88, 255, 12};

    fp = fopen("output.bin", "wb");  // wb mode for "write binary"!

    // In the call to fwrite, the arguments are:
    //
    // * Pointer to data to write
    // * Size of each "piece" of data
    // * Count of each "piece" of data
    // * FILE*

    fwrite(bytes, sizeof(char), 6, fp);

    fclose(fp);
}
```

Hai đối số giữa của `fwrite()` trông hơi kỳ. Nhưng đại khái ý ta muốn nói với hàm là: "Ta có các mục lớn *ngần này*, và muốn ghi *bấy nhiêu* mục." Tiện lợi nếu bạn có bản ghi có độ dài cố định, và có một mảng chúng. Bạn chỉ cần bảo kích cỡ một bản ghi và bao nhiêu bản ghi cần ghi.

Trong ví dụ trên, ta bảo kích cỡ mỗi bản ghi là `char`, và có 6 mục.

Chạy chương trình xong ta có file `output.bin`, nhưng mở nó trong trình soạn thảo văn bản thì chẳng thấy gì thân thiện! Đó là dữ liệu nhị phân, không phải văn bản. Và là dữ liệu nhị phân ngẫu nhiên tôi vừa chế ra ấy chứ!

Nếu đẩy qua chương trình hex dump[4], ta có thể thấy đầu ra dưới dạng các byte:

```
05 25 00 58 ff 0c
```

> Nhiều hệ Unix có sẵn chương trình tên `hexdump` để làm việc này. Bạn có thể dùng như này với cờ `-C` ("canonical") để có đầu ra đẹp:
>
> ```
> $ hexdump -C output.bin
> 00000000  05 25 00 58 ff 0c                                 |.%.X..|
> ```
>
> `00000000` là offset trong file mà dòng đầu ra này bắt đầu. `05 25 00 58 ff 0c` là các giá trị byte (và sẽ dài hơn, tới 16 byte mỗi dòng, nếu có nhiều byte hơn trong file). Và bên phải giữa hai ký hiệu pipe (`|`) là nỗ lực hết mình của `hexdump` để in ra các ký tự ứng với những byte đó. Nó

[3] Thông thường chương trình thứ hai sẽ đọc tất cả byte cùng lúc, *rồi* mới in ra trong vòng lặp. Cách đó hiệu quả hơn. Nhưng ở đây cốt là để demo.

[4] https://en.wikipedia.org/wiki/Hex_dump

> in dấu chấm nếu ký tự không in được. Trường hợp này, vì ta chỉ in dữ liệu nhị phân ngẫu nhiên, phần đầu ra đó chỉ là rác. Nhưng nếu ta in một chuỗi ASCII ra file, sẽ thấy nó ở trong đó.

Và các giá trị hex đó khớp với các giá trị (thập phân) ta đã ghi ra.

Giờ thử đọc lại bằng một chương trình khác. Chương trình này sẽ mở file để đọc nhị phân (mode `"rb"`) và đọc từng byte một trong vòng lặp.

`fread()` có đặc điểm hay ở chỗ trả về số byte đã đọc, hoặc `0` khi EOF. Nên ta có thể lặp đến khi thấy thế, in số ra trong lúc chạy.

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    unsigned char c;

    fp = fopen("output.bin", "rb"); // rb for "read binary"!

    while (fread(&c, sizeof(char), 1, fp) > 0)
        printf("%d\n", c);

    fclose(fp);
}
```

Và, chạy nó, ta thấy lại các con số gốc!

```
5
37
0
88
255
12
```

Woo hoo!

9.6.1 Lưu ý về `struct` và số

Như đã thấy ở phần `struct`, compiler được tự do thêm padding vào `struct` theo cách nó thấy hợp. Và các compiler khác nhau có thể làm khác nhau. Cùng một compiler trên kiến trúc khác nhau có thể làm khác. Cùng một compiler trên cùng kiến trúc cũng có thể làm khác.

Ý tôi là: không portable nếu bạn chỉ `fwrite()` nguyên một `struct` ra file khi không biết padding sẽ nằm đâu.

Fix sao đây? Giữ ý đó lại, ta sẽ xem vài cách làm chuyện này sau khi ngó qua một vấn đề liên quan khác.

Số!

Hoá ra không phải mọi kiến trúc đều biểu diễn số trong bộ nhớ theo cùng một cách.

Xem một `fwrite()` đơn giản của một số 2 byte. Ta sẽ viết nó dạng hex để mỗi byte hiện rõ. Byte có giá trị cao nhất sẽ có giá trị `0x12`, byte thấp nhất sẽ có giá trị `0x34`.

```
unsigned short v = 0x1234;  // Two bytes, 0x12 and 0x34

fwrite(&v, sizeof v, 1, fp);
```

Stream sẽ chứa gì?

Tưởng như phải là `0x12` rồi tới `0x34`, đúng không?

Nhưng nếu tôi chạy cái này trên máy mình và hex dump kết quả, tôi thấy:

```
34 12
```

Đảo ngược rồi! Chuyện gì thế?

Chuyện này liên quan tới cái gọi là *endianess*[5] của kiến trúc. Có nơi ghi byte có giá trị cao trước, có nơi ghi byte có giá trị thấp trước.

Điều này nghĩa là nếu bạn ghi thẳng một số nhiều byte ra từ bộ nhớ, bạn không thể làm portable được[6].

Một vấn đề tương tự tồn tại với số dấu phẩy động. Hầu hết hệ thống dùng cùng format cho số floating point, nhưng vài hệ thì không. Không đảm bảo gì hết!

Vậy... làm sao fix hết đống vấn đề này với số và `struct` để ghi dữ liệu ra một cách portable?

Tóm gọn là *serialize* (tuần tự hoá) dữ liệu, thuật ngữ chung mang nghĩa lấy hết dữ liệu và ghi ra theo một định dạng bạn kiểm soát, rõ ràng, và lập trình được để hoạt động giống nhau trên mọi nền tảng.

Như bạn đoán, đây là bài toán đã giải. Có một loạt thư viện serialization sẵn sàng để dùng, chẳng hạn *protocol buffers*[7] của Google. Chúng lo mọi chi tiết vặt cho bạn, và thậm chí cho phép dữ liệu từ chương trình C của bạn tương tác với các ngôn ngữ khác hỗ trợ cùng phương thức serialization.

Làm ơn một điều cho bản thân và mọi người! Serialize dữ liệu nhị phân khi ghi ra stream! Sẽ giữ mọi thứ gọn và portable, kể cả khi bạn chuyển file dữ liệu từ kiến trúc này sang kiến trúc khác.

[5] https://en.wikipedia.org/wiki/Endianess

[6] Và đây là lý do tôi dùng từng byte riêng trong các ví dụ `fwrite()` và `fread()` ở trên, khôn đấy chứ.

[7] https://en.wikipedia.org/wiki/Protocol_buffers

Chapter 10

`typedef`: Tạo kiểu mới

Thực ra không hẳn là tạo kiểu *mới*, mà là đặt tên mới cho kiểu đã có. Thoạt nghe hơi vô nghĩa, nhưng ta có thể dùng nó để làm code gọn gàng hơn hẳn.

10.1 `typedef` về lý thuyết

Đại khái, bạn lấy một kiểu đã có rồi tạo alias (bí danh) cho nó bằng `typedef`.

Như này:

```
typedef int antelope;  // Make "antelope" an alias for "int"

antelope x = 10;       // Type "antelope" is the same as type "int"
```

Bạn có thể làm thế với bất kỳ kiểu có sẵn nào. Còn có thể tạo nhiều kiểu bằng list ngăn bởi dấu phẩy:

```
typedef int antelope, bagel, mushroom;  // These are all "int"
```

Tiện ghê, nhỉ? Đánh máy được `mushroom` thay cho `int`? Chắc bạn đang *phấn khích lắm* với tính năng này!

Được rồi Giáo sư Chế Giễu, chút nữa sẽ tới những cách dùng phổ biến hơn.

10.1.1 Scoping

`typedef` tuân theo quy tắc scope thông thường.

Vì vậy, rất hay gặp `typedef` ở file scope ("global") để mọi hàm đều dùng được kiểu mới thoải mái.

10.2 `typedef` trong thực tế

Đổi tên `int` thành cái khác thì cũng không thú vị lắm. Xem `typedef` thường xuất hiện ở đâu.

10.2.1 `typedef` và `struct`

Đôi khi một `struct` sẽ được `typedef` ra tên mới để khỏi phải gõ từ `struct` đi `struct` lại.

```
struct animal {
    char *name;
    int leg_count, speed;
};
```

```
//  original name      new name
//            |         |
//            v         v
//      |-----------| |----|
typedef struct animal animal;

struct animal y;  // This works
animal z;         // This also works because "animal" is an alias
```

Cá nhân tôi không thích kiểu này lắm. Tôi thích sự rõ ràng mà code có được khi thêm chữ `struct` vào trước kiểu, lập trình viên biết ngay mình đang xử lý cái gì. Nhưng vì nó phổ biến nên tôi đưa vào đây.

Giờ tôi muốn chạy lại đúng ví dụ đó theo cách bạn hay thấy. Ta sẽ đặt `struct animal` *vào bên trong* `typedef`. Có thể gộp hết vào như này:

```
//  original name
//            |
//            v
//      |-----------|
typedef struct animal {
    char *name;
    int leg_count, speed;
} animal;                         // <-- new name

struct animal y;  // This works
animal z;         // This also works because "animal" is an alias
```

Y chang ví dụ trước, chỉ ngắn gọn hơn.

Nhưng chưa hết! Còn một kiểu rút gọn phổ biến nữa bạn có thể thấy trong code, dùng cái gọi là *anonymous structure* (cấu trúc vô danh)[1]. Hoá ra ở nhiều chỗ bạn không cần đặt tên cho structure, và dùng với `typedef` là một trong những chỗ đó.

Làm lại ví dụ với anonymous structure:

```
//  Anonymous struct! It has no name!
//         |
//         v
//      |----|
typedef struct {
    char *name;
    int leg_count, speed;
} animal;                         // <-- new name

//struct animal y;  // ERROR: this no longer works--no such struct!
animal z;           // This works because "animal" is an alias
```

Một ví dụ khác, có thể gặp thứ kiểu như:

```
typedef struct {
    int x, y;
} point;

point p = {.x=20, .y=40};

printf("%d, %d\n", p.x, p.y);  // 20, 40
```

[1] Ta sẽ nói thêm về chúng sau.

10.2.2 `typedef` và các kiểu khác

Không phải dùng `typedef` với một kiểu đơn giản như `int` là hoàn toàn vô ích, nó giúp bạn trừu tượng hoá các kiểu để dễ đổi sau này.

Ví dụ, nếu `float` rải khắp code của bạn ở 100 tỉ chỗ, sẽ rất đau đầu nếu đổi hết sang `double` khi sau này vì lý do nào đó bạn phải làm thế.

Nhưng nếu bạn chuẩn bị một chút:

```
typedef float app_float;

// and

app_float f1, f2, f3;
```

Thì sau này muốn đổi sang kiểu khác, chẳng hạn `long double`, bạn chỉ cần đổi `typedef`:

```
//        voila!
//      |---------|
typedef long double app_float;

// and no need to change this line:

app_float f1, f2, f3;  // Now these are all long doubles
```

10.2.3 `typedef` và con trỏ

Bạn có thể tạo một kiểu là con trỏ.

```
typedef int *intptr;

int a = 10;
intptr x = &a;  // "intptr" is type "int*"
```

Tôi rất không thích kiểu này. Nó giấu đi chuyện `x` là kiểu con trỏ vì bạn không thấy dấu `*` nào trong khai báo.

IMHO, tốt hơn là thể hiện rõ bạn đang khai báo kiểu con trỏ để các dev khác nhìn thấy rõ và không nhầm `x` là kiểu không phải con trỏ.

Nhưng lần đếm gần nhất thì có chừng 832.007 người không đồng ý với tôi.

10.2.4 `typedef` và cách viết hoa

Tôi đã thấy đủ loại cách viết hoa cho `typedef`.

```
typedef struct {
    int x, y;
} my_point;          // lower snake case

typedef struct {
    int x, y;
} MyPoint;          // CamelCase

typedef struct {
    int x, y;
} Mypoint;          // Leading uppercase
```

```
typedef struct {
    int x, y;
} MY_POINT;          // UPPER SNAKE CASE
```

Đặc tả C11 không ép theo cách nào, và có ví dụ cả viết hoa hết lẫn viết thường hết.

K&R2 chủ yếu dùng leading uppercase, nhưng cũng có vài ví dụ viết hoa hết và snake case (với hậu tố `_t`).

Nếu bạn đang có style guide, theo nó. Nếu chưa, vớ lấy một cái rồi theo.

10.3 Mảng và `typedef`

Cú pháp hơi kỳ, và theo kinh nghiệm của tôi thì hiếm gặp, nhưng bạn có thể `typedef` một mảng với số phần tử xác định.

```
// Make type five_ints an array of 5 ints
typedef int five_ints[5];

five_ints x = {11, 22, 33, 44, 55};
```

Tôi không thích vì nó giấu đi bản chất mảng của biến, nhưng làm được.

Chapter 11

Pointers II: Số học con trỏ

Đến lúc lao vào sâu hơn với một loạt chủ đề mới về con trỏ! Nếu bạn chưa nắm vững con trỏ, xem lại mục đầu trong sách về chủ đề này.

11.1 Số học con trỏ

Hoá ra bạn có thể làm toán trên con trỏ, cụ thể là cộng và trừ.

Nhưng như thế có nghĩa là gì?

Ngắn gọn, nếu bạn có con trỏ tới một kiểu, cộng 1 vào con trỏ sẽ chuyển nó tới item kế tiếp của kiểu đó nằm ngay sau trong bộ nhớ.

Điều **quan trọng** cần nhớ là khi di chuyển con trỏ và nhìn vào các chỗ khác nhau trong bộ nhớ, ta phải đảm bảo con trỏ luôn trỏ đến một chỗ hợp lệ trước khi dereference. Nếu đang lang thang đâu đó ngoài đồng cỏ và cố xem ở đó có gì, hành vi là undefined và chương trình thường sẽ crash.

Chuyện này hơi kiểu gà-với-trứng so với Array/Pointer Equivalence ở dưới, nhưng ta vẫn sẽ thử.

11.1.1 Cộng vào con trỏ

Đầu tiên, lấy một mảng số.

```
int a[5] = {11, 22, 33, 44, 55};
```

Rồi lấy con trỏ tới phần tử đầu tiên của mảng:

```
int a[5] = {11, 22, 33, 44, 55};

int *p = &a[0];  // Or "int *p = a;" works just as well
```

Rồi in giá trị ở đó bằng cách dereference con trỏ:

```
printf("%d\n", *p);  // Prints 11
```

Giờ dùng số học con trỏ để in phần tử kế tiếp trong mảng, phần tử ở index 1:

```
printf("%d\n", *(p + 1));  // Prints 22!!
```

Chuyện gì vừa xảy ra? C biết `p` là con trỏ tới một `int`. Nó biết `sizeof` của một `int`[1] và biết phải nhảy bao nhiêu byte để tới `int` kế tiếp sau cái đầu!

[1]Nhớ rằng toán tử `sizeof` cho biết kích cỡ tính bằng byte của một đối tượng trong bộ nhớ.

Thực ra, ví dụ trước có thể viết hai cách tương đương:

```
printf("%d\n", *p);        // Prints 11
printf("%d\n", *(p + 0));  // Prints 11
```

vì cộng `0` vào con trỏ cho ra cùng một con trỏ.

Rút ra gì? Ta có thể duyệt các phần tử của mảng theo cách này thay vì dùng mảng:

```
int a[5] = {11, 22, 33, 44, 55};

int *p = &a[0];  // Or "int *p = a;" works just as well

for (int i = 0; i < 5; i++) {
    printf("%d\n", *(p + i));  // Same as p[i]!
}
```

Và nó chạy giống hệt như dùng ký hiệu mảng! Oooo! Đến gần hơn với array/pointer equivalence rồi! Sẽ nói thêm ở phần sau của chương.

Nhưng thực chất chuyện gì đang xảy ra ở đây? Nó hoạt động thế nào?

Nhớ từ đầu rằng bộ nhớ giống như một mảng lớn, mỗi index của mảng lưu một byte?

Và index vào bộ nhớ có vài tên gọi:

- Index vào bộ nhớ
- Location
- Address (địa chỉ)
- *Pointer!* (con trỏ)

Vậy con trỏ là index vào bộ nhớ, ở một chỗ nào đó.

Lấy ví dụ ngẫu nhiên, giả sử số 3490 được lưu ở địa chỉ ("index") 23.237.489.202. Nếu ta có một con trỏ `int` tới số 3490 đó, giá trị của con trỏ đó là 23.237.489.202... bởi vì con trỏ là địa chỉ bộ nhớ. Cùng một thứ, chỉ khác cách gọi.

Giờ giả sử ta có thêm số nữa, 4096, được lưu ngay sau 3490 ở địa chỉ 23.237.489.210 (cao hơn 3490 là 8 vì mỗi `int` trong ví dụ này dài 8 byte).

Nếu cộng `1` vào con trỏ, thực ra nó nhảy tới trước `sizeof(int)` byte để tới `int` kế. Nó biết nhảy chừng đó vì là con trỏ `int`. Nếu là con trỏ `float`, nó sẽ nhảy tới trước `sizeof(float)` byte để tới float kế!

Vậy bạn có thể nhìn `int` kế tiếp bằng cách cộng `1` vào con trỏ, cái sau đó bằng cách cộng `2`, v.v.

11.1.2 Thay đổi con trỏ

Ở mục trước ta thấy cách cộng một số nguyên vào con trỏ. Lần này, ta sẽ *tự sửa chính con trỏ.*

Bạn có thể cộng (hoặc trừ) trực tiếp giá trị số nguyên vào (hoặc từ) bất kỳ con trỏ nào!

Làm lại ví dụ đó, nhưng có vài thay đổi. Đầu tiên, tôi sẽ thêm `999` vào cuối dãy số để làm sentinel (giá trị canh). Giá trị đó sẽ báo cho ta biết đâu là cuối dữ liệu.

```
int a[] = {11, 22, 33, 44, 55, 999};  // Add 999 here as a sentinel

int *p = &a[0];  // p points to the 11
```

Và ta cũng có `p` trỏ tới phần tử ở index `0` của `a`, tức `11`, giống như trước.

Giờ, bắt đầu *tăng* `p` để nó trỏ tới các phần tử tiếp theo của mảng. Ta làm vậy cho đến khi `p` trỏ tới `999`, tức là cho đến khi `*p == 999`:

```
while (*p != 999) {       // While the thing p points to isn't 999
    printf("%d\n", *p);   // Print it
    p++;                  // Move p to point to the next int!
}
```

Điên ghê, nhỉ?

Chạy thử, đầu tiên `p` trỏ tới `11`. Rồi tăng `p`, nó trỏ tới `22`, rồi lại tăng, trỏ tới `33`. Cứ thế cho đến khi trỏ tới `999` thì thoát.

11.1.3 Trừ con trỏ

Bạn có thể trừ một giá trị từ con trỏ để lui về địa chỉ trước đó, y như ta cộng vào vậy.

Nhưng ta cũng có thể trừ hai con trỏ để tìm khoảng cách giữa chúng, chẳng hạn ta có thể tính giữa hai `int*` có bao nhiêu `int`. Điểm lưu ý là chuyện này chỉ hoạt động trong cùng một mảng[2], nếu các con trỏ trỏ tới thứ khác, bạn nhận undefined behavior.

Nhớ chuỗi là `char*` trong C chứ? Xem thử có dùng cái này viết một biến thể `strlen()` để tính độ dài chuỗi bằng phép trừ con trỏ được không.

Ý tưởng là nếu có con trỏ tới đầu chuỗi, ta có thể tìm con trỏ tới cuối chuỗi bằng cách quét tới khi gặp ký tự `NUL`.

Và nếu có con trỏ tới đầu chuỗi, và tính được con trỏ tới cuối chuỗi, ta chỉ cần trừ hai con trỏ là ra độ dài!

```
#include <stdio.h>

int my_strlen(char *s)
{
    // Start scanning from the beginning of the string
    char *p = s;

    // Scan until we find the NUL character
    while (*p != '\0')
        p++;

    // Return the difference in pointers
    return p - s;
}

int main(void)
{
    printf("%d\n", my_strlen("Hello, world!"));  // Prints "13"
}
```

Nhớ rằng bạn chỉ được trừ con trỏ giữa hai con trỏ trỏ tới cùng một mảng!

11.2 Array/Pointer Equivalence

Cuối cùng thì cũng đến lúc nói chuyện này! Ta đã thấy kha khá ví dụ chỗ nào đó trộn lẫn ký hiệu mảng, nhưng giờ xin đưa ra *công thức căn bản của array/pointer equivalence*:

```
a[b] == *(a + b)
```

Nghiền đi! Chúng tương đương và dùng thay cho nhau được!

[2]Hoặc chuỗi, thực ra là mảng `char`. Hơi kỳ là bạn cũng có thể có con trỏ tham chiếu tới *một chỗ sau* phần cuối mảng mà vẫn làm toán với nó được. Chỉ là không được dereference khi nó ở đó.

Tôi đã đơn giản hoá một chút, vì trong ví dụ trên `a` và `b` đều có thể là biểu thức, và có khi ta cần thêm ngoặc để ép thứ tự toán tử nếu biểu thức phức tạp.

Spec thì luôn cụ thể, tuyên bố (trong C11 §6.5.2.1¶2):

> `E1[E2]` is identical to `(*((E1)+(E2)))`

nhưng cái đó hơi khó hình dung. Chỉ cần đảm bảo dùng ngoặc nếu biểu thức phức tạp để phép toán diễn ra đúng thứ tự.

Nghĩa là ta có thể *quyết định* dùng ký hiệu mảng hay ký hiệu con trỏ cho bất kỳ mảng hay con trỏ nào (giả định nó trỏ tới một phần tử của một mảng).

Dùng cả mảng và con trỏ với cả hai ký hiệu:

```
#include <stdio.h>

int main(void)
{
    int a[] = {11, 22, 33, 44, 55};

    int *p = a;  // p points to the first element of a, 11

    // Print all elements of the array a variety of ways:

    for (int i = 0; i < 5; i++)
        printf("%d\n", a[i]);      // Array notation with a

    for (int i = 0; i < 5; i++)
        printf("%d\n", p[i]);      // Array notation with p

    for (int i = 0; i < 5; i++)
        printf("%d\n", *(a + i));  // Pointer notation with a

    for (int i = 0; i < 5; i++)
        printf("%d\n", *(p + i));  // Pointer notation with p

    for (int i = 0; i < 5; i++)
        printf("%d\n", *(p++));    // Moving pointer p
        //printf("%d\n", *(a++));    // Moving array variable a--ERROR!
}
```

Có thể thấy nhìn chung, nếu bạn có biến mảng, có thể dùng ký hiệu con trỏ hay ký hiệu mảng để truy cập phần tử. Tương tự với biến con trỏ.

Khác biệt lớn duy nhất là bạn có thể *sửa* một con trỏ để trỏ sang địa chỉ khác, nhưng không làm thế được với biến mảng. Nói cách khác, bạn không gán được vào biến mảng, chỉ gán vào từng phần tử của mảng đó thôi.

Nếu thực sự muốn copy mảng này sang mảng khác, bạn phải dùng hàm như `memcpy()` (hay một vòng lặp).

11.2.1 Array/Pointer Equivalence trong lời gọi hàm

Đây chắc chắn là chỗ bạn gặp khái niệm này nhiều nhất.

Nếu bạn có hàm nhận đối số là con trỏ, ví dụ:

```
int my_strlen(char *s)
```

nghĩa là bạn có thể truyền hoặc mảng, hoặc con trỏ vào hàm này và nó vẫn chạy!

```
char s[] = "Antelopes";
char *t = "Wombats";

printf("%d\n", my_strlen(s));  // Works!
printf("%d\n", my_strlen(t));  // Works, too!
```

Và đó cũng là lý do hai signature hàm này tương đương:

```
int my_strlen(char *s)    // Works!
int my_strlen(char s[])   // Works, too!
```

11.3 Con trỏ `void`

Bạn đã thấy từ khoá `void` dùng với hàm để chỉ không có tham số hay không có giá trị trả về, nhưng cái này là một con thú hoàn toàn tách biệt, không liên quan.

Một `void*` chắc chắn là con trỏ tới một *thứ* đang tồn tại. Nhưng phần `void` chỉ ra rằng ta không biết *kiểu* của thứ đó. Và đôi khi, tin hay không tuỳ bạn, cái đó thực sự hữu ích. Nó cho phép viết code kiểu-bất-khả-tri hơn một chút, một sự linh hoạt rất nice trong một ngôn ngữ có kiểu như C.

Về cơ bản có hai trường hợp sử dụng, xem và giải hoặc một chút bí ẩn.

1. Hàm sẽ xử lý một thứ gì đó theo từng byte. Ví dụ, `memcpy()` chép byte bộ nhớ từ con trỏ này sang con trỏ kia, nhưng các con trỏ đó có thể trỏ tới kiểu bất kỳ. `memcpy()` tận dụng chuyện nếu bạn duyệt qua các `char*`, bạn đang duyệt qua các byte của một đối tượng bất kể đối tượng là kiểu gì. Sẽ nói thêm ở tiểu mục Multibyte Values.
2. Một hàm khác gọi một hàm bạn truyền vào cho nó (callback), và nó truyền dữ liệu cho bạn. Bạn biết kiểu của dữ liệu, nhưng hàm gọi bạn thì không. Nên nó truyền `void*` cho bạn, vì nó không biết kiểu, rồi bạn chuyển cái đó về kiểu mình cần. `qsort()` [3] và `bsearch()` [4] có sẵn đều dùng kỹ thuật này.

Xem ví dụ, hàm `memcpy()` có sẵn:

```
void *memcpy(void *s1, void *s2, size_t n);
```

Hàm này chép `n` byte bộ nhớ bắt đầu từ địa chỉ `s2` vào bộ nhớ bắt đầu từ địa chỉ `s1`.

Nhưng nhìn! `s1` và `s2` là `void*`! Vì sao? Nghĩa là gì? Thử thêm ví dụ.

Chẳng hạn, ta có thể chép một chuỗi bằng `memcpy()` (dù `strcpy()` phù hợp hơn cho chuỗi):

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

int main(void)
{
    char s[] = "Goats!";
    char t[100];

    memcpy(t, s, 7);  // Copy 7 bytes--including the NUL terminator!

    printf("%s\n", t);  // "Goats!"
}
```

Hoặc chép vài `int`:

[3] https://beej.us/guide/bgclr/html/split/stdlib.html#man-qsort
[4] https://beej.us/guide/bgclr/html/split/stdlib.html#man-bsearch

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

int main(void)
{
    int a[] = {11, 22, 33};
    int b[3];

    memcpy(b, a, 3 * sizeof(int));  // Copy 3 ints of data

    printf("%d\n", b[1]);  // 22
}
```

Cái này hơi hoang đấy, thấy ta vừa làm gì với `memcpy()` chứ? Ta chép dữ liệu từ `a` sang `b`, nhưng phải chỉ rõ chép bao nhiêu *byte*, và một `int` thì lớn hơn một byte.

Vậy, một `int` chiếm bao nhiêu byte? Trả lời: tuỳ hệ thống. Nhưng ta có thể biết bao nhiêu byte một kiểu chiếm bằng toán tử `sizeof`.

Vậy đây rồi: một `int` cần `sizeof(int)` byte bộ nhớ để lưu.

Và nếu có 3 cái trong mảng, như ví dụ đó, tổng dung lượng dùng cho 3 `int` phải là `3 * sizeof(int)`.

(Ở ví dụ chuỗi trước, chặt chẽ kỹ thuật hơn thì phải chép `7 * sizeof(char)` byte. Nhưng `char` theo định nghĩa luôn dài một byte, nên cái đó thoái hoá thành `7 * 1`.)

Ta thậm chí có thể chép một `float` hay `struct` bằng `memcpy()`! (Dù đây là lạm dụng, ta nên dùng `=` cho chuyện đó):

```
struct antelope my_antelope;
struct antelope my_clone_antelope;

// ...

memcpy(&my_clone_antelope, &my_antelope, sizeof my_antelope);
```

Nhìn `memcpy()` đa năng chưa! Nếu có con trỏ tới nguồn và con trỏ tới đích, và biết số byte muốn chép, bạn có thể chép *bất kỳ kiểu dữ liệu nào*.

Tưởng tượng nếu không có `void*`. Ta sẽ phải viết các hàm `memcpy()` chuyên biệt cho mỗi kiểu:

```
memcpy_int(int *a, int *b, int count);
memcpy_float(float *a, float *b, int count);
memcpy_double(double *a, double *b, int count);
memcpy_char(char *a, char *b, int count);
memcpy_unsigned_char(unsigned char *a, unsigned char *b, int count);

// etc... blech!
```

Tốt hơn nhiều là cứ dùng `void*` và có một hàm lo hết.

Đó là sức mạnh của `void*`. Bạn có thể viết hàm không quan tâm kiểu biến mà vẫn làm được việc với nó.

Nhưng sức mạnh lớn đi kèm trách nhiệm lớn. Có thể trách nhiệm không *đến mức* đó trong trường hợp này, nhưng có những giới hạn.

1. Không làm được số học con trỏ trên `void*`.
2. Không dereference được `void*`.
3. Không dùng được toán tử mũi tên trên `void*`, vì đó cũng là dereference.

4. Không dùng được ký hiệu mảng trên `void*`, vì đó cũng là dereference[5].

Và nếu nghĩ kỹ, các quy tắc này hợp lý. Tất cả thao tác đó dựa vào việc biết `sizeof` của kiểu dữ liệu được trỏ tới, mà với `void*` ta không biết kích cỡ của dữ liệu được trỏ tới, có thể là bất cứ gì!

Nhưng khoan, nếu không dereference được `void*` thì nó có ích gì cho bạn?

Giống như với `memcpy()`, nó giúp bạn viết các hàm tổng quát xử lý được nhiều kiểu dữ liệu. Nhưng bí mật là, ở cốt lõi, *bạn chuyển `void*` sang kiểu khác trước khi dùng*!

Và chuyển thì dễ: chỉ cần gán vào biến có kiểu mong muốn[6].

```
char a = 'X';  // A single char

void *p = &a;  // p points to the 'X'
char *q = p;   // q also points to the 'X'

printf("%c\n", *p);  // ERROR--cannot dereference void*!
printf("%c\n", *q);  // Prints "X"
```

Viết `memcpy()` của riêng mình để thử. Ta có thể chép byte (`char`), và biết số byte vì nó được truyền vào.

```
void *my_memcpy(void *dest, void *src, int byte_count)
{
    // Convert void*s to char*s
    char *s = src, *d = dest;

    // Now that we have char*s, we can dereference and copy them
    while (byte_count--) {
        *d++ = *s++;
    }

    // Most of these functions return the destination, just in case
    // that's useful to the caller.
    return dest;
}
```

Ngay đầu, ta chép `void*` vào `char*` để có thể dùng chúng như `char*`. Đơn giản vậy thôi.

Rồi vui vẻ trong một vòng while, nơi ta giảm `byte_count` đến khi thành false (`0`). Nhớ rằng với post-decrement, giá trị của biểu thức được tính (cho `while` dùng) *rồi* biến mới được giảm.

Và vui vẻ trong phần copy, nơi ta gán `*d = *s` để chép byte, nhưng làm với post-increment để cả `d` và `s` chuyển sang byte kế sau khi gán xong.

Cuối cùng, hầu hết các hàm về bộ nhớ và chuỗi trả về một bản sao của con trỏ tới đích phòng khi caller cần dùng.

Xong rồi, tôi chỉ muốn nhanh chóng chỉ ra rằng ta có thể dùng kỹ thuật này để duyệt qua các byte của *bất kỳ* đối tượng nào trong C, `float`, `struct`, hay gì cũng được!

Làm thêm một ví dụ thực tế với routine có sẵn `qsort()`, có thể sắp xếp *bất cứ gì* nhờ phép màu của `void*`.

(Trong ví dụ dưới, có thể bỏ qua từ `const`, ta chưa nói tới.)

[5]Vì nhớ rằng ký hiệu mảng chỉ là một dereference cộng chút toán tử con trỏ, mà bạn không dereference được `void*`!

[6]Bạn cũng có thể *cast* `void*` sang kiểu khác, nhưng ta chưa tới cast.

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

// The type of structure we're going to sort
struct animal {
    char *name;
    int leg_count;
};

// This is a comparison function called by qsort() to help it determine
// what exactly to sort by. We'll use it to sort an array of struct
// animals by leg_count.
int compar(const void *elem1, const void *elem2)
{
    // We know we're sorting struct animals, so let's make both
    // arguments pointers to struct animals
    const struct animal *animal1 = elem1;
    const struct animal *animal2 = elem2;

    // Return <0 =0 or >0 depending on whatever we want to sort by.

    // Let's sort ascending by leg_count, so we'll return the difference
    // in the leg_counts
    if (animal1->leg_count > animal2->leg_count)
        return 1;

    if (animal1->leg_count < animal2->leg_count)
        return -1;

    return 0;
}

int main(void)
{
    // Let's build an array of 4 struct animals with different
    // characteristics. This array is out of order by leg_count, but
    // we'll sort it in a second.
    struct animal a[4] = {
        {.name="Dog", .leg_count=4},
        {.name="Monkey", .leg_count=2},
        {.name="Antelope", .leg_count=4},
        {.name="Snake", .leg_count=0}
    };

    // Call qsort() to sort the array. qsort() needs to be told exactly
    // what to sort this data by, and we'll do that inside the compar()
    // function.
    //
    // This call is saying: qsort array a, which has 4 elements, and
    // each element is sizeof(struct animal) bytes big, and this is the
    // function that will compare any two elements.
    qsort(a, 4, sizeof(struct animal), compar);

    // Print them all out
    for (int i = 0; i < 4; i++) {
        printf("%d: %s\n", a[i].leg_count, a[i].name);
    }
```

```
}
```

Chỉ cần bạn đưa cho `qsort()` một hàm có thể so sánh hai item trong mảng cần sort, nó sắp được mọi thứ. Và làm vậy mà không cần phải hard-code kiểu của item ở đâu cả. `qsort()` chỉ sắp xếp lại các khối byte dựa vào kết quả của hàm `compar()` bạn truyền vào.

Chapter 12

Cấp phát bộ nhớ thủ công

Đây là một trong những mảng lớn C có khả năng khác với các ngôn ngữ bạn đã biết: *cấp phát bộ nhớ thủ công*.

Các ngôn ngữ khác dùng reference counting, garbage collection, hay các cách khác để quyết định khi nào cấp phát bộ nhớ mới cho dữ liệu, và khi nào giải phóng khi không còn biến nào tham chiếu tới nó nữa.

Và chuyện đó nice. Nice khi không cần lo nghĩ, cứ thả hết tham chiếu tới một item và tin rằng ở lúc nào đó bộ nhớ gắn với nó sẽ được giải phóng.

Nhưng C không hoàn toàn như vậy.

Dĩ nhiên trong C, một số biến được cấp phát và giải phóng tự động khi chúng vào scope và ra scope. Ta gọi chúng là biến automatic. Đó là các biến "local" block-scope bình thường. Không vấn đề gì.

Nhưng nếu bạn muốn cái gì đó tồn tại lâu hơn một block cụ thể thì sao? Đây là lúc quản lý bộ nhớ thủ công vào cuộc.

Bạn có thể bảo C cấp phát tường minh cho bạn một số byte nhất định để dùng theo ý muốn. Các byte này sẽ vẫn được cấp phát cho đến khi bạn tường minh giải phóng bộ nhớ đó[1].

Quan trọng là giải phóng bộ nhớ khi xong việc với nó! Nếu không, ta gọi đó là *memory leak* (rò rỉ bộ nhớ) và process của bạn sẽ tiếp tục chiếm bộ nhớ đó cho đến khi thoát.

Bạn đã cấp phát thủ công, thì bạn phải giải phóng thủ công khi xong việc.

Vậy làm sao? Ta sẽ học thêm vài hàm, và dùng toán tử `sizeof` để giúp biết cần cấp phát bao nhiêu byte.

Nói kiểu phổ thông trong C, dev hay nói biến automatic local được cấp phát "trên stack", còn bộ nhớ cấp phát thủ công thì "trên heap". Spec không nói tới hai thứ đó, nhưng mọi dev C đều hiểu nếu bạn nhắc tới.

Mọi hàm ta sẽ học trong chương này đều nằm trong `<stdlib.h>`.

12.1 Cấp phát và giải phóng, `malloc()` và `free()`

Hàm `malloc()` nhận số byte cần cấp phát, và trả về con trỏ void tới khối bộ nhớ vừa cấp phát.

Vì nó là `void*`, bạn có thể gán vào bất kỳ kiểu con trỏ nào tuỳ ý, thường con trỏ đó sẽ ứng theo cách nào đó với số byte bạn đang cấp phát.

Vậy nên cấp phát bao nhiêu byte? Ta có thể dùng `sizeof` giúp cho chuyện đó. Nếu muốn cấp phát đủ chỗ cho một `int`, ta có thể dùng `sizeof(int)` và truyền vào `malloc()`.

[1] Hoặc cho đến khi chương trình thoát, lúc đó mọi bộ nhớ được cấp phát sẽ được giải phóng. Dấu sao: một số hệ thống cho phép cấp phát bộ nhớ tồn tại cả sau khi chương trình thoát, nhưng chuyện đó phụ thuộc hệ thống, ngoài phạm vi của sách này, và chắc chắn bạn sẽ không vô tình làm thế.

Sau khi xong việc với đoạn bộ nhớ đã cấp phát, ta gọi `free()` để báo đã xong với bộ nhớ đó và có thể dùng cho việc khác. Đối số là cùng con trỏ bạn nhận từ `malloc()` (hoặc bản sao của nó). Dùng vùng bộ nhớ sau khi `free()` là undefined behavior.

Thử xem. Ta sẽ cấp phát đủ chỗ cho một `int`, rồi lưu cái gì đó vào đó, rồi in ra.

```
// Allocate space for a single int (sizeof(int) bytes-worth):

int *p = malloc(sizeof(int));

*p = 12;  // Store something there

printf("%d\n", *p);  // Print it: 12

free(p);  // All done with that memory

//*p = 3490;  // ERROR: undefined behavior! Use after free()!
```

Trong ví dụ bịa đặt đó, thực sự cũng chẳng lợi gì. Ta có thể dùng một `int` automatic và nó vẫn chạy. Nhưng rồi sẽ thấy khả năng cấp phát bộ nhớ cách này có những lợi thế, đặc biệt với các cấu trúc dữ liệu phức tạp hơn.

Một điều khác hay gặp tận dụng chuyện `sizeof` có thể cho biết kích cỡ của kiểu kết quả của bất kỳ biểu thức hằng nào. Nên bạn cũng có thể đặt tên biến vào đó dùng. Đây là ví dụ, y như cái trước:

```
int *p = malloc(sizeof *p);  // *p is an int, so same as sizeof(int)
```

12.2 Kiểm lỗi

Mọi hàm cấp phát đều trả về con trỏ tới vùng bộ nhớ vừa cấp phát, hoặc `NULL` nếu vì lý do nào đó bộ nhớ không cấp phát được.

Một số OS như Linux có thể được cấu hình sao cho `malloc()` không bao giờ trả `NULL`, kể cả khi hết bộ nhớ. Nhưng dù vậy, bạn vẫn luôn nên viết code có phòng bị.

```
int *x;

x = malloc(sizeof(int) * 10);

if (x == NULL) {
    printf("Error allocating 10 ints\n");
    // do something here to handle it
}
```

Đây là mẫu phổ biến bạn sẽ thấy, gán và kiểm tra trên cùng một dòng:

```
int *x;

if ((x = malloc(sizeof(int) * 10)) == NULL) {
    printf("Error allocating 10 ints\n");
    // do something here to handle it
}
```

12.3 Cấp phát cho mảng

Ta đã xem cách cấp phát cho một thứ, giờ thế nào với một đống chúng trong mảng?

Trong C, mảng là một đống những thứ giống nhau xếp sát nhau trong một vùng bộ nhớ liên tục.

Ta có thể cấp phát một vùng bộ nhớ liên tục, đã thấy cách rồi. Nếu muốn 3490 byte bộ nhớ, cứ đòi:

```
char *p = malloc(3490);  // Voila
```

Và, đúng thế!, đó là mảng 3490 `char` (cũng là một chuỗi!) vì mỗi `char` là 1 byte. Nói cách khác, `sizeof(char)` là `1`.

Chú ý: bộ nhớ vừa cấp phát không được khởi tạo gì, đầy rác. Dọn sạch bằng `memset()` nếu cần, hoặc xem `calloc()` ở dưới.

Nhưng ta chỉ cần nhân kích cỡ thứ ta muốn với số phần tử muốn, rồi truy cập bằng ký hiệu con trỏ hay ký hiệu mảng. Ví dụ!

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

int main(void)
{
    // Allocate space for 10 ints
    int *p = malloc(sizeof(int) * 10);

    // Assign them values 0-45:
    for (int i = 0; i < 10; i++)
        p[i] = i * 5;

    // Print all values 0, 5, 10, 15, ..., 40, 45
    for (int i = 0; i < 10; i++)
        printf("%d\n", p[i]);

    // Free the space
    free(p);
}
```

Chìa khoá ở dòng `malloc()`. Nếu biết mỗi `int` chiếm `sizeof(int)` byte, và muốn 10 cái, ta cấp phát đúng chừng đó byte với:

```
sizeof(int) * 10
```

Và mánh này chạy cho mọi kiểu. Cứ truyền vào `sizeof` rồi nhân với kích cỡ mảng.

12.4 Phương án khác: `calloc()`

Đây là một hàm cấp phát nữa, hoạt động tương tự `malloc()`, với hai khác biệt then chốt:

- Thay vì một đối số, bạn truyền kích cỡ của một phần tử, và số phần tử muốn cấp phát. Cứ như sinh ra để cấp phát mảng.
- Nó xoá bộ nhớ về không.

Bạn vẫn dùng `free()` để giải phóng bộ nhớ lấy qua `calloc()`.

Đây là so sánh giữa `calloc()` và `malloc()`.

```
// Allocate space for 10 ints with calloc(), initialized to 0:
int *p = calloc(10, sizeof(int));

// Allocate space for 10 ints with malloc(), initialized to 0:
```

```
int *q = malloc(10 * sizeof(int));
memset(q, 0, 10 * sizeof(int));   // set to 0
```

Lần nữa, kết quả giống nhau cả hai, chỉ khác `malloc()` không xoá bộ nhớ về 0 mặc định.

12.5 Đổi kích cỡ đã cấp phát với `realloc()`

Nếu bạn đã cấp phát 10 `int`, nhưng sau lại quyết định cần 20, làm sao?

Một lựa chọn là cấp phát chỗ mới rồi `memcpy()` sang... nhưng hoá ra đôi khi không cần dịch chuyển gì. Và có một hàm vừa đủ thông minh để làm điều đúng trong mọi trường hợp đúng: `realloc()`.

Nó nhận con trỏ tới vùng bộ nhớ đã cấp phát trước (qua `malloc()` hoặc `calloc()`) và kích cỡ mới cho vùng bộ nhớ đó.

Nó sẽ nới rộng hoặc thu nhỏ bộ nhớ đó, rồi trả về con trỏ tới nó. Đôi khi nó có thể trả về cùng con trỏ (nếu dữ liệu không cần chuyển đi đâu), hoặc con trỏ khác (nếu dữ liệu phải bị chép đi).

Chắc chắn khi gọi `realloc()`, bạn phải chỉ định số *byte* cần cấp phát, không phải số phần tử mảng! Tức là:

```
num_floats *= 2;

np = realloc(p, num_floats);  // WRONG: need bytes, not number of elements!

np = realloc(p, num_floats * sizeof(float));  // Better!
```

Cấp phát một mảng 20 `float`, rồi đổi ý thành 40 cái.

Ta sẽ gán giá trị trả về của realloc() vào một con trỏ khác để kiểm tra xem nó có phải NULL không. Nếu không phải, ta có thể gán lại vào con trỏ gốc. (Nếu gán thẳng giá trị trả về vào con trỏ gốc, ta sẽ mất con trỏ đó nếu hàm trả về `NULL` mà không có cách nào lấy lại.)

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

int main(void)
{
    // Allocate space for 20 floats
    float *p = malloc(sizeof *p * 20);  // sizeof *p same as sizeof(float)

    // Assign them fractional values 0.0-1.0:
    for (int i = 0; i < 20; i++)
        p[i] = i / 20.0;

    // But wait! Let's actually make this an array of 40 elements
    float *new_p = realloc(p, sizeof *p * 40);

    // Check to see if we successfully reallocated
    if (new_p == NULL) {
        printf("Error reallocing\n");
        return 1;
    }

    // If we did, we can just reassign p
    p = new_p;

    // And assign the new elements values in the range 1.0-2.0
```

```
    for (int i = 20; i < 40; i++)
        p[i] = 1.0 + (i - 20) / 20.0;

    // Print all values 0.0-2.0 in the 40 elements:
    for (int i = 0; i < 40; i++)
        printf("%f\n", p[i]);

    // Free the space
    free(p);
}
```

Chú ý chỗ ta lấy giá trị trả về từ `realloc()` rồi gán lại vào cùng biến con trỏ `p` đã truyền vào. Chuyện này khá phổ biến.

Cũng vậy, nếu dòng 7 trông lạ, với `sizeof *p` ở đó, nhớ rằng `sizeof` hoạt động trên kích cỡ của kiểu của biểu thức. Và kiểu của `*p` là `float`, nên dòng đó tương đương với `sizeof(float)`.

Cuối cùng, có thể hơi lạ là tôi không có `free(new_p)` ở đâu cả, dù đó là con trỏ trả về từ `realloc()`. Lý do là ta chép `new_p` vào `p` ở dòng 23, nên cả hai cùng giá trị; cả hai trỏ tới cùng một khối bộ nhớ, và chỉ có một khối. Nên khi `free()`, thực ra tôi free cái nào cũng được kết quả như nhau.

12.5.1 Đọc dòng có độ dài bất kỳ

Tôi muốn minh hoạ hai chuyện bằng ví dụ đầy đủ này.

1. Dùng `realloc()` để nới buffer khi đọc thêm dữ liệu.
2. Dùng `realloc()` để co buffer về kích cỡ hoàn hảo sau khi đã đọc xong.

Thứ ta thấy đây là một vòng lặp gọi `fgetc()` lặp đi lặp lại để nối vào buffer đến khi thấy ký tự cuối là newline.

Khi tìm thấy newline, nó thu buffer về đúng kích cỡ rồi trả về.

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

// Read a line of arbitrary size from a file
//
// Returns a pointer to the line.
// Returns NULL on EOF or error.
//
// It's up to the caller to free() this pointer when done with it.
//
// Note that this strips the newline from the result. If you need
// it in there, probably best to switch this to a do-while.

char *readline(FILE *fp)
{
    int offset = 0;   // Index next char goes in the buffer
    int bufsize = 4;  // Preferably power of 2 initial size
    char *buf;        // The buffer
    int c;            // The character we've read in

    buf = malloc(bufsize);  // Allocate initial buffer

    if (buf == NULL)   // Error check
        return NULL;
```

```
    // Main loop--read until newline or EOF
    while (c = fgetc(fp), c != '\n' && c != EOF) {

        // Check if we're out of room in the buffer accounting
        // for the extra byte for the NUL terminator
        if (offset == bufsize - 1) {  // -1 for the NUL terminator
            bufsize *= 2;  // 2x the space

            char *new_buf = realloc(buf, bufsize);

            if (new_buf == NULL) {
                free(buf);   // On error, free and bail
                return NULL;
            }

            buf = new_buf;  // Successful realloc
        }

        buf[offset++] = c;  // Add the byte onto the buffer
    }

    // We hit newline or EOF...

    // If at EOF and we read no bytes, free the buffer and
    // return NULL to indicate we're at EOF:
    if (c == EOF && offset == 0) {
        free(buf);
        return NULL;
    }

    // Shrink to fit
    if (offset < bufsize - 1) {  // If we're short of the end
        char *new_buf = realloc(buf, offset + 1); // +1 for NUL terminator

        // If successful, point buf to new_buf;
        // otherwise we'll just leave buf where it is
        if (new_buf != NULL)
            buf = new_buf;
    }

    // Add the NUL terminator
    buf[offset] = '\0';

    return buf;
}

int main(void)
{
    FILE *fp = fopen("foo.txt", "r");

    char *line;

    while ((line = readline(fp)) != NULL) {
        printf("%s\n", line);
        free(line);
    }
```

```
    fclose(fp);
}
```

Khi nới bộ nhớ kiểu này, thường (dù chẳng phải luật) là nhân đôi không gian cần thiết ở mỗi bước để giảm số lần `realloc()`.

Cuối cùng bạn có thể để ý rằng `readline()` trả về con trỏ tới buffer được `malloc()`. Vì vậy, caller có trách nhiệm `free()` tường minh bộ nhớ đó khi xong việc.

12.5.2 `realloc()` với `NULL`

Giờ tới Trivia! Hai dòng này tương đương:

```
char *p = malloc(3490);
char *p = realloc(NULL, 3490);
```

Có thể tiện khi bạn có vòng lặp cấp phát và không muốn xử lý riêng cho `malloc()` lần đầu.

```
int *p = NULL;
int length = 0;

while (!done) {
    // Allocate 10 more ints:
    length += 10;
    p = realloc(p, sizeof *p * length);

    // Do amazing things
    // ...
}
```

Trong ví dụ đó, ta không cần `malloc()` khởi tạo vì `p` ban đầu là `NULL`.

12.6 Cấp phát có canh lề

Chắc bạn sẽ không phải dùng cái này.

Và tôi không muốn đi quá xa chỗ cỏ dại bàn về nó ngay bây giờ, nhưng có một thứ gọi là *memory alignment* (canh lề bộ nhớ), liên quan tới chuyện địa chỉ bộ nhớ (giá trị con trỏ) là bội của một số cụ thể.

Ví dụ, hệ thống có thể yêu cầu các giá trị 16-bit phải bắt đầu ở địa chỉ bộ nhớ là bội của 2. Hoặc giá trị 64-bit phải bắt đầu ở địa chỉ là bội của 2, 4, hay 8, chẳng hạn. Tuỳ CPU.

Vài hệ thống yêu cầu kiểu canh lề này để truy cập bộ nhớ nhanh, hoặc một số hệ thì để truy cập được bộ nhớ tí nào.

Nếu bạn dùng `malloc()`, `calloc()`, hay `realloc()`, C sẽ đưa bạn một khối bộ nhớ được canh lề ổn cho bất kỳ giá trị nào, kể cả `struct`. Chạy trong mọi trường hợp.

Nhưng có thể có lúc bạn biết một số dữ liệu canh được theo biên nhỏ hơn, hoặc vì lý do nào đó phải canh theo biên lớn hơn. Tôi đoán chuyện này phổ biến hơn với lập trình hệ nhúng.

Trong những trường hợp đó, bạn có thể chỉ định một alignment với `aligned_alloc()`.

Alignment là số nguyên luỹ thừa của hai lớn hơn không, nên là `2`, `4`, `8`, `16`, v.v. và bạn đưa cái đó cho `aligned_alloc()` trước số byte bạn quan tâm.

Ràng buộc kia là số byte bạn cấp phát phải là bội của alignment. Nhưng chuyện này có thể đang thay đổi. Xem C Defect Report 460[2]

[2] http://www.open-std.org/jtc1/sc22/wg14/www/docs/summary.htm#dr_460

Ví dụ, cấp phát trên biên 64-byte:

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

int main(void)
{
    // Allocate 256 bytes aligned on a 64-byte boundary
    char *p = aligned_alloc(64, 256);  // 256 == 64 * 4

    // Copy a string in there and print it
    strcpy(p, "Hello, world!");
    printf("%s\n", p);

    // Free the space
    free(p);
}
```

Tôi muốn ghi chú ở đây về `realloc()` và `aligned_alloc()`. `realloc()` không có bảo đảm gì về alignment, nên nếu bạn cần lấy được vùng cấp phát lại đã canh lề, bạn sẽ phải làm kiểu khó với `memcpy()`.

Đây là một hàm `aligned_realloc()` không chuẩn, nếu bạn cần:

```
void *aligned_realloc(void *ptr, size_t old_size, size_t alignment, size_t size)
{
    char *new_ptr = aligned_alloc(alignment, size);

    if (new_ptr == NULL)
        return NULL;

    size_t copy_size = old_size < size? old_size: size;  // get min

    if (ptr != NULL)
        memcpy(new_ptr, ptr, copy_size);

    free(ptr);

    return new_ptr;
}
```

Chú ý rằng nó *luôn luôn* chép dữ liệu, tốn thời gian, trong khi `realloc()` thật sẽ tránh chép nếu có thể. Nên nó kém hiệu quả. Tránh phải cấp phát lại dữ liệu canh lề tuỳ chỉnh.

Chapter 13

Scope

Scope nói về chuyện biến nào nhìn thấy được trong ngữ cảnh nào.

13.1 Block scope

Đây là scope của gần như mọi biến dev định nghĩa. Nó bao gồm cả cái mà ngôn ngữ khác có thể gọi là "function scope", tức biến khai báo bên trong hàm.

Quy tắc cơ bản là nếu bạn khai báo biến trong một block được bao bởi cặp ngoặc xoăn xoắn xuýt, scope của biến đó là block đó.

Nếu có block trong block, thì biến khai báo trong block *bên trong* là local với block đó, không thấy được ở scope ngoài.

Khi scope của một biến kết thúc, biến đó không thể tham chiếu nữa, và bạn có thể coi giá trị của nó đã bay vào bit bucket[1] khổng lồ trên trời.

Ví dụ với scope lồng nhau:

```
#include <stdio.h>

int main(void)
{
    int a = 12;         // Local to outer block, but visible in inner block

    if  (a == 12) {
        int b = 99;     // Local to inner block, not visible in outer block

        printf("%d %d\n", a, b);  // OK: "12 99"
    }

    printf("%d\n", a);  // OK, we're still in a's scope

    printf("%d\n", b);  // ILLEGAL, out of b's scope
}
```

13.1.1 Chỗ nào định nghĩa biến

Một sự thật vui là bạn có thể định nghĩa biến ở bất cứ đâu trong block, trong mức độ hợp lý, chúng có scope của block đó, nhưng không dùng được trước khi được định nghĩa.

[1] https://en.wikipedia.org/wiki/Bit_bucket

```
#include <stdio.h>

int main(void)
{
    int i = 0;

    printf("%d\n", i);     // OK: "0"

    //printf("%d\n", j);   // ILLEGAL--can't use j before it's defined

    int j = 5;

    printf("%d %d\n", i, j);   // OK: "0 5"
}
```

Trước đây, C yêu cầu mọi biến phải định nghĩa trước bất kỳ code nào trong block, nhưng chuẩn C99 không còn thế nữa.

13.1.2 Che biến

Nếu bạn có biến đặt tên giống nhau ở scope trong và scope ngoài, cái ở scope trong được ưu tiên trong khi bạn đang chạy ở scope trong. Tức là nó *che* cái ở scope ngoài suốt thời gian tồn tại.

```
#include <stdio.h>

int main(void)
{
    int i = 10;

    {
        int i = 20;

        printf("%d\n", i);  // Inner scope i, 20 (outer i is hidden)
    }

    printf("%d\n", i);  // Outer scope i, 10
}
```

Có thể bạn đã để ý trong ví dụ đó tôi quẳng luôn một block vào ở dòng 7, chẳng có cả `for` hay `if` khởi sự! Chuyện này hoàn toàn hợp lệ. Thỉnh thoảng dev muốn gom một mớ biến local lại cho một tính toán nhanh và sẽ làm thế, nhưng hiếm khi thấy.

13.2 File scope

Nếu bạn định nghĩa biến ngoài block, biến đó có *file scope.* Nó nhìn thấy được trong mọi hàm trong file xuất hiện sau nó, và được chia sẻ giữa chúng. (Trường hợp ngoại lệ là nếu một block định nghĩa biến cùng tên, nó sẽ che cái ở file scope.)

Đây là cái gần nhất với khái niệm bạn có thể coi là scope "global" trong ngôn ngữ khác.

Ví dụ:

```
#include <stdio.h>

int shared = 10;    // File scope! Visible to the whole file after this!

void func1(void)
```

```
{
    shared += 100;  // Now shared holds 110
}

void func2(void)
{
    printf("%d\n", shared);  // Prints "110"
}

int main(void)
{
    func1();
    func2();
}
```

Chú ý rằng nếu `shared` được khai báo ở cuối file, nó sẽ không compile. Nó phải được khai báo *trước* bất kỳ hàm nào dùng nó.

Có những cách chỉnh thêm các item ở file scope, cụ thể với static và extern, nhưng sẽ nói thêm sau.

13.3 Scope vòng lặp `for`

Thật sự tôi không biết gọi nó là gì, vì C11 §6.8.5.3¶1 không cho nó một tên riêng. Ta đã dùng vài lần trong sách này rồi. Nó là khi bạn khai báo biến bên trong mệnh đề đầu của vòng `for`:

```
for (int i = 0; i < 10; i++)
    printf("%d\n", i);

printf("%d\n", i);  // ILLEGAL--i is only in scope for the for-loop
```

Trong ví dụ đó, thời gian sống của `i` bắt đầu ngay lúc nó được định nghĩa, và tiếp tục trong suốt vòng lặp.

Nếu thân vòng lặp nằm trong một block, các biến định nghĩa trong `for` nhìn thấy được từ scope bên trong đó.

Dĩ nhiên là trừ khi scope trong đó che chúng đi. Ví dụ điên rồ này in `999` năm lần:

```
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 5; i++) {
        int i = 999;  // Hides the i in the for-loop scope
        printf("%d\n", i);
    }
}
```

13.4 Ghi chú về function scope

Spec C có đề cập *function scope*, nhưng nó được dùng chỉ với *label*, thứ ta chưa bàn tới. Sẽ nói thêm hôm khác.

Chapter 14

Types II: Còn nhiều kiểu nữa!

Ta đã quen với các kiểu `char`, `int`, và `float`, nhưng giờ là lúc nâng những thứ đó lên tầm cao mới và xem còn gì nữa ở khoản kiểu!

14.1 Số nguyên có dấu và không dấu

Tới giờ ta dùng `int` như kiểu *có dấu* (signed), tức giá trị có thể âm hoặc dương. Nhưng C còn có các kiểu nguyên *không dấu* (unsigned) cụ thể, chỉ chứa được số dương.

Các kiểu này dùng từ khoá `unsigned` đi trước.

```
int a;           // signed
signed int a;    // signed
signed a;        // signed, "shorthand" for "int" or "signed int", rare
unsigned int b;  // unsigned
unsigned c;      // unsigned, shorthand for "unsigned int"
```

Vì sao? Vì sao bạn quyết định chỉ muốn chứa số dương?

Đáp: với biến unsigned, bạn có thể chứa số lớn hơn so với biến signed.

Nhưng vì sao thế?

Bạn có thể nghĩ về số nguyên được biểu diễn bởi một số *bit*[1]. Trên máy tôi, một `int` được biểu diễn bởi 64 bit.

Và mỗi hoán vị các bit là `1` hay `0` biểu diễn một số. Ta có thể quyết định chia các số này thế nào.

Với số signed, ta dùng (xấp xỉ) một nửa số hoán vị để biểu diễn số âm, nửa kia biểu diễn số dương.

Với unsigned, ta dùng *tất cả* các hoán vị để biểu diễn số dương.

Trên máy tôi với `int` 64-bit dùng two's complement[2] để biểu diễn số, tôi có các giới hạn sau về miền số nguyên:

Kiểu	Min	Max
`int`	`-9,223,372,036,854,775,808`	`9,223,372,036,854,775,807`
`unsigned int`	`0`	`18,446,744,073,709,551,615`

Chú ý số `unsigned int` dương lớn nhất xấp xỉ gấp đôi số `int` dương lớn nhất. Nên bạn có một chút linh hoạt.

[1] "Bit" là viết tắt của *binary digit* (chữ số nhị phân). Nhị phân chỉ là một cách biểu diễn số khác. Thay vì các chữ số 0-9 như ta quen, là các chữ số 0-1.

[2] https://en.wikipedia.org/wiki/Two%27s_complement

14.2 Kiểu ký tự

Nhớ `char` chứ? Kiểu dùng để chứa một ký tự?

```
char c = 'B';

printf("%c\n", c);  // "B"
```

Tôi có tin sốc cho bạn: nó thật ra là một số nguyên.

```
char c = 'B';

// Change this from %c to %d:
printf("%d\n", c);  // 66 (!!)
```

Ở tầng sâu, `char` chỉ là một `int` nhỏ, cụ thể là một số nguyên dùng đúng một byte chỗ, hạn chế miền giá trị xuống còn...

Ở đây spec C hơi khó chịu. Nó đảm bảo rằng `char` là một byte, tức `sizeof(char) == 1`. Nhưng rồi ở C11 §3.6¶3 nó đi thẳng ra nói:

> A byte is composed of a contiguous sequence of bits, *the number of which is implementation-defined.*

Khoan, cái gì? Có lẽ một số bạn quen với ý niệm một byte là 8 bit chứ? Ý tôi là đúng vậy mà, đúng không? Và câu trả lời là, "Gần như chắc chắn."[3] Nhưng C là ngôn ngữ đời cũ, và máy móc thời đó có, hãy nói là, ý kiến *thoáng* hơn về chuyện một byte có bao nhiêu bit. Và qua năm tháng, C vẫn giữ sự linh hoạt đó.

Nhưng giả định byte trong C là 8 bit, như thực tế với gần như mọi máy trên thế giới bạn từng thấy, miền của một `char` là...

Khoan, trước khi tôi nói được, hoá ra `char` có thể signed hoặc unsigned tuỳ compiler. Trừ khi bạn chỉ định rõ.

Nhiều trường hợp, có `char` là ổn vì bạn không quan tâm dấu của dữ liệu. Nhưng nếu cần signed hoặc unsigned `char`, bạn *phải* chỉ định rõ:

```
char a;           // Could be signed or unsigned
signed char b;    // Definitely signed
unsigned char c;  // Definitely unsigned
```

Được rồi, giờ cuối cùng, ta có thể tính miền các số nếu giả định `char` là 8 bit và hệ thống bạn dùng biểu diễn two's complement gần như phổ biến cho signed và unsigned[4].

Với các ràng buộc đó, cuối cùng ta có miền:

Kiểu `char`	Min	Max
`signed char`	`-128`	`127`
`unsigned char`	`0`	`255`

Và miền cho `char` là implementation-defined.

Cho tôi xác nhận lại. `char` thực ra là một số, vậy ta làm toán với nó được không?

Được! Chỉ cần nhớ giữ mọi thứ trong miền của `char`!

[3] Thuật ngữ ngành cho một dãy chính xác, không tranh cãi, 8 bit là *octet*.

[4] Nói chung, nếu bạn có số two's complement n bit, miền signed là -2^{n-1} tới $2^{n-1}-1$. Còn miền unsigned là 0 tới 2^n-1.

```
#include <stdio.h>

int main(void)
{
    char a = 10, b = 20;

    printf("%d\n", a + b);  // 30!
}
```

Thế còn các hằng ký tự trong dấu nháy đơn như `'B'`? Làm sao nó có giá trị số?

Spec ở đây cũng mơ hồ, vì C không được thiết kế để chạy trên một kiểu hệ thống nền duy nhất.

Nhưng cứ giả định tạm rằng bộ ký tự của bạn dựa trên ASCII[5] ít nhất cho 128 ký tự đầu. Trường hợp đó, hằng ký tự sẽ được chuyển thành một `char` có giá trị bằng giá trị ASCII của ký tự đó.

Nghe nhiều nhỉ. Xem ví dụ:

```
#include <stdio.h>

int main(void)
{
    char a = 10;
    char b = 'B';  // ASCII value 66

    printf("%d\n", a + b);  // 76!
}
```

Chuyện này tuỳ vào môi trường thực thi và bộ ký tự được dùng[6]. Một trong những bộ ký tự phổ biến nhất hiện nay là Unicode[7] (vốn là tập cha của ASCII), nên với các 0-9, A-Z, a-z và dấu câu cơ bản, bạn gần như chắc chắn lấy ra được giá trị ASCII.

14.3 Thêm kiểu nguyên: `short`, `long`, `long long`

Tới giờ nói chung ta mới dùng hai kiểu số nguyên:

- `char`
- `int`

và gần đây học thêm các biến thể unsigned của các kiểu nguyên. Và ta đã biết `char` bí mật là một `int` nhỏ trá hình. Nên ta biết `int` có thể tới với nhiều kích cỡ bit.

Nhưng còn vài kiểu nguyên nữa ta nên xem, và miền tối thiểu chúng có thể chứa. (Cài đặt của bạn có thể có miền rộng hơn so với spec yêu cầu, nhưng các miền ở đây là cái bạn có thể **chắc chắn** có portable khắp nơi.)

File header `<limits.h>` định nghĩa các macro chứa miền cho các kiểu khác nhau; dựa vào đó cho chắc, và *đừng bao giờ hard-code hay giả định các giá trị này.*

Các kiểu thêm này là `short int`, `long int`, và `long long int`. Khi dùng các kiểu này, dev C thường bỏ phần `int` đi (ví dụ `long long`), và compiler vẫn vui vẻ.

```
// These two lines are equivalent:
long long int x;
long long x;
```

[5] https://en.wikipedia.org/wiki/ASCII
[6] https://en.wikipedia.org/wiki/List_of_information_system_character_sets
[7] https://en.wikipedia.org/wiki/Unicode

```
// And so are these:
short int x;
short x;
```

Xem các kiểu dữ liệu nguyên và kích cỡ theo thứ tự tăng, nhóm theo tính signed. Lần nữa, các giới hạn min/max này là cái spec đảm bảo portable; hệ thống của bạn có thể có miền rộng hơn.

Kiểu	Min Bytes	Min	Max
`char`	1	-128 or 0	127 or 255[8]
`signed char`	1	-128	127
`short`	2	-32768	32767
`int`	2	-32768	32767
`long`	4	-2147483648	2147483647
`long long`	8	-9223372036854775808	9223372036854775807
`unsigned char`	1	0	255
`unsigned short`	2	0	65535
`unsigned int`	2	0	65535
`unsigned long`	4	0	4294967295
`unsigned long long`	8	0	18446744073709551615

Không có kiểu `long long long`. Bạn không thể cứ thêm `long` vào thế. Đừng ngốc.

> Fan two's complement có thể đã nhận ra gì đó hài về các số đó. Ví dụ, vì sao `signed char` dừng ở -127 chứ không phải -128? Nhớ: đây chỉ là tối thiểu spec yêu cầu. Vài cách biểu diễn số (như sign and magnitude[a]) chặn ở ±127.
>
> [a] https://en.wikipedia.org/wiki/Signed_number_representations#Signed_magnitude_representation

Chạy lại bảng đó trên hệ thống 64-bit two's complement của tôi xem ra gì:

Kiểu	My Bytes	Min	Max
`char`	1	-128	127[9]
`signed char`	1	-128	127
`short`	2	-32768	32767
`int`	4	-2147483648	2147483647
`long`	8	-9223372036854775808	9223372036854775807
`long long`	8	-9223372036854775808	9223372036854775807
`unsigned char`	1	0	255
`unsigned short`	2	0	65535
`unsigned int`	4	0	4294967295
`unsigned long`	8	0	18446744073709551615
`unsigned long long`	8	0	18446744073709551615

Hợp lý hơn chút, nhưng ta có thể thấy hệ thống của tôi có giới hạn lớn hơn so với tối thiểu trong spec.

Vậy các macro trong `<limits.h>` là gì?

Kiểu	Min Macro	Max Macro
`char`	`CHAR_MIN`	`CHAR_MAX`
`signed char`	`SCHAR_MIN`	`SCHAR_MAX`

[8] Tuỳ vào `char` mặc định là `signed char` hay `unsigned char`

[9] `char` của tôi là signed.

Kiểu	Min Macro	Max Macro
`short`	`SHRT_MIN`	`SHRT_MAX`
`int`	`INT_MIN`	`INT_MAX`
`long`	`LONG_MIN`	`LONG_MAX`
`long long`	`LLONG_MIN`	`LLONG_MAX`
`unsigned char`	`0`	`UCHAR_MAX`
`unsigned short`	`0`	`USHRT_MAX`
`unsigned int`	`0`	`UINT_MAX`
`unsigned long`	`0`	`ULONG_MAX`
`unsigned long long`	`0`	`ULLONG_MAX`

Chú ý có cách kín đáo ở đó để xác định hệ thống dùng `char` signed hay unsigned. Nếu `CHAR_MAX == UCHAR_MAX`, nó phải là unsigned.

Cũng chú ý không có macro min cho các biến thể `unsigned`, chúng chỉ là `0`.

14.4 Thêm float: `double` và `long double`

Xem spec nói gì về số dấu phẩy động ở §5.2.4.2.2¶1-2:

> The following parameters are used to define the model for each floating-point type:
>
> | Parameter | Definition |
> |---|---|
> | s | sign (± 1) |
> | b | base or radix of exponent representation (an integer > 1) |
> | e | exponent (an integer between a minimum e_{min} and a maximum e_{max}) |
> | p | precision (the number of base-b digits in the significand) |
> | f_k | nonnegative integers less than b (the significand digits) |
>
> A *floating-point number* (x) is defined by the following model:
>
> $$x = sb^e \sum_{k=1}^{p} f_k b^{-k}, \quad e_{min} \le e \le e_{max}$$

Hy vọng mọi chuyện giờ đã sáng tỏ với bạn.

Thôi được. Lùi lại một bước xem cái gì thực tế.

Chú ý: ta nhắc đến một đống macro trong mục này. Chúng nằm ở header `<float.h>`.

Số dấu phẩy động được mã hoá theo một chuỗi bit cụ thể (định dạng IEEE-754[10] vô cùng phổ biến) trong các byte.

Đào sâu thêm, số đó về cơ bản được biểu diễn là *significand* (phần số, bản thân các chữ số có nghĩa, đôi khi còn gọi là *mantissa*) và *exponent* (số mũ), tức luỹ thừa mà ta dùng để nâng các chữ số lên. Nhớ rằng số mũ âm làm số nhỏ đi.

Tưởng tượng ta dùng 10 làm số để nâng theo số mũ. Ta có thể biểu diễn các số sau bằng cách dùng significand là 12345, và số mũ là -3, 4, và 0 để mã hoá các giá trị dấu phẩy động sau:

$12345 \times 10^{-3} = 12.345$

[10] https://en.wikipedia.org/wiki/IEEE_754

$12345 \times 10^4 = 123450000$

$12345 \times 10^0 = 12345$

Với tất cả các số đó, significand giữ nguyên. Khác biệt duy nhất là số mũ.

Trên máy bạn, base của số mũ có lẽ là 2, không phải 10, vì máy tính thích nhị phân. Bạn có thể kiểm bằng cách in macro `FLT_RADIX`.

Vậy ta có một số biểu diễn bằng một số byte, mã hoá theo cách nào đó. Vì số mẫu bit có giới hạn, số dấu phẩy động biểu diễn được cũng có giới hạn.

Cụ thể hơn, chỉ một số chữ số thập phân có nghĩa nhất định được biểu diễn chính xác.

Làm sao để có nhiều hơn? Bạn dùng kiểu dữ liệu lớn hơn!

Và ta có vài cái. Ta biết `float` rồi, nhưng để có chính xác hơn thì có `double`. Và cho chính xác hơn nữa, có `long double` (không liên quan tới `long int` ngoài cái tên).

Spec không ghi mỗi kiểu nên chiếm bao nhiêu byte lưu trữ, nhưng trên máy tôi, ta có thể thấy kích cỡ tăng tương đối:

Kiểu	`sizeof`
`float`	4
`double`	8
`long double`	16

Nên mỗi kiểu (trên máy tôi) dùng các bit thêm đó cho độ chính xác cao hơn.

Nhưng *chính xác tới đâu*? Bao nhiêu số thập phân có thể biểu diễn được bằng các giá trị này?

Thì C cung cấp cho ta một đống macro trong `<float.h>` để giúp hình dung chuyện đó.

Hơi lắt léo nếu bạn dùng hệ base-2 (nhị phân) để lưu số (thực ra là gần như mọi người trên hành tinh này, có lẽ kể cả bạn), nhưng chịu khó theo khi ta tính ra.

14.4.1 Bao nhiêu chữ số thập phân?

Câu hỏi triệu đô là, "Tôi có thể lưu bao nhiêu chữ số thập phân có nghĩa trong một kiểu dấu phẩy động nhất định để lấy ra đúng số thập phân đó khi in?"

Số chữ số thập phân bạn có thể lưu trong kiểu dấu phẩy động và chắc chắn lấy lại đúng số đó khi in được cho bởi các macro sau:

Kiểu	Chữ số thập phân lưu được	Min
`float`	`FLT_DIG`	6
`double`	`DBL_DIG`	10
`long double`	`LDBL_DIG`	10

Trên máy tôi, `FLT_DIG` là 6, nên tôi có thể chắc rằng nếu in ra `float` 6 chữ số, tôi sẽ lấy lại đúng thứ đó. (Có thể nhiều chữ số hơn, một số số sẽ trở về đúng với nhiều chữ số hơn. Nhưng 6 thì chắc chắn trở về.)

Ví dụ, in ra các `float` theo mẫu tăng dần chữ số, có vẻ ta tới 8 chữ số trước khi có gì đó sai, nhưng sau đó ta quay về 7 chữ số đúng.

```
0.12345
0.123456
0.1234567
0.12345678
```

```
0.123456791  <-- Things start going wrong
0.1234567910
```

Làm demo nữa. Trong code này ta có hai `float` đều chứa số có `FLT_DIG` chữ số thập phân có nghĩa[11]. Rồi ta cộng chúng lại, đáng lẽ được 12 chữ số thập phân có nghĩa. Nhưng thế nhiều hơn ta có thể lưu trong một `float` và khôi phục về chuỗi đúng được, nên ta thấy khi in ra, mọi thứ bắt đầu sai sau chữ số có nghĩa thứ 7.

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

int main(void)
{
    // Both these numbers have 6 significant digits, so they can be
    // stored accurately in a float:

    float f = 3.14159f;
    float g = 0.00000265358f;

    printf("%.5f\n", f);   // 3.14159       -- correct!
    printf("%.11f\n", g);  // 0.00000265358 -- correct!

    // Now add them up
    f += g;                // 3.14159265358 is what f _should_ be

    printf("%.11f\n", f);  // 3.14159274101 -- wrong!
}
```

(Code trên có `f` sau các hằng số, cái này cho biết hằng đó là kiểu `float`, khác với mặc định là `double`. Sẽ nói thêm sau.)

Nhớ rằng `FLT_DIG` là số chữ số an toàn bạn có thể lưu trong một `float` và lấy lại đúng.

Đôi khi bạn có thể lấy ra thêm một hai chữ số. Nhưng đôi khi chỉ có `FLT_DIG` chữ số trở về. Điều chắc chắn: nếu bạn lưu bất kỳ số chữ số nào lên tới và bao gồm `FLT_DIG` trong một `float`, bạn chắc chắn lấy lại chúng đúng.

Vậy là hết chuyện. `FLT_DIG`. Hết.

...Hay chưa hết?

14.4.2 Chuyển sang thập phân và trở lại

Nhưng lưu số base 10 trong số dấu phẩy động và lấy ra chỉ mới là một nửa câu chuyện.

Hoá ra số dấu phẩy động có thể mã hoá các số cần nhiều chữ số thập phân hơn để in ra đầy đủ. Chỉ là số thập phân lớn của bạn có thể không ánh xạ tới một trong các số đó.

Tức là, khi nhìn các số dấu phẩy động đi từ cái này sang cái kế, có khoảng hở. Nếu bạn thử mã hoá một số thập phân trong khoảng hở đó, nó sẽ dùng số dấu phẩy động gần nhất. Đó là vì sao bạn chỉ có thể mã hoá `FLT_DIG` cho một `float`.

Nhưng còn các số dấu phẩy động *không* nằm trong khoảng hở thì sao? Cần bao nhiêu chữ số để in chúng ra chính xác?

Một cách đặt câu hỏi khác là với một số dấu phẩy động bất kỳ, tôi cần giữ bao nhiêu chữ số thập phân nếu muốn chuyển số thập phân đó ngược lại thành cùng số dấu phẩy động? Tức là tôi phải in bao nhiêu chữ số base 10 để khôi phục **tất cả** các chữ số base 2 trong số gốc?

[11] Chương trình này chạy như bình luận cho biết trên hệ thống có `FLT_DIG` là `6`, dùng số dấu phẩy động IEEE-754 base-2. Ngoài ra, bạn có thể có output khác.

Đôi khi có thể chỉ vài cái. Nhưng cho chắc, bạn sẽ muốn chuyển sang thập phân với một số vị trí thập phân an toàn nhất định. Số đó được mã hoá trong các macro sau:

Macro	Mô tả
`FLT_DECIMAL_DIG`	Số chữ số thập phân mã hoá trong `float`.
`DBL_DECIMAL_DIG`	Số chữ số thập phân mã hoá trong `double`.
`LDBL_DECIMAL_DIG`	Số chữ số thập phân mã hoá trong `long double`.
`DECIMAL_DIG`	Giống như cách mã hoá rộng nhất, `LDBL_DECIMAL_DIG`.

Xem ví dụ với `DBL_DIG` là 15 (nên đó là hết ta có thể có trong một hằng), nhưng `DBL_DECIMAL_DIG` là 17 (nên ta phải chuyển sang 17 số thập phân để giữ hết mọi bit của `double` gốc).

Gán số có 15 chữ số có nghĩa `0.123456789012345` cho `x`, và gán số có 1 chữ số có nghĩa `0.0000000000000006` cho `y`.

```
x is exact: 0.12345678901234500    Printed to 17 decimal places
y is exact: 0.00000000000000060
```

Nhưng cộng chúng lại. Đáng lẽ ra `0.1234567890123456`, nhưng thế là nhiều hơn `DBL_DIG`, nên chuyện lạ có thể xảy ra... xem:

```
x + y not quite right: 0.12345678901234559    Should end in 4560!
```

Đó là cái ta có vì in nhiều hơn `DBL_DIG`, đúng không? Nhưng xem này... số đó, ở trên, được biểu diễn chính xác đúng y như vậy!

Nếu ta gán `0.12345678901234559` (17 chữ số) cho `z` rồi in, ta có:

```
z is exact: 0.12345678901234559   17 digits correct! More than DBL_DIG!
```

Nếu ta cắt `z` về 15 chữ số, nó sẽ không còn là cùng một số. Đó là vì sao để giữ toàn bộ bit của một `double`, ta cần `DBL_DECIMAL_DIG`, chứ không chỉ `DBL_DIG` nhỏ hơn.

Tất cả đã nói, rõ ràng khi nghịch số thập phân nói chung, không an toàn khi in nhiều hơn `FLT_DIG`, `DBL_DIG`, hay `LDBL_DIG` chữ số để hợp lý tương ứng với các số base 10 gốc và bất kỳ phép toán tiếp theo.

Nhưng khi chuyển từ `float` sang biểu diễn thập phân rồi *trở lại* `float`, chắc chắn dùng `FLT_DECIMAL_DIG` để bảo toàn chính xác mọi bit.

14.5 Kiểu hằng số

Khi bạn viết một số hằng, như `1234`, nó có kiểu. Nhưng kiểu gì? Xem cách C quyết định kiểu hằng là gì, và cách ép nó chọn kiểu cụ thể.

14.5.1 Hệ cơ số 16 và cơ số 8

Ngoài hệ thập phân cũ kỹ như bà ngoại hay nướng, C còn hỗ trợ hằng ở các hệ cơ số khác.

Nếu bạn mở đầu một số bằng `0x`, nó được đọc như số hex:

```
int a = 0x1A2B;   // Hexadecimal
int b = 0x1a2b;   // Case doesn't matter for hex digits

printf("%x", a);  // Print a hex number, "1a2b"
```

Nếu bạn mở đầu một số bằng `0`, nó được đọc như số bát phân:

```
int a = 012;

printf("%o\n", a);  // Print an octal number, "12"
```

Cái này đặc biệt rắc rối với lập trình viên mới, khi họ cố độn số thập phân bên trái bằng `0` để canh cho đẹp, vô tình đổi luôn cơ số của số:

```
int x = 11111;  // Decimal 11111
int y = 00111;  // Decimal 73 (Octal 111)
int z = 01111;  // Decimal 585 (Octal 1111)
```

14.5.1.1 Ghi chú về nhị phân

Một mở rộng không chính thức[12] trong nhiều compiler C cho phép bạn biểu diễn số nhị phân với tiền tố `0b`:

```
int x = 0b101010;    // Binary 101010

printf("%d\n", x);   // Prints 42 decimal
```

Không có format specifier của `printf()` nào để in số nhị phân. Bạn phải làm từng ký tự một bằng toán tử bitwise.

14.5.2 Hằng số nguyên

Bạn có thể ép một hằng nguyên thành kiểu cụ thể bằng cách gắn hậu tố chỉ kiểu đó.

Ta sẽ làm vài gán để demo, nhưng dev hầu hết bỏ hậu tố trừ khi cần chính xác. Compiler khá giỏi trong việc đảm bảo các kiểu tương thích.

```
int           x = 1234;
long int      x = 1234L;
long long int x = 1234LL

unsigned int           x = 1234U;
unsigned long int      x = 1234UL;
unsigned long long int x = 1234ULL;
```

Hậu tố có thể viết hoa hay thường. Và `U` cùng `L` hay `LL` có thể xuất hiện cái nào trước cũng được.

Kiểu	Hậu tố
`int`	Không
`long int`	`L`
`long long int`	`LL`
`unsigned int`	`U`
`unsigned long int`	`UL`
`unsigned long long int`	`ULL`

Tôi có nhắc trong bảng rằng "không hậu tố" nghĩa là `int`... nhưng thực tế phức tạp hơn thế.

Vậy chuyện gì xảy ra khi bạn có số không hậu tố như:

[12] Tôi khá ngạc nhiên là C chưa có cái này trong spec. Trong tài liệu C99 Rationale, họ viết, "A proposal to add binary constants was rejected due to lack of precedent and insufficient utility." Nghe hơi ngốc khi so với một số tính năng khác họ tống vào! Tôi cược một trong các bản phát hành tới sẽ có.

```
int x = 1234;
```

Nó kiểu gì?

Cái C thường làm là chọn kiểu nhỏ nhất từ `int` trở lên có thể chứa giá trị đó.

Nhưng cụ thể, điều đó cũng tuỳ vào cơ số của số (thập phân, hex, hay bát phân).

Spec có một bảng ngon chỉ ra kiểu nào được dùng cho giá trị không hậu tố nào. Thực ra, tôi sẽ chỉ chép y nguyên vào đây.

C11 §6.4.4.1¶5 ghi, "The type of an integer constant is the first of the first of the corresponding list in which its value can be represented."

Rồi tiếp theo là bảng này:

Hậu tố	Hằng thập phân	Hằng bát phân hoặc hexadecimal

không có	`int`
	`long int`
		`int`
		`unsigned int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

`u` hay `U`	`unsigned int`
	`unsigned long int`
	`unsigned long long int`
		`unsigned int`
		`unsigned long int`
		`unsigned long long int`

`l` hay `L`	`long int`
	`long long int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

Cả `u` hay `U` và `l` hay `L`	`unsigned long int` `unsigned long long int`	`unsigned long int` `unsigned long long int`
`ll` hay `LL`	`long long int`	`long long int` `unsigned long long int`
Cả `u` hay `U` và `ll` hay `LL`	`unsigned long long int`	`unsigned long long int`

Ý nghĩa là, chẳng hạn, nếu bạn chỉ định một số như `123456789U`, C sẽ thử xem nó có vừa `unsigned int` không. Nếu không vừa, nó sẽ thử `unsigned long int`. Rồi `unsigned long long int`. Nó sẽ dùng kiểu nhỏ nhất có thể chứa số đó.

14.5.3 Hằng dấu phẩy động

Bạn nghĩ hằng dấu phẩy động như `1.23` sẽ có kiểu mặc định là `float` chứ?

Bất ngờ! Hoá ra số dấu phẩy động không hậu tố là kiểu `double`! Chúc mừng sinh nhật muộn nhé!

Bạn có thể ép nó thành kiểu `float` bằng cách gắn `f` (hoặc `F`, không phân biệt hoa thường). Bạn có thể ép nó thành `long double` bằng cách gắn `l` (hoặc `L`).

Kiểu	Hậu tố
`float`	`F`
`double`	Không
`long double`	`L`

Ví dụ:

```
float x       = 3.14f;
double x      = 3.14;
long double x = 3.14L;
```

Suốt thời gian qua ta vẫn làm thế này, đúng không?

```
float x = 3.14;
```

Bên trái không phải `float` và bên phải `double` sao? Đúng vậy! Nhưng C khá giỏi với chuyển đổi số tự động, nên hằng dấu phẩy động không hậu tố còn phổ biến hơn có hậu tố. Sẽ nói thêm sau.

14.5.3.1 Ký hiệu khoa học

Nhớ trước đó ta nói về chuyện số dấu phẩy động có thể biểu diễn bởi significand, base, và exponent chứ?

Có một cách viết phổ biến cho số kiểu đó, thể hiện ở đây kèm cái tương đương dễ nhận hơn là thứ bạn có khi thực sự chạy tính toán:

$1.2345 \times 10^3 = 1234.5$

Viết số dạng $s \times b^e$ gọi là *scientific notation*[13] (ký hiệu khoa học). Trong C, chúng được viết bằng "E notation", nên các cái này tương đương:

Scientific Notation	E notation
$1.2345 \times 10^{-3} = 0.0012345$	`1.2345e-3`
$1.2345 \times 10^8 = 123450000$	`1.2345e+8`

Bạn có thể in số theo ký hiệu này với `%e`:

```
printf("%e\n", 123456.0);  // Prints 1.234560e+05
```

Vài sự thật vui về ký hiệu khoa học:

- Bạn không bắt buộc phải viết với đúng một chữ số trước dấu thập phân. Có thể bao nhiêu chữ số cũng được đằng trước.

  ```
  double x = 123.456e+3;  // 123456
  ```

  Tuy nhiên khi in ra, nó sẽ đổi số mũ để chỉ có một chữ số trước dấu thập phân.

- Dấu cộng có thể bỏ ở số mũ, vì đó là mặc định, nhưng theo tôi thấy ít thấy trong thực tế.

  ```
  1.2345e10 == 1.2345e+10
  ```

- Bạn có thể áp hậu tố `F` hay `L` cho hằng E-notation:

[13] https://en.wikipedia.org/wiki/Scientific_notation

```
1.2345e10F
1.2345e10L
```

14.5.3.2 Hằng dấu phẩy động hexadecimal

Nhưng khoan, còn nhiều dấu phẩy động để xử!

Hoá ra cũng có hằng dấu phẩy động hexadecimal!

Chúng hoạt động tương tự số dấu phẩy động thập phân, nhưng bắt đầu bằng `0x` y như số nguyên.

Điều lưu ý là bạn *phải* chỉ định số mũ, và số mũ này tạo ra luỹ thừa của 2. Tức là: 2^x.

Rồi bạn dùng `p` thay cho `e` khi viết số:

Nên `0xa.1p3` là $10.0625 \times 2^3 == 80.5$.

Khi dùng hằng hex dấu phẩy động, Ta có thể in hex scientific notation với `%a`:

```
double x = 0xa.1p3;

printf("%a\n", x);  // 0x1.42p+6
printf("%f\n", x);  // 80.500000
```

Chapter 15

Types III: Chuyển đổi

Ở chương này, ta muốn nói hết về chuyện chuyển đổi từ kiểu này sang kiểu khác. C có nhiều cách để làm điều này, và một số có thể hơi khác bạn quen ở ngôn ngữ khác.

Trước khi nói cách ép chuyển đổi xảy ra, hãy bàn về cách chúng hoạt động khi chúng *đã* xảy ra.

15.1 Chuyển đổi chuỗi

Khác nhiều ngôn ngữ, C không làm chuyển đổi chuỗi-sang-số (và ngược lại) theo kiểu gọn gàng như với chuyển đổi số.

Với mấy thứ này, ta phải gọi hàm để làm việc bẩn.

15.1.1 Giá trị số sang chuỗi

Khi muốn chuyển số sang chuỗi, ta có thể dùng `sprintf()` (phát âm là *SPRINT-f*) hoặc `snprintf()` (*s-n-print-f*)[1]

Mấy cái này về cơ bản hoạt động như `printf()`, chỉ khác chúng xuất ra chuỗi, và bạn có thể in chuỗi đó sau, hay gì tuỳ ý.

Ví dụ, biến một phần giá trị π thành chuỗi:

```
#include <stdio.h>

int main(void)
{
    char s[10];
    float f = 3.14159;

    // Convert "f" to string, storing in "s", writing at most 10 characters
    // including the NUL terminator

    snprintf(s, 10, "%f", f);

    printf("String value: %s\n", s);  // String value: 3.141590
}
```

Bạn có thể dùng `%d` hay `%u` như bạn quen cho số nguyên.

[1] Chúng giống nhau, chỉ khác `snprintf()` cho phép bạn chỉ định số byte xuất tối đa, tránh tràn cuối chuỗi. Nên an toàn hơn.

15.1.2 Chuỗi sang giá trị số

Có hai họ hàm làm việc này trong C. Ta sẽ gọi chúng là họ `atoi` (phát âm *a-to-i*) và họ `strtol` (*stir-to-long*).

Để chuyển đổi cơ bản từ chuỗi sang số, thử các hàm `atoi` từ `<stdlib.h>`. Chúng có đặc tính xử lý lỗi tệ (kể cả undefined behavior nếu bạn truyền chuỗi xấu), nên dùng cẩn thận.

Hàm	Mô tả
`atoi`	Chuỗi sang `int`
`atof`	Chuỗi sang `float`
`atol`	Chuỗi sang `long int`
`atoll`	Chuỗi sang `long long int`

Dù spec không thừa nhận, chữ `a` đầu tên hàm là viết tắt của ASCII[2], nên thực ra `atoi()` là "ASCII-to-integer", nhưng nói thế giờ hơi quy ASCII về làm trung tâm.

Ví dụ chuyển chuỗi sang `float`:

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

int main(void)
{
    char *pi = "3.14159";
    float f;

    f = atof(pi);

    printf("%f\n", f);
}
```

Nhưng, như đã nói, ta có undefined behavior từ những chuyện lạ lùng như:

```
int x = atoi("what");  // "What" ain't no number I ever heard of
```

(Khi chạy cái đó, tôi nhận về `0`, nhưng bạn thật sự không nên trông cậy vào đó bằng bất cứ cách nào. Có thể bạn nhận về thứ hoàn toàn khác.)

Để có đặc tính xử lý lỗi tốt hơn, xem đống hàm `strtol`, cũng trong `<stdlib.h>`. Không chỉ thế, chúng còn chuyển sang nhiều kiểu và nhiều cơ số hơn!

Hàm	Mô tả
`strtol`	Chuỗi sang `long int`
`strtoll`	Chuỗi sang `long long int`
`strtoul`	Chuỗi sang `unsigned long int`
`strtoull`	Chuỗi sang `unsigned long long int`
`strtof`	Chuỗi sang `float`
`strtod`	Chuỗi sang `double`
`strtold`	Chuỗi sang `long double`

Các hàm này đều đi theo mẫu dùng tương tự, và là trải nghiệm đầu tiên của nhiều người với con-trỏ-tới-con-trỏ! Nhưng đừng lo, dễ hơn trông thấy nhiều.

[2] https://en.wikipedia.org/wiki/ASCII

Ví dụ chuyển chuỗi sang `unsigned long`, bỏ qua thông tin lỗi (tức thông tin về ký tự sai trong chuỗi đầu vào):

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

int main(void)
{
    char *s = "3490";

    // Convert string s, a number in base 10, to an unsigned long int.
    // NULL means we don't care to learn about any error information.

    unsigned long int x = strtoul(s, NULL, 10);

    printf("%lu\n", x);  // 3490
}
```

Chú ý vài thứ. Dù ta không hạ cố lấy thông tin gì về ký tự lỗi trong chuỗi, `strtoul()` không cho ta undefined behavior; nó chỉ trả về `0`.

Ta cũng chỉ định đây là số thập phân (base 10).

Thế nghĩa là ta có thể chuyển số ở cơ số khác? Chắc rồi! Làm nhị phân!

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

int main(void)
{
    char *s = "101010";  // What's the meaning of this number?

    // Convert string s, a number in base 2, to an unsigned long int.

    unsigned long int x = strtoul(s, NULL, 2);

    printf("%lu\n", x);  // 42
}
```

Được rồi, vui thú đó, nhưng cái `NULL` trong đó là gì? Để làm gì?

Nó giúp ta biết có lỗi xảy ra khi xử lý chuỗi hay không. Là một con trỏ tới con trỏ tới `char`, nghe đáng sợ, nhưng không còn đáng sợ khi bạn ghép được trong đầu.

Làm ví dụ với số xấu cố tình, xem `strtol()` báo cho ta vị trí của chữ số hợp lệ đầu tiên thế nào.

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

int main(void)
{
    char *s = "34x90";  // "x" is not a valid digit in base 10!
    char *badchar;

    // Convert string s, a number in base 10, to an unsigned long int.

    unsigned long int x = strtoul(s, &badchar, 10);
```

```
    // It tries to convert as much as possible, so gets this far:

    printf("%lu\n", x);  // 34

    // But we can see the offending bad character because badchar
    // points to it!

    printf("Invalid character: %c\n", *badchar);  // "x"
}
```

Thế là `strtoul()` chỉnh cái `badchar` trỏ tới để báo cho ta chỗ có chuyện không hay[3].

Nhưng nếu không có gì trục trặc thì sao? Trường hợp đó, `badchar` sẽ trỏ tới ký tự kết chuỗi `NUL` ở cuối chuỗi. Nên ta có thể kiểm tra:

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

int main(void)
{
    char *s = "3490";  // "x" is not a valid digit in base 10!
    char *badchar;

    // Convert string s, a number in base 10, to an unsigned long int.

    unsigned long int x = strtoul(s, &badchar, 10);

    // Check if things went well

    if (*badchar == '\0') {
        printf("Success! %lu\n", x);
    } else  {
        printf("Partial conversion: %lu\n", x);
        printf("Invalid character: %c\n", *badchar);
    }
}
```

Vậy là xong. Hàm kiểu `atoi()` tốt khi tình huống gấp có kiểm soát, nhưng hàm kiểu `strtol()` cho bạn kiểm soát tốt hơn hẳn về xử lý lỗi và cơ số đầu vào.

15.2 Chuyển đổi `char`

Nếu bạn có một ký tự chứa chữ số, như `'5'`... có giống giá trị `5` không?

Thử xem.

```
printf("%d %d\n", 5, '5');
```

Trên hệ thống UTF-8 của tôi, cái này in:

```
5 53
```

Vậy... không. Còn 53? Là gì? Đó là code point trong UTF-8 (và ASCII) cho ký tự `'5'`[4]

[3] Ta phải truyền con trỏ tới `badchar` cho `strtoul()` chứ không nó không chỉnh được theo cách ta thấy, tương tự lý do bạn phải truyền con trỏ tới `int` cho hàm nếu muốn hàm đó có thể đổi giá trị của `int` đó.

[4] Mỗi ký tự có một giá trị gắn với nó ứng với sơ đồ mã hoá ký tự cụ thể nào đó.

Vậy làm sao chuyển ký tự `'5'` (có giá trị 53 rõ ràng) thành giá trị `5`?

Bằng một trick lanh lợi, đây này!

Chuẩn C đảm bảo các ký tự này có code point liên tiếp và theo thứ tự này:

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

Nghĩ một giây, ta có thể tận dụng thế nào? Spoiler ở dưới...

Xem các ký tự và code point của chúng trong UTF-8:

```
0  1  2  3  4  5  6  7  8  9
48 49 50 51 52 53 54 55 56 57
```

Bạn thấy `'5'` là `53`, y như ta đã có. Và `'0'` là `48`.

Nên ta có thể trừ `'0'` khỏi bất kỳ ký tự chữ số nào để lấy giá trị số của nó:

```
char c = '6';

int x = c;  // x has value 54, the code point for '6'

int y = c - '0'; // y has value 6, just like we want
```

Và ta cũng có thể chuyển chiều kia, chỉ việc cộng giá trị vào.

```
int x = 6;

char c = x + '0';  // c has value 54

printf("%d\n", c);  // prints 54
printf("%c\n", c);  // prints 6 with %c
```

Bạn có thể nghĩ đây là cách lạ lùng để chuyển đổi, và theo chuẩn ngày nay, đúng là thế. Nhưng hồi xưa khi máy tính được làm bằng gỗ theo đúng nghĩa đen, đây là cách chuyển đổi. Và không hư, nên C chưa sửa.

15.3 Chuyển đổi số

15.3.1 Boolean

Nếu bạn chuyển một số 0 sang `bool`, kết quả là `0`. Ngược lại là `1`.

15.3.2 Chuyển giữa số nguyên

Nếu kiểu số nguyên được chuyển sang unsigned mà không vừa, kết quả unsigned sẽ wrap-around kiểu công-tơ-mét cho tới khi vừa kiểu unsigned[5].

Nếu kiểu số nguyên được chuyển sang số signed mà không vừa, kết quả là implementation-defined! Chuyện gì đó có ghi sẽ xảy ra, nhưng bạn phải tra tài liệu[6]

[5]Trong thực tế, cái nhiều khả năng đang xảy ra trên cài đặt của bạn là các bit bậc cao bị bỏ khỏi kết quả, nên số 16-bit `0x1234` khi chuyển sang số 8-bit sẽ thành `0x0034`, hay chỉ `0x34`.

[6]Lần nữa, trong thực tế, cái có khả năng xảy ra trên hệ thống của bạn là mẫu bit của số gốc sẽ bị cắt rồi dùng luôn để biểu diễn số signed, two's complement. Ví dụ, hệ của tôi lấy một `unsigned char` `192` và chuyển thành `signed char` `-64`. Trong two's complement, mẫu bit cho cả hai số này là nhị phân `11000000`.

15.3.3 Chuyển số nguyên và số dấu phẩy động

Nếu kiểu dấu phẩy động được chuyển sang kiểu nguyên, phần lẻ bị vứt đi không thương tiếc[7].

Nhưng, và đây là bẫy, nếu số quá lớn không vừa kiểu nguyên, bạn có undefined behavior. Nên đừng làm thế.

Đi từ số nguyên hay dấu phẩy động sang dấu phẩy động, C cố gắng hết sức để tìm số dấu phẩy động gần nhất với số nguyên.

Lần nữa, nếu giá trị gốc không biểu diễn được, đó là undefined behavior.

15.4 Chuyển đổi ngầm

Đây là các chuyển đổi compiler tự làm cho bạn khi bạn trộn các kiểu.

15.4.1 Integer Promotions

Ở nhiều chỗ, nếu một `int` có thể dùng để biểu diễn giá trị từ `char` hay `short` (signed hay unsigned), giá trị đó được *promote* (nâng) lên `int`. Nếu không vừa `int`, nó được nâng lên `unsigned int`.

Đó là cách ta làm được chuyện như:

```
char x = 10, y = 20;
int i = x + y;
```

Trường hợp đó, `x` và `y` được C nâng lên `int` trước khi phép toán diễn ra.

Integer promotion xảy ra trong The Usual Arithmetic Conversions, với hàm variadic[8], toán tử `+` và `-` đơn, hoặc khi truyền giá trị cho hàm không có prototype[9].

15.4.2 The Usual Arithmetic Conversions

Đây là các chuyển đổi tự động C làm quanh các phép toán số bạn yêu cầu. (Tên gọi thật sự là thế, nhân tiện, theo C11 §6.3.1.8.) Chú ý ở mục này, ta chỉ nói kiểu số, chuỗi sẽ bàn sau.

Các chuyển đổi này trả lời câu hỏi chuyện gì xảy ra khi bạn trộn kiểu, như:

```
int x = 3 + 1.2;   // Mixing int and double
                   // 4.2 is converted to int
                   // 4 is stored in x

float y = 12 * 2;  // Mixing float and int
                   // 24 is converted to float
                   // 24.0 is stored in y
```

Chúng thành `int`? Chúng thành `float`? Hoạt động thế nào?

Đây là các bước, diễn giải lại cho dễ nuốt.

1. Nếu có một thứ trong biểu thức là kiểu dấu phẩy động, chuyển các thứ khác sang kiểu dấu phẩy động đó.
2. Ngược lại, nếu cả hai đều là kiểu nguyên, thực hiện integer promotion trên mỗi cái, rồi làm kiểu của các toán hạng đủ lớn để chứa giá trị lớn chung. Đôi khi việc này liên quan đến chuyện đổi signed sang unsigned.

[7] Thật ra không, cứ vứt như thường.
[8] Hàm có số đối số thay đổi.
[9] Hiếm làm vì compiler sẽ than phiền và có prototype là *Cách làm đúng*. Tôi nghĩ cái này vẫn chạy vì lý do lịch sử, trước khi prototype ra đời.

Nếu muốn biết chi tiết vụn, xem C11 §6.3.1.8. Nhưng chắc bạn không muốn đâu.

Nhớ đại khái là kiểu int thành kiểu float nếu có kiểu dấu phẩy động ở đâu trong đó, và compiler cố gắng đảm bảo các kiểu int trộn không bị tràn.

Cuối cùng, nếu bạn chuyển từ kiểu dấu phẩy động này sang kiểu dấu phẩy động khác, compiler sẽ cố chuyển đổi chính xác. Nếu không được, nó sẽ làm xấp xỉ tốt nhất có thể. Nếu số quá lớn không vừa kiểu bạn đang chuyển qua, *bùm*: undefined behavior!

15.4.3 `void*`

Kiểu `void*` thú vị vì nó có thể chuyển từ hay sang bất kỳ kiểu con trỏ nào.

```
int x = 10;

void *p = &x;  // &x is type int*, but we store it in a void*

int *q = p;    // p is void*, but we store it in an int*
```

15.5 Chuyển đổi tường minh

Đây là các chuyển đổi từ kiểu sang kiểu mà bạn phải yêu cầu, compiler sẽ không tự làm.

Bạn có thể chuyển từ kiểu này sang kiểu khác bằng cách gán với `=`.

Bạn cũng có thể chuyển tường minh bằng *cast*.

15.5.1 Casting

Bạn có thể đổi tường minh kiểu của biểu thức bằng cách đặt một kiểu mới trong ngoặc trước nó. Vài dev C cau mày với cách này trừ khi thật sự cần, nhưng bạn có khả năng gặp một ít code C có cast bên trong.

Làm ví dụ muốn chuyển `int` sang `long` để lưu trong `long`.

Chú ý: ví dụ này bịa đặt và cast ở đây hoàn toàn không cần vì biểu thức `x + 12` sẽ tự chuyển sang `long int` để hợp với kiểu rộng hơn của `y`.

```
int x = 10;
long int y = (long int)x + 12;
```

Trong ví dụ đó, mặc dù `x` là kiểu `int` trước đó, biểu thức `(long int)x` có kiểu `long int`. Ta nói, "Ta cast `x` sang `long int`."

Thường gặp hơn, bạn có thể thấy cast được dùng để chuyển `void*` thành kiểu con trỏ cụ thể để có thể dereference.

Callback từ hàm có sẵn `qsort()` có thể thể hiện hành vi này vì nó có `void*` được truyền vào:

```
int compar(const void *elem1, const void *elem2)
{
    if (*((const int*)elem2) > *((const int*)elem1)) return 1;
    if (*((const int*)elem2) < *((const int*)elem1)) return -1;
    return 0;
}
```

Nhưng bạn cũng có thể viết rõ ràng bằng phép gán:

```
int compar(const void *elem1, const void *elem2)
{
    const int *e1 = elem1;
    const int *e2 = elem2;

    return *e2 - *e1;
}
```

Một chỗ bạn thấy cast phổ biến hơn là để tránh warning khi in giá trị con trỏ với `%p` hiếm dùng, cái này khó tính với bất cứ thứ gì không phải `void*` :

```
int x = 3490;
int *p = &x;

printf("%p\n", p);
```

sinh ra warning này:

```
warning: format ‘%p’ expects argument of type ‘void *’, but argument
         2 has type ‘int *’
```

Bạn có thể fix bằng cast:

```
printf("%p\n", (void *)p);
```

Chỗ khác là với đổi con trỏ tường minh, nếu không muốn dùng `void*` ở giữa, nhưng cái này cũng khá hiếm:

```
long x = 3490;
long *p = &x;
unsigned char *c = (unsigned char *)p;
```

Chỗ thứ ba thường yêu cầu là với các hàm chuyển đổi ký tự trong `<ctype.h>`[10] ở đó bạn nên cast các giá trị signedness đáng ngờ sang `unsigned char` để tránh undefined behavior.

Một lần nữa, cast hiếm khi *cần* trong thực tế. Nếu bạn thấy mình đang cast, có khả năng có cách khác làm cùng chuyện đó, hoặc có thể bạn đang cast không cần thiết.

Hoặc có thể là cần. Cá nhân tôi, tôi cố tránh, nhưng không ngại dùng nếu phải.

[10] https://beej.us/guide/bgclr/html/split/ctype.html

Chapter 16

Types IV: Qualifiers và Specifiers

Giờ ta đã có thêm vài kiểu dưới tay rồi, hóa ra ta có thể gắn cho chúng thêm vài thuộc tính để điều khiển cách chúng cư xử. Đó chính là *type qualifier* (bổ từ kiểu) và *storage-class specifier* (specifier lớp lưu trữ).

16.1 Type Qualifier

Mấy thứ này sẽ cho phép bạn khai báo giá trị hằng, và cũng cho compiler thêm gợi ý tối ưu hóa mà nó có thể dùng.

16.1.1 `const`

Đây là type qualifier phổ biến nhất bạn sẽ gặp. Nó nghĩa là biến đó là hằng, và bất kỳ nỗ lực nào hòng sửa nó sẽ khiến compiler nổi đóa.

```
const int x = 2;

x = 4;  // COMPILER PUKING SOUNDS, can't assign to a constant
```

Bạn không thể đổi giá trị `const`.

Bạn cũng hay thấy `const` trong danh sách tham số của hàm:

```
void foo(const int x)
{
    printf("%d\n", x + 30);  // OK, doesn't modify "x"
}
```

16.1.1.1 `const` và con trỏ

Chỗ này hơi lạ đời, vì có hai cách dùng mang hai ý nghĩa khác nhau khi dính tới con trỏ.

Một là, ta có thể làm sao cho bạn không đổi được thứ mà con trỏ trỏ đến. Bạn làm thế bằng cách đặt `const` ra phía trước cùng với tên kiểu (trước dấu sao) trong khai báo kiểu.

```
int x[] = {10, 20};
const int *p = x;

p++;  // We can modify p, no problem

*p = 30; // Compiler error! Can't change what it points to
```

Hơi khó hiểu tí, nhưng hai thứ sau là tương đương:

```
const int *p;  // Can't modify what p points to
int const *p;  // Can't modify what p points to, just like the previous line
```

Hay rồi, vậy là ta không đổi được thứ con trỏ trỏ đến, nhưng vẫn đổi được bản thân con trỏ. Thế nếu muốn ngược lại thì sao? Ta muốn đổi được thứ con trỏ trỏ đến nhưng *không* đổi được bản thân con trỏ?

Chỉ cần chuyển `const` ra sau dấu sao trong khai báo:

```
int *const p;   // We can't modify "p" with pointer arithmetic

p++;  // Compiler error!
```

Nhưng ta vẫn đổi được thứ nó trỏ đến:

```
int x = 10;
int *const p = &x;

*p = 20;   // Set "x" to 20, no problem
```

Bạn cũng có thể làm cả hai thứ đều `const`:

```
const int *const p;  // Can't modify p or *p!
```

Cuối cùng, nếu bạn có nhiều mức gián tiếp, bạn nên đặt `const` ở đúng mức. Một con trỏ `const` không có nghĩa là con trỏ mà nó trỏ tới cũng phải thế. Bạn có thể đặt tường minh như mấy ví dụ sau:

```
char **p;
p++;     // OK!
(*p)++;  // OK!

char **const p;
p++;     // Error!
(*p)++;  // OK!

char *const *p;
p++;     // OK!
(*p)++;  // Error!

char *const *const p;
p++;     // Error!
(*p)++;  // Error!
```

16.1.1.2 `const` Correctness

Còn một chuyện nữa tôi phải nhắc, là compiler sẽ cảnh báo với thứ kiểu thế này:

```
const int x = 20;
int *p = &x;
```

nói đại loại như:

```
initialization discards 'const' qualifier from pointer type target
```

Có chuyện gì ở đây?

Ta cần nhìn kiểu ở hai bên dấu gán:

```
    const int x = 20;
    int *p = &x;
//    ^       ^
//    |       |
//  int*    const int*
```

Compiler đang cảnh báo rằng giá trị bên phải dấu gán là `const`, còn bên trái thì không. Và compiler đang báo cho ta biết rằng nó đang vứt đi tính "const" của biểu thức bên phải.

Tức là ta *vẫn* có thể thử làm như dưới đây, nhưng nó sai. Compiler sẽ cảnh báo, và đó là hành vi không xác định:

```
const int x = 20;
int *p = &x;

*p = 40;  // Undefined behavior--maybe it modifies "x", maybe not!

printf("%d\n", x);  // 40, if you're lucky
```

16.1.2 `restrict`

TLDR: bạn không bao giờ phải dùng cái này và có thể lờ nó đi mỗi khi thấy. Nếu bạn dùng đúng, rất có thể bạn sẽ được chút hiệu năng. Nếu dùng sai, bạn sẽ được undefined behavior.

`restrict` là gợi ý cho compiler rằng một vùng bộ nhớ cụ thể sẽ chỉ được truy cập qua đúng một con trỏ chứ không phải qua cái khác. (Tức là sẽ không có aliasing với đối tượng mà con trỏ `restrict` trỏ tới.) Nếu dev khai báo một con trỏ là `restrict` rồi truy cập đối tượng đó theo cách khác (ví dụ qua con trỏ khác), hành vi là không xác định.

Về cơ bản bạn đang nói với C: "Này, tôi đảm bảo rằng cái con trỏ duy nhất này là đường duy nhất tôi truy cập bộ nhớ đó, và nếu tôi nói dối thì anh cứ quẳng undefined behavior vào mặt tôi."

Và C dùng thông tin đó để thực hiện một số tối ưu. Ví dụ, nếu bạn dereference con trỏ `restrict` lặp đi lặp lại trong vòng lặp, C có thể quyết định cache kết quả trong một thanh ghi và chỉ lưu kết quả cuối cùng khi vòng lặp xong. Nếu có con trỏ khác cùng trỏ đến vùng nhớ đó và truy cập trong vòng lặp, kết quả sẽ không chính xác.

(Lưu ý là `restrict` không có tác dụng nếu đối tượng được trỏ tới không bao giờ được ghi. Tất cả là về tối ưu quanh chuyện ghi vào bộ nhớ.)

Ta viết thử một hàm hoán đổi hai biến, và sẽ dùng từ khóa `restrict` để cam kết với C rằng ta sẽ không bao giờ truyền vào hai con trỏ cùng trỏ tới một chỗ. Rồi ta làm hỏng cam kết đó và thử truyền hai con trỏ cùng trỏ tới một chỗ.

```
void swap(int *restrict a, int *restrict b)
{
    int t;

    t = *a;
    *a = *b;
    *b = t;
}

int main(void)
{
    int x = 10, y = 20;

    swap(&x, &y);  // OK! "a" and "b", above, point to different things
```

```
    swap(&x, &x);  // Undefined behavior! "a" and "b" point to the same thing
}
```

Nếu ta bỏ các từ khóa `restrict` ra, cả hai lời gọi trên đều an toàn. Nhưng khi đó compiler có thể không tối ưu được.

`restrict` có block scope, tức là hạn chế chỉ kéo dài trong scope nó được dùng. Nếu nó ở danh sách tham số của một hàm, nó ở trong block scope của hàm đó.

Nếu con trỏ bị restrict trỏ vào một mảng, nó chỉ áp dụng cho từng đối tượng trong mảng. Các con trỏ khác vẫn có thể đọc và ghi mảng đó miễn là không đọc hay ghi cùng phần tử mà con trỏ restrict đã chạm vào.

Nếu nó ở ngoài mọi hàm, tức ở file scope, hạn chế áp dụng cho toàn bộ chương trình.

Bạn rất có thể sẽ thấy cái này trong các hàm thư viện như `printf()`:

```
int printf(const char * restrict format, ...);
```

Vẫn thế, nó chỉ báo với compiler rằng bên trong hàm `printf()`, chỉ có đúng một con trỏ nào đó trỏ đến phần bất kỳ của chuỗi `format`.

Một chú ý cuối: nếu vì lý do nào đó bạn dùng ký pháp mảng cho tham số hàm thay vì ký pháp con trỏ, bạn có thể dùng `restrict` như sau:

```
void foo(int p[restrict])     // With no size

void foo(int p[restrict 10])  // Or with a size
```

Nhưng ký pháp con trỏ thì phổ biến hơn.

16.1.3 `volatile`

Bạn ít khi gặp hay cần cái này trừ khi đang trực tiếp làm việc với phần cứng.

`volatile` báo với compiler rằng một giá trị có thể thay đổi sau lưng nó và phải được tra cứu lại mỗi lần.

Ví dụ: compiler đang nhìn vào bộ nhớ ở một địa chỉ mà liên tục được cập nhật trong hậu trường, ví dụ như một bộ đếm thời gian phần cứng.

Nếu compiler quyết định tối ưu chuyện đó bằng cách lưu giá trị trong một thanh ghi suốt một quãng dài, giá trị trong bộ nhớ sẽ đổi còn thanh ghi thì không phản ánh.

Khi khai báo một thứ là `volatile`, bạn đang nói với compiler: "Này, thứ con trỏ này trỏ đến có thể đổi bất cứ lúc nào vì lý do ngoài mã của chương trình này."

```
volatile int *p;
```

16.1.4 `_Atomic`

Đây là một tính năng C tùy chọn mà ta sẽ bàn ở chương Atomics.

16.2 Storage-Class Specifiers

Storage-class specifier tương tự như type qualifier. Chúng cho compiler thêm thông tin về kiểu của một biến.

16.2.1 `auto`

Bạn gần như không bao giờ thấy từ khóa này, vì `auto` là mặc định cho biến ở block scope. Nó được ngầm định.

Hai đoạn sau là như nhau:

```
{
    int a;         // auto is the default...
    auto int a;    // So this is redundant
}
```

Từ khóa `auto` cho biết đối tượng này có *automatic storage duration* (thời gian tồn tại tự động). Tức là nó tồn tại trong scope mà nó được định nghĩa, và tự động được giải phóng khi thoát scope.

Một cái bẫy với biến automatic là giá trị của chúng là không xác định cho đến khi bạn khởi tạo tường minh. Ta nói chúng đầy dữ liệu "random" hoặc "rác", dù tôi không ưa cả hai cách gọi đó lắm. Dù sao, bạn sẽ không biết trong đó có gì trừ khi bạn khởi tạo nó.

Luôn khởi tạo mọi biến automatic trước khi dùng!

16.2.2 `static`

Từ khóa này có hai nghĩa, tùy thuộc biến ở file scope hay block scope.

Bắt đầu với block scope.

16.2.2.1 `static` ở block scope

Ở đây, về cơ bản ta đang nói: "Tôi chỉ muốn một phiên bản duy nhất của biến này tồn tại, dùng chung giữa các lần gọi."

Tức là giá trị của nó sẽ giữ nguyên giữa các lời gọi.

`static` ở block scope kèm initializer sẽ chỉ được khởi tạo đúng một lần lúc khởi động chương trình, chứ không phải mỗi lần hàm được gọi.

Làm ví dụ cái xem:

```
#include <stdio.h>

void counter(void)
{
    static int count = 1;  // This is initialized one time

    printf("This has been called %d time(s)\n", count);

    count++;
}

int main(void)
{
    counter();  // "This has been called 1 time(s)"
    counter();  // "This has been called 2 time(s)"
    counter();  // "This has been called 3 time(s)"
    counter();  // "This has been called 4 time(s)"
}
```

Thấy cách giá trị của `count` còn giữ giữa các lời gọi không?

Một chuyện đáng lưu ý là biến `static` ở block scope mặc định được khởi tạo bằng `0`.

```
static int foo;      // Default starting value is `0`...
static int foo = 0;  // So the `0` assignment is redundant
```

Cuối cùng, hãy nhớ rằng nếu bạn viết chương trình đa luồng, bạn phải chắc chắn không để nhiều luồng cùng dẫm lên một biến.

16.2.2.2 `static` ở file scope

Khi ra tới file scope, ngoài mọi block, nghĩa thay đổi kha khá.

Biến ở file scope vốn đã tồn tại giữa các lời gọi hàm, nên chuyện đó đã sẵn ở đó rồi.

Thay vào đó, `static` trong ngữ cảnh này nghĩa là biến này không nhìn thấy được bên ngoài file mã nguồn này. Hơi giống "global", nhưng chỉ trong file này.

Sẽ nói thêm ở phần build từ nhiều file mã nguồn.

16.2.3 `extern`

Storage-class specifier `extern` cho ta cách tham chiếu đến các đối tượng trong file mã nguồn khác.

Ví dụ, giả sử file `bar.c` chỉ có đúng đoạn sau:

```
// bar.c

int a = 37;
```

Chỉ vậy thôi. Khai báo một `int a` mới ở file scope.

Nhưng nếu ta có file mã nguồn khác là `foo.c`, và muốn tham chiếu đến `a` ở trong `bar.c` thì sao?

Dễ với từ khóa `extern`:

```
// foo.c

extern int a;

int main(void)
{
    printf("%d\n", a);  // 37, from bar.c!

    a = 99;

    printf("%d\n", a);  // Same "a" from bar.c, but it's now 99
}
```

Ta cũng có thể đặt `extern int a` trong block scope, nó vẫn tham chiếu tới `a` trong `bar.c`:

```
// foo.c

int main(void)
{
    extern int a;

    printf("%d\n", a);  // 37, from bar.c!

    a = 99;

    printf("%d\n", a);  // Same "a" from bar.c, but it's now 99
```

```
}
```

Bây giờ, nếu `a` trong `bar.c` đã được đánh dấu `static`, chuyện này sẽ không hoạt động. Biến `static` ở file scope không nhìn thấy được bên ngoài file đó.

Một ghi chú cuối về `extern` với hàm. Với hàm, `extern` là mặc định, nên nó thừa. Bạn có thể khai báo hàm là `static` nếu chỉ muốn nó nhìn thấy được trong một file mã nguồn duy nhất.

16.2.4 `register`

Đây là từ khóa để gợi ý cho compiler rằng biến này được dùng thường xuyên, và nên được làm cho truy cập nhanh nhất có thể. Compiler không có nghĩa vụ phải đồng ý.

Giờ thì, bộ tối ưu của compiler C hiện đại khá giỏi trong việc tự tìm ra chuyện này, nên hiếm khi thấy từ khóa này ngày nay.

Nhưng nếu bạn phải dùng:

```
#include <stdio.h>

int main(void)
{
    register int a;   // Make "a" as fast to use as possible.

    for (a = 0; a < 10; a++)
        printf("%d\n", a);
}
```

Có cái giá đi kèm. Bạn không thể lấy địa chỉ của một register:

```
register int a;
int *p = &a;    // COMPILER ERROR! Can't take address of a register
```

Điều tương tự áp dụng cho bất kỳ phần nào của một mảng:

```
register int a[] = {11, 22, 33, 44, 55};
int *p = a;  // COMPILER ERROR! Can't take address of a[0]
```

Hoặc dereference phần nào đó của mảng:

```
register int a[] = {11, 22, 33, 44, 55};

int a = *(a + 2);  // COMPILER ERROR! Address of a[0] taken
```

Thú vị là, với phiên bản tương đương dùng ký pháp mảng, gcc chỉ cảnh báo:

```
register int a[] = {11, 22, 33, 44, 55};

int a = a[2];  // COMPILER WARNING!
```

với:

```
warning: ISO C forbids subscripting ‘register’ array
```

Việc không lấy được địa chỉ biến register giải phóng compiler để nó có thể tối ưu quanh giả định đó, nếu nó chưa tự tìm ra rồi. Thêm `register` vào biến `const` còn ngăn ta vô tình truyền con trỏ của nó cho hàm khác sẵn sàng ngó lơ tính const[1].

[1] https://gustedt.wordpress.com/2010/08/17/a-common-misconsception-the-register-keyword/

Chút gốc gác lịch sử cho vui: sâu trong CPU có mấy "biến" nhỏ chuyên dụng gọi là *thanh ghi*[2]. Chúng truy cập siêu nhanh so với RAM, nên dùng chúng thì lên được ít tốc độ. Nhưng chúng không ở trong RAM, nên không có địa chỉ bộ nhớ gắn kèm (đó là lý do bạn không thể lấy địa chỉ của hay lấy con trỏ tới chúng).

Nhưng, như tôi nói, compiler hiện đại rất giỏi sinh mã tối ưu, dùng thanh ghi bất cứ khi nào có thể bất kể bạn có ghi từ khóa `register` hay không. Không những thế, spec còn cho phép chúng coi như bạn gõ `auto` nếu nó muốn. Nên không có đảm bảo gì.

16.2.5 `_Thread_local`

Khi bạn dùng nhiều luồng và bạn có vài biến ở global scope hoặc ở `static` block scope, đây là cách để mỗi luồng có bản sao riêng của biến. Nó sẽ giúp bạn tránh race condition và việc các luồng dẫm lên chân nhau.

Nếu bạn đang ở block scope, bạn phải dùng cái này cùng với `extern` hoặc `static`.

Ngoài ra, nếu bạn include `<threads.h>`, bạn có thể dùng `thread_local` dễ nuốt hơn, làm alias cho cái `_Thread_local` xấu xí.

Thông tin thêm có ở phần Threads.

[2]https://en.wikipedia.org/wiki/Processor_register

Chapter 17

Dự án nhiều file

Từ đầu đến giờ ta chỉ xem mấy chương trình đồ chơi mà phần lớn đều nhét vừa trong một file. Nhưng chương trình C phức tạp được tạo từ nhiều file, tất cả được biên dịch và link lại thành một file thực thi.

Chương này ta sẽ xem vài mẫu và lối làm thường gặp khi ghép các dự án lớn hơn lại với nhau.

17.1 Include và function prototype

Một tình huống rất phổ biến là vài hàm của bạn được định nghĩa trong một file, và bạn muốn gọi chúng từ file khác.

Chuyện này thực ra chạy được ngay với một cảnh báo, cứ thử trước rồi ta xem cách đúng để dẹp cảnh báo đó.

Để biên dịch, bạn cần chỉ định mọi file nguồn trên dòng lệnh:

```
# output file   source files
#     v            v
#   |----| |---------|
gcc -o foo foo.c bar.c
```

Trong ví dụ đó, `foo.c` và `bar.c` được build thành file thực thi tên `foo`.

Với mấy ví dụ này, ta để tên file như comment đầu tiên trong nguồn. Xem file nguồn `bar.c`:

```
// File bar.c

int add(int x, int y)
{
    return x + y;
}
```

Và file `foo.c` có main trong đó:

```
// File foo.c

#include <stdio.h>

int main(void)
{
    printf("%d\n", add(2, 3));  // 5!
}
```

Thấy cách từ `main()` ta gọi `add()` chứ, mà `add()` lại nằm trong một file nguồn hoàn toàn khác! Nó ở `bar.c`, còn lời gọi tới nó nằm trong `foo.c`!

Nếu build cái này bằng:

```
gcc -o foo foo.c bar.c
```

ta sẽ nhận được lỗi này:

```
error: implicit declaration of function 'add' is invalid in C99
```

(Hoặc bạn có thể nhận được cảnh báo. Mà thứ bạn không nên bỏ qua. Đừng bao giờ bỏ qua cảnh báo trong C, xử lý hết đi.)

Nếu bạn còn nhớ từ phần về prototype, khai báo ngầm bị cấm trong C hiện đại và không có lý do chính đáng nào để đưa chúng vào code mới. Ta nên sửa nó.

`implicit declaration` nghĩa là ta đang dùng một hàm, ở đây là `add()`, mà không cho C biết trước cái gì về nó cả. C muốn biết nó trả về gì, nhận kiểu gì làm đối số, và các thứ kiểu vậy.

Ta đã thấy cách sửa chuyện đó từ trước với *function prototype*. Đúng thế, nếu ta thêm một cái vào `foo.c` trước khi gọi, mọi thứ sẽ ổn:

```
// File foo.c

#include <stdio.h>

int add(int, int);  // Add the prototype

int main(void)
{
    printf("%d\n", add(2, 3));  // 5!
}
```

Hết lỗi!

Nhưng chuyện đó cũng nhọc, phải gõ prototype mỗi khi muốn dùng một hàm. Ủa kìa, ta vừa dùng `printf()` ngay đó mà đâu cần gõ prototype, vậy là sao?

Thật ra ta đã include prototype cho `printf()` rồi! Nó ở trong file `stdio.h`! Và ta đã include file đó bằng `#include`!

Ta làm tương tự với hàm `add()` của mình được không? Làm prototype cho nó và nhét vào một file header?

Dĩ nhiên được!

Header file trong C theo quy ước có phần mở rộng `.h`. Và chúng thường, dù không phải luôn luôn, có cùng tên với file `.c` tương ứng. Vậy ta tạo file `bar.h` cho file `bar.c`, và nhét prototype vào đó:

```
// File bar.h

int add(int, int);
```

Giờ sửa `foo.c` để include file đó. Giả sử nó ở cùng thư mục, ta include nó bên trong dấu nháy kép (thay vì dấu ngoặc nhọn):

```
// File foo.c

#include <stdio.h>
```

```
#include "bar.h"  // Include from current directory

int main(void)
{
    printf("%d\n", add(2, 3));  // 5!
}
```

Chú ý ta không còn prototype trong `foo.c` nữa, ta include nó từ `bar.h`. Giờ *bất cứ* file nào muốn dùng chức năng `add()` chỉ cần `#include "bar.h"` là có, không cần lo chuyện gõ prototype của hàm.

Như bạn có thể đoán, `#include` theo đúng nghĩa đen đưa file được gọi tên *ngay vào đó* trong mã nguồn của bạn, y như là bạn đã gõ vào.

Rồi build và chạy:

```
./foo
5
```

Đúng rồi, ta nhận được kết quả $2 + 3$! Hú hồn!

Nhưng đừng vội khui chai đồ uống yêu thích. Gần xong thôi! Còn một mẩu boilerplate nữa phải thêm.

17.2 Xử lý include bị lặp

Cũng không hiếm chuyện một file header lại `#include` các header khác cần cho chức năng của các file C tương ứng. Kiểu, sao không?

Và có thể bạn có một header được `#include` nhiều lần từ nhiều chỗ khác nhau. Có khi chẳng sao, có khi lại gây lỗi compiler. Và ta không kiểm soát được có bao nhiêu chỗ `#include` nó!

Tệ hơn, có khi ta rơi vào tình huống điên rồ kiểu header `a.h` include header `b.h`, và `b.h` lại include `a.h`! Đúng là chu kỳ `#include` vô hạn!

Thử build một thứ như vậy sẽ báo lỗi:

```
error: #include nested depth 200 exceeds maximum of 200
```

Biết đâu bước thứ 201 nó đã giải được chu kỳ...

Việc ta cần làm là nếu một file đã được include một lần rồi, các `#include` sau cho cùng file đó sẽ bị lờ đi.

Mấy thứ ta sắp làm phổ biến đến mức mà bạn cứ tự động làm mỗi lần tạo file header!

Và cách phổ biến để làm chuyện này là một biến preprocessor mà ta đặt vào lần đầu tiên `#include` file. Rồi với các `#include` sau, ta kiểm tra trước để chắc rằng biến đó chưa được định nghĩa.

Về tên biến, cực kỳ phổ biến việc lấy tên file header, như `bar.h`, viết hoa lên, và thay dấu chấm bằng gạch dưới: `BAR_H`.

Vậy đặt một kiểm tra ở sát đầu file xem nó đã được include chưa, và coi như comment cả file đi nếu rồi.

(Đừng đặt gạch dưới ở đầu (vì gạch dưới đầu theo sau là chữ hoa đã được reserved) hay hai gạch dưới ở đầu (vì cái đó cũng được reserved).)

```
#ifndef BAR_H   // If BAR_H isn't defined...
#define BAR_H   // Define it (with no particular value)

// File bar.h
```

```
int add(int, int);

#endif          // End of the #ifndef BAR_H
```

Cái này sẽ khiến file header chỉ được include đúng một lần, bất kể bao nhiêu chỗ cố `#include` nó.

17.3 `static` và `extern`

Bạn có thể tham chiếu đến các đối tượng ở file khác bằng `extern`.

Bạn có thể đảm bảo biến và hàm ở file scope *không* nhìn thấy được từ các file nguồn khác (dù có `extern`) bằng từ khóa `static`.

Thêm thông tin, xem các phần trong sách về storage-class specifier `static` và `extern`.

17.4 Biên dịch với object file

Chuyện này không có trong spec, nhưng nó là 99.999% phổ biến trong thế giới C.

Bạn có thể biên dịch file C thành dạng biểu diễn trung gian gọi là *object file*. Chúng chứa mã máy (tức là các bit 1 và 0 của các lệnh thực sự) nhưng chưa được ghép thành file thực thi.

Object file trong Windows có phần mở rộng `.OBJ`; trong các hệ Unix-like, chúng là `.o`.

Trong gcc, ta có thể build mấy cái đó thế này, với cờ `-c` (chỉ compile thôi!):

```
gcc -c foo.c     # produces foo.o
gcc -c bar.c     # produces bar.o
```

Rồi ta có thể *link* chúng lại thành một file thực thi duy nhất:

```
gcc -o foo foo.o bar.o
```

Voila, ta đã tạo ra file thực thi `foo` từ hai object file.

Nhưng bạn nghĩ, tội gì cho khổ? Chẳng phải ta có thể:

```
gcc -o foo foo.c bar.c
```

và hạ[1] hai con chim bằng một viên đá?

Với chương trình nhỏ thì ổn. Tôi vẫn làm vậy suốt.

Nhưng với chương trình lớn hơn, ta có thể tận dụng chuyện biên dịch từ nguồn ra object file thì tương đối chậm, còn link một đống object file lại thì tương đối nhanh.

Điều này thể hiện rõ nhất với công cụ `make`, thứ chỉ build lại những nguồn mới hơn output của chúng.

Giả sử bạn có một nghìn file C. Ban đầu bạn compile tất cả chúng thành object file (chậm) rồi gộp tất cả các object file đó thành file thực thi (nhanh).

Giờ giả sử bạn sửa đúng một trong số các file nguồn C đó, đây mới là phép màu: *bạn chỉ cần build lại đúng object file cho file nguồn đó*! Rồi build lại file thực thi (nhanh). Mọi file C khác không cần đụng tới.

Nói cách khác, nhờ chỉ build lại những object file cần, ta cắt giảm thời gian compile dữ dội. (Dĩ nhiên trừ khi bạn làm build "clean", khi đó tất cả object file đều phải được tạo lại.)

[1] https://en.wikipedia.org/wiki/Boids

Chapter 18

Môi trường bên ngoài

Khi bạn chạy một chương trình, thực ra là bạn đang nói chuyện với shell, kiểu: "Này, chạy giùm cái này với." Rồi shell nói: "Được," rồi nó bảo hệ điều hành: "Này, anh tạo tiến trình mới rồi chạy cái này giùm được không?" Và nếu mọi chuyện suôn sẻ, OS làm theo và chương trình của bạn chạy.

Nhưng ngoài chương trình, trong shell có cả một thế giới mà từ trong C có thể tương tác được. Ta sẽ ngó qua vài thứ trong chương này.

18.1 Tham số dòng lệnh

Nhiều tiện ích dòng lệnh nhận *tham số dòng lệnh*. Ví dụ, nếu ta muốn xem mọi file kết thúc bằng `.txt`, ta có thể gõ đại loại thế này trên hệ Unix-like:

```
ls *.txt
```

(hoặc `dir` thay cho `ls` trên hệ Windows).

Trong trường hợp này, lệnh là `ls`, nhưng tham số của nó là mọi file kết thúc bằng `.txt`[1].

Vậy làm sao để xem thứ gì được truyền vào chương trình từ dòng lệnh?

Giả sử ta có chương trình tên `add` cộng mọi số truyền trên dòng lệnh rồi in kết quả:

```
./add 10 30 5
45
```

Chắc chắn cái này sẽ kiếm đủ tiền trả hóa đơn đây!

Nhưng nghiêm túc, đây là công cụ hay để xem cách lấy tham số từ dòng lệnh rồi xử lý chúng.

Đầu tiên, xem cách lấy chúng ra đã. Cho chuyện này, ta cần một `main()` mới!

Đây là chương trình in ra tất cả tham số dòng lệnh. Ví dụ, nếu đặt tên file thực thi là `foo`, ta chạy thế này:

```
./foo i like turtles
```

và ta sẽ thấy output:

```
arg 0: ./foo
arg 1: i
```

[1]Về mặt lịch sử, chương trình trên MS-DOS và Windows làm chuyện này khác Unix. Ở Unix, shell sẽ *mở rộng* ký tự đại diện thành mọi file khớp trước khi chương trình của bạn thấy được, còn mấy bản của Microsoft sẽ chuyển cả biểu thức ký tự đại diện vào chương trình để tự xử. Dù sao, vẫn có tham số được chuyển vào chương trình.

```
arg 2: like
arg 3: turtles
```

Hơi lạ, vì tham số thứ không là tên file thực thi. Nhưng quen thôi. Các tham số còn lại thì theo sau trực tiếp.

Nguồn:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    for (int i = 0; i < argc; i++) {
        printf("arg %d: %s\n", i, argv[i]);
    }
}
```

Oái! Có chuyện gì với chữ ký của `main()` thế? `argc` và `argv` [2] (đọc là *arg-cee* và *arg-vee*) là gì vậy?

Bắt đầu với cái dễ trước: `argc`. Đây là *argument count*, tức số lượng tham số, bao gồm cả tên chương trình. Nếu bạn hình dung mọi tham số như một mảng chuỗi, mà chúng đúng là vậy, thì `argc` là độ dài của mảng đó, mà nó đúng là thế.

Và vậy chuyện ta làm trong vòng lặp là duyệt qua mọi `argv` và in từng cái một, nên với input:

```
./foo i like turtles
```

ta có output tương ứng:

```
arg 0: ./foo
arg 1: i
arg 2: like
arg 3: turtles
```

Với chừng đó trong đầu, ta đủ đồ để làm chương trình cộng.

Kế hoạch:

- Xem mọi tham số dòng lệnh (qua khỏi `argv[0]`, tên chương trình)
- Đổi chúng sang số nguyên
- Cộng dồn vào tổng đang chạy
- In kết quả

Bắt tay vào!

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

int main(int argc, char **argv)
{
    int total = 0;

    for (int i = 1; i < argc; i++) {  // Start at 1, the first argument
        int value = atoi(argv[i]);    // Use strtol() for better error handling

        total += value;
```

[2] Vì chúng chỉ là tên tham số thông thường, bạn không nhất thiết phải gọi là `argc` và `argv`. Nhưng đó là idiomatic đến mức nếu bạn sáng tạo, dev C khác sẽ nhìn bạn bằng ánh mắt nghi hoặc thật sự đấy!

```
    }

    printf("%d\n", total);
}
```

Vài lần chạy thử:

```
$ ./add
0
$ ./add 1
1
$ ./add 1 2
3
$ ./add 1 2 3
6
$ ./add 1 2 3 4
10
```

Dĩ nhiên nó có thể phun bậy nếu bạn truyền vào cái gì không phải số nguyên, nhưng việc làm cứng cáp chuyện đó xin để lại làm bài tập cho người đọc.

18.1.1 `argv` cuối cùng là `NULL`

Một điều vui vui về `argv` là sau chuỗi cuối cùng là một con trỏ tới `NULL`.

Tức là:

```
argv[argc] == NULL
```

luôn đúng!

Chuyện này có vẻ vô nghĩa, nhưng hóa ra lại hữu ích ở vài chỗ, ta sẽ xem một trong số đó ngay bây giờ.

18.1.2 Dạng thay thế: `char **argv`

Nhớ rằng khi gọi hàm, C không phân biệt ký pháp mảng và ký pháp con trỏ trong chữ ký hàm.

Tức là, hai thứ sau là như nhau:

```
void foo(char a[])
void foo(char *a)
```

Lâu nay ta hình dung `argv` như một mảng chuỗi, tức là một mảng các `char*`, nên cái này nghe hợp lý:

```
int main(int argc, char *argv[])
```

nhưng vì sự tương đương đó, bạn cũng có thể viết:

```
int main(int argc, char **argv)
```

Ừ, con trỏ trỏ tới con trỏ! Nếu thấy dễ hơn, cứ nghĩ nó như con trỏ tới chuỗi. Nhưng thực ra, nó là con trỏ trỏ tới một giá trị mà giá trị đó trỏ tới `char`.

Cũng nhớ rằng hai thứ này tương đương:

```
argv[i]
*(argv + i)
```

nghĩa là bạn có thể làm số học con trỏ trên `argv`.

Vậy một cách khác để tiêu thụ tham số dòng lệnh có thể là đi dọc mảng `argv` bằng cách tăng con trỏ lên cho tới khi chạm `NULL` ở cuối.

Sửa chương trình cộng của ta để làm vậy:

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

int main(int argc, char **argv)
{
    int total = 0;

    // Cute trick to get the compiler to stop warning about the
    // unused variable argc:
    (void)argc;

    for (char **p = argv + 1; *p != NULL; p++) {
        int value = atoi(*p);  // Use strtol() for better error handling

        total += value;
    }

    printf("%d\n", total);
}
```

Cá nhân tôi dùng ký pháp mảng để truy cập `argv`, nhưng tôi vẫn thấy kiểu này lảng vảng đâu đó.

18.1.3 Ít chuyện vui

Còn vài thứ về `argc` và `argv`.

- Vài môi trường có thể không đặt `argv[0]` là tên chương trình. Nếu nó không có, `argv[0]` sẽ là chuỗi rỗng. Tôi chưa bao giờ thấy chuyện này xảy ra.
- Spec thực ra khá thoáng trong chuyện implementation có thể làm gì với `argv` và các giá trị đó đến từ đâu. Nhưng mọi hệ tôi từng dùng đều hoạt động giống nhau, như ta đã bàn ở phần này.
- Bạn có thể sửa `argc`, `argv`, hoặc bất kỳ chuỗi nào mà `argv` trỏ tới. (Chỉ là đừng làm mấy chuỗi đó dài hơn kích thước sẵn có của nó!)
- Trên vài hệ Unix-like, sửa chuỗi `argv[0]` sẽ khiến output của `ps` thay đổi[3].

  Thông thường, nếu bạn có chương trình tên `foo` chạy bằng `./foo`, bạn có thể thấy trong output của `ps`:

  ```
  4078 tty1     S      0:00 ./foo
  ```

  Nhưng nếu sửa `argv[0]` thế này, cẩn thận để chuỗi mới `"Hi!  "` có cùng độ dài với chuỗi cũ `"./foo"`:

  ```
  strcpy(argv[0], "Hi!  ");
  ```

  rồi chạy `ps` khi chương trình `./foo` còn đang chạy, ta sẽ thấy:

  ```
  4079 tty1     S      0:00 Hi!
  ```

[3] `ps`, Process Status, là lệnh Unix để xem tiến trình nào đang chạy lúc đó.

Hành vi này không có trong spec và phụ thuộc rất nhiều vào hệ thống.

18.2 Exit status

Bạn có để ý chữ ký của `main()` trả về kiểu `int` không? Chuyện đó là sao? Nó liên quan tới thứ gọi là *exit status*, một số nguyên có thể được trả lại chương trình đã khởi chạy chương trình của bạn để báo mọi chuyện ra sao.

Có cả đống cách để chương trình thoát trong C, bao gồm `return` từ `main()`, hay gọi một trong các biến thể `exit()`.

Tất cả các cách này nhận `int` làm tham số.

Nhắc lề: bạn có thấy trong phần lớn các ví dụ của tôi, dù `main()` lẽ ra phải trả về `int`, tôi thật ra không `return` gì hết? Trong mọi hàm khác, chuyện này là bất hợp pháp, nhưng có một ngoại lệ đặc biệt trong C: nếu luồng thực thi chạm đến cuối `main()` mà không tìm thấy `return`, nó tự động làm `return 0`.

Nhưng `0` đó nghĩa là gì? Các số khác ta có thể đặt vào đó là gì? Và chúng được dùng ra sao?

Spec vừa rõ vừa mơ hồ về chuyện này, như thường lệ. Rõ vì nó nói ra bạn có thể làm gì, mơ hồ vì nó không giới hạn gì mấy.

Chẳng còn cách nào khác là *cứ tiến lên* và tìm ra!

Ta Inception[4] một chút: hóa ra khi bạn chạy chương trình, *bạn đang chạy nó từ một chương trình khác.*

Thường chương trình kia là kiểu shell[5] nào đó mà tự nó không làm mấy ngoài chuyện khởi chạy chương trình khác.

Nhưng đây là quy trình nhiều giai đoạn, nhất là với shell dòng lệnh thấy rõ:

1. Shell khởi chạy chương trình của bạn
2. Shell thường đi ngủ (với shell dòng lệnh)
3. Chương trình của bạn chạy
4. Chương trình kết thúc
5. Shell thức dậy và đợi lệnh tiếp theo

Bây giờ, có một mẩu thông tin liên lạc diễn ra giữa bước 4 và 5: chương trình có thể trả về một *giá trị trạng thái* mà shell có thể hỏi. Thường giá trị này được dùng để báo chương trình thành công hay thất bại, và nếu thất bại thì kiểu thất bại nào.

Giá trị này là cái ta vẫn `return` từ `main()`. Đó là status.

Giờ, spec C cho phép hai giá trị status khác nhau, có tên macro được định nghĩa trong `<stdlib.h>`:

Status	Mô tả
`EXIT_SUCCESS` hay `0`	Chương trình kết thúc thành công.
`EXIT_FAILURE`	Chương trình kết thúc với lỗi.

Hãy viết một chương trình ngắn nhân hai số từ dòng lệnh. Ta sẽ yêu cầu bạn chỉ định chính xác hai giá trị. Nếu không, ta in thông báo lỗi và thoát với status lỗi.

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

int main(int argc, char **argv)
{
    if (argc != 3) {
```

[4] https://en.wikipedia.org/wiki/Inception
[5] https://en.wikipedia.org/wiki/Shell_(computing)

```
        printf("usage: mult x y\n");
        return EXIT_FAILURE;   // Indicate to shell that it didn't work
    }

    printf("%d\n", atoi(argv[1]) * atoi(argv[2]));

    return 0;  // same as EXIT_SUCCESS, everything was good.
}
```

Giờ thử chạy, ta sẽ thấy đúng như ý cho đến khi truyền đúng số lượng tham số dòng lệnh:

```
$ ./mult
usage: mult x y

$ ./mult 3 4 5
usage: mult x y

$ ./mult 3 4
12
```

Nhưng cái đó không thật sự cho thấy exit status mà ta trả về, đúng không? Ta có thể bắt shell in nó ra. Giả sử bạn đang dùng Bash hoặc shell POSIX khác, có thể dùng `echo $?` để xem[6].

Thử xem:

```
$ ./mult
usage: mult x y
$ echo $?
1

$ ./mult 3 4 5
usage: mult x y
$ echo $?
1

$ ./mult 3 4
12
$ echo $?
0
```

Thú vị! Ta thấy trên hệ của tôi, `EXIT_FAILURE` là `1`. Spec không nói rõ chuyện này, nên nó có thể là số bất kỳ. Nhưng cứ thử, trên hệ của bạn chắc cũng là `1`.

18.2.1 Các giá trị exit status khác

Status `0` chắc chắn nghĩa là thành công, nhưng mọi số nguyên khác, kể cả âm, thì sao?

Ở đây ta đi ra khỏi spec C mà bước vào địa phận Unix. Nhìn chung, `0` là thành công, còn số dương khác không là thất bại. Vậy bạn chỉ có một kiểu thành công, nhưng nhiều kiểu thất bại. Bash nói exit code nên nằm trong khoảng 0 đến 255, dù một số code đã được reserved.

Nói ngắn, nếu muốn báo các exit status lỗi khác nhau trong môi trường Unix, bạn có thể bắt đầu từ `1` và tăng dần.

Trên Linux, nếu bạn thử code nào nằm ngoài khoảng 0-255, nó sẽ AND bitwise code với `0xff`, kẹp nó vào khoảng đó.

Bạn có thể script shell để dùng các code status này quyết định làm gì tiếp theo.

[6] Trên Windows `cmd.exe`, gõ `echo %errorlevel%`. Trong PowerShell, gõ `$LastExitCode`.

18.3 Biến môi trường

Trước khi đi vào phần này, tôi phải cảnh báo rằng C không định nghĩa biến môi trường là gì. Nên tôi sẽ mô tả hệ thống biến môi trường hoạt động trên mọi nền tảng lớn tôi biết.

Về cơ bản, môi trường là chương trình sẽ chạy chương trình của bạn, ví dụ shell bash. Và nó có thể có vài biến bash được định nghĩa. Nếu bạn chưa biết, shell có thể tự tạo biến riêng. Mỗi shell mỗi khác, nhưng trong bash bạn chỉ cần gõ `set` là nó hiện hết cho bạn.

Đây là trích đoạn từ 61 biến được định nghĩa trong bash shell của tôi:

```
HISTFILE=/home/beej/.bash_history
HISTFILESIZE=500
HISTSIZE=500
HOME=/home/beej
HOSTNAME=FBILAPTOP
HOSTTYPE=x86_64
IFS=$' \t\n'
```

Chú ý chúng ở dạng cặp key/value. Ví dụ, một key là `HOSTTYPE` và giá trị là `x86_64`. Từ góc nhìn C, tất cả giá trị là chuỗi, dù chúng là số[7].

Vậy *dù sao*! Chuyện dài kể ngắn, bạn có thể lấy các giá trị này từ bên trong chương trình C của bạn.

Viết chương trình dùng hàm chuẩn `getenv()` để tra một giá trị mà bạn đặt trong shell.

`getenv()` sẽ trả về con trỏ tới chuỗi giá trị, hoặc `NULL` nếu biến môi trường không tồn tại.

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

int main(void)
{
    char *val = getenv("FROTZ");  // Try to get the value

    // Check to make sure it exists
    if (val == NULL) {
        printf("Cannot find the FROTZ environment variable\n");
        return EXIT_FAILURE;
    }

    printf("Value: %s\n", val);
}
```

Nếu chạy trực tiếp, tôi thấy thế này:

```
$ ./foo
Cannot find the FROTZ environment variable
```

chuyện đó hợp lý, vì tôi chưa đặt nó.

Trong bash, tôi có thể đặt nó bằng[8]:

```
$ export FROTZ="C is awesome!"
```

Rồi khi chạy, tôi thấy:

[7]Nếu bạn cần giá trị số, chuyển chuỗi bằng thứ như `atoi()` hay `strtol()`.

[8]Trong Windows CMD.EXE, dùng `set FROTZ=value`. Trong PowerShell, dùng `$Env:FROTZ=value`.

```
$ ./foo
Value: C is awesome!
```

Theo cách này, bạn có thể đặt dữ liệu trong biến môi trường, và có thể lấy nó trong code C rồi thay đổi hành vi tương ứng.

18.3.1 Đặt biến môi trường

Cái này không chuẩn, nhưng nhiều hệ có cách để đặt biến môi trường.

Nếu trên hệ Unix-like, tra tài liệu cho `putenv()`, `setenv()`, và `unsetenv()`. Trên Windows, xem `_putenv()`.

18.3.2 Biến môi trường thay thế trên Unix-like

Nếu bạn đang trên hệ Unix-like, nhiều khả năng bạn có thêm vài cách nữa để truy cập biến môi trường. Lưu ý rằng dù spec chỉ ra đây là phần mở rộng phổ biến, nó không thật sự là phần của chuẩn C. Nhưng nó là phần của chuẩn POSIX.

Một trong số đó là biến tên `environ` phải được khai báo thế này:

```
extern char **environ;
```

Nó là một mảng chuỗi kết thúc bằng con trỏ `NULL`.

Bạn nên tự khai báo trước khi dùng, hoặc có thể tìm thấy nó trong file header không chuẩn `<unistd.h>`.

Mỗi chuỗi ở dạng `"key=value"` nên bạn phải tự tách rồi phân tích nếu muốn lấy key và value ra.

Đây là ví dụ lặp qua và in các biến môi trường bằng vài cách khác nhau:

```
#include <stdio.h>

extern char **environ;  // MUST be extern AND named "environ"

int main(void)
{
    for (char **p = environ; *p != NULL; p++) {
        printf("%s\n", *p);
    }

    // Or you could do this:
    for (int i = 0; environ[i] != NULL; i++) {
        printf("%s\n", environ[i]);
    }
}
```

Cho ra một đống output như thế này:

```
SHELL=/bin/bash
COLORTERM=truecolor
TERM_PROGRAM_VERSION=1.53.2
LOGNAME=beej
HOME=/home/beej
... etc ...
```

Dùng `getenv()` nếu có thể vì nó di động hơn. Nhưng nếu bạn phải lặp qua các biến môi trường, dùng `environ` có thể là cách hợp.

Một cách không chuẩn khác để lấy biến môi trường là làm tham số của `main()`. Nó hoạt động khá giống, nhưng bạn khỏi phải thêm biến `extern environ`. Ngay cả spec POSIX cũng không hỗ trợ cái này[9] theo tôi biết, nhưng nó phổ biến ở chốn Unix.

```
#include <stdio.h>

int main(int argc, char **argv, char **env)  // <-- env!
{
    (void)argc; (void)argv;  // Suppress unused warnings

    for (char **p = env; *p != NULL; p++) {
        printf("%s\n", *p);
    }

    // Or you could do this:
    for (int i = 0; env[i] != NULL; i++) {
        printf("%s\n", env[i]);
    }
}
```

Giống như dùng `environ` nhưng *còn kém di động hơn.* Có mục tiêu là tốt.

[9] https://pubs.opengroup.org/onlinepubs/9699919799/functions/exec.html

Chapter 19

C Preprocessor

Trước khi chương trình được biên dịch, nó thật ra chạy qua một pha gọi là *preprocessing*. Gần như là có một ngôn ngữ *nằm trên* ngôn ngữ C chạy trước. Và nó xuất ra code C, rồi code đó mới được biên dịch.

Ta đã thấy chuyện này phần nào với `#include`! Đó là C Preprocessor! Chỗ nào nó thấy directive đó, nó nhét file được gọi tên ngay vào, y như bạn đã gõ vào đó. Và *rồi* compiler mới build cả mớ.

Nhưng hóa ra nó mạnh hơn nhiều so với chỉ biết include. Bạn có thể định nghĩa *macro* để được thay thế... và thậm chí cả macro nhận tham số!

19.1 `#include`

Bắt đầu với cái ta đã thấy nhiều lần. Đây dĩ nhiên là một cách để include các nguồn khác vào nguồn của bạn. Rất hay dùng với file header.

Dù spec cho phép `#include` hành xử đủ kiểu, ta sẽ chọn cách thực dụng hơn và nói về cách nó hoạt động trên mọi hệ tôi từng gặp.

Ta có thể chia file header thành hai loại: hệ thống và nội bộ. Những thứ dựng sẵn như `stdio.h`, `stdlib.h`, `math.h` và v.v., bạn có thể include bằng ngoặc nhọn:

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

Ngoặc nhọn nói với C: "Này, đừng tìm file header này ở thư mục hiện tại, tìm trong thư mục include chung của hệ thống kìa."

Cái đó, dĩ nhiên, ngầm bảo rằng phải có cách để include file nội bộ từ thư mục hiện tại. Và có: dùng nháy kép:

```
#include "myheader.h"
```

Hoặc rất có thể bạn tìm được ở thư mục tương đối bằng dấu gạch chéo xuôi và dấu chấm, kiểu thế này:

```
#include "mydir/myheader.h"
#include "../someheader.py"
```

Đừng dùng dấu gạch chéo ngược (`\`) làm dấu ngăn đường dẫn trong `#include`! Đó là undefined behavior! Chỉ dùng gạch chéo xuôi (`/`), ngay cả trên Windows.

Tóm lại, dùng ngoặc nhọn (`<` và `>`) cho include hệ thống, và dùng nháy kép (`"`) cho include cá nhân của bạn.

19.2 Macro đơn giản

Macro là một định danh được *mở rộng* thành một mẩu code khác trước khi compiler thấy được. Hãy coi nó như một chỗ trống, khi preprocessor thấy một trong các định danh đó, nó thay bằng giá trị mà bạn đã định nghĩa.

Ta làm chuyện này bằng `#define` (thường đọc là "pound define", hay "hash define", và hiếm khi, nếu có, đọc là "octothorpe define"). Ví dụ:

```
#include <stdio.h>

#define HELLO "Hello, world"
#define PI 3.14159

int main(void)
{
    printf("%s, %f\n", HELLO, PI);
}
```

Ở dòng 3 và 4 ta định nghĩa vài macro. Ở đâu mà chúng xuất hiện trong code (dòng 8), chúng sẽ được thay bằng giá trị đã định nghĩa.

Từ góc nhìn của compiler C, y hệt như ta đã viết thế này:

```
#include <stdio.h>

int main(void)
{
    printf("%s, %f\n", "Hello, world", 3.14159);
}
```

Thấy cách `HELLO` được thay bằng `"Hello, world"` và `PI` được thay bằng `3.14159` chứ? Từ góc nhìn compiler, giống như những giá trị đó đã có mặt ngay trong code.

Lưu ý rằng macro không có kiểu cụ thể, *tự thân nó*. Thật ra chỉ là chúng được thay nguyên xi bằng bất cứ thứ gì được `#define` thành. Nếu code C kết quả không hợp lệ, compiler sẽ phun bậy ra.

Bạn cũng có thể định nghĩa macro không có giá trị:

```
#define EXTRA_HAPPY
```

trong trường hợp đó, macro tồn tại và được định nghĩa, nhưng định nghĩa là không có gì. Nên chỗ nào nó xuất hiện trong văn bản cũng sẽ được thay bằng không có gì. Ta sẽ thấy cách dùng của cái này sau.

Theo quy ước, tên macro viết `ALL_CAPS` dù kỹ thuật thì không bắt buộc.

Nhìn chung, cái này cho bạn cách định nghĩa giá trị hằng gần như toàn cục và dùng được *bất cứ* chỗ nào. Kể cả chỗ mà biến `const` không dùng được, ví dụ trong `case` của `switch` và độ dài mảng cố định.

Dù vậy, cuộc tranh cãi vẫn nổ ra trên mạng về việc dùng biến `const` có kiểu có tốt hơn macro `#define` trong trường hợp chung không.

Nó cũng có thể được dùng để thay thế hoặc sửa các từ khóa, một khái niệm hoàn toàn xa lạ với `const`, dù chuyện này nên dùng tiết kiệm.

19.3 Biên dịch có điều kiện

Có thể bắt preprocessor quyết định có đưa một số block code cho compiler hay không, hoặc xóa chúng hẳn đi trước khi compile.

Ta làm chuyện đó bằng cách bọc code trong các block điều kiện, giống như câu lệnh `if` - `else`.

19.3.1 Nếu đã định nghĩa, `#ifdef` và `#endif`

Đầu tiên, thử compile code cụ thể tùy thuộc vào macro có được định nghĩa hay không.

```
#include <stdio.h>

#define EXTRA_HAPPY

int main(void)
{

#ifdef EXTRA_HAPPY
    printf("I'm extra happy!\n");
#endif

    printf("OK!\n");
}
```

Trong ví dụ đó, ta định nghĩa `EXTRA_HAPPY` (thành không gì cả, nhưng nó *được* định nghĩa), rồi ở dòng 8 ta kiểm tra xem nó có được định nghĩa hay không bằng directive `#ifdef`. Nếu có, code tiếp sau đó sẽ được đưa vào cho đến `#endif`.

Vì nó được định nghĩa, code sẽ được đưa vào để compile và output sẽ là:

```
I'm extra happy!
OK!
```

Nếu ta comment cái `#define` đi, kiểu:

```
//#define EXTRA_HAPPY
```

thì nó sẽ không được định nghĩa, và code sẽ không được đưa vào compile. Và output sẽ chỉ là:

```
OK!
```

Quan trọng là nhớ rằng những quyết định này xảy ra lúc compile! Code thực sự được compile hoặc bị bỏ đi tùy vào điều kiện. Chuyện này khác với lệnh `if` tiêu chuẩn được đánh giá lúc chương trình chạy.

19.3.2 Nếu chưa định nghĩa, `#ifndef`

Cũng có nghĩa phủ định của "nếu đã định nghĩa": "nếu chưa định nghĩa", hay `#ifndef`. Ta có thể sửa ví dụ trước để in ra thứ khác dựa trên việc một thứ có được định nghĩa hay không:

```
#ifdef EXTRA_HAPPY
    printf("I'm extra happy!\n");
#endif

#ifndef EXTRA_HAPPY
    printf("I'm just regular\n");
#endif
```

Ta sẽ thấy cách gọn hơn để làm ở phần sau.

Nối lại với file header, ta đã thấy cách khiến file header chỉ được include đúng một lần bằng cách bọc chúng trong directive preprocessor thế này:

```
#ifndef MYHEADER_H  // First line of myheader.h
#define MYHEADER_H

int x = 12;

#endif  // Last line of myheader.h
```

Cái này cho thấy cách một macro tồn tại qua các file và qua nhiều `#include`. Nếu chưa được định nghĩa, ta định nghĩa nó rồi compile cả file header.

Nhưng lần sau khi nó được include, ta thấy `MYHEADER_H` *đã* được định nghĩa, nên ta không gửi file header cho compiler nữa, nó bị xóa hẳn đi.

19.3.3 `#else`

Nhưng chưa phải tất cả! Còn có `#else` mà ta có thể quẳng vào mớ đó.

Sửa ví dụ trước:

```
#ifdef EXTRA_HAPPY
    printf("I'm extra happy!\n");
#else
    printf("I'm just regular\n");
#endif
```

Giờ nếu `EXTRA_HAPPY` chưa được định nghĩa, nó sẽ trúng nhánh `#else` và in:

```
I'm just regular
```

19.3.4 Else-If: `#elifdef`, `#elifndef`

Tính năng này mới có ở C23!

Thế nếu bạn muốn cái gì phức tạp hơn? Có thể bạn cần cấu trúc if-else nối tầng để code được build đúng?

May là ta có mấy directive này để xài. Ta có thể dùng `#elifdef` cho "else if defined":

```
#ifdef MODE_1
    printf("This is mode 1\n");
#elifdef MODE_2
    printf("This is mode 2\n");
#elifdef MODE_3
    printf("This is mode 3\n");
#else
    printf("This is some other mode\n");
#endif
```

Ngược lại, bạn có thể dùng `#elifndef` cho "else if not defined".

19.3.5 Điều kiện tổng quát: `#if`, `#elif`

Cái này hoạt động rất giống `#ifdef` và `#ifndef` ở chỗ bạn cũng có thể có `#else` và cả mớ được khép lại bằng `#endif`.

Khác biệt duy nhất là biểu thức hằng sau `#if` phải tính ra true (khác không) thì code trong `#if` mới được compile. Nên thay vì việc một thứ có được định nghĩa hay chưa, ta muốn một biểu thức tính ra true.

```
#include <stdio.h>

#define HAPPY_FACTOR 1

int main(void)
{

#if HAPPY_FACTOR == 0
    printf("I'm not happy!\n");
#elif HAPPY_FACTOR == 1
    printf("I'm just regular\n");
#else
    printf("I'm extra happy!\n");
#endif

    printf("OK!\n");
}
```

Lại một lần nữa, với các nhánh `#if` không khớp, compiler sẽ không nhìn thấy những dòng đó. Với code trên, sau khi preprocessor làm xong, tất cả compiler thấy chỉ là:

```
#include <stdio.h>

int main(void)
{

    printf("I'm just regular\n");

    printf("OK!\n");
}
```

Một cách dùng hơi hacker là để comment nhanh một mớ dòng[1].

Nếu bạn đặt `#if 0` ("if false") ở đầu block cần comment và `#endif` ở cuối, bạn được hiệu quả này:

```
#if 0
    printf("All this code"); /* is effectively */
    printf("commented out"); // by the #if 0
#endif
```

Thế nếu bạn trên compiler cũ hơn C23 và không có hỗ trợ directive `#elifdef` hay `#elifndef` thì sao? Làm sao đạt cùng hiệu quả với `#if`? Ví dụ, nếu tôi muốn cái này:

```
#ifdef FOO
    x = 2;
#elifdef BAR  // POTENTIAL ERROR: Not supported before C23
    x = 3;
#endif
```

Làm sao làm được?

Hóa ra có một toán tử preprocessor tên `defined` mà ta có thể dùng với lệnh `#if`.

Các thứ sau là tương đương:

[1] Bạn không thể lúc nào cũng bọc code trong comment `/*` `*/` vì chúng không lồng được.

```
#ifdef FOO
#if defined FOO
#if defined(FOO)   // Parentheses optional
```

Các thứ sau cũng vậy:

```
#ifndef FOO
#if !defined FOO
#if !defined(FOO)   // Parentheses optional
```

Chú ý cách ta dùng toán tử NOT logic tiêu chuẩn (`!`) cho "chưa định nghĩa".

Vậy giờ ta quay lại vùng đất `#if` và có thể dùng `#elif` mà không phải sợ gì!

Đoạn code hỏng này:

```
#ifdef FOO
    x = 2;
#elifdef BAR  // POTENTIAL ERROR: Not supported before C23
    x = 3;
#endif
```

có thể được thay bằng:

```
#if defined FOO
    x = 2;
#elif defined BAR
    x = 3;
#endif
```

19.3.6 Vứt macro đi: `#undef`

Nếu bạn đã định nghĩa gì đó nhưng không cần nữa, bạn có thể bỏ định nghĩa bằng `#undef` .

```
#include <stdio.h>

int main(void)
{
#define GOATS

#ifdef GOATS
    printf("Goats detected!\n");  // prints
#endif

#undef GOATS  // Make GOATS no longer defined

#ifdef GOATS
    printf("Goats detected, again!\n"); // doesn't print
#endif
}
```

19.4 Macro dựng sẵn

Chuẩn định nghĩa một đống macro dựng sẵn mà bạn có thể kiểm tra và dùng cho biên dịch có điều kiện. Xem qua ở đây.

19.4.1 Macro bắt buộc

Tất cả những cái này đều được định nghĩa:

Macro	Mô tả
`__DATE__`	Ngày compile, kiểu lúc bạn đang compile file này, ở định dạng `Mmm dd yyyy`
`__TIME__`	Giờ compile ở định dạng `hh:mm:ss`
`__FILE__`	Một chuỗi chứa tên file này
`__LINE__`	Số dòng của file mà macro này xuất hiện ở đó
`__func__`	Tên hàm mà cái này xuất hiện trong, dưới dạng chuỗi[2]
`__STDC__`	Được định nghĩa bằng `1` nếu đây là compiler C chuẩn
`__STDC_HOSTED__`	Sẽ là `1` nếu compiler là *hosted implementation*[3], ngược lại `0`
`__STDC_VERSION__`	Phiên bản của C, một hằng `long int` ở dạng `yyyymmL`, ví dụ `201710L`

Ghép chúng lại.

```
#include <stdio.h>

int main(void)
{
    printf("This function: %s\n", __func__);
    printf("This file: %s\n", __FILE__);
    printf("This line: %d\n", __LINE__);
    printf("Compiled on: %s %s\n", __DATE__, __TIME__);
    printf("C Version: %ld\n", __STDC_VERSION__);
}
```

Output trên hệ của tôi là:

```
This function: main
This file: foo.c
This line: 7
Compiled on: Nov 23 2020 17:16:27
C Version: 201710
```

`__FILE__`, `__func__` và `__LINE__` đặc biệt hữu ích để báo tình trạng lỗi trong thông điệp cho dev. Macro `assert()` trong `<assert.h>` dùng chúng để chỉ ra chỗ nào trong code assertion thất bại.

19.4.1.1 `__STDC_VERSION__`

Nếu bạn tò mò, đây là các số phiên bản cho những bản phát hành lớn khác nhau của Spec Ngôn ngữ C:

Release	Phiên bản ISO/IEC	`__STDC_VERSION__`
C89	ISO/IEC 9899:1990	không định nghĩa
C89	ISO/IEC 9899:1990/Amd.1:1995	`199409L`
C99	ISO/IEC 9899:1999	`199901L`
C11	ISO/IEC 9899:2011/Amd.1:2012	`201112L`

Chú ý macro này ban đầu không tồn tại trong C89.

[2] Đây không hẳn là macro, về kỹ thuật thì nó là một định danh. Nhưng nó là định danh được định nghĩa trước duy nhất và cảm giác rất giống macro, nên tôi để nó ở đây. Kiểu nổi loạn tí.

[3] Hosted implementation về cơ bản nghĩa là bạn đang chạy chuẩn C đầy đủ, có lẽ trên một hệ điều hành nào đó. Mà chắc là đúng thế. Nếu bạn đang chạy trên phần cứng trần trong kiểu hệ embedded, bạn chắc đang trên *standalone implementation*.

Cũng chú ý rằng kế hoạch là số phiên bản sẽ tăng nghiêm ngặt, nên bạn luôn có thể kiểm tra, chẳng hạn "ít nhất C99" bằng:

```
#if __STDC_VERSION__ >= 1999901L
```

19.4.2 Macro tùy chọn

Implementation của bạn có thể cũng định nghĩa những cái này. Hoặc không.

Macro	Mô tả
`__STDC_ISO_10646__`	Nếu được định nghĩa, `wchar_t` chứa giá trị Unicode, ngược lại thì chứa gì khác
`__STDC_MB_MIGHT_NEQ_WC__`	Giá trị `1` báo rằng các giá trị trong ký tự đa byte có thể không ánh xạ bằng với giá trị trong ký tự wide
`__STDC_UTF_16__`	Giá trị `1` báo rằng hệ thống dùng mã hóa UTF-16 trong kiểu `char16_t`
`__STDC_UTF_32__`	Giá trị `1` báo rằng hệ thống dùng mã hóa UTF-32 trong kiểu `char32_t`
`__STDC_ANALYZABLE__`	Giá trị `1` báo code có thể phân tích được[4]
`__STDC_IEC_559__`	`1` nếu hỗ trợ floating point IEEE-754 (còn gọi là IEC 60559)
`__STDC_IEC_559_COMPLEX__`	`1` nếu hỗ trợ floating point phức IEC 60559
`__STDC_LIB_EXT1__`	`1` nếu implementation này hỗ trợ nhiều hàm thư viện chuẩn "an toàn" thay thế (chúng có hậu tố `_s` trong tên)
`__STDC_NO_ATOMICS__`	`1` nếu implementation này **không** hỗ trợ `_Atomic` hay `<stdatomic.h>`
`__STDC_NO_COMPLEX__`	`1` nếu implementation này **không** hỗ trợ kiểu complex hay `<complex.h>`
`__STDC_NO_THREADS__`	`1` nếu implementation này **không** hỗ trợ `<threads.h>`
`__STDC_NO_VLA__`	`1` nếu implementation này **không** hỗ trợ mảng có độ dài thay đổi

19.5 Macro có tham số

Macro mạnh hơn là chỉ thay thế đơn giản. Bạn có thể cài chúng nhận tham số để thay vào.

Câu hỏi thường nảy ra là khi nào dùng macro có tham số thay cho hàm. Trả lời ngắn: dùng hàm. Nhưng bạn sẽ thấy vô số macro ngoài đời và trong thư viện chuẩn. Người ta hay dùng chúng cho mấy thứ ngắn, tính toán, và cho các tính năng có thể thay đổi tùy nền tảng. Bạn có thể định nghĩa các từ khóa khác nhau cho nền tảng này hay nền tảng khác.

19.5.1 Macro có một tham số

Bắt đầu với cái đơn giản bình phương một số:

```
#include <stdio.h>

#define SQR(x) x * x  // Not quite right, but bear with me

int main(void)
{
    printf("%d\n", SQR(12));  // 144
}
```

Cái đó nói "ở đâu thấy `SQR` với giá trị nào đó, thay bằng giá trị đó nhân với chính nó".

Nên dòng 7 sẽ đổi thành:

[4] Được, tôi biết đó là câu trả lời né tránh. Về cơ bản có một phần mở rộng tùy chọn mà compiler có thể cài, trong đó chúng đồng ý giới hạn vài kiểu undefined behavior để code C dễ làm static code analysis hơn. Ít khả năng bạn cần dùng cái này.

```
    printf("%d\n", 12 * 12);  // 144
```

C thoải mái chuyển thành 144.

Nhưng ta đã phạm lỗi sơ đẳng trong macro đó, lỗi mà ta cần tránh.

Cùng xem. Nếu ta muốn tính `SQR(3 + 4)`? Ừ, $3 + 4 = 7$, nên ta chắc muốn tính $7^2 = 49$. Đấy, `49`, câu trả lời cuối cùng.

Cho vào code và ta được... 19?

```
    printf("%d\n", SQR(3 + 4));  // 19!!??
```

Có chuyện gì?

Nếu ta theo dõi mở rộng macro, ta có

```
    printf("%d\n", 3 + 4 * 3 + 4);  // 19!
```

Ối! Vì nhân có độ ưu tiên cao hơn, ta làm $4 \times 3 = 12$ trước, rồi được $3 + 12 + 4 = 19$. Không phải thứ ta muốn.

Nên ta phải sửa cho nó đúng.

Cái này phổ biến tới mức cứ tự động làm mỗi khi bạn tạo macro tính toán có tham số!

Sửa thì dễ: thêm vài cặp ngoặc đơn!

```
#define SQR(x) (x) * (x)   // Better... but still not quite good enough!
```

Và giờ macro của ta mở rộng thành:

```
    printf("%d\n", (3 + 4) * (3 + 4));  // 49! Woo hoo!
```

Nhưng thật ra ta vẫn còn cùng vấn đề, có thể thò ra nếu quanh đó có toán tử ưu tiên cao hơn nhân (`*`).

Nên cách an toàn, đúng bài để ghép macro là bọc cả cục trong thêm cặp ngoặc đơn, thế này:

```
#define SQR(x) ((x) * (x))   // Good!
```

Cứ biến nó thành thói quen khi tạo macro tính toán là không sai được.

19.5.2 Macro có nhiều hơn một tham số

Bạn có thể xếp chồng mấy thứ này lên bao nhiêu tùy ý:

```
#define TRIANGLE_AREA(w, h) (0.5 * (w) * (h))
```

Làm vài macro giải x dùng công thức nghiệm bậc hai. Phòng khi bạn không còn nhớ nằm lòng, với phương trình dạng:

$ax^2 + bx + c = 0$

bạn có thể giải x bằng công thức nghiệm bậc hai:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Cái đó điên rồ. Để ý cả cái dấu cộng-trừ ($\pm$) trong đó, báo rằng thật ra có hai nghiệm.

Nên ta làm macro cho cả hai:

```
#define QUADP(a, b, c) ((-(b) + sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
#define QUADM(a, b, c) ((-(b) - sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
```

Vậy là có được tí tính toán. Nhưng ta định nghĩa thêm một cái nữa để dùng làm tham số cho `printf()` in cả hai đáp án.

```
//          macro              replacement
//      |-----------| |---------------------------|
#define QUAD(a, b, c) QUADP(a, b, c), QUADM(a, b, c)
```

Đó chỉ là vài giá trị ngăn cách bởi dấu phẩy, và ta có thể dùng cái đó như tham số "ghép" của `printf()` thế này:

```
printf("x = %f or x = %f\n", QUAD(2, 10, 5));
```

Ghép lại thành code:

```
#include <stdio.h>
#include <math.h>  // For sqrt()

#define QUADP(a, b, c) ((-(b) + sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
#define QUADM(a, b, c) ((-(b) - sqrt((b) * (b) - 4 * (a) * (c))) / (2 * (a)))
#define QUAD(a, b, c) QUADP(a, b, c), QUADM(a, b, c)

int main(void)
{
    printf("2*x^2 + 10*x + 5 = 0\n");
    printf("x = %f or x = %f\n", QUAD(2, 10, 5));
}
```

Và cái này cho output:

```
2*x^2 + 10*x + 5 = 0
x = -0.563508 or x = -4.436492
```

Thế giá trị nào vào cũng cho xấp xỉ không (hơi lệch vì các số không chính xác):

$2 \times -0.563508^2 + 10 \times -0.563508 + 5 \approx 0.000003$

19.5.3 Macro với tham số biến

Cũng có cách truyền số lượng tham số biến đổi vào macro, dùng dấu ba chấm (`...`) sau các tham số có tên đã biết. Khi macro mở rộng, mọi tham số thừa sẽ ở trong danh sách ngăn bởi dấu phẩy trong macro `__VA_ARGS__`, và có thể được thay từ đó:

```
#include <stdio.h>

// Combine the first two arguments to a single number,
// then have a commalist of the rest of them:

#define X(a, b, ...) (10*(a) + 20*(b)), __VA_ARGS__

int main(void)
{
    printf("%d %f %s %d\n", X(5, 4, 3.14, "Hi!", 12));
}
```

Việc thay thế diễn ra ở dòng 10 sẽ là:

```
    printf("%d %f %s %d\n", (10*(5) + 20*(4)), 3.14, "Hi!", 12);
```

cho output:

```
130 3.140000 Hi! 12
```

Bạn cũng có thể "stringify" `__VA_ARGS__` bằng cách đặt `#` ở trước:

```
#define X(...) #__VA_ARGS__

printf("%s\n", X(1,2,3));  // Prints "1, 2, 3"
```

19.5.4 Stringification

Như vừa nhắc ở trên, bạn có thể biến tham số thành chuỗi bằng cách đặt `#` phía trước nó trong phần thay thế.

Ví dụ, ta có thể in bất cứ thứ gì dưới dạng chuỗi với macro này và `printf()`:

```
#define STR(x) #x

printf("%s\n", STR(3.14159));
```

Trong trường hợp đó, phép thay thế dẫn đến:

```
printf("%s\n", "3.14159");
```

Xem ta có thể dùng cái này hiệu quả hơn không bằng cách truyền bất kỳ tên biến `int` nào vào macro, rồi cho nó in ra tên và giá trị.

```
#include <stdio.h>

#define PRINT_INT_VAL(x) printf("%s = %d\n", #x, x)

int main(void)
{
    int a = 5;

    PRINT_INT_VAL(a);  // prints "a = 5"
}
```

Ở dòng 9, ta có phép thay thế macro sau:

```
    printf("%s = %d\n", "a", a);
```

19.5.5 Nối chuỗi

Ta cũng có thể nối hai tham số với nhau bằng `##`. Vui quá đi!

```
#define CAT(a, b) a ## b

printf("%f\n", CAT(3.14, 1592));   // 3.141592
```

19.6 Macro nhiều dòng

Có thể kéo dài macro qua nhiều dòng nếu bạn escape ký tự xuống dòng bằng gạch chéo ngược (`\`).

Hãy viết một macro nhiều dòng in các số từ `0` đến tích của hai tham số truyền vào.

```
#include <stdio.h>

#define PRINT_NUMS_TO_PRODUCT(a, b) do { \
    int product = (a) * (b); \
    for (int i = 0; i < product; i++) { \
        printf("%d\n", i); \
    } \
} while(0)

int main(void)
{
    PRINT_NUMS_TO_PRODUCT(2, 4);  // Outputs numbers from 0 to 7
}
```

Vài điều chú ý:

- Escape ở cuối mỗi dòng trừ dòng cuối để báo macro còn tiếp.
- Cả mớ được bọc trong vòng `do` - `while(0)` với ngoặc xoắn xoắn xuýt.

Điểm thứ hai có thể hơi lạ, nhưng tất cả là để nuốt dấu `;` mà coder quăng vào sau macro.

Lúc đầu tôi tưởng chỉ cần ngoặc xoắn là đủ, nhưng có một tình huống nó hỏng nếu coder đặt dấu chấm phẩy sau macro. Đây là tình huống đó:

```
#include <stdio.h>

#define FOO(x) { (x)++; }

int main(void)
{
    int i = 0;

    if (i == 0)
        FOO(i);
    else
        printf(":-(\n");

    printf("%d\n", i);
}
```

Trông có vẻ đơn giản, nhưng nó không build được vì lỗi cú pháp:

```
foo.c:11:5: error: ‘else’ without a previous ‘if’
```

Bạn thấy chưa?

Xem đoạn mở rộng:

```
    if (i == 0) {
        (i)++;
    };              // <-- Trouble with a capital-T!

    else
```

```
        printf(":-(\n");
```

Dấu `;` kết thúc câu lệnh `if`, nên `else` cứ lơ lửng ngoài đó một cách bất hợp pháp[5].

Nên hãy bọc macro nhiều dòng đó trong `do` - `while(0)`.

19.7 Ví dụ: Macro Assert

Thêm assert vào code là cách hay để bắt các tình trạng mà bạn nghĩ không nên xảy ra. C có sẵn chức năng `assert()`. Nó kiểm tra một điều kiện, và nếu là false, chương trình nổ tung báo cho bạn biết file và số dòng mà assertion thất bại.

Nhưng cái này còn thiếu.

1. Trước hết, bạn không thể đặc tả thêm thông điệp đi kèm assert.
2. Thứ hai, không có công tắc bật/tắt dễ dàng cho toàn bộ assert.

Ta có thể giải quyết cái đầu bằng macro.

Về cơ bản, khi tôi có code này:

```
ASSERT(x < 20, "x must be under 20");
```

Tôi muốn cái gì đó như thế này xảy ra (giả sử `ASSERT()` ở dòng 220 của `foo.c`):

```
if (!(x < 20)) {
    fprintf(stderr, "foo.c:220: assertion x < 20 failed: ");
    fprintf(stderr, "x must be under 20\n");
    exit(1);
}
```

Ta có thể lấy tên file từ macro `__FILE__`, và số dòng từ `__LINE__`. Thông điệp đã là chuỗi, còn `x < 20` thì không, nên ta phải stringify nó bằng `#`. Ta có thể làm macro nhiều dòng bằng cách escape gạch chéo ngược ở cuối dòng.

```
#define ASSERT(c, m) \
do { \
    if (!(c)) { \
        fprintf(stderr, __FILE__ ":%d: assertion %s failed: %s\n", \
                        __LINE__, #c, m); \
        exit(1); \
    } \
} while(0)
```

(Nó trông hơi lạ với `__FILE__` ở đầu như vậy, nhưng nhớ rằng nó là string literal, và các string literal nằm cạnh nhau sẽ tự động nối lại. `__LINE__` thì ngược lại, nó chỉ là `int`.)

Và nó chạy! Nếu tôi chạy cái này:

```
int x = 30;

ASSERT(x < 20, "x must be under 20");
```

Tôi được output này:

```
foo.c:23: assertion x < 20 failed: x must be under 20
```

[5] *Breakin' the law... breakin' the law...*

Ngon lành!

Cái còn lại là cách bật/tắt, và ta có thể làm với biên dịch có điều kiện.

Đây là ví dụ hoàn chỉnh:

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

#define ASSERT_ENABLED 1

#if ASSERT_ENABLED
#define ASSERT(c, m) \
do { \
    if (!(c)) { \
        fprintf(stderr, __FILE__ ":%d: assertion %s failed: %s\n", \
                        __LINE__, #c, m); \
        exit(1); \
    } \
} while(0)
#else
#define ASSERT(c, m)  // Empty macro if not enabled
#endif

int main(void)
{
    int x = 30;

    ASSERT(x < 20, "x must be under 20");
}
```

Cái này có output:

```
foo.c:23: assertion x < 20 failed: x must be under 20
```

19.8 Directive `#error`

Directive này bắt compiler báo lỗi ngay khi nó thấy.

Thường thì dùng trong một điều kiện để ngăn biên dịch trừ khi vài điều kiện tiên quyết được thỏa mãn:

```
#ifndef __STDC_IEC_559__
    #error I really need IEEE-754 floating point to compile. Sorry!
#endif
```

Vài compiler có directive `#warning` không chuẩn bổ sung sẽ xuất cảnh báo nhưng không dừng biên dịch, nhưng cái này không có trong spec C11.

19.9 Directive `#embed`

Mới ở C23!

Và hiện chưa chạy với bất kỳ compiler nào của tôi, nên hãy đọc phần này với một hạt muối!

Ý là bạn có thể include các byte của một file dưới dạng hằng số nguyên y như là bạn đã gõ chúng vào.

Ví dụ, nếu bạn có file nhị phân tên `foo.bin` chứa bốn byte với giá trị thập phân 11, 22, 33, và 44, và bạn làm cái này:

```
int a[] = {
#embed "foo.bin"
};
```

Thì y như bạn đã gõ:

```
int a[] = {11,22,33,44};
```

Đây là cách rất mạnh để khởi tạo mảng với dữ liệu nhị phân mà không cần chuyển tất cả sang code trước, preprocessor làm giùm bạn!

Trường hợp dùng điển hình hơn có thể là một file chứa hình ảnh nhỏ để hiển thị mà bạn không muốn nạp lúc runtime.

Đây là ví dụ khác:

```
int a[] = {
#embed <foo.bin>
};
```

Nếu bạn dùng ngoặc nhọn, preprocessor tìm trong một loạt vị trí do implementation định nghĩa để tìm file, giống như `#include` làm. Nếu bạn dùng nháy kép mà tài nguyên không tìm thấy, compiler sẽ thử như là bạn đã dùng ngoặc nhọn, như một nỗ lực tuyệt vọng cuối cùng để tìm file.

`#embed` hoạt động như `#include` ở chỗ nó dán giá trị vào trước khi compiler thấy. Tức là bạn có thể dùng nó ở đủ loại chỗ:

```
return
#embed "somevalue.dat"
;
```

hoặc

```
int x =
#embed "xvalue.dat"
;
```

Giờ, đây có luôn là byte không? Nghĩa là giá trị sẽ từ 0 đến 255, bao gồm cả? Câu trả lời mặc định chắc chắn là "có", trừ khi là "không".

Về kỹ thuật, các phần tử sẽ rộng `CHAR_BIT` bit. Và rất có thể là 8 trên hệ của bạn, nên bạn sẽ có khoảng 0-255 trong các giá trị. (Chúng sẽ luôn không âm.)

Cũng có thể một implementation cho phép ghi đè cái này theo cách nào đó, ví dụ trên dòng lệnh hay với tham số.

Kích thước file tính bằng bit phải là bội của kích thước phần tử. Tức là, nếu mỗi phần tử là 8 bit, kích thước file (tính bằng bit) phải là bội của 8. Trong sử dụng hằng ngày, đây là cách nói vòng vo rằng mỗi file phải là số nguyên lần byte... dĩ nhiên là vậy rồi. Thành thật mà nói, tôi cũng chẳng biết sao mình lại bận tâm viết đoạn này. Đọc spec nếu bạn thật sự tò mò.

19.9.1 Tham số cho `#embed`

Có đủ loại tham số bạn có thể chỉ định cho directive `#embed`. Đây là ví dụ với tham số `limit()` còn chưa giới thiệu:

```
int a[] = {
#embed "/dev/random" limit(5)
};
```

Nhưng nếu bạn đã định nghĩa `limit` ở chỗ khác thì sao? May là bạn có thể đặt `__` quanh từ khóa và nó sẽ hoạt động y vậy:

```
int a[] = {
#embed "/dev/random" __limit__(5)
};
```

Giờ... cái `limit` này là gì?

19.9.2 Tham số `limit()`

Bạn có thể đặt giới hạn số phần tử để embed bằng tham số này.

Đây là giá trị tối đa, không phải tuyệt đối. Nếu file được embed ngắn hơn giới hạn chỉ định, chỉ bấy nhiêu byte sẽ được nhập vào.

Ví dụ `/dev/random` ở trên là ví dụ cho động lực của chuyện này, trong Unix, đó là một *character device file* sẽ trả về dòng số ngẫu nhiên vô hạn.

Embed vô hạn byte thì ác với RAM, nên tham số `limit` cho bạn cách dừng sau một số nhất định.

Cuối cùng, bạn được phép dùng macro `#define` trong `limit`, phòng khi bạn tò mò.

19.9.3 Tham số `if_empty`

Tham số này định nghĩa kết quả embed phải là gì nếu file tồn tại nhưng không chứa dữ liệu. Giả sử file `foo.dat` chứa một byte duy nhất với giá trị 123. Nếu ta làm:

```
int x =
#embed "foo.dat" if_empty(999)
;
```

ta sẽ có:

```
int x = 123;   // When foo.dat contains a 123 byte
```

Nhưng nếu `foo.dat` dài 0 byte (tức là không chứa dữ liệu và rỗng)? Nếu vậy, nó sẽ mở rộng thành:

```
int x = 999;   // When foo.dat is empty
```

Đáng chú ý là nếu `limit` đặt thành `0`, thì `if_empty` sẽ luôn được thay vào. Tức là, giới hạn bằng không có nghĩa là file rỗng.

Cái này sẽ luôn phát ra `x = 999` bất kể trong `foo.dat` có gì:

```
int x =
#embed "foo.dat" limit(0) if_empty(999)
;
```

19.9.4 Tham số `prefix()` và `suffix()`

Đây là cách để thêm dữ liệu vào trước/sau embed.

Chú ý rằng các tham số này chỉ ảnh hưởng dữ liệu không rỗng! Nếu file rỗng, cả `prefix` lẫn `suffix` đều không có tác dụng.

Ví dụ ta embed ba số ngẫu nhiên, nhưng thêm tiền tố `11,` và hậu tố `,99`:

```
int x[] = {
#embed "/dev/urandom" limit(3) prefix(11,) suffix(,99)
};
```

Ví dụ kết quả:

```
int x[] = {11,135,116,220,99};
```

Không bắt buộc phải dùng cả `prefix` và `suffix`. Bạn có thể dùng cả hai, một cái, cái kia, hay không cái nào.

Ta có thể tận dụng đặc tính chỉ áp dụng với file không rỗng này để có hiệu quả hay ho, như trong ví dụ sau ăn cắp trắng trợn từ spec.

Giả sử có file `foo.dat` có dữ liệu trong đó. Và ta muốn dùng nó để khởi tạo một mảng, rồi muốn hậu tố của mảng là một phần tử không.

Không vấn đề, đúng không?

```
int x[] = {
#embed "foo.dat" suffix(,0)
};
```

Nếu `foo.dat` chứa 11, 22, và 33, ta sẽ có:

```
int x[] = {11,22,33,0};
```

Nhưng khoan! Nếu `foo.dat` rỗng thì sao? Ta có:

```
int x[] = {};
```

và cái đó không tốt.

Nhưng ta có thể sửa thế này:

```
int x[] = {
#embed "foo.dat" suffix(,)
    0
};
```

Vì tham số `suffix` bị bỏ đi nếu file rỗng, cái này sẽ biến thành:

```
int x[] = {0};
```

thế là ổn.

19.9.5 Định danh `__has_embed()`

Đây là cách hay để kiểm tra xem một file cụ thể có sẵn sàng để embed không, và cũng xem nó có rỗng hay không.

Bạn dùng nó với directive `#if`.

Đây là một đoạn code lấy 5 số ngẫu nhiên từ character device sinh số ngẫu nhiên. Nếu cái đó không tồn tại, nó thử lấy từ file `myrandoms.dat`. Nếu cái đó không tồn tại, nó dùng vài giá trị cứng:

```
    int random_nums[] = {
#if __has_embed("/dev/urandom")
```

```
    #embed "/dev/urandom" limit(5)
#elif __has_embed("myrandoms.dat")
    #embed "myrandoms.dat" limit(5)
#else
    140,178,92,167,120
#endif
    };
```

Về kỹ thuật, định danh `__has_embed()` phân giải ra một trong ba giá trị:

Kết quả `__has_embed()`	Mô tả
`__STDC_EMBED_NOT_FOUND__`	Nếu file không tìm thấy.
`__STDC_EMBED_FOUND__`	Nếu file tìm thấy và không rỗng.
`__STDC_EMBED_EMPTY__`	Nếu file tìm thấy và rỗng.

Tôi có lý do chính đáng để tin rằng `__STDC_EMBED_NOT_FOUND__` là `0` và mấy cái còn lại khác không (vì điều đó được ngầm chỉ trong đề xuất và hợp logic), nhưng tôi đang vất vả tìm chỗ đó trong bản dự thảo spec này.

TODO

19.9.6 Tham số khác

Một implementation của compiler có thể định nghĩa các tham số embed khác tùy ý, tìm các tham số không chuẩn này trong tài liệu của compiler.

Ví dụ:

```
#embed "foo.bin" limit(12) frotz(lamp)
```

Chúng thường có tiền tố để giúp namespace:

```
#embed "foo.bin" limit(12) fmc::frotz(lamp)
```

Có lẽ hợp lý là thử phát hiện xem những cái này có sẵn không trước khi dùng, và may là ta có thể dùng `__has_embed` để giúp.

Thường, `__has_embed()` sẽ chỉ báo file có ở đó hay không. Nhưng, và đây là phần vui, nó cũng sẽ trả false nếu thêm bất kỳ tham số nào không được hỗ trợ!

Vậy nếu ta đưa nó một file mà ta *biết* tồn tại cùng với tham số mà ta muốn test sự tồn tại, nó sẽ hiệu quả báo tham số đó có được hỗ trợ không.

Nhưng file nào *luôn* tồn tại? Hóa ra ta có thể dùng macro `__FILE__`, mở rộng thành tên file nguồn tham chiếu đến nó! File đó *phải* tồn tại, không thì có chuyện cực kỳ nghiêm trọng trong mảng con gà đẻ trứng.

Test tham số `frotz` xem có dùng được không:

```
#if __has_embed(__FILE__ fmc::frotz(lamp))
    puts("fmc::frotz(lamp) is supported!");
#else
    puts("fmc::frotz(lamp) is NOT supported!");
#endif
```

19.9.7 Embed giá trị nhiều byte

Thế còn việc nhét vài `int` vào thay vì byte đơn thì sao? Thế còn giá trị nhiều byte trong file embed?

Đây không phải cái được chuẩn C23 hỗ trợ, nhưng có thể có các extension implementation định nghĩa cho nó trong tương lai.

19.10 Directive `#pragma`

Đây là một directive kỳ cục, viết tắt của "pragmatic". Bạn có thể dùng để làm... ờ, bất cứ gì mà compiler của bạn hỗ trợ bạn làm với nó.

Về cơ bản, lần duy nhất bạn thêm cái này vào code là khi tài liệu nào đó bảo bạn làm vậy.

19.10.1 Pragma không chuẩn

Đây là một ví dụ không chuẩn dùng `#pragma` để bắt compiler chạy vòng lặp `for` song song trên nhiều thread (nếu compiler hỗ trợ extension OpenMP[6]):

```
#pragma omp parallel for
for (int i = 0; i < 10; i++) { ... }
```

Có đủ loại directive `#pragma` được ghi chép ở khắp bốn góc của địa cầu.

Mọi `#pragma` không nhận diện được đều bị compiler bỏ qua.

19.10.2 Pragma chuẩn

Cũng có vài cái chuẩn, và chúng bắt đầu bằng `STDC`, theo cùng dạng:

```
#pragma STDC pragma_name on-off
```

Phần `on-off` có thể là `ON`, `OFF`, hoặc `DEFAULT`.

Và `pragma_name` có thể là một trong các cái sau:

Tên Pragma	Mô tả
`FP_CONTRACT`	Cho phép biểu thức floating point được rút gọn thành một phép toán duy nhất để tránh lỗi làm tròn có thể xảy ra từ nhiều phép toán.
`FENV_ACCESS`	Đặt `ON` nếu bạn định truy cập các cờ trạng thái floating point. Nếu `OFF`, compiler có thể thực hiện tối ưu gây ra giá trị trong cờ không nhất quán hoặc không hợp lệ.
`CX_LIMITED_RANGE`	Đặt `ON` để compiler bỏ qua kiểm tra tràn khi làm số học phức. Mặc định là `OFF`.

Ví dụ:

```
#pragma STDC FP_CONTRACT OFF
#pragma STDC CX_LIMITED_RANGE ON
```

Về `CX_LIMITED_RANGE`, spec chỉ ra:

> Mục đích của pragma là cho implementation dùng các công thức:
> $(x + iy) \times (u + iv) = (xu - yv) + i(yu + xv)$
> $(x + iy)/(u + iv) = [(xu + yv) + i(yu - xv)]/(u^2 + v^2)$

[6] https://www.openmp.org/

> $|x + iy| = \sqrt{x^2 + y^2}$
> khi lập trình viên có thể xác định rằng chúng an toàn.

19.10.3 Toán tử `_Pragma`

Đây là cách khác để khai báo pragma mà bạn có thể dùng trong một macro.

Hai thứ sau là tương đương:

```
#pragma "Unnecessary" quotes
_Pragma("\"Unnecessary\" quotes")
```

Cái này có thể dùng trong macro, khi cần:

```
#define PRAGMA(x) _Pragma(#x)
```

19.11 Directive `#line`

Cái này cho phép bạn ghi đè giá trị cho `__LINE__` và `__FILE__`. Nếu bạn muốn.

Tôi chưa bao giờ muốn làm chuyện này, nhưng trong K&R2, họ viết:

> Để hữu ích cho các preprocessor khác sinh ra chương trình C [...]

Vậy có khi có chỗ cho nó.

Để ghi đè số dòng thành, chẳng hạn, 300:

```
#line 300
```

và `__LINE__` sẽ tiếp tục đếm lên từ đó.

Để ghi đè số dòng và tên file:

```
#line 300 "newfilename"
```

19.12 Directive Null

Một ký tự `#` trên một dòng đứng một mình sẽ bị preprocessor bỏ qua. Thành thật mà nói, tôi không biết ca dùng cho chuyện này là gì.

Tôi đã thấy ví dụ kiểu này:

```
#ifdef FOO
    #
#else
    printf("Something");
#endif
```

đó chỉ là về mặt thẩm mỹ; dòng với `#` đơn độc có thể bị xóa đi mà không có tác động gì xấu.

Hay có lẽ vì tính nhất quán thẩm mỹ, thế này:

```
#
#ifdef FOO
    x = 2;
#endif
```

```
#
#if BAR == 17
    x = 12;
#endif
#
```

Nhưng, về mặt thẩm mỹ, cái đó chỉ là xấu.

Một bài đăng khác nhắc đến chuyện xóa comment, rằng trong GCC, một comment sau `#` sẽ không được compiler nhìn thấy. Cái đó tôi không nghi ngờ, nhưng spec dường như không nói đây là hành vi chuẩn.

Mấy tìm kiếm của tôi cho nguyên do không mang về nhiều quả. Nên tôi đành nói đây là một thứ C esoterica kiểu cổ điển.

Chapter 20

`struct` II: Nghịch `struct` vui hơn

Hóa ra còn khá nhiều thứ có thể làm với `struct` mà ta chưa bàn, nhưng nó chỉ là một đống thứ linh tinh. Nên ta nhét hết vào chương này.

Nếu bạn đã thạo căn bản về `struct`, bạn có thể làm tròn kiến thức ở đây.

20.1 Khởi tạo `struct` lồng nhau và mảng

Nhớ cách bạn có thể khởi tạo thành viên struct theo các dòng này không?

```
struct foo x = {.a=12, .b=3.14};
```

Hóa ra ta có nhiều sức mạnh trong các initializer này hơn là lúc đầu chia sẻ. Hấp dẫn!

Một chuyện là, nếu bạn có substructure lồng nhau như sau, bạn có thể khởi tạo thành viên của substructure đó bằng cách đi theo tên biến xuống dần:

```
struct foo x = {.a.b.c=12};
```

Xem ví dụ:

```
#include <stdio.h>

struct cabin_information {
    int window_count;
    int o2level;
};

struct spaceship {
    char *manufacturer;
    struct cabin_information ci;
};

int main(void)
{
    struct spaceship s = {
        .manufacturer="General Products",
        .ci.window_count = 8,   // <-- NESTED INITIALIZER!
        .ci.o2level = 21
    };

    printf("%s: %d seats, %d%% oxygen\n",
```

```
        s.manufacturer, s.ci.window_count, s.ci.o2level);
}
```

Xem dòng 16-17! Đó là chỗ ta khởi tạo các thành viên của `struct cabin_information` trong định nghĩa `s`, tức `struct spaceship` của ta.

Và đây là lựa chọn khác cho cùng initializer đó, lần này ta làm một thứ trông chuẩn hơn, nhưng cách nào cũng chạy:

```
    struct spaceship s = {
        .manufacturer="General Products",
        .ci={
            .window_count = 8,
            .o2level = 21
        }
    };
```

Giờ, như thể thông tin ở trên còn chưa đủ ngoạn mục, ta cũng có thể trộn initializer mảng vào đó luôn.

Sửa cái này để có mảng thông tin hành khách, và ta có thể xem initializer hoạt động trong đó ra sao.

```
#include <stdio.h>

struct passenger {
    char *name;
    int covid_vaccinated; // Boolean
};

#define MAX_PASSENGERS 8

struct spaceship {
    char *manufacturer;
    struct passenger passenger[MAX_PASSENGERS];
};

int main(void)
{
    struct spaceship s = {
        .manufacturer="General Products",
        .passenger = {
            // Initialize a field at a time
            [0].name = "Gridley, Lewis",
            [0].covid_vaccinated = 0,

            // Or all at once
            [7] = {.name="Brown, Teela", .covid_vaccinated=1},
        }
    };

    printf("Passengers for %s ship:\n", s.manufacturer);

    for (int i = 0; i < MAX_PASSENGERS; i++)
        if (s.passenger[i].name != NULL)
            printf("    %s (%svaccinated)\n",
                s.passenger[i].name,
                s.passenger[i].covid_vaccinated? "": "not ");
}
```

20.2 `struct` vô danh

Đây là "struct không tên". Ta cũng có nhắc mấy cái này ở phần `typedef`, nhưng ta sẽ ôn lại ở đây.

Đây là `struct` thường:

```
struct animal {
    char *name;
    int leg_count, speed;
};
```

Và đây là phiên bản vô danh tương đương:

```
struct {             // <-- No name!
    char *name;
    int leg_count, speed;
};
```

Ừuuuu. Vậy ta có một `struct` không tên, không có cách nào dùng sau này? Nghe có vẻ vô dụng.

Công nhận, trong ví dụ đó, đúng là thế. Nhưng ta vẫn có thể tận dụng nó bằng vài cách.

Một cách hiếm, nhưng vì `struct` vô danh đại diện một kiểu, ta có thể đặt vài tên biến ngay sau nó và dùng chúng.

```
struct {                  // <-- No name!
    char *name;
    int leg_count, speed;
} a, b, c;                // 3 variables of this struct type

a.name = "antelope";
c.leg_count = 4;          // for example
```

Nhưng cũng không hữu dụng mấy.

Phổ biến hơn nhiều là dùng `struct` vô danh với `typedef` để có thể dùng sau (ví dụ để truyền biến cho hàm).

```
typedef struct {                  // <-- No name!
    char *name;
    int leg_count, speed;
} animal;                         // New type: animal

animal a, b, c;

a.name = "antelope";
c.leg_count = 4;          // for example
```

Cá nhân tôi không dùng nhiều `struct` vô danh. Tôi thấy dễ chịu hơn khi thấy cả `struct animal` trước tên biến trong khai báo.

Nhưng đó chỉ là, kiểu, ý kiến của tôi thôi, anh bạn.

20.3 `struct` tự tham chiếu

Với bất kỳ cấu trúc dữ liệu dạng đồ thị nào, có con trỏ tới các node/đỉnh nối với nó là hữu ích. Nhưng điều này nghĩa là trong định nghĩa một node, bạn cần có con trỏ tới một node. Con gà và quả trứng!

Nhưng hóa ra bạn làm được chuyện này trong C mà không gặp vấn đề gì.

Ví dụ, đây là một node của linked list:

```
struct node {
    int data;
    struct node *next;
};
```

Quan trọng là `next` là con trỏ. Đây là điều cho phép cả mớ build được. Dù compiler chưa biết nguyên `struct node` trông thế nào, mọi con trỏ đều cùng kích thước.

Đây là một chương trình linked list ẩu để thử nó:

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

struct node {
    int data;
    struct node *next;
};

int main(void)
{
    struct node *head;

    // Hackishly set up a linked list (11)->(22)->(33)
    head = malloc(sizeof(struct node));
    head->data = 11;
    head->next = malloc(sizeof(struct node));
    head->next->data = 22;
    head->next->next = malloc(sizeof(struct node));
    head->next->next->data = 33;
    head->next->next->next = NULL;

    // Traverse it
    for (struct node *cur = head; cur != NULL; cur = cur->next) {
        printf("%d\n", cur->data);
    }
}
```

Chạy nó in ra:

```
11
22
33
```

20.4 Flexible array member

Ngày xửa ngày xưa, khi người ta còn đẽo code C từ gỗ, có người nghĩ sẽ hay nếu có thể cấp phát `struct` mà có mảng độ dài biến đổi ở cuối.

Tôi muốn nói rõ rằng phần đầu của đoạn này là cách cũ, và ta sẽ làm cách mới ở phần sau.

Ví dụ, có thể bạn định nghĩa một `struct` để chứa chuỗi cùng độ dài chuỗi đó. Nó sẽ có độ dài và một mảng để chứa dữ liệu. Có khi thế này:

```
struct len_string {
    int length;
```

```
    char data[8];
};
```

Nhưng cái đó có `8` được đóng cứng làm độ dài tối đa của chuỗi, mà không nhiều lắm. Thế nếu ta làm gì đó *ngầu* và chỉ cần `malloc()` thêm không gian ở cuối sau struct, rồi để dữ liệu tràn vào không gian đó?

Làm thế đi, rồi cấp phát thêm 40 byte:

```
struct len_string *s = malloc(sizeof *s + 40);
```

Vì `data` là field cuối của `struct`, nếu ta làm tràn field đó, nó chảy ra không gian mà ta đã cấp phát! Vì vậy, trò này chỉ hoạt động nếu mảng ngắn là field *cuối* của `struct`.

```
// Copy more than 8 bytes!

strcpy(s->data, "Hello, world!");  // Won't crash. Probably.
```

Thật ra, có một cách lách thường gặp cho compiler để làm chuyện này, bạn cấp phát một mảng độ dài không ở cuối:

```
struct len_string {
    int length;
    char data[0];
};
```

Và rồi mỗi byte thừa bạn cấp phát đã sẵn sàng để dùng trong chuỗi đó.

Vì `data` là field cuối của `struct`, nếu ta làm tràn field đó, nó chảy ra không gian mà ta đã cấp phát!

```
// Copy more than 8 bytes!

strcpy(s->data, "Hello, world!");  // Won't crash. Probably.
```

Nhưng dĩ nhiên, truy cập dữ liệu vượt cuối mảng đó là undefined behavior! Trong thời hiện đại, ta không còn hạ mình làm kiểu man rợ đó.

May cho ta, ta vẫn có hiệu quả tương tự với C99 trở về sau, nhưng giờ là hợp pháp.

Chỉ việc đổi định nghĩa trên để mảng không có kích thước[1]:

```
struct len_string {
    int length;
    char data[];
};
```

Vẫn thế, cái này chỉ chạy nếu flexible array member là field *cuối* của `struct`.

Rồi ta có thể cấp phát bao nhiêu không gian tùy ý cho các chuỗi đó bằng cách `malloc()` lớn hơn `struct len_string`, như trong ví dụ này tạo một `struct len_string` mới từ chuỗi C:

```
struct len_string *len_string_from_c_string(char *s)
{
    int len = strlen(s);

    // Allocate "len" more bytes than we'd normally need
```

[1] Kỹ thuật mà nói, ta gọi nó có *incomplete type*.

```
    struct len_string *ls = malloc(sizeof *ls + len);

    ls->length = len;

    // Copy the string into those extra bytes
    memcpy(ls->data, s, len);

    return ls;
}
```

20.5 Byte đệm (padding)

Cẩn thận rằng C được phép thêm byte đệm bên trong hoặc sau một `struct` tùy ý nó. Bạn không thể tin rằng chúng sẽ liền kề nhau trong bộ nhớ[2].

Xem chương trình này. Ta xuất hai số. Một là tổng `sizeof` của từng kiểu field riêng lẻ. Cái kia là `sizeof` cả `struct`.

Lẽ ra ta kỳ vọng chúng bằng nhau. Kích thước của cái toàn thể là tổng kích thước các phần, đúng không?

```
#include <stdio.h>

struct foo {
    int a;
    char b;
    int c;
    char d;
};

int main(void)
{
    printf("%zu\n", sizeof(int) + sizeof(char) + sizeof(int) + sizeof(char));
    printf("%zu\n", sizeof(struct foo));
}
```

Nhưng trên hệ của tôi, cái này xuất:

```
10
16
```

Chúng không bằng nhau! Compiler đã thêm 6 byte đệm để giúp nó chạy nhanh hơn. Có thể bạn nhận kết quả khác với compiler của bạn, nhưng trừ khi bạn ép buộc, bạn không thể chắc không có đệm.

20.6 `offsetof`

Trong đoạn trước, ta thấy compiler có thể chêm byte đệm tùy ý vào trong cấu trúc.

Nếu ta cần biết chúng ở đâu? Ta có thể đo bằng `offsetof`, định nghĩa trong `<stddef.h>`.

Sửa code ở trên để in offset của từng field trong `struct`:

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

struct foo {
```

[2]Dù vài compiler có tùy chọn ép chuyện này xảy ra, tra `__attribute__((packed))` để xem cách làm với GCC.

```
    int a;
    char b;
    int c;
    char d;
};

int main(void)
{
    printf("%zu\n", offsetof(struct foo, a));
    printf("%zu\n", offsetof(struct foo, b));
    printf("%zu\n", offsetof(struct foo, c));
    printf("%zu\n", offsetof(struct foo, d));
}
```

Với tôi, cái này xuất:

```
0
4
8
12
```

cho biết ta đang dùng 4 byte cho mỗi field. Hơi lạ, vì `char` chỉ có 1 byte, đúng không? Compiler đang đặt 3 byte đệm sau mỗi `char` để mọi field đều dài 4 byte. Chắc chuyện này sẽ chạy nhanh hơn trên CPU của tôi.

20.7 OOP giả

Có một trò hơi lạm dụng, kiểu kiểu OOP, mà bạn có thể làm với `struct`.

Vì con trỏ tới `struct` trùng với con trỏ tới phần tử đầu tiên của `struct`, bạn có thể thoải mái ép kiểu con trỏ tới `struct` sang con trỏ tới phần tử đầu tiên.

Điều này có nghĩa là ta có thể dựng tình huống thế này:

```
struct parent {
    int a, b;
};

struct child {
    struct parent super;  // MUST be first
    int c, d;
};
```

Rồi ta có thể truyền con trỏ tới `struct child` cho một hàm mong đợi *hoặc* con trỏ tới `struct parent`!

Vì `struct parent super` là phần tử đầu của `struct child`, con trỏ tới bất kỳ `struct child` nào cũng trùng với con trỏ tới field `super` đó[3].

Dựng ví dụ luôn. Ta làm `struct` như trên, rồi truyền con trỏ tới `struct child` cho hàm cần con trỏ tới `struct parent` ... và vẫn chạy.

```
#include <stdio.h>

struct parent {
    int a, b;
};
```

[3]Nhân tiện, `super` không phải từ khóa. Tôi chỉ mượn vài thuật ngữ OOP thôi.

```
struct child {
    struct parent super;  // MUST be first
    int c, d;
};

// Making the argument `void*` so we can pass any type into it
// (namely a struct parent or struct child)
void print_parent(void *p)
{
    // Expects a struct parent--but a struct child will also work
    // because the pointer points to the struct parent in the first
    // field:
    struct parent *self = p;

    printf("Parent: %d, %d\n", self->a, self->b);
}

void print_child(struct child *self)
{
    printf("Child: %d, %d\n", self->c, self->d);
}

int main(void)
{
    struct child c = {.super.a=1, .super.b=2, .c=3, .d=4};

    print_child(&c);
    print_parent(&c);  // Also works even though it's a struct child!
}
```

Thấy chuyện ta làm ở dòng cuối `main()` chứ? Ta gọi `print_parent()` mà truyền `struct child*` làm tham số! Dù `print_parent()` cần tham số trỏ tới `struct parent`, ta *thoát* được vì field đầu của `struct child` là `struct parent`.

Vẫn vậy, cái này chạy được vì con trỏ tới `struct` có cùng giá trị với con trỏ tới field đầu trong `struct` đó.

Tất cả dựa trên phần này của spec:

> **§6.7.2.1¶15** [...] A pointer to a structure object, suitably converted, points to its initial member [...], and vice versa.

và

> **§6.5¶7** An object shall have its stored value accessed only by an lvalue expression that has one of the following types:
>
> - a type compatible with the effective type of the object
> - [...]

và giả định của tôi rằng "suitably converted" nghĩa là "ép kiểu sang kiểu hiệu lực của phần tử đầu".

20.8 Bit-field

Theo kinh nghiệm của tôi, mấy cái này ít khi dùng, nhưng bạn có thể thấy đây đó, đặc biệt trong ứng dụng tầng thấp nơi người ta dồn bit vào không gian lớn hơn.

Xem đoạn code để minh họa trường hợp dùng:

```
#include <stdio.h>

struct foo {
    unsigned int a;
    unsigned int b;
    unsigned int c;
    unsigned int d;
};

int main(void)
{
    printf("%zu\n", sizeof(struct foo));
}
```

Với tôi, cái này in `16`. Hợp lý, vì `unsigned` là 4 byte trên hệ của tôi.

Nhưng nếu ta biết mọi giá trị sẽ được lưu trong `a` và `b` đều chứa được trong 5 bit, và giá trị trong `c` và `d` chứa được trong 3 bit? Tổng cộng mới 16 bit. Sao lại phải dành 128 bit cho chúng nếu ta chỉ dùng 16?

Ta có thể nói với C làm-ơn-thử-gói các giá trị này lại. Ta có thể chỉ định số bit tối đa mà giá trị có thể chiếm (từ 1 lên kích thước kiểu chứa).

Làm bằng cách đặt dấu hai chấm sau tên field, rồi tới độ rộng field tính bằng bit.

```
struct foo {
    unsigned int a:5;
    unsigned int b:5;
    unsigned int c:3;
    unsigned int d:3;
};
```

Giờ khi tôi hỏi C `struct foo` lớn cỡ nào, nó nói 4! Trước là 16 byte, giờ chỉ còn 4. Nó đã "gói" 4 giá trị đó vào 4 byte, tiết kiệm bộ nhớ gấp bốn.

Đánh đổi dĩ nhiên là field 5-bit chỉ chứa giá trị 0-31 và field 3-bit chỉ chứa giá trị 0-7. Nhưng đời sau cùng vẫn là về thỏa hiệp.

20.8.1 Bit-field không liền kề

Một cái bẫy: C chỉ gộp bit-field **liền kề** thôi. Nếu chúng bị ngắt bởi non-bit-field, bạn không tiết kiệm được gì:

```
struct foo {            // sizeof(struct foo) == 16 (for me)
    unsigned int a:1;   // since a is not adjacent to c.
    unsigned int b;
    unsigned int c:1;
    unsigned int d;
};
```

Trong ví dụ đó, vì `a` không liền kề `c`, chúng đều được "gói" vào `int` riêng của mình.

Nên ta có một `int` cho mỗi `a`, `b`, `c`, `d`. Vì `int` của tôi là 4 byte, tổng cộng 16 byte.

Sắp xếp lại nhanh tiết kiệm không gian từ 16 byte xuống 12 byte (trên hệ của tôi):

```
struct foo {            // sizeof(struct foo) == 12 (for me)
    unsigned int a:1;
    unsigned int c:1;
```

```
    unsigned int b;
    unsigned int d;
};
```

Và giờ, vì `a` kế bên `c`, compiler đặt chúng vào một `int` duy nhất.

Nên ta có một `int` cho `a` và `c` gộp, và một `int` mỗi cái cho `b` và `d`. Tổng cộng 3 `int`, hay 12 byte.

Đặt hết bitfield chung với nhau để compiler gộp chúng.

20.8.2 `int` có dấu hay không dấu

Nếu bạn chỉ khai báo bit-field là `int`, các compiler khác nhau sẽ xử lý nó là `signed` hoặc `unsigned`. Giống tình huống với `char`.

Hãy rõ ràng về dấu khi dùng bit-field.

20.8.3 Bit-field không tên

Trong vài tình huống cụ thể, bạn có thể cần dành một số bit vì lý do phần cứng, nhưng không cần dùng chúng trong code.

Ví dụ, giả sử bạn có một byte mà 2 bit trên có ý nghĩa, 1 bit dưới có ý nghĩa, còn 5 bit giữa không được bạn dùng[4].

Ta *có thể* làm thế này:

```
struct foo {
    unsigned char a:2;
    unsigned char dummy:5;
    unsigned char b:1;
};
```

Và cái đó chạy, trong code ta dùng `a` và `b`, không bao giờ dùng `dummy`. Nó chỉ ở đó để ăn hết 5 bit để chắc rằng `a` và `b` ở đúng vị trí "yêu cầu" (theo ví dụ giả định này) trong byte.

C cho ta một cách dọn cái này: *bit-field không tên.* Bạn chỉ cần bỏ tên (`dummy`) trong trường hợp này, và C hoàn toàn hài lòng cho hiệu quả tương tự:

```
struct foo {
    unsigned char a:2;
    unsigned char :5;   // <-- unnamed bit-field!
    unsigned char b:1;
};
```

20.8.4 Bit-field không tên độ rộng zero

Thêm tí thứ bí hiểm ngoài đây... Giả sử bạn đang gói bit vào một `unsigned int`, và bạn cần vài bit-field liền kề được gói vào `unsigned int` *tiếp theo.*

Tức là, nếu bạn làm thế này:

```
struct foo {
    unsigned int a:1;
    unsigned int b:2;
    unsigned int c:3;
```

[4]Giả sử `char` 8 bit, tức là `CHAR_BIT == 8`.

```
    unsigned int d:4;
};
```

compiler gói hết tất cả vào một `unsigned int` duy nhất. Nhưng nếu bạn cần `a` và `b` trong một `int`, và `c` và `d` trong một cái khác?

Có giải pháp: đặt bit-field không tên độ rộng `0` ở chỗ bạn muốn compiler bắt đầu lại việc gói bit vào `int` khác:

```
struct foo {
    unsigned int a:1;
    unsigned int b:2;
    unsigned int :0;   // <--Zero-width unnamed bit-field
    unsigned int c:3;
    unsigned int d:4;
};
```

Nó tương tự như ngắt trang tường minh trong word processor. Bạn nói với compiler: "Dừng gói bit vào `unsigned` này, và bắt đầu gói vào cái tiếp theo."

Bằng cách thêm bit-field không tên độ rộng zero ở chỗ đó, compiler đặt `a` và `b` vào một `unsigned int`, và `c` với `d` vào một `unsigned int` khác. Tổng cộng hai, cỡ 8 byte trên hệ của tôi (`unsigned int` mỗi cái 4 byte).

20.9 Union

Về cơ bản mấy cái này giống `struct`, chỉ khác là các field chồng lên nhau trong bộ nhớ. `union` sẽ chỉ đủ lớn cho field lớn nhất, và bạn chỉ dùng được một field mỗi lần.

Đây là cách tái sử dụng cùng không gian bộ nhớ cho các kiểu dữ liệu khác nhau.

Bạn khai báo chúng y như `struct`, chỉ đổi sang `union`. Xem cái này:

```
union foo {
    int a, b, c, d, e, f;
    float g, h;
    char i, j, k, l;
};
```

Đấy, có một đống field. Nếu đây là `struct`, hệ tôi sẽ nói cần 36 byte để chứa tất cả.

Nhưng đây là `union`, nên mọi field đó chồng lên cùng đoạn bộ nhớ. Field lớn nhất là `int` (hoặc `float`), chiếm 4 byte trên hệ tôi. Và đúng thế, nếu tôi hỏi `sizeof` của `union foo`, nó nói 4!

Đánh đổi là bạn chỉ dùng được di động một trong các field đó mỗi lần. Tuy nhiên...

20.9.1 Union và type punning

Bạn có thể ghi vào một field của `union` rồi đọc từ field khác, nhưng không di động!

Làm vậy gọi là type punning[5], và bạn dùng nó nếu bạn thật sự biết mình đang làm gì, thường là trong kiểu lập trình tầng thấp nào đó.

Vì các thành viên của union chia sẻ cùng bộ nhớ, ghi vào một thành viên tất yếu ảnh hưởng các thành viên khác. Và nếu bạn đọc từ cái đã được ghi vào cái khác, bạn sẽ có những hiệu ứng kỳ lạ.

[5] https://en.wikipedia.org/wiki/Type_punning

```
#include <stdio.h>

union foo {
    float b;
    short a;
};

int main(void)
{
    union foo x;

    x.b = 3.14159;

    printf("%f\n", x.b);  // 3.14159, fair enough

    printf("%d\n", x.a);  // But what about this?
}
```

Trên hệ của tôi, cái này in ra:

```
3.141590
4048
```

vì dưới mui xe, biểu diễn đối tượng cho float `3.14159` giống hệt biểu diễn đối tượng cho short `4048`. Trên hệ của tôi. Kết quả của bạn có thể khác.

20.9.2 Con trỏ tới `union`

Nếu bạn có con trỏ tới `union`, bạn có thể ép kiểu con trỏ đó sang bất kỳ kiểu nào của các field trong `union` đó và lấy giá trị ra theo cách đó.

Trong ví dụ này, ta thấy `union` có `int` và `float` trong đó. Và ta lấy con trỏ tới `union`, nhưng ép sang kiểu `int*` và `float*` (cast để làm im compiler). Rồi nếu ta dereference chúng, ta thấy chúng có giá trị ta đã lưu trực tiếp trong `union`.

```
#include <stdio.h>

union foo {
    int a, b, c, d, e, f;
    float g, h;
    char i, j, k, l;
};

int main(void)
{
    union foo x;

    int *foo_int_p = (int *)&x;
    float *foo_float_p = (float *)&x;

    x.a = 12;
    printf("%d\n", x.a);           // 12
    printf("%d\n", *foo_int_p);    // 12, again

    x.g = 3.141592;
    printf("%f\n", x.g);           // 3.141592
    printf("%f\n", *foo_float_p);  // 3.141592, again
```

```
}
```

Điều ngược lại cũng đúng. Nếu ta có con trỏ tới một kiểu bên trong `union`, ta có thể ép sang con trỏ tới `union` và truy cập các thành viên của nó.

```
union foo x;
int *foo_int_p = (int *)&x;             // Pointer to int field
union foo *p = (union foo *)foo_int_p;  // Back to pointer to union

p->a = 12;  // This line the same as...
x.a = 12;   // this one.
```

Tất cả chuyện này chỉ cho bạn biết rằng, dưới mui xe, mọi giá trị trong một `union` bắt đầu cùng chỗ trong bộ nhớ, và đó cũng là chỗ cả `union` ở.

20.9.3 Common initial sequence trong union

Nếu bạn có một `union` các `struct`, và tất cả `struct` đó bắt đầu bằng một *common initial sequence*, truy cập các thành viên của sequence đó từ bất kỳ thành viên nào của `union` là hợp lệ.

Gì cơ?

Đây là hai `struct` với common initial sequence:

```
struct a {
    int x;     //
    float y;   // Common initial sequence

    char *p;
};

struct b {
    int x;     //
    float y;   // Common initial sequence

    double *p;
    short z;
};
```

Bạn thấy chưa? Là chúng bắt đầu bằng `int` tiếp theo là `float`, đó là common initial sequence. Các thành viên trong sequence của các `struct` phải là kiểu tương thích. Và ta thấy với `x` và `y`, là `int` và `float`.

Giờ xây một union của mấy cái này:

```
union foo {
    struct a sa;
    struct b sb;
};
```

Quy tắc này nói cho ta rằng ta được đảm bảo các thành viên của common initial sequence có thể hoán đổi cho nhau trong code. Tức là:

- `f.sa.x` giống `f.sb.x`.
- `f.sa.y` giống `f.sb.y`.

Vì field `x` và `y` đều nằm trong common initial sequence.

Ngoài ra, tên của các thành viên trong common initial sequence không quan trọng, chỉ quan trọng kiểu phải giống nhau.

Tất cả gộp lại cho ta cách an toàn để thêm vài thông tin chia sẻ giữa các `struct` trong `union`. Ví dụ hay nhất của chuyện này có lẽ là dùng một field để xác định kiểu `struct` nào trong tất cả các `struct` của `union` đang "được dùng".

Tức là, nếu ta không được phép làm thế và ta truyền `union` cho một hàm, hàm đó làm sao biết thành viên nào của `union` là cái nó nên nhìn?

Xem các `struct` này. Để ý common initial sequence:

```
#include <stdio.h>

struct common {
    int type;   // common initial sequence
};

struct antelope {
    int type;   // common initial sequence

    int loudness;
};

struct octopus {
    int type;   // common initial sequence

    int sea_creature;
    float intelligence;
};
```

Giờ quẳng chúng vào `union`:

```
union animal {
    struct common common;
    struct antelope antelope;
    struct octopus octopus;
};
```

Ngoài ra, làm ơn chiều tôi hai `#define` sau cho ví dụ:

```
#define ANTELOPE 1
#define OCTOPUS  2
```

Tới giờ, chẳng có gì đặc biệt xảy ra ở đây. Có vẻ field `type` hoàn toàn vô dụng.

Nhưng giờ ta làm hàm chung in `union animal`. Nó phải cách nào đó biết được mình đang nhìn vào `struct antelope` hay `struct octopus`.

Nhờ phép thuật của common initial sequence, nó có thể tra kiểu animal ở bất kỳ chỗ nào cho một `union animal x` cụ thể:

```
int type = x.common.type;    // or...
int type = x.antelope.type;  // or...
int type = x.octopus.type;
```

Tất cả đều trỏ đến cùng giá trị trong bộ nhớ.

Và, như bạn có thể đoán, `struct common` ở đó để code có thể nhìn kiểu một cách tổng quát mà không phải nhắc tới con vật cụ thể.

Xem code để in `union animal`:

```
void print_animal(union animal *x)
{
    switch (x->common.type) {
        case ANTELOPE:
            printf("Antelope: loudness=%d\n", x->antelope.loudness);
            break;

        case OCTOPUS:
            printf("Octopus : sea_creature=%d\n", x->octopus.sea_creature);
            printf("          intelligence=%f\n", x->octopus.intelligence);
            break;

        default:
            printf("Unknown animal type\n");
    }

}

int main(void)
{
    union animal a = {.antelope.type=ANTELOPE, .antelope.loudness=12};
    union animal b = {.octopus.type=OCTOPUS, .octopus.sea_creature=1,
                                        .octopus.intelligence=12.8};

    print_animal(&a);
    print_animal(&b);
}
```

Xem cách ở dòng 29 ta chỉ truyền vào `union`, ta không biết kiểu animal `struct` nào đang được dùng bên trong.

Nhưng không sao! Vì ở dòng 31 ta kiểm tra kiểu xem là antelope hay octopus. Rồi ta có thể nhìn vào đúng `struct` để lấy thành viên.

Hoàn toàn có thể có hiệu quả tương tự chỉ dùng `struct`, nhưng bạn có thể làm thế này nếu muốn hiệu quả tiết kiệm bộ nhớ của `union`.

20.10 Union và struct vô danh

Bạn biết cách có `struct` vô danh, thế này:

```
struct {
    int x, y;
} s;
```

Cái đó định nghĩa biến `s` thuộc kiểu `struct` vô danh (vì `struct` không có tag tên), với thành viên `x` và `y`.

Nên những chuyện kiểu này là hợp lệ:

```
s.x = 34;
s.y = 90;

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

Hóa ra bạn có thể thả `struct` vô danh vào `union` y như bạn nghĩ:

```
union foo {
    struct {       // unnamed!
        int x, y;
    } a;

    struct {       // unnamed!
        int z, w;
    } b;
};
```

Rồi truy cập chúng như bình thường:

```
union foo f;

f.a.x = 1;
f.a.y = 2;
f.b.z = 3;
f.b.w = 4;
```

Không sao!

20.11 Truyền và trả `struct` và `union`

Bạn có thể truyền `struct` hoặc `union` cho hàm theo giá trị (thay vì con trỏ tới nó), một bản sao của đối tượng đó sẽ được tạo cho tham số như khi gán thông thường.

Bạn cũng có thể trả `struct` hoặc `union` từ hàm và nó cũng được truyền ngược lại theo giá trị.

```
#include <stdio.h>

struct foo {
    int x, y;
};

struct foo f(void)
{
    return (struct foo){.x=34, .y=90};
}

int main(void)
{
    struct foo a = f();  // Copy is made

    printf("%d %d\n", a.x, a.y);
}
```

Chuyện vui: nếu làm thế, bạn có thể dùng toán tử `.` ngay trên lời gọi hàm:

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

(Dĩ nhiên ví dụ đó gọi hàm hai lần, không hiệu quả.)

Và điều tương tự đúng với việc trả con trỏ tới `struct` và `union`, chỉ cần nhớ dùng toán tử mũi tên `->` trong trường hợp đó.

Chapter 21

Ký tự và chuỗi II

Ta đã nói về chuyện kiểu `char` thực ra chỉ là kiểu số nguyên nhỏ, và ký tự nằm trong dấu nháy đơn cũng vậy.

Nhưng chuỗi trong dấu nháy kép thì có kiểu `const char *`.

Hóa ra còn vài kiểu chuỗi và ký tự nữa, và nó dẫn tới một trong những hang thỏ khét tiếng nhất của ngôn ngữ này: cả cái mớ multibyte/wide/Unicode/localization.

Ta sẽ ghé nhìn xuống hang thỏ đó, nhưng chưa chui vào. Chưa đâu!

21.1 Escape sequence

Ta quen với chuỗi và ký tự gồm chữ cái, dấu câu và số thông thường:

```
char *s = "Hello!";
char t = 'c';
```

Nhưng lỡ ta muốn nhét mấy ký tự đặc biệt mà bàn phím không gõ được vì nó không có ở đó (ví dụ "€"), hay kể cả khi ta muốn một ký tự là dấu nháy đơn, thì sao? Rõ ràng ta không thể viết:

```
char t = ''';
```

Để làm mấy chuyện này, ta dùng thứ gọi là *escape sequence* (chuỗi thoát). Nó là ký tự backslash (`\`) theo sau là một ký tự khác. Hai (hoặc nhiều) ký tự đi với nhau mang nghĩa đặc biệt.

Với ví dụ ký tự nháy đơn, ta có thể đặt một escape (tức là `\`) trước dấu nháy đơn ở giữa để giải quyết:

```
char t = '\'';
```

Giờ C biết `\'` nghĩa là dấu nháy thật mà ta muốn in ra, chứ không phải điểm kết thúc chuỗi ký tự.

Bạn có thể nói "backslash" hoặc "escape" trong ngữ cảnh này ("escape cái nháy đó đi") và dân C sẽ hiểu bạn đang nói gì. Lưu ý "escape" ở đây khác với phím `Esc` hay mã ASCII `ESC`.

21.1.1 Mấy escape hay dùng

Theo ý tôi, mấy escape dưới đây chiếm 99.2%[1] của mọi escape.

Code	Mô tả
`\n`	Ký tự newline, khi in ra, phần sau tiếp tục ở dòng kế
`\'`	Nháy đơn, dùng cho hằng ký tự là dấu nháy đơn

[1] Tôi bịa con số đó, nhưng chắc không sai lệch bao xa

Code	Mô tả
`\"`	Nháy kép, dùng cho dấu nháy kép trong string literal
`\\`	Backslash, dùng cho ký tự `\` theo đúng nghĩa trong chuỗi hay ký tự

Vài ví dụ về escape và cái chúng in ra:

```
printf("Use \\n for newline\n");  // Use \n for newline
printf("Say \"hello\"!\n");       // Say "hello"!
printf("%c\n", '\'');             // '
```

21.1.2 Mấy escape ít dùng

Còn nhiều escape khác nữa! Chỉ là bạn ít gặp chúng hơn.

Code	Mô tả
`\a`	Alert. Khiến terminal kêu hoặc chớp sáng, hoặc cả hai!
`\b`	Backspace. Lùi con trỏ về một ký tự. Không xóa ký tự đó.
`\f`	Formfeed. Nhảy sang "trang" tiếp theo, nhưng chuyện đó chẳng còn mấy ý nghĩa ở thời nay. Trên máy tôi, nó hành xử như `\v`.
`\r`	Return. Về đầu cùng dòng hiện tại.
`\t`	Tab ngang. Nhảy tới tab stop ngang kế tiếp. Trên máy tôi, nó dóng vào các cột là bội số của 8, nhưng YMMV.
`\v`	Tab dọc. Nhảy tới tab stop dọc kế tiếp. Trên máy tôi, nó nhảy sang cùng cột ở dòng kế.
`\?`	Dấu hỏi theo đúng nghĩa. Đôi khi bạn cần cái này để tránh trigraph, sẽ nói bên dưới.

21.1.2.1 Cập nhật trạng thái trên một dòng

Một ca dùng của `\b` hay `\r` là hiển thị cập nhật trạng thái trên cùng một dòng màn hình mà không làm nội dung cuộn. Đây là ví dụ đếm ngược từ 10. (Nếu compiler của bạn không hỗ trợ threading, bạn có thể dùng hàm POSIX không chuẩn `sleep()` từ `<unistd.h>`, nếu không ở hệ Unix-like, tìm nền tảng của bạn cộng với `sleep` để có cái tương đương.)

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

int main(void)
{
    for (int i = 10; i >= 0; i--) {
        printf("\rT minus %d second%s... \b", i, i != 1? "s": "");

        fflush(stdout);  // Force output to update

        // Sleep for 1 second
        thrd_sleep(&(struct timespec){.tv_sec=1}, NULL);
    }

    printf("\rLiftoff!             \n");
}
```

Có kha khá chuyện xảy ra ở dòng 7. Đầu tiên, ta mở đầu bằng `\r` để về đầu dòng hiện tại, rồi ghi đè lên bất cứ thứ gì đang ở đó bằng đoạn đếm ngược hiện tại. (Có toán tử ternary ở đó để đảm bảo ta in `1 second` chứ không phải `1 seconds`.)

Cũng có một khoảng trắng sau `...` Đó là để ta ghi đè đúng dấu `.` cuối cùng khi `i` tụt từ `10` xuống `9` và ta bị hụt đi một cột. Thử bỏ khoảng trắng đó đi để thấy tôi muốn nói gì.

Và ta kết bằng `\b` để lùi qua khoảng trắng đó cho con trỏ nằm đúng cuối dòng, cho đẹp.

Chú ý dòng 15 cũng có nhiều khoảng trắng ở cuối để ghi đè các ký tự còn sót lại từ đoạn đếm ngược.

Cuối cùng, có một dòng `fflush(stdout)` lạ lạ, mà không hiểu nghĩa là gì. Ngắn gọn là phần lớn terminal mặc định *line buffered*, nghĩa là chúng không thật sự in ra gì cho tới khi gặp ký tự newline. Vì ta không có newline (chỉ có `\r`), nếu không có dòng đó, chương trình sẽ ngồi im cho tới lúc `Liftoff!` rồi in tất cả trong một nháy. `fflush()` ghi đè hành vi này và ép output diễn ra *ngay bây giờ*.

21.1.2.2 Escape cho dấu hỏi

Sao phải bận tâm chuyện này? Cái này chạy tốt mà:

```
printf("Doesn't it?\n");
```

Và dùng escape cũng chạy tốt:

```
printf("Doesn't it\?\n");   // Note \?
```

Vậy thì để làm gì??!

Ta nhấn mạnh hơn chút với thêm một dấu hỏi và một dấu chấm than:

```
printf("Doesn't it??!\n");
```

Khi tôi compile cái này, tôi nhận được cảnh báo:

```
foo.c: In function ‘main’:
foo.c:5:23: warning: trigraph ??! converted to | [-Wtrigraphs]
    5 |     printf("Doesn't it??!\n");
      |
```

Và chạy nó thì cho kết quả khó tin:

```
Doesn't it|
```

Vậy *trigraph*? Cái quái gì đây??!

Tôi chắc ta sẽ quay lại cái góc bụi bặm này của ngôn ngữ sau, nhưng ngắn gọn là compiler tìm một số bộ ba ký tự nhất định bắt đầu bằng `??` rồi thay chúng bằng ký tự khác. Vậy nếu bạn đang ngồi trước một terminal cổ lỗ sĩ không có ký hiệu pipe (`|`) trên bàn phím, bạn có thể gõ `??!` thay thế.

Bạn có thể sửa bằng cách escape dấu hỏi thứ hai, kiểu vầy:

```
printf("Doesn't it?\?!\n");
```

Và rồi nó compile và chạy như mong đợi.

Tất nhiên, ngày nay chẳng ai dùng trigraph nữa. Nhưng cái `??!` đó đôi khi vẫn xuất hiện nếu bạn quyết định dùng nó trong một chuỗi để nhấn mạnh.

21.1.3 Escape dạng số

Ngoài ra, còn có các cách để chỉ định hằng số hay giá trị ký tự khác bên trong chuỗi hay hằng ký tự.

Nếu bạn biết biểu diễn octal hay hexadecimal của một byte, bạn có thể đưa nó vào một chuỗi hay hằng ký tự.

Bảng dưới có các con số ví dụ, nhưng bất kỳ số hex hay octal nào cũng dùng được. Pad thêm số 0 đầu nếu cần để đủ số chữ số.

Code	Mô tả
`\123`	Nhúng byte có giá trị octal `123`, đúng 3 chữ số.
`\x4D`	Nhúng byte có giá trị hex `4D`, 2 chữ số.
`\u2620`	Nhúng ký tự Unicode tại code point có giá trị hex `2620`, 4 chữ số.
`\U0001243F`	Nhúng ký tự Unicode tại code point có giá trị hex `1243F`, 8 chữ số.

Đây là ví dụ dùng ký pháp octal ít gặp để biểu diễn chữ `B` nằm giữa `A` và `C`. Thường cách này được dùng cho ký tự đặc biệt không in được, nhưng ta có cách khác để làm thế bên dưới, đây chỉ là demo octal thôi:

```
printf("A\102C\n");  // 102 is `B` in ASCII/UTF-8
```

Chú ý không có số 0 đầu ở số octal khi bạn viết theo kiểu này. Nhưng nó cần đúng ba ký tự, nên hãy pad thêm số 0 đầu nếu cần.

Nhưng phổ biến hơn nhiều ngày nay là dùng hẳng hex. Đây là một demo mà bạn không nên dùng, nhưng nó minh họa việc nhúng các byte UTF-8 0xE2, 0x80, và 0xA2 vào trong một chuỗi, ứng với ký tự Unicode "bullet" (•).

```
printf("\xE2\x80\xA2 Bullet 1\n");
printf("\xE2\x80\xA2 Bullet 2\n");
printf("\xE2\x80\xA2 Bullet 3\n");
```

Sinh ra output sau nếu bạn đang ở console UTF-8 (hoặc có khi là rác nếu không):

```
• Bullet 1
• Bullet 2
• Bullet 3
```

Nhưng đó là cách lởm khởm để làm Unicode. Bạn có thể dùng escape `\u` (16-bit) hoặc `\U` (32-bit) để tham chiếu Unicode bằng số code point thẳng luôn. Bullet là `2022` (hex) trong Unicode, nên bạn có thể làm vầy để có kết quả portable hơn:

```
printf("\u2022 Bullet 1\n");
printf("\u2022 Bullet 2\n");
printf("\u2022 Bullet 3\n");
```

Nhớ pad `\u` đủ số 0 đầu cho đủ bốn ký tự, và `\U` đủ để ra tám.

Ví dụ, cái bullet đó có thể làm bằng `\U` với bốn số 0 đầu:

```
printf("\U00002022 Bullet 1\n");
```

Ai rảnh mà dài dòng thế?

Chapter 22

Kiểu liệt kê: `enum`

C cho ta thêm một cách nữa để có giá trị số nguyên hằng đặt tên: `enum`.

Ví dụ:

```
enum {
  ONE=1,
  TWO=2
};

printf("%d %d", ONE, TWO);  // 1 2
```

Ở vài chỗ, nó có thể tốt hơn, hoặc khác, so với dùng `#define`. Mấy khác biệt chính:

- `enum` chỉ có thể là kiểu số nguyên.
- `#define` thì định nghĩa được bất cứ thứ gì.
- `enum` thường hiện ra bằng tên ký hiệu trong debugger.
- Số `#define` chỉ hiện ra dưới dạng số thô, khó biết ý nghĩa lúc debug.

Vì chúng là kiểu số nguyên, chúng dùng được ở bất cứ đâu mà số nguyên dùng được, kể cả kích thước mảng và câu `case`.

Mổ xẻ thêm nhé.

22.1 Hành vi của `enum`

22.1.1 Đánh số

`enum` được đánh số tự động trừ khi bạn ghi đè.

Chúng bắt đầu từ `0`, rồi tự tăng dần lên, theo mặc định:

```
enum {
    SHEEP,  // Value is 0
    WHEAT,  // Value is 1
    WOOD,   // Value is 2
    BRICK,  // Value is 3
    ORE     // Value is 4
};

printf("%d %d\n", SHEEP, BRICK);  // 0 3
```

Bạn có thể ép giá trị số nguyên cụ thể, như ta đã thấy ở trên:

```
enum {
  X=2,
  Y=18,
  Z=-2
};
```

Trùng giá trị cũng không sao:

```
enum {
  X=2,
  Y=2,
  Z=2
};
```

Nếu giá trị bị bỏ đi, việc đánh số tiếp tục đếm theo hướng dương từ giá trị nào được chỉ định gần nhất. Ví dụ:

```
enum {
  A,    // 0, default starting value
  B,    // 1
  C=4,  // 4, manually set
  D,    // 5
  E,    // 6
  F=3,  // 3, manually set
  G,    // 4
  H     // 5
}
```

22.1.2 Dấu phẩy đuôi

Cái này hoàn toàn ổn, nếu bạn thích kiểu đó:

```
enum {
  X=2,
  Y=18,
  Z=-2,   // <-- Trailing comma
};
```

Mấy thập kỷ gần đây nó phổ biến hơn trong các ngôn ngữ khác, nên bạn có thể thấy vui khi gặp lại.

22.1.3 Phạm vi

`enum` có scope đúng như bạn kỳ vọng. Nếu ở file scope, cả file thấy nó. Nếu trong một block, nó cục bộ trong block đó.

Rất thường gặp chuyện `enum` được định nghĩa trong file header để có thể `#include` vào ở file scope.

22.1.4 Style

Như bạn đã để ý, rất phổ biến việc khai báo ký hiệu `enum` bằng chữ hoa (với gạch dưới).

Đây không phải yêu cầu, nhưng là một idiom rất, rất phổ biến.

22.2 `enum` của bạn là một kiểu

Đây là chuyện quan trọng cần biết về `enum`: chúng là một kiểu, tương tự cách `struct` là một kiểu.

Bạn có thể gán cho chúng một tên tag để về sau tham chiếu kiểu và khai báo biến của kiểu đó.

Mà này, vì `enum` là kiểu số nguyên, sao không xài luôn `int`?

Trong C, lý do tốt nhất là để code rõ ràng, đây là cách đẹp, có kiểu, để diễn tả suy nghĩ của bạn trong code. C (khác với C++) thật ra không ép giá trị phải nằm trong phạm vi của một `enum` cụ thể.

Làm ví dụ khai báo biến `r` kiểu `enum resource` có thể giữ các giá trị đó:

```
// Named enum, type is "enum resource"

enum resource {
    SHEEP,
    WHEAT,
    WOOD,
    BRICK,
    ORE
};

// Declare a variable "r" of type "enum resource"

enum resource r = BRICK;

if (r == BRICK) {
    printf("I'll trade you a brick for two sheep.\n");
}
```

Bạn cũng có thể `typedef` cái này, dĩ nhiên, dù cá nhân tôi không thích.

```
typedef enum {
    SHEEP,
    WHEAT,
    WOOD,
    BRICK,
    ORE
} RESOURCE;

RESOURCE r = BRICK;
```

Một lối tắt khác hợp lệ nhưng hiếm là khai báo biến ngay khi khai báo `enum`:

```
// Declare an enum and some initialized variables of that type:

enum {
    SHEEP,
    WHEAT,
    WOOD,
    BRICK,
    ORE
} r = BRICK, s = WOOD;
```

Bạn cũng có thể đặt tên cho `enum` để về sau dùng lại, đây có lẽ là điều bạn muốn làm trong phần lớn trường hợp:

```
// Declare an enum and some initialized variables of that type:

enum resource {   // <-- type is now "enum resource"
    SHEEP,
    WHEAT,
    WOOD,
```

```
    BRICK,
    ORE
} r = BRICK, s = WOOD;
```

Ngắn gọn, `enum` là cách hay để viết code đẹp, có scope, có kiểu, sạch sẽ.

Chapter 23

Pointer III: Pointer tới pointer và hơn thế

Đây là chỗ ta nói về cách dùng pointer ở mức trung cấp và nâng cao. Nếu bạn chưa nắm chắc pointer, xem lại các chương trước về pointer và pointer arithmetic trước khi bắt đầu mấy thứ này.

23.1 Pointer tới pointer

Nếu bạn có thể có pointer tới một biến, và một biến có thể là pointer, thì bạn có thể có pointer tới một biến mà bản thân nó là pointer không?

Có chứ! Cái này là pointer tới pointer, và nó được giữ trong biến kiểu pointer-pointer.

Trước khi mổ xẻ, tôi muốn cố tạo *cảm giác trực giác* về cách pointer tới pointer hoạt động.

Nhớ là pointer chỉ là một con số. Nó là con số đại diện cho một chỉ số trong bộ nhớ máy tính, thường là chỉ số giữ một giá trị mà ta đang quan tâm vì lý do nào đó.

Cái pointer đó, là một con số, thì cũng phải được lưu ở đâu đó. Và chỗ đó là bộ nhớ, như mọi thứ khác[1].

Nhưng vì nó được lưu trong bộ nhớ, nó phải có một chỉ số nơi nó được lưu, đúng không? Cái pointer đó phải có một chỉ số trong bộ nhớ nơi nó nằm. Và chỉ số đó là một con số. Đó là địa chỉ của pointer. Đó là pointer tới pointer.

Hãy bắt đầu với một pointer thường tới `int`, trở lại từ các chương trước:

```
#include <stdio.h>

int main(void)
{
    int x = 3490;  // Type: int
    int *p = &x;   // Type: pointer to an int

    printf("%d\n", *p);  // 3490
}
```

Khá đơn giản, nhỉ? Ta có hai kiểu được đại diện: `int` và `int*`, và ta dựng `p` để trỏ tới `x`. Rồi ta có thể dereference `p` ở dòng 8 và in ra giá trị `3490`.

Nhưng, như ta đã nói, ta có thể có pointer tới bất kỳ biến nào, vậy có phải nghĩa là ta có thể có pointer tới `p` không?

Nói cách khác, biểu thức này có kiểu gì?

[1] Có chút lắt léo với giá trị được lưu chỉ trong thanh ghi, nhưng ở đây ta có thể bỏ qua an toàn. Với lại spec C không có quan điểm gì về mấy chuyện "thanh ghi" đó ngoài từ khóa `register`, mà phần mô tả của nó cũng không nhắc tới thanh ghi.

```
int x = 3490;  // Type: int
int *p = &x;   // Type: pointer to an int

&p  // <-- What type is the address of p? AKA a pointer to p?
```

Nếu `x` là `int`, thì `&x` là pointer tới `int` mà ta đã lưu trong `p` có kiểu `int*`. Theo được chứ? (Đọc lại đoạn này cho đến khi hiểu!)

Và do đó `&p` là pointer tới `int*`, hay còn gọi là "pointer tới pointer tới `int`". Hay là "`int`-pointer-pointer".

Hiểu rồi chứ? (Đọc lại đoạn trước cho đến khi hiểu!)

Ta viết kiểu này với hai dấu sao: `int **`. Xem nó trong hành động.

```
#include <stdio.h>

int main(void)
{
    int x = 3490;  // Type: int
    int *p = &x;   // Type: pointer to an int
    int **q = &p;  // Type: pointer to pointer to int

    printf("%d %d\n", *p, **q);  // 3490 3490
}
```

Ta bịa ra vài địa chỉ giả cho các giá trị trên làm ví dụ và xem ba biến đó có thể trông thế nào trong bộ nhớ. Các giá trị địa chỉ dưới đây do tôi bịa cho có ví dụ:

Biến	Lưu tại địa chỉ	Giá trị lưu ở đó
`x`	`28350`	`3490`, giá trị từ code
`p`	`29122`	`28350`, địa chỉ của `x`!
`q`	`30840`	`29122`, địa chỉ của `p`!

Thật vậy, hãy thử thật trên máy tôi[2] và in các giá trị pointer bằng `%p` và tôi sẽ lập bảng đó lại với tham chiếu thật (in bằng hex).

Biến	Lưu tại địa chỉ	Giá trị lưu ở đó
`x`	`0x7ffd96a07b94`	`3490`, giá trị từ code
`p`	`0x7ffd96a07b98`	`0x7ffd96a07b94`, địa chỉ của `x`!
`q`	`0x7ffd96a07ba0`	`0x7ffd96a07b98`, địa chỉ của `p`!

Bạn có thể thấy mấy địa chỉ này giống nhau trừ byte cuối, nên chỉ cần để ý byte đó.

Trên hệ của tôi, `int` chiếm 4 byte, vì vậy ta thấy địa chỉ tăng thêm 4 từ `x` sang `p`[3] rồi tăng thêm 8 từ `p` sang `q`. Trên hệ của tôi, mọi pointer chiếm 8 byte.

Có quan trọng chuyện nó là `int*` hay `int**` không? Cái nào nhiều byte hơn cái nào? Không hề! Nhớ rằng mọi pointer đều là địa chỉ, tức là chỉ số vào bộ nhớ. Và trên máy tôi ta có thể biểu diễn một chỉ số bằng 8 byte, chẳng liên quan cái gì được lưu ở chỉ số đó.

Giờ để ý xem ta đã làm gì ở dòng 9 của ví dụ trước: ta *dereference hai lần* `q` để quay lại được `3490`.

Đây là điểm quan trọng về pointer và pointer tới pointer:

[2] Trên máy bạn rất có thể ra số khác.

[3] Spec không nói gì chuyện cái này sẽ luôn hoạt động kiểu này, nhưng tình cờ trên máy tôi nó vậy.

- Bạn có thể lấy pointer tới bất cứ thứ gì bằng `&` (kể cả tới một pointer!)
- Bạn có thể lấy thứ mà một pointer đang trỏ tới bằng `*` (kể cả một pointer!)

Vậy bạn có thể nghĩ `&` được dùng để tạo pointer, còn `*` thì ngược lại, đi theo chiều ngược với `&`, để tới được thứ được trỏ tới.

Về kiểu, mỗi lần bạn `&`, cái đó thêm một mức pointer vào kiểu.

Nếu bạn có	Rồi bạn chạy	Kiểu kết quả là
`int x`	`&x`	`int *`
`int *x`	`&x`	`int **`
`int **x`	`&x`	`int ***`
`int ***x`	`&x`	`int ****`

Và mỗi lần bạn dereference (`*`), nó làm ngược lại:

Nếu bạn có	Rồi bạn chạy	Kiểu kết quả là
`int ****x`	`*x`	`int ***`
`int ***x`	`*x`	`int **`
`int **x`	`*x`	`int *`
`int *x`	`*x`	`int`

Lưu ý bạn có thể dùng nhiều `*` liên tiếp để dereference nhanh, y như trong ví dụ code với `**q` ở trên. Mỗi cái bóc đi một lớp gián tiếp.

Nếu bạn có	Rồi bạn chạy	Kiểu kết quả là
`int ****x`	`***x`	`int *`
`int ***x`	`**x`	`int *`
`int **x`	`**x`	`int`

Tổng quát, `&*E == E`[4]. Dereference "hoàn tác" address-of.

Nhưng `&` thì không chạy kiểu đó, bạn chỉ làm từng cái một được, và phải lưu kết quả vào một biến trung gian:

```
int x = 3490;     // Type: int
int *p = &x;      // Type: int *, pointer to an int
int **q = &p;     // Type: int **, pointer to pointer to int
int ***r = &q;    // Type: int ***, pointer to pointer to pointer to int
int ****s = &r;   // Type: int ****, you get the idea
int *****t = &s;  // Type: int *****
```

23.1.1 Pointer-pointer và `const`

Nếu bạn còn nhớ, khai báo pointer kiểu vầy:

```
int *const p;
```

nghĩa là bạn không thể sửa `p`. Cố `p++` sẽ cho lỗi lúc compile.

Nhưng nó hoạt động ra sao với `int **` hay `int ***`? `const` đi đâu và nghĩa là gì?

Bắt đầu từ phần đơn giản. `const` ngay cạnh tên biến ám chỉ chính biến đó. Vậy nếu bạn muốn một `int***` mà không thể đổi, bạn có thể làm vầy:

[4] Kể cả khi `E` là `NULL`, lạ thay.

```
int ***const p;

p++;  // Not allowed
```

Nhưng đây là chỗ mọi thứ hơi lạ.

Lỡ ta gặp tình huống này thì sao:

```
int main(void)
{
    int x = 3490;
    int *const p = &x;
    int **q = &p;
}
```

Khi tôi build cái đó, tôi nhận được cảnh báo:

```
warning: initialization discards ‘const’ qualifier from pointer target type
    7 |     int **q = &p;
      |               ^
```

Chuyện gì vậy? Compiler đang báo ta rằng ta có một biến `const`, và ta đang gán giá trị của nó vào biến khác không `const` theo cùng cách. Tính "`const`" bị bỏ đi, mà có lẽ đó không phải thứ ta muốn.

Kiểu của `p` là `int *const p`, nên `&p` có kiểu `int *const *`. Và ta cố gán cái đó vào `q`.

Nhưng `q` là `int **`! Một kiểu với tính `const` khác ở dấu `*` đầu tiên! Nên ta nhận cảnh báo là `const` trong `int *const *` của `p` đang bị bỏ qua và vứt đi.

Ta có thể sửa bằng cách đảm bảo kiểu của `q` ít nhất cũng `const` bằng `p`.

```
int x = 3490;
int *const p = &x;
int *const *q = &p;
```

Và giờ ta vui rồi.

Ta có thể làm `q` `const` hơn nữa. Hiện tại ở trên, ta đang nói, "`q` bản thân không `const`, nhưng thứ nó trỏ tới là `const`." Nhưng ta có thể làm cả hai `const`:

```
int x = 3490;
int *const p = &x;
int *const *const q = &p;  // More const!
```

Và chạy ngon. Giờ ta không thể sửa `q`, hay cái pointer mà `q` trỏ tới.

23.2 Giá trị nhiều byte

Ta đã bóng gió chuyện này ở vài chỗ trước đây, nhưng rõ ràng không phải mọi giá trị đều lưu vừa trong một byte bộ nhớ. Mọi thứ chiếm nhiều byte bộ nhớ (giả sử chúng không phải `char`). Bạn có thể biết chiếm bao nhiêu byte bằng `sizeof`. Và bạn có thể biết địa chỉ nào trong bộ nhớ là byte *đầu tiên* của đối tượng bằng toán tử chuẩn `&`, luôn trả về địa chỉ của byte đầu.

Và đây là sự thật thú vị khác! Nếu bạn duyệt qua các byte của bất kỳ đối tượng nào, bạn nhận được *biểu diễn đối tượng* của nó. Hai thứ có cùng biểu diễn đối tượng trong bộ nhớ thì bằng nhau.

Nếu bạn muốn duyệt biểu diễn đối tượng, bạn nên làm với pointer tới `unsigned char`.

Hãy làm phiên bản riêng của `memcpy()` [5] làm đúng chuyện này:

```
void *my_memcpy(void *dest, const void *src, size_t n)
{
    // Make local variables for src and dest, but of type unsigned char

    const unsigned char *s = src;
    unsigned char *d = dest;

    while (n-- > 0)   // For the given number of bytes
        *d++ = *s++;  // Copy source byte to dest byte

    // Most copy functions return a pointer to the dest as a convenience
    // to the caller

    return dest;
}
```

(Trong đó cũng có ví dụ post-increment và post-decrement hay để bạn nghiên cứu.)

Quan trọng mà lưu ý là phiên bản ở trên có lẽ kém hiệu quả hơn cái đi kèm với hệ của bạn.

Nhưng bạn có thể truyền pointer tới bất cứ thứ gì vào nó, và nó sẽ copy các đối tượng đó. Có thể là `int*`, `struct animal*`, hay bất cứ gì.

Làm thêm ví dụ nữa in ra các byte biểu diễn đối tượng của một `struct` để ta xem có padding trong đó không và nó có giá trị gì[6].

```
#include <stdio.h>

struct foo {
    char a;
    int b;
};

int main(void)
{
    struct foo x = {0x12, 0x12345678};
    unsigned char *p = (unsigned char *)&x;

    for (size_t i = 0; i < sizeof x; i++) {
        printf("%02X\n", p[i]);
    }
}
```

Cái ta có đó là `struct foo` được dựng sao cho khuyến khích compiler chèn byte padding vào (dù nó không phải làm). Rồi ta lấy `unsigned char *` tới byte đầu của biến `struct foo` tên `x`.

Từ đó, ta chỉ cần biết `sizeof x` là có thể lặp qua từng ấy byte, in các giá trị ra (bằng hex cho dễ đọc).

Chạy cái này cho ra một đống số. Tôi chú thích bên dưới để chỉ ra các giá trị được lưu ở đâu:

```
12  | x.a == 0x12

AB  |
BF  | padding bytes with "random" value
26  |
```

[5] https://beej.us/guide/bgclr/html/split/stringref.html#man-memcpy

[6] Compiler C không bị bắt buộc phải chèn byte padding, và giá trị của bất cứ byte padding nào được chèn là không xác định.

```
78  | x.b == 0x12345678
56  |
34  |
12  |
```

Trên mọi hệ, `sizeof(char)` là 1, và ta thấy byte đầu tiên ở đầu output giữ giá trị `0x12` mà ta đã lưu ở đó.

Rồi ta có vài byte padding, với tôi, mấy cái này thay đổi giữa các lần chạy.

Cuối cùng, trên hệ của tôi, `sizeof(int)` là 4, và ta thấy 4 byte đó ở cuối. Chú ý chúng chính là các byte trong giá trị hex `0x12345678`, nhưng kỳ lạ là ngược thứ tự[7].

Vậy đó là một cái nhìn hé lộ vào các byte của một thực thể phức tạp hơn trong bộ nhớ.

23.3 Pointer `NULL` và số 0

Mấy thứ này có thể dùng thay qua lại:

- `NULL`
- `0`
- `'\0'`
- `(void *)0`

Cá nhân tôi, tôi luôn dùng `NULL` khi tôi muốn nói `NULL`, nhưng bạn có thể thỉnh thoảng thấy các biến thể khác. Dù `'\0'` (byte với mọi bit bằng 0) cũng so sánh ra bằng, nhưng đem so nó với một pointer thì *kỳ cục*, bạn nên so `NULL` với pointer. (Dĩ nhiên, nhiều lần khi xử lý chuỗi, bạn so sánh *cái mà pointer đang trỏ tới* với `'\0'`, và cái đó thì đúng.)

`0` được gọi là *null pointer constant*, và khi so sánh hay gán vào một pointer khác, nó được chuyển thành null pointer cùng kiểu.

23.4 Pointer như số nguyên

Bạn có thể cast pointer thành số nguyên và ngược lại (vì pointer chỉ là chỉ số vào bộ nhớ), nhưng có lẽ bạn chỉ cần làm điều này nếu đang làm mấy thứ phần cứng mức thấp. Kết quả của những trò như vậy là implementation-defined, nên không portable. Và *chuyện lạ* có thể xảy ra.

Tuy nhiên, C có đảm bảo một chuyện: bạn có thể chuyển một pointer sang kiểu `uintptr_t` và bạn sẽ có thể chuyển nó về lại pointer mà không mất dữ liệu.

`uintptr_t` được định nghĩa trong `<stdint.h>`[8].

Thêm nữa, nếu bạn muốn có dấu, bạn có thể dùng `intptr_t` với tác dụng y vậy.

23.5 Cast pointer sang pointer khác

Chỉ có một cách chuyển pointer an toàn:

1. Chuyển sang `intptr_t` hoặc `uintptr_t`.
2. Chuyển sang và từ `void*`.

HAI! Có hai cách chuyển pointer an toàn.

3. Chuyển sang và từ `char*` (hoặc `signed char*` / `unsigned char*`).

BA! Có ba cách chuyển an toàn!

[7] Cái này sẽ khác nhau tùy kiến trúc, nhưng hệ của tôi *little endian*, nghĩa là byte nhỏ nhất của số được lưu trước. Hệ *big endian* sẽ có `12` trước và `78` sau. Nhưng spec không ra lệnh gì về biểu diễn này.

[8] Đây là tính năng tùy chọn, nên nó có thể không có, nhưng có lẽ có.

4. Chuyển sang và từ pointer tới `struct` và pointer tới thành viên đầu tiên của nó, và ngược lại.

BỐN! Có bốn cách chuyển an toàn!

Nếu bạn cast sang pointer kiểu khác rồi truy cập đối tượng nó trỏ tới, hành vi là không xác định do một thứ gọi là *strict aliasing*.

Aliasing thuần túy nghĩa là khả năng có nhiều hơn một cách để truy cập cùng một đối tượng. Các điểm truy cập là alias của nhau.

Strict aliasing nói bạn chỉ được phép truy cập một đối tượng qua pointer tới *kiểu tương thích* với đối tượng đó.

Ví dụ, cái này chắc chắn được phép:

```
int a = 1;
int *p = &a;
```

`p` là pointer tới `int`, và nó trỏ tới một kiểu tương thích, cụ thể là `int`, nên ta ngon.

Nhưng cái dưới không ổn vì `int` và `float` không phải là các kiểu tương thích:

```
int a = 1;
float *p = (float *)&a;
```

Đây là chương trình demo làm chút aliasing. Nó lấy biến `v` kiểu `int32_t` và alias nó thành pointer tới một `struct words`. `struct` đó có hai `int16_t`. Các kiểu này không tương thích, nên ta đang vi phạm luật strict aliasing. Compiler sẽ giả định rằng hai pointer này không bao giờ trỏ tới cùng một đối tượng, nhưng ta đang làm cho chúng trỏ tới. Điều đó thật hư của ta.

Hãy xem có làm vỡ được gì không.

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

struct words {
    int16_t v[2];
};

void fun(int32_t *pv, struct words *pw)
{
    for (int i = 0; i < 5; i++) {
        (*pv)++;

        // Print the 32-bit value and the 16-bit values:

        printf("%x, %x-%x\n", *pv, pw->v[1], pw->v[0]);
    }
}

int main(void)
{
    int32_t v = 0x12345678;

    struct words *pw = (struct words *)&v;  // Violates strict aliasing

    fun(&v, pw);
}
```

Thấy cách tôi truyền hai pointer không tương thích vào `fun()` chứ? Một kiểu là `int32_t*` và cái kia là `struct words*`.

Nhưng cả hai trỏ tới cùng đối tượng: giá trị 32-bit được khởi tạo bằng `0x12345678`.

Vậy nếu ta nhìn các field trong `struct words`, ta sẽ thấy hai nửa 16-bit của con số đó. Đúng không?

Và trong vòng lặp `fun()`, ta tăng pointer tới `int32_t`. Chỉ vậy. Nhưng vì `struct` trỏ tới cùng bộ nhớ đó, nó cũng sẽ được cập nhật thành cùng giá trị.

Vậy chạy thử và ta nhận được cái này, với giá trị 32-bit bên trái và hai phần 16-bit bên phải. Phải khớp nhau[9]:

```
12345679, 1234-5679
1234567a, 1234-567a
1234567b, 1234-567b
1234567c, 1234-567c
1234567d, 1234-567d
```

và nó có khớp... *CHO TỚI NGÂY MAI!*

Thử compile bằng GCC với `-O3` và `-fstrict-aliasing`:

```
12345679, 1234-5678
1234567a, 1234-5679
1234567b, 1234-567a
1234567c, 1234-567b
1234567d, 1234-567c
```

Lệch nhau một đơn vị! Mà chúng trỏ tới cùng bộ nhớ! Sao có thể? Giải đáp: aliasing bộ nhớ kiểu đó là hành vi không xác định. *Bất cứ chuyện gì cũng có thể xảy ra*, nhưng không phải theo nghĩa tốt.

Nếu code của bạn vi phạm luật strict aliasing, việc nó chạy hay không phụ thuộc vào cách ai đó quyết định compile nó. Và đó là điều đáng tiếc vì nó ngoài tầm kiểm soát của bạn. Trừ khi bạn là một thứ thần thánh toàn năng nào đó.

Khó có khả năng vậy, tiếc.

GCC có thể bị ép không dùng luật strict aliasing bằng `-fno-strict-aliasing`. Compile chương trình demo ở trên với `-O3` và cờ này khiến output đúng như mong đợi.

Cuối cùng, *type punning* là dùng pointer của các kiểu khác nhau để nhìn cùng dữ liệu. Trước khi có strict aliasing, mấy chuyện kiểu này khá phổ biến:

```
int a = 0x12345678;
short b = *((short *)&a);   // Violates strict aliasing
```

Nếu bạn muốn làm type punning (tương đối) an toàn, xem phần Union và Type Punning.

23.6 Hiệu của pointer

Như bạn đã biết từ phần pointer arithmetic, bạn có thể trừ một pointer từ pointer khác[10] để có hiệu giữa chúng theo số phần tử mảng.

Giờ *kiểu của cái hiệu đó* tùy implementation quyết định, nên có thể khác nhau giữa các hệ.

Để portable hơn, bạn có thể lưu kết quả vào biến kiểu `ptrdiff_t` định nghĩa trong `<stddef.h>`.

[9] Tôi in hai giá trị 16-bit theo thứ tự đảo vì tôi đang ở máy little-endian và làm vậy dễ đọc hơn ở đây.
[10] Giả sử chúng trỏ tới cùng một đối tượng mảng.

```
int cats[100];

int *f = cats + 20;
int *g = cats + 60;

ptrdiff_t d = g - f;  // difference is 40
```

Và bạn có thể in nó bằng cách thêm `t` đầu format specifier cho số nguyên:

```
printf("%td\n", d);  // Print decimal: 40
printf("%tX\n", d);  // Print hex:     28
```

23.7 Pointer tới hàm

Hàm chỉ là tập hợp lệnh máy trong bộ nhớ, nên không có lý do gì ta không lấy được pointer tới lệnh đầu tiên của hàm.

Và rồi gọi nó.

Điều này có thể hữu ích khi truyền một pointer tới hàm vào một hàm khác như đối số. Rồi hàm thứ hai có thể gọi bất cứ cái gì được truyền vào.

Tuy nhiên, phần khó với mấy cái này là C cần biết kiểu của biến là pointer tới hàm.

Và nó thật sự muốn biết mọi chi tiết.

Kiểu như "đây là pointer tới hàm nhận hai đối số `int` và trả về `void`".

Viết hết mớ đó ra sao để khai báo được biến?

Hóa ra nó trông rất giống function prototype, chỉ thêm vài cặp ngoặc:

```
// Declare p to be a pointer to a function.
// This function returns a float, and takes two ints as arguments.

float (*p)(int, int);
```

Lưu ý bạn không cần đặt tên cho tham số. Nhưng bạn có thể nếu muốn, chúng chỉ bị bỏ qua.

```
// Declare p to be a pointer to a function.
// This function returns a float, and takes two ints as arguments.

float (*p)(int a, int b);
```

Giờ ta đã biết cách khai báo biến, làm sao biết gán gì vào? Làm sao lấy địa chỉ của một hàm?

Hóa ra có lối tắt y như lấy pointer tới mảng: bạn có thể chỉ viết tên hàm trần không có ngoặc. (Bạn có thể thêm `&` đằng trước nếu thích, nhưng không cần và không idiomatic.)

Một khi có pointer tới hàm, bạn có thể gọi nó bằng cách thêm ngoặc và danh sách đối số.

Làm ví dụ đơn giản tôi đặt hẳn alias cho hàm bằng cách dựng một pointer tới nó. Rồi ta gọi nó.

Code này in ra `3490`:

```
#include <stdio.h>

void print_int(int n)
{
    printf("%d\n", n);
}
```

```
int main(void)
{
    // Assign p to point to print_int:

    void (*p)(int) = print_int;

    p(3490);          // Call print_int via the pointer
}
```

Lưu ý cách kiểu của `p` đại diện cho giá trị trả về và kiểu tham số của `print_int`. Bắt buộc phải thế, không thì C sẽ phàn nàn về kiểu pointer không tương thích.

Thêm một ví dụ nữa cho thấy ta có thể truyền pointer tới hàm như đối số cho hàm khác thế nào.

Ta sẽ viết hàm nhận vài đối số số nguyên, cộng với pointer tới hàm nào đó thao tác trên hai đối số đó. Rồi nó in kết quả.

```
#include <stdio.h>

int add(int a, int b)
{
    return a + b;
}

int mult(int a, int b)
{
    return a * b;
}

void print_math(int (*op)(int, int), int x, int y)
{
    int result = op(x, y);

    printf("%d\n", result);
}

int main(void)
{
    print_math(add, 5, 7);   // 12
    print_math(mult, 5, 7);  // 35
}
```

Dành tí thời gian tiêu hóa chuyện đó. Ý ở đây là ta sẽ truyền một pointer tới hàm vào `print_math()`, và nó sẽ gọi hàm đó để làm vài phép toán.

Bằng cách này ta có thể đổi hành vi của `print_math()` bằng cách truyền hàm khác vào. Bạn thấy ta làm thế ở dòng 22-23 khi truyền vào pointer tới hàm `add` và `mult`, theo thứ tự.

Giờ, ở dòng 13, tôi nghĩ ai cũng đồng ý signature của `print_math()` là một cảnh ngoạn mục. Và, tin hay không, cái này thực ra còn khá thẳng thớm so với vài thứ bạn có thể dựng ra[11].

Nhưng hãy tiêu hóa nó. Hóa ra chỉ có ba tham số, nhưng chúng hơi khó thấy:

```
//                      op             x      y
//              |-----------------|  |---|  |---|
void print_math(int (*op)(int, int), int x, int y)
```

[11]Ngôn ngữ Go lấy cảm hứng cú pháp khai báo kiểu từ điều ngược lại với cái C làm.

Cái đầu, `op`, là pointer tới hàm nhận hai `int` làm đối số và trả về `int`. Cái này khớp signature của cả `add()` lẫn `mult()`.

Cái thứ hai và ba, `x` và `y`, chỉ là tham số `int` chuẩn.

Chậm và có chủ đích, hãy để mắt bạn đi qua signature rồi xác định các phần. Một thứ luôn nhảy ra với tôi là chuỗi `(*op)(`, cặp ngoặc và dấu sao. Đó là dấu hiệu nó là pointer tới hàm.

Cuối cùng, nhảy lại chương *Pointer II* để xem ví dụ pointer-tới-hàm dùng `qsort()` có sẵn.

Chapter 24

Phép bitwise

Mấy phép số này cho phép bạn thao tác từng bit trong biến, cũng hợp vì C là ngôn ngữ mức thấp[1].

Nếu bạn chưa quen với phép bitwise, Wikipedia có bài viết về bitwise khá tốt[2].

24.1 Bitwise AND, OR, XOR và NOT

Với từng phép này, các quy đổi số học thông thường diễn ra trên toán hạng (trong trường hợp này phải là kiểu số nguyên), rồi phép bitwise tương ứng được thực hiện.

Phép	Toán tử	Ví dụ
AND	`&`	`a = b & c`
OR	`|`	`a = b | c`
XOR	`^`	`a = b ^ c`
NOT	`~`	`a = ~c`

Lưu ý chúng giống toán tử Boolean `&&` và `||`.

Mấy cái này có biến thể gán tắt tương tự `+=` và `-=`:

Toán tử	Ví dụ	Tương đương đầy đủ
`&=`	`a &= c`	`a = a & c`
`|=`	`a |= c`	`a = a | c`
`^=`	`a ^= c`	`a = a ^ c`

24.2 Bitwise shift

Với mấy cái này, integer promotion diễn ra trên từng toán hạng (phải là kiểu số nguyên) rồi bitwise shift được thực hiện. Kiểu của kết quả là kiểu của toán hạng trái sau khi promote.

Bit mới được lấp bằng 0, với một ngoại lệ có thể có được nói trong phần hành vi implementation-defined, bên dưới.

Phép	Toán tử	Ví dụ
Shift trái	`<<`	`a = b << c`

[1]Không phải các ngôn ngữ khác không làm vậy, chúng có làm. Thú vị là bao nhiêu ngôn ngữ hiện đại dùng cùng toán tử bitwise như C.

[2]https://en.wikipedia.org/wiki/Bitwise_operation

Phép	Toán tử	Ví dụ
Shift phải	`>>`	`a = b >> c`

Cũng có dạng viết tắt tương tự cho shift:

Toán tử	Ví dụ	Tương đương đầy đủ
`>>=`	`a >>= c`	`a = a >> c`
`<<=`	`a <<= c`	`a = a << c`

Coi chừng hành vi không xác định: không shift số âm, và không shift lớn hơn kích thước của toán hạng trái sau khi promote.

Cũng coi chừng hành vi implementation-defined: nếu bạn shift phải một số âm, kết quả là implementation-defined. (Shift phải một `int` có dấu thì hoàn toàn ổn, chỉ cần nó là số dương.)

Chapter 25

Hàm variadic

Variadic là từ kêu kêu để chỉ hàm nhận số đối số tuỳ ý.

Hàm thường nhận một số đối số cụ thể, ví dụ:

```
int add(int x, int y)
{
    return x + y;
}
```

Bạn chỉ có thể gọi nó với đúng hai đối số tương ứng tham số `x` và `y`.

```
add(2, 3);
add(5, 12);
```

Nhưng nếu thử nhiều hơn, compiler không cho:

```
add(2, 3, 4);  // ERROR
add(5);        // ERROR
```

Hàm variadic vượt qua giới hạn này ở một mức nào đó.

Ta đã thấy một ví dụ nổi tiếng trong `printf()`! Bạn có thể truyền đủ kiểu thứ vào nó.

```
printf("Hello, world!\n");
printf("The number is %d\n", 2);
printf("The number is %d and pi is %f\n", 2, 3.14159);
```

Nó có vẻ chẳng quan tâm bạn đưa bao nhiêu đối số!

Ừ, không hẳn. Không đối số nào sẽ cho lỗi:

```
printf();  // ERROR
```

Điều này dẫn ta tới một giới hạn của hàm variadic trong C: chúng phải có ít nhất một đối số.

Nhưng ngoài chuyện đó, chúng khá linh hoạt, thậm chí cho phép đối số có kiểu khác nhau như `printf()` làm.

Xem chúng hoạt động sao nhé!

25.1 Dấu ba chấm trong signature hàm

Vậy nó chạy thế nào, về cú pháp?

Việc bạn làm là đặt mọi đối số *bắt buộc* phải truyền vào trước (và nhớ phải có ít nhất một) và sau đó, bạn đặt `...`. Như vầy:

```
void func(int a, ...)   // Literally 3 dots here
```

Đây là ít code để demo:

```
#include <stdio.h>

void func(int a, ...)
{
    printf("a is %d\n", a);  // Prints "a is 2"
}

int main(void)
{
    func(2, 3, 4, 5, 6);
}
```

Rồi, hay, ta lấy được đối số đầu ở biến `a`, nhưng còn phần đối số còn lại thì sao? Làm sao tới được chúng?

Đây là chỗ vui bắt đầu!

25.2 Lấy các đối số dư

Bạn sẽ muốn include `<stdarg.h>` để làm mấy chuyện này.

Trước tiên, ta sẽ dùng một biến đặc biệt kiểu `va_list` (variable argument list) để theo dõi ta đang truy cập biến nào tại thời điểm đó.

Ý tưởng là ta bắt đầu xử lý đối số bằng một lời gọi `va_start()`, xử lý từng đối số một bằng `va_arg()`, rồi, khi xong, kết bằng `va_end()`.

Khi bạn gọi `va_start()`, bạn cần truyền *tham số có tên cuối cùng* (cái ngay trước `...`) để nó biết chỗ cần bắt đầu tìm các đối số thêm.

Và khi bạn gọi `va_arg()` để lấy đối số kế, bạn phải cho nó biết kiểu của đối số kế tiếp cần lấy.

Đây là demo cộng lại một số tuỳ ý các số nguyên. Đối số đầu là số lượng số nguyên cần cộng. Ta sẽ dùng nó để biết phải gọi `va_arg()` bao nhiêu lần.

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

int add(int count, ...)
{
    int total = 0;
    va_list va;

    va_start(va, count);   // Start with arguments after "count"

    for (int i = 0; i < count; i++) {
        int n = va_arg(va, int);   // Get the next int

        total += n;
    }
```

```
    va_end(va);  // All done

    return total;
}

int main(void)
{
    printf("%d\n", add(4, 6, 2, -4, 17));  // 6 + 2 - 4 + 17 = 21
    printf("%d\n", add(2, 22, 44));        // 22 + 44 = 66
}
```

(Lưu ý khi gọi `printf()`, nó dùng số `%d` (hay bất cứ thứ gì) trong chuỗi format để biết còn bao nhiêu đối số nữa!)

Nếu cú pháp của `va_arg()` trông lạ với bạn (vì có tên kiểu lơ lửng trong đó), bạn không đơn độc. Chúng được cài đặt bằng macro preprocessor để có được mọi phép màu cần thiết.

25.3 Chức năng `va_list`

Cái biến `va_list` ta đang dùng ở trên là gì? Đó là biến mờ[1] giữ thông tin về việc ta sẽ lấy đối số nào kế tiếp bằng `va_arg()`. Thấy cách ta gọi `va_arg()` lặp đi lặp lại đấy? Biến `va_list` là chỗ giữ chỗ đang theo dõi tiến độ cho tới giờ.

Nhưng ta phải khởi tạo biến đó bằng một giá trị hợp lý. Đó là lúc `va_start()` ra sân.

Khi ta gọi `va_start(va, count)` ở trên, ta đang nói, "Khởi tạo biến `va` để trỏ tới đối số biến *ngay sau* `count`."

Và đó là *lý do* ta cần có ít nhất một biến có tên trong danh sách đối số[2].

Một khi có pointer tới tham số ban đầu, bạn có thể dễ dàng lấy các giá trị đối số sau bằng cách gọi `va_arg()` lặp đi lặp lại. Khi làm vậy, bạn phải truyền vào biến `va_list` của mình (để nó tiếp tục theo dõi bạn đang ở đâu), cùng với kiểu của đối số bạn sắp copy ra.

Tùy bạn, người lập trình, nghĩ ra kiểu bạn sẽ truyền cho `va_arg()`. Trong ví dụ ở trên, ta chỉ làm `int`. Nhưng trong trường hợp `printf()`, nó dùng format specifier để xác định kiểu nào cần lấy kế tiếp.

Và khi xong, gọi `va_end()` để kết lại. Bạn **phải** (spec nói) gọi cái này trên một biến `va_list` cụ thể trước khi bạn quyết định gọi lại `va_start()` hay `va_copy()` trên nó lần nữa. Tôi biết ta chưa nói về `va_copy()`.

Vậy tiến trình chuẩn là:

- `va_start()` để khởi tạo biến `va_list` của bạn
- Lặp lại `va_arg()` để lấy giá trị
- `va_end()` để kết biến `va_list` của bạn

Tôi cũng có nhắc `va_copy()` ở trên; nó làm bản sao biến `va_list` của bạn ở đúng cùng trạng thái. Tức là, nếu bạn chưa bắt đầu dùng `va_arg()` với biến nguồn, biến mới cũng chưa bắt đầu. Nếu bạn đã ngốn 5 biến bằng `va_arg()` cho tới giờ, bản sao cũng phản ánh y vậy.

`va_copy()` có thể hữu ích nếu bạn cần quét trước qua đối số nhưng vẫn cần nhớ vị trí hiện tại.

[1] Nghĩa là đám dev thấp cổ bé họng như ta không phải biết trong đó có gì hay ý nghĩa gì. Spec không ra lệnh chi tiết nó là gì.

[2] Thành thật mà nói, loại bỏ giới hạn này khỏi ngôn ngữ là khả thi, nhưng ý tưởng là các macro `va_start()`, `va_arg()` và `va_end()` có thể viết được bằng C. Và để làm được điều đó, ta cần cách nào đó khởi tạo một pointer tới vị trí của tham số đầu. Và để làm điều đó, ta cần *tên* của tham số đầu. Sẽ cần mở rộng ngôn ngữ để làm được việc này, và tới giờ ủy ban chưa tìm ra lý do chính đáng.

25.4 Hàm thư viện dùng `va_list`

Một trong những cách dùng khác của mấy cái này khá hay: viết biến thể `printf()` tuỳ ý của riêng bạn. Sẽ đau đầu nếu phải xử mọi format specifier đó phải không? Cả tỷ cái?

May thay, có các biến thể `printf()` nhận một `va_list` đang hoạt động làm đối số. Bạn có thể dùng chúng để bọc lại và tự làm `printf()` riêng!

Các hàm này bắt đầu bằng chữ `v`, như `vprintf()`, `vfprintf()`, `vsprintf()` và `vsnprintf()`. Về cơ bản là mọi bản hit kinh điển của `printf()` chỉ thêm `v` đằng trước.

Hãy làm hàm `my_printf()` chạy y `printf()` chỉ khác là nhận thêm một đối số đầu.

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

int my_printf(int serial, const char *format, ...)
{
    va_list va;
    int rv;

    // Do my custom work
    printf("The serial number is: %d\n", serial);

    // Then pass the rest off to vprintf()
    va_start(va, format);
    rv = vprintf(format, va);
    va_end(va);

    return rv;
}

int main(void)
{
    int x = 10;
    float y = 3.2;

    my_printf(3490, "x is %d, y is %f\n", x, y);
}
```

Thấy ta làm gì đó chưa? Ở dòng 12-14 ta mở một biến `va_list` mới, rồi cứ thế truyền thẳng vào `vprintf()`. Và nó biết ngay phải làm gì với nó, vì nó có sẵn mọi đầu óc của `printf()` gài vào.

Tuy vậy, ta vẫn phải gọi `va_end()` khi xong, nên đừng quên!

25.5 Bẫy macro variadic

Như tôi đã nhắc, `va_start()` và `va_end()` có thể là macro. Một hệ quả của chuyện này có thể là chúng có tiềm năng mở ra một scope cục bộ mới. (Tức là, nếu `va_start()` có `{` và `va_end()` chứa `}`.)

Nên ta cần cảnh giác với chuyện scope có thể gặp vấn đề. Lấy ví dụ sau:

```
va_start(va, format);          // may contain {
int rv = vprintf(format, va);
va_end(va);                    // may contain }
```

```
return rv;
```

Nếu `va_start()` mở scope mới, `rv` sẽ cục bộ trong scope đó rồi câu `return` sẽ fail. Nhưng chuyện này sẽ âm thầm chỉ xảy ra trên các compiler tình cờ làm vậy với macro `va`.

Chapter 26

Locale và quốc tế hoá

Localization (bản địa hoá) là quá trình làm cho app của bạn sẵn sàng hoạt động tốt ở các locale (hay quốc gia) khác nhau.

Như bạn có thể biết, không phải ai cũng dùng cùng ký tự cho dấu thập phân hay cho dấu phân cách hàng nghìn, hay cho đơn vị tiền tệ.

Các locale này có tên, và bạn có thể chọn một cái để dùng. Ví dụ, locale Mỹ có thể viết một con số như:

100,000.00

Còn ở Brazil, cùng số đó có thể được viết với dấu phẩy và dấu chấm đổi chỗ:

100.000,00

Chuyện này dễ dàng hơn khi bạn viết code sao cho dễ chuyển sang các quốc tịch khác!

Ừ, kiểu kiểu. Hoá ra C chỉ có đúng một locale sẵn, và nó bị giới hạn. Spec chừa ra khá nhiều chỗ mập mờ ở đây; khó mà thật sự portable hoàn toàn.

Nhưng ta sẽ cố gắng hết sức!

26.1 Đặt localization, nhanh và bẩn

Với các lời gọi này, include `<locale.h>`.

Về cơ bản chỉ có một việc bạn có thể làm portable ở đây khi khai báo một locale cụ thể. Đây rất có thể là điều bạn muốn làm nếu định đụng tới locale:

```
setlocale(LC_ALL, "");  // Use this environment's locale for everything
```

Bạn sẽ muốn gọi nó để chương trình được khởi tạo với locale hiện tại của bạn.

Đi vào chi tiết hơn, có một chuyện nữa bạn làm được mà vẫn portable:

```
setlocale(LC_ALL, "C");  // Use the default C locale
```

nhưng cái đó được gọi mặc định mỗi lần chương trình của bạn khởi chạy, nên không mấy khi cần tự gọi.

Trong chuỗi thứ hai đó, bạn có thể chỉ định bất kỳ locale nào được hệ thống của bạn hỗ trợ. Chuyện này hoàn toàn phụ thuộc hệ, nên sẽ khác nhau. Trên hệ của tôi, tôi có thể chỉ định cái này:

```
setlocale(LC_ALL, "en_US.UTF-8");  // Non-portable!
```

Và cái đó sẽ chạy. Nhưng nó chỉ portable sang các hệ có đúng cùng tên đó cho đúng locale đó, và bạn không thể bảo đảm được.

Bằng cách truyền chuỗi rỗng (`""`) làm đối số thứ hai, bạn đang nói với C, "Này, tự tìm xem locale hiện tại trên hệ này là gì để tôi khỏi phải nói cho."

26.2 Lấy thiết lập locale cho tiền tệ

Vì di chuyển mấy tờ giấy xanh hứa hẹn là chìa khoá tới hạnh phúc[1], hãy nói về locale cho tiền tệ. Khi bạn viết code portable, bạn phải biết phải gõ gì cho tiền mặt, đúng không? Dù nó là "$", "€", "¥", hay "£".

Làm sao viết code đó mà không phát điên? May thay, khi bạn gọi `setlocale(LC_ALL, "")` , bạn có thể tra mấy cái này bằng một lời gọi `localeconv()` :

```
struct lconv *x = localeconv();
```

Hàm này trả về pointer tới một `struct lconv` được cấp phát tĩnh có mọi thông tin ngon lành bạn đang tìm.

Đây là các field của `struct lconv` và nghĩa của chúng.

Trước hết, vài quy ước. `_p_` nghĩa là "positive" (dương), `_n_` nghĩa là "negative" (âm), và `int_` nghĩa là "international" (quốc tế). Dù nhiều cái có kiểu `char` hoặc `char*` , phần lớn (hoặc các chuỗi chúng trỏ tới) thật ra được xem như số nguyên[2].

Trước khi đi tiếp, biết rằng `CHAR_MAX` (từ `<limits.h>`) là giá trị tối đa lưu được trong một `char` . Và nhiều giá trị `char` dưới đây dùng nó để cho biết giá trị không có ở locale đó.

Field	Mô tả
`char *mon_decimal_point`	Ký tự dấu thập phân cho tiền, ví dụ `"."` .
`char *mon_thousands_sep`	Ký tự phân cách hàng nghìn cho tiền, ví dụ `","` .
`char *mon_grouping`	Mô tả cách nhóm cho tiền (xem bên dưới).
`char *positive_sign`	Dấu dương cho tiền, ví dụ `"+"` hoặc `""` .
`char *negative_sign`	Dấu âm cho tiền, ví dụ `"-"` .
`char *currency_symbol`	Ký hiệu tiền tệ, ví dụ `"$"` .
`char frac_digits`	Khi in lượng tiền, in bao nhiêu chữ số sau dấu thập phân, ví dụ `2` .
`char p_cs_precedes`	`1` nếu `currency_symbol` đứng trước giá trị cho lượng tiền không âm, `0` nếu sau.
`char n_cs_precedes`	`1` nếu `currency_symbol` đứng trước giá trị cho lượng tiền âm, `0` nếu sau.
`char p_sep_by_space`	Quy định cách ngăn cách `currency symbol` khỏi giá trị cho lượng không âm (xem bên dưới).
`char n_sep_by_space`	Quy định cách ngăn cách `currency symbol` khỏi giá trị cho lượng âm (xem bên dưới).
`char p_sign_posn`	Quy định vị trí của `positive_sign` cho giá trị không âm.
`char n_sign_posn`	Quy định vị trí của `positive_sign` cho giá trị âm.
`char *int_curr_symbol`	Ký hiệu tiền tệ quốc tế, ví dụ `"USD "` .
`char int_frac_digits`	Giá trị quốc tế cho `frac_digits` .
`char int_p_cs_precedes`	Giá trị quốc tế cho `p_cs_precedes` .
`char int_n_cs_precedes`	Giá trị quốc tế cho `n_cs_precedes` .
`char int_p_sep_by_space`	Giá trị quốc tế cho `p_sep_by_space` .
`char int_n_sep_by_space`	Giá trị quốc tế cho `n_sep_by_space` .
`char int_p_sign_posn`	Giá trị quốc tế cho `p_sign_posn` .

[1]" Hành tinh này có, hay đúng hơn, đã có một vấn đề, đó là: phần lớn những người sống trên nó không hạnh phúc gần như suốt cả thời gian. Nhiều giải pháp được đề xuất cho vấn đề này, nhưng phần lớn đều liên quan đến việc di chuyển các tờ giấy xanh nhỏ, kỳ cục là nhìn chung chẳng phải mấy tờ giấy xanh nhỏ không hạnh phúc." The Hitchhiker's Guide to the Galaxy, Douglas Adams

[2]Nhớ là `char` chỉ là số nguyên cỡ một byte.

Field	Mô tả
`char int_n_sign_posn`	Giá trị quốc tế cho `n_sign_posn`.

26.2.1 Nhóm chữ số cho tiền tệ

Được rồi, cái này hơi chập cheng. `mon_grouping` là `char*`, nên bạn có thể nghĩ nó là chuỗi. Nhưng trong trường hợp này, không, nó thực ra là mảng các `char`. Nó luôn phải kết thúc bằng `0` hoặc `CHAR_MAX`.

Các giá trị này mô tả cách nhóm các tập số trong tiền tệ ở phía *trái* dấu thập phân (phần nguyên).

Ví dụ, ta có thể có:

```
  2   1   0
 --- --- ---
$100,000,000.00
```

Đây là các nhóm ba chữ số. Nhóm 0 (ngay bên trái dấu thập phân) có 3 chữ số. Nhóm 1 (nhóm kế tiếp về trái) có 3 chữ số, và nhóm cuối cũng có 3.

Vậy ta có thể mô tả các nhóm này, từ phải (dấu thập phân) sang trái bằng một loạt số nguyên đại diện cho kích thước nhóm:

```
3 3 3
```

Và chừng đó sẽ ổn với giá trị tới $100,000,000.

Nhưng lỡ ta có nhiều hơn? Ta có thể cứ thêm `3`...

```
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

nhưng kiểu đó điên rồ. May thay, ta có thể chỉ định `0` để báo rằng kích thước nhóm trước được lặp lại:

```
3 0
```

Nghĩa là lặp mỗi 3. Cái đó sẽ xử được $100, $1,000, $10,000, $10,000,000, $100,000,000,000, và cứ thế.

Bạn có thể chính thức phát khùng với mấy cái này để chỉ ra vài kiểu nhóm kỳ cục.

Ví dụ:

```
4 3 2 1 0
```

sẽ cho ra:

```
$1,0,0,0,0,00,000,0000.00
```

Một giá trị nữa có thể xuất hiện là `CHAR_MAX`. Cái này báo rằng không còn nhóm nào nữa, và có thể xuất hiện ở bất kỳ chỗ nào trong mảng, kể cả giá trị đầu.

```
3 2 CHAR_MAX
```

sẽ cho ra:

```
100000000,00,000.00
```

chẳng hạn.

Và chỉ cần có `CHAR_MAX` ở vị trí đầu của mảng là báo cho bạn biết không có nhóm nào hết.

26.2.2 Dấu phân cách và vị trí dấu

Mọi biến thể `sep_by_space` xử lý khoảng trắng quanh ký hiệu tiền tệ. Giá trị hợp lệ là:

Giá trị	Mô tả
`0`	Không có khoảng trắng giữa ký hiệu tiền tệ và giá trị.
`1`	Tách ký hiệu tiền tệ (và dấu, nếu có) khỏi giá trị bằng một khoảng trắng.
`2`	Tách ký hiệu dấu khỏi ký hiệu tiền tệ (nếu kề nhau) bằng khoảng trắng, ngược lại tách ký hiệu dấu khỏi giá trị bằng khoảng trắng.

Các biến thể `sign_posn` được quyết định bởi các giá trị sau:

Giá trị	Mô tả
`0`	Bọc giá trị và ký hiệu tiền tệ bằng cặp ngoặc.
`1`	Đặt chuỗi dấu trước ký hiệu tiền tệ và giá trị.
`2`	Đặt chuỗi dấu sau ký hiệu tiền tệ và giá trị.
`3`	Đặt chuỗi dấu ngay trước ký hiệu tiền tệ.
`4`	Đặt chuỗi dấu ngay sau ký hiệu tiền tệ.

26.2.3 Ví dụ giá trị

Khi tôi lấy các giá trị trên hệ của mình, đây là thứ tôi thấy (chuỗi grouping hiển thị dưới dạng các giá trị byte riêng):

```
mon_decimal_point  = "."
mon_thousands_sep  = ","
mon_grouping       = 3 3 0
positive_sign      = ""
negative_sign      = "-"
currency_symbol    = "$"
frac_digits        = 2
p_cs_precedes      = 1
n_cs_precedes      = 1
p_sep_by_space     = 0
n_sep_by_space     = 0
p_sign_posn        = 1
n_sign_posn        = 1
int_curr_symbol    = "USD "
int_frac_digits    = 2
int_p_cs_precedes  = 1
int_n_cs_precedes  = 1
int_p_sep_by_space = 1
int_n_sep_by_space = 1
int_p_sign_posn    = 1
int_n_sign_posn    = 1
```

26.3 Chi tiết localization

Để ý ta đã truyền macro `LC_ALL` cho `setlocale()` ở trên, chuyện này gợi ý có thể có biến thể khác cho phép bạn chính xác hơn về *phần nào* của locale bạn đang đặt.

Hãy xem các giá trị bạn có thể thấy cho mấy cái này:

Macro	Mô tả
`LC_ALL`	Đặt tất cả những cái dưới đây về locale đã cho.
`LC_COLLATE`	Kiểm soát hành vi của hàm `strcoll()` và `strxfrm()`.
`LC_CTYPE`	Kiểm soát hành vi của các hàm xử lý ký tự[3].
`LC_MONETARY`	Kiểm soát giá trị mà `localeconv()` trả về.
`LC_NUMERIC`	Kiểm soát dấu thập phân cho họ hàm `printf()`.
`LC_TIME`	Kiểm soát định dạng thời gian cho các hàm in thời gian và ngày `strftime()` và `wcsftime()`.

Khá phổ biến thấy `LC_ALL` được đặt, nhưng, này, ít nhất bạn có lựa chọn.

Cũng nên nói `LC_CTYPE` là một trong những cái lớn vì nó gắn với wide character, một mớ bùng nhùng ta sẽ nói sau.

[3]Trừ `isdigit()` và `isxdigit()`.

Chapter 27

Unicode, wide character, và mấy thứ đó

Trước khi bắt đầu, lưu ý đây là vùng ngôn ngữ C đang phát triển sôi động khi nó cố vượt qua vài, ờm, *cơn đau trưởng thành*. Giờ C23 đã ra mắt, cập nhật ở đây là khả năng cao.

Phần lớn mọi người về cơ bản quan tâm câu hỏi tưởng đơn giản nhưng lừa gạt, "Làm sao dùng bộ ký tự này-nọ trong C?" Ta sẽ tới đó. Nhưng như ta sẽ thấy, có khi nó đã chạy sẵn trên hệ của bạn rồi. Hoặc bạn có thể phải đổ qua thư viện bên thứ ba.

Ta sẽ nói về khá nhiều thứ trong chương này, vài cái không phụ thuộc nền tảng, vài cái riêng của C.

Hãy xem sơ đồ những gì ta sắp xem:

- Nền tảng Unicode
- Nền tảng encoding ký tự
- Bộ ký tự nguồn và bộ ký tự thực thi
- Dùng Unicode và UTF-8
- Dùng các kiểu ký tự khác như `wchar_t`, `char16_t`, và `char32_t`

Lao vào nào!

27.1 Unicode là gì?

Ngày xưa, ở Mỹ và phần lớn thế giới, phổ biến dùng encoding 7-bit hay 8-bit cho ký tự trong bộ nhớ. Điều này nghĩa là ta có thể có 128 hay 256 ký tự (kể cả ký tự không in được) tổng cộng. Chừng đó ổn với một thế giới lấy Mỹ làm trung tâm, nhưng hóa ra ngoài kia còn bảng chữ cái khác, ai mà biết được? Tiếng Trung có hơn 50.000 ký tự, và ngần đó không nhét vừa một byte.

Thế là người ta đẻ ra đủ kiểu cách khác nhau để biểu diễn bộ ký tự riêng của mình. Và vậy cũng ổn, nhưng biến thành cơn ác mộng tương thích.

Để thoát khỏi đó, Unicode được phát minh. Một bộ ký tự để cai trị tất cả. Nó trải ra tới vô hạn (về cơ bản) nên ta sẽ không bao giờ hết chỗ cho ký tự mới. Nó có tiếng Trung, Latin, Hy Lạp, chữ hình nêm, ký hiệu cờ vua, emoji... gần như mọi thứ, thật đấy! Và liên tục có thêm cái mới!

27.2 Code point

Tôi muốn nói về hai khái niệm ở đây. Hơi rối vì cả hai đều là số, các số khác nhau cho cùng một thứ. Nhưng ráng theo tôi nào.

Định nghĩa *code point* một cách lỏng lẻo là một giá trị số đại diện cho một ký tự. (Code point cũng có thể đại diện cho ký tự điều khiển không in được, nhưng cứ giả định tôi muốn nói tới cái gì đó như chữ "B" hay ký tự "π".)

Mỗi code point đại diện cho một ký tự duy nhất. Và mỗi ký tự có một code point số duy nhất gắn với nó.

Ví dụ, trong Unicode, giá trị số 66 đại diện cho "B", và 960 đại diện cho "π". Các ánh xạ ký tự khác không phải Unicode dùng giá trị khác, có thể, nhưng hãy quên chúng và tập trung vào Unicode, tương lai!

Vậy đó là một chuyện: có một con số đại diện cho từng ký tự. Trong Unicode, các số này chạy từ 0 tới hơn 1 triệu.

Hiểu rồi chứ?

Vì ta sắp lật bàn tí đây.

27.3 Encoding

Nếu bạn còn nhớ, một byte 8-bit có thể giữ giá trị từ 0-255, gồm cả hai đầu. Chừng đó ổn với "B" là 66, cái đó vừa vặn trong một byte. Nhưng "π" là 960, cái đó không vừa một byte! Ta cần byte khác. Làm sao ta lưu hết mớ đó trong bộ nhớ? Hay mấy số lớn hơn, như 195.024? Cái đó sẽ cần một số byte để giữ.

Câu hỏi lớn: các con số này được biểu diễn ra sao trong bộ nhớ? Đây là cái ta gọi là *encoding* của các ký tự.

Vậy ta có hai thứ: một là code point cho ta biết về cơ bản số sê-ri của một ký tự cụ thể. Và ta có encoding cho ta biết ta sẽ biểu diễn con số đó ra sao trong bộ nhớ.

Có cả đống encoding. Bạn có thể tự nghĩ ra encoding của mình ngay bây giờ, nếu bạn muốn[1]. Nhưng ta sẽ xem vài encoding thực sự phổ biến đang được dùng với Unicode.

Encoding	Mô tả
UTF-8	Encoding hướng byte, dùng số byte thay đổi trên mỗi ký tự. Đây là cái nên dùng.
UTF-16	Encoding 16-bit cho mỗi ký tự[2].
UTF-32	Encoding 32-bit cho mỗi ký tự.

Với UTF-16 và UTF-32, thứ tự byte có ý nghĩa, nên bạn có thể thấy UTF-16BE cho big-endian và UTF-16LE cho little-endian. Y vậy cho UTF-32. Về kỹ thuật, nếu không chỉ định, bạn nên giả định big-endian. Nhưng vì Windows dùng UTF-16 nhiều và nó little-endian, đôi khi điều đó được giả định[3].

Xem vài ví dụ. Tôi sẽ viết giá trị theo hex vì nó đúng hai chữ số cho mỗi byte 8-bit, và làm vậy dễ thấy mọi thứ xếp ra sao trong bộ nhớ hơn.

Ký tự	Code Point	UTF-16BE	UTF-32BE	UTF-16LE	UTF-32LE	UTF-8
A	41	0041	00000041	4100	41000000	41
B	42	0042	00000042	4200	42000000	42
~	7E	007E	0000007E	7E00	7E000000	7E
π	3C0	03C0	000003C0	C003	C0030000	CF80
€	20AC	20AC	000020AC	AC20	AC200000	E282AC

Ngó trong đó tìm mẫu xem. Để ý UTF-16BE và UTF-32BE chỉ là code point được biểu diễn thẳng dưới dạng giá trị 16 và 32-bit[4].

Little-endian cũng y vậy, chỉ khác là các byte theo thứ tự little-endian.

Rồi ta có UTF-8 ở cuối. Đầu tiên bạn có thể để ý code point một-byte được biểu diễn dưới dạng một byte. Cái đó hay. Bạn cũng có thể để ý code point khác nhau chiếm số byte khác nhau. Đây là encoding độ rộng thay đổi.

[1] Ví dụ, ta có thể lưu code point trong một số nguyên 32-bit big-endian. Thẳng thớm! Ta vừa phát minh ra một encoding! Thật ra thì không; đó là cái encoding UTF-32BE. Chao ôi, quay lại với công việc thôi!

[2] Kiểu kiểu vậy. Về kỹ thuật, nó có độ rộng thay đổi, có cách biểu diễn code point lớn hơn 2^{16} bằng cách ghép hai ký tự UTF-16 lại.

[3] Có một ký tự đặc biệt tên *Byte Order Mark* (BOM), code point 0xFEFF, có thể tuỳ chọn đi trước luồng dữ liệu và cho biết endianness. Tuy nhiên nó không bắt buộc.

[4] Lại, điều này chỉ đúng trong UTF-16 cho các ký tự vừa trong hai byte.

Nên ngay khi vượt qua một giá trị nào đó, UTF-8 bắt đầu dùng thêm byte để lưu giá trị. Và chúng có vẻ không tương quan với giá trị code point.

Chi tiết encoding UTF-8[5] nằm ngoài phạm vi sách này, nhưng biết thế này là đủ: nó có số byte thay đổi cho mỗi code point, và các giá trị byte đó không khớp với code point *trừ 128 code point đầu tiên.* Nếu bạn thật sự muốn biết thêm, Computerphile có video về UTF-8 rất hay với Tom Scott[6].

Cái cuối đó là điều thú vị về Unicode và UTF-8 từ góc nhìn Bắc Mỹ: nó tương thích ngược với encoding ASCII 7-bit! Nên nếu bạn quen với ASCII, UTF-8 giống y vậy! Mọi tài liệu được encode bằng ASCII cũng được encode bằng UTF-8! (Dĩ nhiên không phải ngược lại.)

Có lẽ chính điểm cuối này, hơn bất cứ điểm nào khác, đang đẩy UTF-8 thống trị thế giới.

27.4 Bộ ký tự nguồn và thực thi

Khi lập trình C, có (ít nhất) ba bộ ký tự đang chơi:

- Cái code của bạn tồn tại trên đĩa dưới dạng.
- Cái compiler dịch sang ngay khi bắt đầu compile (*bộ ký tự nguồn*). Có thể giống cái trên đĩa, hoặc không.
- Cái compiler dịch bộ ký tự nguồn sang để thực thi (*bộ ký tự thực thi*). Có thể giống bộ ký tự nguồn, hoặc không.

Compiler của bạn có lẽ có tùy chọn để chọn các bộ ký tự này lúc build.

Bộ ký tự cơ bản cho cả nguồn và thực thi sẽ chứa các ký tự sau:

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
! " # % & ' ( ) * + , - . / :
; < = > ? [ \ ] ^ _ { | } ~
space tab vertical-tab
form-feed end-of-line
```

Đó là các ký tự bạn có thể dùng trong nguồn và vẫn portable 100%.

Bộ ký tự thực thi sẽ có thêm ký tự cho alert (chuông/chớp), backspace, carriage return, và newline.

Nhưng phần lớn mọi người không đi tới mức cực đoan đó và thoải mái dùng bộ ký tự mở rộng trong nguồn và chương trình chạy, nhất là bây giờ Unicode và UTF-8 đang phổ biến hơn. Ý tôi là, bộ ký tự cơ bản thậm chí không cho phép `@`, `$`, hay `` ` ``!

Đáng chú ý, đau đầu (dù làm được bằng escape sequence) khi gõ ký tự Unicode chỉ bằng bộ ký tự cơ bản.

27.5 Unicode trong C

Trước khi đi vào encoding trong C, hãy nói về Unicode từ góc độ code point. Có cách trong C để chỉ định ký tự Unicode và chúng sẽ được compiler dịch sang bộ ký tự thực thi[7].

Vậy làm sao ta làm?

Thử ký hiệu euro, code point 0x20AC. (Tôi viết nó bằng hex vì cả hai cách biểu diễn nó trong C đều dùng hex.) Làm sao ta đặt nó vào code C?

Dùng escape `\u` để đặt nó trong chuỗi, ví dụ `"\u20AC"` (viết hoa thường hex không quan trọng). Bạn phải đặt **đúng bốn** chữ số hex sau `\u`, pad bằng số 0 đầu nếu cần.

[5] https://en.wikipedia.org/wiki/UTF-8
[6] https://www.youtube.com/watch?v=MijmeoH9LT4
[7] Có lẽ compiler cố hết sức dịch code point sang encoding output nào đó, nhưng tôi không tìm thấy đảm bảo nào trong spec.

Đây là ví dụ:

```
char *s = "\u20AC1.23";

printf("%s\n", s);  // €1.23
```

Vậy `\u` chạy với code point Unicode 16-bit, còn mấy cái lớn hơn 16-bit thì sao? Cho cái đó, ta cần chữ hoa: `\U`.

Ví dụ:

```
char *s = "\U0001D4D1";

printf("%s\n", s);  // Prints a mathematical letter "B"
```

Giống `\u`, chỉ là 32-bit thay vì 16. Hai cái này tương đương:

```
\u03C0
\U000003C0
```

Lại, các cái này được dịch sang bộ ký tự thực thi lúc compile. Chúng đại diện cho code point Unicode, không phải encoding cụ thể nào. Thêm nữa, nếu một code point Unicode không biểu diễn được trong bộ ký tự thực thi, compiler có thể làm gì với nó cũng được.

Giờ, bạn có thể thắc mắc sao không làm thế này:

```
char *s = "€1.23";

printf("%s\n", s);  // €1.23
```

Và có lẽ bạn làm được, với compiler hiện đại. Bộ ký tự nguồn sẽ được compiler dịch sang bộ ký tự thực thi cho bạn. Nhưng compiler có quyền nôn ra nếu tìm thấy ký tự nào không có trong bộ ký tự mở rộng của nó, và ký hiệu € chắc chắn không có trong bộ ký tự cơ bản.

Lưu ý từ spec: bạn không thể dùng `\u` hay `\U` để encode bất kỳ code point nào dưới 0xA0 trừ 0x24 (`$`), 0x40 (`@`), và 0x60 (`` ` ``), đúng rồi, đó là bộ ba dấu câu phổ biến bị thiếu khỏi bộ ký tự cơ bản. Rõ ràng hạn chế này được nới lỏng trong phiên bản spec sắp tới.

Cuối cùng, bạn cũng có thể dùng các cái này trong định danh trong code của mình, với vài hạn chế. Nhưng tôi không muốn đi vào đó ở đây. Chương này ta chỉ tập trung xử lý chuỗi.

Và đó gần như là toàn bộ về Unicode trong C (trừ encoding).

27.6 Ghi chú nhanh về UTF-8 trước khi lao vào bụi rậm

Có thể file nguồn của bạn trên đĩa, các ký tự nguồn mở rộng, và các ký tự thực thi mở rộng đều ở định dạng UTF-8. Và các thư viện bạn dùng mong đợi UTF-8. Đây là tương lai rực rỡ của UTF-8 ở mọi nơi.

Nếu đúng vậy, và bạn không ngại không portable sang các hệ không như thế, cứ chạy. Nhét ký tự Unicode vào nguồn và dữ liệu thoải mái. Dùng chuỗi C thường và vui vẻ.

Nhiều thứ sẽ chạy được (dù không portable) vì chuỗi UTF-8 có thể kết thúc bằng NUL an toàn y như chuỗi C nào khác. Nhưng có thể đổi tính portable để xử lý ký tự dễ hơn là cái đánh đổi đáng giá với bạn.

Tuy nhiên, có vài lưu ý:

- Những thứ như `strlen()` báo số byte trong chuỗi, không phải số ký tự. (`mbstowcs()` trả về số ký tự trong chuỗi khi bạn chuyển nó sang wide character. POSIX mở rộng cái này để bạn có thể truyền `NULL` làm đối số đầu nếu chỉ muốn đếm số ký tự.)

- Những cái sau sẽ không chạy đúng với ký tự hơn một byte: `strtok()`, `strchr()` (dùng `strstr()` thay thế), họ hàm `strspn()`, `toupper()`, `tolower()`, họ hàm `isalpha()`, và chắc còn nữa. Cảnh giác với bất cứ gì hoạt động trên byte.
- Các biến thể `printf()` cho phép chỉ in ra một số byte của chuỗi[8]. Bạn cần chắc chắn in đúng số byte để kết thúc ở ranh giới ký tự.
- Nếu bạn muốn `malloc()` chỗ cho chuỗi, hay khai báo mảng `char` cho một chuỗi, lưu ý kích thước tối đa có thể nhiều hơn bạn nghĩ. Mỗi ký tự có thể chiếm tới `MB_LEN_MAX` byte (từ `<limits.h>`), trừ các ký tự trong bộ ký tự cơ bản đảm bảo là một byte.

Và chắc còn nữa mà tôi chưa khám phá ra. Cho tôi biết còn cái bẫy nào ngoài kia nữa nhé...

27.7 Các kiểu ký tự khác nhau

Tôi muốn giới thiệu thêm kiểu ký tự. Ta quen với `char`, đúng không?

Nhưng cái đó quá dễ. Hãy làm mọi thứ khó hơn nhiều! Hoan hô!

27.7.1 Ký tự multibyte

Trước hết, tôi muốn có thể thay đổi cách bạn nghĩ về chuỗi (mảng `char`) là gì. Chúng là *chuỗi multibyte* được tạo từ *ký tự multibyte*.

Đúng rồi, cái chuỗi ký tự bình thường của bạn là multibyte. Khi ai đó nói "C string", họ đang nói "chuỗi multibyte C".

Kể cả khi một ký tự cụ thể trong chuỗi chỉ là một byte, hay nếu chuỗi được tạo chỉ từ ký tự đơn, nó vẫn được gọi là chuỗi multibyte.

Ví dụ:

```
char c[128] = "Hello, world!";  // Multibyte string
```

Cái ta đang nói ở đây là một ký tự cụ thể không thuộc bộ ký tự cơ bản có thể được tạo từ nhiều byte. Tối đa `MB_LEN_MAX` byte (từ `<limits.h>`). Chắc, trên màn hình nó chỉ trông như một ký tự, nhưng có thể là nhiều byte.

Bạn cũng có thể ném giá trị Unicode vào đó, như ta thấy trước đó:

```
char *s = "\u20AC1.23";

printf("%s\n", s);  // €1.23
```

Nhưng ở đây ta vào vùng kỳ lạ, vì xem này:

```
char *s = "\u20AC1.23";  // €1.23

printf("%zu\n", strlen(s));  // 7!
```

Độ dài chuỗi của `"€1.23"` là `7`?! Đúng vậy! Ồ, trên hệ của tôi, đúng! Nhớ `strlen()` trả về số byte trong chuỗi, không phải số ký tự. (Khi ta tới "wide character", sắp tới, ta sẽ thấy cách lấy số ký tự trong chuỗi.)

Lưu ý C cho phép hằng `char` multibyte riêng lẻ (khác với `char*`), nhưng hành vi của chúng thay đổi theo implementation và compiler có thể cảnh báo về nó.

GCC, chẳng hạn, cảnh báo về hằng ký tự multi-character cho hai dòng sau (và, trên hệ của tôi, in ra encoding UTF-8):

[8] Với format specifier kiểu `"%.12s"` chẳng hạn.

```
printf("%x\n", '€');
printf("%x\n", '\u20ac');
```

27.7.2 Wide character

Nếu bạn không phải ký tự multibyte, thì bạn là *wide character*.

Wide character là một giá trị đơn có thể đại diện duy nhất cho bất kỳ ký tự nào trong locale hiện tại. Nó tương đương với code point Unicode. Nhưng có thể không. Hoặc có thể.

Về cơ bản, chuỗi ký tự multibyte là mảng các byte, thì chuỗi wide character là mảng các *ký tự*. Nên bạn có thể bắt đầu suy nghĩ theo kiểu từng-ký-tự thay vì từng-byte (cái sau rối mù khi ký tự bắt đầu chiếm số byte thay đổi).

Wide character có thể được biểu diễn bằng một số kiểu, nhưng cái nổi bật nhất là `wchar_t`. Nó là cái chính. Giống `char`, chỉ là wide.

Bạn có thể thắc mắc nếu bạn không biết có phải Unicode hay không, sao mà cho bạn nhiều linh hoạt trong việc viết code? `wchar_t` mở vài cửa đó ra, vì có bộ hàm phong phú bạn có thể dùng để xử lý chuỗi `wchar_t` (như lấy độ dài, v.v.) mà không quan tâm encoding.

27.8 Dùng wide character và `wchar_t`

Đến lúc cho kiểu mới: `wchar_t`. Đây là kiểu wide character chính. Nhớ cách `char` chỉ một byte chứ? Và một byte có thể không đủ đại diện mọi ký tự? Ừ, cái này đủ.

Để dùng `wchar_t`, include `<wchar.h>`.

Nó to bao nhiêu byte? Không rõ lắm. Có thể 16 bit. Có thể 32 bit.

Nhưng khoan, bạn đang nói, nếu chỉ 16 bit, nó không đủ giữ hết code point Unicode đúng không? Đúng, không đủ. Spec không yêu cầu nó phải thế. Nó chỉ phải biểu diễn được mọi ký tự trong locale hiện tại.

Điều này có thể gây đau đầu với Unicode trên nền tảng `wchar_t` 16-bit (ờ hèm, Windows). Nhưng cái đó ngoài phạm vi sách này.

Bạn có thể khai báo chuỗi hay ký tự kiểu này với tiền tố `L`, và bạn có thể in chúng với format specifier `%ls` ("ell ess"). Hoặc in một `wchar_t` riêng lẻ với `%lc`.

```
wchar_t *s = L"Hello, world!";
wchar_t c = L'B';

printf("%ls %lc\n", s, c);
```

Giờ, các ký tự đó có được lưu dưới dạng code point Unicode hay không? Tùy implementation. Nhưng bạn có thể kiểm tra xem có phải không bằng macro `__STDC_ISO_10646__`. Nếu cái này được định nghĩa, câu trả lời là, "Nó là Unicode!"

Chi tiết hơn, giá trị trong macro đó là số nguyên dạng `yyyymm` cho bạn biết bạn có thể dựa vào chuẩn Unicode nào, bất cứ cái nào đang có hiệu lực vào ngày đó.

Nhưng dùng chúng thế nào?

27.8.1 Chuyển multibyte sang `wchar_t`

Vậy làm sao từ chuỗi chuẩn hướng byte sang chuỗi wide hướng ký tự và ngược lại?

Ta có thể dùng vài hàm chuyển chuỗi để làm chuyện này.

Đầu tiên, vài quy ước đặt tên bạn sẽ thấy trong các hàm này:

- `mb`: multibyte

- `wc` : wide character
- `mbs` : multibyte string
- `wcs` : wide character string

Vậy nếu ta muốn chuyển chuỗi multibyte thành chuỗi wide character, ta có thể gọi `mbstowcs()` . Và chiều ngược lại: `wcstombs()` .

Hàm chuyển	Mô tả
`mbtowc()`	Chuyển ký tự multibyte sang wide character.
`wctomb()`	Chuyển wide character sang ký tự multibyte.
`mbstowcs()`	Chuyển chuỗi multibyte sang chuỗi wide.
`wcstombs()`	Chuyển chuỗi wide sang chuỗi multibyte.

Làm demo nhanh ta chuyển chuỗi multibyte thành chuỗi wide character, rồi so độ dài chuỗi của hai cái bằng các hàm tương ứng.

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <string.h>
#include <locale.h>

int main(void)
{
    // Get out of the C locale to one that likely has the euro symbol
    setlocale(LC_ALL, "");

    // Original multibyte string with a euro symbol (Unicode point 20ac)
    char *mb_string = "The cost is \u20ac1.23";  // €1.23
    size_t mb_len = strlen(mb_string);

    // Wide character array that will hold the converted string
    wchar_t wc_string[128];  // Holds up to 128 wide characters

    // Convert the MB string to WC; this returns the number of wide chars
    size_t wc_len = mbstowcs(wc_string, mb_string, 128);

    // Print result--note the %ls for wide char strings
    printf("multibyte: \"%s\" (%zu bytes)\n", mb_string, mb_len);
    printf("wide char: \"%ls\" (%zu characters)\n", wc_string, wc_len);
}
```

Trên hệ của tôi, cái này in ra:

```
multibyte: "The cost is €1.23" (19 bytes)
wide char: "The cost is €1.23" (17 characters)
```

(Máy bạn có thể khác số byte tùy locale.)

Một điều thú vị cần lưu ý là `mbstowcs()` , ngoài việc chuyển chuỗi multibyte sang wide, còn trả về độ dài (tính bằng ký tự) của chuỗi wide character. Trên hệ tuân thủ POSIX, bạn có thể tận dụng chế độ đặc biệt nơi nó *chỉ* trả về độ dài tính bằng ký tự của một chuỗi multibyte cho trước: bạn chỉ truyền `NULL` cho đích, và `0` cho số ký tự tối đa cần chuyển (giá trị này bị bỏ qua).

(Trong code dưới, tôi đang dùng bộ ký tự nguồn mở rộng của mình, bạn có thể phải thay bằng escape `\u` .)

```
setlocale(LC_ALL, "");

// The following string has 7 characters
size_t len_in_chars = mbstowcs(NULL, "§¶°±π€•", 0);

printf("%zu", len_in_chars);  // 7
```

Lại, đó là mở rộng POSIX không portable.

Và, dĩ nhiên, nếu bạn muốn chuyển chiều ngược lại, đó là `wcstombs()`.

27.9 Chức năng wide character

Một khi đã ở xứ wide character, ta có đủ loại chức năng trong tay. Tôi sẽ chỉ tóm tắt một đống hàm ở đây, nhưng về cơ bản cái ta có ở đây là phiên bản wide character của các hàm chuỗi multibyte ta quen thuộc. (Ví dụ, ta biết `strlen()` cho chuỗi multibyte; có `wcslen()` cho chuỗi wide character.)

27.9.1 `wint_t`

Nhiều hàm trong đám này dùng `wint_t` để giữ ký tự đơn, dù chúng được truyền vào hay trả về.

Về bản chất nó có liên quan đến `wchar_t`. `wint_t` là số nguyên có thể đại diện mọi giá trị trong bộ ký tự mở rộng, và cũng một ký tự end-of-file đặc biệt, `WEOF`.

Cái này được dùng bởi một số hàm wide character hướng ký tự đơn.

27.9.2 Hướng luồng I/O

Tóm gọn ở đây là đừng trộn lẫn hàm hướng byte (như `fprintf()`) với hàm hướng wide (như `fwprintf()`). Quyết định xem luồng sẽ hướng byte hay hướng wide và bám lấy kiểu hàm I/O đó.

Chi tiết hơn: luồng có thể hướng byte hoặc hướng wide. Khi luồng mới được tạo, nó không có hướng, nhưng lần đọc hoặc ghi đầu sẽ đặt hướng.

Nếu bạn dùng phép wide trước (như `fwprintf()`) nó sẽ đặt hướng luồng sang wide.

Nếu bạn dùng phép byte trước (như `fprintf()`) nó sẽ đặt hướng luồng theo byte.

Bạn có thể đặt thủ công một luồng chưa có hướng theo cách này hoặc cách kia bằng lời gọi `fwide()`. Bạn có thể dùng cùng hàm đó để lấy hướng của luồng.

Nếu cần đổi hướng giữa chừng, bạn có thể làm bằng `freopen()`.

27.9.3 Hàm I/O

Thường include `<stdio.h>` và `<wchar.h>` cho mấy cái này.

Hàm I/O	Mô tả
`wprintf()`	Output console có định dạng.
`wscanf()`	Input console có định dạng.
`getwchar()`	Input console hướng ký tự.
`putwchar()`	Output console hướng ký tự.
`fwprintf()`	Output file có định dạng.
`fwscanf()`	Input file có định dạng.
`fgetwc()`	Input file hướng ký tự.
`fputwc()`	Output file hướng ký tự.
`fgetws()`	Input file hướng chuỗi.
`fputws()`	Output file hướng chuỗi.

Hàm I/O	Mô tả
`swprintf()`	Output chuỗi có định dạng.
`swscanf()`	Input chuỗi có định dạng.
`vfwprintf()`	Output file có định dạng, variadic.
`vfwscanf()`	Input file có định dạng, variadic.
`vswprintf()`	Output chuỗi có định dạng, variadic.
`vswscanf()`	Input chuỗi có định dạng, variadic.
`vwprintf()`	Output console có định dạng, variadic.
`vwscanf()`	Input console có định dạng, variadic.
`ungetwc()`	Đẩy một wide character ngược lại luồng output.
`fwide()`	Lấy hoặc đặt hướng multibyte/wide của luồng.

27.9.4 Hàm chuyển kiểu

Thường include `<wchar.h>` cho mấy cái này.

Hàm chuyển	Mô tả
`wcstod()`	Chuyển chuỗi sang `double`.
`wcstof()`	Chuyển chuỗi sang `float`.
`wcstold()`	Chuyển chuỗi sang `long double`.
`wcstol()`	Chuyển chuỗi sang `long`.
`wcstoll()`	Chuyển chuỗi sang `long long`.
`wcstoul()`	Chuyển chuỗi sang `unsigned long`.
`wcstoull()`	Chuyển chuỗi sang `unsigned long long`.

27.9.5 Hàm copy chuỗi và bộ nhớ

Thường include `<wchar.h>` cho mấy cái này.

Hàm copy	Mô tả
`wcscpy()`	Copy chuỗi.
`wcsncpy()`	Copy chuỗi, giới hạn độ dài.
`wmemcpy()`	Copy bộ nhớ.
`wmemmove()`	Copy bộ nhớ có thể chồng lấn.
`wcscat()`	Nối chuỗi.
`wcsncat()`	Nối chuỗi, giới hạn độ dài.

27.9.6 Hàm so sánh chuỗi và bộ nhớ

Thường include `<wchar.h>` cho mấy cái này.

Hàm so sánh	Mô tả
`wcscmp()`	So sánh chuỗi theo thứ tự từ điển.
`wcsncmp()`	So sánh chuỗi theo thứ tự từ điển, giới hạn độ dài.
`wcscoll()`	So sánh chuỗi theo thứ tự từ điển của locale.
`wmemcmp()`	So sánh bộ nhớ theo thứ tự từ điển.
`wcsxfrm()`	Biến chuỗi thành phiên bản khiến `wcscmp()` hành xử như `wcscoll()` [9].

[9] `wcscoll()` là `wcsxfrm()` rồi theo sau bởi `wcscmp()`.

27.9.7 Hàm tìm kiếm chuỗi

Thường include `<wchar.h>` cho mấy cái này.

Hàm tìm	Mô tả
`wcschr()`	Tìm một ký tự trong chuỗi.
`wcsrchr()`	Tìm một ký tự trong chuỗi từ phía sau.
`wmemchr()`	Tìm một ký tự trong bộ nhớ.
`wcsstr()`	Tìm chuỗi con trong chuỗi.
`wcspbrk()`	Tìm bất kỳ ký tự nào trong một tập ký tự trong chuỗi.
`wcsspn()`	Tìm độ dài chuỗi con gồm bất kỳ ký tự nào trong tập.
`wcscspn()`	Tìm độ dài chuỗi con trước bất kỳ ký tự nào trong tập.
`wcstok()`	Tìm token trong chuỗi.

27.9.8 Hàm về độ dài/linh tinh

Thường include `<wchar.h>` cho mấy cái này.

Hàm Length/Misc	Mô tả
`wcslen()`	Trả về độ dài chuỗi.
`wmemset()`	Đặt ký tự trong bộ nhớ.
`wcsftime()`	Output ngày và giờ có định dạng.

27.9.9 Hàm phân loại ký tự

Include `<wctype.h>` cho mấy cái này.

Hàm Length/Misc	Mô tả
`iswalnum()`	True nếu ký tự là chữ và số.
`iswalpha()`	True nếu ký tự là chữ cái.
`iswblank()`	True nếu ký tự là khoảng trắng (giống space, nhưng không phải newline).
`iswcntrl()`	True nếu ký tự là ký tự điều khiển.
`iswdigit()`	True nếu ký tự là chữ số.
`iswgraph()`	True nếu ký tự in được (trừ space).
`iswlower()`	True nếu ký tự là chữ thường.
`iswprint()`	True nếu ký tự in được (kể cả space).
`iswpunct()`	True nếu ký tự là dấu câu.
`iswspace()`	True nếu ký tự là khoảng trắng.
`iswupper()`	True nếu ký tự là chữ hoa.
`iswxdigit()`	True nếu ký tự là chữ số hex.
`towlower()`	Chuyển ký tự thành chữ thường.
`towupper()`	Chuyển ký tự thành chữ hoa.

27.10 Parse state, hàm restartable

Ta sẽ đi sâu chút vào ruột gan của chuyển multibyte, nhưng đây là chuyện hay để hiểu, về khái niệm.

Tưởng tượng chương trình của bạn lấy một chuỗi ký tự multibyte và biến chúng thành wide character, hoặc ngược lại. Có lúc, nó có thể đang phân tích dở một ký tự, hoặc có thể phải đợi thêm byte trước khi chốt giá trị cuối.

Parse state này được lưu trong một biến mờ kiểu `mbstate_t` và được dùng mỗi khi chuyển được thực hiện. Đó là cách các hàm chuyển theo dõi chúng đang ở đâu giữa chừng.

Và nếu bạn đổi sang chuỗi ký tự khác giữa chừng, hoặc cố seek sang chỗ khác trong chuỗi input, nó có thể bị rối.

Giờ bạn có thể bắt bẻ tôi: ta vừa chuyển vài cái ở trên, mà tôi chưa bao giờ nhắc tới `mbstate_t` nào.

Đó là vì các hàm chuyển như `mbstowcs()`, `wctomb()`, v.v. mỗi cái có biến `mbstate_t` riêng mà chúng dùng. Tuy nhiên chỉ có một cho mỗi hàm, nên nếu bạn đang viết code đa luồng, chúng không an toàn để dùng.

May thay, C định nghĩa phiên bản *restartable* của các hàm này, trong đó bạn có thể truyền vào `mbstate_t` riêng trên cơ sở từng luồng nếu cần. Nếu bạn làm đa luồng, dùng mấy cái này!

Ghi chú nhanh về khởi tạo biến `mbstate_t`: chỉ cần `memset()` nó về 0. Không có hàm sẵn nào để ép nó khởi tạo.

```
mbstate_t mbs;

// Set the state to the initial state
memset(&mbs, 0, sizeof mbs);
```

Đây là danh sách các hàm chuyển restartable, để ý quy ước đặt tên chèn thêm "`r`" sau kiểu "from":

- `mbrtowc()`, multibyte sang wide character
- `wcrtomb()`, wide character sang multibyte
- `mbsrtowcs()`, chuỗi multibyte sang chuỗi wide character
- `wcsrtombs()`, chuỗi wide character sang chuỗi multibyte

Chúng thực sự giống với đồng nghiệp không restartable, trừ việc chúng yêu cầu bạn truyền vào pointer tới biến `mbstate_t` riêng của bạn. Và chúng cũng sửa pointer chuỗi nguồn (để giúp bạn nếu phát hiện byte không hợp lệ), nên có thể hữu ích khi lưu bản sao của bản gốc.

Đây là ví dụ trước đó trong chương, sửa lại để truyền `mbstate_t` riêng vào.

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <wchar.h>
#include <string.h>
#include <locale.h>

int main(void)
{
    // Get out of the C locale to one that likely has the euro symbol
    setlocale(LC_ALL, "");

    // Original multibyte string with a euro symbol (Unicode point 20ac)
    char *mb_string = "The cost is \u20ac1.23";  // €1.23
    size_t mb_len = strlen(mb_string);

    // Wide character array that will hold the converted string
    wchar_t wc_string[128];  // Holds up to 128 wide characters

    // Set up the conversion state
    mbstate_t mbs;
    memset(&mbs, 0, sizeof mbs);  // Initial state

    // mbsrtowcs() modifies the input pointer to point at the first
    // invalid character, or NULL if successful. Let's make a copy of
    // the pointer for mbsrtowcs() to mess with so our original is
```

```
    // unchanged.
    //
    // This example will probably be successful, but we check farther
    // down to see.
    const char *invalid = mb_string;

    // Convert the MB string to WC; this returns the number of wide chars
    size_t wc_len = mbsrtowcs(wc_string, &invalid, 128, &mbs);

    if (invalid == NULL) {
        printf("No invalid characters found\n");

        // Print result--note the %ls for wide char strings
        printf("multibyte: \"%s\" (%zu bytes)\n", mb_string, mb_len);
        printf("wide char: \"%ls\" (%zu characters)\n", wc_string, wc_len);
    } else {
        ptrdiff_t offset = invalid - mb_string;
        printf("Invalid character at offset %td\n", offset);
    }
}
```

Với các hàm chuyển tự quản state riêng, bạn có thể reset state nội bộ của chúng về trạng thái ban đầu bằng cách truyền `NULL` cho các đối số `char*` của chúng, ví dụ:

```
mbstowcs(NULL, NULL, 0);   // Reset the parse state for mbstowcs()
mbstowcs(dest, src, 100);  // Parse some stuff
```

Với I/O, mỗi luồng wide tự quản `mbstate_t` của mình và dùng nó cho chuyển input và output theo diễn biến.

Và một số hàm I/O hướng byte như `printf()` và `scanf()` giữ state nội bộ riêng khi làm việc.

Cuối cùng, các hàm chuyển restartable này thật ra có state nội bộ riêng nếu bạn truyền `NULL` cho tham số `mbstate_t`. Điều này làm chúng hành xử giống với đồng nghiệp không restartable hơn.

27.11 Encoding Unicode và C

Trong phần này, ta sẽ xem C làm được (và không làm được) gì với ba encoding Unicode cụ thể: UTF-8, UTF-16, và UTF-32.

27.11.1 UTF-8

Để làm mới trước phần này, đọc lại ghi chú nhanh UTF-8 ở trên.

Ngoài ra, C có khả năng UTF-8 gì?

Ồ, không nhiều, tiếc thay.

Bạn có thể nói với C rằng bạn muốn cụ thể một string literal được encode UTF-8, và nó sẽ làm cho bạn. Bạn có thể đặt tiền tố `u8` trước chuỗi:

```
char *s = u8"Hello, world!";

printf("%s\n", s);   // Hello, world!--if you can output UTF-8
```

Giờ, bạn có thể nhét ký tự Unicode vào đó không?

```
char *s = u8"€123";
```

Được! Nếu bộ ký tự nguồn mở rộng hỗ trợ nó. (gcc hỗ trợ.)

Nếu không hỗ trợ thì sao? Bạn có thể chỉ định code point Unicode với người bạn thân `\u` và `\U`, như đã nói ở trên.

Nhưng cái đó là hết. Không có cách portable nào trong thư viện chuẩn để lấy input tùy ý và biến nó thành UTF-8 trừ khi locale của bạn là UTF-8. Hoặc parse UTF-8 trừ khi locale của bạn là UTF-8.

Nên nếu bạn muốn làm, hoặc ở trong locale UTF-8 và:

```
setlocale(LC_ALL, "");
```

hoặc tìm ra tên locale UTF-8 trên máy cục bộ và đặt nó tường minh kiểu:

```
setlocale(LC_ALL, "en_US.UTF-8");  // Non-portable name
```

Hoặc dùng thư viện bên thứ ba.

27.11.2 UTF-16, UTF-32, `char16_t`, và `char32_t`

`char16_t` và `char32_t` là vài kiểu ký tự có tiềm năng wide khác với kích thước 16 bit và 32 bit, tương ứng. Không nhất thiết là wide, vì nếu chúng không thể đại diện mọi ký tự trong locale hiện tại, chúng mất đi tính wide character. Nhưng spec gọi chúng là kiểu "wide character" khắp nơi, nên đành vậy.

Mấy cái này có ở đây để làm mọi thứ thân thiện với Unicode hơn một chút, có tiềm năng.

Để dùng, include `<uchar.h>`. (Đó là "u", không phải "w".)

Header file này không có trên OS X, tiếc. Nếu bạn chỉ muốn kiểu, bạn có thể:

```
#include <stdint.h>

typedef int_least16_t char16_t;
typedef int_least32_t char32_t;
```

Nhưng nếu cũng muốn hàm, cái đó tùy bạn.

Giả sử bạn vẫn ổn để tiếp, bạn có thể khai báo chuỗi hay ký tự các kiểu này với tiền tố `u` và `U`:

```
char16_t *s = u"Hello, world!";
char16_t c = u'B';

char32_t *t = U"Hello, world!";
char32_t d = U'B';
```

Giờ, giá trị trong mấy cái này có được lưu theo UTF-16 hay UTF-32 không? Tùy implementation.

Nhưng bạn có thể kiểm tra xem có phải không. Nếu các macro `__STDC_UTF_16__` hoặc `__STDC_UTF_32__` được định nghĩa (thành `1`) nghĩa là các kiểu giữ UTF-16 hoặc UTF-32, tương ứng.

Nếu bạn tò mò, và tôi biết bạn tò mò, các giá trị, nếu là UTF-16 hay UTF-32, được lưu theo endianness của máy. Tức là, bạn có thể so chúng thẳng với giá trị code point Unicode:

```
char16_t pi = u"\u03C0";  // pi symbol

#if __STDC_UTF_16__
pi == 0x3C0;  // Always true
#else
pi == 0x3C0;  // Probably not true
#endif
```

27.11.3 Chuyển Multibyte

Bạn có thể chuyển từ encoding multibyte sang `char16_t` hay `char32_t` bằng vài hàm hỗ trợ.

(Như tôi đã nói, kết quả có thể không phải UTF-16 hay UTF-32 trừ khi macro tương ứng được đặt thành `1`.)

Mọi hàm này đều restartable (tức là bạn truyền vào `mbstate_t` riêng), và mọi hàm đều thao tác theo từng ký tự[10].

Hàm chuyển	Mô tả
`mbrtoc16()`	Chuyển ký tự multibyte sang ký tự `char16_t`.
`mbrtoc32()`	Chuyển ký tự multibyte sang ký tự `char32_t`.
`c16rtomb()`	Chuyển ký tự `char16_t` sang ký tự multibyte.
`c32rtomb()`	Chuyển ký tự `char32_t` sang ký tự multibyte.

27.11.4 Thư viện bên thứ ba

Cho chuyển hạng nặng giữa các encoding cụ thể khác nhau, có vài thư viện chín muồi đáng xem. Lưu ý tôi chưa dùng cái nào trong số này.

- iconv[11], Internationalization Conversion, một API chuẩn POSIX phổ biến có sẵn trên các nền tảng lớn.
- ICU[12], International Components for Unicode. Ít nhất một blogger thấy cái này dễ dùng.

Nếu bạn biết thư viện đáng chú ý khác, cho tôi biết.

[10] Kiểu kiểu vậy, mọi thứ trở nên lạ với encoding UTF-16 multi- `char16_t`.

[11] https://en.wikipedia.org/wiki/Iconv

[12] http://site.icu-project.org/

Chapter 28

Thoát khỏi chương trình

Hóa ra có khá nhiều cách để làm chuyện này, và còn cả cách cài "móc" để hàm nào đó chạy khi chương trình thoát.

Trong chương này ta sẽ đào vào và xem chúng.

Ta đã nói về ý nghĩa của mã exit status trong phần Exit Status, nên nhảy ngược lại đó và xem lại nếu cần.

Mọi hàm trong phần này nằm ở `<stdlib.h>`.

28.1 Thoát bình thường

Bắt đầu với các cách thoát thường, rồi nhảy sang vài cái hiếm và quái hơn.

Khi bạn thoát chương trình bình thường, mọi luồng I/O mở được flush và file tạm được xóa. Về cơ bản đây là lối thoát đẹp nơi mọi thứ được dọn dẹp và xử lý. Đây là thứ bạn muốn làm gần như mọi lúc, trừ khi có lý do khác.

28.1.1 Trở về từ `main()`

Nếu bạn để ý, `main()` có kiểu trả về là `int` ... nhưng tôi hiếm khi, nếu có, `return` bất cứ gì từ `main()`.

Đó là vì chỉ riêng `main()` (và tôi không thể nhấn mạnh đủ rằng trường hợp đặc biệt này *chỉ* áp dụng cho `main()` chứ không hàm nào khác ở đâu) có *ngầm* `return 0` nếu bạn rơi khỏi đuôi hàm.

Bạn có thể `return` từ `main()` tường minh bất cứ lúc nào bạn muốn, và vài lập trình viên cảm thấy nó *Đúng* hơn khi luôn có `return` ở cuối `main()`. Nhưng nếu bạn bỏ đó, C sẽ đặt một cái đó vào giúp bạn.

Vậy... đây là luật `return` cho `main()`:

- Bạn có thể trả về exit status từ `main()` bằng câu lệnh `return`. `main()` là hàm duy nhất có hành vi đặc biệt này. Dùng `return` trong bất kỳ hàm nào khác chỉ trả về từ hàm đó tới nơi gọi.
- Nếu bạn không `return` tường minh mà chỉ rơi khỏi đuôi của `main()`, y như bạn đã `return 0` hay `EXIT_SUCCESS`.

28.1.2 `exit()`

Cái này cũng đã xuất hiện vài lần. Nếu bạn gọi `exit()` từ bất cứ đâu trong chương trình, nó sẽ thoát tại điểm đó.

Đối số bạn truyền cho `exit()` là exit status.

28.1.3 Cài exit handler với `atexit()`

Bạn có thể đăng ký các hàm được gọi khi chương trình thoát, dù bằng cách return từ `main()` hay gọi hàm `exit()`.

Một lời gọi `atexit()` với tên hàm handler sẽ xong việc. Bạn có thể đăng ký nhiều exit handler, và chúng sẽ được gọi theo thứ tự ngược lại với thứ tự đăng ký.

Đây là ví dụ:

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

void on_exit_1(void)
{
    printf("Exit handler 1 called!\n");
}

void on_exit_2(void)
{
    printf("Exit handler 2 called!\n");
}

int main(void)
{
    atexit(on_exit_1);
    atexit(on_exit_2);

    printf("About to exit...\n");
}
```

Và output là:

```
About to exit...
Exit handler 2 called!
Exit handler 1 called!
```

28.2 Thoát nhanh hơn với `quick_exit()`

Cái này tương tự thoát thường, trừ:

- File mở có thể không được flush.
- File tạm có thể không được xóa.
- Handler `atexit()` sẽ không được gọi.

Nhưng có cách để đăng ký exit handler: gọi `at_quick_exit()` tương tự cách bạn gọi `atexit()`.

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

void on_quick_exit_1(void)
{
    printf("Quick exit handler 1 called!\n");
}

void on_quick_exit_2(void)
{
```

```
    printf("Quick exit handler 2 called!\n");
}

void on_exit(void)
{
    printf("Normal exit--I won't be called!\n");
}

int main(void)
{
    at_quick_exit(on_quick_exit_1);
    at_quick_exit(on_quick_exit_2);

    atexit(on_exit);  // This won't be called

    printf("About to quick exit...\n");

    quick_exit(0);
}
```

Cho ra output:

```
About to quick exit...
Quick exit handler 2 called!
Quick exit handler 1 called!
```

Nó chạy y như `exit()` / `atexit()`, trừ việc flush file và dọn dẹp có thể không được làm.

28.3 Bắn nó từ quỹ đạo: `_Exit()`

Gọi `_Exit()` thoát ngay lập tức, hết chuyện. Không có hàm callback on-exit nào được thực thi. File sẽ không được flush. File tạm sẽ không được xóa.

Dùng cái này nếu bạn phải thoát *ngay tức khắc*.

28.4 Thoát đôi khi: `assert()`

Câu lệnh `assert()` được dùng để ép một điều gì đó phải đúng, không thì chương trình sẽ thoát.

Dev thường dùng assert để bắt lỗi kiểu Never-Should-Happen (không bao giờ nên xảy ra).

```
#define PI 3.14159

assert(PI > 3);   // Sure enough, it is, so carry on
```

so với:

```
goats -= 100;

assert(goats >= 0);  // Can't have negative goats
```

Trong trường hợp đó, nếu tôi cố chạy nó và `goats` tụt dưới `0`, chuyện này xảy ra:

```
goat_counter: goat_counter.c:8: main: Assertion `goats >= 0' failed.
Aborted
```

và tôi bị đá về dòng lệnh.

Cái này không thân thiện lắm với người dùng, nên nó chỉ được dùng cho mấy thứ mà người dùng sẽ không bao giờ thấy. Và thường người ta tự viết macro assert riêng có thể tắt dễ hơn.

28.5 Thoát bất thường: `abort()`

Bạn có thể dùng cái này nếu có gì đó sai khủng khiếp và bạn muốn báo như vậy cho môi trường bên ngoài. Cái này cũng không nhất thiết dọn dẹp file mở nào.

Tôi hiếm thấy cái này được dùng.

Hé lộ chút về *signal*: cái này thực ra hoạt động bằng cách raise một `SIGABRT` sẽ kết thúc tiến trình.

Chuyện gì xảy ra sau đó tùy hệ thống, nhưng trên các hệ Unix-like, thường dump core[1] khi chương trình kết thúc.

[1] https://en.wikipedia.org/wiki/Core_dump

Chapter 29

Xử lý signal

Trước khi bắt đầu, tôi muốn khuyên bạn nên bỏ qua cả chương này và dùng các hàm xử lý signal (rất có thể) ngon hơn của OS. Các hệ Unix-like có họ hàm `sigaction()`, còn Windows thì có... thứ gì đó của nó[1].

Dẹp chuyện đó sang bên, vậy signal là gì?

29.1 Signal là gì?

Một *signal* được *raise* khi có đủ kiểu sự kiện bên ngoài xảy ra. Chương trình bạn có thể được cấu hình để bị ngắt nhằm *handle* signal, và tuỳ chọn, chạy tiếp chỗ bị bỏ dở sau khi đã xử lý xong.

Nghĩ nó như một hàm được gọi tự động khi một trong các sự kiện ngoài này xảy ra.

Các sự kiện này là gì? Trên hệ của bạn, có lẽ có kha khá, nhưng trong spec C chỉ có vài cái:

Signal	Mô tả
`SIGABRT`	Kết thúc bất thường, thứ xảy ra khi `abort()` được gọi.
`SIGFPE`	Ngoại lệ dấu chấm động.
`SIGILL`	Lệnh không hợp lệ.
`SIGINT`	Ngắt, thường là kết quả của việc bấm `CTRL-C`.
`SIGSEGV`	"Segmentation Violation": truy cập bộ nhớ không hợp lệ.
`SIGTERM`	Yêu cầu kết thúc.

Bạn có thể cài chương trình để bỏ qua, xử lý, hoặc cho chạy hành vi mặc định đối với từng signal bằng hàm `signal()`.

29.2 Xử lý signal với `signal()`

Lời gọi `signal()` nhận hai tham số: signal cần quan tâm, và hành động cần làm khi signal đó được raise.

Hành động có thể là một trong ba thứ:

- Một con trỏ tới hàm xử lý (handler).
- `SIG_IGN` để bỏ qua signal.
- `SIG_DFL` để khôi phục handler mặc định cho signal.

Viết một chương trình mà bạn không `CTRL-C` ra nổi. (Đừng lo, trong chương trình sau, bạn cũng có thể bấm `RETURN` để thoát.)

[1] Hình như Windows không làm signal kiểu Unix ở tầng sâu, và chúng được giả lập cho các app console.

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

int main(void)
{
    char s[1024];

    signal(SIGINT, SIG_IGN);    // Ignore SIGINT, caused by ^C

    printf("Try hitting ^C... (hit RETURN to exit)\n");

    // Wait for a line of input so the program doesn't just exit
    fgets(s, sizeof s, stdin);
}
```

Để ý dòng 8, ta bảo chương trình bỏ qua `SIGINT`, signal ngắt được raise khi `CTRL-C` được bấm. Bạn bấm bao nhiêu tuỳ thích, signal vẫn bị ngó lơ. Nếu bạn comment dòng 8 đi, bạn sẽ thấy có thể `CTRL-C` thoải mái và thoát chương trình tại chỗ.

29.3 Viết signal handler

Tôi có nói rằng bạn cũng có thể viết một hàm handler được gọi khi signal được raise.

Mấy cái này khá đơn giản, nhưng cũng rất bị giới hạn về năng lực khi dính tới spec.

Trước khi bắt đầu, xem prototype của `signal()`:

```
void (*signal(int sig, void (*func)(int)))(int);
```

Dễ đọc chưa?

SAI! `:)`

Dành chút để tháo nó ra cho quen tay.

`signal()` nhận hai đối số: một số nguyên `sig` đại diện cho signal, và một con trỏ `func` tới handler (handler trả về `void` và nhận một `int` làm đối số), tô đậm phía dưới:

```
               sig          func
             |-----|  |---------------|
void (*signal(int sig, void (*func)(int)))(int);
```

Về cơ bản, ta sẽ truyền vào số signal cần bắt, và truyền một con trỏ tới hàm có dạng:

```
void f(int x);
```

hàm đó sẽ làm phần bắt signal thực sự.

Giờ, còn phần còn lại của prototype thì sao? Về cơ bản đó là toàn bộ kiểu trả về. Thấy không, `signal()` sẽ trả về bất cứ thứ gì bạn truyền làm `func` khi thành công... tức là nó đang trả về một con trỏ tới hàm trả về `void` và nhận `int` làm đối số.

```
returned
function    indicates we're             and
returns     returning a                 that function
void        pointer to function         takes an int
|--|        |                                  |---|
void       (*signal(int sig, void (*func)(int)))(int);
```

Ngoài ra, nó có thể trả về `SIG_ERR` khi có lỗi.

Làm một ví dụ bạn phải bấm `CTRL-C` hai lần mới thoát.

Tôi muốn nói rõ rằng chương trình này dính hành vi không xác định (undefined behavior) ở vài chỗ. Nhưng nó chắc sẽ chạy với bạn, và khó nghĩ ra demo di động mà không trivial.

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

int count = 0;

void sigint_handler(int signum)
{
    // The compiler is allowed to run:
    //
    //   signal(signum, SIG_DFL)
    //
    // when the handler is called. So we reset the handler here:
    signal(SIGINT, sigint_handler);

    (void)signum;   // Get rid of unused variable warning

    count++;                        // Undefined behavior
    printf("Count: %d\n", count);  // Undefined behavior

    if (count == 2) {
        printf("Exiting!\n");      // Undefined behavior
        exit(0);
    }
}

int main(void)
{
    signal(SIGINT, sigint_handler);

    printf("Try hitting ^C...\n");

    for(;;);  // Wait here forever
}
```

Một điều bạn sẽ để ý là ở dòng 14 ta reset signal handler. Đó là vì C có quyền reset signal handler về hành vi `SIG_DFL` trước khi chạy handler tuỳ chỉnh của bạn. Nói cách khác, nó có thể chỉ chạy một lần. Nên ta reset ngay lập tức để bắt được lần kế tiếp.

Ta bỏ qua giá trị trả về của `signal()` trong trường hợp này. Nếu ta đã set một handler khác trước đó, nó sẽ trả về con trỏ tới handler đó, mà ta có thể lấy kiểu này:

```
// old_handler is type "pointer to function that takes a single
// int parameter and returns void":

void (*old_handler)(int);

old_handler = signal(SIGINT, sigint_handler);
```

Nói thật tôi không rõ use case phổ biến cho chuyện này. Nhưng nếu bạn cần handler cũ vì lý do nào đó, bạn có thể lấy theo cách đó.

Ghi chú nhanh về dòng 16, đó chỉ là cách báo compiler đừng warning rằng ta không dùng biến này. Giống như nói, "Tôi biết tôi không dùng nó, ông không cần cảnh báo tôi đâu."

Và cuối cùng bạn sẽ thấy tôi đã đánh dấu hành vi không xác định ở vài chỗ. Xem thêm ở phần kế tiếp.

29.4 Ta thực sự làm được gì?

Hoá ra ta khá bị giới hạn về những gì có thể và không thể làm trong signal handler. Đây là một trong những lý do tôi bảo bạn đừng thèm dính vào cái này và dùng signal handling của OS thay thế (ví dụ `sigaction()` cho các hệ Unix-like).

Wikipedia nói thẳng rằng thứ duy nhất thực sự di động bạn làm được là gọi `signal()` với `SIG_IGN` hay `SIG_DFL`, thế thôi.

Đây là những gì ta **không thể** làm một cách di động:

- Gọi bất cứ hàm thư viện chuẩn nào.
  - Như `printf()` chẳng hạn.
  - Tôi nghĩ gọi các hàm có thể restart/reentrant là tương đối an toàn, nhưng spec không cho phép cái đặc quyền đó.
- Lấy hay set giá trị từ một biến `static` cục bộ, scope file, hay thread-local.
  - Trừ khi nó là lock-free atomic object hoặc...
  - Bạn đang gán vào biến kiểu `volatile sig_atomic_t`.

Cái cuối đó, `sig_atomic_t`, là tấm vé để bạn đưa dữ liệu ra khỏi signal handler. (Trừ khi bạn muốn dùng lock-free atomic object, vốn nằm ngoài phạm vi phần này[2].) Nó là kiểu số nguyên, có thể có dấu hoặc không. Và nó bị giới hạn bởi thứ bạn có thể nhét vào.

Bạn có thể xem giá trị min và max cho phép trong macro `SIG_ATOMIC_MIN` và `SIG_ATOMIC_MAX`[3].

Gây bối rối là spec cũng nói bạn không được "refer tới bất kỳ object nào có static hay thread storage duration mà không phải lock-free atomic object ngoại trừ bằng cách gán giá trị vào một object được khai báo là `volatile sig_atomic_t` [...]"

Tôi hiểu ý này là bạn không thể đọc hay ghi bất cứ gì không phải lock-free atomic object. Ngoài ra bạn có thể gán vào một object `volatile sig_atomic_t`.

Nhưng bạn đọc từ nó được không? Thật lòng tôi không thấy lý do gì không được, trừ việc spec rất chăm chỉ nhắc chuyện "gán vào". Nhưng nếu bạn phải đọc nó và ra quyết định dựa trên đó, bạn có thể mở ra chỗ cho race condition nào đó.

Có cái đó trong đầu, ta có thể viết lại đoạn "bấm `CTRL-C` hai lần để thoát" sao cho di động hơn chút, tuy output có kiệm lời hơn.

Đổi handler `SIGINT` của ta để không làm gì ngoại trừ tăng một giá trị kiểu `volatile sig_atomic_t`. Nó sẽ đếm số lần `CTRL-C` đã được bấm.

Rồi trong vòng lặp main, ta sẽ kiểm tra xem counter đó đã vượt quá `2` chưa, và bail ra nếu có.

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

volatile sig_atomic_t count = 0;

void sigint_handler(int signum)
{
    (void)signum;                    // Unused variable warning
```

[2] Gây bối rối là `sig_atomic_t` có trước lock-free atomic và không phải cùng một thứ.

[3] Nếu `sig_action_t` có dấu, range sẽ ít nhất là `-127` tới `127`. Nếu không dấu, ít nhất `0` tới `255`.

```
    signal(SIGINT, sigint_handler);  // Reset signal handler

    count++;                         // Undefined behavior
}

int main(void)
{
    signal(SIGINT, sigint_handler);

    printf("Hit ^C twice to exit.\n");

    while(count < 2);
}
```

Lại hành vi không xác định? Tôi đọc đây là có, vì ta phải đọc giá trị để tăng rồi lưu lại. Một thread khác có thể nghịch `count` và làm ta phát cáu. Nhưng trong ví dụ đơn giản này, không có thread khác làm chuyện đó, nên ta bỏ qua được và tận hưởng demo.

Nếu ta chỉ muốn trì hoãn thoát thêm một lần bấm `CTRL-C`, ta làm được mà không khổ lắm. Nhưng thêm nữa thì cần mấy chuỗi hàm nhố nhăng.

Cái ta sẽ làm là xử lý một lần, và handler sẽ reset signal về hành vi mặc định (tức là thoát):

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

void sigint_handler(int signum)
{
    (void)signum;                    // Unused variable warning
    signal(SIGINT, SIG_DFL);         // Reset signal handler
}

int main(void)
{
    signal(SIGINT, sigint_handler);

    printf("Hit ^C twice to exit.\n");

    while(1);
}
```

Sau này khi nhìn vào biến lock-free atomic, ta sẽ thấy cách sửa phiên bản dùng `count` (giả sử biến lock-free atomic có sẵn trên hệ cụ thể của bạn).

Đó là lý do ngay từ đầu tôi đã gợi ý bạn check signal system tích hợp sẵn của OS như phương án nhiều khả năng ngon hơn.

29.5 Bạn Hiền Không Để Bạn Hiền `signal()`

Lần nữa, dùng signal handling tích hợp sẵn của OS hay cái tương đương. Nó không có trong spec, không di động bằng, nhưng có lẽ mạnh hơn nhiều. Cộng thêm OS của bạn có lẽ định nghĩa một số signal không có trong spec C. Và viết code di động dùng `signal()` dù sao cũng khó.

Chapter 30

Mảng độ dài biến đổi (VLA)

C cung cấp cách khai báo mảng mà kích thước được xác định lúc chạy. Cái này cho bạn lợi ích của việc chỉnh kích thước động lúc runtime như với `malloc()`, nhưng không cần lo `free()` bộ nhớ sau đó.

Giờ, nhiều người không thích VLA. Chúng bị cấm trong Linux kernel chẳng hạn. Ta sẽ đào sâu hơn về lý do đó ở sau.

Đây là tính năng tuỳ chọn của ngôn ngữ. Macro `__STDC_NO_VLA__` được set là `1` nếu VLA *không* có. (Chúng bắt buộc trong C99, rồi thành tuỳ chọn trong C11.)

```
#if __STDC_NO_VLA__ == 1
   #error Sorry, need VLAs for this program!
#endif
```

Nhưng vì cả GCC và Clang đều không buồn định nghĩa macro này, bạn có thể chẳng đi được mấy với nó.

Nhảy vào với một ví dụ trước, rồi ta sẽ đi tìm con quỷ trong chi tiết.

30.1 Cơ bản

Một mảng thường được khai báo với kích thước hằng, như sau:

```
int v[10];
```

Nhưng với VLA, ta có thể dùng kích thước xác định lúc runtime để đặt mảng, như sau:

```
int n = 10;
int v[n];
```

Giờ, trông thì giống y như nhau, và ở nhiều mặt nó giống thật, nhưng cái này cho bạn sự linh hoạt để tính kích thước cần, rồi lấy một mảng chính xác kích thước đó.

Ta hãy hỏi người dùng nhập kích thước mảng, rồi lưu chỉ-số-nhân-10 vào mỗi phần tử mảng:

```
#include <stdio.h>

int main(void)
{
    int n;
    char buf[32];

    printf("Enter a number: "); fflush(stdout);
```

```
    fgets(buf, sizeof buf, stdin);
    n = strtoul(buf, NULL, 10);

    int v[n];

    for (int i = 0; i < n; i++)
        v[i] = i * 10;

    for (int i = 0; i < n; i++)
        printf("v[%d] = %d\n", i, v[i]);
}
```

(Ở dòng 7, tôi có `fflush()` để ép dòng được xuất ra dù tôi không có newline ở cuối.)

Dòng 12 là chỗ ta khai báo VLA, một khi thực thi đi qua dòng đó, kích thước mảng được set bằng bất cứ giá trị nào `n` có tại thời điểm đó. Độ dài mảng không thể đổi sau này.

Bạn có thể đặt biểu thức trong ngoặc vuông cũng được:

```
int v[x * 100];
```

Vài hạn chế:

- Bạn không thể khai báo VLA ở file scope, và không thể tạo một VLA `static` trong block scope[1].
- Bạn không thể dùng initializer list để khởi tạo mảng.

Ngoài ra, nhập giá trị âm cho kích thước mảng sẽ gây hành vi không xác định, dù sao thì cũng trong vũ trụ này.

30.2 `sizeof` và VLA

Ta đã quen với việc `sizeof` cho ra kích thước tính bằng byte của một object cụ thể, kể cả mảng. Và VLA cũng không ngoại lệ.

Khác biệt chính là `sizeof` trên VLA được chạy lúc *runtime*, còn trên biến không có kích thước biến đổi thì được tính lúc *compile time*.

Nhưng cách dùng vẫn vậy.

Bạn thậm chí có thể tính số phần tử trong VLA bằng trò mảng quen thuộc:

```
size_t num_elems = sizeof v / sizeof v[0];
```

Có một hàm ý tinh tế và đúng từ dòng trên: số học con trỏ chạy y như bạn kỳ vọng với mảng thường. Nên cứ dùng thoả thích:

```
#include <stdio.h>

int main(void)
{
    int n = 5;
    int v[n];

    int *p = v;

    *(p+2) = 12;
    printf("%d\n", v[2]);  // 12
```

[1]Đây là do VLA thường được cấp phát trên stack, còn biến `static` nằm trên heap. Và cả ý tưởng của VLA là chúng sẽ được giải phóng tự động khi stack frame bị pop ở cuối hàm.

```
    p[3] = 34;
    printf("%d\n", v[3]);  // 34
}
```

Giống như với mảng thường, bạn có thể dùng ngoặc với `sizeof()` để lấy kích thước của một VLA giả-định mà không thực sự khai báo nó:

```
int x = 12;

printf("%zu\n", sizeof(int [x]));  // Prints 48 on my system
```

30.3 VLA nhiều chiều

Bạn có thể tạo đủ kiểu VLA với một hoặc nhiều chiều được set làm biến

```
int w = 10;
int h = 20;

int x[h][w];
int y[5][w];
int z[10][w][20];
```

Lại nữa, bạn có thể điều hướng chúng y như mảng thường.

30.4 Truyền VLA một chiều cho hàm

Truyền VLA đơn-chiều vào hàm không khác gì truyền mảng thường. Cứ thế mà làm.

```
#include <stdio.h>

int sum(int count, int *v)
{
    int total = 0;

    for (int i = 0; i < count; i++)
        total += v[i];

    return total;
}

int main(void)
{
    int x[5];   // Standard array

    int a = 5;
    int y[a];   // VLA

    for (int i = 0; i < a; i++)
        x[i] = y[i] = i + 1;

    printf("%d\n", sum(5, x));
    printf("%d\n", sum(a, y));
}
```

Nhưng còn có thêm chút nữa. Bạn cũng có thể cho C biết rằng mảng là VLA cụ thể kích thước nào đó bằng cách truyền kích thước đó trước rồi ghi chiều đó vào danh sách tham số:

```
int sum(int count, int v[count])
{
    // ...
}
```

Nhân tiện, có vài cách liệt kê prototype cho hàm trên; một trong số đó dùng `*` nếu bạn không muốn chỉ cụ thể tên biến giữ giá trị trong VLA. Nó chỉ báo rằng kiểu là VLA chứ không phải con trỏ thường.

Prototype VLA:

```
void do_something(int count, int v[count]);  // With names
void do_something(int, int v[*]);            // Without names
```

Lại nữa, cái `*` đó chỉ dùng với prototype, trong thân hàm bạn sẽ phải đặt kích thước tường minh.

Giờ, *đa chiều thôi*! Đây là chỗ vui bắt đầu.

30.5 Truyền VLA đa chiều cho hàm

Y như ta đã làm với dạng thứ hai của VLA một chiều ở trên, nhưng lần này ta truyền vào hai chiều và dùng chúng.

Trong ví dụ sau, ta dựng một ma trận bảng cửu chương chiều rộng và chiều cao biến đổi, rồi truyền cho một hàm để in ra.

```
#include <stdio.h>

void print_matrix(int h, int w, int m[h][w])
{
    for (int row = 0; row < h; row++) {
        for (int col = 0; col < w; col++)
            printf("%2d ", m[row][col]);
        printf("\n");
    }
}

int main(void)
{
    int rows = 4;
    int cols = 7;

    int matrix[rows][cols];

    for (int row = 0; row < rows; row++)
        for (int col = 0; col < cols; col++)
            matrix[row][col] = row * col;

    print_matrix(rows, cols, matrix);
}
```

30.5.1 VLA đa chiều một phần

Bạn có thể có một số chiều cố định và một số biến đổi. Giả sử ta có một bản ghi độ dài cố định 5 phần tử, nhưng ta không biết có bao nhiêu bản ghi.

```
#include <stdio.h>

void print_records(int count, int record[count][5])
{
    for (int i = 0; i < count; i++) {
        for (int j = 0; j < 5; j++)
            printf("%2d ", record[i][j]);
        printf("\n");
    }
}

int main(void)
{
    int rec_count = 3;
    int records[rec_count][5];

    // Fill with some dummy data
    for (int i = 0; i < rec_count; i++)
        for (int j = 0; j < 5; j++)
            records[i][j] = (i+1)*(j+2);

    print_records(rec_count, records);
}
```

30.6 Tương thích với mảng thường

Vì VLA y như mảng thường trong bộ nhớ, hoàn toàn cho phép truyền chúng lẫn nhau... miễn là các chiều khớp.

Ví dụ, nếu ta có một hàm muốn mảng 3×5 cụ thể, ta vẫn có thể truyền một VLA vào đó.

```
int foo(int m[5][3]) {...}

\\ ...

int w = 3, h = 5;
int matrix[h][w];

foo(matrix);   // OK!
```

Tương tự, nếu bạn có một hàm VLA, bạn có thể truyền mảng thường vào:

```
int foo(int h, int w, int m[h][w]) {...}

\\ ...

int matrix[3][5];

foo(3, 5, matrix);   // OK!
```

Coi chừng nhé: nếu chiều không khớp, bạn sẽ có hành vi không xác định, rất có khả năng.

30.7 `typedef` và VLA

Bạn có thể `typedef` một VLA, nhưng hành vi có thể không như bạn mong đợi.

Về cơ bản, `typedef` tạo một kiểu mới với các giá trị như chúng tồn tại tại thời điểm `typedef` được chạy.

Nên nó không hẳn là một `typedef` của VLA mà là một kiểu mảng kích thước cố định mới với các chiều tại thời điểm đó.

```
#include <stdio.h>

int main(void)
{
    int w = 10;

    typedef int goat[w];

    // goat is an array of 10 ints
    goat x;

    // Init with squares of numbers
    for (int i = 0; i < w; i++)
        x[i] = i*i;

    // Print them
    for (int i = 0; i < w; i++)
        printf("%d\n", x[i]);

    // Now let's change w...

    w = 20;

    // But goat is STILL an array of 10 ints, because that was the
    // value of w when the typedef executed.
}
```

Nên nó hành xử như một mảng kích thước cố định.

Nhưng bạn vẫn không thể dùng initializer list trên nó.

30.8 Bẫy nhảy lung tung

Bạn phải coi chừng khi dùng `goto` gần VLA vì nhiều thứ không hợp lệ.

Và khi bạn dùng `longjmp()` có trường hợp bạn có thể leak bộ nhớ với VLA.

Nhưng cả hai thứ này ta sẽ bàn trong chương riêng của chúng.

30.9 Vấn đề chung

VLA đã bị cấm khỏi Linux kernel vì vài lý do:

- Nhiều chỗ chúng được dùng lẽ ra nên là kích thước cố định.
- Code đằng sau VLA chậm hơn (tới mức mà đa số không để ý, nhưng tạo khác biệt trong hệ điều hành).
- VLA không được hỗ trợ đồng đều bởi mọi trình biên dịch C.
- Kích thước stack bị giới hạn, và VLA nằm trên stack. Nếu đoạn code nào đó vô tình (hoặc ác ý) truyền giá trị lớn vào một hàm kernel cấp phát VLA, *Chuyện Xấu*™ có thể xảy ra.

Nhiều người khác online chỉ ra rằng không có cách nào phát hiện VLA thất bại khi cấp phát, và chương trình dính vấn đề như vậy khả năng chỉ có crash. Dù mảng kích thước cố định cũng có vấn đề y vậy, khả năng cao hơn nhiều là ai đó lỡ tay làm *VLA Kích Thước Bất Thường* hơn là ai đó vô tình khai báo một mảng cố định ví dụ 30 megabyte.

Chapter 31

`goto`

Câu lệnh `goto` được cả thế giới tôn sùng và có thể trình ra đây không ai cãi được.

Đùa thôi! Qua năm tháng, đã có cả đống tranh cãi qua lại về việc `goto` có bị coi là có hại[1] hay không (thường là có).

Theo ý của programmer này, bạn nên dùng cấu trúc nào dẫn tới code *tốt nhất*, có tính tới bảo trì và tốc độ. Và đôi khi cái đó có thể là `goto`!

Trong chương này, ta sẽ xem `goto` hoạt động sao trong C, rồi ngó qua vài trường hợp hay dùng[2].

31.1 Một ví dụ đơn giản

Trong ví dụ này, ta sẽ dùng `goto` để bỏ qua một dòng code và nhảy tới một *label*. Label là identifier có thể làm đích của `goto`, nó kết thúc bằng dấu hai chấm (`:`).

```
#include <stdio.h>

int main(void)
{
    printf("One\n");
    printf("Two\n");

    goto skip_3;

    printf("Three\n");

skip_3:

    printf("Five!\n");
}
```

Output là:

```
One
Two
Five!
```

`goto` đẩy thực thi nhảy tới label đã chỉ định, bỏ qua mọi thứ ở giữa.

[1] https://en.wikipedia.org/wiki/Goto#Criticism

[2] Tôi muốn nói rõ rằng dùng `goto` trong tất cả các trường hợp này đều tránh được. Bạn có thể dùng biến và vòng lặp thay thế. Chỉ là có người thấy `goto` tạo code *tốt nhất* trong những hoàn cảnh đó.

Bạn có thể nhảy tiến hay lùi với `goto`.

```
infinite_loop:
    print("Hello, world!\n");
    goto infinite_loop;
```

Label bị bỏ qua khi thực thi. Cái sau sẽ in cả ba số theo thứ tự y như thể các label không có mặt:

```
    printf("Zero\n");
label_1:
label_2:
    printf("One\n");
label_3:
    printf("Two\n");
label_4:
    printf("Three\n");
```

Như bạn đã để ý, quy ước phổ biến là căn lề label sát bên trái. Điều này tăng khả năng đọc vì người đọc có thể quét nhanh để tìm đích.

Label có *function scope*. Tức là, dù chúng xuất hiện ở mức block sâu bao nhiêu, bạn vẫn có thể `goto` chúng từ bất cứ đâu trong hàm.

Điều đó cũng có nghĩa là bạn chỉ có thể `goto` các label nằm trong cùng hàm với `goto`. Label ở các hàm khác là ngoài scope theo góc nhìn của `goto`. Và có nghĩa là bạn có thể dùng cùng tên label trong hai hàm khác nhau, chỉ không được dùng cùng tên label trong cùng một hàm.

31.2 `continue` có label

Ở vài ngôn ngữ, bạn thực sự có thể chỉ định label cho câu lệnh `continue`. C không cho, nhưng bạn có thể dễ dàng dùng `goto` thay thế.

Để thấy vấn đề, xem `continue` trong vòng lặp lồng này:

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        printf("%d, %d\n", i, j);
        continue;   // Always goes to next j
    }
}
```

Như ta thấy, `continue` đó, giống như mọi `continue`, đi tới lần lặp kế của vòng lặp bao quanh gần nhất. Nếu ta muốn `continue` ở vòng lặp ngoài kế tiếp, vòng lặp với `i` thì sao?

Thì, ta có thể `break` để ra lại vòng lặp ngoài, đúng không?

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        printf("%d, %d\n", i, j);
        break;     // Gets us to the next iteration of i
    }
}
```

Cái đó giải quyết được hai mức lồng. Nhưng rồi nếu ta lồng thêm vòng nữa, ta hết lựa chọn. Còn cái này, nơi ta không có câu lệnh nào đưa ta ra tới lần lặp kế của `i`?

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        for (int k = 0; k < 3; k++) {
            printf("%d, %d, %d\n", i, j, k);

            continue;  // Gets us to the next iteration of k
            break;     // Gets us to the next iteration of j
            ????;      // Gets us to the next iteration of i???

        }
    }
}
```

Câu lệnh `goto` cho ta lối!

```
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            for (int k = 0; k < 3; k++) {
                printf("%d, %d, %d\n", i, j, k);

                goto continue_i;   // Now continuing the i loop!!
            }
        }
continue_i: ;
    }
```

Ta có `;` ở cuối đó, vì bạn không thể có label chỉ tới chỗ cuối thuần của compound statement (hay trước một khai báo biến).

31.3 Thoát thân

Khi bạn đang lồng cực sâu giữa mớ code, bạn có thể dùng `goto` để thoát ra theo cách thường sạch hơn là lồng thêm `if` và dùng biến cờ.

```
    // Pseudocode

    for(...) {
        for (...) {
            while (...) {
                do {
                    if (some_error_condition)
                        goto bail;

                } while(...);
            }
        }
    }

bail:
    // Cleanup here
```

Không có `goto`, bạn sẽ phải check cờ điều kiện lỗi trong tất cả các vòng lặp để thoát hết.

31.4 break có label

Tình huống rất giống với chuyện `continue` chỉ continue vòng lặp trong cùng. `break` cũng chỉ break khỏi vòng lặp trong cùng.

```
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d, %d\n", i, j);
            break;   // Only breaks out of the j loop
        }
    }

    printf("Done!\n");
```

Nhưng ta có thể dùng `goto` để break xa hơn:

```
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d, %d\n", i, j);
            goto break_i;   // Now breaking out of the i loop!
        }
    }

break_i:

    printf("Done!\n");
```

31.5 Dọn dẹp nhiều tầng

Nếu bạn đang gọi nhiều hàm để khởi tạo nhiều hệ thống con và một trong số đó fail, bạn chỉ nên de-initialize các cái mà bạn đã tới được cho tới giờ.

Làm một ví dụ giả trong đó ta bắt đầu khởi tạo hệ thống và check xem có cái nào trả về lỗi (ta dùng `-1` để báo lỗi). Nếu có, ta phải tắt chỉ những hệ thống mà ta đã khởi tạo đến lúc đó.

```
    if (init_system_1() == -1)
        goto shutdown;

    if (init_system_2() == -1)
        goto shutdown_1;

    if (init_system_3() == -1)
        goto shutdown_2;

    if (init_system_4() == -1)
        goto shutdown_3;

    do_main_thing();   // Run our program

    shutdown_system4();

shutdown_3:
    shutdown_system3();

shutdown_2:
    shutdown_system2();
```

```
shutdown_1:
    shutdown_system1();

shutdown:
    print("All subsystems shut down.\n");
```

Lưu ý rằng ta tắt theo thứ tự ngược với thứ tự khởi tạo hệ thống con. Nên nếu hệ con 4 fail khi khởi động, nó sẽ tắt 3, 2, rồi 1 theo thứ tự đó.

31.6 Tối ưu tail call

Kinda. Chỉ cho hàm đệ quy.

Nếu bạn chưa quen, Tail Call Optimization (TCO)[3] là cách không phí stack space khi gọi hàm khác trong các tình huống rất cụ thể. Không may chi tiết nằm ngoài phạm vi guide này.

Nhưng nếu bạn có một hàm đệ quy bạn biết có thể được tối ưu theo kiểu này, bạn có thể tận dụng kỹ thuật này. (Lưu ý bạn không thể tail call hàm khác vì label có function scope.)

Làm ví dụ thẳng thắn, giai thừa.

Đây là phiên bản đệ quy không phải TCO, nhưng có thể!

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

int factorial(int n, int a)
{
    if (n == 0)
        return a;

    return factorial(n - 1, a * n);
}

int main(void)
{
    for (int i = 0; i < 8; i++)
        printf("%d! == %ld\n", i, factorial(i, 1));
}
```

Để biến nó thành TCO, bạn có thể thay lời gọi bằng hai bước:

1. Set giá trị các tham số sang giá trị sẽ có ở lời gọi kế.
2. `goto` một label ở dòng đầu tiên của hàm.

Thử xem:

```
#include <stdio.h>

int factorial(int n, int a)
{
tco:  // add this

    if (n == 0)
        return a;

    // replace return by setting new parameter values and
```

[3] https://en.wikipedia.org/wiki/Tail_call

```
    // goto-ing the beginning of the function

    //return factorial(n - 1, a * n);

    int next_n = n - 1;  // See how these match up with
    int next_a = a * n;  // the recursive arguments, above?

    n = next_n;   // Set the parameters to the new values
    a = next_a;

    goto tco;   // And repeat!
}

int main(void)
{
    for (int i = 0; i < 8; i++)
        printf("%d! == %d\n", i, factorial(i, 1));
}
```

Tôi đã dùng biến tạm phía trên để set giá trị kế của các tham số trước khi nhảy về đầu hàm. Thấy chúng tương ứng với các đối số đệ quy trong lời gọi đệ quy chưa?

Giờ, tại sao dùng biến tạm? Tôi có thể đã làm vầy thay thế:

```
    a *= n;
    n -= 1;

    goto tco;
```

và cái đó thực tế chạy ổn. Nhưng nếu tôi bất cẩn đảo hai dòng code đó:

```
    n -= 1;  // BAD NEWS
    a *= n;
```

giờ ta gặp rắc rối. Ta sửa đổi `n` trước khi dùng nó để sửa `a`. Đó là Tệ vì đó không phải cách nó chạy khi bạn gọi đệ quy. Dùng biến tạm tránh được vấn đề này kể cả khi bạn không để ý. Và compiler khả năng cao tối ưu chúng đi thôi.

31.7 Khởi động lại system call bị ngắt

Cái này nằm ngoài spec, nhưng thường thấy ở các hệ Unix-like.

Một số system call lâu có thể trả lỗi nếu bị ngắt bởi signal, và `errno` sẽ được set thành `EINTR` để báo rằng syscall vẫn ổn, chỉ là bị ngắt.

Trong các trường hợp đó, rất phổ biến việc lập trình viên muốn chạy lại lời gọi và thử lại.

```
retry:
    byte_count = read(0, buf, sizeof(buf) - 1);  // Unix read() syscall

    if (byte_count == -1) {            // An error occurred...
        if (errno == EINTR) {          // But it was just interrupted
            printf("Restarting...\n");
            goto retry;
        }
```

Nhiều hệ Unix-like có cờ `SA_RESTART` bạn có thể truyền cho `sigaction()` để yêu cầu OS tự khởi động lại các syscall chậm thay vì fail với `EINTR`.

Lại nữa, cái này đặc thù Unix và nằm ngoài chuẩn C.

Nói vậy, có thể dùng kỹ thuật tương tự bất cứ khi nào có hàm nào nên được khởi động lại.

31.8 `goto` và preempt thread

Ví dụ này được lấy thẳng từ *Operating Systems: Three Easy Pieces*, một cuốn sách tuyệt vời nữa từ các tác giả cùng tư tưởng cũng cho rằng sách chất lượng nên được tải miễn phí. Không phải tôi có quan điểm gì đâu.

```
retry:

    pthread_mutex_lock(L1);

    if (pthread_mutex_trylock(L2) != 0) {
        pthread_mutex_unlock(L1);
        goto retry;
    }

    save_the_day();

    pthread_mutex_unlock(L2);
    pthread_mutex_unlock(L1);
```

Ở đó thread vui vẻ lấy được mutex `L1`, nhưng rồi tiềm năng fail khi lấy tài nguyên thứ hai được bảo vệ bởi mutex `L2` (nếu một thread khác không hợp tác đang giữ, chẳng hạn). Nếu thread của ta không lấy được khoá `L2`, nó mở khoá `L1` rồi dùng `goto` để thử lại sạch sẽ.

Ta hy vọng thread anh hùng của ta rốt cuộc lấy được cả hai mutex và cứu cả ngày, tránh được deadlock tà ác.

31.9 `goto` và scope của biến

Ta đã thấy label có function scope, nhưng chuyện lạ có thể xảy ra nếu ta nhảy qua phần khởi tạo biến.

Xem ví dụ này nơi ta nhảy từ một chỗ mà biến `x` ngoài scope vào giữa scope của nó (trong block).

```
    goto label;

    {
        int x = 12345;

label:
        printf("%d\n", x);
    }
```

Cái này sẽ compile và chạy, nhưng cho tôi cảnh báo:

```
warning: ‘x’ is used uninitialized in this function
```

Và rồi in ra `0` khi tôi chạy (kết quả có thể khác với bạn).

Về cơ bản chuyện đã xảy ra là ta nhảy vào scope của `x` (nên ok khi tham chiếu nó trong `printf()`) nhưng ta nhảy qua dòng mà thực sự khởi tạo nó thành `12345`. Nên giá trị không xác định.

Cách sửa dĩ nhiên là đưa phần khởi tạo ra *sau* label theo cách nào đó.

```
    goto label;

    {
        int x;

label:
        x = 12345;
        printf("%d\n", x);
    }
```

Xem thêm một ví dụ nữa.

```
    {
        int x = 10;

label:

        printf("%d\n", x);
    }

    goto label;
```

Chuyện gì xảy ra ở đây?

Lần đầu qua block, ta ngon. `x` là `10` và đó là cái được in.

Nhưng sau `goto`, ta nhảy vào scope của `x`, nhưng qua phần khởi tạo của nó. Tức là ta vẫn có thể in nó, nhưng giá trị không xác định (vì nó chưa được khởi tạo lại).

Trên máy tôi, nó in `10` lần nữa (mãi mãi), nhưng đó chỉ là may mắn. Nó có thể in giá trị bất kỳ sau `goto` vì `x` không được khởi tạo.

31.10 `goto` và VLA

Khi dính tới VLA và `goto`, có một quy tắc: bạn không thể nhảy từ ngoài scope của một VLA vào trong scope của VLA đó.

Nếu tôi cố làm vầy:

```
    int x = 10;

    goto label;

    {
        int v[x];

label:

        printf("Hi!\n");
    }
```

Tôi bị lỗi:

```
error: jump into scope of identifier with variably modified type
```

Bạn có thể nhảy tới trước khai báo VLA, như vầy:

```
    int x = 10;

    goto label;

    {
label:  ;
        int v[x];

        printf("Hi!\n");
    }
```

Vì cách đó VLA được cấp phát đúng cách trước khi chắc chắn bị giải phóng khi ra khỏi scope.

Chapter 32

Types Phần V: Compound Literals và Generic Selections

Đây là chương cuối về types! Ta sẽ nói hai chuyện:

- Làm sao có object "ẩn danh" không tên và lợi ích của nó.
- Làm sao tạo code phụ thuộc kiểu.

Chúng không liên quan lắm, nhưng cũng không đáng mỗi cái một chương. Nên tôi nhét chúng vào đây như một kẻ nổi loạn!

32.1 Compound Literals

Đây là một tính năng hay của ngôn ngữ cho phép bạn tạo một object thuộc kiểu nào đó trên đường đi mà không cần gán nó vào biến. Bạn có thể làm kiểu đơn giản, mảng, `struct`, gì cũng được.

Một trong những cách dùng chính của nó là truyền đối số phức tạp cho hàm khi bạn không muốn tạo biến tạm để giữ giá trị.

Cách bạn tạo compound literal là đặt tên kiểu trong ngoặc đơn, rồi đặt một initializer list phía sau. Ví dụ, một mảng `int` không tên có thể trông như vầy:

```
(int []){1,2,3,4}
```

Giờ, dòng code đó tự nó không làm gì cả. Nó tạo một mảng không tên gồm 4 `int`, rồi vứt đi mà không dùng.

Ta có thể dùng một con trỏ để lưu tham chiếu tới mảng...

```
int *p = (int []){1 ,2 ,3 ,4};

printf("%d\n", p[1]);  // 2
```

Nhưng cái đó có vẻ như kiểu vòng vo để có mảng. Ý là, ta cũng có thể đã làm vầy[1]:

```
int p[] = {1, 2, 3, 4};

printf("%d\n", p[1]);  // 2
```

Vậy hãy xem ví dụ hữu ích hơn.

[1] Cũng không hoàn toàn giống, vì nó là mảng chứ không phải con trỏ tới `int`.

32.1.1 Truyền object không tên cho hàm

Giả sử ta có một hàm tính tổng một mảng `int`:

```
int sum(int p[], int count)
{
    int total = 0;

    for (int i = 0; i < count; i++)
        total += p[i];

    return total;
}
```

Nếu ta muốn gọi nó, thường ta phải làm kiểu này, khai báo mảng và lưu giá trị vào nó để truyền cho hàm:

```
int a[] = {1, 2, 3, 4};

int s = sum(a, 4);
```

Nhưng object không tên cho ta cách bỏ qua biến bằng cách truyền thẳng nó vào (tên tham số liệt kê phía trên). Xem này, ta sẽ thay biến `a` bằng một mảng không tên truyền làm đối số đầu:

```
//                  p[]         count
//           |-----------------|  |
int s = sum((int []){1, 2, 3, 4}, 4);
```

Khá gọn!

32.1.2 `struct` không tên

Ta có thể làm điều tương tự với `struct`.

Trước, hãy làm không dùng object không tên. Ta sẽ định nghĩa một `struct` để giữ toạ độ `x` / `y`. Rồi ta định nghĩa một cái, truyền giá trị vào initializer của nó. Cuối cùng, truyền nó cho một hàm để in giá trị ra:

```
#include <stdio.h>

struct coord {
    int x, y;
};

void print_coord(struct coord c)
{
    printf("%d, %d\n", c.x, c.y);
}

int main(void)
{
    struct coord t = {.x=10, .y=20};

    print_coord(t);   // prints "10, 20"
}
```

Đủ thẳng thắn?

Chỉnh nó để dùng object không tên thay cho biến `t` mà ta đang truyền cho `print_coord()`.

Ta chỉ cần rút `t` ra và thay bằng một `struct` không tên:

```
    //struct coord t = {.x=10, .y=20};

    print_coord((struct coord){.x=10, .y=20});   // prints "10, 20"
```

Vẫn chạy!

32.1.3 Con trỏ tới object không tên

Bạn có thể để ý trong ví dụ cuối rằng dù ta đang dùng `struct`, ta truyền một bản sao của `struct` cho `print_coord()` chứ không phải truyền con trỏ tới `struct`.

Hoá ra, ta có thể lấy địa chỉ của một object không tên bằng `&` như thường.

Đó là vì, nhìn chung, nếu một toán tử chạy được với biến thuộc kiểu đó, bạn có thể dùng toán tử đó trên object không tên thuộc cùng kiểu.

Chỉnh code trên để ta truyền con trỏ tới object không tên

```
#include <stdio.h>

struct coord {
    int x, y;
};

void print_coord(struct coord *c)
{
    printf("%d, %d\n", c->x, c->y);
}

int main(void)
{
    //     Note the &
    //          |
    print_coord(&(struct coord){.x=10, .y=20});   // prints "10, 20"
}
```

Thêm nữa, đây có thể là cách hay ngay cả để truyền con trỏ tới object đơn giản:

```
// Pass a pointer to an int with value 3490
foo(&(int){3490});
```

Dễ vậy thôi.

32.1.4 Object không tên và scope

Vòng đời của object không tên kết thúc ở cuối scope của nó. Cách lớn nhất mà chuyện này có thể cắn bạn là nếu bạn tạo một object không tên mới, lấy con trỏ tới nó, rồi rời khỏi scope của object. Trong trường hợp đó, con trỏ sẽ tham chiếu tới một object đã chết.

Nên cái này là hành vi không xác định:

```
int *p;

{
    p = &(int){10};
}
```

```
printf("%d\n", *p);  // INVALID: The (int){10} fell out of scope
```

Tương tự, bạn không thể trả về một con trỏ tới object không tên từ một hàm. Object được giải phóng khi nó rời khỏi scope:

```
#include <stdio.h>

int *get3490(void)
{
    // Don't do this
    return &(int){3490};
}

int main(void)
{
    printf("%d\n", *get3490());  // INVALID: (int){3490} fell out of scope
}
```

Cứ nghĩ scope của chúng giống như biến cục bộ thông thường. Bạn cũng không thể trả về con trỏ tới biến cục bộ.

32.1.5 Ví dụ object không tên hơi ngớ

Bạn có thể đặt kiểu nào vào đó và tạo object không tên cũng được.

Ví dụ, những cái này thực tế tương đương:

```
int x = 3490;

printf("%d\n", x);               // 3490 (variable)
printf("%d\n", 3490);            // 3490 (constant)
printf("%d\n", (int){3490});     // 3490 (unnamed object)
```

Cái cuối là không tên, nhưng ngớ ngẩn. Thà làm cái đơn giản ở dòng trước.

Nhưng hy vọng nó cho thêm chút rõ ràng về cú pháp.

32.2 Generic Selections

Đây là một biểu thức cho phép bạn chọn các đoạn code khác nhau tuỳ vào *type* của đối số đầu của biểu thức.

Ta sẽ xem ví dụ trong tích tắc, nhưng quan trọng là biết rằng cái này được xử lý tại compile time, *không phải runtime*. Không có phân tích runtime nào xảy ra ở đây.

Biểu thức bắt đầu bằng `_Generic`, chạy kiểu như `switch`, và nhận ít nhất hai đối số.

Đối số đầu là một biểu thức (hay biến[2]) có một *type*. Mọi biểu thức đều có type. Các đối số còn lại cho `_Generic` là các case về việc thay gì vào cho kết quả của biểu thức nếu đối số đầu có type đó.

Cái gì cơ?

Thử coi sao.

```
#include <stdio.h>

int main(void)
{
```

[2]Biến dùng ở đây *là* một biểu thức.

```
    int i;
    float f;
    char c;

    char *s = _Generic(i,
                    int: "that variable is an int",
                    float: "that variable is a float",
                    default: "that variable is some type"
                );

    printf("%s\n", s);
}
```

Xem biểu thức `_Generic` bắt đầu ở dòng 9.

Khi compiler thấy nó, nó nhìn vào type của đối số đầu. (Trong ví dụ này, type của biến `i`.) Rồi nó nhìn qua các case để tìm cái nào thuộc type đó. Và rồi thay đối số vào chỗ toàn bộ biểu thức `_Generic`.

Trong trường hợp này, `i` là `int`, nên nó khớp case đó. Rồi chuỗi được thay vào chỗ biểu thức. Nên dòng trở thành vầy khi compiler thấy:

```
    char *s = "that variable is an int";
```

Nếu compiler không tìm thấy type khớp trong `_Generic`, nó tìm case `default` tuỳ chọn và dùng nó.

Nếu không tìm được type khớp và không có `default`, bạn sẽ bị lỗi compile. Biểu thức đầu **phải** khớp một trong các type hoặc `default`.

Vì viết `_Generic` đi viết lại bất tiện, nó thường được dùng để làm thân của một macro có thể tái dùng dễ dàng.

Hãy làm một macro `TYPESTR(x)` nhận một đối số và trả về chuỗi với type của đối số.

Nên `TYPESTR(1)` sẽ trả về chuỗi `"int"`, chẳng hạn.

Nào:

```
#include <stdio.h>

#define TYPESTR(x) _Generic((x), \
                        int: "int", \
                        long: "long", \
                        float: "float", \
                        double: "double", \
                        default: "something else")

int main(void)
{
    int i;
    long l;
    float f;
    double d;
    char c;

    printf("i is type %s\n", TYPESTR(i));
    printf("l is type %s\n", TYPESTR(l));
    printf("f is type %s\n", TYPESTR(f));
    printf("d is type %s\n", TYPESTR(d));
    printf("c is type %s\n", TYPESTR(c));
```

```
}
```

Cái này xuất ra:

```
i is type int
l is type long
f is type float
d is type double
c is type something else
```

Không có gì bất ngờ, vì như ta đã nói, code trong `main()` được thay bằng cái sau khi compile:

```
    printf("i is type %s\n", "int");
    printf("l is type %s\n", "long");
    printf("f is type %s\n", "float");
    printf("d is type %s\n", "double");
    printf("c is type %s\n", "something else");
```

Và đó đúng là output ta thấy.

Làm thêm cái nữa. Tôi đã kèm vài macro ở đây để khi bạn chạy:

```
int i = 10;
char *s = "Foo!";

PRINT_VAL(i);
PRINT_VAL(s);
```

bạn được output:

```
i = 10
s = Foo!
```

Ta sẽ phải dùng chút phép thuật macro để làm được chuyện đó.

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

// Macro that gives back a format specifier for a type
#define FMTSPEC(x) _Generic((x), \
                        int: "%d", \
                        long: "%ld", \
                        float: "%f", \
                        double: "%f", \
                        char *: "%s")
                        // TODO: add more types

// Macro that prints a variable in the form "name = value"
#define PRINT_VAL(x) do { \
    char fmt[512]; \
    snprintf(fmt, sizeof fmt, #x " = %s\n", FMTSPEC(x)); \
    printf(fmt, (x)); \
} while(0)

int main(void)
{
```

```
    int i = 10;
    float f = 3.14159;
    char *s = "Hello, world!";

    PRINT_VAL(i);
    PRINT_VAL(f);
    PRINT_VAL(s);
}
```

cho output:

```
i = 10
f = 3.141590
s = Hello, world!
```

Ta có thể nhét hết vào một macro to, nhưng tôi chẻ ra hai để tránh chảy máu mắt.

Chapter 33

Mảng Phần II

Chương này ta sẽ đi qua vài thứ linh tinh thêm về mảng.

- Type qualifier với tham số mảng
- Từ khoá `static` với tham số mảng
- Initializer một phần cho mảng đa chiều

Chúng không phải cái hay thấy, nhưng ta sẽ liếc qua vì chúng là một phần của spec mới hơn.

33.1 Type qualifier cho mảng trong danh sách tham số

Nếu bạn nhớ từ trước, hai thứ này tương đương trong danh sách tham số hàm:

```
int func(int *p) {...}
int func(int p[]) {...}
```

Và bạn cũng có thể nhớ rằng bạn có thể thêm type qualifier vào biến con trỏ như vầy:

```
int *const p;
int *volatile p;
int *const volatile p;
// etc.
```

Nhưng làm sao làm được chuyện đó khi ta dùng ký pháp mảng trong danh sách tham số?

Hoá ra nó đi vào trong ngoặc vuông. Và bạn có thể đặt count tuỳ chọn ở sau. Hai dòng sau tương đương:

```
int func(int *const volatile p) {...}
int func(int p[const volatile]) {...}
int func(int p[const volatile 10]) {...}
```

Nếu bạn có mảng đa chiều, bạn cần đặt type qualifier ở bộ ngoặc vuông đầu.

33.2 `static` cho mảng trong danh sách tham số

Tương tự, bạn có thể dùng từ khoá static trong mảng trong danh sách tham số.

Đây là thứ tôi chưa từng thấy ngoài đời. Nó **luôn** theo sau bởi một kích thước:

```
int func(int p[static 4]) {...}
```

Điều này có nghĩa, trong ví dụ trên, compiler sẽ giả định mọi mảng bạn truyền cho hàm sẽ có *ít nhất* 4 phần tử.

Bất cứ gì khác là hành vi không xác định.

```
int func(int p[static 4]) {...}

int main(void)
{
    int a[] = {11, 22, 33, 44};
    int b[] = {11, 22, 33, 44, 55};
    int c[] = {11, 22};

    func(a);  // OK! a is 4 elements, the minimum
    func(b);  // OK! b is at least 4 elements
    func(c);  // Undefined behavior! c is under 4 elements!
}
```

Cái này về cơ bản đặt kích thước tối thiểu của mảng bạn có thể có.

Lưu ý quan trọng: không có gì trong compiler cấm bạn truyền mảng nhỏ hơn. Compiler khả năng sẽ không cảnh báo bạn, và nó cũng không phát hiện lúc runtime.

Khi đặt `static` vào đó, bạn đang nói, "Tôi hứa danh dự gấp đôi là tôi không bao giờ truyền vào mảng nhỏ hơn cái này." Và compiler nói, "Ừ, được rồi," và tin bạn sẽ không làm thế.

Và rồi compiler có thể thực hiện một số tối ưu code nhất định, yên tâm rằng bạn, lập trình viên, sẽ luôn làm đúng.

33.3 Các initializer tương đương

C hơi, nói thế nào nhỉ, *linh hoạt* khi dính tới initializer của mảng.

Ta đã thấy một chút rồi, khi giá trị thiếu được thay bằng zero.

Ví dụ, ta có thể khởi tạo mảng 5 phần tử thành `1,2,0,0,0` với:

```
int a[5] = {1, 2};
```

Hoặc set cả mảng về zero với:

```
int a[5] = {0};
```

Nhưng chuyện thú vị bắt đầu khi khởi tạo mảng đa chiều.

Làm một mảng 3 hàng, 2 cột:

```
int a[3][2];
```

Viết chút code để khởi tạo và in kết quả:

```
#include <stdio.h>

int main(void)
{
    int a[3][2] = {
        {1, 2},
        {3, 4},
        {5, 6}
    };

    for (int row = 0; row < 3; row++) {
```

```
        for (int col = 0; col < 2; col++)
            printf("%d ", a[row][col]);
        printf("\n");
    }
}
```

Và khi chạy, ta có kết quả như kỳ vọng:

```
1 2
3 4
5 6
```

Hãy bỏ bớt vài phần tử initializer và xem chúng được set về zero:

```
    int a[3][2] = {
        {1, 2},
        {3},    // Left off the 4!
        {5, 6}
    };
```

cho ra:

```
1 2
3 0
5 6
```

Giờ bỏ cả phần tử giữa cuối cùng:

```
    int a[3][2] = {
        {1, 2},
        // {3, 4},   // Just cut this whole thing out
        {5, 6}
    };
```

Và giờ ta có cái này, có thể không như bạn nghĩ:

```
1 2
5 6
0 0
```

Nhưng nếu bạn dừng lại suy nghĩ, ta chỉ cung cấp đủ initializer cho hai hàng, nên chúng được dùng cho hai hàng đầu. Và các phần tử còn lại được khởi tạo thành zero.

Đến đây ổn. Nhìn chung, nếu ta bỏ bớt phần của initializer, compiler set các phần tử tương ứng thành `0`.

Nhưng hãy làm *điên* hơn.

```
    int a[3][2] = { 1, 2, 3, 4, 5, 6 };
```

Cái gì—? Đó là mảng 2D, nhưng chỉ có initializer 1D!

Hoá ra chuyện đó hợp lệ (dù GCC sẽ cảnh báo nếu bật đúng warning).

Về cơ bản, nó bắt đầu điền phần tử ở hàng 0, rồi hàng 1, rồi hàng 2 từ trái sang phải.

Nên khi ta in, nó in theo thứ tự:

```
1 2
3 4
5 6
```

Nếu ta bỏ vài cái:

```
    int a[3][2] = { 1, 2, 3 };
```

chúng được điền `0`:

```
1 2
3 0
0 0
```

Nên nếu bạn muốn điền cả mảng bằng `0`, cứ:

```
    int a[3][2] = {0};
```

Nhưng khuyến nghị của tôi là nếu bạn có mảng 2D, dùng initializer 2D. Nó làm code dễ đọc hơn. (Trừ việc khởi tạo cả mảng bằng `0`, trường hợp đó dùng `{0}` là idiom bất kể chiều của mảng.)

Chapter 34

Long Jump với `setjmp`, `longjmp`

Ta đã thấy `goto`, nhảy trong scope hàm. Nhưng `longjmp()` cho phép bạn nhảy ngược về một điểm sớm hơn trong thực thi, về một hàm đã gọi hàm này.

Có cả đống hạn chế và cảnh báo, nhưng đây có thể là hàm hữu ích để thoát từ sâu trong call stack ngược lên trạng thái sớm hơn.

Theo kinh nghiệm của tôi, chức năng này rất hiếm khi được dùng.

34.1 Dùng `setjmp` và `longjmp`

Vũ điệu ta sẽ làm ở đây là về cơ bản đặt một bookmark trong thực thi với `setjmp()`. Sau đó, ta gọi `longjmp()` và nó sẽ nhảy về điểm sớm hơn trong thực thi nơi ta đặt bookmark bằng `setjmp()`.

Và nó có thể làm chuyện này ngay cả khi bạn đã gọi các hàm con.

Đây là demo nhanh trong đó ta gọi vào các hàm sâu vài cấp rồi thoát ra khỏi nó.

Ta sẽ dùng biến file scope `env` để giữ *state* khi gọi `setjmp()` để có thể khôi phục khi gọi `longjmp()` sau này. Đây là biến ta nhớ "vị trí" của mình.

Biến `env` thuộc kiểu `jmp_buf`, một kiểu mờ được khai báo trong `<setjmp.h>`.

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

jmp_buf env;

void depth2(void)
{
    printf("Entering depth 2\n");
    longjmp(env, 3490);           // Bail out
    printf("Leaving depth 2\n");  // This won't happen
}

void depth1(void)
{
    printf("Entering depth 1\n");
    depth2();
    printf("Leaving depth 1\n");  // This won't happen
}

int main(void)
```

```
{
    switch (setjmp(env)) {
      case 0:
          printf("Calling into functions, setjmp() returned 0\n");
          depth1();
          printf("Returned from functions\n");  // This won't happen
          break;

      case 3490:
          printf("Bailed back to main, setjmp() returned 3490\n");
          break;
    }
}
```

Khi chạy, cái này xuất ra:

```
Calling into functions, setjmp() returned 0
Entering depth 1
Entering depth 2
Bailed back to main, setjmp() returned 3490
```

Nếu bạn cố lấy output đó và khớp với code, rõ là có chuyện gì đó rất *quái* đang xảy ra.

Một trong những thứ đáng chú ý nhất là `setjmp()` return *hai lần*. Cái quái gì thế? Phép thuật gì đây?!

Vậy đây là deal: nếu `setjmp()` trả `0` , tức là bạn đã đặt "bookmark" thành công tại điểm đó.

Nếu nó trả khác 0, tức là bạn vừa trở về "bookmark" đã đặt trước đó. (Và giá trị trả về là giá trị bạn truyền cho `longjmp()` .)

Kiểu này bạn có thể phân biệt giữa việc đặt bookmark và trở về nó sau này.

Nên khi code trên gọi `setjmp()` lần đầu, `setjmp()` *lưu* state vào biến `env` và trả về `0` . Sau đó khi ta gọi `longjmp()` với cùng `env` đó, nó khôi phục state và `setjmp()` trả về giá trị đã truyền cho `longjmp()` .

34.2 Bẫy

Dưới mui, cái này khá thẳng thắn. Thông thường *stack pointer* theo dõi vị trí trong bộ nhớ nơi biến cục bộ được lưu, và *program counter* theo dõi địa chỉ của lệnh hiện đang thực thi[1].

Nên nếu ta muốn nhảy về hàm sớm hơn, về cơ bản chỉ là chuyện khôi phục stack pointer và program counter về giá trị giữ trong biến `jmp_buf` , và đảm bảo giá trị trả về được set đúng. Và rồi thực thi sẽ tiếp tục ở đó.

Nhưng đủ kiểu yếu tố làm rối cái này, tạo ra một số lượng đáng kể các bẫy hành vi không xác định.

34.2.1 Giá trị của biến cục bộ

Nếu bạn muốn giá trị của biến cục bộ automatic (không `static` và không `extern`) tồn tại trong hàm đã gọi `setjmp()` sau khi một `longjmp()` xảy ra, bạn phải khai báo các biến đó là `volatile` .

Về mặt kỹ thuật, chúng chỉ cần `volatile` nếu chúng thay đổi giữa lúc `setjmp()` được gọi và lúc `longjmp()` được gọi[2].

Ví dụ, nếu ta chạy code này:

[1] Cả "stack pointer" và "program counter" đều liên quan tới kiến trúc nằm dưới và cài đặt C, và không phải phần của spec.

[2] Lý lẽ ở đây là chương trình có thể lưu giá trị tạm thời trong *CPU register* khi nó đang làm việc với giá trị đó. Trong khoảng thời gian đó, register giữ giá trị đúng, và giá trị trên stack có thể đã cũ. Rồi sau đó giá trị register bị ghi đè và các thay đổi đối với biến bị mất.

```
int x = 20;

if (setjmp(env) == 0) {
    x = 30;
}
```

và sau đó `longjmp()` quay lại, giá trị của `x` sẽ không xác định.

Nếu ta muốn sửa chuyện này, `x` phải là `volatile`:

```
volatile int x = 20;

if (setjmp(env) == 0) {
    x = 30;
}
```

Giờ giá trị sẽ là đúng `30` sau khi một `longjmp()` đưa ta về điểm này.

34.2.2 Bao nhiêu state được lưu?

Khi bạn `longjmp()`, thực thi tiếp tục tại điểm của `setjmp()` tương ứng. Và thế thôi.

Spec chỉ rõ nó giống như bạn đã nhảy về hàm tại điểm đó với biến cục bộ được set về bất cứ giá trị nào chúng có tại lúc gọi `longjmp()`.

Những thứ không được khôi phục bao gồm, diễn giải lại spec:

- Cờ trạng thái dấu chấm động
- File đang mở
- Bất kỳ thành phần nào khác của máy trừu tượng

34.2.3 Bạn không thể đặt tên gì là `setjmp`

Bạn không thể có định danh `extern` nào với tên `setjmp`. Hoặc, nếu `setjmp` là macro, bạn không thể undefine nó.

Cả hai đều là hành vi không xác định.

34.2.4 Bạn không thể `setjmp()` trong biểu thức lớn hơn

Tức là, bạn không thể làm kiểu này:

```
if (x == 12 && setjmp(env) == 0) { ... }
```

Chuyện đó quá phức tạp để spec cho phép vì những cỗ máy cần chạy khi tháo stack và tất cả mấy chuyện đó. Ta không thể `longjmp()` về vào biểu thức phức tạp nào đó mà chỉ mới thực thi một phần.

Nên có giới hạn về độ phức tạp của biểu thức đó.

- Nó có thể là toàn bộ biểu thức điều khiển của điều kiện.

  ```
  if (setjmp(env)) {...}
  ```

  ```
  switch (setjmp(env)) {...}
  ```

- Nó có thể là một phần của biểu thức quan hệ hay đẳng thức, miễn là toán hạng kia là hằng số nguyên. Và toàn bộ là biểu thức điều khiển của điều kiện.

```
if (setjmp(env) == 0) {...}
```

- Toán hạng của phép NOT logic (`!`), là toàn bộ biểu thức điều khiển.

```
if (!setjmp(env)) {...}
```

- Biểu thức đứng một mình, có thể được cast thành `void`.

```
setjmp(env);
```

```
(void)setjmp(env);
```

34.2.5 Khi nào bạn không thể `longjmp()`?

Là hành vi không xác định nếu:

- Bạn không gọi `setjmp()` trước đó
- Bạn gọi `setjmp()` từ thread khác
- Bạn gọi `setjmp()` trong scope của một mảng độ dài biến đổi (VLA), và thực thi rời khỏi scope của VLA đó trước khi `longjmp()` được gọi.
- Hàm chứa `setjmp()` đã thoát trước khi `longjmp()` được gọi.

Ổ cái cuối, "thoát" bao gồm return bình thường khỏi hàm, cũng như trường hợp một `longjmp()` khác nhảy về "sớm hơn" trong call stack so với hàm đang nói tới.

34.2.6 Bạn không thể truyền `0` cho `longjmp()`

Nếu bạn thử truyền giá trị `0` cho `longjmp()`, nó sẽ âm thầm đổi giá trị đó thành `1`.

Vì `setjmp()` rốt cuộc trả giá trị này, và việc `setjmp()` trả `0` có nghĩa đặc biệt, nên trả `0` bị cấm.

34.2.7 `longjmp()` và mảng độ dài biến đổi

Nếu bạn đang trong scope của một VLA và `longjmp()` ra ngoài, bộ nhớ cấp cho VLA có thể bị leak[3].

Chuyện tương tự xảy ra nếu bạn `longjmp()` về qua bất kỳ hàm sớm hơn nào vẫn còn VLA trong scope.

Đây là một thứ thực sự làm tôi thấy phiền về VLA, rằng bạn có thể viết code C hoàn toàn hợp lệ mà phí bộ nhớ. Nhưng thôi, tôi không phải người quyết spec.

[3]Tức là, vẫn được cấp phát tới khi chương trình kết thúc mà không có cách giải phóng.

Chapter 35

Kiểu không hoàn chỉnh (Incomplete Types)

Bạn có thể ngạc nhiên khi biết đoạn này build không lỗi:

```
extern int a[];

int main(void)
{
    struct foo *x;
    union bar *y;
    enum baz *z;
}
```

Ta chưa hề cho kích thước của `a`. Và ta có con trỏ tới `struct foo`, `bar`, và `baz` mà dường như không được khai báo ở đâu cả.

Và cảnh báo duy nhất tôi nhận được là `x`, `y`, và `z` không được dùng.

Đây là các ví dụ về *kiểu không hoàn chỉnh* (incomplete type).

Kiểu không hoàn chỉnh là kiểu mà kích thước (tức là kích thước `sizeof` trả về) chưa biết. Cách nghĩ khác là kiểu bạn chưa khai báo xong.

Bạn có thể có con trỏ tới kiểu không hoàn chỉnh, nhưng bạn không thể dereference nó hay dùng số học con trỏ trên nó. Và bạn không thể `sizeof` nó.

Vậy làm gì được với nó?

35.1 Use case: cấu trúc tự tham chiếu

Tôi chỉ biết một use case thực sự: forward reference tới `struct` hay `union` với các cấu trúc tự tham chiếu hay đồng phụ thuộc. (Tôi sẽ dùng `struct` cho phần còn lại của các ví dụ này, nhưng tất cả đều áp dụng ngang bằng cho `union`.)

Làm ví dụ kinh điển trước.

Nhưng trước đó, biết điều này! Khi bạn khai báo một `struct`, `struct` đó không hoàn chỉnh cho tới khi dấu ngoặc nhọn đóng được tới!

```
struct antelope {             // struct antelope is incomplete here
    int leg_count;            // Still incomplete
    float stomach_fullness;   // Still incomplete
    float top_speed;          // Still incomplete
```

```
    char *nickname;            // Still incomplete
};                             // NOW it's complete.
```

Thì sao? Trông đủ hợp lý.

Nhưng nếu ta đang làm linked list thì sao? Mỗi node trong linked list cần có tham chiếu tới node khác. Nhưng làm sao tạo tham chiếu tới node khác nếu ta còn chưa khai báo xong cái node luôn?

Sự cho phép của C với kiểu không hoàn chỉnh làm điều đó thành khả thi. Ta không thể khai báo một node, nhưng ta *có thể* khai báo một con trỏ tới nó, kể cả khi nó chưa hoàn chỉnh!

```
struct node {
    int val;
    struct node *next;  // struct node is incomplete, but that's OK!
};
```

Dù `struct node` chưa hoàn chỉnh ở dòng 3, ta vẫn có thể khai báo một con trỏ tới nó[1].

Ta có thể làm tương tự nếu ta có hai `struct` khác nhau tham chiếu lẫn nhau:

```
struct a {
    struct b *x;  // Refers to a `struct b`
};

struct b {
    struct a *x;  // Refers to a `struct a`
};
```

Ta không thể nào tạo được cặp cấu trúc đó nếu không có quy tắc thả lỏng cho kiểu không hoàn chỉnh.

35.2 Thông báo lỗi về kiểu không hoàn chỉnh

Bạn có đang nhận các lỗi kiểu này không?

```
invalid application of ‘sizeof’ to incomplete type

invalid use of undefined type

dereferencing pointer to incomplete type
```

Thủ phạm có khả năng nhất: bạn có lẽ quên `#include` file header khai báo kiểu đó.

35.3 Các kiểu không hoàn chỉnh khác

Khai báo `struct` hay `union` không có thân tạo ra kiểu không hoàn chỉnh, ví dụ `struct foo;`.

`enum` không hoàn chỉnh cho tới dấu ngoặc nhọn đóng.

`void` là kiểu không hoàn chỉnh.

Mảng khai báo `extern` không có kích thước là không hoàn chỉnh, ví dụ:

```
extern int a[];
```

Nếu là mảng không-`extern` không có kích thước có initializer theo sau, nó không hoàn chỉnh cho tới dấu ngoặc nhọn đóng của initializer.

[1] Cái này chạy vì trong C, con trỏ có cùng kích thước bất kể kiểu dữ liệu chúng trỏ tới. Nên compiler không cần biết kích thước `struct node` tại điểm này; nó chỉ cần biết kích thước con trỏ.

35.4 Use case: mảng trong file header

Có thể hữu ích khi khai báo kiểu mảng không hoàn chỉnh trong file header. Trong trường hợp đó, phần lưu trữ thực (nơi mảng hoàn chỉnh được khai báo) nên ở trong một file `.c` duy nhất. Nếu bạn đặt nó trong file `.h`, nó sẽ bị nhân đôi mỗi lần file header được include.

Nên cái bạn có thể làm là tạo một file header với kiểu không hoàn chỉnh tham chiếu tới mảng, như vầy:

```
// File: bar.h

#ifndef BAR_H
#define BAR_H

extern int my_array[];  // Incomplete type

#endif
```

Và trong file `.c`, thực sự định nghĩa mảng:

```
// File: bar.c

int my_array[1024];     // Complete type!
```

Rồi bạn có thể include header từ bao nhiêu chỗ tuỳ ý, và mỗi chỗ sẽ tham chiếu tới cùng `my_array` nằm dưới.

```
// File: foo.c

#include <stdio.h>
#include "bar.h"    // includes the incomplete type for my_array

int main(void)
{
    my_array[0] = 10;

    printf("%d\n", my_array[0]);
}
```

Khi compile nhiều file, nhớ chỉ định mọi file `.c` cho compiler, nhưng không cần file `.h`, ví dụ:

```
gcc -o foo foo.c bar.c
```

35.5 Hoàn chỉnh kiểu không hoàn chỉnh

Nếu bạn có kiểu không hoàn chỉnh, bạn có thể hoàn chỉnh nó bằng cách định nghĩa `struct`, `union`, `enum`, hay mảng hoàn chỉnh trong cùng scope.

```
struct foo;        // incomplete type

struct foo *p;     // pointer, no problem

// struct foo f;   // Error: incomplete type!

struct foo {
    int x, y, z;
};                 // Now the struct foo is complete!
```

```
struct foo f;      // Success!
```

Lưu ý rằng dù `void` là kiểu không hoàn chỉnh, không có cách nào hoàn chỉnh nó. Không phải ai đó nghĩ tới chuyện làm cái quái đó. Nhưng nó cũng giải thích tại sao bạn có thể làm cái này:

```
void *p;            // OK: pointer to incomplete type
```

và không thể làm cả hai cái này:

```
void v;             // Error: declare variable of incomplete type

printf("%d\n", *p);  // Error: dereference incomplete type
```

Biết thêm càng tốt...

Chapter 36

Số phức

Một đoạn nhập môn tí hon về số phức[1] ăn cắp thẳng từ Wikipedia:

> **Số phức** là số có thể được biểu diễn dưới dạng $a + bi$, trong đó a và b là số thực [tức là kiểu dấu chấm động trong C], và i đại diện cho đơn vị ảo, thoả mãn phương trình $i^2 = -1$. Vì không có số thực nào thoả phương trình này, i được gọi là số ảo. Với số phức $a + bi$, a được gọi là **phần thực**, và b được gọi là **phần ảo**.

Nhưng tới đây là hết rồi. Ta giả định nếu bạn đọc chương này, bạn biết số phức là gì và bạn muốn làm gì với chúng.

Và tất cả ta cần lo là tiện ích của C để làm điều đó.

Hoá ra, hỗ trợ số phức trong compiler là tính năng *tuỳ chọn*. Không phải compiler tuân chuẩn nào cũng làm được. Và những compiler làm được, có thể làm ở các mức độ hoàn chỉnh khác nhau.

Bạn có thể check xem hệ của bạn có hỗ trợ số phức không với:

```
#ifdef __STDC_NO_COMPLEX__
#error Complex numbers not supported!
#endif
```

Thêm nữa, có một macro báo việc tuân theo chuẩn ISO 60559 (IEEE 754) cho toán dấu chấm động với số phức, cũng như sự hiện diện của kiểu `_Imaginary`.

```
#if __STDC_IEC_559_COMPLEX__ != 1
#error Need IEC 60559 complex support!
#endif
```

Chi tiết thêm về chuyện đó ghi ở Annex G trong spec C11.

36.1 Kiểu phức

Để dùng số phức, `#include <complex.h>`.

Với cái đó, bạn có ít nhất hai kiểu:

```
_Complex
complex
```

Cả hai đều có cùng nghĩa, nên cứ dùng `complex` cho đẹp.

Bạn cũng có vài kiểu cho số ảo nếu cài đặt của bạn tuân IEC 60559:

[1] https://en.wikipedia.org/wiki/Complex_number

```
_Imaginary
imaginary
```

Cả hai này cũng có cùng nghĩa, nên cứ dùng `imaginary` cho đẹp.

Bạn cũng có giá trị cho số ảo i, chính nó:

```
I
_Complex_I
_Imaginary_I
```

Macro `I` được set thành `_Imaginary_I` (nếu có), hoặc `_Complex_I`. Nên cứ dùng `I` cho số ảo.

Lề một chút: tôi đã nói nếu compiler set `__STDC_IEC_559_COMPLEX__` thành `1`, nó phải hỗ trợ kiểu `_Imaginary` để tuân chuẩn. Đó là cách tôi đọc spec. Tuy nhiên, tôi không biết một compiler nào thực sự hỗ trợ `_Imaginary` dù chúng có set `__STDC_IEC_559_COMPLEX__`. Nên tôi sẽ viết một số code với kiểu đó ở đây mà tôi không có cách nào test. Xin lỗi!

OK, giờ ta biết có kiểu `complex`, ta dùng nó sao?

36.2 Gán số phức

Vì số phức có phần thực và phần ảo, nhưng cả hai đều dựa vào số dấu chấm động để lưu giá trị, ta cũng cần báo C dùng độ chính xác nào cho các phần đó của số phức.

Ta làm chuyện đó bằng cách đính kèm `float`, `double`, hay `long double` vào `complex`, trước hay sau đều được.

Định nghĩa một số phức dùng `float` cho các thành phần của nó:

```
float complex c;   // Spec prefers this way
complex float c;   // Same thing--order doesn't matter
```

Vậy khai báo thì ổn rồi, còn khởi tạo hay gán thì sao?

Hoá ra ta được dùng ký pháp khá tự nhiên. Ví dụ!

```
double complex x = 5 + 2*I;
double complex y = 10 + 3*I;
```

Cho $5 + 2i$ và $10 + 3i$, tương ứng.

36.3 Dựng, xé, và in

Ta đang tới đó...

Ta đã thấy một cách viết số phức:

```
double complex x = 5 + 2*I;
```

Cũng không vấn đề gì khi dùng số dấu chấm động khác để dựng nó:

```
double a = 5;
double b = 2;
double complex x = a + b*I;
```

Cũng có một bộ macro để giúp dựng mấy cái này. Code trên có thể được viết dùng macro `CMPLX()`, như vầy:

```
double complex x = CMPLX(5, 2);
```

Theo như tôi tìm hiểu, mấy cái này *gần như* tương đương:

```
double complex x = 5 + 2*I;
double complex x = CMPLX(5, 2);
```

Nhưng macro `CMPLX()` sẽ xử lý zero âm ở phần ảo đúng mỗi lần, còn cách kia có thể chuyển chúng thành zero dương. Tôi *nghĩ*[2] Cái này có vẻ hàm ý rằng nếu có khả năng phần ảo là zero, bạn nên dùng macro... nhưng ai đó nên sửa tôi về điều này nếu tôi nhầm!

Macro `CMPLX()` chạy trên kiểu `double`. Có hai macro khác cho `float` và `long double`: `CMPLXF()` và `CMPLXL()`. (Hậu tố "f" và "l" xuất hiện hầu như trong tất cả các hàm liên quan tới số phức.)

Giờ thử ngược lại: nếu ta có số phức, làm sao tách nó ra phần thực và phần ảo?

Ta có một cặp hàm sẽ trích phần thực và phần ảo từ số: `creal()` và `cimag()`:

```
double complex x = 5 + 2*I;
double complex y = 10 + 3*I;

printf("x = %f + %fi\n", creal(x), cimag(x));
printf("y = %f + %fi\n", creal(y), cimag(y));
```

cho output:

```
x = 5.000000 + 2.000000i
y = 10.000000 + 3.000000i
```

Lưu ý rằng chữ `i` tôi có trong chuỗi format của `printf()` là chữ `i` theo nghĩa đen được in ra, nó không phải phần của format specifier. Cả hai giá trị trả về từ `creal()` và `cimag()` đều là `double`.

Và như thường, có các biến thể `float` và `long double` của các hàm này: `crealf()`, `cimagf()`, `creall()`, và `cimagl()`.

36.4 Số học và so sánh số phức

Số học có thể được thực hiện trên số phức, tuy cách nó chạy về mặt toán học nằm ngoài phạm vi guide này.

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

int main(void)
{
    double complex x = 1 + 2*I;
    double complex y = 3 + 4*I;
    double complex z;

    z = x + y;
    printf("x + y = %f + %fi\n", creal(z), cimag(z));
```

[2]Cái này là cái khó research hơn, và tôi sẽ nhận thêm thông tin ai đó cho tôi. `I` có thể được định nghĩa là `_Complex_I` hay `_Imaginary_I`, nếu cái sau tồn tại. `_Imaginary_I` sẽ xử lý zero có dấu, nhưng `_Complex_I` *có thể* không. Cái này có hàm ý với branch cut và các thứ toán-số-phức khác. Có lẽ. Bạn thấy tôi thực sự ra khỏi khu vực chuyên môn chưa? Dù sao, macro `CMPLX()` hành xử như thể `I` được định nghĩa là `_Imaginary_I`, với zero có dấu, kể cả khi `_Imaginary_I` không tồn tại trên hệ.

```
    z = x - y;
    printf("x - y = %f + %fi\n", creal(z), cimag(z));

    z = x * y;
    printf("x * y = %f + %fi\n", creal(z), cimag(z));

    z = x / y;
    printf("x / y = %f + %fi\n", creal(z), cimag(z));
}
```

cho kết quả:

```
x + y = 4.000000 + 6.000000i
x - y = -2.000000 + -2.000000i
x * y = -5.000000 + 10.000000i
x / y = 0.440000 + 0.080000i
```

Bạn cũng có thể so sánh hai số phức về bằng nhau (hoặc không bằng):

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

int main(void)
{
    double complex x = 1 + 2*I;
    double complex y = 3 + 4*I;

    printf("x == y = %d\n", x == y);  // 0
    printf("x != y = %d\n", x != y);  // 1
}
```

với output:

```
x == y = 0
x != y = 1
```

Chúng bằng nhau nếu cả hai thành phần test bằng. Lưu ý rằng như với mọi dấu chấm động, chúng có thể bằng nếu đủ gần do lỗi làm tròn[3].

36.5 Toán số phức

Nhưng khoan! Còn nhiều hơn chỉ là số học số phức đơn giản!

Đây là bảng tổng hợp mọi hàm toán có sẵn cho bạn với số phức.

Tôi sẽ chỉ liệt kê phiên bản `double` của mỗi hàm, nhưng với tất cả chúng đều có phiên bản `float` bạn lấy được bằng cách thêm `f` vào tên hàm, và phiên bản `long double` bạn lấy được bằng cách thêm `l`.

Ví dụ, hàm `cabs()` dùng tính giá trị tuyệt đối của số phức cũng có biến thể `cabsf()` và `cabsl()`. Tôi bỏ chúng cho ngắn.

36.5.1 Hàm lượng giác

[3]Sự đơn giản của câu này không làm tròn được lượng công sức khủng khiếp đổ vào việc chỉ đơn thuần hiểu dấu chấm động thực sự hoạt động thế nào. https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/

Hàm	Mô tả
`ccos()`	Cosine
`csin()`	Sine
`ctan()`	Tangent
`cacos()`	Arc cosine
`casin()`	Arc sine
`catan()`	Chơi *Settlers of Catan*
`ccosh()`	Hyperbolic cosine
`csinh()`	Hyperbolic sine
`ctanh()`	Hyperbolic tangent
`cacosh()`	Arc hyperbolic cosine
`casinh()`	Arc hyperbolic sine
`catanh()`	Arc hyperbolic tangent

36.5.2 Hàm mũ và logarit

Hàm	Mô tả
`cexp()`	Mũ cơ số e
`clog()`	Logarit tự nhiên (cơ số e)

36.5.3 Hàm luỹ thừa và giá trị tuyệt đối

Hàm	Mô tả
`cabs()`	Giá trị tuyệt đối
`cpow()`	Luỹ thừa
`csqrt()`	Căn bậc hai

36.5.4 Hàm thao tác

Hàm	Mô tả
`creal()`	Trả phần thực
`cimag()`	Trả phần ảo
`CMPLX()`	Dựng số phức
`carg()`	Argument / góc pha
`conj()`	Liên hợp[4]
`cproj()`	Phép chiếu lên mặt cầu Riemann

[4] Đây là cái duy nhất không bắt đầu bằng chữ `c` thêm đằng trước, lạ thay.

Chapter 37

Kiểu số nguyên bề rộng cố định

C có đủ các kiểu số nguyên nhỏ, lớn hơn, và lớn nhất kiểu `int`, `long` và thế nọ thế kia. Và bạn có thể xem trong phần giới hạn để thấy int lớn nhất là gì với `INT_MAX` và tương tự.

Các kiểu đó to bao nhiêu? Tức là, chúng chiếm mấy byte? Ta có thể dùng `sizeof` để ra câu trả lời.

Nhưng nếu tôi muốn đi ngược lại thì sao? Nếu tôi cần kiểu chính xác 32 bit (4 byte) hoặc ít nhất 16 bit hay đại loại thế?

Làm sao khai báo kiểu có kích thước nhất định?

Header `<stdint.h>` cho ta cách.

37.1 Các kiểu theo số bit

Với cả số nguyên có dấu và không dấu, ta có thể chỉ định kiểu có số bit nhất định, với vài cảnh báo, đương nhiên.

Và có ba nhóm chính các kiểu này (trong các ví dụ, `N` sẽ được thay bằng số bit cụ thể):

- Số nguyên chính xác kích thước nào đó (`intN_t`)
- Số nguyên ít nhất kích thước nào đó (`int_leastN_t`)
- Số nguyên ít nhất kích thước nào đó và nhanh hết mức có thể (`int_fastN_t`)[1]

`fast` nhanh hơn bao nhiêu? Chắc chắn có lẽ nhanh hơn một lượng nào đó. Có thể. Spec không nói nhanh hơn bao nhiêu, chỉ nói chúng sẽ là nhanh nhất trên kiến trúc này. Tuy nhiên, đa số compiler C khá tốt, nên bạn chắc sẽ chỉ thấy cái này được dùng ở chỗ cần đảm bảo tốc độ tối đa có thể (chứ không chỉ là hy vọng compiler xuất ra code đủ-nhanh-phết, mà nó đang làm vậy).

Cuối cùng, các kiểu số không dấu này có thêm chữ `u` ở đầu để phân biệt.

Ví dụ, các kiểu này có nghĩa tương ứng được liệt kê:

```
int32_t w;        // w is exactly 32 bits, signed
uint16_t x;       // x is exactly 16 bits, unsigned

int_least8_t y;   // y is at least 8 bits, signed

uint_fast64_t z;  // z is the fastest representation at least 64 bits, unsigned
```

Các kiểu sau được đảm bảo được định nghĩa:

[1] Một số kiến trúc có dữ liệu kích thước khác mà CPU và RAM có thể thao tác với tốc độ nhanh hơn các kiểu khác. Trong các trường hợp đó, nếu bạn cần số 8-bit nhanh nhất, có thể nó cho bạn kiểu 16- hay 32-bit thay thế vì cái đó chỉ đơn giản là nhanh hơn. Nên với cái này, bạn sẽ không biết kiểu to bao nhiêu, nhưng nó sẽ ít nhất to như bạn nói.

```
int_least8_t      uint_least8_t
int_least16_t     uint_least16_t
int_least32_t     uint_least32_t
int_least64_t     uint_least64_t

int_fast8_t       uint_fast8_t
int_fast16_t      uint_fast16_t
int_fast32_t      uint_fast32_t
int_fast64_t      uint_fast64_t
```

Có thể có các kiểu khác với bề rộng khác, nhưng chúng là tuỳ chọn.

Ê! Các kiểu cố định kiểu `int16_t` đâu? Hoá ra chúng hoàn toàn tuỳ chọn... trừ khi có điều kiện nhất định được thoả[2]. Và nếu bạn có hệ máy tính hiện đại trung bình bình thường, các điều kiện đó khả năng cao được thoả. Và nếu thoả, bạn sẽ có các kiểu này:

```
int8_t      uint8_t
int16_t     uint16_t
int32_t     uint32_t
int64_t     uint64_t
```

Các biến thể khác với bề rộng khác có thể được định nghĩa, nhưng chúng tuỳ chọn.

37.2 Kiểu số nguyên kích thước tối đa

Có kiểu bạn có thể dùng giữ số nguyên biểu diễn được lớn nhất có sẵn trên hệ, cả có dấu và không dấu:

```
intmax_t
uintmax_t
```

Dùng các kiểu này khi bạn muốn đi to hết mức.

Rõ ràng là giá trị từ bất kỳ kiểu số nguyên nào khác cùng dấu sẽ vừa vào kiểu này, đương nhiên.

37.3 Dùng hằng số kích thước cố định

Nếu bạn có một hằng mà bạn muốn nó vừa vào số bit nhất định, bạn có thể dùng các macro này để tự động thêm hậu tố đúng vào số (ví dụ `22L` hay `3490ULL`).

```
INT8_C(x)     UINT8_C(x)
INT16_C(x)    UINT16_C(x)
INT32_C(x)    UINT32_C(x)
INT64_C(x)    UINT64_C(x)
INTMAX_C(x)   UINTMAX_C(x)
```

Lại nữa, mấy cái này chỉ chạy với giá trị số nguyên hằng.

Ví dụ, ta có thể dùng một trong số đó để gán giá trị hằng như vầy:

```
uint16_t x = UINT16_C(12);
intmax_t y = INTMAX_C(3490);
```

[2]Cụ thể, hệ có số nguyên 8, 16, 32, hay 64 bit không padding dùng biểu diễn bù 2, trong trường hợp đó biến thể `intN_t` cho số bit cụ thể đó *phải* được định nghĩa.

37.4 Giới hạn của số nguyên kích thước cố định

Ta cũng có một số giới hạn được định nghĩa để bạn lấy được giá trị lớn nhất và nhỏ nhất cho các kiểu này:

```
INT8_MAX           INT8_MIN           UINT8_MAX
INT16_MAX          INT16_MIN          UINT16_MAX
INT32_MAX          INT32_MIN          UINT32_MAX
INT64_MAX          INT64_MIN          UINT64_MAX

INT_LEAST8_MAX     INT_LEAST8_MIN     UINT_LEAST8_MAX
INT_LEAST16_MAX    INT_LEAST16_MIN    UINT_LEAST16_MAX
INT_LEAST32_MAX    INT_LEAST32_MIN    UINT_LEAST32_MAX
INT_LEAST64_MAX    INT_LEAST64_MIN    UINT_LEAST64_MAX

INT_FAST8_MAX      INT_FAST8_MIN      UINT_FAST8_MAX
INT_FAST16_MAX     INT_FAST16_MIN     UINT_FAST16_MAX
INT_FAST32_MAX     INT_FAST32_MIN     UINT_FAST32_MAX
INT_FAST64_MAX     INT_FAST64_MIN     UINT_FAST64_MAX

INTMAX_MAX         INTMAX_MIN         UINTMAX_MAX
```

Lưu ý `MIN` cho mọi kiểu không dấu là `0`, nên như vậy, không có macro cho nó.

37.5 Format specifier

Để in các kiểu này, bạn cần truyền đúng format specifier cho `printf()`. (Và vấn đề tương tự khi lấy input với `scanf()`.)

Nhưng làm sao bạn biết được kiểu to bao nhiêu dưới mui? May thay, lần nữa, C cung cấp vài macro để giúp chuyện này.

Tất cả chuyện này có thể thấy trong `<inttypes.h>`.

Giờ, ta có một đống macro. Kiểu một vụ nổ phức tạp của macro. Nên tôi sẽ thôi liệt ra từng cái và chỉ đặt chữ thường `n` ở chỗ mà bạn nên đặt `8`, `16`, `32`, hay `64` tuỳ nhu cầu.

Nhìn qua các macro cho in số nguyên có dấu:

```
PRIdn    PRIdLEASTn    PRIdFASTn    PRIdMAX
PRIin    PRIiLEASTn    PRIiFASTn    PRIiMAX
```

Nhìn pattern ở đó. Bạn có thể thấy có biến thể cho kiểu fixed, least, fast, và max.

Và bạn cũng có chữ thường `d` và chữ thường `i`. Các chữ đó tương ứng với format specifier `%d` và `%i` của `printf()`.

Nên nếu tôi có thứ kiểu:

```
int_least16_t x = 3490;
```

Tôi có thể in cái đó với format specifier tương đương với `%d` bằng `PRIdLEAST16`.

Nhưng sao? Ta dùng macro đó sao?

Trước hết, macro đó chỉ định một chuỗi chứa chữ cái hay các chữ cái `printf()` cần dùng để in kiểu đó. Ví dụ, nó có thể là `"d"` hay `"ld"`.

Nên tất cả ta cần làm là nhúng nó vào chuỗi format của lời gọi `printf()`.

Để làm chuyện này, ta có thể tận dụng một chuyện về C bạn có thể đã quên: chuỗi literal kề nhau được nối tự động thành một chuỗi. Ví dụ:

```
printf("Hello, " "world!\n");   // Prints "Hello, world!"
```

Và vì các macro này là chuỗi literal, ta có thể dùng chúng như vầy:

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

int main(void)
{
    int_least16_t x = 3490;

    printf("The value is %" PRIdLEAST16 "!\n", x);
}
```

Ta cũng có một đống macro để in kiểu không dấu:

```
PRIon    PRIoLEASTn    PRIoFASTn    PRIoMAX
PRIun    PRIuLEASTn    PRIuFASTn    PRIuMAX
PRIxn    PRIxLEASTn    PRIxFASTn    PRIxMAX
PRIXn    PRIXLEASTn    PRIXFASTn    PRIXMAX
```

Trong trường hợp này, `o`, `u`, `x`, và `X` tương ứng với các format specifier đã documented trong `printf()`.

Và, như trước, chữ thường `n` nên được thay bằng `8`, `16`, `32`, hay `64`.

Nhưng ngay khi bạn tưởng mình đã chán macro, hoá ra ta có một bộ hoàn chỉnh đối ứng cho `scanf()`!

```
SCNdn    SCNdLEASTn    SCNdFASTn    SCNdMAX
SCNin    SCNiLEASTn    SCNiFASTn    SCNiMAX
SCNon    SCNoLEASTn    SCNoFASTn    SCNoMAX
SCNun    SCNuLEASTn    SCNuFASTn    SCNuMAX
SCNxn    SCNxLEASTn    SCNxFASTn    SCNxMAX
```

Nhớ: khi bạn muốn in một kiểu số nguyên kích thước cố định bằng `printf()` hay `scanf()`, lấy format specifier tương ứng đúng từ `<inttypes.h>`.

Chapter 38

Ngày giờ

> "Time is an illusion. Lunchtime doubly so."
> —Ford Prefect, The Hitchhikers Guide to the Galaxy

Cái này không quá phức tạp, nhưng ban đầu có thể hơi nản, cả với các kiểu khác nhau có sẵn và cách ta chuyển qua lại giữa chúng.

Trộn thêm GMT (UTC) và local time và ta có mọi *Niềm Vui Thường Lệ*™ mà người ta có với ngày giờ.

Và đương nhiên đừng bao giờ quên quy tắc vàng của ngày giờ: *Đừng bao giờ cố viết chức năng ngày giờ của riêng bạn. Chỉ dùng cái thư viện cho.*

Thời gian quá phức tạp đối với các lập trình viên phàm phu. Nghiêm túc, ta nợ một điểm mỗi người đã làm việc trên bất kỳ thư viện ngày giờ nào, nên bỏ cái đó vào ngân sách.

38.1 Thuật ngữ và thông tin nhanh

Vài thuật ngữ nhanh phòng khi bạn chưa nắm rõ.

- **UTC**: Coordinated Universal Time là thời gian tuyệt đối được đồng thuận toàn cầu[1]. Mọi người trên hành tinh nghĩ bây giờ là cùng một thời điểm theo UTC... dù họ có giờ địa phương khác nhau.
- **GMT**: Greenwich Mean Time, về cơ bản giống UTC[2]. Bạn có lẽ muốn nói UTC, hay "giờ toàn cầu". Nếu bạn nói cụ thể về múi giờ GMT, nói GMT. Gây lẫn lộn là, nhiều hàm UTC của C có trước UTC và vẫn dùng tên Greenwich Mean Time. Khi bạn thấy thế, biết rằng C ý là UTC.
- **Local time**: giờ ở nơi máy tính đang chạy chương trình. Cái này được mô tả như một độ lệch so với UTC. Dù có nhiều múi giờ trên thế giới, đa số máy tính làm việc ở local time hoặc UTC.

Theo quy tắc chung, nếu bạn đang mô tả sự kiện xảy ra một lần, như một entry log, hay một vụ phóng tên lửa, hay khi con trỏ cuối cùng cũng click trong đầu bạn, dùng UTC.

Mặt khác, nếu là chuyện gì đó xảy ra cùng giờ *ở mọi múi giờ*, như đêm giao thừa hay giờ ăn tối, dùng local time.

Vì nhiều ngôn ngữ chỉ giỏi chuyển qua lại UTC và local time, bạn có thể tự gây đau đầu rất nhiều nếu chọn lưu ngày theo dạng sai. (Hỏi tôi sao tôi biết.)

38.2 Kiểu ngày

Có hai[3] kiểu chính trong C khi dính tới ngày: `time_t` và `struct tm`.

Spec thực ra không nói nhiều về chúng:

[1] Trên Trái Đất, dù sao. Ai biết họ dùng hệ điên rồ gì *ngoài kia*...

[2] OK, đừng giết tôi! GMT về kỹ thuật là múi giờ còn UTC là hệ thời gian toàn cầu. Ngoài ra vài nước có thể chỉnh GMT cho tiết kiệm ánh sáng ban ngày, trong khi UTC không bao giờ được chỉnh cho tiết kiệm ánh sáng ban ngày.

[3] Thực ra là nhiều hơn hai.

- `time_t`: kiểu thực có khả năng giữ một thời gian. Nên theo spec, cái này có thể là kiểu dấu chấm động hay kiểu số nguyên. Trong POSIX (các hệ Unix-like), nó là số nguyên. Cái này giữ *calendar time*. Bạn có thể nghĩ như giờ UTC.
- `struct tm`: giữ các thành phần của một calendar time. Đây là một *broken-down time*, tức là, các thành phần của thời gian, như giờ, phút, giây, ngày, tháng, năm, v.v.

Trên nhiều hệ, `time_t` đại diện cho số giây kể từ *Epoch*[4]. Epoch theo một nghĩa là khởi đầu thời gian theo góc nhìn của máy tính, thường là 1 tháng 1, 1970 UTC. `time_t` có thể âm để đại diện cho thời gian trước Epoch. Windows hoạt động tương tự Unix theo tôi thấy.

Và trong `struct tm` có gì? Các field sau:

```
struct tm {
    int tm_sec;    // seconds after the minute -- [0, 60]
    int tm_min;    // minutes after the hour -- [0, 59]
    int tm_hour;   // hours since midnight -- [0, 23]
    int tm_mday;   // day of the month -- [1, 31]
    int tm_mon;    // months since January -- [0, 11]
    int tm_year;   // years since 1900
    int tm_wday;   // days since Sunday -- [0, 6]
    int tm_yday;   // days since January 1 -- [0, 365]
    int tm_isdst;  // Daylight Saving Time flag
};
```

Lưu ý mọi thứ đều bắt đầu từ zero trừ ngày trong tháng.

Quan trọng là biết rằng bạn có thể đặt bất cứ giá trị nào vào các kiểu này bạn muốn. Có các hàm giúp lấy thời gian *hiện tại*, nhưng kiểu giữ *một* thời gian, không phải *thời gian*.

Nên câu hỏi trở thành: "Làm sao khởi tạo dữ liệu các kiểu này, và làm sao chuyển giữa chúng?"

38.3 Khởi tạo và chuyển giữa các kiểu

Trước hết, bạn có thể lấy thời gian hiện tại và lưu nó vào `time_t` với hàm `time()`.

```
time_t now;  // Variable to hold the time now

now = time(NULL);  // You can get it like this...

time(&now);        // ...or this. Same as the previous line.
```

Tuyệt! Bạn có biến lấy được thời gian hiện tại.

Vui là, chỉ có một cách portable để in ra thứ có trong `time_t`, và đó là hàm `ctime()` hiếm dùng, in giá trị theo local time:

```
now = time(NULL);
printf("%s", ctime(&now));
```

Cái này trả chuỗi có dạng rất cụ thể bao gồm newline ở cuối:

```
Sun Feb 28 18:47:25 2021
```

Nên cái đó hơi cứng nhắc. Nếu bạn muốn kiểm soát hơn, bạn nên chuyển `time_t` đó thành `struct tm`.

[4] https://en.wikipedia.org/wiki/Unix_time

38.3.1 Chuyển `time_t` sang `struct tm`

Có hai cách kỳ diệu để làm chuyển này:

- `localtime()` : hàm này chuyển `time_t` sang `struct tm` theo local time.
- `gmtime()` : hàm này chuyển `time_t` sang `struct tm` theo UTC. (Thấy GMT xưa chui vào tên hàm đó chứ?)

Xem giờ hiện tại bằng cách in ra `struct tm` bằng hàm `asctime()` :

```
printf("Local: %s", asctime(localtime(&now)));
printf("  UTC: %s", asctime(gmtime(&now)));
```

Output (tôi ở múi giờ Pacific Standard):

```
Local: Sun Feb 28 20:15:27 2021
  UTC: Mon Mar  1 04:15:27 2021
```

Một khi bạn có `time_t` trong `struct tm` , nó mở ra đủ loại cánh cửa. Bạn có thể in thời gian theo đủ kiểu, tìm xem một ngày là thứ mấy trong tuần, v.v. Hoặc chuyển nó ngược lại thành `time_t` .

Sẽ nói thêm về cái đó sớm!

38.3.2 Chuyển `struct tm` sang `time_t`

Nếu bạn muốn đi theo chiều ngược, bạn có thể dùng `mktime()` để lấy thông tin đó.

`mktime()` set giá trị của `tm_wday` và `tm_yday` giùm bạn, nên đừng phí công điền chúng vì chúng sẽ bị ghi đè thôi.

Ngoài ra, bạn có thể set `tm_isdst` thành `-1` để nó tự quyết định giùm bạn. Hoặc bạn có thể set thủ công thành true hay false.

```
// Don't be tempted to put leading zeros on these numbers (unless you
// mean for them to be in octal)!

struct tm some_time = {
    .tm_year=82,   // years since 1900
    .tm_mon=3,     // months since January -- [0, 11]
    .tm_mday=12,   // day of the month -- [1, 31]
    .tm_hour=12,   // hours since midnight -- [0, 23]
    .tm_min=0,     // minutes after the hour -- [0, 59]
    .tm_sec=4,     // seconds after the minute -- [0, 60]
    .tm_isdst=-1,  // Daylight Saving Time flag
};

time_t some_time_epoch;

some_time_epoch = mktime(&some_time);

printf("%s", ctime(&some_time_epoch));
printf("Is DST: %d\n", some_time.tm_isdst);
```

Output:

```
Mon Apr 12 12:00:04 1982
Is DST: 0
```

Khi bạn nạp thủ công một `struct tm` như vậy, nó nên là local time. `mktime()` sẽ chuyển local time đó thành `time_t` calendar time.

Lạ là, tuy vậy, chuẩn không cho ta cách nạp một `struct tm` với thời gian UTC và chuyển nó thành `time_t`. Nếu bạn muốn làm vậy với các hệ Unix-like, thử hàm không chuẩn `timegm()`. Trên Windows, `_mkgmtime()`.

38.4 In ngày theo định dạng

Ta đã thấy vài cách in output ngày có định dạng lên màn hình. Với `time_t` ta dùng `ctime()`, và với `struct tm` ta dùng `asctime()`.

```
time_t now = time(NULL);
struct tm *local = localtime(&now);
struct tm *utc = gmtime(&now);

printf("Local time: %s", ctime(&now));     // Local time with time_t
printf("Local time: %s", asctime(local));  // Local time with struct tm
printf("UTC       : %s", asctime(utc));    // UTC with a struct tm
```

Nhưng nếu tôi nói với bạn, độc giả thân mến, rằng có cách kiểm soát nhiều hơn cách ngày được in ra thì sao?

Chắc chắn, ta có thể câu từng field từ `struct tm`, nhưng có một hàm tuyệt gọi là `strftime()` sẽ làm nhiều phần khó cho bạn. Nó giống `printf()` chỉ khác là cho ngày!

Xem vài ví dụ. Trong mỗi cái, ta truyền vào buffer đích, số ký tự tối đa để ghi, và rồi chuỗi format (theo phong cách của, nhưng không giống, `printf()`) bảo `strftime()` thành phần nào của `struct tm` cần in và in sao.

Bạn có thể thêm ký tự hằng khác để đưa vào output trong chuỗi format, cũng như với `printf()`.

Ta lấy `struct tm` trong trường hợp này từ `localtime()`, nhưng bất kỳ nguồn nào cũng ổn.

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

int main(void)
{
    char s[128];
    time_t now = time(NULL);

    // %c: print date as per current locale
    strftime(s, sizeof s, "%c", localtime(&now));
    puts(s);   // Sun Feb 28 22:29:00 2021

    // %A: full weekday name
    // %B: full month name
    // %d: day of the month
    strftime(s, sizeof s, "%A, %B %d", localtime(&now));
    puts(s);   // Sunday, February 28

    // %I: hour (12 hour clock)
    // %M: minute
    // %S: second
    // %p: AM or PM
    strftime(s, sizeof s, "It's %I:%M:%S %p", localtime(&now));
```

```
    puts(s);   // It's 10:29:00 PM

    // %F: ISO 8601 yyyy-mm-dd
    // %T: ISO 8601 hh:mm:ss
    // %z: ISO 8601 time zone offset
    strftime(s, sizeof s, "ISO 8601: %FT%T%z", localtime(&now));
    puts(s);   // ISO 8601: 2021-02-28T22:29:00-0800
}
```

Có *cả tấn* format specifier in ngày cho `strftime()`, nên nhớ xem chúng trong trang tham khảo `strftime()`[5].

38.5 Độ phân giải cao hơn với `timespec_get()`

Bạn có thể lấy số giây và nanosecond kể từ Epoch với `timespec_get()`.

Có thể.

Các cài đặt có thể không có độ phân giải nanosecond (là một phần tỷ giây) nên ai biết bạn sẽ có bao nhiêu chữ số có nghĩa, nhưng cứ thử xem.

`timespec_get()` nhận hai đối số. Một là con trỏ tới `struct timespec` để giữ thông tin thời gian. Và cái kia là `base`, mà spec cho phép bạn set thành `TIME_UTC` báo rằng bạn quan tâm tới số giây kể từ Epoch. (Các cài đặt khác có thể cho bạn thêm lựa chọn cho `base`.)

Và bản thân cấu trúc có hai field:

```
struct timespec {
    time_t tv_sec;   // Seconds
    long   tv_nsec;  // Nanoseconds (billionths of a second)
};
```

Đây là ví dụ ta lấy thời gian và in ra cả giá trị số nguyên lẫn giá trị dấu chấm động:

```
struct timespec ts;

timespec_get(&ts, TIME_UTC);

printf("%ld s, %ld ns\n", ts.tv_sec, ts.tv_nsec);

double float_time = ts.tv_sec + ts.tv_nsec/1000000000.0;
printf("%f seconds since epoch\n", float_time);
```

Ví dụ output:

```
1614581530 s, 806325800 ns
1614581530.806326 seconds since epoch
```

`struct timespec` cũng xuất hiện ở một số hàm threading cần có khả năng chỉ định thời gian với độ phân giải đó.

[5] https://beej.us/guide/bgclr/html/split/time.html#man-strftime

38.6 Khác biệt giữa các thời gian

Một lưu ý nhanh về lấy khác biệt giữa hai `time_t`: vì spec không quy định kiểu đó biểu diễn thời gian sao, bạn có thể không thể chỉ đơn giản trừ hai `time_t` và ra gì có nghĩa[6].

May thay bạn có thể dùng `difftime()` để tính khác biệt tính bằng giây giữa hai ngày.

Trong ví dụ sau, ta có hai sự kiện xảy ra cách nhau một khoảng thời gian, và ta dùng `difftime()` để tính khác biệt.

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

int main(void)
{
    struct tm time_a = {
        .tm_year=82,   // years since 1900
        .tm_mon=3,     // months since January -- [0, 11]
        .tm_mday=12,   // day of the month -- [1, 31]
        .tm_hour=4,    // hours since midnight -- [0, 23]
        .tm_min=00,    // minutes after the hour -- [0, 59]
        .tm_sec=04,    // seconds after the minute -- [0, 60]
        .tm_isdst=-1,  // Daylight Saving Time flag
    };

    struct tm time_b = {
        .tm_year=120,  // years since 1900
        .tm_mon=10,    // months since January -- [0, 11]
        .tm_mday=15,   // day of the month -- [1, 31]
        .tm_hour=16,   // hours since midnight -- [0, 23]
        .tm_min=27,    // minutes after the hour -- [0, 59]
        .tm_sec=00,    // seconds after the minute -- [0, 60]
        .tm_isdst=-1,  // Daylight Saving Time flag
    };

    time_t cal_a = mktime(&time_a);
    time_t cal_b = mktime(&time_b);

    double diff = difftime(cal_b, cal_a);

    double years = diff / 60 / 60 / 24 / 365.2425;  // close enough

    printf("%f seconds (%f years) between events\n", diff, years);
}
```

Output:

```
1217996816.000000 seconds (38.596783 years) between events
```

Và bạn có rồi đó! Nhớ dùng `difftime()` để lấy khác biệt thời gian. Dù bạn có thể chỉ trừ trên hệ POSIX, cứ giữ portable thôi.

[6]Bạn sẽ làm được trên POSIX, nơi `time_t` chắc chắn là số nguyên. Không may cả thế giới không phải POSIX, nên vậy đó.

Chapter 39

Đa luồng (Multithreading)

C11 chính thức đưa đa luồng vào ngôn ngữ C. Nó giống kỳ lạ với POSIX threads[1] nếu bạn đã từng dùng thứ đó.

Còn nếu chưa, đừng lo. Ta sẽ đi qua từng bước.

Lưu ý là tôi không định làm hướng dẫn đa luồng đầy đủ kiểu cổ điển[2]; bạn phải tìm một cuốn sách thật dày khác dành riêng cho chuyện đó. Xin lỗi nhé!

Threads là tính năng tuỳ chọn. Nếu compiler C11+ định nghĩa `__STDC_NO_THREADS__`, threads sẽ **không** có trong thư viện. Tại sao họ quyết định đi theo nghĩa phủ định trong cái macro đó thì tôi chịu, nhưng nó là vậy.

Bạn có thể kiểm tra nó như vầy:

```
#ifdef __STDC_NO_THREADS__
#error I need threads to build this program!
#endif
```

Ngoài ra, bạn có thể cần chỉ định tuỳ chọn linker khi build. Trong trường hợp hệ Unix-like, thử thêm `-lpthreads` vào cuối dòng lệnh để link thư viện `pthreads`[3]:

```
gcc -std=c11 -o foo foo.c -lpthreads
```

Nếu bạn gặp lỗi linker trên hệ thống, có thể là do thư viện phù hợp không được include.

39.1 Bối cảnh

Threads là cách để bạn dùng tất cả các CPU core bóng loáng mà bạn đã trả tiền cùng làm việc cho bạn trong cùng một chương trình.

Bình thường một chương trình C chỉ chạy trên một CPU core. Nhưng nếu bạn biết chia công việc ra, bạn có thể đưa từng phần cho một số threads và để chúng làm song song.

Dù spec không nói, rất có khả năng trên hệ thống của bạn, C (hay OS thay mặt nó) sẽ cố cân bằng threads trên tất cả các CPU core.

Và nếu bạn có nhiều threads hơn cores, không sao. Chỉ là bạn sẽ không nhận được tất cả lợi ích đó nếu chúng đều cạnh tranh thời gian CPU.

[1] https://en.wikipedia.org/wiki/POSIX_Threads

[2] Bản thân tôi thích kiểu shared-nothing hơn, và kỹ năng của tôi với mấy construct đa luồng cổ điển nói nhẹ là đã cũ.

[3] Đúng, `pthreads` với "`p`". Viết tắt cho POSIX threads, thư viện mà C11 đã vay mượn rất nhiều cho cách hiện thực threads của nó.

39.2 Những thứ bạn làm được

Bạn có thể tạo một thread. Nó sẽ bắt đầu chạy hàm bạn chỉ định. Thread cha tạo ra nó cũng sẽ tiếp tục chạy.

Và bạn có thể đợi thread kết thúc. Cái này gọi là *joining*.

Hoặc nếu bạn không quan tâm khi nào thread kết thúc và không muốn đợi, bạn có thể *detach* nó.

Một thread có thể *exit* tường minh, hoặc có thể ngầm kết thúc bằng cách return từ hàm chính của nó.

Một thread cũng có thể *sleep* một khoảng thời gian, không làm gì trong khi thread khác chạy.

Chương trình `main()` cũng là một thread.

Ngoài ra, ta có thread local storage, mutexes, và condition variables. Nhưng mấy cái đó để sau. Giờ cứ xem phần cơ bản đã.

39.3 Data Race và thư viện chuẩn

Một số hàm trong thư viện chuẩn (ví dụ `asctime()` và `strtok()`) trả về hoặc dùng các phần tử dữ liệu `static` không threadsafe. Nhưng nhìn chung trừ khi có nói khác, thư viện chuẩn cố gắng làm nó threadsafe[4].

Nhưng để ý nhé. Nếu một hàm thư viện chuẩn giữ trạng thái giữa các lần gọi trong một biến bạn không sở hữu, hay một hàm trả về con trỏ tới thứ gì đó mà bạn không truyền vào, thì nó không threadsafe.

39.4 Tạo và đợi Threads

Hack gì đó lên thôi!

Ta sẽ tạo vài threads và đợi chúng hoàn thành (join).

Có vài thứ cần hiểu trước đã.

Mỗi thread được xác định bằng một biến mờ (opaque) kiểu `thrd_t`. Đây là ID duy nhất của từng thread trong chương trình. Khi bạn tạo thread, nó được cấp ID mới.

Cũng vậy khi bạn tạo thread, bạn phải đưa cho nó con trỏ đến một hàm để chạy, và một con trỏ đến tham số để truyền cho nó (hoặc `NULL` nếu không có gì để truyền).

Thread sẽ bắt đầu thực thi ở hàm bạn chỉ định.

Khi bạn muốn đợi một thread hoàn thành, bạn phải chỉ định thread ID của nó để C biết đợi cái nào.

Nên ý tưởng cơ bản là:

1. Viết một hàm đóng vai "`main`" của thread. Không phải `main()` chính gốc, nhưng tương tự. Thread sẽ bắt đầu chạy ở đó.
2. Từ thread chính, launch thread mới với `thrd_create()`, và truyền cho nó con trỏ đến hàm cần chạy.
3. Trong hàm đó, cho thread làm bất cứ gì nó phải làm.
4. Cùng lúc đó, thread chính có thể tiếp tục làm bất cứ gì *nó* cần làm.
5. Khi thread chính quyết định, nó có thể đợi thread con hoàn thành bằng cách gọi `thrd_join()`. Thường thì bạn **phải** `thrd_join()` thread để dọn dẹp nó nếu không bạn sẽ leak bộ nhớ[5]

`thrd_create()` nhận con trỏ đến hàm cần chạy, và nó có kiểu `thrd_start_t`, tức là `int (*)(void *)`. Đó là tiếng Hy Lạp cho "con trỏ đến một hàm nhận `void*` làm tham số và trả về `int`."

Hãy tạo một thread! Ta sẽ launch nó từ thread chính bằng `thrd_create()` để chạy một hàm, làm vài thứ khác, rồi đợi nó hoàn thành với `thrd_join()`. Tôi đặt tên hàm chính của thread là `run()`, nhưng bạn có thể đặt tên gì cũng được miễn kiểu khớp với `thrd_start_t`.

[4] Theo §7.1.4¶5.

[5] Trừ khi bạn `thrd_detach()`. Sẽ nói thêm sau.

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

// This is the function the thread will run. It can be called anything.
//
// arg is the argument pointer passed to `thrd_create()`.
//
// The parent thread will get the return value back from `thrd_join()`'
// later.

int run(void *arg)
{
    int *a = arg;  // We'll pass in an int* from thrd_create()

    printf("THREAD: Running thread with arg %d\n", *a);

    return 12;  // Value to be picked up by thrd_join() (chose 12 at random)
}

int main(void)
{
    thrd_t t;  // t will hold the thread ID
    int arg = 3490;

    printf("Launching a thread\n");

    // Launch a thread to the run() function, passing a pointer to 3490
    // as an argument. Also stored the thread ID in t:

    thrd_create(&t, run, &arg);

    printf("Doing other things while the thread runs\n");

    printf("Waiting for thread to complete...\n");

    int res;  // Holds return value from the thread exit

    // Wait here for the thread to complete; store the return value
    // in res:

    thrd_join(t, &res);

    printf("Thread exited with return value %d\n", res);
}
```

Thấy cách ta làm `thrd_create()` ở đó để gọi hàm `run()` không? Rồi ta làm những việc khác trong `main()` rồi dừng và đợi thread hoàn thành với `thrd_join()`.

Kết quả mẫu (của bạn có thể khác):

```
Launching a thread
Doing other things while the thread runs
Waiting for thread to complete...
THREAD: Running thread with arg 3490
Thread exited with return value 12
```

`arg` bạn truyền cho hàm phải có lifetime đủ dài để thread có thể lấy nó trước khi nó biến mất. Và nó cần không bị thread chính ghi đè trước khi thread mới kịp dùng.

Xem ví dụ launch 5 threads. Một thứ cần lưu ý ở đây là cách ta dùng mảng `thrd_t` để theo dõi tất cả thread ID.

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

int run(void *arg)
{
    int i = *(int*)arg;

    printf("THREAD %d: running!\n", i);

    return i;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    int i;

    printf("Launching threads...\n");
    for (i = 0; i < THREAD_COUNT; i++)

        // NOTE! In the following line, we pass a pointer to i,
        // but each thread sees the same pointer. So they'll
        // print out weird things as i changes value here in
        // the main thread! (More in the text, below.)

        thrd_create(t + i, run, &i);

    printf("Doing other things while the thread runs...\n");
    printf("Waiting for thread to complete...\n");

    for (int i = 0; i < THREAD_COUNT; i++) {
        int res;
        thrd_join(t[i], &res);

        printf("Thread %d complete!\n", res);
    }

    printf("All threads complete!\n");
}
```

Khi tôi chạy các threads, tôi đếm `i` từ 0 đến 4 và truyền con trỏ tới nó cho `thrd_create()`. Con trỏ này đi tới hàm `run()` nơi ta tạo một bản sao.

Đủ đơn giản chưa? Đây là output:

```
Launching threads...
THREAD 2: running!
THREAD 3: running!
THREAD 4: running!
THREAD 2: running!
Doing other things while the thread runs...
```

```
Waiting for thread to complete...
Thread 2 complete!
Thread 2 complete!
THREAD 5: running!
Thread 3 complete!
Thread 4 complete!
Thread 5 complete!
All threads complete!
```

Cái giiiiii? `THREAD 0` đâu rồi? Và sao có `THREAD 5` khi rõ ràng `i` không bao giờ lớn hơn `4` lúc gọi `thrd_create()`? Và hai `THREAD 2`? Điên rồi!

Đây là đang bước vào vùng đất vui vẻ của *race conditions*. Thread chính đang sửa `i` trước khi thread có cơ hội copy nó. Thực ra `i` đi tới tận `5` và kết thúc vòng lặp trước khi thread cuối cùng có cơ hội copy nó.

Ta cần có biến per-thread để tham chiếu sao cho có thể truyền vào làm `arg`.

Có thể có mảng lớn. Hoặc có thể `malloc()` chỗ (và free nó ở đâu đó, có thể trong chính thread.)

Thử vậy xem:

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

int run(void *arg)
{
    int i = *(int*)arg;  // Copy the arg

    free(arg);  // Done with this

    printf("THREAD %d: running!\n", i);

    return i;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    int i;

    printf("Launching threads...\n");
    for (i = 0; i < THREAD_COUNT; i++) {

        // Get some space for a per-thread argument:

        int *arg = malloc(sizeof *arg);
        *arg = i;

        thrd_create(t + i, run, arg);
    }

    // ...
```

Chú ý trên dòng 27-30 ta `malloc()` chỗ cho một `int` và copy giá trị của `i` vào đó. Mỗi thread mới

nhận biến `malloc()`-mới-toanh của riêng mình và ta truyền con trỏ tới nó cho hàm `run()`.

Khi `run()` tạo bản copy của `arg` ở dòng 7, nó `free()` cái `int` đã `malloc()`. Và giờ nó có bản sao riêng, muốn làm gì thì làm.

Và chạy cho thấy kết quả:

```
Launching threads...
THREAD 0: running!
THREAD 1: running!
THREAD 2: running!
THREAD 3: running!
Doing other things while the thread runs...
Waiting for thread to complete...
Thread 0 complete!
Thread 1 complete!
Thread 2 complete!
Thread 3 complete!
THREAD 4: running!
Thread 4 complete!
All threads complete!
```

Đấy! Threads 0-4 đều có mặt!

Lần chạy của bạn có thể khác, thread được schedule chạy thế nào nằm ngoài C spec. Ta thấy ví dụ trên thread 4 không thậm chí bắt đầu đến khi threads 0-1 đã xong. Thực tế, nếu chạy lại tôi có thể ra output khác. Ta không thể đảm bảo thứ tự thực thi thread.

39.5 Detach Threads

Nếu bạn muốn fire-and-forget một thread (tức là bạn không phải `thrd_join()` nó sau này), bạn làm được với `thrd_detach()`.

Cái này loại bỏ khả năng của thread cha lấy return value từ thread con, nhưng nếu bạn không quan tâm và chỉ muốn threads tự dọn dẹp đẹp đẽ, thì đây là cách đi.

Về cơ bản ta sẽ làm vầy:

```
thrd_create(&t, run, NULL);
thrd_detach(t);
```

chỗ gọi `thrd_detach()` là thread cha nói, "Này, tôi sẽ không đợi thread con này hoàn thành với `thrd_join()`. Nên cứ tự dọn dẹp nó khi xong nhé."

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

int run(void *arg)
{
    (void)arg;

    //printf("Thread running! %lu\n", thrd_current()); // non-portable!
    printf("Thread running!\n");

    return 0;
}

#define THREAD_COUNT 10
```

```
int main(void)
{
    thrd_t t;

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_create(&t, run, NULL);
        thrd_detach(t);               // <-- DETACH!
    }

    // Sleep for a second to let all the threads finish
    thrd_sleep(&(struct timespec){.tv_sec=1}, NULL);
}
```

Lưu ý trong code này ta cho thread chính sleep 1 giây bằng `thrd_sleep()`, nói thêm sau.

Ngoài ra trong hàm `run()`, tôi có dòng comment-out in thread ID dưới dạng `unsigned long`. Cái này không portable, vì spec không nói kiểu bên dưới của `thrd_t` là gì, nó có thể là một `struct` ai biết. Nhưng dòng đó chạy được trên máy tôi.

Một điều thú vị tôi thấy khi chạy code trên và in thread ID là vài threads có ID trùng nhau! Tưởng chừng là không thể, nhưng C cho phép *reuse* thread ID sau khi thread tương ứng đã thoát. Nên cái tôi thấy là vài threads đã xong trước khi threads khác được launch.

39.6 Dữ liệu cục bộ theo Thread

Threads thú vị vì chúng không có bộ nhớ riêng ngoài biến cục bộ. Nếu bạn muốn biến `static` hay biến phạm vi file, tất cả threads sẽ thấy cùng biến đó.

Cái này có thể dẫn tới race conditions, nơi bạn gặp *Chuyện Lạ*™ xảy ra.

Xem ví dụ này. Ta có biến `static foo` trong block scope ở `run()`. Biến này sẽ thấy được bởi tất cả threads đi qua hàm `run()`. Và các threads thực sự có thể giẫm chân nhau.

Mỗi thread copy `foo` vào biến cục bộ `x` (không chia sẻ giữa threads, tất cả threads có call stack riêng). Nên chúng *nên* bằng nhau, đúng không?

Và lần đầu in ra, chúng bằng[6]. Nhưng rồi ngay sau đó, ta kiểm tra xem chúng có còn bằng không.

Và *thường thì* bằng. Nhưng không phải luôn luôn!

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

int run(void *arg)
{
    int n = *(int*)arg;  // Thread number for humans to differentiate

    free(arg);

    static int foo = 10;  // Static value shared between threads

    int x = foo;  // Automatic local variable--each thread has its own

    // We just assigned x from foo, so they'd better be equal here.
    // (In all my test runs, they were, but even this isn't guaranteed!)
```

[6]Dù tôi không nghĩ chúng phải vậy. Chỉ là threads có vẻ không được reschedule cho đến khi xảy ra system call nào đó như `printf()` ... đó là lý do tôi để `printf()` trong đó.

```
    printf("Thread %d: x = %d, foo = %d\n", n, x, foo);

    // And they should be equal here, but they're not always!
    // (Sometimes they were, sometimes they weren't!)

    // What happens is another thread gets in and increments foo
    // right now, but this thread's x remains what it was before!

    if (x != foo) {
        printf("Thread %d: Craziness! x != foo! %d != %d\n", n, x, foo);
    }

    foo++;  // Increment shared value

    return 0;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    for (int i = 0; i < THREAD_COUNT; i++) {
        int *n = malloc(sizeof *n);  // Holds a thread serial number
        *n = i;
        thrd_create(t + i, run, n);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }
}
```

Đây là một output mẫu (thay đổi giữa các lần chạy):

```
Thread 0: x = 10, foo = 10
Thread 1: x = 10, foo = 10
Thread 1: Craziness! x != foo! 10 != 11
Thread 2: x = 12, foo = 12
Thread 4: x = 13, foo = 13
Thread 3: x = 14, foo = 14
```

Trong thread 1, giữa hai lần `printf()`, giá trị `foo` bằng cách nào đó đổi từ `10` thành `11`, dù rõ ràng không có increment nào giữa hai `printf()`!

Đó là một thread khác chen vào (có vẻ là thread 0 theo hình thù) và increment `foo` sau lưng thread 1!

Hãy giải quyết vấn đề này theo hai cách khác nhau. (Nếu bạn muốn tất cả threads cùng dùng chung biến *và* không giẫm chân nhau, bạn phải đọc tiếp tới phần mutex.)

39.6.1 Storage-Class `_Thread_local`

Trước hết, cứ nhìn cách dễ nhất để đi vòng: storage-class `_Thread_local`.

Về cơ bản ta chỉ dán nó vào trước biến `static` block scope và chuyện sẽ chạy! Nó bảo C rằng mỗi thread nên có phiên bản riêng của biến này, nên không có thread nào giẫm chân nhau.

File header `<threads.h>` định nghĩa `thread_local` là alias của `_Thread_local` nên code bạn khỏi xấu.

Lấy ví dụ trước và biến `foo` thành biến `thread_local` để ta không chia sẻ dữ liệu.

```
int run(void *arg)
{
    int n = *(int*)arg;  // Thread number for humans to differentiate

    free(arg);

    thread_local static int foo = 10;  // <-- No longer shared!!
```

Và chạy ta được:

```
Thread 0: x = 10, foo = 10
Thread 1: x = 10, foo = 10
Thread 2: x = 10, foo = 10
Thread 4: x = 10, foo = 10
Thread 3: x = 10, foo = 10
```

Không còn rắc rối lạ nữa!

Một điều: nếu biến `thread_local` là block scope, nó **phải** `static`. Luật là vậy. (Nhưng vẫn OK vì biến không-`static` đã là per-thread rồi do mỗi thread có biến không-`static` riêng.)

Hơi nói dối chút: biến `thread_local` block scope còn có thể `extern`.

39.6.2 Một lựa chọn khác: Thread-Specific Storage

Thread-specific storage (TSS) là một cách khác để có dữ liệu per-thread.

Một tính năng thêm là các hàm này cho phép bạn chỉ định destructor sẽ được gọi trên dữ liệu khi biến TSS bị xoá. Thông thường destructor là `free()` để tự động dọn dữ liệu `malloc()` per-thread. Hoặc `NULL` nếu không cần destroy gì.

Destructor có kiểu `tss_dtor_t` là con trỏ đến hàm trả về `void` và nhận `void*` làm tham số (cái `void*` trỏ tới dữ liệu lưu trong biến). Nói cách khác, nó là `void (*)(void*)`, nếu có rõ thêm. Tôi thừa nhận chắc là không. Xem ví dụ bên dưới.

Nói chung `thread_local` chắc là lựa chọn mặc định, nhưng nếu bạn thích ý tưởng destructor thì tận dụng.

Cách dùng hơi lạ ở chỗ ta cần biến kiểu `tss_t` sống để đại diện giá trị trên cơ sở per-thread. Rồi ta khởi tạo nó với `tss_create()`. Cuối cùng xoá với `tss_delete()`. Chú ý gọi `tss_delete()` không chạy tất cả destructors, chính `thrd_exit()` (hoặc return từ hàm run) mới làm. `tss_delete()` chỉ giải phóng bộ nhớ do `tss_create()` cấp phát.

Ở giữa, threads có thể gọi `tss_set()` và `tss_get()` để đặt và lấy giá trị.

Trong code sau, ta set up biến TSS trước khi tạo threads, rồi dọn dẹp sau khi threads xong.

Trong hàm `run()`, threads `malloc()` chỗ cho một chuỗi và lưu con trỏ đó vào biến TSS.

Khi thread thoát, hàm destructor (`free()` trong trường hợp này) được gọi cho *tất cả* threads.

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

```
tss_t str;

void some_function(void)
{
    // Retrieve the per-thread value of this string
    char *tss_string = tss_get(str);

    // And print it
    printf("TSS string: %s\n", tss_string);
}

int run(void *arg)
{
    int serial = *(int*)arg;  // Get this thread's serial number
    free(arg);

    // malloc() space to hold the data for this thread
    char *s = malloc(64);
    sprintf(s, "thread %d! :)", serial);  // Happy little string

    // Set this TSS variable to point at the string
    tss_set(str, s);

    // Call a function that will get the variable
    some_function();

    return 0;   // Equivalent to thrd_exit(0)
}

#define THREAD_COUNT 15

int main(void)
{
    thrd_t t[THREAD_COUNT];

    // Make a new TSS variable, the free() function is the destructor
    tss_create(&str, free);

    for (int i = 0; i < THREAD_COUNT; i++) {
        int *n = malloc(sizeof *n);  // Holds a thread serial number
        *n = i;
        thrd_create(t + i, run, n);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }

    // All threads are done, so we're done with this
    tss_delete(str);
}
```

Lại nữa, đây là cách làm khá đau so với `thread_local`, nên trừ khi bạn thực sự cần chức năng destructor, tôi sẽ dùng `thread_local`.

39.7 Mutexes

Nếu bạn chỉ muốn cho một thread vào critical section của code tại một thời điểm, bạn có thể bảo vệ đoạn đó bằng mutex[7].

Ví dụ, nếu ta có biến `static` và muốn có thể lấy và set nó bằng hai thao tác mà không thread khác nhảy vào giữa làm hỏng, ta có thể dùng mutex.

Bạn có thể acquire mutex hay release nó. Nếu bạn cố acquire mutex và thành công, bạn có thể tiếp tục thực thi. Nếu cố mà thất bại (vì người khác đang giữ), bạn sẽ *block*[8] đến khi mutex được release.

Nếu nhiều threads bị block đợi một mutex được release, một trong chúng sẽ được chọn để chạy (ngẫu nhiên từ góc nhìn của ta), còn các thread khác tiếp tục ngủ.

Kế hoạch là đầu tiên ta sẽ khởi tạo biến mutex để sẵn sàng dùng bằng `mtx_init()`.

Rồi các threads tiếp theo có thể gọi `mtx_lock()` và `mtx_unlock()` để lấy và release mutex.

Khi ta đã hoàn toàn xong với mutex, có thể destroy với `mtx_destroy()`, đối lập logic của `mtx_init()`.

Đầu tiên xem code *không* dùng mutex và cố in ra serial number chia sẻ (`static`) rồi tăng nó. Vì ta không dùng mutex trên việc lấy giá trị (để in) và set (để tăng), threads có thể giẫm chân nhau trong critical section đó.

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

int run(void *arg)
{
    (void)arg;

    static int serial = 0;   // Shared static variable!

    printf("Thread running! %d\n", serial);

    serial++;

    return 0;
}

#define THREAD_COUNT 10

int main(void)
{
    thrd_t t[THREAD_COUNT];

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_create(t + i, run, NULL);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }
}
```

Khi tôi chạy, tôi được cái gì đó như:

[7] Viết tắt của "mutual exclusion", cũng gọi là "lock" trên một đoạn code mà chỉ một thread được phép thực thi.
[8] Tức là process của bạn sẽ đi ngủ.

```
Thread running! 0
Thread running! 0
Thread running! 0
Thread running! 3
Thread running! 4
Thread running! 5
Thread running! 6
Thread running! 7
Thread running! 8
Thread running! 9
```

Rõ ràng nhiều threads chen vào chạy `printf()` trước khi ai đó có cơ hội cập nhật biến `serial`.

Cái ta muốn làm là bọc việc lấy biến và set biến thành một đoạn code được mutex bảo vệ duy nhất.

Ta sẽ thêm biến mới đại diện mutex kiểu `mtx_t` trong phạm vi file, khởi tạo, rồi threads có thể lock và unlock nó trong hàm `run()`.

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

mtx_t serial_mtx;     // <-- MUTEX VARIABLE

int run(void *arg)
{
    (void)arg;

    static int serial = 0;   // Shared static variable!

    // Acquire the mutex--all threads will block on this call until
    // they get the lock:

    mtx_lock(&serial_mtx);           // <-- ACQUIRE MUTEX

    printf("Thread running! %d\n", serial);

    serial++;

    // Done getting and setting the data, so free the lock. This will
    // unblock threads on the mtx_lock() call:

    mtx_unlock(&serial_mtx);         // <-- RELEASE MUTEX

    return 0;
}

#define THREAD_COUNT 10

int main(void)
{
    thrd_t t[THREAD_COUNT];

    // Initialize the mutex variable, indicating this is a normal
    // no-frills, mutex:

    mtx_init(&serial_mtx, mtx_plain);        // <-- CREATE MUTEX
```

```
    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_create(t + i, run, NULL);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }

    // Done with the mutex, destroy it:

    mtx_destroy(&serial_mtx);               // <-- DESTROY MUTEX
}
```

Xem cách dòng 38 và 50 trong `main()` ta khởi tạo và destroy mutex.

Nhưng từng thread acquire mutex ở dòng 15 và release ở dòng 24.

Giữa `mtx_lock()` và `mtx_unlock()` là *critical section*, vùng code mà ta không muốn nhiều threads lộn xộn cùng lúc.

Và giờ ta có output đúng!

```
Thread running! 0
Thread running! 1
Thread running! 2
Thread running! 3
Thread running! 4
Thread running! 5
Thread running! 6
Thread running! 7
Thread running! 8
Thread running! 9
```

Nếu bạn cần nhiều mutexes, không vấn đề: cứ có nhiều biến mutex.

Và nhớ Quy Tắc Số Một của Nhiều Mutex: *Unlock mutexes theo thứ tự ngược với thứ tự lock!*

39.7.1 Các kiểu Mutex khác nhau

Như đã gợi ý, ta có vài kiểu mutex tạo được với `mtx_init()`. (Một số kiểu là kết quả của phép bitwise-OR, như ghi trong bảng.)

Kiểu	Mô tả
`mtx_plain`	Mutex bình thường
`mtx_timed`	Mutex hỗ trợ timeouts
`mtx_plain|mtx_recursive`	Mutex đệ quy
`mtx_timed|mtx_recursive`	Mutex đệ quy hỗ trợ timeouts

"Đệ quy" nghĩa là người giữ lock có thể gọi `mtx_lock()` nhiều lần trên cùng một lock. (Họ phải unlock một số lần bằng nhau trước khi ai khác có thể lấy mutex.) Cái này có thể làm code dễ hơn đôi khi, đặc biệt khi bạn gọi một hàm cần lock mutex trong khi bạn đã giữ mutex.

Và timeout cho thread cơ hội *cố* lấy lock một lúc, nhưng rồi bỏ cuộc nếu không lấy được trong khung thời gian đó.

Với mutex timeout, nhớ tạo với `mtx_timed`:

```
mtx_init(&serial_mtx, mtx_timed);
```

Rồi khi bạn đợi nó, bạn phải chỉ định thời gian theo UTC khi nào unlock[9].

Hàm `timespec_get()` từ `<time.h>` có thể trợ giúp. Nó cho bạn thời gian hiện tại theo UTC trong `struct timespec` đúng cái ta cần. Thực tế có vẻ nó tồn tại chỉ vì mục đích này.

Nó có hai field: `tv_sec` giữ thời gian hiện tại tính bằng giây kể từ epoch, và `tv_nsec` giữ nanosecond (phần tỉ của giây) như phần "fractional".

Nên bạn có thể load nó với thời gian hiện tại, rồi cộng thêm để có timeout cụ thể.

Rồi gọi `mtx_timedlock()` thay vì `mtx_lock()`. Nếu nó trả về giá trị `thrd_timedout`, là đã timeout.

```
struct timespec timeout;

timespec_get(&timeout, TIME_UTC);  // Get current time
timeout.tv_sec += 1;               // Timeout 1 second after now

int result = mtx_timedlock(&serial_mtx, &timeout));

if (result == thrd_timedout) {
    printf("Mutex lock timed out!\n");
}
```

Ngoài cái đó ra, timed lock giống regular lock.

39.8 Condition Variables

Condition Variables là mảnh ghép cuối ta cần để làm các ứng dụng đa luồng có hiệu năng và compose các cấu trúc đa luồng phức tạp hơn.

Condition variable cung cấp cách để threads đi ngủ cho đến khi một event nào đó trên thread khác xảy ra.

Nói cách khác, ta có thể có một số threads đã sẵn sàng chạy, nhưng phải đợi đến khi event nào đó là true trước khi tiếp tục. Về cơ bản chúng được bảo "đợi đấy!" đến khi được thông báo.

Và cái này đi đôi với mutex vì cái ta đợi thường phụ thuộc vào giá trị của dữ liệu nào đó, và dữ liệu đó thường cần được mutex bảo vệ.

Quan trọng là bản thân condition variable không phải người giữ dữ liệu cụ thể nào từ góc nhìn của ta. Nó chỉ là biến qua đó C theo dõi trạng thái waiting/not-waiting của một thread hay nhóm threads cụ thể.

Hãy viết một chương trình giả tạo đọc vào nhóm 5 số từ thread chính một lần một số. Rồi khi 5 số đã nhập, thread con thức dậy, cộng 5 số đó, và in kết quả.

Các số sẽ được lưu trong mảng chia sẻ toàn cục, cũng như index của mảng cho số sắp được nhập.

Vì đây là giá trị chia sẻ, ta ít nhất phải giấu chúng sau mutex cho cả thread chính và thread con. (Chính sẽ ghi dữ liệu vào, còn con sẽ đọc dữ liệu ra.)

Nhưng chừng đó chưa đủ. Thread con cần block ("ngủ") đến khi 5 số đã được đọc vào mảng. Rồi thread cha cần đánh thức thread con để nó làm việc.

Và khi thức dậy, nó cần đang giữ mutex đó. Và sẽ vậy! Khi một thread đợi trên condition variable, nó cũng acquire mutex khi thức dậy.

Tất cả cái này xảy ra quanh một biến thêm kiểu `cnd_t` chính là *condition variable*. Ta tạo biến này bằng hàm `cnd_init()` và destroy khi xong với hàm `cnd_destroy()`.

[9] Bạn có thể đã nghĩ đó là "thời gian kể từ bây giờ", nhưng bạn chỉ muốn nghĩ vậy thôi phải không!

Nhưng hoạt động thế nào? Xem phác thảo những gì thread con sẽ làm:

1. Lock mutex với `mtx_lock()`
2. Nếu chưa nhập hết số, đợi trên condition variable với `cnd_wait()`
3. Làm việc cần làm
4. Unlock mutex với `mtx_unlock()`

Cùng lúc thread chính sẽ làm:

1. Lock mutex với `mtx_lock()`
2. Lưu số vừa đọc vào mảng
3. Nếu mảng đầy, signal con thức dậy với `cnd_signal()`
4. Unlock mutex với `mtx_unlock()`

Nếu bạn không đọc lướt quá (OK, tôi không giận), bạn có thể nhận ra điều lạ: làm sao thread chính giữ lock mutex và signal con, nếu con phải giữ lock mutex để đợi signal? Cả hai không cùng giữ được!

Và đúng vậy! Có tí ma thuật sau cánh gà với condition variable: khi bạn `cnd_wait()`, nó release mutex bạn chỉ định và thread đi ngủ. Và khi ai đó signal thread thức dậy, nó reacquire lock như không có gì xảy ra.

Bên `cnd_signal()` thì hơi khác. Cái này không làm gì với mutex. Thread signal vẫn phải tự release mutex trước khi threads đang đợi thức dậy được.

Một điều nữa về `cnd_wait()`. Bạn sẽ gọi `cnd_wait()` nếu điều kiện nào đó[10] chưa đạt (ví dụ trong trường hợp này, nếu chưa nhập hết số). Thoả thuận là: điều kiện này nên trong `while` loop, không phải `if`. Tại sao?

Vì một hiện tượng bí ẩn gọi là *spurious wakeup*. Đôi khi, trong vài implementation, một thread có thể bị đánh thức từ giấc ngủ `cnd_wait()` mà dường như *không có lý do. [nhạc X-Files]*[11]. Và ta phải kiểm tra xem điều kiện ta cần có còn thực sự đúng khi thức dậy không. Nếu không, đi ngủ lại!

Nào, làm thôi! Bắt đầu với thread chính:

- Thread chính sẽ set up mutex và condition variable, và launch thread con.
- Rồi trong vòng lặp vô hạn, lấy số làm input từ console.
- Nó cũng acquire mutex để lưu số đã nhập vào mảng toàn cục.
- Khi mảng có 5 số, thread chính sẽ signal thread con rằng đến lúc thức dậy làm việc.
- Rồi thread chính unlock mutex và quay lại đọc số tiếp từ console.

Trong khi đó, thread con làm trò riêng:

- Thread con lấy mutex.
- Trong khi điều kiện chưa đạt (tức mảng chia sẻ chưa có 5 số), thread con ngủ bằng cách đợi trên condition variable. Khi đợi, nó ngầm unlock mutex.
- Khi thread chính signal thread con thức dậy, nó thức dậy làm việc và lấy lại mutex lock.
- Thread con cộng các số và reset biến index vào mảng.
- Rồi release mutex và chạy lại trong vòng lặp vô hạn.

Và đây là code! Nhìn kỹ để thấy chỗ các mảnh trên được xử lý:

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

#define VALUE_COUNT_MAX 5
```

[10] Và đó là lý do gọi là *condition variables*!

[11] Tôi không nói là người ngoài hành tinh... nhưng là người ngoài hành tinh. OK, thực tế nhiều khả năng hơn là một thread khác đã được đánh thức và làm được việc trước.

```
int value[VALUE_COUNT_MAX];  // Shared global
int value_count = 0;   // Shared global, too

mtx_t value_mtx;   // Mutex around value
cnd_t value_cnd;   // Condition variable on value

int run(void *arg)
{
    (void)arg;

    for (;;) {
        mtx_lock(&value_mtx);      // <-- GRAB THE MUTEX

        while (value_count < VALUE_COUNT_MAX) {
            printf("Thread: is waiting\n");
            cnd_wait(&value_cnd, &value_mtx);  // <-- CONDITION WAIT
        }

        printf("Thread: is awake!\n");

        int t = 0;

        // Add everything up
        for (int i = 0; i < VALUE_COUNT_MAX; i++)
            t += value[i];

        printf("Thread: total is %d\n", t);

        // Reset input index for main thread
        value_count = 0;

        mtx_unlock(&value_mtx);   // <-- MUTEX UNLOCK
    }

    return 0;
}

int main(void)
{
    thrd_t t;

    // Spawn a new thread

    thrd_create(&t, run, NULL);
    thrd_detach(t);

    // Set up the mutex and condition variable

    mtx_init(&value_mtx, mtx_plain);
    cnd_init(&value_cnd);

    for (;;) {
        int n;

        scanf("%d", &n);

        mtx_lock(&value_mtx);    // <-- LOCK MUTEX
```

```
        value[value_count++] = n;

        if (value_count == VALUE_COUNT_MAX) {
            printf("Main: signaling thread\n");
            cnd_signal(&value_cnd);  // <-- SIGNAL CONDITION
        }

        mtx_unlock(&value_mtx);  // <-- UNLOCK MUTEX
    }

    // Clean up (I know that's an infinite loop above here, but I
    // want to at least pretend to be proper):

    mtx_destroy(&value_mtx);
    cnd_destroy(&value_cnd);
}
```

Và đây là output mẫu (các số trên từng dòng là input của tôi):

```
Thread: is waiting
1
1
1
1
1
Main: signaling thread
Thread: is awake!
Thread: total is 5
Thread: is waiting
2
8
5
9
0
Main: signaling thread
Thread: is awake!
Thread: total is 24
Thread: is waiting
```

Đây là cách dùng phổ biến của condition variable trong tình huống producer-consumer như vầy. Nếu ta không có cách cho thread con ngủ trong khi đợi điều kiện nào đó đạt, nó sẽ buộc phải poll, lãng phí CPU kinh khủng.

39.8.1 Timed Condition Wait

Có một biến thể của `cnd_wait()` cho phép bạn chỉ định timeout để ngừng đợi.

Vì thread con phải relock mutex, cái này không có nghĩa là bạn sẽ bật ngay trở lại khoảnh khắc timeout xảy ra; bạn vẫn phải đợi các threads khác release mutex.

Nhưng có nghĩa là bạn sẽ không đợi cho đến khi `cnd_signal()` xảy ra.

Để dùng, gọi `cnd_timedwait()` thay vì `cnd_wait()`. Nếu nó trả về giá trị `thrd_timedout`, là timeout.

Timestamp là thời gian tuyệt đối theo UTC, không phải time-from-now. May thay hàm `timespec_get()` trong `<time.h>` có vẻ được làm ra đúng cho trường hợp này.

```
struct timespec timeout;

timespec_get(&timeout, TIME_UTC);  // Get current time
timeout.tv_sec += 1;               // Timeout 1 second after now

int result = cnd_timedwait(&condition, &mutex, &timeout));

if (result == thrd_timedout) {
    printf("Condition variable timed out!\n");
}
```

39.8.2 Broadcast: Đánh thức mọi Thread đang đợi

`cnd_signal()` chỉ đánh thức một thread để tiếp tục làm việc. Tuỳ logic bạn làm, có thể hợp lý khi đánh thức nhiều hơn một thread tiếp tục khi điều kiện đạt.

Tất nhiên chỉ một thread có thể lấy mutex, nhưng nếu bạn có tình huống:

- Thread mới thức dậy chịu trách nhiệm đánh thức thread kế, và,
- Có khả năng điều kiện loop spurious-wakeup sẽ ngăn nó làm vậy,

thì bạn sẽ muốn broadcast wake up để chắc chắn lấy được ít nhất một thread ra khỏi loop đó để launch thread tiếp.

Cách làm?

Đơn giản dùng `cnd_broadcast()` thay vì `cnd_signal()`. Dùng giống hệt, chỉ khác `cnd_broadcast()` đánh thức **tất cả** sleeping threads đang đợi trên condition variable đó.

39.9 Chạy một hàm đúng một lần

Giả sử bạn có một hàm *có thể* được chạy bởi nhiều threads, nhưng bạn không biết khi nào, và không đáng công viết tất cả logic đó.

Có cách đi vòng: dùng `call_once()`. Cả đống threads có thể cố chạy hàm, nhưng chỉ thread đầu tiên được tính[12]

Để làm, bạn cần một biến flag đặc biệt bạn khai báo để theo dõi xem cái đó đã được chạy chưa. Và bạn cần một hàm để chạy, không nhận tham số và không trả giá trị.

```
once_flag of = ONCE_FLAG_INIT;  // Initialize it like this

void run_once_function(void)
{
    printf("I'll only run once!\n");
}

int run(void *arg)
{
    (void)arg;

    call_once(&of, run_once_function);

    // ...
```

Trong ví dụ này, không quan trọng bao nhiêu threads tới hàm `run()`, `run_once_function()` sẽ chỉ được gọi một lần duy nhất.

[12]Sự sống sót của kẻ khoẻ nhất! Đúng không? Tôi thừa nhận thực ra không giống vậy chút nào.

Chapter 40

Atomics

> *"They tried and failed, all of them?"*
> *"Oh, no." She shook her head. "They tried and died."*
>
> —Paul Atreides and The Reverend Mother Gaius Helen Mohiam, *Dune*

Đây là một trong những khía cạnh thử thách hơn của đa luồng với C. Nhưng ta sẽ cố gắng thong thả.

Về cơ bản tôi sẽ nói về các cách dùng đơn giản hơn của biến atomic, chúng là gì, và hoạt động thế nào, vân vân. Và tôi sẽ nhắc tới vài con đường điên-rồ-phức-tạp có sẵn cho bạn.

Nhưng tôi sẽ không đi theo những con đường đó. Không chỉ vì tôi hiếm đủ khả năng để viết về chúng, mà tôi nghĩ nếu bạn biết mình cần chúng, bạn đã biết nhiều hơn tôi rồi.

Nhưng ngay cả phần cơ bản cũng có những thứ lạ. Nên cài dây an toàn nào mọi người, vì Kansas sắp tạm biệt rồi đây.

40.1 Kiểm tra hỗ trợ Atomic

Atomics là tính năng tuỳ chọn. Có một macro `__STDC_NO_ATOMICS__` bằng `1` nếu bạn *không* có atomics.

Macro đó có thể không tồn tại trước C11, nên ta nên kiểm tra phiên bản ngôn ngữ với `__STDC_VERSION__`[1].

```
#if __STDC_VERSION__ < 201112L || __STDC_NO_ATOMICS__ == 1
#define HAS_ATOMICS 0
#else
#define HAS_ATOMICS 1
#endif
```

Nếu những test đó qua, bạn có thể an toàn include `<stdatomic.h>`, header làm cơ sở cho phần còn lại của chương này. Nhưng nếu không có hỗ trợ atomic, header đó có thể thậm chí không tồn tại.

Trên vài hệ thống, bạn có thể cần thêm `-latomic` vào cuối dòng lệnh compile để dùng các hàm trong header đó.

40.2 Biến Atomic

Đây là *một phần* của cách biến atomic hoạt động:

Nếu bạn có biến atomic chia sẻ và ghi vào nó từ một thread, lần ghi đó sẽ là *all-or-nothing* trong thread khác.

[1] Macro `__STDC_VERSION__` không tồn tại trong C89 đời đầu, nên nếu bạn lo về điều đó, kiểm tra với `#ifdef`.

Tức là thread khác sẽ thấy toàn bộ lần ghi, ví dụ giá trị 32 bit. Không phải một nửa. Không có cách nào để một thread ngắt thread khác đang ở *giữa* một lần ghi atomic nhiều byte.

Gần như có một cái lock nhỏ quanh việc lấy và set biến đó. (Và có *thể* có thật! Xem Biến Atomic Lock-Free bên dưới.)

Nhân đây, bạn có thể thoát khỏi việc dùng atomic nếu bạn dùng mutex để lock critical section. Chỉ là có một lớp *cấu trúc dữ liệu lock-free* luôn cho phép các thread khác tiến tới thay vì bị block bởi mutex... nhưng cái này tạo ra đúng từ đầu khá khó, và là một trong những thứ nằm ngoài phạm vi guide này, buồn thay.

Đó mới chỉ là một phần. Nhưng là phần ta bắt đầu.

Trước khi đi tiếp, làm sao khai báo biến là atomic?

Đầu tiên, include `<stdatomic.h>`.

Cái này cho ta các kiểu như `atomic_int`.

Rồi ta có thể đơn giản khai báo biến có kiểu đó.

Nhưng hãy làm demo có hai thread. Thread thứ nhất chạy một hồi rồi set một biến thành giá trị cụ thể, rồi thoát. Thread kia chạy cho đến khi thấy giá trị đó được set, rồi thoát.

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

atomic_int x;   // THE POWER OF ATOMICS! BWHAHAHA!

int thread1(void *arg)
{
    (void)arg;

    printf("Thread 1: Sleeping for 1.5 seconds\n");
    thrd_sleep(&(struct timespec){.tv_sec=1, .tv_nsec=500000000}, NULL);

    printf("Thread 1: Setting x to 3490\n");
    x = 3490;

    printf("Thread 1: Exiting\n");
    return 0;
}

int thread2(void *arg)
{
    (void)arg;

    printf("Thread 2: Waiting for 3490\n");
    while (x != 3490) {}  // spin here

    printf("Thread 2: Got 3490--exiting!\n");
    return 0;
}

int main(void)
{
    x = 0;

    thrd_t t1, t2;
```

```
    thrd_create(&t1, thread1, NULL);
    thrd_create(&t2, thread2, NULL);

    thrd_join(t1, NULL);
    thrd_join(t2, NULL);

    printf("Main    : Threads are done, so x better be 3490\n");
    printf("Main    : And indeed, x == %d\n", x);
}
```

Thread thứ hai spin tại chỗ, nhìn vào cờ và đợi nó được set thành giá trị `3490`. Và thread thứ nhất làm điều đó.

Và tôi nhận được output này:

```
Thread 1: Sleeping for 1.5 seconds
Thread 2: Waiting for 3490
Thread 1: Setting x to 3490
Thread 1: Exiting
Thread 2: Got 3490--exiting!
Main    : Threads are done, so x better be 3490
Main    : And indeed, x == 3490
```

Nhìn nè, mẹ ơi! Ta đang truy cập biến từ các thread khác nhau mà không dùng mutex! Và cái đó chạy mọi lần nhờ bản chất atomic của biến atomic.

Bạn có thể đang thắc mắc chuyện gì xảy ra nếu đó là `int` thường không-atomic. Trên máy tôi vẫn chạy... trừ khi tôi build có optimization thì nó hang trên thread 2 đợi thấy 3490 được set[2].

Nhưng đó mới chỉ là đầu câu chuyện. Phần tiếp sẽ cần nhiều năng lực não hơn và có liên quan một thứ gọi là *synchronization*.

40.3 Synchronization

Phần tiếp của câu chuyện là về khi nào các lần ghi bộ nhớ trong một thread trở nên có thể thấy với các thread khác.

Bạn có thể nghĩ là ngay lập tức, đúng không? Nhưng không phải. Nhiều thứ có thể sai. Sai một cách kỳ lạ.

Compiler có thể đã sắp xếp lại các truy cập bộ nhớ nên khi bạn nghĩ đã set giá trị tương đối so với cái khác có thể không đúng. Và dù compiler không làm, CPU của bạn có thể đã làm on the fly. Hoặc có thể có gì đó khác về kiến trúc đó gây ra việc ghi trên một CPU bị delay trước khi có thể thấy trên CPU khác.

Tin tốt là ta có thể gom tất cả rắc rối tiềm ẩn này vào một: các truy cập bộ nhớ không đồng bộ có thể xuất hiện không theo thứ tự tuỳ thread nào đang quan sát, như thể các dòng code đã bị sắp xếp lại.

Lấy ví dụ, cái nào xảy ra trước trong code sau, ghi `x` hay ghi `y`?

```
int x, y;  // global

// ...

x = 2;
y = 3;
```

[2]Lý do là khi optimize, compiler của tôi đã đặt giá trị `x` vào register để làm `while` loop nhanh. Nhưng register không có cách nào biết biến đã được cập nhật trong thread khác, nên nó không bao giờ thấy `3490`. Cái này không thực sự liên quan phần *all-or-nothing* của atomicity, mà liên quan hơn đến các khía cạnh đồng bộ trong phần tiếp.

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

Đáp án: ta không biết. Compiler hay CPU có thể âm thầm đảo dòng 5 và 6 mà ta không hay. Code sẽ chạy single-thread *như thể* nó được thực thi theo thứ tự code.

Trong kịch bản đa luồng, ta có thể có pseudocode như vầy:

```
int x = 0, y = 0;

thread1() {
    x = 2;
    y = 3;
}

thread2() {
    while (y != 3) {}  // spin
    printf("x is now %d\n", x);  // 2? ...or 0?
}
```

Output của thread 2 là gì?

Nếu `x` được gán `2` *trước* khi `y` được gán `3`, tôi kỳ vọng output rất hợp lý là:

```
x is now 2
```

Nhưng cái gì đó lén lút có thể sắp xếp lại dòng 4 và 5 làm ta thấy giá trị `0` của `x` khi in.

Nói cách khác, mọi thứ không chắc chắn trừ khi ta có thể bằng cách nào đó nói, "Tại điểm này, tôi kỳ vọng tất cả các lần ghi trước đó trong thread khác đều thấy được trong thread này."

Hai thread *đồng bộ* khi chúng thống nhất về trạng thái bộ nhớ chia sẻ. Như đã thấy, chúng không phải luôn đồng ý với code. Vậy chúng đồng ý cách nào?

Dùng biến atomic có thể ép sự đồng ý đó[3]. Nếu một thread ghi vào biến atomic, nó đang nói "ai đọc biến atomic này trong tương lai cũng sẽ thấy tất cả thay đổi tôi đã làm với bộ nhớ (atomic hay không) cho đến và bao gồm biến atomic này".

Hay theo kiểu người hơn, cùng ngồi quanh bàn họp và bảo đảm rằng ta cùng chung một trang về các mảnh bộ nhớ chia sẻ nào giữ giá trị nào. Bạn đồng ý rằng các thay đổi bộ nhớ bạn đã làm cho đến và bao gồm lần store atomic sẽ thấy được với tôi sau khi tôi load cùng biến atomic đó.

Nên ta có thể dễ dàng sửa ví dụ:

```
int x = 0;
atomic int y = 0;  // Make y atomic

thread1() {
    x = 2;
    y = 3;             // Synchronize on write
}

thread2() {
    while (y != 3) {}  // Synchronize on read
    printf("x is now %d\n", x);  // 2, period.
}
```

Vì các thread đồng bộ qua `y`, tất cả các lần ghi trong thread 1 xảy ra *trước* lần ghi vào `y` đều thấy được trong thread 2 *sau* lần đọc từ `y` (trong `while` loop).

[3]Cho đến khi tôi nói khác, tôi đang nói chung về các thao tác *sequentially consistent*. Nói thêm ý nghĩa của nó sớm thôi.

Quan trọng chú ý vài điều ở đây:

1. Không có gì ngủ. Synchronization không phải thao tác blocking. Cả hai thread chạy hết ga đến khi thoát. Thậm chí cái bị kẹt trong spin loop cũng không block ai khác khỏi chạy.
2. Synchronization xảy ra khi một thread đọc biến atomic mà thread khác đã ghi. Nên khi thread 2 đọc `y`, tất cả lần ghi bộ nhớ trước trong thread 1 (cụ thể là set `x`) sẽ thấy được trong thread 2.
3. Chú ý `x` không atomic. Vẫn OK vì ta không đang đồng bộ qua `x`, và việc đồng bộ qua `y` khi ta ghi nó trong thread 1 nghĩa là tất cả lần ghi trước, bao gồm `x`, trong thread 1 sẽ thấy được với các thread khác... nếu các thread đó đọc `y` để đồng bộ.

Ép synchronization này kém hiệu quả và có thể chậm hơn nhiều so với dùng biến thường. Đây là lý do ta không dùng atomic trừ khi phải dùng cho ứng dụng cụ thể.

Đó là cơ bản. Xem sâu hơn nào.

40.4 Acquire và Release

Thêm thuật ngữ! Học giờ thì có lợi sau.

Khi một thread đọc biến atomic, đó được gọi là thao tác *acquire*.

Khi một thread ghi biến atomic, đó được gọi là thao tác *release*.

Những cái này là gì? Xếp chúng vào các thuật ngữ bạn đã biết về biến atomic:

Read = Load = Acquire. Như khi bạn so sánh biến atomic hay đọc nó để copy sang giá trị khác.

Write = Store = Release. Như khi bạn gán giá trị vào biến atomic.

Khi dùng biến atomic với ngữ nghĩa acquire/release, C nêu rõ chuyện gì có thể xảy ra khi nào.

Acquire/release tạo cơ sở cho synchronization ta vừa nói.

Khi một thread acquire biến atomic, nó có thể thấy giá trị đã set trong thread khác đã release cùng biến đó.

Nói cách khác:

Khi một thread đọc biến atomic, nó có thể thấy giá trị đã set trong thread khác đã ghi cùng biến đó.

Synchronization xảy ra qua cặp acquire/release.

Chi tiết thêm:

Với read/load/acquire một biến atomic cụ thể:

- Tất cả lần ghi (atomic hay không) trong thread khác xảy ra trước khi thread đó write/store/release biến atomic này giờ thấy được trong thread này.
- Giá trị mới của biến atomic do thread khác set cũng thấy được trong thread này.
- Không có lần đọc hay ghi biến/bộ nhớ nào trong thread hiện tại có thể bị sắp xếp lại xảy ra trước acquire này.
- Acquire đóng vai rào chắn một chiều khi sắp xếp lại code; các lần đọc và ghi trong thread hiện tại có thể bị di chuyển xuống từ *trước* acquire thành *sau* nó. Nhưng quan trọng hơn với synchronization, không gì có thể di chuyển lên từ *sau* acquire thành *trước* nó.

Với write/store/release một biến atomic cụ thể:

- Tất cả lần ghi (atomic hay không) trong thread hiện tại xảy ra trước release này trở nên thấy được với các thread khác đã read/load/acquire cùng biến atomic.
- Giá trị thread này ghi vào biến atomic này cũng thấy được với các thread khác.
- Không có lần đọc hay ghi biến/bộ nhớ nào trong thread hiện tại có thể bị sắp xếp lại xảy ra sau release này.

- Release đóng vai rào chắn một chiều khi sắp xếp lại code: các lần đọc và ghi trong thread hiện tại có thể bị di chuyển lên từ *sau* release thành *trước* nó. Nhưng quan trọng hơn với synchronization, không gì có thể di chuyển xuống từ *trước* release thành *sau* nó.

Lại nữa, kết quả là synchronization bộ nhớ từ thread này sang thread khác. Thread thứ hai có thể chắc chắn rằng biến và bộ nhớ được ghi theo thứ tự lập trình viên mong muốn.

```
int x, y, z = 0;
atomic_int a = 0;

thread1() {
    x = 10;
    y = 20;
    a = 999;  // Release
    z = 30;
}

thread2()
{
    while (a != 999) { } // Acquire

    assert(x == 10);  // never asserts, x is always 10
    assert(y == 20);  // never asserts, y is always 20

    assert(z == 0);  // might assert!!
}
```

Trong ví dụ trên, `thread2` có thể chắc chắn về giá trị của `x` và `y` sau khi nó acquire `a` vì chúng được set trước khi `thread1` release atomic `a`.

Nhưng `thread2` không thể chắc chắn về giá trị `z` vì nó xảy ra sau release. Có thể việc gán cho `z` bị di chuyển lên trước việc gán cho `a`.

Chú ý quan trọng: release một biến atomic không có tác dụng lên acquire các biến atomic khác. Mỗi biến cô lập với các biến khác.

40.5 Sequential Consistency

Bạn vẫn còn trụ được chứ? Ta đã qua phần nội dung chính của cách dùng atomic đơn giản hơn. Và vì ta không định nói về các cách dùng phức tạp hơn ở đây, bạn có thể thư giãn chút.

Sequential consistency là cái gọi là *memory ordering*. Có nhiều memory ordering, nhưng sequential consistency là tỉnh táo nhất[4] mà C có. Nó cũng là mặc định. Bạn phải cố tình để dùng các memory ordering khác.

Tất cả thứ ta đã nói từ đầu đến giờ đều xảy ra trong lãnh địa của sequential consistency.

Ta đã nói về cách compiler hay CPU có thể sắp xếp lại lần đọc và ghi bộ nhớ trong một thread miễn là tuân theo quy tắc *as-if*.

Và ta đã thấy cách phanh hành vi này bằng cách đồng bộ qua biến atomic.

Hãy chính thức hoá thêm chút.

Nếu các thao tác là *sequentially consistent*, nghĩa là cuối ngày, khi mọi thứ đã nói xong, tất cả các thread có thể gác chân, mở đồ uống yêu thích, và đồng ý về thứ tự các thay đổi bộ nhớ xảy ra trong lần chạy. Và thứ tự đó là cái được quy định bởi code.

Một cái sẽ không nói, "Nhưng chẳng phải *B* xảy ra trước *A* sao?" nếu các cái khác nói, "*A* chắc chắn xảy ra trước *B*". Tất cả đều bạn bè nhau ở đây.

[4] Tỉnh táo nhất từ góc nhìn của lập trình viên.

Đặc biệt, trong một thread, không có acquire và release nào có thể bị sắp xếp lại so với nhau. Cái này thêm vào các quy tắc về những truy cập bộ nhớ khác có thể bị sắp xếp lại quanh chúng.

Quy tắc này cho thêm một cấp độ tinh táo cho tiến trình các load/ acquire và store/release atomic.

Mọi memory order khác trong C đều liên quan việc nới lỏng các quy tắc sắp xếp lại, cho acquires/releases hoặc cho các truy cập bộ nhớ khác, atomic hay không. Bạn sẽ làm vậy nếu *thực sự* biết mình đang làm gì và cần tăng tốc. *Đây là đất của đội quân rồng...*

Nói thêm sau, nhưng giờ cứ dùng cái an toàn thực dụng.

40.6 Gán Atomic và các Toán tử

Một số toán tử trên biến atomic là atomic. Và những cái khác thì không.

Hãy bắt đầu với một phản ví dụ:

```
atomic_int x = 0;

thread1() {
    x = x + 3;  // NOT atomic!
}
```

Vì có lần đọc `x` ở bên phải phép gán và lần ghi hiệu quả ở bên trái, đây là hai thao tác. Một thread khác có thể chen vào giữa và làm bạn phật lòng.

Nhưng bạn *có thể* dùng shorthand `+=` để được thao tác atomic:

```
atomic_int x = 0;

thread1() {
    x += 3;   // ATOMIC!
}
```

Trong trường hợp đó, `x` sẽ được tăng atomic thêm `3`, không thread nào khác có thể nhảy vào giữa.

Đặc biệt, các toán tử sau là thao tác atomic read-modify-write với sequential consistency, nên cứ dùng thoải mái trong niềm vui. (Trong ví dụ, `a` là atomic.)

```
a++       a--       --a       ++a
a += b    a -= b    a *= b    a /= b    a %= b
a &= b    a |= b    a ^= b    a >>= b   a <<= b
```

40.7 Các hàm thư viện tự đồng bộ

Đến giờ ta đã nói cách đồng bộ với biến atomic, nhưng hoá ra có vài hàm thư viện tự làm đồng bộ hạn chế sau cánh gà.

```
call_once()      thrd_create()       thrd_join()
mtx_lock()       mtx_timedlock()     mtx_trylock()
malloc()         calloc()            realloc()
aligned_alloc()
```

`call_once()`: Đồng bộ với tất cả các lần gọi tiếp theo tới `call_once()` cho một flag cụ thể. Cách này các lần gọi tiếp theo có thể yên tâm rằng nếu thread khác set flag, chúng sẽ thấy.

`thrd_create()`: Đồng bộ với phần đầu của thread mới. Thread mới có thể chắc chắn nó sẽ thấy tất cả các lần ghi bộ nhớ chia sẻ từ thread cha trước khi gọi `thrd_create()`.

`thrd_join()`: Khi một thread chết, nó đồng bộ với hàm này. Thread đã gọi `thrd_join()` có thể yên tâm rằng nó có thể thấy tất cả các lần ghi chia sẻ của thread đã chết.

`mtx_lock()`: Các lần gọi trước tới `mtx_unlock()` trên cùng mutex đồng bộ với lần gọi này. Đây là trường hợp phản chiếu nhiều nhất tiến trình acquire/release ta đã nói. `mtx_unlock()` thực hiện release trên biến mutex, bảo đảm bất kỳ thread sau nào acquire với `mtx_lock()` có thể thấy tất cả thay đổi bộ nhớ chia sẻ trong critical section.

`mtx_timedlock()` và **`mtx_trylock()`**: Tương tự tình huống với `mtx_lock()`, nếu lần gọi này thành công, các lần gọi trước tới `mtx_unlock()` đồng bộ với cái này.

Các hàm bộ nhớ động: nếu bạn cấp phát bộ nhớ, nó đồng bộ với lần giải phóng trước đó của cùng bộ nhớ. Và các lần cấp phát và giải phóng vùng bộ nhớ đó xảy ra theo một thứ tự tổng thể duy nhất mà tất cả thread có thể đồng ý. Tôi *nghĩ* ý tưởng ở đây là lần giải phóng có thể xoá sạch vùng nếu nó chọn, và ta muốn chắc rằng lần cấp phát sau không thấy dữ liệu không bị xoá. Ai đó báo tôi biết nếu còn gì khác.

40.8 Bộ chỉ định kiểu Atomic, Qualifier

Hạ một chút xem ta có các kiểu nào sẵn, và làm sao tạo kiểu atomic mới.

Đầu tiên, xem các kiểu atomic có sẵn và chúng được `typedef` tới cái gì. (Spoiler: `_Atomic` là một type qualifier!)

Kiểu Atomic	Dạng dài tương đương
`atomic_bool`	`_Atomic _Bool`
`atomic_char`	`_Atomic char`
`atomic_schar`	`_Atomic signed char`
`atomic_uchar`	`_Atomic unsigned char`
`atomic_short`	`_Atomic short`
`atomic_ushort`	`_Atomic unsigned short`
`atomic_int`	`_Atomic int`
`atomic_uint`	`_Atomic unsigned int`
`atomic_long`	`_Atomic long`
`atomic_ulong`	`_Atomic unsigned long`
`atomic_llong`	`_Atomic long long`
`atomic_ullong`	`_Atomic unsigned long long`
`atomic_char16_t`	`_Atomic char16_t`
`atomic_char32_t`	`_Atomic char32_t`
`atomic_wchar_t`	`_Atomic wchar_t`
`atomic_int_least8_t`	`_Atomic int_least8_t`
`atomic_uint_least8_t`	`_Atomic uint_least8_t`
`atomic_int_least16_t`	`_Atomic int_least16_t`
`atomic_uint_least16_t`	`_Atomic uint_least16_t`
`atomic_int_least32_t`	`_Atomic int_least32_t`
`atomic_uint_least32_t`	`_Atomic uint_least32_t`
`atomic_int_least64_t`	`_Atomic int_least64_t`
`atomic_uint_least64_t`	`_Atomic uint_least64_t`
`atomic_int_fast8_t`	`_Atomic int_fast8_t`
`atomic_uint_fast8_t`	`_Atomic uint_fast8_t`
`atomic_int_fast16_t`	`_Atomic int_fast16_t`
`atomic_uint_fast16_t`	`_Atomic uint_fast16_t`
`atomic_int_fast32_t`	`_Atomic int_fast32_t`
`atomic_uint_fast32_t`	`_Atomic uint_fast32_t`

Kiểu Atomic	Dạng dài tương đương
`atomic_int_fast64_t`	`_Atomic int_fast64_t`
`atomic_uint_fast64_t`	`_Atomic uint_fast64_t`
`atomic_intptr_t`	`_Atomic intptr_t`
`atomic_uintptr_t`	`_Atomic uintptr_t`
`atomic_size_t`	`_Atomic size_t`
`atomic_ptrdiff_t`	`_Atomic ptrdiff_t`
`atomic_intmax_t`	`_Atomic intmax_t`
`atomic_uintmax_t`	`_Atomic uintmax_t`

Dùng chúng thoải mái! Chúng nhất quán với các alias atomic trong C++, nếu điều đó có ích.

Nhưng nếu bạn muốn nhiều hơn?

Bạn có thể làm với type qualifier hoặc type specifier.

Đầu tiên, specifier! Đó là từ khoá `_Atomic` với kiểu trong ngoặc sau[5], phù hợp dùng với `typedef`:

```
typedef _Atomic(double) atomic_double;

atomic_double f;
```

Hạn chế với specifier: kiểu bạn đang làm atomic không thể là kiểu mảng hay hàm, cũng không thể là atomic hay đã qualified kiểu khác.

Tiếp, qualifier! Đó là từ khoá `_Atomic` *không* có kiểu trong ngoặc sau.

Nên hai cái này làm việc tương tự[6]:

```
_Atomic(int) i;   // type specifier
_Atomic int  j;   // type qualifier
```

Điểm khác là bạn có thể include type qualifier khác với cái sau:

```
_Atomic volatile int k;   // qualified atomic variable
```

Hạn chế với qualifier: kiểu bạn đang làm atomic không thể là kiểu mảng hay hàm.

40.9 Biến Atomic Lock-Free

Kiến trúc phần cứng bị hạn chế về lượng dữ liệu có thể atomic đọc và ghi. Tuỳ vào cách nó được kết nối. Và khác nhau.

Nếu bạn dùng kiểu atomic, bạn có thể yên tâm rằng truy cập kiểu đó sẽ atomic... nhưng có một điều: nếu phần cứng không làm được, nó được làm bằng lock thay.

Nên truy cập atomic trở thành lock-access-unlock, chậm hơn khá và có một số ngụ ý với signal handler.

Atomic flags bên dưới là kiểu atomic duy nhất được đảm bảo lock-free trong tất cả implementation tuân chuẩn. Trong thế giới desktop/laptop máy tính thông thường, các kiểu lớn khác có thể cũng lock-free.

May thay, ta có vài cách để xác định liệu kiểu cụ thể có phải atomic lock-free hay không.

Trước hết, vài macro, bạn có thể dùng ở compile time với `#if`. Chúng áp dụng cho cả kiểu signed lẫn unsigned.

[5] Có vẻ C++23 thêm cái này như một macro.
[6] Spec lưu ý chúng có thể khác về kích thước, biểu diễn, và căn chỉnh.

Kiểu Atomic	Macro Lock Free
`atomic_bool`	`ATOMIC_BOOL_LOCK_FREE`
`atomic_char`	`ATOMIC_CHAR_LOCK_FREE`
`atomic_char16_t`	`ATOMIC_CHAR16_T_LOCK_FREE`
`atomic_char32_t`	`ATOMIC_CHAR32_T_LOCK_FREE`
`atomic_wchar_t`	`ATOMIC_WCHAR_T_LOCK_FREE`
`atomic_short`	`ATOMIC_SHORT_LOCK_FREE`
`atomic_int`	`ATOMIC_INT_LOCK_FREE`
`atomic_long`	`ATOMIC_LONG_LOCK_FREE`
`atomic_llong`	`ATOMIC_LLONG_LOCK_FREE`
`atomic_intptr_t`	`ATOMIC_POINTER_LOCK_FREE`

Các macro này thú vị có thể có *ba* giá trị khác nhau:

Giá trị	Ý nghĩa
`0`	Không bao giờ lock-free.
`1`	*Đôi khi* lock-free.
`2`	Luôn lock-free.

Khoan, cái gì đó *đôi khi* lock-free được là sao? Nghĩa là đáp án không biết tại compile-time, nhưng có thể biết sau tại runtime. Có thể đáp án khác tuỳ bạn đang chạy code trên Genuine Intel hay AMD hay gì đó[7].

Nhưng bạn luôn có thể test tại runtime với hàm `atomic_is_lock_free()`. Hàm này trả về true hay false nếu kiểu cụ thể là atomic ngay bây giờ.

Tại sao ta quan tâm?

Lock-free nhanh hơn, nên có thể có vấn đề tốc độ bạn muốn code tránh theo cách khác. Hoặc có thể bạn cần dùng biến atomic trong signal handler.

40.9.1 Signal Handlers và Atomic Lock-Free

Nếu bạn đọc hay ghi biến chia sẻ (thời lượng lưu trữ static hay `_Thread_Local`) trong signal handler, đó là hành vi không xác định [gasp!]... Trừ khi bạn làm một trong các điều sau:

1. Ghi vào biến kiểu `volatile sig_atomic_t`.
2. Đọc hay ghi biến atomic lock-free.

Theo tôi thấy, biến atomic lock-free là một trong số ít cách portable lấy thông tin ra khỏi signal handler.

Spec hơi mơ hồ, theo cách tôi đọc, về memory order khi acquire hay release biến atomic trong signal handler. C++ nói, và hợp lý, rằng các truy cập đó là không tuần tự so với phần còn lại của chương trình[8]. Signal có thể được raise bất cứ lúc nào. Nên tôi giả định hành vi của C tương tự.

40.10 Atomic Flags

Chỉ có một kiểu mà chuẩn đảm bảo sẽ là lock-free atomic: `atomic_flag`. Đây là kiểu mờ (opaque) cho các thao tác test-and-set[9].

Nó có thể là *set* hoặc *clear*. Bạn có thể khởi tạo nó thành clear với:

[7] Tôi chỉ lấy ví dụ đó từ không khí. Có thể không quan trọng trên Intel/AMD, nhưng có thể quan trọng ở đâu đó đấy!
[8] C++ nói thêm nếu signal là kết quả của lần gọi `raise()`, nó tuần tự *sau* `raise()`.
[9] https://en.wikipedia.org/wiki/Test-and-set

```
atomic_flag f = ATOMIC_FLAG_INIT;
```

Bạn có thể set flag atomic với `atomic_flag_test_and_set()`, sẽ set flag và trả về trạng thái trước của nó dưới dạng `_Bool` (true cho set).

Bạn có thể clear flag atomic với `atomic_flag_clear()`.

Đây là ví dụ ta init flag thành clear, set hai lần, rồi clear lại.

```
#include <stdio.h>
#include <stdbool.h>   // Not needed in C23
#include <stdatomic.h>

atomic_flag f = ATOMIC_FLAG_INIT;

int main(void)
{
    bool r = atomic_flag_test_and_set(&f);
    printf("Value was: %d\n", r);           // 0

    r = atomic_flag_test_and_set(&f);
    printf("Value was: %d\n", r);           // 1

    atomic_flag_clear(&f);
    r = atomic_flag_test_and_set(&f);
    printf("Value was: %d\n", r);           // 0
}
```

40.11 `struct` và `union` Atomic

Dùng qualifier hay specifier `_Atomic`, bạn có thể tạo `struct` hay `union` atomic! Khá đáng kinh ngạc.

Nếu không có nhiều dữ liệu bên trong (tức là vài byte), kiểu atomic tạo ra có thể lock-free. Test bằng `atomic_is_lock_free()`.

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

int main(void)
{
    struct point {
        float x, y;
    };

    _Atomic(struct point) p;

    printf("Is lock free: %d\n", atomic_is_lock_free(&p));
}
```

Đây là cái bắt: bạn không thể truy cập field của `struct` hay `union` atomic... nên có ý nghĩa gì? À, bạn có thể atomic *copy* toàn bộ `struct` vào biến không-atomic rồi dùng. Bạn cũng có thể atomic copy ngược lại.

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

```
int main(void)
{
    struct point {
        float x, y;
    };

    _Atomic(struct point) p;
    struct point t;

    p = (struct point){1, 2};  // Atomic copy

    //printf("%f\n", p.x);  // Error

    t = p;   // Atomic copy

    printf("%f\n", t.x);  // OK!
}
```

Bạn cũng có thể khai báo `struct` mà các field riêng lẻ là atomic. Là implementation-defined xem kiểu atomic có được phép trên bitfield hay không.

40.12 Con trỏ Atomic

Chỉ ghi chú ở đây về vị trí `_Atomic` khi nói tới con trỏ.

Đầu tiên, con trỏ tới atomic (tức là giá trị con trỏ không atomic, nhưng thứ nó trỏ tới thì atomic):

```
_Atomic int x;
_Atomic int *p;  // p is a pointer to an atomic int

p = &x;  // OK!
```

Thứ hai, con trỏ atomic tới giá trị không-atomic (tức là giá trị con trỏ tự thân atomic, nhưng thứ nó trỏ tới thì không):

```
int x;
int * _Atomic p;  // p is an atomic pointer to an int

p = &x;  // OK!
```

Cuối cùng, con trỏ atomic tới giá trị atomic (tức là con trỏ và thứ nó trỏ tới đều atomic):

```
_Atomic int x;
_Atomic int * _Atomic p;  // p is an atomic pointer to an atomic int

p = &x;  // OK!
```

40.13 Memory Order

Ta đã nói về sequential consistency, cái hợp lý trong nhóm. Nhưng còn một số cái khác:

`memory_order`	Mô tả
`memory_order_seq_cst`	Sequential Consistency
`memory_order_acq_rel`	Acquire/Release
`memory_order_release`	Release

memory_order	Mô tả
memory_order_acquire	Acquire
memory_order_consume	Consume
memory_order_relaxed	Relaxed

Bạn có thể chỉ định các cái khác với một số hàm thư viện. Ví dụ, bạn có thể cộng giá trị vào biến atomic như vầy:

```
atomic_int x = 0;

x += 5;  // Sequential consistency, the default
```

Hay bạn có thể làm tương tự với hàm thư viện này:

```
atomic_int x = 0;

atomic_fetch_add(&x, 5);  // Sequential consistency, the default
```

Hay bạn có thể làm tương tự với memory ordering tường minh:

```
atomic_int x = 0;

atomic_fetch_add_explicit(&x, 5, memory_order_seq_cst);
```

Nhưng nếu ta không muốn sequential consistency? Và muốn acquire/release thay vào đó vì lý do gì đó? Cứ gọi tên nó:

```
atomic_int x = 0;

atomic_fetch_add_explicit(&x, 5, memory_order_acq_rel);
```

Ta sẽ chia nhỏ các memory order khác bên dưới. Đừng nghịch bất kỳ cái gì khác ngoài sequential consistency trừ khi bạn biết đang làm gì. Rất dễ mắc lỗi gây ra các failure hiếm, khó tái hiện.

40.13.1 Sequential Consistency

- Thao tác Load acquire (xem bên dưới).
- Thao tác Store release (xem bên dưới).
- Thao tác Read-modify-write acquire rồi release.

Cũng vậy, để duy trì tổng thứ tự của acquire và release, không có acquire hay release nào bị sắp xếp lại so với nhau. (Quy tắc acquire/release không cấm sắp xếp lại một release theo sau là acquire. Nhưng quy tắc sequentially consistent thì cấm.)

40.13.2 Acquire

Đây là chuyện xảy ra trên thao tác load/read một biến atomic.

- Nếu thread khác đã release biến atomic này, tất cả các lần ghi thread đó làm giờ thấy được trong thread này.
- Các truy cập bộ nhớ trong thread này xảy ra sau lần load này không thể bị sắp xếp lại trước nó.

40.13.3 Release

Đây là chuyện xảy ra trên store/write một biến atomic.

- Nếu thread khác sau này acquire biến atomic này, tất cả các lần ghi bộ nhớ trong thread này trước lần ghi atomic của nó trở nên thấy được với thread khác đó.
- Các truy cập bộ nhớ trong thread này xảy ra trước release không thể bị sắp xếp lại sau nó.

40.13.4 Consume

Cái này hơi lạ, tương tự phiên bản ít nghiêm khắc hơn của acquire. Nó ảnh hưởng các truy cập bộ nhớ *phụ thuộc dữ liệu* vào biến atomic.

"Phụ thuộc dữ liệu" mơ hồ nghĩa là biến atomic được dùng trong một phép tính.

Tức là nếu một thread consume biến atomic thì tất cả các thao tác trong thread đó tiếp tục dùng biến atomic đó sẽ có thể thấy các lần ghi bộ nhớ trong thread đang release.

So với acquire nơi các lần ghi bộ nhớ trong thread đang release sẽ thấy được với *tất cả* các thao tác trong thread hiện tại, không chỉ những cái phụ thuộc dữ liệu.

Cũng giống acquire, có hạn chế về thao tác nào có thể bị sắp xếp lại *trước* consume. Với acquire, bạn không thể sắp xếp lại bất cứ gì trước nó. Với consume, bạn không thể sắp xếp lại bất cứ gì phụ thuộc giá trị atomic đã load trước nó.

40.13.5 Acquire/Release

Cái này chỉ áp dụng cho thao tác read-modify-write. Là một acquire và release gom vào một.

- Acquire xảy ra cho lần read.
- Release xảy ra cho lần write.

40.13.6 Relaxed

Không có quy tắc; là hỗn loạn! Ai cũng có thể sắp xếp lại mọi thứ mọi nơi! Chó với mèo sống chung, loạn lớn!

Thực ra có một quy tắc. Lần đọc và ghi atomic vẫn là all-or-nothing. Nhưng các thao tác có thể bị sắp xếp lại tuỳ hứng và không có synchronization giữa các thread.

Có vài use case cho memory order này, bạn có thể tìm với một ít tìm kiếm, ví dụ các counter đơn giản.

Và bạn có thể dùng fence để ép synchronization sau một loạt lần ghi relaxed.

40.14 Fences

Bạn biết cách release và acquire biến atomic xảy ra khi bạn đọc và ghi chúng đúng không?

Thì ra cũng có thể làm release hay acquire mà *không* có biến atomic.

Cái này gọi là *fence*. Nên nếu bạn muốn tất cả các lần ghi trong một thread thấy được ở nơi khác, bạn có thể đặt release fence trong một thread và acquire fence trong thread khác, giống cách biến atomic hoạt động.

Vì thao tác consume không thực sự có nghĩa trên fence[10], `memory_order_consume` được xử lý như acquire.

Bạn có thể đặt fence với bất kỳ order nào được chỉ định:

```
atomic_thread_fence(memory_order_release);
```

Còn có phiên bản fence nhẹ để dùng với signal handler, gọi là `atomic_signal_fence()`.

Nó hoạt động y như `atomic_thread_fence()`, trừ:

[10] Vì consume là về các thao tác phụ thuộc giá trị biến atomic đã acquire, và không có biến atomic với fence.

- Nó chỉ liên quan khả năng thấy giá trị trong cùng thread; không có synchronization với thread khác.
- Không phát ra lệnh fence phần cứng.

Nếu bạn muốn chắc rằng side effect của thao tác không-atomic (và thao tác atomic relaxed) thấy được trong signal handler, bạn có thể dùng fence này.

Ý tưởng là signal handler đang thực thi trong *thread này*, không phải thread khác, nên đây là cách nhẹ hơn để đảm bảo thay đổi bên ngoài signal handler thấy được bên trong nó (tức là chúng không bị sắp xếp lại).

40.15 Tham khảo

Nếu bạn muốn học thêm về mấy thứ này, đây là một số thứ đã giúp tôi cày qua nó:

- Herb Sutter's `atomic<>` *Weapons* talk:
  - Part 1[11]
  - part 2[12]
- Jeff Preshing's materials[13], đặc biệt:
  - An Introduction to Lock-Free Programming[14]
  - Acquire and Release Semantics[15]
  - The *Happens-Before* Relation[16]
  - The *Synchronizes-With* Relation[17]
  - The Purpose of `memory_order_consume` in C++11[18]
  - You Can Do Any Kind of Atomic Read-Modify-Write Operation[19]
- CPPReference:
  - Memory Order[20]
  - Atomic Types[21]
- Bruce Dawson's Lockless Programming Considerations[22]
- Những người nhiệt tình và am hiểu trên r/C_Programming[23]

[11] https://www.youtube.com/watch?v=A8eCGOqgvH4
[12] https://www.youtube.com/watch?v=KeLBd2EJLOU
[13] https://preshing.com/archives/
[14] https://preshing.com/20120612/an-introduction-to-lock-free-programming/
[15] https://preshing.com/20120913/acquire-and-release-semantics/
[16] https://preshing.com/20130702/the-happens-before-relation/
[17] https://preshing.com/20130823/the-synchronizes-with-relation/
[18] https://preshing.com/20140709/the-purpose-of-memory_order_consume-in-cpp11/
[19] https://preshing.com/20150402/you-can-do-any-kind-of-atomic-read-modify-write-operation/
[20] https://en.cppreference.com/w/c/atomic/memory_order
[21] https://en.cppreference.com/w/c/language/atomic
[22] https://docs.microsoft.com/en-us/windows/win32/dxtecharts/lockless-programming
[23] https://www.reddit.com/r/C_Programming/

Chapter 41

Function Specifier, Alignment Specifier/Operator

Theo kinh nghiệm của tôi, mấy thứ này không được dùng nhiều lắm, nhưng cứ trình bày cho đủ.

41.1 Function Specifier

Khi bạn khai báo một hàm, bạn có thể cho compiler vài gợi ý về cách hàm đó có thể hay sẽ được dùng. Điều này cho phép hoặc khuyến khích compiler thực hiện một số tối ưu hoá.

41.1.1 `inline` để tăng tốc, có lẽ

Bạn có thể khai báo hàm là inline như vầy:

```
static inline int add(int x, int y) {
    return x + y;
}
```

Ý nghĩa là khuyến khích compiler làm lời gọi hàm này nhanh nhất có thể. Và trong lịch sử, một cách để làm điều đó là *inlining*, tức là thân hàm sẽ được nhúng nguyên vẹn tại nơi gọi. Cái này tránh tất cả overhead set up lời gọi hàm và tháo dỡ nó, đổi lại kích thước code lớn hơn vì hàm được copy khắp nơi thay vì tái sử dụng.

Những điều nhanh-gọn cần nhớ:

1. Bạn có lẽ không cần dùng `inline` để tăng tốc. Compiler hiện đại biết cái gì tốt nhất.
2. Nếu bạn dùng nó để tăng tốc, dùng với phạm vi file, tức là `static inline`. Cái này tránh quy tắc lộn xộn của external linkage và hàm inline.

Đừng đọc phần này nữa.

Kẻ thèm bị trừng phạt hả?

Thử bỏ `static` đi nào.

```
#include <stdio.h>

inline int add(int x, int y)
{
    return x + y;
}

int main(void)
```

```
{
    printf("%d\n", add(1, 2));
}
```

`gcc` báo lỗi linker trên `add()` [1]. Spec yêu cầu nếu bạn có một hàm inline không-`extern` thì bạn cũng phải cung cấp một phiên bản có external linkage.

Nên bạn sẽ phải có phiên bản `extern` ở đâu đó khác để cái này chạy. Nếu compiler có cả hàm `inline` trong file hiện tại và phiên bản external của cùng hàm ở nơi khác, nó được chọn gọi cái nào. Nên tôi khuyên mạnh là chúng giống nhau.

Một cách khác bạn có thể làm là khai báo hàm là `extern inline`. Cái này sẽ thử inline trong cùng file (để tăng tốc), nhưng cũng tạo phiên bản có external linkage.

41.1.2 `noreturn` và `_Noreturn`

Cái này báo cho compiler rằng một hàm cụ thể sẽ không bao giờ return về chỗ gọi, tức là chương trình sẽ thoát bằng cơ chế nào đó trước khi hàm return.

Nó cho phép compiler có thể thực hiện một số tối ưu quanh lời gọi hàm.

Nó cũng cho phép bạn báo cho các dev khác rằng có logic chương trình phụ thuộc vào một hàm *không* return.

Có lẽ bạn sẽ không bao giờ cần dùng cái này, nhưng bạn sẽ thấy nó trên một số lời gọi thư viện như `exit()` [2] và `abort()` [3].

Từ khoá có sẵn là `_Noreturn`, nhưng nếu nó không làm hỏng code sẵn có của bạn, mọi người đều khuyên include `<stdnoreturn.h>` và dùng `noreturn` dễ đọc hơn.

Là hành vi không xác định nếu một hàm được chỉ định là `noreturn` thực sự return. Cái đó không trung thực về mặt tính toán, thấy đó.

Đây là ví dụ dùng `noreturn` đúng:

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

noreturn void foo(void) // This function should never return!
{
    printf("Happy days\n");

    exit(1);            // And it doesn't return--it exits here!
}

int main(void)
{
    foo();
}
```

Nếu compiler phát hiện một hàm `noreturn` có thể return, nó có thể cảnh báo bạn, hữu ích.

Thay thế hàm `foo()` bằng cái này:

```
noreturn void foo(void)
{
```

[1] Trừ khi bạn compile có bật optimization (có lẽ)! Nhưng tôi nghĩ khi nó làm vậy, nó không tuân spec.
[2] https://beej.us/guide/bgclr/html/split/stdlib.html#man-exit
[3] https://beej.us/guide/bgclr/html/split/stdlib.html#man-abort

```
    printf("Breakin' the law\n");
}
```

cho tôi một cảnh báo:

```
foo.c:7:1: warning: function declared 'noreturn' should not return
```

41.2 Alignment Specifier và Operator

Alignment[4] là về các bội số của địa chỉ mà các đối tượng có thể được lưu. Bạn có thể lưu cái này ở địa chỉ bất kỳ? Hay phải là địa chỉ bắt đầu chia hết cho 2? Hay 8? Hay 16?

Nếu bạn đang code thứ gì đó thấp cấp như bộ cấp phát bộ nhớ giao tiếp với OS, bạn có thể cần nghĩ đến điều này. Phần lớn dev đi hết sự nghiệp mà không dùng chức năng này trong C.

41.2.1 `alignas` và `_Alignas`

Cái này không phải hàm. Đây là một *alignment specifier* bạn có thể dùng với một khai báo biến.

Specifier có sẵn là `_Alignas`, nhưng header `<stdalign.h>` định nghĩa nó là `alignas` cho đẹp hơn.

Nếu bạn cần `char` của mình được căn chỉnh như `int`, bạn có thể ép như vầy khi khai báo:

```
char alignas(int) c;
```

Bạn cũng có thể truyền giá trị hằng hay biểu thức vào làm alignment. Cái này phải là thứ được hệ thống hỗ trợ, nhưng spec ngừng trước chuyện quy định bạn có thể đưa giá trị nào vào. Các luỹ thừa nhỏ của 2 (1, 2, 4, 8, và 16) nhìn chung là đặt cược an toàn.

```
char alignas(8) c;   // align on 8-byte boundaries
```

Nếu bạn muốn căn chỉnh ở alignment lớn nhất hệ thống bạn dùng, include `<stddef.h>` và dùng kiểu `max_align_t`, như sau:

```
char alignas(max_align_t) c;
```

Bạn có thể *over-align* bằng cách chỉ định alignment lớn hơn của `max_align_t`, nhưng chuyện đó có được phép hay không phụ thuộc hệ thống.

41.2.2 `alignof` và `_Alignof`

Toán tử này sẽ trả về bội số địa chỉ mà một kiểu cụ thể dùng cho alignment trên hệ thống này. Ví dụ, có thể `char` được căn chỉnh mỗi 1 địa chỉ, và `int` được căn chỉnh mỗi 4 địa chỉ.

Toán tử có sẵn là `_Alignof`, nhưng header `<stdalign.h>` định nghĩa nó là `alignof` nếu bạn muốn trông chất hơn.

Đây là chương trình sẽ in ra alignment của nhiều kiểu khác nhau. Lại nữa, chúng sẽ thay đổi từ hệ thống này sang hệ thống khác. Chú ý kiểu `max_align_t` sẽ cho bạn alignment lớn nhất hệ thống dùng.

```
#include <stdalign.h>
#include <stdio.h>     // for printf()
#include <stddef.h>    // for max_align_t

struct t {
```

[4] https://en.wikipedia.org/wiki/Data_structure_alignment

```
    int a;
    char b;
    float c;
};

int main(void)
{
    printf("char       : %zu\n", alignof(char));
    printf("short      : %zu\n", alignof(short));
    printf("int        : %zu\n", alignof(int));
    printf("long       : %zu\n", alignof(long));
    printf("long long  : %zu\n", alignof(long long));
    printf("double     : %zu\n", alignof(double));
    printf("long double: %zu\n", alignof(long double));
    printf("struct t   : %zu\n", alignof(struct t));
    printf("max_align_t: %zu\n", alignof(max_align_t));
}
```

Output trên hệ của tôi:

```
char       : 1
short      : 2
int        : 4
long       : 8
long long  : 8
double     : 8
long double: 16
struct t   : 16
max_align_t: 16
```

41.3 Hàm `memalignment()`

Mới trong C23!

(Lưu ý: không compiler nào của tôi hỗ trợ hàm này chưa, nên code chủ yếu chưa được test.)

`alignof` hay nếu bạn biết kiểu dữ liệu. Nhưng nếu bạn *ngu dốt đáng thương* về kiểu, và chỉ có con trỏ tới dữ liệu?

Sao điều đó xảy ra được?

À, với người bạn tốt `void*` của ta, tất nhiên. Ta không thể truyền cái đó cho `alignof`, nhưng nếu ta cần biết alignment của thứ nó trỏ tới?

Ta có thể muốn biết điều này nếu ta sắp dùng bộ nhớ cho thứ gì đó có nhu cầu alignment đáng kể. Ví dụ, kiểu atomic và floating thường hành xử xấu nếu không căn chỉnh đúng.

Với hàm này ta có thể kiểm tra alignment của một số dữ liệu miễn là có con trỏ tới dữ liệu đó, ngay cả khi là `void*`.

Hãy làm một test nhanh xem một void pointer có được căn chỉnh tốt để dùng như kiểu atomic hay không, và nếu có, lấy một biến dùng như kiểu đó:

```
void foo(void *p)
{
    if (memalignment(p) >= alignof(atomic int)) {
        atomic int *i = p;
        do_things(i);
    } else
```

```
        puts("This pointer is no good as an atomic int\n");

...
```

Tôi ngờ bạn sẽ hiếm khi (đến mức không bao giờ, có lẽ) cần dùng hàm này trừ khi bạn đang làm thứ gì đó thấp cấp.

Và xong! Alignment!

Index