Beej's Guide to C Programming

Library Reference

Brian "Beej Jorgensen" Hall

Contents

Chapter 1

Foreword

Cánh cửa từ từ kẽo kẹt mở ra, để lộ một hành lang dài với những chồng sách truyền thuyết phủ bụi…

Thôi, có lẽ không phải vậy đâu.

Nhưng bạn đã tìm ra phần Tham chiếu Thư viện (Library Reference) của Beej's Guide to C!

Đây không phải một tutorial, mà là một bộ manual page đầy đủ (hay *man page* theo kiểu gọi của mấy dân Unix), định nghĩa *mọi* hàm trong Thư viện chuẩn C, kèm ví dụ.

> *"Thưa ngài, cuốn này chứa mọi chữ của thứ tiếng mà chúng ta yêu quý."*
> *"Từng chữ một, thưa ngài?"*
> *"Từng chữ một, thưa ngài!"*
> *"Ồ, vậy thì, thưa ngài, tôi mong ngài không phiền nếu tôi cũng gửi đến vị bác sĩ lời chúc phản-miếng-bánh-ngụm (contrafribularities) nồng nhiệt nhất của mình."*
>
> —Blackadder trêu Tiến sĩ Samuel Johnson

Thực ra có một số hàm bị bỏ ngoài guide này, đáng chú ý nhất là mấy hàm "safe" tuỳ chọn (có hậu tố `_s`).

Nhưng mọi thứ bạn có thể muốn thì chắc chắn đều có ở đây. Kèm ví dụ.

Chắc vậy.

1.1 Đối tượng

Guide này dành cho người đã thạo C ở mức kha khá.

Nếu bạn chưa thuộc nhóm đó mà muốn trở thành, tôi xin hết lòng, hoàn toàn không thiên vị, giới thiệu cuốn *Beej's Guide to C Programming*[1], có sẵn miễn phí ở bất cứ đâu Internet được bán.

1.2 Cách đọc cuốn này

Dùng mục lục hoặc index để tìm hàm hoặc chủ đề bạn cần.

Rồi lấy một tô ngũ cốc yêu thích ra, và chén thoải mái đống chữ nghĩa ngon lành trong đây.

1.3 Nền tảng và Compiler

Tôi sẽ cố bám vào ISO-standard C[2] cổ điển. Ừ, phần lớn là vậy. Thi thoảng tôi có thể nổi hứng nói về POSIX[3] hay gì đó, để xem đã.

[1] https://beej.us/guide/bgc/
[2] https://en.wikipedia.org/wiki/ANSI_C
[3] https://en.wikipedia.org/wiki/POSIX

Unix (Linux, BSD, v.v.) thử gõ `cc` hoặc `gcc` ngoài command line, có thể bạn đã có sẵn compiler rồi. Nếu chưa, tìm cách cài `gcc` hoặc `clang` trên distro của bạn.

Windows thử Visual Studio Community[4]. Hoặc, nếu bạn muốn trải nghiệm kiểu Unix (khuyến khích!), cài WSL[5] rồi cài `gcc`.

Mac thì cài XCode[6], nhớ bật command line tools.

Có rất nhiều compiler, và hầu như cái nào cũng dùng được cho cuốn này. Và compiler C++ sẽ compile được khá nhiều code C (nhưng không phải tất cả!). Tốt nhất là dùng compiler C thực thụ nếu có.

1.4 Trang chủ chính thức

Địa chỉ chính thức của tài liệu này là `https://beej.us/guide/bgclr/`[7]. Trước đây ở đây có một ghi chú về việc di chuyển khỏi máy chủ ở Chico State (trường cũ của tôi), nhưng đó là chuyện tỷ năm trước rồi và câu chữ vẫn còn đây chỉ vì nó được copy từ Network Guide sang, [*hít thở*] mà tôi đã khá lâu rồi không đọc lại toàn bộ.

Hết chuyện.

1.5 Chính sách Email

Nói chung tôi luôn sẵn lòng giúp trả lời câu hỏi qua email, cứ viết thoải mái, nhưng tôi không đảm bảo sẽ trả lời. Tôi sống khá bận, có những lúc đơn giản là không trả lời nổi. Những lúc đó tôi thường xoá luôn tin nhắn. Không phải chuyện cá nhân gì đâu; chỉ là tôi không bao giờ đủ thời gian để đưa ra câu trả lời chi tiết mà bạn cần.

Theo kinh nghiệm, câu hỏi càng phức tạp, tôi càng ít khả năng trả lời. Nếu bạn chịu khó thu hẹp câu hỏi trước khi gửi, và nhớ kèm mọi thông tin liên quan (như nền tảng, compiler, các thông báo lỗi bạn gặp, và bất cứ gì bạn nghĩ sẽ giúp tôi debug), bạn có cơ hội được trả lời cao hơn nhiều.

Nếu bạn không được trả lời, cứ cày thêm, thử tự tìm ra đáp án, và nếu nó vẫn lẩn trốn, hãy viết lại cho tôi với thông tin bạn đã thu thập được, và hy vọng chừng đó là đủ để tôi giúp được.

Giờ khi đã cằn nhằn bạn về cách nên và không nên viết thư cho tôi, tôi cũng muốn nói là tôi *hết sức* trân trọng mọi lời khen mà cuốn guide nhận được qua nhiều năm nay. Đó là một liều tinh thần thực sự, và tôi mừng khi biết nó đang được dùng cho mục đích tốt! `:-)` Cảm ơn nhé!

1.6 Mirror

Bạn hoàn toàn được hoan nghênh mirror trang này, dù công khai hay riêng tư. Nếu bạn mirror công khai và muốn tôi link tới từ trang chính, gửi cho tôi một dòng ở `beej@beej.us`.

1.7 Ghi chú cho người dịch

Nếu bạn muốn dịch guide sang ngôn ngữ khác, viết cho tôi ở `beej@beej.us` và tôi sẽ link tới bản dịch của bạn từ trang chính. Bạn có thể thêm tên và thông tin liên hệ của mình vào bản dịch.

Lưu ý các ràng buộc giấy phép ở mục Bản quyền và Phân phối bên dưới.

1.8 Bản quyền và Phân phối

[4] https://visualstudio.microsoft.com/vs/community/
[5] https://docs.microsoft.com/en-us/windows/wsl/install-win10
[6] https://developer.apple.com/xcode/
[7] https://beej.us/guide/bgclr/

Với một số ngoại lệ cụ thể cho source code và bản dịch (xem bên dưới), tác phẩm này được cấp phép theo Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 License. Để xem bản sao của giấy phép, vào `https://creativecommons.org/licenses/by-nc-nd/3.0/` hoặc gửi thư tới Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Một ngoại lệ cụ thể đối với phần "No Derivative Works" của giấy phép là: guide này có thể được dịch tự do sang bất cứ ngôn ngữ nào, miễn là bản dịch chính xác và guide được in lại trọn vẹn. Các ràng buộc giấy phép áp dụng cho bản dịch giống như với bản gốc. Bản dịch cũng có thể kèm tên và thông tin liên hệ của người dịch.

Source code C xuất hiện trong tài liệu này được tuyên bố đưa vào public domain, hoàn toàn không bị ràng buộc giấy phép.

Các nhà giáo dục được khuyến khích tự do giới thiệu hoặc cung cấp bản sao của guide này cho học viên của mình.

Liên hệ `beej@beej.us` để biết thêm.

1.9 Lời đề tặng

Những chuyện khó nhất khi viết mấy cuốn guide này là:

- Học đủ sâu để có thể giải thích được
- Tìm ra cách giải thích sao cho rõ, một quá trình lặp đi lặp lại tưởng chừng vô tận
- Đặt mình ra đó như một cái gọi là *chuyên gia*, trong khi thực ra tôi chỉ là một con người bình thường đang cố hiểu mọi thứ, giống mọi người khác
- Giữ lửa với nó khi nhiều thứ khác cứ đòi sự chú ý của tôi

Rất nhiều người đã giúp tôi qua quá trình này, và tôi muốn ghi công những ai đã góp phần làm nên cuốn sách này.

- Mọi người trên Internet đã quyết định chia sẻ kiến thức của mình theo cách này hay cách khác. Việc chia sẻ kiến thức hướng dẫn một cách tự do chính là thứ làm nên một Internet tuyệt vời như ta thấy.
- Các tình nguyện viên ở cppreference.com[8], những người xây cây cầu nối từ spec sang thực tế.
- Những người tốt bụng và hiểu biết ở comp.lang.c[9] và r/C_Programming[10], những người đã kéo tôi qua các phần khó của ngôn ngữ.
- Tất cả những ai đã gửi corrections và pull-request cho đủ thứ từ hướng dẫn gây hiểu lầm đến lỗi chính tả.

Cảm ơn! ♥

[8] https://en.cppreference.com/
[9] https://groups.google.com/g/comp.lang.c
[10] https://www.reddit.com/r/C_Programming/

Chapter 2

The C Language

Đây là bản tổng quan nhanh các điểm nhấn thời thượng và vui vẻ của cú pháp, keyword, và những "con thú" khác trong sở thú C.

2.1 Bối cảnh

Một vài thứ bạn cần để hiểu các ví dụ bên dưới.

2.1.1 Comment

Comment trong C bắt đầu bằng `//` và kéo đến cuối dòng.

Comment nhiều dòng bắt đầu bằng `/*` và kéo đến khi gặp `*/` đóng.

2.1.2 Dấu phân cách

Biểu thức trong C được phân cách bằng dấu chấm phẩy (`;`). Chúng thường xuất hiện ở cuối dòng.

2.1.3 Biểu thức

Nếu nó không phải keyword hay ký tự đặc biệt, khả năng nó là một biểu thức. Cứ nghĩ "toán học cộng thêm lời gọi hàm".

2.1.4 Câu lệnh

Cứ nghĩ `if` , `while` , v.v. Các keyword thực thi.

2.1.5 Boolean

Bỏ qua kiểu `bool` , số không là false và khác không là true.

2.1.6 Block

Nhiều biểu thức và keyword điều khiển luồng có thể được gói trong một block, gồm `{` theo sau là một hay nhiều biểu thức hoặc câu lệnh, rồi `}` .

2.1.7 Ví dụ code

Chúng nhằm cho bạn hình dung cách dùng các câu lệnh khác nhau, chứ không phải liệt kê ví dụ toàn diện.

Trong các ví dụ dưới đây, nếu chỗ đó có thể là biểu thức hoặc câu lệnh, từ `code` được chèn vào.

2.2 Toán tử

2.2.1 Toán tử số học

Các toán tử số học: `+`, `-`, `*`, `/`, `%` (phần dư).

Phép chia là chia số nguyên nếu mọi toán hạng đều là số nguyên. Nếu không, kết quả là số thực.

Bạn cũng có thể đổi dấu một biểu thức bằng cách đặt `-` trước nó. (Bạn cũng có thể đặt `+` trước, việc này không làm gì về mặt toán học, nhưng nó khiến Usual Arithmetic Conversions được thực hiện trên biểu thức đó.)

2.2.2 Pre- và Post-Increment, Pre- và Post-Decrement

Toán tử post-increment (`++`) và post-decrement (`--`) (đặt sau biến) làm việc của chúng *sau khi* phần còn lại của biểu thức đã được tính.

```
int x = 10;
int y = 20;
int z = 30;

int w = (x++) + (y--) + (z++);

print("%d %d %d %d\n", x, y, z, w);  // 11 19 31 60
```

Toán tử pre-increment (`++`) và pre-decrement (`--`) (đặt trước biến) làm việc của chúng *trước khi* phần còn lại của biểu thức được tính.

```
int x = 10;
int y = 20;
int z = 30;

int w = (++x) + (--y) + (++z);

print("%d %d %d %d\n", x, y, z, w);  // 11 19 31 61
```

2.2.3 Toán tử so sánh

Tất cả đều trả về một giá trị true hoặc false kiểu Boolean.

Nhỏ hơn, lớn hơn, và bằng lần lượt là: `<`, `>`, `==`.

Nhỏ hơn hoặc bằng và lớn hơn hoặc bằng là `<=` và `>=`.

Không bằng là `!=`.

2.2.4 Toán tử con trỏ

`*` đặt trước biến con trỏ dereference biến đó.

`&` đặt trước biến lấy địa chỉ của biến đó.

Các toán tử số học `+` và `-` cũng chạy trên con trỏ để làm số học con trỏ.

2.2.5 Toán tử cho Struct và Union

Toán tử dấu chấm (`.`) lấy giá trị một field từ `struct` hoặc `union`.

Toán tử mũi tên (`->`) lấy giá trị một field từ con trỏ tới `struct` hay `union`. Hai cách dưới đây tương đương nhau, giả sử `p` chính là loại con trỏ đó:

```
(*p).bar;
p->bar;
```

2.2.6 Toán tử mảng

Toán tử ngoặc vuông có thể tham chiếu một giá trị trong mảng:

```
a[10] = 99;
```

Đây là "syntactic sugar" bọc ngoài số học con trỏ và dereference. Dòng trên tương đương với:

```
*(a + 10) = 99;
```

2.2.7 Toán tử bit

Dịch bit phải: `>>`, dịch bit trái: `<<`.

```
int i = x << 3;  // dịch trái 3 bit
```

Việc dịch phải trên một giá trị có dấu có được mở rộng dấu hay không là implementation-defined.

Bitwise AND, OR, NOT, và XOR lần lượt là `&`, `|`, `~`, và `^`.

2.2.8 Toán tử gán

`=` đứng một mình là phép gán cơ bản.

Nhưng còn có các phép gán kết hợp, kiểu viết tắt. Ví dụ, hai dòng sau về cơ bản là tương đương:

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

Có các toán tử gán kết hợp cho phần lớn các toán tử khác.

Số học: `+=`, `-=`, `*=`, `/=`, và `%=`.

Bit: `|=`, `&=`, `~=`, và `^=`.

2.2.9 Toán tử `sizeof`

Đây là toán tử lúc compile, cho bạn kích thước tính bằng byte của kiểu của đối số. Kiểu của biểu thức được dùng; biểu thức không được đánh giá. `sizeof` làm việc với bất kỳ kiểu nào, kể cả kiểu hợp do người dùng định nghĩa.

Kiểu trả về là kiểu integer `size_t`.

```
float f;
size_t x = sizeof f;

printf("f is %zu bytes\n", x);
```

Bạn cũng có thể chỉ định trực tiếp tên kiểu bằng cách bọc nó trong ngoặc:

```
size_t x = sizeof(int);

printf("int is %zu bytes\n", x);
```

2.2.10 Type Cast

Bạn có thể ép một biểu thức sang kiểu khác (trong giới hạn hợp lý) bằng cách *cast* sang kiểu đó.

Bạn đặt tên kiểu mới trong ngoặc.

Ở đây ta đang ép biểu thức con `x` thành kiểu `float` ngay trước phép chia[1]. Cái này làm cho phép chia, nếu không sẽ là chia số nguyên, trở thành chia số thực.

```
int x = 17;
int y = 2;

float f = (float)x / y;
```

2.2.11 Toán tử `_Alignof`

Bạn có thể lấy byte alignment của bất kỳ kiểu nào với toán tử compile-time `_Alignof`. Nếu bạn include `<stdalign.h>`, bạn có thể dùng `alignof` thay thế.

Bất kỳ kiểu nào cũng có thể là đối số của toán tử, và phải đặt trong ngoặc. Khác với `sizeof`, đối số không thể là biểu thức.

```
printf("Alignment of int is %zu\n", alignof(int));
```

2.2.12 Toán tử dấu phẩy

Bạn có thể phân tách các biểu thức con bằng dấu phẩy, mỗi biểu thức sẽ được tính từ trái sang phải, và giá trị của toàn biểu thức sẽ là giá trị của biểu thức con đứng sau dấu phẩy cuối cùng.

```
int x = (1, 2, 3);  // Cách ngôć đểgán `x = 3`
```

Thường thì cái này được dùng trong các mệnh đề của vòng lặp. Ví dụ, ta có thể gán nhiều lần trong vòng `for`, và có nhiều biểu thức post như thế này:

```
for (i = 2, j = 10; i < 100; i++, j += 4) { ... }
```

2.3 Type Specifier

Các kiểu integer từ nhỏ nhất đến lớn nhất: `char`, `short`, `int`, `long`, `long long`.

Bất kỳ kiểu integer nào cũng có thể có tiền tố `signed` (mặc định, trừ `char`) hoặc `unsigned`.

Việc `char` có dấu hay không là implementation-defined.

Các kiểu floating từ ít chính xác nhất đến nhiều nhất: `float`, `double`, `long double`.

`void` là kiểu biểu diễn "không có kiểu".

`_Bool` là kiểu Boolean. Kiểu này thành `bool` trong C23. Các phiên bản C trước đó phải include `<stdbool.h>` để có `bool`.

`_Complex` chỉ một kiểu số phức floating, khi kết hợp với một kiểu đó. Include `<complex.h>` để dùng `complex` thay thế.

```
complex float x = 1.2 + 2.3*I;
complex double y = 1.2 + 2.3*I;
```

[1]Việc này không thay đổi kiểu của `x` ở ngữ cảnh khác, nó chỉ có hiệu lực trong lần dùng cụ thể này trong biểu thức này.

`_Imaginary` là keyword tuỳ chọn, dùng để chỉ một kiểu tưởng tượng (phần imaginary của một số phức) khi kết hợp với kiểu floating. Include `<complex.h>` để dùng `imaginary` thay thế. Cả GCC và clang đều không hỗ trợ cái này.

```
imaginary float f = 2.3*I;
```

`_Generic` là "cái chuyển kiểu", cho phép bạn sinh ra code khác nhau lúc compile tuỳ vào kiểu của dữ liệu.

2.4 Kiểu hằng

Bạn có thể khai báo hằng với kiểu cụ thể (dù đôi khi nó là kiểu lớn hơn). Trong ví dụ dưới, với các kiểu không qualifier, hoa thường không quan trọng, và `U` có thể đứng trước hoặc sau `L` hoặc `LL`.

```
123             int hoặc lớn hơn
123L            long int hoặc lớn hơn
123LL           long long int

123U            unsigned int hoặc lớn hơn
123UL           unsigned long int hoặc lớn hơn
123ULL          unsigned long long int

123.4F          float
123.4           double
123.4L          long double

'a'             char
"hello, world"  char* (string)
```

Bạn cũng có thể chỉ định hằng ở cơ số khác:

```
123             thập phân
0x123           hexa
0123            bát phân
```

Bạn cũng có thể chỉ định hằng floating theo ký hiệu luỹ thừa cơ số 10:

```
1.2e3           1.2 x 10^3
```

Và bạn có thể chỉ định float ở hex! Chỉ có điều trong trường hợp này số mũ vẫn ở thập phân, còn cơ số là 2 thay vì 10:

```
0x1.2p3         0x1.2 x 2^3
```

2.5 Kiểu hợp

2.5.1 Kiểu `struct`

Bạn có thể dựng một kiểu hợp từ các kiểu khác bằng `struct` rồi khai báo biến có kiểu đó.

```
struct animal {
    char *name;
    int leg_count;
};
```

```
struct animal a;
struct animal b = {"goat", 4};
struct animal c = {.name="goat", .leg_count=4};
```

Truy cập bằng toán tử dấu chấm (`.`), hoặc nếu biến là con trỏ tới `struct` , bằng toán tử mũi tên (`->`).

```
struct animal *p = &b;  // b ở trên

printf("%d\n", b.leg_count);
printf("%d\n", p->leg_count);
```

2.5.2 Kiểu `union`

Chúng giống kiểu `struct` về cách dùng, chỉ khác là bạn chỉ dùng được một field tại một thời điểm. (Các field chia sẻ cùng vùng bộ nhớ.)

```
union dt {
    float distance;
    int time;
};

union dt a;
union dt b = {6};            // Khởi tạo "distance", field đầu
union dt c = {.distance=6};  // Khởi tạo "distance"
union dt d = {.time=6};      // Khởi tạo "time"
```

Truy cập bằng toán tử dấu chấm (`.`), hoặc nếu biến là con trỏ tới `union` , bằng toán tử mũi tên (`->`).

```
union dt *p = &b;

printf("%d\n", b.time);
printf("%d\n", p->time);
```

2.5.3 Kiểu `enum`

Cho bạn cách có kiểu để đặt tên cho các giá trị hằng integer. Chúng dùng được với `switch()` , hay làm kích thước mảng, hay ở bất cứ chỗ nào cần giá trị hằng.

Theo thông lệ, tên viết hoa.

```
enum animal {
    ANTELOPE,
    BADGER,
    CAT,
    DOG,
    ELEPHANT,
    FISH
};

enum animal a = CAT;

if (a == CAT)
    printf("The animal is a cat.\n");
```

Các tên có giá trị số bắt đầu từ 0 và đếm lên. (Trong ví dụ trên, `DOG` sẽ là `3` .)

Có thể ghi đè giá trị số bằng cách chỉ định một số nguyên chính xác. Các giá trị sau tăng từ giá trị đã chỉ định đó.

```
enum animal {
    ANTELOPE = 4,
    BADGER,         // Sẽ là 5
    CAT,            // Sẽ là 6
    DOG = 3,
    ELEPHANT,       // Sẽ là 4
    FISH            // Sẽ là 5
};
```

Như trên, giá trị trùng không phải là bất hợp pháp, nhưng hữu dụng thì cũng chẳng bao.

2.6 Initializer

Bạn có thể làm vầy khi biến được định nghĩa, nhưng không được làm ở chỗ khác.

Khởi tạo các kiểu cơ bản:

```
int x = 12;
float y = 1.2;
char c = 'a';
char *s = "Hello, world!";
```

Khởi tạo các kiểu mảng:

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

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

Khởi tạo các kiểu con trỏ:

```
int q;
int *p = &q;
```

Khởi tạo `struct`:

```
struct s {
    int a;
    float b;
};

struct s x0 = {1, 2.2}; // Khởi tạo các field theo thứ tự

struct s x0 = {.a=1, .b=2.2}; // Khởi tạo các field theo tên
struct s x0 = {.b=2.2, .a=1}; // Cùng ý nghĩa

struct s x0 = {.b=2.2}; // Các field còn lại được khởi tạo về 0
struct s x0 = {.b=2.2, .a-=0};  // Cùng ý nghĩa
```

Khởi tạo `union`:

```
union u {
    int a;
    float b;
};

union u x0 = {1};  // Khởi tạo field đầu tiên (a)

union u x0 = {.a=1};  // Khởi tạo field theo tên
union u x0 = {.b=2.2};

//union u x0 = {1, 2};       // BẤT HỢP PHÁP
//union u x0 = {.a=1, .b=2};  // BẤT HỢP PHÁP
```

2.7 Compound Literal

Bạn có thể khai báo object "không tên" trong C. Cái này thường hữu dụng khi truyền một `struct` cho hàm mà không cần đặt tên nó.

Bạn dùng tên kiểu trong ngoặc, theo sau là một initializer để tạo object.

Đây là ví dụ truyền một compound literal cho hàm. Chú ý không có biến `struct s` nào trong `main()`:

```
#include <stdio.h>

struct s {
    int a, b;
};

int add(struct s x)
{
    return x.a + x.b;
}

int main(void)
{
    int t = add((struct s){.a=2, .b=4});  // <-- Đây

    printf("%d\n", t);
}
```

Compound literal có thời gian sống đúng bằng scope của chúng.

Bạn cũng có thể truyền con trỏ tới một compound literal bằng cách lấy địa chỉ của nó:

```
foo(&(struct s){1, 2});
```

2.8 Type Alias

Bạn có thể dựng type alias cho tiện hoặc để trừu tượng hoá.

Ở đây ta sẽ tạo kiểu mới tên `time_counter`, thực ra chỉ là `int`. Nó chỉ dùng được hệt như `int`. Nó chỉ là alias của `int`.

```
typedef int time_counter;

time_counter t = 3490;
```

Cũng chạy với `struct` hay `union`:

```
struct foo {
    int bar;
    float baz;
};

typedef struct foo funtype;

funtype f = {1, 2}; // "funtype" là alias của "struct foo";
```

Nó cũng chạy inline, với `struct` hay `union` có tên hoặc không tên:

```
typedef struct {
    int bar;
    float baz;
} funtype;

funtype f = {1, 2}; // "funtype" là alias cho struct không tên
```

2.9 Specifier khác liên quan đến kiểu

Bạn có thể cho compiler nhiều gợi ý hơn về tính chất mà một kiểu nên có thông qua các specifier và qualifier này.

2.9.1 Storage Class Specifier

Chúng có thể đặt trước một kiểu để chỉ dẫn thêm về cách dùng kiểu đó.

```
auto int a
register int a
static int a
extern int a
thread_local int a
```

`auto` là mặc định, nên về cơ bản chẳng ai dùng. Nó chỉ ra storage duration tự động (những thứ như biến local được tự động giải phóng khi hết scope). Trong C23 keyword này đổi nghĩa thành "suy luận kiểu" kiểu C++.

`register` chỉ ra rằng việc truy cập biến này nên càng nhanh càng tốt. Nó hạn chế một số cách dùng biến để compiler có cơ hội tối ưu. Ít gặp trong công việc hàng ngày.

`static` ở function scope chỉ ra giá trị biến này nên tồn tại qua các lần gọi. Ở file scope nó chỉ ra biến này không nên visible ngoài file nguồn này.

`extern` chỉ ra biến này tham chiếu tới một biến được khai báo ở file nguồn khác.

`_Thread_local` nghĩa là mỗi thread có bản copy riêng của biến này. Bạn có thể dùng `thread_local` nếu include `<threads.h>`.

2.9.2 Type Qualifier

Chúng có thể đặt trước một kiểu để chỉ dẫn thêm về cách dùng kiểu đó.

```
const int a
const int *p
int * const p
```

```
const int * const p
int * restrict p
volatile int a
atomic int a
```

`const` nghĩa là giá trị không được sửa. Bạn cũng có thể dùng với con trỏ:

```
const int a = 10;        // Không sửa được "a"

const int *p = &b        // Không sửa được thứ "p" trỏ tới ("b")
int *const p = &b        // Không sửa được "p"
const int *const p = &b  // Không sửa được cả "p" lẫn thứ nó trỏ tới
```

`restrict` trên con trỏ nghĩa là sẽ chỉ có một con trỏ trỏ tới item đó, cho compiler tự do hơn để tối ưu.

`volatile` chỉ ra giá trị trong biến có thể thay đổi bất cứ lúc nào, và nên được load từ bộ nhớ thay vì giữ trong register. Thường dùng với phần cứng memory-mapped.

`_Atomic` (hoặc `atomic` nếu bạn include `<stdatomic.h>`) bảo compiler rằng việc đọc hoặc ghi kiểu này phải xảy ra atomic. (Việc này có thể được thực hiện bằng lock, tuỳ platform và kiểu.)

2.9.2.1 Pseudo-Type có qualifier của C23: `QVoid*`, `QChar*`, v.v.

Trong C23 có một số hàm generic sẽ trả về kiểu đã được `const`-qualify nếu một trong các tham số là `const`, và không như vậy trong các trường hợp khác.

Spec tự chế một kiểu giả cho mục đích này, với chữ `Q` đằng trước (cho "qualified"). Đây không phải kiểu thật và sẽ không compile được, chỉ để làm tài liệu.

Các pseudo-type này là:

- `QVoid *`
- `QChar *`
- `QWchar_t *`

Ví dụ, hàm `strchr()`, tìm một ký tự trong chuỗi, có prototype thế này trong spec:

```
QChar *strchr(QChar *s, int c);
```

Nó là gì? Nó có nghĩa là nếu `s` có kiểu `const char *`, thì kiểu trả về của hàm cũng sẽ là `const char *`.

Nếu `s` chỉ là `char *`, thì kiểu trả về của hàm cũng chỉ là `char *`.

Nói cách khác, tính `const` của `s` được giữ nguyên ở giá trị trả về.

Nhìn kiểu khác, thì dòng này:

```
QChar *strchr(QChar *s, int c);
```

tương đương với:

```
char *strchr(char *s, int c);
const char *strchr(const char *s, int c);
```

Tóm lại khi bạn thấy cái này, bỏ `Q` đầu đi rồi đổi chữ tiếp theo thành chữ thường là xong.

2.9.3 Function Specifier

Được dùng trên hàm để chỉ dẫn thêm cho compiler.

`_Noreturn` chỉ ra rằng một hàm sẽ không bao giờ return. Nó chỉ có thể chạy mãi hoặc thoát hẳn chương trình. Nếu bạn include `<stdnoreturn.h>`, bạn có thể dùng `noreturn` thay thế.

`inline` chỉ ra rằng các lời gọi hàm này nên càng nhanh càng tốt. Ý định là code của hàm được dời *inline* để bỏ overhead của lời gọi và return. Compiler coi `inline` là gợi ý, không phải yêu cầu.

2.9.4 Alignment Specifier

Bạn có thể ép alignment của một biến trong bộ nhớ bằng `_Alignas`. Nếu bạn include `<stdalign.h>` bạn có thể dùng `alignas` thay thế.

`alignas(0)` không có tác dụng gì.

```
alignas(16) int a = 12;    // alignment 16-byte
alignas(long) int b = 34;  // cùng alignment với "long"
```

2.10 Câu lệnh `if`

```
if (boolean_expression) code;

if (boolean_expression) {
    code;
    code;
    code;
}

if (boolean_expression) {
    code;
    code;
} else
    code;

if (boolean_expression) {
    code;
    code;
} else if {
    code;
    code;
    code;
} else {
    code;
}
```

2.11 Câu lệnh `for`

Vòng `for` cổ điển.

Phần trong ngoặc gồm ba phần phân cách bằng dấu chấm phẩy:

- Khởi tạo, chạy một lần.
- Điều kiện vào block, được đánh giá trước mỗi lần vào thân vòng lặp.
- Biểu thức post, đánh giá sau mỗi lần chạy thân vòng lặp.

Ví dụ, khởi tạo `i` về `0`, vào thân vòng lặp khi `i < 10`, và tăng `i` sau mỗi vòng lặp:

```
for (i = 0; i < 10; i++) {
    code;
    code;
    code;
}
```

Bạn có thể khai báo biến cục bộ trong vòng lặp bằng cách chỉ định kiểu:

```
for (int i = 0; i < 10; i++) {
    code;
    code;
}
```

Bạn có thể phân tách các phần của biểu thức bằng toán tử dấu phẩy:

```
for (i = 0, j = 5; i < 10; i++, j *= 3) {
    code;
    code;
}
```

2.12 Câu lệnh `while`

Vòng lặp này sẽ không vào nếu biểu thức Boolean là false. Kiểm tra điều kiện tiếp tục xảy ra trước thân vòng lặp.

```
while (boolean_expression) code;

while (boolean_expression) {
    code;
    code;
}
```

2.13 Câu lệnh `do` - `while`

Vòng lặp này sẽ chạy ít nhất một lần ngay cả khi biểu thức Boolean là false. Kiểm tra điều kiện tiếp tục không xảy ra cho đến sau thân vòng lặp.

```
do code while (boolean_expression);

do {
    code;
    code;
} while (boolean_expression);
```

2.14 Câu lệnh `switch`

Thực hiện hành động dựa trên giá trị của một biểu thức. Các case so khớp phải là giá trị hằng.

Nếu có `default` tuỳ chọn, code đó được chạy khi không case nào khớp. Không bắt buộc có ngoặc nhọn quanh các case.

```
switch (expression) {
    case constant:
        code;
        code;
        break;

    case constant:
        code;
        code;
        break;

    default:
        code;
        break;
}
```

`break` cuối trong `switch` là không cần thiết nếu không còn case nào sau nó.

Nếu `break` không có, `case` rơi qua case kế tiếp. Nên ghi comment cho chuyện đó để các dev khác không ghét bạn.

```
switch (expression) {
    case constant:
        code;
        code;
        // fall through!

    case constant:
        code;
        break;
}
```

2.15 Câu lệnh `break`

Lệnh này thoát khỏi một case trong `switch`, nhưng cũng có thể thoát khỏi bất kỳ vòng lặp nào.

```
while (boolean_expression) {
    code;

    if (boolean_expression)
        break;

    code;
}
```

2.16 Câu lệnh `continue`

Có thể dùng để short-circuit một vòng lặp và đi tới kiểm tra điều kiện tiếp theo mà không hoàn tất thân vòng lặp.

```
while (boolean_expression) {
    code;
    code;

    if (boolean_expression_2)
```

```
        continue;

    // Nếu boolean_expression_2, code dưới đây sẽ bị bỏ qua:

    code;
    code;
}
```

2.17 Câu lệnh `goto`

Bạn có thể nhảy đến bất kỳ đâu trong một hàm bằng `goto`. (Bạn không thể `goto` giữa các hàm, chỉ trong cùng hàm với `goto`.)

Đích của `goto` là một *label*, là một identifier theo sau là dấu hai chấm (`:`). Label thường được canh sát lề trái để dễ nhìn.

```
{
    // Code minh hoạ kiểu lạm dụng mà đáng lẽ phải là vòng while

    int i = 0;

loop:

    printf("%d\n", i++);

    if (i < 10)
        goto loop;
}
```

2.18 Câu lệnh `return`

Đây là cách bạn về từ một hàm. Bạn có thể `return` nhiều lần hoặc chỉ một lần.

Nếu một hàm kiểu trả về `void` chạy hết, `return` là ngầm định.

Nếu kiểu trả về không phải `void`, câu lệnh `return` phải chỉ định giá trị trả về cùng kiểu.

Không cần ngoặc quanh giá trị return (vì đây là câu lệnh, không phải hàm).

```
int increment(int a)
{
    return a + 1;
}
```

2.19 Câu lệnh `_Static_assert`

Đây là cách ngăn *compilation* một chương trình nếu một điều kiện hằng nào đó không thoả.

```
_Static_assert(__STDC_VERSION__ >= 201112L, "You need at least C11!")
```

2.20 Hàm

Bạn cần chỉ định kiểu trả về và kiểu tham số cho hàm, thân hàm đặt trong block sau đó.

Biến trong hàm là local với hàm đó.

```
// Hàm cộng hai sô´

int add(int x, int y)
{
    int sum = x + y;

    return sum;
}
```

Hàm không return gì nên có kiểu trả về `void`. Hàm không nhận tham số nên có `void` làm danh sách tham số.

```
// Toàn tác dụng phụ, mọi lúc!

void foo(void)
{
    some_global = 12;
    printf("Here we go!\n");
}
```

2.20.1 Hàm `main()`

Đây là hàm chạy khi bạn bắt đầu chương trình. Nó sẽ ở một trong các dạng sau:

```
int main(void)
int main(int argc, char *argv[])
```

Dạng đầu bỏ qua mọi tham số command line.

Dạng thứ hai lưu số lượng tham số command line vào `argc`, và lưu chính các tham số đó thành mảng chuỗi trong `argv`. Cái đầu tiên, `argv[0]`, thường là tên của file thực thi. Con trỏ `argv` cuối cùng có giá trị `NULL`.

Giá trị trả về thường xuất hiện thành exit status code trong OS. Nếu không có `return`, chạy hết `main()` được ngầm coi là `return 0`[2].

2.20.2 Hàm Variadic

Một số hàm có thể nhận số lượng tham số biến đổi. Mọi hàm đều phải có ít nhất một tham số. Các tham số còn lại được chỉ định bằng `...` và có thể đọc qua các macro `va_start()`, `va_arg()`, và `va_end()`.

Đây là ví dụ cộng số lượng biến đổi các giá trị integer.

```
int add(int count, ...)
{
    int total = 0;
    va_list va;

    va_start(va, count);   // Bắt đầu với tham sô´ sau "count"

    for (int i = 0; i < count; i++) {
        int n = va_arg(va, int);   // Lấy int kê´ tiêṕ

        total += n;
```

[2] Lưu ý ngầm định này chỉ áp dụng cho `main()`, không áp dụng cho bất cứ hàm nào khác.

```
    }

    va_end(va);  // Xong

    return total;
}
```

Chapter 3

`<assert.h>` Runtime and Compile-time Diagnostics

Macro	Mô tả
`assert()`	Assertion lúc chạy
`static_assert()`	Assertion lúc compile

Chức năng này liên quan tới những thứ Không-Bao-Giờ-Nên-Xảy-Ra™. Nếu có điều gì đó lẽ ra không bao giờ đúng và bạn muốn chương trình nổ tung khi nó xảy ra, đây là header dành cho bạn.

Có hai loại assertion: assertion lúc compile (gọi là "static assertion") và assertion lúc chạy (runtime). Nếu assertion *thất bại* (nghĩa là thứ bạn cần đúng thì hoá ra không đúng), chương trình sẽ nổ tung ở compile-time hoặc runtime.

3.1 Macro

Nếu bạn define macro `NDEBUG` **trước khi** include `<assert.h>`, macro `assert()` sẽ không còn tác dụng. Bạn có thể define `NDEBUG` thành bất cứ thứ gì, nhưng `1` có vẻ là giá trị hợp lý.

Vì `assert()` làm chương trình nổ tung lúc chạy, chắc là bạn không muốn hành vi đó khi lên production. Define `NDEBUG` sẽ khiến `assert()` bị bỏ qua.

`NDEBUG` không ảnh hưởng tới `static_assert()`.

3.2 `assert()`

Nổ tung lúc chạy nếu một điều kiện thất bại

Synopsis

```
#include <assert.h>

void assert(scalar expression);
```

Mô tả

Bạn truyền một biểu thức vào macro này. Nếu nó đánh giá ra false, chương trình sẽ crash với assertion failure (bằng cách gọi hàm `abort()`).

Về cơ bản, bạn đang nói, "Này, tôi đang giả định điều kiện này là true, và nếu không phải vậy, tôi không muốn chạy tiếp nữa."

Cái này được dùng khi debug để đảm bảo không có điều kiện bất ngờ nào phát sinh. Và nếu trong quá trình phát triển bạn thấy điều kiện đó có phát sinh thật, có thể bạn nên sửa code để xử lý nó trước khi lên production.

Nếu bạn đã define macro `NDEBUG` thành bất kỳ giá trị nào trước khi `<assert.h>` được include, macro `assert()` sẽ bị bỏ qua. Đây là ý hay trước khi ra production.

Khác với `static_assert()`, macro này không cho bạn in một thông báo tuỳ ý. Nếu bạn muốn làm vậy, có thể tự viết một assert riêng dạng preprocessor macro:

```
#define ASSERT(c, m) \
do { \
    if (!(c)) { \
        fprintf(stderr, __FILE__ ":%d: assertion %s failed: %s\n", \
                        __LINE__, #c, m); \
        exit(1); \
    } \
} while(0)
```

Giá trị trả về

Macro này không return (vì nó gọi `abort()`, không bao giờ return).

Nếu `NDEBUG` được set, macro đánh giá ra `((void)0)`, không làm gì cả.

Ví dụ

Đây là hàm chia kích thước đàn dê của ta. Nhưng ta giả định sẽ không bao giờ bị truyền `0` vào.

Nên ta assert `amount != 0`... và nếu có, chương trình abort.

```
//#define NDEBUG 1   // bỏ comment để tắt assert

#include <stdio.h>
#include <assert.h>

int goat_count = 10;

void divide_goat_herd_by(int amount)
{
    assert(amount != 0);

    goat_count /= amount;
}

int main(void)
{
    divide_goat_herd_by(2);  // OK

    divide_goat_herd_by(0);  // Kích hoạt assert
}
```

Khi tôi chạy và truyền `0` cho hàm, tôi nhận được output sau trên hệ của tôi (output cụ thể có thể khác):

```
assert: assert.c:10: divide_goat_herd_by: Assertion `amount != 0' failed.
```

Xem thêm

`static_assert()`, `abort()`

3.3 `static_assert()`

Nổ tung lúc compile nếu một điều kiện thất bại

Synopsis

```
#include <assert.h>

static_assert(constant-expression, string-literal);
```

Mô tả

Macro này ngăn chương trình của bạn được compile luôn nếu một điều kiện không đúng.

Và nó in ra string literal bạn đưa cho nó.

Cơ bản là nếu `constant-expression` là false, việc compile sẽ dừng và `string-literal` sẽ được in ra.

Constant expression phải thực sự là hằng, chỉ giá trị, không biến. String literal cũng vậy: không biến, chỉ là literal string đặt trong dấu ngoặc kép. (Phải như vậy vì chương trình chưa chạy ở thời điểm này.)

Giá trị trả về

Không áp dụng, vì đây là chức năng compile-time.

Ví dụ

Đây là một ví dụ phần, với một thuật toán mà hiệu năng hay bộ nhớ chắc là tệ nếu kích thước mảng local quá lớn. Ta phòng trường hợp đó ngay lúc compile bằng cách bắt nó với `static_assert()`.

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

#define ARRAY_SIZE 16

int main(void)
{
    static_assert(ARRAY_SIZE > 32, "ARRAY_SIZE too small");

    int a[ARRAY_SIZE];

    a[32] = 10;

    printf("%d\n", a[32]);
}
```

Trên hệ của tôi, khi tôi cố compile, nó in ra (output có thể khác trên hệ khác):

```
In file included from static_assert.c:2:
static_assert.c: In function ‘main’:
static_assert.c:8:5: error: static assertion failed: "ARRAY_SIZE too small"
    8 |     static_assert(ARRAY_SIZE > 32, "ARRAY_SIZE too small");
      |     ^~~~~~~~~~~~~
```

Xem thêm

`assert()`

Chapter 4

`<complex.h>` Chức năng Số phức

Các hàm số phức trong phần tham chiếu này đều có ba phiên bản: `double complex`, `float complex`, và `long double complex`.

Biến thể `float` kết thúc bằng `f` còn biến thể `long double` kết thúc bằng `l`, ví dụ với cosine của số phức:

```
ccos()   double complex
ccosf()  float complex
ccosl()  long double complex
```

Bảng dưới chỉ liệt kê phiên bản `double complex` cho gọn.

Hàm	Mô tả
`cabs()`	Tính giá trị tuyệt đối của số phức
`cacos()`	Tính arc-cosine phức
`cacosh()`	Tính arc hyperbolic cosine phức
`carg()`	Tính argument phức
`casin()`	Tính arc-sine phức
`casinh()`	Tính arc hyperbolic sine phức
`catan()`	Tính arc-tangent phức
`catanh()`	Tính arc hyperbolic tangent phức
`ccos()`	Tính cosine phức
`ccosh()`	Tính hyperbolic cosine phức
`cexp()`	Tính hàm mũ cơ số e phức
`cimag()`	Trả về phần ảo của một số phức
`clog()`	Tính logarithm phức
`CMPLX()`	Dựng một giá trị phức từ kiểu thực và ảo
`conj()`	Tính liên hợp của một số phức
`cproj()`	Tính phép chiếu của một số phức
`creal()`	Trả về phần thực của một số phức
`csin()`	Tính sine phức
`csinh()`	Tính hyperbolic sine phức
`csqrt()`	Tính căn bậc hai phức
`ctan()`	Tính tangent phức
`ctanh()`	Tính hyperbolic tangent phức

Bạn có thể kiểm tra hỗ trợ số phức bằng cách xem macro `__STDC_NO_COMPLEX__`. Nếu nó được định nghĩa, tức là số phức không có sẵn.

Có thể có hai loại số được định nghĩa: *complex* và *imaginary*. Hiện tôi không biết hệ thống nào hiện thực kiểu imaginary cả.

Các kiểu complex, tức một giá trị thực cộng với bội của i, là:

```
float complex
double complex
long double complex
```

Các kiểu imaginary, tức chỉ chứa bội của i, là:

```
float imaginary
double imaginary
long double imaginary
```

Giá trị toán học $i = \sqrt{-1}$ được biểu diễn bằng ký hiệu `_Complex_I` hoặc `_Imaginary_I`, nếu có.

Macro `I` sẽ được ưu tiên đặt thành `_Imaginary_I` (nếu có), hoặc là `_Complex_I` nếu không.

Bạn có thể viết literal ảo (nếu được hỗ trợ) theo cú pháp này:

```
double imaginary x = 3.4 * I;
```

Bạn có thể viết literal phức bằng ký pháp phức thông thường:

```
double complex x = 1.2 + 3.4 * I;
```

hoặc dựng chúng bằng macro `CMPLX()`:

```
double complex x = CMPLX(1.2, 3.4);  // Giống 1.2 + 3.4 * I
```

Cách sau có lợi thế là xử lý đúng các trường hợp đặc biệt của số phức (như những trường hợp dính vô cùng hoặc số không có dấu) như thể `_Imaginary_I` đang có mặt, dù thực ra không có.

Mọi giá trị góc đều tính bằng radian.

Một số hàm có các điểm gián đoạn gọi là *branch cut* (nhát cắt nhánh). Thú thật tôi không phải dân toán nên không bàn nghiêm túc về chuyện này được, nhưng nếu bạn đang ở đây thì tôi tin bạn biết mình đang làm gì ở mảng này.

Nếu hệ của bạn có số không có dấu, bạn có thể biết mình đang ở phía nào của nhát cắt dựa vào dấu. Còn nếu không có thì chịu. Spec viết thêm:

> Những hiện thực không hỗ trợ số không có dấu [...] không thể phân biệt hai phía của nhát cắt nhánh. Các hiện thực này phải ánh xạ nhát cắt sao cho hàm liên tục khi tiếp cận nhát cắt đi quanh điểm đầu hữu hạn của nhát cắt theo chiều ngược kim đồng hồ. (Các nhát cắt nhánh cho những hàm ở đây đều chỉ có một điểm đầu hữu hạn.) Ví dụ, với hàm căn bậc hai, đi ngược kim đồng hồ quanh điểm đầu hữu hạn của nhát cắt dọc theo trục thực âm thì tiếp cận nhát cắt từ phía trên, vậy nên nhát cắt ánh xạ vào trục ảo dương.

Cuối cùng, có một pragma tên `CX_LIMITED_RANGE` có thể bật/tắt (mặc định là tắt). Bạn bật nó như thế này:

```
#pragma STDC CX_LIMITED_RANGE ON
```

Nó cho phép một số phép toán trung gian được tràn dưới, tràn trên, hoặc xử lý lỏng tay với vô cùng, đại khái là đánh đổi lấy tốc độ. Nếu bạn chắc chắn những lỗi kiểu đó không xảy ra với các số bạn đang dùng VÀ bạn đang cố vắt kiệt tốc độ, bạn có thể bật macro này lên.

Spec cũng viết thêm:

> Mục đích của pragma là cho phép hiện thực dùng các công thức:
> $(x + iy) \times (u + iv) = (xu - yv) + i(yu + xv)$
> $(x + iy)/(u + iv) = [(xu + yv) + i(yu - xv)]/(u^2 + v^2)$
> $|x + iy| = \sqrt{x^2 + y^2}$
> khi lập trình viên xác định được là chúng an toàn.

4.1 `cacos()`, `cacosf()`, `cacosl()`

Tính arc-cosine phức

Synopsis

```
#include <complex.h>

double complex cacos(double complex z);

float complex cacosf(float complex z);

long double complex cacosl(long double complex z);
```

Mô tả

Tính arc-cosine phức của một số phức.

Số phức `z` sẽ có phần ảo trong khoảng $[0, \pi]$, còn phần thực không bị chặn.

Có các nhát cắt nhánh nằm ngoài khoảng $[-1, +1]$ trên trục thực.

Giá trị trả về

Trả về arc-cosine phức của `z`.

Ví dụ

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

int main(void)
{
    double complex x = 8 + 1.5708 * I;

    double complex y = cacos(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 0.195321 + -2.788006i
```

Xem thêm

`ccos()`, `casin()`, `catan()`

4.2 `casin()`, `casinf()`, `casinl()`

Tính arc-sine phức

Synopsis

```
#include <complex.h>

double complex casin(double complex z);

float complex casinf(float complex z);

long double complex casinl(long double complex z);
```

Mô tả

Tính arc-sine phức của một số phức.

Số phức `z` sẽ có phần ảo trong khoảng $[-\pi/2, +\pi/2]$, còn phần thực không bị chặn.

Có các nhát cắt nhánh nằm ngoài khoảng $[-1, +1]$ trên trục thực.

Giá trị trả về

Trả về arc-sine phức của `z`.

Ví dụ

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

int main(void)
{
    double complex x = 8 + 1.5708 * I;

    double complex y = casin(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 1.375476 + 2.788006i
```

Xem thêm

`csin()`, `cacos()`, `catan()`

4.3 `catan()`, `catanf()`, `catanl()`

Tính arc-tangent phức

Synopsis

```
#include <complex.h>

double complex catan(double complex z);

float complex catanf(float complex z);

long double complex catanl(long double complex z);
```

Mô tả

Tính arc-tangent phức của một số phức.

Số phức `z` sẽ có phần thực trong khoảng $[-\pi/2, +\pi/2]$, còn phần ảo không bị chặn.

Có các nhát cắt nhánh nằm ngoài khoảng $[-i, +i]$ trên trục ảo.

Giá trị trả về

Trả về arc-tangent phức của `z`.

Ví dụ

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

int main(void)
{
    double wheat = 8;
    double sheep = 1.5708;

    double complex x = wheat + sheep * I;

    double complex y = catan(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 1.450947 + 0.023299i
```

Xem thêm

`ctan()`, `cacos()`, `casin()`

4.4 `ccos()`, `ccosf()`, `ccosl()`

Tính cosine phức

Synopsis

```
#include <complex.h>

double complex ccos(double complex z);

float complex ccosf(float complex z);

long double complex ccosl(long double complex z);
```

Mô tả

Tính cosine phức của một số phức.

Giá trị trả về

Trả về cosine phức của `z`.

Ví dụ

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

int main(void)
{
    double complex x = 8 + 1.5708 * I;

    double complex y = ccos(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: -0.365087 + -2.276818i
```

Xem thêm

`csin()`, `ctan()`, `cacos()`

4.5 `csin()`, `csinf()`, `csinl()`

Tính sine phức

Synopsis

```
#include <complex.h>

double complex csin(double complex z);

float complex csinf(float complex z);

long double complex csinl(long double complex z);
```

Mô tả

Tính sine phức của một số phức.

Giá trị trả về

Trả về sine phức của `z`.

Ví dụ

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

int main(void)
{
    double complex x = 8 + 1.5708 * I;

    double complex y = csin(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 2.482485 + -0.334840i
```

Xem thêm

`ccos()`, `ctan()`, `casin()`

4.6 `ctan()`, `ctanf()`, `ctanl()`

Tính tangent phức

Synopsis

```
#include <complex.h>

double complex ctan(double complex z);

float complex ctanf(float complex z);

long double complex ctanl(long double complex z);
```

Mô tả

Tính tangent phức của một số phức.

Giá trị trả về

Trả về tangent phức của `z`.

Ví dụ

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

int main(void)
{
    double complex x = 8 + 1.5708 * I;

    double complex y = ctan(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: -0.027073 + 1.085990i
```

Xem thêm

`ccos()`, `csin()`, `catan()`

4.7 `cacosh()`, `cacoshf()`, `cacoshl()`

Tính arc hyperbolic cosine phức

Synopsis

```
#include <complex.h>

double complex cacosh(double complex z);

float complex cacoshf(float complex z);

long double complex cacoshl(long double complex z);
```

Mô tả

Tính arc hyperbolic cosine phức của một số phức.

Có một nhát cắt nhánh tại các giá trị nhỏ hơn 1 trên trục thực.

Giá trị trả về sẽ không âm trên trục số thực, và nằm trong khoảng $[-i\pi, +i\pi]$ trên trục ảo.

Giá trị trả về

Trả về arc hyperbolic cosine phức của `z`.

Ví dụ

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

int main(void)
{
```

```
    double complex x = 8 + 1.5708 * I;

    double complex y = cacosh(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 2.788006 + 0.195321i
```

Xem thêm

`casinh()`, `catanh()`, `acosh()`

4.8 `casinh()`, `casinhf()`, `casinhl()`

Tính arc hyperbolic sine phức

Synopsis

```
#include <complex.h>

double complex casinh(double complex z);

float complex casinhf(float complex z);

long double complex casinhl(long double complex z);
```

Mô tả

Tính arc hyperbolic sine phức của một số phức.

Có các nhát cắt nhánh nằm ngoài $[-i, +i]$ trên trục ảo.

Giá trị trả về không bị chặn trên trục số thực, và nằm trong khoảng $[-i\pi/2, +i\pi/2]$ trên trục ảo.

Giá trị trả về

Trả về arc hyperbolic sine phức của `z`.

Ví dụ

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

int main(void)
{
    double complex x = 8 + 1.5708 * I;

    double complex y = casinh(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 2.794970 + 0.192476i
```

Xem thêm

`cacosh()`, `catanh()`, `asinh()`

4.9 `catanh()`, `catanhf()`, `catanhl()`

Tính arc hyperbolic tangent phức

Synopsis

```
#include <complex.h>

double complex catanh(double complex z);

float complex catanhf(float complex z);

long double complex catanhl(long double complex z);
```

Mô tả

Tính arc hyperbolic tangent phức của một số phức.

Có các nhát cắt nhánh nằm ngoài $[-1, +1]$ trên trục thực.

Giá trị trả về không bị chặn trên trục số thực, và nằm trong khoảng $[-i\pi/2, +i\pi/2]$ trên trục ảo.

Giá trị trả về

Trả về arc hyperbolic tangent phức của `z`.

Ví dụ

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

int main(void)
{
    double complex x = 8 + 1.5708 * I;

    double complex y = catanh(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 0.120877 + 1.546821i
```

Xem thêm

`cacosh()`, `casinh()`, `atanh()`

4.10 `ccosh()`, `ccoshf()`, `ccoshl()`

Tính hyperbolic cosine phức

Synopsis

```
#include <complex.h>

double complex ccosh(double complex z);

float complex ccoshf(float complex z);

long double complex ccoshl(long double complex z);
```

Mô tả

Tính hyperbolic cosine phức của một số phức.

Giá trị trả về

Trả về hyperbolic cosine phức của `z`.

Ví dụ

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

int main(void)
{
    double complex x = 8 + 1.5708 * I;

    double complex y = ccosh(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: -0.005475 + 1490.478826i
```

Xem thêm

`csinh()`, `ctanh()`, `ccos()`

4.11 `csinh()`, `csinhf()`, `csinhl()`

Tính hyperbolic sine phức

Synopsis

```
#include <complex.h>

double complex csinh(double complex z);

float complex csinhf(float complex z);

long double complex csinhl(long double complex z);
```

Mô tả

Tính hyperbolic sine phức của một số phức.

Giá trị trả về

Trả về hyperbolic sine phức của `z`.

Ví dụ

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

int main(void)
{
    double complex x = 8 + 1.5708 * I;

    double complex y = csinh(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: -0.005475 + 1490.479161i
```

Xem thêm

`ccosh()`, `ctanh()`, `csin()`

4.12 `ctanh()`, `ctanhf()`, `ctanhl()`

Tính hyperbolic tangent phức

Synopsis

```
#include <complex.h>

double complex ctanh(double complex z);

float complex ctanhf(float complex z);

long double complex ctanhl(long double complex z);
```

Mô tả

Tính hyperbolic tangent phức của một số phức.

Giá trị trả về

Trả về hyperbolic tangent phức của `z`.

Ví dụ

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

int main(void)
{
    double complex x = 8 + 1.5708 * I;

    double complex y = ctanh(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 1.000000 + -0.000000i
```

Xem thêm

`ccosh()`, `csinh()`, `ctan()`

4.13 `cexp()`, `cexpf()`, `cexpl()`

Tính hàm mũ cơ số e phức

Synopsis

```
#include <complex.h>

double complex cexp(double complex z);

float complex cexpf(float complex z);

long double complex cexpl(long double complex z);
```

Mô tả

Tính hàm mũ cơ số e phức của `z`.

Giá trị trả về

Trả về hàm mũ cơ số e phức của `z`.

Ví dụ

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

int main(void)
{
    double complex x = 1 + 2 * I;

    double complex y = cexp(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: -1.131204 + 2.471727i
```

Xem thêm

`cpow()`, `clog()`, `exp()`

4.14 `clog()`, `clogf()`, `clogl()`

Tính logarithm phức

Synopsis

```
#include <complex.h>

double complex clog(double complex z);

float complex clogf(float complex z);

long double complex clogl(long double complex z);
```

Mô tả

Tính logarithm cơ số e phức của `z`. Có một nhát cắt nhánh trên trục thực âm.

Giá trị trả về không bị chặn trên trục thực và nằm trong khoảng $[-i\pi, +i\pi]$ trên trục ảo.

Giá trị trả về

Trả về logarithm cơ số e phức của `z`.

Ví dụ

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

int main(void)
{
    double complex x = 1 + 2 * I;
```

```
    double complex y = clog(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 0.804719 + 1.107149i
```

Xem thêm

`cexp()`, `log()`

4.15 `cabs()`, `cabsf()`, `cabsl()`

Tính giá trị tuyệt đối phức

Synopsis

```
#include <complex.h>

double cabs(double complex z);

float cabsf(float complex z);

long double cabsl(long double complex z);
```

Mô tả

Tính giá trị tuyệt đối phức của `z`.

Giá trị trả về

Trả về giá trị tuyệt đối phức của `z`.

Ví dụ

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

int main(void)
{
    double complex x = 1 + 2 * I;

    double complex y = cabs(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 2.236068 + 0.000000i
```

Xem thêm

`fabs()`, `abs()`

4.16 `cpow()`, `cpowf()`, `cpowl()`

Tính luỹ thừa phức

Synopsis

```
#include <complex.h>

double complex cpow(double complex x, double complex y);

float complex cpowf(float complex x, float complex y);

long double complex cpowl(long double complex x,
                          long double complex y);
```

Mô tả

Tính x^y phức.

Có một nhát cắt nhánh cho `x` dọc theo trục thực âm.

Giá trị trả về

Trả về x^y phức.

Ví dụ

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

int main(void)
{
    double complex x = 1 + 2 * I;
    double complex y = 3 + 4 * I;

    double r = cpow(x, y);

    printf("Result: %f + %fi\n", creal(r), cimag(r));
}
```

Result:

```
Result: 0.129010 + 0.000000i
```

Xem thêm

`csqrt()`, `cexp()`

4.17 `csqrt()`, `csqrtf()`, `csqrtl()`

Tính căn bậc hai phức

Synopsis

```
#include <complex.h>

double complex csqrt(double complex z);

float complex csqrtf(float complex z);

long double complex csqrtl(long double complex z);
```

Mô tả

Tính căn bậc hai phức của `z`.

Có một nhát cắt nhánh dọc theo trục thực âm.

Giá trị trả về nằm ở nửa phải của mặt phẳng phức và bao gồm cả trục ảo.

Giá trị trả về

Trả về căn bậc hai phức của `z`.

Ví dụ

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

int main(void)
{
    double complex x = 1 + 2 * I;

    double complex y = csqrt(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 1.272020 + 0.786151i
```

Xem thêm

`cpow()`, `sqrt()`

4.18 `carg()`, `cargf()`, `cargl()`

Tính argument phức

Synopsis

```
#include <complex.h>

double carg(double complex z);

float cargf(float complex z);

long double cargl(long double complex z);
```

Mô tả

Tính argument phức (còn gọi là góc pha) của `z`.

Có một nhát cắt nhánh dọc theo trục thực âm.

Trả về giá trị trong khoảng $[-\pi, +\pi]$.

Giá trị trả về

Trả về argument phức của `z`.

Ví dụ

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

int main(void)
{
    double complex x = 1 + 2 * I;

    double y = carg(x);

    printf("Result: %f\n", y);
}
```

Output:

```
Result: 1.107149
```

4.19 `cimag()`, `cimagf()`, `cimagl()`

Trả về phần ảo của một số phức

Synopsis

```
#include <complex.h>

double cimag(double complex z);

float cimagf(float complex z);

long double cimagl(long double complex z);
```

Mô tả

Trả về phần ảo của `z`.

Nói ngoài lề, spec chỉ ra rằng bất kỳ số phức `x` nào cũng tuân theo đẳng thức sau:

```
x == creal(x) + cimag(x) * I;
```

Giá trị trả về

Trả về phần ảo của `z`.

Ví dụ

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

int main(void)
{
    double complex x = 1 + 2 * I;

    double y = cimag(x);

    printf("Result: %f\n", y);
}
```

Output—chỉ phần ảo:

```
Result: 2.000000
```

Xem thêm

`creal()`

4.20 `CMPLX()`, `CMPLXF()`, `CMPLXL()`

Dựng một giá trị phức từ kiểu thực và ảo

Synopsis

```
#include <complex.h>

double complex CMPLX(double x, double y);

float complex CMPLXF(float x, float y);

long double complex CMPLXL(long double x, long double y);
```

Mô tả

Các macro này dựng một giá trị phức từ các kiểu thực và ảo.

Chắc là bạn đang nghĩ, "Nhưng tôi đã có thể dựng giá trị phức từ kiểu thực và ảo bằng macro `I` rồi mà, như trong ví dụ sắp tới."

```
double complex x = 1 + 2 * I;
```

Và điều đó đúng.

Nhưng thực tế vấn đề lạ và phức.

Có thể `I` đã bị undefine, hoặc có thể bạn đã redefine nó.

Hoặc có thể `I` được định nghĩa là `_Complex_I`, thứ không nhất thiết giữ được dấu của giá trị không.

Như spec chỉ ra, các macro này dựng số phức như thể `_Imaginary_I` đang có mặt (nhờ đó giữ được dấu không của bạn) ngay cả khi nó không có. Cụ thể, chúng được định nghĩa tương đương với:

```
#define CMPLX(x, y)  ((double complex)((double)(x) + \
                     _Imaginary_I * (double)(y)))

#define CMPLXF(x, y) ((float complex)((float)(x) + \
                     _Imaginary_I * (float)(y)))

#define CMPLXL(x, y) ((long double complex)((long double)(x) + \
                     _Imaginary_I * (long double)(y)))
```

Giá trị trả về

Trả về số phức cho các phần thực `x` và ảo `y` đã cho.

Ví dụ

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

int main(void)
{
    double complex x = CMPLX(1, 2);  // Giống 1 + 2 * I

    printf("Result: %f + %fi\n", creal(x), cimag(x));
}
```

Output:

```
Result: 1.000000 + 2.000000i
```

Xem thêm

`creal()`, `cimag()`

4.21 `conj()`, `conjf()`, `conjl()`

Tính liên hợp của một số phức

Synopsis

```
#include <complex.h>

double complex conj(double complex z);

float complex conjf(float complex z);

long double complex conjl(long double complex z);
```

Mô tả

Hàm này tính liên hợp phức[1] của `z`. Hình như nó làm vậy bằng cách đảo dấu phần ảo, nhưng trời ạ, tôi là lập trình viên chứ không phải dân toán, Jim ơi!

Giá trị trả về

Trả về liên hợp phức của `z`

Ví dụ

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

int main(void)
{
    double complex x = 1 + 2 * I;

    double complex y = conj(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 1.000000 + -2.000000i
```

4.22 `cproj()`, `cproj()`, `cproj()`

Tính phép chiếu của một số phức

Synopsis

```
#include <complex.h>

double complex cproj(double complex z);

float complex cprojf(float complex z);

long double complex cprojl(long double complex z);
```

[1] https://en.wikipedia.org/wiki/Complex_conjugate

Mô tả

Tính phép chiếu của `z` lên một mặt cầu Riemann[2].

Giờ thì *thực sự* ngoài vùng chuyên môn của tôi rồi. Spec viết thế này, tôi trích nguyên văn vì không đủ hiểu biết để viết lại cho gọn. Mong là nó hiểu được với ai cần dùng hàm này.

> `z` chiếu thành `z` trừ khi mọi vô cùng phức (kể cả những vô cùng có một phần vô cùng và một phần `NaN`) đều chiếu thành vô cùng dương trên trục thực. Nếu `z` có phần vô cùng, thì `cproj(z)` tương đương với
>
> ```
> INFINITY + I * copysign(0.0, cimag(z))
> ```

Đấy, có vậy thôi.

Giá trị trả về

Trả về phép chiếu của `z` lên một mặt cầu Riemann.

Ví dụ

Bắt chéo ngón tay cầu cho ví dụ này có tí hợp lý...

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

int main(void)
{
    double complex x = 1 + 2 * I;

    double complex y = cproj(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));

    x = INFINITY + 2 * I;
    y = cproj(x);

    printf("Result: %f + %fi\n", creal(y), cimag(y));
}
```

Output:

```
Result: 1.000000 + 2.000000i
Result: inf + 0.000000i
```

4.23 `creal()`, `crealf()`, `creall()`

Trả về phần thực của một số phức

[2] https://en.wikipedia.org/wiki/Riemann_sphere

Synopsis

```
#include <complex.h>

double creal(double complex z);

float crealf(float complex z);

long double creall(long double complex z);
```

Mô tả

Trả về phần thực của `z`.

Nói ngoài lề, spec chỉ ra rằng bất kỳ số phức `x` nào cũng tuân theo đẳng thức sau:

```
x == creal(x) + cimag(x) * I;
```

Giá trị trả về

Trả về phần thực của `z`.

Ví dụ

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

int main(void)
{
    double complex x = 1 + 2 * I;

    double y = creal(x);

    printf("Result: %f\n", y);
}
```

Output—chỉ phần thực:

```
Result: 1.000000
```

Xem thêm

`cimag()`

Chapter 5

`<ctype.h>` Phân loại và Chuyển đổi Ký tự

Hàm	Mô tả
`isalnum()`	Kiểm tra ký tự có là chữ cái hoặc chữ số
`isalpha()`	Trả về true nếu ký tự là chữ cái
`isblank()`	Kiểm tra ký tự có là whitespace ngăn từ
`iscntrl()`	Kiểm tra ký tự có là ký tự điều khiển
`isdigit()`	Kiểm tra ký tự có là chữ số
`isgraph()`	Kiểm tra ký tự có in được và không phải space
`islower()`	Kiểm tra ký tự có là chữ thường
`isprint()`	Kiểm tra ký tự có in được
`ispunct()`	Kiểm tra ký tự có là dấu câu
`isspace()`	Kiểm tra ký tự có là whitespace
`isupper()`	Kiểm tra ký tự có là chữ hoa
`isxdigit()`	Kiểm tra ký tự có là chữ số hex
`tolower()`	Chuyển chữ cái sang chữ thường
`toupper()`	Chuyển chữ cái sang chữ hoa

Bộ macro này tiện để kiểm tra xem một ký tự có thuộc một nhóm nào đó không, chẳng hạn chữ cái, chữ số, ký tự điều khiển, v.v.

Đáng ngạc nhiên là chúng nhận tham số `int` thay vì kiểu `char` nào đó. Lý do là để bạn tiện nhét `EOF` vào nếu có biểu diễn int của nó. Nếu không phải `EOF`, giá trị truyền vào phải biểu diễn được bằng `unsigned char`. Ngược lại thì (tèn tèn TÈÈÈN) undefined behavior. Cho nên quên việc nhét ký tự UTF-8 nhiều byte vào đây đi.

Bạn có thể tránh undefined behavior này kiểu portable bằng cách ép kiểu các đối số của những hàm này thành `(unsigned char)`. Công nhận nghe phiền và xấu. Các giá trị trong basic character set đều dùng được an toàn vì chúng là giá trị dương vừa vặn trong `unsigned char`.

Ngoài ra, hành vi của những hàm này cũng thay đổi theo locale.

Trong nhiều trang ở section này, tôi có kèm vài ví dụ. Chúng chạy ở locale "C", và có thể khác đi nếu bạn đã đặt locale khác.

Chú ý là ký tự rộng (wide character) có bộ hàm phân loại riêng của nó, nên đừng có thử dùng mấy hàm này trên `wchar_t`. Nghe không thì *biết tay*!

5.1 `isalnum()`

Kiểm tra ký tự có là chữ cái hoặc chữ số

Synopsis

```
#include <ctype.h>

int isalnum(int c);
```

Mô tả

Kiểm tra xem ký tự có là chữ cái (`A` - `Z` hoặc `a` - `z`) hoặc chữ số (`0` - `9`).

Tương đương với:

```
isalpha(c) || isdigit(c)
```

Giá trị trả về

Trả về true nếu ký tự là chữ cái (`A` - `Z` hoặc `a` - `z`) hoặc chữ số (`0` - `9`).

Ví dụ

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

int main(void)
{
    //             kiểm tra ký tự này
    //                      v
    printf("%s\n", isalnum('a')? "yes": "no");  // yes
    printf("%s\n", isalnum('B')? "yes": "no");  // yes
    printf("%s\n", isalnum('5')? "yes": "no");  // yes
    printf("%s\n", isalnum('?')? "yes": "no");  // no
}
```

Xem thêm

`isalpha()`, `isdigit()`

5.2 `isalpha()`

Trả về true nếu ký tự là chữ cái

Synopsis

```
#include <ctype.h>

int isalpha(int c);
```

Mô tả

Trả về true cho các ký tự chữ cái (`A` - `Z` hoặc `a` - `z`).

Về mặt kỹ thuật (và ở locale "C") tương đương với:

```
isupper(c) || islower(c)
```

Cực kỳ siêu kỹ thuật, vì tôi biết bạn đang khát khao đoạn này phức tạp một cách không cần thiết, nó còn có thể bao gồm một số ký tự đặc thù locale mà cái này đúng:

```
!iscntrl(c) && !isdigit(c) && !ispunct(c) && !isspace(c)
```

và cái này đúng:

```
isupper(c) || islower(c)
```

Giá trị trả về

Trả về true cho các ký tự chữ cái (`A` - `Z` hoặc `a` - `z`).

Hoặc bất kỳ thứ điên rồ nào khác trong phần mô tả ở trên.

Ví dụ

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

int main(void)
{
    //             kiểm tra ký tự này
    //                      v
    printf("%s\n", isalpha('a')? "yes": "no");  // yes
    printf("%s\n", isalpha('B')? "yes": "no");  // yes
    printf("%s\n", isalpha('5')? "yes": "no");  // no
    printf("%s\n", isalpha('?')? "yes": "no");  // no
}
```

Xem thêm

`isalnum()`

5.3 `isblank()`

Kiểm tra ký tự có là whitespace ngăn từ

Synopsis

```
#include <ctype.h>

int isblank(int c);
```

Mô tả

True nếu ký tự là whitespace dùng để ngăn các từ trên cùng một dòng.

Ví dụ, space (`' '`) hoặc tab ngang (`'\t'`). Locale khác có thể định nghĩa các ký tự blank khác.

Giá trị trả về

Trả về true nếu ký tự là whitespace dùng để ngăn các từ trên cùng một dòng.

Ví dụ

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

int main(void)
{
    //             kiểm tra ký tự này
    //                      v
    printf("%s\n", isblank(' ')? "yes": "no");   // yes
    printf("%s\n", isblank('\t')? "yes": "no");  // yes
    printf("%s\n", isblank('\n')? "yes": "no");  // no
    printf("%s\n", isblank('a')? "yes": "no");   // no
    printf("%s\n", isblank('?')? "yes": "no");   // no
}
```

Xem thêm

`isspace()`

5.4 `iscntrl()`

Kiểm tra ký tự có là ký tự điều khiển

Synopsis

```
#include <ctype.h>

int iscntrl(int c);
```

Mô tả

Ký tự điều khiển (control character) là ký tự không in được, tuỳ theo locale.

Với locale “C”, nghĩa là các ký tự điều khiển nằm trong khoảng 0x00 đến 0x1F (ký tự ngay trước SPACE) và 0x7F (ký tự DEL).

Nói chung nếu nó không phải ký tự ASCII (hoặc Unicode nhỏ hơn 128) in được, thì nó là ký tự điều khiển ở locale “C”.

Giá trị trả về

Trả về true nếu `c` là ký tự điều khiển.

Ví dụ

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

int main(void)
{
    //             kiểm tra ký tự này
    //                      v
    printf("%s\n", iscntrl('\t')? "yes": "no");  // yes (tab)
    printf("%s\n", iscntrl('\n')? "yes": "no");  // yes (newline)
    printf("%s\n", iscntrl('\r')? "yes": "no");  // yes (return)
    printf("%s\n", iscntrl('\a')? "yes": "no");  // yes (bell)
    printf("%s\n", iscntrl(' ')? "yes": "no");   // no
    printf("%s\n", iscntrl('a')? "yes": "no");   // no
    printf("%s\n", iscntrl('?')? "yes": "no");   // no
}
```

Xem thêm

`isgraph()`, `isprint()`

5.5 `isdigit()`

Kiểm tra ký tự có là chữ số

Synopsis

```
#include <ctype.h>

int isdigit(int c);
```

Mô tả

Kiểm tra xem `c` có là chữ số trong khoảng `0`-`9` không.

Giá trị trả về

Trả về true nếu ký tự là chữ số, không có gì bất ngờ.

Ví dụ

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

int main(void)
{
    //             kiểm tra ký tự này
    //                      v
    printf("%s\n", isdigit('0')? "yes": "no");   // yes
    printf("%s\n", isdigit('5')? "yes": "no");   // yes
    printf("%s\n", isdigit('a')? "yes": "no");   // no
    printf("%s\n", isdigit('B')? "yes": "no");   // no
    printf("%s\n", isdigit('?')? "yes": "no");   // no
```

```
}
```

Xem thêm

`isalnum()`, `isxdigit()`

5.6 `isgraph()`

Kiểm tra ký tự có in được và không phải space

Synopsis

```
#include <ctype.h>

int isgraph(int c);
```

Mô tả

Kiểm tra xem `c` có là ký tự in được nào mà không phải space (`' '`) không.

Giá trị trả về

Trả về true nếu `c` là ký tự in được nào mà không phải space (`' '`).

Ví dụ

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

int main(void)
{
    //             kiểm tra ký tự này
    //                      v
    printf("%s\n", isgraph('0')? "yes": "no");   // yes
    printf("%s\n", isgraph('a')? "yes": "no");   // yes
    printf("%s\n", isgraph('B')? "yes": "no");   // yes
    printf("%s\n", isgraph('?')? "yes": "no");   // yes
    printf("%s\n", isgraph(' ')? "yes": "no");   // no
    printf("%s\n", isgraph('\n')? "yes": "no");  // no
}
```

Xem thêm

`iscntrl()`, `isprint()`

5.7 `islower()`

Kiểm tra ký tự có là chữ thường

Synopsis

```
#include <ctype.h>

int islower(int c);
```

Mô tả

Kiểm tra xem ký tự có là chữ thường trong khoảng `a` - `z` không.

Ở các locale khác, có thể có những ký tự chữ thường khác. Trong mọi trường hợp, để là chữ thường, điều sau đây phải đúng:

```
!iscntrl(c) && !isdigit(c) && !ispunct(c) && !isspace(c)
```

Giá trị trả về

Trả về true nếu ký tự là chữ thường.

Ví dụ

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

int main(void)
{
    //             kiểm tra ký tự này
    //                      v
    printf("%s\n", islower('c')? "yes": "no");   // yes
    printf("%s\n", islower('0')? "yes": "no");   // no
    printf("%s\n", islower('B')? "yes": "no");   // no
    printf("%s\n", islower('?')? "yes": "no");   // no
    printf("%s\n", islower(' ')? "yes": "no");   // no
}
```

Xem thêm

`isupper()`, `isalpha()`, `toupper()`, `tolower()`

5.8 `isprint()`

Kiểm tra ký tự có in được

Synopsis

```
#include <ctype.h>

int isprint(int c);
```

Mô tả

Kiểm tra xem ký tự có in được không, kể cả space (`' '`). Nên là giống `isgraph()`, nhưng lần này space không bị bỏ rơi ngoài trời lạnh.

Giá trị trả về

Trả về true nếu ký tự in được, kể cả space (`' '`).

Ví dụ

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

int main(void)
{
    //             kiểm tra ký tự này
    //                      v
    printf("%s\n", isprint('c')? "yes": "no");   // yes
    printf("%s\n", isprint('0')? "yes": "no");   // yes
    printf("%s\n", isprint(' ')? "yes": "no");   // yes
    printf("%s\n", isprint('\r')? "yes": "no");  // no
}
```

Xem thêm

`isgraph()`, `iscntrl()`

5.9 `ispunct()`

Kiểm tra ký tự có là dấu câu

Synopsis

```
#include <ctype.h>

int ispunct(int c);
```

Mô tả

Kiểm tra xem ký tự có là dấu câu.

Ở locale "C", điều đó có nghĩa là:

```
!isspace(c) && !isalnum(c)
```

Ở các locale khác, có thể có các ký tự dấu câu khác (nhưng cũng không thể là space hoặc chữ-số).

Giá trị trả về

True nếu ký tự là dấu câu.

Ví dụ

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

int main(void)
{
```

```
    //             kiểm tra ký tự này
    //                      v
    printf("%s\n", ispunct(',')? "yes": "no");   // yes
    printf("%s\n", ispunct('!')? "yes": "no");   // yes
    printf("%s\n", ispunct('c')? "yes": "no");   // no
    printf("%s\n", ispunct('0')? "yes": "no");   // no
    printf("%s\n", ispunct(' ')? "yes": "no");   // no
    printf("%s\n", ispunct('\n')? "yes": "no");  // no
}
```

Xem thêm

`isspace()`, `isalnum()`

5.10 `isspace()`

Kiểm tra ký tự có là whitespace

Synopsis

```
#include <ctype.h>

int isspace(int c);
```

Mô tả

Kiểm tra xem `c` có là ký tự whitespace không. Bao gồm:

- Space (`' '`)
- Formfeed (`'\f'`)
- Newline (`'\n'`)
- Carriage Return (`'\r'`)
- Horizontal Tab (`'\t'`)
- Vertical Tab (`'\v'`)

Các locale khác có thể xác định các ký tự whitespace khác. `isalnum()` là false với mọi ký tự whitespace.

Giá trị trả về

True nếu ký tự là whitespace.

Ví dụ

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

int main(void)
{
    //             kiểm tra ký tự này
    //                      v
    printf("%s\n", isspace(' ')? "yes": "no");   // yes
    printf("%s\n", isspace('\n')? "yes": "no");  // yes
    printf("%s\n", isspace('\t')? "yes": "no");  // yes
    printf("%s\n", isspace(',')? "yes": "no");   // no
```

```
    printf("%s\n", isspace('!')? "yes": "no");   // no
    printf("%s\n", isspace('c')? "yes": "no");   // no
}
```

Xem thêm

`isblank()`

5.11 `isupper()`

Kiểm tra ký tự có là chữ hoa

Synopsis

```
#include <ctype.h>

int isupper(int c);
```

Mô tả

Kiểm tra xem ký tự có là chữ hoa trong khoảng `A` - `Z` không.

Ở các locale khác, có thể có những ký tự chữ hoa khác. Trong mọi trường hợp, để là chữ hoa, điều sau đây phải đúng:

```
!iscntrl(c) && !isdigit(c) && !ispunct(c) && !isspace(c)
```

Giá trị trả về

Trả về true nếu ký tự là chữ hoa.

Ví dụ

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

int main(void)
{
    //             kiểm tra ký tự này
    //                      v
    printf("%s\n", isupper('B')? "yes": "no");   // yes
    printf("%s\n", isupper('c')? "yes": "no");   // no
    printf("%s\n", isupper('0')? "yes": "no");   // no
    printf("%s\n", isupper('?')? "yes": "no");   // no
    printf("%s\n", isupper(' ')? "yes": "no");   // no
}
```

Xem thêm

`islower()`, `isalpha()`, `toupper()`, `tolower()`

5.12 `isxdigit()`

Kiểm tra ký tự có là chữ số hex

Synopsis

```
#include <ctype.h>

int isxdigit(int c);
```

Mô tả

Trả về true nếu ký tự là chữ số hex. Cụ thể là `0` - `9`, `a` - `f`, hoặc `A` - `F`.

Giá trị trả về

True nếu ký tự là chữ số hex.

Ví dụ

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

int main(void)
{
    //             kiểm tra ký tự này
    //                      v
    printf("%s\n", isxdigit('B')? "yes": "no");   // yes
    printf("%s\n", isxdigit('c')? "yes": "no");   // yes
    printf("%s\n", isxdigit('2')? "yes": "no");   // yes
    printf("%s\n", isxdigit('G')? "yes": "no");   // no
    printf("%s\n", isxdigit('?')? "yes": "no");   // no
}
```

Xem thêm

`isdigit()`

5.13 `tolower()`

Chuyển chữ cái sang chữ thường

Synopsis

```
#include <ctype.h>

int tolower(int c);
```

Mô tả

Nếu ký tự là chữ hoa (tức `isupper(c)` là true), hàm này trả về chữ thường tương ứng.

Các locale khác nhau có thể có tập chữ hoa và chữ thường khác nhau.

Giá trị trả về

Trả về giá trị chữ thường của một chữ hoa. Nếu ký tự không phải chữ hoa, trả về không đổi.

Ví dụ

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

int main(void)
{
    //             đổi ký tự này
    //                      v
    printf("%c\n", tolower('B'));  // b (đã chuyển thường!)
    printf("%c\n", tolower('e'));  // e (không đổi)
    printf("%c\n", tolower('!'));  // ! (không đổi)
}
```

Xem thêm

`toupper()`, `islower()`, `isupper()`

5.14 `toupper()`

Chuyển chữ cái sang chữ hoa

Synopsis

```
#include <ctype.h>

int toupper(int c);
```

Mô tả

Nếu ký tự là chữ thường (tức `islower(c)` là true), hàm này trả về chữ hoa tương ứng.

Các locale khác nhau có thể có tập chữ hoa và chữ thường khác nhau.

Giá trị trả về

Trả về giá trị chữ hoa của một chữ thường. Nếu ký tự không phải chữ thường, trả về không đổi.

Ví dụ

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

int main(void)
{
    //             đổi ký tự này
    //                      v
    printf("%c\n", toupper('B'));  // B (không đổi)
    printf("%c\n", toupper('e'));  // E (đã chuyển hoa!)
    printf("%c\n", toupper('!'));  // ! (không đổi)
```

```
}
```

Xem thêm

`tolower()`, `islower()`, `isupper()`

Chapter 6

`<errno.h>` Thông tin Lỗi

Biến	Mô tả
`errno`	Giữ trạng thái lỗi của lời gọi gần nhất

Header này định nghĩa duy nhất một biến[1], `errno`, có thể kiểm tra để xem đã xảy ra lỗi hay chưa.

`errno` được set thành `0` lúc khởi động, nhưng không có hàm thư viện nào set nó về `0`. Nếu bạn chỉ dựa vào nó để kiểm tra lỗi, set nó về `0` trước khi gọi rồi kiểm tra sau đó. Không chỉ vậy, nếu không có lỗi, mọi hàm thư viện sẽ để giá trị `errno` nguyên không đổi.

Tuy nhiên, thông thường bạn sẽ nhận được tín hiệu lỗi nào đó từ hàm được gọi rồi mới kiểm tra `errno` để xem chuyện gì đã sai.

Cái này thường dùng chung với `perror()` để lấy một thông báo lỗi dạng con người đọc được ứng với lỗi cụ thể.

Mẹo An Toàn Quan Trọng: Đừng bao giờ tự tạo biến của riêng bạn tên là `errno`—đó là undefined behavior.

Chú ý là spec C chỉ định nghĩa vài ba giá trị mà `errno` có thể nhận. Unix định nghĩa thêm kha khá[2], Windows cũng vậy[3].

6.1 `errno`

Giữ trạng thái lỗi của lời gọi gần nhất

Synopsis

```
errno   // Kiểu không xác định, nhưng gán được
```

Mô tả

Cho biết trạng thái lỗi của lời gọi gần nhất (chú ý không phải mọi lời gọi đều set giá trị này).

Giá trị	Mô tả
`0`	Không lỗi

[1] Thật ra nó chỉ cần là modifiable lvalue, nên không nhất thiết là biến. Nhưng bạn cứ xem nó như biến cũng được.
[2] https://man.archlinux.org/man/errno.3.en
[3] https://docs.microsoft.com/en-us/cpp/c-runtime-library/errno-constants?view=msvc-160

Giá trị	Mô tả
`EDOM`	Lỗi miền xác định (từ toán)
`EILSEQ`	Lỗi encoding (từ chuyển đổi ký tự)
`ERANGE`	Lỗi miền giá trị (từ toán)

Nếu bạn đang làm một loạt hàm toán, bạn có thể gặp `EDOM` hoặc `ERANGE`.

Với các hàm chuyển đổi ký tự multibyte/wide, bạn có thể thấy `EILSEQ`.

Và hệ thống của bạn có thể định nghĩa thêm bao nhiêu giá trị khác mà `errno` có thể nhận, tất cả đều bắt đầu bằng chữ `E`.

Chuyện Vui: bạn có thể dùng `EDOM`, `EILSEQ`, và `ERANGE` với các chỉ thị preprocessor như `#ifdef`. Nhưng thật lòng, tôi không chắc bạn sẽ làm vậy để làm gì ngoài việc kiểm tra sự tồn tại của chúng.

Ví dụ

Đoạn sau in ra thông báo lỗi, vì truyền `2.0` vào `acos()` là ngoài miền xác định của hàm.

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

int main(void)
{
    double x;

    errno = 0;       // Đảm bảo clear trước khi gọi

    x = acos(2.0);   // Đôí sô´không hợp lệ cho acos()

    if (errno == EDOM)
        perror("acos");
    else
        printf("Answer is %f\n", x);

    return 0;
}
```

Output:

```
acos: Numerical argument out of domain
```

Đoạn sau in ra thông báo lỗi (trên hệ của tôi), vì truyền `1e+30` vào `exp()` tạo ra kết quả ngoài miền giá trị của `double`.

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

int main(void)
{
    double x;

    errno = 0;       // Đảm bảo clear trước khi gọi

    x = exp(1e+30);  // Truyền vào sô´quá khổng lô`
```

```
    if (errno == ERANGE)
        perror("exp");
    else
        printf("Answer is %f\n", x);

    return 0;
}
```

Output:

```
exp: Numerical result out of range
```

Ví dụ này thử chuyển một ký tự không hợp lệ sang ký tự rộng (wide), thất bại. Nó set `errno` thành `EILSEQ`. Rồi ta dùng `perror()` để in ra thông báo lỗi.

```
#include <stdio.h>
#include <string.h>
#include <wchar.h>
#include <errno.h>
#include <locale.h>

int main(void)
{
    setlocale(LC_ALL, "");

    char *bad_str = "\xff";  // Có lẽ là char không hợp lệ ở locale này
    wchar_t wc;
    size_t result;
    mbstate_t ps;

    memset(&ps, 0, sizeof ps);

    result = mbrtowc(&wc, bad_str, 1, &ps);

    if (result == (size_t)(-1))
        perror("mbrtowc");  // mbrtowc: Illegal byte sequence
    else
        printf("Converted to L'%lc'\n", wc);

    return 0;
}
```

Output:

```
mbrtowc: Invalid or incomplete multibyte or wide character
```

Xem thêm

`perror()`, `mbrtoc16()`, `c16rtomb()`, `mbrtoc32()`, `c32rtomb()`, `fgetwc()`, `fputwc()`, `mbrtowc()`, `wcrtomb()`, `mbsrtowcs()`, `wcsrtombs()`, `<math.h>`,

Chapter 7

`<fenv.h>` Exception và Môi trường Dấu chấm động

Hàm	Mô tả
`feclearexcept()`	Xoá các exception dấu chấm động
`fegetexceptflag()`	Lưu các cờ exception dấu chấm động
`fesetexceptflag()`	Khôi phục các cờ exception dấu chấm động
`feraiseexcept()`	Raise một exception dấu chấm động bằng phần mềm
`fetestexcept()`	Kiểm tra xem một exception đã xảy ra hay chưa
`fegetround()`	Lấy hướng làm tròn
`fesetround()`	Đặt hướng làm tròn
`fegetenv()`	Lưu toàn bộ môi trường dấu chấm động
`fesetenv()`	Khôi phục toàn bộ môi trường dấu chấm động
`feholdexcept()`	Lưu trạng thái dấu chấm động và bật chế độ non-stop
`feupdateenv()`	Khôi phục môi trường dấu chấm động và áp dụng các exception gần nhất

7.1 Kiểu và Macro

Có hai kiểu được định nghĩa trong header này:

Kiểu	Mô tả
`fenv_t`	Toàn bộ môi trường dấu chấm động
`fexcept_t`	Một tập exception dấu chấm động

"Môi trường" có thể xem như trạng thái tại thời điểm hiện tại của hệ thống xử lý dấu chấm động: gồm exception, làm tròn, v.v. Nó là kiểu opaque, nên bạn không thể truy cập trực tiếp mà phải làm qua các hàm đúng cách.

Nếu các hàm nói đến có tồn tại trên hệ của bạn (có thể không!), thì bạn cũng sẽ có các macro sau được định nghĩa để biểu diễn các exception khác nhau:

Macro	Mô tả
`FE_DIVBYZERO`	Chia cho 0
`FE_INEXACT`	Kết quả không chính xác, bị làm tròn
`FE_INVALID`	Lỗi miền xác định
`FE_OVERFLOW`	Tràn trên

Macro	Mô tả
`FE_UNDERFLOW`	Tràn dưới
`FE_ALL_EXCEPT`	Tất cả ở trên gộp lại

Ý tưởng là bạn có thể bitwise-OR chúng với nhau để biểu diễn nhiều exception, ví dụ `FE_INVALID|FE_OVERFLOW`.

Các hàm bên dưới có tham số `excepts` sẽ nhận các giá trị này.

Xem `<math.h>` để biết hàm nào raise exception nào và khi nào.

7.2 Pragma

Bình thường C được tự do tối ưu đủ thứ chuyện có thể khiến các cờ không giống như bạn mong đợi. Vậy nên nếu bạn định dùng mớ này, nhớ set pragma này:

```
#pragma STDC FENV_ACCESS ON
```

Nếu bạn làm vậy ở scope toàn cục, nó có hiệu lực cho đến khi bạn tắt đi:

```
#pragma STDC FENV_ACCESS OFF
```

Nếu bạn làm ở block scope, nó phải đứng trước mọi statement hoặc declaration. Trong trường hợp đó, nó có hiệu lực cho đến khi block kết thúc (hoặc đến khi bị tắt đi rõ ràng).

Một cảnh báo: pragma này không được hỗ trợ trên cả hai compiler tôi đang có (gcc và clang) ở thời điểm viết, nên dù tôi có build mớ code dưới đây, nó không được test kỹ lắm.

7.3 `feclearexcept()`

Xoá các exception dấu chấm động

Synopsis

```
#include <fenv.h>

int feclearexcept(int excepts);
```

Mô tả

Nếu có exception dấu chấm động đã xảy ra, hàm này có thể xoá nó.

Set `excepts` thành danh sách exception nối bằng bitwise-OR cần xoá.

Truyền `0` thì không có tác dụng gì.

Giá trị trả về

Trả về `0` nếu thành công và khác 0 nếu thất bại.

Ví dụ

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

int main(void)
{
    #pragma STDC FENV_ACCESS ON

    double f = sqrt(-1);

    int r = feclearexcept(FE_INVALID);

    printf("%d %f\n", r, f);
}
```

Xem thêm

`feraiseexcept()`, `fetestexcept()`

7.4 `fegetexceptflag()` `fesetexceptflag()`

Lưu hoặc khôi phục các cờ exception dấu chấm động

Synopsis

```
#include <fenv.h>

int fegetexceptflag(fexcept_t *flagp, int excepts);

int fesetexceptflag(fexcept_t *flagp, int excepts);
```

Mô tả

Dùng các hàm này để lưu hoặc khôi phục môi trường dấu chấm động hiện tại trong một biến.

Set `excepts` thành tập exception bạn muốn lưu hoặc khôi phục trạng thái. Set thành `FE_ALL_EXCEPT` sẽ lưu hoặc khôi phục toàn bộ trạng thái.

Chú ý `fexcept_t` là kiểu opaque—bạn không biết bên trong nó có gì.

`excepts` có thể set thành 0 để không có tác dụng gì.

Giá trị trả về

Trả về `0` nếu thành công hoặc nếu `excepts` là 0.

Trả về khác 0 nếu thất bại.

Ví dụ

Chương trình này lưu trạng thái (trước khi có lỗi nào xảy ra), rồi cố ý gây lỗi miền xác định bằng cách thử tính $\sqrt{-1}$.

Sau đó, nó khôi phục trạng thái dấu chấm động về trước khi có lỗi, nhờ đó xoá lỗi đi.

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

int main(void)
{
    #pragma STDC FENV_ACCESS ON

    fexcept_t flag;

    fegetexceptflag(&flag, FE_ALL_EXCEPT);  // Lưu trạng thái

    double f = sqrt(-1);                    // Tôi đoán cái này không chạy được
    printf("%f\n", f);                      // "nan"

    if (fetestexcept(FE_INVALID))
        printf("1: Domain error\n");        // Dòng này in!
    else
        printf("1: No domain error\n");

    fesetexceptflag(&flag, FE_ALL_EXCEPT);  // Khôi phục về`trước khi có lỗĩ

    if (fetestexcept(FE_INVALID))
        printf("2: Domain error\n");
    else
        printf("2: No domain error\n");     // Dòng này in!
}
```

7.5 `feraiseexcept()`

Raise một exception dấu chấm động bằng phần mềm

Synopsis

```
#include <fenv.h>

int feraiseexcept(int excepts);
```

Mô tả

Cái này cố raise một exception dấu chấm động như thể nó đã xảy ra.

Bạn có thể chỉ định nhiều exception để raise.

Nếu `FE_UNDERFLOW` hoặc `FE_OVERFLOW` được raise, C *có thể* raise thêm `FE_INEXACT`.

Nếu `FE_UNDERFLOW` hoặc `FE_OVERFLOW` được raise cùng lúc với `FE_INEXACT`, thì `FE_UNDERFLOW` hoặc `FE_OVERFLOW` sẽ được raise *trước* `FE_INEXACT` phía sau hậu trường.

Thứ tự các exception khác được raise thì không xác định.

Giá trị trả về

Trả về `0` nếu mọi exception đều được raise hoặc nếu `excepts` là `0`.

Trả về khác 0 trong trường hợp khác.

Ví dụ

Đoạn code này cố ý raise exception chia cho 0 rồi phát hiện nó.

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

int main(void)
{
    #pragma STDC FENV_ACCESS ON

    feraiseexcept(FE_DIVBYZERO);

    if (fetestexcept(FE_DIVBYZERO) == FE_DIVBYZERO)
        printf("Detected division by zero\n");  // Dòng này in!!
    else
        printf("This is fine.\n");
}
```

Xem thêm

`feclearexcept()`, `fetestexcept()`

7.6 `fetestexcept()`

Kiểm tra xem một exception đã xảy ra hay chưa

Synopsis

```
#include <fenv.h>

int fetestexcept(int excepts);
```

Mô tả

Đặt các exception bạn muốn kiểm tra vào `excepts`, bitwise-OR chúng lại với nhau.

Giá trị trả về

Trả về bitwise-OR của các exception đã được raise.

Ví dụ

Đoạn code này cố ý raise exception chia cho 0 rồi phát hiện nó.

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

int main(void)
{
    #pragma STDC FENV_ACCESS ON

    feraiseexcept(FE_DIVBYZERO);
```

```
    if (fetestexcept(FE_DIVBYZERO) == FE_DIVBYZERO)
        printf("Detected division by zero\n");  // Dòng này in!!
    else
        printf("This is fine.\n");
}
```

Xem thêm

`feclearexcept()`, `feraiseexcept()`

7.7 `fegetround()` `fesetround()`

Lấy hoặc đặt hướng làm tròn

Synopsis

```
#include <fenv.h>

int fegetround(void);

int fesetround(int round);
```

Mô tả

Dùng mấy hàm này để lấy hoặc đặt hướng làm tròn được dùng bởi một đống hàm toán.

Cơ bản là khi một hàm "làm tròn" một số, nó muốn biết làm tròn thế nào. Mặc định, nó làm theo cách ta hay mong đợi: nếu phần phân số nhỏ hơn 0.5, làm tròn xuống về phía 0, ngược lại làm tròn lên xa 0.

Macro	Mô tả
`FE_TONEAREST`	Làm tròn về số nguyên gần nhất, mặc định
`FE_TOWARDZERO`	Luôn làm tròn về phía 0
`FE_DOWNWARD`	Làm tròn về số nguyên nhỏ hơn kế tiếp
`FE_UPWARD`	Làm tròn về số nguyên lớn hơn kế tiếp

Một số hiện thực không hỗ trợ làm tròn. Nếu có, các macro trên sẽ được định nghĩa.

Chú ý hàm `round()` luôn là "về-gần-nhất" và không quan tâm đến chế độ làm tròn.

Giá trị trả về

`fegetround()` trả về hướng làm tròn hiện tại, hoặc giá trị âm nếu lỗi.

`fesetround()` trả về 0 nếu thành công, khác 0 nếu thất bại.

Ví dụ

Ví dụ này làm tròn vài số

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

```
// Hàm phụ in ra chế độ làm tròn
const char *rounding_mode_str(int mode)
{
    switch (mode) {
        case FE_TONEAREST:  return "FE_TONEAREST";
        case FE_TOWARDZERO: return "FE_TOWARDZERO";
        case FE_DOWNWARD:   return "FE_DOWNWARD";
        case FE_UPWARD:     return "FE_UPWARD";
    }

    return "Unknown";
}

int main(void)
{
    #pragma STDC FENV_ACCESS ON

    int rm;

    rm = fegetround();

    printf("%s\n", rounding_mode_str(rm));    // In chế độ hiện tại
    printf("%f %f\n", rint(2.1), rint(2.7));  // Thử làm tròn

    fesetround(FE_TOWARDZERO);                // Đổi chế độ

    rm = fegetround();

    printf("%s\n", rounding_mode_str(rm));    // In ra
    printf("%f %f\n", rint(2.1), rint(2.7));  // Thử lại xem!
}
```

Output:

```
FE_TONEAREST
2.000000 3.000000
FE_TOWARDZERO
2.000000 2.000000
```

Xem thêm

`nearbyint()`, `nearbyintf()`, `nearbyintl()`, `rint()`, `rintf()`, `rintl()`, `lrint()`, `lrintf()`, `lrintl()`, `llrint()`, `llrintf()`, `llrintl()`

7.8 `fegetenv()` `fesetenv()`

Lưu hoặc khôi phục toàn bộ môi trường dấu chấm động

Synopsis

```
#include <fenv.h>

int fegetenv(fenv_t *envp);
int fesetenv(const fenv_t *envp);
```

Mô tả

Bạn có thể lưu môi trường (exception, hướng làm tròn, v.v.) bằng cách gọi `fegetenv()` và khôi phục bằng `fesetenv()`.

Dùng cái này nếu bạn muốn khôi phục trạng thái sau khi gọi hàm, nghĩa là giấu đi khỏi caller rằng có vài exception dấu chấm động hoặc thay đổi đã xảy ra.

Giá trị trả về

`fegetenv()` và `fesetenv()` trả về `0` nếu thành công, khác 0 trong trường hợp khác.

Ví dụ

Ví dụ này lưu môi trường, quậy với rounding và exception, rồi khôi phục lại. Sau khi môi trường được khôi phục, ta thấy rounding đã về mặc định và exception đã bị xoá.

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

void show_status(void)
{
    printf("Rounding is FE_TOWARDZERO: %d\n",
           fegetround() == FE_TOWARDZERO);

    printf("FE_DIVBYZERO is set: %d\n",
           fetestexcept(FE_DIVBYZERO) != 0);
}

int main(void)
{
    #pragma STDC FENV_ACCESS ON

    fenv_t env;

    fegetenv(&env);  // Lưu môi trường

    fesetround(FE_TOWARDZERO);    // Đổi rounding
    feraiseexcept(FE_DIVBYZERO);  // Raise exception

    show_status();

    fesetenv(&env);  // Khôi phục môi trường

    show_status();
}
```

Output:

```
Rounding is FE_TOWARDZERO: 1
FE_DIVBYZERO is set: 1
Rounding is FE_TOWARDZERO: 0
FE_DIVBYZERO is set: 0
```

Xem thêm

`feholdexcept()`, `feupdateenv()`

7.9 `feholdexcept()`

Lưu trạng thái dấu chấm động và bật chế độ non-stop

Synopsis

```
#include <fenv.h>

int feholdexcept(fenv_t *envp);
```

Mô tả

Cái này giống `fegetenv()` chỉ khác là nó cập nhật môi trường hiện tại sang chế độ *non-stop*, tức là nó sẽ không dừng lại ở bất kỳ exception nào.

Nó giữ nguyên trạng thái này cho đến khi bạn khôi phục trạng thái bằng `fesetenv()` hoặc `feupdateenv()`.

Giá trị trả về

Ví dụ

Ví dụ này lưu môi trường và vào chế độ non-stop, quậy với rounding và exception, rồi khôi phục lại. Sau khi khôi phục môi trường, ta thấy rounding đã về mặc định và exception đã bị xoá. Ta cũng sẽ không còn ở chế độ non-stop nữa.

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

void show_status(void)
{
    printf("Rounding is FE_TOWARDZERO: %d\n",
           fegetround() == FE_TOWARDZERO);

    printf("FE_DIVBYZERO is set: %d\n",
           fetestexcept(FE_DIVBYZERO) != 0);
}

int main(void)
{
    #pragma STDC FENV_ACCESS ON

    fenv_t env;

    // Lưu môi trường và không dừng lại khi gặp exception
    feholdexcept(&env);

    fesetround(FE_TOWARDZERO);    // Đổi rounding
    feraiseexcept(FE_DIVBYZERO);  // Raise exception

    show_status();

    fesetenv(&env);  // Khôi phục môi trường
```

```
    show_status();
}
```

Xem thêm

`fegetenv()`, `fesetenv()`, `feupdateenv()`

7.10 `feupdateenv()`

Khôi phục môi trường dấu chấm động và áp dụng các exception gần nhất

Synopsis

```
#include <fenv.h>

int feupdateenv(const fenv_t *envp);
```

Mô tả

Cái này giống `fesetenv()` chỉ khác là nó chỉnh lại môi trường truyền vào sao cho được cập nhật với các exception đã xảy ra trong lúc đó.

Ví dụ bạn có một hàm có thể raise exception, nhưng bạn muốn giấu chúng đi khỏi caller. Một lựa chọn là:

1. Lưu môi trường bằng `fegetenv()` hoặc `feholdexcept()`.
2. Làm gì thì làm mà có thể raise exception.
3. Khôi phục môi trường bằng `fesetenv()`, qua đó giấu đi các exception đã xảy ra ở bước 2.

Nhưng cách đó giấu *toàn bộ* exception. Lỡ bạn chỉ muốn giấu vài cái thì sao? Bạn có thể dùng `feupdateenv()` như sau:

1. Lưu môi trường bằng `fegetenv()` hoặc `feholdexcept()`.
2. Làm gì thì làm mà có thể raise exception.
3. Gọi `feclearexcept()` để xoá các exception bạn muốn giấu khỏi caller.
4. Gọi `feupdateenv()` để khôi phục môi trường trước đó và cập nhật nó với các exception khác đã xảy ra.

Nên nó giống một cách khôi phục môi trường có năng lực hơn so với chỉ đơn giản `fegetenv()` / `fesetenv()`.

Giá trị trả về

Trả về `0` nếu thành công, khác 0 trong trường hợp khác.

Ví dụ

Chương trình này lưu trạng thái, raise vài exception, rồi xoá một trong các exception, rồi khôi phục và cập nhật trạng thái.

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

void show_status(void)
{
    printf("FE_DIVBYZERO: %d\n", fetestexcept(FE_DIVBYZERO) != 0);
```

```
    printf("FE_INVALID  : %d\n", fetestexcept(FE_INVALID) != 0);
    printf("FE_OVERFLOW : %d\n\n", fetestexcept(FE_OVERFLOW) != 0);
}

int main(void)
{
    #pragma STDC FENV_ACCESS ON

    fenv_t env;

    feholdexcept(&env);  // Lưu môi trường

    // Giả vờ có chút toán tệ xảy ra ở đây:
    feraiseexcept(FE_DIVBYZERO);  // Raise exception
    feraiseexcept(FE_INVALID);    // Raise exception
    feraiseexcept(FE_OVERFLOW);   // Raise exception

    show_status();

    feclearexcept(FE_INVALID);

    feupdateenv(&env);  // Khôi phục môi trường

    show_status();
}
```

Trong output, lúc đầu ta không có exception nào. Rồi ta có ba cái đã raise. Sau đó khi khôi phục/cập nhật môi trường, ta thấy cái đã xoá (`FE_INVALID`) không được áp dụng:

```
FE_DIVBYZERO: 0
FE_INVALID  : 0
FE_OVERFLOW : 0

FE_DIVBYZERO: 1
FE_INVALID  : 1
FE_OVERFLOW : 1

FE_DIVBYZERO: 1
FE_INVALID  : 0
FE_OVERFLOW : 1
```

Xem thêm

`fegetenv()` , `fesetenv()` , `feholdexcept()` , `feclearexcept()`

Chapter 8

`<float.h>` Giới hạn Dấu chấm động

Macro	Độ lớn tối thiểu	Mô tả
`FLT_ROUNDS`		Chế độ làm tròn hiện tại
`FLT_EVAL_METHOD`		Các kiểu dùng để đánh giá
`FLT_HAS_SUBNORM`		Hỗ trợ subnormal cho `float`
`DBL_HAS_SUBNORM`		Hỗ trợ subnormal cho `double`
`LDBL_HAS_SUBNORM`		Hỗ trợ subnormal cho `long double`
`FLT_RADIX`	`2`	Cơ số dấu chấm động (radix)
`FLT_MANT_DIG`		Số chữ số cơ số `FLT_RADIX` trong `float`
`DBL_MANT_DIG`		Số chữ số cơ số `FLT_RADIX` trong `double`
`LDBL_MANT_DIG`		Số chữ số cơ số `FLT_RADIX` trong `long double`
`FLT_DECIMAL_DIG`	`6`	Số chữ số thập phân cần để mã hoá một `float`
`DBL_DECIMAL_DIG`	`10`	Số chữ số thập phân cần để mã hoá một `double`
`LDBL_DECIMAL_DIG`	`10`	Số chữ số thập phân cần để mã hoá một `long double`
`DECIMAL_DIG`	`10`	Số chữ số thập phân cần để mã hoá số dấu chấm động rộng nhất được hỗ trợ
`FLT_DIG`	`6`	Số chữ số thập phân có thể lưu an toàn trong `float`
`DBL_DIG`	`10`	Số chữ số thập phân có thể lưu an toàn trong `double`
`LDBL_DIG`	`10`	Số chữ số thập phân có thể lưu an toàn trong `long double`
`FLT_MIN_EXP`		`FLT_RADIX` mũ `FLT_MIN_EXP-1` là `float` chuẩn hoá nhỏ nhất
`DBL_MIN_EXP`		`FLT_RADIX` mũ `DBL_MIN_EXP-1` là `double` chuẩn hoá nhỏ nhất
`LDBL_MIN_EXP`		`FLT_RADIX` mũ `LDBL_MIN_EXP-1` là `long double` chuẩn hoá nhỏ nhất
`FLT_MIN_10_EXP`	`-37`	Số mũ tối thiểu sao cho `10` luỹ thừa số này là `float` chuẩn hoá
`DBL_MIN_10_EXP`	`-37`	Số mũ tối thiểu sao cho `10` luỹ thừa số này là `double` chuẩn hoá
`LDBL_MIN_10_EXP`	`-37`	Số mũ tối thiểu sao cho `10` luỹ thừa số này là `long_double` chuẩn hoá
`FLT_MAX_EXP`		`FLT_RADIX` mũ `FLT_MAX_EXP-1` là `float` hữu hạn lớn nhất
`DBL_MAX_EXP`		`FLT_RADIX` mũ `DBL_MAX_EXP-1` là `double` hữu hạn lớn nhất
`LDBL_MAX_EXP`		`FLT_RADIX` mũ `LDBL_MAX_EXP-1` là `long double` hữu hạn lớn nhất

Macro	Độ lớn tối thiểu	Mô tả
`FLT_MAX_10_EXP`	`-37`	Số mũ tối thiểu sao cho `10` luỹ thừa số này là `float` hữu hạn
`DBL_MAX_10_EXP`	`-37`	Số mũ tối thiểu sao cho `10` luỹ thừa số này là `double` hữu hạn
`LDBL_MAX_10_EXP`	`-37`	Số mũ tối thiểu sao cho `10` luỹ thừa số này là `long_double` hữu hạn
`FLT_MAX`	`1E+37`	`float` hữu hạn lớn nhất
`DBL_MAX`	`1E+37`	`double` hữu hạn lớn nhất
`LDBL_MAX`	`1E+37`	`long double` hữu hạn lớn nhất

Macro	Giá trị tối đa	Mô tả
`FLT_EPSILON`	`1E-5`	Chênh lệch giữa 1 và `float` biểu diễn được lớn hơn kế tiếp
`DBL_EPSILON`	`1E-9`	Chênh lệch giữa 1 và `double` biểu diễn được lớn hơn kế tiếp
`LDBL_EPSILON`	`1E-9`	Chênh lệch giữa 1 và `long double` biểu diễn được lớn hơn kế tiếp
`FLT_MIN`	`1E-37`	`float` chuẩn hoá dương nhỏ nhất
`DBL_MIN`	`1E-37`	`double` chuẩn hoá dương nhỏ nhất
`LDBL_MIN`	`1E-37`	`long double` chuẩn hoá dương nhỏ nhất
`FLT_TRUE_MIN`	`1E-37`	`float` dương nhỏ nhất
`DBL_TRUE_MIN`	`1E-37`	`double` dương nhỏ nhất
`LDBL_TRUE_MIN`	`1E-37`	`long double` dương nhỏ nhất

Các giá trị tối thiểu và tối đa ở đây lấy từ spec—đó là mức tối thiểu bạn có thể mong đợi trên mọi nền tảng. Máy siêu xịn của bạn có thể làm tốt hơn nữa!

8.1 Bối cảnh

Spec cho phép khá thoáng trong chuyện C biểu diễn số dấu chấm động như thế nào. Header này đánh vần ra các giới hạn của những con số đó.

Nó đưa ra một mô hình có thể mô tả bất kỳ số dấu chấm động nào mà tôi *biết* chắc chắn bạn sẽ mê tít. Trông như vầy:

$$x = sb^e \sum_{k=1}^{p} f_k b^{-k}, e_{min} \leq e \leq e_{max}$$

trong đó:

Biến	Ý nghĩa
s	Dấu, -1 hoặc 1
b	Cơ số (radix), trên hệ của bạn chắc là 2
e	Số mũ
p	Độ chính xác: có bao nhiêu chữ số cơ số b trong số
f_k	Từng chữ số của số, tức significand

Nhưng tạm thời cứ bỏ qua mớ đó đi cho nhẹ đầu.

Giả sử máy của bạn dùng cơ số 2 cho dấu chấm động (chắc là có). Và trong ví dụ dưới đây các số 1-và-0 là ở hệ nhị phân, còn lại ở thập phân.

Tóm lại là bạn có thể có các số dấu chấm động như trong ví dụ này:

$-0.10100101 \times 2^5 = -10100.101 = -20.625$

Đó là phần phân số nhân với cơ số luỹ thừa số mũ. Số mũ điều khiển dấu thập phân nằm ở đâu. Nó "trôi" loanh quanh!

8.2 Chi tiết `FLT_ROUNDS`

Cái này cho bạn biết chế độ làm tròn. Có thể thay đổi bằng một lời gọi `fesetround()`.

Chế độ	Mô tả
`-1`	Không xác định được
`0`	Về phía 0
`1`	Về gần nhất
`2`	Về phía dương vô cực
`3`	Về phía âm vô cực... và xa hơn nữa!

Không giống mọi macro khác trong header này, `FLT_ROUNDS` có thể không phải là biểu thức hằng.

8.3 Chi tiết `FLT_EVAL_METHOD`

Cái này cơ bản cho bạn biết các giá trị dấu chấm động được promote sang kiểu khác thế nào trong biểu thức.

Phương pháp	Mô tả
`-1`	Không xác định được
`0`	Đánh giá mọi phép toán và hằng ở độ chính xác của kiểu tương ứng
`1`	Đánh giá phép toán `float` và `double` như `double`, phép toán `long double` như `long double`
`2`	Đánh giá mọi phép toán và hằng như `long double`

8.4 Số Subnormal

Các macro `FLT_HAS_SUBNORM`, `DBL_HAS_SUBNORM`, và `LDBL_HAS_SUBNORM` đều cho bạn biết các kiểu đó có hỗ trợ số subnormal[1] không.

Giá trị	Mô tả
`-1`	Không xác định được
`0`	Subnormal không được hỗ trợ cho kiểu này
`1`	Subnormal được hỗ trợ cho kiểu này

8.5 Tôi dùng được bao nhiêu chữ số thập phân?

Còn tuỳ bạn muốn làm gì.

An toàn là nếu bạn không bao giờ dùng quá `FLT_DIG` chữ số cơ số 10 trong `float` của mình, bạn ổn. (Tương tự với `DBL_DIG` và `LDBL_DIG` cho kiểu của chúng.)

Và "dùng" tôi nói đây là in ra, có trong code, đọc từ bàn phím, v.v.

[1] https://en.wikipedia.org/wiki/Subnormal_number

Bạn có thể in ra từng ấy chữ số thập phân bằng `printf()` và format specifier `%g`:

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

int main(void)
{
    float pi = 3.14159265358979323846264338327950288419 71;

    // Với %g hoặc %G, precision chỉ số´chữ số´có nghĩa:

    printf("%.*g\n", FLT_DIG, pi);  // Với tôi: 3.14159

    // Nhưng %f in quá nhiều, vì precision là số´chữ số´bên phải
    // dấú thập phân--nó không đếm các chữ số´bên trái:

    printf("%.*f\n", FLT_DIG, pi);  // Với tôi: 3.14159... 3 ???
}
```

Đấy là hết, nhưng mời đón xem phần kết hấp dẫn của "Tôi dùng được bao nhiêu chữ số thập phân?"

Vì cơ số 10 và cơ số 2 (`FLT_RADIX` tiêu biểu của bạn) không ăn ý với nhau lắm, bạn có thể thực sự có nhiều hơn `FLT_DIG` chữ số trong `float`; các bit lưu trữ kéo dài thêm chút nữa. Nhưng chúng có thể làm tròn theo cách bạn không ngờ tới.

Nhưng nếu bạn muốn chuyển một số dấu chấm động sang cơ số 10 rồi có thể chuyển nó ngược lại thành cùng một số dấu chấm động y hệt, bạn sẽ cần `FLT_DECIMAL_DIG` chữ số từ `float` để đảm bảo mấy bit lưu trữ dư ra được thể hiện đầy đủ. (Và `DBL_DECIMAL_DIG` và `LDBL_DECIMAL_DIG` cho các kiểu tương ứng.)

Dưới đây là ví dụ output cho thấy giá trị được lưu có thể có vài chữ số thập phân thừa ở cuối.

```
#include <stdio.h>
#include <math.h>
#include <assert.h>
#include <float.h>

int main(void)
{
    printf("FLT_DIG = %d\n", FLT_DIG);
    printf("FLT_DECIMAL_DIG = %d\n\n", FLT_DECIMAL_DIG);

    assert(FLT_DIG == 6);  // Code dưới giả định điều này

    for (float x = 0.123456; x < 0.12346; x += 0.000001) {
        printf("As written: %.*g\n", FLT_DIG, x);
        printf("As stored:  %.*g\n\n", FLT_DECIMAL_DIG, x);
    }
}
```

Và output trên máy của tôi, bắt đầu tại `0.123456` và tăng dần `0.000001` mỗi lần:

```
FLT_DIG = 6
FLT_DECIMAL_DIG = 9

As written: 0.123456
As stored:  0.123456001
```

```
As written: 0.123457
As stored:  0.123457

As written: 0.123458
As stored:  0.123457998

As written: 0.123459
As stored:  0.123458996

As written: 0.12346
As stored:  0.123459995
```

Bạn có thể thấy giá trị lưu không phải lúc nào cũng là giá trị ta mong đợi, vì cơ số 2 không biểu diễn chính xác được mọi phân số cơ số 10. Tốt nhất nó có thể làm là lưu thêm vài chữ số rồi làm tròn.

Cũng để ý là dù ta đã cố dừng vòng `for` *trước* `0.123460`, nó thực sự chạy qua cả giá trị đó vì bản lưu của số đó là `0.123459995`, vẫn nhỏ hơn `0.123460`.

Số dấu chấm động vui ghê nhỉ?

8.6 Ví dụ Toàn diện

Đây là chương trình in ra chi tiết cho một máy cụ thể:

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

int main(void)
{
    printf("FLT_RADIX: %d\n", FLT_RADIX);
    printf("FLT_ROUNDS: %d\n", FLT_ROUNDS);
    printf("FLT_EVAL_METHOD: %d\n", FLT_EVAL_METHOD);
    printf("DECIMAL_DIG: %d\n\n", DECIMAL_DIG);

    printf("FLT_HAS_SUBNORM: %d\n", FLT_HAS_SUBNORM);
    printf("FLT_MANT_DIG: %d\n", FLT_MANT_DIG);
    printf("FLT_DECIMAL_DIG: %d\n", FLT_DECIMAL_DIG);
    printf("FLT_DIG: %d\n", FLT_DIG);
    printf("FLT_MIN_EXP: %d\n", FLT_MIN_EXP);
    printf("FLT_MIN_10_EXP: %d\n", FLT_MIN_10_EXP);
    printf("FLT_MAX_EXP: %d\n", FLT_MAX_EXP);
    printf("FLT_MAX_10_EXP: %d\n", FLT_MAX_10_EXP);
    printf("FLT_MIN: %.*e\n", FLT_DECIMAL_DIG, FLT_MIN);
    printf("FLT_MAX: %.*e\n", FLT_DECIMAL_DIG, FLT_MAX);
    printf("FLT_EPSILON: %.*e\n", FLT_DECIMAL_DIG, FLT_EPSILON);
    printf("FLT_TRUE_MIN: %.*e\n\n", FLT_DECIMAL_DIG, FLT_TRUE_MIN);

    printf("DBL_HAS_SUBNORM: %d\n", DBL_HAS_SUBNORM);
    printf("DBL_MANT_DIG: %d\n", DBL_MANT_DIG);
    printf("DBL_DECIMAL_DIG: %d\n", DBL_DECIMAL_DIG);
    printf("DBL_DIG: %d\n", DBL_DIG);
    printf("DBL_MIN_EXP: %d\n", DBL_MIN_EXP);
    printf("DBL_MIN_10_EXP: %d\n", DBL_MIN_10_EXP);
    printf("DBL_MAX_EXP: %d\n", DBL_MAX_EXP);
    printf("DBL_MAX_10_EXP: %d\n", DBL_MAX_10_EXP);
    printf("DBL_MIN: %.*e\n", DBL_DECIMAL_DIG, DBL_MIN);
    printf("DBL_MAX: %.*e\n", DBL_DECIMAL_DIG, DBL_MAX);
```

```
    printf("DBL_EPSILON: %.*e\n", DBL_DECIMAL_DIG, DBL_EPSILON);
    printf("DBL_TRUE_MIN: %.*e\n\n", DBL_DECIMAL_DIG, DBL_TRUE_MIN);

    printf("LDBL_HAS_SUBNORM: %d\n", LDBL_HAS_SUBNORM);
    printf("LDBL_MANT_DIG: %d\n", LDBL_MANT_DIG);
    printf("LDBL_DECIMAL_DIG: %d\n", LDBL_DECIMAL_DIG);
    printf("LDBL_DIG: %d\n", LDBL_DIG);
    printf("LDBL_MIN_EXP: %d\n", LDBL_MIN_EXP);
    printf("LDBL_MIN_10_EXP: %d\n", LDBL_MIN_10_EXP);
    printf("LDBL_MAX_EXP: %d\n", LDBL_MAX_EXP);
    printf("LDBL_MAX_10_EXP: %d\n", LDBL_MAX_10_EXP);
    printf("LDBL_MIN: %.*Le\n", LDBL_DECIMAL_DIG, LDBL_MIN);
    printf("LDBL_MAX: %.*Le\n", LDBL_DECIMAL_DIG, LDBL_MAX);
    printf("LDBL_EPSILON: %.*Le\n", LDBL_DECIMAL_DIG, LDBL_EPSILON);
    printf("LDBL_TRUE_MIN: %.*Le\n\n", LDBL_DECIMAL_DIG, LDBL_TRUE_MIN);

    printf("sizeof(float): %zu\n", sizeof(float));
    printf("sizeof(double): %zu\n", sizeof(double));
    printf("sizeof(long double): %zu\n", sizeof(long double));
}
```

Và đây là output trên máy của tôi:

```
FLT_RADIX: 2
FLT_ROUNDS: 1
FLT_EVAL_METHOD: 0
DECIMAL_DIG: 21

FLT_HAS_SUBNORM: 1
FLT_MANT_DIG: 24
FLT_DECIMAL_DIG: 9
FLT_DIG: 6
FLT_MIN_EXP: -125
FLT_MIN_10_EXP: -37
FLT_MAX_EXP: 128
FLT_MAX_10_EXP: 38
FLT_MIN: 1.175494351e-38
FLT_MAX: 3.402823466e+38
FLT_EPSILON: 1.192092896e-07
FLT_TRUE_MIN: 1.401298464e-45

DBL_HAS_SUBNORM: 1
DBL_MANT_DIG: 53
DBL_DECIMAL_DIG: 17
DBL_DIG: 15
DBL_MIN_EXP: -1021
DBL_MIN_10_EXP: -307
DBL_MAX_EXP: 1024
DBL_MAX_10_EXP: 308
DBL_MIN: 2.22507385850720138e-308
DBL_MAX: 1.79769313486231571e+308
DBL_EPSILON: 2.22044604925031308e-16
DBL_TRUE_MIN: 4.94065645841246544e-324

LDBL_HAS_SUBNORM: 1
LDBL_MANT_DIG: 64
LDBL_DECIMAL_DIG: 21
```

```
LDBL_DIG: 18
LDBL_MIN_EXP: -16381
LDBL_MIN_10_EXP: -4931
LDBL_MAX_EXP: 16384
LDBL_MAX_10_EXP: 4932
LDBL_MIN: 3.362103143112093506263e-4932
LDBL_MAX: 1.189731495357231765021e+4932
LDBL_EPSILON: 1.084202172485504434007e-19
LDBL_TRUE_MIN: 3.645199531882474602528e-4951

sizeof(float): 4
sizeof(double): 8
sizeof(long double): 16
```

Chapter 9

`<inttypes.h>` Các phép chuyển đổi số nguyên thêm

Hàm	Mô tả
`imaxabs()`	Tính giá trị tuyệt đối của một `intmax_t`
`imaxdiv()`	Tính thương và số dư của các `intmax_t`
`strtoimax()`	Chuyển string sang kiểu `intmax_t`
`strtoumax()`	Chuyển string sang kiểu `uintmax_t`
`wcstoimax()`	Chuyển wide string sang kiểu `intmax_t`
`wcstoumax()`	Chuyển wide string sang kiểu `uintmax_t`

Header này làm các phép chuyển đổi sang số nguyên cỡ lớn nhất, chia với số nguyên cỡ lớn nhất, và cũng cung cấp format specifier cho `printf()` và `scanf()` cho một loạt kiểu được định nghĩa trong `<stdint.h>`.

Header `<stdint.h>` được include bởi header này.

9.1 Macro

Mấy macro này là để giúp `printf()` và `scanf()` khi bạn dùng một kiểu như `int_least16_t`... bạn dùng format specifier gì?

Bắt đầu với `printf()`—mấy macro này bắt đầu bằng `PRI` rồi theo sau là format specifier mà bạn thường dùng cho kiểu đó. Cuối cùng, thêm số bit.

Ví dụ, format specifier cho số nguyên 64-bit là `PRId64`—chữ `d` là vì thường bạn in số nguyên với `"%d"`.

Một số nguyên không dấu 16-bit có thể được in với `PRIu16`.

Mấy macro này mở rộng thành string literal. Ta có thể tận dụng việc C tự động nối các string literal kề nhau và dùng các specifier này như sau:

```
#include <stdio.h>     // cho printf()
#include <inttypes.h>

int main(void)
{
    int16_t x = 32;

    printf("The value is %" PRId16 "!\n", x);
```

```
}
```

Không có gì ma thuật xảy ra ở dòng 8 bên trên đâu. Thật vậy, nếu tôi in giá trị của macro:

```
printf("%s\n", PRId16);
```

ta nhận được cái này trên hệ của tôi:

```
hd
```

là một format specifier của `printf()` có nghĩa "số nguyên có dấu ngắn".

Nên quay lại dòng 8, sau khi nối string literal, nó y hệt như tôi đã gõ:

```
    printf("The value is %hd!\n", x);
```

Đây là bảng mọi macro bạn có thể dùng cho format specifier của `printf()` ... thay số bit vào *N*, thường là 8, 16, 32, hoặc 64.

`PRId`*N*	`PRIdLEAST`*N*	`PRIdFAST`*N*	`PRIdMAX`	`PRIdPTR`
`PRIi`*N*	`PRIiLEAST`*N*	`PRIiFAST`*N*	`PRIiMAX`	`PRIiPTR`
`PRIo`*N*	`PRIoLEAST`*N*	`PRIoFAST`*N*	`PRIoMAX`	`PRIoPTR`
`PRIu`*N*	`PRIuLEAST`*N*	`PRIuFAST`*N*	`PRIuMAX`	`PRIuPTR`
`PRIx`*N*	`PRIxLEAST`*N*	`PRIxFAST`*N*	`PRIxMAX`	`PRIxPTR`
`PRIX`*N*	`PRIXLEAST`*N*	`PRIXFAST`*N*	`PRIXMAX`	`PRIXPTR`

Để ý lần nữa là chữ chữ thường ở giữa đại diện cho các format specifier thông thường bạn truyền cho `printf()`: `d`, `i`, `o`, `u`, `x`, và `X`.

Và ta có một bộ macro tương tự cho `scanf()` để đọc các kiểu này:

`SCNd`*N*	`SCNdLEAST`*N*	`SCNdFAST`*N*	`SCNdMAX`	`SCNdPTR`
`SCNi`*N*	`SCNiLEAST`*N*	`SCNiFAST`*N*	`SCNiMAX`	`SCNiPTR`
`SCNo`*N*	`SCNoLEAST`*N*	`SCNoFAST`*N*	`SCNoMAX`	`SCNoPTR`
`SCNu`*N*	`SCNuLEAST`*N*	`SCNuFAST`*N*	`SCNuMAX`	`SCNuPTR`
`SCNx`*N*	`SCNxLEAST`*N*	`SCNxFAST`*N*	`SCNxMAX`	`SCNxPTR`

Quy tắc là với mỗi kiểu được định nghĩa trong `<stdint.h>` sẽ có các macro `printf()` và `scanf()` tương ứng được định nghĩa ở đây.

9.2 `imaxabs()`

Tính giá trị tuyệt đối của một `intmax_t`

Synopsis

```
#include <inttypes.h>

intmax_t imaxabs(intmax_t j);
```

Mô tả

Khi bạn cần giá trị tuyệt đối của kiểu số nguyên lớn nhất trên hệ thống, đây là hàm dành cho bạn.

Spec ghi chú rằng nếu giá trị tuyệt đối của số không biểu diễn được, hành vi là undefined. Chuyện này xảy ra nếu bạn thử lấy giá trị tuyệt đối của số âm nhỏ nhất có thể trên hệ thống biểu diễn bù hai (two's complement).

Giá trị trả về

Trả về giá trị tuyệt đối của đầu vào, $|j|$.

Ví dụ

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

int main(void)
{
    intmax_t j = -3490;

    printf("%jd\n", imaxabs(j));    // 3490
}
```

Xem thêm

`fabs()`

9.3 `imaxdiv()`

Tính thương và số dư của các `intmax_t`

Synopsis

```
#include <inttypes.h>

imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
```

Mô tả

Khi bạn muốn làm phép chia nguyên và lấy số dư trong cùng một phép, hàm này sẽ làm cho bạn.

Nó tính `numer/denom` và `numer%denom` rồi trả về kết quả trong một struct kiểu `imaxdiv_t`.

Struct này có hai field kiểu `imaxdiv_t`, `quot` và `rem`, mà bạn dùng để lấy các giá trị mong muốn.

Giá trị trả về

Trả về một `imaxdiv_t` chứa thương và số dư của phép toán.

Ví dụ

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

int main(void)
{
    intmax_t numer = INTMAX_C(3490);
    intmax_t denom = INTMAX_C(17);

    imaxdiv_t r = imaxdiv(numer, denom);

    printf("Quotient: %jd, remainder: %jd\n", r.quot, r.rem);
}
```

Output:

```
Quotient: 205, remainder: 5
```

Xem thêm

`remquo()`

9.4 `strtoimax()` `strtoumax()`

Chuyển string sang kiểu `intmax_t` và `uintmax_t`

Synopsis

```
#include <inttypes.h>

intmax_t strtoimax(const char * restrict nptr, char ** restrict endptr,
                   int base);

uintmax_t strtoumax(const char * restrict nptr, char ** restrict endptr,
                   int base);
```

Mô tả

Mấy cái này hoạt động y hệt họ hàm `strtol()`, chỉ khác là trả về `intmax_t` hoặc `uintmax_t`.

Xem trang tham chiếu `strtol()` để biết chi tiết.

Giá trị trả về

Trả về string đã chuyển dưới dạng `intmax_t` hoặc `uintmax_t`.

Nếu kết quả nằm ngoài miền giá trị, giá trị trả về sẽ là `INTMAX_MAX`, `INTMAX_MIN`, hoặc `UINTMAX_MAX`, tuỳ trường hợp. Và biến `errno` sẽ được set thành `ERANGE`.

Ví dụ

Ví dụ sau chuyển một số cơ số 10 sang `intmax_t`. Rồi nó thử chuyển một số cơ số 2 không hợp lệ, bắt lỗi.

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

int main(void)
{
    intmax_t r;
    char *endptr;

    // Số cơ số 10 hợp lệ
    r = strtoimax("123456789012345", &endptr, 10);

    if (*endptr != '\0')
        printf("Invalid digit: %c\n", *endptr);
    else
        printf("Value is %jd\n", r);

    // Số nhị phân sau chứa chữ số không hợp lệ
    r = strtoimax("0100102010101101", &endptr, 2);

    if (*endptr != '\0')
        printf("Invalid digit: %c\n", *endptr);
    else
        printf("Value is %jd\n", r);
}
```

Output:

```
Value is 123456789012345
Invalid digit: 2
```

Xem thêm

`strtol()`, `errno`

9.5 `wcstoimax()` `wcstoumax()`

Chuyển wide string sang kiểu `intmax_t` và `uintmax_t`

Synopsis

```
#include <stddef.h> // cho wchar_t
#include <inttypes.h>

intmax_t wcstoimax(const wchar_t * restrict nptr,
                   wchar_t ** restrict endptr, int base);

uintmax_t wcstoumax(const wchar_t * restrict nptr,
                    wchar_t ** restrict endptr, int base);
```

Mô tả

Mấy cái này hoạt động y hệt họ hàm `wcstol()`, chỉ khác là trả về `intmax_t` hoặc `uintmax_t`.

Xem trang tham chiếu `wcstol()` để biết chi tiết.

Giá trị trả về

Trả về wide string đã chuyển dưới dạng `intmax_t` hoặc `uintmax_t`.

Nếu kết quả nằm ngoài miền giá trị, giá trị trả về sẽ là `INTMAX_MAX`, `INTMAX_MIN`, hoặc `UINTMAX_MAX`, tuỳ trường hợp. Và biến `errno` sẽ được set thành `ERANGE`.

Ví dụ

Ví dụ sau chuyển một số cơ số 10 sang `intmax_t`. Rồi nó thử chuyển một số cơ số 2 không hợp lệ, bắt lỗi.

```
#include <wchar.h>
#include <inttypes.h>

int main(void)
{
    intmax_t r;
    wchar_t *endptr;

    // Số cơ số 10 hợp lệ
    r = wcstoimax(L"123456789012345", &endptr, 10);

    if (*endptr != '\0')
        wprintf(L"Invalid digit: %lc\n", *endptr);
    else
        wprintf(L"Value is %jd\n", r);

    // Số nhị phân sau chứa chữ số không hợp lệ
    r = wcstoimax(L"0100102010101101", &endptr, 2);

    if (*endptr != '\0')
        wprintf(L"Invalid digit: %lc\n", *endptr);
    else
        wprintf(L"Value is %jd\n", r);
}
```

```
Value is 123456789012345
Invalid digit: 2
```

Xem thêm

`wcstol()`, `errno`

Chapter 10

`<iso646.h>` Cách Viết Thay Thế Cho Toán Tử

ISO-646 là một chuẩn mã hóa ký tự rất giống ASCII. Nhưng nó thiếu vài ký tự đáng chú ý như `|`, `^`, và `~`.

Vì đây là các toán tử hoặc thành phần của toán tử trong C, header này định nghĩa một số macro bạn có thể dùng phòng khi bàn phím của bạn không có những ký tự đó. (Và C++ cũng xài được cùng bộ thay thế này.)

Toán tử	Tương đương trong `<iso646.h>`
`&&`	`and`
`&=`	`and_eq`
`&`	`bitand`
`|`	`bitor`
`~`	`compl`
`!`	`not`
`!=`	`not_eq`
`||`	`or`
`|=`	`or_eq`
`^`	`xor`
`^=`	`xor_eq`

Điều thú vị là không có `eq` cho `==`, và `&` với `!` vẫn được đưa vào dù chúng có trong ISO-646.

Ví dụ dùng:

```
#include <stdio.h>
#include <iso646.h>

int main(void)
{
    int x = 12;
    int y = 30;

    if (x == 12 and y not_eq 40)
        printf("Now we know.\n");
}
```

Cá nhân tôi chưa từng thấy file này được include, nhưng chắc thỉnh thoảng cũng có người dùng.

Chapter 11

`<limits.h>` Giới Hạn Số

Lưu ý quan trọng: "minimum magnitude" (độ lớn tối thiểu) trong bảng bên dưới là giá trị tối thiểu mà spec yêu cầu. Rất có khả năng giá trị trên hệ thống xịn xò của bạn còn vượt xa những con số đó.

Macro	Độ lớn tối thiểu	Mô tả
`CHAR_BIT`	`8`	Số bit trong một byte
`SCHAR_MIN`	`-127`	Giá trị nhỏ nhất của `signed char`
`SCHAR_MAX`	`127`	Giá trị lớn nhất của `signed char`
`UCHAR_MAX`	`255`	Giá trị lớn nhất của `unsigned char` [1]
`CHAR_MIN`	`0` hoặc `SCHAR_MIN`	Xem chi tiết bên dưới
`CHAR_MAX`	`SCHAR_MAX` hoặc `UCHAR_MAX`	Xem chi tiết bên dưới
`MB_LEN_MAX`	`1`	Số byte tối đa trong một ký tự multibyte ở bất cứ locale nào
`SHRT_MIN`	`-32767`	Giá trị nhỏ nhất của `short`
`SHRT_MAX`	`32767`	Giá trị lớn nhất của `short`
`USHRT_MAX`	`65535`	Giá trị lớn nhất của `unsigned short`
`INT_MIN`	`-32767`	Giá trị nhỏ nhất của `int`
`INT_MAX`	`32767`	Giá trị lớn nhất của `int`
`UINT_MAX`	`65535`	Giá trị lớn nhất của `unsigned int`
`LONG_MIN`	`-2147483647`	Giá trị nhỏ nhất của `long`
`LONG_MAX`	`2147483647`	Giá trị lớn nhất của `long`
`ULONG_MAX`	`4294967295`	Giá trị lớn nhất của `unsigned long`
`LLONG_MIN`	`-9223372036854775807`	Giá trị nhỏ nhất của `long long`
`LLONG_MAX`	`9223372036854775807`	Giá trị lớn nhất của `long long`
`ULLONG_MAX`	`18446744073709551615`	Giá trị lớn nhất của `unsigned long long`

11.1 `CHAR_MIN` và `CHAR_MAX`

Với `CHAR_MIN` và `CHAR_MAX`, mọi thứ phụ thuộc vào việc kiểu `char` mặc định trên hệ bạn là signed hay unsigned. Nhớ là C để tùy implementation quyết định chuyện đó chứ? Không nhớ? Thật đấy.

Vậy nếu nó là signed, `CHAR_MIN` và `CHAR_MAX` có giá trị giống như `SCHAR_MIN` và `SCHAR_MAX`.

Còn nếu nó là unsigned, `CHAR_MIN` và `CHAR_MAX` giống `0` và `UCHAR_MAX`.

[1] Giá trị nhỏ nhất của `unsigned char` là `0`. Với `unsigned short` và `unsigned long` cũng vậy. Hay bất kỳ kiểu unsigned nào, cũng thế.

Tiện thể: bạn có thể biết lúc runtime hệ có `char` signed hay unsigned bằng cách kiểm tra xem `CHAR_MIN` có bằng `0` không.

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

int main(void)
{
    printf("chars are %ssigned\n", CHAR_MIN == 0? "un": "");
}
```

Trên hệ của tôi, `char` là signed.

11.2 Chọn Kiểu Cho Đúng

Nếu bạn muốn siêu portable, hãy chọn một kiểu mà bảng bên trên đảm bảo ít nhất đủ to như bạn cần.

Nói vậy chứ, khá nhiều code—vì tốt hơn hoặc (khả năng cao hơn) tệ hơn—giả định rằng `int` là 32-bit, trong khi thực ra spec chỉ đảm bảo 16-bit thôi.

Nếu bạn cần kích thước bit được đảm bảo, xem các kiểu `int_leastN_t` trong `<stdint.h>`.

11.3 Còn Two's Complement Thì Sao?

Nếu bạn tinh mắt và lại có hiểu biết sẵn về chủ đề này, bạn có thể đã nghĩ tôi viết sai giá trị nhỏ nhất của các macro bên trên.

"`short` đi từ `32767` tới `-32767`? Chẳng phải nó phải tới `-32768` à?"

Không, tôi viết đúng rồi. Spec liệt kê độ lớn tối thiểu cho các macro đó, và một số hệ thống cổ lỗ có thể đã dùng cách encode khác cho số signed mà chỉ đi xa được tới đó.

Hầu như mọi hệ thống hiện đại đều dùng Two's Complement[2] (bù hai) cho số signed, và với kiểu đó `short` sẽ đi từ `32767` tới `-32768`. Hệ của bạn chắc cũng vậy.

11.4 Chương Trình Demo

Đây là chương trình in ra giá trị các macro:

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

int main(void)
{
    printf("CHAR_BIT = %d\n", CHAR_BIT);
    printf("SCHAR_MIN = %d\n", SCHAR_MIN);
    printf("SCHAR_MAX = %d\n", SCHAR_MAX);
    printf("UCHAR_MAX = %d\n", UCHAR_MAX);
    printf("CHAR_MIN = %d\n", CHAR_MIN);
    printf("CHAR_MAX = %d\n", CHAR_MAX);
    printf("MB_LEN_MAX = %d\n", MB_LEN_MAX);
    printf("SHRT_MIN = %d\n", SHRT_MIN);
    printf("SHRT_MAX = %d\n", SHRT_MAX);
    printf("USHRT_MAX = %u\n", USHRT_MAX);
    printf("INT_MIN = %d\n", INT_MIN);
    printf("INT_MAX = %d\n", INT_MAX);
```

[2] https://en.wikipedia.org/wiki/Two%27s_complement

```
    printf("UINT_MAX = %u\n", UINT_MAX);
    printf("LONG_MIN = %ld\n", LONG_MIN);
    printf("LONG_MAX = %ld\n", LONG_MAX);
    printf("ULONG_MAX = %lu\n", ULONG_MAX);
    printf("LLONG_MIN = %lld\n", LLONG_MIN);
    printf("LLONG_MAX = %lld\n", LLONG_MAX);
    printf("ULLONG_MAX = %llu\n", ULLONG_MAX);
}
```

Trên hệ Intel 64-bit của tôi với clang, output là:

```
CHAR_BIT = 8
SCHAR_MIN = -128
SCHAR_MAX = 127
UCHAR_MAX = 255
CHAR_MIN = -128
CHAR_MAX = 127
MB_LEN_MAX = 6
SHRT_MIN = -32768
SHRT_MAX = 32767
USHRT_MAX = 65535
INT_MIN = -2147483648
INT_MAX = 2147483647
UINT_MAX = 4294967295
LONG_MIN = -9223372036854775808
LONG_MAX = 9223372036854775807
ULONG_MAX = 18446744073709551615
LLONG_MIN = -9223372036854775808
LLONG_MAX = 9223372036854775807
ULLONG_MAX = 18446744073709551615
```

Có vẻ hệ của tôi dùng encoding two's-complement cho số signed, `char` là signed, và `int` là 32-bit.

Chapter 12

`<locale.h>` Xử Lý Locale

Hàm	Mô tả
`setlocale()`	Đặt locale
`localeconv()`	Lấy thông tin về locale hiện tại

"Locale" là các chi tiết về cách chương trình nên chạy tùy theo vị trí địa lý của nó trên trái đất.

Ví dụ, ở locale này, một đơn vị tiền tệ có thể in ra là `$123`, còn ở locale khác là `€123`.

Hoặc một locale dùng encoding ASCII và locale khác dùng UTF-8.

Mặc định, chương trình chạy trong locale "C". Nó có một tập ký tự cơ bản với encoding single-byte. Nếu bạn thử in ký tự UTF-8 trong locale C, sẽ không có gì in ra. Bạn phải chuyển sang một locale thích hợp.

12.1 `setlocale()`

Đặt locale

Synopsis

```
#include <locale.h>

char *setlocale(int category, const char *locale);
```

Mô tả

Đặt `locale` cho `category` chỉ định.

Category là một trong các giá trị sau:

Category	Mô tả
`LC_ALL`	Tất cả các category dưới đây
`LC_COLLATE`	Ảnh hưởng tới hàm `strcoll()` và `strxfrm()`
`LC_CTYPE`	Ảnh hưởng tới các hàm trong `<ctype.h>`
`LC_MONETARY`	Ảnh hưởng tới thông tin tiền tệ mà `localeconv()` trả về

Category	Mô tả
`LC_NUMERIC`	Ảnh hưởng tới dấu thập phân cho I/O định dạng và hàm xử lý chuỗi định dạng, cùng thông tin tiền tệ mà `localeconv()` trả về
`LC_TIME`	Ảnh hưởng tới hàm `strftime()` và `wcsftime()`

Và có ba giá trị portable bạn có thể truyền cho `locale`; mọi chuỗi khác đều là implementation-defined và không portable.

Locale	Mô tả
`"C"`	Đặt chương trình về locale C
`""`	(Chuỗi rỗng) Đặt chương trình về locale native của hệ
`NULL`	Không thay đổi gì; chỉ trả về locale hiện tại
Khác	Đặt chương trình về một locale implementation-defined

Lời gọi phổ biến nhất, tôi dám cá, là:

```
// Đặt mọi cài đặt locale về`locale native của hệ

setlocale(LC_ALL, "");
```

Tiện là `setlocale()` trả về locale vừa được đặt, nên bạn có thể thấy locale thực tế trên hệ của mình là gì.

Giá trị trả về

Nếu thành công, trả về con trỏ tới chuỗi biểu diễn locale hiện tại. Bạn không được modify chuỗi này, và nó có thể bị thay đổi bởi các lời gọi `setlocale()` kế tiếp.

Nếu thất bại, trả về `NULL`.

Ví dụ

Ở đây ta lấy locale hiện tại. Rồi đặt nó về locale native, và in ra.

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

int main(void)
{
    char *loc;

    // Lấy locale hiện tại
    loc = setlocale(LC_ALL, NULL);

    printf("Starting locale: %s\n", loc);

    // Đặt (và lấy) locale về`locale native
    loc = setlocale(LC_ALL, "");

    printf("Native locale: %s\n", loc);
}
```

Output trên hệ của tôi:

```
Starting locale: C
Native locale: en_US.UTF-8
```

Lưu ý locale native của tôi (trên máy Linux) có thể khác với những gì bạn thấy.

Dù vậy, tôi vẫn có thể đặt nó thẳng tay trên hệ của mình, hoặc đặt về bất kỳ locale nào đã cài:

```
    loc = setlocale(LC_ALL, "en_US.UTF-8");   // Không portable
```

Nhưng lần nữa, hệ của bạn có thể có các locale khác được định nghĩa.

Xem thêm

`localeconv()`, `strcoll()`, `strxfrm()`, `strftime()`, `wcsftime()`, `printf()`, `scanf()`, `<ctype.h>`

12.2 `localeconv()`

Lấy thông tin về locale hiện tại

Synopsis

```
#include <locale.h>

struct lconv *localeconv(void);
```

Mô tả

Hàm này chỉ trả về một con trỏ tới `struct lconv`, nhưng vẫn là một tay có sức mạnh đáng kể.

Struct trả về chứa *cả tấn* thông tin về locale. Dưới đây là các trường của `struct lconv` và ý nghĩa của chúng.

Trước hết, vài quy ước. Trong tên trường bên dưới, `_p_` nghĩa là "positive" (dương), `_n_` nghĩa là "negative" (âm), và `int_` nghĩa là "international" (quốc tế). Dù khá nhiều trong số này có kiểu `char` hoặc `char*`, hầu hết (hoặc chuỗi mà chúng trỏ tới) thực ra được đối xử như integer[1].

Trước khi đi tiếp, bạn cần biết `CHAR_MAX` (từ `<limits.h>`) là giá trị lớn nhất mà một `char` có thể giữ. Và khá nhiều giá trị `char` bên dưới dùng nó để báo hiệu rằng giá trị không có sẵn trong locale đó.

Trường	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 số tiền, in bao nhiêu chữ số sau dấu thập phân, ví dụ `2`.
`char p_cs_precedes`	`1` nếu `currency_symbol` đi trước giá trị cho số tiền không âm, `0` nếu đi sau.
`char n_cs_precedes`	`1` nếu `currency_symbol` đi trước giá trị cho số tiền âm, `0` nếu đi sau.

[1] Nhớ là `char` chỉ là integer cỡ một byte.

Trường	Mô tả
`char p_sep_by_space`	Xác định cách phân cách `currency symbol` khỏi giá trị cho số không âm (xem bên dưới).
`char n_sep_by_space`	Xác định cách phân cách `currency symbol` khỏi giá trị cho số âm (xem bên dưới).
`char p_sign_posn`	Xác định vị trí `positive_sign` cho số không âm.
`char p_sign_posn`	Xác định vị trí `positive_sign` cho số 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`.
`char int_n_sign_posn`	Giá trị quốc tế cho `n_sign_posn`.

Mặc dù khá nhiều trong số này có kiểu `char`, giá trị chứa trong đó được truy cập như integer.

Tất cả biến thể `sep_by_space` xử lý khoảng cách quanh ký hiệu tiền tệ. Giá trị hợp lệ:

Giá trị	Mô tả
`0`	Không có khoảng trắng giữa ký hiệu tiền tệ và giá trị.
`1`	Phân cách ký hiệu tiền tệ (và dấu, nếu có) khỏi giá trị bằng khoảng trắng.
`2`	Phân cách dấu khỏi ký hiệu tiền tệ (nếu kề nhau) bằng khoảng trắng, ngược lại phân cách dấu khỏi giá trị bằng khoảng trắng.

Các biến thể `sign_posn` được xác định bởi giá trị sau:

Giá trị	Mô tả
`0`	Đặt ngoặc đơn quanh giá trị và ký hiệu tiền tệ.
`1`	Đặt chuỗi dấu ở 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.

Giá trị trả về

Trả về con trỏ tới struct chứa thông tin locale.

Chương trình không được modify struct này.

Các lời gọi `localeconv()` kế tiếp có thể ghi đè struct này, cũng như các lời gọi `setlocale()` với `LC_ALL`, `LC_MONETARY`, hoặc `LC_NUMERIC`.

Ví dụ

Đây là chương trình in thông tin locale cho locale native.

```
#include <stdio.h>
#include <locale.h>
#include <limits.h>  // for CHAR_MAX

void print_grouping(char *mg)
```

```
{
    int done = 0;

    while (!done) {
        if (*mg == CHAR_MAX)
            printf("CHAR_MAX ");
        else
            printf("%c ", *mg + '0');
        done = *mg == CHAR_MAX || *mg == 0;
        mg++;
    }
}

int main(void)
{
    setlocale(LC_ALL, "");

    struct lconv *lc = localeconv();

    printf("mon_decimal_point : %s\n", lc->mon_decimal_point);
    printf("mon_thousands_sep : %s\n", lc->mon_thousands_sep);
    printf("mon_grouping      : ");
    print_grouping(lc->mon_grouping);
    printf("\n");
    printf("positive_sign     : %s\n", lc->positive_sign);
    printf("negative_sign     : %s\n", lc->negative_sign);
    printf("currency_symbol   : %s\n", lc->currency_symbol);
    printf("frac_digits       : %c\n", lc->frac_digits);
    printf("p_cs_precedes     : %c\n", lc->p_cs_precedes);
    printf("n_cs_precedes     : %c\n", lc->n_cs_precedes);
    printf("p_sep_by_space    : %c\n", lc->p_sep_by_space);
    printf("n_sep_by_space    : %c\n", lc->n_sep_by_space);
    printf("p_sign_posn       : %c\n", lc->p_sign_posn);
    printf("p_sign_posn       : %c\n", lc->p_sign_posn);
    printf("int_curr_symbol   : %s\n", lc->int_curr_symbol);
    printf("int_frac_digits   : %c\n", lc->int_frac_digits);
    printf("int_p_cs_precedes : %c\n", lc->int_p_cs_precedes);
    printf("int_n_cs_precedes : %c\n", lc->int_n_cs_precedes);
    printf("int_p_sep_by_space: %c\n", lc->int_p_sep_by_space);
    printf("int_n_sep_by_space: %c\n", lc->int_n_sep_by_space);
    printf("int_p_sign_posn   : %c\n", lc->int_p_sign_posn);
    printf("int_n_sign_posn   : %c\n", lc->int_n_sign_posn);
}
```

Output trên hệ của tôi:

```
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
p_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
```

Xem thêm

`setlocale()`

Chapter 13

`<math.h>` Toán Học

Nhiều hàm trong phần này có các phiên bản `float` và `long double` như mô tả ở dưới (ví dụ `pow()`, `powf()`, `powl()`). Các biến thể `float` và `long double` được bỏ qua trong bảng dưới để mắt bạn khỏi nổ tung.

Hàm	Mô tả
`acos()`	Tính arc cosine của một số.
`acosh()`	Tính arc hyperbolic cosine.
`asin()`	Tính arc sine của một số.
`asinh()`	Tính arc hyperbolic sine.
`atan()`, `atan2()`	Tính arc tangent của một số.
`atanh()`	Tính arc hyperbolic tangent.
`cbrt()`	Tính căn bậc ba.
`ceil()`	Ceiling—trả về số nguyên (phần nguyên hướng lên) không nhỏ hơn số đã cho.
`copysign()`	Chép dấu của một giá trị sang một giá trị khác.
`cos()`	Tính cosine của một số.
`cosh()`	Tính hyperbolic cosine.
`erf()`	Tính hàm lỗi (error function) của giá trị cho trước.
`erfc()`	Tính hàm lỗi bù (complementary error function) của một giá trị.
`exp()`	Tính e mũ lên.
`exp2()`	Tính 2 mũ lên.
`expm1()`	Tính $e^x - 1$.
`fabs()`	Tính giá trị tuyệt đối.
`fdim()`	Trả về hiệu số dương giữa hai số, chặn dưới ở 0.
`floor()`	Tính số nguyên lớn nhất không lớn hơn giá trị cho trước.
`fma()`	Nhân và cộng Floating (còn gọi là "Fast").
`fmax()`, `fmin()`	Trả về giá trị lớn nhất hoặc nhỏ nhất trong hai số.
`fmod()`	Tính phần dư floating point.
`fpclassify()`	Trả về phân loại của một số floating point cho trước.
`frexp()`	Tách một số thành phần phân số và phần mũ (theo luỹ thừa của 2).
`hypot()`	Tính độ dài cạnh huyền của tam giác.
`ilogb()`	Trả về số mũ của một số floating point.
`isfinite()`	True nếu số không phải vô cùng hay NaN.
`isgreater()`	True nếu một đối số lớn hơn đối số khác.
`isgreatereequal()`	True nếu một đối số lớn hơn hoặc bằng đối số khác.
`isinf()`	True nếu số là vô cùng.
`isless()`	True nếu một đối số nhỏ hơn đối số khác.
`islesseequal()`	True nếu một đối số nhỏ hơn hoặc bằng đối số khác.

Hàm	Mô tả
`islessgreater()`	Kiểm tra số floating point này nhỏ hơn hoặc lớn hơn số kia.
`isnan()`	True nếu số là Not-a-Number.
`isnormal()`	True nếu số là số "normal".
`isunordered()`	Macro trả về true nếu bất kỳ đối số floating point nào là NaN.
`ldexp()`	Nhân một số với luỹ thừa nguyên của 2.
`lgamma()`	Tính logarithm tự nhiên của giá trị tuyệt đối của $\Gamma(x)$.
`log()`	Tính logarithm tự nhiên.
`log10()`	Tính log cơ số 10 của một số.
`log2()`	Tính logarithm cơ số 2 của một số.
`logb()`	Trích số mũ của một số theo cơ số `FLT_RADIX`.
`log1p()`	Tính logarithm tự nhiên của một số cộng 1.
`lrint()`	Trả về `x` được làm tròn theo hướng làm tròn hiện tại dưới dạng số nguyên.
`lround()`, `llround()`	Làm tròn một số theo cách cổ điển, trả về số nguyên.
`modf()`	Tách phần nguyên và phần phân số của một số.
`nan()`	Trả về `NAN`.
`nearbyint()`	Làm tròn giá trị theo hướng làm tròn hiện tại.
`nextafter()`	Lấy giá trị floating point kế tiếp (hoặc trước đó) biểu diễn được.
`nexttoward()`	Lấy giá trị floating point kế tiếp (hoặc trước đó) biểu diễn được.
`pow()`	Tính một giá trị luỹ thừa.
`remainder()`	Tính phần dư kiểu IEC 60559.
`remquo()`	Tính phần dư và (một phần của) thương.
`rint()`	Làm tròn một giá trị theo hướng làm tròn hiện tại.
`round()`	Làm tròn một số theo cách cổ điển.
`scalbn()`, `scalbln()`	Tính $x \times r^n$ hiệu quả, với r là `FLT_RADIX`.
`signbit()`	Trả về dấu của một số.
`sin()`	Tính sine của một số.
`sqrt()`	Tính căn bậc hai của một số.
`tan()`	Tính tangent của một số.
`tanh()`	Tính hyperbolic tangent.
`tgamma()`	Tính hàm gamma, $\Gamma(x)$.
`trunc()`	Cắt phần phân số của một giá trị floating point.

Môn học yêu thích của bạn đây: Toán học! Xin chào, tôi là Tiến sĩ Math, và tôi sẽ làm cho toán trở nên VUI và DỄ!

[tiếng ói mửa]

Được rồi, tôi biết toán không phải thứ tuyệt vời nhất với một số bạn, nhưng đây chỉ là những hàm làm toán nhanh gọn, thứ toán mà bạn hoặc biết, hoặc cần, hoặc chẳng quan tâm. Tóm lại là vậy đó.

13.1 Các idiom hàm Toán học

Nhiều hàm toán học này tồn tại ở ba dạng, tương ứng với kiểu đối số và/hoặc kiểu trả về mà hàm sử dụng: `float`, `double`, hoặc `long double`.

Dạng thay thế cho `float` được tạo bằng cách thêm `f` vào cuối tên hàm.

Dạng thay thế cho `long double` được tạo bằng cách thêm `l` vào cuối tên hàm.

Ví dụ, hàm `pow()` dùng để tính x^y tồn tại ở các dạng sau:

```
double      pow(double x, double y);             // double
float       powf(float x, float y);              // float
long double powl(long double x, long double y);  // long double
```

Nhớ là tham số nhận giá trị như thể bạn gán vào chúng. Vậy nên nếu bạn truyền một `double` vào `powf()`, nó sẽ chọn `float` gần nhất có thể để giữ giá trị double đó. Nếu `double` không vừa, hành vi không xác định (undefined behavior) sẽ xảy ra.

13.2 Các kiểu Toán học

Chúng ta có hai kiểu mới rất hấp dẫn trong `<math.h>`:

- `float_t`
- `double_t`

Kiểu `float_t` ít nhất chính xác bằng `float`, và kiểu `double_t` ít nhất chính xác bằng `double`.

Ý tưởng của các kiểu này là chúng có thể biểu diễn cách lưu trữ số hiệu quả nhất để đạt tốc độ tối đa.

Kiểu thực tế thay đổi theo implementation, nhưng có thể xác định qua giá trị của macro `FLT_EVAL_METHOD`.

`FLT_EVAL_METHOD`	Kiểu `float_t`	Kiểu `double_t`
`0`	`float`	`double`
`1`	`double`	`double`
`2`	`long double`	`long double`
Khác	Phụ thuộc implementation	Phụ thuộc implementation

Với mọi giá trị xác định của `FLT_EVAL_METHOD`, `float_t` là kiểu có độ chính xác thấp nhất được dùng cho mọi phép tính floating point.

13.3 Các Macro Toán học

Thực ra có khá nhiều macro được định nghĩa, nhưng chúng ta sẽ nói hầu hết chúng trong các phần tham chiếu tương ứng bên dưới.

Nhưng đây là vài cái:

`NAN` biểu diễn Not-A-Number.

Định nghĩa trong `<float.h>` là `FLT_RADIX`: cơ số mà floating point dùng. Thường là `2`, nhưng có thể là bất kỳ giá trị nào.

13.4 Các lỗi Toán học

Như ta biết, không gì có thể sai trong toán học... ngoại trừ *mọi thứ*!

Vậy nên có vài loại lỗi có thể xảy ra khi dùng các hàm này.

- **Range error (lỗi miền giá trị)** nghĩa là kết quả vượt quá những gì có thể lưu trong kiểu trả về.
- **Domain error (lỗi miền xác định)** nghĩa là bạn truyền vào một đối số không có kết quả xác định với hàm này.
- **Pole error** nghĩa là giới hạn của hàm khi x tiến tới đối số cho trước là vô cùng.
- **Overflow error (lỗi tràn trên)** là khi kết quả rất lớn, nhưng không thể lưu mà không gây sai số làm tròn (rounding) lớn.

- **Underflow error (lỗi tràn dưới)** giống overflow, nhưng với các số rất nhỏ.

Giờ, thư viện toán của C có thể làm vài thứ khi các lỗi này xảy ra:

- Đặt `errno` thành một giá trị nào đó, hoặc...
- Raise một floating point exception.

Hệ thống của bạn có thể xử lý khác nhau. Bạn có thể kiểm tra bằng cách xem giá trị của biến `math_errhandling`. Nó sẽ bằng một trong những giá trị sau[1]:

`math_errhandling`	Mô tả
`MATH_ERRNO`	Hệ thống dùng `errno` cho các lỗi toán.
`MATH_ERREXCEPT`	Hệ thống dùng exception cho các lỗi toán.
`MATH_ERRNO | MATH_ERREXCEPT`	Hệ thống làm cả hai! (Đó là phép OR bit!)

Bạn không được phép thay đổi `math_errhandling`.

Muốn mô tả đầy đủ hơn về cách exception hoạt động và ý nghĩa của chúng, xem phần `<fenv.h>`.

13.5 Các Pragma Toán học

Nói gọn, pragma cho phép ta điều khiển hành vi của trình biên dịch theo nhiều cách. Ở đây, ta nói về chuyện điều khiển cách thư viện toán của C hoạt động.

Cụ thể, chúng ta có pragma `FP_CONTRACT` có thể bật/tắt.

Nó có nghĩa gì?

Trước tiên, nhớ rằng bất kỳ phép toán nào trong một biểu thức đều có thể gây rounding error. Nên mỗi bước của biểu thức có thể tạo thêm rounding error.

Nhưng sẽ thế nào nếu compiler biết một cách *siêu bí mật* để lấy biểu thức bạn viết và chuyển nó thành một lệnh duy nhất, giảm số bước đến mức rounding error trung gian không xảy ra?

Nó có được dùng cách đó không? Ý là, kết quả sẽ khác so với khi bạn để rounding error tích tụ qua từng bước...

Vì kết quả sẽ khác, bạn có thể bảo compiler rằng bạn có cho phép làm thế hay không.

Nếu bạn cho phép:

```
#pragma STDC FP_CONTRACT ON
```

và để không cho phép:

```
#pragma STDC FP_CONTRACT OFF
```

Nếu bạn làm điều này ở phạm vi toàn cục, nó sẽ giữ trạng thái bạn đặt cho đến khi bạn đổi nó.

Nếu bạn làm ở phạm vi block, nó sẽ quay về giá trị bên ngoài block khi block kết thúc.

Giá trị ban đầu của pragma `FP_CONTRACT` thay đổi tuỳ hệ thống.

13.6 `fpclassify()`

Trả về phân loại của một số floating point cho trước.

[1] Dù hệ thống định nghĩa `MATH_ERRNO` là `1` và `MATH_ERREXCEPT` là `2`, tốt nhất luôn dùng tên ký hiệu. Phòng hờ.

Synopsis

```
#include <math.h>

int fpclassify(any_floating_type x);
```

Mô tả

Số floating point này biểu diễn loại thực thể nào? Có những lựa chọn nào?

Chúng ta quen thuộc với số floating point là những thứ bình thường như `3.14` hay `3490.0001`.

Nhưng số floating point cũng có thể biểu diễn những thứ như vô cùng. Hay Not-A-Number (NAN). Hàm này cho bạn biết đối số là loại floating point nào.

Đây là một macro, nên bạn có thể dùng với `float`, `double`, `long double`, hay bất kỳ kiểu tương tự nào.

Giá trị trả về

Trả về một trong các macro sau tuỳ theo phân loại của đối số:

Phân loại	Mô tả
`FP_INFINITE`	Số là vô cùng.
`FP_NAN`	Số là Not-A-Number (NAN).
`FP_NORMAL`	Chỉ là một số bình thường.
`FP_SUBNORMAL`	Số là số sub-normal.
`FP_ZERO`	Số là không.

Bàn về số subnormal nằm ngoài phạm vi guide này, và là thứ mà phần lớn dev đi hết đời mà không phải đụng tới. Nói ngắn gọn, đó là cách biểu diễn các số rất nhỏ mà bình thường sẽ bị làm tròn về không. Nếu muốn biết thêm, xem trang Wikipedia về số denormal[2].

Ví dụ

In các phân loại số khác nhau.

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

const char *get_classification(double n)
{
    switch (fpclassify(n)) {
        case FP_INFINITE: return "infinity";
        case FP_NAN: return "not a number";
        case FP_NORMAL: return "normal";
        case FP_SUBNORMAL: return "subnormal";
        case FP_ZERO: return "zero";
    }

    return "unknown";
}

int main(void)
{
```

[2] https://en.wikipedia.org/wiki/Denormal_number

```
    printf("    1.23: %s\n", get_classification(1.23));
    printf("     0.0: %s\n", get_classification(0.0));
    printf("sqrt(-1): %s\n", get_classification(sqrt(-1)));
    printf("1/tan(0): %s\n", get_classification(1/tan(0)));
    printf("  1e-310: %s\n", get_classification(1e-310));  // rất nhỏ!
}
```

Output[3]:

```
    1.23: normal
     0.0: zero
sqrt(-1): not a number
1/tan(0): infinity
  1e-310: subnormal
```

Xem thêm

`isfinite()`, `isinf()`, `isnan()`, `isnormal()`, `signbit()`

13.7 `isfinite()`, `isinf()`, `isnan()`, `isnormal()`

Trả về true nếu số khớp một phân loại.

Synopsis

```
#include <math.h>

int isfinite(any_floating_type x);

int isinf(any_floating_type x);

int isnan(any_floating_type x);

int isnormal(any_floating_type x);
```

Mô tả

Đây là các macro trợ giúp cho `fpclassify()`. Vì là macro, chúng hoạt động trên mọi kiểu floating point.

Macro	Mô tả
`isfinite()`	True nếu số không phải vô cùng hoặc NaN.
`isinf()`	True nếu số là vô cùng.
`isnan()`	True nếu số là Not-a-Number.
`isnormal()`	True nếu số là normal.

Để biết thêm thảo luận sơ lược về số normal và subnormal, xem `fpclassify()`.

Giá trị trả về

Trả về khác không nếu true, và không nếu false.

[3]Đây là trên máy tôi. Một số hệ thống sẽ có ngưỡng mà số trở thành subnormal khác, hoặc có thể không hỗ trợ giá trị subnormal.

Ví dụ

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

int main(void)
{
    printf("  isfinite(1.23): %d\n", isfinite(1.23));    // 1
    printf(" isinf(1/tan(0)): %d\n", isinf(1/tan(0)));   // 1
    printf(" isnan(sqrt(-1)): %d\n", isnan(sqrt(-1)));   // 1
    printf("isnormal(1e-310): %d\n", isnormal(1e-310));  // 0
}
```

Xem thêm

`fpclassify()`, `signbit()`,

13.8 `signbit()`

Trả về dấu của một số.

Synopsis

```
#include <math.h>

int signbit(any_floating_type x);
```

Mô tả

Macro này nhận vào một số floating point bất kỳ và trả về giá trị cho biết dấu của số đó, dương hay âm.

Giá trị trả về

Trả về `1` nếu dấu là âm, ngược lại là `0`.

Ví dụ

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

int main(void)
{
    printf("%d\n", signbit(3490.0));  // 0
    printf("%d\n", signbit(-37.0));   // 1
}
```

Xem thêm

`fpclassify()`, `isfinite()`, `isinf()`, `isnan()`, `isnormal()`, `copysign()`

13.9 `acos()`, `acosf()`, `acosl()`

Tính arc cosine của một số.

Synopsis

```
#include <math.h>

double acos(double x);
float acosf(float x);
long double acosl(long double x);
```

Mô tả

Tính arc cosine của một số theo radian. (Tức là giá trị mà cosine của nó bằng `x`.) Số phải nằm trong khoảng -1.0 đến 1.0.

Với những bạn đã quên, radian là một cách đo góc khác, giống như độ. Để chuyển đổi giữa độ và radian, dùng đoạn code sau:

```
pi = 3.14159265358979;
degrees = radians * 180 / pi;
radians = degrees * pi / 180;
```

Giá trị trả về

Trả về arc cosine của `x`, trừ khi `x` ngoài khoảng. Trong trường hợp đó, `errno` sẽ được đặt thành EDOM và giá trị trả về sẽ là NaN. Các biến thể trả về kiểu khác nhau.

Ví dụ

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

int main(void)
{
    double acosx;
    long double ldacosx;

    acosx = acos(0.2);
    ldacosx = acosl(0.3L);

    printf("%f\n", acosx);
    printf("%Lf\n", ldacosx);
}
```

Xem thêm

`asin()`, `atan()`, `atan2()`, `cos()`

13.10 `asin()`, `asinf()`, `asinl()`

Tính arc sine của một số.

Synopsis

```
#include <math.h>

double asin(double x);
float asinf(float x);
long double asinl(long double x);
```

Mô tả

Tính arc sine của một số theo radian. (Tức là giá trị mà sine của nó bằng `x`.) Số phải nằm trong khoảng -1.0 đến 1.0.

Với những bạn đã quên, radian là một cách đo góc khác, giống như độ. Để chuyển đổi giữa độ và radian, dùng đoạn code sau:

```
pi = 3.14159265358979;
degrees = radians * 180 / pi;
radians = degrees * pi / 180;
```

Giá trị trả về

Trả về arc sine của `x`, trừ khi `x` ngoài khoảng. Trong trường hợp đó, `errno` sẽ được đặt thành EDOM và giá trị trả về sẽ là NaN. Các biến thể trả về kiểu khác nhau.

Ví dụ

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

int main(void)
{
    double asinx;
    long double ldasinx;

    asinx = asin(0.2);
    ldasinx = asinl(0.3L);

    printf("%f\n", asinx);
    printf("%Lf\n", ldasinx);
}
```

Xem thêm

`acos()`, `atan()`, `atan2()`, `sin()`

13.11 `atan()`, `atanf()`, `atanl()`, `atan2()`, `atan2f()`, `atan2l()`

Tính arc tangent của một số.

Synopsis

```
#include <math.h>

double atan(double x);
float atanf(float x);
long double atanl(long double x);

double atan2(double y, double x);
float atan2f(float y, float x);
long double atan2l(long double y, long double x);
```

Mô tả

Tính arc tangent của một số theo radian. (Tức là giá trị mà tangent của nó bằng `x`.)

Các biến thể `atan2()` khá giống gọi `atan()` với tham số `y / x` ... ngoại trừ chuyện `atan2()` sẽ dùng các giá trị đó để xác định đúng góc phần tư (quadrant) của kết quả.

Với những bạn đã quên, radian là một cách đo góc khác, giống như độ. Để chuyển đổi giữa độ và radian, dùng đoạn code sau:

```
pi = 3.14159265358979;
degrees = radians * 180 / pi;
radians = degrees * pi / 180;
```

Giá trị trả về

Các hàm `atan()` trả về arc tangent của `x`, nằm giữa PI/2 và -PI/2. Các hàm `atan2()` trả về một góc giữa PI và -PI.

Ví dụ

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

int main(void)
{
    double atanx;
    long double ldatanx;

    atanx = atan(0.7);
    ldatanx = atanl(0.3L);

    printf("%f\n", atanx);
    printf("%Lf\n", ldatanx);

    atanx = atan2(7, 10);
    ldatanx = atan2l(3L, 10L);

    printf("%f\n", atanx);
    printf("%Lf\n", ldatanx);
}
```

Xem thêm

`tan()`, `asin()`, `atan()`

13.12 `cos()`, `cosf()`, `cosl()`

Tính cosine của một số.

Synopsis

```
#include <math.h>

double cos(double x)
float cosf(float x)
long double cosl(long double x)
```

Mô tả

Tính cosine của giá trị `x`, với `x` tính bằng radian.

Với những bạn đã quên, radian là một cách đo góc khác, giống như độ. Để chuyển đổi giữa độ và radian, dùng đoạn code sau:

```
pi = 3.14159265358979;
degrees = radians * 180 / pi;
radians = degrees * pi / 180;
```

Giá trị trả về

Trả về cosine của `x`. Các biến thể trả về kiểu khác nhau.

Ví dụ

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

int main(void)
{
    double cosx;
    long double ldcosx;

    cosx = cos(3490.0); // quay mòng mòng!
    ldcosx = cosl(3.490L);

    printf("%f\n", cosx);
    printf("%Lf\n", ldcosx);
}
```

Xem thêm

`sin()`, `tan()`, `acos()`

13.13 `sin()`, `sinf()`, `sinl()`

Tính sine của một số.

Synopsis

```
#include <math.h>

double sin(double x);
float sinf(float x);
long double sinl(long double x);
```

Mô tả

Tính sine của giá trị `x`, với `x` tính bằng radian.

Với những bạn đã quên, radian là một cách đo góc khác, giống như độ. Để chuyển đổi giữa độ và radian, dùng đoạn code sau:

```
pi = 3.14159265358979;
degrees = radians * 180 / pi;
radians = degrees * pi / 180;
```

Giá trị trả về

Trả về sine của `x`. Các biến thể trả về kiểu khác nhau.

Ví dụ

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

int main(void)
{
    double sinx;
    long double ldsinx;

    sinx = sin(3490.0); // quay mòng mòng!
    ldsinx = sinl(3.490L);

    printf("%f\n", sinx);
    printf("%Lf\n", ldsinx);
}
```

Xem thêm

`cos()`, `tan()`, `asin()`

13.14 `tan()`, `tanf()`, `tanl()`

Tính tangent của một số.

Synopsis

```
#include <math.h>

double tan(double x)
float tanf(float x)
```

```
long double tanl(long double x)
```

Mô tả

Tính tangent của giá trị `x`, với `x` tính bằng radian.

Với những bạn đã quên, radian là một cách đo góc khác, giống như độ. Để chuyển đổi giữa độ và radian, dùng đoạn code sau:

```
pi = 3.14159265358979;
degrees = radians * 180 / pi;
radians = degrees * pi / 180;
```

Giá trị trả về

Trả về tangent của `x`. Các biến thể trả về kiểu khác nhau.

Ví dụ

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

int main(void)
{
    double tanx;
    long double ldtanx;

    tanx = tan(3490.0); // quay mòng mòng!
    ldtanx = tanl(3.490L);

    printf("%f\n", tanx);
    printf("%Lf\n", ldtanx);
}
```

Xem thêm

`sin()`, `cos()`, `atan()`, `atan2()`

13.15 `acosh()`, `acoshf()`, `acoshl()`

Tính arc hyperbolic cosine.

Synopsis

```
#include <math.h>

double acosh(double x);

float acoshf(float x);

long double acoshl(long double x);
```

Mô tả

Fan trig mừng đi! C có arc hyperbolic cosine!

Các hàm này trả về acosh không âm của `x`, với `x` phải lớn hơn hoặc bằng `1`.

Giá trị trả về

Trả về arc hyperbolic cosine trong khoảng $[0, +\infty]$.

Ví dụ

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

int main(void)
{
    printf("acosh 1.8 = %f\n", acosh(1.8));  // 1.192911
}
```

Xem thêm

`asinh()`

13.16 `asinh()`, `asinhf()`, `asinhl()`

Tính arc hyperbolic sine.

Synopsis

```
#include <math.h>

double asinh(double x);

float asinhf(float x);

long double asinhl(long double x);
```

Mô tả

Fan trig mừng đi! C có arc hyperbolic sine!

Các hàm này trả về asinh của `x`.

Giá trị trả về

Trả về arc hyperbolic sine.

Ví dụ

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

int main(void)
{
```

```
    printf("asinh 1.8 = %f\n", asinh(1.8));  // 1.350441
}
```

Xem thêm

`acosh()`

13.17 `atanh()`, `atanhf()`, `atanhl()`

Tính arc hyperbolic tangent.

Synopsis

```
#include <math.h>

double atanh(double x);

float atanhf(float x);

long double atanhl(long double x);
```

Mô tả

Các hàm này tính arc hyperbolic tangent của `x`, với `x` phải nằm trong khoảng $[-1, +1]$. Truyền đúng -1 hoặc $+1$ có thể gây pole error.

Giá trị trả về

Trả về arc hyperbolic tangent của `x`.

Ví dụ

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

int main(void)
{
    printf("atanh 0.5 = %f\n", atanh(0.5));  // 0.549306
}
```

Xem thêm

`acosh()`, `asinh()`

13.18 `cosh()`, `coshf()`, `coshl()`

Tính hyperbolic cosine.

Synopsis

```
#include <math.h>

double cosh(double x);

float coshf(float x);

long double coshl(long double x);
```

Mô tả

Như bạn đoán, các hàm này tính hyperbolic cosine của `x`. Range error có thể xảy ra nếu `x` quá lớn.

Giá trị trả về

Trả về hyperbolic cosine của `x`.

Ví dụ

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

int main(void)
{
    printf("cosh 0.5 = %f\n", cosh(0.5));  // 1.127626
}
```

Xem thêm

`sinh()`, `tanh()`

13.19 `sinh()`, `sinhf()`, `sinhl()`

Tính hyperbolic sine.

Synopsis

```
#include <math.h>

double sinh(double x);

float sinhf(float x);

long double sinhl(long double x);
```

Mô tả

Như bạn đoán, các hàm này tính hyperbolic sine của `x`. Range error có thể xảy ra nếu `x` quá lớn.

Giá trị trả về

Trả về hyperbolic sine của `x`.

Ví dụ

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

int main(void)
{
    printf("sinh 0.5 = %f\n", sinh(0.5));  // 0.521095
}
```

Xem thêm

`sinh()`, `tanh()`

13.20 `tanh()`, `tanhf()`, `tanhl()`

Tính hyperbolic tangent.

Synopsis

```
#include <math.h>

double tanh(double x);

float tanhf(float x);

long double tanhl(long double x);
```

Mô tả

Như bạn đoán, các hàm này tính hyperbolic tangent của `x`.

May thay, đây là trang man liên quan trig cuối cùng tôi sẽ viết.

Giá trị trả về

Trả về hyperbolic tangent của `x`.

Ví dụ

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

int main(void)
{
    printf("tanh 0.5 = %f\n", tanh(0.5));  // 0.462117
}
```

Xem thêm

`cosh()`, `sinh()`

13.21 `exp()`, `expf()`, `expl()`

Tính e mũ lên.

Synopsis

```
#include <math.h>

double exp(double x);

float expf(float x);

long double expl(long double x);
```

Mô tả

Tính e^x với e là hằng số Euler[4].

Số e được đặt theo tên Leonard Euler, sinh ngày 15/4/1707, ông là người chịu trách nhiệm, ngoài những việc khác, cho việc làm trang tham chiếu này dài hơn cần thiết.

Giá trị trả về

Trả về e^x.

Ví dụ

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

int main(void)
{
    printf("exp(1) = %f\n", exp(1));  // 2.718282
    printf("exp(2) = %f\n", exp(2));  // 7.389056
}
```

Xem thêm

`exp2()`, `expm1()`, `pow()`, `log()`

13.22 `exp2()`, `exp2f()`, `exp2l()`

Tính 2 mũ lên.

Synopsis

```
#include <math.h>

double exp2(double x);

float exp2f(float x);
```

[4]https://en.wikipedia.org/wiki/E_(mathematical_constant)

```
long double exp2l(long double x);
```

Mô tả

Các hàm này tính 2 mũ một số. Rất thú vị, vì máy tính chơi đùa toàn với luỹ thừa của 2!

Các hàm này có thể nhanh hơn dùng `pow()` để làm cùng việc.

Chúng cũng hỗ trợ số mũ phân số.

Range error xảy ra nếu `x` quá lớn.

Giá trị trả về

`exp2()` trả về 2^x.

Ví dụ

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

int main(void)
{
    printf("2^3 = %f\n", exp2(3));      // 2^3 = 8.000000
    printf("2^8 = %f\n", exp2(8));      // 2^8 = 256.000000
    printf("2^0.5 = %f\n", exp2(0.5));  // 2^0.5 = 1.414214
}
```

Xem thêm

`exp()`, `pow()`

13.23 `expm1()`, `expm1f()`, `expm1l()`

Tính $e^x - 1$.

Synopsis

```
#include <math.h>

double expm1(double x);

float expm1f(float x);

long double expm1l(long double x);
```

Mô tả

Cái này giống hệt `exp()` ngoại trừ—*twist bất ngờ!*—nó tính kết quả đó trừ đi một.

Để thảo luận thêm về e là gì, xem trang man `exp()`.

Nếu `x` khổng lồ, range error có thể xảy ra.

Với các giá trị `x` nhỏ gần không, `expm1(x)` có thể chính xác hơn `exp(x)-1`.

Giá trị trả về

Trả về $e^x - 1$.

Ví dụ

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

int main(void)
{
    printf("%f\n", expm1(2.34));  // 9.381237
}
```

Xem thêm

`exp()`

13.24 `frexp()`, `frexpf()`, `frexpl()`

Tách một số thành phần phân số và phần mũ (theo luỹ thừa của 2).

Synopsis

```
#include <math.h>

double frexp(double value, int *exp);

float frexpf(float value, int *exp);

long double frexpl(long double value, int *exp);
```

Mô tả

Nếu có một số floating point, bạn có thể tách nó thành phần phân số và phần mũ (theo luỹ thừa của 2).

Ví dụ, nếu có số 1234.56, nó có thể biểu diễn dưới dạng bội của luỹ thừa 2 như sau:

$1234.56 = 0.6028125 \times 2^{11}$

Và bạn có thể dùng hàm này để lấy phần 0.6028125 và 11 của phương trình đó.

Còn vì sao, câu trả lời của tôi đơn giản thôi: tôi không biết. Tìm không ra chỗ nào dùng. K&R2 và mọi nguồn tôi tìm được chỉ nói *cách* dùng, chứ không nói *vì sao* bạn lại muốn dùng.

Tài liệu C99 Rationale viết:

> Các hàm `frexp`, `ldexp`, và `modf` là nguyên thuỷ được phần còn lại của thư viện sử dụng.
>
> Có ý kiến muốn bỏ chúng vì lý do tương tự như `ecvt`, `fcvt`, và `gcvt` bị bỏ, nhưng phe ủng hộ giữ chúng lại cho dùng chung. Việc dùng chúng có vấn đề: trên kiến trúc không nhị phân, ldexp có thể mất độ chính xác và frexp có thể không hiệu quả.

Vậy đó. Nếu bạn cần dùng.

Giá trị trả về

`frexp()` trả về phần phân số của `value` trong khoảng 0.5 (bao gồm) đến 1 (không bao gồm), hoặc 0. Và nó lưu số mũ luỹ thừa 2 vào biến mà `exp` trỏ tới.

Nếu bạn truyền vào không, cả giá trị trả về lẫn biến `exp` trỏ tới đều là không.

Ví dụ

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

int main(void)
{
    double frac;
    int expt;

    frac = frexp(1234.56, &expt);
    printf("1234.56 = %.7f x 2^%d\n", frac, expt);
}
```

Output:

```
1234.56 = 0.6028125 x 2^11
```

Xem thêm

`ldexp()`, `ilogb()`, `modf()`

13.25 `ilogb()`, `ilogbf()`, `ilogbl()`

Trả về số mũ của một số floating point.

Synopsis

```
#include <math.h>

int ilogb(double x);

int ilogbf(float x);

int ilogbl(long double x);
```

Mô tả

Cái này cho bạn số mũ của số cho trước... hơi lạ, vì số mũ phụ thuộc vào giá trị của `FLT_RADIX`. Giờ, giá trị này rất thường là `2`—nhưng không có gì đảm bảo!

Thực ra nó trả về $\log_r |x|$ với r là `FLT_RADIX`.

Domain hoặc range error có thể xảy ra với giá trị `x` không hợp lệ, hoặc với giá trị trả về nằm ngoài phạm vi của kiểu trả về.

Giá trị trả về

Số mũ của giá trị tuyệt đối của số đã cho, phụ thuộc `FLT_RADIX`.

Cụ thể là $\log_r |x|$ với r là `FLT_RADIX`.

Nếu bạn truyền vào `0`, nó sẽ trả về `FP_ILOGB0`.

Nếu bạn truyền vào vô cùng, nó sẽ trả về `INT_MAX`.

Nếu bạn truyền vào NaN, nó sẽ trả về `FP_ILOGBNAN`.

Spec còn nói giá trị của `FP_ILOGB0` sẽ là `INT_MIN` hoặc `-INT_MAX`. Và giá trị của `FP_ILOGBNAN` sẽ là `INT_MAX` hoặc `INT_MIN`, phòng khi có ích.

Ví dụ

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

int main(void)
{
    printf("%d\n", ilogb(257));  // 8
    printf("%d\n", ilogb(256));  // 8
    printf("%d\n", ilogb(255));  // 7
}
```

Xem thêm

`frexp()`, `logb()`

13.26 `ldexp()`, `ldexpf()`, `ldexpl()`

Nhân một số với luỹ thừa nguyên của 2.

Synopsis

```
#include <math.h>

double ldexp(double x, int exp);

float ldexpf(float x, int exp);

long double ldexpl(long double x, int exp);
```

Mô tả

Các hàm này nhân số `x` với 2 mũ `exp`.

Giá trị trả về

Trả về $x \times 2^{exp}$.

Ví dụ

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

int main(void)
{
    printf("1 x 2^10 = %f\n", ldexp(1, 10));
    printf("5.67 x 2^7 = %f\n", ldexp(5.67, 7));
}
```

Output:

```
1 x 2^10 = 1024.000000
5.67 x 2^7 = 725.760000
```

Xem thêm

`exp()`

13.27 `log()`, `logf()`, `logl()`

Tính logarithm tự nhiên.

Synopsis

```
#include <math.h>

double log(double x);

float logf(float x);

long double logl(long double x);
```

Mô tả

Logarithm tự nhiên! Và mọi người cùng reo mừng.

Các hàm này tính logarithm cơ số e của một số, $\log_e x$, $\ln x$.

Nói cách khác, với một x cho trước, giải $x = e^y$ tìm y.

Giá trị trả về

Logarithm cơ số e của giá trị đã cho, $\log_e x$, $\ln x$.

Ví dụ

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

int main(void)
{
    const double e = 2.718281828459045;
```

```
    printf("%f\n", log(3490.2));  // 8.157714
    printf("%f\n", log(e));       // 1.000000
}
```

Xem thêm

`exp()`, `log10()`, `log1p()`

13.28 `log10()`, `log10f()`, `log10l()`

Tính log cơ số 10 của một số.

Synopsis

```
#include <math.h>

double log10(double x);

float log10f(float x);

long double log10l(long double x);
```

Mô tả

Ngay khi bạn tưởng mình sắp phải dùng Định luật Logarithm để tính cái này, đây là một hàm xuất hiện bất ngờ để cứu bạn.

Các hàm này tính logarithm cơ số 10 của một số, $\log_{10} x$.

Nói cách khác, với một x cho trước, giải $x = 10^y$ tìm y.

Giá trị trả về

Trả về log cơ số 10 của `x`, $\log_{10} x$.

Ví dụ

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

int main(void)
{
    printf("%f\n", log10(3490.2));   // 3.542850
    printf("%f\n", log10(10));       // 1.000000
}
```

Xem thêm

`pow()`, `log()`

13.29 `log1p()`, `log1pf()`, `log1pl()`

Tính logarithm tự nhiên của một số cộng 1.

Synopsis

```
#include <math.h>

double log1p(double x);

float log1pf(float x);

long double log1pl(long double x);
```

Mô tả

Hàm này tính $\log_e(1 + x)$, $\ln(1 + x)$.

Nó hoạt động y như gọi:

```
log(1 + x)
```

ngoại trừ là có thể chính xác hơn với giá trị `x` nhỏ.

Vậy nếu `x` có độ lớn nhỏ, hãy dùng cái này.

Giá trị trả về

Trả về $\log_e(1 + x)$, $\ln(1 + x)$.

Ví dụ

Tính vài giá trị logarithm lớn và nhỏ để xem khác biệt giữa `log1p()` và `log()`:

```
#include <stdio.h>
#include <float.h> // cho LDBL_DECIMAL_DIG
#include <math.h>

int main(void)
{
    printf("Big log1p()  : %.*Lf\n", LDBL_DECIMAL_DIG-1, log1pl(9));
    printf("Big log()    : %.*Lf\n", LDBL_DECIMAL_DIG-1, logl(1 + 9));

    printf("Small log1p(): %.*Lf\n", LDBL_DECIMAL_DIG-1, log1pl(0.01));
    printf("Small log()  : %.*Lf\n", LDBL_DECIMAL_DIG-1, logl(1 + 0.01));
}
```

Output trên máy tôi:

```
Big log1p()  : 2.30258509299404568403
Big log()    : 2.30258509299404568403
Small log1p(): 0.00995033085316808305
Small log()  : 0.00995033085316809164
```

Xem thêm

`log()`

13.30 `log2()`, `log2f()`, `log2l()`

Tính logarithm cơ số 2 của một số.

Synopsis

```
#include <math.h>

double log2(double x);

float log2f(float x);

long double log2l(long double x);
```

Mô tả

Wow! Bạn nghĩ mình đã xong với các hàm logarithm? Chúng ta chỉ mới bắt đầu!

Hàm này tính $\log_2 x$. Tức là, tính y thoả mãn $x = 2^y$.

Yêu luỹ thừa của 2 đó!

Giá trị trả về

Trả về logarithm cơ số 2 của giá trị đã cho, $\log_2 x$.

Ví dụ

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

int main(void)
{
    printf("%f\n", log2(3490.2));  // 11.769094
    printf("%f\n", log2(256));     // 8.000000
}
```

Xem thêm

`log()`

13.31 `logb()`, `logbf()`, `logbl()`

Trích số mũ của một số theo cơ số `FLT_RADIX`.

Synopsis

```
#include <math.h>

double logb(double x);

float logbf(float x);

long double logbl(long double x);
```

Mô tả

Hàm này trả về phần nguyên của số mũ của số với cơ số `FLT_RADIX`, cụ thể là phần nguyên của $\log_r |x|$ với r là `FLT_RADIX`. Phần phân số bị cắt.

Nếu số là subnormal[5], `logb()` xử lý như thể nó đã được chuẩn hoá.

Nếu `x` là `0`, có thể có domain error hoặc pole error.

Giá trị trả về

Hàm này trả về phần nguyên của $\log_r |x|$ với r là `FLT_RADIX`.

Ví dụ

```
#include <stdio.h>
#include <float.h>  // Cho FLT_RADIX
#include <math.h>

int main(void)
{
    printf("FLT_RADIX = %d\n", FLT_RADIX);
    printf("%f\n", logb(3490.2));
    printf("%f\n", logb(256));
}
```

Output:

```
FLT_RADIX = 2
11.000000
8.000000
```

Xem thêm

`ilogb()`

13.32 `modf()`, `modff()`, `modfl()`

Tách phần nguyên và phần phân số của một số.

Synopsis

```
#include <math.h>

double modf(double value, double *iptr);

float modff(float value, float *iptr);

long double modfl(long double value, long double *iptr);
```

[5] https://en.wikipedia.org/wiki/Denormal_number

Mô tả

Nếu bạn có một số floating point, như `123.456`, hàm này sẽ trích phần nguyên (`123.0`) và phần phân số (`0.456`). Trùng hợp tuyệt đối là đây cũng là nội dung cốt truyện phim hành động mới nhất của Jason Statham.

Cả phần nguyên và phần phân số đều giữ dấu của `value` được truyền vào.

Phần nguyên được lưu vào địa chỉ mà `iptr` trỏ tới.

Xem ghi chú trong `frexp()` về lý do nó nằm trong thư viện.

Giá trị trả về

Các hàm này trả về phần phân số của số. Phần nguyên được lưu vào địa chỉ mà `iptr` trỏ tới. Cả phần nguyên và phần phân số đều giữ dấu của `value` được truyền vào.

Ví dụ

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

void print_parts(double x)
{
    double i, f;

    f = modf(x, &i);

    printf("Entire number  : %f\n", x);
    printf("Integral part  : %f\n", i);
    printf("Fractional part: %f\n\n", f);
}

int main(void)
{
    print_parts(123.456);
    print_parts(-123.456);
}
```

Output:

```
Entire number  : 123.456000
Integral part  : 123.000000
Fractional part: 0.456000

Entire number  : -123.456000
Integral part  : -123.000000
Fractional part: -0.456000
```

Xem thêm

`frexp()`

13.33 `scalbn()`, `scalbnf()`, `scalbnl()` `scalbln()`, `scalblnf()`, `scalblnl()`

Tính $x \times r^n$ hiệu quả, với r là `FLT_RADIX`.

Synopsis

```
#include <math.h>

double scalbn(double x, int n);

float scalbnf(float x, int n);

long double scalbnl(long double x, int n);

double scalbln(double x, long int n);

float scalblnf(float x, long int n);

long double scalblnl(long double x, long int n);
```

Mô tả

Các hàm này tính $x \times r^n$ hiệu quả, với r là `FLT_RADIX`.

Nếu `FLT_RADIX` tình cờ là `2` (không hứa hẹn gì đâu!), cái này chạy giống `exp2()`.

Tên hàm chắc hẳn phải có ý nghĩa rõ ràng với bạn. Rõ ràng là tất cả đều bắt đầu bằng tiền tố "scalb" nghĩa là...

...OK, tôi thú nhận! Tôi chẳng biết nó nghĩa gì. Tra hoài không ra!

Nhưng hãy nhìn vào các hậu tố:

Hậu tố	Ý nghĩa
`n`	`scalbn()` —số mũ `n` là `int`
`nf`	`scalbnf()` —phiên bản `float` của `scalbn()`
`nl`	`scalbnl()` —phiên bản `long double` của `scalbn()`
`ln`	`scalbln()` —số mũ `n` là `long int`
`lnf`	`scalblnf()` —phiên bản `float` của `scalbln()`
`lnl`	`scalblnl()` —phiên bản `long double` của `scalbln()`

Nên dù vẫn mù tịt về "scalb", ít nhất tôi đã nắm được phần đó.

Range error có thể xảy ra với giá trị lớn.

Giá trị trả về

Trả về $x \times r^n$, với r là `FLT_RADIX`.

Ví dụ

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

```
int main(void)
{
    printf("FLT_RADIX = %d\n\n", FLT_RADIX);
    printf("scalbn(3, 8)      = %f\n", scalbn(2, 8));
    printf("scalbnf(10.2, 20) = %f\n", scalbnf(10.2, 20));
}
```

Output trên máy tôi:

```
FLT_RADIX = 2

scalbn(3, 8)       = 512.000000
scalbn(10.2, 20.7) = 10695475.200000
```

Xem thêm

`exp2()`, `pow()`

13.34 `cbrt()`, `cbrtf()`, `cbrtl()`

Tính căn bậc ba.

Synopsis

```
#include <math.h>

double cbrt(double x);

float cbrtf(float x);

long double cbrtl(long double x);
```

Mô tả

Tính căn bậc ba của `x`, $x^{1/3}$, $\sqrt[3]{x}$.

Giá trị trả về

Trả về căn bậc ba của `x`, $x^{1/3}$, $\sqrt[3]{x}$.

Ví dụ

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

int main(void)
{
    printf("cbrt(1729.03) = %f\n", cbrt(1729.03));
}
```

Output:

```
cbrt(1729.03) = 12.002384
```

Xem thêm

`sqrt()`, `pow()`

13.35 `fabs()`, `fabsf()`, `fabsl()`

Tính giá trị tuyệt đối.

Synopsis

```
#include <math.h>

double fabs(double x);

float fabsf(float x);

long double fabsl(long double x);
```

Mô tả

Các hàm này trả về thẳng giá trị tuyệt đối của `x`, tức $|x|$.

Nếu bạn đã quên giá trị tuyệt đối, nó chỉ đơn giản nghĩa là kết quả sẽ dương, kể cả khi `x` là âm. Đơn giản là lột bỏ dấu âm thôi.

Giá trị trả về

Trả về giá trị tuyệt đối của `x`, $|x|$.

Ví dụ

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

int main(void)
{
    printf("fabs(3490.0)  = %f\n", fabs(3490.0));  // 3490.000000
    printf("fabs(-3490.0) = %f\n", fabs(3490.0));  // 3490.000000
}
```

Xem thêm

`abs()`, `copysign()`, `imaxabs()`

13.36 `hypot()`, `hypotf()`, `hypotl()`

Tính độ dài cạnh huyền của tam giác.

Synopsis

```
#include <math.h>

double hypot(double x, double y);

float hypotf(float x, float y);

long double hypotl(long double x, long double y);
```

Mô tả

Fan Định lý Pythagoras[6] mừng lên! Đây là hàm bạn đã chờ đợi!

Nếu bạn biết độ dài hai cạnh góc vuông của tam giác vuông, `x` và `y`, bạn có thể tính độ dài cạnh huyền (cạnh dài nhất, chéo) bằng hàm này.

Cụ thể, nó tính căn bậc hai của tổng bình phương hai cạnh: $\sqrt{x^2 + y^2}$.

Giá trị trả về

Trả về độ dài cạnh huyền của tam giác vuông có độ dài các cạnh là `x` và `y`: $\sqrt{x^2 + y^2}$.

Ví dụ

```
printf("%f\n", hypot(3, 4));  // 5.000000
```

Xem thêm

`sqrt()`

13.37 `pow()`, `powf()`, `powl()`

Tính một giá trị luỹ thừa.

Synopsis

```
#include <math.h>

double pow(double x, double y);

float powf(float x, float y);

long double powl(long double x, long double y);
```

Mô tả

Tính `x` luỹ thừa `y`: x^y.

Các đối số có thể là phân số.

[6] https://en.wikipedia.org/wiki/Pythagorean_theorem

Giá trị trả về

Trả về `x` luỹ thừa `y`: x^y.

Domain error có thể xảy ra nếu:

- `x` là số âm hữu hạn và `y` là số không nguyên hữu hạn
- `x` là không và `y` là không.

Domain error hoặc pole error có thể xảy ra nếu `x` là không và `y` là âm.

Range error có thể xảy ra với giá trị lớn.

Ví dụ

```
printf("%f\n", pow(3, 4));    // 3^4    = 81.000000
printf("%f\n", pow(2, 0.5));  // sqrt 2 = 1.414214
```

Xem thêm

`exp()`, `exp2()`, `sqrt()`, `cbrt()`

13.38 `sqrt()`

Tính căn bậc hai của một số.

Synopsis

```
#include <math.h>

double sqrt(double x);

float sqrtf(float x);

long double sqrtl(long double x);
```

Mô tả

Tính căn bậc hai của một số: $\sqrt{x}$. Với ai chưa biết căn bậc hai là gì, tôi sẽ không giải thích. Chỉ cần nói, căn bậc hai của một số cho ra một giá trị mà khi bình phương (nhân với chính nó) sẽ cho lại số ban đầu.

Ok, được rồi—tôi đã giải thích rồi, nhưng chỉ vì tôi muốn khoe thôi. Không phải kiểu tôi cho bạn ví dụ hay gì đâu, ví dụ như căn bậc hai của chín là ba, bởi vì khi nhân ba với ba bạn được chín, hay gì đó. Không ví dụ. Tôi ghét ví dụ!

Và tôi đoán bạn cũng muốn có vài thông tin thực tế ở đây. Bạn có thể thấy bộ ba hàm quen thuộc ở đây—tất cả đều tính căn bậc hai, nhưng nhận các kiểu đối số khác nhau. Khá đơn giản thôi.

Domain error xảy ra nếu `x` là âm.

Giá trị trả về

Trả về (và tôi biết đây hẳn là điều bất ngờ với bạn) căn bậc hai của `x`: $\sqrt{x}$.

Ví dụ

```
// ví dụ dùng sqrt()

float something = 10;

double x1 = 8.2, y1 = -5.4;
double x2 = 3.8, y2 = 34.9;
double dx, dy;

printf("square root of 10 is %.2f\n", sqrtf(something));

dx = x2 - x1;
dy = y2 - y1;
printf("distance between points (x1, y1) and (x2, y2): %.2f\n",
    sqrt(dx*dx + dy*dy));
```

Và output là:

```
square root of 10 is 3.16
distance between points (x1, y1) and (x2, y2): 40.54
```

Xem thêm

`hypot()`, `pow()`

13.39 `erf()`, `erff()`, `erfl()`

Tính hàm lỗi của giá trị cho trước.

Synopsis

```
#include <math.h>

double erfc(double x);

float erfcf(float x);

long double erfcl(long double x);
```

Mô tả

Các hàm này tính hàm lỗi[7] của một giá trị.

Giá trị trả về

Trả về hàm lỗi của `x`:

$$\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

[7] https://en.wikipedia.org/wiki/Error_function

Ví dụ

```
for (float i = -2; i <= 2; i += 0.5)
    printf("% .1f: %f\n", i, erf(i));
```

Output:

```
-2.0: -0.995322
-1.5: -0.966105
-1.0: -0.842701
-0.5: -0.520500
 0.0: 0.000000
 0.5: 0.520500
 1.0: 0.842701
 1.5: 0.966105
 2.0: 0.995322
```

Xem thêm

`erfc()`

13.40 `erfc()`, `erfcf()`, `erfcl()`

Tính hàm lỗi bù của một giá trị.

Synopsis

```
#include <math.h>

double erfc(double x);

float erfcf(float x);

long double erfcl(long double x);
```

Mô tả

Các hàm này tính hàm lỗi bù[8] của một giá trị.

Cái này giống như:

```
1 - erf(x)
```

Range error có thể xảy ra nếu `x` quá lớn.

Giá trị trả về

Trả về `1 - erf(x)`, cụ thể là:

$$\frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2}\, dt$$

[8] https://en.wikipedia.org/wiki/Error_function

Ví dụ

```
for (float i = -2; i <= 2; i += 0.5)
    printf("% .1f: %f\n", i, erfc(i));
```

Output:

```
-2.0: 1.995322
-1.5: 1.966105
-1.0: 1.842701
-0.5: 1.520500
 0.0: 1.000000
 0.5: 0.479500
 1.0: 0.157299
 1.5: 0.033895
 2.0: 0.004678
```

Xem thêm

`erf()`

13.41 `lgamma()`, `lgammaf()`, `lgammal()`

Tính logarithm tự nhiên của giá trị tuyệt đối của $\Gamma(x)$.

Synopsis

```
#include <math.h>

double lgamma(double x);

float lgammaf(float x);

long double lgammal(long double x);
```

Mô tả

Tính log tự nhiên của giá trị tuyệt đối của hàm gamma[9] `x`, $\log_e |\Gamma(x)|$.

Range error có thể xảy ra nếu `x` quá lớn.

Pole error có thể xảy ra nếu `x` không dương.

Giá trị trả về

Trả về $\log_e |\Gamma(x)|$.

Ví dụ

```
for (float i = 0.5; i <= 4; i += 0.5)
    printf("%.1f: %f\n", i, lgamma(i));
```

Output:

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

```
0.5: 0.572365
1.0: 0.000000
1.5: -0.120782
2.0: 0.000000
2.5: 0.284683
3.0: 0.693147
3.5: 1.200974
4.0: 1.791759
```

Xem thêm

`tgamma()`

13.42 `tgamma()`, `tgammaf()`, `tgammal()`

Tính hàm gamma, $\Gamma(x)$.

Synopsis

```
#include <math.h>

double tgamma(double x);

float tgammaf(float x);

long double tgammal(long double x);
```

Mô tả

Tính hàm gamma[10] của `x`, $\Gamma(x)$.

Domain hoặc pole error có thể xảy ra nếu `x` không dương.

Range error có thể xảy ra nếu `x` quá lớn hoặc quá nhỏ.

Giá trị trả về

Trả về hàm gamma của `x`, $\Gamma(x)$.

Ví dụ

```
for (float i = 0.5; i <= 4; i += 0.5)
    printf("%.1f: %f\n", i, tgamma(i));
```

Output:

```
0.5: 1.772454
1.0: 1.000000
1.5: 0.886227
2.0: 1.000000
2.5: 1.329340
3.0: 2.000000
3.5: 3.323351
```

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

```
4.0: 6.000000
```

Xem thêm

`lgamma()`

13.43 `ceil()`, `ceilf()`, `ceill()`

Ceiling—trả về số nguyên không nhỏ hơn số đã cho.

Synopsis

```
#include <math.h>

double ceil(double x);

float ceilf(float x);

long double ceill(long double x);
```

Mô tả

Trả về ceiling của `x`: $\lceil x \rceil$.

Đây là số nguyên kế tiếp không nhỏ hơn `x`.

Coi chừng con rồng nhỏ này: nó không đơn giản là "làm tròn lên". Ồ, với số dương thì đúng vậy, nhưng với số âm thực ra lại làm tròn về không. (Vì hàm ceiling hướng tới số nguyên lớn hơn kế tiếp, và -4 lớn hơn -5.)

Giá trị trả về

Trả về số nguyên lớn hơn kế tiếp lớn hơn `x`.

Ví dụ

Chú ý với số âm nó hướng về không, tức là hướng tới số nguyên lớn hơn kế tiếp—giống như số dương hướng tới số nguyên lớn hơn kế tiếp.

```
printf("%f\n", ceil(4.0));   //  4.000000
printf("%f\n", ceil(4.1));   //  5.000000
printf("%f\n", ceil(-2.0));  // -2.000000
printf("%f\n", ceil(-2.1));  // -2.000000
printf("%f\n", ceil(-3.1));  // -3.000000
```

Xem thêm

`floor()`, `round()`

13.44 `floor()`, `floorf()`, `floorl()`

Tính số nguyên lớn nhất không lớn hơn giá trị cho trước.

Synopsis

```
#include <math.h>
double floor(double x);
float floorf(float x);
long double floorl(long double x);
```

Mô tả

Trả về floor của giá trị: $\lfloor x \rfloor$. Đây là ngược lại của `ceil()`.

Đây là số nguyên lớn nhất không lớn hơn `x`.

Với số dương, cái này giống làm tròn xuống: `4.5` thành `4.0`.

Với số âm, nó giống làm tròn lên: `-3.6` thành `-4.0`.

Trong cả hai trường hợp, những kết quả đó là số nguyên lớn nhất không lớn hơn số đã cho.

Giá trị trả về

Trả về số nguyên lớn nhất không lớn hơn `x`: $\lfloor x \rfloor$.

Ví dụ

Chú ý cách số âm thực ra làm tròn ra xa không, không giống số dương.

```
printf("%f\n", floor(4.0));   //  4.000000
printf("%f\n", floor(4.1));   //  4.000000
printf("%f\n", floor(-2.0));  // -2.000000
printf("%f\n", floor(-2.1));  // -3.000000
printf("%f\n", floor(-3.1));  // -4.000000
```

Xem thêm

`ceil()`, `round()`

13.45 `nearbyint()`, `nearbyintf()`, `nearbyintl()`

Làm tròn giá trị theo hướng làm tròn hiện tại.

Synopsis

```
#include <math.h>

double nearbyint(double x);

float nearbyintf(float x);

long double nearbyintl(long double x);
```

Mô tả

Hàm này làm tròn `x` đến số nguyên gần nhất theo hướng làm tròn (rounding) hiện tại.

Hướng làm tròn có thể được đặt bằng `fesetround()` trong `<fenv.h>`.

`nearbyint()` không raise floating point exception "inexact".

Giá trị trả về

Trả về `x` được làm tròn theo hướng làm tròn hiện tại.

Ví dụ

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

int main(void)
{
    #pragma STDC FENV_ACCESS ON        // Nếu được hỗ trợ

    fesetround(FE_TONEAREST);          // làm tròn đến gần nhất

    printf("%f\n", nearbyint(3.14));   // 3.000000
    printf("%f\n", nearbyint(3.74));   // 4.000000

    fesetround(FE_TOWARDZERO);         // làm tròn về không

    printf("%f\n", nearbyint(1.99));   // 1.000000
    printf("%f\n", nearbyint(-1.99));  // -1.000000
}
```

Xem thêm

`rint()`, `lrint()`, `round()`, `fesetround()`, `fegetround()`

13.46 `rint()`, `rintf()`, `rintl()`

Làm tròn giá trị theo hướng làm tròn hiện tại.

Synopsis

```
#include <math.h>

double rint(double x);

float rintf(float x);

long double rintl(long double x);
```

Mô tả

Hàm này hoạt động y như `nearbyint()` ngoại trừ là nó có thể raise floating point exception "inexact".

Giá trị trả về

Trả về `x` được làm tròn theo hướng làm tròn hiện tại.

Ví dụ

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

int main(void)
{
    #pragma STDC FENV_ACCESS ON

    fesetround(FE_TONEAREST);

    printf("%f\n", rint(3.14));   // 3.000000
    printf("%f\n", rint(3.74));   // 4.000000

    fesetround(FE_TOWARDZERO);

    printf("%f\n", rint(1.99));   // 1.000000
    printf("%f\n", rint(-1.99));  // -1.000000
}
```

Xem thêm

`nearbyint()`, `lrint()`, `round()`, `fesetround()`, `fegetround()`

13.47 `lrint()`, `lrintf()`, `lrintl()`, `llrint()`, `llrintf()`, `llrintl()`

Trả về `x` được làm tròn theo hướng làm tròn hiện tại dưới dạng số nguyên.

Synopsis

```
#include <math.h>

long int lrint(double x);
long int lrintf(float x);
long int lrintl(long double x);

long long int llrint(double x);
long long int llrintf(float x);
long long int llrintl(long double x);
```

Mô tả

Làm tròn một số floating point theo hướng làm tròn hiện tại, nhưng lần này trả về số nguyên thay vì float. Bạn biết đó, chỉ để thay đổi không khí.

Các hàm này có hai biến thể:

- `lrint()` —trả về `long int`
- `llrint()` —trả về `long long int`

Nếu kết quả không vừa kiểu trả về, domain hoặc range error có thể xảy ra.

Giá trị trả về

Giá trị `x` được làm tròn thành số nguyên theo hướng làm tròn hiện tại.

Ví dụ

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

int main(void)
{
    #pragma STDC FENV_ACCESS ON

    fesetround(FE_TONEAREST);

    printf("%ld\n", lrint(3.14));   // 3
    printf("%ld\n", lrint(3.74));   // 4

    fesetround(FE_TOWARDZERO);

    printf("%ld\n", lrint(1.99));   // 1
    printf("%ld\n", lrint(-1.99));  // -1
}
```

Xem thêm

`nearbyint()`, `rint()`, `round()`, `fesetround()`, `fegetround()`

13.48 `round()`, `roundf()`, `roundl()`

Làm tròn một số theo cách cổ điển.

Synopsis

```
#include <math.h>

double round(double x);

float roundf(float x);

long double roundl(long double x);
```

Mô tả

Làm tròn một số đến giá trị nguyên gần nhất.

Với trường hợp chính giữa, làm tròn ra xa không (tức "tròn lên" về độ lớn).

Trò Jedi của hướng làm tròn hiện tại không có tác dụng với hàm này.

Giá trị trả về

Giá trị làm tròn của `x`.

Ví dụ

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

int main(void)
{
    printf("%f\n", round(3.14));   // 3.000000
    printf("%f\n", round(3.5));    // 4.000000

    printf("%f\n", round(-1.5));   // -2.000000
    printf("%f\n", round(-1.14));  // -1.000000
}
```

Xem thêm

`lround()`, `nearbyint()`, `rint()`, `lrint()`, `trunc()`

13.49 `lround()`, `lroundf()`, `lroundl()` `llround()`, `llroundf()`, `llroundl()`

Làm tròn một số theo cách cổ điển, trả về số nguyên.

Synopsis

```
#include <math.h>

long int lround(double x);
long int lroundf(float x);
long int lroundl(long double x);

long long int llround(double x);
long long int llroundf(float x);
long long int llroundl(long double x);
```

Mô tả

Các hàm này giống như `round()` ngoại trừ chúng trả về số nguyên.

Giá trị chính giữa làm tròn ra xa không, ví dụ 1.5 làm tròn thành 2 và -1.5 làm tròn thành -2.

Các hàm được nhóm theo kiểu trả về:

- `lround()` —trả về `long int`
- `llround()` —trả về `long long int`

Nếu giá trị làm tròn không vừa kiểu trả về, domain hoặc range error có thể xảy ra.

Giá trị trả về

Trả về giá trị làm tròn của `x` dưới dạng số nguyên.

Ví dụ

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

int main(void)
{
    printf("%ld\n", lround(3.14));   // 3
    printf("%ld\n", lround(3.5));    // 4

    printf("%ld\n", lround(-1.5));   // -2
    printf("%ld\n", lround(-1.14));  // -1
}
```

Xem thêm

`round()`, `nearbyint()`, `rint()`, `lrint()`, `trunc()`

13.50 `trunc()`, `truncf()`, `truncl()`

Cắt phần phân số của một giá trị floating point.

Synopsis

```
#include <math.h>

double trunc(double x);

float truncf(float x);

long double truncl(long double x);
```

Mô tả

Các hàm này đơn giản bỏ phần phân số của một số floating point. Bùm.

Nói cách khác, chúng luôn làm tròn về không.

Giá trị trả về

Trả về số floating point đã bị cắt.

Ví dụ

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

int main(void)
{
    printf("%f\n", trunc(3.14));   // 3.000000
    printf("%f\n", trunc(3.8));    // 3.000000

    printf("%f\n", trunc(-1.5));   // -1.000000
    printf("%f\n", trunc(-1.14));  // -1.000000
```

```
}
```

Xem thêm

`round()`, `lround()`, `nearbyint()`, `rint()`, `lrint()`

13.51 `fmod()`, `fmodf()`, `fmodl()`

Tính phần dư floating point.

Synopsis

```
#include <math.h>

double fmod(double x, double y);

float fmodf(float x, float y);

long double fmodl(long double x, long double y);
```

Mô tả

Trả về phần dư của $\frac{x}{y}$. Kết quả có cùng dấu với `x`.

Dưới vỏ bọc, phép tính thực hiện là:

```
x - trunc(x / y) * y
```

Nhưng dễ hơn thì cứ nghĩ về phần dư thôi.

Giá trị trả về

Trả về phần dư của $\frac{x}{y}$ cùng dấu với `x`.

Ví dụ

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

int main(void)
{
    printf("%f\n", fmod(-9.2, 5.1));  // -4.100000
    printf("%f\n", fmod(9.2, 5.1));   //  4.100000
}
```

Xem thêm

`remainder()`

13.52 `remainder()`, `remainderf()`, `remainderl()`

Tính phần dư kiểu IEC 60559.

Synopsis

```
#include <math.h>

double remainder(double x, double y);

float remainderf(float x, float y);

long double remainderl(long double x, long double y);
```

Mô tả

Cái này tương tự `fmod()`, nhưng không hoàn toàn giống. `fmod()` có lẽ là cái bạn muốn nếu bạn mong phần dư xoay vòng như đồng hồ đo đường xe chạy (odometer).

Spec C trích IEC 60559 về cách nó hoạt động:

> Khi $y \neq 0$, phần dư $r = x$ REM y được xác định bất kể mode làm tròn bằng quan hệ toán học $r = x - ny$, với n là số nguyên gần nhất với giá trị chính xác của x/y; khi $|n - x/y| = 1/2$, thì n là số chẵn. Nếu $r = 0$, dấu của nó sẽ là dấu của x.

Hy vọng đã sáng tỏ!

OK, có thể chưa. Tóm lại:

Bạn biết là nếu `fmod()` cho gì đó, ví dụ `2.0`, bạn được kết quả nào đó nằm giữa `0.0` và `2.0`? Và nếu bạn cứ tăng số mà bạn đang lấy mod với `2.0`, bạn thấy kết quả leo lên `2.0` rồi xoay vòng về `0.0` như đồng hồ đo đường của ô tô?

`remainder()` hoạt động y vậy, ngoại trừ nếu `y` là `2.0`, nó xoay vòng từ `-1.0` đến `1.0` thay vì từ `0.0` đến `2.0`.

Nói cách khác, phạm vi của hàm chạy từ `-y/2` đến `y/2`. Khác với `fmod()` chạy từ `0.0` đến `y`, output của `remainder()` chỉ là dịch xuống nửa `y`.

Và không-dư-bất-kỳ-cái-gì là `0`.

Ngoại trừ nếu `y` là không, hàm có thể trả về không hoặc domain error có thể xảy ra.

Giá trị trả về

Kết quả IEC 60559 của `x` -dư- `y`.

Ví dụ

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

int main(void)
{
    printf("%f\n", remainder(3.7, 4));  // -0.300000
    printf("%f\n", remainder(4.3, 4));  //  0.300000
}
```

Xem thêm

`fmod()`, `remquo()`

13.53 `remquo()`, `remquof()`, `remquol()`

Tính phần dư và (một phần của) thương.

Synopsis

```
#include <math.h>

double remquo(double x, double y, int *quo);

float remquof(float x, float y, int *quo);

long double remquol(long double x, long double y, int *quo);
```

Mô tả

Đây là một thứ hơi lạ đời.

Thứ nhất, giá trị trả về là phần dư, giống hàm `remainder()`, nên xem cái đó.

Và thương trở về trong con trỏ `quo`.

Hoặc ít nhất *một phần của nó.* Bạn sẽ nhận ít nhất 3 bit của thương.

Nhưng *vì sao*?

Vài lý do.

Một là thương của một số floating point rất lớn có thể dễ dàng quá khổng lồ để vừa cả `long long unsigned int`. Nên dù sao cũng phải cắt bớt một phần.

Nhưng 3 bit? Có ích gì? Chỉ cho bạn từ 0 đến 7!

Tài liệu C99 Rationale nói:

> Các hàm `remquo` dành cho việc triển khai các phép giảm đối số có thể khai thác vài bit thấp của thương. Lưu ý rằng x có thể quá lớn về độ lớn so với y đến mức biểu diễn chính xác của thương là không khả thi.

Vậy là... triển khai các phép giảm đối số... có thể khai thác vài bit thấp... Ồ ờ okay.

CPPReference có nói thế này[11] về chuyện này, nói hay quá nên tôi sẽ trích nguyên văn:

> Hàm này hữu ích khi triển khai các hàm tuần hoàn có chu kỳ biểu diễn chính xác được bằng giá trị floating point: khi tính $\sin(\pi x)$ với `x` rất lớn, gọi `sin` trực tiếp có thể cho sai số lớn, nhưng nếu đối số hàm được giảm trước bằng `remquo`, các bit thấp của thương có thể dùng để xác định dấu và octant của kết quả trong chu kỳ, còn phần dư có thể dùng để tính giá trị với độ chính xác cao.

Đấy. Nếu bạn có ví dụ khác dùng được... xin chúc mừng! :)

Giá trị trả về

Trả về giống `remainder`: Kết quả IEC 60559 của `x` -dư- `y`.

Ngoài ra, ít nhất 3 bit thấp nhất của thương sẽ được lưu vào `quo` với cùng dấu của `x/y`.

Ví dụ

Có ví dụ `cos()` hay ở CPPReference[12] bao quát một use case thực tế.

[11] https://en.cppreference.com/w/c/numeric/math/remquo
[12] https://en.cppreference.com/w/c/numeric/math/remquo

Nhưng thay vì "mượn" nó, tôi sẽ đăng một ví dụ đơn giản ở đây và bạn có thể ghé trang họ để xem ví dụ thật.

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

int main(void)
{
    int quo;
    double rem;

    rem = remquo(12.75, 2.25, &quo);

    printf("%d remainder %f\n", quo, rem);  // 6 remainder -0.750000
}
```

Xem thêm

`remainder()`, `imaxdiv()`

13.54 `copysign()`, `copysignf()`, `copysignl()`

Chép dấu của một giá trị sang một giá trị khác.

Synopsis

```
#include <math.h>

double copysign(double x, double y);

float copysignf(float x, float y);

long double copysignl(long double x, long double y);
```

Mô tả

Các hàm này trả về một số có độ lớn của `x` và dấu của `y`. Bạn có thể dùng chúng để ép dấu theo giá trị khác.

Tất nhiên là cả `x` lẫn `y` đều không bị thay đổi. Giá trị trả về chứa kết quả.

Giá trị trả về

Trả về giá trị có độ lớn của `x` và dấu của `y`.

Ví dụ

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

int main(void)
{
    double x = 34.9;
    double y = -999.9;
```

```
    double z = 123.4;

    printf("%f\n", copysign(x, y)); // -34.900000
    printf("%f\n", copysign(x, z)); //  34.900000
}
```

Xem thêm

`signbit()`

13.55 `nan()`, `nanf()`, `nanl()`

Trả về `NAN`.

Synopsis

```
#include <math.h>

double nan(const char *tagp);

float nanf(const char *tagp);

long double nanl(const char *tagp);
```

Mô tả

Các hàm này trả về một quiet NaN[13]. Nó được tạo như thể gọi `strtod()` với `"NAN"` (hoặc biến thể của nó) làm đối số.

`tagp` trỏ tới một chuỗi có thể là nhiều thứ, bao gồm cả rỗng. Nội dung của chuỗi quyết định biến thể NaN nào có thể được trả về tuỳ implementation.

Phiên bản NaN nào? Bạn có biết là có thể đi sâu tới mức này với một thứ không phải là số không?

Trường hợp 1, bạn truyền vào chuỗi rỗng, trong trường hợp đó các dòng sau tương đương:

```
nan("");

strtod("NAN()", NULL);
```

Trường hợp 2, chuỗi chỉ chứa chữ số 0-9, chữ cái a-z, chữ cái A-Z, và/hoặc gạch dưới:

```
nan("goats");

strtod("NAN(goats)", NULL);
```

Và Trường hợp 3, chuỗi chứa bất cứ thứ gì khác và bị bỏ qua:

```
nan("!");

strtod("NAN", NULL);
```

Còn `strtod()` làm gì với các giá trị trong ngoặc đơn, xem trang tham chiếu [`strtod()`]. Spoiler: phụ thuộc implementation.

[13] Một *quiet NaN* là NaN không raise bất kỳ exception nào.

Giá trị trả về

Trả về quiet NaN được yêu cầu, hoặc 0 nếu hệ thống của bạn không hỗ trợ những thứ này.

Ví dụ

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

int main(void)
{
    printf("%f\n", nan(""));       // nan
    printf("%f\n", nan("goats"));  // nan
    printf("%f\n", nan("!"));      // nan
}
```

Xem thêm

`strtod()`

13.56 `nextafter()`, `nextafterf()`, `nextafterl()`

Lấy giá trị floating point kế tiếp (hoặc trước đó) biểu diễn được.

Synopsis

```
#include <math.h>

double nextafter(double x, double y);

float nextafterf(float x, float y);

long double nextafterl(long double x, long double y);
```

Mô tả

Như bạn có lẽ đã biết, số floating point không thể biểu diễn *mọi* số thực khả dĩ. Có giới hạn.

Và, như vậy, tồn tại một số "kế tiếp" và "trước đó" sau hoặc trước bất kỳ số floating point nào.

Các hàm này trả về số kế tiếp (hoặc trước đó) biểu diễn được. Nghĩa là, không có số floating point nào tồn tại giữa số đã cho và số kế tiếp.

Cách nó xác định là nó chạy từ `x` theo hướng `y`, trả lời câu hỏi "số kế tiếp biểu diễn được từ `x` khi ta đi về phía `y` là gì".

Giá trị trả về

Trả về giá trị floating point kế tiếp biểu diễn được từ `x` theo hướng `y`.

Nếu `x` bằng `y`, trả về `y`. Và cả `x` nữa, tôi đoán vậy.

Ví dụ

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

int main(void)
{
    printf("%.*f\n", DBL_DECIMAL_DIG, nextafter(0.5, 1.0));
    printf("%.*f\n", DBL_DECIMAL_DIG, nextafter(0.349, 0.0));
}
```

Output trên máy tôi:

```
0.50000000000000011
0.34899999999999992
```

Xem thêm

`nexttoward()`

13.57 `nexttoward()`, `nexttowardf()`, `nexttowardl()`

Lấy giá trị floating point kế tiếp (hoặc trước đó) biểu diễn được.

Synopsis

```
include <math.h>

double nexttoward(double x, long double y);

float nexttowardf(float x, long double y);

long double nexttowardl(long double x, long double y);
```

Mô tả

Các hàm này giống `nextafter()` ngoại trừ tham số thứ hai luôn là `long double`.

Giá trị trả về

Trả về giống `nextafter()` ngoại trừ nếu `x` bằng `y`, trả về `y` cast sang kiểu trả về của hàm.

Ví dụ

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

int main(void)
{
    printf("%.*f\n", DBL_DECIMAL_DIG, nexttoward(0.5, 1.0));
    printf("%.*f\n", DBL_DECIMAL_DIG, nexttoward(0.349, 0.0));
}
```

Output trên máy tôi:

```
0.50000000000000011
0.34899999999999992
```

Xem thêm

`nextafter()`

13.58 `fdim()`, `fdimf()`, `fdiml()`

Trả về hiệu số dương giữa hai số, chặn dưới ở 0.

Synopsis

```
#include <math.h>

double fdim(double x, double y);

float fdimf(float x, float y);

long double fdiml(long double x, long double y);
```

Mô tả

Hiệu số dương giữa `x` và `y` là hiệu... ngoại trừ nếu hiệu nhỏ hơn `0`, nó bị kẹp ở `0`.

Các hàm này có thể ném range error.

Giá trị trả về

Trả về hiệu của `x-y` nếu hiệu lớn hơn `0`. Ngược lại trả về `0`.

Ví dụ

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

int main(void)
{
    printf("%f\n", fdim(10.0, 3.0));   // 7.000000
    printf("%f\n", fdim(3.0, 10.0));   // 0.000000, bị kẹp
}
```

13.59 `fmax()`, `fmaxf()`, `fmaxl()`, `fmin()`, `fminf()`, `fminl()`

Trả về giá trị lớn nhất hoặc nhỏ nhất trong hai số.

Synopsis

```
#include <math.h>

double fmax(double x, double y);

float fmaxf(float x, float y);

long double fmaxl(long double x, long double y);

double fmin(double x, double y);

float fminf(float x, float y);

long double fminl(long double x, long double y);
```

Mô tả

Đơn giản, các hàm này trả về giá trị nhỏ nhất hoặc lớn nhất trong hai số đã cho.

Nếu một trong các số là NaN, các hàm trả về số không phải NaN. Nếu cả hai đối số đều là NaN, các hàm trả về NaN.

Giá trị trả về

Trả về giá trị nhỏ nhất hoặc lớn nhất, với NaN được xử lý như trên.

Ví dụ

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

int main(void)
{
    printf("%f\n", fmin(10.0, 3.0));   //  3.000000
    printf("%f\n", fmax(3.0, 10.0));   // 10.000000
}
```

13.60 `fma()`, `fmaf()`, `fmal()`

Nhân và cộng Floating (còn gọi là “Fast”).

Synopsis

```
#include <math.h>

double fma(double x, double y, double z);

float fmaf(float x, float y, float z);

long double fmal(long double x, long double y, long double z);
```

Mô tả

Hàm này thực hiện phép toán $(x \times y) + z$, nhưng theo một cách rất đỉnh. Nó làm phép tính như thể có độ chính xác vô hạn, rồi làm tròn kết quả cuối cùng về kiểu dữ liệu cuối cùng theo mode làm tròn hiện tại.

So với cách bạn tự làm toán, trong đó có thể làm tròn ở mỗi bước.

Ngoài ra, một số kiến trúc có lệnh CPU để làm đúng phép tính này, nên có thể làm siêu nhanh. (Nếu không có, đáng kể chậm hơn.)

Bạn có thể biết CPU của mình có hỗ trợ phiên bản nhanh không bằng cách kiểm tra macro `FP_FAST_FMA` được đặt thành `1`. (Biến thể `float` và `long` của `fma()` có thể được kiểm tra với `FP_FAST_FMAF` và `FP_FAST_FMAL` tương ứng.)

Các hàm này có thể gây range error.

Giá trị trả về

Trả về `(x * y) + z`.

Ví dụ

```
printf("%f\n", fma(1.0, 2.0, 3.0));  // 5.000000
```

13.61 `isgreater()`, `isgreaterequal()`, `isless()`, `islessequal()`

Các macro so sánh floating point.

Synopsis

```
#include <math.h>

int isgreater(any_floating_type x, any_floating_type y);

int isgreaterequal(any_floating_type x, any_floating_type y);

int isless(any_floating_type x, any_floating_type y);

int islessequal(any_floating_type x, any_floating_type y);
```

Mô tả

Các macro này so sánh các số floating point. Vì là macro, ta có thể truyền bất kỳ kiểu floating point nào.

Bạn có thể nghĩ mình đã làm được điều đó chỉ với các toán tử so sánh bình thường—và bạn nói đúng!

Trừ một ngoại lệ: toán tử so sánh raise floating exception "invalid" nếu một hoặc nhiều toán hạng là NaN. Các macro này không như vậy.

Lưu ý là bạn chỉ được truyền kiểu floating point vào các hàm này. Truyền số nguyên hay bất kỳ kiểu nào khác là hành vi không xác định.

Giá trị trả về

`isgreater()` trả về kết quả của `x > y`.

`isgreaterequal()` trả về kết quả của `x >= y`.

`isless()` trả về kết quả của `x < y`.

`islessequal()` trả về kết quả của `x <= y`.

Ví dụ

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

int main(void)
{
    printf("%d\n", isgreater(10.0, 3.0));        // 1
    printf("%d\n", isgreaterequal(10.0, 10.0));  // 1
    printf("%d\n", isless(10.0, 3.0));           // 0
    printf("%d\n", islessequal(10.0, 3.0));      // 0
}
```

Xem thêm

`islessgreater()`, `isunordered()`

13.62 `islessgreater()`

Kiểm tra số floating point này nhỏ hơn hoặc lớn hơn số kia.

Synopsis

```
#include <math.h>

int islessgreater(any_floating_type x, any_floating_type y);
```

Mô tả

Macro này tương tự `isgreater()` và đồng bọn, nhưng nó làm tên section quá dài nếu tôi bao nó vào trên kia. Nên nó được có chỗ riêng.

Cái này trả về true nếu $x < y$ hoặc $x > y$.

Dù là macro, ta có thể yên tâm rằng `x` và `y` chỉ được tính giá trị một lần.

Và kể cả khi `x` hoặc `y` là NaN, cái này sẽ không ném exception "invalid", không giống các toán tử so sánh thường.

Nếu bạn truyền kiểu không phải floating point, hành vi không xác định.

Giá trị trả về

Trả về `(x < y) || (x > y)`.

Ví dụ

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

int main(void)
{
    printf("%d\n", islessgreater(10.0, 3.0));   // 1
    printf("%d\n", islessgreater(10.0, 30.0));  // 1
    printf("%d\n", islessgreater(10.0, 10.0));  // 0
}
```

Xem thêm

`isgreater()`, `isgreaterequal()`, `isless()`, `islessequal()`, `isunordered()`

13.63 `isunordered()`

Macro trả về true nếu bất kỳ đối số floating point nào là NaN.

Synopsis

```
#include <math.h>

int isunordered(any_floating_type x, any_floating_type y);
```

Mô tả

Spec viết:

> Macro isunordered xác định xem các đối số của nó có unordered không.

Thấy chưa? Tôi đã nói C dễ mà!

Nó cũng nói thêm là các đối số là unordered nếu một hoặc cả hai là NaN.

Giá trị trả về

Macro này trả về true nếu một hoặc cả hai đối số là NaN.

Ví dụ

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

int main(void)
{
    printf("%d\n", isunordered(1.0, 2.0));       // 0
    printf("%d\n", isunordered(1.0, sqrt(-1)));  // 1
    printf("%d\n", isunordered(NAN, 30.0));      // 1
    printf("%d\n", isunordered(NAN, NAN));       // 1
}
```

Xem thêm

`isgreater()`, `isgreaterequal()`, `isless()`, `islessequal()`, `islessgreater()`

Chapter 14

`<setjmp.h>` Goto Không Cục Bộ

Các hàm này cho phép bạn tua ngược call stack về một điểm trước đó, với một đống cái bẫy đi kèm. Hiếm khi được dùng.

Hàm	Mô tả
`longjmp()`	Quay về chỗ đánh dấu đã đặt trước đó
`setjmp()`	Đánh dấu chỗ này để quay về sau

Còn có một kiểu mờ (opaque type) mới, `jmp_buf`, giữ toàn bộ thông tin cần thiết để làm được trò ảo thuật này.

Nếu bạn muốn biến local tự động của mình vẫn đúng sau lời gọi `longjmp()`, khai báo chúng là `volatile` ở chỗ bạn gọi `setjmp()`.

14.1 `setjmp()`

Lưu vị trí này để quay về sau

Synopsis

```
#include <setjmp.h>

int setjmp(jmp_buf env);
```

Mô tả

Đây là cách bạn lưu vị trí để sau này có thể `longjmp()` quay về. Coi như đang set điểm warp để dùng sau.

Về cơ bản, bạn gọi hàm này, đưa cho nó một `env` để nó điền vào mọi thông tin cần thiết để quay lại đây sau. Cái `env` này là thứ bạn sẽ truyền cho `longjmp()` sau này khi muốn teleport về đây.

Và phần funky thực sự là hàm này có thể trả về theo hai kiểu khác nhau:

1. Có thể trả về `0` từ lời gọi set up điểm jump đích.
2. Có thể trả về non-zero khi bạn thực sự warp về đây từ lời gọi `longjmp()`.

Bạn có thể kiểm tra giá trị trả về để biết case nào đang xảy ra.

Bạn chỉ được gọi `setjmp()` trong một số trường hợp giới hạn.

1. Như một biểu thức đứng riêng:

   ```
   setjmp(env);
   ```

   Bạn cũng có thể cast nó về `(void)` nếu vì lý do gì đó muốn làm vậy.

2. Như biểu thức điều khiển đầy đủ trong `if` hoặc `switch`.

   ```
   if (setjmp(env)) { ... }

   switch (setjmp(env)) { ... }
   ```

   Nhưng không được thế này, vì nó không phải biểu thức điều khiển đầy đủ trong trường hợp này:

   ```
   if (x == 2 && setjmp()) { ... }   // Undefined behavior
   ```

3. Giống (2) phía trên, trừ việc có so sánh với integer constant:

   ```
   if (setjmp(env) == 0) { ... }

   if (setjmp(env) > 2) { ... }
   ```

4. Như toán hạng cho toán tử phủ định (`!`):

   ```
   if (!setjmp(env)) { ... }
   ```

Mọi thứ khác đều là (bạn đoán đúng rồi) undefined behavior!

Đây có thể là macro hoặc hàm, nhưng bạn xử nó cùng cách trong mọi trường hợp.

Giá trị trả về

Cái này funky. Nó trả về một trong hai thứ:

Trả về `0` nếu đây là lời gọi `setjmp()` để set up.

Trả về non-zero nếu việc đang ở đây là kết quả của lời gọi `longjmp()`. (Cụ thể, nó trả về giá trị được truyền vào hàm `longjmp()`.)

Ví dụ

Đây là một hàm gọi `setjmp()` để set up (nơi nó trả về `0`), rồi gọi xuống sâu vài level vào các hàm, và cuối cùng cắt ngang đường return bằng `longjmp()` quay về chỗ `setjmp()` đã được gọi trước đó. Lần này, nó truyền `3490` làm giá trị, và `setjmp()` sẽ trả về giá trị đó.

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

jmp_buf env;

void depth2(void)
{
    printf("Entering depth 2\n");
    longjmp(env, 3490);           // Nhảy về`setjmp()!!
    printf("Leaving depth 2\n");  // Cái này không chạy
}
```

```
void depth1(void)
{
    printf("Entering depth 1\n");
    depth2();
    printf("Leaving depth 1\n");  // Cái này không chạy
}

int main(void)
{
    switch (setjmp(env)) {
        case 0:
            printf("Calling into functions, setjmp() returned 0\n");
            depth1();
            printf("Returned from functions\n");  // Cái này không chạy
            break;

        case 3490:
            printf("Bailed back to main, setjmp() returned 3490\n");
            break;
    }
}
```

Chạy thì output:

```
Calling into functions, setjmp() returned 0
Entering depth 1
Entering depth 2
Bailed back to main, setjmp() returned 3490
```

Chú ý `printf()` thứ hai trong `case 0` không chạy; nó bị `longjmp()` nhảy qua!

Xem thêm

`longjmp()`

14.2 `longjmp()`

Quay về vị trí `setjmp()` trước đó

Synopsis

```
 #include <setjmp.h>

_Noreturn void longjmp(jmp_buf env, int val);
```

Mô tả

Hàm này trả về một lời gọi `setjmp()` trước đó trong call history. `setjmp()` sẽ trả về giá trị `val` được truyền vào `longjmp()`.

`env` truyền cho `setjmp()` nên là chính cái bạn truyền cho `longjmp()`.

Có cả đống vấn đề tiềm tàng khi làm chuyện này, nên bạn phải cẩn thận tránh undefined behavior bằng cách không làm mấy điều sau:

1. Đừng gọi `longjmp()` nếu lời gọi `setjmp()` tương ứng nằm trong thread khác.
2. Đừng gọi `longjmp()` nếu bạn chưa gọi `setjmp()` trước đó.
3. Đừng gọi `longjmp()` nếu hàm đã gọi `setjmp()` đã hoàn thành.
4. Đừng gọi `longjmp()` nếu lời gọi `setjmp()` có một variable length array (VLA) trong scope và scope đó đã kết thúc.
5. Đừng gọi `longjmp()` nếu có bất kỳ VLA nào trong các scope đang active giữa `setjmp()` và `longjmp()`. Nguyên tắc nằm lòng ở đây là đừng mix VLA với `longjmp()`.

Dù `longjmp()` cố gắng khôi phục máy về trạng thái lúc `setjmp()`, gồm cả biến local, vẫn có vài thứ không được hồi sinh lại:

- Biến local không phải volatile mà có thể đã thay đổi
- Floating point status flags
- File đang mở
- Bất kỳ thành phần nào khác của abstract machine

Giá trị trả về

Cái này cũng funky ở chỗ nó là một trong số ít hàm của C không bao giờ trả về!

Ví dụ

Đây là một hàm gọi `setjmp()` để set up (nơi nó trả về `0`), rồi gọi xuống sâu vài level vào các hàm, và cuối cùng cắt ngang đường return bằng `longjmp()` quay về chỗ `setjmp()` đã được gọi trước đó. Lần này, nó truyền `3490` làm giá trị, và `setjmp()` sẽ trả về giá trị đó.

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

jmp_buf env;

void depth2(void)
{
    printf("Entering depth 2\n");
    longjmp(env, 3490);           // Nhảy về`setjmp()!!
    printf("Leaving depth 2\n");  // Cái này không chạy
}

void depth1(void)
{
    printf("Entering depth 1\n");
    depth2();
    printf("Leaving depth 1\n");  // Cái này không chạy
}

int main(void)
{
    switch (setjmp(env)) {
        case 0:
            printf("Calling into functions, setjmp() returned 0\n");
            depth1();
            printf("Returned from functions\n");  // Cái này không chạy
            break;

        case 3490:
```

```
            printf("Bailed back to main, setjmp() returned 3490\n");
            break;
    }
}
```

Chạy thì output:

```
Calling into functions, setjmp() returned 0
Entering depth 1
Entering depth 2
Bailed back to main, setjmp() returned 3490
```

Chú ý `printf()` thứ hai trong `case 0` không chạy; nó bị `longjmp()` nhảy qua!

Xem thêm

`setjmp()`

Chapter 15

`<signal.h>` Xử Lý Signal

Hàm	Mô tả
`signal()`	Đặt signal handler cho một signal
`raise()`	Làm cho một signal được raise lên

Xử lý signal kiểu portable, đại khái!

Các signal này được raise vì đủ lý do như người dùng bấm CTRL-C, yêu cầu terminate từ chương trình ngoài, vi phạm truy cập bộ nhớ, vân vân.

OS của bạn nhiều khả năng còn định nghĩa thêm cả đống signal khác.

Hệ thống này khá giới hạn, như bạn sẽ thấy bên dưới. Nếu bạn trên Unix, gần như chắc chắn OS của bạn có khả năng xử lý signal xịn hơn hẳn so với thư viện chuẩn C. Xem `sigaction`[1].

15.1 `signal()`

Đặt signal handler cho một signal

Synopsis

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

Mô tả

Cái khai báo hàm *đó* nhìn sao ta?

Thôi bỏ qua nó một chút, nói về hàm này làm gì đã.

Khi một signal được raise, *điều gì đó* sẽ xảy ra. Hàm này cho phép bạn chọn một trong các việc sau để làm khi signal xảy ra:

- Bỏ qua signal
- Thực hiện hành động mặc định
- Có một hàm cụ thể được gọi

[1] https://man.archlinux.org/man/sigaction.2.en

Hàm `signal()` nhận hai tham số. Tham số đầu, `sig`, là tên signal cần xử lý.

Signal	Mô tả
`SIGABRT`	Được raise khi `abort()` được gọi
`SIGFPE`	Exception số học floating-point
`SIGILL`	CPU thử thực thi lệnh không hợp lệ
`SIGINT`	Signal interrupt, như khi `CTRL-C` được bấm
`SIGSEGV`	Segmentation Violation: thử truy cập bộ nhớ bị cấm
`SIGTERM`	Yêu cầu terminate[2]

Vậy đó là phần đầu khi bạn gọi `signal()` —nói cho nó biết signal cần xử:

```
signal(SIGINT, ...
```

Nhưng tham số `func` kia là gì?

Tiết lộ luôn, đó là pointer tới một hàm nhận tham số `int` và trả về `void`. Ta có thể dùng nó để gọi một hàm tuỳ ý khi signal xảy ra.

Trước khi làm chuyện đó, xem mấy cái dễ trước: bảo hệ thống bỏ qua signal hoặc thực hiện hành động mặc định (đây là cái nó làm mặc định nếu bạn không bao giờ gọi `signal()`).

Bạn có thể đặt `func` bằng một trong hai giá trị đặc biệt để làm chuyện này:

`func`	Mô tả
`SIG_DFL`	Thực hiện hành động mặc định cho signal này
`SIG_IGN`	Bỏ qua signal này

Ví dụ:

```
signal(SIGTERM, SIG_DFL);  // Hành động mặc định với SIGTERM
signal(SIGINT, SIG_IGN);   // Bỏ qua SIGINT
```

Nhưng nếu bạn muốn handler của riêng mình làm gì đó thay vì default hay ignore thì sao? Bạn có thể truyền vào một hàm của bạn để được gọi. Đó là lý do cái function signature điên rồ kia tồn tại (một phần). Nó nói rằng tham số có thể là con trỏ tới hàm nhận tham số `int` và trả về `void`.

Vậy nếu bạn muốn gọi handler của mình, bạn có thể viết code thế này:

```
int handler(int sig)
{
    // Xử lý signal
}

int main(void)
{
    signal(SIGINT, handler);
```

Trong signal handler bạn có thể làm gì? Không nhiều.

Nếu signal là do `abort()` hoặc `raise()`, handler không được gọi `raise()`.

Nếu signal **không** phải do `abort()` hoặc `raise()`, bạn chỉ được gọi mấy hàm sau từ thư viện chuẩn (dù spec không cấm gọi các hàm non-library khác):

[2] Như kiểu được gửi từ lệnh `kill` trên Unix.]

- `abort()`
- `_Exit()`
- `quick_exit()`
- Hàm trong `<stdatomic.h>` khi tham số atomic là lock-free
- `signal()` với tham số đầu bằng tham số được truyền vào handler

Thêm nữa, nếu signal **không** phải do `abort()` hoặc `raise()`, handler không được truy cập bất kỳ object nào có static hoặc thread-storage duration trừ khi nó là lock-free.

Ngoại lệ là bạn có thể gán cho biến kiểu `volatile sig_atomic_t`. Spec không rõ liệu bạn có đọc được biến kiểu đó không. Tôi nghĩ bạn làm vậy an toàn, nhưng lúc bạn quyết định vừa đọc vừa ghi biến đó trong handler, bạn đã mở cửa cho khả năng race condition. Chắc tốt nhất là chỉ set nó như một flag báo signal đã xảy ra và để thế giới bên ngoài xử lý.

Tuỳ implementation, nhưng signal handler có thể bị reset về `SIG_DFL` ngay trước khi handler được gọi.

Gọi `signal()` trong chương trình multithread là undefined behavior.

Return từ handler cho `SIGFPE`, `SIGILL`, `SIGSEGV`, hoặc bất kỳ giá trị implementation-defined nào là undefined behavior. Bạn phải exit.

Implementation có thể ngăn hoặc không ngăn các signal khác phát sinh khi đang ở trong signal handler.

Giá trị trả về

Khi thành công, `signal()` trả về con trỏ tới signal handler trước đó đã được đặt qua lời gọi `signal()` cho signal number đó. Nếu bạn chưa từng gọi `signal()` để đặt, trả về `SIG_DFL`.

Khi thất bại, `SIG_ERR` được trả về và `errno` được set thành một giá trị dương.

Ví dụ

Đây là chương trình khiến `SIGINT` bị bỏ qua. Thông thường bạn trigger signal này bằng cách bấm `CTRL-C`.

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

int main(void)
{
    signal(SIGINT, SIG_IGN);

    printf("You can't hit CTRL-C to exit this program. Try it!\n\n");
    printf("Press return to exit, instead.");
    fflush(stdout);
    getchar();
}
```

Output:

```
You can't hit CTRL-C to exit this program. Try it!

Press return to exit, instead.^C^C^C^C^C^C^C^C^C^C^C
```

Chương trình này đặt signal handler, rồi raise signal. Signal handler được kích hoạt.

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

```
void handler(int sig)
{
    // Undefined behavior khi gọi printf() nếu handler này không
    // phải do raise(), tức là nếu bạn bấm CTRL-C.

    printf("Got signal %d!\n", sig);

    // Thường reset handler phòng khi implementation set nó về`
    // SIG_DFL khi signal xảy ra.

    signal(sig, handler);
}

int main(void)
{
    signal(SIGINT, handler);

    raise(SIGINT);
    raise(SIGINT);
    raise(SIGINT);
}
```

Output:

```
Got signal 2!
Got signal 2!
Got signal 2!
```

Ví dụ này bắt `SIGINT` rồi set một flag thành `1`. Rồi main loop thấy flag và exit.

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

volatile sig_atomic_t x;

void handler(int sig)
{
    x = 1;
}

int main(void)
{
    signal(SIGINT, handler);

    printf("Hit CTRL-C to exit\n");
    while (x != 1);
}
```

Xem thêm

`raise()`, `abort()`

15.2 `raise()`

Làm cho một signal được raise lên

Synopsis

```
#include <signal.h>

int raise(int sig);
```

Mô tả

Làm cho signal handler cho signal `sig` được gọi. Nếu handler là `SIG_DFL` hoặc `SIG_IGN`, thì hành động mặc định hoặc không gì cả sẽ xảy ra.

`raise()` trả về sau khi signal handler đã chạy xong.

Thú vị là, nếu bạn gây ra signal bằng `raise()`, bạn có thể gọi các hàm thư viện từ trong signal handler mà không gây undefined behavior. Tuy nhiên tôi không chắc sự thật đó hữu dụng thực tế ở chỗ nào.

Giá trị trả về

Trả về `0` khi thành công. Khác `0` nếu không.

Ví dụ

Chương trình này đặt signal handler, rồi raise signal. Signal handler được kích hoạt.

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

void handler(int sig)
{
    // Undefined behavior khi gọi printf() nếu handler này không
    // phải do raise(), tức là nếu bạn bấm CTRL-C.

    printf("Got signal %d!\n", sig);

    // Thường reset handler phòng khi implementation set nó về`
    // SIG_DFL khi signal xảy ra.

    signal(sig, handler);
}

int main(void)
{
    signal(SIGINT, handler);

    raise(SIGINT);
    raise(SIGINT);
    raise(SIGINT);
}
```

Output:

```
Got signal 2!
Got signal 2!
Got signal 2!
```

Xem thêm

`signal()`

Chapter 16

`<stdalign.h>` Macro Cho Alignment

Nếu bạn đang code thứ gì đó low-level như một memory allocator giao tiếp với OS, bạn có thể cần header này. Nhưng hầu hết dev C trải cả sự nghiệp mà chưa bao giờ dùng nó.

Alignment[1] là về bội số của địa chỉ mà object có thể được lưu. Object có thể lưu ở bất kỳ địa chỉ nào? Hay phải là địa chỉ chia hết cho 2? Hay 8? Hay 16?

Tên	Mô tả
`alignas()`	Chỉ định alignment, expand thành `_Alignas`
`alignof()`	Lấy alignment, expand thành `_Alignof`

Hai macro bổ sung sau được định nghĩa bằng `1`:

```
__alignas_is_defined
__alignof_is_defined
```

Ghi chú nhanh: các alignment lớn hơn của `max_align_t` được gọi là *overalignment* và là implementation-defined.

16.1 `alignas()` `_Alignas()`

Bắt biến phải có một alignment nhất định

Synopsis

```
#include <stdalign.h>

alignas(type-name)
alignas(constant-expression)
```

```
_Alignas(type-name)
_Alignas(constant-expression)
```

Mô tả

Dùng *alignment specifier* này để bắt buộc alignment cho biến cụ thể. Ví dụ, ta có thể khai báo `c` là `char`, nhưng align giống như nó là `int`:

[1] https://en.wikipedia.org/wiki/Data_structure_alignment

```
char alignas(int) c;
```

Bạn cũng có thể đặt biểu thức integer hằng vào đó. Compiler có lẽ sẽ áp limit lên giá trị mà biểu thức này có thể nhận. Lũy thừa nhỏ của 2 (1, 2, 4, 8, và 16) thường là đặt cược an toàn.

```
char alignas(8) c;   // align theo ranh giới 8-byte
```

Cho tiện, bạn cũng có thể chỉ định `0` nếu muốn alignment mặc định (như thể bạn không nói `alignas()` gì cả):

```
char alignas(0) c;   // dùng alignment mặc định cho kiểu này
```

Ví dụ

```
#include <stdalign.h>
#include <stdio.h>     // for printf()
#include <stddef.h>    // for max_align_t

int main(void)
{
    int i, j;
    char alignas(max_align_t) a, b;
    char alignas(int) c, d;
    char e, f;

    printf("i: %p\n", (void *)&i);
    printf("j: %p\n\n", (void *)&j);
    printf("a: %p\n", (void *)&a);
    printf("b: %p\n\n", (void *)&b);
    printf("c: %p\n", (void *)&c);
    printf("d: %p\n\n", (void *)&d);
    printf("e: %p\n", (void *)&e);
    printf("f: %p\n", (void *)&f);
}
```

Output trên hệ của tôi bên dưới. Chú ý sự khác biệt giữa các cặp giá trị.

- `i` và `j`, đều là `int`, align theo ranh giới 4-byte.
- `a` và `b` bị bắt về ranh giới của kiểu `max_align_t`, tức mỗi 16 byte trên hệ của tôi.
- `c` và `d` bị bắt về cùng alignment với `int`, tức 4 byte, giống như `i` và `j`.
- `e` và `f` không có alignment chỉ định, nên chúng được lưu với alignment mặc định là 1 byte.

```
i: 0x7ffee7dfb4cc    <-- difference of 4 bytes
j: 0x7ffee7dfb4c8

a: 0x7ffee7dfb4c0    <-- difference of 16 bytes
b: 0x7ffee7dfb4b0

c: 0x7ffee7dfb4ac    <-- difference of 4 bytes
d: 0x7ffee7dfb4a8

e: 0x7ffee7dfb4a7    <-- difference of 1 byte
f: 0x7ffee7dfb4a6
```

Xem thêm

`alignof` , `max_align_t` , `memalignment()`

16.2 `alignof()` `_Alignof()`

Lấy alignment của một kiểu

Synopsis

```
#include <stdalign.h>

alignof(type-name)
```

```
_Alignof(type-name)
```

Mô tả

Biểu thức này eval thành một giá trị kiểu `size_t` cho biết alignment của một kiểu cụ thể trên hệ của bạn.

Giá trị trả về

Trả về giá trị alignment, tức là địa chỉ bắt đầu của object kiểu đã cho phải ở ranh giới chia hết cho số này.

Ví dụ

In ra alignment của đủ loại kiểu khác nhau.

```
#include <stdalign.h>
#include <stdio.h>     // for printf()
#include <stddef.h>    // for max_align_t

struct t {
    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
```

Xem thêm

`alignas`, `max_align_t`, `memalignment()`

Chapter 17

`<stdarg.h>` Tham Số Biến Đổi

Macro	Mô tả
`va_arg()`	Lấy tham số biến đổi kế tiếp
`va_copy()`	Copy một `va_list` và công việc đã làm tới thời điểm đó
`va_end()`	Báo hiệu ta đã xử lý xong tham số biến đổi
`va_start()`	Khởi tạo một `va_list` để bắt đầu xử lý tham số biến đổi

Header file này là cái cho phép bạn viết hàm nhận số lượng tham số biến đổi.

Ngoài các macro, bạn còn có một kiểu mới giúp C theo dõi vị trí đang xử lý đến đâu trong quá trình xử lý tham số biến đổi: `va_list`. Kiểu này là opaque, và bạn sẽ truyền nó quanh các macro khác nhau để giúp lấy tham số.

Chú ý mỗi hàm variadic cần ít nhất một tham số non-variable. Bạn cần cái này để khởi động xử lý với `va_start()`.

17.1 `va_arg()`

Lấy tham số biến đổi kế tiếp

Synopsis

```
#include <stdarg.h>

type va_arg(va_list ap, type);
```

Mô tả

Nếu bạn có variable argument list đã được khởi tạo bằng `va_start()`, truyền nó vào đây cùng với kiểu của tham số bạn đang muốn lấy, ví dụ:

```
int   x = va_arg(args, int);
float y = va_arg(args, float);
```

Giá trị trả về

Eval thành giá trị và kiểu của tham số biến đổi kế tiếp.

Ví dụ

Đây là demo cộng số lượng integer tuỳ ý lại với nhau. Tham số đầu tiên là số lượng integer cần cộng. Ta sẽ dùng cái đó 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);   // Bắt đầu với tham số sau "count"

    for (int i = 0; i < count; i++) {
        int n = va_arg(va, int);   // Lấy int kế tiếp

        total += n;
    }

    va_end(va);  // Xong hết

    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
}
```

Xem thêm

`va_start()`, `va_end()`

17.2 `va_copy()`

Copy một `va_list` và công việc đã làm tới thời điểm đó

Synopsis

```
#include <stdarg.h>

void va_copy(va_list dest, va_list src);
```

Mô tả

Mục đích chính của hàm này là lưu trạng thái giữa chừng trong quá trình xử lý tham số biến đổi để bạn có thể scan tới rồi rewind lại chỗ đã lưu.

Bạn truyền vào `src` `va_list` và nó copy sang `dest`.

Nếu đã gọi hàm này một lần cho một `dest` cụ thể, bạn không được gọi lại (hay gọi `va_start()`) với cùng `dest` đó trừ khi bạn gọi `va_end()` cho cái `dest` đó trước.

```
va_copy(dest, src);
va_copy(dest, src2);  // SAI!

va_copy(dest, src);
va_start(dest, var);  // SAI!

va_copy(dest, src);
va_end(dest);
va_copy(dest, src2);  // OK!

va_copy(dest, src);
va_end(dest);
va_start(dest, var);  // OK!
```

Giá trị trả về

Không trả về gì.

Ví dụ

Đây là ví dụ cộng tất cả tham số biến đổi lại, rồi quay lại cộng thêm tất cả các số từ thứ ba trở đi nữa, ví dụ nếu tham số là:

```
10 20 30 40
```

Đầu tiên ta cộng tất cả được `100`, rồi cộng thêm mọi thứ từ số thứ ba, tức cộng `30+40`, tổng ra `170`.

Ta sẽ làm vậy bằng cách lưu vị trí trong quá trình xử lý tham số biến đổi với `va_copy`, rồi dùng nó sau để xử lý lại phần tham số đuôi.

(Và đúng rồi, tôi biết có cách toán học làm chuyện này mà không cần rewind, nhưng tôi đang khó lòng nghĩ ra ví dụ hay!)

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

// Cộng tất cả các sô´lại, rôi cộng lại lần nữa tất cả các sô´
// từ sau sô´thứ hai.
int contrived_adder(int count, ...)
{
    if (count < 3) return 0; // OK, tôi đang lười. Bắt được rôi đó.

    int total = 0;

    va_list args, mid_args;

    va_start(args, count);

    for (int i = 0; i < count; i++) {

        // Nêú ta đang ở sô´thứ hai, lưu chô˜vào
        // mid_args:

        if (i == 2)
            va_copy(mid_args, args);

        total += va_arg(args, int);
```

```
    }

    va_end(args); // Xong cái này

    // Nhưng giờ bắt đầu với mid_args và cộng hết những cái đó:
    for (int i = 0; i < count - 2; i++)
        total += va_arg(mid_args, int);

    va_end(mid_args); // Xong cái này luôn

    return total;
}

int main(void)
{
    // 10+20+30 + 30 == 90
    printf("%d\n", contrived_adder(3, 10, 20, 30));

    // 10+20+30+40+50 + 30+40+50 == 270
    printf("%d\n", contrived_adder(5, 10, 20, 30, 40, 50));
}
```

Xem thêm

`va_start()`, `va_arg()`, `va_end()`

17.3 `va_end()`

Báo hiệu ta đã xử lý xong tham số biến đổi

Synopsis

```
#include <stdarg.h>

void va_end(va_list ap);
```

Mô tả

Sau khi bạn đã `va_start()` hoặc `va_copy` một `va_list` mới, bạn **phải** gọi `va_end()` với nó trước khi nó ra khỏi scope.

Bạn cũng phải làm vậy nếu định gọi `va_start()` hoặc `va_copy()` *lại* trên biến mà bạn đã làm vậy rồi.

Đó là luật nếu bạn muốn tránh undefined behavior.

Nhưng cứ nghĩ nó như cleanup thôi. Bạn gọi `va_start()`, nên bạn sẽ gọi `va_end()` khi xong.

Giá trị trả về

Không trả về gì.

Ví dụ

Đây là demo cộng số lượng integer tuỳ ý lại với nhau. Tham số đầu tiên là số lượng integer cần cộng. Ta sẽ dùng cái đó 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);   // Bắt đầu với tham số sau "count"

    for (int i = 0; i < count; i++) {
        int n = va_arg(va, int);   // Lấy int kế tiếp

        total += n;
    }

    va_end(va);  // Xong hết

    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
}
```

Xem thêm

`va_start()`, `va_copy()`

17.4 `va_start()`

Khởi tạo một `va_list` để bắt đầu xử lý tham số biến đổi

Synopsis

```
#include <stdarg.h>

void va_start(va_list ap, parmN);
```

Mô tả

Bạn đã khai báo một biến kiểu `va_list` để theo dõi quá trình xử lý tham số biến đổi... giờ làm sao khởi tạo nó để có thể bắt đầu gọi `va_arg()` lấy các tham số đó?

`va_start()` ra tay cứu nguy!

Việc bạn làm là truyền `va_list` của bạn vào, ở đây thể hiện qua tham số `ap`. Chỉ truyền list, không truyền pointer tới nó.

Rồi cho tham số thứ hai của `va_start()`, bạn đưa tên tham số mà bạn muốn bắt đầu xử lý tham số *sau* nó. Cái này phải là tham số ngay trước `...` trong danh sách tham số.

Nếu bạn đã gọi `va_start()` trên một `va_list` cụ thể và muốn gọi `va_start()` lại trên nó, bạn **phải** gọi `va_end()` trước!

Giá trị trả về

Không trả về gì!

Ví dụ

Đây là demo cộng số lượng integer tuỳ ý lại với nhau. Tham số đầu tiên là số lượng integer cần cộng. Ta sẽ dùng cái đó 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);   // Bắt đầu với tham số sau "count"

    for (int i = 0; i < count; i++) {
        int n = va_arg(va, int);   // Lấy int kế tiếp

        total += n;
    }

    va_end(va);  // Xong hết

    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
}
```

Xem thêm

`va_arg()`, `va_end()`

Chapter 18

`<stdatomic.h>` Các Hàm Liên Quan Đến Atomic

Hàm	Mô tả
`atomic_compare_exchange_strong_explicit()`	Compare-and-exchange nguyên tử, strong, explicit
`atomic_compare_exchange_strong()`	Compare-and-exchange nguyên tử, strong
`atomic_compare_exchange_weak_explicit()`	Compare-and-exchange nguyên tử, weak, explicit
`atomic_compare_exchange_weak()`	Compare-and-exchange nguyên tử, weak
`atomic_exchange_explicit()`	Thay giá trị trong một object nguyên tử, explicit
`atomic_exchange()`	Thay giá trị trong một object nguyên tử
`atomic_fetch_add_explicit()`	Cộng nguyên tử vào một số nguyên atomic, explicit
`atomic_fetch_add()`	Cộng nguyên tử vào một số nguyên atomic
`atomic_fetch_and_explicit()`	AND bit nguyên tử một số nguyên atomic, explicit
`atomic_fetch_and()`	AND bit nguyên tử một số nguyên atomic
`atomic_fetch_or_explicit()`	OR bit nguyên tử một số nguyên atomic, explicit
`atomic_fetch_or()`	OR bit nguyên tử một số nguyên atomic
`atomic_fetch_sub_explicit()`	Trừ nguyên tử khỏi một số nguyên atomic, explicit
`atomic_fetch_sub()`	Trừ nguyên tử khỏi một số nguyên atomic
`atomic_fetch_xor_explicit()`	XOR bit nguyên tử một số nguyên atomic, explicit
`atomic_fetch_xor()`	XOR bit nguyên tử một số nguyên atomic
`atomic_flag_clear_explicit()`	Xoá một atomic flag, explicit
`atomic_flag_clear()`	Xoá một atomic flag
`atomic_flag_test_and_set_explicit()`	Test-and-set một atomic flag, explicit
`atomic_flag_test_and_set()`	Test-and-set một atomic flag
`atomic_init()`	Khởi tạo một biến atomic
`atomic_is_lock_free()`	Xác định xem một kiểu atomic có lock-free không
`atomic_load_explicit()`	Trả về giá trị từ một biến atomic, explicit
`atomic_load()`	Trả về giá trị từ một biến atomic
`atomic_signal_fence()`	Fence cho signal handler trong cùng thread
`atomic_store_explicit()`	Lưu một giá trị vào biến atomic, explicit
`atomic_store()`	Lưu một giá trị vào biến atomic
`atomic_thread_fence()`	Dựng một fence
`ATOMIC_VAR_INIT()`	Tạo initializer cho một biến atomic

Hàm	Mô tả
`kill_dependency()`	Kết thúc một chuỗi dependency

Trên các hệ điều hành kiểu Unix, bạn có thể cần thêm `-latomic` vào dòng lệnh biên dịch.

18.1 Các Kiểu Atomic

Header này định nghĩa sẵn một đống kiểu:

Kiểu atomic	Dạng đầy đủ 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`
`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`

Bạn có thể tự làm thêm kiểu của mình bằng type qualifier `_Atomic`:

```
_Atomic double x;
```

hoặc type specifier `_Atomic()`:

```
_Atomic(double) x;
```

18.2 Các Macro Lock-free

Các macro này cho bạn biết một kiểu có lock-free hay không. Có thể.

Chúng có thể dùng ở compile time với `#if`. Chúng áp dụng cho cả kiểu signed và unsigned.

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`

Thú vị là các macro này có thể có tới *ba* giá trị khác nhau:

Giá trị	Ý nghĩa
`0`	Không bao giờ lock-free.
`1`	*Đôi khi* lock-free[1].
`2`	Luôn lock-free.

18.3 Atomic Flag

Kiểu mờ (opaque type) `atomic_flag` là thứ duy nhất được đảm bảo là lock-free. Dù implementation trên PC của bạn chắc sẽ làm được nhiều hơn thế.

Nó được truy cập qua các hàm `atomic_flag_test_and_set()` và `atomic_flag_clear()`.

Trước khi dùng, có thể khởi tạo nó về trạng thái clear bằng:

```
atomic_flag f = ATOMIC_FLAG_INIT;
```

18.4 Memory Order (Thứ tự bộ nhớ)

Header này giới thiệu một kiểu `enum` mới tên là `memory_order`. Nó được dùng bởi một đống hàm để chỉ định các memory order (thứ tự bộ nhớ) khác với sequential consistency (tính nhất quán tuần tự).

[1] Có thể nó phụ thuộc vào môi trường run-time và không thể biết ở compile-time.

`memory_order`	Mô tả
`memory_order_seq_cst`	Sequential Consistency
`memory_order_acq_rel`	Acquire/Release
`memory_order_release`	Release
`memory_order_acquire`	Acquire
`memory_order_consume`	Consume
`memory_order_relaxed`	Relaxed

Bạn có thể truyền mấy thứ này vào các hàm atomic có hậu tố `_explicit`.

Các phiên bản không có `_explicit` hoạt động y như khi bạn gọi phiên bản `_explicit` tương ứng với `memory_order_seq_cst`.

18.5 `ATOMIC_VAR_INIT()`

Tạo một initializer cho biến atomic

Synopsis

```
#include <stdatomic.h>

#define ATOMIC_VAR_INIT(C value)   // Deprecated
```

Mô tả

Macro này mở rộng thành một initializer, nên bạn có thể dùng nó khi định nghĩa biến.

Kiểu của `value` phải là kiểu cơ sở của biến atomic.

Buồn cười là, bản thân việc khởi tạo *không* phải thao tác nguyên tử (atomic).

CPPReference nói rằng cái này đã bị deprecated[2] và nhiều khả năng sẽ bị bỏ. Tài liệu tiêu chuẩn p1138r0[3] giải thích thêm rằng macro này bị hạn chế ở chỗ không thể khởi tạo đúng các atomic `struct`, và lý do tồn tại ban đầu của nó hoá ra chẳng hữu dụng.

Cứ khởi tạo biến trực tiếp đi là xong.

Giá trị trả về

Mở rộng thành initializer phù hợp cho biến atomic này.

Ví dụ

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

int main(void)
{
    atomic_int x = ATOMIC_VAR_INIT(3490);  // Deprecated
    printf("%d\n", x);
}
```

[2] https://en.cppreference.com/w/cpp/atomic/ATOMIC_VAR_INIT
[3] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1138r0.pdf

Xem thêm

`atomic_init()`

18.6 `atomic_init()`

Khởi tạo một biến atomic

Synopsis

```
#include <stdatomic.h>

void atomic_init(volatile A *obj, C value);
```

Mô tả

Bạn có thể dùng nó để khởi tạo một biến atomic.

Kiểu của `value` phải là kiểu cơ sở của biến atomic.

Buồn cười là, bản thân việc khởi tạo *không* phải thao tác nguyên tử.

Theo như tôi thấy, chẳng có gì khác biệt giữa cái này và việc gán trực tiếp cho biến atomic. Spec nói nó có mặt để cho phép compiler chèn thêm bất kỳ việc khởi tạo bổ sung nào cần làm, nhưng mọi thứ vẫn ổn nếu không có nó. Nếu ai đó có thêm thông tin, gửi cho tôi nhé.

Giá trị trả về

Không trả về gì cả!

Ví dụ

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

int main(void)
{
    atomic_int x;

    atomic_init(&x, 3490);

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

Xem thêm

`ATOMIC_VAR_INIT()`, `atomic_store()`, `atomic_store_explicit()`

18.7 `kill_dependency()`

Kết thúc một chuỗi dependency

Synopsis

```
#include <stdatomic.h>

type kill_dependency(type y);
```

Mô tả

Cái này có khả năng hữu ích để tối ưu nếu bạn đang dùng `memory_order_consume` ở đâu đó.

Và nếu bạn biết mình đang làm gì. Nếu không chắc, tìm hiểu thêm trước khi thử dùng.

Giá trị trả về

Trả về giá trị được truyền vào.

Ví dụ

Trong ví dụ này, `i` mang theo một dependency vào `x`. Và cũng sẽ mang vào `y`, nhưng vì có lời gọi `kill_dependency()` nên không.

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

int main(void)
{
    atomic_int a;
    int i = 10, x, y;

    atomic_store_explicit(&a, 3490, memory_order_release);

    i = atomic_load_explicit(&a, memory_order_consume);
    x = i;
    y = kill_dependency(i);

    printf("%d %d\n", x, y);  // 3490 and either 3490 or 10
}
```

18.8 `atomic_thread_fence()`

Dựng một fence (hàng rào)

Synopsis

```
#include <stdatomic.h>

void atomic_thread_fence(memory_order order);
```

Mô tả

Hàm này dựng một memory fence (rào cản bộ nhớ) với `order` chỉ định.

`order`	Mô tả
`memory_order_seq_cst`	Fence acquire/release theo sequential consistency
`memory_order_acq_rel`	Fence acquire/release
`memory_order_release`	Fence release
`memory_order_acquire`	Fence acquire
`memory_order_consume`	Fence acquire (again)
`memory_order_relaxed`	Không có fence gì cả—gọi với cái này chẳng có ý nghĩa gì

Bạn có thể cố tránh dùng mấy cái này và cứ bám vào các chế độ khác nhau với `atomic_store_explicit()` và `atomic_load_explicit()`. Hoặc không.

Giá trị trả về

Không trả về gì cả!

Ví dụ

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

atomic_int shared_1 = 1;
atomic_int shared_2 = 2;

int thread_1(void *arg)
{
    (void)arg;

    atomic_store_explicit(&shared_1, 10, memory_order_relaxed);

    atomic_thread_fence(memory_order_release);

    atomic_store_explicit(&shared_2, 20, memory_order_relaxed);

    return 0;
}

int thread_2(void *arg)
{
    (void)arg;

    // If this fence runs after the release fence, we're
    // guaranteed to see thread_1's changes to the shared
    // varaibles.

    atomic_thread_fence(memory_order_acquire);

    if (shared_2 == 20) {
        printf("Shared_1 better be 10 and it's %d\n", shared_1);
    } else {
        printf("Anything's possible: %d %d\n", shared_1, shared_2);
    }

    return 0;
```

```
}

int main(void)
{
    thrd_t t1, t2;

    thrd_create(&t2, thread_2, NULL);
    thrd_create(&t1, thread_1, NULL);

    thrd_join(t1, NULL);
    thrd_join(t2, NULL);
}
```

Xem thêm

`atomic_store_explicit()`, `atomic_load_explicit()`, `atomic_signal_fence()`

18.9 `atomic_signal_fence()`

Fence cho signal handler trong cùng thread

Synopsis

```
#include <stdatomic.h>

void atomic_signal_fence(memory_order order);
```

Mô tả

Hàm này hoạt động giống `atomic_thread_fence()` nhưng mục đích là trong phạm vi một thread duy nhất; đáng chú ý là để dùng trong signal handler của thread đó.

Vì signal có thể xảy ra bất cứ lúc nào, ta có thể cần một cách để chắc chắn rằng mọi write của thread xảy ra trước signal handler sẽ nhìn thấy được bên trong signal handler đó.

Giá trị trả về

Không trả về gì cả!

Ví dụ

Demo một phần. (Lưu ý rằng về mặt kỹ thuật thì việc gọi `printf()` trong signal handler là undefined behavior.)

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

int global;

void handler(int sig)
{
    (void)sig;
```

```
    // If this runs before the release, the handler will
    // potentially see global == 0.
    //
    // Otherwise, it will definitely see global == 10.

    atomic_signal_fence(memory_order_acquire);

    printf("%d\n", global);
}

int main(void)
{
    signal(SIGINT, handler);

    global = 10;

    atomic_signal_fence(memory_order_release);

    // If the signal handler runs after the release
    // it will definitely see the value 10 in global.
}
```

Xem thêm

`atomic_thread_fence()`, `signal()`

18.10 `atomic_is_lock_free()`

Xác định xem một kiểu atomic có lock-free không

Synopsis

```
#include <stdatomic.h>

_Bool atomic_is_lock_free(const volatile A *obj);
```

Mô tả

Xác định xem biến `obj` kiểu `A` có lock-free không. Dùng được với bất kỳ kiểu nào.

Khác với các macro lock-free có thể dùng ở compile-time, đây thuần tuý là hàm run-time. Vì vậy ở những chỗ mà macro trả lời "có thể", hàm này sẽ chắc chắn cho bạn biết biến atomic có lock-free hay không.

Cái này hữu ích khi bạn tự định nghĩa biến atomic của mình và muốn biết trạng thái lock-free của chúng.

Giá trị trả về

True nếu biến lock-free, false nếu không.

Ví dụ

Kiểm tra xem một cặp `struct` và một `double` atomic có lock-free không. Trên hệ thống của tôi, `struct` lớn hơn thì to quá không lock-free được, nhưng hai cái còn lại thì OK.

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

int main(void)
{
    struct foo {
        int x, y;
    };

    struct bar {
        int x, y, z;
    };

    _Atomic(double) a;
    struct foo b;
    struct bar c;

    printf("a is lock-free: %d\n", atomic_is_lock_free(&a));
    printf("b is lock-free: %d\n", atomic_is_lock_free(&b));
    printf("c is lock-free: %d\n", atomic_is_lock_free(&c));
}
```

Output trên hệ thống của tôi (YMMV):

```
a is lock-free: 1
b is lock-free: 1
c is lock-free: 0
```

Xem thêm

Các Macro Lock-free

18.11 `atomic_store()`

Lưu một giá trị vào biến atomic

Synopsis

```
#include <stdatomic.h>

void atomic_store(volatile A *object, C desired);

void atomic_store_explicit(volatile A *object,
                           C desired, memory_order order);
```

Mô tả

Lưu một giá trị vào biến atomic, có thể được đồng bộ.

Cái này giống như một phép gán thông thường, nhưng linh hoạt hơn.

Mấy cái sau có cùng hiệu ứng lưu trữ với một `atomic_int x`:

```
x = 10;
atomic_store(&x, 10);
atomic_store_explicit(&x, 10, memory_order_seq_cst);
```

Nhưng hàm cuối, `atomic_store_explicit()`, cho bạn chỉ định memory order.

Vì đây là thao tác kiểu "release-y", không có memory order kiểu "acquire-y" nào hợp lệ. `order` chỉ có thể là `memory_order_seq_cst`, `memory_order_release`, hoặc `memory_order_relaxed`.

`order` không thể là `memory_order_acq_rel`, `memory_order_acquire`, hoặc `memory_order_consume`.

Giá trị trả về

Không trả về gì cả!

Ví dụ

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

int main(void)
{
    atomic_int x = 0;
    atomic_int y = 0;

    atomic_store(&x, 10);

    atomic_store_explicit(&y, 20, memory_order_relaxed);

    // Will print either "10 20" or "10 0":
    printf("%d %d\n", x, y);
}
```

Xem thêm

`atomic_init()`, `atomic_load()`, `atomic_load_explicit()`, `atomic_exchange()`, `atomic_exchange_explicit()`, `atomic_compare_exchange_strong()`, `atomic_compare_exchange_strong_explicit()`, `atomic_compare_exchange_weak()`, `atomic_compare_exchange_weak_explicit()`, `atomic_fetch_*()`

18.12 `atomic_load()`

Trả về giá trị từ một biến atomic

Synopsis

```
#include <stdatomic.h>

C atomic_load(const volatile A *object);

C atomic_load_explicit(const volatile A *object, memory_order order);
```

Mô tả

Với một con trỏ tới `object` kiểu `A`, nguyên tử trả về giá trị `C` của nó. Đây là hàm generic có thể dùng với bất kỳ kiểu nào.

Hàm `atomic_load_explicit()` cho bạn chỉ định memory order.

Vì đây là thao tác kiểu "acquire-y", không có memory order kiểu "release-y" nào hợp lệ. `order` chỉ có thể là `memory_order_seq_cst`, `memory_order_acquire`, `memory_order_consume`, hoặc `memory_order_relaxed`.

`order` không thể là `memory_order_acq_rel` hoặc `memory_order_release`.

Giá trị trả về

Trả về giá trị được lưu trong `object`.

Ví dụ

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

int main(void)
{
    atomic_int x = 10;

    int v = atomic_load(&x);

    printf("%d\n", v);  // 10
}
```

Xem thêm

`atomic_store()`, `atomic_store_explicit()`

18.13 `atomic_exchange()`

Thay giá trị trong một object nguyên tử

Synopsis

```
#include <stdatomic.h>

C atomic_exchange(volatile A *object, C desired);

C atomic_exchange_explicit(volatile A *object, C desired,
                           memory_order order);
```

Mô tả

Đặt giá trị trong `object` thành `desired`.

`object` có kiểu `A`, một kiểu atomic nào đó.

`desired` có kiểu `C`, kiểu non-atomic tương ứng với `A`.

Cái này rất giống `atomic_store()`, trừ việc giá trị trước đó được trả về một cách nguyên tử.

Giá trị trả về

Trả về giá trị trước đó của `object`.

Ví dụ

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

int main(void)
{
    atomic_int x = 10;

    int previous = atomic_exchange(&x, 20);

    printf("x is  %d\n", x);
    printf("x was %d\n", previous);
}
```

Output:

```
x is  20
x was 10
```

Xem thêm

`atomic_init()`, `atomic_load()`, `atomic_load_explicit()`, `atomic_store()`, `atomic_store_explicit()` `atomic_compare_exchange_strong()`, `atomic_compare_exchange_strong_explicit()`, `atomic_compare_exchange_weak()`, `atomic_compare_exchange_weak_explicit()`

18.14 `atomic_compare_exchange_*()`

Compare-and-exchange nguyên tử

Synopsis

```
#include <stdatomic.h>

_Bool atomic_compare_exchange_strong(volatile A *object,
                                     C *expected, C desired);

_Bool atomic_compare_exchange_strong_explicit(volatile A *object,
                                              C *expected, C desired,
                                              memory_order success,
                                              memory_order failure);

_Bool atomic_compare_exchange_weak(volatile A *object,
                                   C *expected, C desired);

_Bool atomic_compare_exchange_weak_explicit(volatile A *object,
                                            C *expected, C desired,
                                            memory_order success,
                                            memory_order failure);
```

Mô tả

Nền tảng lâu đời cho vô số thứ lock-free: compare-and-exchange (CAS).

Trong các prototype ở trên, `A` là kiểu của object atomic, và `C` là kiểu cơ sở tương đương.

Bỏ qua các phiên bản `_explicit` một lúc, các hàm này làm:

- Nếu giá trị được trỏ tới bởi `object` bằng giá trị được trỏ tới bởi `expected`, thì giá trị được trỏ tới bởi `object` được đặt thành `desired`. Và hàm trả về `true` cho biết trao đổi đã diễn ra.
- Ngược lại, giá trị được trỏ tới bởi `expected` (vâng, `expected`) được đặt thành `desired` và hàm trả về `false` cho biết trao đổi không diễn ra.

Pseudocode cho trao đổi sẽ trông như thế này[4]:

```
bool compare_exchange(atomic_A *object, C *expected, C desired)
{
    if (*object is the same as *expected) {
        *object = desired
        return true
    }

    *expected = desired
    return false
}
```

Các biến thể `_weak` có thể thất bại một cách tự phát, nên ngay cả khi `*object == *desired`, nó có thể không đổi giá trị và sẽ trả về `false`. Vì thế bạn sẽ muốn đặt nó trong vòng lặp nếu dùng[5].

Các biến thể `_explicit` có hai memory order: `success` nếu `*object` được đặt thành `desired`, và `failure` nếu không.

Đây là các hàm test-and-set, nên bạn có thể dùng `memory_order_acq_rel` với các biến thể `_explicit`.

Giá trị trả về

Trả về `true` nếu `*object` là `*expected`. Ngược lại, `false`.

Ví dụ

Một ví dụ gượng ép, nơi nhiều thread cộng `2` vào một giá trị chia sẻ theo cách lock-free.

(Ngoài đời thực thì tốt hơn là dùng `+= 2` để làm việc này, trừ khi bạn đang dùng phép thuật `_explicit` nào đó.)

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

#define LOOP_COUNT 10000

atomic_int value;

int run(void *arg)
```

[4]Hiệu quả thì cái này làm cùng việc, nhưng rõ ràng nó không nguyên tử.

[5]Spec nói: "Cái thất bại tự phát này cho phép cài đặt compare-and-exchange trên một lớp máy rộng hơn, ví dụ như các máy load-locked store-conditional." Và thêm: "Khi compare-and-exchange nằm trong vòng lặp, phiên bản weak sẽ cho hiệu năng tốt hơn trên một số nền tảng. Khi một compare-and-exchange weak sẽ cần vòng lặp còn strong thì không, thì strong được ưu tiên hơn."

```
{
    (void)arg;

    for(int i = 0; i < LOOP_COUNT; i++) {

        int cur = value;
        int next;

        do {
            next = cur + 2;
        } while (!atomic_compare_exchange_strong(&value, &cur, next));
    }

    return 0;
}

int main(void)
{
    thrd_t t1, t2;

    thrd_create(&t1, run, NULL);
    thrd_create(&t2, run, NULL);

    thrd_join(t1, NULL);
    thrd_join(t2, NULL);

    printf("%d should equal %d\n", value, LOOP_COUNT * 4);
}
```

Chỉ cần thay cái này bằng `value = value + 2` thôi là gây ra chuyện đạp dữ liệu.

Xem thêm

`atomic_load()`, `atomic_load_explicit()`, `atomic_store()`, `atomic_store_explicit()`, `atomic_exchange()`, `atomic_exchange_explicit()`, `atomic_fetch_*()`

18.15 `atomic_fetch_*()`

Sửa đổi biến atomic một cách nguyên tử

Synopsis

```
#include <stdatomic.h>

C atomic_fetch_KEY(volatile A *object, M operand);

C atomic_fetch_KEY_explicit(volatile A *object, M operand,
                            memory_order order);
```

Mô tả

Thật ra đây là một nhóm gồm 10 hàm. Bạn thay `KEY` bằng một trong các từ dưới để thực hiện thao tác đó:

- `add`

- `sub`
- `or`
- `xor`
- `and`

Vậy các hàm này có thể cộng hoặc trừ giá trị vào/khỏi một biến atomic, hoặc có thể thực hiện OR, XOR, hoặc AND bit trên chúng.

Dùng với các kiểu integer hoặc pointer. Dù spec có hơi mơ hồ về vấn đề này, các kiểu khác sẽ làm C không vui. Nó còn cố tránh undefined behavior với signed integer:

C18 §7.17.7.5 ¶3:

> Đối với các kiểu signed integer, số học được định nghĩa dùng biểu diễn bù hai với quấn vòng im lặng khi tràn; không có kết quả undefined nào.

Trong synopsis ở trên, `A` là kiểu atomic, và `M` là kiểu non-atomic tương ứng với `A` (hoặc `ptrdiff_t` cho pointer atomic), và `C` là kiểu non-atomic tương ứng với `A`.

Ví dụ, đây là một số thao tác trên một `atomic_int`.

```
atomic_fetch_add(&x, 20);
atomic_fetch_sub(&x, 37);
atomic_fetch_xor(&x, 3490);
```

Chúng tương đương `+=`, `-=`, `|=`, `^=` và `&=`, trừ việc giá trị trả về là giá trị *trước đó* của object atomic. (Với các toán tử gán, giá trị của biểu thức là giá trị *sau* khi đánh giá.)

```
atomic_int x = 10;
int prev = atomic_fetch_add(&x, 20);
printf("%d %d\n", prev, x);  // 10 30
```

so với:

```
atomic_int x = 10;
int prev = (x += 20);
printf("%d %d\n", prev, x);  // 30 30
```

Và, tất nhiên, phiên bản `_explicit` cho phép bạn chỉ định memory order còn tất cả các toán tử gán đều là `memory_order_seq_cst`.

Giá trị trả về

Trả về giá trị trước đó của object atomic trước khi sửa đổi.

Ví dụ

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

int main(void)
{
    atomic_int x = 0;
    int prev;

    atomic_fetch_add(&x, 3490);
    atomic_fetch_sub(&x, 12);
    atomic_fetch_xor(&x, 444);
```

```
    atomic_fetch_or(&x, 12);
    prev = atomic_fetch_and(&x, 42);

    printf("%d %d\n", prev, x);   // 3118 42
}
```

Xem thêm

`atomic_exchange()`, `atomic_exchange_explicit()`, `atomic_compare_exchange_strong()`, `atomic_compare_exchange_strong_explicit()`, `atomic_compare_exchange_weak()`, `atomic_compare_exchange_weak_explicit()`

18.16 `atomic_flag_test_and_set()`

Test-and-set một atomic flag

Synopsis

```
#include <stdatomic.h>

_Bool atomic_flag_test_and_set(volatile atomic_flag *object);

_Bool atomic_flag_test_and_set_explicit(volatile atomic_flag *object,
                                        memory_order order);
```

Mô tả

Một trong các hàm lâu đời đáng kính của lập trình lock-free, hàm này set atomic flag được chỉ định trong `object`, và trả về giá trị trước đó của flag.

Như thường lệ, `_explicit` cho phép bạn chỉ định memory order khác.

Giá trị trả về

Trả về `true` nếu flag đã được set trước đó, và `false` nếu chưa.

Ví dụ

Dùng test-and-set để cài đặt một spin lock[6]:

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

// Shared non-atomic struct
struct {
    int x, y, z;
} s = {1, 2, 3};

atomic_flag f = ATOMIC_FLAG_INIT;
```

[6]Đừng dùng cái này trừ khi bạn biết mình đang làm gì—dùng chức năng mutex của thread thay vào đó. Nó sẽ cho phép thread bị chặn ngủ và ngừng gặm CPU.

```
int run(void *arg)
{
    int tid = *(int*)arg;

    printf("Thread %d: waiting for lock...\n", tid);

    while (atomic_flag_test_and_set(&f));

    printf("Thread %d: got lock, s is {%d, %d, %d}\n", tid,
                                                       s.x, s.y, s.z);
    s.x = (tid + 1) * 5 + 0;
    s.y = (tid + 1) * 5 + 1;
    s.z = (tid + 1) * 5 + 2;
    printf("Thread %d: set s to {%d, %d, %d}\n", tid, s.x, s.y, s.z);

    printf("Thread %d: releasing lock...\n", tid);
    atomic_flag_clear(&f);

    return 0;
}

int main(void)
{
    thrd_t t1, t2;
    int tid[] = {0, 1};

    thrd_create(&t1, run, tid+0);
    thrd_create(&t2, run, tid+1);

    thrd_join(t1, NULL);
    thrd_join(t2, NULL);
}
```

Output ví dụ (thay đổi giữa các lần chạy):

```
Thread 0: waiting for lock...
Thread 0: got lock, s is {1, 2, 3}
Thread 1: waiting for lock...
Thread 0: set s to {5, 6, 7}
Thread 0: releasing lock...
Thread 1: got lock, s is {5, 6, 7}
Thread 1: set s to {10, 11, 12}
Thread 1: releasing lock...
```

Xem thêm

`atomic_flag_clear()`

18.17 `atomic_flag_clear()`

Xoá một atomic flag

Synopsis

```
#include <stdatomic.h>

void atomic_flag_clear(volatile atomic_flag *object);

void atomic_flag_clear_explicit(volatile atomic_flag *object,
                                memory_order order);
```

Mô tả

Xoá một atomic flag.

Như thường lệ, `_explicit` cho phép bạn chỉ định memory order khác.

Giá trị trả về

Không trả về gì cả!

Ví dụ

Dùng test-and-set để cài đặt một spin lock[7]:

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

// Shared non-atomic struct
struct {
    int x, y, z;
} s = {1, 2, 3};

atomic_flag f = ATOMIC_FLAG_INIT;

int run(void *arg)
{
    int tid = *(int*)arg;

    printf("Thread %d: waiting for lock...\n", tid);

    while (atomic_flag_test_and_set(&f));

    printf("Thread %d: got lock, s is {%d, %d, %d}\n", tid,
                                                       s.x, s.y, s.z);
    s.x = (tid + 1) * 5 + 0;
    s.y = (tid + 1) * 5 + 1;
    s.z = (tid + 1) * 5 + 2;
    printf("Thread %d: set s to {%d, %d, %d}\n", tid, s.x, s.y, s.z);

    printf("Thread %d: releasing lock...\n", tid);
    atomic_flag_clear(&f);

    return 0;
}
```

[7]Đừng dùng cái này trừ khi bạn biết mình đang làm gì—dùng chức năng mutex của thread thay vào đó. Nó sẽ cho phép thread bị chặn ngủ và ngừng gặm CPU.

```
int main(void)
{
    thrd_t t1, t2;
    int tid[] = {0, 1};

    thrd_create(&t1, run, tid+0);
    thrd_create(&t2, run, tid+1);

    thrd_join(t1, NULL);
    thrd_join(t2, NULL);
}
```

Output ví dụ (thay đổi giữa các lần chạy):

```
Thread 0: waiting for lock...
Thread 0: got lock, s is {1, 2, 3}
Thread 1: waiting for lock...
Thread 0: set s to {5, 6, 7}
Thread 0: releasing lock...
Thread 1: got lock, s is {5, 6, 7}
Thread 1: set s to {10, 11, 12}
Thread 1: releasing lock...
```

Xem thêm

`atomic_flag_test_and_set()`

Chapter 19

`<stdbit.h>` Các Hàm Liên Quan Đến Bit

Header file này cùng với toàn bộ các hàm bên trong đều là mới trong C23!

Lưu ý: không có compiler nào của tôi hỗ trợ cái này, nên toàn bộ code chưa được kiểm thử!

Hàm	Mô tả
`stdc_bit_ceil()`	Trả về lũy thừa của 2 nhỏ nhất không nhỏ hơn một số
`stdc_bit_floor()`	Trả về lũy thừa của 2 lớn nhất không lớn hơn một số
`stdc_bit_width()`	Trả về số bit cần dùng để lưu một giá trị
`stdc_count_ones()`	Đếm số bit 1 trong một số unsigned
`stdc_count_zeros()`	Đếm số bit 0 trong một số unsigned
`stdc_first_leading_one()`	Tìm bit 1 đầu tiên từ phía trước trong một số unsigned
`stdc_first_leading_zero()`	Tìm bit 0 đầu tiên từ phía trước trong một số unsigned
`stdc_first_trailing_one()`	Tìm bit 1 đầu tiên từ phía sau trong một số unsigned
`stdc_first_trailing_zero()`	Tìm bit 0 đầu tiên từ phía sau trong một số unsigned
`stdc_has_single_bit()`	Kiểm tra xem một số nguyên unsigned có đúng một bit được set không
`stdc_leading_ones()`	Đếm số 1 đầu (đếm số 1 đầu) trong một số unsigned
`stdc_leading_zeros()`	Đếm số 0 đầu (đếm số 0 đầu) trong một số unsigned
`stdc_trailing_ones()`	Đếm số 1 cuối (đếm số 1 cuối) trong một số unsigned
`stdc_trailing_zeros()`	Đếm số 0 cuối (đếm số 0 cuối) trong một số unsigned

19.1 Macro Kiểm Tra Sự Tồn Tại

Vì đây là một tính năng mới, bạn có thể kiểm tra sự tồn tại của nó bằng cách đảm bảo macro `__STDC_VERSION_STDBIT_H__` tồn tại và có giá trị là `202311L` hoặc lớn hơn.

19.2 Các Macro Endian

Header file này định nghĩa một số macro có thể dùng để xác định *endian* của hệ thống. (Nghĩa là, trong một giá trị nhiều byte, byte đầu tiên có biểu diễn phần có trọng số lớn nhất của giá trị không, hay nhỏ nhất? Hay chẳng phải cái nào?)

Có hai giá trị khác nhau cho các endian đã định nghĩa là `__STDC_ENDIAN_BIG__` và `__STDC_ENDIAN_LITTLE__`.

Ngoài ra, còn có một macro `__STDC_ENDIAN_NATIVE__` (native endian) cho bạn biết endian của *hệ thống này*. Bạn có thể so sánh nó với hai macro kia để xem mình đang có gì. Nếu nó không bằng cái nào trong hai, thì chắc bạn đang ở trên một hệ thống mixed-endian nào đó.

Thử xem hệ thống này là gì nào:

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

int main(void)
{
    switch(__STDC_ENDIAN_NATIVE__) {
        case __STDC_ENDIAN_BIG__:
            puts("Big-endian!");
            break;

        case __STDC_ENDIAN_LITTLE__:
            puts("Little-endian!");
            break;

        default:
            puts("Other-endian!");
    }
}
```

19.3 Cấu Trúc Chung Của Các Hàm Này

Tất cả các hàm trong header này đều đi theo một khuôn mẫu chuẩn, nên ta cũng nên điểm qua một lần ở đây ngay từ đầu để khỏi dài dòng lê thê về sau.

Mỗi hàm đều có sáu dạng thần thánh, tùy theo các kiểu liên quan.

Trước hết, chúng chỉ làm việc trên các kiểu số nguyên unsigned, nên ta gạt chuyện đó ra khỏi đầu luôn.

Và chúng *phần lớn* sẽ trả về `unsigned int`.

Và chúng đi theo hai dạng con: dạng type-specific và dạng type-generic.

Ta nhìn dạng type-specific trước. Ta sẽ dùng hàm đếm số bit 0 đầu trong một giá trị làm ví dụ. Đây:

```
unsigned int stdc_leading_zeros_uc(unsigned char value);
unsigned int stdc_leading_zeros_us(unsigned short value);
unsigned int stdc_leading_zeros_ui(unsigned int value);
unsigned int stdc_leading_zeros_ul(unsigned long value);
unsigned int stdc_leading_zeros_ull(unsigned long long value);
```

Ôi chao. OK, ta có gì nào?

Để tránh ô nhiễm namespace, tất cả đều bắt đầu bằng `stdc_`.

Còn mấy cái `uc`, `us`, `ull`, v.v. thì sao? Chúng tương ứng với kiểu của tham số, bạn sẽ thấy nếu để ý một chút.

Hậu tố	Kiểu tham số
`_uc`	`unsigned char`
`_us`	`unsigned short`
`_ui`	`unsigned int`
`_ul`	`unsigned long`
`_ull`	`unsigned long long`

Nhưng khoan! Còn nữa!

Mỗi họ hàm này trong header còn có một biến thể *generic*.

Lại lấy `count_leading_zeros` làm ví dụ, có một phiên bản không cần chỉ định kiểu, nó chỉ là tên hàm không có hậu tố nào.

```
generic_return_type stdc_leading_zeros(generic_value_type value);
```

Trong ví dụ đó, các chữ `generic_return_type` và `generic_value_type` **không** phải là từ khóa C. Chúng chỉ là placeholder để báo cho bạn biết chỗ đó cần thay bằng thứ đúng.

```
unsigned short x = 3490;

unsigned int a = stdc_leading_zeros(x);
auto b = stdc_leading_zeros(1234);
```

Các hàm generic làm việc với tất cả các kiểu unsigned, không tính `bool`.

19.4 `stdc_leading_zeros()`

Đếm số 0 đầu (đếm số 0 đầu) trong một số unsigned

Synopsis

Mới trong C23!

```
#include <stdbit.h>

unsigned int stdc_leading_zeros_uc(unsigned char value);
unsigned int stdc_leading_zeros_us(unsigned short value);
unsigned int stdc_leading_zeros_ui(unsigned int value);
unsigned int stdc_leading_zeros_ul(unsigned long value);
unsigned int stdc_leading_zeros_ull(unsigned long long value);

generic_return_type stdc_leading_zeros(generic_value_type value);
```

Mô tả

Hàm này trả về số bit 0 ở đầu của một giá trị nhất định, tính từ bit có trọng số lớn nhất. Con số này sẽ bị ảnh hưởng bởi kích thước của tham số.

Giá trị trả về

Trả về số bit 0 ở đầu.

Ví dụ

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

int main(void)
{
    unsigned int value = 3490;
    unsigned int count = stdc_leading_zeros_ui(value);
```

```
    printf("%u\n", count);

    unsigned long long value2 = 3490;
    auto count2 = stdc_leading_zeros(value2);

    printf("%u\n", count2);
}
```

Xem thêm

`stdc_leading_ones()`, `stdc_trailing_zeros()`

19.5 `stdc_leading_ones()`

Đếm số 1 đầu (đếm số 1 đầu) trong một số unsigned

Synopsis

Mới trong C23!

```
#include <stdbit.h>

unsigned int stdc_leading_ones_uc(unsigned char value);
unsigned int stdc_leading_ones_us(unsigned short value);
unsigned int stdc_leading_ones_ui(unsigned int value);
unsigned int stdc_leading_ones_ul(unsigned long value);
unsigned int stdc_leading_ones_ull(unsigned long long value);

generic_return_type stdc_leading_ones(generic_value_type value);
```

Mô tả

Hàm này trả về số bit 1 ở đầu của một giá trị nhất định, tính từ bit có trọng số lớn nhất. Con số này sẽ bị ảnh hưởng bởi kích thước của tham số.

Giá trị trả về

Trả về số bit 1 ở đầu.

Ví dụ

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

int main(void)
{
    unsigned int value = 3490;
    unsigned int count = stdc_leading_ones_ui(value);

    printf("%u\n", count);

    unsigned long long value2 = 3490;
    auto count2 = stdc_leading_ones(value2);
```

```
    printf("%u\n", count2);
}
```

Xem thêm

`stdc_leading_zeros()`, `stdc_trailing_ones()`

19.6 `stdc_trailing_zeros()`

Đếm số 0 cuối (đếm số 0 cuối) trong một số unsigned

Synopsis

Mới trong C23!

```
#include <stdbit.h>

unsigned int stdc_trailing_zeros_uc(unsigned char value);
unsigned int stdc_trailing_zeros_us(unsigned short value);
unsigned int stdc_trailing_zeros_ui(unsigned int value);
unsigned int stdc_trailing_zeros_ul(unsigned long value);
unsigned int stdc_trailing_zeros_ull(unsigned long long value);

generic_return_type stdc_trailing_zeros(generic_value_type value);
```

Mô tả

Hàm này trả về số bit 0 ở cuối của một giá trị nhất định, tính từ bit có trọng số nhỏ nhất. Con số này sẽ bị ảnh hưởng bởi kích thước của tham số.

Giá trị trả về

Trả về số bit 0 ở cuối.

Ví dụ

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

int main(void)
{
    unsigned int value = 3490;
    unsigned int count = stdc_trailing_zeros_ui(value);

    printf("%u\n", count);

    unsigned long long value2 = 3490;
    auto count2 = stdc_trailing_zeros(value2);

    printf("%u\n", count2);
}
```

Xem thêm

`stdc_trailing_ones()`, `stdc_leading_zeros()`

19.7 `stdc_trailing_ones()`

Đếm số 1 cuối (đếm số 1 cuối) trong một số unsigned

Synopsis

Mới trong C23!

```
#include <stdbit.h>

unsigned int stdc_trailing_ones_uc(unsigned char value);
unsigned int stdc_trailing_ones_us(unsigned short value);
unsigned int stdc_trailing_ones_ui(unsigned int value);
unsigned int stdc_trailing_ones_ul(unsigned long value);
unsigned int stdc_trailing_ones_ull(unsigned long long value);

generic_return_type stdc_trailing_ones(generic_value_type value);
```

Mô tả

Hàm này trả về số bit 1 ở cuối của một giá trị nhất định, tính từ bit có trọng số nhỏ nhất. Con số này sẽ bị ảnh hưởng bởi kích thước của tham số.

Giá trị trả về

Trả về số bit 1 ở cuối.

Ví dụ

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

int main(void)
{
    unsigned int value = 3490;
    unsigned int count = stdc_trailing_ones_ui(value);

    printf("%u\n", count);

    unsigned long long value2 = 3490;
    auto count2 = stdc_trailing_ones(value2);

    printf("%u\n", count2);
}
```

Xem thêm

`stdc_trailing_zeros()`, `stdc_leading_ones()`

19.8 `stdc_first_leading_zero()`

Tìm bit 0 đầu tiên từ phía trước trong một số unsigned

Synopsis

Mới trong C23!

```
#include <stdbit.h>

unsigned int stdc_first_leading_zero_uc(unsigned char value);
unsigned int stdc_first_leading_zero_us(unsigned short value);
unsigned int stdc_first_leading_zero_ui(unsigned int value);
unsigned int stdc_first_leading_zero_ul(unsigned long value);
unsigned int stdc_first_leading_zero_ull(unsigned long long value);

generic_return_type stdc_first_leading_zero(generic_value_type value);
```

Mô tả

Hàm này tìm chỉ số (index) của bit 0 đầu tiên từ phía trước trong một số. Chỉ số được đánh từ `1` là vị trí bit có trọng số lớn nhất (bit bên trái nhất). (Có thể cái này khác với cách bạn vẫn quen đánh số chỉ số bit.)

Nó đánh theo cơ số 1 để bạn có thể dùng nhanh giá trị trả về như một biểu thức Boolean để xem có tìm thấy bit 0 hay không.

Giá trị trả về

Trả về chỉ số bắt đầu từ 1, tính từ bit có trọng số lớn nhất, của bit 0 đầu tiên trong `value`, hoặc `0` nếu không có bit 0 nào.

Ví dụ

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

int main(void)
{
    unsigned int value = UINT_MAX;
    unsigned int index = stdc_first_leading_zero_ui(value);

    printf("%u\n", index);

    unsigned long long value2 = UINT_MAX >> 2;
    auto index2 = stdc_first_leading_zero(value2);

    printf("%u\n", index2);
}
```

Xem thêm

`stdc_first_leading_one()`, `stdc_first_trailing_zero()`

19.9 `stdc_first_leading_one()`

Tìm bit 1 đầu tiên từ phía trước trong một số unsigned

Synopsis

Mới trong C23!

```
#include <stdbit.h>

unsigned int stdc_first_leading_one_uc(unsigned char value);
unsigned int stdc_first_leading_one_us(unsigned short value);
unsigned int stdc_first_leading_one_ui(unsigned int value);
unsigned int stdc_first_leading_one_ul(unsigned long value);
unsigned int stdc_first_leading_one_ull(unsigned long long value);

generic_return_type stdc_first_leading_one(generic_value_type value);
```

Mô tả

Hàm này tìm chỉ số của bit 1 đầu tiên từ phía trước trong một số. Chỉ số được đánh từ `1` là vị trí bit có trọng số lớn nhất (bit bên trái nhất). (Có thể cái này khác với cách bạn vẫn quen đánh số chỉ số bit.)

Nó đánh theo cơ số 1 để bạn có thể dùng nhanh giá trị trả về như một biểu thức Boolean để xem có tìm thấy bit 1 hay không.

Giá trị trả về

Trả về chỉ số bắt đầu từ 1, tính từ bit có trọng số lớn nhất, của bit 1 đầu tiên trong `value`, hoặc `0` nếu không có bit 1 nào.

Ví dụ

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

int main(void)
{
    unsigned int value = UINT_MAX;
    unsigned int index = stdc_first_leading_one_ui(value);

    printf("%u\n", index);

    unsigned long long value2 = UINT_MAX >> 2;
    auto index2 = stdc_first_leading_one(value2);

    printf("%u\n", index2);
}
```

Xem thêm

`stdc_first_leading_zero()`, `stdc_first_trailing_one()`

19.10 `stdc_first_trailing_zero()`

Tìm bit 0 đầu tiên từ phía sau trong một số unsigned

Synopsis

Mới trong C23!

```
#include <stdbit.h>

unsigned int stdc_first_trailing_zero_uc(unsigned char value);
unsigned int stdc_first_trailing_zero_us(unsigned short value);
unsigned int stdc_first_trailing_zero_ui(unsigned int value);
unsigned int stdc_first_trailing_zero_ul(unsigned long value);
unsigned int stdc_first_trailing_zero_ull(unsigned long long value);

generic_return_type stdc_first_trailing_zero(generic_value_type value);
```

Mô tả

Hàm này tìm chỉ số của bit 0 đầu tiên từ phía sau trong một số. Chỉ số được đánh từ `1` là vị trí bit có trọng số lớn nhất (bit bên trái nhất). (Có thể cái này khác với cách bạn vẫn quen đánh số chỉ số bit.)

Nó đánh theo cơ số 1 để bạn có thể dùng nhanh giá trị trả về như một biểu thức Boolean để xem có tìm thấy bit 0 hay không.

Giá trị trả về

Trả về chỉ số bắt đầu từ 1, tính từ bit có trọng số lớn nhất, của bit 0 đầu tiên trong `value`, hoặc `0` nếu không có bit 0 nào.

Ví dụ

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

int main(void)
{
    unsigned int value = UINT_MAX;
    unsigned int index = stdc_first_trailing_zero_ui(value);

    printf("%u\n", index);

    unsigned long long value2 = UINT_MAX >> 2;
    auto index2 = stdc_first_trailing_zero(value2);

    printf("%u\n", index2);
}
```

Xem thêm

`stdc_first_leading_zero()`, `stdc_first_trailing_one()`

19.11 `stdc_first_trailing_one()`

Tìm bit 1 đầu tiên từ phía sau trong một số unsigned

Synopsis

Mới trong C23!

```
#include <stdbit.h>

unsigned int stdc_first_trailing_one_uc(unsigned char value);
unsigned int stdc_first_trailing_one_us(unsigned short value);
unsigned int stdc_first_trailing_one_ui(unsigned int value);
unsigned int stdc_first_trailing_one_ul(unsigned long value);
unsigned int stdc_first_trailing_one_ull(unsigned long long value);

generic_return_type stdc_first_trailing_one(generic_value_type value);
```

Mô tả

Hàm này tìm chỉ số của bit 1 đầu tiên từ phía sau trong một số. Chỉ số được đánh từ `1` là vị trí bit có trọng số lớn nhất (bit bên trái nhất). (Có thể cái này khác với cách bạn vẫn quen đánh số chỉ số bit.)

Nó đánh theo cơ số 1 để bạn có thể dùng nhanh giá trị trả về như một biểu thức Boolean để xem có tìm thấy bit 1 hay không.

Giá trị trả về

Trả về chỉ số bắt đầu từ 1, tính từ bit có trọng số lớn nhất, của bit 1 đầu tiên trong `value`, hoặc `0` nếu không có bit 1 nào.

Ví dụ

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

int main(void)
{
    unsigned int value = UINT_MAX;
    unsigned int index = stdc_first_trailing_one_ui(value);

    printf("%u\n", index);

    unsigned long long value2 = UINT_MAX >> 2;
    auto index2 = stdc_first_trailing_one(value2);

    printf("%u\n", index2);
}
```

Xem thêm

`stdc_first_leading_one()`, `stdc_first_trailing_zero()`

19.12 stdc_count_zeros()

Đếm số bit 0 trong một số unsigned

Synopsis

Mới trong C23!

```
#include <stdbit.h>

unsigned int stdc_count_zeros_uc(unsigned char value);
unsigned int stdc_count_zeros_us(unsigned short value);
unsigned int stdc_count_zeros_ui(unsigned int value);
unsigned int stdc_count_zeros_ul(unsigned long value);
unsigned int stdc_count_zeros_ull(unsigned long long value);

generic_return_type stdc_count_zeros(generic_value_type value);
```

Mô tả

Hàm này trả về số bit 0 trong một giá trị nhất định. Con số này sẽ bị ảnh hưởng bởi kích thước của tham số.

Giá trị trả về

Trả về số bit 0.

Ví dụ

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

int main(void)
{
    unsigned int value = 3490;
    unsigned int count = stdc_count_zeros_ui(value);

    printf("%u\n", count);

    unsigned long long value2 = 123456;
    auto count2 = stdc_count_zeros(value2);

    printf("%u\n", count2);
}
```

Xem thêm

`stdc_count_ones()()` , `stdc_leading_zeros()()` , `stdc_trailing_zeros()()`

19.13 stdc_count_ones()

Đếm số bit 1 (population count / popcount — đếm số bit 1) trong một số unsigned

Synopsis

Mới trong C23!

```
#include <stdbit.h>

unsigned int stdc_count_ones_uc(unsigned char value);
unsigned int stdc_count_ones_us(unsigned short value);
unsigned int stdc_count_ones_ui(unsigned int value);
unsigned int stdc_count_ones_ul(unsigned long value);
unsigned int stdc_count_ones_ull(unsigned long long value);

generic_return_type stdc_count_ones(generic_value_type value);
```

Mô tả

Hàm này trả về số bit 1 trong một giá trị nhất định. Con số này sẽ bị ảnh hưởng bởi kích thước của tham số.

Giá trị trả về

Trả về số bit 1.

Ví dụ

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

int main(void)
{
    unsigned int value = 3490;
    unsigned int count = stdc_count_ones_ui(value);

    printf("%u\n", count);

    unsigned long long value2 = 123456;
    auto count2 = stdc_count_ones(value2);

    printf("%u\n", count2);
}
```

Xem thêm

`stdc_count_zeros()()`, `stdc_leading_ones()()`, `stdc_trailing_ones()()`

19.14 `stdc_has_single_bit()`

Kiểm tra xem một số nguyên unsigned có đúng một bit được set không

Synopsis

Mới trong C23!

```
#include <stdbit.h>

bool stdc_has_single_bit_uc(unsigned char value);
bool stdc_has_single_bit_us(unsigned short value);
bool stdc_has_single_bit_ui(unsigned int value);
bool stdc_has_single_bit_ul(unsigned long value);
bool stdc_has_single_bit_ull(unsigned long long value);

bool stdc_has_single_bit(generic_value_type value);
```

Mô tả

Các hàm này trả về true nếu có đúng một bit bằng `1` trong `value` unsigned.

Nếu true, nó cũng có nghĩa là `value` là một lũy thừa của 2 (lũy thừa của 2).

Giá trị trả về

Trả về true nếu `value` có đúng một bit được set bằng `1`.

Ví dụ

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

int main(void)
{
    unsigned int value = 0b0001000;
    unsigned int result = stdc_has_single_bit_ui(value);

    printf("%d\n", result);    // 1

    unsigned long long value2 = 0b000011000
    auto result2 = stdc_has_single_bit(value2);

    printf("%d\n", result2);   // 0
}
```

19.15 `stdc_bit_width()`

Trả về số bit (bit width / độ rộng bit) cần dùng để lưu một giá trị

Synopsis

Mới trong C23!

```
#include <stdbit.h>

unsigned int stdc_bit_width_uc(unsigned char value);
unsigned int stdc_bit_width_us(unsigned short value);
unsigned int stdc_bit_width_ui(unsigned int value);
unsigned int stdc_bit_width_ul(unsigned long value);
unsigned int stdc_bit_width_ull(unsigned long long value);
```

```
generic_return_type stdc_bit_width(generic_value_type value);
```

Mô tả

Cho một giá trị số nguyên unsigned, số bit nhỏ nhất cần dùng để lưu nó là bao nhiêu? Đó là câu hỏi mà hàm này trả lời.

Giá trị trả về

Trả về số bit cần dùng để lưu một `value` dương. Trả về `0` nếu `value` truyền vào là `0`.

Ví dụ

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

int main(void)
{
    unsigned int value = 0b0001000;
    unsigned int result = stdc_bit_width_ui(value);

    printf("%d\n", result);    // 4

    unsigned long long value2 = 0b000011010
    auto result2 = stdc_bit_width(value2);

    printf("%d\n", result2);   // 5
}
```

19.16 `stdc_bit_floor()`

Trả về lũy thừa của 2 (lũy thừa của 2) lớn nhất không lớn hơn một số

Synopsis

Mới trong C23!

```
#include <stdbit.h>

unsigned char stdc_bit_floor_uc(unsigned char value);
unsigned short stdc_bit_floor_us(unsigned short value);
unsigned int stdc_bit_floor_ui(unsigned int value);
unsigned long stdc_bit_floor_ul(unsigned long value);
unsigned long long stdc_bit_floor_ull(unsigned long long value);

generic_value_type stdc_bit_floor(generic_value_type value);
```

Mô tả

Hàm này trả về lũy thừa của 2 lớn nhất không lớn hơn `value`.

Nói cách khác, "nhỏ hơn hoặc bằng" `value`.

Nói cách khác, làm tròn xuống (floor) đến lũy thừa của 2 gần nhất.

Nói cách khác, trả về giá trị của bit được set cao nhất trong một số.

Ví dụ: | `value` |Giá trị trả về| |-|-| | `0b101` | `0b100` | | `0b1000` | `0b1000` | | `0b100101` | `0b100000` |

Giá trị trả về

Trả về lũy thừa của 2 lớn nhất không lớn hơn `value`. Trả về `0` nếu `value` là `0`.

Ví dụ

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

int main(void)
{
    unsigned int value = 0b0001000;
    unsigned int result = stdc_bit_floor_ui(value);

    printf("%b\n", result);    // 1000

    unsigned long long value2 = 0b000011010
    auto result2 = stdc_bit_floor(value2);

    printf("%b\n", result2);   // 10000
}
```

Xem thêm

`stdc_bit_ceil()`

19.17 `stdc_bit_ceil()`

Trả về lũy thừa của 2 nhỏ nhất không nhỏ hơn một số

Synopsis

Mới trong C23!

```
#include <stdbit.h>

unsigned char stdc_bit_ceil_uc(unsigned char value);
unsigned short stdc_bit_ceil_us(unsigned short value);
unsigned int stdc_bit_ceil_ui(unsigned int value);
unsigned long stdc_bit_ceil_ul(unsigned long value);
unsigned long long stdc_bit_ceil_ull(unsigned long long value);

generic_value_type stdc_bit_ceil(generic_value_type value);
```

Mô tả

Hàm này trả về lũy thừa của 2 nhỏ nhất không nhỏ hơn `value`.

Nói cách khác, "lớn hơn hoặc bằng" `value`.

Nói cách khác, làm tròn lên (ceil) đến lũy thừa của 2 gần nhất.

Ví dụ: | `value` |Giá trị trả về| |-|-| | `0b101` | `0b1000` | | `0b1000` | `0b1000` | | `0b100101` | `0b1000000` |

Nếu kết quả không vừa với kiểu trả về, hành vi là không xác định (undefined).

Giá trị trả về

Trả về lũy thừa của 2 nhỏ nhất không nhỏ hơn `value`.

Ví dụ

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

int main(void)
{
    unsigned int value = 0b0001000;
    unsigned int result = stdc_bit_ceil_ui(value);

    printf("%b\n", result);    // 1000

    unsigned long long value2 = 0b000011010
    auto result2 = stdc_bit_ceil(value2);

    printf("%b\n", result2);   // 100000
}
```

Xem thêm

`stdc_bit_floor()`

Chapter 20

`<stdbool.h>` Kiểu Boolean

Đây là header file nhỏ định nghĩa vài macro Boolean tiện tay. Nếu bạn thực sự cần mấy thứ này.

Macro	Mô tả
`bool`	Kiểu cho Boolean, expand thành `_Bool`
`true`	Giá trị true, expand thành `1`
`false`	Giá trị false, expand thành `0`

Còn một macro nữa tôi không cho vào bảng vì tên dài ngoẵng sẽ phá nát bảng:

```
__bool_true_false_are_defined
```

expand thành `1`.

20.1 Ví dụ

Đây là ví dụ hết sức tầm thường để show off mấy macro này.

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

int main(void)
{
    bool x;

    x = (3 > 2);

    if (x == true)
        printf("The universe still makes sense.\n");

    x = false;

    printf("x is now %d\n", x);  // 0
}
```

Output:

```
The universe still makes sense.
x is now 0
```

20.2 `_Bool`?

`_Bool` là sao? Sao họ không đặt luôn là `bool`?

Chà, ngoài kia có cả đống code C mà người ta đã tự định nghĩa kiểu `bool` của riêng mình, và thêm một `bool` chính thức sẽ làm hỏng những `typedef` đó.

Nhưng C đã reserve sẵn mọi định danh bắt đầu bằng dấu gạch dưới theo sau là chữ cái viết hoa, nên rõ ràng cách làm là nặn ra kiểu `_Bool` mới và xài nó.

Và, nếu bạn biết code của mình xử được, bạn có thể include header này để lấy đống syntax ngon nghẻ này.

Một ghi chú nữa về chuyện chuyển đổi: khác với chuyển sang `int`, thứ *duy nhất* chuyển thành `false` trong một `_Bool` là giá trị scalar bằng không. Bất cứ thứ gì khác không, như `-3490`, `0.12`, hay `NaN`, đều chuyển thành `true`.

Chapter 21

`<stddef.h>` Vài Định Nghĩa Chuẩn

Dù tên gọi vậy, tôi hiếm khi thấy header này được include.

Nó gồm vài kiểu và macro.

Tên	Mô tả
`ptrdiff_t`	Integer có dấu cho hiệu giữa hai pointer
`size_t`	Kiểu integer không dấu mà `sizeof` trả về
`max_align_t`	Khai báo một kiểu với alignment lớn nhất có thể
`wchar_t`	Kiểu ký tự rộng
`NULL`	Con trỏ `NULL`, được định nghĩa ở vài chỗ
`offsetof`	Lấy byte offset của trường trong `struct` hoặc `union`

21.1 `ptrdiff_t`

Kiểu này giữ hiệu giữa hai pointer. Bạn có thể lưu kết quả này vào kiểu khác, nhưng kiểu của kết quả phép trừ pointer là implementation-defined; portable tối đa thì dùng `ptrdiff_t`.

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

int main(void)
{
    int cats[100];

    int *f = cats + 20;
    int *g = cats + 60;

    ptrdiff_t d = g - f;  // hiệu là 40
```

Và bạn có thể in nó bằng cách thêm prefix `t` vào format specifier integer:

```
    printf("%td\n", d);  // In thập phân: 40
    printf("%tX\n", d);  // In hex:       28
}
```

21.2 `size_t`

Đây là kiểu mà `sizeof` trả về và được dùng ở vài chỗ khác. Nó là integer không dấu.

Bạn có thể in nó dùng prefix `z` trong `printf()`:

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

int main(void)
{
    size_t x;

    x = sizeof(int);

    printf("%zu\n", x);
```

Một số hàm trả về số âm cast thành `size_t` làm giá trị lỗi (như `mbrtoc16()`). Nếu muốn in mấy cái đó dưới dạng giá trị âm, bạn có thể dùng `%zd`:

```
    char16_t a;
    mbstate_t mbs;
    memset(&mbs, 0, sizeof mbs);

    x = mbrtoc16(&a, "b", 8, &mbs);

    printf("%zd\n", x);
}
```

21.3 `max_align_t`

Theo hiểu của tôi, cái này tồn tại để cho phép tính runtime alignment[1] fundamental tối đa trên nền tảng hiện tại. Ai biết thêm công dụng khác làm ơn mail cho tôi.

Có lẽ bạn cần cái này nếu đang viết memory allocator của riêng mình hay gì đó.

```
#include <stddef.h>
#include <stdio.h>      // For printf()
#include <stdalign.h>   // For alignof

int main(void)
{
    int max = alignof(max_align_t);

    printf("Maximum fundamental alignment: %d\n", max);
}
```

Trên hệ của tôi, in ra:

```
Maximum fundamental alignment: 16
```

Xem thêm `alignas`, `alignof`.

[1] https://en.wikipedia.org/wiki/Data_structure_alignment

21.4 `wchar_t`

Cái này tương tự `char`, chỉ khác là nó dành cho ký tự rộng (wide character).

Đây là kiểu integer có tầm đủ lớn để giữ giá trị duy nhất cho mọi ký tự trong mọi locale được hỗ trợ.

Giá trị `0` là ký tự `NUL` rộng.

Cuối cùng, giá trị của các hằng ký tự từ tập ký tự cơ bản sẽ bằng với giá trị `wchar_t` tương ứng... trừ khi `__STDC_MB_MIGHT_NEQ_WC__` được định nghĩa.

21.5 `offsetof`

Nếu bạn có `struct` hoặc `union`, bạn có thể dùng cái này để lấy byte offset của các trường trong kiểu đó.

Cách dùng:

```
offsetof(type, fieldname);
```

Giá trị kết quả có kiểu `size_t`.

Đây là ví dụ in offset trường của một `struct`:

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

struct foo {
    int a;
    char b;
    char c;
    float d;
};

int main(void)
{
    printf("a: %zu\n", offsetof(struct foo, a));
    printf("b: %zu\n", offsetof(struct foo, b));
    printf("c: %zu\n", offsetof(struct foo, c));
    printf("d: %zu\n", offsetof(struct foo, d));
}
```

Trên hệ của tôi, output là:

```
a: 0
b: 4
c: 5
d: 8
```

Và bạn không thể dùng `offsetof` trên bitfield, nên đừng hi vọng hão.

Chapter 22

`<stdint.h>` Thêm Kiểu Integer

Header này cho ta quyền truy cập (có thể) các kiểu có số bit cố định, hoặc, ít ra, các kiểu có ít nhất từng đó bit.

Nó cũng cho ta mấy macro tiện tay.

22.1 Integer Kích Thước Cụ Thể

Có ba nhóm kiểu chính được định nghĩa ở đây, có dấu và không dấu:

- Integer chính xác một kích thước (`int` N `_t` , `uint` N `_t`)
- Integer ít nhất một kích thước (`int_least` N `_t` , `uint_least` N `_t`)
- Integer ít nhất một kích thước và nhanh hết mức có thể (`int_fast` N `_t` , `uint_fast` N `_t`)

Chỗ N xuất hiện, bạn thay vào số bit, thường là bội của 8, ví dụ `uint16_t` .

Các kiểu sau được đảm bảo có định nghĩa:

```
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
```

Những cái còn lại là tùy chọn, nhưng bạn có thể sẽ có luôn các kiểu sau, chúng được yêu cầu khi hệ thống có integer đúng kích thước đó không có padding và dùng biểu diễn two's-complement... đây là trường hợp với Mac, PC và cả đống hệ thống khác. Tóm lại, bạn nhiều khả năng có sẵn mấy cái này:

```
int8_t      uint8_t
int16_t     uint16_t
int32_t     uint32_t
int64_t     uint64_t
```

Số bit khác cũng có thể được hỗ trợ nếu implementation muốn làm chuyện điên rồ.

Ví dụ:

```
#include <stdint.h>

int main(void)
{
    int16_t x = 32;
    int_fast32_t y = 3490;

    // ...
```

22.2 Các Kiểu Integer Khác

Có vài kiểu tùy chọn là integer có khả năng giữ kiểu pointer.

```
intptr_t
uintptr_t
```

Bạn có thể convert `void*` thành một trong các kiểu này, rồi trở lại. Và các `void*` sẽ so sánh bằng nhau.

Use case là bất cứ chỗ nào bạn cần một integer đại diện cho pointer vì lý do nào đó.

Ngoài ra, có vài kiểu tồn tại chỉ để là integer lớn nhất mà hệ của bạn hỗ trợ:

```
intmax_t
uintmax_t
```

Fact vui: bạn có thể in các kiểu này với format specifier `"%jd"` và `"%ju"` của `printf()`.

Cũng có cả đống macro trong `<inttypes.h>` (#inttypes) bạn có thể dùng để in bất kỳ kiểu nào ở trên.

22.3 Macros

Các macro sau định nghĩa giá trị min và max 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

INTPTR_MAX         INTPTR_MIN         UINTPTR_MAX
```

Với kiểu signed kích thước chính xác, min chính xác là $-(2^{N-1})$ và max chính xác là $2^{N-1}-1$. Và với kiểu unsigned kích thước chính xác, max chính xác là 2^N-1.

Với biến thể signed "least" và "fast", độ lớn và dấu của min ít nhất là $-(2^{N-1}-1)$ và max ít nhất là $2^{N-1}-1$. Và với unsigned, ít nhất là 2^N-1.

`INTMAX_MAX` ít nhất là $2^{63}-1$, `INTMAX_MIN` ít nhất là $-(2^{63}-1)$ xét về dấu và độ lớn. Và `UINTMAX_MAX` ít nhất là $2^{64}-1$.

Cuối cùng, `INTPTR_MAX` ít nhất là $2^{15}-1$, `INTPTR_MIN` ít nhất là $-(2^{15}-1)$ xét về dấu và độ lớn. Và `UINTPTR_MAX` ít nhất là $2^{16}-1$.

22.4 Giới Hạn Khác

Có một đống kiểu trong `<inttypes.h>` (#inttypes) có giới hạn được định nghĩa ở đây. (`<inttypes.h>` include `<stdint.h>` .)

Macro	Mô tả
`PTRDIFF_MIN`	Giá trị min của `ptrdiff_t`
`PTRDIFF_MAX`	Giá trị max của `ptrdiff_t`
`SIG_ATOMIC_MIN`	Giá trị min của `sig_atomic_t`
`SIG_ATOMIC_MAX`	Giá trị max của `sig_atomic_t`
`SIZE_MAX`	Giá trị max của `size_t`
`WCHAR_MIN`	Giá trị min của `wchar_t`
`WCHAR_MAX`	Giá trị max của `wchar_t`
`WINT_MIN`	Giá trị min của `wint_t`
`WINT_MAX`	Giá trị max của `wint_t`

Spec nói `PTRDIFF_MIN` về độ lớn sẽ ít nhất là -65535. Và `PTRDIFF_MAX` với `SIZE_MAX` sẽ ít nhất là 65535.

`SIG_ATOMIC_MIN` và `MAX` sẽ hoặc là -127 và 127 (nếu signed) hoặc 0 và 255 (nếu unsigned).

Tương tự với `WCHAR_MIN` và `MAX` .

`WINT_MIN` và `MAX` sẽ hoặc là -32767 và 32767 (nếu signed) hoặc 0 và 65535 (nếu unsigned).

22.5 Macro Để Khai Báo Hằng

Nếu bạn còn nhớ, bạn có thể chỉ định kiểu cho hằng integer:

```
int x = 12;
long int y = 12L;
unsigned long long int z = 12ULL;
```

Bạn có thể dùng macro `INT` N `_C()` và `UINT` N `()` trong đó N là `8` , `16` , `32` hoặc `64` .

```
uint_least16_t x = INT16_C(3490);
uint_least64_t y = INT64_C(1122334455);
```

Biến thể của mấy cái này là `INTMAX_C()` và `UINTMAX_C()` . Chúng tạo một hằng phù hợp để lưu trong `intmax_t` hoặc `uintmax_t` .

```
intmax_t x = INTMAX_C(3490);
uintmax_t x = UINTMAX_C(1122334455);
```

Chapter 23

`<stdio.h>` Thư Viện I/O Chuẩn

Hàm	Mô tả
`clearerr()`	Xoá cờ trạng thái `feof` và `ferror`
`fclose()`	Đóng một file đang mở
`feof()`	Trả về trạng thái end-of-file (cuối file) của file
`ferror()`	Trả về trạng thái lỗi của file
`fflush()`	Flush (đẩy đệm) toàn bộ output có đệm ra file
`fgetc()`	Đọc một ký tự từ một file
`fgetpos()`	Lấy vị trí I/O của file
`fgets()`	Đọc một dòng từ file
`fopen()`	Mở một file
`fprintf()`	In output có định dạng ra file
`fputc()`	In một ký tự ra file
`fputs()`	In một chuỗi ra file
`fread()`	Đọc dữ liệu nhị phân từ file
`freopen()`	Đổi file gắn với một stream
`fscanf()`	Đọc input có định dạng từ file
`fseek()`	Đặt vị trí I/O của file
`fsetpos()`	Đặt vị trí I/O của file
`ftell()`	Lấy vị trí I/O của file
`fwrite()`	Ghi dữ liệu nhị phân ra file
`getc()`	Lấy một ký tự từ `stdin`
`getchar()`	Lấy một ký tự từ `stdin`
`gets()`	Lấy một chuỗi từ `stdin` (đã bị gỡ bỏ trong C11)
`perror()`	In một thông báo lỗi dễ đọc cho người dùng
`printf()`	In output có định dạng ra `stdout`
`putc()`	In một ký tự ra `stdout`
`putchar()`	In một ký tự ra `stdout`
`puts()`	In một chuỗi ra `stdout`
`remove()`	Xoá một file khỏi đĩa
`rename()`	Đổi tên hoặc di chuyển một file trên đĩa
`rewind()`	Đặt vị trí I/O về đầu file
`scanf()`	Đọc input có định dạng từ `stdin`
`setbuf()`	Cấu hình đệm cho các thao tác I/O
`setvbuf()`	Cấu hình đệm cho các thao tác I/O
`snprintf()`	In output có định dạng ra chuỗi với giới hạn độ dài
`sprintf()`	In output có định dạng ra chuỗi

Hàm	Mô tả
`sscanf()`	Đọc input có định dạng từ chuỗi
`tmpfile()`	Tạo một file tạm
`tmpnam()`	Sinh một tên duy nhất cho file tạm
`ungetc()`	Đẩy ngược một ký tự trở lại input stream
`vfprintf()`	Phiên bản variadic của in output có định dạng ra file
`vfscanf()`	Phiên bản variadic của đọc input có định dạng từ file
`vprintf()`	Phiên bản variadic của in output có định dạng ra `stdout`
`vscanf()`	Phiên bản variadic của đọc input có định dạng từ `stdin`
`vsnprintf()`	Phiên bản variadic của in output có định dạng ra chuỗi với giới hạn độ dài
`vsprintf()`	Phiên bản variadic của in output có định dạng ra chuỗi
`vsscanf()`	Phiên bản variadic của đọc input có định dạng từ chuỗi

Cơ bản nhất trong tất cả các thư viện của thư viện chuẩn C chính là thư viện I/O chuẩn. Nó được dùng để đọc và ghi file. Tôi biết bạn đang rất háo hức chuyện này.

Vậy nên tôi sẽ kể tiếp. Nó cũng được dùng để đọc và ghi ra console, như chúng ta đã thấy nhiều lần với hàm `printf()`.

(Một bí mật nhỏ ở đây—trong nhiều hệ điều hành, rất nhiều thứ thực ra sâu bên trong đều là file, và console cũng không phải ngoại lệ. "*Mọi thứ trong Unix đều là file!*" `:-)`)

Chắc bạn sẽ muốn có prototype của các hàm mình có thể dùng phải không? Để đặt đôi bàn tay nhỏ xíu bẩn thỉu của bạn lên chúng, bạn cần include `stdio.h`.

Anyway, chúng ta có thể làm đủ trò hay ho với I/O file. PHÁT HIỆN NÓI DỐI. Được rồi, được rồi. Chúng ta có thể làm đủ thứ với I/O file. Về cơ bản, chiến lược như sau:

1. Dùng `fopen()` để lấy một con trỏ đến cấu trúc file kiểu `FILE*`. Con trỏ này là thứ bạn sẽ truyền vào nhiều hàm I/O file khác.
2. Dùng một vài hàm file khác, như `fscanf()`, `fgets()`, `fprintf()`, v.v. bằng `FILE*` mà `fopen()` trả về.
3. Khi xong, gọi `fclose()` với `FILE*`. Điều này cho hệ điều hành biết rằng bạn đã thực sự xong với file, không có take-back nào nữa.

Trong `FILE*` có gì? Như bạn có thể đoán, nó trỏ tới một `struct` chứa đủ loại thông tin về vị trí đọc ghi hiện tại trong file, file được mở như thế nào, và những thứ tương tự. Nhưng thành thật mà nói, ai quan tâm. Không ai cả. Cấu trúc `FILE` là *opaque* đối với bạn là một lập trình viên; nghĩa là bạn không cần biết bên trong có gì, và bạn cũng không *muốn* biết bên trong có gì. Bạn chỉ cần truyền nó vào các hàm I/O chuẩn khác và chúng biết phải làm gì.

Thực ra điều này khá quan trọng: cố gắng đừng nghịch ngợm bên trong cấu trúc `FILE`. Nó thậm chí còn khác nhau giữa các hệ thống, và bạn sẽ kết cục viết code rất không portable (không di động).

Một điều nữa cần nhắc về thư viện I/O chuẩn: rất nhiều hàm thao tác trên file dùng tiền tố "f" trong tên hàm. Hàm tương đương thao tác trên console sẽ bỏ "f" đi. Ví dụ, nếu muốn in ra console bạn dùng `printf()`, còn nếu muốn in ra file thì dùng `fprintf()`, thấy chưa?

Khoan! Nếu ghi ra console, về cơ bản, giống như ghi ra file vì mọi thứ trong Unix đều là file, thì tại sao lại có hai hàm? Câu trả lời: tiện hơn. Nhưng quan trọng hơn, có tồn tại `FILE*` nào gắn với console mà bạn có thể dùng không? Câu trả lời: CÓ!

Thực tế, có *ba* (đếm đi!) `FILE*` đặc biệt mà bạn có sẵn chỉ bằng việc include `stdio.h`. Một cho input, và hai cho output.

Nghe có vẻ không công bằng lắm—sao output lại được hai file, còn input chỉ có một?

Đừng vội—cứ xem chúng trước đã:

Stream	Mô tả
`stdin`	Input từ console.
`stdout`	Output ra console.
`stderr`	Output ra console trên file stream lỗi.

Standard input (`stdin`) mặc định là những gì bạn gõ từ bàn phím. Bạn có thể dùng nó với `fscanf()` nếu muốn, như thế này:

```
/* dòng này: */
scanf("%d", &x);

/* tương đương dòng này: */
fscanf(stdin, "%d", &x);
```

Và `stdout` cũng hoạt động tương tự:

```
printf("Hello, world!\n");
fprintf(stdout, "Hello, world!\n"); /* giống dòng trên! */
```

Vậy `stderr` là cái gì? Chuyện gì xảy ra khi bạn output ra đó? Thông thường nó cũng ra console giống `stdout`, nhưng người ta dùng nó cho các thông báo lỗi nói riêng. Tại sao? Trên nhiều hệ thống, bạn có thể redirect output của chương trình vào một file từ dòng lệnh...và đôi khi bạn chỉ quan tâm tới phần output lỗi thôi. Vậy nên nếu chương trình ngoan và ghi hết lỗi ra `stderr`, người dùng có thể redirect chỉ `stderr` vào một file và chỉ xem phần đó. Đó là một điều hay mà bạn, với tư cách lập trình viên, có thể làm.

Cuối cùng, khá nhiều hàm ở đây trả về `int` trong khi bạn có thể mong đợi `char`. Đó là vì hàm có thể trả về một ký tự *hoặc* end-of-file (`EOF`), và `EOF` có khả năng là một số nguyên. Nếu không nhận được `EOF` làm giá trị trả về, bạn có thể an toàn lưu kết quả vào một `char`.

23.1 `remove()`

Xoá một file

Synopsis

```
#include <stdio.h>

int remove(const char *filename);
```

Mô tả

Xoá file đã chỉ định khỏi hệ thống tập tin. Đơn giản là xoá. Không có gì phép thuật. Chỉ cần gọi hàm này và hiến tế một con gà nhỏ, file bạn yêu cầu sẽ bị xoá.

Giá trị trả về

Trả về 0 khi thành công, và `-1` khi lỗi, đồng thời set `errno`.

Ví dụ

```
#include <stdio.h>

int main(void)
{
    char *filename = "evidence.txt";

    remove(filename);
}
```

Xem thêm

`rename()`

23.2 `rename()`

Đổi tên file và tuỳ chọn di chuyển nó sang vị trí mới

Synopsis

```
#include <stdio.h>

int rename(const char *old, const char *new);
```

Mô tả

Đổi tên file `old` thành `new`. Dùng hàm này nếu bạn đã chán cái tên cũ của file và sẵn sàng đổi mới. Đôi khi đổi tên file đơn giản lại khiến nó cảm giác như mới, và có thể tiết kiệm tiền so với việc mua toàn bộ file mới!

Một điều hay nữa bạn có thể làm với hàm này là thực sự di chuyển một file từ thư mục này sang thư mục khác bằng cách chỉ định đường dẫn khác cho tên mới.

Giá trị trả về

Trả về 0 khi thành công, và `-1` khi lỗi, đồng thời set `errno`.

Ví dụ

```
#include <stdio.h>

int main(void)
{
    // Đổi tên một file
    rename("foo", "bar");

    // Đổi tên và chuyển sang thư mục khác:
    rename("/home/beej/evidence.txt", "/tmp/nothing.txt");
}
```

Xem thêm

`remove()`

23.3 `tmpfile()`

Tạo một file tạm

Synopsis

```
#include <stdio.h>

FILE *tmpfile(void);
```

Mô tả

Đây là một hàm nho nhỏ tiện lợi, nó sẽ tạo và mở cho bạn một file tạm, và trả về một `FILE*` để bạn dùng. File được mở ở chế độ "`r+b`", nên phù hợp để đọc, ghi và dữ liệu nhị phân.

Bằng một chút phép thuật, file tạm sẽ tự động bị xoá khi nó được `close()` hoặc khi chương trình của bạn kết thúc. (Cụ thể, theo cách Unix, `tmpfile()` *unlinks*[1] file ngay sau khi mở. Nghĩa là nó đã được đặt trong trạng thái sẵn sàng bị xoá khỏi đĩa, nhưng vẫn tồn tại vì tiến trình của bạn vẫn đang mở nó. Ngay khi tiến trình của bạn kết thúc, mọi file mở đều được đóng lại, và file tạm biến vào hư không.)

Giá trị trả về

Hàm này trả về một `FILE*` đã mở khi thành công, hoặc `NULL` khi thất bại.

Ví dụ

```
#include <stdio.h>

int main(void)
{
    FILE *temp;
    char s[128];

    temp = tmpfile();

    fprintf(temp, "What is the frequency, Alexander?\n");

    rewind(temp); // quay vê`đâù

    fscanf(temp, "%s", s); // đọc lại ra

    fclose(temp); // đóng (và xoá một cách thần kỳ)
}
```

Xem thêm

`fopen()`, `fclose()`, `tmpnam()`

23.4 `tmpnam()`

Sinh một tên duy nhất cho file tạm

[1] https://man.archlinux.org/man/unlinkat.2.en#DESCRIPTION

Synopsis

```
#include <stdio.h>

char *tmpnam(char *s);
```

Mô tả

Hàm này nhìn kỹ các file đang tồn tại trên hệ thống của bạn, rồi nghĩ ra một cái tên duy nhất cho file mới phù hợp để dùng làm file tạm.

Giả sử bạn có một chương trình cần lưu dữ liệu trong thời gian ngắn, nên bạn tạo một file tạm cho dữ liệu đó, sẽ bị xoá khi chương trình kết thúc. Giờ tưởng tượng bạn đặt tên file này là `foo.txt`. Tất cả đều ổn, cho đến khi người dùng đã có một file tên `foo.txt` ở thư mục bạn đang chạy chương trình. Bạn sẽ ghi đè file của họ, họ sẽ không vui và stalk bạn mãi mãi. Và bạn chắc không muốn thế đúng không?

Ok, nên bạn khôn ra và quyết định đặt file vào `/tmp` để không ghi đè nội dung quan trọng nào. Nhưng khoan! Lỡ có người khác đang chạy chương trình cùng lúc và cả hai muốn dùng cùng một tên file thì sao? Hoặc lỡ có chương trình khác đã tạo sẵn file đó?

Thấy chưa, tất cả những vấn đề đáng sợ này có thể tránh hoàn toàn nếu bạn dùng `tmpnam()` để lấy một tên file an toàn sẵn dùng.

Vậy dùng nó thế nào? Có hai cách tuyệt vời. Một, bạn khai báo một mảng (hoặc `malloc()` nó—sao cũng được) đủ lớn để chứa tên file tạm. Lớn bao nhiêu? May thay, đã có một macro sẵn dành cho bạn, `L_tmpnam`, cho biết mảng phải lớn bao nhiêu.

Và cách thứ hai: chỉ cần truyền `NULL` cho tên file. `tmpnam()` sẽ lưu tên tạm trong một mảng tĩnh và trả về con trỏ đến đó. Những lần gọi sau với tham số `NULL` sẽ ghi đè mảng tĩnh, nên hãy chắc là bạn đã dùng xong trước khi gọi `tmpnam()` lần nữa.

Một lần nữa, hàm này chỉ tạo tên file cho bạn. Việc tự `fopen()` file và dùng nó là tuỳ bạn.

Một lưu ý nữa: một số compiler cảnh báo không nên dùng `tmpnam()` vì có hệ thống có hàm tốt hơn (như hàm `mkstemp()` trên Unix). Bạn có thể xem tài liệu local để tìm tuỳ chọn tốt hơn. Tài liệu Linux thậm chí còn nói, "Đừng bao giờ dùng hàm này. Dùng `mkstemp()` thay thế."

Tuy nhiên, tôi sẽ làm kẻ khó chịu và không nói về `mkstemp()`[2] vì nó không nằm trong chuẩn mà tôi đang viết. Nyaah.

Macro `TMP_MAX` chứa số lượng tên file duy nhất có thể được `tmpnam()` sinh ra. Trớ trêu thay, đó là số *tối thiểu* các tên đó.

Giá trị trả về

Trả về con trỏ đến tên file tạm. Đây có thể là con trỏ đến chuỗi bạn truyền vào, hoặc con trỏ đến vùng nhớ tĩnh nội bộ nếu bạn truyền `NULL`. Khi lỗi (ví dụ không tìm được tên tạm duy nhất nào), `tmpnam()` trả về `NULL`.

Ví dụ

```
#include <stdio.h>

int main(void)
{
    char filename[L_tmpnam];
    char *another_filename;
```

[2] https://man.archlinux.org/man/mkstemp.3.en

```
    if (tmpnam(filename) != NULL)
        printf("We got a temp file name: \"%s\"\n", filename);
    else
        printf("Something went wrong, and we got nothing!\n");

    another_filename = tmpnam(NULL);

    printf("We got another temp file name: \"%s\"\n", another_filename);
    printf("And we didn't error check it because we're too lazy!\n");
}
```

Trên hệ Linux của tôi, chương trình này cho ra output sau:

```
We got a temp file name: "/tmp/filew9PMuZ"
We got another temp file name: "/tmp/fileOwrgPO"
And we didn't error check it because we're too lazy!
```

Xem thêm

`fopen()`, `tmpfile()`

23.5 `fclose()`

Ngược với `fopen()` —đóng một file khi bạn xong với nó để giải phóng tài nguyên hệ thống

Synopsis

```
#include <stdio.h>

int fclose(FILE *stream);
```

Mô tả

Khi bạn mở một file, hệ thống dành ra một ít tài nguyên để duy trì thông tin về file đang mở đó. Thường chỉ có thể mở đến một giới hạn số file cùng lúc. Dù sao, Việc Đúng Đắn là đóng file khi bạn xong dùng để tài nguyên hệ thống được giải phóng.

Ngoài ra, có thể bạn sẽ thấy rằng không phải mọi thông tin bạn đã ghi vào file đã thực sự được ghi xuống đĩa cho đến khi file được đóng. (Bạn có thể ép chuyện này bằng cách gọi `fflush()`.)

Khi chương trình của bạn thoát bình thường, nó sẽ đóng tất cả file đang mở hộ bạn. Nhiều khi, bạn có chương trình chạy dài, và tốt hơn là đóng file trước đó. Dù sao, không đóng file bạn đã mở khiến bạn trông tệ hại. Vậy nên nhớ `fclose()` file khi bạn xong dùng nó!

Giá trị trả về

Khi thành công, trả về `0`. Thường không ai kiểm tra cái này. Khi lỗi trả về `EOF`. Thường cũng không ai kiểm tra.

Ví dụ

```
#include <stdio.h>

int main(void)
{
    FILE *fp;

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

    if (fp == NULL) {
        printf("Error opening file\n");
    } else {
        printf("Opened file just fine!\n");
        fclose(fp);  // Xong hết!
    }
}
```

Xem thêm

`fopen()`

23.6 `fflush()`

Xử lý toàn bộ I/O có đệm của một stream ngay bây giờ

Synopsis

```
#include <stdio.h>

int fflush(FILE *stream);
```

Mô tả

Khi bạn làm I/O chuẩn, như đã nhắc trong phần về hàm `setvbuf()`, dữ liệu thường được lưu trong buffer cho đến khi nhập xong một dòng, hoặc buffer đầy, hoặc file được đóng. Đôi khi, bạn thực sự muốn output xảy ra *ngay giây này*, không chờ trong buffer. Bạn có thể ép điều đó bằng cách gọi `fflush()`.

Lợi ích của việc buffered (có đệm) là OS không cần chạm đĩa mỗi lần bạn gọi `fprintf()`. Nhược điểm là nếu bạn nhìn vào file trên đĩa sau khi gọi `fprintf()`, có thể nó vẫn chưa được ghi ra thực sự. ("Tôi gọi `fputs()`, nhưng file vẫn dài 0 byte! Sao thế?!") Trong hầu hết mọi tình huống, lợi ích của đệm lớn hơn nhược điểm; với những tình huống còn lại, dùng `fflush()`.

Lưu ý rằng theo spec, `fflush()` chỉ được thiết kế để dùng trên output stream. Chuyện gì xảy ra nếu bạn thử nó trên input stream? Dùng giọng rùng rợn nào: *có trờiiiii mà biếếếết!*

Giá trị trả về

Khi thành công, `fflush()` trả về 0. Nếu có lỗi, nó trả về `EOF` và đặt điều kiện lỗi cho stream (xem `ferror()`).

Ví dụ

Trong ví dụ này, chúng ta sẽ dùng carriage return, `'\r'`. Nó giống newline (xuống dòng) (`'\n'`), chỉ khác là không chuyển sang dòng kế. Nó chỉ quay về đầu dòng hiện tại.

Điều chúng ta định làm là một thanh trạng thái văn bản nho nhỏ như nhiều chương trình dòng lệnh vẫn dùng. Nó sẽ đếm ngược từ 10 xuống 0, in đè lên cùng một dòng.

Điểm mấu chốt là gì, và nó liên quan tới `fflush()` ra sao? Mấu chốt là terminal rất có khả năng đang "line buffered" (đệm theo dòng) (xem mục `setvbuf()` để biết thêm), nghĩa là nó sẽ không hiển thị gì cho đến khi in ra newline. Nhưng chúng ta không in newline; chỉ in carriage return, nên cần một cách để ép output xảy ra dù vẫn đang ở cùng một dòng. Đúng vậy, `fflush()!`

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

void sleep_seconds(int s)
{
    thrd_sleep(&(struct timespec){.tv_sec=s}, NULL);
}

int main(void)
{
    int count;

    for(count = 10; count >= 0; count--) {
        printf("\rSeconds until launch: ");  // mở đâù bằng CR
        if (count > 0)
            printf("%2d", count);
        else
            printf("blastoff!\n");

        // ép output ngay!!
        fflush(stdout);

        sleep_seconds(1);
    }
}
```

Xem thêm

`setbuf()`, `setvbuf()`

23.7 `fopen()`

Mở một file để đọc hoặc ghi

Synopsis

```
#include <stdio.h>

FILE *fopen(const char *path, const char *mode);
```

Mô tả

`fopen()` mở một file để đọc hoặc ghi.

Tham số `path` có thể là đường dẫn tương đối hoặc đường dẫn đầy đủ kèm tên file.

Tham số `mode` báo cho `fopen()` cách mở file (đọc, ghi, hay cả hai) và có phải file nhị phân không. Các mode khả dụng:

Mode	Mô tả
`r`	Mở file để đọc (chỉ đọc).
`w`	Mở file để ghi (chỉ ghi). File sẽ được tạo nếu chưa tồn tại.
`r+`	Mở file để đọc và ghi. File phải đã tồn tại.
`w+`	Mở file để ghi và đọc. File sẽ được tạo nếu chưa tồn tại.
`a`	Mở file để append (ghi thêm vào cuối). Giống mở để ghi, nhưng đặt file pointer ở cuối file, nên lần ghi kế sẽ nối vào cuối. File sẽ được tạo nếu chưa tồn tại.
`a+`	Mở file để đọc và append. File sẽ được tạo nếu chưa tồn tại.

Bất kỳ mode nào cũng có thể thêm chữ "`b`" ở cuối, như "`wb`" ("write binary"), để báo rằng file đó là file *binary* (chế độ nhị phân). ("Binary" ở đây thường nghĩa là file chứa các ký tự phi chữ số trông giống rác đối với mắt người). Nhiều hệ thống (như Unix) không phân biệt giữa file nhị phân và không nhị phân, nên "`b`" là thừa. Nhưng nếu dữ liệu là binary, thêm "`b`" cũng không hại, và có thể giúp người khác khi port code của bạn sang hệ thống khác.

Macro `FOPEN_MAX` cho biết (ít nhất) có thể mở bao nhiêu stream cùng lúc.

Macro `FILENAME_MAX` cho biết tên file hợp lệ có thể dài nhất bao nhiêu. Đừng làm quá nhé.

Giá trị trả về

`fopen()` trả về một `FILE*` có thể dùng trong các lời gọi liên quan tới file sau đó.

Nếu có chuyện gì không ổn (ví dụ bạn thử mở để đọc một file không tồn tại), `fopen()` sẽ trả về `NULL`.

Ví dụ

```
#include <stdio.h>

int main(void)
{
    FILE *fp;

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

    if (fp == NULL) {
        printf("Error opening file\n");
    } else {
        printf("Opened file just fine!\n");
        fclose(fp);  // Xong hết!
    }
}
```

Xem thêm

`fclose()`, `freopen()`

23.8 `freopen()`

Mở lại một `FILE*` đã có, gắn nó với một đường dẫn mới

Synopsis

```
#include <stdio.h>

FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

Mô tả

Giả sử bạn có một `FILE*` stream đã mở, nhưng đột nhiên muốn nó dùng một file khác thay vì file đang dùng. Bạn có thể dùng `freopen()` để "re-open" stream với một file mới.

Vì sao trên đời bạn lại muốn làm thế? Lý do phổ biến nhất là nếu bạn có chương trình thường đọc từ `stdin`, nhưng bạn muốn nó đọc từ một file. Thay vì đổi mọi `scanf()` sang `fscanf()`, bạn có thể đơn giản reopen `stdin` trên file mà bạn muốn đọc.

Một cách dùng khác được một số hệ thống cho phép là truyền `NULL` cho `filename` và chỉ định `mode` mới cho `stream`. Vậy bạn có thể đổi file từ "`r+`" (đọc và ghi) thành chỉ "`r`" (đọc), chẳng hạn. Việc mode nào có thể đổi là tuỳ cài đặt.

Khi bạn gọi `freopen()`, `stream` cũ sẽ bị đóng. Ngoài ra, hàm hoạt động y hệt `fopen()` chuẩn.

Giá trị trả về

`freopen()` trả về `stream` nếu mọi thứ ổn.

Nếu có chuyện gì không ổn (ví dụ bạn thử mở để đọc một file không tồn tại), `freopen()` sẽ trả về `NULL`.

Ví dụ

```
#include <stdio.h>

int main(void)
{
    int i, i2;

    scanf("%d", &i); // đọc i từ stdin

    // giờ đổi stdin để trỏ đến file thay vì bàn phím
    freopen("someints.txt", "r", stdin);

    scanf("%d", &i2); // lần này đọc từ file "someints.txt"

    printf("Hello, world!\n"); // in ra màn hình

    // đổi stdout để đi ra file thay vì terminal:
    freopen("output.txt", "w", stdout);

    printf("This goes to the file \"output.txt\"\n");

    // được phép trên một số hệ thống--có thể đổi mode của file:
    freopen(NULL, "wb", stdout); // đổi sang "wb" thay vì "w"
}
```

Xem thêm

`fclose()`, `fopen()`

23.9 `setbuf()`, `setvbuf()`

Cấu hình đệm cho các thao tác I/O chuẩn

Synopsis

```
#include <stdio.h>

void setbuf(FILE *stream, char *buf);

int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

Mô tả

Bình tĩnh đi vì điều sắp nói có thể hơi bất ngờ: khi bạn `printf()` hoặc `fprintf()` hoặc dùng bất kỳ hàm I/O kiểu như thế, *thường nó không chạy ngay lập tức.* Vì hiệu năng, và để làm bạn bực mình, I/O trên một `FILE*` stream được buffered (có đệm) cất an toàn cho đến khi thoả mãn điều kiện nhất định, và chỉ lúc đó I/O thực sự mới được thực hiện. Các hàm `setbuf()` và `setvbuf()` cho phép bạn đổi những điều kiện đó và hành vi đệm.

Vậy có những kiểu hành vi đệm nào? Lớn nhất là "full buffering" (đệm đầy đủ), khi đó toàn bộ I/O được lưu trong buffer lớn đến khi đầy, rồi mới đổ ra đĩa (hoặc bất kỳ file nào). Kế tiếp là "line buffering" (đệm theo dòng); với line buffering, I/O được gom từng dòng một (đến khi gặp ký tự newline (`'\n'`)) rồi dòng đó mới được xử lý. Cuối cùng là "unbuffered" (không đệm), nghĩa là I/O được xử lý ngay lập tức với mỗi lời gọi I/O chuẩn.

Có thể bạn từng thấy và thắc mắc vì sao gọi `putchar()` hết lần này đến lần khác mà không thấy output nào cho đến khi gọi `putchar('\n')`; đúng rồi— `stdout` là line-buffered!

Vì `setbuf()` chỉ là phiên bản đơn giản hoá của `setvbuf()`, chúng ta sẽ nói về `setvbuf()` trước.

`stream` là `FILE*` bạn muốn chỉnh. Chuẩn nói rằng bạn *phải* gọi `setvbuf()` *trước* khi bất kỳ thao tác I/O nào được thực hiện trên stream, không thì đến lúc đó đã quá muộn.

Tham số tiếp theo, `buf`, cho phép bạn tự làm vùng buffer (dùng `malloc()` hoặc chỉ một mảng `char`) để dùng cho đệm. Nếu không quan tâm, cứ đặt `buf` là `NULL`.

Giờ đến phần quan trọng của hàm: `mode` cho phép bạn chọn loại đệm muốn dùng trên `stream` này. Đặt là một trong các giá trị sau:

Mode	Mô tả
`_IOFBF`	`stream` sẽ được đệm đầy đủ (full buffered).
`_IOLBF`	`stream` sẽ được đệm theo dòng (line buffered).
`_IONBF`	`stream` sẽ không có đệm (unbuffered).

Cuối cùng, tham số `size` là kích thước mảng bạn truyền vào cho `buf` ...trừ khi bạn truyền `NULL` cho `buf`, trong trường hợp đó nó sẽ resize buffer hiện có thành kích thước bạn chỉ định.

Còn hàm "hạng nhỏ" `setbuf()` thì sao? Nó chỉ như gọi `setvbuf()` với các tham số nhất định, chỉ khác là `setbuf()` không trả về giá trị. Ví dụ sau cho thấy tính tương đương:

```
// hai dòng này tương đương:
setbuf(stream, buf);
setvbuf(stream, buf, _IOFBF, BUFSIZ); // đệm đầy đủ

// và hai dòng này tương đương:
setbuf(stream, NULL);
setvbuf(stream, NULL, _IONBF, BUFSIZ); // không đệm
```

Giá trị trả về

`setvbuf()` trả về 0 khi thành công, và khác 0 khi thất bại. `setbuf()` không có giá trị trả về.

Ví dụ

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    char lineBuf[1024];

    fp = fopen("somefile.txt", "w");
    setvbuf(fp, lineBuf, _IOLBF, 1024);  // đặt đệm theo dòng
    fprintf(fp, "You won't see this in the file yet. ");
    fprintf(fp, "But now you will because of this newline.\n");
    fclose(fp);

    fp = fopen("anotherfile.txt", "w");
    setbuf(fp, NULL); // đặt không đệm
    fprintf(fp, "You will see this in the file now.");
    fclose(fp);
}
```

Xem thêm

`fflush()`

23.10 `printf()`, `fprintf()`, `sprintf()`, `snprintf()`

In một chuỗi có định dạng ra console hoặc ra file

Synopsis

```
#include <stdio.h>

int printf(const char *format, ...);

int fprintf(FILE *stream, const char *format, ...);

int sprintf(char * restrict s, const char * restrict format, ...);

int snprintf(char * restrict s, size_t n, const char * restrict format, ...);
```

Mô tả

Các hàm này in output có định dạng ra nhiều đích khác nhau.

Hàm	Đích output
`printf()`	In ra console (mặc định thường là màn hình).
`fprintf()`	In ra một file.
`sprintf()`	In ra một chuỗi.
`snprintf()`	In ra một chuỗi (an toàn).

Khác biệt duy nhất giữa chúng là các tham số đứng trước chuỗi `format` mà bạn truyền vào.

Hàm	Thứ bạn truyền trước `format`
`printf()`	Không có gì đứng trước `format`.
`fprintf()`	Truyền vào `FILE*`.
`sprintf()`	Truyền vào `char*` đến buffer để in vào.
`snprintf()`	Truyền vào `char*` đến buffer và độ dài buffer tối đa.

Hàm `printf()` là huyền thoại về sự linh hoạt, một trong những hệ thống xuất có khả năng tùy biến nhất từng được nghĩ ra. Nó cũng có thể hơi ma quái đôi chỗ, rõ nhất là ở chuỗi `format`. Ta sẽ đi từng bước.

Cách dễ nhất nhìn vào chuỗi format là: nó sẽ in mọi thứ trong chuỗi như nguyên bản, *trừ khi* một ký tự có dấu phần trăm (`%`) đứng trước. Đó là lúc phép thuật xảy ra: tham số kế tiếp trong danh sách tham số của `printf()` sẽ được in theo cách mô tả bởi mã phần trăm. Những mã phần trăm này gọi là *format specifiers* (bộ chỉ định định dạng).

Đây là những format specifier phổ biến nhất.

Specifier	Mô tả
`%d`	In tham số kế tiếp dưới dạng số thập phân có dấu, như `3490`. Tham số in theo cách này phải là `int`, hoặc thứ được promote lên `int`.
`%f`	In tham số kế tiếp dưới dạng số thực có dấu, như `3.14159`. Tham số in theo cách này phải là `double`, hoặc thứ được promote lên `double`.
`%c`	In tham số kế tiếp như một ký tự, như `'B'`. Tham số in theo cách này phải là biến thể của `char`.
`%s`	In tham số kế tiếp như một chuỗi, như `"Did you remember your mittens?"`. Tham số in theo cách này phải là `char*` hoặc `char[]`.
`%%`	Không có tham số nào được chuyển đổi, in một dấu phần trăm bình thường. Đây là cách in ký tự '%' bằng `printf()`.

Đó là phần cơ bản. Tôi sẽ cho bạn thêm format specifier lát nữa, nhưng trước hết hãy lấy thêm bề rộng. Thực ra còn rất nhiều thứ bạn có thể chỉ định sau dấu phần trăm.

Một điều, bạn có thể đặt field width (độ rộng trường) trong đó—đây là con số cho `printf()` biết để bao nhiêu khoảng trắng vào một bên hoặc bên kia của giá trị bạn đang in. Giúp bạn xếp thành cột đẹp đẽ. Nếu số này âm, kết quả được canh trái thay vì canh phải. Ví dụ:

```
printf("%10d", x);  /* in X ở phía phải của trường 10 khoảng trắng */
printf("%-10d", x); /* in X ở phía trái của trường 10 khoảng trắng */
```

Nếu bạn không biết trước field width, bạn có thể dùng kung-foo nhỏ để lấy nó từ danh sách tham số ngay trước tham số cần in. Làm bằng cách đặt ghế và bàn ăn ở vị trí thẳng đứng. Dây an toàn được cài

bằng cách—ho khụ. Có vẻ tôi đã bay quá nhiều gần đây. Bỏ qua sự thật vô dụng đó hoàn toàn, bạn có thể chỉ định field width động bằng cách đặt `*` thay cho width. Nếu bạn không sẵn sàng hoặc không thể thực hiện tác vụ này, vui lòng báo tiếp viên và chúng tôi sẽ xếp lại chỗ cho bạn.

```
int width = 12;
int value = 3490;

printf("%*d\n", width, value);
```

Bạn cũng có thể đặt "0" trước con số nếu muốn nó được pad bằng số 0:

```
int x = 17;
printf("%05d", x);  /* "00017" */
```

Với số thực, bạn cũng có thể chỉ định muốn in bao nhiêu chữ số thập phân bằng một field width dạng "`x.y`" trong đó `x` là field width (bạn có thể bỏ qua nếu muốn nó chỉ đủ rộng) và `y` là số chữ số sau dấu thập phân cần in:

```
float f = 3.1415926535;

printf("%.2f", f);  /* "3.14" */
printf("%7.3f", f); /* "  3.141" <-- 7 khoảng trắng ngang qua */
```

Ok, các phần trên chắc chắn là cách dùng `printf()` phổ biến nhất, nhưng hãy lấy *full coverage* nào.

23.10.0.1 Bố cục Format Specifier

Về mặt kỹ thuật, bố cục của format specifier gồm các phần sau theo thứ tự này:

1. `%`, theo sau bởi...
2. Tuỳ chọn: không hoặc nhiều flag, canh trái, pad bằng số 0, v.v.
3. Tuỳ chọn: Field width, độ rộng của trường output.
4. Tuỳ chọn: Precision (độ chính xác), hay bao nhiêu chữ số thập phân cần in.
5. Tuỳ chọn: Length modifier, cho việc in những thứ lớn hơn `int` hoặc `double`.
6. Conversion specifier (chỉ định chuyển đổi), như `d`, `f`, v.v.

Tóm lại, toàn bộ format specifier được xếp như thế này:

```
%[flags][fieldwidth][.precision][lengthmodifier]conversionspecifier
```

Còn gì dễ hơn?

23.10.0.2 Conversion Specifier

Hãy nói về conversion specifier trước. Mỗi cái sau đây chỉ định kiểu có thể in, nhưng cũng có thể in bất cứ thứ gì được promote lên kiểu đó. Ví dụ, `%d` có thể in `int`, `short`, và `char`.

Specifier nhị phân là mới trong C23!

Conversion Specifier	Mô tả
`d`	In tham số `int` dưới dạng số thập phân.
`i`	Giống hệt `d`.
`b`	In `unsigned int` dưới dạng nhị phân (cơ số 2).
`B`	Giống hệt `b`, trừ dạng thay thế (alternate form), xem dưới.
`o`	In `unsigned int` dưới dạng bát phân (cơ số 8).
`u`	In `unsigned int` dưới dạng thập phân.
`x`	In `unsigned int` dưới dạng hex với chữ cái thường.

Conversion Specifier	Mô tả
`X`	In `unsigned int` dưới dạng hex với chữ cái hoa; cũng chú ý dạng thay thế, xem dưới.
`f`	In `double` dưới dạng thập phân. Vô cực được in là `infinity` hoặc `inf`, và NaN được in là `nan`, bất kỳ cái nào cũng có thể có dấu trừ đứng đầu.
`F`	Giống `f`, nhưng in `INFINITY`, `INF`, hoặc `NAN` toàn chữ hoa.
`e`	In số theo ký hiệu khoa học, ví dụ `1.234e56`. Xử lý vô cực và NaN như `f`.
`E`	Giống `e`, nhưng in số mũ `E` (và vô cực và NaN) bằng chữ hoa.
`g`	In số nhỏ như `f` và số lớn như `e`. Xem lưu ý dưới.
`G`	In số nhỏ như `F` và số lớn như `E`. Xem lưu ý dưới.
`a`	In `double` dưới dạng hex `0xh.hhhhpd` trong đó `h` là chữ số hex thường và `d` là số mũ thập phân của 2. Vô cực và NaN dạng như `f`. Xem thêm bên dưới.
`A`	Giống `a` nhưng mọi thứ viết hoa.
`c`	Chuyển tham số `int` sang `unsigned char` và in dưới dạng ký tự.
`s`	In chuỗi bắt đầu từ `char*` đã cho.
`p`	In một `void*` dưới dạng số, chắc là địa chỉ số, có thể ở dạng hex.
`n`	Lưu số ký tự đã ghi tính đến giờ vào `int*` đã cho. Không in gì. Xem dưới.
`%`	In dấu phần trăm literal.

23.10.0.2.1 Lưu ý về `%a` và `%A` Khi in số thực dạng hex, có một chữ số trước dấu thập phân, và phần còn lại tính đến precision.

```
double pi = 3.14159265358979;

printf("%.3a\n", pi);  // 0x1.922p+1
```

C có thể chọn chữ số đầu sao cho các chữ số tiếp theo canh theo ranh giới 4-bit.

Nếu precision bị bỏ qua và macro `FLT_RADIX` là luỹ thừa của 2, đủ precision sẽ được dùng để biểu diễn số chính xác. Nếu `FLT_RADIX` không phải luỹ thừa của 2, đủ precision được dùng để có thể phân biệt bất kỳ hai giá trị floating nào.

Nếu precision là `0` và flag `#` không được chỉ định, dấu thập phân được bỏ đi.

23.10.0.2.2 Lưu ý về `%g` và `%G` Ý nghĩa chung là dùng ký hiệu khoa học khi số trở nên quá "extreme", và dùng ký hiệu thập phân thường trong các trường hợp khác.

Hành vi chính xác về việc in như `%f` hay `%e` phụ thuộc vào vài yếu tố:

Nếu số mũ lớn hơn hoặc bằng -4 **và** precision lớn hơn số mũ, chúng ta dùng `%f`. Trong trường hợp này, precision được chuyển theo $p = p - (x + 1)$, trong đó p là precision đã chỉ định và x là số mũ.

Ngược lại chúng ta dùng `%e`, và precision trở thành $p - 1$.

Các số 0 ở cuối phần thập phân bị xoá. Và nếu không còn cái nào, dấu thập phân cũng bị xoá luôn. Tất cả những điều này trừ khi flag `#` được chỉ định.

23.10.0.2.3 Lưu ý về `%n` Specifier này ngầu và khác biệt, và hiếm khi cần. Nó thực ra không in gì, nhưng lưu số ký tự đã in tính đến lúc đó vào tham số pointer (con trỏ) kế tiếp trong danh sách.

```
int numChars;
float a = 3.14159;
int b = 3490;

printf("%f %d%n\n", a, b, &numChars);
printf("The above line contains %d characters.\n", numChars);
```

Ví dụ trên sẽ in các giá trị `a` và `b`, rồi lưu số ký tự đã in đến lúc đó vào biến `numChars`. Lời gọi `printf()` kế in kết quả đó.

```
3.141590 3490
The above line contains 13 characters
```

23.10.0.3 Length Modifier

Bạn có thể dán *length* modifier trước mỗi conversion specifier, nếu muốn. Phần lớn các format specifier hoạt động với kiểu `int` hoặc `double`, nhưng nếu bạn muốn kiểu lớn hơn hay nhỏ hơn thì sao? Đó là lúc mấy cái này có ích.

Ví dụ, bạn có thể in `long long int` với modifier `ll`:

```
long long int x = 3490;

printf("%lld\n", x);  // 3490
```

Length Modifier	Conversion Specifier	Mô tả
`hh`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Chuyển tham số sang `char` (signed hoặc unsigned tuỳ ngữ cảnh) trước khi in.
`h`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Chuyển tham số sang `short int` (signed hoặc unsigned tuỳ ngữ cảnh) trước khi in.
`l`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Tham số là `long int` (signed hoặc unsigned tuỳ ngữ cảnh).
`ll`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Tham số là `long long int` (signed hoặc unsigned tuỳ ngữ cảnh).
`j`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Tham số là `intmax_t` hoặc `uintmax_t` (tuỳ ngữ cảnh).
`z`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Tham số là `size_t`.
`t`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Tham số là `ptrdiff_t`.
`L`	`a`, `A`, `e`, `E`, `f`, `F`, `g`, `G`	Tham số là `long double`.
`l`	`c`	Tham số nằm trong `wint_t`, một wide character (ký tự rộng).
`l`	`s`	Tham số nằm trong `wchar_t*`, một chuỗi wide character.
`hh`	`n`	Lưu kết quả vào tham số `signed char*`.
`h`	`n`	Lưu kết quả vào tham số `short int*`.
`l`	`n`	Lưu kết quả vào tham số `long int*`.
`ll`	`n`	Lưu kết quả vào tham số `long long int*`.
`j`	`n`	Lưu kết quả vào tham số `intmax_t*`.
`z`	`n`	Lưu kết quả vào tham số `size_t*`.
`t`	`n`	Lưu kết quả vào tham số `ptrdiff_t*`.

23.10.0.4 Precision

Trước length modifier, bạn có thể đặt precision (độ chính xác), mà thường có nghĩa là bạn muốn bao nhiêu chữ số thập phân cho số thực.

Để làm điều này, bạn đặt dấu chấm (`.`) và số chữ số thập phân sau đó.

Ví dụ, chúng ta có thể in π làm tròn hai chữ số thập phân như sau:

```
double pi = 3.14159265358979;

printf("%.2f\n", pi);  // 3.14
```

Conversion Specifier	Ý nghĩa giá trị Precision
b, d, i, o, u, x, X	Với kiểu số nguyên, số chữ số tối thiểu (sẽ pad bằng số 0 đầu)
a, e, f, A, E, F	Với kiểu số thực, precision là số chữ số sau dấu thập phân.
g, G	Với kiểu số thực, precision là số chữ số có nghĩa được in.
s	Số byte tối đa (không phải ký tự multibyte (đa byte)!) được ghi.

Nếu không có số nào sau dấu thập phân trong precision, precision là 0.

Nếu `*` được chỉ định sau dấu thập phân, điều kỳ diệu xảy ra! Nó có nghĩa là tham số `int` truyền vào `printf()` ngay trước số cần in chứa precision. Bạn có thể dùng cái này nếu không biết precision lúc compile.

```
int precision;
double pi = 3.14159265358979;

printf("Enter precision: "); fflush(stdout);
scanf("%d", &precision);

printf("%.*f\n", precision, pi);
```

Kết quả:

```
Enter precision: 4
3.1416
```

23.10.0.5 Field Width

Trước precision tuỳ chọn, bạn có thể chỉ định field width (độ rộng trường). Đây là số thập phân chỉ ra vùng in tham số nên rộng bao nhiêu. Vùng đó sẽ được pad bằng khoảng trắng đầu (hoặc cuối) để đảm bảo đủ rộng.

Nếu field width chỉ định quá nhỏ để chứa output, nó bị bỏ qua.

Xem trước, bạn có thể cho field width âm để canh hướng khác.

Hãy in một số trong trường rộng 10. Chúng ta sẽ đặt vài dấu ngoặc nhọn quanh nó để nhìn rõ khoảng trắng pad trong output.

```
printf("<<%10d>>\n", 3490);   // canh phải
printf("<<%-10d>>\n", 3490);  // canh trái
```

```
<<      3490>>
<<3490      >>
```

Giống precision, bạn có thể dùng dấu sao (`*`) cho field width

```
int field_width;
int val = 3490;

printf("Enter field_width: "); fflush(stdout);
scanf("%d", &field_width);
```

```
printf("<<%*d>>\n", field_width, val);
```

23.10.0.6 Flags

Trước field width, bạn có thể đặt vài flag tuỳ chọn để kiểm soát thêm output của các trường sau. Ta vừa thấy flag `-` có thể dùng để canh trái hoặc phải. Nhưng còn nhiều nữa!

Flag	Mô tả
`-`	Với field width, canh trái trong trường (mặc định là phải).
`+`	Nếu số có dấu, luôn đặt `+` hoặc `-` ở đầu.
[SPACE]	Nếu số có dấu, đặt khoảng trắng cho số dương, hoặc `-` cho số âm.
`0`	Pad trường canh phải bằng số 0 đầu thay vì khoảng trắng đầu.
`#`	In theo dạng thay thế (alternate form). Xem bên dưới.

Ví dụ, chúng ta có thể pad một số hex với số 0 đầu đến field width 8 bằng:

```
printf("%08x\n", 0x1234);  // 00001234
```

Kết quả của "alternate form" `#` phụ thuộc vào conversion specifier.

Conversion Specifier	Ý nghĩa dạng thay thế (`#`)
`o`	Tăng precision của số khác không vừa đủ để có một số `0` đứng đầu số bát phân.
`b`	Thêm tiền tố `0b` cho số khác không.
`B`	Giống `b` , nhưng viết hoa `0B` .
`x`	Thêm tiền tố `0x` cho số khác không.
`X`	Giống `x` , nhưng viết hoa `0X` .
`a` , `e` , `f`	Luôn in dấu thập phân, ngay cả khi không có gì theo sau.
`A` , `E` , `F`	Giống hệt `a` , `e` , `f` .
`g` , `G`	Luôn in dấu thập phân, ngay cả khi không có gì theo sau, và giữ các số 0 cuối.

23.10.0.7 Chi tiết `sprintf()` và `snprintf()`

Cả `sprintf()` và `snprintf()` đều có tính chất là nếu bạn truyền `NULL` làm buffer, không gì được ghi—nhưng bạn vẫn có thể kiểm tra giá trị trả về để xem sẽ ghi được bao nhiêu ký tự *nếu được ghi*.

`snprintf()` **luôn** kết thúc chuỗi bằng ký tự `NUL` . Nên nếu bạn thử ghi nhiều hơn số ký tự tối đa chỉ định, vũ trụ sẽ kết thúc.

Đùa thôi. Nếu bạn làm thế, `snprintf()` sẽ ghi $n - 1$ ký tự để nó còn đủ chỗ ghi ký tự kết thúc ở cuối.

Giá trị trả về

Trả về số ký tự đã output, hoặc một số âm khi lỗi.

Ví dụ

```
#include <stdio.h>

int main(void)
{
    int a = 100;
    float b = 2.717;
    char *c = "beej!";
```

```
    char d = 'X';
    int e = 5;

    printf("%d\n", a); /* "100"      */
    printf("%f\n", b); /* "2.717000" */
    printf("%s\n", c); /* "beej!"    */
    printf("%c\n", d); /* "X"        */
    printf("110%%\n"); /* "110%"     */

    printf("%10d\n", a);   /* "       100" */
    printf("%-10d\n", a);  /* "100       " */
    printf("%*d\n", e, a); /* "  100"      */
    printf("%.2f\n", b);   /* "2.72"       */

    printf("%hhd\n", d); /* "88" <-- mã ASCII của 'X' */

    printf("%5d %5.2f %c\n", a, b, d); /* "  100  2.72 X" */
}
```

Xem thêm

`sprintf()`, `vprintf()`

23.11 `scanf()`, `fscanf()`, `sscanf()`

Đọc chuỗi, ký tự, hoặc dữ liệu số có định dạng từ console hoặc từ file

Synopsis

```
#include <stdio.h>

int scanf(const char *format, ...);

int fscanf(FILE *stream, const char *format, ...);

int sscanf(const char * restrict s, const char * restrict format, ...);
```

Mô tả

Các hàm này đọc output có định dạng từ nhiều nguồn khác nhau.

Hàm	Nguồn input
`scanf()`	Đọc từ console (mặc định thường là bàn phím).
`fscanf()`	Đọc từ file.
`sscanf()`	Đọc từ một chuỗi.

Khác biệt duy nhất giữa chúng là các tham số đứng trước chuỗi `format` mà bạn truyền vào.

Hàm	Thứ bạn truyền trước `format`
`scanf()`	Không có gì đứng trước `format`.
`fscanf()`	Truyền vào `FILE*`.

Hàm	Thứ bạn truyền trước `format`
`sscanf()`	Truyền vào `char*` đến buffer để đọc từ đó.

Họ hàm `scanf()` đọc dữ liệu từ console hoặc từ `FILE` stream, phân tích và lưu kết quả vào các biến bạn đưa vào trong danh sách tham số.

Chuỗi format rất giống của `printf()` ở chỗ bạn có thể bảo nó đọc một `"%d"`, chẳng hạn cho `int`. Nhưng nó cũng có thêm khả năng, nổi bật là nó có thể nuốt những ký tự khác trong input mà bạn chỉ định trong chuỗi format.

Nhưng hãy bắt đầu đơn giản, xem cách dùng cơ bản nhất trước khi đắm vào chiều sâu của hàm. Ta sẽ bắt đầu bằng cách đọc một `int` từ bàn phím:

```
int a;

scanf("%d", &a);
```

`scanf()` hiển nhiên cần một pointer đến biến nếu nó định thay đổi biến đó, nên ta dùng toán tử address-of để lấy pointer.

Trong trường hợp này, `scanf()` đi theo chuỗi format, tìm thấy "`%d`", rồi biết nó cần đọc một số nguyên và lưu vào biến kế tiếp trong danh sách tham số, `a`.

Đây là vài format specifier khác bạn có thể đặt trong chuỗi format:

Format Specifier	Mô tả
`%d`	Đọc một số nguyên để lưu vào `int`. Số này có thể có dấu.
`%u`	Đọc một số nguyên để lưu vào `unsigned int`.
`%f`	Đọc một số thực, để lưu vào `float`.
`%s`	Đọc một chuỗi cho đến ký tự whitespace (khoảng trắng) đầu tiên.
`%c`	Đọc một `char`.

Và đó là hết chuyện!

Haha! Đùa thôi. Nếu bạn vừa từ trang `printf()` qua, bạn biết còn gần như vô tận nội dung thêm.

23.11.0.1 Tiêu thụ ký tự khác

`scanf()` sẽ đi dọc chuỗi format khớp với bất kỳ ký tự nào bạn bỏ vào.

Ví dụ, bạn có thể đọc một ngày có gạch nối như thế này:

```
scanf("%u-%u-%u", &yyyy, &mm, &dd);
```

Trong trường hợp đó, `scanf()` sẽ cố tiêu thụ một số thập phân unsigned, rồi một dấu gạch, rồi một số unsigned nữa, rồi một dấu gạch nữa, rồi một số unsigned nữa.

Nếu tại bất kỳ điểm nào không khớp (ví dụ người dùng nhập "foo"), `scanf()` sẽ bỏ đi mà không tiêu thụ những ký tự lạ.

Và nó sẽ trả về số biến được chuyển đổi thành công. Trong ví dụ trên, nếu người dùng nhập chuỗi hợp lệ, `scanf()` sẽ trả về `3`, một cho mỗi biến được đọc thành công.

23.11.0.2 Vấn đề với `scanf()`

Tôi (và C FAQ, và nhiều người) khuyên *không* nên dùng `scanf()` để đọc trực tiếp từ bàn phím. Quá dễ để nó ngừng tiêu thụ ký tự khi người dùng nhập dữ liệu xấu.

Nếu bạn có dữ liệu trong file và tự tin nó ở tình trạng tốt, `fscanf()` có thể thực sự hữu dụng.

Nhưng trong trường hợp bàn phím hoặc file, bạn luôn có thể dùng `fgets()` để đọc một dòng đầy đủ vào buffer, rồi dùng `sscanf()` để tách mọi thứ ra khỏi buffer. Cách này cho bạn điểm tốt của cả hai bên.

23.11.0.3 Vấn đề với `sscanf()`

Cách đây không lâu, một lập trình viên bên thứ ba nổi tiếng vì nghĩ ra cách cắt thời gian load của *GTA Online* đi 70%[3].

Điều họ phát hiện là cài đặt của `sscanf()` đầu tiên sẽ gọi ngầm `strlen()` ... nên ngay cả khi bạn chỉ dùng `sscanf()` để tách vài ký tự đầu khỏi chuỗi, nó vẫn chạy tới tận cuối chuỗi trước.

Trên chuỗi ngắn, không vấn đề, nhưng trên chuỗi dài với lời gọi lặp lại (đúng cái xảy ra trong *GTA*) nó trở nên *chậmmmmmmm...*

Vậy nên nếu bạn chỉ chuyển chuỗi thành số, hãy cân nhắc `atoi()`, `atof()`, hoặc họ hàm `strtol()` và `strtod()`.

(Lập trình viên đó nhận được bug bounty $10.000 cho công sức đó.)

23.11.0.4 Chi tiết sâu

Hãy xem một `scanf()`

Và đây là vài mã nữa, nhưng những cái này không hay dùng thường xuyên. Đương nhiên bạn có thể dùng chúng nhiều như bạn muốn!

Đầu tiên, chuỗi format. Như đã nhắc, nó có thể chứa ký tự bình thường và các format specifier `%`. Và ký tự whitespace.

Ký tự whitespace có vai trò đặc biệt: một ký tự whitespace sẽ khiến `scanf()` tiêu thụ càng nhiều ký tự whitespace càng tốt cho đến ký tự non-whitespace tiếp theo. Bạn có thể dùng cái này để bỏ qua mọi whitespace đầu hoặc cuối.

Ngoài ra, tất cả format specifier trừ `s`, `c`, và `[` tự động tiêu thụ whitespace đầu.

Nhưng tôi biết bạn đang nghĩ gì: phần thịt của hàm này nằm trong các format specifier. Chúng trông thế nào?

Chúng gồm các phần sau, theo thứ tự:

1. Dấu `%`
2. Tuỳ chọn: một `*` để chặn việc gán—sẽ nói sau
3. Tuỳ chọn: field width—số ký tự tối đa để đọc
4. Tuỳ chọn: length modifier, để chỉ định kiểu dài hơn hoặc ngắn hơn
5. Conversion specifier, như `d` hay `f` chỉ ra kiểu cần đọc

23.11.0.5 Conversion Specifier

Hãy bắt đầu với cái hay nhất và cuối cùng: *conversion specifier*.

Đây là phần của format specifier cho ta biết `scanf()` nên đọc vào kiểu biến nào, như `%d` hay `%f`.

Chuyển đổi nhị phân là mới trong C23!

Conversion Specifier	Mô tả
`d`	Khớp `int` thập phân. Có thể có dấu đầu.
`b`	Khớp `unsigned int` nhị phân (cơ số 2). Có thể có dấu đầu.
`i`	Giống `d`, chỉ khác là xử lý được nếu bạn đặt `0x` (hex) hay `0` (bát phân) hay `0b` (nhị phân) đứng đầu số.
`o`	Khớp `unsigned int` bát phân (cơ số 8). Bỏ qua số 0 đầu.

[3] https://nee.lv/2021/02/28/How-I-cut-GTA-Online-loading-times-by-70/

Conversion Specifier	Mô tả
`u`	Khớp `unsigned int` thập phân.
`x`	Khớp `unsigned int` hex (cơ số 16).
`f`	Khớp số thực (hoặc ký hiệu khoa học, hoặc bất cứ thứ gì `strtod()` xử lý được).
`c`	Khớp một `char`, hoặc nhiều `char` nếu có field width.
`s`	Khớp một chuỗi `char` non-whitespace.
`[`	Khớp một chuỗi ký tự từ một tập. Tập kết thúc bằng `]`. Xem thêm bên dưới.
`p`	Khớp một pointer, ngược lại với `%p` của `printf()`.
`n`	Lưu số ký tự đã ghi tính đến lúc đó vào `int*` đã cho. Không tiêu thụ gì.
`%`	Khớp dấu phần trăm literal.

Tất cả những cái sau đều tương đương với specifier `f`: `a`, `e`, `g`, `A`, `E`, `F`, `G`.

Và `X` hoa tương đương với `x` thường.

23.11.0.5.1 Conversion Specifier Scanset `%[]`

Đây chắc là format specifier kỳ quặc nhất. Nó cho phép bạn chỉ định một tập ký tự (*scanset*) để được lưu (khả năng cao vào mảng `char`). Việc chuyển đổi dừng khi gặp ký tự không thuộc tập.

Ví dụ, `%[0-9]` nghĩa là "khớp mọi số từ 0 đến 9". Và `%[AD-G34]` nghĩa là "khớp A, D đến G, 3, hoặc 4".

Giờ, để rối rắm thêm, bạn có thể bảo `scanf()` khớp các ký tự *không* nằm trong tập bằng cách đặt dấu caret (`^`) ngay sau `%[` và theo sau là tập, như: `%[^A-C]`, nghĩa là "khớp mọi ký tự *không* từ A đến C."

Để khớp dấu ngoặc vuông đóng, đặt nó làm ký tự đầu tiên trong tập, như: `%[]A-C]` hoặc `%[^]A-C]`. (Tôi thêm "`A-C`" để rõ rằng "`]`" đầu tiên trong tập.)

Để khớp dấu gạch nối, đặt nó làm ký tự cuối cùng trong tập, ví dụ để khớp A-đến-C hoặc gạch nối: `%[A-C-]`.

Vậy nếu chúng ta muốn khớp mọi chữ cái *trừ* "%", "^", "]", "B", "C", "D", "E", và "-", ta có thể dùng chuỗi format này: `%[^]%^B-E-]`.

Hiểu chưa? Giờ ta có thể qua hàm tiếp—khoan! Còn nữa! Vâng, vẫn còn nữa để biết về `scanf()`. Không bao giờ kết thúc à? Hãy thử tưởng tượng tôi cảm thấy thế nào khi viết về nó!

23.11.0.6 Length Modifier

Vậy bạn biết "`%d`" lưu vào `int`. Nhưng làm sao lưu vào `long`, `short`, hoặc `double`?

Chà, giống như trong `printf()`, bạn có thể thêm modifier trước type specifier để báo `scanf()` rằng bạn có kiểu dài hơn hoặc ngắn hơn. Sau đây là bảng các modifier khả dụng:

Length Modifier	Conversion Specifier	Mô tả
`hh`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Chuyển input sang `char` (signed hoặc unsigned tuỳ ngữ cảnh) trước khi in.
`h`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Chuyển input sang `short int` (signed hoặc unsigned tuỳ ngữ cảnh) trước khi in.
`l`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Chuyển input sang `long int` (signed hoặc unsigned tuỳ ngữ cảnh).
`ll`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Chuyển input sang `long long int` (signed hoặc unsigned tuỳ ngữ cảnh).
`j`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Chuyển input sang `intmax_t` hoặc `uintmax_t` (tuỳ ngữ cảnh).
`z`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Chuyển input sang `size_t`.

Length Modifier	Conversion Specifier	Mô tả
`t`	`b`, `d`, `i`, `o`, `u`, `x`, `X`	Chuyển input sang `ptrdiff_t`.
`L`	`a`, `A`, `e`, `E`, `f`, `F`, `g`, `G`	Chuyển input sang `long double`.
`l`	`c`, `s`, `[`	Chuyển input sang `wchar_t`, một wide character.
`l`	`s`	Tham số nằm trong `wchar_t*`, một chuỗi wide character.
`hh`	`n`	Lưu kết quả vào tham số `signed char*`.
`h`	`n`	Lưu kết quả vào tham số `short int*`.
`l`	`n`	Lưu kết quả vào tham số `long int*`.
`ll`	`n`	Lưu kết quả vào tham số `long long int*`.
`j`	`n`	Lưu kết quả vào tham số `intmax_t*`.
`z`	`n`	Lưu kết quả vào tham số `size_t*`.
`t`	`n`	Lưu kết quả vào tham số `ptrdiff_t*`.

23.11.0.7 Field Widths

Field width nói chung cho phép bạn chỉ định số ký tự tối đa được tiêu thụ. Nếu thứ bạn cố khớp ngắn hơn field width, input đó sẽ ngừng được xử lý trước khi đạt field width.

Vậy một chuỗi sẽ ngừng tiêu thụ khi tìm thấy whitespace, ngay cả khi khớp ít hơn field width ký tự.

Và một số float sẽ ngừng tiêu thụ ở cuối số, ngay cả khi ít ký tự hơn field width được khớp.

Nhưng `%c` thì thú vị—nó không ngừng tiêu thụ ký tự với bất cứ điều gì. Nên nó sẽ đi đúng đến field width. (Hoặc 1 ký tự nếu không có field width.)

23.11.0.8 Bỏ qua Input với `*`

Nếu bạn đặt `*` trong format specifier, nó báo `scanf()` thực hiện chuyển đổi đã chỉ định, nhưng không lưu vào đâu cả. Nó chỉ đơn giản vứt dữ liệu đi khi đọc. Đây là thứ bạn dùng nếu muốn `scanf()` ăn chút dữ liệu nhưng không muốn lưu vào đâu; bạn không truyền tham số cho `scanf()` cho chuyển đổi này.

```
// Đọc 3 int, nhưng bỏ cái ở giữa
scanf("%d %*d %d", &int1, &int3);
```

Giá trị trả về

`scanf()` trả về số item được gán vào biến. Vì việc gán vào biến dừng khi gặp input không hợp lệ cho một format specifier nào đó, nó có thể cho bạn biết đã nhập đủ dữ liệu đúng chưa.

Ngoài ra, `scanf()` trả về `EOF` khi gặp end-of-file.

Ví dụ

```
#include <stdio.h>

int main(void)
{
    int a;
    long int b;
    unsigned int c;
    float d;
    double e;
```

```
    long double f;
    char s[100];

    scanf("%d", &a);  // lưu một int
    scanf(" %d", &a); // ăn whitespace, rồi lưu một int
    scanf("%s", s); // lưu một chuỗi
    scanf("%Lf", &f); // lưu một long double

    // lưu một unsigned, đọc whitespace, rồi lưu một long int:
    scanf("%u %ld", &c, &b);

    // lưu một int, đọc whitespace, đọc "blendo", đọc whitespace,
    // rồi lưu một float:
    scanf("%d blendo %f", &a, &d);

    // đọc whitespace, rồi lưu mọi ký tự cho đến newline
    scanf(" %[^\n]", s);

    // lưu một float, đọc (và bỏ) một int, rồi lưu một double:
    scanf("%f %*d %lf", &d, &e);

    // lưu 10 ký tự:
    scanf("%10c", s);
}
```

Xem thêm

`sscanf()`, `vscanf()`, `vsscanf()`, `vfscanf()`

23.12 `vprintf()`, `vfprintf()`, `vsprintf()`, `vsnprintf()`

Biến thể `printf()` dùng danh sách tham số biến đổi (`va_list`)

Synopsis

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

int vprintf(const char * restrict format, va_list arg);

int vfprintf(FILE * restrict stream, const char * restrict format,
             va_list arg);

int vsprintf(char * restrict s, const char * restrict format, va_list arg);

int vsnprintf(char * restrict s, size_t n, const char * restrict format,
              va_list arg);
```

Mô tả

Các hàm này giống biến thể `printf()` chỉ khác là thay vì nhận số tham số biến đổi thực, chúng nhận một số cố định—cuối cùng là một `va_list` tham chiếu đến các tham số biến đổi.

Giống với `printf()`, các biến thể khác nhau gửi output đến những nơi khác nhau.

Hàm	Đích output
`vprintf()`	In ra console (mặc định thường là màn hình).
`vfprintf()`	In ra file.
`vsprintf()`	In ra chuỗi.
`vsnprintf()`	In ra chuỗi (an toàn).

Cả `vsprintf()` và `vsnprintf()` đều có tính chất là nếu bạn truyền `NULL` làm buffer, không gì được ghi—nhưng bạn vẫn có thể kiểm tra giá trị trả về để xem đã ghi được bao nhiêu ký tự *nếu được ghi.*

Nếu bạn thử ghi nhiều hơn số ký tự tối đa, `vsnprintf()` sẽ lịch sự chỉ ghi $n - 1$ ký tự để còn đủ chỗ cho ký tự kết thúc ở cuối.

Còn vì sao đời bạn lại muốn làm thế, lý do phổ biến nhất là để tạo phiên bản chuyên biệt riêng của các hàm kiểu `printf()`, ăn bám lên mọi điều tốt đẹp của `printf()`.

Xem ví dụ để có ví dụ, dự đoán được thôi.

Giá trị trả về

`vprintf()` và `vfprintf()` trả về số ký tự đã in, hoặc giá trị âm khi lỗi.

`vsprintf()` trả về số ký tự đã in vào buffer, không tính ký tự kết thúc NUL, hoặc giá trị âm nếu có lỗi.

`vnsprintf()` trả về số ký tự đã in vào buffer. Hoặc số *sẽ* được in nếu buffer đủ lớn.

Ví dụ

Trong ví dụ này, ta tạo phiên bản riêng của `printf()` gọi là `logger()`, có timestamp output. Chú ý các lời gọi `logger()` có đủ đồ chơi của `printf()`.

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

int logger(char *format, ...)
{
    va_list va;
    time_t now_secs = time(NULL);
    struct tm *now = gmtime(&now_secs);

    // Output timestamp dạng "YYYY-MM-DD hh:mm:ss : "
    printf("%04d-%02d-%02d %02d:%02d:%02d : ",
        now->tm_year + 1900, now->tm_mon + 1, now->tm_mday,
        now->tm_hour, now->tm_min, now->tm_sec);

    va_start(va, format);
    int result = vprintf(format, va);
    va_end(va);

    printf("\n");

    return result;
}

int main(void)
{
    int x = 12;
```

```
    float y = 3.2;

    logger("Hello!");
    logger("x = %d and y = %.2f", x, y);
}
```

Output:

```
2021-03-30 04:25:49 : Hello!
2021-03-30 04:25:49 : x = 12 and y = 3.20
```

Xem thêm

`printf()`

23.13 `vscanf()`, `vfscanf()`, `vsscanf()`

Biến thể `scanf()` dùng danh sách tham số biến đổi (`va_list`)

Synopsis

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

int vscanf(const char * restrict format, va_list arg);

int vfscanf(FILE * restrict stream, const char * restrict format,
            va_list arg);

int vsscanf(const char * restrict s, const char * restrict format,
            va_list arg);
```

Mô tả

Các hàm này giống biến thể `scanf()` chỉ khác là thay vì nhận số tham số biến đổi thực, chúng nhận một số cố định—cuối cùng là một `va_list` tham chiếu đến các tham số biến đổi.

Hàm	Nguồn input
`vscanf()`	Đọc từ console (mặc định thường là bàn phím).
`vfscanf()`	Đọc từ file.
`vsscanf()`	Đọc từ chuỗi.

Giống với họ hàm `vprintf()`, đây là cách tốt để thêm chức năng phụ tận dụng sức mạnh mà `scanf()` mang lại.

Giá trị trả về

Trả về số item scan thành công, hoặc `EOF` khi gặp end-of-file hoặc lỗi.

Ví dụ

Tôi phải thú nhận đã vắt óc để nghĩ khi nào bạn muốn dùng cái này. Ví dụ tốt nhất tôi tìm được là một cái trên Stack Overflow[4] kiểm tra lỗi giá trị trả về từ `scanf()` so với mong đợi. Một biến thể của nó được trình bày bên dưới.

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

int error_check_scanf(int expected_count, char *format, ...)
{
    va_list va;

    va_start(va, format);
    int count = vscanf(format, va);
    va_end(va);

    // Dòng này sẽ crash chương trình nếu điều kiện sai:
    assert(count == expected_count);

    return count;
}

int main(void)
{
    int a, b;
    float c;

    error_check_scanf(3, "%d, %d/%f", &a, &b, &c);
    error_check_scanf(2, "%d", &a);
}
```

Xem thêm

`scanf()`

23.14 `getc()`, `fgetc()`, `getchar()`

Lấy một ký tự đơn từ console hoặc từ file

Synopsis

```
#include <stdio.h>

int getc(FILE *stream);

int fgetc(FILE *stream);

int getchar(void);
```

[4] https://stackoverflow.com/questions/17017331/c99-vscanf-for-dummies/17018046#17018046

Mô tả

Tất cả các hàm này, theo cách này hay cách khác, đọc một ký tự đơn từ console hoặc từ một `FILE`. Khác biệt khá nhỏ, và đây là mô tả:

`getc()` trả về một ký tự từ `FILE` đã chỉ định. Về mặt sử dụng, nó tương đương với lời gọi `fgetc()` tương tự, và `fgetc()` thường gặp hơn. Chỉ khác ở cách cài đặt của hai hàm.

`fgetc()` trả về một ký tự từ `FILE` đã chỉ định. Về mặt sử dụng, nó tương đương với lời gọi `getc()` tương tự, chỉ khác là `fgetc()` thường gặp hơn. Chỉ khác ở cách cài đặt của hai hàm.

Đúng vậy, tôi đã gian lận và copy-paste đoạn trên.

`getchar()` trả về một ký tự từ `stdin`. Thực ra, nó giống gọi `getc(stdin)`.

Giá trị trả về

Cả ba hàm trả về `unsigned char` mà chúng đọc được, trừ việc nó được cast sang `int`.

Nếu gặp end-of-file hoặc lỗi, cả ba hàm đều trả về `EOF`.

Ví dụ

Ví dụ này đọc tất cả ký tự từ một file, chỉ output những chữ 'b' nó tìm thấy..

```
#include <stdio.h>

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

    fp = fopen("spoon.txt", "r"); // nhớ kiểm tra lỗi cái này!

    // câu while dưới gán vào c, rồi so với EOF:

    while((c = fgetc(fp)) != EOF) {
        if (c == 'b') {
            putchar(c);
        }
    }

    putchar('\n');

    fclose(fp);
}
```

Xem thêm

23.15 `gets()`, `fgets()`

Đọc một chuỗi từ console hoặc file

Synopsis

```
#include <stdio.h>

char *fgets(char *s, int size, FILE *stream);
char *gets(char *s);
```

Mô tả

Đây là những hàm lấy một chuỗi kết thúc bằng newline từ console hoặc một file. Nói cách khác bình thường, chúng đọc một dòng văn bản. Hành vi hơi khác, và vì thế, cách dùng cũng khác. Ví dụ, đây là cách dùng `gets()`:

Đừng dùng `gets()`. Thực tế, từ C11 nó đã bị loại bỏ! Đây là một trong những trường hợp hiếm hoi một hàm bị *gỡ* khỏi chuẩn.

Phải thừa nhận là có giải thích lý do sẽ hữu ích, đúng không? Thứ nhất, `gets()` không cho phép bạn chỉ định độ dài của buffer để lưu chuỗi. Điều này cho phép người dùng tiếp tục nhập dữ liệu qua khỏi đuôi buffer của bạn, và tin tôi đi, đó sẽ là Tin Xấu.

Và đó là công dụng của tham số `size` trong `fgets()`. `fgets()` sẽ đọc tối đa `size-1` ký tự rồi đặt một terminator `NUL` sau đó.

Tôi định thêm lý do khác, nhưng về cơ bản đó là lý do chính và duy nhất không dùng `gets()`. Như bạn có thể đoán, `fgets()` cho phép bạn chỉ định độ dài chuỗi tối đa.

Một khác biệt giữa hai hàm: `gets()` sẽ nuốt và vứt đi ký tự newline cuối dòng, còn `fgets()` sẽ lưu nó vào cuối chuỗi của bạn (nếu còn chỗ).

Đây là ví dụ dùng `fgets()` từ console, làm cho nó cư xử giống `gets()` hơn (ngoại trừ việc chứa newline):

```
char s[100];
gets(s);  // đừng dùng cái này--đọc một dòng (từ stdin)
fgets(s, sizeof(s), stdin); // đọc một dòng từ stdin
```

Trong trường hợp này, toán tử `sizeof()` cho ta tổng kích thước mảng theo byte, và vì `char` là một byte, nó tiện lợi cho ta tổng kích thước của mảng.

Dĩ nhiên, như tôi đã nói, chuỗi trả về từ `fgets()` rất có khả năng có newline ở cuối mà bạn có thể không muốn. Bạn có thể viết một hàm ngắn cắt newline đi—thực tế, hãy gộp luôn nó vào phiên bản `gets()` của riêng ta

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

char *ngets(char *s, int size)
{
    char *rv = fgets(s, size, stdin);

    if (rv == NULL)
        return NULL;

    char *p = strchr(s, '\n');  // Tìm một newline

    if (p != NULL)  // nếu có newline
        *p = '\0';  // cắt chuôĩ ở đó
```

```
    return s;
}
```

Vậy, tóm lại, dùng `fgets()` để đọc một dòng văn bản từ bàn phím hoặc file, và đừng dùng `gets()`.

Giá trị trả về

Cả `gets()` và `fgets()` đều trả về con trỏ đến chuỗi đã truyền vào.

Khi lỗi hoặc end-of-file, các hàm trả về `NULL`.

Ví dụ

```
#include <stdio.h>

int main(void)
{
    FILE *fp;
    char s[100];

    gets(s); // đọc từ standard input (đừng dùng--dùng fgets()!)

    fgets(s, sizeof s, stdin); // đọc 100 byte từ standard input

    fp = fopen("spoon.txt", "r"); // (bạn nên kiểm tra lỗi cái này)
    fgets(s, 100, fp); // đọc 100 byte từ file datafile.dat
    fclose(fp);

    fgets(s, 20, stdin); // đọc tối đa 20 byte từ stdin
}
```

Xem thêm

`getc()`, `fgetc()`, `getchar()`, `puts()`, `fputs()`, `ungetc()`

23.16 `putc()`, `fputc()`, `putchar()`

Ghi một ký tự đơn ra console hoặc ra file

Synopsis

```
#include <stdio.h>

int putc(int c, FILE *stream);

int fputc(int c, FILE *stream);

int putchar(int c);
```

Mô tả

Cả ba hàm output một ký tự đơn, hoặc ra console hoặc ra một `FILE`.

`putc()` nhận một tham số ký tự, và output ra `FILE` đã chỉ định. `fputc()` làm y hệt, chỉ khác `putc()` ở cách cài đặt. Hầu hết mọi người dùng `fputc()`.

`putchar()` ghi ký tự ra console, và giống gọi `putc(c, stdout)`.

Giá trị trả về

Cả ba hàm trả về ký tự đã ghi khi thành công, hoặc `EOF` khi lỗi.

Ví dụ

In bảng chữ cái:

```
#include <stdio.h>

int main(void)
{
    char i;

    for(i = 'A'; i <= 'Z'; i++)
        putchar(i);

    putchar('\n'); // đặt một newline ở cuối cho đẹp
}
```

Xem thêm

23.17 `puts()`, `fputs()`

Ghi một chuỗi ra console hoặc ra file

Synopsis

```
#include <stdio.h>

int puts(const char *s);

int fputs(const char *s, FILE *stream);
```

Mô tả

Cả hai hàm này output một chuỗi kết thúc bằng NUL. `puts()` output ra console, còn `fputs()` cho phép bạn chỉ định file để output.

Giá trị trả về

Cả hai hàm trả về không âm khi thành công, hoặc `EOF` khi lỗi.

Ví dụ

Đọc chuỗi từ console và lưu chúng vào file:

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

```
int main(void)
{
    FILE *fp;
    char s[100];

    fp = fopen("somefile.txt", "w"); // nhớ kiểm tra lỗi cái này!

    while(fgets(s, sizeof(s), stdin) != NULL) { // đọc một chuỗi
        fputs(s, fp);  // ghi vào file đã mở
    }

    fclose(fp);
}
```

Xem thêm

23.18 `ungetc()`

Đẩy một ký tự trở lại input stream

Synopsis

```
#include <stdio.h>

int ungetc(int c, FILE *stream);
```

Mô tả

Bạn biết `getc()` đọc ký tự kế tiếp từ một file stream chứ? Đây là ngược lại—nó đẩy một ký tự trở lại file stream để nó sẽ xuất hiện lại ở lần đọc kế tiếp từ stream, như thể bạn chưa từng lấy nó bằng `getc()` ngay từ đầu.

Vì sao, nhân danh tất cả những gì linh thiêng, bạn lại muốn làm thế? Có lẽ bạn có một stream dữ liệu mà bạn đang đọc từng ký tự một, và bạn sẽ không biết khi nào dừng cho đến khi lấy được một ký tự nhất định, nhưng bạn muốn có thể đọc ký tự đó lại sau. Bạn có thể đọc ký tự, thấy rằng nó là cái cần dừng, rồi `ungetc()` nó để nó xuất hiện ở lần đọc kế.

Đúng rồi, chuyện này không xảy ra thường xuyên, nhưng thì đó.

Đây là điểm cần chú ý: chuẩn chỉ đảm bảo bạn có thể đẩy lại *một ký tự*. Một số cài đặt có thể cho phép đẩy nhiều hơn, nhưng không có cách nào biết chắc mà vẫn portable.

Giá trị trả về

Khi thành công, `ungetc()` trả về ký tự bạn truyền vào. Khi thất bại, nó trả về `EOF`.

Ví dụ

Ví dụ này đọc một dấu câu, rồi mọi thứ sau nó cho đến dấu câu kế. Nó trả về dấu câu đầu, và lưu phần còn lại vào một chuỗi.

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

```
int read_punctstring(FILE *fp, char *s)
{
    int origpunct, c;

    origpunct = fgetc(fp);

    if (origpunct == EOF)  // trả về EOF khi end-of-file
        return EOF;

    while (c = fgetc(fp), !ispunct(c) && c != EOF)
        *s++ = c;  // lưu vào chuỗi

    *s = '\0'; // kết thúc chuỗi bằng NUL

    // nếu vừa đọc là dấu câu, ungetc nó để lần sau fgetc lấy lại:
    if (ispunct(c))
        ungetc(c, fp);

    return origpunct;
}

int main(void)
{
    char s[128];
    char c;

    while((c = read_punctstring(stdin, s)) != EOF) {
        printf("%c: %s\n", c, s);
    }
}
```

Input mẫu:

```
!foo#bar*baz
```

Output mẫu:

```
!: foo
#: bar
*: baz
```

Xem thêm

`fgetc()`

23.19 `fread()`

Đọc dữ liệu nhị phân từ file

Synopsis

```
#include <stdio.h>

size_t fread(void *p, size_t size, size_t nmemb, FILE *stream);
```

Mô tả

Có thể bạn còn nhớ rằng bạn có thể gọi `fopen()` với flag "`b`" trong chuỗi mode để mở file ở chế độ "binary" (chế độ nhị phân). File mở ở chế độ không nhị phân (ASCII hay text mode (chế độ văn bản)) có thể được đọc bằng các lời gọi hướng ký tự chuẩn như `fgetc()` hay `fgets()`. File mở ở binary mode thường được đọc bằng hàm `fread()`.

Cái hàm này làm là nói, "Ê, đọc bằng này thứ, mỗi thứ là một số byte nhất định, rồi lưu cả đống vào bộ nhớ bắt đầu từ con trỏ này."

Cái này có thể rất có ích, tin tôi đi, khi bạn muốn làm thứ như lưu 20 `int` vào file.

Nhưng khoan—không phải bạn có thể dùng `fprintf()` với format specifier "`%d`" để lưu `int` vào file text và lưu chúng kiểu đó sao? Ừ, chắc chắn. Cái đó có lợi thế là con người có thể mở file và đọc số. Nhược điểm là chuyển số từ `int` sang text chậm hơn, và số có khả năng chiếm nhiều chỗ hơn trong file. (Nhớ rằng một `int` khả năng là 4 byte, nhưng chuỗi "12345678" là 8 byte.)

Vậy nên lưu dữ liệu nhị phân chắc chắn có thể gọn hơn và đọc nhanh hơn.

Giá trị trả về

Hàm này trả về số item đọc thành công. Nếu mọi item yêu cầu đều đọc được, giá trị trả về sẽ bằng tham số `nmemb`. Nếu gặp EOF, giá trị trả về sẽ là 0.

Để làm bạn bối rối, nó cũng sẽ trả về 0 nếu có lỗi. Bạn có thể dùng `feof()` hoặc `ferror()` để biết cái nào thực sự xảy ra.

Ví dụ

Đọc 10 số từ file và lưu chúng vào mảng:

```
#include <stdio.h>

int main(void)
{
    int i;
    int n[10]
    FILE *fp;

    fp = fopen("numbers.dat", "rb");
    fread(n, sizeof(int), 10, fp);  // đọc 10 int
    fclose(fp);

    // in chúng ra:
    for(i = 0; i < 10; i++)
        printf("n[%d] == %d\n", i, n[i]);
}
```

Xem thêm

`fopen()`, `fwrite()`, `feof()`, `ferror()`

23.20 `fwrite()`

Ghi dữ liệu nhị phân ra file

Synopsis

```
#include <stdio.h>

size_t fwrite(const void *p, size_t size, size_t nmemb, FILE *stream);
```

Mô tả

Đây là đối tác của hàm `fread()`. Nó ghi khối dữ liệu nhị phân ra đĩa. Để hiểu điều đó nghĩa là gì, xem mục `fread()`.

Giá trị trả về

`fwrite()` trả về số item ghi thành công, hy vọng sẽ là `nmemb` mà bạn truyền vào. Nó sẽ trả về 0 khi lỗi.

Ví dụ

Lưu 10 số ngẫu nhiên vào file:

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

int main(void)
{
    int i;
    int n[10];
    FILE *fp;

    // điền mảng bằng sô´ngâũ nhiên:
    for(i = 0; i < 10; i++) {
        n[i] = rand();
        printf("n[%d] = %d\n", i, n[i]);
    }

    // lưu sô´ngâũ nhiên (10 int) vào file
    fp = fopen("numbers.dat", "wb");
    fwrite(n, sizeof(int), 10, fp); // ghi 10 int
    fclose(fp);
}
```

Xem thêm

`fopen()`, `fread()`

23.21 `fgetpos()`, `fsetpos()`

Lấy vị trí hiện tại trong file, hoặc đặt vị trí hiện tại trong file. Trên hầu hết hệ thống thì giống `ftell()` và `fseek()`

Synopsis

```
#include <stdio.h>

int fgetpos(FILE *stream, fpos_t *pos);

int fsetpos(FILE *stream, fpos_t *pos);
```

Mô tả

Các hàm này giống `ftell()` và `fseek()`, chỉ khác là thay vì đếm theo byte, chúng dùng một cấu trúc dữ liệu *opaque* để giữ thông tin vị trí về file. (Opaque, ở đây, nghĩa là bạn không được phép biết kiểu dữ liệu được tạo từ cái gì.)

Trên gần như mọi hệ thống (và chắc chắn trên mọi hệ thống tôi biết), người ta không dùng những hàm này, mà dùng `ftell()` và `fseek()` thay thế. Các hàm này tồn tại đề phòng hệ thống của bạn không nhớ được vị trí file dưới dạng offset byte đơn giản.

Vì biến `pos` là opaque, bạn phải gán vào nó bằng chính lời gọi `fgetpos()`. Rồi bạn lưu giá trị cho sau này và dùng nó để reset vị trí bằng `fsetpos()`.

Giá trị trả về

Cả hai hàm trả về 0 khi thành công, và `-1` khi lỗi.

Ví dụ

```
#include <stdio.h>

int main(void)
{
    char s[100];
    fpos_t pos;
    FILE *fp;

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

    fgets(s, sizeof(s), fp); // đọc một dòng từ file
    printf("%s", s);

    fgetpos(fp, &pos);   // lưu vị trí sau khi đọc

    fgets(s, sizeof(s), fp); // đọc một dòng khác từ file
    printf("%s", s);

    fsetpos(fp, &pos);   // giờ khôi phục vị trí đã lưu

    fgets(s, sizeof(s), fp); // đọc lại dòng trước đó
    printf("%s", s);

    fclose(fp);
}
```

Xem thêm

`fseek()`, `ftell()`, `rewind()`

23.22 `fseek()`, `rewind()`

Định vị file pointer để chuẩn bị cho lần đọc hoặc ghi kế tiếp

Synopsis

```
#include <stdio.h>

int fseek(FILE *stream, long offset, int whence);

void rewind(FILE *stream);
```

Mô tả

Khi đọc ghi một file, OS theo dõi bạn đang ở đâu trong file bằng một bộ đếm gọi chung là file pointer. Bạn có thể đặt lại file pointer đến một điểm khác trong file bằng lời gọi `fseek()`. Coi nó như cách truy cập ngẫu nhiên file.

Tham số đầu là file đang nói đến, hiển nhiên rồi. Tham số `offset` là vị trí bạn muốn seek (seek / di chuyển) đến, và `whence` là cái offset đó so với cái gì.

Dĩ nhiên, chắc bạn muốn nghĩ offset là tính từ đầu file. Ý tôi là, "Seek đến vị trí 3490, cái đó nên là 3490 byte tính từ đầu file." Ồ, nó *có thể* thế, nhưng không bắt buộc. Hãy tưởng tượng sức mạnh bạn đang nắm giữ. Cố kiềm chế sự hứng thú của bạn.

Bạn có thể đặt giá trị `whence` thành một trong ba thứ:

`whence`	Mô tả
`SEEK_SET`	`offset` là tương đối so với đầu file. Đây chắc là cái bạn đang nghĩ, và là giá trị dùng phổ biến nhất cho `whence`.
`SEEK_CUR`	`offset` là tương đối so với vị trí file pointer hiện tại. Nên, thực tế, bạn có thể nói, "Di chuyển đến vị trí hiện tại cộng 30 byte," hoặc, "di chuyển đến vị trí hiện tại trừ 20 byte."
`SEEK_END`	`offset` là tương đối so với cuối file. Giống `SEEK_SET` nhưng tính từ đầu kia của file. Nhớ dùng giá trị âm cho `offset` nếu muốn lùi từ cuối file, thay vì đi quá đuôi vào hư vô.

Nhắc đến seek ra khỏi cuối file, làm được không? Được luôn. Thực tế, bạn có thể seek xa tít khỏi cuối rồi ghi một ký tự; file sẽ được mở rộng ra đủ lớn để chứa cả đống số 0 ra đến ký tự đó.

Giờ hàm phức tạp đã xong, `rewind()` mà tôi vừa đề cập là gì? Nó đặt lại file pointer về đầu file:

```
fseek(fp, 0, SEEK_SET); // giống rewind()
rewind(fp);             // giống fseek(fp, 0, SEEK_SET)
```

Giá trị trả về

Với `fseek()`, khi thành công trả về 0; `-1` khi thất bại.

Lời gọi `rewind()` không bao giờ thất bại.

Ví dụ

```
#include <stdio.h>

int main(void)
{
    FILE *fp;

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

    fseek(fp, 100, SEEK_SET); // seek đến byte thứ 100 của file
    printf("100: %c\n", fgetc(fp));

    fseek(fp, -31, SEEK_CUR); // seek lùi 30 byte từ vị trí hiện tại
    printf("31 back: %c\n", fgetc(fp));

    fseek(fp, -12, SEEK_END); // seek đến byte thứ 10 trước cuối file
    printf("12 from end: %c\n", fgetc(fp));

    fseek(fp, 0, SEEK_SET);   // seek về đầu file
    rewind(fp);               // cũng seek về đầu file
    printf("Beginning: %c\n", fgetc(fp));

    fclose(fp);
}
```

Xem thêm

`ftell()`, `fgetpos()`, `fsetpos()`

23.23 `ftell()`

Cho bạn biết một file sắp đọc từ hay ghi vào chỗ nào

Synopsis

```
#include <stdio.h>

long ftell(FILE *stream);
```

Mô tả

Hàm này là ngược của `fseek()`. Nó cho bạn biết vị trí trong file mà thao tác file kế tiếp sẽ xảy ra, tính từ đầu file.

Hữu ích nếu bạn muốn nhớ vị trí hiện tại trong file, `fseek()` đi chỗ khác, rồi quay lại sau. Bạn có thể lấy giá trị trả về từ `ftell()` và đưa lại vào `fseek()` (với tham số `whence` đặt là `SEEK_SET`) khi muốn quay về vị trí trước đó.

Giá trị trả về

Trả về offset hiện tại trong file, hoặc `-1` khi lỗi.

Ví dụ

```
#include <stdio.h>

int main(void)
{
    char c[6];
    FILE *fp;

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

    long pos;

    // seek tới 10 byte:
    fseek(fp, 10, SEEK_SET);

    // lưu vị trí hiện tại vào biến "pos":
    pos = ftell(fp);

    // Đọc vài byte
    fread(c, sizeof c  - 1, 1, fp);
    c[5] = '\0';
    printf("Read: \"%s\"\n", c);

    // và quay về vị trí bắt đầu, đã lưu trong "pos":
    fseek(fp, pos, SEEK_SET);

    // Đọc lại đúng những byte đó
    fread(c, sizeof c  - 1, 1, fp);
    c[5] = '\0';
    printf("Read: \"%s\"\n", c);

    fclose(fp);
}
```

Xem thêm

`fseek()`, `rewind()`, `fgetpos()`, `fsetpos()`

23.24 `feof()`, `ferror()`, `clearerr()`

Xác định xem file đã đến end-of-file chưa hoặc có lỗi không

Synopsis

```
#include <stdio.h>

int feof(FILE *stream);

int ferror(FILE *stream);

void clearerr(FILE *stream);
```

Mô tả

Mỗi `FILE*` bạn dùng để đọc ghi dữ liệu với file đều chứa các flag mà hệ thống set khi xảy ra sự kiện nhất định. Nếu bị lỗi, nó set cờ lỗi; nếu đạt cuối file trong lúc đọc, nó set cờ EOF. Khá đơn giản.

Các hàm `feof()` và `ferror()` cho bạn cách đơn giản để test các flag này: chúng sẽ trả về khác 0 (true) nếu được set.

Khi các flag đã được set cho một stream nào đó, chúng giữ nguyên đến khi bạn gọi `clearerr()` để xoá.

Giá trị trả về

`feof()` và `ferror()` trả về khác 0 (true) nếu file đã đạt EOF hoặc có lỗi, tương ứng.

Ví dụ

Đọc dữ liệu nhị phân, kiểm tra EOF hoặc lỗi:

```
#include <stdio.h>

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

    fp = fopen("numbers.dat", "r");

    // đọc từng int một, dừng khi EOF hoặc lôĩ:

    while(fread(&a, sizeof(int), 1, fp), !feof(fp) && !ferror(fp)) {
        printf("Read %d\n", a);
    }

    if (feof(fp))
        printf("End of file was reached.\n");

    if (ferror(fp))
        printf("An error occurred.\n");

    fclose(fp);
}
```

Xem thêm

`fopen()`, `fread()`

23.25 `perror()`

In thông báo lỗi gần nhất ra `stderr`

Synopsis

```
#include <stdio.h>
#include <errno.h>  // chỉ cần nếu bạn muốn dùng trực tiếp biến "errno"

void perror(const char *s);
```

Mô tả

Nhiều hàm, khi gặp điều kiện lỗi vì bất cứ lý do gì, sẽ set cho bạn một biến toàn cục tên `errno` (trong `<errno.h>`). `errno` chỉ là một số nguyên đại diện cho một lỗi duy nhất.

Nhưng với bạn, người dùng, một con số thường không hữu ích lắm. Vì lý do đó, bạn có thể gọi `perror()` sau khi xảy ra lỗi để in ra lỗi thực sự đã xảy ra dưới dạng chuỗi dễ đọc cho con người.

Và để giúp bạn, bạn có thể truyền tham số `s` sẽ được đặt trước chuỗi lỗi.

Một mẹo khéo khác là kiểm tra giá trị của `errno` (phải include `errno.h` mới thấy nó) đối với các lỗi cụ thể và cho code của bạn làm những việc khác nhau. Có lẽ bạn muốn bỏ qua một số lỗi nhưng không phải lỗi khác, chẳng hạn.

Chuẩn chỉ định nghĩa ba giá trị cho `errno`, nhưng hệ thống của bạn chắc chắn định nghĩa nhiều hơn. Ba giá trị được định nghĩa là:

`errno`	Mô tả
`EDOM`	Phép toán ngoài miền.
`EILSEQ`	Sequence (chuỗi) không hợp lệ trong encoding multibyte (đa byte) sang wide character.
`ERANGE`	Kết quả phép toán không vừa kiểu đã chỉ định.

Điểm cần chú ý là các hệ thống khác nhau định nghĩa giá trị `errno` khác nhau, nên không portable lắm ngoài 3 cái trên. Tin tốt là ít nhất các giá trị *phần lớn* portable giữa các hệ thống giống Unix.

Giá trị trả về

Không trả về gì hết! Xin lỗi!

Ví dụ

`fseek()` trả về `-1` khi lỗi, và set `errno`, vậy nên dùng nó. Seek trên `stdin` vô nghĩa, nên sẽ sinh lỗi:

```
#include <stdio.h>
#include <errno.h> // phải include cái này để thấy "errno" trong ví dụ này

int main(void)
{
    if (fseek(stdin, 10L, SEEK_SET) < 0)
        perror("fseek");

    fclose(stdin); // ngừng dùng stream này

    if (fseek(stdin, 20L, SEEK_CUR) < 0) {

        // kiểm tra cụ thể errno để xem loại lỗi
        // nào đã xảy ra...cái này hoạt động trên Linux,
        // nhưng trên hệ khác có thể khác!

        if (errno == EBADF) {
            perror("fseek again, EBADF");
        } else {
            perror("fseek again");
        }
    }
}
```

Và output là:

```
fseek: Illegal seek
fseek again, EBADF: Bad file descriptor
```

Xem thêm

`feof()`, `ferror()`, `strerror()`

Chapter 24

`<stdlib.h>` Các Hàm Thư Viện Chuẩn

Một số hàm dưới đây có các biến thể xử lý các kiểu khác nhau: `atoi()`, `strtod()`, `strtol()`, `abs()`, và `div()`. Ở đây tôi chỉ liệt kê một cái cho gọn.

Hàm	Mô tả
`_Exit()`	Thoát (exit) chương trình đang chạy và không ngoái đầu nhìn lại
`abort()`	Kết thúc chương trình đột ngột
`abs()`	Tính giá trị tuyệt đối của một số nguyên
`aligned_alloc()`	Cấp phát bộ nhớ được căn chỉnh cụ thể
`at_quick_exit()`	Đăng ký handler chạy khi chương trình thoát nhanh
`atexit()`	Đăng ký handler chạy khi chương trình thoát
`atof()`	Chuyển chuỗi thành giá trị dấu phẩy động
`atoi()`	Chuyển một số nguyên trong chuỗi sang kiểu số nguyên
`bsearch()`	Binary Search (có thể) trên một mảng đối tượng
`calloc()`	Cấp phát và điền 0 bộ nhớ để dùng tuỳ ý
`div()`	Tính thương và phần dư của hai số
`exit()`	Thoát chương trình đang chạy
`free()`	Giải phóng một vùng bộ nhớ
`getenv()`	Lấy giá trị của một biến môi trường
`malloc()`	Cấp phát bộ nhớ để dùng tuỳ ý
`mblen()`	Trả về số byte của một ký tự multibyte
`mbstowcs()`	Chuyển chuỗi multibyte thành chuỗi ký tự rộng
`mbtowc()`	Chuyển một ký tự multibyte thành ký tự rộng
`qsort()`	Quicksort (có thể) dữ liệu
`quick_exit()`	Thoát chương trình đang chạy một cách nhanh chóng
`rand()`	Trả về một số giả ngẫu nhiên
`realloc()`	Đổi kích thước một vùng bộ nhớ đã cấp phát trước đó
`srand()`	Seed bộ sinh số giả ngẫu nhiên có sẵn
`strtod()`	Chuyển chuỗi thành số dấu phẩy động
`strtol()`	Chuyển chuỗi thành số nguyên
`system()`	Chạy một chương trình bên ngoài
`wcstombs()`	Chuyển chuỗi ký tự rộng thành chuỗi multibyte
`wctomb()`	Chuyển ký tự rộng thành ký tự multibyte

Header `<stdlib.h>` gói đủ mọi loại hàm—dám nói luôn—linh tinh nhét hết vào đây. Bao gồm:

- Chuyển đổi từ số sang chuỗi
- Chuyển đổi từ chuỗi sang số
- Sinh số giả ngẫu nhiên

- Cấp phát bộ nhớ động
- Nhiều cách để thoát chương trình
- Khả năng chạy chương trình bên ngoài
- Binary search (hoặc vài kiểu tìm kiếm nhanh nào đó)
- Quicksort (hoặc vài kiểu sắp xếp nhanh nào đó)
- Các hàm số học với số nguyên
- Chuyển đổi ký tự và chuỗi multibyte và ký tự rộng

Vậy đó... mỗi thứ một tí.

24.1 Kiểu và Macro của `<stdlib.h>`

Vài kiểu và macro mới được giới thiệu, dù một số có thể cũng được định nghĩa ở chỗ khác:

Kiểu	Mô tả
`size_t`	Được trả về từ `sizeof` và dùng ở nhiều chỗ khác
`wchar_t`	Cho các thao tác với ký tự rộng
`div_t`	Cho hàm `div()`
`ldiv_t`	Cho hàm `ldiv()`
`lldiv_t`	Cho hàm `lldiv()`

Và một vài macro:

Kiểu	Mô tả
`NULL`	Người bạn con trỏ tốt của chúng ta
`EXIT_SUCCESS`	Mã thoát đẹp khi mọi thứ suôn sẻ
`EXIT_FAILURE`	Mã thoát đẹp khi mọi thứ tệ hại
`RAND_MAX`	Giá trị lớn nhất mà hàm `rand()` có thể trả về
`MB_CUR_MAX`	Số byte tối đa trong một ký tự multibyte tại locale hiện tại

Vậy đó. Bao nhiêu hàm vui vẻ và hữu ích trong đây. Cùng xem thử nào!

24.2 `atof()`

Chuyển chuỗi thành giá trị dấu phẩy động

Synopsis

```
#include <stdlib.h>

double atof(const char *nptr);
```

Mô tả

Ngày xưa cái tên này viết tắt cho "ASCII-To-Floating" hồi đó[1], nhưng bây giờ không ai dám dùng kiểu ngôn ngữ thô thiển đó nữa.

[1] http://man.cat-v.org/unix-1st/3/atof

Nhưng ý tưởng thì vẫn như cũ: chúng ta sẽ chuyển một chuỗi chứa chữ số và (tuỳ chọn) một dấu thập phân thành giá trị dấu phẩy động. Khoảng trắng đầu chuỗi bị bỏ qua, và việc chuyển đổi dừng ở ký tự không hợp lệ đầu tiên.

Nếu kết quả không vừa trong `double`, hành vi là không xác định.

Nó thường hoạt động như thể bạn đã gọi `strtod()`:

```
strtod(nptr, NULL)
```

Nên xem trang tham chiếu đó để biết thêm thông tin.

Thực ra `strtod()` xịn hơn và bạn chắc nên dùng nó.

Giá trị trả về

Trả về chuỗi đã được chuyển thành `double`.

Ví dụ

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

int main(void)
{
    double x = atof("3.141593");

    printf("%f\n", x);  // 3.141593
}
```

Xem thêm

`atoi()`, `strtod()`

24.3 `atoi()`, `atol()`, `atoll()`

Chuyển một số nguyên trong chuỗi sang kiểu số nguyên

Synopsis

```
#include <stdlib.h>

int atoi(const char *nptr);

long int atol(const char *nptr);

long long int atoll(const char *nptr);
```

Mô tả

Ngày xưa, `atoi()` viết tắt cho "ASCII-To_Integer"[2] nhưng giờ spec không nhắc đến chuyện đó nữa.

Các hàm này nhận một chuỗi có số trong đó và chuyển nó thành số nguyên của kiểu trả về tương ứng. Khoảng trắng đầu bị bỏ qua. Việc chuyển đổi dừng ở ký tự không hợp lệ đầu tiên.

[2] http://man.cat-v.org/unix-1st/3/atoi

Nếu kết quả không vừa trong kiểu trả về, hành vi là không xác định.

Nó thường hoạt động như thể bạn đã gọi họ hàm `strtol()`:

```
atoi(nptr)                // về cơ bản giống như...
(int)strtol(nptr, NULL, 10)

atol(nptr)                // về cơ bản giống như...
strtol(nptr, NULL, 10)

atoll(nptr)               // về cơ bản giống như...
strtoll(nptr, NULL, 10)
```

Lần nữa, các hàm `strtol()` nhìn chung tốt hơn, nên tôi khuyên dùng chúng thay vì mấy hàm này.

Giá trị trả về

Trả về kết quả số nguyên tương ứng với kiểu trả về.

Ví dụ

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

int main(void)
{
    int x = atoi("3490");

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

Xem thêm

`atof()`, `strtol()`

24.4 `strtod()`, `strtof()`, `strtold()`

Chuyển chuỗi thành số dấu phẩy động

Synopsis

```
#include <stdlib.h>

double strtod(const char * restrict nptr, char ** restrict endptr);

float strtof(const char * restrict nptr, char ** restrict endptr);

long double strtold(const char * restrict nptr, char ** restrict endptr);
```

Mô tả

Đây là mấy hàm hay ho, chuyển chuỗi thành số dấu phẩy động (thậm chí cả NaN hoặc Infinity) và lại còn cung cấp chút kiểm tra lỗi nữa.

Đầu tiên, khoảng trắng đầu chuỗi bị bỏ qua.

Rồi các hàm cố gắng chuyển từng ký tự thành kết quả dấu phẩy động. Cuối cùng, khi gặp ký tự không hợp lệ (hoặc ký tự NUL), chúng đặt `endptr` trỏ tới ký tự không hợp lệ đó.

Đặt `endptr` thành `NULL` nếu bạn không quan tâm đến vị trí ký tự không hợp lệ đầu tiên ở đâu.

Nếu bạn không đặt `endptr` thành `NULL`, nó sẽ trỏ tới ký tự NUL nếu quá trình chuyển đổi không gặp ký tự xấu nào. Nghĩa là:

```
if (*endptr == '\0') {
    printf("What a perfectly-formed number!\n");
} else {
    printf("I found badness in your number: \"%s\"\n", endptr);
}
```

Nhưng đoán xem! Bạn cũng có thể chuyển chuỗi thành các giá trị đặc biệt, như NaN và Infinity!

Nếu `nptr` trỏ tới chuỗi chứa `INF` hoặc `INFINITY` (chữ hoa hay chữ thường cũng được), giá trị Infinity sẽ được trả về.

Nếu `nptr` trỏ tới chuỗi chứa `NAN`, thì (một NaN yên lặng, không báo hiệu) sẽ được trả về. Bạn có thể gắn tag cho `NAN` bằng một chuỗi ký tự từ tập `0`-`9`, `a`-`z`, `A`-`Z`, và `_` bằng cách bao chúng trong dấu ngoặc:

```
NAN(foobar_3490)
```

Việc compiler của bạn làm gì với cái này là do implementation quyết định, nhưng nó có thể được dùng để chỉ định các kiểu NaN khác nhau.

Bạn cũng có thể chỉ định số ở dạng thập lục phân với số mũ luỹ thừa 2 (2^x) nếu bắt đầu bằng `0x` (hoặc `0X`). Với phần số mũ, dùng `p` theo sau là số mũ cơ số 10. (Bạn không thể dùng `e` vì đó là chữ số hex hợp lệ!)

Ví dụ:

```
0xabc.123p15
```

Có giá trị bằng $0xabc.123 \times 2^{15}$.

Bạn có thể đưa vào `FLT_DECIMAL_DIG`, `DBL_DECIMAL_DIG`, hoặc `LDBL_DECIMAL_DIG` chữ số và nhận được kết quả làm tròn chính xác cho kiểu đó.

Giá trị trả về

Trả về số đã chuyển đổi. Nếu không có số nào, trả về `0`. `endptr` được đặt trỏ tới ký tự không hợp lệ đầu tiên, hoặc tới ký tự NUL kết thúc nếu tất cả ký tự đều được xử lý.

Nếu có tràn số, `HUGE_VAL`, `HUGE_VALF`, hoặc `HUGE_VALL` sẽ được trả về, mang dấu của input, và `errno` được đặt thành `ERANGE`.

Nếu có underflow, nó trả về số nhỏ nhất gần 0 nhất với dấu của input. `errno` có thể được đặt thành `ERANGE`.

Ví dụ

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

int main(void)
{
    char *inp = "   123.4567beej";
```

```
    char *badchar;

    double val = strtod(inp, &badchar);

    printf("Converted string to %f\n", val);
    printf("Encountered bad characters: %s\n", badchar);

    val = strtod("987.654321beej", NULL);
    printf("Ignoring bad chars for result: %f\n", val);

    val = strtod("11.2233", &badchar);

    if (*badchar == '\0')
        printf("No bad chars: %f\n", val);
    else
        printf("Found bad chars: %f, %s\n", val, badchar);
}
```

Output:

```
Converted string to 123.456700
Encountered bad characters: beej
Ignoring bad chars: 987.654321
No bad chars: 11.223300
```

Xem thêm

`atof()`, `strtol()`

24.5 `strtol()`, `strtoll()`, `strtoul()`, `strtoull()`

Chuyển chuỗi thành số nguyên

Synopsis

```
#include <stdlib.h>

long int strtol(const char * restrict nptr,
                char ** restrict endptr, int base);

long long int strtoll(const char * restrict nptr,
                      char ** restrict endptr, int base);

unsigned long int strtoul(const char * restrict nptr,
                          char ** restrict endptr, int base);

unsigned long long int strtoull(const char * restrict nptr,
                                char ** restrict endptr, int base);
```

Mô tả

Mấy hàm này chuyển chuỗi thành số nguyên giống `atoi()`, nhưng chúng có thêm vài tính năng hay ho.

Đáng chú ý nhất, chúng có thể cho bạn biết chỗ nào việc chuyển đổi bắt đầu sai, tức là ký tự không hợp lệ, nếu có, xuất hiện ở đâu. Khoảng trắng đầu bị bỏ qua. Dấu `+` hoặc `-` có thể đứng trước số.

Ý tưởng cơ bản là nếu mọi thứ ổn, các hàm này sẽ trả về giá trị số nguyên chứa trong chuỗi. Và nếu bạn truyền vào `endptr` kiểu `char**`, nó sẽ đặt biến đó trỏ vào ký tự NUL ở cuối chuỗi.

Nếu mọi thứ không ổn, chúng sẽ đặt `endptr` trỏ vào ký tự đầu tiên có vấn đề. Nghĩa là, nếu bạn đang chuyển giá trị `103z2!` ở hệ 10, chúng sẽ đặt `endptr` trỏ vào ký tự `z` vì đó là ký tự không phải số đầu tiên.

Bạn có thể truyền `NULL` cho `endptr` nếu không muốn làm kiểu kiểm tra lỗi đó.

Khoan—tôi vừa nói là chúng ta có thể chỉ định cơ số cho việc chuyển đổi à? Đúng rồi! Đúng, tôi nói vậy. Mà cơ số[3] thì nằm ngoài phạm vi tài liệu này rồi, nhưng chắc chắn một số cơ số quen thuộc hơn là nhị phân (cơ số 2), bát phân (cơ số 8), thập phân (cơ số 10), và thập lục phân (cơ số 16).

Bạn có thể chỉ định cơ số để chuyển đổi làm tham số thứ ba. Các cơ số từ 2 đến 36 được hỗ trợ, với các chữ số không phân biệt hoa thường chạy từ `0` tới `Z`.

Nếu bạn chỉ định cơ số là `0`, hàm sẽ cố gắng tự xác định nó. Mặc định là cơ số 10 trừ vài trường hợp:

Chuyển nhị phân là mới trong C23!

- Nếu số bắt đầu bằng `0b` hoặc `0B`, nó sẽ là nhị phân (cơ số 2)
- Nếu số bắt đầu bằng `0`, nó sẽ là bát phân (cơ số 8)
- Nếu số bắt đầu bằng `0x` hoặc `0X`, nó sẽ là hex (cơ số 16)

Locale có thể ảnh hưởng đến hành vi của các hàm này.

Giá trị trả về

Trả về giá trị đã chuyển đổi.

`endptr`, nếu không phải `NULL`, được đặt trỏ vào ký tự không hợp lệ đầu tiên, hoặc vào đầu chuỗi nếu không có chuyển đổi nào được thực hiện, hoặc vào ký tự NUL kết thúc chuỗi nếu tất cả ký tự đều hợp lệ.

Nếu có tràn số, một trong các giá trị sau sẽ được trả về: `LONG_MIN`, `LONG_MAX`, `LLONG_MIN`, `LLONG_MAX`, `ULONG_MAX`, `ULLONG_MAX`. Và `errno` được đặt thành `ERANGE`.

Ví dụ

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

int main(void)
{
    // Tất cả output ở hệ thập phân (cơ sô´10)

    printf("%ld\n", strtol("123", NULL, 0));      // 123
    printf("%ld\n", strtol("123", NULL, 10));     // 123
    printf("%ld\n", strtol("101010", NULL, 2));   // nhị phân, 42
    printf("%ld\n", strtol("123", NULL, 8));      // bát phân, 83
    printf("%ld\n", strtol("123", NULL, 16));     // hex, 291

    printf("%ld\n", strtol("0123", NULL, 0));     // bát phân, 83
    printf("%ld\n", strtol("0x123", NULL, 0));    // hex, 291

    char *badchar;
    long int x = strtol("   1234beej", &badchar, 0);
```

[3] https://en.wikipedia.org/wiki/Radix

```
    printf("Value is %ld\n", x);               // Value is 1234
    printf("Bad chars at \"%s\"\n", badchar);  // Bad chars at "beej"
}
```

Output:

```
123
123
42
83
291
83
291
Value is 1234
Bad chars at "beej"
```

Xem thêm

`atoi()`, `strtod()`, `setlocale()`, `strtoimax()`, `strtoumax()`

24.6 `rand()`

Trả về một số giả ngẫu nhiên

Synopsis

```
#include <stdlib.h>

int rand(void);
```

Mô tả

Hàm này trả về một số giả ngẫu nhiên trong khoảng `0` tới `RAND_MAX`, bao gồm cả hai đầu. (`RAND_MAX` sẽ tối thiểu là 32767.)

Nếu muốn ép nó vào một khoảng nhất định, cách cổ điển là dùng toán tử modulo `%`, dù việc này có gây ra bias[4] nếu `RAND_MAX+1` không phải bội số của số bạn chia dư. Xử lý chuyện này nằm ngoài phạm vi guide này.

Nếu muốn tạo một số dấu phẩy động giữa `0` và `1` bao gồm cả hai, bạn có thể chia kết quả cho `RAND_MAX`. Hoặc `RAND_MAX+1` nếu không muốn bao gồm `1`. Nhưng tất nhiên, cũng có những vấn đề ngoài phạm vi nữa[5].

Tóm lại, `rand()` là cách tuyệt vời để lấy các số ngẫu nhiên có thể tệ một cách dễ dàng. Chắc cũng đủ dùng cho trò chơi bạn đang viết.

Spec nói rõ thêm:

> Không có đảm bảo nào về chất lượng của chuỗi ngẫu nhiên được sinh ra và một số implementation được biết là sinh ra các chuỗi với các bit thấp kém ngẫu nhiên đến mức đáng lo. Ứng dụng có yêu cầu riêng nên dùng bộ sinh được biết là đủ tốt cho nhu cầu của chúng.

[4] https://stackoverflow.com/questions/10984974/why-do-people-say-there-is-modulo-bias-when-using-a-random-number-generator

[5] https://mumble.net/~campbell/2014/04/28/uniform-random-float

Hệ thống của bạn chắc có bộ sinh số ngẫu nhiên tốt nếu bạn cần nguồn mạnh hơn. Người dùng Linux có `getrandom()`, ví dụ vậy, và Windows có `CryptGenRandom()`.

Với các công việc số ngẫu nhiên đòi hỏi cao hơn, bạn có thể thấy một thư viện như GNU Scientific Library[6] hữu ích.

Với đa số implementation, các số do `rand()` sinh ra sẽ giống nhau qua các lần chạy. Để khắc phục, bạn cần bắt đầu nó ở một vị trí khác bằng cách truyền một *seed* vào bộ sinh số ngẫu nhiên. Bạn có thể làm việc này với `srand()`.

Giá trị trả về

Trả về một số ngẫu nhiên trong khoảng `0` tới `RAND_MAX`, bao gồm cả hai đầu.

Ví dụ

Lưu ý rằng tất cả các ví dụ này đều không sinh ra phân phối hoàn toàn đều. Nhưng đủ tốt cho mắt người không luyện, và thực sự rất phổ biến trong sử dụng chung khi chất lượng số ngẫu nhiên trung bình là chấp nhận được.

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

int main(void)
{
    printf("RAND_MAX = %d\n", RAND_MAX);

    printf("0 to 9: %d\n", rand() % 10);

    printf("10 to 44: %d\n", rand() % 35 + 10);
    printf("0 to 0.99999: %f\n", rand() / ((float)RAND_MAX + 1));
    printf("10.5 to 15.7: %f\n", 10.5 + 5.2 * rand() / (float)RAND_MAX);
}
```

Output trên máy tôi:

```
RAND_MAX = 2147483647
0 to 9: 3
10 to 44: 21
0 to 0.99999: 0.783099
10.5 to 15.7: 14.651888
```

Ví dụ seed cho RNG bằng thời gian:

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

int main(void)
{
    // time(NULL) gần như chắc chắn trả về số giây kể từ
    // ngày 1 tháng 1 năm 1970:

    srand(time(NULL));

    for (int i = 0; i < 5; i++)
        printf("%d\n", rand());
```

[6] https://www.gnu.org/software/gsl/doc/html/rng.html

```
}
```

Xem thêm

`srand()`

24.7 `srand()`

Seed bộ sinh số giả ngẫu nhiên có sẵn

Synopsis

```
#include <stdlib.h>

void srand(unsigned int seed);
```

Mô tả

Bí mật nho nhỏ bẩn thỉu của việc sinh số giả ngẫu nhiên là chúng hoàn toàn xác định (deterministic). Không có gì ngẫu nhiên về chúng cả. Chúng chỉ *trông* ngẫu nhiên thôi.

Nếu bạn dùng `rand()` và chạy chương trình nhiều lần, bạn có thể thấy một chuyện *đáng ngờ*: chúng sinh ra cùng các số ngẫu nhiên lặp đi lặp lại.

Để thay đổi, chúng ta cần cho bộ sinh số giả ngẫu nhiên một "điểm bắt đầu" mới, gọi là vậy. Chúng ta gọi đó là *seed*. Nó chỉ là một số, nhưng nó được dùng làm nền tảng cho việc sinh số tiếp theo. Cho seed khác, bạn sẽ có dãy số ngẫu nhiên khác. Cho seed giống nhau, bạn sẽ có cùng dãy số ngẫu nhiên tương ứng[7].

Nên nếu bạn gọi `srand(3490)` trước khi bắt đầu sinh số với `rand()`, bạn sẽ có cùng dãy mỗi lần. `srand(37)` cũng sẽ cho bạn cùng dãy mỗi lần, nhưng đó sẽ là dãy khác với dãy bạn có từ `srand(3490)`.

Nhưng nếu bạn không thể hardcode seed (vì làm vậy sẽ cho bạn cùng dãy mỗi lần), thì bạn làm thế nào?

Rất phổ biến là dùng số giây kể từ ngày 1 tháng 1 năm 1970 (ngày này được biết đến là *Unix epoch*[8]) để seed cho bộ sinh. Cái này nghe có vẻ khá tuỳ tiện nhưng thực ra đó chính xác là giá trị mà đa số implementation trả về từ lời gọi thư viện `time(NULL)`[9].

Chúng ta sẽ làm vậy trong ví dụ.

Nếu bạn không gọi `srand()`, nó như thể bạn đã gọi `srand(1)`.

Giá trị trả về

Không trả về gì!

---

[7] Fan của Minecraft có thể nhớ rằng khi tạo thế giới mới, họ được cho tuỳ chọn nhập vào một seed số ngẫu nhiên. Giá trị duy nhất đó được dùng để sinh ra toàn bộ thế giới ngẫu nhiên đó. Và nếu bạn của bạn bắt đầu thế giới với cùng seed bạn dùng, họ sẽ nhận được cùng thế giới bạn có.

[8] https://en.wikipedia.org/wiki/Unix_time

[9] Spec của C không quy định chính xác `time(NULL)` sẽ trả về gì, nhưng spec POSIX thì có! Và gần như ai cũng trả về đúng thứ đó: số giây kể từ epoch.

Ví dụ

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>    // cho lời gọi time()

int main(void)
{
    srand(time(NULL));

    for (int i = 0; i < 5; i++)
        printf("%d\n", rand() % 32);
}
```

Output:

```
4
20
22
14
9
```

Output từ lần chạy tiếp theo:

```
19
0
31
31
24
```

Xem thêm

`rand()`, `time()`

24.8 `aligned_alloc()`

Cấp phát bộ nhớ được căn chỉnh cụ thể

Synopsis

```
#include <stdlib.h>

void *aligned_alloc(size_t alignment, size_t size);
```

Mô tả

Có lẽ bạn muốn dùng `malloc()` hoặc `calloc()` thay vì cái này. Nhưng nếu chắc chắn là không, đọc tiếp!

Bình thường bạn không cần nghĩ về chuyện này, vì `malloc()` và `realloc()` cả hai đều cung cấp vùng bộ nhớ được căn chỉnh[10] phù hợp để dùng với bất kỳ kiểu dữ liệu nào.

Nhưng nếu bạn cần căn chỉnh cụ thể hơn, bạn có thể chỉ định nó với hàm này.

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

Khi xong việc với vùng bộ nhớ, nhớ giải phóng nó bằng một lời gọi `free()`.

Đừng truyền `0` cho size. Nó chắc không làm gì bạn muốn đâu.

Nếu bạn thắc mắc, tất cả bộ nhớ cấp phát động đều được hệ thống tự động giải phóng khi chương trình kết thúc. Dù vậy, việc `free()` rõ ràng mọi thứ bạn đã cấp phát được coi là *Good Form* (phong cách tốt). Cách này các lập trình viên khác sẽ không nghĩ bạn cẩu thả.

Giá trị trả về

Trả về con trỏ tới vùng bộ nhớ vừa được cấp phát, căn chỉnh như yêu cầu. Trả về `NULL` nếu có gì đó sai.

Ví dụ

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

int main(void)
{
    int *p = aligned_alloc(256, 10 * sizeof(int));

    // Cho vui, chuyển sang intptr_t và mod với 256
    // để chắc chắn chúng ta thật sự căn chỉnh trên biên 256 byte.
    //
    // Chuyện này chắc là một kiểu hành vi do implementation định nghĩa,
    // nhưng tôi cá là nó hoạt động.

    intptr_t ip = (intptr_t)p;

    printf("%ld\n", ip % 256);   // 0!

    // Giải phóng
    free(p);
}
```

Xem thêm

`malloc()`, `calloc()`, `free()`

24.9 `calloc()`, `malloc()`

Cấp phát bộ nhớ để dùng tuỳ ý

Synopsis

```
#include <stdlib.h>

void *calloc(size_t nmemb, size_t size);

void *malloc(size_t size);
```

Mô tả

Cả hai hàm này đều cấp phát bộ nhớ cho mục đích chung. Vùng nhớ sẽ được căn chỉnh sao cho có thể dùng lưu trữ bất kỳ kiểu dữ liệu nào.

`malloc()` cấp phát chính xác số byte bộ nhớ được chỉ định trong một khối liên tục. Vùng bộ nhớ có thể đầy dữ liệu rác. (Bạn có thể xoá nó bằng `memset()`, nếu muốn.)

`calloc()` thì khác ở chỗ nó cấp phát chỗ cho `nmemb` đối tượng, mỗi cái `size` byte. (Bạn có thể làm điều tương tự với `malloc()`, nhưng bạn phải tự nhân.)

`calloc()` có thêm tính năng: nó xoá toàn bộ bộ nhớ về `0`.

Nên nếu bạn định sao cũng zero out bộ nhớ, `calloc()` có lẽ là lựa chọn đúng. Nếu không, bạn có thể tránh chi phí đó bằng cách gọi `malloc()`.

Khi xong việc với vùng bộ nhớ, giải phóng nó bằng một lời gọi `free()`.

Đừng truyền `0` cho size. Nó chắc không làm gì bạn muốn đâu.

Nếu bạn thắc mắc, tất cả bộ nhớ cấp phát động đều được hệ thống tự động giải phóng khi chương trình kết thúc. Dù vậy, việc `free()` rõ ràng mọi thứ bạn đã cấp phát được coi là *Good Form* (phong cách tốt). Cách này các lập trình viên khác sẽ không nghĩ bạn cẩu thả.

Giá trị trả về

Cả hai hàm trả về con trỏ tới bộ nhớ mới toanh, sáng bóng vừa được cấp phát. Hoặc `NULL` nếu có chuyện gì đó xảy ra.

Ví dụ

So sánh `malloc()` và `calloc()` khi cấp phát 5 `int`:

```
#include <stdlib.h>

int main(void)
{
    // Cấp phát chỗ cho 5 int
    int *p = malloc(5 * sizeof(int));

    p[0] = 12;
    p[1] = 30;

    // Cấp phát chỗ cho 5 int
    // (Đồng thời xoá bộ nhớ đó về 0)
    int *q = calloc(5, sizeof(int));

    q[0] = 12;
    q[1] = 30;

    // Xong rồi
    free(p);
    free(q);
}
```

Xem thêm

`aligned_alloc()`, `free()`

24.10 `free()`

Giải phóng một vùng bộ nhớ

Synopsis

```
#include <stdlib.h>

void free(void *ptr);
```

Mô tả

Bạn biết con trỏ bạn nhận được từ `malloc()`, `calloc()`, hoặc `aligned_alloc()` chứ? Bạn truyền con trỏ đó cho `free()` để giải phóng bộ nhớ gắn với nó.

Nếu bạn không làm vậy, bộ nhớ sẽ vẫn được cấp phát MÃI MÃI! (Ồ, cho đến khi chương trình thoát, dù sao đi nữa.)

Sự thật thú vị: `free(NULL)` không làm gì cả. Bạn có thể gọi nó an toàn. Đôi khi điều đó tiện.

Đừng `free()` một con trỏ đã được `free()` trước đó. Đừng `free()` một con trỏ mà bạn không nhận được từ một trong các hàm cấp phát. Như vậy sẽ là *Bad*[11].

Giá trị trả về

Không trả về gì!

Ví dụ

```
#include <stdlib.h>

int main(void)
{
    // Cấp phát chỗ cho 5 int
    int *p = malloc(5 * sizeof(int));

    p[0] = 12;
    p[1] = 30;

    // Giải phóng chỗ đó
    free(p);
}
```

Xem thêm

`malloc()`, `calloc()`, `aligned_alloc()`

24.11 `realloc()`

Đổi kích thước một vùng bộ nhớ đã cấp phát trước đó

[11] "Thử tưởng tượng mọi sự sống như bạn biết dừng ngay lập tức và mọi phân tử trong cơ thể bạn nổ tung ở tốc độ ánh sáng." —Egon Spengler

Synopsis

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size);
```

Mô tả

Hàm này lấy một con trỏ tới vùng bộ nhớ đã cấp phát trước đó với `malloc()` hoặc `calloc()` và đổi kích thước nó thành kích thước mới.

Nếu kích thước mới nhỏ hơn kích thước cũ, phần dữ liệu vượt quá kích thước mới sẽ bị loại bỏ.

Nếu kích thước mới lớn hơn kích thước cũ, phần lớn hơn mới sẽ không được khởi tạo. (Bạn có thể xoá nó bằng `memset()`.)

Lưu ý quan trọng: bộ nhớ có thể bị di chuyển! Nếu bạn đổi kích thước, hệ thống có thể cần chuyển bộ nhớ sang một khối liên tục lớn hơn. Nếu chuyện này xảy ra, `realloc()` sẽ sao chép dữ liệu cũ sang vị trí mới cho bạn.

Vì chuyện này, việc lưu giá trị trả về vào con trỏ của bạn để cập nhật vị trí mới khi có di chuyển là quan trọng. (Cũng nhớ kiểm tra lỗi để không ghi đè con trỏ cũ bằng `NULL`, gây rò rỉ bộ nhớ.)

Bạn cũng có thể `relloc()` bộ nhớ cấp phát bởi `aligned_alloc()`, nhưng nó có thể mất căn chỉnh nếu khối bị di chuyển.

Giá trị trả về

Trả về con trỏ tới vùng bộ nhớ đã đổi kích thước. Nó có thể bằng với `ptr` truyền vào, hoặc có thể ở vị trí khác.

Ví dụ

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

int main(void)
{
    // Cấp phát chỗ cho 5 int
    int *p = malloc(5 * sizeof(int));

    p[0] = 12;
    p[1] = 30;

    // Realloc cho 10 byte
    int *new_p = realloc(p, 10 * sizeof(int));

    if (new_p == NULL) {
        printf("Error reallocing\n");
    } else {
        p = new_p;  // Ổn rồi; giữ lại thôi
        p[7] = 99;
    }

    // Xong
    free(p);
}
```

Xem thêm

`malloc()`, `calloc()`

24.12 `abort()`

Kết thúc chương trình đột ngột

Synopsis

```
#include <stdlib.h>

_Noreturn void abort(void);
```

Mô tả

Hàm này kết thúc việc thực thi chương trình *không bình thường* và ngay lập tức. Dùng nó trong những tình huống hiếm và bất ngờ.

Các stream đang mở có thể không được flush. File tạm đã tạo có thể không bị xoá. Exit handler không được gọi.

Mã thoát khác 0 được trả về môi trường.

Trên một số hệ thống, `abort()` có thể dump core[12], nhưng chuyện này nằm ngoài phạm vi spec.

Bạn có thể gây ra tương đương `abort()` bằng cách gọi `raise(SIGABRT)`, nhưng tôi không biết vì sao bạn lại làm vậy.

Cách portable duy nhất để dừng một lời gọi `abort()` giữa chừng là dùng `signal()` để bắt `SIGABRT` rồi `exit()` trong signal handler.

Giá trị trả về

Hàm này không bao giờ trả về.

Ví dụ

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

int main(void)
{
    int bad_thing = 1;

    if (bad_thing) {
        printf("This should never have happened!\n");
        fflush(stdout);  // Đảm bảo thông điệp ra được
        abort();
    }
}
```

Trên máy tôi, output là:

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

```
This should never have happened!
zsh: abort (core dumped)  ./foo
```

Xem thêm

`signal()`

24.13 `atexit()`, `at_quick_exit()`

Đăng ký handler chạy khi chương trình thoát

Synopsis

```
#include <stdlib.h>

int atexit(void (*func)(void));

int at_quick_exit(void (*func)(void));
```

Mô tả

Khi chương trình thoát bình thường bằng `exit()` hoặc `return` từ `main()`, nó sẽ tìm các handler đã đăng ký trước đó để gọi trên đường ra. Các handler này được đăng ký bằng lời gọi `atexit()`.

Cứ nghĩ nó như là, "Này, lúc chuẩn bị thoát, làm mấy việc thêm này nhé."

Với lời gọi `quick_exit()`, bạn có thể dùng hàm `at_quick_exit()` để đăng ký handler cho nó[13]. Không có sự chồng chéo trong handler giữa `exit()` và `quick_exit()`, tức là khi gọi một hàm, không handler nào của hàm kia được kích hoạt.

Bạn có thể đăng ký nhiều handler để chạy—ít nhất 32 handler được hỗ trợ bởi cả `exit()` và `quick_exit()`.

Tham số `func` trong các hàm nhìn hơi lạ—nó là con trỏ tới một hàm để gọi. Về cơ bản cứ đặt tên hàm cần gọi vào đó (không có dấu ngoặc đằng sau). Xem ví dụ bên dưới.

Nếu bạn gọi `atexit()` từ bên trong handler `atexit()` của bạn (hoặc tương đương trong handler `at_quick_exit()`), thì không rõ liệu nó có được gọi hay không. Vậy nên hãy đăng ký tất cả trước khi thoát.

Khi thoát, các hàm sẽ được gọi theo thứ tự ngược với thứ tự chúng được đăng ký.

Giá trị trả về

Các hàm này trả về `0` khi thành công, hoặc khác 0 khi thất bại.

Ví dụ

`atexit()`:

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

void exit_handler_1(void)
```

[13] `quick_exit()` khác `exit()` ở chỗ file đang mở có thể không được flush và file tạm có thể không bị xoá.

```
{
    printf("Exit handler 1 called!\n");
}

void exit_handler_2(void)
{
    printf("Exit handler 2 called!\n");
}

int main(void)
{
    atexit(exit_handler_1);
    atexit(exit_handler_2);

    exit(0);
}
```

Output:

```
Exit handler 2 called!
Exit handler 1 called!
```

Và một ví dụ tương tự với `quick_exit()`:

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

void exit_handler_1(void)
{
    printf("Exit handler 1 called!\n");
}

void exit_handler_2(void)
{
    printf("Exit handler 2 called!\n");
}

int main(void)
{
    at_quick_exit(exit_handler_1);
    at_quick_exit(exit_handler_2);

    quick_exit(0);
}
```

Xem thêm

`exit()`, `quick_exit()`

24.14 `exit()`, `quick_exit()`, `_Exit()`

Thoát chương trình đang chạy

Synopsis

```
#include <stdlib.h>

_Noreturn void exit(int status);

_Noreturn void quick_exit(int status);

_Noreturn void _Exit(int status);
```

Mô tả

Tất cả các hàm này khiến chương trình thoát, với các mức dọn dẹp khác nhau.

`exit()` dọn dẹp nhiều nhất và là cách thoát bình thường nhất.

`quick_exit()` là cách thứ hai về mức độ dọn dẹp.

`_Exit()` bỏ hết mọi thứ không một chút thủ tục và ragequit tại chỗ.

Gọi `exit()` hoặc `quick_exit()` khiến các handler tương ứng `atexit()` hoặc `at_quick_exit()` được gọi theo thứ tự ngược với thứ tự chúng được đăng ký.

`exit()` sẽ flush tất cả stream và xoá tất cả file tạm.

`quick_exit()` hoặc `_Exit()` có thể không làm các thao tác lịch sự đó.

`_Exit()` cũng không gọi các at-exit handler nào cả.

Với tất cả các hàm, mã thoát `status` được trả về môi trường.

Các mã thoát được định nghĩa:

Status	Mô tả
`EXIT_SUCCESS`	Thường trả về khi chuyện tốt đẹp xảy ra
`0`	Giống `EXIT_SUCCESS`
`EXIT_FAILURE`	Ôi không! Chắc chắn là thất bại!
Giá trị dương bất kỳ	Thường chỉ một loại thất bại khác

Lưu ý OS X: `quick_exit()` không được hỗ trợ.

Giá trị trả về

Không hàm nào trong số này từng trả về.

Ví dụ

```
#include <stdlib.h>

int main(void)
{
    int contrived_exit_type = 1;

    switch(contrived_exit_type) {
        case 1:
            exit(EXIT_SUCCESS);

        case 2:
```

```
            // Không được hỗ trợ trong OS X
            quick_exit(EXIT_SUCCESS);

        case 3:
            _Exit(2);
    }
}
```

Xem thêm

`atexit()`, `at_quick_exit()`

24.15 `getenv()`

Lấy giá trị của một biến môi trường

Synopsis

```
#include <stdlib.h>

char *getenv(const char *name);
```

Mô tả

Environment (môi trường) thường cung cấp các biến được đặt trước khi chương trình chạy mà bạn có thể truy cập lúc runtime.

Tất nhiên chi tiết cụ thể tuỳ hệ thống, nhưng các biến này là cặp key/value, và bạn có thể lấy value bằng cách truyền key cho `getenv()` qua tham số `name`.

Bạn không được phép ghi đè chuỗi trả về.

Cái này khá hạn chế trong standard, nhưng OS của bạn thường cung cấp chức năng tốt hơn.

Giá trị trả về

Trả về con trỏ tới giá trị của biến môi trường, hoặc `NULL` nếu biến không tồn tại.

Ví dụ

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

int main(void)
{
    printf("PATH is %s\n", getenv("PATH"));
}
```

Output (bị cắt trong trường hợp của tôi):

```
PATH is /usr/bin:/usr/local/bin:/usr/sbin:/home/beej/.cargo/bin [...]
```

24.16 `system()`

Chạy một chương trình bên ngoài

Synopsis

```
#include <stdlib.h>

int system(const char *string);
```

Mô tả

Hàm này sẽ chạy một chương trình bên ngoài rồi trả về cho bên gọi.

Cách nó chạy chương trình là do hệ thống quy định, nhưng thường là bạn có thể truyền cho nó cái gì đó giống như bạn gõ trên dòng lệnh, tìm trong `PATH`, v.v.

Không phải hệ thống nào cũng có khả năng này, nhưng bạn có thể kiểm tra bằng cách truyền `NULL` cho `system()` và xem nó trả về 0 (không có command processor) hay khác 0 (có command processor! Yay!)

Nếu bạn đang lấy input từ người dùng và truyền nó cho `system()`, hãy *cực kỳ* cẩn thận escape tất cả ký tự đặc biệt của shell (tất cả ký tự không phải chữ cái/số) bằng dấu backslash để ngăn kẻ xấu chạy cái gì đó bạn không muốn.

Giá trị trả về

Nếu truyền `NULL`, trả về khác 0 nếu có command processor (tức là `system()` sẽ hoạt động).

Ngược lại trả về giá trị do implementation định nghĩa.

Ví dụ

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

int main(void)
{
    printf("Here's a directory listing:\n\n");

    system("ls -l");   // Chạy lệnh này và trả về`

    printf("\nAll done!\n");
}
```

Output:

```
Here's a directory listing:

total 92
drwxr-xr-x 3 beej beej  4096 Oct 14 21:38 bin
drwxr-xr-x 2 beej beej  4096 Dec 20 20:07 examples
-rwxr-xr-x 1 beej beej 16656 Feb 23 21:49 foo
-rw-rw-rw- 1 beej beej   155 Feb 23 21:49 foo.c
-rw-r--r-- 1 beej beej  1350 Jan 27 22:11 Makefile
-rw-r--r-- 1 beej beej  4644 Jan 18 09:12 README.md
drwxr-xr-x 3 beej beej  4096 Feb 23 20:21 src
drwxr-xr-x 6 beej beej  4096 Feb 21 20:24 stage
```

```
drwxr-xr-x 2 beej beej  4096 Sep 27 20:54 translations
drwxr-xr-x 2 beej beej  4096 Sep 27 20:54 website

All done!
```

24.17 `bsearch()`

Binary Search (có thể) trên một mảng đối tượng

Synopsis

```
#include <stdlib.h>

// Trước C23:

void *bsearch(const void *key, const void *base,
              size_t nmemb, size_t size,
              int (*compar)(const void *, const void *));
// C23:

QVoid *bsearch(const void *key, QVoid *base,
               size_t nmemb, size_t size,
               int (*compar)(const void *, const void *));
```

Mô tả

Hàm trông điên rồ này tìm kiếm một giá trị trong mảng.

Nó chắc là binary search hoặc một kiểu tìm kiếm nhanh, hiệu quả nào đó. Nhưng spec không nói rõ.

Tuy nhiên, mảng phải được sắp xếp! Nên binary search có vẻ khả thi.

- `key` là con trỏ tới giá trị cần tìm.
- `base` là con trỏ tới đầu mảng—mảng phải được sắp xếp!
- `nmemb` là số phần tử trong mảng.
- `size` là `sizeof` của mỗi phần tử trong mảng.
- `compar` là con trỏ tới hàm so sánh key với các giá trị khác.

Hàm so sánh nhận key làm đối số thứ nhất và giá trị cần so sánh làm đối số thứ hai. Nó nên trả về số âm nếu key nhỏ hơn giá trị, `0` nếu key bằng giá trị, và số dương nếu key lớn hơn giá trị.

Kiểu này thường được tính bằng cách lấy hiệu giữa key và giá trị cần so sánh. Nếu phép trừ được hỗ trợ.

Giá trị trả về từ hàm `strcmp()` có thể dùng để so sánh chuỗi.

Lần nữa, mảng phải được sắp xếp theo thứ tự của hàm so sánh trước khi chạy `bsearch()`. May cho bạn, bạn chỉ cần gọi `qsort()` với cùng hàm so sánh để lo việc này.

Nó là hàm đa dụng—nó sẽ tìm bất kỳ kiểu mảng nào cho bất cứ thứ gì. Đánh đổi là bạn phải viết hàm so sánh.

Và chuyện đó không đáng sợ như nhìn. Nhảy xuống ví dụ.

Giá trị trả về

Hàm trả về con trỏ tới giá trị tìm được, hoặc `NULL` nếu không tìm thấy.

Mới trong C23: Nếu `base` là `const`, kiểu trả về của hàm `bsearch()` cũng sẽ là `const`.

Ví dụ

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

int compar(const void *key, const void *value)
{
    const int *k = key, *v = value;  // Cần int, không phải void

    return *k - *v;
}

int main(void)
{
    int a[9] = {2, 6, 9, 12, 13, 18, 20, 32, 47};

    int *r, key;

    key = 12;  // 12 có trong mảng
    r = bsearch(&key, a, 9, sizeof(int), compar);
    printf("Found %d\n", *r);

    key = 30;  // Không tìm thấy 30
    r = bsearch(&key, a, 9, sizeof(int), compar);
    if (r == NULL)
        printf("Didn't find 30\n");

    // Tìm với key không tên, con trỏ tới 32
    r = bsearch(&(int){32}, a, 9, sizeof(int), compar);
    printf("Found %d\n", *r);  // Tìm thấy
}
```

Output:

```
Found 12
Didn't find 30
Found 32
```

Xem thêm

`strcmp()`, `qsort()`

24.18 `qsort()`

Quicksort (có thể) dữ liệu

Synopsis

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Mô tả

Hàm này sẽ quicksort (hoặc một kiểu sắp xếp khác, chắc cũng nhanh) một mảng dữ liệu tại chỗ[14].

Giống `bsearch()`, nó không biết gì về dữ liệu. Bất kỳ dữ liệu nào có thể định nghĩa thứ tự tương đối đều có thể sắp xếp, dù là `int`, `struct`, hay bất kỳ thứ gì khác.

Cũng giống `bsearch()`, bạn phải cho một hàm so sánh để thực hiện việc so sánh thực sự.

- `base` là con trỏ tới đầu mảng cần sắp xếp.
- `nmemb` là số phần tử trong mảng.
- `size` là `sizeof` của mỗi phần tử.
- `compar` là con trỏ tới hàm so sánh.

Hàm so sánh nhận con trỏ tới hai phần tử của mảng làm đối số và so sánh chúng. Nó nên trả về số âm nếu đối số thứ nhất nhỏ hơn đối số thứ hai, `0` nếu chúng bằng nhau, và số dương nếu đối số thứ nhất lớn hơn đối số thứ hai.

Kiểu này thường được tính bằng cách lấy hiệu giữa đối số thứ nhất và đối số thứ hai. Nếu phép trừ được hỗ trợ.

Giá trị trả về từ hàm `strcmp()` có thể cung cấp thứ tự sắp xếp cho chuỗi.

Nếu bạn phải sắp xếp một `struct`, chỉ việc trừ theo trường cụ thể mà bạn muốn sắp xếp theo.

Hàm so sánh này có thể được dùng bởi `bsearch()` để tìm kiếm sau khi danh sách đã được sắp xếp.

Để đảo ngược thứ tự sắp xếp, trừ đối số thứ hai cho đối số thứ nhất, tức là đảo dấu giá trị trả về từ `compar()`.

Giá trị trả về

Không trả về gì!

Ví dụ

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

int compar(const void *elem0, const void *elem1)
{
    const int *x = elem0, *y = elem1;  // Cần int, không phải void

    if (*x > *y) return 1;
    if (*x < *y) return -1;
    return 0;
}

int main(void)
{
    int a[9] = {14, 2, 3, 17, 10, 8, 6, 1, 13};

    // Sắp xếp danh sách

    qsort(a, 9, sizeof(int), compar);

    // In danh sách đã sắp xếp

    for (int i = 0; i < 9; i++)
        printf("%d ", a[i]);
```

[14] "Tại chỗ" (in-place) nghĩa là mảng gốc sẽ chứa kết quả; không có mảng mới nào được cấp phát.

```
    putchar('\n');

    // Dùng cùng hàm compar() để binary search
    // cho 17 (truyền vào như một đối tượng không tên)

    int *r = bsearch(&(int){17}, a, 9, sizeof(int), compar);
    printf("Found %d!\n", *r);
}
```

Output:

```
1 2 3 6 8 10 13 14 17
Found 17!
```

Xem thêm

`strcmp()`, `bsearch()`

24.19 `abs()`, `labs()`, `llabs()`

Tính giá trị tuyệt đối của một số nguyên

Synopsis

```
#include <stdlib.h>

int abs(int j);

long int labs(long int j);

long long int llabs(long long int j);
```

Mô tả

Tính giá trị tuyệt đối của `j`. Nếu bạn không nhớ, đó là khoảng cách từ `j` đến 0.

Nói cách khác, nếu `j` âm, trả về nó dưới dạng dương. Nếu nó dương, trả về nó dưới dạng dương. Lúc nào cũng dương. Tận hưởng cuộc sống đi.

Nếu kết quả không thể biểu diễn được, hành vi là không xác định. Đặc biệt để ý nửa trên của các số unsigned.

Giá trị trả về

Trả về giá trị tuyệt đối của `j`, $|j|$.

Ví dụ

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

int main(void)
{
    printf("|-2| = %d\n", abs(-2));
```

```
    printf("|4|  = %d\n", abs(4));
}
```

Output:

```
|-2| = 2
|4|  = 4
```

Xem thêm

`fabs()`

24.20 `div()`, `ldiv()`, `lldiv()`

Tính thương và phần dư của hai số

Synopsis

```
#include <stdlib.h>

div_t div(int numer, int denom);

ldiv_t ldiv(long int numer, long int denom);

lldiv_t lldiv(long long int numer, long long int denom);
```

Mô tả

Các hàm này cho bạn thương và phần dư của một cặp số trong một lần.

Chúng trả về một cấu trúc có hai trường, `quot` và `rem`, kiểu của chúng khớp với kiểu của `numer` và `denom`. Để ý mỗi hàm trả về một biến thể khác của `div_t`.

Các biến thể `div_t` này tương đương với:

```
typedef struct {
    int quot, rem;
} div_t;

typedef struct {
    long int quot, rem;
} ldiv_t;

typedef struct {
    long long int quot, rem;
} lldiv_t;
```

Tại sao lại dùng mấy cái này thay vì toán tử chia?

C99 Rationale nói:

> Vì C89 có ngữ nghĩa do implementation định nghĩa cho phép chia số nguyên có dấu khi có toán hạng âm, `div` và `ldiv`, và `lldiv` trong C99, được phát minh để cung cấp ngữ nghĩa được xác định rõ cho chia và phép dư số nguyên có dấu. Ngữ nghĩa được lấy giống như trong Fortran. Vì các hàm này trả về cả thương và phần dư, chúng cũng đóng vai trò là cách tiện lợi để mô hình

> hiệu quả phần cứng bên dưới tính cả hai kết quả như một phần của cùng thao tác. Bảng 7.2 tóm tắt ngữ nghĩa của các hàm này.

Thật vậy, K&R2 (C89) nói:

> Hướng truncation cho `/` và dấu của kết quả cho `%` phụ thuộc máy với toán hạng âm [...]

Rationale sau đó tiếp tục nói rõ dấu của thương và phần dư sẽ là gì cho trước dấu của tử và mẫu khi dùng các hàm `div()`:

`numer`	`denom`	`quot`	`rem`
+	+	+	+
−	+	−	−
+	−	−	+
−	−	+	−

Giá trị trả về

Một cấu trúc `div_t`, `ldiv_t`, hoặc `lldiv_t` với các trường `quot` và `rem` chứa thương và phần dư của phép toán `numer/denom`.

Ví dụ

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

int main(void)
{
    div_t d = div(64, -7);

    printf("64 / -7 = %d\n", d.quot);
    printf("64 %% -7 = %d\n", d.rem);
}
```

Output:

```
64 / -7 = -9
64 % -7 = 1
```

Xem thêm

`fmod()`, `remainder()`

24.21 `mblen()`

Trả về số byte của một ký tự multibyte

Synopsis

```
#include <stdlib.h>

int mblen(const char *s, size_t n);
```

Mô tả

Nếu bạn có một ký tự multibyte trong một chuỗi, hàm này sẽ cho bạn biết nó dài bao nhiêu byte.

`n` là số byte tối đa `mblen()` sẽ quét trước khi bỏ cuộc.

Nếu `s` là con trỏ `NULL`, kiểm tra xem encoding này có phụ thuộc trạng thái không, như đã nói ở phần giá trị trả về bên dưới. Nó cũng reset trạng thái, nếu có.

Hành vi của hàm này chịu ảnh hưởng bởi locale.

Giá trị trả về

Trả về số byte được dùng để mã hoá ký tự này, hoặc `-1` nếu không có ký tự multibyte hợp lệ trong `n` byte tiếp theo.

Hoặc, nếu `s` là NULL, trả về true nếu encoding này có phụ thuộc trạng thái.

Ví dụ

Cho ví dụ, tôi dùng bộ ký tự mở rộng để đặt ký tự Unicode vào source. Nếu cái này không hoạt động với bạn, dùng escape `\uXXXX`.

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

int main(void)
{
    setlocale(LC_ALL, "");

    printf("State dependency: %d\n", mblen(NULL, 0));
    printf("Bytes for €: %d\n", mblen("€", 5));
    printf("Bytes for \u00e9: %d\n", mblen("\u00e9", 5));  // \u00e9 == é
    printf("Bytes for &: %d\n", mblen("&", 5));
}
```

Output (trong trường hợp của tôi, encoding là UTF-8, nhưng của bạn có thể khác):

```
State dependency: 0
Bytes for €: 3
Bytes for é: 2
Bytes for &: 1
```

Xem thêm

`mbtowc()`, `mbstowcs())`, `setlocale()`

24.22 `mbtowc()`

Chuyển một ký tự multibyte thành ký tự rộng

Synopsis

```
#include <stdlib.h>

int mbtowc(wchar_t * restrict pwc, const char * restrict s, size_t n);
```

Mô tả

Nếu bạn có một ký tự multibyte, hàm này sẽ chuyển nó thành ký tự rộng và lưu tại địa chỉ do `pwc` trỏ tới. Tối đa `n` byte của ký tự multibyte sẽ được phân tích.

Nếu `pwc` là `NULL`, ký tự kết quả sẽ không được lưu. (Hữu ích khi chỉ muốn lấy giá trị trả về.)

Nếu `s` là con trỏ `NULL`, kiểm tra xem encoding này có phụ thuộc trạng thái không, như đã nói ở phần giá trị trả về bên dưới. Nó cũng reset trạng thái, nếu có.

Hành vi của hàm này chịu ảnh hưởng bởi locale.

Giá trị trả về

Trả về số byte được dùng trong ký tự rộng đã mã hoá, hoặc `-1` nếu không có ký tự multibyte hợp lệ trong `n` byte tiếp theo.

Trả về `0` nếu `s` trỏ tới ký tự NUL.

Hoặc, nếu `s` là NULL, trả về true nếu encoding này có phụ thuộc trạng thái.

Ví dụ

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>

int main(void)
{
    setlocale(LC_ALL, "");

    printf("State dependency: %d\n", mbtowc(NULL, NULL, 0));

    wchar_t wc;
    int bytes;

    bytes = mbtowc(&wc, "€", 5);

    printf("L'%lc' takes %d bytes as multibyte char '€'\n", wc, bytes);
}
```

Output trên máy tôi:

```
State dependency: 0
L'€' takes 3 bytes as multibyte char '€'
```

Xem thêm

`mblen()`, `mbstowcs()`, `wcstombs()`, `setlocale()`

24.23 `wctomb()`

Chuyển ký tự rộng thành ký tự multibyte

Synopsis

```
#include <stdlib.h>

int wctomb(char *s, wchar_t wc);
```

Mô tả

Nếu bạn đang nắm trong tay một ký tự rộng, bạn có thể dùng cái này để biến nó thành multibyte.

Ký tự rộng `wc` được lưu dưới dạng ký tự multibyte trong chuỗi được trỏ bởi `s`. Buffer mà `s` trỏ tới nên dài ít nhất `MB_CUR_MAX` ký tự. Lưu ý `MB_CUR_MAX` thay đổi theo locale.

Nếu `wc` là ký tự rộng NUL, một NUL sẽ được lưu vào `s` sau các byte cần thiết để reset shift state (nếu có).

Nếu `s` là con trỏ `NULL`, kiểm tra xem encoding này có phụ thuộc trạng thái không, như đã nói ở phần giá trị trả về bên dưới. Nó cũng reset trạng thái, nếu có.

Hành vi của hàm này chịu ảnh hưởng bởi locale.

Giá trị trả về

Trả về số byte được dùng trong ký tự multibyte đã mã hoá, hoặc `-1` nếu `wc` không tương ứng với ký tự multibyte hợp lệ nào.

Hoặc, nếu `s` là NULL, trả về true nếu encoding này có phụ thuộc trạng thái.

Ví dụ

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>

int main(void)
{
    setlocale(LC_ALL, "");

    printf("State dependency: %d\n", mbtowc(NULL, NULL, 0));

    int bytes;
    char mb[MB_CUR_MAX + 1];

    bytes = wctomb(mb, L'€');
    mb[bytes] = '\0';

    printf("L'€' takes %d bytes as multibyte char '%s'\n", bytes, mb);
}
```

Output trên máy tôi:

```
State dependency: 0
L'€' takes 3 bytes as multibyte char '€'
```

Xem thêm

`mbtowc()`, `mbstowcs()`, `wcstombs()`, `setlocale()`

24.24 `mbstowcs()`

Chuyển chuỗi multibyte thành chuỗi ký tự rộng

Synopsis

```
#include <stdlib.h>

size_t mbstowcs(wchar_t * restrict pwcs, const char * restrict s, size_t n);
```

Mô tả

Nếu bạn có một chuỗi multibyte (hay chuỗi thường), bạn có thể chuyển nó thành chuỗi ký tự rộng với hàm này.

Tối đa `n` ký tự rộng được ghi vào đích `pwcs` từ nguồn `s`.

Ký tự NUL được lưu dưới dạng ký tự NUL rộng.

Phần mở rộng POSIX không portable: nếu bạn đang dùng một thư viện tuân thủ POSIX, hàm này cho phép `pwcs` là `NULL` nếu bạn chỉ quan tâm tới giá trị trả về. Đáng chú ý nhất là, việc này sẽ cho bạn số ký tự trong chuỗi multibyte (trái ngược với `strlen()` đếm số byte.)

Giá trị trả về

Trả về số ký tự rộng được ghi vào đích `pwcs`.

Nếu tìm thấy ký tự multibyte không hợp lệ, trả về `(size_t)(-1)`.

Nếu giá trị trả về là `n`, nghĩa là kết quả *không* được kết thúc bằng NUL.

Ví dụ

Source này dùng bộ ký tự mở rộng. Nếu compiler của bạn không hỗ trợ, bạn sẽ phải thay chúng bằng escape `\u`.

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

int main(void)
{
    setlocale(LC_ALL, "");

    wchar_t wcs[128];
    char *s = "€200 for this spoon?";  // 20 ký tự

    size_t char_count, byte_count;

    char_count = mbstowcs(wcs, s, 128);
    byte_count = strlen(s);
```

```
    printf("Wide string: L\"%ls\"\n", wcs);
    printf("Char count : %zu\n", char_count);    // 20
    printf("Byte count : %zu\n\n", byte_count);  // 22 trên máy tôi

    // Phần mở rộng POSIX cho phép bạn truyền NULL cho
    // đích để chỉ dùng giá trị trả về (là số ký tự của
    // chuỗi, nếu không có lỗi xảy ra)

    s = "§¶°±π€•";  // 7 ký tự

    char_count = mbstowcs(NULL, s, 0);  // Chỉ POSIX, không portable
    byte_count = strlen(s);

    printf("Multibyte str: \"%s\"\n", s);
    printf("Char count   : %zu\n", char_count);  // 7
    printf("Byte count   : %zu\n", byte_count);  // 16 trên máy tôi
}
```

Output trên máy tôi (số byte sẽ tuỳ thuộc vào encoding của bạn):

```
Wide string: L"€200 for this spoon?"
Char count : 20
Byte count : 22

Multibyte str: "§¶°±π€•"
Char count   : 7
Byte count   : 16
```

Xem thêm

`mblen()`, `mbtowc()`, `wcstombs()`, `setlocale()`

24.25 `wcstombs()`

Chuyển chuỗi ký tự rộng thành chuỗi multibyte

Synopsis

```
#include <stdlib.h>

size_t wcstombs(char * restrict s, const wchar_t * restrict pwcs, size_t n);
```

Mô tả

Nếu bạn có chuỗi ký tự rộng và muốn biến nó thành chuỗi multibyte, đây là hàm cho bạn!

Nó sẽ lấy các ký tự rộng được `pwcs` trỏ tới và chuyển chúng thành ký tự multibyte lưu vào `s`. Không quá `n` byte sẽ được ghi vào `s`.

Phần mở rộng POSIX không portable: nếu bạn đang dùng một thư viện tuân thủ POSIX, hàm này cho phép `s` là `NULL` nếu bạn chỉ quan tâm tới giá trị trả về. Đáng chú ý nhất, việc này sẽ cho bạn số byte cần thiết để mã hoá các ký tự rộng thành chuỗi multibyte.

Giá trị trả về

Trả về số byte được ghi vào `s`, hoặc `(size_t)(-1)` nếu một trong các ký tự không thể mã hoá thành chuỗi multibyte.

Nếu giá trị trả về là `n`, nghĩa là kết quả *không* được kết thúc bằng NUL.

Ví dụ

Source này dùng bộ ký tự mở rộng. Nếu compiler của bạn không hỗ trợ, bạn sẽ phải thay chúng bằng escape `\u`.

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

int main(void)
{
    setlocale(LC_ALL, "");

    char mbs[128];
    wchar_t *wcs = L"€200 for this spoon?";  // 20 ký tự

    size_t byte_count;

    byte_count = wcstombs(mbs, wcs, 128);

    printf("Wide string: L\"%ls\"\n", wcs);
    printf("Multibyte  : \"%s\"\n", mbs);
    printf("Byte count : %zu\n\n", byte_count);  // 22 trên máy tôi

    // Phần mở rộng POSIX cho phép bạn truyền NULL cho
    // đích để chỉ dùng giá trị trả về (là số ký tự của
    // chuỗi, nếu không có lỗi xảy ra)

    wcs = L"§¶°±π€•";  // 7 ký tự

    byte_count = wcstombs(NULL, wcs, 0);  // Chỉ POSIX, không portable

    printf("Wide string: L\"%ls\"\n", wcs);
    printf("Byte count : %zu\n", byte_count);  // 16 trên máy tôi
}
```

Output trên máy tôi (số byte sẽ tuỳ thuộc vào encoding của bạn):

```
Wide string: L"€200 for this spoon?"
Multibyte  : "€200 for this spoon?"
Byte count : 22

Wide string: L"§¶°±π€•"
Byte count : 16
```

Xem thêm

`mblen()`, `wctomb()`, `mbstowcs()`, `setlocale()`

24.26 `memalignment()`

Synopsis

Mới trong C23!

```
#include <stdlib.h>

size_t memalignment(const void * p);
```

Mô tả

Hàm này sẽ cho bạn biết memory alignment (căn chỉnh bộ nhớ) (tính bằng byte) của đối tượng được trỏ tới.

Ý tưởng là chúng ta sẽ lấy được cùng thông tin như khi dùng `alignof`, chỉ khác là chúng ta có thể làm việc đó với `void*` không có kiểu.

Và bạn có thể muốn làm điều này vì dữ liệu căn chỉnh sai có thể chậm hơn hoặc không dùng được trên nhiều nền tảng.

Giá trị trả về

Trả về căn chỉnh dạng số nguyên của `p`, sẽ là luỹ thừa của 2. Trả về `0` nếu truyền vào `NULL`.

`0` nghĩa là con trỏ không thể dùng để truy cập đối tượng của bất kỳ kiểu nào.

Ví dụ

(Lưu ý: chưa có compiler nào của tôi hỗ trợ hàm này, nên đoạn code phần lớn chưa được test.)

Đề xuất cho tính năng này gợi ý rằng một macro có thể hữu ích để xác định xem con trỏ có căn chỉnh tốt cho một kiểu cụ thể hay không, nên chúng ta sẽ ăn cắp cái đó làm ví dụ:

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

#define isaligned(P, T) (memalignment (P) >= alignof(T))

void check(void *p)
{
    printf("%lu %lu\n", memalignment(p), alignof(int));
    if (isaligned(p, int)) {
        puts("The pointer p is well-aligned for an int! :)");

        // Nên tôi thoải mái làm thế´này:

        int *i = p;
        *i = 3490;

    } else
        puts("The pointer p is poorly-aligned for an int! :(");
}

int main(void)
{
    char *p = malloc(10);
```

```
    check(p);
    check(p + 1);
}
```

Xem thêm

`alignof()`, `alignas()`, `max_align_t`

Chapter 25

`<stdnoreturn.h>` Macro Cho Hàm Không Trả Về

Header này cung cấp một macro `noreturn` là bí danh tiện tay cho `_Noreturn`.

Dùng macro này để báo cho compiler biết một hàm sẽ không bao giờ trả về chỗ gọi. Undefined behavior nếu hàm được đánh dấu như vậy mà vẫn trả về.

Ví dụ sử dụng:

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

noreturn void foo(void) // Hàm này không bao giờ nên return!
{
    printf("Happy days\n");

    exit(1);            // Và nó không return--nó exit ở đây!
}

int main(void)
{
    foo();
}
```

Có vậy thôi.

Chapter 26

`<string.h>` Thao Tác Chuỗi

Hàm	Mô tả
`memccpy()`	Sao chép một vùng bộ nhớ sang vùng khác, dừng lại tại một ký tự được chỉ định.
`memchr()`	Tìm lần xuất hiện đầu tiên của một ký tự trong bộ nhớ.
`memcmp()`	So sánh hai vùng bộ nhớ.
`memcpy()`	Sao chép một vùng bộ nhớ sang vùng khác.
`memmove()`	Di chuyển một vùng bộ nhớ (có thể chồng nhau).
`memset()`	Đặt một vùng bộ nhớ về một giá trị.
`memset_explicit()`	Đặt một vùng bộ nhớ về một giá trị theo cách không thể bị tối ưu hoá bỏ đi.
`strcat()`	Concatenate (nối) hai chuỗi lại với nhau.
`strchr()`	Tìm lần xuất hiện đầu tiên của một ký tự trong chuỗi.
`strcmp()`	So sánh hai chuỗi.
`strcoll()`	So sánh hai chuỗi có tính đến locale.
`strcpy()`	Sao chép một chuỗi.
`strcspn()`	Tìm độ dài của một chuỗi không chứa một tập các ký tự nào đó.
`strdup()`	Nhân bản một chuỗi trên heap.
`strerror()`	Trả về thông báo lỗi dễ đọc ứng với một mã cho trước.
`strlen()`	Trả về độ dài của một chuỗi.
`strncat()`	Concatenate (nối) hai chuỗi, có giới hạn độ dài.
`strncmp()`	So sánh hai chuỗi, có giới hạn độ dài.
`strncpy()`	Sao chép hai chuỗi, có giới hạn độ dài.
`strndup()`	Nhân bản một chuỗi trên heap, có giới hạn độ dài.
`strpbrk()`	Tìm trong chuỗi một trong các ký tự của một tập hợp.
`strrchr()`	Tìm lần xuất hiện cuối cùng của một ký tự trong chuỗi.
`strspn()`	Tìm độ dài của một chuỗi chỉ gồm một tập các ký tự.
`strstr()`	Tìm một chuỗi con trong một chuỗi.
`strtok()`	Tách chuỗi thành các token.
`strxfrm()`	Chuẩn bị một chuỗi để so sánh như thể bằng `strcoll()`.

Như đã nhắc từ sớm trong guide, một string (chuỗi) trong C là một chuỗi các byte trong bộ nhớ, được kết thúc bằng một ký tự NUL (‘`\0`’). Cái NUL ở cuối rất quan trọng, vì nó cho tất cả các hàm chuỗi này (và `printf()` và `puts()` và mọi thứ khác có dính dáng đến chuỗi) biết điểm kết thúc thực sự của chuỗi ở đâu.

May thay, khi bạn thao tác với một chuỗi bằng một trong rất nhiều hàm có sẵn này, chúng sẽ tự thêm null terminator (kết thúc bằng NUL) giúp bạn, nên thực ra hiếm khi bạn phải tự theo dõi nó. (Đôi lúc bạn cũng phải làm, đặc biệt nếu bạn đang tự tay dựng một chuỗi từ đầu bằng cách thêm từng ký tự một hoặc đại loại vậy.)

Trong mục này bạn sẽ tìm thấy các hàm để lấy chuỗi con ra khỏi chuỗi, concatenate (nối) các chuỗi với nhau, lấy độ dài (length) của một chuỗi, vân vân và mây mây.

26.1 `memcpy()`, `memccpy()`, `memmove()`

Sao chép byte từ vị trí này sang vị trí khác trong bộ nhớ

Synopsis

`memccpy()` là mới trong C23!

```
#include <string.h>

void *memcpy(void * restrict s1, const void * restrict s2, size_t n);

void *memccpy(void * restrict s1, const void * restrict s2, int c,
              size_t n);

void *memmove(void *s1, const void *s2, size_t n);
```

Mô tả

Mấy hàm này sao chép bộ nhớ—bao nhiêu byte bạn muốn cũng được! Từ nguồn sang đích, với số byte mà bạn chỉ định!

`memccpy()` cho bạn thêm một tuỳ chọn: dừng lại sau khi gặp một ký tự nhất định trong nguồn.

Khác biệt chính giữa các biến thể `memcpy()` và `memmove()` là cái trước không thể sao chép an toàn các vùng bộ nhớ overlap (chồng) nhau, còn cái sau thì có.

Tôi không chắc vì sao bạn lại muốn dùng `memcpy()` thay vì `memmove()`, nhưng tôi đoán chắc nó nhanh hơn một chút.

Các tham số có thứ tự đặc biệt: đích trước, rồi đến nguồn. Tôi nhớ thứ tự này vì nó giống như một phép gán “`=`”: đích nằm ở bên trái.

Giá trị trả về

Cả hai hàm đều trả về đúng cái bạn truyền vào ở tham số `s1` cho tiện.

Ví dụ

```
#include <string.h>

int main(void)
{
    char s[100] = "Goats";
```

```
    char t[100];

    memcpy(t, s, 6);       // Sao chép bộ nhớ không chồng nhau

    memmove(s + 2, s, 6);  // Sao chép bộ nhớ chồng nhau
}
```

Xem thêm

`strcpy()`, `strncpy()`

26.2 `strcpy()`, `strncpy()`

Sao chép một chuỗi

Synopsis

```
#include <string.h>

char *strcpy(char *dest, char *src);

char *strncpy(char *dest, char *src, size_t n);
```

Mô tả

Mấy hàm này sao chép một chuỗi từ một địa chỉ sang địa chỉ khác, dừng lại tại null terminator trên chuỗi `src`.

`strncpy()` cũng giống `strcpy()`, chỉ khác là chỉ `n` ký tự đầu được sao chép thực sự. Cẩn thận nhé: nếu bạn chạm tới giới hạn `n` trước khi gặp null terminator trên chuỗi `src`, chuỗi `dest` của bạn sẽ không được NUL-terminated (kết thúc bằng NUL). Cẩn thận! CẨN THẬN!

(Nếu chuỗi `src` có ít hơn `n` ký tự, nó hoạt động đúng như `strcpy()`.)

Bạn có thể tự mình kết thúc chuỗi bằng cách nhét `'\0'` vào:

```
char s[10];
char foo = "My hovercraft is full of eels."; // nhiều hơn 10 ký tự

strncpy(s, foo, 9); // chỉ chép 9 ký tự vào vị trí 0-8
s[9] = '\0';        // vị trí 9 nhận ký tự kết thúc
```

Giá trị trả về

Cả hai hàm đều trả về `dest` cho tiện, không tính thêm phí gì.

Ví dụ

```
#include <string.h>

int main(void)
{
    char *src = "hockey hockey hockey hockey hockey hockey hockey hockey";
    char dest[20];
```

```
    int len;

    strcpy(dest, "I like "); // dest giờ là "I like "

    len = strlen(dest);

    // hơi hóc búa, nhưng ta dùng chút sô´học con trỏ đê^append
    // được càng nhiêù src vào cuôí dest càng tôt, -1 trong độ dài đê^
    // chừa chô~cho ký tự kêt thúc:
    strncpy(dest+len, src, sizeof(dest)-len-1);

    // nhớ rằng sizeof() trả vê`kích thước của mảng theo byte
    // và một char là một byte:
    dest[sizeof(dest)-1] = '\0'; // kêt thúc chuôĩ

    // dest giờ là:      v null terminator
    // I like hockey hocke
    // 012345678901234567890123456
}
```

Xem thêm

`memcpy()`, `strcat()`, `strncat()`

26.3 `strdup()`, `strndup()`

Nhân bản (duplicate) một chuỗi trên heap

Synopsis

Mới trong C23!

```
#include <string.h>

char *strdup(const char *s);

char *strndup(const char *s, size_t size);
```

Mô tả

Cái này giống `strcpy()`, chỉ khác là nó tự cấp phát chỗ cho chuỗi mới, như thể được làm bằng `malloc()`.

Con trỏ tới chuỗi nhân bản mà nó trả về nên được giải phóng bằng một lần gọi `free()` khi bạn dùng xong.

`strndup()` hoạt động tương tự, chỉ khác là bạn có thể giới hạn số byte được duplicate. `strndup()` luôn tạo ra một chuỗi NUL-terminated.

Giá trị trả về

Cả hai hàm đều trả về con trỏ tới chuỗi mới được tạo, hoặc `NULL` nếu không cấp phát được bộ nhớ cho bản nhân bản.

Ví dụ

Đây là ví dụ nhân bản một chuỗi rồi viết hoa chữ cái đầu của bản nhân bản.

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

char *dup_cap(const char *s)
{
    char *d = strdup(s);

    if (d != NULL)
        d[0] = toupper(d[0]);

    return d;
}

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

    if (s != NULL)
        puts(s);  // Hello, world!
    else
        puts("Error duplicating string");

    free(s);
}
```

Xem thêm

`strcpy()`, `strncpy()`, `malloc()`, `free()`

26.4 `strcat()`, `strncat()`

Concatenate (nối) hai chuỗi thành một chuỗi duy nhất

Synopsis

```
#include <string.h>

int strcat(const char *dest, const char *src);

int strncat(const char *dest, const char *src, size_t n);
```

Mô tả

"Concatenate", cho ai chưa biết, nghĩa là "dán vào nhau". Mấy hàm này nhận hai chuỗi, dán chúng lại, rồi lưu kết quả vào chuỗi đầu tiên.

Các hàm này không tính đến kích thước của chuỗi thứ nhất khi nối. Dịch ra thực tế nghĩa là bạn có thể cố nhét một chuỗi 2 megabyte vào một chỗ chỉ có 10 byte. Điều này sẽ dẫn đến hậu quả ngoài ý muốn, trừ khi bạn cố ý dẫn đến hậu quả ngoài ý muốn, trong trường hợp đó nó sẽ dẫn đến hậu quả ngoài-ý-muốn-một-cách-có-chủ-ý.

Gạt chuyện đùa kỹ thuật sang một bên, sếp và/hoặc giáo sư của bạn sẽ nổi điên.

Nếu bạn muốn chắc chắn không tràn chuỗi đầu tiên, hãy kiểm tra độ dài của các chuỗi trước và dùng một phép trừ cực kỳ cao siêu để đảm bảo mọi thứ vừa vặn.

Thực ra bạn có thể chỉ concatenate `n` ký tự đầu của chuỗi thứ hai bằng cách dùng `strncat()` và chỉ định số ký tự tối đa cần sao chép.

Giá trị trả về

Cả hai hàm đều trả về một con trỏ tới chuỗi đích, như hầu hết các hàm xử lý chuỗi khác.

Ví dụ

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

int main(void)
{
    char dest[30] = "Hello";
    char *src = ", World!";
    char numbers[] = "12345678";

    printf("dest before strcat: \"%s\"\n", dest); // "Hello"

    strcat(dest, src);
    printf("dest after strcat:  \"%s\"\n", dest); // "Hello, world!"

    strncat(dest, numbers, 3); // strcat 3 ký tự đầù của numbers
    printf("dest after strncat: \"%s\"\n", dest); // "Hello, world!123"
}
```

Để ý tôi có trộn lẫn ký pháp con trỏ và mảng ở đó với `src` và `numbers`; với các hàm chuỗi thì chuyện đó không sao cả.

Xem thêm

`strlen()`

26.5 `strcmp()`, `strncmp()`, `memcmp()`

So sánh hai chuỗi hoặc hai vùng bộ nhớ và trả về phần chênh lệch

Synopsis

```
#include <string.h>

int strcmp(const char *s1, const char *s2);

int strncmp(const char *s1, const char *s2, size_t n);

int memcmp(const void *s1, const void *s2, size_t n);
```

Mô tả

Tất cả các hàm này đều so sánh các khối byte trong bộ nhớ.

`strcmp()` và `strncmp()` làm việc với chuỗi NUL-terminated, trong khi `memcmp()` sẽ so sánh đúng số byte bạn chỉ định, bơ đi mọi ký tự NUL mà nó vô tình gặp trên đường.

`strcmp()` so sánh toàn bộ chuỗi đến cuối, còn `strncmp()` chỉ so sánh `n` ký tự đầu của các chuỗi.

Giá trị chúng trả về hơi kỳ kỳ. Về cơ bản nó là phần chênh lệch của các chuỗi, nên nếu các chuỗi giống nhau thì nó trả về 0 (vì chênh lệch bằng 0). Nó sẽ trả về khác 0 nếu các chuỗi khác nhau; về cơ bản nó sẽ tìm ký tự đầu tiên không khớp và trả về giá trị nhỏ hơn 0 nếu ký tự đó trong `s1` nhỏ hơn ký tự tương ứng trong `s2`. Nó sẽ trả về giá trị lớn hơn 0 nếu ký tự đó trong `s1` lớn hơn trong `s2`.

Vậy nếu chúng trả về `0`, phép so sánh là bằng nhau (tức là chênh lệch bằng `0`.)

Các hàm này có thể dùng làm hàm so sánh cho `qsort()` nếu bạn có một mảng các `char*` cần sắp xếp.

Giá trị trả về

Trả về 0 nếu các chuỗi hay các vùng bộ nhớ giống nhau, nhỏ hơn 0 nếu ký tự khác biệt đầu tiên trong `s1` nhỏ hơn ký tự tương ứng trong `s2`, hoặc lớn hơn 0 nếu ký tự khác biệt đầu tiên trong `s1` lớn hơn trong `s2`.

Ví dụ

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

int main(void)
{
    char *s1 = "Muffin";
    char *s2 = "Muffin Sandwich";
    char *s3 = "Muffin";

    int r1 = strcmp("Biscuits", "Kittens");
    printf("%d\n", r1); // in ra < 0 vì 'B' < 'K'

    int r2 = strcmp("Kittens", "Biscuits");
    printf("%d\n", r2); // in ra > 0 vì 'K' > 'B'

    if (strcmp(s1, s2) == 0)
        printf("This won't get printed because the strings differ\n");

    if (strcmp(s1, s3) == 0)
        printf("This will print because s1 and s3 are the same\n");

    // hơi lạ...nhưng nếu các chuỗi giống nhau, nó sẽ trả về`
    // 0, mà cũng có thể hiểu là "false". Not-false
    // là "true", nên (!strcmp()) sẽ là true nếu các chuỗi giống
    // nhau. Ừ, kỳ thật, nhưng bạn sẽ thấy cái này suốt ngoài đời
    // nên cứ quen dần đi:

    if (!strcmp(s1, s3))
        printf("The strings are the same!\n");

    if (!strncmp(s1, s2, 6))
        printf("The first 6 characters of s1 and s2 are the same\n");
}
```

Xem thêm

`memcmp()`, `qsort()`

26.6 `strcoll()`

So sánh hai chuỗi có tính đến locale

Synopsis

```
#include <string.h>

int strcoll(const char *s1, const char *s2);
```

Mô tả

Đây về cơ bản là `strcmp()`, chỉ khác là nó xử lý các ký tự có dấu tốt hơn tuỳ theo locale.

Ví dụ, `strcmp()` của tôi báo rằng ký tự "é" (có dấu) lớn hơn "f". Nhưng chuyện đó chẳng có mấy tác dụng cho việc sắp xếp theo bảng chữ cái.

Bằng cách set giá trị locale `LC_COLLATE` (theo tên hoặc qua `LC_ALL`), bạn có thể khiến `strcoll()` sắp xếp theo cách có ý nghĩa hơn với locale hiện tại. Ví dụ, có "é" xuất hiện một cách tử tế *trước* "f".

Nó cũng chậm hơn `strcmp()` kha khá nên chỉ dùng khi bắt buộc. Xem `strxfrm()` để có cách tăng tốc tiềm năng.

Giá trị trả về

Giống các hàm so sánh chuỗi khác, `strcoll()` trả về giá trị âm nếu `s1` nhỏ hơn `s2`, hoặc giá trị dương nếu `s1` lớn hơn `s2`. Hoặc `0` nếu chúng bằng nhau.

Ví dụ

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

int main(void)
{
    setlocale(LC_ALL, "");

    // Nêú bộ ký tự nguôǹ của bạn không hô˜trợ "é" trong chuôĩ
    // bạn có thể˙thay bằng `\u00e9`, là code point Unicode
    // của "é".

    printf("%d\n", strcmp("é", "f"));   // Báo é > f, ẹ.
    printf("%d\n", strcoll("é", "f"));  // Báo é < f, hoan hô!
}
```

Xem thêm

`strcmp()`

26.7 `strxfrm()`

Chuyển đổi (transform) một chuỗi để so sánh dựa trên locale

Synopsis

```
#include <string.h>

size_t strxfrm(char * restrict s1, const char * restrict s2, size_t n);
```

Mô tả

Đây là một hàm be bé kỳ lạ, nên chịu khó theo tôi nhé.

Trước tiên, nếu bạn chưa làm quen, hãy tìm hiểu `strcoll()` vì cái này liên quan mật thiết tới nó.

OK! Giờ bạn đã quay lại, bạn có thể nghĩ về `strxfrm()` như phần đầu trong ruột của `strcoll()`. Về cơ bản, `strcoll()` phải transform (chuyển đổi) một chuỗi sang một dạng có thể đem so sánh bằng `strcmp()`. Và nó làm việc này bằng `strxfrm()` cho cả hai chuỗi mỗi lần bạn gọi.

`strxform()` lấy chuỗi `s2` và transform nó (chuẩn bị cho `strcmp()`) lưu kết quả vào `s1`. Nó ghi không quá `n` byte, bảo vệ ta khỏi những vụ tràn buffer (buffer overflow) kinh khủng.

Nhưng khoan—còn một chế độ khác nữa! Nếu bạn truyền `NULL` cho `s1` và `0` cho `n`, nó sẽ trả về số byte mà chuỗi đã transform *đáng ra sẽ* cần[1]. Cái này hữu ích khi bạn cần cấp phát sẵn chỗ để chứa chuỗi đã transform trước khi `strcmp()` nó với chuỗi khác.

Ý tôi là, nói thẳng ra, `strcoll()` chậm so với `strcmp()`. Nó phải làm thêm rất nhiều việc khi chạy `strxfrm()` trên tất cả các chuỗi.

Thực tế, ta có thể thấy cách nó hoạt động bằng cách tự viết một bản như thế này:

```
int my_strcoll(char *s1, char *s2)
{
    // Dùng n = 0 để chỉ lấy độ dài của các chuỗi đã transform
    int len1 = strxfrm(NULL, s1, 0) + 1;
    int len2 = strxfrm(NULL, s2, 0) + 1;

    // Cấp phát đủ chỗ cho mỗi chuỗi
    char *d1 = malloc(len1);
    char *d2 = malloc(len2);

    // Transform các chuỗi để so sánh
    strxfrm(d1, s1, len1);
    strxfrm(d2, s2, len2);

    // So sánh các chuỗi đã transform
    int result = strcmp(d1, d2);

    // Giải phóng các chuỗi đã transform
    free(d2);
    free(d1);

    return result;
}
```

[1] Nó luôn trả về số byte mà chuỗi đã transform chiếm, nhưng trong trường hợp này vì `s1` là `NULL`, nó không thực sự ghi ra một chuỗi đã transform.

Bạn thấy ở dòng 12, 13 và 16 phía trên, ta transform hai chuỗi input rồi gọi `strcmp()` trên kết quả.

Vậy tại sao ta có hàm này? Không thể cứ gọi `strcoll()` cho xong chuyện à?

Ý tưởng là: nếu bạn có một chuỗi sẽ được đem so sánh với rất nhiều chuỗi khác, có lẽ bạn chỉ muốn transform chuỗi đó một lần, rồi dùng `strcmp()` nhanh hơn để tiết kiệm cả đống công việc ta đã phải làm trong hàm phía trên.

Ta sẽ làm điều đó trong ví dụ.

Giá trị trả về

Trả về số byte của chuỗi đã transform. Nếu giá trị lớn hơn `n`, kết quả trong `s1` không có ý nghĩa.

Ví dụ

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

// Transform một chuỗi để so sánh, trả về kết quả đã malloc
char *get_xfrm_str(char *s)
{
    int len = strxfrm(NULL, s, 0) + 1;
    char *d = malloc(len);

    strxfrm(d, s, len);

    return d;
}

// Làm một nửa công việc của strcoll() thông thường vì chuỗi thứ hai
// đã được transform sẵn khi truyền vào.
int half_strcoll(char *s1, char *s2_transformed)
{
    char *s1_transformed = get_xfrm_str(s1);

    int result = strcmp(s1_transformed, s2_transformed);

    free(s1_transformed);

    return result;
}

int main(void)
{
    setlocale(LC_ALL, "");

    // Transform trước chuỗi dùng để so sánh
    char *s = get_xfrm_str("éfg");

    // So sánh lặp đi lặp lại với "éfg"
    printf("%d\n", half_strcoll("fgh", s));  // "fgh" > "éfg"
    printf("%d\n", half_strcoll("àbc", s));  // "àbc" < "éfg"
    printf("%d\n", half_strcoll("ĥij", s));  // "ĥij" > "éfg"

    free(s);
```

```
}
```

Xem thêm

`strcoll()`

26.8 `strchr()`, `strrchr()`, `memchr()`

Tìm một ký tự trong chuỗi

Synopsis

```
// Pre-C23:

#include <string.h>

char *strchr(char *str, int c);

char *strrchr(char *str, int c);

void *memchr(const void *s, int c, size_t n);


// C23:

QChar *strchr(QChar *s, int c);

QChar *strrchr(QChar *s, int c);

QVoid *memchr(QVoid *s, int c, size_t n);
```

Mô tả

Các hàm `strchr()` và `strrchr` lần lượt tìm lần xuất hiện đầu tiên hoặc cuối cùng của một chữ cái trong chuỗi. (Chữ "r" thừa trong `strrchr()` là viết tắt của "reverse" (ngược)—nó bắt đầu tìm từ cuối chuỗi và đi ngược về.) Mỗi hàm trả về con trỏ tới char đó, hoặc `NULL` nếu không tìm thấy chữ cái trong chuỗi.

`memchr()` tương tự, chỉ khác là thay vì dừng lại ở ký tự NUL đầu tiên, nó tiếp tục tìm trong bao nhiêu byte bạn chỉ định.

Khá thẳng thắn.

Một việc bạn có thể làm nếu muốn tìm lần xuất hiện tiếp theo của ký tự sau khi tìm được lần đầu, là gọi lại hàm với giá trị trả về trước đó cộng một. (Nhớ số học con trỏ chứ?) Hoặc trừ một nếu bạn tìm ngược. Đừng lỡ tay đi quá cuối chuỗi!

Giá trị trả về

Trả về con trỏ tới nơi xuất hiện của ký tự trong chuỗi, hoặc `NULL` nếu không tìm thấy ký tự.

Ví dụ

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

int main(void)
{
    // "Hello, world!"
    //       ^  ^   ^
    //       A  B   C

    char *str = "Hello, world!";
    char *p;

    p = strchr(str, ',');       // p giờ trỏ vào vị trí A
    p = strrchr(str, 'o');      // p giờ trỏ vào vị trí B

    p = memchr(str, '!', 13);   // p giờ trỏ vào vị trí C

    // tìm lặp lại tất cả các lần xuất hiện của chữ 'B'
    str = "A BIG BROWN BAT BIT BEEJ";

    for(p = strchr(str, 'B'); p != NULL; p = strchr(p + 1, 'B')) {
        printf("Found a 'B' here: %s\n", p);
    }
}
```

Output:

```
Found a 'B' here: BIG BROWN BAT BIT BEEJ
Found a 'B' here: BROWN BAT BIT BEEJ
Found a 'B' here: BAT BIT BEEJ
Found a 'B' here: BIT BEEJ
Found a 'B' here: BEEJ
```

26.9 `strspn()`, `strcspn()`

Trả về độ dài của một chuỗi chỉ gồm một tập các ký tự, hoặc không thuộc một tập các ký tự

Synopsis

```
#include <string.h>

size_t strspn(char *str, const char *accept);

size_t strcspn(char *str, const char *reject);
```

Mô tả

`strspn()` sẽ cho bạn biết độ dài của một chuỗi chỉ gồm toàn bộ các ký tự trong tập `accept`. Tức là, nó bắt đầu đi dọc `str` cho đến khi tìm thấy một ký tự *không* nằm trong tập (nghĩa là một ký tự không được chấp nhận), rồi trả về độ dài của chuỗi tính đến đó.

`strcspn()` hoạt động tương tự, chỉ khác là nó đi dọc `str` cho tới khi tìm thấy một ký tự trong tập `reject` (tập loại trừ—nghĩa là một ký tự cần bị loại bỏ.) Rồi nó trả về độ dài của chuỗi tính đến đó.

Giá trị trả về

Độ dài của chuỗi gồm toàn bộ ký tự trong `accept` (với `strspn()`), hoặc độ dài của chuỗi gồm toàn bộ ký tự không thuộc `reject` (với `strcspn()`).

Ví dụ

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

int main(void)
{
    char str1[] = "a banana";
    char str2[] = "the bolivian navy on maenuvers in the south pacific";
    int n;

    // có bao nhiêu chữ cái trong str1 trước khi gặp thứ không phải nguyên âm?
    n = strspn(str1, "aeiou");
    printf("%d\n", n);  // n == 1, chỉ "a"

    // có bao nhiêu chữ cái trong str1 trước khi gặp thứ không phải a, b,
    // hay khoảng trắńg?
    n = strspn(str1, "ab ");
    printf("%d\n", n);  // n == 4, "a ba"

    // có bao nhiêu chữ cái trong str2 trước khi gặp "y"?
    n = strcspn(str2, "y");
    printf("%d\n", n);  // n = 16, "the bolivian nav"
}
```

Xem thêm

`strchr()`, `strrchr()`

26.10 `strpbrk()`

Tìm trong chuỗi một trong các ký tự của một tập hợp

Synopsis

```
#include <string.h>

// Pre-C23:

char *strpbrk(const char *s1, const char *s2);

// C23:

QChar *strpbrk(QChar *s1, const char *s2);
```

Mô tả

Hàm này tìm trong chuỗi `s1` bất kỳ ký tự nào có trong chuỗi `s2`.

Nó giống cách `strchr()` tìm một ký tự cụ thể trong một chuỗi, chỉ khác là nó sẽ khớp *bất kỳ* ký tự nào có trong `s2`.

Nghĩ về sức mạnh đó đi!

Giá trị trả về

Trả về con trỏ tới ký tự đầu tiên được khớp trong `s1`, hoặc NULL nếu không tìm thấy.

Ví dụ

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

int main(void)
{
    //  p trỏ vào đây sau strpbrk
    //              v
    char *s1 = "Hello, world!";
    char *s2 = "dow!";  // Khớp bất kỳ ký tự nào trong sô´này

    char *p = strpbrk(s1, s2);  // p trỏ vào chữ o

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

Xem thêm

`strchr()`, `memchr()`

26.11 `strstr()`

Tìm một chuỗi trong một chuỗi khác

Synopsis

```
#include <string.h>

// Pre-C23:

char *strstr(const char *str, const char *substr);

// C23:

QChar *strstr(QChar *s1, const char *s2);
```

Mô tả

Giả sử bạn có một chuỗi dài loằng ngoằng, và bạn muốn tìm một từ, hoặc bất kỳ chuỗi con nào khiến bạn thích thú, bên trong chuỗi đầu tiên. Vậy thì `strstr()` là dành cho bạn! Nó sẽ trả về con trỏ tới `substr` nằm trong `str`!

Giá trị trả về

Bạn nhận lại một con trỏ tới nơi xuất hiện của `substr` bên trong `str`, hoặc `NULL` nếu không tìm thấy chuỗi con.

Ví dụ

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

int main(void)
{
    char *str = "The quick brown fox jumped over the lazy dogs.";
    char *p;

    p = strstr(str, "lazy");
    printf("%s\n", p == NULL? "null": p); // "lazy dogs."

    // p là NULL sau dòng này, vì chuỗi "wombat" không có trong str:
    p = strstr(str, "wombat");
    printf("%s\n", p == NULL? "null": p); // "null"
}
```

Xem thêm

`strchr()`, `strrchr()`, `strspn()`, `strcspn()`

26.12 `strtok()`

Tách chuỗi thành các token

Synopsis

```
#include <string.h>

char *strtok(char *str, const char *delim);
```

Mô tả

Nếu bạn có một chuỗi chứa một đống dấu phân cách, và bạn muốn chia chuỗi đó thành từng mẩu riêng, hàm này có thể làm việc đó cho bạn.

Cách dùng hơi kỳ kỳ, nhưng ít ra mỗi khi bạn thấy hàm này ngoài đời, nó đều kỳ một cách nhất quán.

Về cơ bản, lần đầu tiên gọi nó, bạn truyền chuỗi `str` bạn muốn chia ra làm tham số đầu tiên. Cho mỗi lần gọi tiếp theo để lấy thêm token từ chuỗi, bạn truyền `NULL`. Hơi kỳ, nhưng `strtok()` ghi nhớ chuỗi ban đầu bạn đã truyền vào, và tiếp tục lột các token ra cho bạn.

Lưu ý rằng nó làm việc này bằng cách thực sự đặt một null terminator sau token, rồi trả về con trỏ tới đầu token. Vì vậy chuỗi gốc bạn truyền vào bị phá huỷ, có thể nói thế. Nếu bạn cần giữ chuỗi nguyên, hãy chắc chắn truyền một bản sao của nó vào `strtok()` để chuỗi gốc không bị phá.

Giá trị trả về

Con trỏ tới token tiếp theo. Nếu hết token, `NULL` sẽ được trả về.

Ví dụ

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

int main(void)
{
    // chia chuôĩ thành một loạt từ được phân tách bởi khoảng trăńg
    // hoặc dâú câu
    char str[] = "Where is my bacon, dude?";
    char *token;

    // Lưu ý cái câú trúc if-do-while sau đây rât rât rât rât rât
    // phô̂biến khi dùng strtok().

    // lâý token đâù tiên (nhớ đảm bảo có một token đâù tiên!)
    if ((token = strtok(str, ".,?! ")) != NULL) {
        do {
            printf("Word: \"%s\"\n", token);

            // giờ, điêù kiện tiếp tục của while lâý
            // token kê´tiếp (bằng cách truyêǹ NULL làm tham sô´đâù)
            // và tiếp tục nêú token không phải NULL:
        } while ((token = strtok(NULL, ".,?! ")) != NULL);
    }
}
```

Output:

```
Word: "Where"
Word: "is"
Word: "my"
Word: "bacon"
Word: "dude"
```

Xem thêm

`strchr()`, `strrchr()`, `strspn()`, `strcspn()`

26.13 `memset()`, ‘memset_explicit

Đặt một vùng bộ nhớ về một giá trị nhất định

Synopsis

`memset_explicit()` là mới trong C23!

```
#include <string.h>

void *memset(void *s, int c, size_t n);

void *memset_explicit(void *s, int c, size_t n);
```

Mô tả

Hàm này là thứ bạn dùng để đặt một vùng bộ nhớ về một giá trị cụ thể, cụ thể là `c` được chuyển sang `unsigned char`.

Cách dùng phổ biến nhất là để zero out (đặt về 0) một mảng hoặc một `struct`.

`memset_explicit()` chỉ khác ở chỗ nó sẽ không bao giờ bị tối ưu hoá bỏ đi (như `memset()` có thể bị). Ý tưởng là bạn có thể dùng nó để xoá sạch thông tin nhạy cảm (như mật khẩu) khỏi bộ nhớ trước khi đám hacker khó ưa nào đó chạm tay vào.

Giá trị trả về

`memset()` và `memset_explicit()` trả về đúng cái bạn truyền vào làm `s` cho tiện vui vẻ.

Ví dụ

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

int main(void)
{
    struct banana {
        float ripeness;
        char *peel_color;
        int grams;
    };

    struct banana b;

    memset(&b, 0, sizeof b);

    printf("%d\n", b.ripeness == 0.0);     // True
    printf("%d\n", b.peel_color == NULL);  // True
    printf("%d\n", b.grams == 0);          // True
}
```

Xem thêm

`memcpy()`, `memmove()`

26.14 `strerror()`

Lấy phiên bản chuỗi của một mã lỗi

Synopsis

```
#include <string.h>

char *strerror(int errnum);
```

Mô tả

Hàm này gắn rất chặt với `perror()` (hàm in thông báo lỗi dễ đọc tương ứng với `errno`). Nhưng thay vì in ra, `strerror()` trả về con trỏ tới chuỗi thông báo lỗi tuỳ theo locale.

Nên nếu bạn vì lý do nào đó cần cái chuỗi đó (ví dụ bạn định `fprintf()` nó ra file chẳng hạn), hàm này sẽ đưa nó cho bạn. Bạn chỉ cần truyền `errno` làm đối số. (Nhớ rằng `errno` được set làm trạng thái lỗi bởi rất nhiều hàm khác nhau.)

Thực ra bạn có thể truyền số nguyên bất kỳ vào `errnum` tuỳ ý. Hàm sẽ trả về *một* thông báo nào đó, kể cả khi số đó không ứng với giá trị `errno` đã biết nào.

Các giá trị của `errno` và các chuỗi được `strerror()` trả về phụ thuộc vào hệ thống.

Giá trị trả về

Một chuỗi thông báo lỗi ứng với mã lỗi cho trước.

Bạn không được phép chỉnh sửa chuỗi được trả về.

Ví dụ

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

int main(void)
{
    FILE *fp = fopen("NONEXISTENT_FILE.TXT", "r");

    if (fp == NULL) {
        char *errmsg = strerror(errno);
        printf("Error %d opening file: %s\n", errno, errmsg);
    }
}
```

Output:

```
Error 2 opening file: No such file or directory
```

Xem thêm

`perror()`

26.15 `strlen()`

Trả về độ dài của một chuỗi

Synopsis

```
#include <string.h>

size_t strlen(const char *s);
```

Mô tả

Hàm này trả về độ dài của chuỗi null-terminated được truyền vào (không tính ký tự NUL ở cuối). Nó làm điều này bằng cách đi dọc chuỗi và đếm byte cho tới ký tự NUL, nên nó hơi tốn thời gian một chút. Nếu bạn phải lấy độ dài của cùng một chuỗi nhiều lần, hãy lưu nó vào một biến đâu đó.

Giá trị trả về

Trả về số byte trong chuỗi. Lưu ý rằng con số này có thể khác với số ký tự trong một chuỗi multibyte.

Ví dụ

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

int main(void)
{
    char *s = "Hello, world!"; // 13 ký tự

    // in ra "The string is 13 characters long.":

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

Xem thêm

Chapter 27

`<tgmath.h>` Hàm Toán Type-Generic

Đây là các macro type-generic wrap quanh các hàm toán trong `<math.h>` và `<complex.h>`. Header này include cả hai cái đó.

Nhưng nhìn bề mặt, bạn có thể nghĩ về chúng như khả năng dùng, ví dụ, hàm `sqrt()` với bất kỳ kiểu nào mà không cần nghĩ xem nó là `double` hay `long double` hay thậm chí `complex`.

Đây là các macro được định nghĩa—vài cái không có đối ứng trong không gian thực hoặc phức. Type suffix được bỏ qua trong bảng ở các cột Real và Complex. Không macro generic nào có type suffix.

Hàm Thực	Hàm Phức	Macro Generic
`acos`	`cacos`	`acos`
`asin`	`casin`	`asin`
`atan`	`catan`	`atan`
`acosh`	`cacosh`	`acosh`
`asinh`	`casinh`	`asinh`
`atanh`	`catanh`	`atanh`
`cos`	`ccos`	`cos`
`sin`	`csin`	`sin`
`tan`	`ctan`	`tan`
`cosh`	`ccosh`	`cosh`
`sinh`	`csinh`	`sinh`
`tanh`	`ctanh`	`tanh`
`exp`	`cexp`	`exp`
`log`	`clog`	`log`
`pow`	`cpow`	`pow`
`sqrt`	`csqrt`	`sqrt`
`fabs`	`cabs`	`fabs`
`atan2`	—	`atan2`
`fdim`	—	`fdim`
`cbrt`	—	`cbrt`
`floor`	—	`floor`
`ceil`	—	`ceil`
`fma`	—	`fma`
`copysign`	—	`copysign`
`fmax`	—	`fmax`
`erf`	—	`erf`
`fmin`	—	`fmin`
`erfc`	—	`erfc`
`fmod`	—	`fmod`
`exp2`	—	`exp2`

Hàm Thực	Hàm Phức	Macro Generic
`frexp`	—	`frexp`
`expm1`	—	`expm1`
`hypot`	—	`hypot`
`ilogb`	—	`ilogb`
`ldexp`	—	`ldexp`
`lgamma`	—	`lgamma`
`llrint`	—	`llrint`
`llround`	—	`llround`
`log10`	—	`log10`
`log1p`	—	`log1p`
`log2`	—	`log2`
`logb`	—	`logb`
`lrint`	—	`lrint`
`lround`	—	`lround`
`nearbyint`	—	`nearbyint`
`nextafter`	—	`nextafter`
`nexttoward`	—	`nexttoward`
`remainder`	—	`remainder`
`remquo`	—	`remquo`
`rint`	—	`rint`
`round`	—	`round`
`scalbn`	—	`scalbn`
`scalbln`	—	`scalbln`
`tgamma`	—	`tgamma`
`trunc`	—	`trunc`
—	`carg`	`carg`
—	`cimag`	`cimag`
—	`conj`	`conj`
—	`cproj`	`cproj`
—	`creal`	`creal`

27.1 Ví dụ

Đây là ví dụ ta gọi hàm `sqrt()` type-generic trên nhiều kiểu khác nhau.

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

int main(void)
{
    double x = 12.8;
    long double y = 34.9;
    double complex z = 1 + 2 * I;

    double x_result;
    long double y_result;
    double complex z_result;

    // Ta gọi cùng hàm sqrt()--nó là type-generic!
    x_result = sqrt(x);
    y_result = sqrt(y);
    z_result = sqrt(z);
```

```
    printf("x_result: %f\n", x_result);
    printf("y_result: %Lf\n", y_result);
    printf("z_result: %f + %fi\n", creal(z_result), cimag(z_result));
}
```

Output:

```
x_result: 3.577709
y_result: 5.907622
z_result: 1.272020 + 0.786151i
```

Chapter 28

`<threads.h>` Các Hàm Multithreading

Hàm	Mô tả
`call_once()`	Gọi một hàm đúng một lần dù có bao nhiêu thread cùng thử
`cnd_broadcast()`	Đánh thức tất cả thread đang chờ trên một condition variable
`cnd_destroy()`	Giải phóng tài nguyên của một condition variable
`cnd_init()`	Khởi tạo một condition variable để sẵn sàng dùng
`cnd_signal()`	Đánh thức một thread đang chờ trên một condition variable
`cnd_timedwait()`	Chờ trên một condition variable có timeout
`cnd_wait()`	Chờ một signal trên một condition variable
`mtx_destroy()`	Dọn dẹp một mutex khi xong việc
`mtx_init()`	Khởi tạo một mutex để dùng
`mtx_lock()`	Giành lock trên một mutex
`mtx_timedlock()`	Khoá một mutex cho phép timeout
`mtx_trylock()`	Thử khoá một mutex, trả về ngay nếu không được
`mtx_unlock()`	Giải phóng một mutex khi xong critical section
`thrd_create()`	Tạo một thread thực thi mới
`thrd_current()`	Lấy ID của thread đang gọi
`thrd_detach()`	Tự động dọn dẹp thread khi thoát
`thrd_equal()`	So sánh hai thread descriptor xem có bằng nhau không
`thrd_exit()`	Dừng và thoát thread này
`thrd_join()`	Chờ một thread thoát
`thrd_yield()`	Dừng chạy để các thread khác có thể chạy
`tss_create()`	Tạo thread-specific storage mới
`tss_delete()`	Dọn dẹp một biến thread-specific storage
`tss_get()`	Lấy dữ liệu thread-specific
`tss_set()`	Đặt dữ liệu thread-specific

Cái header này cho chúng ta cả đống thứ hay ho:

- Threads (thread)
- Mutex
- Condition Variable (biến điều kiện)
- Thread-Specific Storage (bộ nhớ riêng cho thread)
- Và, cuối cùng nhưng không kém phần vui vẻ, hàm `call_once()` luôn thú vị!

Tận hưởng nào!

28.1 `call_once()`

Gọi một hàm đúng một lần dù có bao nhiêu thread cùng thử

Synopsis

```
#include <threads.h>

void call_once(once_flag *flag, void (*func)(void));
```

Mô tả

Nếu bạn có một đống thread cùng chạy trên cùng đoạn code có gọi một hàm, nhưng bạn chỉ muốn hàm đó chạy đúng một lần, thì `call_once()` có thể giúp bạn.

Điểm bất tiện là hàm được gọi không trả về gì và không nhận tham số nào cả.

Nếu cần nhiều hơn thế, bạn sẽ phải tự đặt một cờ threadsafe, kiểu như `atomic_flag`, hoặc một cờ được bảo vệ bằng mutex.

Để dùng được nó, bạn cần truyền vào một con trỏ tới hàm cần thực thi, `func`, và cả một con trỏ tới cờ kiểu `once_flag`.

`once_flag` là kiểu opaque, nên tất cả những gì bạn cần biết là bạn khởi tạo nó bằng giá trị `ONCE_FLAG_INIT`.

Giá trị trả về

Không trả về gì cả.

Ví dụ

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

once_flag of = ONCE_FLAG_INIT;  // Khởi tạo như thế´này

void run_once_function(void)
{
    printf("I'll only run once!\n");
}

int run(void *arg)
{
    (void)arg;

    printf("Thread running!\n");

    call_once(&of, run_once_function);

    return 0;
}
```

```
#define THREAD_COUNT 5

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);
}
```

Output (có thể khác giữa các lần chạy):

```
Thread running!
Thread running!
I'll only run once!
Thread running!
Thread running!
Thread running!
```

28.2 `cnd_broadcast()`

Đánh thức tất cả thread đang chờ trên một condition variable

Synopsis

```
#include <threads.h>

int cnd_broadcast(cnd_t *cond);
```

Mô tả

Hàm này giống hệt `cnd_signal()` ở chỗ nó đánh thức các thread đang chờ trên một condition variable... khác biệt là thay vì chỉ khều dậy một thread, nó đánh thức toàn bộ.

Tất nhiên, chỉ một thread sẽ lấy được mutex, và mấy thread còn lại phải chờ đến lượt. Nhưng thay vì ngủ để chờ signal, chúng sẽ ngủ để chờ giành lại mutex. Nói cách khác, chúng đã sẵn sàng nhảy vào cuộc.

Điều này tạo nên khác biệt trong một tình huống cụ thể mà `cnd_signal()` có thể để bạn kẹt cứng.

Nếu bạn dựa vào các thread tiếp theo để phát `cnd_signal()` kế tiếp, nhưng `cnd_wait()` của bạn lại nằm trong vòng `while`[1] không cho thread nào thoát ra được, bạn sẽ kẹt luôn. Không còn thread nào được đánh thức từ việc chờ nữa.

Nhưng nếu bạn `cnd_broadcast()`, tất cả thread đều được đánh thức, và giả định là ít nhất một trong số đó sẽ được phép thoát khỏi vòng `while`, rồi nó lại broadcast cho lần đánh thức tiếp theo khi xong việc.

Giá trị trả về

Trả về `thrd_success` hoặc `thrd_error` tuỳ theo mọi chuyện diễn ra thế nào.

[1] Đúng ra nên như vậy vì có spurious wakeup.

Ví dụ

Trong ví dụ dưới đây, chúng ta khởi động một đống thread, nhưng chúng chỉ được phép chạy nếu ID của chúng khớp với ID hiện tại. Nếu không, chúng quay lại chờ.

Nếu bạn dùng `cnd_signal()` để đánh thức thread tiếp theo, có thể nó không phải là thread có ID đúng để chạy. Nếu không phải, nó quay lại ngủ và chúng ta treo (vì không còn thread nào đang thức để gọi `cnd_signal()` lần nữa).

Nhưng nếu bạn `cnd_broadcast()` để đánh thức tất cả, tất cả chúng sẽ lần lượt thử thoát khỏi vòng `while`. Và một trong số đó sẽ thành công.

Thử đổi `cnd_broadcast()` sang `cnd_signal()` để thấy deadlock có khả năng xảy ra. Không xảy ra mỗi lần, nhưng thường là có.

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

cnd_t condvar;
mtx_t mutex;

int run(void *arg)
{
    int id = *(int*)arg;

    static int current_id = 0;

    mtx_lock(&mutex);

    while (id != current_id) {
        printf("THREAD %d: waiting\n", id);
        cnd_wait(&condvar, &mutex);

        if (id != current_id)
            printf("THREAD %d: woke up, but it's not my turn!\n", id);
        else
            printf("THREAD %d: woke up, my turn! Let's go!\n", id);
    }

    current_id++;

    printf("THREAD %d: signaling thread %d to run\n", id, current_id);

    //cnd_signal(&condvar);
    cnd_broadcast(&condvar);
    mtx_unlock(&mutex);

    return 0;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];
    int id[] = {4, 3, 2, 1, 0};

    mtx_init(&mutex, mtx_plain);
```

```
    cnd_init(&condvar);

    for (int i = 0; i < THREAD_COUNT; i++)
        thrd_create(t + i, run, id + i);

    for (int i = 0; i < THREAD_COUNT; i++)
        thrd_join(t[i], NULL);

    mtx_destroy(&mutex);
    cnd_destroy(&condvar);
}
```

Một lần chạy mẫu với `cnd_broadcast()`:

```
THREAD 4: waiting
THREAD 1: waiting
THREAD 3: waiting
THREAD 2: waiting
THREAD 0: signaling thread 1 to run
THREAD 2: woke up, but it's not my turn!
THREAD 2: waiting
THREAD 4: woke up, but it's not my turn!
THREAD 4: waiting
THREAD 3: woke up, but it's not my turn!
THREAD 3: waiting
THREAD 1: woke up, my turn! Let's go!
THREAD 1: signaling thread 2 to run
THREAD 4: woke up, but it's not my turn!
THREAD 4: waiting
THREAD 3: woke up, but it's not my turn!
THREAD 3: waiting
THREAD 2: woke up, my turn! Let's go!
THREAD 2: signaling thread 3 to run
THREAD 4: woke up, but it's not my turn!
THREAD 4: waiting
THREAD 3: woke up, my turn! Let's go!
THREAD 3: signaling thread 4 to run
THREAD 4: woke up, my turn! Let's go!
THREAD 4: signaling thread 5 to run
```

Một lần chạy mẫu với `cnd_signal()`:

```
THREAD 4: waiting
THREAD 1: waiting
THREAD 3: waiting
THREAD 2: waiting
THREAD 0: signaling thread 1 to run
THREAD 4: woke up, but it's not my turn!
THREAD 4: waiting

[deadlock at this point]
```

Thấy `THREAD 0` đã signal rằng tới lượt `THREAD 1` chưa? Nhưng—tin xấu—`THREAD 4` lại là đứa được đánh thức. Thế là không còn ai tiếp tục quá trình. `cnd_broadcast()` thì sẽ đánh thức tất cả, nên sau cùng `THREAD 1` sẽ chạy, thoát khỏi `while`, và broadcast để thread tiếp theo chạy.

Xem thêm

`cnd_signal()`, `mtx_lock()`, `mtx_unlock()`

28.3 `cnd_destroy()`

Giải phóng tài nguyên của một condition variable

Synopsis

```
#include <threads.h>

void cnd_destroy(cnd_t *cond);
```

Mô tả

Đây là cặp đối nghịch của `cnd_init()` và nên được gọi khi tất cả thread đã dùng xong một condition variable.

Giá trị trả về

Không trả về gì cả!

Ví dụ

Ví dụ condition variable tổng quát ở đây, nhưng bạn có thể thấy `cnd_destroy()` ở gần cuối.

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

cnd_t condvar;
mtx_t mutex;

int run(void *arg)
{
    (void)arg;

    mtx_lock(&mutex);

    printf("Thread: waiting...\n");
    cnd_wait(&condvar, &mutex);
    printf("Thread: running again!\n");

    mtx_unlock(&mutex);

    return 0;
}

int main(void)
{
    thrd_t t;

    mtx_init(&mutex, mtx_plain);
    cnd_init(&condvar);
```

```
    printf("Main creating thread\n");
    thrd_create(&t, run, NULL);

    // Ngủ 0.1s để thread kia có thời gian vào trạng thái chờ
    thrd_sleep(&(struct timespec){.tv_nsec=100000000L}, NULL);

    mtx_lock(&mutex);
    printf("Main: signaling thread\n");
    cnd_signal(&condvar);
    mtx_unlock(&mutex);

    thrd_join(t, NULL);

    mtx_destroy(&mutex);
    cnd_destroy(&condvar);  // <-- HUỶ CONDITION VARIABLE
}
```

Output:

```
Main creating thread
Thread: waiting...
Main: signaling thread
Thread: running again!
```

Xem thêm

`cnd_init()`

28.4 `cnd_init()`

Khởi tạo một condition variable để sẵn sàng dùng

Synopsis

```
#include <threads.h>

int cnd_init(cnd_t *cond);
```

Mô tả

Đây là cặp đối nghịch của `cnd_destroy()`. Hàm này chuẩn bị một condition variable để dùng, làm đủ thứ việc hậu trường trên nó.

Đừng dùng một condition variable mà chưa gọi hàm này trước!

Giá trị trả về

Nếu mọi chuyện suôn sẻ, trả về `thrd_success`. Nếu không suôn sẻ, nó có thể trả về `thrd_nomem` khi hệ thống hết bộ nhớ, hoặc `thread_error` trong trường hợp lỗi khác.

Ví dụ

Ví dụ condition variable tổng quát ở đây, nhưng bạn có thể thấy `cnd_init()` ở đầu `main()`.

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

cnd_t condvar;
mtx_t mutex;

int run(void *arg)
{
    (void)arg;

    mtx_lock(&mutex);

    printf("Thread: waiting...\n");
    cnd_wait(&condvar, &mutex);
    printf("Thread: running again!\n");

    mtx_unlock(&mutex);

    return 0;
}

int main(void)
{
    thrd_t t;

    mtx_init(&mutex, mtx_plain);
    cnd_init(&condvar);      // <-- KHỞI TẠO CONDITION VARIABLE

    printf("Main creating thread\n");
    thrd_create(&t, run, NULL);

    // Ngủ 0.1s để thread kia có thời gian vào trạng thái chờ
    thrd_sleep(&(struct timespec){.tv_nsec=100000000L}, NULL);

    mtx_lock(&mutex);
    printf("Main: signaling thread\n");
    cnd_signal(&condvar);
    mtx_unlock(&mutex);

    thrd_join(t, NULL);

    mtx_destroy(&mutex);
    cnd_destroy(&condvar);
}
```

Output:

```
Main creating thread
Thread: waiting...
Main: signaling thread
Thread: running again!
```

Xem thêm

`cnd_destroy()`

28.5 cnd_signal()

Đánh thức một thread đang chờ trên một condition variable

Synopsis

```
#include <threads.h>

int cnd_signal(cnd_t *cond);
```

Mô tả

Nếu bạn có một thread (hoặc một đống thread) đang chờ trên một condition variable, hàm này sẽ đánh thức một trong số đó để chạy.

So với `cnd_broadcast()` vốn đánh thức tất cả thread. Xem trang `cnd_broadcast()` để biết thêm thông tin về lúc nào nên dùng cái kia và lúc nào nên dùng cái này.

Giá trị trả về

Trả về `thrd_success` hoặc `thrd_error` tuỳ xem chương trình của bạn đang vui hay buồn.

Ví dụ

Ví dụ condition variable tổng quát ở đây, nhưng bạn có thể thấy `cnd_signal()` ở giữa `main()`.

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

cnd_t condvar;
mtx_t mutex;

int run(void *arg)
{
    (void)arg;

    mtx_lock(&mutex);

    printf("Thread: waiting...\n");
    cnd_wait(&condvar, &mutex);
    printf("Thread: running again!\n");

    mtx_unlock(&mutex);

    return 0;
}

int main(void)
{
    thrd_t t;

    mtx_init(&mutex, mtx_plain);
    cnd_init(&condvar);

    printf("Main creating thread\n");
    thrd_create(&t, run, NULL);
```

```
    // Ngủ 0.1s để thread kia có thời gian vào trạng thái chờ
    thrd_sleep(&(struct timespec){.tv_nsec=100000000L}, NULL);

    mtx_lock(&mutex);
    printf("Main: signaling thread\n");
    cnd_signal(&condvar);    // <-- SIGNAL THREAD CON TẠI ĐÂY!
    mtx_unlock(&mutex);

    thrd_join(t, NULL);

    mtx_destroy(&mutex);
    cnd_destroy(&condvar);
}
```

Output:

```
Main creating thread
Thread: waiting...
Main: signaling thread
Thread: running again!
```

Xem thêm

`cnd_init()`, `cnd_destroy()`

28.6 `cnd_timedwait()`

Chờ trên một condition variable có timeout

Synopsis

```
#include <threads.h>

int cnd_timedwait(cnd_t *restrict cond, mtx_t *restrict mtx,
                  const struct timespec *restrict ts);
```

Mô tả

Hàm này giống `cnd_wait()` chỉ khác là chúng ta được phép chỉ định thêm một timeout.

Lưu ý là thread vẫn phải giành lại mutex để làm thêm việc kể cả sau khi timeout. Khác biệt chính là `cnd_wait()` thông thường chỉ thử giành mutex sau khi có `cnd_signal()` hoặc `cnd_broadcast()`, trong khi `cnd_timedwait()` cũng làm vậy, **và** còn thử giành mutex sau khi timeout.

Timeout được chỉ định dưới dạng thời gian tuyệt đối (thời gian tuyệt đối) theo UTC tính từ Epoch. Bạn có thể lấy giá trị này qua hàm `timespec_get()` rồi cộng thêm vào kết quả để timeout muộn hơn thời điểm hiện tại, như trong ví dụ.

Coi chừng chuyện bạn không được có nhiều hơn 999999999 nano giây (nanosecond) trong trường `tv_nsec` của `struct timespec`. Dùng phép mod để chúng nằm trong khoảng cho phép.

Giá trị trả về

Nếu thread thức dậy vì lý do không phải timeout (ví dụ signal hay broadcast), trả về `thrd_success`. Nếu thức dậy do timeout, trả về `thrd_timedout`. Ngược lại trả về `thrd_error`.

Ví dụ

Ví dụ này cho một thread chờ trên một condition variable tối đa 1.75 giây. Và nó luôn timeout vì chả có ai gửi signal cả. Bi kịch.

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

cnd_t condvar;
mtx_t mutex;

int run(void *arg)
{
    (void)arg;

    mtx_lock(&mutex);

    struct timespec ts;

    // Lấy thời gian hiện tại
    timespec_get(&ts, TIME_UTC);

    // Cộng thêm 1.75 giây kể từ bây giờ
    ts.tv_sec += 1;
    ts.tv_nsec += 750000000L;

    // Xử lý tràn nsec
    ts.tv_sec += ts.tv_nsec / 1000000000L;
    ts.tv_nsec = ts.tv_nsec % 1000000000L;

    printf("Thread: waiting...\n");
    int r = cnd_timedwait(&condvar, &mutex, &ts);

    switch (r) {
        case thrd_success:
            printf("Thread: signaled!\n");
            break;

        case thrd_timedout:
            printf("Thread: timed out!\n");
            return 1;

        case thrd_error:
            printf("Thread: Some kind of error\n");
            return 2;
    }

    mtx_unlock(&mutex);

    return 0;
}

int main(void)
{
    thrd_t t;

    mtx_init(&mutex, mtx_plain);
```

```
    cnd_init(&condvar);

    printf("Main creating thread\n");
    thrd_create(&t, run, NULL);

    // Ngủ 3s để thread kia có thời gian timeout
    thrd_sleep(&(struct timespec){.tv_sec=3}, NULL);

    thrd_join(t, NULL);

    mtx_destroy(&mutex);
    cnd_destroy(&condvar);
}
```

Output:

```
Main creating thread
Thread: waiting...
Thread: timed out!
```

Xem thêm

`cnd_wait()`, `timespec_get()`

28.7 `cnd_wait()`

Chờ một signal trên một condition variable

Synopsis

```
#include <threads.h>

int cnd_wait(cnd_t *cond, mtx_t *mtx);
```

Mô tả

Hàm này đưa thread đang gọi vào giấc ngủ cho tới khi nó được đánh thức bởi một lời gọi `cnd_signal()` hoặc `cnd_broadcast()`.

Giá trị trả về

Nếu mọi thứ tuyệt vời, trả về `thrd_success`. Ngược lại, nó trả về `thrd_error` để báo rằng có gì đó đã sai nghiêm trọng đến mức kinh hoàng.

Ví dụ

Ví dụ condition variable tổng quát ở đây, nhưng bạn có thể thấy `cnd_wait()` trong hàm `run()`.

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

cnd_t condvar;
mtx_t mutex;
```

```
int run(void *arg)
{
    (void)arg;

    mtx_lock(&mutex);

    printf("Thread: waiting...\n");
    cnd_wait(&condvar, &mutex);       // <-- CHỜ TẠI ĐÂY!
    printf("Thread: running again!\n");

    mtx_unlock(&mutex);

    return 0;
}

int main(void)
{
    thrd_t t;

    mtx_init(&mutex, mtx_plain);
    cnd_init(&condvar);

    printf("Main creating thread\n");
    thrd_create(&t, run, NULL);

    // Ngủ 0.1s để thread kia có thời gian vào trạng thái chờ
    thrd_sleep(&(struct timespec){.tv_nsec=100000000L}, NULL);

    mtx_lock(&mutex);
    printf("Main: signaling thread\n");
    cnd_signal(&condvar);    // <-- SIGNAL THREAD CON TẠI ĐÂY!
    mtx_unlock(&mutex);

    thrd_join(t, NULL);

    mtx_destroy(&mutex);
    cnd_destroy(&condvar);
}
```

Output:

```
Main creating thread
Thread: waiting...
Main: signaling thread
Thread: running again!
```

Xem thêm

`cnd_timedwait()`

28.8 `mtx_destroy()`

Dọn dẹp một mutex khi xong việc

Synopsis

```
#include <threads.h>

void mtx_destroy(mtx_t *mtx);
```

Mô tả

Ngược lại của `mtx_init()`, hàm này giải phóng mọi tài nguyên liên quan đến mutex được truyền vào.

Bạn nên gọi hàm này khi tất cả thread đã xong việc với mutex đó.

Giá trị trả về

Không trả về gì cả, cái đồ vô ơn ích kỷ!

Ví dụ

Ví dụ mutex tổng quát ở đây, nhưng bạn có thể thấy `mtx_destroy()` ở gần cuối.

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

cnd_t condvar;
mtx_t mutex;

int run(void *arg)
{
    (void)arg;

    static int count = 0;

    mtx_lock(&mutex);

    printf("Thread: I got %d!\n", count);
    count++;

    mtx_unlock(&mutex);

    return 0;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    mtx_init(&mutex, mtx_plain);

    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);

    mtx_destroy(&mutex);   // <-- HUỶ MUTEX TẠI ĐÂY
```

```
}
```

Output:

```
Thread: I got 0!
Thread: I got 1!
Thread: I got 2!
Thread: I got 3!
Thread: I got 4!
```

Xem thêm

`mtx_init()`

28.9 `mtx_init()`

Khởi tạo một mutex để dùng

Synopsis

```
#include <threads.h>

int mtx_init(mtx_t *mtx, int type);
```

Mô tả

Trước khi bạn có thể dùng một biến mutex, bạn phải khởi tạo nó bằng lời gọi này để chuẩn bị mọi thứ sẵn sàng.

Nhưng khoan! Không đơn giản thế đâu. Bạn phải nói cho nó biết bạn muốn tạo `type` mutex loại gì.

Loại	Mô tả
`mtx_plain`	Mutex xưa nay thường thấy
`mtx_timed`	Mutex hỗ trợ timeout
`mtx_plain|mtx_recursive`	Mutex recursive (đệ quy)
`mtx_timed|mtx_recursive`	Mutex recursive hỗ trợ timeout

Như bạn thấy, bạn có thể biến mutex plain hoặc timed thành *recursive* bằng cách OR bit với `mtx_recursive`.

"Recursive" nghĩa là kẻ đang giữ lock có thể gọi `mtx_lock()` nhiều lần trên cùng một lock. (Nó phải unlock cũng bằng số lần tương đương thì người khác mới lấy được mutex.) Điều này có thể giúp code dễ chịu hơn trong một số trường hợp, đặc biệt nếu bạn gọi một hàm cần khoá mutex mà bạn đã đang giữ mutex đó rồi.

Và timeout cho thread một cơ hội *thử* giành lock trong một khoảng thời gian, rồi bỏ cuộc nếu không giành được trong khung thời gian đó. Bạn dùng hàm `mtx_timedlock()` với mutex kiểu `mtx_timed`.

Giá trị trả về

Trả về `thrd_success` trong một thế giới hoàn hảo, và có thể là `thrd_error` trong một thế giới không hoàn hảo.

Ví dụ

Ví dụ mutex tổng quát ở đây, nhưng bạn có thể thấy `mtx_init()` ở đầu `main()`:

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

cnd_t condvar;
mtx_t mutex;

int run(void *arg)
{
    (void)arg;

    static int count = 0;

    mtx_lock(&mutex);

    printf("Thread: I got %d!\n", count);
    count++;

    mtx_unlock(&mutex);

    return 0;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    mtx_init(&mutex, mtx_plain);  // <-- TẠO MUTEX TẠI ĐÂY

    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);

    mtx_destroy(&mutex);   // <-- HUỶ MUTEX TẠI ĐÂY
}
```

Output:

```
Thread: I got 0!
Thread: I got 1!
Thread: I got 2!
Thread: I got 3!
Thread: I got 4!
```

Xem thêm

`mtx_destroy()`

28.10 mtx_lock()

Giành lock trên một mutex

Synopsis

```
#include <threads.h>

int mtx_lock(mtx_t *mtx);
```

Mô tả

Nếu bạn là một thread và muốn bước vào critical section (đoạn tới hạn), tôi có một hàm hợp ý bạn đây!

Một thread gọi hàm này sẽ chờ cho tới khi giành được mutex, rồi nó sẽ tóm lấy, tỉnh dậy và chạy!

Nếu mutex là recursive và đã bị khoá bởi chính thread này rồi, nó sẽ được khoá lần nữa và số đếm lock sẽ tăng lên. Nếu mutex không phải recursive và thread đã giữ nó rồi, lời gọi này sẽ báo lỗi.

Giá trị trả về

Trả về `thrd_success` khi mọi thứ tốt đẹp và `thrd_error` khi trục trặc.

Ví dụ

Ví dụ mutex tổng quát ở đây, nhưng bạn có thể thấy `mtx_lock()` trong hàm `run()`:

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

cnd_t condvar;
mtx_t mutex;

int run(void *arg)
{
    (void)arg;

    static int count = 0;

    mtx_lock(&mutex);  // <-- KHOÁ TẠI ĐÂY

    printf("Thread: I got %d!\n", count);
    count++;

    mtx_unlock(&mutex);

    return 0;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    mtx_init(&mutex, mtx_plain);  // <-- TẠO MUTEX TẠI ĐÂY
```

```
    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);

    mtx_destroy(&mutex);   // <-- HUỶ MUTEX TẠI ĐÂY
}
```

Output:

```
Thread: I got 0!
Thread: I got 1!
Thread: I got 2!
Thread: I got 3!
Thread: I got 4!
```

Xem thêm

`mtx_unlock()`, `mtx_trylock()`, `mtx_timedlock()`

28.11 `mtx_timedlock()`

Khoá một mutex cho phép timeout

Synopsis

```
#include <threads.h>

int mtx_timedlock(mtx_t *restrict mtx, const struct timespec *restrict ts);
```

Mô tả

Giống hệt `mtx_lock()` chỉ khác là bạn có thể thêm timeout nếu không muốn chờ mãi mãi.

Timeout được chỉ định dưới dạng thời gian tuyệt đối theo UTC tính từ Epoch. Bạn có thể lấy giá trị này qua hàm `timespec_get()` rồi cộng thêm vào kết quả để timeout muộn hơn thời điểm hiện tại, như trong ví dụ.

Coi chừng chuyện bạn không được có nhiều hơn 999999999 nano giây trong trường `tv_nsec` của `struct timespec`. Dùng phép mod để chúng nằm trong khoảng cho phép.

Giá trị trả về

Nếu mọi chuyện ổn và mutex được giành, trả về `thrd_success`. Nếu timeout xảy ra trước, trả về `thrd_timedout`.

Ngoài ra trả về `thrd_error`. Vì nếu chả có gì đúng thì mọi thứ đều sai.

Ví dụ

Ví dụ này cho một thread chờ trên mutex tối đa 1.75 giây. Và nó luôn timeout vì chả có ai gửi signal cả.

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

mtx_t mutex;

int run(void *arg)
{
    (void)arg;

    struct timespec ts;

    // Lấy thời gian hiện tại
    timespec_get(&ts, TIME_UTC);

    // Cộng thêm 1.75 giây kể từ bây giờ
    ts.tv_sec += 1;
    ts.tv_nsec += 750000000L;

    // Xử lý tràn nsec
    ts.tv_sec += ts.tv_nsec / 1000000000L;
    ts.tv_nsec = ts.tv_nsec % 1000000000L;

    printf("Thread: waiting for lock...\n");
    int r = mtx_timedlock(&mutex, &ts);

    switch (r) {
        case thrd_success:
            printf("Thread: grabbed lock!\n");
            break;

        case thrd_timedout:
            printf("Thread: timed out!\n");
            break;

        case thrd_error:
            printf("Thread: Some kind of error\n");
            break;
    }

    mtx_unlock(&mutex);

    return 0;
}

int main(void)
{
    thrd_t t;

    mtx_init(&mutex, mtx_plain);

    mtx_lock(&mutex);

    printf("Main creating thread\n");
    thrd_create(&t, run, NULL);

    // Ngủ 3s để thread kia có thời gian timeout
```

```
    thrd_sleep(&(struct timespec){.tv_sec=3}, NULL);

    mtx_unlock(&mutex);

    thrd_join(t, NULL);

    mtx_destroy(&mutex);
}
```

Output:

```
Main creating thread
Thread: waiting for lock...
Thread: timed out!
```

Xem thêm

`mtx_lock()`, `mtx_trylock()`, `timespec_get()`

28.12 `mtx_trylock()`

Thử khoá một mutex, trả về ngay nếu không được

Synopsis

```
#include <threads.h>

int mtx_trylock(mtx_t *mtx);
```

Mô tả

Hàm này hoạt động giống hệt `mtx_lock` chỉ khác là nó trả về ngay lập tức nếu không thể giành được lock.

Spec có ghi rằng có khả năng `mtx_trylock()` có thể thất bại kiểu spurious với `thrd_busy` ngay cả khi không có thread nào khác đang giữ lock. Tôi không chắc tại sao lại vậy, nhưng bạn nên viết code phòng thủ chống lại điều này.

Giá trị trả về

Trả về `thrd_success` nếu mọi chuyện ổn. Hoặc `thrd_busy` nếu có thread khác đang giữ lock. Hoặc `thrd_error`, nghĩa là có gì đó đã đi đúng. À ý tôi là "sai".

Ví dụ

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

mtx_t mutex;

int run(void *arg)
{
```

```
    int id = *(int*)arg;

    int r = mtx_trylock(&mutex);   // <-- THỬ GIÀNH LOCK

    switch (r) {
        case thrd_success:
            printf("Thread %d: grabbed lock!\n", id);
            break;

        case thrd_busy:
            printf("Thread %d: lock already taken :(\n", id);
            return 1;

        case thrd_error:
            printf("Thread %d: Some kind of error\n", id);
            return 2;
    }

    mtx_unlock(&mutex);

    return 0;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];
    int id[THREAD_COUNT];

    mtx_init(&mutex, mtx_plain);

    for (int i = 0; i < THREAD_COUNT; i++) {
        id[i] = i;
        thrd_create(t + i, run, id + i);
    }

    for (int i = 0; i < THREAD_COUNT; i++)
        thrd_join(t[i], NULL);

    mtx_destroy(&mutex);
}
```

Output (khác nhau giữa các lần chạy):

```
Thread 0: grabbed lock!
Thread 1: lock already taken :(
Thread 4: lock already taken :(
Thread 3: grabbed lock!
Thread 2: lock already taken :(
```

Xem thêm

`mtx_lock()`, `mtx_timedlock()`, `mtx_unlock()`

28.13 `mtx_unlock()`

Giải phóng một mutex khi xong critical section

Synopsis

```
#include <threads.h>

int mtx_unlock(mtx_t *mtx);
```

Mô tả

Sau khi bạn đã làm xong hết mấy thứ nguy hiểm, nơi mà các thread liên quan không được giẫm chân lên nhau... bạn có thể buông tay siết chặt của mình khỏi mutex bằng cách gọi `mtx_unlock()`.

Giá trị trả về

Trả về `thrd_success` khi thành công. Hoặc `thrd_error` khi lỗi. Không gì sáng tạo lắm trong khoản này.

Ví dụ

Ví dụ mutex tổng quát ở đây, nhưng bạn có thể thấy `mtx_unlock()` trong hàm `run()`:

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

cnd_t condvar;
mtx_t mutex;

int run(void *arg)
{
    (void)arg;

    static int count = 0;

    mtx_lock(&mutex);

    printf("Thread: I got %d!\n", count);
    count++;

    mtx_unlock(&mutex);  // <-- UNLOCK TẠI ĐÂY

    return 0;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    mtx_init(&mutex, mtx_plain);

    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);

    mtx_destroy(&mutex);
}
```

Output:

```
Thread: I got 0!
Thread: I got 1!
Thread: I got 2!
Thread: I got 3!
Thread: I got 4!
```

Xem thêm

`mtx_lock()`, `mtx_timedlock()`, `mtx_trylock()`

28.14 `thrd_create()`

Tạo một thread thực thi mới

Synopsis

```
#include <threads.h>

int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
```

Mô tả

Giờ thì *bạn* có QUYỀN NĂNG!

Phải không nào?

Đây là cách bạn khởi động thread mới để chương trình làm nhiều việc cùng lúc[2]!

Để làm được chuyện này, bạn cần truyền vào một con trỏ tới `thrd_t` sẽ được dùng để đại diện cho thread mà bạn đang spawn (tạo).

Thread đó sẽ bắt đầu chạy hàm mà bạn truyền con trỏ vào qua tham số `func`. Đây là một giá trị kiểu `thrd_start_t`, tức là con trỏ tới một hàm trả về `int` và nhận một tham số `void*` duy nhất, ví dụ:

```
int thread_run_func(void *arg)
```

Và, như bạn có thể đoán, con trỏ bạn truyền vào `thrd_create()` cho tham số `arg` sẽ được chuyển tiếp đến hàm `func`. Đây là cách bạn cung cấp thêm thông tin cho thread khi nó khởi động.

Tất nhiên, với `arg`, bạn phải chắc chắn truyền con trỏ tới một object thread-safe hoặc mỗi thread một cái riêng.

Nếu thread trả về từ hàm đó, nó thoát y như thể đã gọi `thrd_exit()`.

Cuối cùng, giá trị mà hàm `func` trả về có thể được thread cha nhặt bằng `thrd_join()`.

[2] Ừm, ít nhất là chừng nào bạn có đủ core rảnh. OS sẽ lên lịch cho chúng tuỳ khả năng.

Giá trị trả về

Trong trường hợp tốt đẹp, trả về `thrd_success`. Nếu bạn hết bộ nhớ, sẽ trả về `thrd_nomem`. Ngược lại, `thrd_error`.

Ví dụ

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

int run(void *arg)
{
    int id = *(int*)arg;

    printf("Thread %d: I'm alive!!\n", id);

    return id;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];
    int id[THREAD_COUNT];  // Mỗi thread một cái

    for (int i = 0; i < THREAD_COUNT; i++) {
        id[i] = i; // Truyền số thứ tự thread làm ID
        thrd_create(t + i, run, id + i);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        int res;

        thrd_join(t[i], &res);

        printf("Main: thread %d exited with code %d\n", i, res);
    }
}
```

Output (có thể khác nhau giữa các lần chạy):

```
Thread 1: I'm alive!!
Thread 0: I'm alive!!
Thread 3: I'm alive!!
Thread 2: I'm alive!!
Main: thread 0 exited with code 0
Main: thread 1 exited with code 1
Main: thread 2 exited with code 2
Main: thread 3 exited with code 3
Thread 4: I'm alive!!
Main: thread 4 exited with code 4
```

Xem thêm

`thrd_exit()`, `thrd_join()`

28.15 `thrd_current()`

Lấy ID của thread đang gọi

Synopsis

```
#include <threads.h>

thrd_t thrd_current(void);
```

Mô tả

Mỗi thread có một ID opaque kiểu `thrd_t`. Đây là giá trị mà ta thấy được khởi tạo khi gọi `thrd_create()`.

Nhưng nếu bạn muốn lấy ID của thread đang chạy thì sao?

Không vấn đề gì! Gọi hàm này thôi là nó trả về cho bạn.

Để làm gì ư? Ai mà biết!

Thực ra, thành thật mà nói, tôi có thể thấy nó được dùng ở vài chỗ.

1. Bạn có thể dùng để một thread tự detach (detach) chính mình bằng `thrd_detach()`. Tôi không rõ tại sao bạn lại muốn làm điều này.
2. Bạn có thể dùng để so sánh ID của thread hiện tại với một ID khác mà bạn lưu trong biến nào đó bằng hàm `thrd_equal()`. Cái này nghe hợp lý nhất.
3. ...
4. Kiếm lời!

Nếu ai có cách dùng khác, cho tôi biết nhé.

Giá trị trả về

Trả về ID của thread đang gọi.

Ví dụ

Đây là ví dụ chung chung cho thấy cách lấy ID thread hiện tại và so sánh với một ID thread đã ghi trước đó rồi đưa ra hành động kịch tính dựa trên kết quả! Với sự tham gia của Arnold Schwarzenegger!

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

thrd_t first_thread_id;

int run(void *arg)
{
    (void)arg;

    thrd_t my_id = thrd_current();   // <-- LẤY ID THREAD CỦA TÔI

    if (thrd_equal(my_id, first_thread_id))
        printf("I'm the first thread!\n");
    else
        printf("I'm not the first!\n");

    return 0;
}
```

```
int main(void)
{
    thrd_t t;

    thrd_create(&first_thread_id, run, NULL);
    thrd_create(&t, run, NULL);

    thrd_join(first_thread_id, NULL);
    thrd_join(t, NULL);
}
```

Output:

```
Come on, you got what you want, Cohaagen! Give deez people ay-ah!
```

À không, đợi đã, đó là câu thoại của Arnold Schwarzenegger trong *Total Recall*, một trong những phim khoa học viễn tưởng hay nhất mọi thời đại. Xem nó ngay rồi quay lại đọc tiếp trang tham khảo này.

Trời—kết phim mới đỉnh làm sao! Còn Johnny Cab? Xuất sắc. Thôi tiếp nào!

Output:

```
I'm the first thread!
I'm not the first!
```

Xem thêm

`thrd_equal()`, `thrd_detach()`

28.16 `thrd_detach()`

Tự động dọn dẹp thread khi thoát

Synopsis

```
#include <threads.h>

int thrd_detach(thrd_t thr);
```

Mô tả

Bình thường bạn phải `thrd_join()` để mấy tài nguyên liên quan đến thread đã chết được dọn dẹp. (Đáng chú ý nhất là exit status của nó vẫn còn lởn vởn chờ được nhặt.)

Nhưng nếu bạn gọi `thrd_detach()` trên thread trước, thì không cần dọn thủ công nữa. Chúng cứ thế thoát và được OS dọn dẹp.

(Lưu ý rằng khi main thread chết thì tất cả thread đều chết, dù trong trường hợp nào.)

Giá trị trả về

`thrd_success` nếu thread được detach thành công, ngược lại là `thrd_error`.

Ví dụ

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

thrd_t first_thread_id;

int run(void *arg)
{
    (void)arg;

    printf("Thread running!\n");

    return 0;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t;

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_create(&t, run, NULL);
        thrd_detach(t);
    }

    // Không cần thrd_join()!

    // Ngủ 1/4 giây để chúng kịp chạy xong hết
    thrd_sleep(&(struct timespec){.tv_nsec=250000000}, NULL);
}
```

Xem thêm

`thrd_join()`, `thrd_exit()`

28.17 `thrd_equal()`

So sánh hai thread descriptor xem có bằng nhau không

Synopsis

```
#include <threads.h>

int thrd_equal(thrd_t thr0, thrd_t thr1);
```

Mô tả

Nếu bạn có hai thread descriptor trong các biến `thrd_t`, bạn có thể kiểm tra chúng có bằng nhau không bằng hàm này.

Ví dụ, có thể một trong các thread có "siêu năng lực" đặc biệt mà các thread khác không có, và hàm run cần phân biệt được chúng, như trong ví dụ.

Giá trị trả về

Trả về khác 0 nếu hai thread bằng nhau. Trả về `0` nếu không.

Ví dụ

Đây là ví dụ chung chung cho thấy cách lấy ID thread hiện tại và so sánh với một ID thread đã ghi trước đó rồi đưa ra hành động nhàm chán dựa trên kết quả.

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

thrd_t first_thread_id;

int run(void *arg)
{
    (void)arg;

    thrd_t my_id = thrd_current();

    if (thrd_equal(my_id, first_thread_id))  // <-- SO SÁNH!
        printf("I'm the first thread!\n");
    else
        printf("I'm not the first!\n");

    return 0;
}

int main(void)
{
    thrd_t t;

    thrd_create(&first_thread_id, run, NULL);
    thrd_create(&t, run, NULL);

    thrd_join(first_thread_id, NULL);
    thrd_join(t, NULL);
}
```

Output:

```
I'm the first thread!
I'm not the first!
```

Xem thêm

`thrd_current()`

28.18 `thrd_exit()`

Dừng và thoát thread này

Synopsis

```
#include <threads.h>

_Noreturn void thrd_exit(int res);
```

Mô tả

Thread thường thoát bằng cách return từ hàm run của nó. Nhưng nếu nó muốn thoát sớm (có thể từ sâu trong call stack), hàm này sẽ làm việc đó.

Mã `res` có thể được một thread khác nhặt lấy qua `thrd_join()`, và nó tương đương với việc trả về một giá trị từ hàm run.

Như khi return từ hàm run, hàm này cũng sẽ dọn dẹp đúng cách toàn bộ thread-specific storage liên quan đến thread này—tất cả destructor cho các biến TSS của thread sẽ được gọi. Nếu vẫn còn biến TSS với destructor sau đợt dọn dẹp đầu tiên[3], các destructor còn lại sẽ được gọi. Việc này lặp lại cho tới khi không còn gì nữa, hoặc số vòng tàn sát đạt `TSS_DTOR_ITERATIONS`.

Nếu main thread gọi hàm này, nó như thể bạn đã gọi `exit(EXIT_SUCCESS)`.

Giá trị trả về

Hàm này không bao giờ trả về vì thread gọi nó bị giết trong quá trình này. Ảo diệu thiệt!

Ví dụ

Các thread trong ví dụ này thoát sớm với kết quả `22` nếu chúng nhận được giá trị `NULL` cho `arg`.

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

thrd_t first_thread_id;

int run(void *arg)
{
    (void)arg;

    if (arg == NULL)
        thrd_exit(22);

    return 0;
}

#define THREAD_COUNT 5

int main(void)
{
    thrd_t t[THREAD_COUNT];

    for (int i = 0; i < THREAD_COUNT; i++)
        thrd_create(t + i, run, i == 2? NULL: "spatula");


    for (int i = 0; i < THREAD_COUNT; i++) {
        int res;
        thrd_join(t[i], &res);
```

[3]Ví dụ, nếu một destructor làm cho nhiều biến khác được đặt lại.

```
        printf("Thread %d exited with code %d\n", i, res);
    }
}
```

Output:

```
Thread 0 exited with code 0
Thread 1 exited with code 0
Thread 2 exited with code 22
Thread 3 exited with code 0
Thread 4 exited with code 0
```

Xem thêm

`thrd_join()`

28.19 `thrd_join()`

Chờ một thread thoát

Synopsis

```
#include <threads.h>

int thrd_join(thrd_t thr, int *res);
```

Mô tả

Khi thread cha bắn ra một đống thread con, nó có thể chờ chúng chạy xong bằng lời gọi này

Giá trị trả về

Ví dụ

Các thread trong ví dụ này thoát sớm với kết quả `22` nếu chúng nhận được giá trị `NULL` cho `arg`. Thread cha nhặt mã kết quả này bằng `thrd_join()`.

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

thrd_t first_thread_id;

int run(void *arg)
{
    (void)arg;

    if (arg == NULL)
        thrd_exit(22);

    return 0;
}

#define THREAD_COUNT 5
```

```
int main(void)
{
    thrd_t t[THREAD_COUNT];

    for (int i = 0; i < THREAD_COUNT; i++)
        thrd_create(t + i, run, i == 2? NULL: "spatula");


    for (int i = 0; i < THREAD_COUNT; i++) {
        int res;
        thrd_join(t[i], &res);

        printf("Thread %d exited with code %d\n", i, res);
    }
}
```

Output:

```
Thread 0 exited with code 0
Thread 1 exited with code 0
Thread 2 exited with code 22
Thread 3 exited with code 0
Thread 4 exited with code 0
```

Xem thêm

`thrd_exit()`

28.20 `thrd_sleep()`

Ngủ trong một số giây và nano giây nhất định

Synopsis

```
#include <threads.h>

int thrd_sleep(const struct timespec *duration, struct timespec *remaining);
```

Mô tả

Hàm này đưa thread hiện tại vào giấc ngủ (ngủ) một lúc[4] cho phép các thread khác chạy.

Thread đang gọi sẽ thức dậy sau khi hết thời gian, hoặc nếu nó bị signal ngắt hoặc gì đó.

Nếu không bị ngắt, nó sẽ ngủ ít nhất là bằng chừng bạn yêu cầu. Có khi lâu hơn chút. Bạn biết đấy, rời khỏi giường khó cỡ nào.

Cấu trúc nhìn như thế này:

```
struct timespec {
    time_t tv_sec;   // Giây
    long   tv_nsec;  // Nano giây (phần tỷ của một giây)
};
```

[4]Các hệ kiểu Unix có syscall `sleep()` ngủ theo số giây nguyên. Nhưng `thrd_sleep()` có lẽ portable hơn và thêm nữa còn cho độ phân giải dưới-giây!

Đừng đặt `tv_nsec` lớn hơn 999,999,999. Tôi không rõ chuyện gì sẽ chính thức xảy ra nếu bạn làm vậy, nhưng trên máy tôi thì `thrd_sleep()` trả về `-2` và thất bại.

Giá trị trả về

Trả về `0` khi hết thời gian, hoặc `-1` nếu bị signal ngắt. Hoặc bất kỳ giá trị âm nào khác khi có lỗi khác. Kỳ lạ là spec cho phép "giá trị âm khác cho lỗi khác" cũng là `-1`, nên chúc may mắn với chuyện đó.

Ví dụ

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

int main(void)
{
    // Ngủ 3.25 giây
    thrd_sleep(&(struct timespec){.tv_sec=3, .tv_nsec=250000000}, NULL);

    return 0;
}
```

Xem thêm

`thrd_yield()`

28.21 `thrd_yield()`

Dừng chạy để các thread khác có thể chạy

Synopsis

```
#include <threads.h>

void thrd_yield(void);
```

Mô tả

Nếu bạn có một thread đang hốc hết CPU và bạn muốn cho các thread khác thời gian chạy, bạn có thể gọi `thrd_yield()`. Nếu hệ thống thấy hợp lý, nó sẽ đưa thread đang gọi vào giấc ngủ và một trong các thread khác sẽ chạy thay.

Đây là cách hay để "lịch sự" với các thread khác trong chương trình nếu bạn muốn khuyến khích chúng chạy thay.

Giá trị trả về

Không trả về gì cả!

Ví dụ

Ví dụ này hơi tệ vì OS dù sao cũng sẽ tự lên lịch lại các thread trên output, nhưng nó thể hiện được ý chính.

Main thread cho các thread khác một cơ hội chạy sau mỗi khối công việc ngu ngốc mà nó làm.

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

int run(void *arg)
{
    int main_thread = arg != NULL;

    if (main_thread) {
        long int total = 0;

        for (int i = 0; i < 10; i++) {
            for (long int j = 0; j < 1000L; j++)
                total++;

            printf("Main thread yielding\n");
            thrd_yield();                       // <-- YIELD TẠI ĐÂY
        }
    } else
        printf("Other thread running!\n");

    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, i == 0? "main": NULL);

    for (int i = 0; i < THREAD_COUNT; i++)
        thrd_join(t[i], NULL);

    return 0;
}
```

Output sẽ khác giữa các lần chạy. Để ý rằng kể cả sau `thrd_yield()`, các thread khác có thể vẫn chưa sẵn sàng chạy và main thread sẽ tiếp tục.

```
Main thread yielding
Main thread yielding
Main thread yielding
Other thread running!
Other thread running!
Other thread running!
Other thread running!
Main thread yielding
Other thread running!
Other thread running!
Main thread yielding
Main thread yielding
Main thread yielding
Other thread running!
Main thread yielding
```

```
Main thread yielding
Main thread yielding
Other thread running!
Other thread running!
```

Xem thêm

`thrd_sleep()`

28.22 `tss_create()`

Tạo thread-specific storage mới

Synopsis

```
#include <threads.h>

int tss_create(tss_t *key, tss_dtor_t dtor);
```

Mô tả

Hàm này giúp bạn khi cần lưu các giá trị khác nhau theo từng thread.

Một chỗ hay gặp là khi bạn có một biến ở phạm vi file được chia sẻ giữa cả đống hàm và thường xuyên bị trả về. Cái đó không threadsafe. Một cách refactor là thay nó bằng thread-specific storage để mỗi thread có code riêng của mình và không giẫm chân lên nhau.

Để làm việc này, bạn truyền vào con trỏ tới một khoá `tss_t` —đây là biến bạn sẽ dùng trong các lời gọi `tss_set()` và `tss_get()` sau đó để đặt và lấy giá trị gắn với khoá này.

Phần thú vị là con trỏ destructor `dtor` kiểu `tss_dtor_t`. Nó thực ra là con trỏ tới một hàm nhận tham số `void*` và trả về `void`, tức là

```
void dtor(void *p) { ... }
```

Hàm này sẽ được gọi cho mỗi thread khi thread thoát bằng `thrd_exit()` (hoặc return từ hàm run).

Gọi hàm này khi các destructor của thread khác đang chạy là hành vi không xác định (unspecified behavior).

Giá trị trả về

Không trả về gì cả!

Ví dụ

Đây là ví dụ TSS chung chung. Để ý biến TSS được tạo ở gần đầu `main()`.

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

tss_t str;

void some_function(void)
```

```
{
    // Lấy giá trị per-thread của chuỗi này
    char *tss_string = tss_get(str);

    // Và in nó ra
    printf("TSS string: %s\n", tss_string);
}

int run(void *arg)
{
    int serial = *(int*)arg;  // Lấy số thứ tự của thread này
    free(arg);

    // malloc() chỗ chứa dữ liệu cho thread này
    char *s = malloc(64);
    sprintf(s, "thread %d! :)", serial);  // Một chuỗi nho nhỏ vui tươi

    // Đặt biến TSS này trỏ tới chuỗi
    tss_set(str, s);

    // Gọi một hàm sẽ lấy biến
    some_function();

    return 0; // Tương đương thrd_exit(0); kích hoạt destructor
}

#define THREAD_COUNT 15

int main(void)
{
    thrd_t t[THREAD_COUNT];

    // Tạo biến TSS mới, hàm free() là destructor
    tss_create(&str, free);                  // <-- TẠO BIẾN TSS!

    for (int i = 0; i < THREAD_COUNT; i++) {
        int *n = malloc(sizeof *n);  // Chứa số thứ tự thread
        *n = i;
        thrd_create(t + i, run, n);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }

    // Và tất cả thread đã xong, nên giải phóng cái này
    tss_delete(str);
}
```

Output:

```
TSS string: thread 0! :)
TSS string: thread 2! :)
TSS string: thread 1! :)
TSS string: thread 5! :)
TSS string: thread 3! :)
TSS string: thread 6! :)
```

```
TSS string: thread 4! :)
TSS string: thread 7! :)
TSS string: thread 8! :)
TSS string: thread 9! :)
TSS string: thread 10! :)
TSS string: thread 13! :)
TSS string: thread 12! :)
TSS string: thread 11! :)
TSS string: thread 14! :)
```

Xem thêm

`tss_delete()`, `tss_set()`, `tss_get()`, `thrd_exit()`

28.23 `tss_delete()`

Dọn dẹp một biến thread-specific storage

Synopsis

```
#include <threads.h>

void tss_delete(tss_t key);
```

Mô tả

Đây là cặp đối nghịch của `tss_create()`. Bạn tạo (khởi tạo) biến TSS trước khi dùng, rồi khi tất cả thread cần dùng đã xong, bạn xoá (huỷ/giải phóng) nó bằng hàm này.

Hàm này không gọi destructor nào cả! Mấy cái đó đều do `thrd_exit()` gọi!

Giá trị trả về

Không trả về gì cả!

Ví dụ

Đây là ví dụ TSS chung chung. Để ý biến TSS được xoá ở gần cuối `main()`.

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

tss_t str;

void some_function(void)
{
    // Lấy giá trị per-thread của chuỗi này
    char *tss_string = tss_get(str);

    // Và in nó ra
    printf("TSS string: %s\n", tss_string);
}
```

```
int run(void *arg)
{
    int serial = *(int*)arg;  // Lấy số´thứ tự của thread này
    free(arg);

    // malloc() chô˜chứa dữ liệu cho thread này
    char *s = malloc(64);
    sprintf(s, "thread %d! :)", serial);  // Một chuôĩ nho nhỏ vui tươi

    // Đặt biến TSS này trỏ tới chuôĩ
    tss_set(str, s);

    // Gọi một hàm sẽ lấy biến
    some_function();

    return 0; // Tương đương thrd_exit(0); kích hoạt destructor
}

#define THREAD_COUNT 15

int main(void)
{
    thrd_t t[THREAD_COUNT];

    // Tạo biến TSS mới, hàm free() là destructor
    tss_create(&str, free);

    for (int i = 0; i < THREAD_COUNT; i++) {
        int *n = malloc(sizeof *n);  // Chứa số´thứ tự thread
        *n = i;
        thrd_create(t + i, run, n);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }

    // Và tất cả thread đã xong, nên giải phóng cái này
    tss_delete(str);    // <-- XOÁ BIẾN TSS!
}
```

Output:

```
TSS string: thread 0! :)
TSS string: thread 2! :)
TSS string: thread 1! :)
TSS string: thread 5! :)
TSS string: thread 3! :)
TSS string: thread 6! :)
TSS string: thread 4! :)
TSS string: thread 7! :)
TSS string: thread 8! :)
TSS string: thread 9! :)
TSS string: thread 10! :)
TSS string: thread 13! :)
TSS string: thread 12! :)
TSS string: thread 11! :)
```

```
TSS string: thread 14! :)
```

Xem thêm

`tss_create()`, `tss_set()`, `tss_get()`, `thrd_exit()`

28.24 `tss_get()`

Lấy dữ liệu thread-specific

Synopsis

```
#include <threads.h>

void *tss_get(tss_t key);
```

Mô tả

Một khi bạn đã đặt một biến bằng `tss_set()`, bạn có thể lấy giá trị qua `tss_get()`—chỉ cần truyền vào khoá là bạn nhận lại được con trỏ tới giá trị.

Đừng gọi hàm này từ trong destructor.

Giá trị trả về

Trả về giá trị đã lưu cho `key` được cho, hoặc `NULL` nếu có trục trặc.

Ví dụ

Đây là ví dụ TSS chung chung. Để ý biến TSS được lấy trong `some_function()`, bên dưới.

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

tss_t str;

void some_function(void)
{
    // Lấy giá trị per-thread của chuỗi này
    char *tss_string = tss_get(str);    // <-- LẤY GIÁ TRỊ

    // Và in nó ra
    printf("TSS string: %s\n", tss_string);
}

int run(void *arg)
{
    int serial = *(int*)arg;  // Lấy số thứ tự của thread này
    free(arg);

    // malloc() chỗ chứa dữ liệu cho thread này
    char *s = malloc(64);
    sprintf(s, "thread %d! :)", serial);  // Một chuỗi nho nhỏ vui tươi
```

```
    // Đặt biến TSS này trỏ tới chuỗi
    tss_set(str, s);

    // Gọi một hàm sẽ lấy biến
    some_function();

    return 0; // Tương đương thrd_exit(0); kích hoạt destructor
}

#define THREAD_COUNT 15

int main(void)
{
    thrd_t t[THREAD_COUNT];

    // Tạo biến TSS mới, hàm free() là destructor
    tss_create(&str, free);

    for (int i = 0; i < THREAD_COUNT; i++) {
        int *n = malloc(sizeof *n);  // Chứa số thứ tự thread
        *n = i;
        thrd_create(t + i, run, n);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }

    // Và tất cả thread đã xong, nên giải phóng cái này
    tss_delete(str);
}
```

Output:

```
TSS string: thread 0! :)
TSS string: thread 2! :)
TSS string: thread 1! :)
TSS string: thread 5! :)
TSS string: thread 3! :)
TSS string: thread 6! :)
TSS string: thread 4! :)
TSS string: thread 7! :)
TSS string: thread 8! :)
TSS string: thread 9! :)
TSS string: thread 10! :)
TSS string: thread 13! :)
TSS string: thread 12! :)
TSS string: thread 11! :)
TSS string: thread 14! :)
```

Xem thêm

`tss_set()`

28.25 `tss_set()`

Đặt dữ liệu thread-specific

Synopsis

```
#include <threads.h>

int tss_set(tss_t key, void *val);
```

Mô tả

Một khi bạn đã thiết lập biến TSS bằng `tss_create()`, bạn có thể đặt nó theo từng thread qua `tss_set()`.

`key` là định danh của dữ liệu này, còn `val` là con trỏ tới dữ liệu.

Destructor được chỉ định trong `tss_create()` sẽ được gọi cho giá trị được đặt khi thread thoát.

Ngoài ra, nếu có destructor *và* đã có giá trị sẵn cho khoá này rồi, destructor sẽ không được gọi cho giá trị đã có. Thực tế, hàm này không bao giờ khiến destructor được gọi. Nên bạn phải tự lo lấy—tốt nhất là dọn dẹp giá trị cũ trước khi ghi đè bằng giá trị mới.

Giá trị trả về

Trả về `thrd_success` khi vui, và `thrd_error` khi không.

Ví dụ

Đây là ví dụ TSS chung chung. Để ý biến TSS được đặt trong `run()`, bên dưới.

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

tss_t str;

void some_function(void)
{
    // Lấy giá trị per-thread của chuỗi này
    char *tss_string = tss_get(str);

    // Và in nó ra
    printf("TSS string: %s\n", tss_string);
}

int run(void *arg)
{
    int serial = *(int*)arg;  // Lấy số thứ tự của thread này
    free(arg);

    // malloc() chỗ chứa dữ liệu cho thread này
    char *s = malloc(64);
    sprintf(s, "thread %d! :)", serial);  // Một chuỗi nho nhỏ vui tươi

    // Đặt biến TSS này trỏ tới chuỗi
    tss_set(str, s);    // <-- ĐẶT BIẾN TSS
```

```
    // Gọi một hàm sẽ lấy biến
    some_function();

    return 0; // Tương đương thrd_exit(0); kích hoạt destructor
}

#define THREAD_COUNT 15

int main(void)
{
    thrd_t t[THREAD_COUNT];

    // Tạo biến TSS mới, hàm free() là destructor
    tss_create(&str, free);

    for (int i = 0; i < THREAD_COUNT; i++) {
        int *n = malloc(sizeof *n);  // Chứa số thứ tự thread
        *n = i;
        thrd_create(t + i, run, n);
    }

    for (int i = 0; i < THREAD_COUNT; i++) {
        thrd_join(t[i], NULL);
    }

    // Và tất cả thread đã xong, nên giải phóng cái này
    tss_delete(str);
}
```

Output:

```
TSS string: thread 0! :)
TSS string: thread 2! :)
TSS string: thread 1! :)
TSS string: thread 5! :)
TSS string: thread 3! :)
TSS string: thread 6! :)
TSS string: thread 4! :)
TSS string: thread 7! :)
TSS string: thread 8! :)
TSS string: thread 9! :)
TSS string: thread 10! :)
TSS string: thread 13! :)
TSS string: thread 12! :)
TSS string: thread 11! :)
TSS string: thread 14! :)
```

Xem thêm

`tss_get()`

Chapter 29

`<time.h>` Các hàm về ngày và giờ

Hàm	Mô tả
`clock()`	Xem tiến trình này đã dùng bao nhiêu thời gian CPU
`difftime()`	Tính chênh lệch giữa hai mốc thời gian
`mktime()`	Chuyển `struct tm` thành `time_t`
`time()`	Lấy calendar time (thời gian lịch) hiện tại
`timespec_get()`	Lấy thời gian có độ phân giải cao hơn, thường là ngay bây giờ
`asctime()`	Trả về phiên bản dễ đọc cho người của `struct tm`
`ctime()`	Trả về phiên bản dễ đọc cho người của `time_t`
`gmtime()`	Chuyển calendar time thành broken-down time (thời gian đã tách thành trường) theo UTC
`localtime()`	Chuyển calendar time thành broken-down time theo giờ địa phương
`strftime()`	In output ngày giờ có định dạng

Khi nói đến thời gian trong C, có hai kiểu chính cần để ý:

- **`time_t`** chứa một *calendar time* (thời gian lịch). Đây là một kiểu số có tiềm năng là opaque (không lộ bên trong), đại diện cho một thời điểm tuyệt đối và có thể chuyển đổi sang UTC[1] hoặc giờ địa phương.
- **`struct tm`** chứa một *broken-down time* (thời gian đã tách thành trường). Nó có các thứ như ngày trong tuần, ngày trong tháng, giờ, phút, giây, v.v.

Trên các hệ thống POSIX và Windows, `time_t` là một số nguyên và đại diện cho số giây đã trôi qua kể từ 1 tháng 1 năm 1970 lúc 00:00 UTC.

Một `struct tm` chứa các trường sau:

```
struct tm {
    int tm_sec;    // số giây sau phút -- [0, 60]
    int tm_min;    // số phút sau giờ -- [0, 59]
    int tm_hour;   // số giờ kể từ nửa đêm -- [0, 23]
    int tm_mday;   // ngày trong tháng -- [1, 31]
    int tm_mon;    // số tháng kể từ tháng Một -- [0, 11]
    int tm_year;   // số năm kể từ 1900
    int tm_wday;   // số ngày kể từ Chủ Nhật -- [0, 6]
    int tm_yday;   // số ngày kể từ 1 tháng 1 -- [0, 365]
    int tm_isdst;  // cờ Daylight Saving Time
};
```

[1] Khi bạn nói GMT, trừ khi bạn đang nói cụ thể về múi giờ chứ không phải về thời điểm, có lẽ thứ bạn muốn nói là "UTC".

Bạn có thể chuyển qua lại giữa hai kiểu này bằng `mktime()`, `gmtime()`, và `localtime()`.

Bạn có thể in thông tin thời gian ra chuỗi bằng `ctime()`, `asctime()`, và `strftime()`.

29.1 Cảnh báo về Thread Safety

`asctime()`, `ctime()`: Hai hàm này trả về con trỏ tới một vùng nhớ `static`. Cả hai có thể trả về cùng một con trỏ. Nếu bạn cần thread-safe (an toàn với thread), bạn sẽ cần một mutex bao quanh chúng. Nếu bạn cần cả hai kết quả cùng lúc, `strcpy()` một trong hai ra chỗ khác.

Mọi vấn đề với `asctime()` và `ctime()` có thể né được bằng cách dùng hàm `strftime()` thay thế—nó linh hoạt và thread-safe hơn.

`localtime()`, `gmtime()`: Hai hàm này cũng trả về con trỏ tới một vùng nhớ `static`. Cả hai có thể trả về cùng một con trỏ. Nếu bạn cần thread-safe, bạn sẽ cần một mutex bao quanh chúng. Nếu bạn cần cả hai kết quả cùng lúc, hãy sao chép `struct` sang một chỗ khác.

29.2 `clock()`

Xem tiến trình này đã dùng bao nhiêu thời gian CPU

Synopsis

```
#include <time.h>

clock_t clock(void);
```

Mô tả

Bộ xử lý của bạn lúc này đang tung hứng rất nhiều thứ. Chỉ vì một tiến trình đã sống được 20 phút không có nghĩa là nó đã dùng 20 phút "thời gian CPU".

Phần lớn thời gian, tiến trình trung bình của bạn ngồi không (đang ngủ), và cái đó không tính vào thời gian CPU đã dùng.

Hàm này trả về một kiểu opaque đại diện cho số "clock tick" (tick đồng hồ)[2] mà tiến trình đã tiêu tốn để chạy.

Bạn có thể lấy ra số giây từ đó bằng cách chia cho macro `CLOCKS_PER_SEC`. Đây là số nguyên, nên bạn sẽ phải ép một phần của biểu thức sang kiểu số thực để có thời gian có phần lẻ.

Chú ý rằng đây không phải "wall clock time" (thời gian đồng hồ treo tường) của chương trình. Nếu bạn muốn lấy cái đó thì dùng một cách đại khái `time()` và `difftime()` (có thể chỉ cho độ phân giải 1 giây) hoặc `timespec_get()` (cũng có thể chỉ cho độ phân giải thấp, nhưng ít ra nó *có thể* xuống tới mức nano giây).

Giá trị trả về

Trả về lượng thời gian CPU mà tiến trình này đã dùng. Giá trị này trả về dưới dạng có thể chia cho `CLOCKS_PER_SEC` để ra được thời gian tính bằng giây.

[2] Spec thực ra không nói "clock tick", nhưng tôi... thì nói.

Ví dụ

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

// Fibonacci cô´tình ngây thơ
long long int fib(long long int n) {
    if (n <= 1) return n;

    return fib(n-1) + fib(n-2);
}

int main(void)
{
    printf("The 42nd Fibonacci Number is %lld\n", fib(42));

    printf("CPU time: %f\n", clock() / (double)CLOCKS_PER_SEC);
}
```

Output trên máy của tôi:

```
The 42nd Fibonacci Number is 267914296
CPU time: 1.863078
```

Xem thêm

`time()`, `difftime()`, `timespec_get()`

29.3 `difftime()`

Tính chênh lệch giữa hai mốc thời gian

Synopsis

```
#include <time.h>

double difftime(time_t time1, time_t time0);
```

Mô tả

Vì kiểu `time_t` về mặt kỹ thuật là opaque, bạn không thể cứ thế lấy thẳng ra rồi trừ đi để có chênh lệch giữa hai cái[3]. Hãy dùng hàm này để làm việc đó.

Không có gì đảm bảo về độ phân giải của chênh lệch này, nhưng chắc tới mức giây.

Giá trị trả về

Trả về chênh lệch giữa hai `time_t` tính bằng giây.

[3]Trừ khi bạn đang ở trên một hệ thống POSIX nơi `time_t` chắc chắn là số nguyên, lúc đó bạn có thể trừ. Nhưng bạn vẫn nên dùng `difftime()` để portable (khả chuyển) tối đa.

Ví dụ

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

int main(void)
{
    // 12 tháng 4, 1982 và lẻ
    struct tm time_a = { .tm_year=82, .tm_mon=3, .tm_mday=12,
        .tm_hour=4, .tm_min=00, .tm_sec=04, .tm_isdst=-1,
    };

    // 15 tháng 11, 2020 và lẻ
    struct tm time_b = { .tm_year=120, .tm_mon=10, .tm_mday=15,
        .tm_hour=16, .tm_min=27, .tm_sec=00, .tm_isdst=-1,
    };

    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;  // gần đủ rồi

    printf("%f seconds (%f years) between events\n", diff, years);
}
```

Output:

```
1217996816.000000 seconds (38.596783 years) between events
```

Xem thêm

`time()`, `mktime()`

29.4 `mktime()`

Chuyển `struct tm` với giờ địa phương thành `time_t`

Synopsis

```
#include <time.h>

time_t mktime(struct tm *timeptr);
```

Mô tả

Nếu bạn có ngày và giờ ở local (giờ địa phương) và muốn chuyển nó sang `time_t` (để rồi `difftime()` nó hay làm gì đó), bạn có thể chuyển đổi bằng hàm này.

Về cơ bản bạn điền vào các trường trong `struct tm` của mình theo giờ địa phương, và `mktime()` sẽ chuyển chúng sang `time_t` UTC tương ứng.

Vài lưu ý:

- Đừng bận tâm điền `tm_wday` hay `tm_yday`. `mktime()` sẽ điền chúng giúp bạn.
- Bạn có thể đặt `tm_isdst` thành `0` để báo rằng giờ của bạn không phải Daylight Saving Time (DST / giờ tiết kiệm ánh sáng), `1` để báo là có, và `-1` để `mktime()` tự điền theo sở thích của locale.

Nếu bạn không có compiler C23 và cần nhập input theo UTC, xem các hàm không chuẩn `timegm()` [4] cho các hệ Unix-like và `_mkgmtime()` [5] cho Windows.

Giá trị trả về

Trả về giờ địa phương trong `struct tm` dưới dạng `time_t` calendar time.

Trả về `(time_t)(-1)` nếu có lỗi.

Ví dụ

Trong ví dụ sau, chúng ta để `mktime()` cho biết thời điểm đó có phải DST hay không.

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

int main(void)
{
    struct tm broken_down_time = {
        .tm_year=82,   // số năm kể từ 1900
        .tm_mon=3,     // số tháng kể từ tháng Một -- [0, 11]
        .tm_mday=12,   // ngày trong tháng -- [1, 31]
        .tm_hour=4,    // số giờ kể từ nửa đêm -- [0, 23]
        .tm_min=00,    // số phút sau giờ -- [0, 59]
        .tm_sec=04,    // số giây sau phút -- [0, 60]
        .tm_isdst=-1,  // cờ Daylight Saving Time
    };

    time_t calendar_time = mktime(&broken_down_time);

    char *days[] = {"Sunday", "Monday", "Tuesday",
        "Wednesday", "Furzeday", "Friday", "Saturday"};

    // Cái này sẽ in ra những gì có trong broken_down_time
    printf("Local time : %s", asctime(localtime(&calendar_time)));
    printf("Is DST     : %d\n", broken_down_time.tm_isdst);
    printf("Day of week: %s\n\n", days[broken_down_time.tm_wday]);

    // Cái này sẽ in UTC cho giờ địa phương ở trên
    printf("UTC        : %s", asctime(gmtime(&calendar_time)));
}
```

Output (đối với tôi ở Pacific Time—UTC đi trước 8 tiếng):

```
Local time : Mon Apr 12 04:00:04 1982
Is DST     : 0
Day of week: Monday

UTC        : Mon Apr 12 12:00:04 1982
```

[4] https://man.archlinux.org/man/timegm.3.en
[5] https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/mkgmtime-mkgmtime32-mkgmtime64?view=msvc-160

Xem thêm

`timegm()`, `localtime()`, `gmtime()`

29.5 `timegm()`

Chuyển `struct tm` với giờ UTC thành `time_t`

Synopsis

Mới trong C23!

```
#include <time.h>

time_t timegm(struct tm *timeptr);
```

Mô tả

Nếu bạn có ngày và giờ ở UTC và muốn chuyển nó sang `time_t` (để rồi `difftime()` nó hay làm gì đó), bạn có thể chuyển đổi bằng hàm này.

Về cơ bản bạn điền các trường trong `struct tm` của mình theo giờ địa phương, và `mktime()` sẽ chuyển chúng sang `time_t` UTC tương ứng.

Vài lưu ý:

- Đừng bận tâm điền `tm_wday` hay `tm_yday`. `mktime()` sẽ điền chúng giúp bạn.
- Spec không nói gì về cờ Daylight Saving `tm_isdst`, nhưng vì UTC miễn nhiễm với DST, tôi đoán là nó bị bỏ qua hoặc được đặt thành `0` giùm ta.

Nếu bạn không có compiler C23 và cần nhập input theo UTC, xem các hàm không chuẩn `timegm()`[6] cho các hệ Unix-like và `_mkgmtime()`[7] cho Windows.

Giá trị trả về

Trả về giờ UTC trong `struct tm` dưới dạng `time_t` calendar time.

Trả về `(time_t)(-1)` nếu có lỗi.

Ví dụ

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

int main(void)
{
    struct tm broken_down_time = {
        .tm_year=82,   // số năm kể từ 1900
        .tm_mon=3,     // số tháng kể từ tháng Một -- [0, 11]
        .tm_mday=12,   // ngày trong tháng -- [1, 31]
        .tm_hour=4,    // số giờ kể từ nửa đêm -- [0, 23]
        .tm_min=00,    // số phút sau giờ -- [0, 59]
        .tm_sec=04,    // số giây sau phút -- [0, 60]
        .tm_isdst=-1,  // cờ Daylight Saving Time
```

[6] https://man.archlinux.org/man/timegm.3.en
[7] https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/mkgmtime-mkgmtime32-mkgmtime64?view=msvc-160

```
    };

    time_t calendar_time = timegm(&broken_down_time);

    char *days[] = {"Sunday", "Monday", "Tuesday",
        "Wednesday", "Furzeday", "Friday", "Saturday"};

    // Cái này sẽ in ra những gì có trong broken_down_time
    printf("UTC        : %s", asctime(gmtime(&calendar_time)));
    printf("Day of week: %s\n\n", days[broken_down_time.tm_wday]);

    // Cái này sẽ in UTC cho giờ địa phương ở trên
    printf("Local time : %s", asctime(localtime(&calendar_time)));
}
```

Xem thêm

`mktime()`, `timegm()`, `localtime()`, `gmtime()`

29.6 `time()`

Lấy calendar time hiện tại

Synopsis

```
#include <time.h>

time_t time(time_t *timer);
```

Mô tả

Trả về calendar time ngay lúc này. Ý tôi là, bây giờ. Không, bây giờ!

Nếu `timer` không phải `NULL`, nó cũng sẽ được nạp thời gian hiện tại.

Giá trị này có thể chuyển thành `struct tm` bằng `localtime()` hoặc `gmtime()`, hoặc in trực tiếp bằng `ctime()`.

Giá trị trả về

Trả về calendar time hiện tại. Cũng nạp `timer` với thời gian hiện tại nếu nó không phải `NULL`.

Hoặc trả về `(time_t)(-1)` nếu không lấy được thời gian vì bạn đã rơi khỏi dòng không-thời gian và/hoặc hệ thống không hỗ trợ thời gian.

Ví dụ

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

int main(void)
{
    time_t now = time(NULL);
```

```
    printf("The local time is %s", ctime(&now));
}
```

Ví dụ output:

```
The local time is Mon Mar  1 18:45:14 2021
```

Xem thêm

`localtime()`, `gmtime()`, `ctime()`

29.7 `timespec_get()`

Lấy thời gian có độ phân giải cao hơn, thường là ngay bây giờ

Synopsis

```
#include <time.h>

int timespec_get(struct timespec *ts, int base);
```

Mô tả

Hàm này nạp thời gian UTC hiện tại (trừ khi được chỉ định khác) vào `struct timespec`, `ts`, bạn đưa vào.

Struct đó có hai trường:

```
struct timespec {
    time_t tv_sec;   // Sô´nguyên giây
    long   tv_nsec;  // Nano giây, 0-999999999
}
```

Nano giây là một phần tỷ của giây. Bạn có thể chia cho 1000000000.0 để chuyển sang giây.

Tham số `base` theo spec chỉ có một giá trị được định nghĩa: `TIME_UTC`. Vậy nên để đảm bảo tính khả chuyển thì đặt nó là cái đó. Việc này sẽ nạp vào `ts` thời gian hiện tại tính bằng số giây kể từ một Epoch[8] (mốc thời gian) do hệ thống định nghĩa, thường là 1 tháng 1 năm 1970 lúc 00:00 UTC.

Implementation (bản cài đặt) của bạn có thể định nghĩa các giá trị khác cho `base`.

Giá trị trả về

Khi `base` là `TIME_UTC`, hàm nạp vào `ts` thời gian UTC hiện tại.

Khi thành công, trả về `base`, các giá trị hợp lệ của nó sẽ luôn khác không. Khi có lỗi, trả về `0`.

Ví dụ

```
struct timespec ts;

timespec_get(&ts, TIME_UTC);
```

[8] https://en.wikipedia.org/wiki/Unix_time

```
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:

```
1614654187 s, 825540756 ns
1614654187.825541 seconds since epoch
```

Đây là một hàm tiện ích để cộng giá trị vào một `struct timespec`, có xử lý giá trị âm và tràn nano giây.

```
#include <stdlib.h>

// Cộng delta giây và delta nano giây vào ts.
// Cho phép giá trị âm. Mỗi thành phần được cộng riêng rẽ.
//
// Trừ đi 1.5 giây khỏi giá trị hiện tại:
//
// timespec_add(&ts, -1, -500000000L);

struct timespec *timespec_add(struct timespec *ts, long dsec, long dnsec)
{
    long sec = (long)ts->tv_sec + dsec;
    long nsec = ts->tv_nsec + dnsec;

    ldiv_t qr = ldiv(nsec, 1000000000L);

    if (qr.rem < 0) {
        nsec = 1000000000L + qr.rem;
        sec += qr.quot - 1;
    } else {
        nsec = qr.rem;
        sec += qr.quot;
    }

    ts->tv_sec = sec;
    ts->tv_nsec = nsec;

    return ts;
}
```

Và đây là vài hàm để chuyển từ `long double` sang `struct timespec` và ngược lại, phòng khi bạn thích suy nghĩ dưới dạng số thập phân. Cách này có ít chữ số có nghĩa hơn so với dùng các giá trị số nguyên.

```
#include <math.h>

// Chuyển struct timespec thành long double
long double timespec_to_ld(struct timespec *ts)
{
    return ts->tv_sec + ts->tv_nsec / 1000000000.0;
}

// Chuyển long double thành struct timespec
```

```
struct timespec ld_to_timespec(long double t)
{
    long double f;
    struct timespec ts;
    ts.tv_nsec = modfl(t, &f) * 1000000000L;
    ts.tv_sec = f;

    return ts;
}
```

Xem thêm

`time()`, `mtx_timedlock()`, `cnd_timedwait()`

29.8 `asctime()`

Trả về phiên bản dễ đọc cho người của `struct tm`

Synopsis

```
#include <time.h>

char *asctime(const struct tm *timeptr)
```

Mô tả

Hàm này lấy thời gian trong `struct tm` và trả về một chuỗi chứa ngày đó ở dạng:

```
Sun Sep 16 01:03:52 1973
```

có kèm một newline ở cuối, khá là không hữu ích. (`strftime()` sẽ cho bạn nhiều linh hoạt hơn.)

Nó y hệt `ctime()`, chỉ khác là nhận `struct tm` thay vì `time_t`.

CẢNH BÁO: Hàm này trả về con trỏ tới một vùng `static char*` mà không phải thread-safe và có thể dùng chung với hàm `ctime()`. Nếu bạn cần thread-safe, hãy dùng `strftime()` hoặc dùng một mutex bao cả `ctime()` lẫn `asctime()`.

Hành vi là không xác định đối với:

- Năm nhỏ hơn 1000
- Năm lớn hơn 9999
- Bất cứ thành viên nào của `timeptr` nằm ngoài khoảng

Giá trị trả về

Trả về con trỏ tới chuỗi ngày dễ đọc cho người.

Ví dụ

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

int main(void)
```

```
{
    time_t now = time(NULL);

    printf("Local: %s", asctime(localtime(&now)));
    printf("UTC  : %s", asctime(gmtime(&now)));
}
```

Ví dụ output:

```
Local: Mon Mar  1 21:17:34 2021
UTC  : Tue Mar  2 05:17:34 2021
```

Xem thêm

`ctime()`, `localtime()`, `gmtime()`

29.9 `ctime()`

Trả về phiên bản dễ đọc cho người của `time_t`

Synopsis

```
#include <time.h>

char *ctime(const time_t *timer);
```

Mô tả

Hàm này nhận thời gian trong một `time_t` và trả về một chuỗi chứa giờ địa phương và ngày ở dạng:

```
Sun Sep 16 01:03:52 1973
```

có kèm một newline ở cuối, khá là không hữu ích. (`strftime()` sẽ cho bạn nhiều linh hoạt hơn.)

Nó y hệt `asctime()`, chỉ khác là nhận `time_t` thay vì `struct tm`.

CẢNH BÁO: Hàm này trả về con trỏ tới một vùng `static char*` mà không phải thread-safe và có thể dùng chung với hàm `asctime()`. Nếu bạn cần thread-safe, hãy dùng `strftime()` hoặc dùng một mutex bao cả `ctime()` lẫn `asctime()`.

Hành vi là không xác định đối với:

- Năm nhỏ hơn 1000
- Năm lớn hơn 9999
- Bất cứ thành viên nào của `timeptr` nằm ngoài khoảng

Giá trị trả về

Một con trỏ tới chuỗi giờ địa phương và ngày dễ đọc cho người.

Ví dụ

```
time_t now = time(NULL);

printf("Local: %s", ctime(&now));
```

Ví dụ output:

```
Local: Mon Mar  1 21:32:23 2021
```

Xem thêm

`asctime()`

29.10 `gmtime()`

Chuyển calendar time thành broken-down time theo UTC

Synopsis

```
#include <time.h>

struct tm *gmtime(const time_t *timer);
```

Mô tả

Nếu bạn có một `time_t`, bạn có thể cho nó chạy qua hàm này để nhận về một `struct tm` đầy thông tin broken-down time theo UTC tương ứng.

Nó y hệt `localtime()`, chỉ khác là làm theo UTC thay vì giờ địa phương.

Khi đã có `struct tm` đó, bạn có thể đưa nó cho `strftime()` để in ra.

CẢNH BÁO: Hàm này trả về con trỏ tới một vùng `static struct tm*` mà không phải thread-safe và có thể dùng chung với hàm `localtime()`. Nếu bạn cần thread-safe, hãy dùng một mutex bao cả `gmtime()` lẫn `localtime()`.

Giá trị trả về

Trả về con trỏ tới broken-down time theo UTC, hoặc `NULL` nếu không lấy được.

Ví dụ

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

int main(void)
{
    time_t now = time(NULL);

    printf("UTC  : %s", asctime(gmtime(&now)));
    printf("Local: %s", asctime(localtime(&now)));
}
```

Ví dụ output:

```
UTC  : Tue Mar  2 05:40:05 2021
Local: Mon Mar  1 21:40:05 2021
```

Xem thêm

`localtime()`, `asctime()`, `strftime()`

29.11 `localtime()`

Chuyển calendar time thành broken-down time theo giờ địa phương

Synopsis

```
#include <time.h>

struct tm *localtime(const time_t *timer);
```

Mô tả

Nếu bạn có một `time_t`, bạn có thể cho nó chạy qua hàm này để nhận về một `struct tm` đầy thông tin broken-down time theo giờ địa phương tương ứng.

Nó y hệt `gmtime()`, chỉ khác là làm theo giờ địa phương thay vì UTC.

Khi đã có `struct tm` đó, bạn có thể đưa nó cho `strftime()` để in ra.

CẢNH BÁO: Hàm này trả về con trỏ tới một vùng `static struct tm*` mà không phải thread-safe và có thể dùng chung với hàm `gmtime()`. Nếu bạn cần thread-safe, hãy dùng một mutex bao cả `gmtime()` lẫn `localtime()`.

Giá trị trả về

Trả về con trỏ tới broken-down time theo giờ địa phương, hoặc `NULL` nếu không lấy được.

Ví dụ

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

int main(void)
{
    time_t now = time(NULL);

    printf("Local: %s", asctime(localtime(&now)));
    printf("UTC  : %s", asctime(gmtime(&now)));
}
```

Ví dụ output:

```
Local: Mon Mar  1 21:40:05 2021
UTC  : Tue Mar  2 05:40:05 2021
```

Xem thêm

`gmtime()`, `asctime()`, `strftime()`

29.12 `strftime()`

In output ngày giờ có định dạng

Synopsis

```
#include <time.h>

size_t strftime(char * restrict s, size_t maxsize,
                const char * restrict format,
                const struct tm * restrict timeptr);
```

Mô tả

Đây là `sprintf()` dành cho các hàm ngày giờ. Nó sẽ nhận một `struct tm` và tạo ra một chuỗi ở gần như bất cứ dạng nào bạn muốn, ví dụ:

```
2021-03-01
Monday, March 1 at 9:54 PM
It's Monday!
```

Đây là phiên bản siêu linh hoạt của `asctime()`. Và còn thread-safe nữa, vì nó không dựa vào một buffer static để giữ kết quả.

Về cơ bản thứ bạn làm là đưa cho nó một điểm đích `s`, và kích thước tối đa tính bằng byte của nó trong `maxsize`. Ngoài ra, cung cấp một chuỗi `format` tương tự như chuỗi format của `printf()`, nhưng với các format specifier khác. Và cuối cùng là một `struct tm` chứa thông tin broken-down time để dùng cho việc in.

Chuỗi `format` hoạt động như thế này, ví dụ:

```
"It's %A, %B %d!"
```

Sẽ cho ra:

```
It's Monday, March 1!
```

`%A` là tên đầy đủ của ngày-trong-tuần, `%B` là tên đầy đủ của tháng, và `%d` là ngày trong tháng. `strftime()` thay thế đúng thứ cần để tạo ra kết quả. Tuyệt vời!

Vậy thì có những format specifier nào? Mừng là bạn đã hỏi!

Tôi sẽ lười và chép nguyên cái bảng này từ spec.

Specifier	Mô tả
`%a`	Tên viết tắt ngày-trong-tuần theo locale. [`tm_wday`]
`%A`	Tên đầy đủ ngày-trong-tuần theo locale. [`tm_wday`]
`%b`	Tên viết tắt của tháng theo locale. [`tm_mon`]
`%B`	Tên đầy đủ của tháng theo locale. [`tm_mon`]
`%c`	Cách biểu diễn ngày và giờ phù hợp theo locale.
`%C`	Năm chia cho 100 rồi cắt lấy số nguyên, dưới dạng số thập phân (00–99). [`tm_year`]

Specifier	Mô tả
`%d`	Ngày trong tháng dưới dạng số thập phân (01–31). [`tm_mday`]
`%D`	Tương đương `"%m/%d/%y"` . [`tm_mon` , `tm_mday` , `tm_year`]
`%e`	Ngày trong tháng dưới dạng số thập phân (1–31); một chữ số đơn được đặt trước bởi một dấu cách. [`tm_mday`]
`%F`	Tương đương “%Y-%m-%d” (định dạng ngày ISO 8601). [`tm_year` , `tm_mon` , `tm_mday`]
`%g`	Hai chữ số cuối của năm dựa-trên-tuần (xem dưới) dưới dạng số thập phân (00–99). [`tm_year` , `tm_wday` , `tm_yday`]
`%G`	Năm dựa-trên-tuần (xem dưới) dưới dạng số thập phân (ví dụ 1997). [`tm_year` , `tm_wday` , `tm_yday`]
`%h`	Tương đương “%b”. [`tm_mon`]
`%H`	Giờ (đồng hồ 24 giờ) dưới dạng số thập phân (00–23). [`tm_hour`]
`%I`	Giờ (đồng hồ 12 giờ) dưới dạng số thập phân (01–12). [`tm_hour`]
`%j`	Ngày trong năm dưới dạng số thập phân (001–366). [`tm_yday`]
`%m`	Tháng dưới dạng số thập phân (01–12).
`%M`	Phút dưới dạng số thập phân (00–59). [`tm_min`]
`%n`	Một ký tự new-line.
`%p`	Cái tương đương theo locale của ký hiệu AM/PM gắn với đồng hồ 12 giờ. [`tm_hour`]
`%r`	Giờ dạng 12 giờ theo locale. [`tm_hour` , `tm_min` , `tm_sec`]
`%R`	Tương đương `"%H:%M"` . [`tm_hour` , `tm_min`]
`%S`	Giây dưới dạng số thập phân (00–60). [`tm_sec`]
`%t`	Một ký tự tab ngang.
`%T`	Tương đương `"%H:%M:%S"` (định dạng giờ ISO 8601). [`tm_hour` , `tm_min` , `tm_sec`]
`%u`	Ngày trong tuần theo ISO 8601 dưới dạng số thập phân (1–7), với thứ Hai là 1. [`tm_wday`]
`%U`	Số tuần trong năm (Chủ Nhật đầu tiên là ngày đầu của tuần 1) dưới dạng số thập phân (00–53). [`tm_year` , `tm_wday` , `tm_yday`]
`%V`	Số tuần theo ISO 8601 (xem dưới) dưới dạng số thập phân (01–53). [`tm_year` , `tm_wday` , `tm_yday`]
`%w`	Ngày trong tuần dưới dạng số thập phân (0–6), với Chủ Nhật là 0.
`%W`	Số tuần trong năm (thứ Hai đầu tiên là ngày đầu của tuần 1) dưới dạng số thập phân (00–53). [`tm_year` , `tm_wday` , `tm_yday`]
`%x`	Cách biểu diễn ngày phù hợp theo locale.
`%X`	Cách biểu diễn giờ phù hợp theo locale.
`%y`	Hai chữ số cuối của năm dưới dạng số thập phân (00–99). [`tm_year`]
`%Y`	Năm dưới dạng số thập phân (ví dụ 1997). [`tm_year`]
`%z`	Độ lệch so với UTC ở định dạng ISO 8601 `"-0430"` (nghĩa là chậm hơn UTC 4 tiếng 30 phút, phía tây Greenwich), hoặc không có ký tự nào nếu không xác định được múi giờ. [`tm_isdst`]
`%Z`	Tên hoặc viết tắt múi giờ theo locale, hoặc không có ký tự nào nếu không xác định được múi giờ. [`tm_isdst`]
`%%`	Một dấu % bình thường

Phù. Đúng là tình yêu.

`%G` , `%g` , và `%v` hơi khác người ở chỗ chúng dùng thứ gọi là năm dựa-trên-tuần ISO 8601. Tôi chưa từng nghe tới. Nhưng, lại chép từ spec, các quy tắc là:

> `%g` , `%G` , và `%V` cho các giá trị theo năm dựa-trên-tuần ISO 8601. Trong hệ này, tuần bắt đầu vào thứ Hai và tuần 1 của năm là tuần bao gồm ngày 4 tháng 1, cũng là tuần bao gồm thứ Năm đầu tiên của năm, và cũng là tuần đầu tiên chứa ít nhất bốn ngày trong năm. Nếu thứ Hai đầu tiên của tháng Một là ngày 2, 3, hoặc 4, các ngày trước đó thuộc về tuần cuối cùng của năm trước; do đó, với thứ Bảy ngày 2 tháng 1 năm 1999, `%G` được thay bằng `1998` và `%V` được thay bằng `53` . Nếu ngày 29, 30, hoặc 31 tháng 12 là thứ Hai, ngày đó và các ngày tiếp theo là phần của

> tuần 1 của năm kế tiếp. Do đó, với thứ Ba ngày 30 tháng 12 năm 1997, `%G` được thay bằng `1998` và `%V` được thay bằng `01`.

Mỗi ngày học được điều mới! Nếu bạn muốn biết thêm, Wikipedia có một trang về nó[9].

Nếu bạn đang ở locale "C", các specifier sinh ra như sau (lại chép từ spec):

Specifier	Mô tả
`%a`	Ba ký tự đầu của `%A`.
`%A`	Một trong `Sunday`, `Monday`, … , `Saturday`.
`%b`	Ba ký tự đầu của `%B`.
`%B`	Một trong `January`, `February`, … , `December`.
`%c`	Tương đương `%a %b %e %T %Y`.
`%p`	Một trong `AM` hoặc `PM`.
`%r`	Tương đương `%I:%M:%S %p`.
`%x`	Tương đương `%m/%d/%y`.
`%X`	Tương đương `%T`.
`%Z`	Tuỳ implementation.

Có thêm các biến thể của format specifier để báo rằng bạn muốn dùng định dạng thay thế của locale. Chúng không tồn tại với tất cả các locale. Đó là một trong các format specifier ở trên, với tiền tố `E` hoặc `O`:

```
%Ec %EC %Ex %EX %Ey %EY %Od %Oe %OH %OI
%Om %OM %OS %Ou %OU %OV %Ow %OW %Oy
```

Các tiền tố `E` và `O` bị bỏ qua ở locale "C".

Giá trị trả về

Trả về tổng số byte đã đặt vào chuỗi kết quả, không tính ký tự NUL kết thúc.

Nếu kết quả không vừa trong chuỗi, trả về 0 và giá trị trong `s` là không xác định.

Ví dụ

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

int main(void)
{
    char s[128];
    time_t now = time(NULL);

    // %c: in ngày theo locale hiện tại
    strftime(s, sizeof s, "%c", localtime(&now));
    puts(s);   // Sun Feb 28 22:29:00 2021

    // %A: tên đầy đủ ngày-trong-tuần
    // %B: tên đầy đủ của tháng
    // %d: ngày trong tháng
    strftime(s, sizeof s, "%A, %B %d", localtime(&now));
    puts(s);   // Sunday, February 28

    // %I: giờ (đồng hồ 12 giờ)
```

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

```
    // %M: phút
    // %S: giây
    // %p: AM hoặc 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: độ lệch múi giờ ISO 8601
    strftime(s, sizeof s, "ISO 8601: %FT%T%z", localtime(&now));
    puts(s);   // ISO 8601: 2021-02-28T22:29:00-0800
}
```

Xem thêm

`ctime()`, `asctime()`

Chapter 30

`<uchar.h>` Hàm Tiện Ích Unicode

Hàm	Mô tả
`c16rtomb()`	Chuyển `char16_t` sang ký tự multibyte
`c32rtomb()`	Chuyển `char32_t` sang ký tự multibyte
`mbrtoc16()`	Chuyển ký tự multibyte sang `char16_t`
`mbrtoc32()`	Chuyển ký tự multibyte sang `char32_t`

Các hàm này là *restartable*, tức là nhiều luồng có thể cùng gọi chúng an toàn. Chúng xử được chuyện đó nhờ giữ biến conversion state riêng (kiểu `mbstate_t`) cho mỗi lần gọi.

30.1 Kiểu

Header này định nghĩa bốn kiểu.

Kiểu	Mô tả
`char16_t`	Kiểu giữ ký tự 16-bit
`char32_t`	Kiểu giữ ký tự 32-bit
`mbstate_t`	Giữ conversion state cho các hàm restartable (cũng định nghĩa trong `<wchar.h>`)
`size_t`	Giữ các count (cũng định nghĩa trong `<stddef.h>`)

String literal cho các kiểu ký tự này là `u` cho `char16_t` và `U` cho `char32_t`.

```
char16_t *str1 = u"Hello, world!";
char32_t *str2 = U"Hello, world!";

char16_t *chr1 = u'A';
char32_t *chr2 = U'B';
```

Lưu ý `char16_t` và `char32_t` *có thể* chứa Unicode. Hoặc không. Nếu `__STDC_UTF_16__` hoặc `__STDC_UTF_32__` được định nghĩa bằng `1`, thì `char16_t` và `char32_t` tương ứng dùng Unicode. Không thì chúng không dùng và giá trị thật lưu lại phụ thuộc locale. Và nếu bạn không xài Unicode, tôi xin chia buồn.

30.2 Vấn đề trên OS X

Header này không có trên OS X—buồn nhỉ. Nếu chỉ cần mấy kiểu thôi, bạn có thể làm:

```
#include <stdint.h>

typedef int_least16_t char16_t;
typedef int_least32_t char32_t;
```

Nhưng nếu bạn còn cần cả các hàm nữa thì tự lo.

30.3 `mbrtoc16()` `mbrtoc32()`

Chuyển ký tự multibyte sang `char16_t` hoặc `char32_t` kiểu restartable

Synopsis

```
#include <uchar.h>

size_t mbrtoc16(char16_t * restrict pc16, const char * restrict s, size_t n,
                mbstate_t * restrict ps);

size_t mbrtoc32(char32_t * restrict pc32, const char * restrict s, size_t n,
                mbstate_t * restrict ps);
```

Description

Cho một source string `s` và buffer đích `pc16` (hoặc `pc32` với `mbrtoc32()`), chuyển ký tự đầu tiên của nguồn thành các `char16_t` (hoặc `char32_t` với `mbrtoc32()`).

Đại loại bạn có một ký tự thường và muốn nó ở dạng `char16_t` hay `char32_t`. Dùng mấy hàm này là xong. Lưu ý chỉ một ký tự được chuyển thôi bất kể `s` có bao nhiêu ký tự.

Khi các hàm scan `s`, bạn không muốn chúng chạy quá khỏi đầu cuối. Nên bạn truyền `n` là số byte tối đa được kiểm. Hàm sẽ dừng sau khi xét đủ từng ấy byte hoặc khi có đủ một ký tự multibyte hoàn chỉnh, tùy cái nào tới trước.

Vì chúng restartable, truyền vào một biến conversion state cho hàm làm việc.

Và kết quả sẽ được đặt vào `pc16` (hoặc `pc32` với `mbrtoc32()`).

Return Value

Khi thành công, hàm trả về số giữa `1` và `n` (bao gồm cả hai đầu) biểu thị số byte tạo nên ký tự multibyte đó.

Hoặc, cũng tính là thành công, chúng có thể trả `0` nếu ký tự nguồn là NUL (giá trị `0`).

Khi không hoàn toàn thành công, chúng có thể trả đủ loại code. Tất cả đều kiểu `size_t`, nhưng là giá trị âm cast sang kiểu đó.

Giá trị trả về	Mô tả
`(size_t)(-1)`	Lỗi encoding—đây không phải chuỗi byte hợp lệ. `errno` được đặt thành `EILSEQ`.
`(size_t)(-2)`	`n` byte đã được xét và tạo thành *một phần* ký tự hợp lệ, nhưng chưa hoàn chỉnh.
`(size_t)(-3)`	Một giá trị tiếp theo của ký tự không biểu diễn được bằng một giá trị đơn. Xem dưới.

Trường hợp `(size_t)(-3)` hơi lạ. Cơ bản là có vài ký tự không biểu diễn được bằng 16 bit nên không lưu được trong `char16_t`. Các ký tự này được lưu trong cái (gọi trong thế giới Unicode) là *surrogate pair*. Tức là có *hai* giá trị 16-bit đi liền nhau biểu diễn một giá trị Unicode lớn hơn.

Ví dụ, nếu bạn muốn đọc ký tự Unicode `\U0001fbc5` (là một hình người que[1]—tôi không để trong text vì font của tôi không render nó được) thì nó dài hơn 16 bit. Nhưng mỗi lần gọi `mbrtoc16()` chỉ trả một `char16_t`!

Nên các lần gọi `mbrtoc16()` kế tiếp sẽ giải giá trị *tiếp theo* trong surrogate pair và trả về `(size_t)(-3)` để báo cho bạn biết chuyện này đã xảy ra.

Bạn cũng có thể truyền `NULL` cho `pc16` hoặc `pc32`. Làm vậy sẽ không lưu kết quả, nhưng bạn có thể dùng nếu chỉ quan tâm đến giá trị trả về của hàm.

Cuối cùng, nếu bạn truyền `NULL` cho `s`, lệnh gọi tương đương với:

```
mbrtoc16(NULL, "", 1, ps)
```

Vì ký tự là NUL trong trường hợp đó, chuyện này có tác dụng đặt state trong `ps` về conversion state ban đầu.

Example

Ví dụ use case thường thấy, lấy hai giá trị ký tự đầu tiên từ chuỗi multibyte `"€Zillion"`:

```
#include <uchar.h>
#include <stdio.h>   // cho printf()
#include <locale.h>  // cho setlocale()
#include <string.h>  // cho memset()

int main(void)
{
    char *s = "\u20acZillion";  // 20ac là "€"
    char16_t pc16;
    size_t r;
    mbstate_t mbs;

    setlocale(LC_ALL, "");
    memset(&mbs, 0, sizeof mbs);

    // Kiểm 8 byte kế xem có ký tự nào trong đó
    r = mbrtoc16(&pc16, s, 8, &mbs);

    printf("%zu\n", r);     // In một giá trị >= 1 (3 trong locale UTF-8)
    printf("%#x\n", pc16);  // In 0x20ac cho "€"

    s += r;  // Sang ký tự kế

    // Kiểm 8 byte kế xem có ký tự nào trong đó
    r = mbrtoc16(&pc16, s, 8, &mbs);

    printf("%zu\n", r);     // In 1
    printf("%#x\n", pc16);  // In 0x5a cho "Z"
}
```

Ví dụ với surrogate pair. Trong trường hợp này ta đọc đủ để có toàn bộ ký tự, nhưng kết quả phải được lưu trong hai `char16_t`, cần hai lần gọi để lấy cả hai.

[1] https://en.wikipedia.org/wiki/Symbols_for_Legacy_Computing

```
#include <uchar.h>
#include <stdio.h>   // cho printf()
#include <string.h>  // cho memset()
#include <locale.h>  // cho setlocale()

int main(void)
{
    char *s = "\U0001fbc5*";   // Glyph hình người que, hơn 16 bit
    char16_t pc16;
    mbstate_t mbs;
    size_t r;

    setlocale(LC_ALL, "");
    memset(&mbs, 0, sizeof mbs);

    r = mbrtoc16(&pc16, s, 8, &mbs);

    printf("%zd\n", r);     // r là 4 byte trong locale UTF-8
    printf("%#x\n", pc16);  // Giá trị đầu của surrogate pair

    s += r;  // Sang ký tự kế´

    r = mbrtoc16(&pc16, s, 8, &mbs);

    printf("%zd\n", r);     // r là (size_t)(-3) ở đây để báo...
    printf("%#x\n", pc16);  // ...Giá trị thứ hai của surrogate pair

    // Vì r là -3, nghĩa là ta vẫn đang xử cùng một ký tự,
    // nên ĐỪNG sang ký tự kế´lần này
    //s += r;  // Comment đi

    r = mbrtoc16(&pc16, s, 8, &mbs);

    printf("%zd\n", r);     // 1 byte cho "*"
    printf("%#x\n", pc16);  // 0x2a cho "*"
}
```

Output trên hệ của tôi, cho thấy ký tự đầu được biểu diễn bởi cặp `(0xd83e, 0xdfc5)` và ký tự thứ hai được biểu diễn bởi `0x2a`:

```
4
0xd83e
-3
0xdfc5
1
0x2a
```

See Also

`c16rtomb()`, `c32rtomb()`

30.4 `c16rtomb()` `c32rtomb()`

Chuyển `char16_t` hoặc `char32_t` sang ký tự multibyte kiểu restartable

Synopsis

```
#include <uchar.h>

size_t c16rtomb(char * restrict s, char16_t c16, mbstate_t * restrict ps);

size_t c32rtomb(char * restrict s, char32_t c32, mbstate_t * restrict ps);
```

Description

Nếu bạn có một ký tự trong `char16_t` hoặc `char32_t`, dùng các hàm này để chuyển sang ký tự multibyte.

Các hàm này tính ra cần bao nhiêu byte cho ký tự multibyte trong locale hiện tại và lưu vào buffer được trỏ bởi `s`.

Nhưng buffer đó phải bự cỡ nào? May là có một macro giúp: nó không cần lớn hơn `MB_CUR_MAX`.

Trường hợp đặc biệt, nếu `s` là `NULL`, nó giống như gọi

```
c16rtomb(buf, L'\0', ps);  // hoặc...
c32rtomb(buf, L'\0', ps);
```

trong đó `buf` là buffer do hệ thống quản mà bạn không truy cập được.

Chuyện này có tác dụng đặt state `ps` về trạng thái ban đầu.

Cuối cùng với surrogate pair (khi ký tự bị chia thành hai `char16_t`), bạn gọi lần đầu với phần tử đầu của cặp—lúc này hàm sẽ trả `0`. Rồi gọi lại với phần tử thứ hai của cặp, hàm sẽ trả số byte và lưu kết quả trong mảng `s`.

Return Value

Trả về số byte đã lưu vào mảng được trỏ bởi `s`.

Trả về 0 nếu việc xử lý ký tự hiện tại chưa xong, như trong trường hợp surrogate pair.

Nếu có lỗi encoding, hàm trả `(size_t)(-1)` và `errno` được đặt thành `EILSEQ`.

Example

```
#include <uchar.h>
#include <stdlib.h>  // cho MB_CUR_MAX
#include <stdio.h>   // cho printf()
#include <string.h>  // cho memset()
#include <locale.h>  // cho setlocale()

int main(void)
{
    char16_t c16 = 0x20ac;  // Unicode cho ký hiệu Euro
    char dest[MB_CUR_MAX];
    size_t r;
    mbstate_t mbs;

    setlocale(LC_ALL, "");
    memset(&mbs, 0, sizeof mbs);  // Reset conversion state

    // Convert
    r = c16rtomb(dest, c16, &mbs);
```

```
    printf("r == %zd\n", r);  // r == 3 trên hệ của tôi

    // Và cái này sẽ in ký hiệu Euro
    printf("dest == \"%s\"\n", dest);
}
```

Output trên hệ của tôi:

```
r == 3
dest == "€"
```

Đây là ví dụ phức tạp hơn, chuyển một ký tự giá trị lớn trong chuỗi multibyte thành surrogate pair (như ví dụ `mbrtoc16()` ở trên) rồi chuyển ngược lại thành chuỗi multibyte để in.

```
#include <uchar.h>
#include <stdlib.h>  // cho MB_CUR_MAX
#include <stdio.h>   // cho printf()
#include <string.h>  // cho memset()
#include <locale.h>  // cho setlocale()

int main(void)
{
    char *src = "\U0001fbc5*";   // Glyph hình người que, hơn 16 bit
    char dest[MB_CUR_MAX];
    char16_t surrogate0, surrogate1;
    mbstate_t mbs;
    size_t r;

    setlocale(LC_ALL, "");
    memset(&mbs, 0, sizeof mbs);  // Reset conversion state

    // Lấy ký tự surrogate đầu
    r = mbrtoc16(&surrogate0, src, 8, &mbs);

    // Lấy ký tự surrogate kê´
    src += r;  // Sang ký tự kê´
    r = mbrtoc16(&surrogate1, src, 8, &mbs);

    printf("Surrogate pair: %#x, %#x\n", surrogate0, surrogate1);

    // Giờ đảo ngược lại
    memset(&mbs, 0, sizeof mbs);  // Reset conversion state

    // Xư ́ký tự surrogate đầu
    r = c16rtomb(dest, surrogate0, &mbs);

    // r nên là 0 lúc này, vì ký tự chưa được xư ́xong.
    // Và dest sẽ chưa có gì xài được... chưa đâu!
    printf("r == %zd\n", r);   // r == 0

    // Xư ́ký tự surrogate thứ hai
    r = c16rtomb(dest, surrogate1, &mbs);

    // Giờ thì ổn rồi. r có sô´byte, và dest giữ ký tự.
    printf("r == %zd\n", r);  // r == 4 trên hệ của tôi
```

```
    // Và cái này sẽ in hình người que, nếu font của bạn hỗ trợ
    printf("dest == \"%s\"\n", dest);
}
```

See Also

`mbrtoc16()`, `mbrtoc32()`

Chapter 31

`<wchar.h>` Xử Lý Wide Character

Hàm	Mô tả
`btowc()`	Chuyển ký tự một byte sang wide character
`fgetwc()`	Lấy một wide character từ một wide stream
`fgetws()`	Đọc một wide string từ một wide stream
`fputwc()`	Ghi một wide character ra một wide stream
`fputws()`	Ghi một wide string ra một wide stream
`fwide()`	Lấy hoặc đặt orientation của stream
`fwprintf()`	Xuất wide có định dạng ra một wide stream
`fwscanf()`	Nhập wide có định dạng từ một wide stream
`getwchar()`	Lấy một wide character từ `stdin`
`getwc()`	Lấy một wide character từ `stdin`
`mbrlen()`	Tính số byte của một ký tự multibyte kiểu restartable
`mbrtowc()`	Chuyển multibyte sang wide character kiểu restartable
`mbsinit()`	Kiểm tra một `mbstate_t` có đang ở conversion state ban đầu không
`mbsrtowcs()`	Chuyển chuỗi multibyte sang chuỗi wide character kiểu restartable
`putwchar()`	Ghi một wide character ra `stdout`
`putwc()`	Ghi một wide character ra `stdout`
`swprintf()`	Xuất wide có định dạng ra một wide string
`swscanf()`	Nhập wide có định dạng từ một wide string
`ungetwc()`	Đẩy một wide character trở lại input stream
`vfwprintf()`	Xuất wide có định dạng variadic ra một wide stream
`vfwscanf()`	Nhập wide có định dạng variadic từ một wide stream
`vswprintf()`	Xuất wide có định dạng variadic ra một wide string
`vswscanf()`	Nhập wide có định dạng variadic từ một wide string
`vwprintf()`	Xuất wide có định dạng variadic
`vwscanf()`	Nhập wide có định dạng variadic
`wcscat()`	Nối wide string kiểu nguy hiểm
`wcschr()`	Tìm một wide character trong một wide string
`wcscmp()`	So sánh wide string
`wcscoll()`	So sánh hai wide string có tính đến locale
`wcscpy()`	Copy một wide string kiểu nguy hiểm
`wcscspn()`	Đếm các ký tự không thuộc tập bắt đầu từ đầu một wide string
`wcsftime()`	Xuất ngày giờ có định dạng
`wcslen()`	Trả về độ dài của một wide string
`wcsncat()`	Nối wide string an toàn hơn
`wcsncmp()`	So sánh wide string, giới hạn độ dài

Hàm	Mô tả
`wcsncpy()`	Copy một wide string an toàn hơn
`wcspbrk()`	Tìm một wide character trong một tập ở một wide string
`wcsrchr()`	Tìm một wide character trong một wide string từ cuối
`wcsrtombs()`	Chuyển chuỗi wide character sang chuỗi multibyte kiểu restartable
`wcsspn()`	Đếm các ký tự thuộc một tập ở đầu một wide string
`wcsstr()`	Tìm một wide string trong một wide string khác
`wcstod()`	Chuyển một wide string sang `double`
`wcstof()`	Chuyển một wide string sang `float`
`wcstok()`	Tách token một wide string
`wcstold()`	Chuyển một wide string sang `long double`
`wcstoll()`	Chuyển một wide string sang `long long`
`wcstol()`	Chuyển một wide string sang `long`
`wcstoull()`	Chuyển một wide string sang `unsigned long long`
`wcstoul()`	Chuyển một wide string sang `unsigned long`
`wcsxfrm()`	Biến đổi một wide string để so sánh dựa trên locale
`wctob()`	Chuyển một wide character sang ký tự một byte
`wctombr()`	Chuyển wide sang multibyte kiểu restartable
`wmemcmp()`	So sánh wide character trong bộ nhớ
`wmemcpy()`	Copy bộ nhớ wide character
`wmemmove()`	Copy bộ nhớ wide character, có thể đè nhau
`wprintf()`	Xuất wide có định dạng
`wscanf()`	Nhập wide có định dạng

Đây là các biến thể wide character của những hàm có trong `<stdio.h>`.

Nhớ là bạn không được mix-and-match các hàm xuất multibyte (như `printf()`) với các hàm xuất wide character (như `wprintf()`). Output stream có một *orientation* (định hướng) hoặc là multibyte hoặc là wide, được đặt ngay lời gọi I/O đầu tiên lên stream đó. (Hoặc có thể đặt bằng `fwide()`.)

Nên chọn một trong hai và bám lấy nó.

Bạn có thể đặc tả wide character constant và string literal bằng cách thêm tiền tố `L` ở đầu:

```
wchar_t *s = L"Hello, world!";
wchar_t c = L'B';
```

Header này cũng giới thiệu kiểu `wint_t` được các hàm I/O ký tự dùng. Đây là kiểu có thể giữ bất kỳ wide character đơn nào, và *cũng* giữ được macro `WEOF` để báo wide end-of-file.

31.1 Hàm Restartable

Cuối cùng, vài lời về các hàm "restartable" có trong đây. Khi conversion đang diễn ra, một số encoding yêu cầu C phải theo dõi một ít *state* (trạng thái) về tiến độ conversion cho đến thời điểm đó.

Với nhiều hàm, C dùng một biến nội bộ cho state đó, dùng chung giữa các lời gọi hàm. Vấn đề là nếu bạn viết code đa luồng, state này có thể bị các luồng khác dẫm đạp.

Để tránh chuyện đó, mỗi luồng cần tự giữ state của riêng nó trong một biến kiểu opaque `mbstate_t`. Và các hàm "restartable" cho phép bạn truyền state đó vào để mỗi luồng dùng biến của riêng mình.

31.2 `wprintf()`, `fwprintf()`, `swprintf()`

Xuất có định dạng với một wide string

Synopsis

```
#include <stdio.h>   // For fwprintf()
#include <wchar.h>

int wprintf(const wchar_t * restrict format, ...);

int fwprintf(FILE * restrict stream, const wchar_t * restrict format, ...);

int swprintf(wchar_t * restrict s, size_t n,
             const wchar_t * restrict format, ...);
```

Description

Đây là các phiên bản wide của `printf()`, `fprintf()](#man-printf), và [`sprintf()‘.

Xem các trang đó để biết cách dùng cụ thể.

Chúng giống hệt nhau trừ cái `format` string là một chuỗi wide character thay vì một chuỗi multibyte.

Và `swprintf()` thì tương tự `snprintf()` ở chỗ cả hai đều nhận kích thước của mảng đích làm tham số.

Một chuyện nữa: precision đặc tả cho một `%s` specifier tương ứng với số wide character được in, không phải số byte. Nếu bạn biết khác biệt nào khác, báo tôi.

Return Value

Trả về số wide character đã xuất, hoặc `-1` nếu có lỗi.

Example

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

int main(void)
{
    char *mbs = "multibyte";
    wchar_t *ws = L"wide";

    wprintf(L"We're all wide for %s and %ls!\n", mbs, ws);

    double pi = 3.14159265358979;
    wprintf(L"pi = %f\n", pi);
}
```

Output:

```
We're all wide for multibyte and wide!
pi = 3.141593
```

See Also

`printf()`, `vwprintf()`

31.3 `wscanf()` `fwscanf()` `swscanf()`

Scan một wide stream hoặc wide string để nhập có định dạng

Synopsis

```
#include <stdio.h>  // for fwscanf()
#include <wchar.h>

int wscanf(const wchar_t * restrict format, ...);

int fwscanf(FILE * restrict stream, const wchar_t * restrict format, ...);

int swscanf(const wchar_t * restrict s, const wchar_t * restrict format, ...);
```

Description

Đây là các biến thể wide của `scanf()`, `fscanf()`, và `sscanf()`.

Xem trang `scanf()` để biết mọi chi tiết.

Return Value

Trả về số mục scan được thành công, hoặc `EOF` nếu có lỗi input nào đó.

Example

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

int main(void)
{
    int quantity;
    wchar_t item[100];

    wprintf(L"Enter \"quantity: item\"\n");

    if (wscanf(L"%d:%99ls", &quantity, item) != 2)
        wprintf(L"Malformed input!\n");
    else
        wprintf(L"You entered: %d %ls\n", quantity, item);
}
```

Output (với input là `12: apples`):

```
Enter "quantity: item"
12: apples
You entered: 12 apples
```

See Also

`scanf()`, `vwscanf()`

31.4 `vwprintf()` `vfwprintf()` `vswprintf()`

Các biến thể của `wprintf()` dùng danh sách tham số biến thiên (`va_list`)

Synopsis

```
#include <stdio.h>   // For vfwprintf()
#include <stdarg.h>
#include <wchar.h>

int vwprintf(const wchar_t * restrict format, va_list arg);

int vswprintf(wchar_t * restrict s, size_t n,
              const wchar_t * restrict format, va_list arg);

int vfwprintf(FILE * restrict stream, const wchar_t * restrict format,
              va_list arg);
```

Description

Các hàm này là biến thể wide character của các hàm `vprintf()`. Bạn có thể xem trang tham khảo đó để biết thêm chi tiết.

Return Value

Trả về số wide character đã lưu, hoặc một giá trị âm nếu có lỗi.

Example

Trong ví dụ này, chúng ta tự làm phiên bản riêng của `wprintf()` tên là `wlogger()` để chấm timestamp vào output. Để ý rằng các lời gọi `wlogger()` có đủ hết những thứ hay ho của `wprintf()`.

```
#include <stdarg.h>
#include <wchar.h>
#include <time.h>

int wlogger(wchar_t *format, ...)
{
    va_list va;
    time_t now_secs = time(NULL);
    struct tm *now = gmtime(&now_secs);

    // In timestamp theo định dạng "YYYY-MM-DD hh:mm:ss : "
    wprintf(L"%04d-%02d-%02d %02d:%02d:%02d : ",
        now->tm_year + 1900, now->tm_mon + 1, now->tm_mday,
        now->tm_hour, now->tm_min, now->tm_sec);

    va_start(va, format);
    int result = vwprintf(format, va);
    va_end(va);
```

```
    wprintf(L"\n");

    return result;
}

int main(void)
{
    int x = 12;
    float y = 3.2;

    wlogger(L"Hello!");
    wlogger(L"x = %d and y = %.2f", x, y);
}
```

Output:

```
2021-03-30 04:25:49 : Hello!
2021-03-30 04:25:49 : x = 12 and y = 3.20
```

See Also

`printf()`, `vprintf()`

31.5 `vwscanf()`, `vfwscanf()`, `vswscanf()`

Các biến thể của `wscanf()` dùng danh sách tham số biến thiên (`va_list`)

Synopsis

```
#include <stdio.h>   // For vfwscanf()
#include <stdarg.h>
#include <wchar.h>

int vwscanf(const wchar_t * restrict format, va_list arg);

int vfwscanf(FILE * restrict stream, const wchar_t * restrict format,
             va_list arg);

int vswscanf(const wchar_t * restrict s, const wchar_t * restrict format,
             va_list arg);
```

Description

Đây là các đối tác wide của tập hàm `vscanf()`. Xem trang tham khảo của chúng để biết chi tiết.

Return Value

Trả về số mục scan được thành công, hoặc `EOF` nếu có lỗi input nào đó.

Example

Tôi phải thú nhận là tôi vắt óc mãi mới nghĩ ra bao giờ thì bạn muốn dùng cái này. Ví dụ hay nhất tôi tìm được là một cái trên Stack Overflow[1] kiểm tra lỗi cho giá trị trả về từ `scanf()` so với giá trị kỳ

[1] https://stackoverflow.com/questions/17017331/c99-vscanf-for-dummies/17018046#17018046

vọng. Một biến thể của cái đó ở dưới.

```
#include <stdarg.h>
#include <wchar.h>
#include <assert.h>

int error_check_wscanf(int expected_count, wchar_t *format, ...)
{
    va_list va;

    va_start(va, format);
    int count = vwscanf(format, va);
    va_end(va);

    // Dòng này sẽ làm chương trình crash nếu điều kiện sai:
    assert(count == expected_count);

    return count;
}

int main(void)
{
    int a, b;
    float c;

    error_check_wscanf(3, L"%d, %d/%f", &a, &b, &c);
    error_check_wscanf(2, L"%d", &a);
}
```

See Also

`wscanf()`

31.6 `getwc()` `fgetwc()` `getwchar()`

Lấy một wide character từ một input stream

Synopsis

```
#include <stdio.h>   // For getwc() and fgetwc()
#include <wchar.h>

wint_t getwchar(void);

wint_t getwc(FILE *stream);

wint_t fgetwc(FILE *stream);
```

Description

Đây là các biến thể wide của `fgetc()`.

`fgetwc()` và `getwc()` giống hệt nhau, chỉ khác `getwc()` có thể được hiện thực dưới dạng macro và được phép evaluate `stream` nhiều lần.

`getwchar()` giống hệt `getwc()` với `stream` là `stdin`.

Tôi chẳng hiểu sao bạn lại dùng `getwc()` thay vì `fgetwc()`, nhưng nếu ai biết thì báo tôi một tiếng.

Return Value

Trả về wide character tiếp theo trong input stream. Trả về `WEOF` ở end-of-file hoặc khi có lỗi.

Nếu có lỗi I/O, cờ lỗi cũng được đặt trên stream.

Nếu gặp byte sequence không hợp lệ, `errno` được đặt thành `ILSEQ`.

Example

Đọc tất cả ký tự từ một file, chỉ xuất ra những chữ 'b' nó tìm thấy trong file:

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

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

    fp = fopen("datafile.txt", "r"); // error check this!

    // this while-statement assigns into c, and then checks against EOF:

    while((c = fgetc(fp)) != WEOF)
        if (c == L'b')
            fputwc(c, stdout);

    fclose(fp);
}
```

See Also

`fputwc`, `fgetws`, `errno`

31.7 `fgetws()`

Đọc một wide string từ một file

Synopsis

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

wchar_t *fgetws(wchar_t * restrict s, int n, FILE * restrict stream);
```

Description

Đây là phiên bản wide của `fgets()`. Xem trang tham khảo của nó để biết chi tiết.

Một ký tự wide `NUL` được dùng để kết thúc chuỗi.

Return Value

Trả về `s` nếu thành công, hoặc con trỏ `NULL` nếu end-of-file hoặc có lỗi.

Example

Ví dụ sau đây đọc từng dòng từ một file và đánh số vào đầu mỗi dòng:

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

#define BUF_SIZE 1024

int main(void)
{
    FILE *fp;
    wchar_t buf[BUF_SIZE];

    fp = fopen("textfile.txt", "r"); // error check this!

    int line_count = 0;

    while ((fgetws(buf, BUF_SIZE, fp)) != NULL)
        wprintf(L"%04d: %ls", ++line_count, buf);

    fclose(fp);
}
```

Output ví dụ cho một file có các dòng (không có số đã thêm):

```
0001: line 1
0002: line 2
0003: something
0004: line 4
```

See Also

`fgetwc()`, `fgets()`

31.8 `putwchar()` `putwc()` `fputwc()`

Ghi một wide character đơn ra console hoặc ra file

Synopsis

```
#include <stdio.h>   // For putwc() and fputwc()
#include <wchar.h>

wint_t putwchar(wchar_t c);

wint_t putwc(wchar_t c, FILE *stream);

wint_t fputwc(wchar_t c, FILE *stream);
```

Description

Đây là các tương đương wide character của nhóm hàm 'fputc()'. Bạn có thể tìm thêm thông tin 'ở phần tham khảo đó'.

`fputwc()` và `putwc()` giống hệt nhau, chỉ khác `putwc()` có thể được hiện thực dưới dạng macro và được phép evaluate `stream` nhiều lần.

`putwchar()` giống hệt `putwc()` với `stream` là `stdin`.

Tôi chẳng hiểu sao bạn lại dùng `putwc()` thay vì `fputwc()`, nhưng nếu ai biết thì báo tôi một tiếng.

Return Value

Trả về wide character đã ghi, hoặc `WEOF` nếu có lỗi.

Nếu là lỗi I/O, cờ lỗi sẽ được đặt cho stream.

Nếu là lỗi encoding, `errno` sẽ được đặt thành `EILSEQ`.

Example

Đọc tất cả ký tự từ một file, chỉ xuất ra những chữ 'b' nó tìm thấy trong file:

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

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

    fp = fopen("datafile.txt", "r"); // error check this!

    // this while-statement assigns into c, and then checks against EOF:

    while((c = fgetc(fp)) != WEOF)
        if (c == L'b')
            fputwc(c, stdout);

    fclose(fp);
}
```

See Also

`fgetwc()`, `fputc()`, `errno`

31.9 `fputws()`

Ghi một wide string ra một file

Synopsis

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

int fputws(const wchar_t * restrict s, FILE * restrict stream);
```

Description

Đây là phiên bản wide của `fputs()`.

Truyền vào một wide string và một output stream, và nó sẽ được ghi ra.

Return Value

Trả về một giá trị không âm nếu thành công, hoặc `EOF` nếu có lỗi.

Example

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

int main(void)
{
    fputws(L"Hello, world!\n", stdout);
}
```

See Also

`fputwc()` `fputs()`

31.10 `fwide()`

Lấy hoặc đặt orientation của stream

Synopsis

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

int fwide(FILE *stream, int mode);
```

Description

Stream có thể là wide-oriented (nghĩa là các hàm wide đang được dùng) hoặc byte-oriented (các hàm multibyte thường đang được dùng). Hoặc, trước khi orientation được chọn, thì là unoriented.

Có hai cách đặt orientation cho một stream unoriented:

- Ngầm: chỉ cần dùng một hàm như `printf()` (byte oriented) hoặc `wprintf()` (wide oriented), và orientation sẽ được đặt.
- Tường minh: dùng hàm này để đặt.

Bạn có thể đặt orientation cho stream bằng cách truyền các số khác nhau vào `mode`:

`mode`	Mô tả
`0`	Không thay đổi orientation
`-1`	Đặt stream thành byte-oriented
`1`	Đặt stream thành wide-oriented

(Tôi nói `-1` và `1` ở đó, nhưng thực ra có thể là bất kỳ số dương hay âm nào.)

Hầu hết mọi người chọn các hàm wide hoặc byte (`printf()` hoặc `wprintf()`) và cứ xài, không bao giờ dùng `fwide()` để đặt orientation.

Và một khi orientation đã được đặt, bạn không đổi được. Nên bạn cũng chẳng dùng `fwide()` cho việc đó được.

Vậy dùng nó để làm gì?

Bạn có thể *kiểm tra* xem một stream đang ở orientation nào bằng cách truyền `0` vào `mode` và kiểm tra giá trị trả về.

Return Value

Trả về số lớn hơn 0 nếu stream là wide-oriented.

Trả về số nhỏ hơn 0 nếu stream là byte-oriented.

Trả về 0 nếu stream là unoriented.

Example

Ví dụ đặt thành byte-oriented:

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

int main(void)
{
    printf("Hello world!\n");  // Ngầm đặt thành byte

    int mode = fwide(stdout, 0);

    printf("Stream is %s-oriented\n", mode < 0? "byte": "wide");
}
```

Output:

```
Hello world!
Stream is byte-oriented
```

Ví dụ đặt thành wide-oriented:

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

int main(void)
{
    wprintf(L"Hello world!\n");  // Ngầm đặt thành wide

    int mode = fwide(stdout, 0);

    wprintf(L"Stream is %ls-oriented\n", mode < 0? L"byte": L"wide");
}
```

Output:

```
Hello world!
Stream is wide-oriented
```

31.11 `ungetwc()`

Đẩy một wide character trở lại input stream

Synopsis

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

wint_t ungetwc(wint_t c, FILE *stream);
```

Description

Đây là biến thể wide character của `ungetc()`.

Nó làm điều ngược với `fgetwc()`, đẩy một ký tự trở lại input stream.

Spec bảo đảm bạn làm được chuyện này một lần liên tiếp. Có thể bạn làm được nhiều lần hơn, nhưng tuỳ implementation. Nếu gọi quá nhiều lần mà không có lời đọc xen vào, có thể sẽ trả về lỗi.

Đặt vị trí file sẽ huỷ mọi ký tự đã được `ungetwc()` đẩy vào mà chưa đọc lại.

Cờ end-of-file sẽ được xoá sau một lời gọi thành công.

Return Value

Trả về giá trị của ký tự đã đẩy nếu thành công, hoặc `WEOF` nếu thất bại.

Example

Ví dụ này đọc một dấu câu, rồi mọi thứ sau nó cho đến dấu câu tiếp theo. Nó trả về dấu câu dẫn đầu và lưu phần còn lại vào một chuỗi.

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

wint_t read_punctstring(FILE *fp, wchar_t *s)
{
    wint_t origpunct, c;

    origpunct = fgetwc(fp);

    if (origpunct == WEOF)  // trả về EOF khi end-of-file
        return WEOF;

    while (c = fgetwc(fp), !iswpunct(c) && c != WEOF)
        *s++ = c;  // lưu vào chuỗi

    *s = L'\0'; // nul-terminate chuỗi

    // nếu đọc được dấu câu cuối cùng, ungetc nó để lần sau fgetc
    // lấy lại được:
    if (iswpunct(c))
        ungetwc(c, fp);

    return origpunct;
}
```

```
int main(void)
{
    wchar_t s[128];
    wint_t c;

    while ((c = read_punctstring(stdin, s)) != WEOF) {
        wprintf(L"%lc: %ls\n", c, s);
    }
}
```

Sample Input:

```
!foo#bar*baz
```

Sample output:

```
!: foo
#: bar
*: baz
```

See Also

`fgetwc()`, `ungetc()`

31.12 `wcstod()` `wcstof()` `wcstold()`

Chuyển một wide string sang số dấu phẩy động

Synopsis

```
#include <wchar.h>

double wcstod(const wchar_t * restrict nptr, wchar_t ** restrict endptr);

float wcstof(const wchar_t * restrict nptr, wchar_t ** restrict endptr);

long double wcstold(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
```

Description

Đây là các đối tác wide của họ hàm `strtod()`. Xem trang tham khảo của chúng để biết chi tiết.

Return Value

Trả về chuỗi đã được chuyển sang giá trị dấu phẩy động.

Trả về `0` nếu không có số hợp lệ trong chuỗi.

Khi overflow, trả về `HUGE_VAL`, `HUGE_VALF`, hoặc `HUGE_VALL` với dấu thích hợp tuỳ kiểu trả về, và `errno` được đặt thành `ERANGE`.

Khi underflow, trả về một số không lớn hơn số dương normalized nhỏ nhất, có dấu thích hợp. Implementation *có thể* đặt `errno` thành `ERANGE`.

Example

```
#include <wchar.h>

int main(void)
{
    wchar_t *inp = L"   123.4567beej";
    wchar_t *badchar;

    double val = wcstod(inp, &badchar);

    wprintf(L"Converted string to %f\n", val);
    wprintf(L"Encountered bad characters: %ls\n", badchar);

    val = wcstod(L"987.654321beej", NULL);
    wprintf(L"Ignoring bad chars: %f\n", val);

    val = wcstod(L"11.2233", &badchar);

    if (*badchar == L'\0')
        wprintf(L"No bad chars: %f\n", val);
    else
        wprintf(L"Found bad chars: %f, %ls\n", val, badchar);
}
```

Output:

```
Converted string to 123.456700
Encountered bad characters: beej
Ignoring bad chars: 987.654321
No bad chars: 11.223300
```

See Also

`wcstol()`, `strtod()`, `errno`

31.13 `wcstol()` `wcstoll()` `wcstoul()` `wcstoull()`

Chuyển một wide string sang giá trị số nguyên

Synopsis

```
#include <wchar.h>

long int wcstol(const wchar_t * restrict nptr,
                wchar_t ** restrict endptr, int base);

long long int wcstoll(const wchar_t * restrict nptr,
                      wchar_t ** restrict endptr, int base);

unsigned long int wcstoul(const wchar_t * restrict nptr,
                          wchar_t ** restrict endptr, int base);

unsigned long long int wcstoull(const wchar_t * restrict nptr,
```

```
                                wchar_t ** restrict endptr, int base);
```

Description

Đây là các đối tác wide của họ hàm `strtol()`, nên xem trang tham khảo của chúng để biết chi tiết.

Return Value

Trả về giá trị số nguyên của chuỗi.

Nếu không tìm được gì, trả về `0`.

Nếu kết quả nằm ngoài phạm vi, giá trị trả về là một trong `LONG_MIN`, `LONG_MAX`, `LLONG_MIN`, `LLONG_MAX`, `ULONG_MAX` hoặc `ULLONG_MAX`, tuỳ trường hợp. Và `errno` được đặt thành `ERANGE`.

Example

```
#include <wchar.h>

int main(void)
{
    // All output in decimal (base 10)

    wprintf(L"%ld\n", wcstol(L"123", NULL, 0));     // 123
    wprintf(L"%ld\n", wcstol(L"123", NULL, 10));    // 123
    wprintf(L"%ld\n", wcstol(L"101010", NULL, 2));  // binary, 42
    wprintf(L"%ld\n", wcstol(L"123", NULL, 8));     // octal, 83
    wprintf(L"%ld\n", wcstol(L"123", NULL, 16));    // hex, 291

    wprintf(L"%ld\n", wcstol(L"0123", NULL, 0));    // octal, 83
    wprintf(L"%ld\n", wcstol(L"0x123", NULL, 0));   // hex, 291

    wchar_t *badchar;
    long int x = wcstol(L"   1234beej", &badchar, 0);

    wprintf(L"Value is %ld\n", x);                  // Value is 1234
    wprintf(L"Bad chars at \"%ls\"\n", badchar);    // Bad chars at "beej"
}
```

Output:

```
123
123
42
83
291
83
291
Value is 1234
Bad chars at "beej"
```

See Also

`wcstod()`, `strtol()`, `errno`, `wcstoimax()`, `wcstoumax()`

31.14 `wcscpy()` `wcsncpy()`

Copy một wide string

Synopsis

```
#include <wchar.h>

wchar_t *wcscpy(wchar_t * restrict s1, const wchar_t * restrict s2);

wchar_t *wcsncpy(wchar_t * restrict s1,
                 const wchar_t * restrict s2, size_t n);
```

Description

Đây là các phiên bản wide của `strcpy()` và `strncpy()`.

Chúng copy một chuỗi đến khi gặp wide NUL. Hoặc, với phiên bản an toàn hơn `wcsncpy()`, đến đó hoặc đến khi `n` wide character đã được copy.

Nếu chuỗi trong `s1` ngắn hơn `n`, `wcsncpy()` sẽ đệm `s2` bằng các wide NUL character cho đến khi chạm wide character thứ `n`.

Dù `wcsncpy()` an toàn hơn vì nó sẽ không bao giờ chạy vượt cuối `s2` (giả sử bạn đặt `n` đúng), nó vẫn không an toàn nếu không tìm thấy NUL trong `n` ký tự đầu của `s1`. Trong trường hợp đó, `s2` sẽ không được NUL-terminate. Luôn đảm bảo `n` lớn hơn độ dài chuỗi `s1`!

Return Value

Trả về `s1`.

Example

```
#include <wchar.h>

int main(void)
{
    wchar_t *s1 = L"Hello!";
    wchar_t s2[10];

    wcsncpy(s2, s1, 10);

    wprintf(L"\"%ls\"\n", s2);  // "Hello!"
}
```

See Also

`wmemcpy()`, `wmemmove()` `strcpy()`, `strncpy()`

31.15 `wmemcpy()` `wmemmove()`

Copy wide character

Synopsis

```
#include <wchar.h>

wchar_t *wmemcpy(wchar_t * restrict s1,
                 const wchar_t * restrict s2, size_t n);

wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

Đây là các phiên bản wide của `memcpy()` và `memmove()`.

Chúng copy `n` wide character từ `s2` vào `s1`.

Chúng giống nhau trừ chuyện `wmemmove()` được bảo đảm hoạt động với các vùng nhớ đè nhau, còn `wmemcpy()` thì không.

Return Value

Cả hai hàm đều trả về con trỏ `s1`.

Example

```
#include <wchar.h>

int main(void)
{
    wchar_t s[100] = L"Goats";
    wchar_t t[100];

    wmemcpy(t, s, 6);       // Copy bộ nhớ không đè nhau

    wmemmove(s + 2, s, 6);  // Copy bộ nhớ đè nhau

    wprintf(L"s is \"%ls\"\n", s);
    wprintf(L"t is \"%ls\"\n", t);
}
```

Output:

```
s is "GoGoats"
t is "Goats"
```

See Also

`wcscpy()`, `wcsncpy()`, `memcpy()`, `memmove()`

31.16 `wcscat()` `wcsncat()`

Nối wide string

Synopsis

```
#include <wchar.h>

wchar_t *wcscat(wchar_t * restrict s1, const wchar_t * restrict s2);

wchar_t *wcsncat(wchar_t * restrict s1,
                 const wchar_t * restrict s2, size_t n);
```

Description

Đây là các biến thể wide của `strcat()` và `strncat()`.

Chúng nối `s2` vào cuối `s1`.

Chúng giống nhau trừ `wcsncat()` cho bạn chọn giới hạn số wide character được nối.

Lưu ý `wcsncat()` luôn thêm một NUL terminator vào cuối, kể cả khi `n` ký tự đã được nối. Nên nhớ chừa chỗ cho cái đó.

Return Value

Cả hai hàm đều trả về con trỏ `s1`.

Example

```
#include <wchar.h>

int main(void)
{
    wchar_t dest[30] = L"Hello";
    wchar_t *src = L", World!";
    wchar_t numbers[] = L"12345678";

    wprintf(L"dest before strcat: \"%ls\"\n", dest); // "Hello"

    wcscat(dest, src);
    wprintf(L"dest after strcat:  \"%ls\"\n", dest); // "Hello, world!"

    wcsncat(dest, numbers, 3); // strcat 3 ký tự đầu của numbers
    wprintf(L"dest after strncat: \"%ls\"\n", dest); // "Hello, world!123"
}
```

See Also

`strcat()`, `strncat()`

31.17 `wcscmp()`, `wcsncmp()`, `wmemcmp()`

So sánh wide string hoặc bộ nhớ

Synopsis

```
#include <wchar.h>

int wcscmp(const wchar_t *s1, const wchar_t *s2);

int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);

int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

Description

Đây là các biến thể wide của `memcmp()`, `strcmp()`, và `strncmp()`.

`wcscmp()` và `wcsncmp()` đều so sánh chuỗi đến ký tự NUL.

`wcsncmp()` còn thêm hạn chế là chỉ so sánh `n` ký tự đầu.

`wmemcmp()` giống `wcsncmp()` trừ chuyện nó không dừng ở NUL.

So sánh được thực hiện dựa trên giá trị ký tự (có thể (hoặc không) là Unicode code point của nó).

Return Value

Trả về 0 nếu cả hai vùng bằng nhau.

Trả về một số âm nếu vùng `s1` trỏ tới nhỏ hơn `s2`.

Trả về một số dương nếu vùng `s1` trỏ tới lớn hơn `s2`.

Example

```
#include <wchar.h>

int main(void)
{
    wchar_t *s1 = L"Muffin";
    wchar_t *s2 = L"Muffin Sandwich";
    wchar_t *s3 = L"Muffin";

    wprintf(L"%d\n", wcscmp(L"Biscuits", L"Kittens")); // <0 since 'B' < 'K'
    wprintf(L"%d\n", wcscmp(L"Kittens", L"Biscuits")); // >0 since 'K' > 'B'

    if (wcscmp(s1, s2) == 0)
        wprintf(L"This won't get printed because the strings differ\n");

    if (wcscmp(s1, s3) == 0)
        wprintf(L"This will print because s1 and s3 are the same\n");

    // hơi lạ...nhưng nếu các chuỗi giống nhau, nó sẽ trả về 0,
    // mà 0 cũng có thể hiểu là "false". Not-false là "true",
    // nên (!wcscmp()) sẽ là true nếu các chuỗi giống nhau. Vâng,
    // lạ thật, nhưng bạn thấy hoài ngoài kia nên cứ quen dần đi:

    if (!wcscmp(s1, s3))
        wprintf(L"The strings are the same!\n");

    if (!wcsncmp(s1, s2, 6))
        wprintf(L"The first 6 characters of s1 and s2 are the same\n");
```

```
}
```

Output:

```
-1
1
This will print because s1 and s3 are the same
The strings are the same!
The first 6 characters of s1 and s2 are the same
```

See Also

`wcscoll()`, `memcmp()`, `strcmp()`, `strncmp()`

31.18 `wcscoll()`

So sánh hai wide string có tính đến locale

Synopsis

```
#include <wchar.h>

int wcscoll(const wchar_t *s1, const wchar_t *s2);
```

Description

Đây là phiên bản wide của `strcoll()`. Xem `trang tham khảo đó` để biết chi tiết.

Cái này chậm hơn `wcscmp()`, nên chỉ dùng khi bạn cần so sánh theo locale.

Return Value

Trả về 0 nếu cả hai vùng bằng nhau trong locale này.

Trả về một số âm nếu vùng `s1` trỏ tới nhỏ hơn `s2` trong locale này.

Trả về một số dương nếu vùng `s1` trỏ tới lớn hơn `s2` trong locale này.

Example

```
#include <wchar.h>
#include <locale.h>

int main(void)
{
    setlocale(LC_ALL, "");

    // Nếu source character set không hỗ trợ "é" trong chuỗi thì
    // có thể thay bằng `\u00e9`, Unicode code point của "é".

    wprintf(L"%d\n", wcscmp(L"é", L"f"));   // Reports é > f, yuck.
    wprintf(L"%d\n", wcscoll(L"é", L"f"));  // Reports é < f, yay!
}
```

See Also

`wcscmp()`, `wcsxfrm()`, `strcoll()`

31.19 `wcsxfrm()`

Biến đổi một wide string để so sánh dựa trên locale

Synopsis

```
#include <wchar.h>

size_t wcsxfrm(wchar_t * restrict s1,
               const wchar_t * restrict s2, size_t n);
```

Description

Đây là biến thể wide của `strxfrm()`. Xem trang tham khảo đó để biết chi tiết.

Return Value

Trả về độ dài của wide string đã biến đổi tính theo wide character.

Nếu giá trị trả về lớn hơn `n`, kết quả trong `s1` coi như bỏ, không đoán được gì.

Example

```
#include <wchar.h>
#include <locale.h>
#include <stdlib.h>

// Biến đổi một chuỗi để so sánh, trả về kết quả đã malloc'd
wchar_t *get_xfrm_str(wchar_t *s)
{
    int len = wcsxfrm(NULL, s, 0) + 1;
    wchar_t *d = malloc(len * sizeof(wchar_t));

    wcsxfrm(d, s, len);

    return d;
}

// Làm một nửa công việc của wcscoll() thường vì chuỗi thứ hai
// đến đã được biến đổi sẵn.
int half_wcscoll(wchar_t *s1, wchar_t *s2_transformed)
{
    wchar_t *s1_transformed = get_xfrm_str(s1);

    int result = wcscmp(s1_transformed, s2_transformed);

    free(s1_transformed);

    return result;
}
```

```
int main(void)
{
    setlocale(LC_ALL, "");

    // Biến đổi trước chuỗi để so sánh
    wchar_t *s = get_xfrm_str(L"éfg");

    // So sánh lặp lại với "éfg"
    wprintf(L"%d\n", half_wcscoll(L"fgh", s));  // "fgh" > "éfg"
    wprintf(L"%d\n", half_wcscoll(L"àbc", s));  // "àbc" < "éfg"
    wprintf(L"%d\n", half_wcscoll(L"ĥij", s));  // "ĥij" > "éfg"

    free(s);
}
```

Output:

```
1
-1
1
```

See Also

`wcscmp()`, `wcscoll()`, `strxfrm()`

31.20 `wcschr()` `wcsrchr()`

Tìm một wide character trong một wide string

Synopsis

```
#include <wchar.h>

// Pre-C23:

wchar_t *wcschr(const wchar_t *s, wchar_t c);

wchar_t *wcsrchr(const wchar_t *s, wchar_t c);

wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);

// C23:

QWchar_t *wcschr(QWchar_t *s, wchar_t c);

QWchar_t *wcsrchr(QWchar_t *s, wchar_t c);

QWchar_t *wmemchr(QWchar_t *s, wchar_t c, size_t n);
```

Description

Đây là các tương đương wide của `strchr()`, `strrchr()`, và `memchr()`.

Chúng tìm wide character trong một wide string từ đầu (`wcschr()`), từ cuối (`wcsrchr()`) hoặc tìm trong một số lượng wide character tuỳ ý (`wmemchr()`).

Return Value

Cả ba hàm đều trả về con trỏ đến wide character tìm được, hoặc `NULL` nếu không tìm thấy (buồn thật).

Example

```
#include <wchar.h>

int main(void)
{
    // "Hello, world!"
    //       ^  ^   ^
    //       A  B   C

    wchar_t *str = L"Hello, world!";
    wchar_t *p;

    p = wcschr(str, ',');       // p bây giờ trỏ đến vị trí A
    p = wcsrchr(str, 'o');      // p bây giờ trỏ đến vị trí B

    p = wmemchr(str, '!', 13);   // p bây giờ trỏ đến vị trí C

    // Lặp để tìm tất cả các chữ 'B'
    str = L"A BIG BROWN BAT BIT BEEJ";

    for(p = wcschr(str, 'B'); p != NULL; p = wcschr(p + 1, 'B')) {
        wprintf(L"Found a 'B' here: %ls\n", p);
    }
}
```

Output:

```
Found a 'B' here: BIG BROWN BAT BIT BEEJ
Found a 'B' here: BROWN BAT BIT BEEJ
Found a 'B' here: BAT BIT BEEJ
Found a 'B' here: BIT BEEJ
Found a 'B' here: BEEJ
```

See Also

`strchr()`, `strrchr()`, `memchr()`

31.21 `wcsspn()` `wcscspn()`

Trả về độ dài của một wide string gồm toàn các ký tự thuộc một tập wide character, hoặc không thuộc một tập wide character

Synopsis

```
#include <wchar.h>

size_t wcsspn(const wchar_t *s1, const wchar_t *s2);

size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
```

Description

Đây là các đối tác wide character của [`strspn()`] (#man-strspn) và `strcspn()`.

Chúng tính độ dài của chuỗi `s1` trỏ tới gồm toàn các ký tự có trong `s2`. Hoặc, với `wcscspn()`, các ký tự *không* có trong `s2`.

Return Value

Độ dài của chuỗi `s1` trỏ tới gồm toàn các ký tự trong `s2` (với `wcsspn()`) hoặc toàn các ký tự *không* có trong `s2` (với `wcscspn()`).

Example

```
#include <wchar.h>

int main(void)
{
    wchar_t str1[] = L"a banana";
    wchar_t str2[] = L"the bolivian navy on maneuvers in the south pacific";
    int n;

    // có bao nhiêu ký tự trong str1 trước khi gặp ký tự không phải nguyên âm?
    n = wcsspn(str1, L"aeiou");
    wprintf(L"%d\n", n);  // n == 1, just "a"

    // có bao nhiêu ký tự trong str1 trước khi gặp ký tự không phải
    // a, b, hoặc space?
    n = wcsspn(str1, L"ab ");
    wprintf(L"%d\n", n);  // n == 4, "a ba"

    // có bao nhiêu ký tự trong str2 trước khi gặp "y"?
    n = wcscspn(str2, L"y");
    wprintf(L"%d\n", n);  // n = 16, "the bolivian nav"
}
```

See Also

`wcschr()`, `wcsrchr()`, `strspn()`

31.22 `wcspbrk()`

Tìm một wide character trong một tập ở một wide string

Synopsis

```
#include <wchar.h>

// Pre-C23:

wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2);

// C23:
```

```
QWchar_t *wcspbrk(QWchar_t *s1, const wchar_t *s2);
```

Description

Đây là biến thể wide character của `strpbrk()`.

Nó tìm vị trí xuất hiện đầu tiên của bất kỳ ký tự nào trong một tập wide character có trong một wide string.

Return Value

Trả về con trỏ đến ký tự đầu tiên trong chuỗi `s1` tồn tại trong chuỗi `s2`.

Hoặc `NULL` nếu không tìm thấy ký tự nào của `s2` trong `s1`.

Example

```
#include <wchar.h>

int main(void)
{
    //  p trỏ vào đây sau wcspbrk
    //                  v
    wchar_t *s1 = L"Hello, world!";
    wchar_t *s2 = L"dow!";  // Match bất kỳ ký tự nào trong đây

    wchar_t *p = wcspbrk(s1, s2);  // p trỏ vào chữ o

    wprintf(L"%ls\n", p);  // "o, world!"
}
```

See Also

`wcschr()`, `wmemchr()`, `strpbrk()`

31.23 `wcsstr()`

Tìm một wide string trong một wide string khác

Synopsis

```
#include <wchar.h>

// Pre-C23:

wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);

// C23:

QWchar_t *wcsstr(QWchar_t *s1, const wchar_t *s2);
```

Description

Đây là biến thể wide của `strstr()`.

Nó tìm vị trí một chuỗi con trong một chuỗi.

Return Value

Trả về con trỏ đến vị trí trong `s1` có chứa `s2`.

Hoặc `NULL` nếu không tìm thấy `s2` trong `s1`.

Example

```
#include <wchar.h>

int main(void)
{
    wchar_t *str = L"The quick brown fox jumped over the lazy dogs.";
    wchar_t *p;

    p = wcsstr(str, L"lazy");
    wprintf(L"%ls\n", p == NULL? L"null": p); // "lazy dogs."

    // p sẽ là NULL sau đoạn này, vì chuỗi "wombat" không có trong str:
    p = wcsstr(str, L"wombat");
    wprintf(L"%ls\n", p == NULL? L"null": p); // "null"
}
```

See Also

`wcschr()`, `wcsrchr()`, `wcsspn()`, `wcscspn()`, `strstr()`

31.24 `wcstok()`

Tách token một wide string

Synopsis

```
#include <wchar.h>
wchar_t *wcstok(wchar_t * restrict s1, const wchar_t * restrict s2,
                wchar_t ** restrict ptr);
```

Description

Đây là phiên bản wide của `strtok()`.

Và, cũng như nó, nó chỉnh sửa chuỗi `s1`. Nên hãy copy nó ra trước nếu muốn giữ chuỗi gốc.

Một khác biệt quan trọng là `wcstok()` có thể threadsafe vì bạn truyền vào con trỏ `ptr` trỏ đến state hiện tại của quá trình biến đổi. Cái này được khởi tạo tự động khi `s1` được truyền vào lần đầu dưới dạng khác `NULL`. (Các lần gọi sau với `s1` là `NULL` làm state cập nhật.)

Return Value

Example

```
#include <wchar.h>

int main(void)
{
    // tách chuôĩ thành chuôĩ các từ cách nhau bởi space
    // hoặc dâú câu
    wchar_t str[] = L"Where is my bacon, dude?";
    wchar_t *token;
    wchar_t *state;

    // Lưu ý câú trúc if-do-while sau đây rất rất rất
    // rất rất thường thấy khi dùng strtok().

    // lấy token đầu tiên (đảm bảo có token đầu tiên!)
    if ((token = wcstok(str, L".,?! ", &state)) != NULL) {
        do {
            wprintf(L"Word: \"%ls\"\n", token);

            // bây giờ, điều kiện tiếp tục của while lấy token
            // tiếp theo (bằng cách truyền NULL làm tham sô´đầu)
            // và tiếp tục nếú token không NULL:
        } while ((token = wcstok(NULL, L".,?! ", &state)) != NULL);
    }
}
```

Output:

```
Word: "Where"
Word: "is"
Word: "my"
Word: "bacon"
Word: "dude"
```

See Also

`strtok()`

31.25 `wcslen()`

Trả về độ dài của một wide string

Synopsis

```
#include <wchar.h>

size_t wcslen(const wchar_t *s);
```

Description

Đây là đối tác wide của `strlen()`.

Return Value

Trả về số wide character trước wide NUL terminator.

Example

```
#include <wchar.h>

int main(void)
{
    wchar_t *s = L"Hello, world!"; // 13 ký tự

    // in "The string is 13 characters long.":

    wprintf(L"The string is %zu characters long.\n", wcslen(s));
}
```

See Also

`strlen()`

31.26 `wcsftime()`

Xuất ngày giờ có định dạng

Synopsis

```
#include <time.h>
#include <wchar.h>

size_t wcsftime(wchar_t * restrict s, size_t maxsize,
                const wchar_t * restrict format,
                const struct tm * restrict timeptr);
```

Description

Đây là tương đương wide của `strftime()`. Xem trang tham khảo đó để biết chi tiết.

`maxsize` ở đây là số wide character tối đa có thể có trong chuỗi kết quả.

Return Value

Nếu thành công, trả về số wide character đã ghi.

Nếu thất bại vì kết quả không vừa chỗ đã cấp, trả về `0` và nội dung của chuỗi có thể là bất kỳ thứ gì.

Example

```
#include <wchar.h>
#include <time.h>

#define BUFSIZE 128

int main(void)
{
```

```
    wchar_t s[BUFSIZE];
    time_t now = time(NULL);

    // %c: in ngày theo locale hiện hành
    wcsftime(s, BUFSIZE, L"%c", localtime(&now));
    wprintf(L"%ls\n", s);   // Sun Feb 28 22:29:00 2021

    // %A: tên đầy đủ của thứ
    // %B: tên đầy đủ của tháng
    // %d: ngày trong tháng
    wcsftime(s, BUFSIZE, L"%A, %B %d", localtime(&now));
    wprintf(L"%ls\n", s);   // Sunday, February 28

    // %I: giờ (đồng hồ 12 giờ)
    // %M: phút
    // %S: giây
    // %p: AM hoặc PM
    wcsftime(s, BUFSIZE, L"It's %I:%M:%S %p", localtime(&now));
    wprintf(L"%ls\n", s);   // It's 10:29:00 PM

    // %F: ISO 8601 yyyy-mm-dd
    // %T: ISO 8601 hh:mm:ss
    // %z: ISO 8601 offset múi giờ
    wcsftime(s, BUFSIZE, L"ISO 8601: %FT%T%z", localtime(&now));
    wprintf(L"%ls\n", s);   // ISO 8601: 2021-02-28T22:29:00-0800
}
```

See Also

`strftime()`

31.27 `btowc()` `wctob()`

Chuyển một ký tự một byte sang một wide character

Synopsis

```
#include <wchar.h>

wint_t btowc(int c);

int wctob(wint_t c);
```

Description

Các hàm này chuyển qua lại giữa ký tự một byte và wide character.

Tuy có `int` trong đây, đừng để nó đánh lừa; thực chất chúng được chuyển thành `unsigned char` bên trong.

Các ký tự trong basic character set được đảm bảo là một byte.

Return Value

`btowc()` trả về ký tự một byte dưới dạng wide character. Trả về `WEOF` nếu `EOF` được truyền vào, hoặc byte không tương ứng wide character hợp lệ.

`wctob()` trả về wide character dưới dạng ký tự một byte. Trả về `EOF` nếu `WEOF` được truyền vào, hoặc wide character không tương ứng ký tự một byte hợp lệ.

Xem `mbtowc()` và `wctomb()` để chuyển multibyte sang wide character.

Example

```
#include <wchar.h>

int main(void)
{
    wint_t wc = btowc('B');    // Chuyển byte đơn sang wide char

    wprintf(L"Wide character: %lc\n", wc);

    unsigned char c = wctob(wc);  // Chuyển ngược về byte đơn

    wprintf(L"Single-byte character: %c\n", c);
}
```

Output:

```
Wide character: B
Single-byte character: B
```

See Also

`mbtowc()`, `wctomb()`

31.28 `mbsinit()`

Kiểm tra một `mbstate_t` có đang ở conversion state ban đầu không

Synopsis

```
#include <wchar.h>

int mbsinit(const mbstate_t *ps);
```

Description

Với một conversion state cho sẵn trong biến `mbstate_t`, hàm này xác định xem nó có đang ở conversion state ban đầu không.

Return Value

Trả về khác 0 nếu giá trị `ps` trỏ tới đang ở conversion state ban đầu, hoặc nếu `ps` là `NULL`.

Trả về `0` nếu giá trị `ps` trỏ tới **không** ở conversion state ban đầu.

Example

Với tôi, ví dụ này chẳng có gì thú vị, nói rằng biến `mbstate_t` luôn luôn ở state ban đầu. Yay.

Nhưng nếu bạn có một encoding có state như 2022-JP, thử nghịch với cái này xem có vào được state trung gian nào không.

Chương trình này có một đoạn code ở đầu báo xem encoding của locale bạn có cần state nào không.

```
#include <locale.h>   // For setlocale()
#include <string.h>   // For memset()
#include <stdlib.h>   // For mbtowc()
#include <wchar.h>

int main(void)
{
    mbstate_t state;
    wchar_t wc[128];

    setlocale(LC_ALL, "");

    int is_state_dependent = mbtowc(NULL, NULL, 0);

    wprintf(L"Is encoding state dependent? %d\n", is_state_dependent);

    memset(&state, 0, sizeof state);  // Đặt về`state ban đầu

    wprintf(L"In initial conversion state? %d\n", mbsinit(&state));

    mbrtowc(wc, "B", 5, &state);

    wprintf(L"In initial conversion state? %d\n", mbsinit(&state));
}
```

See Also

`mbtowc()`, `wctomb()`, `mbrtowc()`, `wcrtomb()`

31.29 `mbrlen()`

Tính số byte của một ký tự multibyte, kiểu restartable

Synopsis

```
#include <wchar.h>

size_t mbrlen(const char * restrict s, size_t n, mbstate_t * restrict ps);
```

Description

Đây là phiên bản restartable của `mblen()`.

Nó xét tối đa `n` byte của chuỗi `s` xem có bao nhiêu byte trong ký tự này.

Conversion state được lưu trong `ps`.

Hàm này không có chức năng của `mblen()` cho phép bạn truy vấn xem encoding ký tự này có stateful không và reset state nội bộ.

Return Value

Trả về số byte cần cho ký tự multibyte này.

Trả về `(size_t)(-1)` nếu dữ liệu trong `s` không phải ký tự multibyte hợp lệ.

Trả về `(size_t)(-2)` nếu dữ liệu trong `s` hợp lệ nhưng chưa đủ một ký tự multibyte hoàn chỉnh.

Example

Nếu character set của bạn không hỗ trợ ký hiệu Euro "€", thay bằng chuỗi escape Unicode `\u20ac`, ở dưới.

```
#include <locale.h>   // For setlocale()
#include <string.h>   // For memset()
#include <wchar.h>

int main(void)
{
    mbstate_t state;
    int len;

    setlocale(LC_ALL, "");

    memset(&state, 0, sizeof state);  // Đặt về`state ban đầù

    len = mbrlen("B", 5, &state);

    wprintf(L"Length of 'B' is %d byte(s)\n", len);

    len = mbrlen("€", 5, &state);

    wprintf(L"Length of '€' is %d byte(s)\n", len);
}
```

Output:

```
Length of 'B' is 1 byte(s)
Length of '€' is 3 byte(s)
```

See Also

`mblen()`

31.30 `mbrtowc()`

Chuyển multibyte sang wide character kiểu restartable

Synopsis

```
#include <wchar.h>

size_t mbrtowc(wchar_t * restrict pwc, const char * restrict s,
               size_t n, mbstate_t * restrict ps);
```

Description

Đây là đối tác restartable của `mbtowc()`.

Nó chuyển các ký tự đơn từ multibyte sang wide, theo dõi conversion state trong biến được `ps` trỏ tới.

Tối đa `n` byte được xét để chuyển sang một wide character.

Hai biến thể sau đây giống nhau và làm state mà `ps` trỏ tới được đặt về conversion state ban đầu:

```
mbrtowc(NULL, NULL, 0, &state);
mbrtowc(NULL, "", 1, &state);
```

Ngoài ra, nếu bạn chỉ quan tâm đến độ dài tính bằng byte của ký tự multibyte, có thể truyền `NULL` cho `pwc` và không có gì được lưu cho wide character:

```
int len = mbrtowc(NULL, "€", 5, &state);
```

Hàm này không có chức năng của `mbtowc()` cho phép bạn truy vấn xem encoding ký tự này có stateful không và reset state nội bộ.

Return Value

Nếu thành công, trả về một số dương tương ứng với số byte trong ký tự multibyte.

Trả về `0` nếu ký tự được encode là wide NUL character.

Trả về `(size_t)(-1)` nếu dữ liệu trong `s` không phải ký tự multibyte hợp lệ.

Trả về `(size_t)(-2)` nếu dữ liệu trong `s` hợp lệ nhưng chưa đủ một ký tự multibyte hoàn chỉnh.

Example

Nếu character set của bạn không hỗ trợ ký hiệu Euro “€”, thay bằng chuỗi escape Unicode `\u20ac`, ở dưới.

```
#include <string.h>  // For memset()
#include <stdlib.h>  // For mbtowc()
#include <locale.h>  // For setlocale()
#include <wchar.h>

int main(void)
{
    mbstate_t state;

    memset(&state, 0, sizeof state);

    setlocale(LC_ALL, "");

    wprintf(L"State dependency: %d\n", mbtowc(NULL, NULL, 0));

    wchar_t wc;
    int bytes;

    bytes = mbrtowc(&wc, "€", 5, &state);

    wprintf(L"L'%lc' takes %d bytes as multibyte char '€'\n", wc, bytes);
}
```

Output trên máy tôi:

```
State dependency: 0
L'€' takes 3 bytes as multibyte char '€'
```

See Also

`mbtowc()`, `wcrtomb()`

31.31 `wcrtomb()`

Chuyển wide sang multibyte kiểu restartable

Synopsis

```
#include <wchar.h>

size_t wcrtomb(char * restrict s, wchar_t wc, mbstate_t * restrict ps);
```

Description

Đây là đối tác restartable của `wctomb()`.

Nó chuyển các ký tự đơn từ wide sang multibyte, theo dõi conversion state trong biến được `ps` trỏ tới.

Mảng đích `s` cần có kích thước tối thiểu `MB_CUR_MAX` [2] byte—bạn sẽ không nhận được thứ gì lớn hơn từ hàm này.

Lưu ý các giá trị trong mảng kết quả này sẽ không được NUL-terminate.

Nếu bạn truyền vào một wide NUL character, kết quả sẽ chứa các byte cần thiết để phục hồi conversion state về state ban đầu, theo sau là một NUL character, và state `ps` trỏ tới sẽ được reset về state ban đầu:

```
// Reset state
wcrtomb(mb, L'\0', &state)
```

Nếu bạn không quan tâm kết quả (nghĩa là bạn chỉ muốn reset state hoặc lấy giá trị trả về), có thể làm vậy bằng cách truyền `NULL` cho `s`:

```
wcrtomb(NULL, L'\0', &state);               // Reset state

int byte_count = wctomb(NULL, "X", &state);  // Count bytes in 'X'
```

Hàm này không có chức năng của `wctomb()` cho phép bạn truy vấn xem encoding ký tự này có stateful không và reset state nội bộ.

Return Value

Nếu thành công, trả về số byte cần để encode wide character này trong locale hiện hành.

Nếu input là wide character không hợp lệ, `errno` sẽ được đặt thành `EILSEQ` và hàm trả về `(size_t)(-1)`. Khi chuyện này xảy ra, conversion state coi như bỏ, bạn cứ reset luôn cho rồi.

[2] Đây là một biến, không phải macro, nên nếu bạn dùng nó để định nghĩa mảng, đó sẽ là mảng có độ dài biến thiên (variable-length array).

Example

Nếu character set của bạn không hỗ trợ ký hiệu Euro "€", thay bằng chuỗi escape Unicode `\u20ac`, ở dưới.

```
#include <string.h>  // For memset()
#include <stdlib.h>  // For mbtowc()
#include <locale.h>  // For setlocale()
#include <wchar.h>

int main(void)
{
    mbstate_t state;

    memset(&state, 0, sizeof state);

    setlocale(LC_ALL, "");

    wprintf(L"State dependency: %d\n", mbtowc(NULL, NULL, 0));

    char mb[10] = {0};
    int bytes = wcrtomb(mb, L'€', &state);

    wprintf(L"L'€' takes %d bytes as multibyte char '%s'\n", bytes, mb);
}
```

See Also

`mbrtowc()`, `wctomb()`, `errno`

31.32 `mbsrtowcs()`

Chuyển một chuỗi multibyte sang chuỗi wide character kiểu restartable

Synopsis

```
#include <wchar.h>

size_t mbsrtowcs(wchar_t * restrict dst, const char ** restrict src,
                 size_t len, mbstate_t * restrict ps);
```

Description

Đây là phiên bản restartable của `mbstowcs()`.

Nó chuyển một chuỗi multibyte sang chuỗi wide character.

Kết quả được đặt vào buffer mà `dst` trỏ tới, và con trỏ `src` được cập nhật để chỉ ra phần chuỗi đã được tiêu thụ (trừ khi `dst` là `NULL`).

Tối đa `len` wide character sẽ được lưu.

Hàm này cũng nhận con trỏ đến biến `mbstate_t` của riêng nó trong `ps` để giữ conversion state.

Bạn có thể đặt `dst` thành `NULL` nếu chỉ quan tâm đến giá trị trả về. Chuyện này có thể hữu ích khi muốn lấy số ký tự trong chuỗi multibyte.

Trong trường hợp bình thường, chuỗi `src` sẽ được tiêu thụ đến ký tự NUL, và kết quả sẽ được lưu vào buffer `dst`, bao gồm cả wide NUL character. Trong trường hợp này, con trỏ được `src` trỏ tới sẽ được đặt thành `NULL`. Và conversion state sẽ được đặt về conversion state ban đầu.

Nếu có sự cố vì chuỗi nguồn không phải chuỗi ký tự hợp lệ, conversion sẽ dừng và con trỏ được `src` trỏ tới sẽ được đặt thành địa chỉ ngay sau ký tự multibyte được biến đổi thành công cuối cùng.

Return Value

Nếu thành công, trả về số ký tự đã được chuyển, không tính NUL terminator.

Nếu chuỗi multibyte không hợp lệ, hàm trả về `(size_t)(-1)` và `errno` được đặt thành `EILSEQ`.

Example

Ở đây chúng ta sẽ chuyển chuỗi “€5 ± π” sang chuỗi wide character:

```
#include <locale.h>  // For setlocale()
#include <string.h>  // For memset()
#include <wchar.h>

#define WIDE_STR_SIZE 10

int main(void)
{
    const char *mbs = "€5 ± π";  // That's the exact price range

    wchar_t wcs[WIDE_STR_SIZE];

    setlocale(LC_ALL, "");

    mbstate_t state;
    memset(&state, 0, sizeof state);

    size_t count = mbsrtowcs(wcs, &mbs, WIDE_STR_SIZE, &state);

    wprintf(L"Wide string L\"%ls\" is %d characters\n", wcs, count);
}
```

Output:

```
Wide string L"€5 ± π" is 6 characters
```

Đây là một ví dụ khác dùng `mbsrtowcs()` để lấy độ dài tính theo ký tự của một chuỗi multibyte kể cả khi chuỗi đầy ký tự multibyte. Cái này ngược với `strlen()`, hàm trả về tổng số byte trong chuỗi.

```
#include <stdio.h>   // For printf()
#include <locale.h>  // For setlocale()

#include <string.h>  // For memset()
#include <stdint.h>  // For SIZE_MAX
#include <wchar.h>

size_t mbstrlen(const char *mbs)
{
    mbstate_t state;

    memset(&state, 0, sizeof state);
```

```
    return mbsrtowcs(NULL, &mbs, SIZE_MAX, &state);
}

int main(void)
{
    setlocale(LC_ALL, "");

    char *mbs = "€5 ± π";  // That's the exact price range

    printf("\"%s\" is %zu characters...\n", mbs, mbstrlen(mbs));
    printf("but it's %zu bytes!\n", strlen(mbs));
}
```

Output trên máy tôi:

```
"€5 ± π" is 6 characters...
but it's 10 bytes!
```

See Also

`mbrtowc()`, `mbstowcs()`, `wcsrtombs()`, `strlen()`, `errno`

31.33 `wcsrtombs()`

Chuyển một chuỗi wide character sang chuỗi multibyte kiểu restartable

Synopsis

```
#include <wchar.h>

size_t wcsrtombs(char * restrict dst, const wchar_t ** restrict src,
                 size_t len, mbstate_t * restrict ps);
```

Description

Nếu bạn có một chuỗi wide character, bạn có thể chuyển nó sang chuỗi ký tự multibyte trong locale hiện hành bằng hàm này.

Tối đa `len` byte dữ liệu sẽ được lưu vào buffer mà `dst` trỏ tới. Conversion sẽ dừng ngay sau khi NUL terminator được copy, hoặc `len` byte đã được copy, hoặc có lỗi nào khác xảy ra.

Nếu `dst` là con trỏ `NULL`, không có kết quả nào được lưu. Bạn có thể làm vậy nếu chỉ quan tâm đến giá trị trả về (thường là số byte chuỗi này dùng trong chuỗi multibyte, không tính NUL terminator).

Nếu `dst` không phải `NULL`, con trỏ được `src` trỏ tới sẽ được chỉnh để cho biết bao nhiêu dữ liệu đã được copy. Nếu ở cuối nó chứa `NULL`, nghĩa là mọi chuyện ổn. Trong trường hợp này, state `ps` sẽ được đặt về conversion state ban đầu.

Nếu `len` đã chạm hoặc có lỗi xảy ra, nó sẽ trỏ tới địa chỉ ngay sau `dst+len`.

Return Value

Nếu mọi chuyện ổn, trả về số byte cần cho chuỗi multibyte, không tính NUL terminator.

Nếu có ký tự nào trong chuỗi không tương ứng ký tự multibyte hợp lệ trong locale hiện hành, nó trả về `(size_t)(-1)` và `EILSEQ` được lưu vào `errno`.

Example

Ở đây chúng ta sẽ chuyển chuỗi wide "€5 ± π" sang chuỗi ký tự multibyte:

```
#include <locale.h>  // For setlocale()
#include <string.h>  // For memset()
#include <wchar.h>

#define MB_STR_SIZE 20

int main(void)
{
    const wchar_t *wcs = L"€5 ± π";  // That's the exact price range

    char mbs[MB_STR_SIZE];

    setlocale(LC_ALL, "");

    mbstate_t state;
    memset(&state, 0, sizeof state);

    size_t count = wcsrtombs(mbs, &wcs, MB_STR_SIZE, &state);

    wprintf(L"Multibyte string \"%s\" is %d bytes\n", mbs, count);
}
```

Đây là một ví dụ helper function khác `malloc()` vừa đủ bộ nhớ để giữ chuỗi đã chuyển, rồi trả về kết quả. (Cái này về sau dĩ nhiên phải free, để tránh leak bộ nhớ.)

```
#include <stdlib.h>  // For malloc()
#include <locale.h>  // For setlocale()
#include <string.h>  // For memset()
#include <stdint.h>  // For SIZE_MAX
#include <wchar.h>

char *get_mb_string(const wchar_t *wcs)
{
    setlocale(LC_ALL, "");

    mbstate_t state;
    memset(&state, 0, sizeof state);

    // Cần copy cái này vì wcsrtombs thay đổi nó
    const wchar_t *p = wcs;

    // Tính số byte cần để giữ kết quả
    size_t bytes_needed = wcsrtombs(NULL, &p, SIZE_MAX, &state);

    // Nếu chuyển chưa đủ đàng hoàng, bỏ qua
    if (bytes_needed == (size_t)(-1))
        return NULL;

    // Cấp chỗ cho kết quả
    char *mbs = malloc(bytes_needed + 1);  // +1 cho NUL terminator

    // Đặt conversion state về state ban đầu
    memset(&state, 0, sizeof state);
```

```
    // Chuyển và lưu kết quả
    wcsrtombs(mbs, &wcs, bytes_needed + 1, &state);

    // Đảm bảo mọi chuyện ổn
    if (wcs != NULL) {
        free(mbs);
        return NULL;
    }

    // Thành công!
    return mbs;
}

int main(void)
{
    char *mbs = get_mb_string(L"€5 ± π");

    wprintf(L"Multibyte result: \"%s\"\n", mbs);

    free(mbs);
}
```

See Also

`wcrtomb()`, `wcstombs()`, `mbsrtowcs()`, `errno`

Chapter 32

`<wctype.h>` Phân Loại và Biến Đổi Wide Character

Hàm	Mô tả
`iswalnum()`	Kiểm tra wide character có là chữ cái hoặc chữ số.
`iswalpha()`	Kiểm tra wide character có là chữ cái
`iswblank()`	Kiểm tra đây có phải wide blank character
`iswcntrl()`	Kiểm tra đây có phải wide control character.
`iswctype()`	Xác định phân loại wide character
`iswdigit()`	Kiểm tra wide character có là chữ số
`iswgraph()`	Kiểm tra wide character có in được mà không phải space
`iswlower()`	Kiểm tra wide character có là chữ thường
`iswprint()`	Kiểm tra wide character có in được
`iswpunct()`	Kiểm tra wide character có là dấu câu
`iswspace()`	Kiểm tra wide character có là whitespace
`iswupper()`	Kiểm tra wide character có là chữ hoa
`iswxdigit()`	Kiểm tra wide character có là chữ số hex
`towctrans()`	Chuyển wide character sang chữ hoa hoặc chữ thường
`towlower()`	Chuyển wide character hoa sang chữ thường
`towupper()`	Chuyển wide character thường sang chữ hoa
`wctrans()`	Hàm phụ trợ cho `towctrans()`
`wctype()`	Hàm phụ trợ cho `iswctype()`

Cái này giống `<ctype.h>` nhưng dành cho wide character (ký tự rộng).

Với nó bạn có thể kiểm tra phân loại ký tự (kiểu "ký tự này có phải whitespace không?") hoặc làm vài chuyển đổi ký tự cơ bản (kiểu "ép ký tự này thành chữ thường").

32.1 `iswalnum()`

Kiểm tra wide character có là chữ cái hoặc chữ số.

Synopsis

```
#include <wctype.h>

int iswalnum(wint_t wc);
```

Description

Về cơ bản là kiểm tra xem một ký tự có là chữ cái (`A` - `Z` hoặc `a` - `z`) hoặc chữ số (`0` - `9`). Nhưng vài ký tự khác cũng có thể tính là hợp lệ tùy theo locale.

Cái này tương đương với việc kiểm tra `iswalpha()` hoặc `iswdigit()` có true không.

Return Value

Trả về true nếu ký tự là chữ cái hoặc chữ số.

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                  testing this char
    //                              v
    wprintf(L"%ls\n", iswalnum(L'a')? L"yes": L"no");  // yes
    wprintf(L"%ls\n", iswalnum(L'B')? L"yes": L"no");  // yes
    wprintf(L"%ls\n", iswalnum(L'5')? L"yes": L"no");  // yes
    wprintf(L"%ls\n", iswalnum(L'?')? L"yes": L"no");  // no
}
```

See Also

`iswalpha()` , `iswdigit()` , `isalnum()`

32.2 `iswalpha()`

Kiểm tra wide character có là chữ cái

Synopsis

```
#include <wctype.h>

int iswalpha(wint_t wc);
```

Description

Về cơ bản là kiểm tra một ký tự có là chữ cái (`A` - `Z` hoặc `a` - `z`). Nhưng vài ký tự khác cũng có thể tính là hợp lệ tùy theo locale. (Nếu ký tự khác được tính, chúng sẽ không phải ký tự điều khiển, chữ số, dấu câu hay space.)

Cái này giống như kiểm tra `iswupper()` hoặc `iswlower()` .

Return Value

Trả về true nếu ký tự là chữ cái.

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                  testing this char
    //                            v
    wprintf(L"%ls\n", iswalpha(L'a')? L"yes": L"no");  // yes
    wprintf(L"%ls\n", iswalpha(L'B')? L"yes": L"no");  // yes
    wprintf(L"%ls\n", iswalpha(L'5')? L"yes": L"no");  // no
    wprintf(L"%ls\n", iswalpha(L'?')? L"yes": L"no");  // no
}
```

See Also

`iswalnum()`, `isalpha()`

32.3 `iswblank()`

Kiểm tra đây có phải wide blank character

Synopsis

```
#include <wctype.h>

int iswblank(wint_t wc);
```

Description

Blank character là whitespace đồng thời được dùng làm dấu ngăn từ trên cùng một dòng. Ở locale “C”, các blank character duy nhất là space và tab.

Các locale khác có thể định nghĩa blank character khác.

Return Value

Trả về true nếu đây là blank character.

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                   testing this char
    //                             v
    wprintf(L"%ls\n", iswblank(L' ')? L"yes": L"no");   // yes
```

```
    wprintf(L"%ls\n", iswblank(L'\t')? L"yes": L"no");  // yes
    wprintf(L"%ls\n", iswblank(L'\n')? L"yes": L"no");  // no
    wprintf(L"%ls\n", iswblank(L'a')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswblank(L'?')? L"yes": L"no");   // no
}
```

See Also

`iswspace()`, `isblank()`

32.4 `iswcntrl()`

Kiểm tra đây có phải wide control character.

Synopsis

```
#include <wctype.h>

int iswcntrl(wint_t wc);
```

Description

Spec khá trống trải ở chỗ này. Nhưng tôi cứ giả định là nó hoạt động giống bản không-wide. Vậy thì xem bản kia.

Control character (ký tự điều khiển) là ký tự không in được, phụ thuộc locale.

Với locale "C", nghĩa là control character nằm trong khoảng 0x00 đến 0x1F (ngay trước ký tự SPACE) và 0x7F (ký tự DEL).

Về cơ bản nếu không phải ký tự ASCII in được (hoặc Unicode nhỏ hơn 128), thì đó là control character trong locale "C".

Chắc là vậy.

Return Value

Trả về true nếu đây là control character.

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                   testing this char
    //                            v
    wprintf(L"%ls\n", iswcntrl(L'\t')? L"yes": L"no");  // yes (tab)
    wprintf(L"%ls\n", iswcntrl(L'\n')? L"yes": L"no");  // yes (newline)
    wprintf(L"%ls\n", iswcntrl(L'\r')? L"yes": L"no");  // yes (return)
    wprintf(L"%ls\n", iswcntrl(L'\a')? L"yes": L"no");  // yes (bell)
    wprintf(L"%ls\n", iswcntrl(L' ')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswcntrl(L'a')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswcntrl(L'?')? L"yes": L"no");   // no
```

```
}
```

See Also

`iscntrl()`

32.5 `iswdigit()`

Kiểm tra wide character có là chữ số

Synopsis

```
#include <wctype.h>

int iswdigit(wint_t wc);
```

Description

Kiểm tra wide character có là chữ số (`0` - `9`).

Return Value

Trả về true nếu ký tự là chữ số.

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                   testing this char
    //                             v
    wprintf(L"%ls\n", iswdigit(L'0')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswdigit(L'5')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswdigit(L'a')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswdigit(L'B')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswdigit(L'?')? L"yes": L"no");   // no
}
```

See Also

`iswalnum()` , `isdigit()`

32.6 `iswgraph()`

Kiểm tra wide character có in được mà không phải space

Synopsis

```
#include <wctype.h>

int iswgraph(wint_t wc);
```

Description

Trả về true nếu đây là ký tự in được (không phải control) và cũng không phải whitespace.

Về cơ bản nếu `iswprint()` true và `iswspace()` false.

Return Value

Trả về true nếu đây là ký tự in được mà không phải space.

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                   testing this char
    //                            v
    wprintf(L"%ls\n", iswgraph(L'0')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswgraph(L'a')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswgraph(L'B')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswgraph(L'?')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswgraph(L' ')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswgraph(L'\n')? L"yes": L"no");  // no
}
```

See Also

`iswprint()`, `iswspace()`, `isgraph()`

32.7 `iswlower()`

Kiểm tra wide character có là chữ thường

Synopsis

```
#include <wctype.h>

int iswlower(wint_t wc);
```

Description

Kiểm tra một ký tự có là chữ thường, trong khoảng `a` - `z`.

Ở các locale khác, có thể có ký tự chữ thường khác. Trong mọi trường hợp, để là chữ thường thì những điều sau phải đúng:

```
!iswcntrl(c) && !iswdigit(c) && !iswpunct(c) && !iswspace(c)
```

Return Value

Trả về true nếu wide character là chữ thường.

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                   testing this char
    //                            v
    wprintf(L"%ls\n", iswlower(L'c')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswlower(L'0')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswlower(L'B')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswlower(L'?')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswlower(L' ')? L"yes": L"no");   // no
}
```

See Also

`islower()`, `iswupper()`, `iswalpha()`, `towupper()`, `towlower()`

32.8 `iswprint()`

Kiểm tra wide character có in được

Synopsis

```
#include <wctype.h>

int iswprint(wint_t wc);
```

Description

Kiểm tra wide character có in được, bao gồm cả space (`' '`). Nên giống như `isgraph()`, chỉ khác là space không bị bỏ rơi ngoài trời lạnh.

Return Value

Trả về true nếu wide character in được, bao gồm cả space (`' '`).

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
```

```
    //                  testing this char
    //                           v
    wprintf(L"%ls\n", iswprint(L'c')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswprint(L'0')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswprint(L' ')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswprint(L'\r')? L"yes": L"no");  // no
}
```

See Also

`isprint()`, `iswgraph()`, `iswcntrl()`

32.9 `iswpunct()`

Kiểm tra wide character có là dấu câu

Synopsis

```
#include <wctype.h>

int iswpunct(wint_t wc);
```

Description

Kiểm tra wide character có là dấu câu.

Ổ bất kỳ locale nào, điều này có nghĩa là:

```
!isspace(c) && !isalnum(c)
```

Return Value

True nếu wide character là dấu câu.

Example

Kết quả có thể khác nhau tùy locale.

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                  testing this char
    //                           v
    wprintf(L"%ls\n", iswpunct(L',')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswpunct(L'!')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswpunct(L'c')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswpunct(L'0')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswpunct(L' ')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswpunct(L'\n')? L"yes": L"no");  // no
}
```

See Also

`ispunct()`, `iswspace()`, `iswalnum()`

32.10 `iswspace()`

Kiểm tra wide character có là whitespace

Synopsis

```
#include <wctype.h>

int iswspace(wint_t wc);
```

Description

Kiểm tra `c` có là ký tự whitespace. Chắc là bao gồm:

- Space (`' '`)
- Formfeed (`'\f'`)
- Newline (`'\n'`)
- Carriage Return (`'\r'`)
- Horizontal Tab (`'\t'`)
- Vertical Tab (`'\v'`)

Các locale khác có thể đặc tả ký tự whitespace khác. `iswalnum()`, `iswgraph()`, và `iswpunct()` đều false với mọi ký tự whitespace.

Return Value

True nếu ký tự là whitespace.

Example

Kết quả có thể khác nhau tùy locale.

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                   testing this char
    //                            v
    wprintf(L"%ls\n", iswspace(L' ')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswspace(L'\n')? L"yes": L"no");  // yes
    wprintf(L"%ls\n", iswspace(L'\t')? L"yes": L"no");  // yes
    wprintf(L"%ls\n", iswspace(L',')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswspace(L'!')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswspace(L'c')? L"yes": L"no");   // no
}
```

See Also

`isspace()`, `iswblank()`

32.11 iswupper()

Kiểm tra wide character có là chữ hoa

Synopsis

```
#include <wctype.h>

int iswupper(wint_t wc);
```

Description

Kiểm tra một ký tự có là chữ hoa trong locale hiện tại.

Để là chữ hoa, những điều sau phải đúng:

```
!iscntrl(c) && !isdigit(c) && !ispunct(c) && !isspace(c)
```

Return Value

Trả về true nếu wide character là chữ hoa.

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                   testing this char
    //                              v
    wprintf(L"%ls\n", iswupper(L'B')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswupper(L'c')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswupper(L'0')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswupper(L'?')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswupper(L' ')? L"yes": L"no");   // no
}
```

See Also

`isupper()`, `iswlower()`, `iswalpha()`, `towupper()`, `towlower()`

32.12 iswxdigit()

Kiểm tra wide character có là chữ số hex

Synopsis

```
#include <wctype.h>

int iswxdigit(wint_t wc);
```

Description

Trả về true nếu wide character là chữ số hex. Cụ thể là nếu nó thuộc `0` - `9`, `a` - `f`, hoặc `A` - `F`.

Return Value

True nếu ký tự là chữ số hex.

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                    testing this char
    //                              v
    wprintf(L"%ls\n", iswxdigit(L'B')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswxdigit(L'c')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswxdigit(L'2')? L"yes": L"no");   // yes
    wprintf(L"%ls\n", iswxdigit(L'G')? L"yes": L"no");   // no
    wprintf(L"%ls\n", iswxdigit(L'?')? L"yes": L"no");   // no
}
```

See Also

`isxdigit()`, `iswdigit()`

32.13 `iswctype()`

Xác định phân loại wide character

Synopsis

```
#include <wctype.h>

int iswctype(wint_t wc, wctype_t desc);
```

Description

Đây là con dao Thụy Sĩ của các hàm phân loại; nó gom hết mấy hàm kia vào một.

Bạn gọi nó kiểu này:

```
if (iswctype(c, wctype("digit")))  // or "alpha" or "space" or...
```

và nó hành xử y như bạn đã gọi:

```
if (iswdigit(c))
```

Khác biệt là bạn có thể đặc tả kiểu matching muốn làm dưới dạng chuỗi lúc runtime, nghe thì tiện.

`iswctype()` dựa vào giá trị trả về của lời gọi `wctype()` để làm việc.

Chôm từ spec, đây là các lời gọi `iswctype()` và các hàm tương đương:

Lời gọi `iswctype()`	Tương đương hard-code
`iswctype(c, wctype("alnum"))`	`iswalnum(c)`
`iswctype(c, wctype("alpha"))`	`iswalpha(c)`
`iswctype(c, wctype("blank"))`	`iswblank(c)`
`iswctype(c, wctype("cntrl"))`	`iswcntrl(c)`
`iswctype(c, wctype("digit"))`	`iswdigit(c)`
`iswctype(c, wctype("graph"))`	`iswgraph(c)`
`iswctype(c, wctype("lower"))`	`iswlower(c)`
`iswctype(c, wctype("print"))`	`iswprint(c)`
`iswctype(c, wctype("punct"))`	`iswpunct(c)`
`iswctype(c, wctype("space"))`	`iswspace(c)`
`iswctype(c, wctype("upper"))`	`iswupper(c)`
`iswctype(c, wctype("xdigit"))`	`iswxdigit(c)`

Xem tài liệu `wctype()` để biết hàm phụ trợ đó hoạt động ra sao.

Return Value

Trả về true nếu wide character `wc` khớp với lớp ký tự ở `desc`.

Example

Kiểm tra một phân loại ký tự nào đó khi không biết phân loại tại thời điểm compile:

```
#include <stdio.h>  // for fflush(stdout)
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    wchar_t c;        // Holds a single wide character (to test)
    char desc[128];   // Holds the character class

    // Get the character and classification from the user
    wprintf(L"Enter a character and character class: ");
    fflush(stdout);
    wscanf(L"%lc %s", &c, desc);

    // Compute the type from the given class
    wctype_t t = wctype(desc);

    if (t == 0)
        // If the type is 0, it's an unknown class
        wprintf(L"Unknown character class: \"%s\"\n", desc);
    else {
        // Otherwise, let's test the character and see if its that
        // classification
        if (iswctype(c, t))
            wprintf(L"Yes! '%lc' is %s!\n", c, desc);
        else
            wprintf(L"Nope! '%lc' is not %s.\n", c, desc);
    }
}
```

Output:

```
Enter a character and character class: 5 digit
Yes! '5' is digit!

Enter a character and character class: b digit
Nope! 'b' is not digit.

Enter a character and character class: x alnum
Yes! 'x' is alnum!
```

See Also

`wctype()`

32.14 `wctype()`

Hàm phụ trợ cho `iswctype()`

Synopsis

```
#include <wctype.h>

wctype_t wctype(const char *property);
```

Description

Hàm này trả về một giá trị opaque cho `property` cho trước, dùng để truyền làm tham số thứ hai cho `iswctype()`.

Giá trị trả về có kiểu `wctype_t`.

Các property hợp lệ trong mọi locale là:

```
"alnum"   "alpha"   "blank"   "cntrl"
"digit"   "graph"   "lower"   "print"
"punct"   "space"   "upper"   "xdigit"
```

Các property khác có thể được định nghĩa tùy category `LC_CTYPE` của locale hiện tại.

Xem trang tham khảo `iswctype()` để biết chi tiết sử dụng.

Return Value

Trả về giá trị `wctype_t` tương ứng với `property` cho trước.

Nếu truyền giá trị không hợp lệ cho `property`, trả về `0`.

Example

Kiểm tra một phân loại ký tự nào đó khi không biết phân loại tại thời điểm compile:

```
#include <stdio.h>  // for fflush(stdout)
#include <wchar.h>
#include <wctype.h>

int main(void)
```

```
{
    wchar_t c;        // Holds a single wide character (to test)
    char desc[128];   // Holds the character class

    // Get the character and classification from the user
    wprintf(L"Enter a character and character class: ");
    fflush(stdout);
    wscanf(L"%lc %s", &c, desc);

    // Compute the type from the given class
    wctype_t t = wctype(desc);

    if (t == 0)
        // If the type is 0, it's an unknown class
        wprintf(L"Unknown character class: \"%s\"\n", desc);
    else {
        // Otherwise, let's test the character and see if its that
        // classification
        if (iswctype(c, t))
            wprintf(L"Yes! '%lc' is %s!\n", c, desc);
        else
            wprintf(L"Nope! '%lc' is not %s.\n", c, desc);
    }
}
```

Output:

```
Enter a character and character class: 5 digit
Yes! '5' is digit!

Enter a character and character class: b digit
Nope! 'b' is not digit.

Enter a character and character class: x alnum
Yes! 'x' is alnum!
```

See Also

`iswctype()`

32.15 `towlower()`

Chuyển wide character hoa sang chữ thường

Synopsis

```
#include <wctype.h>

wint_t towlower(wint_t wc);
```

Description

Nếu ký tự là chữ hoa (tức `iswupper(c)` true), hàm này trả về chữ thường tương ứng.

Các locale khác nhau có thể có chữ hoa và chữ thường khác nhau.

Return Value

Nếu chữ cái `wc` là chữ hoa, phiên bản chữ thường của chữ đó sẽ được trả về theo locale hiện tại.

Nếu chữ cái không phải chữ hoa, `wc` được trả về nguyên không đổi.

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                  changing this char
    //                           v
    wprintf(L"%lc\n", towlower(L'B'));  // b (made lowercase!)
    wprintf(L"%lc\n", towlower(L'e'));  // e (unchanged)
    wprintf(L"%lc\n", towlower(L'!'));  // ! (unchanged)
}
```

See Also

`tolower()`, `towupper()`, `iswlower()`, `iswupper()`

32.16 `towupper()`

Chuyển wide character thường sang chữ hoa

Synopsis

```
#include <wctype.h>

wint_t towupper(wint_t wc);
```

Description

Nếu ký tự là chữ thường (tức `iswlower(c)` true), hàm này trả về chữ hoa tương ứng.

Các locale khác nhau có thể có chữ hoa và chữ thường khác nhau.

Return Value

Nếu chữ cái `wc` là chữ thường, phiên bản chữ hoa của chữ đó sẽ được trả về theo locale hiện tại.

Nếu chữ cái không phải chữ thường, `wc` được trả về nguyên không đổi.

Example

```
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    //                  changing this char
    //                           v
```

```
    wprintf(L"%lc\n", towupper(L'B'));  // B (unchanged)
    wprintf(L"%lc\n", towupper(L'e'));  // E (made uppercase!)
    wprintf(L"%lc\n", towupper(L'!'));  // ! (unchanged)
}
```

See Also

`toupper()`, `towlower()`, `iswlower()`, `iswupper()`

32.17 `towctrans()`

Chuyển wide character sang chữ hoa hoặc chữ thường

Synopsis

```
#include <wctype.h>

wint_t towctrans(wint_t wc, wctrans_t desc);
```

Description

Đây là con dao Thụy Sĩ của các hàm chuyển đổi ký tự; nó gom hết mấy hàm kia vào một. Và "mấy hàm kia" ở đây nghĩa là `towupper()` và `towlower()`, vì đó là tất cả những gì có.

Bạn gọi nó kiểu này:

```
if (towctrans(c, wctrans("toupper")))  // or "tolower"
```

và nó hành xử y như bạn đã gọi:

```
towupper(c);
```

Khác biệt là bạn có thể đặc tả kiểu chuyển đổi muốn làm dưới dạng chuỗi lúc runtime, nghe thì tiện.

`towctrans()` dựa vào giá trị trả về của lời gọi `wctrans()` để làm việc.

Lời gọi `towctrans()`	Tương đương hard-code
`towctrans(c, wctrans("toupper"))`	`towupper(c)`
`towctrans(c, wctrans("tolower"))`	`towlower(c)`

Xem tài liệu `wctrans()` để biết hàm phụ trợ đó hoạt động ra sao.

Return Value

Trả về ký tự `wc` như thể đã chạy qua `towupper()` hoặc `towlower()`, tùy giá trị của `desc`.

Nếu ký tự đã khớp sẵn với phân loại, nó được trả về nguyên xi.

Example

```
#include <stdio.h>  // for fflush(stdout)
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    wchar_t c;        // Holds a single wide character (to test)
    char desc[128];   // Holds the conversion type

    // Get the character and conversion type from the user
    wprintf(L"Enter a character and conversion type: ");
    fflush(stdout);
    wscanf(L"%lc %s", &c, desc);

    // Compute the type from the given conversion type
    wctrans_t t = wctrans(desc);

    if (t == 0)
        // If the type is 0, it's an unknown conversion type
        wprintf(L"Unknown conversion: \"%s\"\n", desc);
    else {
        // Otherwise, let's do the conversion
        wint_t result = towctrans(c, t);
        wprintf(L"'%lc' -> %s -> '%lc'\n", c, desc, result);
    }
}
```

Output trên máy tôi:

```
Enter a character and conversion type: b toupper
'b' -> toupper -> 'B'

Enter a character and conversion type: B toupper
'B' -> toupper -> 'B'

Enter a character and conversion type: B tolower
'B' -> tolower -> 'b'

Enter a character and conversion type: ! toupper
'!' -> toupper -> '!'
```

See Also

`wctrans()`, `towupper()`, `towlower()`

32.18 `wctrans()`

Hàm phụ trợ cho `towctrans()`

Synopsis

```
#include <wctype.h>

wctrans_t wctrans(const char *property);
```

Description

Đây là hàm phụ trợ để sinh tham số thứ hai cho `towctrans()`.

Bạn có thể truyền vào một trong hai thứ cho `property`:

- `toupper` để `towctrans()` hành xử như `towupper()`
- `tolower` để `towctrans()` hành xử như `towlower()`

Return Value

Nếu thành công, trả về giá trị có thể dùng làm tham số `desc` cho `towctrans()`.

Ngược lại, nếu `property` không nhận ra, trả về `0`.

Example

```
#include <stdio.h>  // for fflush(stdout)
#include <wchar.h>
#include <wctype.h>

int main(void)
{
    wchar_t c;        // Holds a single wide character (to test)
    char desc[128];   // Holds the conversion type

    // Get the character and conversion type from the user
    wprintf(L"Enter a character and conversion type: ");
    fflush(stdout);
    wscanf(L"%lc %s", &c, desc);

    // Compute the type from the given conversion type
    wctrans_t t = wctrans(desc);

    if (t == 0)
        // If the type is 0, it's an unknown conversion type
        wprintf(L"Unknown conversion: \"%s\"\n", desc);
    else {
        // Otherwise, let's do the conversion
        wint_t result = towctrans(c, t);
        wprintf(L"'%lc' -> %s -> '%lc'\n", c, desc, result);
    }
}
```

Output trên máy tôi:

```
Enter a character and conversion type: b toupper
'b' -> toupper -> 'B'

Enter a character and conversion type: B toupper
'B' -> toupper -> 'B'
```

```
Enter a character and conversion type: B tolower
'B' -> tolower -> 'b'

Enter a character and conversion type: ! toupper
'!' -> toupper -> '!'
```

See Also

`towctrans()`

Index