**MISRA C:2012**  
**Hướng dẫn sử dụng ngôn ngữ C trong các hệ thống quan trọng**

Tháng 8 năm 2024

**TUYÊN BỐ SỨ MỆNH CỦA MISRA**

Chúng tôi cung cấp các hướng dẫn về thực hành tốt nhất hàng đầu thế giới để ứng dụng an toàn cho cả hệ thống điều khiển nhúng và phần mềm độc lập.

MISRA, Hiệp hội Độ tin cậy Phần mềm Công nghiệp Ô tô, là sự hợp tác giữa các nhà sản xuất, nhà cung cấp linh kiện và các công ty tư vấn kỹ thuật nhằm thúc đẩy thực hành tốt nhất trong việc phát triển các hệ thống nhúng liên quan đến an toàn và các ứng dụng phần mềm khác đòi hỏi sự phức tạp. Để đạt được điều này, MISRA xuất bản các tài liệu cung cấp thông tin dễ tiếp cận cho các kỹ sư và nhà quản lý, và tổ chức các sự kiện nhằm cho phép trao đổi kinh nghiệm giữa các nhà thực hành.

[www.misra.org.uk](http://www.misra.org.uk)

**TUYÊN BỐ TỪ CHỐI TRÁCH NHIỆM**

Tuân thủ các yêu cầu của tài liệu này không tự đảm bảo phần mềm không có lỗi, đáng tin cậy hoặc đảm bảo khả năng di động và tái sử dụng.

Tuân thủ các yêu cầu của tài liệu này, hoặc bất kỳ tiêu chuẩn nào khác, tự nó không miễn trừ các nghĩa vụ pháp lý.

**LỜI NÓI ĐẦU**

Lúc đầu, phiên bản thứ ba của Hướng dẫn MISRA C này có thể trông khá đáng sợ. Vì có kích thước gấp đôi phiên bản trước, người ta có thể nghĩ rằng nó chứa gấp đôi số lượng hướng dẫn, và việc tuân thủ những hướng dẫn đó có thể tốn gấp đôi công sức.

Thực tế, số lượng hướng dẫn tăng không đáng kể, chỉ khoảng 10%. Phần còn lại của sự gia tăng kích thước là do những cải tiến trong hướng dẫn, chẳng hạn như:

* Lý do thuyết phục hơn cho các hướng dẫn;
* Mô tả chính xác hơn;
* Ví dụ mã, thể hiện sự tuân thủ và không tuân thủ, cho hầu hết các hướng dẫn;
* Hướng dẫn chi tiết hơn về việc kiểm tra tuân thủ và quy trình chấp nhận sai lệch;
* Các bảng kiểm tra có thể được sử dụng để hỗ trợ một tuyên bố tuân thủ.

Cuối cùng, tôi muốn nhấn mạnh đến các phần giới thiệu của tài liệu này. Chúng không chỉ chứa các hướng dẫn thực tế về cách sử dụng MISRA C, mà đồng thời, cũng đã được làm rõ hơn so với các phiên bản trước. Tôi khuyến khích tất cả người dùng làm quen với tài liệu này.

Steve Montgomery MA (Cantab), PhD Chủ tịch, Nhóm Công tác MISRA C.

[1. A 6](#_Toc175596666)

[2. B 6](#_Toc175596667)

[3. C 6](#_Toc175596668)

[4. D 6](#_Toc175596669)

[5. E 6](#_Toc175596670)

[6. F 6](#_Toc175596671)

[7. G 6](#_Toc175596672)

[8. Quy tắc 6](#_Toc175596673)

[8.1. Một môi trường C tiêu chuẩn 6](#_Toc175596674)

[8.1.1. Quy tắc 1.1 6](#_Toc175596675)

[8.1.2. Quy tắc 1.2 8](#_Toc175596676)

[8.1.3. Quy tắc 1.3 9](#_Toc175596677)

[8.2. Mã không sử dụng 10](#_Toc175596678)

[8.2.1. Quy tắc 2.1 10](#_Toc175596679)

[8.2.2. Quy tắc 2.2 13](#_Toc175596680)

[8.2.3. Quy tắc 2.3 14](#_Toc175596681)

[8.2.4. Quy tắc 2.4 15](#_Toc175596682)

[8.2.5. Quy tắc 2.5 16](#_Toc175596683)

[8.2.6. Quy tắc 2.6 17](#_Toc175596684)

[8.2.7. Quy tắc 2.7 17](#_Toc175596685)

[8.3. Bình luận (Comments) 18](#_Toc175596686)

[8.3.1. Quy tắc 3.1 18](#_Toc175596687)

[8.3.2. Quy tắc 3.2 19](#_Toc175596688)

[8.4. 8.4 Bộ ký tự và quy ước từ vựng 20](#_Toc175596689)

[8.4.1. Quy tắc 4.1 20](#_Toc175596690)

[8.4.2. Quy tắc 4.2 21](#_Toc175596691)

[8.5. Định danh 22](#_Toc175596692)

[8.5.1. Quy tắc 5.1 23](#_Toc175596693)

[8.5.2. Quy tắc 5.2 24](#_Toc175596694)

[8.5.3. Quy tắc 5.3 26](#_Toc175596695)

[8.5.4. Quy tắc 5.4 29](#_Toc175596696)

[8.5.5. Quy tắc 5.5 31](#_Toc175596697)

[8.5.6. Quy tắc 5.6 32](#_Toc175596698)

[8.5.7. Quy tắc 5.7 34](#_Toc175596699)

[8.5.8. Quy tắc 5.8 36](#_Toc175596700)

[8.5.9. Quy tắc 5.9 37](#_Toc175596701)

[8.6. Các loại 39](#_Toc175596702)

[8.6.1. Quy tắc 6.1 39](#_Toc175596703)

[8.6.2. Quy tắc 6.2 41](#_Toc175596704)

[8.7. Hằng số và hằng số số học 41](#_Toc175596705)

[8.7.1. Quy tắc 7.1 41](#_Toc175596706)

[8.7.2. Quy tắc 7.2 42](#_Toc175596707)

[8.7.3. Quy tắc 7.3 44](#_Toc175596708)

[8.7.4. Quy tắc 7.4 45](#_Toc175596709)

[8.8. Khai báo và định nghĩa 47](#_Toc175596710)

[8.8.1. Quy tắc 8.1 47](#_Toc175596711)

[8.8.2. Quy tắc 8.2 49](#_Toc175596712)

[8.8.3. Quy tắc 8.3 51](#_Toc175596713)

[8.8.4. Quy tắc 8.4 53](#_Toc175596714)

[8.8.5. Quy tắc 8.5 55](#_Toc175596715)

[8.8.6. Quy tắc 8.6 56](#_Toc175596716)

[8.8.7. Quy tắc 8.7 57](#_Toc175596717)

[8.8.8. Quy tắc 8.8 58](#_Toc175596718)

## Tổng quan

MISRA C 2012 là một bộ quy tắc giúp lập trình viên viết mã C an toàn và ít lỗi hơn, đặc biệt trong các hệ thống đòi hỏi độ an toàn cao.

### Mục tiêu của MISRA C

MISRA C ra đời nhằm giảm thiểu các lỗi trong lập trình C bằng cách cung cấp các quy tắc và hướng dẫn cụ thể. Nó đặc biệt cần thiết trong các hệ thống đòi hỏi độ an toàn cao, ví dụ như trong lĩnh vực ô tô, hàng không, y tế,... Những hệ thống này yêu cầu mã nguồn phải được kiểm tra rất kỹ để tránh lỗi gây hậu quả nghiêm trọng. MISRA C không chỉ hữu ích cho các hệ thống an toàn, mà còn cho các ứng dụng cần độ chính xác và độ tin cậy cao.

### Hướng dẫn của MISRA C

* **Tài liệu hướng dẫn**: Đây là tài liệu nhằm giúp các lập trình viên hiểu rõ hơn về các quy tắc của MISRA C và cách áp dụng chúng. Nó giúp lập trình viên phát triển phần mềm an toàn hơn, hiệu quả hơn.
* **Tài liệu tham khảo**: Dành cho những người phát triển các công cụ phân tích mã nguồn, giúp tự động kiểm tra và xác minh mã có tuân theo các quy tắc MISRA C hay không.

### Phiên bản trước của MISRA C

Các phiên bản MISRA C trước đây dựa trên chuẩn ISO C năm 1990 (C90). Nhưng khi chuẩn ISO C năm 1999 (C99) ra đời và được sử dụng rộng rãi, đặc biệt trong hệ thống nhúng (embedded systems), vì vậy MISRA C 2012 ra đời và cập nhật để phù hợp với các chuẩn mới này.

### Mục tiêu của phiên bản MISRA C 2012

* **Hỗ trợ cả hai chuẩn ISO C 1999 và 1990**: Phiên bản mới vẫn duy trì hỗ trợ cho các dự án đang sử dụng chuẩn C cũ (1990), đồng thời hỗ trợ các tính năng mới của chuẩn C năm 1999.
* **Sửa lỗi từ các phiên bản trước**: Các vấn đề trong các phiên bản trước được xem xét và sửa chữa nếu cần thiết.
* **Thêm hướng dẫn mới**: Các hướng dẫn mới được bổ sung dựa trên những lý do hợp lý, tức là các lỗi hoặc vấn đề phát sinh khi lập trình sẽ được cân nhắc để đưa ra các quy tắc mới.
* **Cải tiến quy tắc hiện có**: Các quy tắc cũ được viết lại hoặc điều chỉnh để rõ ràng và dễ hiểu hơn.
* **Loại bỏ các quy tắc lỗi thời**: Nếu một quy tắc nào đó không còn phù hợp hoặc không còn lý do hợp lý để áp dụng, nó sẽ bị loại bỏ.
* **Tăng số lượng quy tắc có thể kiểm tra tự động**: MISRA C 2012 cố gắng tạo ra nhiều quy tắc có thể được kiểm tra bằng các công cụ phân tích tĩnh (static analysis tools). Điều này giúp lập trình viên phát hiện lỗi sớm và dễ dàng hơn trong quá trình phát triển phần mềm.

### Lợi ích của phiên bản MISRA C 2012

* Lập trình viên có thể viết mã C an toàn hơn, ít lỗi hơn.
* Các quy tắc được viết rõ ràng và dễ hiểu, giúp lập trình viên dễ dàng tuân thủ.
* Việc sử dụng các công cụ phân tích tĩnh để kiểm tra mã giúp tăng tốc độ phát hiện lỗi.

## Nền tảng của MISRA C

### Sự phổ biến của ngôn ngữ C

Ngôn ngữ lập trình C rất phổ biến trong phát triển phần mềm, đặc biệt là các hệ thống nhúng và hệ thống quan trọng, do những ưu điểm sau:

* **Sự linh hoạt về trình biên dịch**: Trình biên dịch C có mặt trên rất nhiều nền tảng, từ máy tính cá nhân đến các vi xử lý nhúng. Điều này làm cho C trở thành ngôn ngữ có tính tương thích cao, có thể chạy trên nhiều thiết bị và kiến trúc phần cứng khác nhau.
* **Hiệu suất cao**: Chương trình viết bằng C có thể được biên dịch thành mã máy rất hiệu quả. Điều này giúp tối ưu hóa tốc độ chạy của chương trình và sử dụng tài nguyên hệ thống (CPU, bộ nhớ) một cách tiết kiệm nhất.
* **Tiêu chuẩn quốc tế**: Ngôn ngữ C tuân theo các tiêu chuẩn quốc tế như ISO C, giúp đảm bảo tính nhất quán giữa các nền tảng. Điều này nghĩa là một chương trình viết bằng C sẽ có cơ hội chạy trên nhiều hệ thống khác nhau mà không cần thay đổi nhiều về mã nguồn.
* **Truy cập phần cứng trực tiếp**: C cung cấp cơ chế cho phép lập trình viên truy cập trực tiếp vào phần cứng của hệ thống, ví dụ như địa chỉ bộ nhớ hoặc các thanh ghi của vi xử lý. Điều này rất quan trọng trong lập trình nhúng và các ứng dụng điều khiển phần cứng.
* **Sự phổ biến trong các hệ thống quan trọng**: Ngôn ngữ C đã được sử dụng rộng rãi trong các hệ thống đòi hỏi tính an toàn và độ tin cậy cao như hàng không, ô tô, y tế,... Do đó, đã có rất nhiều kinh nghiệm và kiến thức tích lũy trong việc sử dụng C để xây dựng các ứng dụng quan trọng.
* **Hỗ trợ công cụ phân tích và kiểm tra phần mềm**: Ngôn ngữ C có nhiều công cụ mạnh mẽ hỗ trợ việc phân tích mã nguồn và kiểm tra phần mềm, như các công cụ phân tích tĩnh. Các công cụ này giúp phát hiện lỗi sớm trong quá trình phát triển và cải thiện chất lượng mã nguồn.

### Nhược điểm của ngôn ngữ C

Dù ngôn ngữ C có nhiều ưu điểm, nhưng nó cũng có một số nhược điểm khiến việc lập trình với C trở nên phức tạp hơn, đặc biệt là trong các hệ thống đòi hỏi an toàn cao. MISRA C ra đời để giúp giảm thiểu những nhược điểm này.

#### Định nghĩa ngôn ngữ

Tiêu chuẩn ISO C không định nghĩa tất cả các chi tiết của ngôn ngữ mà để lại một số khía cạnh cho từng trình biên dịch hoặc nền tảng triển khai tự quyết định. Điều này nhằm giúp C tương thích với nhiều loại bộ vi xử lý khác nhau, nhưng lại dẫn đến một số vấn đề:

* **Hành vi không được định nghĩa rõ ràng (undefined behavior)**: Đây là những trường hợp mà chuẩn C không quy định cách hệ thống phải xử lý. Ví dụ, nếu chương trình sử dụng một biến chưa được khởi tạo, hành vi của chương trình sẽ không đoán trước được.
* **Hành vi không được xác định cụ thể (unspecified behavior)**: Đây là những trường hợp mà cách hành xử có thể thay đổi tùy thuộc vào trình biên dịch hoặc môi trường cụ thể, nhưng không nhất thiết phải gây lỗi. Ví dụ, thứ tự đánh giá các toán tử có thể thay đổi giữa các trình biên dịch.
* **Hành vi phụ thuộc vào cách triển khai (implementation-defined behavior)**: Trong trường hợp này, trình biên dịch hoặc môi trường triển khai có thể tự quyết định cách thực hiện, miễn là hành vi đó được tài liệu hóa rõ ràng. Điều này có thể dẫn đến việc chương trình hoạt động khác nhau trên các hệ thống khác nhau.

Những vấn đề trên có thể làm cho mã nguồn khó dự đoán và khó chuyển giữa các hệ thống. Ngoài ra, nếu mã phụ thuộc quá nhiều vào hành vi không được định nghĩa hoặc do triển khai quyết định, việc sử dụng công cụ phân tích tĩnh để kiểm tra mã cũng sẽ gặp khó khăn.

#### Lạm dụng ngôn ngữ

Ngôn ngữ C tuy mạnh mẽ và linh hoạt, nhưng cũng dễ bị lạm dụng, dẫn đến mã nguồn khó đọc và dễ gây nhầm lẫn. Một số ví dụ về lạm dụng trong C:

* **Toán tử dễ gây lỗi**: Một trong những lỗi phổ biến nhất trong C là nhầm lẫn giữa phép gán (=) và phép so sánh (==).

Ví dụ:

|  |
| --- |
| if ( a == b )  // // So sánh a và b có bằng nhau không  if ( a = b )    // Gán b cho a, sau đó kiểm tra xem a có khác 0 không |

Cả hai câu lệnh này đều hợp lệ với trình biên dịch, nhưng chúng có ý nghĩa hoàn toàn khác nhau. Lỗi này thường khó phát hiện vì không gây ra lỗi biên dịch, nhưng có thể dẫn đến lỗi logic rất nghiêm trọng.

#### Hiểu lầm về ngôn ngữ

Ngôn ngữ C có một số đặc điểm khiến lập trình viên dễ hiểu lầm hoặc gặp khó khăn, đặc biệt khi họ chưa quen với cách thức hoạt động của nó:

* **Toán tử và mức độ ưu tiên**: Ngôn ngữ C có rất nhiều toán tử với các mức độ ưu tiên khác nhau. Điều này có thể gây khó khăn cho lập trình viên trong việc hiểu rõ các quy tắc kết hợp và thứ tự thực thi các toán tử. Ví dụ, trong một biểu thức phức tạp với nhiều toán tử, việc xác định toán tử nào được thực thi trước có thể không luôn trực quan, đặc biệt đối với những người mới làm quen với ngôn ngữ. Việc hiểu nhầm về mức độ ưu tiên có thể dẫn đến các lỗi logic khó phát hiện.
* **Quy tắc về kiểu dữ liệu**: Trong C, các phép toán có thể tự động "nâng cấp" các toán hạng lên kiểu dữ liệu rộng hơn. Ví dụ, khi thực hiện một phép toán giữa một số nguyên (int) và một số dấu phẩy động (float), số nguyên có thể tự động được chuyển đổi thành số dấu phẩy động trước khi phép toán diễn ra. Điều này đôi khi không rõ ràng và dễ gây nhầm lẫn, đặc biệt đối với các lập trình viên quen thuộc với các ngôn ngữ lập trình có kiểu dữ liệu chặt chẽ và không cho phép chuyển đổi kiểu ngầm định.

Một ví dụ điển hình về hiểu nhầm này là khi làm việc với các giá trị nhỏ, như kiểu char, có thể tự động được nâng cấp lên kiểu int khi tham gia vào các phép toán, điều này dẫn đến kết quả không như mong đợi. Các lỗi kiểu dữ liệu này có thể không được trình biên dịch cảnh báo, khiến chúng khó phát hiện trong quá trình lập trình.

#### Kiểm tra lỗi thời gian chạy (Runtime Error Checking)

Ngôn ngữ C được thiết kế để tối ưu hóa cho hiệu suất và kích thước mã biên dịch. Tuy nhiên, điều này cũng dẫn đến một số hạn chế trong việc kiểm tra lỗi trong quá trình chạy chương trình. Không giống như một số ngôn ngữ hiện đại có cơ chế kiểm tra lỗi runtime mạnh mẽ, C yêu cầu lập trình viên phải chủ động thực hiện các kiểm tra này. Dưới đây là một số loại lỗi thời gian chạy mà ngôn ngữ C không tự động kiểm tra:

* **Ngoại lệ số học**: Ví dụ, khi xảy ra phép chia cho 0, chương trình có thể gây ra lỗi, nhưng C không có cơ chế tự động bắt và xử lý lỗi này như một số ngôn ngữ khác.
* **Tràn số (integer overflow)**: Khi giá trị của một biến vượt quá giới hạn của kiểu dữ liệu, C sẽ không đưa ra cảnh báo mà tiếp tục thực hiện phép toán. Kết quả có thể không chính xác và gây ra các lỗi logic nghiêm trọng trong chương trình. Ví dụ, với một biến kiểu unsigned int khi tràn số, nó sẽ quay trở về giá trị nhỏ nhất của kiểu đó thay vì cảnh báo lỗi.
* **Lỗi con trỏ**: Con trỏ là một phần quan trọng của ngôn ngữ C, nhưng chúng cũng là một nguồn lớn gây lỗi. C không tự động kiểm tra tính hợp lệ của con trỏ, ví dụ khi truy cập một vùng nhớ không hợp lệ hoặc sử dụng con trỏ chưa được khởi tạo. Điều này có thể dẫn đến lỗi "segmentation fault" (vi phạm bộ nhớ), khiến chương trình dừng đột ngột.
* **Lỗi phạm vi mảng (Array bounds)**: C cho phép lập trình viên truy cập các phần tử của mảng mà không kiểm tra xem chỉ số có nằm trong phạm vi hợp lệ của mảng hay không. Điều này có thể dẫn đến truy cập ngoài phạm vi (out-of-bounds), gây ra các lỗi khó lường, từ việc làm hỏng dữ liệu đến sụp đổ hệ thống.

Trong các ngôn ngữ lập trình có cơ chế quản lý lỗi runtime tốt hơn, chẳng hạn như Java, nhiều lỗi này sẽ được phát hiện và xử lý ngay trong quá trình chạy chương trình, giúp bảo vệ chương trình khỏi các lỗi tiềm ẩn. Tuy nhiên, trong C, lập trình viên cần phải tự thực hiện các kiểm tra này, điều này đòi hỏi sự cẩn thận và kỹ năng. Trong các hệ thống yêu cầu độ an toàn cao như hệ thống nhúng hoặc hệ thống quan trọng (critical systems), việc không xử lý tốt các lỗi runtime có thể dẫn đến những hậu quả nghiêm trọng.

## Lựa chọn Tiêu chuẩn C và MISRA C

### Các phiên bản của tiêu chuẩn C

Ngôn ngữ lập trình C có nhiều phiên bản tiêu chuẩn khác nhau, mỗi phiên bản được quy định bởi các tổ chức tiêu chuẩn quốc tế và thường có sự cải tiến để phù hợp với các yêu cầu mới của lập trình và phần cứng. Dưới đây là một số phiên bản tiêu chuẩn C phổ biến:

* **C90 (ISO/IEC 9899:1990)**: Đây là một trong những phiên bản đầu tiên của ngôn ngữ C được tiêu chuẩn hóa và vẫn còn được sử dụng rộng rãi. Nhiều trình biên dịch hiện nay vẫn hỗ trợ tiêu chuẩn này vì có một lượng lớn mã nguồn cũ được viết dựa trên C90, và các dự án lớn, lâu đời thường cần duy trì tính tương thích với mã cũ.
* **C99 (ISO/IEC 9899:1999)**: Tiêu chuẩn C99 mang lại nhiều cải tiến đáng kể so với C90, bao gồm hỗ trợ thêm các tính năng như biến cục bộ trong khối lệnh, khai báo biến động thời gian chạy (VLA), và nhiều cải tiến khác về cú pháp và tính năng ngôn ngữ.
* **C11 (ISO/IEC 9899:2011)**: Đây là phiên bản cải tiến tiếp theo của C. Tuy nhiên, **Hướng dẫn MISRA C** chưa hỗ trợ trực tiếp tiêu chuẩn này, do C11 ra đời sau khi các hướng dẫn MISRA đã hoàn thiện.

### Lựa chọn giữa Tiêu chuẩn C90 và C99

Khi lựa chọn giữa C90 và C99, bạn cần cân nhắc các yếu tố sau:

* **Khối lượng mã nguồn cũ cần tái sử dụng**: Nếu dự án của bạn có nhiều mã nguồn cũ dựa trên C90, việc chọn C90 có thể giúp bạn duy trì tính tương thích và giảm thiểu việc viết lại hoặc chỉnh sửa mã.
* **Trình biên dịch hỗ trợ bộ vi xử lý mục tiêu**: Tùy vào bộ vi xử lý bạn sử dụng, bạn cần kiểm tra xem trình biên dịch có hỗ trợ phiên bản tiêu chuẩn nào (C90, C99, hay C11). Một số trình biên dịch có thể chỉ hỗ trợ một trong hai tiêu chuẩn này, hoặc cung cấp tùy chọn để lập trình viên chọn tiêu chuẩn phù hợp với dự án.

Ngoài ra, trình biên dịch lý tưởng cần tuân thủ đầy đủ tiêu chuẩn ISO cho phiên bản C mà bạn chọn, và nhà cung cấp trình biên dịch nên cung cấp thông tin về các thử nghiệm tuân thủ để đảm bảo tính tương thích.

### Cách lựa chọn trình biên dịch

Khi gặp khó khăn trong việc chọn trình biên dịch, bạn có thể làm theo các bước sau:

* **Kiểm tra quy trình phát triển phần mềm của nhà phát triển trình biên dịch**: Ví dụ, xem xét liệu nhà phát triển có tuân thủ các quy trình đạt chuẩn ISO 9001, đảm bảo chất lượng sản phẩm.
* **Đọc đánh giá từ cộng đồng**: Tham khảo các phản hồi và kinh nghiệm sử dụng từ những lập trình viên khác trong cộng đồng.
* **Tự kiểm thử**: Bạn có thể kiểm tra trình biên dịch bằng cách biên dịch các ứng dụng mẫu hoặc sử dụng các bộ kiểm thử tuân thủ độc lập để đảm bảo chất lượng và tính tuân thủ.

### Công cụ phân tích tĩnh

**Công cụ phân tích tĩnh** là các công cụ tự động kiểm tra mã nguồn để đảm bảo rằng mã tuân thủ các hướng dẫn của MISRA C và không có lỗi tiềm ẩn. Mặc dù có thể kiểm tra mã nguồn theo cách thủ công, điều này rất tốn thời gian và dễ mắc lỗi. Do đó, việc sử dụng công cụ phân tích tĩnh là rất quan trọng và cần thiết trong bất kỳ quy trình kiểm tra mã nào.

#### Các tiêu chí lựa chọn công cụ phân tích tĩnh

Một công cụ phân tích tĩnh tốt cần đảm bảo các yêu cầu sau:

* **Phát hiện tất cả các vi phạm MISRA C**: Công cụ phải có khả năng nhận diện tất cả các vi phạm của hướng dẫn MISRA C trong mã nguồn.
* **Giảm thiểu cảnh báo sai**: Một trong những vấn đề với công cụ phân tích tĩnh là có thể tạo ra cảnh báo sai (false positives), tức là cảnh báo về các lỗi không thực sự tồn tại. Một công cụ lý tưởng nên chỉ báo cáo các lỗi thực sự và giảm thiểu tối đa các cảnh báo sai.

#### Cân bằng giữa tốc độ và độ chính xác

* **Công cụ nhanh**: Các công cụ phân tích tĩnh nhanh thường chỉ mất vài giây để chạy, nhưng có thể tạo ra nhiều cảnh báo sai hơn do quá trình phân tích không sâu.
* **Công cụ kỹ lưỡng**: Ngược lại, các công cụ phân tích kỹ lưỡng hơn có thể mất nhiều thời gian (thậm chí vài ngày) để phân tích toàn bộ mã nguồn, nhưng chúng có độ chính xác cao hơn và ít cảnh báo sai hơn.

Do đó, bạn cần phải cân nhắc giữa **tốc độ phân tích** và **độ chính xác của kết quả** khi chọn công cụ phân tích tĩnh. Đôi khi, việc kết hợp sử dụng nhiều công cụ phân tích khác nhau cũng là một giải pháp hợp lý để đảm bảo bao phủ toàn diện các vấn đề tiềm ẩn trong mã nguồn.

#### Công cụ phân tích tĩnh và yêu cầu an toàn

Trong các hệ thống yêu cầu độ an toàn cao, các tiêu chuẩn như **IEC 61508**, **ISO 26262**, và **DO-178C** cũng yêu cầu các công cụ phân tích tĩnh phải đáp ứng một số tiêu chí về trình độ chuyên môn và tuân thủ các quy trình an toàn. Điều này đảm bảo rằng công cụ phân tích có đủ khả năng kiểm tra và xác nhận rằng mã nguồn đáp ứng các yêu cầu an toàn nghiêm ngặt của hệ thống.

## Kiến thức cần thiết

### Kiến thức nền tảng

Để đảm bảo lập trình viên C có đủ kỹ năng và kiến thức cần thiết, các khóa đào tạo chuyên biệt là rất quan trọng. Một số lĩnh vực chính cần đào tạo bao gồm:

* **Sử dụng C cho các ứng dụng nhúng**: Trong các hệ thống nhúng, lập trình viên cần hiểu cách mã C tương tác trực tiếp với phần cứng. Điều này bao gồm làm việc với các thanh ghi, quản lý tài nguyên hệ thống như bộ nhớ và bộ đếm thời gian, cũng như hiểu cách trình biên dịch tối ưu mã cho các hệ thống tài nguyên giới hạn.
* **Sử dụng C trong các hệ thống có độ an toàn và toàn vẹn cao**: Các hệ thống đòi hỏi tính an toàn cao như hàng không, y tế, hoặc ô tô cần lập trình viên hiểu rõ về các phương pháp lập trình đảm bảo tính ổn định, đáng tin cậy, và an toàn. MISRA C thường được sử dụng trong các hệ thống này, nên lập trình viên cần nắm vững các quy tắc của nó.

Ngoài ra, với các **trình biên dịch** và **công cụ phân tích tĩnh** là những phần mềm phức tạp, việc đào tạo chuyên sâu về cách sử dụng chúng là cần thiết. Các khóa học về cách kiểm tra mã theo tiêu chuẩn MISRA C sẽ giúp lập trình viên dễ dàng nhận diện và khắc phục vi phạm trong mã nguồn.

### Hiểu về trình biên dịch

Trong ngữ cảnh của MISRA C, "Trình biên dịch" không chỉ đơn giản là phần mềm biên dịch mà còn bao gồm nhiều công cụ liên quan khác như:

* **Trình liên kết (linker)**: Kết nối các đoạn mã và thư viện lại thành một tệp thực thi duy nhất.
* **Trình quản lý thư viện**: Quản lý các thư viện mã dùng chung, cho phép lập trình viên tái sử dụng mã mà không cần viết lại.
* **Công cụ chuyển đổi định dạng tệp thực thi**: Chuyển đổi mã sau khi biên dịch thành định dạng tệp có thể thực thi trên nền tảng mục tiêu.

Trình biên dịch thường cung cấp các tùy chọn để kiểm soát quá trình biên dịch mã, và lập trình viên cần hiểu rõ các tùy chọn này vì chúng có thể ảnh hưởng đến:

* **Khả năng mở rộng**: Mã có thể dễ dàng chuyển đổi và chạy trên các nền tảng và môi trường khác nhau.
* **Tuân thủ tiêu chuẩn ISO C**: Đảm bảo rằng mã nguồn tuân thủ các quy định của tiêu chuẩn ISO C, như C90, C99 hoặc C11.
* **Tài nguyên hệ thống**: Tối ưu hóa hiệu suất chương trình, bao gồm cả tốc độ xử lý và dung lượng bộ nhớ.
* **Rủi ro lỗi từ trình biên dịch**: Đặc biệt khi sử dụng các tùy chọn tối ưu hóa phức tạp, trình biên dịch có thể gây ra lỗi mà lập trình viên khó phát hiện.

Các yếu tố cần hiểu về trình biên dịch:

* **Tính năng "xác định bởi việc triển khai"**: Một số tính năng trong ngôn ngữ C phụ thuộc vào cách triển khai cụ thể của trình biên dịch. Lập trình viên cần nắm rõ những tính năng này vì chúng có thể thay đổi khi chạy trên các trình biên dịch khác nhau hoặc các nền tảng khác nhau.
* **Mở rộng ngôn ngữ**: Một số trình biên dịch có thể hỗ trợ các tính năng mở rộng ngoài tiêu chuẩn ISO. Hiểu về những mở rộng này rất quan trọng nếu bạn đang làm việc trong một môi trường mà chúng được sử dụng.

Ngoài ra, nhà phát triển trình biên dịch thường cung cấp danh sách các lỗi đã biết và giải pháp khắc phục. Lập trình viên cần nắm rõ danh sách này để tránh các vấn đề tiềm ẩn trong quá trình phát triển. Nếu không có danh sách chính thức từ nhà phát triển, việc duy trì danh sách lỗi phát hiện được và báo cáo lại cho nhà phát triển là một thói quen tốt để tránh lặp lại lỗi.

### Hiểu về các công cụ phân tích tĩnh

Công cụ phân tích tĩnh đóng vai trò quan trọng trong việc kiểm tra mã nguồn tuân thủ tiêu chuẩn MISRA C. Để sử dụng hiệu quả các công cụ này, lập trình viên cần hiểu rõ cách chúng hoạt động thông qua việc tham khảo tài liệu hướng dẫn. Một số điểm chính cần xem xét bao gồm:

* **Cấu hình công cụ phù hợp với trình biên dịch**: Một số cài đặt của trình biên dịch, chẳng hạn như kích thước kiểu số nguyên (int, long), có thể ảnh hưởng đến cách mã nguồn được phân tích. Việc hiểu rõ cách cấu hình công cụ phân tích tĩnh cho phù hợp với trình biên dịch đang sử dụng là rất quan trọng.
* **Phạm vi hỗ trợ của công cụ**: Không phải tất cả các công cụ phân tích tĩnh đều hỗ trợ toàn bộ hướng dẫn của MISRA C. Lập trình viên cần biết công cụ có thể kiểm tra những quy tắc nào để tránh bỏ sót vi phạm.
* **Hỗ trợ mở rộng ngôn ngữ và tính năng không chuẩn**: Nếu trình biên dịch của bạn sử dụng các tính năng mở rộng không nằm trong chuẩn ISO, cần kiểm tra xem công cụ phân tích tĩnh có thể xử lý được các tính năng này không.

Hiểu rõ những yếu tố này giúp lập trình viên sử dụng công cụ phân tích tĩnh một cách hiệu quả hơn, từ đó đảm bảo rằng mã nguồn tuân thủ đúng tiêu chuẩn và giảm thiểu lỗi trong quá trình phát triển phần mềm.

## Áp dụng và sử dụng MISRA C

### Áp dụng

MISRA C nên được **áp dụng ngay từ đầu dự án** để đảm bảo mã nguồn tuân thủ tiêu chuẩn an toàn và đáng tin cậy.

Tuy nhiên, nếu dự án đang xây dựng dựa trên **mã nguồn hiện có**, đã được kiểm chứng là ổn định, thì việc áp dụng MISRA C có thể tiềm ẩn rủi ro. Điều này là do trong quá trình điều chỉnh mã để tuân thủ, có thể vô tình tạo ra lỗi mới. Trong những trường hợp như vậy, **quyết định áp dụng MISRA C** nên dựa trên việc cân nhắc kỹ lưỡng về **lợi ích ròng** mang lại, tức là liệu lợi ích của việc tuân thủ có lớn hơn rủi ro không.

### Quy trình phát triển phần mềm

MISRA C được thiết kế để hoạt động tốt nhất trong một quy trình phát triển phần mềm có tài liệu đầy đủ và được tổ chức cẩn thận. Mặc dù có thể sử dụng MISRA C một cách độc lập, nó sẽ mang lại hiệu quả cao hơn khi được tích hợp vào một quy trình phát triển phần mềm chuẩn, bao gồm các bước như:

* **Yêu cầu phần mềm**: Đảm bảo rằng các yêu cầu phần mềm, đặc biệt là yêu cầu về an toàn, được xác định đầy đủ, rõ ràng và chính xác.
* **Đặc tả thiết kế**: Đảm bảo các thiết kế phần mềm chính xác và phù hợp với các yêu cầu, không có các chức năng ngoài yêu cầu.
* **Mã hóa và biên dịch**: Đảm bảo rằng các mô-đun được biên dịch hoạt động chính xác như đã được thiết kế.
* **Kiểm tra mô-đun**: Kiểm tra các mô-đun riêng lẻ và cùng nhau để phát hiện và loại bỏ lỗi.

Việc áp dụng MISRA C ngay từ giai đoạn đầu của quy trình phát triển là rất quan trọng. Nếu kiểm tra tuân thủ MISRA C diễn ra muộn trong quá trình phát triển, dự án có thể phải mất nhiều thời gian để sửa đổi, viết lại và kiểm tra lại mã. Vì vậy, quy trình phát triển phần mềm nên yêu cầu lập trình viên tuân thủ các nguyên tắc của MISRA C ngay từ giai đoạn đầu để tránh rủi ro và tiết kiệm thời gian.

MISRA C thường được áp dụng trong các quy trình phát triển phần mềm liên quan đến an toàn, nhưng yêu cầu chi tiết về các quy trình này nằm ngoài phạm vi của tài liệu MISRA. Tuy nhiên, các tiêu chuẩn và hướng dẫn phát triển phần mềm liên quan đến an toàn như **IEC 61508**, **ISO 26262**, **DO-178C**, **EN 50128**, và **IEC 62304** cung cấp các ví dụ cụ thể về quy trình phát triển.

Phần tiếp theo của tài liệu sẽ bàn về sự **tương tác giữa MISRA C** và quy trình phát triển phần mềm để đảm bảo tuân thủ và tính an toàn.

#### Cách hoạt động quy trình yêu cầu tới MISRA C

Để áp dụng MISRA C một cách hiệu quả, cần phải phát triển và ghi lại những yếu tố quan trọng sau:

1. **Ma trận tuân thủ**: Đây là một tài liệu theo dõi quá trình tuân thủ từng hướng dẫn của MISRA C. Nó chỉ ra cách mỗi quy tắc của MISRA C sẽ được kiểm tra và chứng minh trong mã nguồn. Ma trận này giúp đảm bảo mọi quy tắc đều được đánh giá kỹ lưỡng.
2. **Quy trình xử lý các trường hợp không tuân thủ**: Không phải lúc nào cũng có thể tuân thủ hoàn toàn tất cả các hướng dẫn của MISRA C. Trong trường hợp đó, cần có quy trình biện minh cho việc không tuân thủ, nhận ủy quyền, và ghi lại lý do. Quy trình này phải được quản lý chặt chẽ để đảm bảo sự nhất quán và tính minh bạch.

Ngoài ra, quy trình phát triển phần mềm cần ghi lại các bước để **tránh lỗi thời gian chạy** và chứng minh rằng các lỗi này đã được phòng ngừa. Ví dụ:

* Đảm bảo **môi trường thực thi** cung cấp đủ **tài nguyên** cho chương trình (như thời gian xử lý và dung lượng ngăn xếp).
* Đảm bảo các **lỗi thời gian chạy** như tràn số học không xảy ra. Điều này có thể được thực hiện thông qua mã kiểm tra giới hạn giá trị đầu vào, nhằm đảm bảo các phép tính không gây ra lỗi.

#### Cách hoạt động và quy trình được mong đợi MISRA C

MISRA C nhận ra rằng một **phong cách lập trình nhất quán** sẽ giúp lập trình viên dễ dàng hiểu mã nguồn do người khác viết. Tuy nhiên, vì phong cách lập trình có thể thay đổi theo từng tổ chức, MISRA C không đưa ra khuyến nghị cụ thể về phong cách lập trình. Thay vào đó, các tổ chức nên phát triển **các quy tắc phong cách lập trình nội bộ** phù hợp với quy trình phát triển của mình và áp dụng chúng một cách nhất quán.

Một số **chỉ số phần mềm** thường được các tiêu chuẩn quy trình phát triển phần mềm khuyến nghị sử dụng để xác định các khu vực mã cần xem xét kỹ lưỡng hơn hoặc cần được kiểm tra thêm. Tuy nhiên, tính chất của các chỉ số này và các ngưỡng tương ứng của chúng sẽ được xác định dựa trên ngành công nghiệp, tổ chức và yêu cầu dự án cụ thể. Do đó, tài liệu MISRA C không đưa ra hướng dẫn cụ thể nào về việc thu thập các chỉ số này, mà để các tổ chức tự quyết định dựa trên nhu cầu và đặc điểm của dự án.

### Tuân thủ

Để đảm bảo mã nguồn tuân thủ đầy đủ các hướng dẫn MISRA C, việc tạo ra một ma trận tuân thủ là rất cần thiết. Ma trận này sẽ liệt kê từng hướng dẫn của MISRA C và chỉ ra cách mà mỗi hướng dẫn được kiểm tra và xác minh. Cách kiểm tra hiệu quả nhất thường bao gồm:

* **Sử dụng công cụ phân tích tĩnh**: Đây là phương pháp nhanh chóng, đáng tin cậy và hiệu quả về chi phí để kiểm tra nhiều hướng dẫn MISRA C.
* **Trình biên dịch và các công cụ khác**: Ngoài công cụ phân tích tĩnh, có thể sử dụng trình biên dịch và các công cụ hỗ trợ khác để kiểm tra sự tuân thủ của mã.
* **Kiểm tra thủ công**: Đối với những hướng dẫn mà các công cụ không thể kiểm tra hoàn toàn, việc **xem xét thủ công** là cần thiết. Điều này đòi hỏi sự thận trọng và kỹ lưỡng từ lập trình viên hoặc người kiểm tra.
* **Tuyên bố tuân thủ từ các công cụ tạo mã tự động**

Một số công cụ **tạo mã tự động** có khả năng cung cấp **tuyên bố tuân thủ** kèm theo. Tuyên bố này đảm bảo rằng mã được tạo ra bởi công cụ sẽ không vi phạm các hướng dẫn MISRA C, nếu công cụ được cấu hình đúng cách. Các tuyên bố này có thể:

* **Liệt kê các hướng dẫn** mà mã tạo ra không vi phạm.
* **Đảm bảo** rằng tất cả các hướng dẫn bắt buộc và yêu cầu đều được tuân thủ.

Việc sử dụng các **tuyên bố tuân thủ** này có thể **giảm đáng kể khối lượng công việc** cần thiết để kiểm tra các hướng dẫn MISRA C theo cách thủ công hoặc bằng các phương pháp khác.

* **Tài liệu tham khảo hỗ trợ việc tạo ma trận tuân thủ**

Để hỗ trợ quá trình tạo ma trận tuân thủ đầy đủ, bạn có thể tham khảo **Bảng 1** trong tài liệu để xem ví dụ minh họa. Ngoài ra, **Phụ lục A** cung cấp một bản tóm tắt các hướng dẫn MISRA C, giúp tổ chức và theo dõi quá trình kiểm tra tuân thủ dễ dàng hơn.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Guideline | Compilers | | Checking tools | | Manual review |
| ‘A’ | ‘B’ | ‘A’ | ‘B’ |
| Dir 1.1 |  |  |  |  | Procedure x |
| Dir 2.1 | no errors | no errors |  |  |  |
| … |  |  |  |  |  |
| Rule 4.1 |  |  | message 38 |  |  |
| Rule 4.2 |  |  |  | warning 97 |  |
| Rule 5.1 | warning 347 |  |  |  |  |
| … |  |  |  |  |  |

**Bảng 1**: Giới thiệu về ma trận tuân thủ

**Giải thích:**

|  |
| --- |
| Ma trận này minh họa cách kiểm tra các hướng dẫn MISRA C qua việc sử dụng các **trình biên dịch**, **công cụ kiểm tra** và **xem xét thủ công**. Dưới đây là các phần chính trong ma trận:   * **Cột Guideline**: Liệt kê các hướng dẫn của MISRA C (ví dụ: Dir 1.1, Rule 4.1). * **Cột Compilers**: Chỉ ra thông tin kiểm tra từ hai trình biên dịch được gọi là 'A' và 'B'. Ví dụ:   + Đối với Dir 2.1, cả hai trình biên dịch 'A' và 'B' không báo cáo lỗi.   + Đối với Rule 5.1, trình biên dịch 'A' báo cảnh báo số 347. * **Cột Checking tools**: Chỉ ra thông tin kiểm tra từ hai công cụ phân tích tĩnh 'A' và 'B'. Ví dụ:   + Đối với Rule 4.1, công cụ 'A' báo cáo "message 38".   + Đối với Rule 4.2, công cụ 'B' báo cáo "warning 97". * **Cột Manual review**: Liệt kê các thủ tục hoặc quy trình xem xét thủ công. Ví dụ, đối với Dir 1.1, "Procedure x" được thực hiện để kiểm tra tuân thủ.   Ma trận này giúp tổ chức và theo dõi việc tuân thủ các quy tắc MISRA C bằng cách sử dụng các công cụ tự động và thủ tục kiểm tra thủ công. |

Khi đã có ma trận tuân thủ, trình biên dịch và trình phân tích có thể được cấu hình và sử dụng để tạo danh sách các thông báo cần điều tra.

Các thông tin sau đây cần được ghi lại cho từng công cụ được sử dụng trong quá trình kiểm tra:

* Số phiên bản;
* Các tùy chọn được sử dụng khi gọi công cụ;
* Mọi dữ liệu cấu hình mà công cụ này sử dụng.

#### Cấu hình trình biên dịch

Khi sử dụng trình biên dịch, cấu hình của nó cần phải phù hợp với các yêu cầu của dự án. Đặc biệt:

* **Lựa chọn giữa C90 và C99**: Nếu trình biên dịch hỗ trợ cả hai phiên bản C90 và C99, cần chọn đúng phiên bản mà dự án sử dụng.
* **Cấu hình theo mục tiêu bộ xử lý**: Nếu trình biên dịch cho phép chọn mục tiêu phần cứng cụ thể (ví dụ, loại bộ vi xử lý), phải đảm bảo rằng trình biên dịch được cấu hình chính xác cho phần cứng mục tiêu của dự án.

Khi lựa chọn **tùy chọn tối ưu hóa**, cần xem xét cẩn thận để đạt được sự cân bằng tốt nhất giữa tốc độ thực thi và kích thước mã. Cần lưu ý rằng **tối ưu hóa mạnh mẽ** có thể tăng nguy cơ gặp lỗi từ trình biên dịch, do đó, cần kiểm tra kỹ lưỡng khi sử dụng các tùy chọn này.

Ngay cả khi không dùng trình biên dịch để kiểm tra tuân thủ MISRA C, nó vẫn có thể tạo ra các **cảnh báo** hữu ích trong quá trình biên dịch, cho thấy các lỗi tiềm ẩn. Các thông báo này cần được xem xét, và nếu trình biên dịch cho phép kiểm soát số lượng và loại cảnh báo, cần chọn mức độ phù hợp để xem xét các cảnh báo liên quan.

#### Cấu hình công cụ phân tích tatic

Khác với trình biên dịch, công cụ phân tích tĩnh thường phục vụ nhiều mục đích và không phụ thuộc vào một loại bộ xử lý cụ thể. Vì vậy, cần cấu hình công cụ này sao cho phù hợp với các quyết định triển khai của trình biên dịch. Ví dụ:

* **Kích thước kiểu dữ liệu**: Công cụ phân tích cần biết kích thước các kiểu dữ liệu (như int, long) mà trình biên dịch đang sử dụng.
* **Hỗ trợ C90 hoặc C99**: Nếu công cụ hỗ trợ cả C90 và C99, cần cấu hình để nó phù hợp với phiên bản C mà dự án đang dùng.

Nếu công cụ phân tích tĩnh hỗ trợ **các mở rộng ngôn ngữ** (ví dụ, mở rộng cụ thể cho trình biên dịch), nó nên được cấu hình để hỗ trợ các phần mở rộng này. Nếu không thể cấu hình được, cần có quy trình thay thế để kiểm tra liệu mã nguồn có tuân thủ đúng ngôn ngữ mở rộng hay không.

Ngoài ra, công cụ phân tích tĩnh có thể cần cấu hình để kiểm tra các **nguyên tắc nhất định** của MISRA C. Ví dụ, nếu dự án không sử dụng \_Bool cho dữ liệu Boolean, thì công cụ cần được biết về kiểu dữ liệu Boolean cơ bản mà dự án đang sử dụng để kiểm tra tuân thủ.

#### Điều tra thông báo

Các thông báo được tạo ra trong quá trình kiểm tra tuân thủ hoặc dịch thuật sẽ thuộc một trong ba loại sau:

1. **Chẩn đoán chính xác hành vi vi phạm**: Công cụ phát hiện đúng các vi phạm hướng dẫn của MISRA C.
2. **Chẩn đoán khả năng vi phạm**: Công cụ cho thấy có khả năng vi phạm nguyên tắc MISRA C, nhưng cần xác minh thêm.
3. **Chẩn đoán sai**: Cảnh báo không chính xác về vi phạm hướng dẫn MISRA C (cảnh báo giả).

Mọi thông báo nào không rõ ràng cần được **điều tra** để xác định tính chính xác. Đôi khi, giải pháp đơn giản nhất là **chỉnh sửa mã nguồn** để loại bỏ thông báo, nhưng điều này không phải lúc nào cũng có thể hoặc mong muốn. Trong trường hợp không thể sửa mã, cần **lưu giữ hồ sơ điều tra** để giải thích lý do tại sao cảnh báo không được xử lý hoặc không cần sửa mã.

**Mục đích của việc ghi lại** quá trình điều tra là để đảm bảo:

* Mọi vi phạm đã được xác minh và xem xét kỹ lưỡng.
* Có tài liệu lưu trữ để tham khảo trong các lần kiểm tra hoặc rà soát mã nguồn sau này.

### Thủ tục sai lệch

Trong một số trường hợp, bạn có thể cần **sai lệch** khỏi các hướng dẫn của MISRA C. Ví dụ, việc truy cập vào các **cổng I/O được ánh xạ bộ nhớ** tại các địa chỉ cố định không tuân thủ hoàn toàn MISRA C, vì nó liên quan đến việc chuyển đổi số nguyên thành con trỏ, như trong đoạn mã sau:

|  |
| --- |
| #define PORT (\*(volatile unsigned char \*)0x0002)  PORT = 0x10u; |

Những sai lệch như vậy cần được **ghi lại và phê duyệt chính thức**. Các lập trình viên không nên tự ý thực hiện sai lệch mà phải tuân theo quy trình ủy quyền chính thức. Việc lưu giữ **hồ sơ chính thức** giúp hỗ trợ bất kỳ luận điểm nào liên quan đến **an toàn phần mềm** và đảm bảo tính minh bạch trong quá trình phát triển.

**Quy trình sai lệch** nên được thiết lập rõ ràng ngay từ đầu trong quá trình phát triển phần mềm. MISRA C không yêu cầu một quy trình cụ thể, vì các tổ chức có thể sử dụng các **phương pháp khác nhau** để quản lý sai lệch, chẳng hạn như:

* Cách thức nêu lên, xem xét và phê duyệt sai lệch.
* **Mức độ xem xét** và **bằng chứng** cần thiết, có thể thay đổi tùy thuộc vào mức độ quan trọng của quy tắc bị sai lệch.

MISRA C phân loại các quy tắc theo **mức độ quan trọng**, điều này ảnh hưởng đến cách áp dụng sai lệch (xem **Phần 6.2**).

Sai lệch được chia thành hai loại:

* **Sai lệch dự án**: Áp dụng cho nhiều trường hợp trong dự án. Ví dụ, nếu bạn sử dụng I/O được ánh xạ bộ nhớ trong nhiều tệp trong trình điều khiển, bạn có thể cấp sai lệch dự án cho toàn bộ hệ thống đó.
* **Sai lệch cụ thể**: Áp dụng cho một trường hợp cụ thể trong một tệp mã duy nhất.

### Tuyên bố tuân thủ

**Tuyên bố tuân thủ** chỉ có thể được thực hiện cho **một dự án cụ thể**, không phải cho toàn bộ tổ chức. Khi đưa ra tuyên bố tuân thủ theo MISRA C cho một dự án, bạn đang khẳng định rằng:

* **Ma trận tuân thủ** đã được hoàn thành, mô tả cách kiểm tra tuân thủ cho từng quy tắc MISRA C.
* Tất cả mã C trong dự án **tuân thủ các hướng dẫn** của MISRA C, hoặc có **sai lệch** đã được phê duyệt.
* **Hồ sơ** về tất cả các sai lệch đã được phê duyệt đều được lưu giữ.
* Các hướng dẫn từ **Phần 3** và **Phần 4** của tài liệu này đã được thực hiện.
* Nhân sự có đủ **kỹ năng và kinh nghiệm** để đảm bảo tuân thủ.

**Lưu ý**: Khi bạn tuyên bố rằng mã tuân thủ MISRA C, điều này giả định rằng tất cả các công cụ kiểm tra hoặc người đánh giá đã xác định được tất cả các vi phạm. Tuy nhiên, các công cụ hoặc người đánh giá khác nhau có thể phát hiện ra các vi phạm khác, vì vậy, tuyên bố tuân thủ không phải là tuyệt đối mà phụ thuộc vào **quy trình kiểm tra** được sử dụng.

Một **danh sách kiểm tra** tóm tắt các hướng dẫn được nêu ở mục (4) có thể được tìm thấy trong **Phụ lục F** của tài liệu.

## Giới thiệu về các hướng dẫn

Phần này giải thích cách các hướng dẫn được trình bày trong **Phần 7** và **Phần 8** của tài liệu. Nó cung cấp một cái nhìn tổng quan để giúp hiểu rõ hơn về các hướng dẫn trong những phần tiếp theo.

### Phân loại hướng dẫn

Mỗi hướng dẫn trong MISRA C được phân loại thành hai loại chính:

* **Chỉ dẫn (Directive)**: Đây là các hướng dẫn mà không thể kiểm tra hoàn toàn bằng mã nguồn. Để kiểm tra chỉ dẫn, cần thêm thông tin bổ sung từ tài liệu thiết kế hoặc yêu cầu hệ thống. Các công cụ phân tích tĩnh có thể hỗ trợ việc kiểm tra chỉ dẫn, nhưng do tính chất không đầy đủ của chúng, mỗi công cụ có thể đưa ra các cách giải thích khác nhau về sự tuân thủ hay không tuân thủ.
* **Quy tắc (Rule)**: Quy tắc có thể kiểm tra hoàn toàn bằng cách xem xét mã nguồn. Mô tả của quy tắc đã cung cấp đầy đủ thông tin để kiểm tra tuân thủ mà không cần thêm tài liệu bổ sung. Các công cụ phân tích tĩnh có thể được sử dụng để kiểm tra các quy tắc này, mặc dù có một số hạn chế (xem **Phần 6.5** để biết thêm chi tiết).
* **Phần 7** chứa các chỉ dẫn.
* **Phần 8** chứa các quy tắc.

### Các loại hướng dẫn

Mỗi hướng dẫn trong MISRA C được phân thành một trong ba loại: **bắt buộc**, **cần thiết**, hoặc **khuyến nghị**. Các loại này không phản ánh tầm quan trọng của hướng dẫn; thay vào đó, chúng chỉ ra mức độ tuân thủ cần thiết. Tất cả các loại hướng dẫn nên được coi là quan trọng và cần tuân thủ theo cách thích hợp.

#### Hướng dẫn bắt buộc

Các hướng dẫn **bắt buộc** phải được tuân thủ hoàn toàn trong mã C được tuyên bố là tuân thủ MISRA C. **Không được phép sai lệch** với các hướng dẫn bắt buộc.

* **Lưu ý**: Nếu một công cụ kiểm tra báo cáo một cảnh báo vi phạm, điều đó không nhất thiết có nghĩa là quy tắc đã bị vi phạm, vì có thể có những yếu tố khác được mô tả trong **Phần 6.5**.

#### Hướng dẫn cần thiết

Các hướng dẫn **cần thiết** cũng phải được tuân thủ, tuy nhiên, trong trường hợp không thể tuân thủ hoàn toàn, có thể **sai lệch** thông qua quy trình phê duyệt chính thức (như được mô tả trong **Phần 5.4**).

* Một tổ chức hoặc dự án có thể quyết định coi các hướng dẫn cần thiết như thể chúng là bắt buộc.

#### Hướng dẫn khuyến nghị

Các hướng dẫn **khuyến nghị** không bắt buộc, nhưng vẫn nên được tuân thủ đến **mức độ hợp lý**. Không cần có quy trình lệch chuẩn chính thức cho các hướng dẫn khuyến nghị, nhưng nếu không tuân thủ, nên có phương án ghi lại và theo dõi sự không tuân thủ này.

* Một tổ chức hoặc dự án có thể chọn cách coi các hướng dẫn khuyến nghị như thể chúng là bắt buộc hoặc cần thiết.

### Tổ chức các hướng dẫn

Các hướng dẫn trong MISRA C được sắp xếp theo **chủ đề** khác nhau của ngôn ngữ C. Điều này giúp người đọc dễ dàng tìm thấy các quy tắc và chỉ dẫn liên quan đến một chủ đề cụ thể. Tuy nhiên, đôi khi có **sự chồng chéo** khi một hướng dẫn có thể áp dụng cho nhiều chủ đề. Trong trường hợp này, hướng dẫn sẽ được đặt trong chủ đề phù hợp nhất.

### Sự dư thừa trong các hướng dẫn

Có một số trường hợp trong tài liệu này, **một hướng dẫn** có thể đề cập đến một tính năng mà **bị cấm** hoặc **không khuyến khích** ở một phần khác của tài liệu. Điều này là có chủ ý. Ví dụ, bạn có thể chọn sử dụng tính năng bị cấm bằng cách **sai lệch khỏi hướng dẫn bắt buộc**, hoặc không tuân theo một hướng dẫn khuyến nghị. Khi đó, **hướng dẫn thứ hai** (hạn chế tính năng đó) sẽ trở nên có liên quan.

### Tính khả quyết của các quy tắc

Các quy tắc trong MISRA C được phân thành hai loại: **khả quyết** (decidable) và **bất khả quyết** (undecidable). Phân loại này dựa trên khả năng của công cụ phân tích tĩnh để xác định một cách chắc chắn xem mã có tuân thủ quy tắc hay không.

* **Quy tắc khả quyết**: Là quy tắc mà công cụ phân tích tĩnh có thể trả lời chính xác **"Có" hoặc "Không"** cho câu hỏi: "Mã này có tuân thủ quy tắc này không?". Điều này có nghĩa là với một công cụ phân tích tĩnh hoàn thiện, nếu nó báo cáo vi phạm một quy tắc khả quyết, thì đó là vi phạm thực sự. Nếu không có vi phạm nào được báo cáo, mã tuân thủ quy tắc đó.

Ví dụ về các quy tắc khả quyết:

* + **Quy tắc 5.2**: Phụ thuộc vào tên và phạm vi của các định danh.
  + **Quy tắc 11.3**: Phụ thuộc vào kiểu con trỏ nguồn và con trỏ đích.
  + **Quy tắc 20.7**: Phụ thuộc vào cú pháp của kết quả từ việc mở rộng macro.
* **Quy tắc bất khả quyết**: Là quy tắc mà công cụ phân tích tĩnh **không thể** luôn trả lời chính xác. Việc phát hiện vi phạm có thể phụ thuộc vào các yếu tố **thời gian chạy**, chẳng hạn như giá trị của một đối tượng hoặc liệu điều khiển có đến một điểm cụ thể trong chương trình hay không.

Đối với quy tắc bất khả quyết:

* + **Một vi phạm được báo cáo** không nhất thiết chỉ ra vi phạm thực sự. Một số công cụ có thể đưa ra cảnh báo để nhắc nhở về sự không chắc chắn.
  + **Không có vi phạm được báo cáo** cũng không đảm bảo rằng mã tuân thủ quy tắc đó, vì công cụ có thể không phát hiện được vi phạm.

Ví dụ về các quy tắc bất khả quyết:

* + **Quy tắc 12.2**: Phụ thuộc vào giá trị của toán hạng phải của toán tử dịch chuyển.
  + **Quy tắc 2.1**: Phụ thuộc vào việc biết liệu chương trình có thực sự đến một điểm nhất định hay không.

Như đã nêu trong Mục 5.3.3, một quy trình cần được phát triển để phân tích kết quả của phân tích tĩnh và ghi lại kết quả. Cần chú ý đặc biệt đến quy trình phân tích bất kỳ đầu ra nào liên quan đến các quy tắc bất khả quyết.

### Phân loại quy tắc theo phạm vi kiểm tra

Mỗi quy tắc trong MISRA C được phân loại dựa trên phạm vi mã cần kiểm tra để phát hiện vi phạm. Giống như tính khả quyết, **khái niệm phạm vi phân tích** không áp dụng cho các chỉ thị mà chỉ áp dụng cho các quy tắc. Có hai phạm vi kiểm tra chính:

1. **Đơn vị Biên dịch Độc lập** (Single Translation Unit)
2. **Hệ thống** (System)

**1. Quy tắc "Đơn vị Biên dịch Độc lập"**

Nếu một quy tắc được phân loại là "Đơn vị Biên dịch Độc lập", có thể kiểm tra vi phạm chỉ bằng cách xem xét **từng đơn vị biên dịch** một cách độc lập. Điều này có nghĩa là mỗi đơn vị biên dịch (tức là mỗi tệp mã nguồn) được kiểm tra riêng biệt mà không cần quan tâm đến các đơn vị biên dịch khác.

**Ví dụ**: Quy tắc 16.4 yêu cầu mỗi câu lệnh switch phải có nhãn default. Việc kiểm tra xem một đơn vị biên dịch có vi phạm quy tắc này hay không chỉ phụ thuộc vào mã trong đơn vị đó, và không bị ảnh hưởng bởi các đơn vị biên dịch khác.

**2. Quy tắc "Hệ thống"**

Nếu một quy tắc yêu cầu kiểm tra trên cơ sở **Hệ thống**, thì việc phát hiện vi phạm đòi hỏi phải kiểm tra **toàn bộ mã nguồn** của dự án. Điều này có nghĩa là bạn cần phân tích nhiều hơn một đơn vị biên dịch, và có thể phải kiểm tra toàn bộ hệ thống (bao gồm tất cả các tệp mã) để đảm bảo tuân thủ.

**Ví dụ**: Quy tắc 8.3 yêu cầu rằng tất cả các khai báo và định nghĩa của cùng một đối tượng phải sử dụng cùng loại và định danh. Nếu một dự án có hai đơn vị biên dịch A và B, việc kiểm tra từng đơn vị biên dịch độc lập không đủ để đảm bảo rằng các khai báo trong A và B đều thống nhất. Để kiểm tra quy tắc này, cần phải kiểm tra toàn bộ mã nguồn khi các đơn vị biên dịch được biên dịch và liên kết lại với nhau.

**Quy tắc bất khả quyết và phạm vi "Hệ thống"**

Tất cả các quy tắc **bất khả quyết** (undecidable) đều yêu cầu kiểm tra trên cơ sở **Hệ thống**. Điều này là do những quy tắc này phụ thuộc vào thông tin từ nhiều đơn vị biên dịch hoặc hành vi của chương trình trong thời gian chạy.

**Ví dụ**: Quy tắc 9.1 yêu cầu các biến tự động phải được khởi tạo trước khi sử dụng. Trong đoạn mã dưới đây, giá trị của biến x trong hàm g phụ thuộc vào hành vi của hàm f, vốn được định nghĩa trong một đơn vị biên dịch khác. Vì vậy, không thể chỉ kiểm tra một đơn vị biên dịch để xác định xem quy tắc này có bị vi phạm hay không.

|  |
| --- |
| extern void f (uint16\_t \*p);  uint16\_t y;  void g (void) {      uint16\_t x;  /\* x không được gán giá trị \*/      f(&x);       /\* f có thể thay đổi đối tượng mà con trỏ p trỏ đến \*/      y = x;       /\* x có thể hoặc không được gán giá trị \*/  } |

### Dự án đa tổ chức

Trong các dự án phần mềm lớn, mã nguồn có thể đến từ nhiều tổ chức khác nhau, bao gồm:

* **Mã Thư viện Chuẩn** từ nhà cung cấp trình biên dịch.
* **Mã trình điều khiển cấp thấp** từ nhà cung cấp thiết bị.
* **Mã hệ điều hành và trình điều khiển cấp cao** từ các nhà cung cấp chuyên biệt.
* **Mã ứng dụng** có thể được chia sẻ giữa các tổ chức hợp tác, tùy theo chuyên môn của họ.

#### Mã Thư viện Chuẩn

Mã trong **Thư viện Chuẩn** thường ưu tiên về hiệu suất và có thể sử dụng các kỹ thuật đặc thù như:

* **Ép kiểu con trỏ** từ một loại đối tượng này sang loại đối tượng khác.
* Sử dụng chi tiết cụ thể về **bố trí ngăn xếp**, đặc biệt là các tham số hàm.
* **Nhúng câu lệnh assembly** trong mã C.

Vì thư viện này là một phần của môi trường thực thi và được định nghĩa trong tiêu chuẩn ISO C, **mã Thư viện Chuẩn không bắt buộc phải tuân thủ MISRA C**. Tuy nhiên, **các giao diện** được cung cấp bởi các khai báo và macro trong các tệp tiêu đề của Thư viện Chuẩn vẫn phải tuân thủ các **quy tắc về loại cơ bản** (essential type rules) của MISRA C, đặc biệt khi chúng liên quan đến các tham số truyền vào hàm và kết quả trả về.

#### Mã từ các tổ chức khác

Tất cả mã khác, ngoài Thư viện Chuẩn, nên **tuân thủ MISRA C** tối đa có thể. Trong trường hợp tất cả mã nguồn đều có sẵn, toàn bộ chương trình có thể được kiểm tra tính tuân thủ. Tuy nhiên, trong nhiều dự án đa tổ chức, việc yêu cầu cung cấp **toàn bộ mã nguồn** từ các tổ chức khác là không khả thi. Điều này thường do lý do **bảo vệ tài sản trí tuệ**.

Khi chỉ có một phần mã nguồn có sẵn để phân tích, cần áp dụng các phương pháp khác để **xác minh sự tuân thủ**, chẳng hạn như:

* **Tuyên bố tuân thủ MISRA C**: Các nhà cung cấp mã thương mại có thể phát hành một tuyên bố về sự tuân thủ.
* **Quy trình và công cụ chung**: Các tổ chức hợp tác có thể đồng ý về một quy trình và công cụ chung để kiểm tra mã nguồn của nhau.
* **Mã nguồn mẫu (stub)**: Các tổ chức có thể cung cấp các phiên bản mã mẫu để cho phép kiểm tra chéo giữa các bên.

#### Xử lý mã từ các tổ chức khác

Một dự án nên quyết định cách xử lý mã từ các tổ chức khác. Có ba phương pháp chính:

1. **Xử lý như mã nội bộ**: Nếu mã nguồn có sẵn, nó được kiểm tra tuân thủ MISRA C giống như mã đang phát triển nội bộ.
2. **Xử lý như Thư viện Chuẩn**: Nếu nhà cung cấp cung cấp một tuyên bố tuân thủ MISRA C, mã này được xử lý tương tự như mã Thư viện Chuẩn.
3. **Kiểm tra tệp tiêu đề và giao diện**: Nếu không có tuyên bố tuân thủ và không có mã nguồn ngoài các tệp tiêu đề, chỉ kiểm tra sự tuân thủ của các tệp tiêu đề và giao diện của chúng. Trước khi sử dụng mã, các kỹ thuật **xác minh và xác nhận bổ sung** nên được áp dụng.

**Lưu ý**

Một số tiêu chuẩn quy trình, chẳng hạn như **ISO 26262**, yêu cầu quản lý cẩn thận đối với các dự án phát triển đa tổ chức. Ví dụ, **Phần 8 Khoản 5** của ISO 26262 nêu rõ các yêu cầu về quản lý giao diện trong phát triển phân phối.

### Mã tự động sinh

MISRA C **áp dụng cho cả mã tự động sinh** và mã thủ công. Trách nhiệm đảm bảo tuân thủ MISRA C đối với mã tự động sinh thuộc về cả **nhà phát triển công cụ tạo mã tự động** và **nhà phát triển mô hình** từ đó mã được sinh ra.

Do có nhiều gói phần mềm mô hình khác nhau, và mỗi gói có thể có nhiều công cụ tạo mã tự động, nên không thể phân bổ trách nhiệm tuân thủ **từng quy tắc MISRA C** một cách cụ thể cho từng công cụ hay nhà phát triển. Thay vào đó, người sử dụng các công cụ tạo mã và gói phần mềm mô hình nên áp dụng các hướng dẫn liên quan như:

* **MISRA AC GMG**: Hướng dẫn quản lý mã tự động sinh.
* **MISRA AC SLSF**: Hướng dẫn cho các công cụ mô hình hóa Stateflow và Simulink.
* **MISRA AC TL**: Hướng dẫn về tạo mã tự động từ các công cụ mô hình hóa.
* **Phân loại hướng dẫn cho mã tự động sinh**

Phân loại của một hướng dẫn MISRA C khi áp dụng cho mã tự động sinh **không nhất thiết** giống như khi áp dụng cho mã thủ công. Điều này là do đặc thù của quá trình sinh mã tự động có thể khác với quá trình mã hóa thủ công. Cần lưu ý rằng nếu một gói phần mềm mô hình **cho phép chèn mã thủ công** vào mô hình, thì mã đó **không được xem là mã tự động sinh** khi xác định phân loại của một hướng dẫn.

**Phụ lục E** của tài liệu MISRA C liệt kê những hướng dẫn có phân loại khác nhau khi áp dụng cho mã tự động sinh so với mã thủ công. Phụ lục này cũng nêu rõ các **yêu cầu tài liệu** cho các nhà phát triển công cụ tạo mã tự động.

### Trình bày các hướng dẫn

Các yêu cầu riêng lẻ của tài liệu này được trình bày theo định dạng sau:

Danh mục danh mục [Source ref]

Tham chiếu Nguồn

Phân tích quyết định tính , phạm vi

Áp dụng cho Cxx

Trong đó:

· "Ident" là một mã nhận dạng duy nhất cho hướng dẫn;

· "Requirement text" là bản thân hướng dẫn;

· "Tham chiếu nguồn" chỉ ra nguồn gốc chính của (các) mục hoặc nhóm mục, nếu có. Xem Mục 6.10 để biết giải thích về ý nghĩa của các tham chiếu này và chìa khóa để tìm hiểu tài liệu nguồn;

· "Danh mục" là một trong các danh mục "Bắt buộc", "Yêu cầu" hoặc "Tư vấn", như đã giải thích trong Mục 6.2;

· "Quyết định tính" là một trong hai tính chất "Có thể quyết định" hoặc "Không thể quyết định", như đã giải thích trong Mục 6.5;

· "Phạm vi" là một trong hai phạm vi "Hệ thống" hoặc "Đơn vị Biên dịch Đơn lẻ", như đã giải thích trong Mục 6.6;

· "Cxx" là một hoặc nhiều phiên bản "C90" và "C99", ngăn cách bằng dấu phẩy, chỉ ra những phiên bản của ngôn ngữ C mà hướng dẫn áp dụng.

Lưu ý: vì các chỉ thị không có tính quyết định hoặc phạm vi phân tích, dòng "Phân tích" được bỏ qua đối với các chỉ thị.

Ngoài ra, văn bản hỗ trợ được cung cấp cho từng mục hoặc nhóm các mục liên quan. Văn bản cung cấp, khi cần thiết, một số giải thích về các vấn đề cơ bản được hướng dẫn xử lý và các ví dụ về cách áp dụng hướng dẫn.

Trong văn bản hỗ trợ, có thể có một tiêu đề "Khuếch đại", tiếp theo là văn bản cung cấp mô tả chi tiết hơn về hướng dẫn. Một phần khuếch đại là mang tính chuẩn mực; nếu nó mâu thuẫn với tiêu đề chính, phần khuếch đại sẽ được ưu tiên. Cơ chế này thuận tiện vì nó cho phép một khái niệm phức tạp được truyền tải bằng một tiêu đề ngắn gọn.

Trong văn bản hỗ trợ, có thể có một tiêu đề "Ngoại lệ", tiếp theo là văn bản mô tả các tình huống trong đó quy tắc không áp dụng. Việc sử dụng ngoại lệ cho phép đơn giản hóa mô tả của một số hướng dẫn. Điều quan trọng cần lưu ý là ngoại lệ là một tình huống trong đó quy tắc không áp dụng. Mã tuân thủ một hướng dẫn nhờ vào một ngoại lệ không yêu cầu phải có sự chệch hướng

Văn bản hỗ trợ không nhằm mục đích làm tài liệu hướng dẫn về tính năng ngôn ngữ liên quan, vì người đọc được giả định đã có kiến thức thực tế về ngôn ngữ. Thông tin thêm về các tính năng ngôn ngữ có thể được thu thập bằng cách tham khảo phần liên quan của tiêu chuẩn ngôn ngữ hoặc các sách tham khảo ngôn ngữ C khác. Khi một tham chiếu nguồn được đưa ra cho một hoặc nhiều "Vấn đề về Khả năng Di chuyển" được liệt kê trong Tiêu chuẩn, thì vấn đề gốc được nêu ra trong đó có thể cung cấp thêm sự giúp đỡ trong việc hiểu hướng dẫn.

Trong các hướng dẫn và văn bản hỗ trợ, các kiểu phông chữ sau được sử dụng để đại diện cho từ khóa C và mã C:

Các từ khóa C xuất hiện dưới dạng văn bản in nghiêng;

Các mục được định nghĩa trong Thuật ngữ xuất hiện dưới dạng văn bản in nghiêng;

Mã C xuất hiện dưới dạng phông chữ đơn cách, hoặc trong văn bản khác;

Hoặc dưới dạng các đoạn mã riêng biệt.

Lưu ý: nơi mã được trích dẫn, các đoạn có thể không đầy đủ (ví dụ một câu lệnh if mà không có thân của nó). Điều này nhằm mục đích ngắn gọn.

Trong các đoạn mã, các kiểu đã được typedef như sau đã được giả định (điều này để tuân thủ Dir 4.6):

uint8\_t /\* unsigned 8-bit integer \*/

uint16\_t    /\* unsigned 16-bit integer \*/

uint32\_t    /\* unsigned 32-bit integer \*/

int8\_t  /\* signed 8-bit integer \*/

int16\_t /\* signed 16-bit integer    \*/

int32\_t /\* signed 32-bit integer \*/

float32\_t  /\* 32-bit floating-point \*/

float64\_t    /\* 64-bit floating-point  \*/

Tên của các đối tượng không cụ thể được cấu trúc để gợi ý về loại của chúng. Ví dụ::

#uint8\_t u8a; /\* 8-bit unsigned integer \*/

int16\_t s16b; /\* 16-bit signed integer \*/

int32\_t s32c; /\* 32-bit signed integer \*/

bool\_t  bla;  /\* essentially Boolean  \*/

enum atag ena;  /\* enumerated type   \*/

char  chb;  /\* character    \*/

float32\_t f32a; /\* 32-bit floating-point \*/

float64\_t f64b;  /\* 64-bit floating-point  \*/

### Hiểu tài liệu tham khảo

Khi một hướng dẫn bắt nguồn từ một hoặc nhiều nguồn đã xuất bản, các nguồn này được chỉ định trong dấu ngoặc vuông sau hướng dẫn. Điều này có hai mục đích. Thứ nhất, các nguồn cụ thể có thể được người đọc tham khảo để hiểu rõ hơn về lý do đằng sau hướng dẫn (ví dụ, khi xem xét một yêu cầu chệch hướng). Thứ hai, đối với các Vấn đề về Khả năng Di chuyển được mô tả trong tiêu chuẩn ISO C, hình thức của nguồn cung cấp thêm thông tin về bản chất của vấn đề.

Những quy tắc không có tham chiếu nguồn có thể bắt nguồn từ tiêu chuẩn nội bộ của một công ty đóng góp, hoặc đã được đề xuất bởi một người đánh giá, hoặc là thực hành tốt được chấp nhận rộng rãi.

### Tham chiếu vấn đề về tính di động của ISO C

Các vấn đề về tính di động của ISO C được mô tả trong các tiểu mục của Phụ lục trong tiêu chuẩn liên quan. Cần lưu ý rằng đối với cả C90 và C99, số tham chiếu được lấy từ tiêu chuẩn gốc và không phải từ bất kỳ phiên bản nào sau này có sửa đổi và bổ sung. Quyết định này một phần là do lý do lịch sử, vì MISRA C:2004 đã sử dụng hệ thống này, và một phần do thực tế là không có phiên bản chính thức nào của Tiêu chuẩn C99 được phát hành có kết hợp các Sửa đổi Kỹ thuật (Technical Corrigenda), mặc dù có tồn tại một bản nháp

Các tiểu mục của tiêu chuẩn liên quan tương ứng với loại tham chiếu này là:

|  |  |  |
| --- | --- | --- |
| Tham chiếu | Phụ lục G của ISO 9899:1990 | Phụ lục J của ISO 9899:1999 |
| Không xác định | G.1 | J.1 |
| Không được định nghĩa | G.2 | J.2 |
| Thực hiện | G.3 | J.3 |
| Locale | G.4 | J.4 |

Khi có văn bản đi kèm tham chiếu, nó có nghĩa như sau:

· Tham chiếu từ Phụ lục G của [2]: Văn bản xác định số của một hoặc nhiều mục trong phần liên quan của Phụ lục, được đánh số từ đầu phần đó. Ví dụ, [Locale 2] đề cập đến mục thứ hai trong phần G.4 của Tiêu chuẩn và [Undefined 3, 5] đề cập đến mục thứ ba và thứ năm trong Phần G.2 của Tiêu chuẩn.

· Tham chiếu từ Phụ lục J của [8]: Đối với các phần J.1, J.2 và J.4, cách giải thích giống như trên. Đối với phần J.3, văn bản xác định tiểu mục của J.3, tiếp theo là số của một hoặc nhiều mục trong dấu ngoặc. Ví dụ, [Unspecified 6] đề cập đến mục thứ sáu trong Phần J.1 của Tiêu chuẩn và [Implementation J3.4(2, 5)] đề cập đến mục thứ hai và thứ năm trong Phần J.3.4 của Tiêu chuẩn.

Khi một hướng dẫn dựa trên các vấn đề từ Phụ lục G (C90) hoặc Phụ lục J (C99) của Tiêu chuẩn, việc hiểu sự khác biệt giữa các tham chiếu Không xác định, Không được định nghĩa, Thực hiện và Locale sẽ hữu ích cho người đọc. Những điều này được giải thích ngắn gọn dưới đây, và thông tin chi tiết hơn có thể được tìm thấy trong Hatton [3]..

· Không xác định: Đây là các cấu trúc ngôn ngữ phải được biên dịch thành công, nhưng trong đó nhà viết trình biên dịch có một số tự do về việc cấu trúc sẽ hoạt động như thế nào. Ví dụ cho điều này là "thứ tự đánh giá" được mô tả trong Quy tắc 13.2. Mặc dù việc sử dụng một số hành vi không xác định là không thể tránh khỏi, nhưng không nên giả định rằng trình biên dịch tạo ra mã đối tượng hoạt động theo một cách cụ thể; trình biên dịch thậm chí không nhất thiết phải hoạt động nhất quán trên tất cả các cấu trúc có thể.

· Không được định nghĩa: Đây về cơ bản là các lỗi lập trình, nhưng nhà viết trình biên dịch không bắt buộc phải cung cấp thông báo lỗi. Ví dụ là các tham số không hợp lệ cho các hàm hoặc các hàm có đối số không khớp với các tham số được định nghĩa. Điều này đặc biệt quan trọng từ góc độ an toàn, vì chúng đại diện cho các lỗi lập trình có thể không nhất thiết phải được bắt bởi trình biên dịch

Implementation

Các vấn đề thuộc nhóm này tương tự như các vấn đề "không xác định", với sự khác biệt chính là nhà viết trình biên dịch phải áp dụng một cách tiếp cận nhất quán và ghi lại nó. Nói cách khác, chức năng có thể khác nhau giữa các trình biên dịch, làm cho mã không thể di động được, nhưng trên bất kỳ một trình biên dịch nào thì hành vi đó nên được định nghĩa rõ ràng. Ví dụ về điều này là hành vi của các toán tử chia nguyên và lấy phần dư, / và %, khi áp dụng cho một số dương và một số âm..

Các hành vi được định nghĩa bởi người thực hiện có xu hướng ít quan trọng hơn từ quan điểm an toàn, với điều kiện là nhà viết trình biên dịch đã ghi chép đầy đủ cách tiếp cận dự định và sau đó thực hiện nó một cách nhất quán. Tuy nhiên, nên tránh các vấn đề này nếu có thể..

Locale

Các hành vi liên quan đến locale (địa phương hóa) tạo thành một tập hợp nhỏ các tính năng có thể thay đổi theo yêu cầu quốc tế. Ví dụ về điều này là khả năng đại diện cho dấu thập phân bằng ký tự “,” thay vì ký tự “.”. Không có vấn đề nào phát sinh từ nguồn này được đề cập trong tài liệu này..

### Các tham chiếu khác

Các tham chiếu khác ngoài các vấn đề về tính di động của ISO C được lấy từ các nguồn sau:

|  |  |
| --- | --- |
| Tham chiếu | nguồn |
| MISRA Guidelines | Hướng dẫn của MISRA [[15]](#_bookmark147) |
| Koenig | “C Traps and Pitfalls”, Koenig [[31]](#_bookmark148) |
| IEC 61508 | IEC 61508:2010 [[32]](#_bookmark148) |
| ISO 26262 | ISO 26262:2011 [[23]](#_bookmark148) |
| DO-178C | DO-178C [[24]](#_bookmark148) |

Trừ khi được nêu rõ khác đi, văn bản sau một tham chiếu sẽ cung cấp số trang liên quan.

## Chỉ thị (Directives)

### Về việc triển khai (The implementation)

**Dir 1.1** Bất kỳ hành vi nào phụ thuộc vào việc triển khai (implementation-defined behaviour) mà kết quả của chương trình phụ thuộc vào, phải được **tài liệu hóa** và **hiểu rõ**.  
(Các tiêu chuẩn: C90 [Phụ lục G.3], C99 [Phụ lục J.3])

**Loại:** Bắt buộc (Required)  
**Áp dụng cho:** C90, C99

##### Giải thích thêm:

Phụ lục G của tài liệu này liệt kê các hành vi phụ thuộc vào việc triển khai (đối với cả C90 và C99) mà:

* Được coi là có khả năng gây ra những hoạt động không mong muốn của chương trình.
* Có thể xuất hiện trong chương trình ngay cả khi nó tuân thủ tất cả các hướng dẫn khác của MISRA C.

Tất cả các hành vi phụ thuộc vào việc triển khai mà kết quả chương trình phụ thuộc vào phải:

* Được tài liệu hóa.
* Được hiểu rõ bởi các nhà phát triển.

**Lưu ý:** Một triển khai tuân thủ (conforming implementation) cần phải tài liệu hóa cách nó xử lý tất cả các hành vi phụ thuộc vào việc triển khai. Nếu tài liệu còn thiếu, nhà phát triển của triển khai đó cần được tham khảo.

##### Lý do (Rationale):

Việc hiểu rõ kết quả của chương trình là quan trọng để biết rằng kết quả đó là **có chủ đích** chứ không phải do ngẫu nhiên. Một số hành vi phụ thuộc vào việc triển khai phổ biến mà phần mềm nhúng liên quan đến an toàn thường dựa vào được mô tả dưới đây.

##### Hành vi cốt lõi (Core behaviour):

Các hành vi phụ thuộc vào việc triển khai cơ bản, mà phần lớn các chương trình đều yêu cầu, bao gồm:

* Cách xác định thông báo chẩn đoán được tạo ra trong quá trình biên dịch.
* Kiểu dữ liệu của hàm main, thường được khai báo là void main(void) trong các triển khai độc lập.
* Số lượng ký tự quan trọng trong các định danh — cần thiết để cấu hình công cụ phân tích theo Quy tắc 5.2.
* Các tập ký tự nguồn và thực thi.
* Kích thước của các loại số nguyên.
* Cách một tên tệp được bao gồm (thông qua #include) được ánh xạ thành tên tệp và được tìm thấy trong hệ thống tệp của máy chủ.

### Tiện ích mở rộng (Extensions)

Các tiện ích mở rộng thường được sử dụng trong các hệ thống nhúng để truy cập các thiết bị ngoại vi và đặt các đối tượng vào các vùng nhớ có thuộc tính đặc biệt, chẳng hạn như Flash EEPROM hoặc RAM truy cập nhanh. Một triển khai tuân thủ có thể cung cấp các tiện ích mở rộng với điều kiện chúng không thay đổi ý nghĩa của bất kỳ chương trình nào tuân thủ nghiêm ngặt.

Một số triển khai C90 có thể cung cấp các tiện ích mở rộng, trong đó có một phần tính năng của C99. Các cách thức mà một triển khai có thể cung cấp tiện ích mở rộng bao gồm:

* Chỉ thị tiền xử lý #pragma hoặc toán tử \_Pragma (chỉ có trong C99).
* Các từ khóa mới.

### Thư viện chuẩn (The Standard Library)

Một số khía cạnh của triển khai Thư viện Chuẩn có thể quan trọng bao gồm:

* Các giá trị được gán cho biến errno khi sử dụng một số hàm của Thư viện Chuẩn.
* Việc triển khai các hàm clock và time.
* Đặc điểm của hệ thống tệp.

### Giao diện Nhị phân Ứng dụng (The Application Binary Interface - ABI)

Trong một số trường hợp, cần phải kết nối mã C với ngôn ngữ lắp ráp (assembly), ví dụ để tăng tốc độ thực thi ở những phần quan trọng. Cũng có thể cần phải kết nối mã từ các trình biên dịch khác nhau, có thể cho các ngôn ngữ khác nhau.

Giao diện Nhị phân Ứng dụng (ABI) của một trình biên dịch cung cấp thông tin cần thiết để thực hiện nhiệm vụ này, bao gồm một số hành vi phụ thuộc vào triển khai. ABI thường xác định:

* Cách tham số hàm được truyền trong các thanh ghi và trên ngăn xếp.
* Cách giá trị trả về của hàm được xử lý.
* Những thanh ghi nào cần được bảo toàn trong quá trình thực thi hàm.
* Cách các đối tượng có thời lượng lưu trữ tự động được cấp phát trên ngăn xếp.
* Yêu cầu căn chỉnh cho mỗi kiểu dữ liệu.
* Cách bố trí cấu trúc dữ liệu và cách các trường bit được phân bổ vào các đơn vị lưu trữ.

Một số bộ xử lý có ABI chuẩn, sẽ được sử dụng bởi tất cả các triển khai. Khi không có ABI chuẩn, mỗi triển khai sẽ cung cấp ABI riêng.

### Chia số nguyên (Integer division)

Trong C90, phép chia số nguyên có dấu hoặc phép toán lấy phần dư mà trong đó một trong các toán hạng có giá trị âm có thể làm tròn xuống hoặc về 0. Trong C99, việc làm tròn được đảm bảo luôn về 0.

### Triển khai số dấu phẩy động (Floating-point implementation)

Việc triển khai các kiểu số dấu phẩy động có thể có ảnh hưởng đáng kể đến hành vi của chương trình, chẳng hạn như:

* Phạm vi giá trị và độ chính xác khi chúng được lưu trữ.
* Hướng làm tròn sau các phép toán số dấu phẩy động.

### 7.1 Làm tròn và các hành vi của kiểu số dấu phẩy động

Khi làm tròn các kiểu số dấu phẩy động hẹp hơn hoặc chuyển đổi sang kiểu số nguyên, các yếu tố sau cần xem xét:

* **Hướng làm tròn** khi chuyển sang các kiểu số dấu phẩy động hẹp hơn hoặc kiểu số nguyên.
* **Hành vi khi có lỗi**: hiện tượng **underflow** (tràn số xuống), **overflow** (tràn số lên) và **NaNs** (giá trị không phải là số).
* **Hành vi của các hàm thư viện** trong trường hợp lỗi về miền giá trị hoặc phạm vi.

**Tham khảo thêm**: Quy tắc 5.1, Quy tắc 5.2.

### 7.2 Biên dịch và xây dựng chương trình (Compilation and Build)

**Dir 2.1**: Tất cả các tệp nguồn phải được biên dịch mà không có lỗi biên dịch.  
**Loại:** Bắt buộc (Required)  
**Áp dụng cho:** C90, C99

**Lý do**: Trình biên dịch có thể tạo ra mô-đun đối tượng (object module) mặc dù có lỗi biên dịch, nhưng chương trình kết quả có thể hoạt động không như mong đợi.

**Tham khảo thêm**: Quy tắc 1.1.

### 7.3 Khả năng truy vết yêu cầu (Requirements Traceability)

**Dir 3.1**: Tất cả mã nguồn phải có khả năng truy vết đến các yêu cầu đã được tài liệu hóa.  
**Loại:** Bắt buộc (Required)  
**Áp dụng cho:** C90, C99

**Lý do**: Các tính năng không cần thiết để đáp ứng yêu cầu của dự án sẽ tạo ra các đường đi không cần thiết. Ví dụ, nhà phát triển có thể thêm mã để bật/tắt một chân xuất của vi xử lý mỗi khi chương trình đạt đến một điểm nhất định. Mặc dù chân này không được đề cập trong đặc tả yêu cầu phần mềm, nhưng nó có thể được kết nối với một thiết bị ngoại vi, gây ra các hiệu ứng không mong muốn.

**Phương pháp**: Mã nguồn cần được kiểm tra đối chiếu với các tài liệu thiết kế và yêu cầu.

### 7.4 Thiết kế mã nguồn (Code Design)

**Dir 4.1**: Cần giảm thiểu tối đa các lỗi thời gian chạy.  
**Loại:** Bắt buộc (Required)  
**Áp dụng cho:** C90, C99

**Lý do**: Ngôn ngữ C cung cấp rất ít kiểm tra thời gian chạy tích hợp. Điều này giúp tạo mã thực thi nhỏ và nhanh, nhưng đặt gánh nặng kiểm tra thời gian chạy lên lập trình viên. Để đảm bảo độ tin cậy, lập trình viên cần xem xét kỹ việc thêm các kiểm tra động, đặc biệt ở những nơi có thể xảy ra lỗi thời gian chạy.

Nếu có thể chứng minh rằng các giá trị của toán hạng không gây ra lỗi thời gian chạy, thì không cần kiểm tra động, miễn là có tài liệu hóa giải thích và các giả định rõ ràng.

### Các loại lỗi cần xem xét kiểm tra động:

* **Lỗi toán học**: Bao gồm các lỗi trong biểu thức như **overflow**, **underflow**, chia cho 0 hoặc mất bit quan trọng khi dịch chuyển. Trong phép tính với số nguyên không dấu, các giá trị có thể "quay vòng" thay vì tràn số, gây ra các giá trị không mong muốn.

Ví dụ:

c

Sao chép mã

float32\_t f1 = 1E38f;

float32\_t f2 = 10.0f;

float32\_t f3 = 0.1f;

float32\_t f4 = (f1 \* f2) \* f3; /\* (f1 \* f2) sẽ tràn số \*/

float32\_t f5 = f1 \* (f2 \* f3); /\* không tràn số vì (f2 \* f3) xấp xỉ 1 \*/

* **Toán tử con trỏ**: Khi tính toán địa chỉ bằng con trỏ, cần đảm bảo địa chỉ hợp lệ và trỏ đến một vùng nhớ hợp lý.
* **Lỗi giới hạn mảng**: Đảm bảo chỉ số của mảng nằm trong giới hạn kích thước mảng.
* **Tham số hàm**: Kiểm tra tính hợp lệ của các đối số trước khi truyền vào các hàm thư viện.
* **Truy cập con trỏ**: Nếu không chắc con trỏ có giá trị khác NULL, cần kiểm tra trước khi truy cập giá trị mà nó trỏ đến.

Ví dụ về kiểm tra con trỏ:

c

Sao chép mã

const char \*msg\_body(const char \*msg) {

const char \*body = NULL;

if (msg != NULL) {

if (msg\_header\_valid(msg)) {

body = &msg[MSG\_HEADER\_SIZE];

}

}

return body;

}

* **Bộ nhớ động**: Nếu sử dụng bộ nhớ động, cần kiểm tra xem việc cấp phát có thành công hay không, và có chiến lược phục hồi phù hợp.

**Tham khảo thêm**: Dir 4.11, Dir 4.12, Quy tắc 1.3, Quy tắc 18.1, 18.2, 18.3.

### ****Dir 4.2****: Mọi sử dụng ngôn ngữ assembly phải được tài liệu hóa

**Loại**: Khuyến nghị (Advisory)  
**Áp dụng cho**: C90, C99

**Giải thích**: Lý do sử dụng mã assembly và cách thức giao tiếp giữa mã C và mã assembly cần được tài liệu hóa rõ ràng.

**Lý do**: Mã assembly là do từng hệ thống định nghĩa, do đó không có tính di động.

### ****Dir 4.3****: Mã assembly phải được đóng gói và cô lập

**Loại**: Bắt buộc (Required)  
**Áp dụng cho**: C90, C99

**Giải thích**: Khi sử dụng các lệnh assembly, chúng cần được đóng gói và cô lập trong:

* Các hàm assembly;
* Các hàm C (ưu tiên dùng hàm nội tuyến cho C99);
* Các macro C.

**Lý do**: Việc đóng gói mã assembly mang lại nhiều lợi ích như:

* **Cải thiện khả năng đọc**;
* Tên và tài liệu của macro hoặc hàm chứa mã assembly sẽ làm rõ ý định sử dụng của mã đó;
* Tất cả các lần sử dụng mã assembly cho một mục đích cụ thể có thể dùng chung việc đóng gói, cải thiện khả năng bảo trì;
* Mã assembly có thể dễ dàng thay thế cho các nền tảng khác hoặc phục vụ phân tích tĩnh.

**Lưu ý**: Việc sử dụng mã assembly nội tuyến là phần mở rộng của ngôn ngữ C, do đó vi phạm Quy tắc 1.2.

**Ví dụ**:

c

Sao chép mã

#define NOP asm("NOP")

### ****Dir 4.4****: Các đoạn mã không nên bị "comment" ra

**Loại**: Khuyến nghị (Advisory)  
**Áp dụng cho**: C90, C99

**Giải thích**: Quy tắc này áp dụng cho cả hai kiểu nhận xét // và /\* ... \*/.

**Lý do**: Khi cần loại trừ một phần mã nguồn không được biên dịch, nên sử dụng biên dịch có điều kiện (ví dụ: #if hoặc #ifdef với một nhận xét giải thích). Sử dụng dấu nhận xét để loại trừ mã là nguy hiểm vì C không hỗ trợ nhận xét lồng nhau, và các nhận xét có sẵn trong đoạn mã sẽ có thể gây ra lỗi.

**Tham khảo thêm**: Quy tắc 3.1, 3.2.

### ****Dir 4.5****: Các định danh (identifiers) trong cùng một không gian tên có tầm nhìn chồng lấn phải dễ phân biệt bằng ký tự

**Loại**: Khuyến nghị (Advisory)  
**Áp dụng cho**: C90, C99

**Giải thích**: Định nghĩa về "dễ phân biệt" nên được xác định theo dự án, xem xét bảng chữ cái và ngôn ngữ sử dụng trong mã nguồn.  
Với bảng chữ cái Latinh trong tiếng Anh, khuyến nghị không sử dụng định danh khác nhau chỉ bằng sự khác biệt giữa các yếu tố sau:

* Thay thế chữ cái viết thường bằng chữ viết hoa tương ứng;
* Sự xuất hiện hoặc không của ký tự gạch dưới \_;
* Thay đổi giữa chữ "O" và số "0";
* Thay đổi giữa chữ "I" và số "1";
* Thay đổi giữa chữ "I" và chữ "l" (el);
* Thay đổi giữa chữ "l" (el) và số "1";
* Thay đổi giữa chữ "S" và số "5";
* Thay đổi giữa chữ "Z" và số "2";
* Thay đổi giữa chữ "n" và chữ "h";
* Thay đổi giữa chữ "B" và số "8";
* Thay đổi giữa chuỗi "rn" (chữ "r" và "n") và chữ "m".

**Lý do**: Tùy thuộc vào phông chữ sử dụng, các ký tự có thể trông giống nhau mặc dù khác nhau. Điều này có thể khiến nhà phát triển nhầm lẫn định danh này với định danh khác.

**Ví dụ**:  
Giả sử quy tắc cho bảng chữ cái Latinh trong tiếng Anh:

c

Sao chép mã

int32\_t id1\_a\_b\_c;

int32\_t id1\_abc; // Không tuân thủ

int32\_t id2\_abc;

int32\_t id2\_ABC; // Không tuân thủ

int32\_t id3\_a\_bc;

int32\_t id3\_ab\_c; // Không tuân thủ

int32\_t id4\_I;

int32\_t id4\_1; // Không tuân thủ

int32\_t id5\_Z;

int32\_t id5\_2; // Không tuân thủ

int32\_t id6\_O;

int32\_t id6\_0; // Không tuân thủ

int32\_t id7\_B;

int32\_t id7\_8; // Không tuân thủ

int32\_t id8\_rn;

int32\_t id8\_m; // Không tuân thủ

int32\_t id9\_rn;

struct {

int32\_t id9\_m; // Tuân thủ

};

### ****Dir 4.6****: Nên sử dụng typedefs để chỉ ra kích thước và dấu của kiểu số thay vì dùng kiểu số cơ bản

**Loại**: Khuyến nghị (Advisory)  
**Áp dụng cho**: C90, C99

**Giải thích**: Không nên sử dụng các kiểu số cơ bản như char, short, int, long, float, double. Thay vào đó, nên sử dụng typedefs chỉ rõ kích thước.

Đối với C99, nên sử dụng các kiểu trong <stdint.h>. Đối với C90, nên định nghĩa các kiểu tương đương và sử dụng.

**Lưu ý**: Không cần phải sử dụng typedefs trong khai báo các trường bit (bit-fields).

### ****Dir 4.6****: Sử dụng typedef chỉ ra kích thước và dấu của kiểu số thay vì dùng kiểu số cơ bản

**Loại**: Khuyến nghị (Advisory)  
**Áp dụng cho**: C90, C99

**Giải thích**: Các kiểu số cơ bản như char, int, long nên được thay thế bằng các typedef chỉ rõ kích thước như int32\_t, uint16\_t để đảm bảo tính rõ ràng về kích thước và dấu của kiểu dữ liệu.

**Lý do**: Sử dụng các kiểu số cụ thể giúp đảm bảo tính nhất quán về kích thước bộ nhớ được phân bổ và giúp dễ dàng kiểm soát tài nguyên. Tuy nhiên, việc sử dụng các kiểu này không đảm bảo tính di động hoàn toàn, vì việc xử lý có thể phụ thuộc vào kích thước của kiểu int trong hệ thống (có thể gây ra lỗi thăng cấp số nguyên - integer promotion).

**Ví dụ**:

c

Sao chép mã

/\* Không tuân thủ - sử dụng kiểu int cơ bản \*/

int x = 0;

/\* Tuân thủ - định nghĩa kiểu dữ liệu cụ thể \*/

typedef int SINT\_16;

/\* Không tuân thủ - kiểu dữ liệu không chỉ rõ kích thước \*/

typedef int speed\_t;

/\* Tuân thủ - định nghĩa kiểu trừu tượng nhưng không chỉ rõ kích thước \*/

typedef int16\_t torque\_t;

### ****Dir 4.7****: Nếu một hàm trả về thông tin lỗi, thì thông tin lỗi đó phải được kiểm tra

**Loại**: Bắt buộc (Required)  
**Áp dụng cho**: C90, C99

**Giải thích**: Danh sách các hàm trả về thông tin lỗi phải được xác định bởi dự án. Thông tin lỗi trả về từ hàm phải được kiểm tra một cách có ý nghĩa.

**Lý do**: Khi một hàm cung cấp cơ chế báo lỗi (dưới dạng cờ lỗi hoặc giá trị trả về đặc biệt), chương trình gọi phải kiểm tra lỗi ngay sau khi hàm hoàn thành. Việc kiểm tra giá trị đầu vào trước khi gọi hàm được xem là cách tốt hơn để ngăn ngừa lỗi, thay vì kiểm tra lỗi sau khi hàm đã hoàn thành.

**Ngoại lệ**: Nếu có thể chứng minh rằng một hàm không bao giờ trả về lỗi, không cần phải kiểm tra.

### ****Dir 4.8****: Nếu con trỏ tới cấu trúc hoặc liên hiệp không bao giờ bị giải tham chiếu trong một đơn vị dịch, thì việc triển khai đối tượng đó nên bị ẩn

**Loại**: Khuyến nghị (Advisory)  
**Áp dụng cho**: C90, C99

**Giải thích**: Việc triển khai đối tượng nên được ẩn thông qua một con trỏ đến kiểu không hoàn chỉnh.

**Lý do**: Nếu con trỏ tới một cấu trúc hoặc liên hiệp không bao giờ bị giải tham chiếu, các chi tiết triển khai của đối tượng đó không cần thiết và cần được bảo vệ khỏi những thay đổi không mong muốn. Việc ẩn triển khai giúp tạo ra một kiểu "mờ" (opaque type), mà nội dung của nó không thể bị truy cập trực tiếp.

**Ví dụ**:

c

Sao chép mã

/\* Opaque.h \*/

#ifndef OPAQUE\_H

#define OPAQUE\_H

typedef struct OpaqueType \*pOpaqueType;

#endif

/\* Opaque.c \*/

#include "Opaque.h"

struct OpaqueType

{

/\* Triển khai đối tượng \*/

};

/\* UseOpaque.c \*/

#include "Opaque.h"

void f ( void )

{

pOpaqueType pObject;

pObject = GetObject(); /\* Lấy đối tượng OpaqueType \*/

UseObject(pObject); /\* Sử dụng đối tượng \*/

}

### ****Dir 4.9****: Một hàm nên được sử dụng thay vì macro giống-hàm khi có thể thay thế được

**Loại**: Khuyến nghị (Advisory)  
**Áp dụng cho**: C90, C99

**Giải thích**: Chỉ thị này áp dụng khi cú pháp và ràng buộc của ngôn ngữ cho phép sử dụng hàm thay vì macro.

**Lý do**: Hàm kiểm tra kiểu của tham số và chỉ đánh giá tham số một lần, tránh các vấn đề về tác dụng phụ xảy ra nhiều lần. Hàm cũng dễ bảo trì và gỡ lỗi hơn macro. Tuy nhiên, macro có thể hữu ích trong một số trường hợp cụ thể như đánh giá hằng tại thời gian biên dịch.

**Ví dụ**:

c

Sao chép mã

/\* Macro giống hàm không thể thay thế bằng hàm vì có chứa toán tử \*/

#define EVAL\_BINOP(OP, L, R) ((L) OP (R))

uint32\_t x = EVAL\_BINOP(+, 1, 2);

/\* Macro hợp lệ trong khởi tạo đối tượng với thời lượng lưu trữ tĩnh \*/

#define DIV2(X) ((X) / 2)

void f (void) {

static uint16\_t x = DIV2(10); /\* Tuân thủ - không thể thay thế bằng hàm \*/

uint16\_t y = DIV2(10); /\* Không tuân thủ - có thể thay thế bằng hàm \*/

}

### ****Dir 4.10****: Phải có biện pháp để ngăn chặn một tệp tiêu đề bị bao gồm nhiều lần

**Loại**: Bắt buộc (Required)  
**Áp dụng cho**: C90, C99

**Giải thích**: Khi một đơn vị dịch chứa một cấu trúc phức tạp của các tệp tiêu đề lồng nhau, có thể xảy ra việc một tệp tiêu đề bị bao gồm nhiều lần, gây ra sự nhầm lẫn hoặc xung đột.

**Ví dụ**:

c

Sao chép mã

/\* file.h \*/

#ifndef FILE\_H

#define FILE\_H

/\* Nội dung của tệp \*/

#endif

**Lưu ý**: Mỗi dự án nên có một định danh duy nhất cho mỗi tệp tiêu đề để tránh bị bao gồm nhiều lần.

### Dir 4.11 Giá trị của các tham số truyền vào hàm thư viện phải được kiểm tra

**C90 [Undefined 60, 63, 96; Implementation 45–47]  
C99 [Unspecified 30, 31, 44, 48–50; Undefined 102, 103, 107, 112, 180, 181, 183, 187, 189; Implementation J.3(8–11)]**  
**Loại:** Bắt buộc  
**Áp dụng cho:** C90, C99  
**Phân tích:**  
Tính chất và tổ chức của dự án sẽ quyết định những thư viện và hàm nào trong các thư viện đó cần tuân theo chỉ thị này.

**Lý do:**  
Nhiều hàm trong Thư viện Chuẩn không yêu cầu kiểm tra tính hợp lệ của các tham số được truyền vào. Ngay cả khi có yêu cầu kiểm tra từ Chuẩn, hoặc khi các trình biên dịch tuyên bố rằng họ kiểm tra tham số, cũng không có gì đảm bảo rằng quá trình kiểm tra được thực hiện đầy đủ. Tương tự, phần mô tả giao diện của các hàm trong các thư viện khác có thể không chỉ rõ các kiểm tra cần thực hiện. Cũng có rủi ro rằng các kiểm tra được chỉ rõ có thể không được thực hiện đầy đủ.

Lập trình viên phải cung cấp các kiểm tra phù hợp cho giá trị đầu vào của tất cả các hàm thư viện có một miền giá trị đầu vào bị hạn chế (Thư viện Chuẩn, thư viện bên thứ ba, và các thư viện nội bộ).

Ví dụ về các hàm trong Thư viện Chuẩn có miền đầu vào bị hạn chế và cần được kiểm tra gồm có:

* Nhiều hàm toán học trong <math.h>, ví dụ:
  + Không được truyền số âm vào các hàm sqrt hoặc log;
  + Tham số thứ hai của hàm fmod không được bằng 0;
* Một số triển khai có thể tạo ra kết quả không mong muốn khi hàm toupper nhận vào một tham số không phải là chữ thường (và tương tự đối với tolower);
* Các hàm kiểm tra ký tự trong <ctype.h> sẽ gây ra hành vi không xác định nếu nhận vào giá trị không hợp lệ;
* Hàm abs áp dụng lên số nguyên nhỏ nhất sẽ tạo ra hành vi không xác định.

Mặc dù hầu hết các hàm trong thư viện toán học <math.h> đã định nghĩa các miền đầu vào hợp lệ, các giá trị trả về khi xảy ra lỗi miền có thể khác nhau giữa các trình biên dịch. Do đó, việc kiểm tra tính hợp lệ của các giá trị đầu vào trước là đặc biệt quan trọng với các hàm này.

Lập trình viên nên xác định các ràng buộc miền nào nên áp dụng hợp lý cho một hàm đang được sử dụng (có thể hoặc không được tài liệu giao diện mô tả), và cung cấp các kiểm tra thích hợp để đảm bảo các giá trị đầu vào nằm trong miền này. Tất nhiên, giá trị có thể bị giới hạn hơn nữa nếu cần thiết, dựa trên hiểu biết về tham số và phạm vi giá trị hợp lý của nó.

Có nhiều cách để thỏa mãn yêu cầu của hướng dẫn này, bao gồm:

* Kiểm tra các giá trị trước khi gọi hàm;
* Kiểm tra các giá trị trong hàm thư viện đã gọi — điều này đặc biệt áp dụng cho các thư viện tự phát triển, mặc dù cũng có thể áp dụng cho các thư viện mua ngoài nếu nhà cung cấp có thể chứng minh rằng họ đã xây dựng các kiểm tra;
* Tạo phiên bản "gói" của các hàm để thực hiện kiểm tra, sau đó gọi hàm gốc;
* Chứng minh tĩnh rằng các tham số đầu vào không bao giờ có giá trị không hợp lệ.

**Xem thêm:**  
Dir 4.1, Dir 4.7

### Dir 4.12 Không sử dụng phân bổ bộ nhớ động

**Loại:** Bắt buộc  
**Áp dụng cho:** C90, C99  
**Phân tích:**  
Quy tắc này áp dụng cho tất cả các gói phân bổ bộ nhớ động bao gồm:

* Các gói được cung cấp bởi Thư viện Chuẩn;
* Các gói từ bên thứ ba.

**Lý do:**  
Các hàm phân bổ và giải phóng bộ nhớ động của Thư viện Chuẩn có thể dẫn đến hành vi không xác định như đã mô tả trong Quy tắc 21.3. Bất kỳ hệ thống phân bổ bộ nhớ động nào khác cũng có khả năng biểu hiện các hành vi không xác định tương tự như của Thư viện Chuẩn.

Cần kiểm tra kỹ các hàm từ bên thứ ba để đảm bảo rằng không sử dụng phân bổ bộ nhớ động một cách vô ý.

Nếu quyết định sử dụng bộ nhớ động, phải đảm bảo rằng phần mềm hoạt động theo cách dự đoán được. Ví dụ, có nguy cơ:

* Không đủ bộ nhớ để đáp ứng yêu cầu — cần phải đảm bảo có phản hồi an toàn và thích hợp khi phân bổ bộ nhớ thất bại;
* Thời gian thực thi phân bổ hoặc giải phóng bộ nhớ có thể dao động lớn tùy thuộc vào mô hình sử dụng và mức độ phân mảnh bộ nhớ.

**Ví dụ:**  
Ví dụ này sử dụng các hàm phân bổ bộ nhớ động của Thư viện Chuẩn vì giao diện của chúng được biết đến rộng rãi.

Trong ví dụ này, hành vi là không xác định sau lần gọi đầu tiên tới free vì giá trị của con trỏ p trở nên không xác định. Mặc dù giá trị được lưu trong con trỏ không thay đổi sau khi gọi free, có thể trong một số hệ thống, bộ nhớ mà nó trỏ tới không còn tồn tại, và việc sao chép con trỏ đó có thể gây ra lỗi bộ nhớ.

c

Sao chép mã

#include <stdlib.h>

void f(void) {

char \*p = (char \*)malloc(10);

char \*q;

free(p);

q = p; /\* Hành vi không xác định - giá trị của p là không xác định \*/

p = (char \*)malloc(20);

free(p);

p = NULL; /\* Gán NULL cho con trỏ đã được giải phóng khiến nó có giá trị xác định \*/

}

**Xem thêm:**  
Dir 4.1, Quy tắc 18.7, Quy tắc 21.3, Quy tắc 22.1, Quy tắc 22.2

### Dir 4.13 Các hàm được thiết kế để thực hiện các thao tác trên một tài nguyên phải được gọi theo đúng thứ tự

**Loại:** Khuyến nghị  
**Áp dụng cho:** C90, C99  
**Phân tích:**  
Một tập hợp các hàm thực hiện các thao tác trên một tài nguyên thường bao gồm ba loại thao tác:

1. Phân bổ tài nguyên, ví dụ: mở một tệp;
2. Giải phóng tài nguyên, ví dụ: đóng tệp;
3. Các thao tác khác, ví dụ: đọc từ tệp.

Đối với mỗi tập hợp hàm này, tất cả các thao tác của nó cần phải được gọi theo đúng thứ tự.

**Lý do:**  
Các công cụ phân tích tĩnh có khả năng cung cấp các kiểm tra phân tích đường dẫn, giúp xác định các đường dẫn trong chương trình mà hàm giải phóng không được gọi. Để tối đa hóa lợi ích của các kiểm tra tự động này, các nhà phát triển nên thiết kế và khai báo tập hợp các hàm cân bằng để công cụ phân tích tĩnh có thể nhận diện.

**Xem thêm:** Dir 4.1, Dir 4.7

Ví dụ

c

Sao chép mã

/\* Các hàm này được thiết kế để kết hợp với nhau \*/

extern mutex\_t mutex\_lock(void);

extern void mutex\_unlock(mutex\_t m);

extern int16\_t x;

void f(void) {

mutex\_t m = mutex\_lock();

if (x > 0) {

mutex\_unlock(m);

} else {

/\* Mutex không được mở khóa trên đường này \*/

}

}

**Xem thêm:**  
Quy tắc 22.1, Quy tắc 22.2, Quy tắc 22.6

## Quy tắc

### Một môi trường C tiêu chuẩn

#### Quy tắc 1.1

**Nội dung**

Chương trình không được vi phạm cú pháp tiêu chuẩn C và các ràng buộc, và không được vượt quá giới hạn dịch của môi trường triển khai.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại |  |
| Phân tích |  |
| Áp dụng |  |

- [Bảng 3 Hướng dẫn MISRA], [IEC 61508-7: Bảng C.1], [ISO 26262-6: Bảng 1]

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

**Mở rộng**

Chương trình chỉ nên sử dụng những tính năng của ngôn ngữ C và thư viện của nó được chỉ định trong phiên bản tiêu chuẩn đã chọn (xem Mục 3.1).

Tiêu chuẩn cho phép các môi trường triển khai cung cấp các mở rộng ngôn ngữ và việc sử dụng những mở rộng này được quy tắc này cho phép.

Trừ khi sử dụng một mở rộng ngôn ngữ, chương trình không được:

- Chứa bất kỳ vi phạm nào của cú pháp ngôn ngữ được mô tả trong Tiêu chuẩn;

- Chứa bất kỳ vi phạm nào của các ràng buộc do Tiêu chuẩn áp đặt.

Chương trình không được vượt quá các giới hạn dịch do môi trường triển khai áp đặt. Các giới hạn dịch tối thiểu được quy định bởi Tiêu chuẩn nhưng môi trường triển khai có thể cung cấp các giới hạn cao hơn.

**Lưu ý**

- Một triển khai tuân thủ sẽ tạo ra một chẩn đoán cho các vi phạm cú pháp và ràng buộc, nhưng hãy lưu ý rằng:

- Chẩn đoán không nhất thiết phải là một lỗi mà có thể là một cảnh báo;

- Chương trình có thể được dịch và một tệp thực thi có thể được tạo ra mặc dù có sự vi phạm cú pháp hoặc ràng buộc;

- Một triển khai tuân thủ không cần phải tạo ra chẩn đoán khi một giới hạn dịch bị vượt quá; một tệp thực thi có thể được tạo ra nhưng không được đảm bảo sẽ thực thi đúng.

**Lý do**

Các vấn đề liên quan đến các tính năng ngôn ngữ nằm ngoài các phiên bản được hỗ trợ của ISO/IEC 9899 không được xem xét trong quá trình phát triển các hướng dẫn này.

Có bằng chứng kể lại về việc một số triển khai không tuân thủ không phát hiện được các vi phạm ràng buộc, ví dụ như trong [38] trang 135, ví dụ 2 có tiêu đề “Lỗi ghi vào khu vực const”.

**Ví dụ**

Một số trình biên dịch C90 hỗ trợ các hàm nội tuyến (inline functions) bằng cách sử dụng từ khóa \_\_inline. Một chương trình C90 sử dụng \_\_inline sẽ tuân thủ quy tắc này miễn là nó được dịch bằng một trình biên dịch như vậy.

Nhiều trình biên dịch cho các mục tiêu nhúng cung cấp các từ khóa bổ sung xác định loại đối tượng với các thuộc tính của khu vực bộ nhớ mà đối tượng đó nằm, ví dụ:

- \_\_zpage — đối tượng có thể được truy cập bằng một lệnh ngắn

- \_\_near — con trỏ tới đối tượng có thể được lưu trong 16 bit

- \_\_far — con trỏ tới đối tượng có thể được lưu trong 24 bit

Một chương trình sử dụng các từ khóa bổ sung này sẽ tuân thủ quy tắc này miễn là trình biên dịch hỗ trợ những từ khóa này như một phần mở rộng ngôn ngữ.

**Xem thêm**

- Dir 2.1, Quy tắc 1.2

#### Quy tắc 1.2

Không nên sử dụng các phần mở rộng ngôn ngữ

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại |  |
| Phân tích |  |
| Áp dụng |  |

- Phân loại: Tư vấn

- Phân tích: Không quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

**Lý do**

Một chương trình phụ thuộc vào các phần mở rộng ngôn ngữ có thể kém di động hơn so với một chương trình không sử dụng. Mặc dù Tiêu chuẩn yêu cầu một triển khai tuân thủ phải tài liệu hóa bất kỳ phần mở rộng nào mà nó cung cấp cho ngôn ngữ, có nguy cơ rằng tài liệu này có thể không cung cấp mô tả đầy đủ về hành vi trong mọi hoàn cảnh.

Nếu quy tắc này không được áp dụng, quyết định sử dụng mỗi phần mở rộng ngôn ngữ nên được giải thích trong tài liệu thiết kế của dự án. Các phương pháp đảm bảo việc sử dụng hợp lệ của mỗi phần mở rộng, ví dụ như kiểm tra trình biên dịch và các chẩn đoán của nó, cũng nên được tài liệu hóa.

Cần nhận ra rằng việc sử dụng các phần mở rộng ngôn ngữ trong hệ thống nhúng là cần thiết. Tiêu chuẩn yêu cầu rằng một phần mở rộng không được thay đổi hành vi của bất kỳ chương trình nào tuân thủ nghiêm ngặt. Ví dụ, một trình biên dịch có thể triển khai, như một phần mở rộng, việc đánh giá đầy đủ các toán tử logic nhị phân mặc dù Tiêu chuẩn chỉ định rằng việc đánh giá dừng lại ngay khi kết quả có thể được xác định. Phần mở rộng như vậy không tuân thủ Tiêu chuẩn vì các tác dụng phụ trong toán hạng phải của một toán tử AND logic sẽ luôn xảy ra, dẫn đến một hành vi khác.

Xem thêm

- Quy tắc 1.1

#### Quy tắc 1.3

Không được có bất kỳ hành vi nào không xác định hoặc không xác định quan trọng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại |  |
| Phân tích |  |
| Áp dụng |  |

- Phân loại: Bắt buộc

- Phân tích: Không quyết định được, Hệ thống

- Áp dụng cho: C90, C99

#Mở rộng

Một số hành vi không xác định và không xác định được xử lý bởi các quy tắc cụ thể. Quy tắc này ngăn chặn tất cả các hành vi không xác định và không xác định quan trọng khác. Phụ lục H liệt kê các hành vi không xác định và những hành vi không xác định được coi là quan trọng.

Dưới đây là bản dịch của phần Rationale từ tài liệu MISRA Autosar từ tiếng Anh sang tiếng Việt:

---

### Lý do

Bất kỳ chương trình nào có hành vi không xác định hoặc không xác định có thể không hoạt động theo cách mong đợi. Trong nhiều trường hợp, hiệu ứng là làm cho chương trình không di động nhưng cũng có thể xảy ra các vấn đề nghiêm trọng hơn. Ví dụ, hành vi không xác định có thể ảnh hưởng đến kết quả của một phép tính. Nếu hoạt động chính xác của phần mềm phụ thuộc vào phép tính này thì an toàn hệ thống có thể bị ảnh hưởng. Vấn đề này đặc biệt khó phát hiện nếu hành vi không xác định chỉ xuất hiện trong những tình huống hiếm hoi.

Nhiều hướng dẫn của MISRA C được thiết kế để tránh một số hành vi không xác định và không xác định. Ví dụ, tuân thủ tất cả Quy tắc 11.4, Quy tắc 11.8 và Quy tắc 19.2 đảm bảo rằng không thể trong C tạo ra một con trỏ không phải là const tới một đối tượng được khai báo với kiểu const. Điều này tránh được C90 [Không xác định 39] và C99 [Không xác định 61]. Tuy nhiên, các hành vi khác không được bao phủ bởi các hướng dẫn cụ thể vì:

- Không có khả năng hành vi này sẽ được gặp phải;

- Không có hướng dẫn thực tiễn nào có thể được đưa ra ngoài việc tuyên bố rõ ràng rằng hành vi này nên được tránh.

Thay vì giới thiệu một hướng dẫn cho mỗi hành vi không xác định và không xác định quan trọng, Hướng dẫn MISRA C trực tiếp giải quyết những hành vi được coi là quan trọng nhất và có khả năng xảy ra nhất trong thực tế. Những hành vi không có hướng dẫn cụ thể đều được bao phủ cùng nhau bởi quy tắc này. Phụ lục H liệt kê tất cả các hành vi không xác định và không xác định quan trọng, cùng với các hướng dẫn MISRA C ngăn chặn sự xuất hiện của chúng. Do đó, nó chỉ ra những hành vi nào được kỳ vọng sẽ được ngăn chặn bởi quy tắc này và những hành vi nào được bao phủ bởi các quy tắc khác.

Lưu ý: một số triển khai có thể cung cấp hành vi xác định rõ cho một số hành vi không xác định và không xác định được liệt kê trong Tiêu chuẩn. Nếu dựa vào những hành vi xác định rõ này, bao gồm cả việc sử dụng phần mở rộng ngôn ngữ, sẽ cần phải lệch khỏi quy tắc này đối với những hành vi đó.

**Xem thêm**

- Dir 4.1

### Mã không sử dụng

#### Quy tắc 2.1

Một dự án không được chứa mã không thể truy cập

- [IEC 61508-7 Phần C.5.9], [DO-178C Phần 6.4.4.3.c]

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại |  |
| Phân tích |  |
| Áp dụng |  |

- Phân loại: Bắt buộc

- Phân tích: Không quyết định được, Hệ thống

- Áp dụng cho: C90, C99

**Lý do**

Với điều kiện là một chương trình không biểu hiện bất kỳ hành vi không xác định nào, mã không thể truy cập không thể được thực thi và không thể có bất kỳ ảnh hưởng nào đến đầu ra của chương trình. Do đó, sự hiện diện của mã không thể truy cập có thể chỉ ra một lỗi trong logic của chương trình.

Dưới đây là bản dịch của phần nội dung thêm từ tài liệu MISRA Autosar từ tiếng Anh sang tiếng Việt:

---

Một trình biên dịch được phép loại bỏ bất kỳ mã không thể truy cập nào mặc dù nó không bắt buộc phải làm như vậy. Mã không thể truy cập không bị loại bỏ bởi trình biên dịch gây lãng phí tài nguyên, ví dụ:

- Nó chiếm không gian trong bộ nhớ của máy mục tiêu;

- Sự hiện diện của nó có thể khiến trình biên dịch chọn các lệnh nhảy dài hơn và chậm hơn khi chuyển điều khiển xung quanh mã không thể truy cập;

- Trong một vòng lặp, nó có thể ngăn toàn bộ vòng lặp cư trú trong bộ nhớ đệm lệnh.

Đôi khi, cần thiết chèn mã dường như không thể truy cập để xử lý các trường hợp ngoại lệ. Ví dụ, trong một câu lệnh switch trong đó mọi giá trị có thể của biểu thức điều khiển đều được bao phủ bởi một case cụ thể, một mệnh đề default phải có mặt theo Quy tắc 16.4. Mục đích của mệnh đề default là để bẫy một giá trị không nên xuất hiện bình thường nhưng có thể đã được tạo ra do:

- Hành vi không xác định có mặt trong chương trình;

- Lỗi của phần cứng bộ vi xử lý.

Nếu một trình biên dịch có thể chứng minh rằng một mệnh đề default là không thể truy cập, nó có thể loại bỏ nó, do đó loại bỏ hành động phòng thủ. Với giả định rằng hành động phòng thủ là quan trọng, cần phải chứng minh rằng trình biên dịch không loại bỏ mã mặc dù nó không thể truy cập, hoặc thực hiện các bước để làm cho mã phòng thủ có thể truy cập. Hành động đầu tiên yêu cầu một sự lệch khỏi quy tắc này, có lẽ bằng cách xem xét mã đối tượng hoặc kiểm tra đơn vị để hỗ trợ sự lệch như vậy. Hành động thứ hai thường có thể được thực hiện bằng cách truy cập thông qua một lvalue đủ điều kiện volatile. Ví dụ, một trình biên dịch có thể xác định rằng phạm vi các giá trị được giữ bởi x được bao phủ bởi các mệnh đề case trong một câu lệnh switch như sau:

```c

uint16\_t x;

switch ( x )

```

Bằng cách buộc x được truy cập bằng một lvalue đủ điều kiện volatile, trình biên dịch phải giả định rằng biểu thức điều khiển có thể lấy bất kỳ giá trị nào:

```c

switch ( \*( volatile uint16\_t \* ) &x )

```

Lưu ý: mã đã bị loại trừ có điều kiện bởi các chỉ thị tiền xử lý không thuộc quy tắc này vì nó không được trình bày cho các giai đoạn dịch sau.

Ví dụ

```c

enum light { red, amber, red\_amber, green };

enum light next\_light ( enum light c )

{

enum light res;

switch ( c )

{

case red:

res = red\_amber;

break;

case red\_amber:

res = green;

break;

case green:

res = amber;

break;

case amber:

res = red;

break;

default:

// Mã xử lý phòng thủ cho giá trị không xác định

res = red; // Hoặc một hành động phù hợp khác

break;

}

return res;

}

```

Dưới đây là phần tiếp theo của bản dịch tài liệu MISRA Autosar từ tiếng Anh sang tiếng Việt:

---

```c

default:

{

/\*

\* Mệnh đề default này sẽ chỉ có thể truy cập nếu tham số c

\* giữ một giá trị không phải là thành viên của enum light.

\*/

error\_handler();

break;

}

}

return res;

res = c; /\* Không tuân thủ - câu lệnh này chắc chắn không thể truy cập \*/

}

```

Xem thêm

- Quy tắc 14.3, Quy tắc 16.4

#### Quy tắc 2.2

Không được có mã chết

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại |  |
| Phân tích |  |
| Áp dụng |  |

- [IEC 61508-7 Phần C.5.10], [ISO 26262-6 Phần 9.4.5], [DO-178C Phần 6.4.4.3.c]

- Phân loại: Bắt buộc

- Phân tích: Không quyết định được, Hệ thống

- Áp dụng cho: C90, C99

#Mở rộng

Bất kỳ thao tác nào được thực thi nhưng việc loại bỏ nó không ảnh hưởng đến hành vi của chương trình đều cấu thành mã chết. Các thao tác được giới thiệu bởi các phần mở rộng ngôn ngữ luôn được giả định có ảnh hưởng đến hành vi của chương trình.

Lưu ý: Hành vi của một hệ thống nhúng thường được xác định không chỉ bởi bản chất của các hành động của nó, mà còn bởi thời điểm chúng xảy ra.

Lưu ý: Mã không thể truy cập không phải là mã chết vì nó không thể được thực thi.

#Lý do

Sự hiện diện của mã chết có thể chỉ ra một lỗi trong logic của chương trình. Vì mã chết có thể bị loại bỏ bởi trình biên dịch, sự hiện diện của nó có thể gây ra sự nhầm lẫn.

#Ngoại lệ

Một ép kiểu sang void được giả định là để chỉ ra một giá trị không được sử dụng một cách có chủ đích. Do đó, ép kiểu không phải là mã chết. Nó được coi là sử dụng toán hạng của nó và do đó cũng không phải là mã chết.

Dưới đây là phần dịch của các quy tắc tiếp theo từ tài liệu MISRA Autosar từ tiếng Anh sang tiếng Việt:

---

#### Quy tắc 2.3

Một dự án không nên chứa các khai báo kiểu không sử dụng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại |  |
| Phân tích |  |
| Áp dụng |  |

- Phân loại: Tư vấn

- Phân tích: Quyết định được, Hệ thống

- Áp dụng cho: C90, C99

#Lý do

Nếu một kiểu được khai báo nhưng không được sử dụng, thì người đánh giá sẽ không rõ liệu kiểu đó là dư thừa hay đã bị bỏ qua một cách nhầm lẫn.

#Ví dụ

```c

int16\_t unusedtype ( void )

{

typedef int16\_t local\_Type; /\* Không tuân thủ \*/

return 67;

}

```

#### Quy tắc 2.4

Một dự án không nên chứa các khai báo nhãn không sử dụng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại |  |
| Phân tích |  |
| Áp dụng |  |

- Phân loại: Tư vấn

- Phân tích: Quyết định được, Hệ thống

- Áp dụng cho: C90, C99

Lý do

Nếu một nhãn được khai báo nhưng không sử dụng, thì người đánh giá sẽ không rõ liệu nhãn đó là dư thừa hay đã bị bỏ qua một cách nhầm lẫn.

Ví dụ

Trong ví dụ sau, nhãn `state` không được sử dụng và khai báo có thể được viết mà không cần nhãn này.

```c

void unusedtag ( void )

{

enum state { S\_init, S\_run, S\_sleep }; /\* Không tuân thủ \*/

}

```

Trong ví dụ sau, nhãn `record\_t` chỉ được sử dụng trong typedef của `record1\_t` mà được sử dụng trong phần còn lại của đơn vị dịch bất cứ khi nào cần kiểu này. Typedef này có thể được viết một cách tuân thủ bằng cách bỏ qua nhãn như được hiển thị trong định nghĩa của `record2\_t`.

```c

typedef struct record\_t /\* Không tuân thủ \*/

{

uint16\_t key;

uint16\_t val;

} record1\_t;

typedef struct /\* Tuân thủ \*/

{

uint16\_t key;

uint16\_t val;

} record2\_t;

```

Dưới đây là bản dịch các quy tắc tiếp theo từ tài liệu MISRA Autosar từ tiếng Anh sang tiếng Việt:

---

#### Quy tắc 2.5

Một dự án không nên chứa các khai báo macro không sử dụng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại |  |
| Phân tích |  |
| Áp dụng |  |

- Phân loại: Tư vấn

- Phân tích: Quyết định được, Hệ thống

- Áp dụng cho: C90, C99

#Lý do

Nếu một macro được khai báo nhưng không được sử dụng, thì người đánh giá sẽ không rõ liệu macro đó là dư thừa hay đã bị bỏ qua một cách nhầm lẫn.

#Ví dụ

```c

void use\_macro ( void )

{

#define SIZE 4

/\* Không tuân thủ - DATA không được sử dụng \*/

#define DATA 3

use\_int16 ( SIZE );

}

```

#### Quy tắc 2.6

Một hàm không nên chứa các khai báo nhãn không sử dụng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại |  |
| Phân tích |  |
| Áp dụng |  |

- Phân loại: Tư vấn

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

Lý do

Nếu một nhãn được khai báo nhưng không được sử dụng, thì người đánh giá sẽ không rõ liệu nhãn đó là dư thừa hay đã bị bỏ qua một cách nhầm lẫn.

Ví dụ

```c

void unused\_label ( void )

{

int16\_t x = 6;

label1: /\* Không tuân thủ \*/

use\_int16 ( x );

}

```

#### Quy tắc 2.7

Không nên có các tham số không sử dụng trong các hàm

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại |  |
| Phân tích |  |
| Áp dụng |  |

- Phân loại: Tư vấn

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

Lý do

Hầu hết các hàm sẽ được chỉ định là sử dụng từng tham số của chúng. Nếu một tham số hàm không được sử dụng, có thể việc triển khai hàm không khớp với đặc tả của nó. Quy tắc này nhấn mạnh những sự không khớp tiềm năng như vậy.

Dưới đây là phần dịch các quy tắc và ví dụ tiếp theo từ tài liệu MISRA Autosar từ tiếng Anh sang tiếng Việt:

---

Ví dụ

```c

void withunusedpara ( uint16\_t \*para1, int16\_t unusedpara ) /\* Không tuân thủ - không sử dụng \*/

{

\*para1 = 42U;

}

```

### Bình luận (Comments)

#### Quy tắc 3.1

Chuỗi ký tự /\* và // không được sử dụng trong một bình luận.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại |  |
| Phân tích |  |
| Áp dụng |  |

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Lý do

Nếu một chuỗi bắt đầu bình luận, /\* hoặc //, xuất hiện trong một bình luận /\*, rất có thể là do thiếu chuỗi kết thúc bình luận \*/.

Nếu một chuỗi bắt đầu bình luận xuất hiện trong một bình luận //, có thể là vì một vùng mã đã được bình luận ra sử dụng //.

#Ngoại lệ

Chuỗi // được phép trong một bình luận //.

#Ví dụ

Xem xét đoạn mã sau:

```c

/\* một số bình luận, dấu kết thúc bình luận vô tình bị bỏ qua

<<New Page>>

Perform\_Critical\_Safety\_Function( X );

/\* bình luận này không tuân thủ \*/

```

Khi xem xét trang chứa lời gọi đến hàm, giả định là nó được thực thi. Vì vô tình bỏ sót dấu kết thúc bình luận, lời gọi đến hàm an toàn quan trọng sẽ không được thực thi.

Trong ví dụ C99 sau, sự hiện diện của các bình luận // thay đổi ý nghĩa của chương trình:

```c

x = y // /\*

+ z

// \*/

;

```

Điều này cho x = y + z; nhưng sẽ là x = y; nếu không có hai chuỗi bắt đầu bình luận //.

Xem thêm

- Dir 4.4

#### Quy tắc 3.2

Không được sử dụng nối dòng trong các bình luận //

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại |  |
| Phân tích |  |
| Áp dụng |  |

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C99

Mở rộng

Nối dòng xảy ra khi ký tự \ được theo sau ngay lập tức bởi ký tự xuống dòng. Nếu tệp nguồn chứa các ký tự đa byte, chúng sẽ được chuyển đổi sang bộ ký tự nguồn trước khi xảy ra nối dòng.

Lý do

Nếu dòng nguồn chứa một bình luận // kết thúc bằng ký tự \ trong bộ ký tự nguồn, dòng tiếp theo trở thành một phần của bình luận. Điều này có thể dẫn đến việc loại bỏ mã không mong muốn.

Lưu ý: nối dòng được mô tả trong Mục 5.1.1.2(2) của cả C90 và C99.

Ví dụ

Trong ví dụ không tuân thủ sau, dòng vật lý chứa từ khóa if về mặt logic là một phần của dòng trước đó và do đó là một bình luận.

```c

extern bool\_t b;

void f ( void )

{

uint16\_t x = 0; // bình luận \

if ( b )

{

++x; /\* Điều này luôn được thực thi \*/

}

}

```

Xem thêm

- Dir 4.4

### 8.4 Bộ ký tự và quy ước từ vựng

#### Quy tắc 4.1

Các chuỗi thoát bát phân và thập lục phân phải được kết thúc.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại |  |
| Phân tích |  |
| Áp dụng |  |

- C90 [Triển khai 11], C99 [Triển khai J.3.4(7, 8)]

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Mở rộng

Một chuỗi thoát bát phân hoặc thập lục phân phải được kết thúc bởi một trong các điều sau:

- Bắt đầu của một chuỗi thoát khác, hoặc

- Kết thúc của hằng ký tự hoặc kết thúc của chuỗi ký tự.

Lý do

Có khả năng gây nhầm lẫn nếu một chuỗi thoát bát phân hoặc thập lục phân được theo sau bởi các ký tự khác. Ví dụ, hằng ký tự '\x1f' bao gồm một ký tự duy nhất trong khi hằng ký tự '\x1g' bao gồm hai ký tự '\x1' và 'g'. Cách mà các hằng ký tự đa ký tự được biểu diễn dưới dạng số nguyên là do triển khai định nghĩa.

Khả năng gây nhầm lẫn sẽ giảm nếu mọi chuỗi thoát bát phân hoặc thập lục phân trong hằng ký tự hoặc chuỗi ký tự đều được kết thúc.

Ví dụ

Trong ví dụ này, mỗi chuỗi được trỏ tới bởi s1, s2 và s3 đều tương đương với “Ag”.

```c

const char \*s1 = "\x41g"; /\* Không tuân thủ \*/

const char \*s2 = "\x41" "g"; /\* Tuân thủ - kết thúc bởi kết thúc của chuỗi \*/

const char \*s3 = "\x41\x67"; /\* Tuân thủ - kết thúc bởi một chuỗi thoát khác \*/

int c1 = '\141t'; /\* Không tuân thủ \*/

int c2 = '\141\t'; /\* Tuân thủ - kết thúc bởi một chuỗi thoát khác \*/

```

Xem thêm

- C90: Phần 6.1.3.4, C99: Phần 6.4.4.4

#### Quy tắc 4.2

Không nên sử dụng các trigraf

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại |  |
| Phân tích |  |
| Áp dụng |  |

- Phân loại: Tư vấn

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Lý do

Các trigraf được biểu thị bằng một chuỗi hai dấu hỏi theo sau bởi một ký tự thứ ba được chỉ định (ví dụ: ??- biểu thị một ký tự ~ (dấu ngã) và ??) biểu thị một ] ). Chúng có thể gây nhầm lẫn tình cờ với các cách sử dụng hai dấu hỏi khác.

Lưu ý: các digraf gọi là:

- <: :> <% %> %: %:%:

được phép vì chúng là các token. Các trigraf được thay thế bất cứ nơi nào chúng xuất hiện trong chương trình trước khi tiền xử lý.

#Ví dụ

Ví dụ chuỗi

```c

"(Date should be in the form ??-??-??)"

```

sẽ không hoạt động như mong đợi, thực sự được trình biên dịch hiểu là

```c

"(Date should be in the form ~~]"

```

---

Dưới đây là bản dịch các quy tắc tiếp theo từ tài liệu MISRA Autosar từ tiếng Anh sang tiếng Việt:

---

### Định danh

#### Quy tắc 5.1

Các định danh bên ngoài phải khác biệt.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại |  |
| Phân tích |  |
| Áp dụng |  |

- C90 [Không xác định 7], C99 [Không xác định 7; Không xác định 28]

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Hệ thống

- Áp dụng cho: C90, C99

#Mở rộng

Quy tắc này yêu cầu các định danh bên ngoài khác nhau phải khác biệt trong giới hạn do triển khai áp đặt.

Định nghĩa của sự khác biệt phụ thuộc vào triển khai và phiên bản của ngôn ngữ C đang được sử dụng:

- Trong C90, yêu cầu tối thiểu là 6 ký tự đầu tiên của các định danh bên ngoài là quan trọng nhưng không yêu cầu phân biệt chữ hoa chữ thường;

- Trong C99, yêu cầu tối thiểu là 31 ký tự đầu tiên của các định danh bên ngoài là quan trọng, với mỗi ký tự toàn cầu hoặc ký tự nguồn mở rộng tương ứng chiếm từ 6 đến 10 ký tự.

Trên thực tế, nhiều triển khai cung cấp các giới hạn lớn hơn. Ví dụ, thường thấy các định danh bên ngoài trong C90 là phân biệt chữ hoa chữ thường và ít nhất 31 ký tự đầu tiên là quan trọng.

#Lý do

Nếu hai định danh chỉ khác nhau ở các ký tự không quan trọng, hành vi là không xác định.

Nếu khả năng di chuyển là một mối quan tâm, nên áp dụng quy tắc này bằng cách sử dụng các giới hạn tối thiểu được chỉ định trong Tiêu chuẩn.

Các định danh dài có thể làm giảm khả năng đọc của mã. Mặc dù nhiều hệ thống tạo mã tự động tạo ra các định danh dài, có lý do tốt để giữ độ dài định danh dưới mức giới hạn này.

Lưu ý: Trong C99, nếu một ký tự nguồn mở rộng xuất hiện trong một định danh bên ngoài và ký tự đó không có ký tự toàn cầu tương ứng, Tiêu chuẩn không chỉ định bao nhiêu ký tự nó chiếm.

#Ví dụ

Trong ví dụ sau, tất cả các định nghĩa đều xảy ra trong cùng một đơn vị dịch. Triển khai liên quan hỗ trợ 31 ký tự phân biệt chữ hoa chữ thường trong các định danh bên ngoài.

```c

/\* 1234567890123456789012345678901\*\*\*\*\*\*\*\*\* Characters \*/

int32\_t engine\_exhaust\_gas\_temperature\_raw;

int32\_t engine\_exhaust\_gas\_temperature\_scaled; /\* Không tuân thủ \*/

/\* 1234567890123456789012345678901\*\*\*\*\*\*\*\*\* Characters \*/

int32\_t engine\_exhaust\_gas\_temp\_raw;

int32\_t engine\_exhaust\_gas\_temp\_scaled; /\* Tuân thủ \*/

```

Dưới đây là phần dịch các quy tắc và ví dụ tiếp theo từ tài liệu MISRA Autosar từ tiếng Anh sang tiếng Việt:

---

Trong ví dụ không tuân thủ sau, triển khai hỗ trợ 6 ký tự phân biệt chữ hoa chữ thường trong các định danh bên ngoài. Các định danh trong hai đơn vị dịch khác nhau nhưng không khác biệt trong các ký tự quan trọng.

```c

/\* file1.c \*/

int32\_t abc = 0;

/\* file2.c \*/

int32\_t ABC = 0;

```

Xem thêm

- Dir 1.1, Quy tắc 5.2, Quy tắc 5.4, Quy tắc 5.5

#### Quy tắc 5.2

Các định danh được khai báo trong cùng phạm vi và không gian tên phải khác biệt.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại |  |
| Phân tích |  |
| Áp dụng |  |

- C90 [Không xác định 7], C99 [Không xác định 28]

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Mở rộng

Quy tắc này không áp dụng nếu cả hai định danh đều là định danh bên ngoài vì trường hợp này được bao phủ bởi Quy tắc 5.1.

Quy tắc này không áp dụng nếu một trong hai định danh là định danh macro vì trường hợp này được bao phủ bởi Quy tắc 5.4 và Quy tắc 5.5.

Định nghĩa của sự khác biệt phụ thuộc vào triển khai và phiên bản của ngôn ngữ C đang được sử dụng:

- Trong C90, yêu cầu tối thiểu là 31 ký tự đầu tiên là quan trọng;

- Trong C99, yêu cầu tối thiểu là 63 ký tự đầu tiên là quan trọng, với mỗi ký tự toàn cầu hoặc ký tự nguồn mở rộng tính là một ký tự.

#Lý do

Nếu hai định danh chỉ khác nhau ở các ký tự không quan trọng, hành vi là không xác định.

Nếu khả năng di chuyển là một mối quan tâm, nên áp dụng quy tắc này bằng cách sử dụng các giới hạn tối thiểu được chỉ định trong Tiêu chuẩn.

Các định danh dài có thể làm giảm khả năng đọc của mã. Mặc dù nhiều hệ thống tạo mã tự động tạo ra các định danh dài, có lý do tốt để giữ độ dài định danh dưới mức giới hạn này.

#Ví dụ

Trong ví dụ sau, triển khai liên quan hỗ trợ 31 ký tự phân biệt chữ hoa chữ thường trong các định danh không có liên kết bên ngoài. Định danh `engine\_exhaust\_gas\_temperature\_local` tuân thủ quy tắc này. Mặc dù nó không khác biệt so với định danh `engine\_exhaust\_gas\_temperature\_raw`, nó nằm trong một phạm vi khác. Tuy nhiên, nó không tuân thủ Quy tắc 5.3.

```c

int32\_t engine\_exhaust\_gas\_temperature\_raw;

void f(void)

{

int32\_t engine\_exhaust\_gas\_temperature\_local; /\* Tuân thủ quy tắc 5.2 \*/

/\* Không tuân thủ quy tắc 5.3 nếu được sử dụng cùng với định danh trên \*/

}

```

Dưới đây là phần dịch tiếp theo của các quy tắc và ví dụ từ tài liệu MISRA Autosar từ tiếng Anh sang tiếng Việt:

---

```c

/\* 1234567890123456789012345678901\*\*\*\*\*\*\*\*\* Characters \*/

extern int32\_t engine\_exhaust\_gas\_temperature\_raw;

static int32\_t engine\_exhaust\_gas\_temperature\_scaled; /\* Không tuân thủ \*/

void f ( void )

{

/\* 1234567890123456789012345678901\*\*\*\*\*\*\*\*\* Characters \*/

int32\_t engine\_exhaust\_gas\_temperature\_local; /\* Tuân thủ \*/

}

/\* 1234567890123456789012345678901\*\*\*\*\*\*\*\*\* Characters \*/

static int32\_t engine\_exhaust\_gas\_temp\_raw;

static int32\_t engine\_exhaust\_gas\_temp\_scaled; /\* Tuân thủ \*/

```

Xem thêm

- Dir 1.1, Quy tắc 5.1, Quy tắc 5.3, Quy tắc 5.4, Quy tắc 5.5

#### Quy tắc 5.3

Một định danh được khai báo trong một phạm vi bên trong không được che khuất một định danh được khai báo trong một phạm vi bên ngoài

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Mở rộng

Một định danh được khai báo trong một phạm vi bên trong phải khác biệt so với bất kỳ định danh nào được khai báo trong một phạm vi bên ngoài. Định nghĩa của sự khác biệt phụ thuộc vào triển khai và phiên bản của ngôn ngữ C đang được sử dụng:

- Trong C90, yêu cầu tối thiểu là 31 ký tự đầu tiên là quan trọng;

- Trong C99, yêu cầu tối thiểu là 63 ký tự đầu tiên là quan trọng, với mỗi ký tự toàn cầu hoặc ký tự nguồn mở rộng tính là một ký tự.

#Lý do

Nếu một định danh được khai báo trong một phạm vi bên trong nhưng không khác biệt so với một định danh đã tồn tại trong một phạm vi bên ngoài, thì khai báo bên trong nhất sẽ "che khuất" cái bên ngoài. Điều này có thể dẫn đến sự nhầm lẫn cho nhà phát triển.

Lưu ý: Một định danh được khai báo trong một không gian tên không che khuất một định danh được khai báo trong một không gian tên khác.

Các thuật ngữ phạm vi bên ngoài và phạm vi bên trong được định nghĩa như sau:

- Các định danh có phạm vi tệp có thể được coi là có phạm vi ngoài cùng;

- Các định danh có phạm vi khối có phạm vi bên trong hơn;

- Các khối lồng nhau kế tiếp nhau, giới thiệu các phạm vi bên trong hơn.

#Ví dụ

```c

extern int32\_t engine\_exhaust\_gas\_temperature\_raw;

void f ( void )

{

int32\_t engine\_exhaust\_gas\_temperature\_local; /\* Không tuân thủ \*/

}

```

Trong ví dụ này, định danh `engine\_exhaust\_gas\_temperature\_local` được khai báo trong phạm vi bên trong của hàm `f`, nhưng không khác biệt so với định danh `engine\_exhaust\_gas\_temperature\_raw` được khai báo trong phạm vi bên ngoài, do đó vi phạm quy tắc này.

Dưới đây là bản dịch tiếp theo của các quy tắc và ví dụ từ tài liệu MISRA Autosar từ tiếng Anh sang tiếng Việt:

---

Ví dụ

```c

void fn1 ( void )

{

int16\_t i; /\* Khai báo một đối tượng "i" \*/

{

int16\_t i; /\* Không tuân thủ - che khuất "i" trước đó \*/

i = 3; /\* Có thể gây nhầm lẫn là "i" nào được tham chiếu \*/

}

}

struct astruct

{

int16\_t m;

};

extern void g ( struct astruct \*p );

int16\_t xyz = 0; /\* Khai báo một đối tượng "xyz" \*/

void fn2 ( struct astruct xyz ) /\* Không tuân thủ - "xyz" ngoài bị che khuất bởi tên tham số \*/

{

g ( &xyz );

}

uint16\_t speed;

void fn3 ( void )

{

typedef float32\_t speed; /\* Không tuân thủ - kiểu che khuất đối tượng \*/

}

```

Xem thêm

- Quy tắc 5.2, Quy tắc 5.8

#### Quy tắc 5.4

Các định danh macro phải khác biệt

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- C90 [Không xác định 7], C99 [Không xác định 7; Không xác định 28]

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Mở rộng

Quy tắc này yêu cầu rằng, khi một macro được định nghĩa, tên của nó phải khác biệt với:

- Tên của các macro khác hiện đang được định nghĩa; và

- Tên của các tham số của chúng.

Nó cũng yêu cầu rằng các tên của các tham số của một macro nhất định phải khác biệt với nhau nhưng không yêu cầu rằng các tên tham số macro phải khác biệt giữa hai macro khác nhau.

Dưới đây là phần dịch tiếp theo của các quy tắc và ví dụ từ tài liệu MISRA Autosar từ tiếng Anh sang tiếng Việt:

---

Định nghĩa của sự khác biệt phụ thuộc vào triển khai và phiên bản của ngôn ngữ C đang được sử dụng:

- Trong C90, yêu cầu tối thiểu là 31 ký tự đầu tiên của các định danh macro là quan trọng;

- Trong C99, yêu cầu tối thiểu là 63 ký tự đầu tiên của các định danh macro là quan trọng.

Trên thực tế, các triển khai có thể cung cấp giới hạn lớn hơn. Quy tắc này yêu cầu các định danh macro phải khác biệt trong giới hạn do triển khai áp đặt.

#Lý do

Nếu hai định danh macro chỉ khác nhau ở các ký tự không quan trọng, hành vi là không xác định. Vì các tham số macro chỉ hoạt động trong quá trình mở rộng của macro, không có vấn đề với các tham số trong một macro bị nhầm lẫn với các tham số trong một macro khác.

Nếu khả năng di chuyển là một mối quan tâm, nên áp dụng quy tắc này bằng cách sử dụng các giới hạn tối thiểu được chỉ định trong Tiêu chuẩn.

Các định danh macro dài có thể làm giảm khả năng đọc của mã. Mặc dù nhiều hệ thống tạo mã tự động tạo ra các định danh macro dài, có lý do tốt để giữ độ dài định danh macro dưới mức giới hạn này.

Lưu ý: Trong C99, nếu một ký tự nguồn mở rộng xuất hiện trong tên macro và ký tự đó không có ký tự toàn cầu tương ứng, Tiêu chuẩn không chỉ định bao nhiêu ký tự nó chiếm.

#Ví dụ

Trong ví dụ sau, triển khai liên quan hỗ trợ 31 ký tự phân biệt chữ hoa chữ thường trong các định danh macro.

```c

/\* 1234567890123456789012345678901\*\*\*\*\*\*\*\*\* Characters \*/

#define engine\_exhaust\_gas\_temperature\_raw egt\_r

#define engine\_exhaust\_gas\_temperature\_scaled egt\_s /\* Không tuân thủ \*/

/\* 1234567890123456789012345678901\*\*\*\*\*\*\*\*\* Characters \*/

#define engine\_exhaust\_gas\_temp\_raw egt\_r

#define engine\_exhaust\_gas\_temp\_scaled egt\_s /\* Tuân thủ \*/

```

Xem thêm

- Quy tắc 5.1, Quy tắc 5.2, Quy tắc 5.5

Dưới đây là phần dịch tiếp theo của các quy tắc và ví dụ từ tài liệu MISRA Autosar từ tiếng Anh sang tiếng Việt:

---

#### Quy tắc 5.5

Các định danh phải khác biệt so với tên macro

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- C90 [Không xác định 7], C99 [Không xác định 7; Không xác định 28]

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Mở rộng

Quy tắc này yêu cầu rằng tên của các macro tồn tại trước khi tiền xử lý phải khác biệt so với các định danh tồn tại sau khi tiền xử lý. Nó áp dụng cho các định danh, bất kể phạm vi hoặc không gian tên, và cho bất kỳ macro nào đã được định nghĩa bất kể định nghĩa đó còn hiệu lực khi định danh được khai báo.

Định nghĩa của sự khác biệt phụ thuộc vào triển khai và phiên bản của ngôn ngữ C đang được sử dụng:

- Trong C90, yêu cầu tối thiểu là 31 ký tự đầu tiên là quan trọng;

- Trong C99, yêu cầu tối thiểu là 63 ký tự đầu tiên là quan trọng, với mỗi ký tự toàn cầu hoặc ký tự nguồn mở rộng tính là một ký tự.

#Lý do

Giữ cho tên macro và định danh khác biệt có thể giúp tránh sự nhầm lẫn của nhà phát triển.

#Ví dụ

Trong ví dụ không tuân thủ sau, tên của macro dạng hàm Sum cũng được sử dụng như một định danh. Khai báo của đối tượng Sum không phải chịu sự mở rộng macro vì nó không được theo sau bởi một ký tự (. Do đó, định danh tồn tại sau khi tiền xử lý đã được thực hiện.

```c

#define Sum(x, y) ( ( x ) + ( y ) )

int16\_t Sum;

```

Ví dụ sau đây tuân thủ vì không có trường hợp nào của định danh Sum sau khi tiền xử lý.

```c

#define Sum(x, y) ( ( x ) + ( y ) )

int16\_t x = Sum ( 1, 2 );

```

Trong ví dụ sau, triển khai liên quan hỗ trợ 31 ký tự phân biệt chữ hoa chữ thường trong các định danh không có liên kết bên ngoài. Ví dụ này không tuân thủ vì tên macro không khác biệt so với tên định danh có liên kết nội bộ trong 31 ký tự đầu tiên.

```c

/\* 1234567890123456789012345678901\*\*\*\*\*\*\*\*\* Characters \*/

#define low\_pressure\_turbine\_temperature\_1 lp\_tb\_temp\_1

static int32\_t low\_pressure\_turbine\_temperature\_2;

```

Xem thêm

- Quy tắc 5.1, Quy tắc 5.2, Quy tắc 5.4

Dưới đây là phần dịch tiếp theo của các quy tắc và ví dụ từ tài liệu MISRA Autosar từ tiếng Anh sang tiếng Việt:

---

#### Quy tắc 5.6

Tên typedef phải là định danh duy nhất

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Hệ thống

- Áp dụng cho: C90, C99

#Mở rộng

Tên typedef phải duy nhất trên tất cả các không gian tên và đơn vị dịch. Nhiều khai báo cùng một tên typedef chỉ được phép theo quy tắc này nếu định nghĩa kiểu được thực hiện trong một tệp tiêu đề và tệp tiêu đề đó được bao gồm trong nhiều tệp nguồn.

#Lý do

Sử dụng lại tên typedef như một tên typedef khác hoặc là tên của một hàm, đối tượng hoặc hằng số liệt kê có thể dẫn đến sự nhầm lẫn cho nhà phát triển.

#Ngoại lệ

Tên typedef có thể giống với tên thẻ cấu trúc, liên hiệp hoặc liệt kê liên quan đến typedef đó.

#Ví dụ

```c

void func ( void )

{

{

typedef unsigned char u8\_t;

}

{

typedef unsigned char u8\_t; /\* Không tuân thủ - sử dụng lại \*/

}

}

typedef float mass;

void func1 ( void )

{

float32\_t mass = 0.0f; /\* Không tuân thủ - sử dụng lại \*/

}

typedef struct list

{

struct list \*next;

uint16\_t element;

} list; /\* Tuân thủ - ngoại lệ \*/

typedef struct

{

struct chain

{

struct chain \*list;

uint16\_t element;

} s1;

uint16\_t length;

} chain; /\* Không tuân thủ - thẻ "chain" không liên quan đến typedef \*/

```

Xem thêm

- Quy tắc 5.7

Dưới đây là phần dịch tiếp theo của các quy tắc và ví dụ từ tài liệu MISRA Autosar từ tiếng Anh sang tiếng Việt:

---

#### Quy tắc 5.7

Tên thẻ phải là định danh duy nhất

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Hệ thống

- Áp dụng cho: C90, C99

#Mở rộng

Tên thẻ phải duy nhất trên tất cả các không gian tên và đơn vị dịch. Tất cả các khai báo của thẻ phải chỉ định cùng một kiểu. Nhiều khai báo hoàn chỉnh của cùng một thẻ chỉ được phép theo quy tắc này nếu thẻ được khai báo trong một tệp tiêu đề và tệp tiêu đề đó được bao gồm trong nhiều tệp nguồn.

#Lý do

Sử dụng lại tên thẻ có thể dẫn đến sự nhầm lẫn cho nhà phát triển. Có hành vi không xác định liên quan đến việc sử dụng lại tên thẻ trong C90 mặc dù điều này không được liệt kê trong Phụ lục của Tiêu chuẩn. Hành vi không xác định này đã được công nhận trong C99 như một ràng buộc trong Mục 6.7.2.3.

#Ngoại lệ

Tên thẻ có thể giống với tên typedef mà nó liên quan.

#Ví dụ

```c

struct stag

{

uint16\_t a;

uint16\_t b;

};

struct stag a1 = { 0, 0 }; /\* Tuân thủ - tương thích với trên \*/

union stag a2 = { 0, 0 }; /\* Không tuân thủ - khai báo kiểu khác so với struct stag. Vi phạm ràng buộc trong C99 \*/

```

Ví dụ sau đây cũng vi phạm Quy tắc 5.3

```c

struct deer

{

uint16\_t a;

uint16\_t b;

};

void foo ( void )

{

struct deer

{

uint16\_t a;

}; /\* Không tuân thủ - tên thẻ "deer" được sử dụng lại \*/

}

```

```c

typedef struct coord

{

uint16\_t x;

uint16\_t y;

} coord; /\* Tuân thủ theo ngoại lệ \*/

```

Dưới đây là phần dịch tiếp theo của các quy tắc và ví dụ từ tài liệu MISRA Autosar từ tiếng Anh sang tiếng Việt:

---

```c

struct elk

{

uint16\_t x;

};

struct elk /\* Không tuân thủ - khai báo kiểu khác Vi phạm ràng buộc trong C99 \*/

{

uint32\_t x;

};

```

Xem thêm

- Quy tắc 5.6

#### Quy tắc 5.8

Các định danh định nghĩa đối tượng hoặc hàm với liên kết bên ngoài phải là duy nhất.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Hệ thống

- Áp dụng cho: C90, C99

#Mở rộng

Một định danh được sử dụng làm định danh bên ngoài không được sử dụng cho bất kỳ mục đích nào khác trong bất kỳ không gian tên hoặc đơn vị dịch nào, ngay cả khi nó biểu thị một đối tượng không có liên kết.

#Lý do

Đảm bảo tính duy nhất của tên định danh theo cách này giúp tránh sự nhầm lẫn. Các định danh của các đối tượng không có liên kết không cần phải duy nhất vì có rất ít nguy cơ nhầm lẫn.

#Ví dụ

Trong ví dụ sau, file1.c và file2.c đều là một phần của cùng một dự án.

```c

/\* file1.c \*/

int32\_t count; /\* "count" có liên kết bên ngoài \*/

void foo ( void ) /\* "foo" có liên kết bên ngoài \*/

{

int16\_t index; /\* "index" không có liên kết \*/

}

/\* file2.c \*/

static void foo ( void ) /\* Không tuân thủ - "foo" không duy nhất vì đã được định nghĩa với liên kết bên ngoài trong file1.c \*/

{

int16\_t count; /\* Không tuân thủ - "count" không có liên kết nhưng xung đột với định danh có liên kết bên ngoài \*/

int32\_t index; /\* Tuân thủ - "index" không có liên kết \*/

}

```

Xem thêm

- Quy tắc 5.3

#### Quy tắc 5.9

Các định danh định nghĩa đối tượng hoặc hàm với liên kết nội bộ nên là duy nhất.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- Phân loại: Tư vấn

- Phân tích: Quyết định được, Hệ thống

- Áp dụng cho: C90, C99

#Mở rộng

Tên định danh nên là duy nhất trên tất cả các không gian tên và đơn vị dịch. Bất kỳ định danh nào được sử dụng theo cách này không nên có cùng tên với bất kỳ định danh nào khác, ngay cả khi định danh đó biểu thị một đối tượng không có liên kết.

#Lý do

Đảm bảo tính duy nhất của tên định danh theo cách này giúp tránh sự nhầm lẫn.

#Ngoại lệ

Một hàm nội tuyến với liên kết nội bộ có thể được định nghĩa trong nhiều đơn vị dịch nếu tất cả các định nghĩa đó được thực hiện trong cùng một tệp tiêu đề được bao gồm trong mỗi đơn vị dịch.

#Ví dụ

Trong ví dụ sau, file1.c và file2.c đều là một phần của cùng một dự án.

```c

/\* file1.c \*/

static int32\_t count; /\* "count" có liên kết nội bộ \*/

static void foo ( void ) /\* "foo" có liên kết nội bộ \*/

{

int16\_t count; /\* Không tuân thủ - "count" không có liên kết nhưng xung đột với định danh có liên kết nội bộ \*/

int16\_t index; /\* "index" không có liên kết \*/

}

void bar1 ( void )

{

static int16\_t count; /\* Không tuân thủ - "count" không có liên kết nhưng xung đột với định danh có liên kết nội bộ \*/

int16\_t index; /\* Tuân thủ - "index" không duy nhất nhưng không có liên kết \*/

foo ( );

}

/\* End of file1.c \*/

/\* file2.c \*/

static int8\_t count; /\* Không tuân thủ - "count" có liên kết nội bộ nhưng xung đột với các định danh khác cùng tên \*/

static void foo ( void ) /\* Không tuân thủ - "foo" có liên kết nội bộ nhưng xung đột với một hàm cùng tên \*/

{

int32\_t index; /\* Tuân thủ - cả "index" và "nbytes" đều không duy nhất nhưng không có liên kết \*/

int16\_t nbytes; /\* không có liên kết \*/

}

void bar2 ( void )

{

static uint8\_t nbytes; /\* Tuân thủ - "nbytes" không duy nhất nhưng không có liên kết và lớp lưu trữ không liên quan \*/

}

/\* End of file2.c \*/

```

Xem thêm

- Quy tắc 8.10

### Các loại

#### Quy tắc 6.1

Các trường bit chỉ nên được khai báo với loại thích hợp.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- C90 [Không xác định 38; Triển khai 29], C99 [Triển khai J.3.9(1, 2)]

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Mở rộng

Các loại trường bit thích hợp là:

- C90: hoặc unsigned int hoặc signed int;

- C99: một trong các loại:

- hoặc unsigned int hoặc signed int;

- một loại số nguyên khác được ký rõ ràng hoặc không ký rõ ràng được cho phép bởi triển khai;

- \_Bool.

Lưu ý: Sử dụng typedef để chỉ định một loại thích hợp được cho phép.

#Lý do

Sử dụng int là do triển khai định nghĩa vì các trường bit kiểu int có thể là kiểu signed hoặc unsigned. Sử dụng enum, short, char hoặc bất kỳ loại nào khác cho các trường bit không được phép trong C90 vì hành vi là không xác định.

Trong C99, triển khai có thể xác định các loại số nguyên khác được cho phép trong các khai báo trường bit.

Dưới đây là bản dịch tiếp theo của các quy tắc và ví dụ từ tài liệu MISRA Autosar từ tiếng Anh sang tiếng Việt:

---

Ví dụ

Ví dụ sau đây áp dụng cho C90 và các triển khai C99 không cung cấp bất kỳ loại trường bit bổ sung nào. Giả sử rằng loại int là 16-bit.

```c

typedef unsigned int UINT\_16;

struct s {

unsigned int b1:2; /\* Tuân thủ \*/

int b2:2; /\* Không tuân thủ - không cho phép sử dụng int \*/

UINT\_16 b3:2; /\* Tuân thủ - typedef chỉ định unsigned int \*/

signed long b4:2; /\* Không tuân thủ ngay cả khi long và int có cùng kích thước \*/

};

```

#### Quy tắc 6.2

Các trường bit được đặt tên chỉ có một bit không được là loại có dấu.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Lý do

Theo Tiêu chuẩn C99 Mục 6.2.6.2, một trường bit có dấu chỉ có một bit có một (một) bit dấu và không (không) bit giá trị. Trong bất kỳ cách biểu diễn số nguyên nào, 0 bit giá trị không thể chỉ định một giá trị có ý nghĩa.

Do đó, một trường bit có dấu chỉ có một bit có khả năng không hoạt động theo cách hữu ích và sự hiện diện của nó có thể chỉ ra sự nhầm lẫn của lập trình viên.

Mặc dù Tiêu chuẩn C90 không cung cấp nhiều chi tiết về cách biểu diễn các loại, nhưng các cân nhắc tương tự áp dụng như đối với C99.

Lưu ý: quy tắc này không áp dụng cho các trường bit không được đặt tên vì giá trị của chúng không thể được truy cập.

### Hằng số và hằng số số học

#### Quy tắc 7.1

Không được sử dụng hằng bát phân.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- [Koenig 9]

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Lý do

Các nhà phát triển viết các hằng số có chữ số 0 đứng đầu có thể mong đợi chúng được hiểu là các hằng số thập phân.

Lưu ý: quy tắc này không áp dụng cho các chuỗi thoát bát phân vì việc sử dụng ký tự \ đứng đầu có nghĩa là có ít khả năng gây nhầm lẫn hơn.

#Ngoại lệ

Hằng số nguyên 0 (viết dưới dạng một chữ số số), là một hằng bát phân nhưng được cho phép ngoại lệ với quy tắc này.

#Ví dụ

```c

extern uint16\_t code[ 10 ];

code[ 1 ] = 109; /\* Tuân thủ - thập phân 109 \*/

code[ 2 ] = 100; /\* Tuân thủ - thập phân 100 \*/

code[ 3 ] = 052; /\* Không tuân thủ - thập phân 42 \*/

code[ 4 ] = 071; /\* Không tuân thủ - thập phân 57 \*/

```

#### Quy tắc 7.2

Một hậu tố “u” hoặc “U” phải được áp dụng cho tất cả các hằng số nguyên được biểu diễn ở loại không dấu.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Mở rộng

Quy tắc này áp dụng cho:

- Các hằng số nguyên xuất hiện trong các biểu thức điều khiển của các chỉ thị tiền xử lý #if và #elif;

- Bất kỳ hằng số nguyên nào khác tồn tại sau khi tiền xử lý.

Lưu ý: trong quá trình tiền xử lý, loại của một hằng số nguyên được xác định theo cách tương tự như sau khi tiền xử lý ngoại trừ:

- Tất cả các loại số nguyên có dấu hoạt động như thể chúng là long (C90) hoặc intmax\_t (C99);

- Tất cả các loại số nguyên không dấu hoạt động như thể chúng là unsigned long (C90) hoặc uintmax\_t (C99).

#Lý do

Loại của một hằng số nguyên là một nguồn tiềm ẩn của sự nhầm lẫn, vì nó phụ thuộc vào một tổ hợp phức tạp của các yếu tố bao gồm:

- Độ lớn của hằng số;

- Kích thước được triển khai của các loại số nguyên;

- Sự hiện diện của bất kỳ hậu tố nào;

- Cơ số mà giá trị được biểu diễn (ví dụ: thập phân, bát phân hoặc thập lục phân).

Ví dụ, hằng số nguyên 40000 là loại signed int trong một môi trường 32-bit nhưng là loại signed long trong một môi trường 16-bit. Giá trị 0x8000 là loại unsigned int trong một môi trường 16-bit, nhưng là loại signed int trong một môi trường 32-bit.

#Lưu ý

- Bất kỳ giá trị nào có hậu tố “U” là loại không dấu;

- Một giá trị thập phân không có hậu tố nhỏ hơn 2^31 là loại có dấu.

Dưới đây là bản dịch tiếp theo của các quy tắc và ví dụ từ tài liệu MISRA Autosar từ tiếng Anh sang tiếng Việt:

---

Nhưng:

- Một giá trị thập lục phân không có hậu tố lớn hơn hoặc bằng 2^15 có thể là loại có dấu hoặc không dấu;

- Đối với C90, một giá trị thập phân không có hậu tố lớn hơn hoặc bằng 2^31 có thể là loại có dấu hoặc không dấu.

Tính có dấu của các hằng số nên rõ ràng. Nếu một hằng số là loại không dấu, việc áp dụng hậu tố "U" làm rõ rằng lập trình viên hiểu rằng hằng số này là không dấu.

Lưu ý: quy tắc này không phụ thuộc vào ngữ cảnh mà hằng số được sử dụng; sự nâng cấp và các chuyển đổi khác có thể được áp dụng cho hằng số không liên quan đến việc xác định tuân thủ quy tắc này.

#Ví dụ

Ví dụ sau đây giả định một máy có loại int 16-bit và loại long 32-bit. Nó cho thấy loại của mỗi hằng số nguyên được xác định theo Tiêu chuẩn. Hằng số nguyên 0x8000 không tuân thủ vì nó là loại không dấu nhưng không có hậu tố "U".

| Hằng số | Loại | Tuân thủ |

|---------|------|----------|

| 32767 | signed int | Tuân thủ |

| 0x7fff | signed int | Tuân thủ |

| 32768 | signed long | Tuân thủ |

| 32768u | unsigned int | Tuân thủ |

| 0x8000 | unsigned int | Không tuân thủ |

| 0x8000u | unsigned int | Tuân thủ |

#### Quy tắc 7.3

Ký tự chữ thường “l” không được sử dụng trong hậu tố của hằng số.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Lý do

Sử dụng hậu tố chữ hoa “L” loại bỏ khả năng nhầm lẫn giữa “1” (chữ số 1) và “l” (chữ cái “el”) khi khai báo các hằng số.

#Ví dụ

Lưu ý: các ví dụ chứa hậu tố long long chỉ áp dụng cho C99.

```c

const int64\_t a = 0L;

const int64\_t b = 0l; /\* Không tuân thủ \*/

const uint64\_t c = 0Lu;

const uint64\_t d = 0lU; /\* Không tuân thủ \*/

const uint64\_t e = 0ULL;

const uint64\_t f = 0Ull; /\* Không tuân thủ \*/

const int128\_t g = 0LL;

const int128\_t h = 0ll; /\* Không tuân thủ \*/

const float128\_t m = 1.2L;

const float128\_t n = 2.4l; /\* Không tuân thủ \*/

```

#### Quy tắc 7.4

Một chuỗi ký tự không được gán cho một đối tượng trừ khi loại của đối tượng là “pointer to const-qualified char”.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- C90 [Không xác định 12], C99 [Không xác định 14; Không xác định 30]

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Mở rộng

Không được cố gắng sửa đổi một chuỗi ký tự hoặc chuỗi ký tự rộng trực tiếp. Kết quả của toán tử địa chỉ &, áp dụng cho một chuỗi ký tự không được gán cho một đối tượng trừ khi loại của đối tượng đó là “pointer to array of const-qualified char”.

Các cân nhắc tương tự áp dụng cho các chuỗi ký tự rộng. Một chuỗi ký tự rộng không được gán cho một đối tượng trừ khi loại của đối tượng đó là “pointer to const-qualified wchar\_t”. Kết quả của toán tử địa chỉ &, áp dụng cho một chuỗi ký tự rộng không được gán cho một đối tượng trừ khi loại của đối tượng đó là “pointer to array of const-qualified wchar\_t”.

#Lý do

Bất kỳ cố gắng nào để sửa đổi một chuỗi ký tự dẫn đến hành vi không xác định. Ví dụ, một số triển khai có thể lưu trữ các chuỗi ký tự trong bộ nhớ chỉ đọc trong trường hợp đó một cố gắng sửa đổi chuỗi ký tự sẽ thất bại và có thể dẫn đến một ngoại lệ hoặc sự cố.

Quy tắc này, khi được áp dụng cùng với các quy tắc khác, ngăn chặn một chuỗi ký tự bị sửa đổi.

C99 không chỉ rõ liệu các chuỗi ký tự chia sẻ một kết thúc chung có được lưu trữ ở các vị trí bộ nhớ riêng biệt hay không. Do đó, ngay cả khi cố gắng sửa đổi một chuỗi ký tự dường như thành công, có thể một chuỗi ký tự khác có thể bị thay đổi vô tình.

#Ví dụ

Ví dụ sau cho thấy một cố gắng sửa đổi trực tiếp một chuỗi ký tự.

```c

"0123456789"[0] = '\*'; /\* Không tuân thủ \*/

```

Các ví dụ này cho thấy cách ngăn chặn sửa đổi các chuỗi ký tự gián tiếp.

```c

/\* Không tuân thủ - s không có const-qualified \*/

char \*s = "string";

/\* Tuân thủ - p có const-qualified; các định danh bổ sung được phép \*/

const volatile char \*p = "string";

extern void f1 ( char \*s1 );

extern void f2 ( const char \*s2 );

void g ( void )

{

f1 ( "string" ); /\* Không tuân thủ - tham số s1 không có const-qualified \*/

f2 ( "string" ); /\* Tuân thủ \*/

}

char \*name1 ( void )

{

return ( "MISRA" ); /\* Không tuân thủ - kiểu trả về không có const-qualified \*/

}

const char \*name2 ( void )

{

return ( "MISRA" ); /\* Tuân thủ \*/

}

```

Xem thêm

- Quy tắc 11.4, Quy tắc 11.8

### Khai báo và định nghĩa

#### Quy tắc 8.1

Các loại phải được chỉ định rõ rang.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90

#Lý do

Tiêu chuẩn C90 cho phép các loại được bỏ qua trong một số trường hợp, trong trường hợp đó loại int được chỉ định ngầm định. Các trường hợp mà một int ngầm định có thể được sử dụng bao gồm:

- Khai báo đối tượng;

- Khai báo tham số;

- Khai báo thành viên;

- Khai báo typedef;

- Các kiểu trả về của hàm.

Việc bỏ sót một loại rõ ràng có thể dẫn đến sự nhầm lẫn. Ví dụ, trong khai báo:

```c

extern void g ( char c, const k );

```

loại của k là const int trong khi const char có thể đã được mong đợi.

Dưới đây là bản dịch tiếp theo của các quy tắc và ví dụ từ tài liệu MISRA Autosar từ tiếng Anh sang tiếng Việt:

---

#Ví dụ

Các ví dụ sau đây cho thấy các khai báo đối tượng tuân thủ và không tuân thủ:

```c

extern x; /\* Không tuân thủ - loại int ngầm định \*/

extern int16\_t x; /\* Tuân thủ - loại rõ ràng \*/

const y; /\* Không tuân thủ - loại int ngầm định \*/

const int16\_t y; /\* Tuân thủ - loại rõ ràng \*/

```

Các ví dụ sau đây cho thấy các khai báo loại hàm tuân thủ và không tuân thủ:

```c

extern f ( void ); /\* Không tuân thủ - loại trả về int ngầm định \*/

extern int16\_t f ( void ); /\* Tuân thủ \*/

extern void g ( char c, const k ); /\* Không tuân thủ - loại int ngầm định cho tham số k \*/

extern void g ( char c, const int16\_t k ); /\* Tuân thủ \*/

```

Các ví dụ sau đây cho thấy các định nghĩa loại tuân thủ và không tuân thủ:

```c

typedef ( \*pfi ) ( void ); /\* Không tuân thủ - loại trả về int ngầm định \*/

typedef int16\_t ( \*pfi ) ( void ); /\* Tuân thủ \*/

typedef void ( \*pfv ) ( const x ); /\* Không tuân thủ - loại int ngầm định cho tham số x \*/

typedef void ( \*pfv ) ( int16\_t x ); /\* Tuân thủ \*/

```

Các ví dụ sau đây cho thấy các khai báo thành viên tuân thủ và không tuân thủ:

```c

struct str

{

int16\_t x; /\* Tuân thủ \*/

const y; /\* Không tuân thủ - loại int ngầm định cho thành viên y \*/

} s;

```

Xem thêm

- Quy tắc 8.2

#### Quy tắc 8.2

Các loại hàm phải ở dạng nguyên mẫu với các tham số có tên.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- C90 [Không xác định 22-25], C99 [Không xác định 36-39, 73, 79]

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Lý do

Phiên bản đầu của C, thường được gọi là K&R C, không cung cấp cơ chế để kiểm tra số lượng đối số hoặc loại của chúng so với các tham số tương ứng. Loại của một đối tượng hoặc hàm không phải được khai báo trong K&R C vì loại mặc định của một đối tượng và loại trả về mặc định của một hàm là int.

Tiêu chuẩn C90 đã giới thiệu các nguyên mẫu hàm, một dạng khai báo hàm trong đó các loại tham số được khai báo. Điều này cho phép kiểm tra các loại đối số so với các loại tham số. Nó cũng cho phép kiểm tra số lượng đối số ngoại trừ khi một nguyên mẫu hàm chỉ định rằng mong đợi một số lượng đối số thay đổi. Tiêu chuẩn C90 không yêu cầu sử dụng các nguyên mẫu hàm vì lý do tương thích ngược với mã hiện có. Vì cùng lý do, nó tiếp tục cho phép bỏ qua các loại trong trường hợp đó loại sẽ mặc định là int.

Tiêu chuẩn C99 đã loại bỏ loại int mặc định khỏi ngôn ngữ nhưng tiếp tục cho phép các loại hàm kiểu K&R trong đó không có phương tiện để cung cấp thông tin loại tham số trong khai báo và việc cung cấp thông tin loại tham số trong định nghĩa là tùy chọn.

Sự không khớp giữa số lượng đối số và tham số, các loại của chúng và loại trả về dự kiến và thực tế của một hàm cung cấp tiềm năng cho hành vi không xác định. Mục đích của quy tắc này cùng với Quy tắc 8.1 và Quy tắc 8.4 là để tránh hành vi không xác định này bằng cách yêu cầu các loại tham số và loại trả về của hàm được chỉ định rõ ràng. Quy tắc 17.3 đảm bảo rằng thông tin này có sẵn tại thời điểm gọi hàm, do đó yêu cầu trình biên dịch chẩn đoán bất kỳ sự không khớp nào được phát hiện.

Quy tắc này cũng yêu cầu rằng các tên được chỉ định cho tất cả các tham số trong khai báo. Các tên tham số có thể cung cấp thông tin hữu ích về giao diện hàm và một sự không khớp giữa khai báo và định nghĩa có thể chỉ ra một lỗi lập trình.

Lưu ý: Một danh sách tham số trống không hợp lệ trong một nguyên mẫu. Nếu một loại hàm không có tham số, dạng nguyên mẫu của nó sử dụng từ khóa void.

#Ví dụ

Ví dụ đầu tiên cho thấy các khai báo của một số hàm và các định nghĩa tương ứng cho một số hàm đó.

```c

/\* Tuân thủ \*/

extern int16\_t func1 ( int16\_t n );

/\* Không tuân thủ - tên tham số không được chỉ định \*/

extern void func2 ( int16\_t );

/\* Không tuân thủ - không ở dạng nguyên mẫu \*/

static int16\_t func3 ( );

/\* Tuân thủ - nguyên mẫu chỉ định 0 tham số \*/

static int16\_t func4 ( void );

/\* Tuân thủ \*/

int16\_t func1 ( int16\_t n )

{

return n;

}

/\* Không tuân thủ - danh sách định danh và khai báo kiểu cũ \*/

static int16\_t func3 ( vec, n )

int16\_t \*vec;

int16\_t n;

{

return vec[ n - 1 ];

}

```

Phần ví dụ này cho thấy việc áp dụng quy tắc cho các loại hàm khác ngoài các khai báo và định nghĩa hàm.

```c

/\* Không tuân thủ - không có nguyên mẫu \*/

int16\_t ( \*pf1 ) ( );

/\* Tuân thủ - nguyên mẫu chỉ định 0 tham số \*/

int16\_t ( \*pf1 ) ( void );

/\* Không tuân thủ - tên tham số không được chỉ định \*/

typedef int16\_t ( \*pf2\_t ) ( int16\_t );

/\* Tuân thủ \*/

typedef int16\_t ( \*pf3\_t ) ( int16\_t n );

```

Xem thêm

- Quy tắc 8.1, Quy tắc 8.4, Quy tắc 17.3

#### Quy tắc 8.3

Tất cả các khai báo của một đối tượng hoặc hàm phải sử dụng cùng tên và các định danh loại.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- C90 [Không xác định 10], C99 [Không xác định 14], [Koenig 59-62]

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Hệ thống

- Áp dụng cho: C90, C99

#Mở rộng

Các chỉ định lớp lưu trữ không nằm trong phạm vi của quy tắc này.

#Lý do

Sử dụng các loại và định danh nhất quán trên các khai báo của cùng một đối tượng hoặc hàm khuyến khích gõ mạnh hơn.

Chỉ định tên tham số trong các nguyên mẫu hàm cho phép định nghĩa hàm được kiểm tra tính nhất quán giao diện với các khai báo của nó.

#Ngoại lệ

Các phiên bản tương thích của cùng một loại cơ bản có thể được sử dụng thay thế lẫn nhau. Ví dụ, int, signed và signed int đều tương đương.

#Ví dụ

```c

extern void f ( signed int );

void f ( int ); /\* Tuân thủ - Ngoại lệ \*/

extern void g ( int \* const );

void g ( int \* ); /\* Không tuân thủ - định danh loại \*/

```

Lưu ý: tất cả các ví dụ trên đều không tuân thủ với Dir 4.6.

Dưới đây là phần dịch tiếp theo của các quy tắc và ví dụ từ tài liệu MISRA Autosar từ tiếng Anh sang tiếng Việt:

---

```c

extern int16\_t func ( int16\_t num, int16\_t den );

/\* Không tuân thủ - tên tham số không khớp \*/

int16\_t func ( int16\_t den, int16\_t num )

{

return num / den;

}

```

Trong ví dụ này, định nghĩa của `area` sử dụng một tên loại khác cho tham số `h` so với tên được sử dụng trong khai báo. Điều này không tuân thủ quy tắc mặc dù `width\_t` và `height\_t` là cùng một loại cơ bản.

```c

typedef uint16\_t width\_t;

typedef uint16\_t height\_t;

typedef uint32\_t area\_t;

extern area\_t area ( width\_t w, height\_t h );

area\_t area ( width\_t w, width\_t h )

{

return ( area\_t ) w \* h;

}

```

Quy tắc này không yêu cầu rằng khai báo con trỏ hàm sử dụng cùng tên với khai báo hàm. Vì vậy, ví dụ sau tuân thủ.

```c

extern void f1 ( int16\_t x );

extern void f2 ( int16\_t y );

void f ( bool\_t b )

{

void ( \*fp1 ) ( int16\_t z ) = b ? f1 : f2;

}

```

Xem thêm

- Quy tắc 8.4

#### Quy tắc 8.4

Một khai báo tương thích phải hiện diện khi một đối tượng hoặc hàm có liên kết bên ngoài được định nghĩa.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- C90 [Không xác định 24], C99 [Không xác định 39]

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Mở rộng

Một khai báo tương thích là một khai báo mà khai báo một loại tương thích cho đối tượng hoặc hàm đang được định nghĩa.

#Lý do

Nếu một khai báo cho một đối tượng hoặc hàm hiện diện khi đối tượng hoặc hàm đó được định nghĩa, một trình biên dịch phải kiểm tra rằng khai báo và định nghĩa là tương thích. Trong sự hiện diện của các nguyên mẫu hàm, như yêu cầu bởi Quy tắc 8.2, việc kiểm tra mở rộng đến số lượng và loại tham số hàm.

Phương pháp khuyến nghị để triển khai các khai báo của các đối tượng và hàm có liên kết bên ngoài là khai báo chúng trong một tệp tiêu đề, và sau đó bao gồm tệp tiêu đề đó trong tất cả các tệp mã cần chúng, bao gồm cả tệp định nghĩa chúng (Xem Quy tắc 8.5).

#Ví dụ

Trong các ví dụ này, không có khai báo hoặc định nghĩa của các đối tượng hoặc hàm nào khác ngoài những cái có trong mã.

```c

extern int16\_t count;

int16\_t count = 0; /\* Tuân thủ \*/

extern uint16\_t speed = 6000u; /\* Không tuân thủ - không có khai báo trước định nghĩa này \*/

uint8\_t pressure = 101u; /\* Không tuân thủ - không có khai báo trước định nghĩa này \*/

extern void func1 ( void );

extern void func2 ( int16\_t x, int16\_t y );

extern void func3 ( int16\_t x, int16\_t y );

void func1 ( void )

{

/\* Tuân thủ \*/

}

```

Định nghĩa không tuân thủ sau của `func3` cũng vi phạm Quy tắc 8.3.

```c

void func2 ( int16\_t x, int16\_t y )

{

/\* Tuân thủ \*/

}

void func3 ( int16\_t x, uint16\_t y )

{

/\* Không tuân thủ - loại tham số khác nhau \*/

}

void func4 ( void )

{

/\* Không tuân thủ - không có khai báo của func4 trước định nghĩa này \*/

}

static void func5 ( void )

{

/\* Tuân thủ - quy tắc không áp dụng cho các đối tượng/hàm có liên kết nội bộ \*/

}

```

Xem thêm

- Quy tắc 8.2, Quy tắc 8.3, Quy tắc 8.5, Quy tắc 17.3

#### Quy tắc 8.5

**Nội dung**

Một đối tượng hoặc hàm có liên kết bên ngoài phải được khai báo một lần trong một và chỉ một tệp.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- [Koenig 66]

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Hệ thống

- Áp dụng cho: C90, C99

#Mở rộng

Quy tắc này chỉ áp dụng cho các khai báo không định nghĩa.

#Lý do

Thông thường, một khai báo duy nhất sẽ được thực hiện trong một tệp tiêu đề sẽ được bao gồm trong bất kỳ đơn vị dịch nào mà định danh được định nghĩa hoặc sử dụng. Điều này đảm bảo sự nhất quán giữa:

- Khai báo và định nghĩa;

- Các khai báo trong các đơn vị dịch khác nhau.

Lưu ý: có thể có nhiều tệp tiêu đề trong một dự án, nhưng mỗi đối tượng hoặc hàm bên ngoài chỉ được khai báo trong một tệp tiêu đề.

#Ví dụ

```c

/\* featureX.h \*/

extern int16\_t a; /\* Khai báo a \*/

/\* file.c \*/

#include "featureX.h"

int16\_t a = 0; /\* Định nghĩa a \*/

```

Xem thêm

- Quy tắc 8.4

#### Quy tắc 8.6

**Nội dung**

Một định danh có liên kết bên ngoài phải có chính xác một định nghĩa bên ngoài.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- C90 [Không xác định 44], C99 [Không xác định 78], [Koenig 55, 63-65]

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Hệ thống

- Áp dụng cho: C90, C99

#Lý do

Hành vi là không xác định nếu một định danh được sử dụng mà có nhiều định nghĩa tồn tại (trong các tệp khác nhau) hoặc không có định nghĩa nào cả. Nhiều định nghĩa trong các tệp khác nhau không được phép bởi quy tắc này ngay cả khi các định nghĩa là giống nhau. Hành vi là không xác định nếu các khai báo khác nhau, hoặc khởi tạo định danh với các giá trị khác nhau.

Dưới đây là phần dịch tiếp theo của các quy tắc và ví dụ từ tài liệu MISRA Autosar từ tiếng Anh sang tiếng Việt:

---

#Ví dụ

Trong ví dụ này, đối tượng `i` được định nghĩa hai lần.

```c

/\* file1.c \*/

int16\_t i = 10;

/\* file2.c \*/

int16\_t i = 20; /\* Không tuân thủ - hai định nghĩa của i \*/

```

Trong ví dụ này, đối tượng `j` có một định nghĩa tạm thời và một định nghĩa bên ngoài.

```c

/\* file3.c \*/

int16\_t j; /\* Định nghĩa tạm thời \*/

int16\_t j = 1; /\* Tuân thủ - định nghĩa bên ngoài \*/

```

Ví dụ sau đây không tuân thủ vì đối tượng `k` có hai định nghĩa bên ngoài. Định nghĩa tạm thời trong `file4.c` trở thành một định nghĩa bên ngoài khi kết thúc đơn vị dịch.

```c

/\* file4.c \*/

int16\_t k; /\* Định nghĩa tạm thời - trở thành bên ngoài \*/

/\* file5.c \*/

int16\_t k = 0; /\* Định nghĩa bên ngoài \*/

```

#### Quy tắc 8.7

**Nội dung**

Các hàm và đối tượng không nên được định nghĩa với liên kết bên ngoài nếu chúng chỉ được tham chiếu trong một đơn vị dịch.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- [Koenig 56, 57]

- Phân loại: Tư vấn

- Phân tích: Quyết định được, Hệ thống

- Áp dụng cho: C90, C99

#Lý do

Hạn chế tầm nhìn của một đối tượng bằng cách cho nó liên kết nội bộ hoặc không có liên kết giảm khả năng nó có thể được truy cập vô ý. Tương tự, giảm tầm nhìn của một hàm bằng cách cho nó liên kết nội bộ giảm khả năng nó được gọi vô ý.

Tuân thủ quy tắc này cũng tránh bất kỳ khả năng nhầm lẫn nào giữa một định danh và một định danh giống hệt trong một đơn vị dịch khác hoặc một thư viện.

#### Quy tắc 8.8

**Nội dung**

Từ khóa lớp lưu trữ static phải được sử dụng trong tất cả các khai báo của các đối tượng và hàm có liên kết nội bộ.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Mở rộng

Vì các định nghĩa cũng là các khai báo, quy tắc này áp dụng tương tự cho các định nghĩa.

#Lý do

Tiêu chuẩn quy định rằng nếu một đối tượng hoặc hàm được khai báo với từ khóa lớp lưu trữ `extern` và một khai báo khác của đối tượng hoặc hàm đã có sẵn, liên kết là liên kết được chỉ định bởi khai báo trước đó. Điều này có thể gây nhầm lẫn vì có thể mong đợi rằng từ khóa lớp lưu trữ `extern` tạo ra liên kết bên ngoài. Do đó, từ khóa lớp lưu trữ `static` phải được áp dụng nhất quán cho các đối tượng và hàm có liên kết nội bộ.

#Ví dụ

```c

static int32\_t x = 0; /\* định nghĩa: liên kết nội bộ \*/

extern int32\_t x; /\* Không tuân thủ \*/

static int32\_t f ( void ); /\* khai báo: liên kết nội bộ \*/

int32\_t f ( void ) /\* Không tuân thủ \*/

{

return 1;

}

static int32\_t g ( void ); /\* khai báo: liên kết nội bộ \*/

extern int32\_t g ( void ) /\* Không tuân thủ \*/

{

return 1;

}

```

Quy tắc 8.9: Một đối tượng nên được định nghĩa ở phạm vi khối nếu định danh của nó chỉ xuất hiện trong một hàm duy nhất.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- Phân loại: Tư vấn

- Phân tích: Quyết định được, Hệ thống

- Áp dụng cho: C90, C99

#Lý do

Định nghĩa một đối tượng ở phạm vi khối giảm khả năng đối tượng có thể bị truy cập vô ý và làm rõ ý định rằng nó không nên được truy cập ở nơi khác.

Trong một hàm, liệu các đối tượng được định nghĩa ở khối ngoài cùng hay trong cùng chủ yếu là vấn đề phong cách.

#Ví dụ

Trong ví dụ tuân thủ này, `i` được khai báo ở phạm vi khối vì nó là một bộ đếm vòng lặp. Không cần các hàm khác trong cùng tệp sử dụng đối tượng này cho bất kỳ mục đích nào khác.

```c

void func ( void )

{

int32\_t i;

for ( i = 0; i < N; ++i )

{

}

}

```

Trong ví dụ tuân thủ này, hàm `count` theo dõi số lần nó đã được gọi và trả về số đó. Không có hàm nào khác cần biết chi tiết của việc triển khai `count` nên bộ đếm cuộc gọi được định nghĩa với phạm vi khối.

```c

uint32\_t count ( void )

{

static uint32\_t call\_count = 0;

++call\_count;

return call\_count;

}

```

Quy tắc 8.10: Một hàm nội tuyến phải được khai báo với từ khóa lớp lưu trữ static.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- C99 [Không xác định 20; Không xác định 67]

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C99

#Lý do

Nếu một hàm nội tuyến được khai báo với liên kết bên ngoài nhưng không được định nghĩa trong cùng đơn vị dịch, hành vi là không xác định.

Một cuộc gọi đến một hàm nội tuyến được khai báo với liên kết bên ngoài có thể gọi định nghĩa bên ngoài của hàm, hoặc nó có thể sử dụng định nghĩa nội tuyến. Mặc dù điều này không ảnh hưởng đến hành vi của hàm được gọi, nhưng nó có thể ảnh hưởng đến thời gian thực thi và do đó ảnh hưởng đến một chương trình thời gian thực.

Lưu ý: một hàm nội tuyến có thể được làm cho sẵn có cho nhiều đơn vị dịch bằng cách đặt định nghĩa của nó trong một tệp tiêu đề.

Xem thêm

- Quy tắc 5.9

Quy tắc 8.11: Khi một mảng có liên kết bên ngoài được khai báo, kích thước của nó nên được chỉ định rõ rang.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- Phân loại: Tư vấn

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Mở rộng

Quy tắc này chỉ áp dụng cho các khai báo không định nghĩa. Có thể định nghĩa một mảng và chỉ định kích thước của nó một cách ngầm định bằng cách khởi tạo.

#Lý do

Mặc dù có thể khai báo một mảng với loại không đầy đủ và truy cập các phần tử của nó, nhưng sẽ an toàn hơn khi làm điều đó khi kích thước của mảng có thể được xác định rõ ràng. Cung cấp thông tin kích thước cho mỗi khai báo cho phép chúng được kiểm tra tính nhất quán. Nó cũng có thể cho phép một trình kiểm tra tĩnh thực hiện một số phân tích giới hạn mảng mà không cần phân tích nhiều hơn một đơn vị dịch.

#Ví dụ

```c

extern int32\_t array1[ 10 ]; /\* Tuân thủ \*/

extern int32\_t array2[ ]; /\* Không tuân thủ \*/

```

Dưới đây là phần dịch tiếp theo của các quy tắc và ví dụ từ tài liệu MISRA Autosar từ tiếng Anh sang tiếng Việt:

---

Quy tắc 8.12: Trong danh sách liệt kê, giá trị của một hằng số liệt kê được chỉ định ngầm định phải là duy nhất

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Lý do

Một hằng số liệt kê được chỉ định ngầm định có giá trị lớn hơn 1 so với giá trị của hằng số trước đó. Nếu hằng số liệt kê đầu tiên được chỉ định ngầm định thì giá trị của nó là 0.

Một hằng số liệt kê được chỉ định rõ ràng có giá trị của biểu thức hằng số liên kết. Nếu hằng số được chỉ định ngầm định và rõ ràng được trộn lẫn trong một danh sách liệt kê, có thể xảy ra sự trùng lặp giá trị. Sự trùng lặp này có thể là không cố ý và có thể gây ra hành vi không mong muốn.

Quy tắc này yêu cầu bất kỳ sự trùng lặp nào của các hằng số liệt kê phải được chỉ định rõ ràng, do đó làm rõ ý định.

#Ví dụ

Trong ví dụ sau, các hằng số liệt kê `green` và `yellow` được gán giá trị giống nhau.

```c

/\* Không tuân thủ - yellow trùng lặp với giá trị của green được chỉ định ngầm định \*/

enum colour { red = 3, blue, green, yellow = 5 };

/\* Tuân thủ \*/

enum colour { red = 3, blue, green = 5, yellow = 5 };

```

Quy tắc 8.13: Một con trỏ nên trỏ tới một loại có định danh const bất cứ khi nào có thể.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- Phân loại: Tư vấn

- Phân tích: Không quyết định được, Hệ thống

- Áp dụng cho: C90, C99

#Mở rộng

Một con trỏ nên trỏ tới một loại có định danh const trừ khi:

- Nó được sử dụng để sửa đổi một đối tượng, hoặc

- Nó được sao chép sang một con trỏ khác trỏ tới một loại không có định danh const thông qua:

- Gán, hoặc

- Các hàm di chuyển hoặc sao chép bộ nhớ.

Vì mục đích đơn giản, quy tắc này được viết theo các con trỏ và các loại mà chúng trỏ tới. Tuy nhiên, nó áp dụng tương tự cho các mảng và các loại của các phần tử mà chúng chứa. Một mảng nên có các phần tử với loại có định danh const trừ khi:

- Bất kỳ phần tử nào của mảng được sửa đổi, hoặc

- Nó được sao chép sang một con trỏ trỏ tới một loại không có định danh const theo các phương pháp mô tả trên.

#Lý do

Quy tắc này khuyến khích thực hành tốt nhất bằng cách đảm bảo rằng các con trỏ không được sử dụng vô ý để sửa đổi các đối tượng. Về mặt lý thuyết, nó tương đương với việc khai báo ban đầu:

- Tất cả các mảng có các phần tử với loại có định danh const, và

- Tất cả các con trỏ trỏ tới các loại có định danh const.

và sau đó chỉ loại bỏ định danh const khi cần thiết để tuân thủ các ràng buộc của tiêu chuẩn ngôn ngữ.

#Ví dụ

Trong ví dụ không tuân thủ sau, `p` không được sử dụng để sửa đổi một đối tượng nhưng loại mà nó trỏ tới không có định danh const.

```c

uint16\_t f ( uint16\_t \*p )

{

return \*p;

}

```

Mã sẽ tuân thủ nếu hàm được định nghĩa với:

```c

uint16\_t g ( const uint16\_t \*p )

```

Ví dụ sau vi phạm một ràng buộc vì cố gắng sử dụng một con trỏ có định danh const để sửa đổi một đối tượng.

```c

void h ( const uint16\_t \*p )

{

\*p = 0;

}

```

Trong ví dụ sau, con trỏ `s` có định danh const nhưng loại mà nó trỏ tới thì không. Vì `s` không được sử dụng để sửa đổi một đối tượng, điều này là không tuân thủ.

```c

#include <string.h>

char last\_char ( char \* const s )

{

return s[ strlen ( s ) - 1u ];

}

```

Mã sẽ tuân thủ nếu hàm được định nghĩa với:

```c

char last\_char ( const char \* const s )

```

Trong ví dụ không tuân thủ này, không có phần tử nào của mảng `a` được sửa đổi nhưng loại phần tử không có định danh const.

```c

uint16\_t first ( uint16\_t a[ 5 ] )

{

return a[ 0 ];

}

```

Mã sẽ tuân thủ nếu hàm được định nghĩa với:

```c

uint16\_t first ( const uint16\_t a[ 5 ] )

```

Quy tắc 8.14: Không được sử dụng định danh loại restrict.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- C99 [Không xác định 65, 66]

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C99

#Lý do

Khi được sử dụng cẩn thận, định danh loại `restrict` có thể cải thiện hiệu suất của mã được tạo bởi trình biên dịch. Nó cũng có thể cho phép cải thiện phân tích tĩnh. Tuy nhiên, để sử dụng định danh loại `restrict`, lập trình viên phải chắc chắn rằng các vùng bộ nhớ được thao tác bởi hai hoặc nhiều con trỏ không chồng chéo.

Có một rủi ro đáng kể rằng trình biên dịch sẽ tạo ra mã không hoạt động như mong đợi nếu `restrict` được sử dụng không đúng cách.

#Ví dụ

Ví dụ sau đây tuân thủ vì các Hướng dẫn MISRA C không áp dụng cho các hàm thư viện tiêu chuẩn. Lập trình viên phải đảm bảo rằng các vùng được định nghĩa bởi `p`, `q` và `n` không chồng chéo.

```c

#include <string.h>

void f ( void )

{

/\* memcpy có các tham số được định danh restrict \*/

memcpy ( p, q, n );

}

```

Ví dụ sau đây không tuân thủ vì một hàm đã được định nghĩa sử dụng `restrict`.

```c

void user\_copy ( void \* restrict p, void \* restrict q, size\_t n )

{

}

```

### Khởi tạo

#### Quy tắc 9.1:

Giá trị của một đối tượng với thời gian lưu trữ tự động không được đọc trước khi nó được thiết lập.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- C90 [Không xác định 41], C99 [Không xác định 10, 17]

- Phân loại: Bắt buộc

- Phân tích: Không quyết định được, Hệ thống

- Áp dụng cho: C90, C99

#Mở rộng

Vì mục đích của quy tắc này, một phần tử mảng hoặc thành viên cấu trúc sẽ được coi là một đối tượng riêng biệt.

Dưới đây là phần dịch tiếp theo của các quy tắc và ví dụ từ tài liệu MISRA Autosar từ tiếng Anh sang tiếng Việt:

---

#Lý do

Theo Tiêu chuẩn, các đối tượng có thời gian lưu trữ tĩnh tự động được khởi tạo về 0 trừ khi được khởi tạo rõ ràng. Các đối tượng có thời gian lưu trữ tự động không được khởi tạo tự động và do đó có thể có các giá trị không xác định.

Lưu ý: đôi khi có thể việc khởi tạo rõ ràng của một đối tượng tự động bị bỏ qua. Điều này sẽ xảy ra khi một lệnh nhảy đến một nhãn sử dụng câu lệnh `goto` hoặc `switch` "bỏ qua" khai báo của đối tượng; đối tượng sẽ được khai báo như mong đợi nhưng bất kỳ khởi tạo rõ ràng nào sẽ bị bỏ qua.

#Ví dụ

```c

void f ( bool\_t b, uint16\_t \*p )

{

if ( b )

{

\*p = 3U;

}

}

void g ( void )

{

uint16\_t u;

f ( false, &u );

if ( u == 3U )

{

/\* Không tuân thủ - u chưa được gán giá trị \*/

}

}

```

Trong ví dụ không tuân thủ C99 sau, câu lệnh `goto` nhảy qua việc khởi tạo `x`.

Lưu ý: Ví dụ này cũng không tuân thủ Quy tắc 15.1.

```c

{

goto L1;

uint16\_t x = 10u;

L1:

x = x + 1u; /\* Không tuân thủ - x chưa được gán giá trị \*/

}

```

Xem thêm

- Quy tắc 15.1, Quy tắc 15.3

#### Quy tắc 9.2:

Bộ khởi tạo cho một tập hợp hoặc liên hiệp phải được bao quanh bởi dấu ngoặc nhọn.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- C90 [Không xác định 42], C99 [Không xác định 76, 77]

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Mở rộng

Quy tắc này áp dụng cho các bộ khởi tạo cho cả các đối tượng và đối tượng con. Một bộ khởi tạo dạng `{ 0 }`, đặt tất cả các giá trị về 0, có thể được sử dụng để khởi tạo các đối tượng con mà không cần các dấu ngoặc nhọn lồng nhau.

Lưu ý: quy tắc này không yêu cầu việc khởi tạo rõ ràng của các đối tượng hoặc đối tượng con.

#Lý do

Sử dụng dấu ngoặc nhọn để chỉ ra việc khởi tạo các đối tượng con cải thiện độ rõ ràng của mã và buộc các lập trình viên phải xem xét việc khởi tạo các phần tử trong các cấu trúc dữ liệu phức tạp như các mảng nhiều chiều hoặc các mảng cấu trúc.

#Ngoại lệ

1. Một mảng có thể được khởi tạo bằng cách sử dụng một chuỗi ký tự.

2. Một cấu trúc hoặc liên hiệp tự động có thể được khởi tạo bằng cách sử dụng một biểu thức với loại cấu trúc hoặc liên hiệp tương thích.

3. Một bộ khởi tạo được chỉ định có thể được sử dụng để khởi tạo một phần của đối tượng con.

#Ví dụ

Ba khởi tạo sau đây, được phép bởi Tiêu chuẩn, là tương đương. Dạng đầu tiên không được phép bởi quy tắc này vì nó không sử dụng dấu ngoặc nhọn để hiển thị rõ ràng việc khởi tạo các mảng con.

```c

int16\_t y[ 3 ][ 2 ] = { 1, 2, 0, 0, 5, 6 }; /\* Không tuân thủ \*/

int16\_t y[ 3 ][ 2 ] = { { 1, 2 }, { 0 }, { 5, 6 } }; /\* Tuân thủ \*/

int16\_t y[ 3 ][ 2 ] = { { 1, 2 }, { 0, 0 }, { 5, 6 } }; /\* Tuân thủ \*/

```

Trong ví dụ sau, việc khởi tạo `z1` tuân thủ theo Ngoại lệ 3 vì một bộ khởi tạo được chỉ định được sử dụng để khởi tạo đối tượng con `z1[ 1 ]`. Việc khởi tạo `z2` cũng tuân thủ vì lý do tương tự. Việc khởi tạo `z3` không tuân thủ vì một phần của đối tượng con `z3[ 1 ]` được khởi tạo bằng bộ khởi tạo theo vị trí nhưng không được bao quanh bởi dấu ngoặc nhọn. Việc khởi tạo `z4` tuân thủ vì một bộ khởi tạo được chỉ định được sử dụng để khởi tạo đối tượng con `z4[ 0 ]` và bộ khởi tạo cho đối tượng con `z4[ 1 ]` được bao quanh bởi dấu ngoặc nhọn.

```c

int16\_t z1[ 2 ][ 2 ] = { { 0 }, [ 1 ][ 1 ] = 1 }; /\* Tuân thủ \*/

int16\_t z2[ 2 ][ 2 ] = { { 0 },

[ 1 ][ 1 ] = 1, [ 1 ][ 0 ] = 0

}; /\* Tuân thủ \*/

int16\_t z3[ 2 ][ 2 ] = { { 0 }, [ 1 ][ 0 ] = 0, 1 }; /\* Không tuân thủ \*/

int16\_t z4[ 2 ][ 2 ] = { [ 0 ][ 1 ] = 0, { 0, 1 } }; /\* Tuân thủ \*/

```

Dưới đây là phần dịch tiếp theo của các quy tắc và ví dụ từ tài liệu MISRA Autosar từ tiếng Anh sang tiếng Việt:

---

#Ví dụ

Dòng đầu tiên trong ví dụ sau khởi tạo 3 mảng con mà không sử dụng các dấu ngoặc nhọn lồng nhau. Dòng thứ hai và thứ ba cho thấy các cách tương đương để viết cùng một bộ khởi tạo.

```c

float32\_t a[ 3 ][ 2 ] = { 0 }; /\* Tuân thủ \*/

float32\_t a[ 3 ][ 2 ] = { { 0 }, { 0 }, { 0 } }; /\* Tuân thủ \*/

float32\_t a[ 3 ][ 2 ] = { { 0.0f, 0.0f }, { 0.0f, 0.0f }, { 0.0f, 0.0f } }; /\* Tuân thủ \*/

```

Khởi tạo liên hiệp:

```c

union u1 {

int16\_t i;

float32\_t f;

} u = { 0 }; /\* Tuân thủ \*/

```

Khởi tạo cấu trúc:

```c

struct s1 {

uint16\_t len;

char buf[ 8 ];

} s[ 3 ] = {

{ 5u, { 'a', 'b', 'c', 'd', 'e', '\0', '\0', '\0' } },

{ 2u, { 0 } },

{ .len = 0u } /\* Tuân thủ - buf khởi tạo ngầm định \*/

}; /\* Tuân thủ - s[] khởi tạo đầy đủ \*/

```

#### Quy tắc 9.3:

Mảng không được khởi tạo một phần.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Mở rộng

Nếu bất kỳ phần tử nào của một đối tượng mảng hoặc đối tượng con được khởi tạo rõ ràng, thì toàn bộ đối tượng hoặc đối tượng con phải được khởi tạo rõ ràng.

#Lý do

Cung cấp một khởi tạo rõ ràng cho mỗi phần tử của một mảng làm rõ rằng mọi phần tử đã được xem xét.

#Ngoại lệ

1. Một bộ khởi tạo dạng `{ 0 }` có thể được sử dụng để khởi tạo rõ ràng tất cả các phần tử của một đối tượng mảng hoặc đối tượng con.

2. Một mảng có bộ khởi tạo chỉ bao gồm các bộ khởi tạo được chỉ định có thể được sử dụng, ví dụ như để thực hiện khởi tạo rời rạc.

3. Một mảng được khởi tạo bằng cách sử dụng một chuỗi ký tự không cần một bộ khởi tạo cho mỗi phần tử.

#Ví dụ

```c

/\* Tuân thủ \*/

int32\_t x[ 3 ] = { 0, 1, 2 };

/\* Không tuân thủ - y[ 2 ] được khởi tạo ngầm định \*/

int32\_t y[ 3 ] = { 0, 1 };

/\* Không tuân thủ - t[ 0 ] và t[ 3 ] được khởi tạo ngầm định \*/

float32\_t t[ 4 ] = { [ 1 ] = 1.0f, 2.0f };

/\* Tuân thủ - bộ khởi tạo được chỉ định cho ma trận rời rạc \*/

float32\_t z[ 50 ] = { [ 1 ] = 1.0f, [ 25 ] = 2.0f };

```

Trong ví dụ tuân thủ sau, mỗi phần tử của mảng `arr` được khởi tạo:

```c

float32\_t arr[ 3 ][ 2 ] =

{

{ 0.0f, 0.0f },

{ PI / 4.0f, -PI / 4.0f },

{ 0 } /\* khởi tạo tất cả các phần tử của đối tượng con mảng arr[ 2 ] \*/

};

```

Trong ví dụ sau, các phần tử mảng từ 6 đến 9 được khởi tạo ngầm định là '\0':

```c

char h[ 10 ] = "Hello"; /\* Tuân thủ theo Ngoại lệ 3 \*/

```

#### Quy tắc 9.4:

Một phần tử của một đối tượng không được khởi tạo nhiều hơn một lần.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C99

#Mở rộng

Quy tắc này áp dụng cho các bộ khởi tạo cho cả các đối tượng và đối tượng con.

Việc cung cấp các bộ khởi tạo được chỉ định trong C99 cho phép đặt tên các thành phần của một tập hợp (cấu trúc hoặc mảng) hoặc của một liên hiệp để được khởi tạo trong một danh sách khởi tạo và cho phép các phần tử của đối tượng được khởi tạo theo bất kỳ thứ tự nào bằng cách chỉ định các chỉ số mảng hoặc tên thành viên cấu trúc mà chúng áp dụng (các phần tử không có giá trị khởi tạo giả định giá trị mặc định cho các đối tượng không được khởi tạo).

#Lý do

Cần thận trọng khi sử dụng các bộ khởi tạo được chỉ định vì việc khởi tạo các phần tử của đối tượng có thể vô tình lặp lại dẫn đến việc ghi đè các phần tử đã được khởi tạo trước đó. Tiêu chuẩn C99 không chỉ rõ liệu các tác dụng phụ trong một bộ khởi tạo bị ghi đè có xảy ra hay không mặc dù điều này không được liệt kê trong Phụ lục J.

Để cho phép các mảng và cấu trúc rời rạc, chấp nhận chỉ khởi tạo những phần tử cần thiết cho ứng dụng.

#Ví dụ

Khởi tạo mảng:

```c

/\*

\* Hành vi yêu cầu sử dụng khởi tạo theo vị trí

\* Tuân thủ - a1 là -5, -4, -3, -2, -1

\*/

int16\_t a1[ 5 ] = { -5, -4, -3, -2, -1 };

/\*

\* Hành vi tương tự sử dụng các bộ khởi tạo được chỉ định

\* Tuân thủ - a2 là -5, -4, -3, -2, -1

\*/

int16\_t a2[ 5 ] = { [ 0 ] = -5, [ 1 ] = -4, [ 2 ] = -3, [ 3 ] = -2, [ 4 ] = -1 };

/\*

\* Các giá trị phần tử bộ khởi tạo được chỉ định lặp lại ghi đè các giá trị trước đó

\* Không tuân thủ - a3 là -5, -4, -2, 0, -1

\*/

int16\_t a3[ 5 ] = { [ 0 ] = -5, [ 1 ] = -4, [ 2 ] = -3, [ 2 ] = -2, [ 4 ] = -1 };

```

Trong ví dụ không tuân thủ sau, không rõ liệu tác dụng phụ có xảy ra hay không:

```c

uint16\_t \*p;

void f ( void )

{

uint16\_t a[ 2 ] = { [ 0 ] = \*p++, [ 0 ] = 1 };

}

```

Khởi tạo cấu trúc:

```c

struct mystruct

{

int32\_t a;

int32\_t b;

int32\_t c;

int32\_t d;

};

/\*

\* Hành vi yêu cầu sử dụng khởi tạo theo vị trí

\* Tuân thủ - s1 là 100, -1, 42, 999

\*/

struct mystruct s1 = { 100, -1, 42, 999 };

/\*

\* Hành vi tương tự sử dụng các bộ khởi tạo được chỉ định

\* Tuân thủ - s2 là 100, -1, 42, 999

\*/

struct mystruct s2 = { .a = 100, .b = -1, .c = 42, .d = 999 };

/\*

\* Các giá trị phần tử bộ khởi tạo được chỉ định lặp lại ghi đè các giá trị trước đó

\* Không tuân thủ - s3 là 42, -1, 0, 999

\*/

struct mystruct s3 = { .a = 100, .b = -1, .a = 42, .d = 999 };

```

#### Quy tắc 9.5:

Khi các bộ khởi tạo được chỉ định được sử dụng để khởi tạo một đối tượng mảng, kích thước của mảng phải được chỉ định rõ rang.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C99

#Mở rộng

Quy tắc này áp dụng tương tự cho một đối tượng con mảng là một thành viên mảng linh hoạt.

#Lý do

Nếu kích thước của một mảng không được chỉ định rõ ràng, nó được xác định bởi chỉ số cao nhất của bất kỳ phần tử nào được khởi tạo. Khi sử dụng các bộ khởi tạo được

chỉ định, có thể không luôn rõ ràng bộ khởi tạo nào có chỉ số cao nhất, đặc biệt khi bộ khởi tạo chứa một số lượng lớn các phần tử.

Để làm rõ ý định, kích thước của mảng phải được khai báo rõ ràng. Điều này cung cấp một số bảo vệ nếu, trong quá trình phát triển chương trình, các chỉ số của các phần tử được khởi tạo bị thay đổi vì đó là vi phạm ràng buộc (C99 Section 6.7.8) để khởi tạo một phần tử ngoài giới hạn của một mảng.

#Ví dụ

```c

/\* Không tuân thủ - có thể vô tình chỉ có một phần tử \*/

int a1[ ] = { [ 0 ] = 1 };

/\* Tuân thủ \*/

int a2[ 10 ] = { [ 0 ] = 1 };

```

### ### 8.10 Mô hình loại cơ bản

#### 8.10.1 Lý do

Các quy tắc trong phần này cùng nhau định nghĩa mô hình loại cơ bản và hạn chế hệ thống loại của C để:

1. Hỗ trợ hệ thống kiểm tra loại mạnh hơn;

2. Cung cấp cơ sở hợp lý để định nghĩa các quy tắc kiểm soát việc sử dụng các chuyển đổi loại ngầm định và rõ ràng;

3. Khuyến khích các thực hành mã hóa di động;

4. Giải quyết một số bất thường trong chuyển đổi loại được tìm thấy trong ISO C.

Mô hình loại cơ bản làm điều này bằng cách gán một loại cơ bản cho các đối tượng và biểu thức mà ISO C coi là loại số học. Ví dụ, việc thêm một `int` vào một `char` sẽ cho kết quả có loại cơ bản là ký tự thay vì loại `int` được thực hiện bởi sự thăng cấp số nguyên.

Lý do đầy đủ đằng sau mô hình loại cơ bản được đưa ra trong Phụ lục C với Phụ lục D cung cấp một định nghĩa toàn diện về loại cơ bản của bất kỳ biểu thức số học nào.

#### 8.10.2 Loại cơ bản

Loại cơ bản của một đối tượng hoặc biểu thức được định nghĩa bởi danh mục loại cơ bản và kích thước của nó. Danh mục loại cơ bản của một biểu thức phản ánh hành vi cơ bản của nó và có thể là:

- Cơ bản là Boolean;

- Cơ bản là ký tự;

- Cơ bản là enum;

- Cơ bản là có dấu;

- Cơ bản là không dấu;

- Cơ bản là số thực.

Lưu ý: mỗi loại enum được liệt kê là một loại enum cơ bản duy nhất được xác định là `enum<i>`. Điều này cho phép các loại enum khác nhau được xử lý như các loại riêng biệt, hỗ trợ hệ thống kiểm tra loại mạnh hơn. Một ngoại lệ là việc sử dụng một loại enum để định nghĩa một giá trị Boolean trong C90. Các loại như vậy được coi là có loại cơ bản là Boolean. Một ngoại lệ khác là việc sử dụng các enum ẩn danh như được định nghĩa trong Phụ lục D. Các enum ẩn danh là một cách để định nghĩa một tập hợp các số nguyên liên quan và được coi là có loại cơ bản là có dấu.

#### ### 8.10.3

Các quy tắc về loại cơ bản

Bảng ánh xạ các loại số nguyên chuẩn sang các danh mục loại cơ bản:

| Danh mục loại cơ bản | Các loại tương ứng |

|----------------------|--------------------|

| Boolean | `\_Bool` |

| Character | `char` |

| Signed | `signed char`, `signed short`, `signed int`, `signed long`, `signed long long` |

| Unsigned | `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, `unsigned long long` |

| Enum | `named enum` |

| Floating | `float`, `double`, `long double` |

Lưu ý: Các triển khai C99 có thể cung cấp các loại số nguyên mở rộng, mỗi loại sẽ được gán một vị trí phù hợp với cấp bậc và tính dấu của nó (xem C99 Section 6.3.1.1).

Các hạn chế được áp đặt bởi các quy tắc trong phần này cũng áp dụng cho các toán tử gán hợp chất (ví dụ: `^=`) vì chúng tương đương với việc gán kết quả thu được từ việc sử dụng một trong các toán tử số học, bitwise hoặc shift. Ví dụ:

```c

u8a += u8b + 1U;

```

tương đương với:

```c

u8a = u8a + ( u8b + 1U );

```

#### Quy tắc 10.1:

Các toán hạng không được có loại cơ bản không phù hợp.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- C90 [Không xác định 23; Triển khai 14, 17, 19, 32]

- C99 [Không xác định 13, 49; Triển khai J3.4(2, 5), J3.5(5), J3.9(6)]

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Mở rộng

Trong bảng dưới đây, một số trong một ô chỉ ra nơi áp dụng hạn chế đối với việc sử dụng một loại cơ bản như một toán hạng cho một toán tử. Các số này tương ứng với các đoạn trong phần Lý do bên dưới và chỉ ra lý do tại sao mỗi hạn chế được áp đặt.

| Toán tử | Toán hạng | Danh mục loại cơ bản của toán hạng số học |

|---------|-----------|-------------------------------------------|

| [ ] | integer | 3, 4, 1 |

| + (unary) | | 3, 4, 5 |

| - (unary) | | 3, 4, 5, 8 |

| + - | either | 3, 5 |

| \* / | either | 3, 4, 5 |

| % | either | 3, 4, 5, 1 |

| < > <= >= | either | 3 |

| == != | either | |

| ! && || | any | 2, 2, 2, 2, 2 |

| << >> left | | 3, 4, 5, 6, 6, 1 |

| << >> right | | 3, 4, 7, 7, 1 |

| ~ & | ^ | any | 3, 4, 5, 6, 6, 1 |

| ?: 1st | any | 2, 2, 2, 2, 2 |

| ?: 2nd and 3rd | | |

Theo quy tắc này, các toán tử ++ và -- hoạt động giống như các toán tử + và - nhị phân.

Các quy tắc khác đặt ra các hạn chế bổ sung về sự kết hợp của các loại cơ bản có thể được sử dụng trong một biểu thức.

Lý do

1. Việc sử dụng một biểu thức có loại cơ bản là số thực cho các toán hạng này là vi phạm ràng buộc.

2. Một biểu thức có loại cơ bản là Boolean nên luôn được sử dụng khi một toán hạng được diễn giải như một giá trị Boolean.

3. Một toán hạng có loại cơ bản là Boolean không nên được sử dụng khi một toán hạng được diễn giải như một giá trị số.

4. Một toán hạng có loại cơ bản là ký tự không nên được sử dụng khi một toán hạng được diễn giải như một giá trị số. Các giá trị số của dữ liệu ký tự là do triển khai xác định.

5. Một toán hạng có loại cơ bản là enum không nên được sử dụng trong một phép toán số học vì một đối tượng enum sử dụng một loại số nguyên do triển khai xác định. Một phép toán liên quan đến một đối tượng enum do đó có thể tạo ra một kết quả với loại không mong đợi. Lưu ý rằng một hằng số liệt kê từ một enum ẩn danh có loại cơ bản là có dấu.

6. Các phép toán dịch chuyển và bitwise chỉ nên được thực hiện trên các toán hạng có loại cơ bản là không dấu. Giá trị số thu được từ việc sử dụng chúng trên các loại có dấu là do triển khai xác định.

7. Toán hạng bên phải của một toán tử dịch chuyển nên có loại cơ bản là không dấu để đảm bảo rằng không có hành vi không xác định xảy ra từ một dịch chuyển âm.

8. Một toán hạng có loại cơ bản là không dấu không nên được sử dụng làm toán hạng cho toán tử trừ đơn, vì tính dấu của kết quả được xác định bởi kích thước đã triển khai của `int`.

#Ngoại lệ

Một biểu thức hằng số số nguyên không âm có loại cơ bản là có dấu có thể được sử dụng làm toán hạng bên phải cho một toán tử dịch chuyển.

#Ví dụ

```c

enum enuma { a1, a2, a3 } ena, enb; /\* Cơ bản là enum<enuma> \*/

enum { K1 = 1, K2 = 2 }; /\* Cơ bản là có dấu \*/

```

Các ví dụ sau đây không tuân thủ. Các nhận xét đề cập đến mục lý do được đánh số dẫn đến việc không tuân thủ.

```c

f32a & 2U /\* Lý do 1 - vi phạm ràng buộc \*/

f32a << 2 /\* Lý do 1 - vi phạm ràng buộc \*/

cha && bla /\* Lý do 2 - loại char được sử dụng làm giá trị Boolean \*/

ena ? a1 : a2 /\* Lý do 2 - loại enum được sử dụng làm giá trị Boolean \*/

s8a && bla /\* Lý do 2 - loại có dấu được sử dụng làm giá trị Boolean \*/

u8a ? a1 : a2 /\* Lý do 2 - loại không dấu được sử dụng làm giá trị Boolean \*/

f32a && bla /\* Lý do 2 - loại số thực được sử dụng làm giá trị Boolean \*/

bla \* blb /\* Lý do 3 - Boolean được sử dụng làm giá trị số \*/

bla > blb /\* Lý do 3 - Boolean được sử dụng làm giá trị số \*/

cha & chb /\* Lý do 4 - loại char được sử dụng làm giá trị số \*/

cha << 1 /\* Lý do 4 - loại char được sử dụng làm giá trị số \*/

ena-- /\* Lý do 5 - loại enum được sử dụng trong phép toán số học \*/

ena \* a1 /\* Lý do 5 - loại enum được sử dụng trong phép toán số học \*/

s8a & 2 /\* Lý do 6 - phép toán bitwise trên loại có dấu \*/

50 << 3U /\* Lý do 6 - phép toán dịch chuyển trên loại có dấu \*/

u8a << s8a /\* Lý do 7 - độ lớn dịch chuyển sử dụng loại có dấu \*/

u8a << -1 /\* Lý do 7 - độ lớn dịch chuyển sử dụng loại có dấu \*/

-u8a /\* Lý do 8 - trừ đơn trên loại không dấu \*/

```

Ví dụ sau không tuân thủ quy tắc này và cũng vi phạm Quy tắc 10.3:

```c

ena += a1 /\* Lý do 5 - loại enum được sử dụng trong phép toán số học \*/

```

Các ví dụ sau tuân thủ:

```c

bla && blb

bla ? u8a : u8b

cha - chb

cha > chb

ena > a1

K1 \* s8a /\* Tuân thủ vì K1 từ enum ẩn danh \*/

s8a + s16b

-( s8a ) \* s8b

s8a > 0

--s16b

u8a + u16b

u8a & 2U

u8a > 0U

u8a << 2U

u8a << 1 /\* Tuân thủ theo ngoại lệ \*/

f32a + f32b

f32a > 0.0

```

Ví dụ sau tuân thủ quy tắc này nhưng vi phạm Quy tắc 10.2:

```c

cha + chb

```

Xem thêm

- Quy tắc 10.2

#### ### Quy tắc 10.2:

Các biểu thức có loại cơ bản là ký tự không được sử dụng không phù hợp trong các phép toán cộng và trừ.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Mở rộng

Các sử dụng phù hợp là:

1. Đối với toán tử `+`, một toán hạng phải có loại cơ bản là ký tự và toán hạng còn lại phải có loại cơ bản là có dấu hoặc không dấu. Kết quả của phép toán có loại cơ bản là ký tự.

2. Đối với toán tử `-`, toán hạng đầu tiên phải có loại cơ bản là ký tự và toán hạng thứ hai phải có loại cơ bản là có dấu, không dấu hoặc ký tự. Nếu cả hai toán hạng có loại cơ bản là ký tự thì kết quả có loại chuẩn (thường là int trong trường hợp này), nếu không thì kết quả có loại cơ bản là ký tự.

#Lý do

Các biểu thức với loại cơ bản là ký tự (dữ liệu ký tự) không nên được sử dụng như số học vì dữ liệu không đại diện cho các giá trị số.

Các sử dụng trên được phép vì chúng cho phép thao tác hợp lý với dữ liệu ký tự. Ví dụ:

- Phép trừ hai toán hạng có loại cơ bản là ký tự có thể được sử dụng để chuyển đổi giữa các chữ số trong phạm vi '0' đến '9' và giá trị thứ tự tương ứng;

- Phép cộng một loại cơ bản là ký tự và một loại không dấu có thể được sử dụng để chuyển đổi giá trị thứ tự thành chữ số tương ứng trong phạm vi '0' đến '9';

- Phép trừ một loại không dấu từ một loại ký tự có thể được sử dụng để chuyển đổi ký tự từ chữ thường sang chữ hoa.

#Ví dụ

Các ví dụ sau đây tuân thủ:

```c

'0' + u8a /\* Chuyển u8a thành chữ số \*/

s8a + '0' /\* Chuyển s8a thành chữ số \*/

cha - '0' /\* Chuyển cha thành giá trị thứ tự \*/

'0' - s8a /\* Chuyển -s8a thành chữ số \*/

```

Các ví dụ sau đây không tuân thủ:

```c

s16a - 'a'

'0' + f32a

cha + ':'

cha - ena

```

#### Quy tắc 10.3:

Giá trị của một biểu thức không được gán cho một đối tượng có loại cơ bản hẹp hơn hoặc có danh mục loại cơ bản khác

- C90 [Không xác định 15; Triển khai 16]

- C99 [Không xác định 15, 16; Triển khai 3.5(4)]

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Mở rộng

Các hoạt động sau được bao phủ bởi quy tắc này:

1. Gán như được định nghĩa trong Từ điển thuật ngữ;

2. Việc chuyển đổi biểu thức hằng số trong nhãn của câu lệnh `switch` sang loại được thăng cấp của biểu thức điều khiển.

#Lý do

Ngôn ngữ C cho phép lập trình viên tự do đáng kể và sẽ cho phép các phép gán giữa các loại số học khác nhau được thực hiện tự động. Tuy nhiên, việc sử dụng các chuyển đổi ngầm định này có thể dẫn đến kết quả không mong muốn, có khả năng làm mất giá trị, dấu hoặc độ chính xác. Các chi tiết bổ sung về các vấn đề với hệ thống loại của C có thể được tìm thấy trong Phụ lục C.

Việc sử dụng kiểu mạnh hơn, như được áp đặt bởi mô hình loại cơ bản của MISRA, giảm khả năng xảy ra các vấn đề này.

#Ngoại lệ

1. Một biểu thức hằng số số nguyên không âm có loại cơ bản là có dấu có thể được gán cho một đối tượng có loại cơ bản là không dấu nếu giá trị của nó có thể được biểu diễn trong loại đó.

2. Bộ khởi tạo `{ 0 }` có thể được sử dụng để khởi tạo một loại tập hợp hoặc liên hiệp.

#Ví dụ

```c

enum enuma { A1, A2, A3 } ena;

enum enumb { B1, B2, B3 } enb;

enum { K1=1, K2=128 };

```

Các ví dụ sau đây tuân thủ:

```c

uint8\_t u8a = 0; /\* Theo ngoại lệ \*/

bool\_t flag = ( bool\_t ) 0;

bool\_t set = true; /\* true là về cơ bản là Boolean \*/

bool\_t get = ( u8b > u8c );

ena = A1;

s8a = K1; /\* Giá trị hằng phù hợp \*/

u8a = 2; /\* Theo ngoại lệ \*/

u8a = 2 \* 24; /\* Theo ngoại lệ \*/

cha += 1; /\* cha = cha + 1 gán ký tự cho ký tự \*/

pu8a = pu8b; /\* Cùng loại cơ bản \*/

u8a = u8b + u8c + u8d; /\* Cùng loại cơ bản \*/

u8a = ( uint8\_t ) s8a; /\* Cast cho cùng loại cơ bản \*/

u32a = u16a; /\* Gán cho loại rộng hơn \*/

u32a = 2U + 125U; /\* Gán cho loại rộng hơn \*/

use\_uint16 ( u8a ); /\* Gán cho loại rộng hơn \*/

use\_uint16 ( u8a + u16b ); /\* Gán cho cùng loại cơ bản \*/

```

Các ví dụ sau không tuân thủ vì chúng có các danh mục loại cơ bản khác nhau:

```c

uint8\_t u8a = 1.0f; /\* không dấu và số thực \*/

bool\_t bla = 0; /\* boolean và có dấu \*/

cha = 7; /\* ký tự và có dấu \*/

u8a = 'a'; /\* không dấu và ký tự \*/

u8b = 1 - 2; /\* không dấu và có dấu \*/

u8c += 'a'; /\* u8c = u8c + 'a' gán ký tự cho không dấu \*/

use\_uint32 ( s32a ); /\* có dấu và không dấu \*/

```

Các ví dụ sau không tuân thủ vì chúng chứa các phép gán cho loại hẹp hơn:

```c

s8a = K2; /\* Giá trị hằng không phù hợp \*/

u16a = u32a; /\* uint32\_t sang uint16\_t \*/

use\_uint16 ( u32a ); /\* uint32\_t sang uint16\_t \*/

uint8\_t foo1 ( uint16\_t x )

{

return x; /\* uint16\_t sang uint8\_t \*/

}

```

Xem thêm

- Quy tắc 10.4, Quy tắc 10.5, Quy tắc 10.6

#### ### Quy tắc 10.4:

Cả hai toán hạng của một toán tử trong đó các chuyển đổi số học thông thường được thực hiện phải có cùng danh mục loại cơ bản.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- C90 [Triển khai 21], C99 [Triển khai 3.6(4)]

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Mở rộng

Quy tắc này áp dụng cho các toán tử được mô tả trong các chuyển đổi số học thông thường (xem C90 Section 6.2.1.5, C99 Section 6.3.1.8). Điều này bao gồm tất cả các toán tử nhị phân, trừ các toán tử dịch chuyển, logic `&&`, logic `||` và dấu phẩy. Ngoài ra, toán hạng thứ hai và thứ ba của toán tử ba ngôi cũng được bao phủ bởi quy tắc này.

Lưu ý: các toán tử tăng và giảm không được bao phủ bởi quy tắc này.

#Lý do

Ngôn ngữ C cho phép lập trình viên tự do đáng kể và sẽ cho phép các chuyển đổi giữa các loại số học khác nhau được thực hiện tự động. Tuy nhiên, việc sử dụng các chuyển đổi ngầm định này có thể dẫn đến kết quả không mong muốn, có khả năng làm mất giá trị, dấu hoặc độ chính xác. Các chi tiết bổ sung về các vấn đề với hệ thống loại của C có thể được tìm thấy trong Phụ lục C.

Việc sử dụng kiểu mạnh hơn, như được áp đặt bởi mô hình loại cơ bản của MISRA, cho phép hạn chế các chuyển đổi ngầm định vào những chuyển đổi mà sau đó nên tạo ra kết quả mong đợi của lập trình viên.

### Quy tắc 10.4: Ngoại lệ

Các ngoại lệ sau đây được phép để cho phép một hình thức thao tác ký tự phổ biến được sử dụng:

1. Các toán tử + và += nhị phân có thể có một toán hạng có loại cơ bản là ký tự và toán hạng còn lại có loại cơ bản là có dấu hoặc không dấu.

2. Các toán tử - và -= nhị phân có thể có toán hạng bên trái có loại cơ bản là ký tự và toán hạng bên phải có loại cơ bản là có dấu hoặc không dấu.

#Ví dụ

```c

enum enuma { A1, A2, A3 } ena;

enum enumb { B1, B2, B3 } enb;

```

Các ví dụ sau đây tuân thủ vì chúng có cùng danh mục loại cơ bản:

```c

ena > A1

u8a + u16b

```

Ví dụ sau tuân thủ theo ngoại lệ 1:

```c

cha += u8a

```

Ví dụ sau không tuân thủ quy tắc này và cũng vi phạm Quy tắc 10.3:

```c

s8a += u8a /\* có dấu và không dấu \*/

```

Các ví dụ sau không tuân thủ:

```c

u8b + 2 /\* không dấu và có dấu \*/

enb > A1 /\* enum<enumb> và enum<enuma> \*/

ena == enb /\* enum<enuma> và enum<enumb> \*/

u8a += cha /\* không dấu và ký tự \*/

```

Xem thêm

- Quy tắc 10.3, Quy tắc 10.7

#### ### Quy tắc 10.5:

Giá trị của một biểu thức không nên được ép kiểu thành loại cơ bản không phù hợp.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- Phân loại: Tư vấn

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Mở rộng

Các phép ép kiểu nên tránh được hiển thị trong bảng sau, nơi các giá trị được ép kiểu (chuyển đổi rõ ràng) sang danh mục loại cơ bản của cột đầu tiên.

| Danh mục loại cơ bản | Boolean | Character | Enum | Signed | Unsigned | Floating |

|----------------------|---------|-----------|------|--------|----------|----------|

| Boolean | Avoid | Avoid | Avoid | Avoid | Avoid | |

| Character | Avoid | | | | | Avoid |

| Enum | Avoid | | Avoid | Avoid | Avoid | Avoid |

| Signed | Avoid | | | | | |

| Unsigned | Avoid | | | | | |

| Floating | Avoid | | | | | |

Lưu ý: Một loại enum có thể được ép kiểu sang một loại enum khác với điều kiện phép ép kiểu đó là cùng loại enum cơ bản. Các phép ép kiểu như vậy là dư thừa.

#Lý do

Một phép ép kiểu rõ ràng có thể được giới thiệu vì lý do chức năng hợp pháp, ví dụ:

- Để thay đổi loại trong đó một phép toán số học được thực hiện;

- Để cắt ngắn một giá trị một cách có chủ ý;

- Để làm cho một chuyển đổi loại rõ ràng vì lợi ích của sự rõ ràng.

Tuy nhiên, một số phép ép kiểu rõ ràng được coi là không phù hợp:

- Trong C99, kết quả của một phép ép kiểu hoặc phép gán sang `\_Bool` luôn là 0 hoặc 1. Điều này không nhất thiết đúng khi ép kiểu sang một loại khác được xác định là về cơ bản là Boolean;

- Một phép ép kiểu sang loại cơ bản là enum có thể dẫn đến một giá trị không nằm trong tập hợp các hằng số enum cho loại đó;

- Một phép ép kiểu từ về cơ bản là Boolean sang bất kỳ loại nào khác không có ý nghĩa;

- Chuyển đổi giữa các loại số thực và ký tự không có ý nghĩa vì không có ánh xạ chính xác giữa hai biểu diễn này.

#Ngoại lệ

Một biểu thức hằng số số nguyên với giá trị 0 hoặc 1 của bất kỳ dấu nào có thể được ép kiểu sang một loại được xác định là về cơ bản là Boolean. Điều này cho phép thực hiện các mô hình Boolean không phải C99.

#Ví dụ

```c

(bool\_t) false /\* Tuân thủ - 'false' trong C99 là về cơ bản là Boolean \*/

(int32\_t) 3U /\* Tuân thủ \*/

(bool\_t) 0 /\* Tuân thủ - theo ngoại lệ \*/

(bool\_t) 3U /\* Không tuân thủ \*/

(int32\_t) ena /\* Tuân thủ \*/

(enum enuma) 3 /\* Không tuân thủ \*/

(char) enc /\* Tuân thủ \*/

```

Xem thêm

- Quy tắc 10.3, Quy tắc 10.8

#### ### 8.10.3

Các toán tử và biểu thức hợp chất

Một số mối quan ngại được đề cập trong Phụ lục C có thể được tránh bằng cách hạn chế các chuyển đổi ngầm định và rõ ràng có thể áp dụng cho các biểu thức không tầm thường. Bao gồm:

- Sự nhầm lẫn về loại trong đó các biểu thức số nguyên được đánh giá, vì điều này phụ thuộc vào loại của các toán hạng sau bất kỳ sự thăng cấp số nguyên nào. Loại của kết quả của một phép toán số học phụ thuộc vào kích thước được triển khai của int;

- Sự hiểu lầm phổ biến trong số các lập trình viên rằng loại trong đó một phép tính được thực hiện bị ảnh hưởng bởi loại mà kết quả được gán hoặc ép kiểu. Sự kỳ vọng sai này có thể dẫn đến kết quả không mong muốn.

Ngoài các quy tắc trước đó, mô hình loại cơ bản áp đặt thêm các hạn chế đối với các biểu thức có toán hạng là các biểu thức hợp chất, như được định nghĩa dưới đây.

Các toán tử sau đây được định nghĩa là toán tử hợp chất trong tài liệu này:

- Toán tử nhân (\*, /, %)

- Toán tử cộng (nhị phân +, nhị phân -)

- Toán tử bitwise (&, |, ^)

- Toán tử dịch chuyển (<<, >>)

- Toán tử điều kiện (?:) nếu toán hạng thứ hai hoặc thứ ba là biểu thức hợp chất

Một phép gán hợp chất tương đương với một phép gán kết quả của toán tử hợp chất tương ứng của nó.

Một biểu thức hợp chất được định nghĩa trong tài liệu này là một biểu thức không hằng số là kết quả trực tiếp của một toán tử hợp chất.

Lưu ý:

- Kết quả của một toán tử gán hợp chất không phải là biểu thức hợp chất;

- Một biểu thức hợp chất được đặt trong dấu ngoặc cũng là một biểu thức hợp chất;

- Một biểu thức hằng số không phải là biểu thức hợp chất.

#### ### Quy tắc 10.6:

Giá trị của một biểu thức hợp chất không được gán cho một đối tượng có loại cơ bản rộng hơn.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Mở rộng

Quy tắc này bao gồm các hoạt động gán được mô tả trong Quy tắc 10.3.

#Lý do

Lý do được mô tả trong phần giới thiệu về các toán tử và biểu thức hợp chất (xem Phần 8.10.3).

#Ví dụ

Các ví dụ sau đây tuân thủ:

```c

u16c = u16a + u16b; /\* Cùng loại cơ bản \*/

u32a = (uint32\_t) u16a + u16b; /\* Ép kiểu gây ra phép cộng trong uint32\_t \*/

```

Các ví dụ sau đây không tuân thủ:

```c

u32a = u16a + u16b; /\* Chuyển đổi ngầm định khi gán \*/

use\_uint32(u16a + u16b); /\* Chuyển đổi ngầm định của đối số hàm \*/

```

Xem thêm

- Quy tắc 10.3, Quy tắc 10.7, Phần 8.10.3

### Quy tắc 10.7: Nếu một biểu thức hợp chất được sử dụng làm một toán hạng của một toán tử trong đó các chuyển đổi số học thông thường được thực hiện, thì toán hạng còn lại không được có loại cơ bản rộng hơn

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Lý do

Lý do được mô tả trong phần giới thiệu về các toán tử và biểu thức hợp chất (xem Phần 8.10.3).

Hạn chế các chuyển đổi ngầm định trên các biểu thức hợp chất có nghĩa là các chuỗi phép toán số học trong một biểu thức phải được thực hiện trong cùng một loại cơ bản. Điều này giảm bớt sự nhầm lẫn của lập trình viên.

#Ví dụ

Các ví dụ sau đây tuân thủ:

```c

u32a \* u16a + u16b /\* Không có chuyển đổi hợp chất \*/

(u32a \* u16a) + u16b /\* Không có chuyển đổi hợp chất \*/

u32a \* ((uint32\_t) u16a + u16b) /\* Cả hai toán hạng của \* có cùng loại cơ bản \*/

u32a += (u32b + u16b) /\* Không có chuyển đổi hợp chất \*/

```

Các ví dụ sau đây không tuân thủ:

```c

u32a \* (u16a + u16b) /\* Chuyển đổi ngầm định của (u16a + u16b) \*/

u32a += (u16a + u16b) /\* Chuyển đổi ngầm định của (u16a + u16b) \*/

```

Xem thêm

- Quy tắc 10.4, Quy tắc 10.6, Phần 8.10.3

#### ### Quy tắc 10.8:

Giá trị của một biểu thức hợp chất không được ép kiểu sang một danh mục loại cơ bản khác hoặc loại cơ bản rộng hơn

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Lý do

Lý do được mô tả trong phần giới thiệu về các toán tử và biểu thức hợp chất (xem Phần 8.10.3).

Ép kiểu sang loại rộng hơn không được phép vì kết quả có thể khác nhau giữa các triển khai. Xem xét ví dụ sau:

```c

(uint32\_t) (u16a + u16b);

```

Trên một máy 16-bit, phép cộng sẽ được thực hiện trong 16-bit với kết quả được làm tròn modulo-2 trước khi được ép kiểu sang 32-bit. Tuy nhiên, trên một máy 32-bit, phép cộng sẽ diễn ra trong 32-bit và sẽ giữ lại các bit cao hơn mà sẽ bị mất trên máy 16-bit.

Ép kiểu sang loại hẹp hơn cùng danh mục loại cơ bản là chấp nhận được vì việc cắt ngắn rõ ràng của kết quả luôn dẫn đến cùng một mất mát thông tin.

#Ví dụ

Các ví dụ sau đây tuân thủ:

```c

(uint16\_t) (u32a + u32b) /\* Tuân thủ \*/

(uint16\_t) s32a /\* Tuân thủ - s32a không phải hợp chất \*/

```

Các ví dụ sau đây không tuân thủ:

```c

(uint16\_t) (s32a + s32b) /\* Không tuân thủ - danh mục loại cơ bản khác \*/

(uint32\_t) (u16a + u16b) /\* Không tuân thủ - ép kiểu sang loại cơ bản rộng hơn \*/

```

Xem thêm

- Quy tắc 10.5, Phần 8.10.3

### Chuyển đổi loại con trỏ

Các loại con trỏ có thể được phân loại như sau:

- Con trỏ đến đối tượng;

- Con trỏ đến hàm;

- Con trỏ đến không hoàn chỉnh;

- Con trỏ đến void;

- Một hằng số con trỏ null, tức là giá trị 0, có thể được ép kiểu sang void \*.

Các chuyển đổi duy nhất liên quan đến con trỏ được phép bởi Tiêu chuẩn là:

- Chuyển đổi từ một loại con trỏ sang void;

- Chuyển đổi từ một loại con trỏ sang một loại số học;

- Chuyển đổi từ một loại số học sang một loại con trỏ;

- Chuyển đổi từ một loại con trỏ sang một loại con trỏ khác.

Mặc dù được phép bởi các ràng buộc của ngôn ngữ, chuyển đổi giữa các con trỏ và bất kỳ loại số học nào khác ngoài các loại số nguyên là không xác định.

Các chuyển đổi con trỏ được phép sau đây không yêu cầu ép kiểu rõ ràng:

- Chuyển đổi từ một loại con trỏ sang \_Bool (chỉ C99);

- Chuyển đổi từ một hằng số con trỏ null sang một loại con trỏ;

- Chuyển đổi từ một loại con trỏ sang một loại con trỏ tương thích với điều kiện loại đích có tất cả các định tính của loại nguồn;

- Chuyển đổi giữa một con trỏ đến một đối tượng hoặc loại không hoàn chỉnh và void \*, hoặc phiên bản đủ điều kiện của nó, với điều kiện loại đích có tất cả các định tính của loại nguồn.

Trong C99, bất kỳ chuyển đổi ngầm định nào không thuộc tập hợp con này của các chuyển đổi con trỏ vi phạm một ràng buộc (C99 Phần 6.5.4 và 6.5.16.1).

Trong C90, bất kỳ chuyển đổi ngầm định nào không thuộc tập hợp con này của các chuyển đổi con trỏ dẫn đến hành vi không xác định (C90 Phần 6.3.4 và 6.3.16.1).

Chuyển đổi giữa các loại con trỏ và các loại số nguyên là do triển khai xác định.

#### ### Quy tắc 11.1:

Không được thực hiện các chuyển đổi giữa một con trỏ đến một hàm và bất kỳ loại nào khác.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- C90 [Không xác định 24, 27–29], C99 [Không xác định 21, 23, 39, 41]

- Phân loại: Bắt buộc

- Phân tích: Quyết định được, Đơn vị Dịch thuật Đơn

- Áp dụng cho: C90, C99

#Mở rộng

Một con trỏ đến một hàm chỉ được chuyển đổi thành hoặc từ một con trỏ đến một hàm với một loại tương thích.

### Quy tắc 11.1: Không được thực hiện các chuyển đổi giữa một con trỏ đến một hàm và bất kỳ loại nào khác.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- \*\*C90\*\* [Không xác định 24, 27–29], \*\*C99\*\* [Không xác định 21, 23, 39, 41]

- \*\*Phân loại\*\*: Bắt buộc

- \*\*Phân tích\*\*: Quyết định được, Đơn vị Dịch thuật Đơn

- \*\*Áp dụng cho\*\*: C90, C99

Lý do

Việc chuyển đổi một con trỏ đến một hàm thành hoặc từ bất kỳ loại nào sau đây:

- Con trỏ đến đối tượng;

- Con trỏ đến không hoàn chỉnh;

- void \*

kết quả là hành vi không xác định.

Nếu một hàm được gọi thông qua một con trỏ mà loại của nó không tương thích với hàm được gọi, hành vi là không xác định. Chuyển đổi một con trỏ đến một hàm thành một con trỏ đến một hàm có loại khác nhau được cho phép bởi Tiêu chuẩn. Chuyển đổi một số nguyên thành một con trỏ đến một hàm cũng được phép bởi Tiêu chuẩn. Tuy nhiên, cả hai đều bị cấm bởi quy tắc này để tránh hành vi không xác định sẽ xảy ra khi gọi một hàm bằng cách sử dụng một loại con trỏ không tương thích.

#Ngoại lệ

1. Một hằng số con trỏ null có thể được chuyển đổi thành một con trỏ đến một hàm;

2. Một con trỏ đến một hàm có thể được chuyển đổi thành void;

3. Một loại hàm có thể được chuyển đổi ngầm định thành một con trỏ đến loại hàm đó.

Lưu ý: ngoại lệ 3 bao gồm các chuyển đổi ngầm định được mô tả trong \*\*C90\*\* Phần 6.2.2.1 và \*\*C99\*\* Phần 6.3.2.1. Các chuyển đổi này thường xảy ra khi:

- Một hàm được gọi trực tiếp, tức là sử dụng một định danh hàm để chỉ định hàm được gọi;

- Một hàm được gán cho một con trỏ hàm.

#Ví dụ

```c

typedef void ( \*fp16 ) ( int16\_t n );

typedef void ( \*fp32 ) ( int32\_t n );

#include <stdlib.h> /\* Để lấy macro NULL \*/

fp16 fp1 = NULL; /\* Tuân thủ - ngoại lệ 1 \*/

fp32 fp2 = ( fp32 ) fp1; /\* Không tuân thủ - con trỏ hàm vào con trỏ hàm khác \*/

if ( fp2 != NULL ) /\* Tuân thủ - ngoại lệ 1 \*/

{

}

fp16 fp3 = ( fp16 ) 0x8000; /\* Không tuân thủ - số nguyên vào con trỏ hàm \*/

fp16 fp4 = ( fp16 ) 1.0e6F; /\* Không tuân thủ - số thực vào con trỏ hàm \*/

typedef fp16 ( \*pfp16 ) ( void );

pfp16 pfp1;

( void ) ( \*pfp1 ( ) ); /\* Tuân thủ - ngoại lệ 2 - ép con trỏ hàm vào void \*/

extern void f ( int16\_t n );

f ( 1 ); /\* Tuân thủ - ngoại lệ 3 - chuyển đổi ngầm định của f thành con trỏ đến hàm \*/

fp16 fp5 = f; /\* Tuân thủ - ngoại lệ 3 \*/

```

#### ### Quy tắc 11.2:

Không được thực hiện các chuyển đổi giữa một con trỏ đến loại không hoàn chỉnh và bất kỳ loại nào khác.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- \*\*C90\*\* [Không xác định 29], \*\*C99\*\* [Không xác định 21, 22, 41]

- \*\*Phân loại\*\*: Bắt buộc

- \*\*Phân tích\*\*: Quyết định được, Đơn vị Dịch thuật Đơn

- \*\*Áp dụng cho\*\*: C90, C99

#Mở rộng

Một con trỏ đến loại không hoàn chỉnh không được chuyển đổi thành loại khác.

Không được chuyển đổi thành một con trỏ đến loại không hoàn chỉnh.

Mặc dù một con trỏ đến void cũng là một con trỏ đến loại không hoàn chỉnh, quy tắc này không áp dụng cho con trỏ đến void vì chúng được bao phủ bởi Quy tắc 11.5.

#Lý do

Chuyển đổi thành hoặc từ một con trỏ đến loại không hoàn chỉnh có thể dẫn đến một con trỏ không được căn chỉnh chính xác, dẫn đến hành vi không xác định. Chuyển đổi một con trỏ đến loại không hoàn chỉnh thành hoặc từ một loại số thực luôn dẫn đến hành vi không xác định. Con trỏ đến loại không hoàn chỉnh đôi khi được sử dụng để ẩn đi cấu trúc của một đối tượng. Chuyển đổi một con trỏ đến loại không hoàn chỉnh thành một con trỏ đến đối tượng sẽ phá vỡ sự bao đóng này.

#Ngoại lệ

1. Một hằng số con trỏ null có thể được chuyển đổi thành một con trỏ đến loại không hoàn chỉnh.

2. Một con trỏ đến loại không hoàn chỉnh có thể được chuyển đổi thành void.

#Ví dụ

```c

struct s; /\* Loại không hoàn chỉnh \*/

struct t; /\* Một loại không hoàn chỉnh khác \*/

struct s \*sp;

struct t \*tp;

int16\_t \*ip;

#include <stdlib.h> /\* Để lấy macro NULL \*/

ip = (int16\_t \*) sp; /\* Không tuân thủ \*/

sp = (struct s \*) 1234; /\* Không tuân thủ \*/

tp = (struct t \*) sp; /\* Không tuân thủ - ép con trỏ thành một loại không hoàn chỉnh khác \*/

sp = NULL; /\* Tuân thủ - ngoại lệ 1 \*/

struct s \*f (void);

(void) f(); /\* Tuân thủ - ngoại lệ 2 \*/

```

Xem thêm

- Quy tắc 11.5

#### ### Quy tắc 11.3:

Không được thực hiện ép kiểu giữa một con trỏ đến loại đối tượng và một con trỏ đến một loại đối tượng khác.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- \*\*C90\*\* [Không xác định 20], \*\*C99\*\* [Không xác định 22, 34]

- \*\*Phân loại\*\*: Bắt buộc

- \*\*Phân tích\*\*: Quyết định được, Đơn vị Dịch thuật Đơn

- \*\*Áp dụng cho\*\*: C90, C99

#Mở rộng

Quy tắc này áp dụng cho các loại không đủ điều kiện mà các con trỏ chỉ đến.

#Lý do

Ép kiểu một con trỏ đến loại đối tượng thành một con trỏ đến một loại đối tượng khác có thể dẫn đến một con trỏ không được căn chỉnh chính xác, dẫn đến hành vi không xác định. Ngay cả khi chuyển đổi được biết là tạo ra một con trỏ được căn chỉnh chính xác, hành vi có thể không xác định nếu con trỏ đó được sử dụng để truy cập một đối tượng. Ví dụ, nếu một đối tượng có loại int được truy cập như một short, hành vi là không xác định ngay cả khi int và short có cùng cấu trúc và yêu cầu căn chỉnh. Xem \*\*C90\*\* Phần 6.3, \*\*C99\*\* Phần 6.5, đoạn 7 để biết chi tiết.

#Ngoại lệ

Được phép chuyển đổi một con trỏ đến loại đối tượng thành một con trỏ đến một trong các loại đối tượng char, signed char hoặc unsigned char. Tiêu chuẩn đảm bảo rằng các con trỏ đến các loại này có thể được sử dụng để truy cập các byte riêng lẻ của một đối tượng.

#Ví dụ

```c

uint8\_t \*p1;

uint32\_t \*p2;

/\* Không tuân thủ - có thể không căn chỉnh \*/

p2 = (uint32\_t \*) p1;

extern uint32\_t read\_value(void);

extern void print(uint32\_t n);

void f(void)

{

uint32\_t u = read\_value();

uint16\_t \*hi\_p = (uint16\_t \*)&u; /\* Không tuân thủ mặc dù có thể được căn chỉnh chính xác \*/

\*hi\_p = 0; /\* Cố gắng xóa 16-bit cao trên máy big-endian \*/

print(u); /\* Dòng trên có thể không được thực hiện \*/

}

```

Ví dụ sau đây tuân thủ vì quy tắc áp dụng cho các loại con trỏ không đủ điều kiện. Nó không ngăn cản việc thêm các định tính vào loại đối tượng.

```c

uint32\_t u = 0;

const uint32\_t \*p = &u; /\* Tuân thủ \*/

```

### Quy tắc 11.3: Không được thực hiện ép kiểu giữa một con trỏ đến loại đối tượng và một con trỏ đến một loại đối tượng khác

\*\*Ví dụ bổ sung\*\*:

```c

const short \*p;

const volatile short \*q;

q = (const volatile short \*) p; /\* Tuân thủ \*/

```

Ví dụ sau không tuân thủ vì các loại con trỏ không đủ điều kiện là khác nhau, cụ thể là "con trỏ đến int đủ điều kiện const" và "con trỏ đến int".

```c

int \* const \* pcpi;

const int \* const \* pcpci;

pcpci = (const int \* const \*) pcpi; /\* Không tuân thủ \*/

```

Xem thêm:

- Quy tắc 11.4

- Quy tắc 11.5

- Quy tắc 11.8

#### ### Quy tắc 11.4:

Không nên thực hiện chuyển đổi giữa con trỏ đến đối tượng và loại số nguyên.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- \*\*C90\*\* [Không xác định 20; Thực hiện 24]

- \*\*C99\*\* [Không xác định 21, 34; Thực hiện J.3.7(1)]

- \*\*Phân loại\*\*: Khuyến cáo

- \*\*Phân tích\*\*: Quyết định được, Đơn vị Dịch thuật Đơn

- \*\*Áp dụng cho\*\*: C90, C99

Mở rộng

Không nên chuyển đổi con trỏ thành số nguyên.

Không nên chuyển đổi số nguyên thành con trỏ.

Lý do

Chuyển đổi một số nguyên thành một con trỏ đến đối tượng có thể dẫn đến một con trỏ không được căn chỉnh chính xác, dẫn đến hành vi không xác định.

Chuyển đổi một con trỏ đến đối tượng thành một số nguyên có thể tạo ra một giá trị không thể được biểu diễn trong loại số nguyên đã chọn, dẫn đến hành vi không xác định.

Lưu ý: các loại C99 `intptr\_t` và `uintptr\_t`, được khai báo trong `<stdint.h>`, lần lượt là các loại số nguyên có dấu và không dấu có khả năng biểu diễn các giá trị con trỏ. Mặc dù vậy, chuyển đổi giữa một con trỏ đến đối tượng và các loại này không được phép bởi quy tắc này vì việc sử dụng chúng không tránh được hành vi không xác định liên quan đến các con trỏ không căn chỉnh.

Ép kiểu giữa một con trỏ và một loại số nguyên nên được tránh nếu có thể, nhưng có thể cần thiết khi xử lý các thanh ghi ánh xạ bộ nhớ hoặc các tính năng đặc biệt khác của phần cứng. Nếu ép kiểu giữa số nguyên và con trỏ được sử dụng, cần phải đảm bảo rằng bất kỳ con trỏ nào được tạo ra không dẫn đến hành vi không xác định đã thảo luận trong Quy tắc 11.3.

Ngoại lệ

Một hằng số con trỏ null có loại số nguyên có thể được chuyển đổi thành một con trỏ đến đối tượng.

Ví dụ

```c

uint8\_t \*PORTA = (uint8\_t \*) 0x0002; /\* Không tuân thủ \*/

uint16\_t \*p;

int32\_t addr = (int32\_t) &p; /\* Không tuân thủ \*/

uint8\_t \*q = (uint8\_t \*) addr; /\* Không tuân thủ \*/

bool\_t b = (bool\_t) p; /\* Không tuân thủ \*/

enum etag { A, B } e = (enum etag) p; /\* Không tuân thủ \*/

```

Xem thêm:

- Quy tắc 11.3

- Quy tắc 11.7

- Quy tắc 11.9

#### ### Quy tắc 11.5:

Không nên thực hiện chuyển đổi từ con trỏ đến void thành con trỏ đến đối tượng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- \*\*C90\*\* [Không xác định 20]

- \*\*C99\*\* [Không xác định 22, 34]

- \*\*Phân loại\*\*: Khuyến cáo

- \*\*Phân tích\*\*: Quyết định được, Đơn vị Dịch thuật Đơn

- \*\*Áp dụng cho\*\*: C90, C99

Lý do

Chuyển đổi từ một con trỏ đến void thành một con trỏ đến đối tượng có thể dẫn đến một con trỏ không được căn chỉnh chính xác, dẫn đến hành vi không xác định. Điều này nên được tránh nếu có thể, nhưng có thể cần thiết, ví dụ khi xử lý các hàm cấp phát bộ nhớ. Nếu chuyển đổi từ một con trỏ đến đối tượng thành một con trỏ đến void được sử dụng, cần phải đảm bảo rằng bất kỳ con trỏ nào được tạo ra không dẫn đến hành vi không xác định đã thảo luận trong Quy tắc 11.3.

Ngoại lệ

Một hằng số con trỏ null có loại con trỏ đến void có thể được chuyển đổi thành con trỏ đến đối tượng.

Ví dụ

```c

uint32\_t \*p32;

void \*p;

uint16\_t \*p16;

p = p32; /\* Tuân thủ - con trỏ đến uint32\_t thành con trỏ đến void \*/

p16 = p; /\* Không tuân thủ \*/

p = (void \*) p16; /\* Tuân thủ \*/

p32 = (uint32\_t \*) p; /\* Không tuân thủ \*/

```

Xem thêm:

- Quy tắc 11.2

- Quy tắc 11.3

#### ### Quy tắc 11.6:

Không được thực hiện ép kiểu giữa con trỏ đến void và loại số học

- \*\*C90\*\* [Không xác định 29; Thực hiện 24]

- \*\*C99\*\* [Không xác định 21, 41; Thực hiện J.3.7(1)]

- \*\*Phân loại\*\*: Bắt buộc

- \*\*Phân tích\*\*: Quyết định được, Đơn vị Dịch thuật Đơn

- \*\*Áp dụng cho\*\*: C90, C99

Lý do

Chuyển đổi một số nguyên thành một con trỏ đến void có thể dẫn đến một con trỏ không được căn chỉnh chính xác, dẫn đến hành vi không xác định.

Chuyển đổi một con trỏ đến void thành một số nguyên có thể tạo ra một giá trị không thể được biểu diễn trong loại số nguyên đã chọn, dẫn đến hành vi không xác định.

Chuyển đổi giữa bất kỳ loại số học không phải số nguyên và con trỏ đến void là không xác định.

Ngoại lệ

Một biểu thức hằng số nguyên với giá trị 0 có thể được ép kiểu thành con trỏ đến void.

Ví dụ

```c

void \*p;

uint32\_t u;

/\* Không tuân thủ - triển khai xác định \*/

p = (void \*) 0x1234u;

/\* Không tuân thủ - không xác định \*/

p = (void \*) 1024.0f;

/\* Không tuân thủ - triển khai xác định \*/

u = (uint32\_t) p;

```

#### ### Quy tắc 11.7:

Không được thực hiện ép kiểu giữa con trỏ đến đối tượng và loại số học không phải số nguyên

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- \*\*C90\*\* [Không xác định 29; Thực hiện 24]

- \*\*C99\*\* [Không xác định 21, 41; Thực hiện J.3.7(1)]

- \*\*Phân loại\*\*: Bắt buộc

- \*\*Phân tích\*\*: Quyết định được, Đơn vị Dịch thuật Đơn

- \*\*Áp dụng cho\*\*: C90, C99

Mở rộng

Đối với mục đích của quy tắc này, một loại số học không phải số nguyên bao gồm:

- Loại Boolean

- Loại ký tự

- Loại enum

- Loại số thực

Lý do

Chuyển đổi một loại Boolean, ký tự hoặc enum thành một con trỏ đến đối tượng có thể dẫn đến một con trỏ không được căn chỉnh chính xác, dẫn đến hành vi không xác định.

Chuyển đổi một con trỏ đến đối tượng thành một loại Boolean, ký tự hoặc enum có thể tạo ra một giá trị không thể được biểu diễn trong loại số nguyên đã chọn, dẫn đến hành vi không xác định.

Chuyển đổi một con trỏ đến đối tượng thành hoặc từ một loại số thực dẫn đến hành vi không xác định.

Ví dụ

```c

int16\_t \*p;

float32\_t f;

f = (float32\_t) p; /\* Không tuân thủ \*/

p = (int16\_t \*) f; /\* Không tuân thủ \*/

```

Xem thêm:

- Quy tắc 11.4

#### ### Quy tắc 11.8:

Không được thực hiện ép kiểu bỏ bất kỳ định tính const hoặc volatile nào từ loại được trỏ tới bởi con trỏ

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- \*\*C90\*\* [Không xác định 12, 39, 40]

- \*\*C99\*\* [Không xác định 30, 61, 62]

- \*\*Phân loại\*\*: Bắt buộc

- \*\*Phân tích\*\*: Quyết định được, Đơn vị Dịch thuật Đơn

- \*\*Áp dụng cho\*\*: C90, C99

Lý do

Bất kỳ nỗ lực nào để loại bỏ định tính liên quan đến loại được trỏ đến bằng cách sử dụng ép kiểu là vi phạm nguyên tắc định tính loại.

Lưu ý: định tính đề cập ở đây không giống với bất kỳ định tính nào có thể được áp dụng cho bản thân con trỏ.

Một số vấn đề có thể phát sinh nếu một định tính bị loại bỏ khỏi đối tượng được địa chỉ hóa là:

- Loại bỏ định tính const có thể làm mất trạng thái chỉ đọc của một đối tượng và dẫn đến việc nó bị sửa đổi;

- Loại bỏ định tính const có thể dẫn đến một ngoại lệ khi đối tượng được truy cập;

- Loại bỏ định tính volatile có thể dẫn đến việc truy cập đối tượng bị tối ưu hóa.

Lưu ý: việc loại bỏ định tính restrict trong C99 là vô hại.

Ví dụ

```c

uint16\_t x;

uint16\_t \* const cpi = &x; /\* con trỏ const \*/

uint16\_t \* const \*pcpi; /\* con trỏ đến con trỏ const \*/

uint16\_t \*\*ppi;

const uint16\_t \*pci; /\* con trỏ đến const \*/

volatile uint16\_t \*pvi; /\* con trỏ đến volatile \*/

uint16\_t \*pi;

pi = cpi; /\* Tuân thủ - không cần chuyển đổi \*/

pi = (uint16\_t \*)pci; /\* Không tuân thủ \*/

pi = (uint16\_t \*)pvi; /\* Không tuân thủ \*/

ppi = (uint16\_t \*\*)pcpi; /\* Không tuân thủ \*/

```

Xem thêm:

- Quy tắc 11.3

#### ### Quy tắc 11.9:

Macro NULL phải là dạng duy nhất được phép của hằng số con trỏ null số nguyên

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

- \*\*C90\*\* [Không xác định 24]

- \*\*C99\*\* [Không xác định 21, 41]

- \*\*Phân loại\*\*: Bắt buộc

- \*\*Phân tích\*\*: Quyết định được, Đơn vị Dịch thuật Đơn

- \*\*Áp dụng cho\*\*: C90, C99

Mở rộng

Một biểu thức hằng số nguyên có giá trị 0 phải được mở rộng từ macro NULL nếu nó xuất hiện trong bất kỳ ngữ cảnh nào sau đây:

- Là giá trị được gán cho một con trỏ;

- Là một toán hạng của toán tử == hoặc != mà toán hạng còn lại là một con trỏ;

- Là toán hạng thứ hai của toán tử ?: mà toán hạng thứ ba là một con trỏ;

- Là toán hạng thứ ba của toán tử ?: mà toán hạng thứ hai là một con trỏ.

Bỏ qua khoảng trắng và bất kỳ dấu ngoặc bao quanh nào, bất kỳ biểu thức hằng số nguyên nào như vậy phải đại diện cho toàn bộ mở rộng của NULL.

Lưu ý: một hằng số con trỏ null dạng (void \*)0 được phép, cho dù nó có được mở rộng từ NULL hay không.

Lý do

Sử dụng NULL thay vì 0 làm rõ rằng một hằng số con trỏ null đã được dự định.

Ví dụ

Trong ví dụ sau, việc khởi tạo p2 là tuân thủ vì biểu thức hằng số nguyên 0 không xuất hiện trong một ngữ cảnh bị cấm bởi quy tắc này.

```c

int32\_t \*p1 = 0; /\* Không tuân thủ \*/

int32\_t \*p2 = (void \*) 0; /\* Tuân thủ \*/

```

### Rule 11.9: The macro NULL shall be the only permitted form of integer null pointer constant

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

Category: Required

Analysis: Decidable, Single Translation Unit

Applies to: C90, C99

\*\*Amplification\*\*

An integer constant expression with the value 0 shall be derived from expansion of the macro NULL if it appears in any of the following contexts:

- As the value being assigned to a pointer;

- As an operand of an == or != operator whose other operand is a pointer;

- As the second operand of a ?: operator whose third operand is a pointer;

- As the third operand of a ?: operator whose second operand is a pointer.

Ignoring whitespace and any surrounding parentheses, any such integer constant expression shall represent the entire expansion of NULL.

\*\*Rationale\*\*

Using NULL rather than 0 makes it clear that a null pointer constant was intended.

\*\*Example\*\*

In the following example, the initialization of `p2` is compliant because the integer constant expression 0 does not appear in one of the contexts prohibited by this rule.

```c

int32\_t \*p1 = 0; // Non-compliant

int32\_t \*p2 = (void \*) 0; // Compliant

```

In the following example, the comparison between `p2` and `(void \*)0` is compliant because the integer constant expression 0 appears as the operand of a cast and not in one of the contexts prohibited by this rule.

```c

#define MY\_NULL\_1 0

#define MY\_NULL\_2 (void \*) 0

if (p1 == MY\_NULL\_1) { // Non-compliant

}

if (p2 == MY\_NULL\_2) { // Compliant

}

```

The following example is compliant because use of the macro NULL provided by the implementation is always permitted, even if it expands to an integer constant expression with value 0.

```c

#include <stddef.h>

extern void f(uint8\_t \*p);

// Compliant for any conforming definition of NULL, such as:

// 0

// (void \*) 0

// (((0)))

// (((1 - 1)))

f(NULL);

```

\*\*See also\*\*

Rule 11.4

---

### Biểu thức

#### Quy tắc 12.1:

Độ ưu tiên của các toán tử trong các biểu thức cần được làm rõ ràng.

**Loại**: Khuyến nghị

**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất

**Áp dụng cho**: C90, C99

**Giải thích chi tiết**: Bảng dưới đây được sử dụng để định nghĩa quy tắc này.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

Dưới đây là bản dịch tiếng Việt:

Bảng dưới đây được sử dụng để định nghĩa quy tắc này.

| **Mô tả** | **Toán tử hoặc Toán hạng** | **Độ ưu tiên** |
| --- | --- | --- |
| **Chính** | định danh, hằng số, chuỗi ký tự, (biểu thức) | 16 (cao) |
| **Hậu tố** | [] () (gọi hàm) . -> ++ (tăng hậu tố) -- (giảm hậu tố) {} (C99: literal ghép) | 15 |
| **Đơn nhất** | ++ (tăng tiền tố) -- (giảm tiền tố) & \* + - ~ ! sizeof defined (bộ tiền xử lý) | 14 |
| **Ép kiểu** | () | 13 |
| **Nhân chia** | \* / % | 12 |
| **Cộng trừ** | + - | 11 |
| **Dịch bit** | << >> | 10 |
| **Quan hệ** | < > <= >= | 9 |
| **Bằng nhau** | == != | 8 |
| **AND bit** | & | 7 |
| **XOR bit** | ^ | 6 |
| **OR bit** |  |  |
| **AND logic** | && | 4 |
| **OR logic** |  |  |
| **Điều kiện** | ?: | 2 |
| **Gán** | = \*= /= %= += -= <<= >>= &= ^= | = |
| **Dấu phẩy** | , | 0 (thấp) |

Lưu ý

Độ ưu tiên được sử dụng trong bảng này được chọn để mô tả ngắn gọn quy tắc. Chúng không nhất thiết phải giống với độ ưu tiên có thể gặp trong các mô tả khác về độ ưu tiên của toán tử.

Giải thích về quy tắc này

Đối với quy tắc này, độ ưu tiên của một biểu thức là độ ưu tiên của phần tử (toán tử hoặc toán hạng) tại gốc của cây phân tích cú pháp cho biểu thức đó.

**Ví dụ**: Cây phân tích cú pháp cho biểu thức a << b + c có thể được biểu diễn như sau:

|  |
| --- |
|  |

css

Sao chép mã

<<

/ \

a +

/ \

b c

Phần tử tại gốc của cây phân tích cú pháp này là <<, do đó biểu thức có độ ưu tiên là 10.

Các lời khuyên sau được đưa ra:

* Toán hạng của toán tử sizeof nên được đặt trong dấu ngoặc đơn.
* Một biểu thức có độ ưu tiên trong khoảng từ 2 đến 12 nên được đặt trong dấu ngoặc đơn với bất kỳ toán hạng nào có:
  + Độ ưu tiên nhỏ hơn 13, và
  + Độ ưu tiên lớn hơn độ ưu tiên của biểu thức.

Lý do

Ngôn ngữ C có một số lượng lớn các toán tử và độ ưu tiên tương đối của chúng không mang tính trực quan. Điều này có thể khiến các lập trình viên ít kinh nghiệm mắc sai lầm. Việc sử dụng dấu ngoặc để làm rõ độ ưu tiên của toán tử giúp loại bỏ khả năng kỳ vọng của lập trình viên bị sai lệch. Nó cũng làm rõ ý định ban đầu của lập trình viên cho những người xem xét hoặc bảo trì mã nguồn.

Điều này được công nhận rằng việc lạm dụng dấu ngoặc có thể làm mã trở nên lộn xộn và giảm khả năng đọc hiểu. Quy tắc này nhằm đạt được sự cân bằng giữa mã khó hiểu do chứa quá nhiều hoặc quá ít dấu ngoặc.

Ví dụ

Dưới đây là ví dụ về các biểu thức với toán tử đơn nhất hoặc hậu tố có các toán hạng là các biểu thức chính hoặc các biểu thức có toán tử mức cao nhất với độ ưu tiên 15.

|  |
| --- |
|  |

a[ i ]->n; /\* Hợp lệ - không cần viết ( a[ i ] )->n \*/

\*p++; /\* Hợp lệ - không cần viết \*( p++ ) \*/

sizeof x + y; /\* Không hợp lệ - viết là sizeof ( x ) + y hoặc sizeof ( x + y ) \*/

Dưới đây là ví dụ về các biểu thức chứa các toán tử ở cùng mức độ ưu tiên. Tất cả đều hợp lệ nhưng tùy thuộc vào kiểu của a, b, và c, bất kỳ biểu thức nào có hơn một toán tử có thể vi phạm các quy tắc khác.

|  |
| --- |
|  |

a + b;

a + b + c;

( a + b ) + c;

a + ( b + c );

a + b - c + d;

( a + b ) - ( c + d );

Dưới đây là ví dụ về các biểu thức hỗn hợp toán tử:

|  |
| --- |
|  |

/\* Hợp lệ - không cần viết f ( ( a + b ), c ) \*/

x = f ( a + b, c );

/\* Không hợp lệ

\* Toán hạng của toán tử điều kiện (độ ưu tiên 2) là:

\* == (độ ưu tiên 8) cần dấu ngoặc

\* a (độ ưu tiên 16) không cần dấu ngoặc

\* - (độ ưu tiên 11) cần dấu ngoặc

\*/

x = a == b ? a : a - b;

/\* Hợp lệ \*/

x = ( a == b ) ? a : ( a - b );

Hy vọng bản dịch này giúp bạn hiểu rõ hơn về quy tắc và lời khuyên trong việc sử dụng dấu ngoặc khi lập trình C theo chuẩn MISRA.

Các ví dụ tuân thủ và không tuân thủ quy tắc

|  |
| --- |
|  |

/\* Hợp lệ

\* Toán hạng của toán tử << (độ ưu tiên 10) là:

\* a (độ ưu tiên 16) không cần dấu ngoặc

\* ( E ) (độ ưu tiên 16) đã được đặt trong dấu ngoặc

\*/

x = a << ( b + c );

/\* Hợp lệ

\* Toán hạng của toán tử && (độ ưu tiên 4) là:

\* a (độ ưu tiên 16) không cần dấu ngoặc

\* && (độ ưu tiên 4) không cần dấu ngoặc

\*/

if ( a && b && c )

{

}

/\* Hợp lệ

\* Toán hạng của toán tử && (độ ưu tiên 4) là:

\* defined(X) (độ ưu tiên 14) không cần dấu ngoặc

\* (E) (độ ưu tiên 16) đã được đặt trong dấu ngoặc

\*/

#if defined ( X ) && ( ( X + Y ) > Z )

/\* Hợp lệ

\* Toán hạng của toán tử && (độ ưu tiên 4) là:

\* !defined ( X ) (độ ưu tiên 14) không cần dấu ngoặc

\* defined ( Y ) (độ ưu tiên 14) không cần dấu ngoặc

\* Toán hạng của toán tử ! (độ ưu tiên 14) là:

\* defined ( X ) (độ ưu tiên 14) không cần dấu ngoặc

\*/

#if !defined ( X ) && defined ( Y )

**Lưu ý**: Quy tắc này không yêu cầu các toán hạng của toán tử dấu phẩy (,) phải đặt trong dấu ngoặc. Tuy nhiên, việc sử dụng toán tử , bị cấm bởi Quy tắc 12.3.

|  |
| --- |
|  |

x = a, b; /\* Hợp lệ - được phân tích cú pháp thành ( x = a ), b \*/

#### Quy tắc 12.2:

Toán hạng bên phải của toán tử dịch chuyển phải nằm trong phạm vi từ 0 đến một số nhỏ hơn bề rộng tính theo bit của loại cơ bản của toán hạng bên trái.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 32], **C99** [Không xác định 48]  
**Loại**: Bắt buộc  
**Phân tích**: Không quyết định được, Hệ thống  
**Áp dụng cho**: C90, C99

Lý do

Nếu toán hạng bên phải là số âm, hoặc lớn hơn hoặc bằng bề rộng của toán hạng bên trái, hành vi sẽ không được xác định.

Ví dụ, nếu toán hạng bên trái của phép dịch trái hoặc dịch phải là một số nguyên 16-bit, thì điều quan trọng là phải đảm bảo rằng chỉ dịch chuyển bởi một số nằm trong phạm vi từ 0 đến 15.

Xem **Mục 8.10** để biết mô tả về loại cơ bản và các giới hạn đối với các loại cơ bản của toán hạng trong các toán tử dịch chuyển.

Có nhiều cách để đảm bảo tuân thủ quy tắc này. Cách đơn giản nhất là để toán hạng bên phải là một hằng số (giá trị của nó có thể được kiểm tra tĩnh). Việc sử dụng loại số nguyên không dấu sẽ đảm bảo rằng toán hạng không âm, do đó chỉ cần kiểm tra giới hạn trên (có thể thực hiện động tại thời gian chạy hoặc bằng cách xem xét mã). Nếu không, cả hai giới hạn sẽ cần được kiểm tra.

Ví dụ

|  |
| --- |
|  |

u8a = u8a << 7; /\* Hợp lệ \*/

u8a = u8a << 8; /\* Không hợp lệ \*/

u16a = ( uint16\_t ) u8a << 9; /\* Hợp lệ \*/

/\* Lưu ý: loại cơ bản của 1u là unsigned char, trong khi loại cơ bản của 1UL là unsigned long. \*/

1u << 10u; /\* Không hợp lệ \*/

( uint16\_t ) 1u << 10u; /\* Hợp lệ \*/

1UL << 10u; /\* Hợp lệ \*/

Hy vọng phần dịch này giúp bạn nắm rõ hơn về các quy tắc và ví dụ cụ thể khi lập trình C theo tiêu chuẩn MISRA.

#### Quy tắc 12.3:

Toán tử dấu phẩy không nên được sử dụng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Khuyến nghị  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Lý do**  
Việc sử dụng toán tử dấu phẩy thường gây bất lợi cho khả năng đọc hiểu mã nguồn, và hiệu ứng tương tự thường có thể đạt được bằng các phương pháp khác.

**Ví dụ**

|  |
| --- |
|  |

f ( ( 1, 2 ), 3 ); /\* Không hợp lệ - có bao nhiêu tham số? \*/

Ví dụ sau đây không tuân thủ quy tắc này và các quy tắc khác:

|  |
| --- |
|  |

for ( i = 0, p = &a[ 0 ]; i < N; ++i, ++p )

{

}

#### Quy tắc 12.4:

Việc đánh giá các biểu thức hằng số không nên dẫn đến tràn số nguyên không dấu

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Khuyến nghị  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Quy tắc này áp dụng cho các biểu thức đáp ứng các ràng buộc của một biểu thức hằng số, cho dù chúng có xuất hiện trong một ngữ cảnh yêu cầu một biểu thức hằng số hay không. Nếu một biểu thức không được đánh giá, chẳng hạn vì nó xuất hiện ở toán hạng bên phải của toán tử && mà toán hạng bên trái luôn là false, thì quy tắc này không áp dụng.

**Lý do**  
Các biểu thức số nguyên không dấu không thực sự tràn, mà thay vào đó là "quay vòng". Mặc dù có thể có lý do chính đáng để sử dụng toán học theo modulo khi chạy chương trình, nhưng việc sử dụng nó ở thời điểm biên dịch ít có khả năng là cố ý.

**Ví dụ**

Biểu thức gắn với nhãn case cần phải là một biểu thức hằng số. Nếu xảy ra hiện tượng quay vòng không dấu khi đánh giá biểu thức case, điều đó có khả năng không phải là cố ý. Trên một máy với kiểu int 16-bit, bất kỳ giá trị nào của BASE lớn hơn hoặc bằng 65.024 sẽ dẫn đến hiện tượng quay vòng trong ví dụ sau:

|  |
| --- |
|  |

#define BASE 65024u

switch ( x )

{

case BASE + 0u:

f ( );

break;

case BASE + 1u:

g ( );

break;

case BASE + 512u: /\* Không hợp lệ - quay vòng thành 0 \*/

h ( );

break;

}

Biểu thức điều khiển của một chỉ thị tiền xử lý #if hoặc #elif cần phải là một biểu thức hằng số.

|  |
| --- |
|  |

#if 1u + ( 0u - 10u ) /\* Không hợp lệ vì ( 0u - 10u ) quay vòng \*/

Trong ví dụ này, biểu thức DELAY + WIDTH có giá trị là 70.000 nhưng sẽ quay vòng thành 4.464 trên máy với kiểu int 16-bit.

|  |
| --- |
|  |

#define DELAY 10000u

#define WIDTH 60000u

void fixed\_pulse ( void )

{

uint16\_t off\_time16 = DELAY + WIDTH; /\* Không hợp lệ \*/

}

Quy tắc này không áp dụng cho biểu thức c + 1 trong ví dụ hợp lệ sau vì nó truy cập một đối tượng và do đó không đáp ứng ràng buộc của một biểu thức hằng số:

|  |
| --- |
|  |

const uint16\_t c = 0xffffu;

void f ( void )

{

uint16\_t y = c + 1u; /\* Hợp lệ \*/

}

Trong ví dụ sau, biểu thức con ( 0u - 1u ) dẫn đến hiện tượng quay vòng số nguyên không dấu. Trong quá trình khởi tạo x, biểu thức con không được đánh giá và do đó là hợp lệ. Tuy nhiên, trong quá trình khởi tạo y, nó có thể được đánh giá và biểu thức do đó không hợp lệ.

|  |
| --- |
|  |

bool\_t b;

void g ( void )

{

uint16\_t x = ( 0u == 0u ) ? 0u : ( 0u - 1u ); /\* Hợp lệ \*/

uint16\_t y = b ? 0u : ( 0u - 1u ); /\* Không hợp lệ \*/

}

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và lý do của chúng khi lập trình C theo tiêu chuẩn MISRA.

### 8.13 Tác động phụ

**Quy tắc 13.1**:

Danh sách khởi tạo không được chứa các tác động phụ dai dẳng.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C99** [Không xác định 17, 22]  
**Loại**: Bắt buộc  
**Phân tích**: Không quyết định được, Hệ thống  
**Áp dụng cho**: C99

**Lý do**  
C90 giới hạn các khởi tạo viên cho các đối tượng tự động có kiểu tổng hợp chỉ chứa các biểu thức hằng số. Tuy nhiên, C99 cho phép các khởi tạo viên tổng hợp tự động chứa các biểu thức được đánh giá tại thời gian chạy. Nó cũng cho phép các literal tổng hợp hoạt động như các đối tượng khởi tạo ẩn danh. Thứ tự xảy ra của các tác động phụ trong quá trình đánh giá các biểu thức trong một danh sách khởi tạo không được xác định, do đó hành vi khởi tạo sẽ không thể dự đoán được nếu những tác động phụ đó là dai dẳng.

**Ví dụ**

|  |
| --- |
|  |

volatile uint16\_t v1;

void f ( void )

{

/\* Không hợp lệ - truy cập volatile là tác động phụ dai dẳng \*/

uint16\_t a[2] = { v1, 0 };

}

void g ( uint16\_t x, uint16\_t y )

{

/\* Hợp lệ - không có tác động phụ \*/

uint16\_t a[2] = { x + y, x - y };

}

uint16\_t x = 0u;

extern void p ( uint16\_t a[2] );

void h ( void )

{

/\* Không hợp lệ - hai tác động phụ \*/

p( ( uint16\_t[2] ) { x++, x++ } );

}

**Xem thêm**  
Quy tắc 13.2

### Quy tắc 13.2: Giá trị của một biểu thức và các tác động phụ dai dẳng của nó phải giống nhau trong tất cả các thứ tự đánh giá được phép

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Ví dụ**  
Khi macro COPY\_ELEMENT được gọi trong ví dụ không hợp lệ sau đây, i được đọc hai lần và sửa đổi hai lần. Thứ tự thực hiện các phép toán trên i là không xác định:

* Đọc, sửa đổi, đọc, sửa đổi, hoặc
* Đọc, đọc, sửa đổi, sửa đổi.

|  |
| --- |
|  |

#define COPY\_ELEMENT ( index ) ( a[( index )] = b[( index )] )

COPY\_ELEMENT ( i++ );

Trong ví dụ không hợp lệ này, thứ tự mà v1 và v2 được đọc là không xác định:

|  |
| --- |
|  |

extern volatile uint16\_t v1, v2;

uint16\_t t;

t = v1 + v2;

Trong ví dụ hợp lệ này, PORT được đọc và sửa đổi:

|  |
| --- |
|  |

extern volatile uint8\_t PORT;

PORT = PORT & 0x80u;

Thứ tự đánh giá các tham số hàm là không xác định, cũng như thứ tự xảy ra của các tác động phụ như được chỉ ra trong ví dụ không hợp lệ sau:

|  |
| --- |
|  |

uint16\_t i = 0;

/\*

\* Không xác định liệu lời gọi này tương đương với:

\* f ( 0, 0 )

\* hoặc f ( 0, 1 )

\*/

f ( i++, i );

Thứ tự tương đối của đánh giá một hàm chỉ định và các tham số hàm cũng là không xác định. Trong ví dụ không hợp lệ này, nếu lời gọi đến g sửa đổi p, thì không xác định liệu hàm chỉ định p->f sử dụng giá trị của p trước hay sau lời gọi g.

|  |
| --- |
|  |

p->f ( g ( &p ) );

**Xem thêm**  
Hướng dẫn 4.9, Quy tắc 13.1, Quy tắc 13.3, Quy tắc 13.4

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 7–9; Không xác định 18], **C99** [Không xác định 15–18; Không xác định 32]  
**Loại**: Bắt buộc  
**Phân tích**: Không quyết định được, Hệ thống  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Giữa bất kỳ hai điểm trình tự liền kề hoặc trong bất kỳ biểu thức đầy đủ nào:

1. Không đối tượng nào được phép sửa đổi nhiều hơn một lần;
2. Không đối tượng nào vừa được sửa đổi vừa được đọc trừ khi bất kỳ lần đọc nào như vậy đóng góp vào việc tính toán giá trị được lưu trữ vào đối tượng;
3. Không được có hơn một lần truy cập sửa đổi với loại được định nghĩa volatile;
4. Không được có hơn một lần truy cập đọc với loại được định nghĩa volatile.

**Lưu ý**: Một đối tượng có thể được truy cập gián tiếp, bằng con trỏ hoặc hàm được gọi, cũng như được truy cập trực tiếp bởi biểu thức.

**Lưu ý**: Phần mở rộng này nghiêm ngặt hơn so với tiêu đề của quy tắc. Do đó, các biểu thức như:

c

Sao chép mã

x = x = 0;

không được phép theo quy tắc này, ngay cả khi giá trị và các tác động phụ dai dẳng, với điều kiện x không phải là volatile, không phụ thuộc vào thứ tự đánh giá hoặc các tác động phụ.

Các điểm trình tự được tóm tắt trong Phụ lục C của cả tiêu chuẩn C90 và C99. Các điểm trình tự trong C90 là một tập con của những điểm trong C99.  
Các biểu thức đầy đủ được định nghĩa trong Mục 6.6 của tiêu chuẩn C90 và Mục 6.8 của tiêu chuẩn C99.

**Lý do**  
Tiêu chuẩn cho phép trình biên dịch có sự linh hoạt đáng kể khi đánh giá các biểu thức. Hầu hết các toán tử có thể có các toán hạng của chúng được đánh giá theo bất kỳ thứ tự nào. Các ngoại lệ chính là:

* Toán tử && logic, trong đó toán hạng thứ hai chỉ được đánh giá nếu toán hạng thứ nhất khác không;
* Toán tử || logic, trong đó toán hạng thứ hai chỉ được đánh giá nếu toán hạng thứ nhất bằng không;
* Toán tử điều kiện ?:, trong đó toán hạng thứ nhất luôn được đánh giá và sau đó là toán hạng thứ hai hoặc thứ ba được đánh giá;
* Toán tử dấu phẩy ,, trong đó toán hạng thứ nhất được đánh giá và sau đó toán hạng thứ hai được đánh giá.

**Lưu ý**: Sự hiện diện của dấu ngoặc đơn có thể thay đổi thứ tự áp dụng các toán tử. Tuy nhiên, điều này không ảnh hưởng đến thứ tự đánh giá của các toán hạng mức thấp nhất, chúng có thể được đánh giá theo bất kỳ thứ tự nào.

Việc tuân theo lời khuyên của Quy tắc 13.3 và Quy tắc 13.4 có thể tránh được nhiều trường hợp hành vi không thể đoán trước liên quan đến đánh giá biểu thức.

Ví dụ

Khi macro COPY\_ELEMENT được gọi trong ví dụ không hợp lệ sau, i được đọc hai lần và sửa đổi hai lần. Thứ tự thực hiện các phép toán trên i là không xác định:

* Đọc, sửa đổi, đọc, sửa đổi, hoặc
* Đọc, đọc, sửa đổi, sửa đổi.

|  |
| --- |
|  |

#define COPY\_ELEMENT ( index ) ( a[( index )] = b[( index )] )

COPY\_ELEMENT ( i++ );

Trong ví dụ không hợp lệ này, thứ tự đọc v1 và v2 là không xác định:

|  |
| --- |
|  |

extern volatile uint16\_t v1, v2;

uint16\_t t;

t = v1 + v2;

Trong ví dụ hợp lệ này, PORT được đọc và sửa đổi:

|  |
| --- |
|  |

extern volatile uint8\_t PORT;

PORT = PORT & 0x80u;

Thứ tự đánh giá các tham số hàm là không xác định, cũng như thứ tự xảy ra của các tác động phụ, như được minh họa trong ví dụ không hợp lệ sau:

|  |
| --- |
|  |

uint16\_t i = 0;

/\*

\* Không xác định liệu lời gọi này tương đương với:

\* f ( 0, 0 )

\* hoặc f ( 0, 1 )

\*/

f ( i++, i );

Thứ tự tương đối của việc đánh giá một hàm chỉ định và các tham số hàm cũng là không xác định. Trong ví dụ không hợp lệ này, nếu lời gọi tới g sửa đổi p, thì không xác định liệu hàm chỉ định p->f sử dụng giá trị của p trước hay sau lời gọi tới g.

|  |
| --- |
|  |

p->f ( g ( &p ) );

**Xem thêm**  
Hướng dẫn 4.9, Quy tắc 13.1, Quy tắc 13.3, Quy tắc 13.4

### Quy tắc 13.3:

Một biểu thức đầy đủ chứa toán tử tăng (++) hoặc giảm (--) không nên có các tác động phụ tiềm năng khác ngoài những tác động do toán tử tăng hoặc giảm gây ra

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 7, 8; Không xác định 18], **C99** [Không xác định 15; Không xác định 32]  
**Loại**: Khuyến nghị  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Một lời gọi hàm được xem là một tác động phụ theo quy tắc này.  
Tất cả các biểu thức con của biểu thức đầy đủ được coi là đã được đánh giá theo quy tắc này, ngay cả khi tiêu chuẩn chỉ rõ rằng chúng không được đánh giá.

**Lý do**  
Việc sử dụng toán tử tăng và giảm kết hợp với các toán tử khác không được khuyến khích vì:

* Nó có thể làm giảm đáng kể khả năng đọc hiểu của mã;
* Nó thêm các tác động phụ vào một câu lệnh với khả năng dẫn đến hành vi không xác định (được đề cập trong Quy tắc 13.2).

Việc sử dụng các toán tử này độc lập với các toán tử khác sẽ rõ ràng hơn.

**Ví dụ**

Biểu thức sau:

|  |
| --- |
|  |

u8a = u8b++;

là không hợp lệ. Câu lệnh biểu thức không hợp lệ:

|  |
| --- |
|  |

u8a = ++u8b + u8c--;

sẽ rõ ràng hơn khi được viết thành chuỗi lệnh sau:

|  |
| --- |
|  |

++u8b;

u8a = u8b + u8c;

u8c--;

Các ví dụ sau đây đều hợp lệ vì tác động phụ duy nhất trong mỗi biểu thức là do toán tử tăng hoặc giảm gây ra:

|  |
| --- |
|  |

x++;

a[i]++;

b.x++;

c->x++;

++(\*p);

\*p++;

(\*p)++;

Hy vọng bản dịch này giúp bạn hiểu rõ hơn về các quy tắc và ví dụ khi lập trình C theo chuẩn MISRA.

### Các ví dụ không hợp lệ

Tất cả các ví dụ sau đây đều không hợp lệ vì chúng chứa lời gọi hàm cũng như toán tử tăng hoặc giảm:

|  |
| --- |
|  |

if ( ( f ( ) + --u8a ) == 0u )

{

}

g ( u8b++ );

Tất cả các ví dụ sau đây đều không hợp lệ ngay cả khi biểu thức con chứa toán tử tăng hoặc giảm hoặc một tác động phụ khác không được đánh giá:

|  |
| --- |
|  |

u8a = ( 1u == 1u ) ? 0u : u8b++;

if ( u8a++ == ( ( 1u == 1u ) ? 0u : f ( ) ) )

{

}

**Xem thêm**  
Quy tắc 13.2

### Quy tắc 13.4: Kết quả của một toán tử gán không nên được sử dụng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 7, 8; Không xác định 18], **C99** [Không xác định 15, 18; Không xác định 32]  
[Koenig 6]  
**Loại**: Khuyến nghị  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Quy tắc này áp dụng ngay cả khi biểu thức chứa toán tử gán không được đánh giá.

**Lý do**

Việc sử dụng các toán tử gán, đơn giản hoặc phức tạp, kết hợp với các toán tử số học khác không được khuyến khích vì:

* Nó có thể làm giảm đáng kể khả năng đọc hiểu của mã;
* Nó thêm các tác động phụ vào một câu lệnh, làm khó khăn hơn trong việc tránh hành vi không xác định được đề cập trong Quy tắc 13.2.

**Ví dụ**

|  |
| --- |
|  |

x = y; /\* Hợp lệ \*/

a[ x ] = a[ x = y ]; /\* Không hợp lệ - giá trị của x = y được sử dụng \*/

/\*

\* Không hợp lệ - giá trị của bool\_var = false được sử dụng nhưng

\* có thể bool\_var == false mới là ý định.

\*/

if ( bool\_var = false )

{

}

/\* Không hợp lệ ngay cả khi bool\_var = true không được đánh giá \*/

if ( ( 0u == 0u ) || ( bool\_var = true ) )

{

}

/\* Không hợp lệ - giá trị của x = f() được sử dụng \*/

if ( ( x = f ( ) ) != 0 )

{

}

/\* Không hợp lệ - giá trị của b += c được sử dụng \*/

a[ b += c ] = a[ b ];

/\* Không hợp lệ - giá trị của c = 0 và b = c = 0 được sử dụng \*/

a = b = c = 0;

**Xem thêm**  
Quy tắc 13.2

Quy tắc 13.5: Toán hạng bên phải của toán tử logic && hoặc || không được chứa các tác động phụ dai dẳng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Không quyết định được, Hệ thống  
**Áp dụng cho**: C90, C99

**Lý do**

Việc đánh giá toán hạng bên phải của các toán tử && và || phụ thuộc vào giá trị của toán hạng bên trái. Nếu toán hạng bên phải chứa các tác động phụ, thì các tác động phụ đó có thể xảy ra hoặc không xảy ra, điều này có thể trái với kỳ vọng của lập trình viên.

Nếu việc đánh giá toán hạng bên phải tạo ra các tác động phụ không dai dẳng tại thời điểm xuất hiện của biểu thức trong chương trình, thì việc toán hạng bên phải có được đánh giá hay không sẽ không quan trọng.

Thuật ngữ **tác động phụ dai dẳng** được định nghĩa trong Phụ lục J.

Hy vọng bản dịch này giúp bạn hiểu rõ hơn về các quy tắc lập trình C theo chuẩn MISRA.

Ví dụ

|  |
| --- |
|  |

uint16\_t f ( uint16\_t y )

{

/\* Các tác động phụ này không dai dẳng khi nhìn từ góc độ người gọi \*/

uint16\_t temp = y;

temp = y + 0x8080U;

return temp;

}

uint16\_t h ( uint16\_t y )

{

static uint16\_t temp = 0;

/\* Tác động phụ này là dai dẳng \*/

temp = y + temp;

return temp;

}

void g ( void )

{

/\* Hợp lệ - f() không có tác động phụ dai dẳng \*/

if ( ishigh && ( a == f ( x ) ) )

{

}

/\* Không hợp lệ - h() có tác động phụ dai dẳng \*/

if ( ishigh && ( a == h ( x ) ) )

{

}

}

volatile uint16\_t v;

uint16\_t x;

/\* Không hợp lệ - truy cập đến volatile v là dai dẳng \*/

if ( ( x == 0u ) || ( v == 1u ) )

{

}

/\* Không hợp lệ nếu fp trỏ đến một hàm có tác động phụ dai dẳng \*/

( fp != NULL ) && ( \*fp ) ( 0 );

### Quy tắc 13.6: Toán hạng của toán tử sizeof không được chứa bất kỳ biểu thức nào có khả năng gây ra tác động phụ

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C99** [Không xác định 21]  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Bất kỳ biểu thức nào xuất hiện trong toán hạng của toán tử sizeof thường không được đánh giá. Quy tắc này yêu cầu rằng việc đánh giá bất kỳ biểu thức nào như vậy không được chứa tác động phụ, dù cho biểu thức đó có được đánh giá hay không.

Lời gọi hàm được xem là một tác động phụ theo quy tắc này.

**Lý do**  
Toán hạng của toán tử sizeof có thể là một biểu thức hoặc có thể chỉ định một kiểu. Nếu toán hạng chứa một biểu thức, một lỗi lập trình có thể xảy ra khi lập trình viên mong đợi biểu thức đó được đánh giá trong khi thực tế không phải lúc nào cũng vậy.

Tiêu chuẩn C90 quy định rằng các biểu thức xuất hiện trong toán hạng không được đánh giá tại thời gian chạy.  
Trong C99, các biểu thức xuất hiện trong toán hạng thường không được đánh giá tại thời gian chạy. Tuy nhiên, nếu toán hạng chứa một kiểu mảng có độ dài thay đổi thì biểu thức kích thước mảng sẽ được đánh giá nếu cần. Nếu kết quả có thể được xác định mà không cần đánh giá biểu thức kích thước mảng thì việc đánh giá có được thực hiện hay không là không xác định.

**Ngoại lệ**  
Một biểu thức có dạng sizeof ( V ), trong đó V là một giá trị lvalue với loại được định nghĩa volatile không phải là mảng có độ dài thay đổi, được cho phép.

**Ví dụ**

|  |
| --- |
|  |

volatile int32\_t i;

int32\_t j;

size\_t s;

s = sizeof ( j ); /\* Hợp lệ \*/

s = sizeof ( j++ ); /\* Không hợp lệ \*/

s = sizeof ( i ); /\* Hợp lệ - ngoại lệ \*/

s = sizeof ( int32\_t ); /\* Hợp lệ \*/

Trong ví dụ này, biểu thức sizeof cuối cùng minh họa cách một biểu thức kích thước mảng có độ dài thay đổi có thể không ảnh hưởng đến kích thước của kiểu. Toán hạng là kiểu "mảng các con trỏ tới hàm với tham số kiểu mảng của v int32\_t". Vì toán hạng có kiểu mảng có độ dài thay đổi, nên nó được đánh giá. Tuy nhiên, kích thước của mảng các con trỏ hàm không bị ảnh hưởng bởi danh sách tham số của các kiểu con trỏ hàm đó. Do đó, đối tượng được định nghĩa volatile là v có thể được đánh giá hoặc không, và các tác động phụ của nó có thể hoặc không xảy ra.

|  |
| --- |
|  |

volatile uint32\_t v;

void f ( int32\_t n )

{

size\_t s;

s = sizeof ( int32\_t[ n ] ); /\* Hợp lệ \*/

s = sizeof ( int32\_t[ n++ ] ); /\* Không hợp lệ \*/

s = sizeof ( void ( \*[ n ] ) ( int32\_t a[ v ] ) ); /\* Không hợp lệ \*/

}

**Xem thêm**  
Quy tắc 18.8

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách tuân thủ khi lập trình C theo chuẩn MISRA.

### 8.14 Biểu thức trong câu lệnh điều khiển

Thuật ngữ **biến đếm vòng lặp** được sử dụng trong một số quy tắc trong phần này. Một biến đếm vòng lặp được định nghĩa là một đối tượng, phần tử mảng hoặc thành viên của một cấu trúc hoặc hợp mà thỏa mãn các điều kiện sau:

1. Có kiểu dữ liệu dạng vô hướng (scalar);
2. Giá trị của nó thay đổi đơn điệu trong mỗi lần lặp của một phiên bản cụ thể của vòng lặp; và
3. Nó tham gia vào quyết định thoát khỏi vòng lặp.

**Lưu ý**: Điều kiện thứ hai có nghĩa là giá trị của biến đếm vòng lặp phải thay đổi trong mỗi lần lặp của vòng lặp và luôn thay đổi theo cùng một hướng trong một phiên bản cụ thể của vòng lặp. Tuy nhiên, nó có thể thay đổi theo hướng khác nhau trong các phiên bản khác nhau, ví dụ như đôi khi đọc các phần tử của một mảng từ cuối đến đầu và đôi khi đọc từ đầu đến cuối.

Theo định nghĩa này, một vòng lặp không nhất thiết phải có chỉ một biến đếm vòng lặp: có thể có vòng lặp không có biến đếm hoặc có nhiều hơn một biến đếm. Xem **Quy tắc 14.2** để biết thêm các hạn chế về biến đếm trong vòng lặp for.

### Ví dụ về các đoạn mã với vòng lặp và biến đếm tương ứng

Trong vòng lặp sau, i là một biến đếm vòng lặp vì nó có kiểu dữ liệu vô hướng, thay đổi đơn điệu (tăng dần) và tham gia vào điều kiện kết thúc vòng lặp.

c

Sao chép mã

for ( uint16\_t i = 0; a[i] < 10; ++i )

{

}

Trong vòng lặp sau, không có biến đếm vòng lặp. Đối tượng count có kiểu vô hướng và thay đổi đơn điệu nhưng không tham gia vào điều kiện kết thúc vòng lặp.

c

Sao chép mã

extern volatile bool\_t b;

uint16\_t count = 0;

while ( b )

{

count = count + 1U;

}

Trong đoạn mã sau, cả i và sum đều có kiểu vô hướng và thay đổi đơn điệu (giảm dần và tăng dần tương ứng). Tuy nhiên, sum không phải là biến đếm vòng lặp vì nó không tham gia vào quyết định thoát khỏi vòng lặp.

c

Sao chép mã

uint16\_t sum = 0;

for ( uint16\_t i = 10U; i != 0U; --i )

{

sum += i;

}

Trong vòng lặp sau, p là một biến đếm vòng lặp. Nó không tham gia vào biểu thức điều khiển vòng lặp nhưng tham gia vào quyết định thoát khỏi vòng lặp thông qua lệnh break.

c

Sao chép mã

extern volatile bool\_t b;

extern char \*p;

do

{

if ( \*p == '\0' )

{

break;

}

++p;

} while ( b );

Biến đếm vòng lặp trong ví dụ sau là p->count.

c

Sao chép mã

struct s

{

uint16\_t count;

uint16\_t a[10];

};

extern struct s \*p;

for ( p->count = 0U; p->count < 10U; ++( p->count ) )

{

p->a[p->count] = 0U;

}

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về khái niệm và các ví dụ về biến đếm vòng lặp trong lập trình C theo chuẩn MISRA.

### Quy tắc 14.1: Một biến đếm vòng lặp không được có kiểu dữ liệu cơ bản là số thực (floating type)

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Không quyết định được, Hệ thống  
**Áp dụng cho**: C90, C99

**Lý do**  
Khi sử dụng biến đếm vòng lặp kiểu số thực, việc tích lũy các lỗi làm tròn có thể dẫn đến sự khác biệt giữa số lần lặp dự kiến và thực tế. Điều này có thể xảy ra khi bước lặp không phải là lũy thừa của cơ số số thực và bị làm tròn về một giá trị có thể biểu diễn được.

Ngay cả khi một vòng lặp với biến đếm kiểu số thực có vẻ hoạt động đúng trên một trình biên dịch, nó có thể cho số lần lặp khác nhau trên một trình biên dịch khác.

**Ví dụ**

Trong ví dụ không hợp lệ sau, giá trị của counter có thể không đạt đến 1.000 sau khi vòng lặp kết thúc:

|  |
| --- |
|  |

uint32\_t counter = 0u;

for ( float32\_t f = 0.0f; f < 1.0f; f += 0.001f )

{

++counter;

}

Ví dụ hợp lệ sau sử dụng biến đếm vòng lặp kiểu nguyên để đảm bảo 1.000 lần lặp và sử dụng nó để tạo giá trị f dùng trong vòng lặp:

|  |
| --- |
|  |

float32\_t f;

for ( uint32\_t counter = 0u; counter < 1000u; ++counter )

{

f = ( float32\_t ) counter \* 0.001f;

}

Vòng lặp while sau là không hợp lệ vì f được sử dụng làm biến đếm vòng lặp:

|  |
| --- |
|  |

float32\_t f = 0.0f;

while ( f < 1.0f )

{

f += 0.001f;

}

Vòng lặp while sau là hợp lệ vì f không được sử dụng làm biến đếm vòng lặp:

|  |
| --- |
|  |

float32\_t f;

uint32\_t u32a;

f = read\_float32();

do

{

u32a = read\_u32();

/\* f không thay đổi trong vòng lặp nên không thể là biến đếm vòng lặp \*/

} while ( ( ( float32\_t ) u32a - f ) > 10.0f );

**Xem thêm**  
Quy tắc 14.2

### Quy tắc 14.2: Vòng lặp for phải được định dạng đúng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Không quyết định được, Hệ thống  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Ba phần của câu lệnh for bao gồm:

* **Phần đầu tiên**:
  + Phải để trống, hoặc
  + Phải gán giá trị cho biến đếm vòng lặp, hoặc
  + Phải định nghĩa và khởi tạo biến đếm vòng lặp (C99).
* **Phần thứ hai**:
  + Phải là một biểu thức không có tác động phụ dai dẳng, và
  + Phải sử dụng biến đếm vòng lặp và tùy chọn thêm cờ điều khiển vòng lặp, và
  + Không được sử dụng bất kỳ đối tượng nào khác bị sửa đổi trong thân vòng lặp for.
* **Phần thứ ba**:
  + Phải là một biểu thức với tác động phụ dai dẳng duy nhất là thay đổi giá trị của biến đếm vòng lặp, và
  + Không được sử dụng các đối tượng bị sửa đổi trong thân vòng lặp for.

Chỉ được có một biến đếm vòng lặp trong vòng lặp for, và biến này không được sửa đổi trong thân vòng lặp.

**Cờ điều khiển vòng lặp** được định nghĩa là một định danh đơn biểu thị một đối tượng có kiểu dữ liệu cơ bản là kiểu Boolean và được sử dụng trong **Phần thứ hai**.

Hành vi của thân vòng lặp for bao gồm cả hành vi của bất kỳ hàm nào được gọi trong câu lệnh đó.

**Lý do**  
Câu lệnh for cung cấp một cơ chế lặp tổng quát. Sử dụng một dạng vòng lặp hạn chế giúp mã dễ xem xét và phân tích hơn.

**Ngoại lệ**  
Cả ba phần có thể để trống, ví dụ for ( ; ; ), để cho phép vòng lặp vô hạn.

**Ví dụ**

Trong ví dụ C99 sau, i là biến đếm vòng lặp và flag là cờ điều khiển vòng lặp:

|  |
| --- |
|  |

bool\_t flag = false;

for ( int16\_t i = 0; ( i < 5 ) && !flag; i++ )

{

if ( C )

{

flag = true; /\* Hợp lệ - cho phép thoát sớm khỏi vòng lặp \*/

}

i = i + 3; /\* Không hợp lệ - thay đổi biến đếm vòng lặp \*/

}

**Xem thêm**  
Quy tắc 14.1, Quy tắc 14.3, Quy tắc 14.4

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và ví dụ khi lập trình C theo chuẩn MISRA.

### Quy tắc 14.3: Biểu thức điều khiển không được bất biến

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Không quyết định được, Hệ thống  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Quy tắc này áp dụng cho:

* Biểu thức điều khiển của các câu lệnh if, while, for, do … while và switch;
* Toán hạng đầu tiên của toán tử ?:.

**Lý do**  
Nếu một biểu thức điều khiển có giá trị bất biến, có thể có lỗi lập trình. Bất kỳ mã nào không thể đạt được do sự hiện diện của một biểu thức bất biến có thể bị loại bỏ bởi trình biên dịch. Điều này có thể dẫn đến việc loại bỏ mã phòng thủ, chẳng hạn, khỏi tệp thực thi.

**Ngoại lệ**

1. Các biểu thức bất biến được sử dụng để tạo vòng lặp vô hạn được cho phép.
2. Vòng lặp do … while với biểu thức điều khiển có kiểu cơ bản là Boolean và đánh giá bằng 0 được cho phép.

**Ví dụ**

|  |
| --- |
|  |

s8a = ( u16a < 0u ) ? 0 : 1; /\* Không hợp lệ - u16a luôn >= 0 \*/

if ( u16a <= 0xffffu )

{

/\* Không hợp lệ - luôn đúng \*/

}

if ( 2 > 3 )

{

/\* Không hợp lệ - luôn sai \*/

}

for ( s8a = 0; s8a < 130; ++s8a )

{

/\* Không hợp lệ - luôn đúng \*/

}

if ( ( s8a < 10 ) && ( s8a > 20 ) )

{

/\* Không hợp lệ - luôn sai \*/

}

if ( ( s8a < 10 ) || ( s8a > 5 ) )

{

/\* Không hợp lệ - luôn đúng \*/

}

while ( s8a > 10 )

{

if ( s8a > 5 )

{

/\* Không hợp lệ - s8a không phải là volatile \*/

}

}

while ( true )

{

/\* Hợp lệ theo ngoại lệ 1 \*/

}

do

{

/\* Hợp lệ theo ngoại lệ 2 \*/

} while ( 0u == 1u );

const uint8\_t numcyl = 4u;

/\*

\* Không hợp lệ - trình biên dịch có thể giả định rằng numcyl luôn

\* có giá trị 4

\*/

if ( numcyl == 4u )

{

}

const volatile uint8\_t numcyl\_cal = 4u;

/\*

\* Hợp lệ - trình biên dịch giả định rằng numcyl\_cal có thể được thay đổi

\* bởi một phương pháp bên ngoài, ví dụ: công cụ hiệu chỉnh trong ô tô,

\* mặc dù chương trình không thể thay đổi giá trị của nó

\*/

if ( numcyl\_cal == 4u )

{

}

uint16\_t n; /\* 10 <= n <= 100 \*/

uint16\_t sum;

sum = 0;

for ( uint16\_t i = ( n - 6u ); i < n; ++i )

{

sum += i;

}

if ( ( sum % 2u ) == 0u )

{

/\*

\* Không hợp lệ - sum là tổng của 6 số nguyên không âm liên tiếp

\* nên phải là số lẻ. Biểu thức điều khiển của câu lệnh if sẽ luôn sai.

\*/

}

**Xem thêm**  
Quy tắc 2.1, Quy tắc 14.2

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng khi lập trình C theo chuẩn MISRA.

### Quy tắc 14.4: Biểu thức điều khiển của câu lệnh if và biểu thức điều khiển của câu lệnh lặp phải có kiểu dữ liệu cơ bản là Boolean

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Biểu thức điều khiển của câu lệnh for là tùy chọn. Quy tắc này không yêu cầu biểu thức phải có mặt nhưng yêu cầu nó phải có kiểu dữ liệu cơ bản là Boolean nếu có mặt.

**Lý do**  
Kiểu dữ liệu mạnh yêu cầu biểu thức điều khiển của câu lệnh if hoặc câu lệnh lặp phải có kiểu dữ liệu cơ bản là Boolean.

**Ví dụ**

|  |
| --- |
|  |

int32\_t \*p, \*q;

while ( p ) /\* Không hợp lệ - p là con trỏ \*/

{

}

while ( q != NULL ) /\* Hợp lệ \*/

{

}

while ( true ) /\* Hợp lệ \*/

{

}

extern bool\_t flag;

while ( flag ) /\* Hợp lệ \*/

{

}

int32\_t i;

if ( i ) /\* Không hợp lệ \*/

{

}

if ( i != 0 ) /\* Hợp lệ \*/

{

}

**Xem thêm**  
Quy tắc 14.2, Quy tắc 20.8

### 8.15 Luồng điều khiển

**Quy tắc 15.1**: Không nên sử dụng câu lệnh goto.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Khuyến nghị  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Lý do**  
Việc sử dụng goto không được kiểm soát có thể dẫn đến các chương trình không có cấu trúc và cực kỳ khó hiểu.

Trong một số trường hợp, việc cấm hoàn toàn goto yêu cầu phải giới thiệu các cờ (flags) để đảm bảo luồng điều khiển đúng, và có thể các cờ này còn ít rõ ràng hơn cả goto mà chúng thay thế. Do đó, nếu quy tắc này không được tuân thủ, thì việc sử dụng goto có thể được chấp nhận nếu tuân theo hướng dẫn trong **Quy tắc 15.2** và **Quy tắc 15.3**.

**Xem thêm**  
Quy tắc 9.1, Quy tắc 15.2, Quy tắc 15.3, Quy tắc 15.4

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng khi lập trình C theo chuẩn MISRA.

### Quy tắc 15.2: Câu lệnh goto phải nhảy đến một nhãn được khai báo sau đó trong cùng hàm

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Lý do**  
Việc sử dụng goto không được kiểm soát có thể dẫn đến các chương trình không có cấu trúc và cực kỳ khó hiểu. Hạn chế việc sử dụng goto để tránh nhảy lùi đảm bảo rằng vòng lặp chỉ xảy ra khi sử dụng các câu lệnh lặp được ngôn ngữ cung cấp, giúp giảm độ phức tạp trực quan của mã nguồn.

**Ví dụ**

|  |
| --- |
|  |

void f ( void )

{

int32\_t j = 0;

L1:

++j;

if ( 10 == j )

{

goto L2; /\* Hợp lệ \*/

}

goto L1; /\* Không hợp lệ \*/

L2:

++j;

}

**Xem thêm**  
Quy tắc 15.1, Quy tắc 15.3, Quy tắc 15.4

### Quy tắc 15.3: Bất kỳ nhãn nào được tham chiếu bởi câu lệnh goto phải được khai báo trong cùng khối hoặc trong bất kỳ khối nào bao quanh câu lệnh goto

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Với mục đích của quy tắc này, một switch-clause không bao gồm một câu lệnh tổng hợp được xem như thể nó là một khối.

**Lý do**  
Việc sử dụng goto không được kiểm soát có thể dẫn đến các chương trình không có cấu trúc và cực kỳ khó hiểu. Ngăn chặn việc nhảy giữa các khối hoặc vào các khối lồng nhau giúp giảm độ phức tạp trực quan của mã nguồn.

**Lưu ý**: C99 hạn chế hơn khi sử dụng các loại biến đổi (variably modified types). Nếu cố gắng nhảy từ bên ngoài phạm vi của một định danh có kiểu biến đổi vào phạm vi đó sẽ dẫn đến vi phạm ràng buộc.

**Ví dụ**

|  |
| --- |
|  |

void f1 ( int32\_t a )

{

if ( a <= 0 )

{

goto L2; /\* Không hợp lệ \*/

}

goto L1; /\* Hợp lệ \*/

if ( a == 0 )

{

goto L1; /\* Hợp lệ \*/

}

goto L2; /\* Không hợp lệ \*/

L1:

if ( a > 0 )

{

L2:

;

}

}

Trong ví dụ sau, nhãn L1 được định nghĩa trong một khối bao quanh khối chứa câu lệnh goto. Tuy nhiên, việc nhảy qua lại giữa các điều khoản switch làm cho câu lệnh goto không hợp lệ vì các điều khoản switch được xem như các khối trong quy tắc này.

|  |
| --- |
|  |

switch ( x )

{

case 0:

if ( x == y )

{

goto L1;

}

break;

case 1:

y = x;

L1:

++x;

break;

default:

break;

}

**Xem thêm**  
Quy tắc 9.1, Quy tắc 15.1, Quy tắc 15.2, Quy tắc 15.4, Quy tắc 16.1

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng khi lập trình C theo chuẩn MISRA.

### Quy tắc 15.4: Không được sử dụng nhiều hơn một câu lệnh break hoặc goto để kết thúc bất kỳ câu lệnh lặp nào

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Khuyến nghị  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Lý do**  
Hạn chế số lần thoát ra khỏi vòng lặp giúp giảm độ phức tạp trực quan của mã nguồn. Việc sử dụng một câu lệnh break hoặc goto cho phép tạo ra một đường thoát thứ cấp duy nhất khi cần kết thúc sớm vòng lặp.

**Ví dụ**

Cả hai vòng lặp lồng nhau sau đây đều hợp lệ vì mỗi vòng lặp chỉ có một câu lệnh break được sử dụng để kết thúc sớm.

|  |
| --- |
|  |

for ( x = 0; x < LIMIT; ++x )

{

if ( ExitNow ( x ) )

{

break;

}

for ( y = 0; y < x; ++y )

{

if ( ExitNow ( LIMIT - y ) )

{

break;

}

}

}

Vòng lặp sau đây không hợp lệ vì có nhiều câu lệnh break và goto được sử dụng để kết thúc sớm vòng lặp.

|  |
| --- |
|  |

for ( x = 0; x < LIMIT; ++x )

{

if ( BreakNow ( x ) )

{

break;

}

else if ( GotoNow ( x ) )

{

goto EXIT;

}

else

{

KeepGoing ( x );

}

}

EXIT:

;

Trong ví dụ sau, vòng lặp while lồng nhau bên trong hợp lệ vì chỉ có một câu lệnh goto có thể kết thúc sớm vòng lặp. Tuy nhiên, vòng lặp while bên ngoài không hợp lệ vì nó có thể bị kết thúc sớm bằng câu lệnh break hoặc bởi câu lệnh goto trong vòng lặp lồng nhau.

|  |
| --- |
|  |

while ( x != 0u )

{

x = calc\_new\_x ( );

if ( x == 1u )

{

break;

}

while ( y != 0u )

{

y = calc\_new\_y ( );

if ( y == 1u )

{

goto L1;

}

}

}

L1:

z = x + y;

**Xem thêm**  
Quy tắc 15.1, Quy tắc 15.2, Quy tắc 15.3

### Quy tắc 15.5: Một hàm nên có một điểm kết thúc duy nhất ở cuối

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**[IEC 61508-3 Table B.9], [ISO 26262-6 Table 8]**  
**Loại**: Khuyến nghị  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**

Một hàm không nên có nhiều hơn một câu lệnh return. Khi sử dụng câu lệnh return, nó nên là câu lệnh cuối cùng trong câu lệnh tổng hợp tạo thành thân hàm.

**Lý do**

Một điểm thoát duy nhất được yêu cầu bởi IEC 61508 và ISO 26262 như một phần của các yêu cầu cho phương pháp tiếp cận module. Việc thoát ra sớm có thể dẫn đến việc vô tình bỏ qua mã kết thúc hàm.

Nếu một hàm có các điểm thoát xen lẫn với các câu lệnh tạo ra tác động phụ dai dẳng, thì sẽ không dễ dàng để xác định các tác động phụ nào sẽ xảy ra khi hàm được thực thi.

**Ví dụ**

Trong ví dụ không hợp lệ sau, các câu lệnh return được sử dụng để kiểm tra tính hợp lệ của tham số hàm.

|  |
| --- |
|  |

bool\_t f ( uint16\_t n, char \*p )

{

if ( n > MAX )

{

return false;

}

if ( p == NULL )

{

return false;

}

return true;

}

**Xem thêm**  
Quy tắc 17.4

### Quy tắc 15.6: Thân của câu lệnh lặp hoặc câu lệnh lựa chọn phải là một câu lệnh tổng hợp

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**[Koenig 24]**  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**

Thân của câu lệnh lặp (while, do … while hoặc for) hoặc câu lệnh lựa chọn (if, else, switch) phải là một câu lệnh tổng hợp.

**Lý do**

Có khả năng lập trình viên tin rằng một chuỗi câu lệnh tạo thành thân của một câu lệnh lặp hoặc lựa chọn dựa trên việc thụt lề. Việc vô tình bao gồm dấu chấm phẩy sau biểu thức điều khiển là một mối nguy hiểm cụ thể, dẫn đến một câu lệnh điều khiển null. Sử dụng câu lệnh tổng hợp giúp xác định rõ ràng các câu lệnh nào thực sự tạo thành thân.

Ngoài ra, việc thụt lề có thể dẫn đến việc lập trình viên gán một câu lệnh else cho câu lệnh if sai.

**Ngoại lệ**

Một câu lệnh if ngay sau else không cần phải nằm trong một câu lệnh tổng hợp.

**Ví dụ**

Cách bố trí cho câu lệnh tổng hợp và dấu ngoặc bao quanh của nó là vấn đề phong cách không được đề cập trong tài liệu này; phong cách được sử dụng trong các ví dụ sau không bắt buộc.

Bảo trì mã sau:

|  |
| --- |
|  |

while ( data\_available )

process\_data ( ); /\* Không hợp lệ \*/

có thể vô tình trở thành:

|  |
| --- |
|  |

while ( data\_available )

process\_data ( ); /\* Không hợp lệ \*/

service\_watchdog ( );

trong đó service\_watchdog() lẽ ra phải được thêm vào thân vòng lặp. Việc sử dụng câu lệnh tổng hợp giúp giảm đáng kể khả năng xảy ra điều này.

Ví dụ tiếp theo dường như chỉ ra rằng action\_2() là câu lệnh else của if đầu tiên.

|  |
| --- |
|  |

if ( flag\_1 )

if ( flag\_2 ) /\* Không hợp lệ \*/

action\_1 ( ); /\* Không hợp lệ \*/

else

action\_2 ( ); /\* Không hợp lệ \*/

nhưng hành vi thực tế là:

|  |
| --- |
|  |

if ( flag\_1 )

{

if ( flag\_2 )

{

action\_1 ( );

}

else

{

action\_2 ( );

}

}

Việc sử dụng câu lệnh tổng hợp đảm bảo rằng mối quan hệ if và else được xác định rõ ràng.

Ngoại lệ cho phép sử dụng else if, như được minh họa bên dưới:

|  |
| --- |
|  |

if ( flag\_1 )

{

action\_1 ( );

}

else if ( flag\_2 ) /\* Hợp lệ theo ngoại lệ \*/

{

action\_2 ( );

}

else

{

;

}

Ví dụ sau đây cho thấy cách dấu chấm phẩy không đáng có có thể dẫn đến lỗi:

|  |
| --- |
|  |

while ( flag ); /\* Không hợp lệ \*/

{

flag = fn ( );

}

Ví dụ sau đây cho thấy phương pháp hợp lệ để viết một vòng lặp với thân rỗng:

|  |
| --- |
|  |

while ( !data\_available )

{

}

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng khi lập trình C theo chuẩn MISRA.

### Quy tắc 15.7: Tất cả các cấu trúc if … else if phải kết thúc bằng một câu lệnh else

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**

Cần phải cung cấp một câu lệnh else cuối cùng bất cứ khi nào một câu lệnh if được theo sau bởi một chuỗi một hoặc nhiều cấu trúc else if. Câu lệnh else phải chứa ít nhất một tác động phụ hoặc một chú thích.

**Lý do**

Việc kết thúc một chuỗi if … else if bằng một câu lệnh else là lập trình phòng ngừa và bổ sung cho yêu cầu về câu lệnh default trong câu lệnh switch (xem Quy tắc 16.5). Câu lệnh else được yêu cầu có tác động phụ hoặc chú thích để đảm bảo có sự chỉ định rõ ràng về hành vi mong muốn, giúp quá trình xem xét mã dễ dàng hơn.

**Lưu ý**: Không cần câu lệnh else cuối cùng cho một câu lệnh if đơn giản.

**Ví dụ**

Ví dụ sau là không hợp lệ vì không có sự chỉ định rõ ràng rằng không cần hành động nào trong phần kết thúc else.

|  |
| --- |
|  |

if ( flag\_1 )

{

action\_1 ( );

}

else if ( flag\_2 )

{

action\_2 ( );

}

/\* Không hợp lệ \*/

Ví dụ sau đây cho thấy cách kết thúc else hợp lệ.

|  |
| --- |
|  |

else

{

; /\* Không cần hành động - ; là tùy chọn \*/

}

**Xem thêm**  
Quy tắc 16.5

### 8.16 Câu lệnh switch

**Quy tắc 16.1**: Tất cả các câu lệnh switch phải được định dạng đúng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Một câu lệnh switch được coi là định dạng đúng nếu nó tuân theo tập hợp con của các câu lệnh switch trong C được chỉ định bởi các quy tắc cú pháp sau. Nếu một quy tắc cú pháp được đưa ra ở đây có cùng tên với một quy tắc được định nghĩa trong Tiêu chuẩn, thì nó thay thế phiên bản chuẩn cho phạm vi của câu lệnh switch; nếu không, tất cả các quy tắc cú pháp được đưa ra trong Tiêu chuẩn không thay đổi.

**Cú pháp câu lệnh switch**

|  |
| --- |
|  |

switch-statement:

switch ( switch-expression ) { case-label-clause-list final-default-clause-list }

switch ( switch-expression ) { initial-default-clause-list case-label-clause-list }

case-label-clause-list:

case-clause-list

case-label-clause-list case-clause-list

case-clause-list:

case-label switch-clause

case-label case-clause-list

case-label:

case constant-expression:

final-default-clause-list:

default: switch-clause

case-label final-default-clause-list

initial-default-clause-list:

default: switch-clause

default: case-clause-list

switch-clause:

statement-listopt break;

C90: { declaration-listopt statement-listopt break; }

C99: { block-item-listopt break; }

Ngoại trừ những trường hợp được cho phép rõ ràng bởi cú pháp này, các từ khóa case và default không được xuất hiện ở bất kỳ đâu trong thân câu lệnh switch.

**Lưu ý**: Một số hạn chế áp đặt lên câu lệnh switch bởi quy tắc này được mở rộng trong các quy tắc được tham khảo trong phần “Xem thêm”. Do đó, có thể mã vi phạm cả quy tắc này và một trong những quy tắc cụ thể hơn.

**Lưu ý**: Thuật ngữ **nhãn switch** được sử dụng trong văn bản của các quy tắc câu lệnh switch cụ thể để chỉ một nhãn case hoặc một nhãn default.

**Lý do**

Cú pháp cho câu lệnh switch trong C không đặc biệt nghiêm ngặt và có thể cho phép hành vi phức tạp, không có cấu trúc. Quy tắc này và các quy tắc khác áp dụng một cấu trúc đơn giản và nhất quán cho câu lệnh switch.

**Ví dụ**

Các quy tắc còn lại trong phần này đưa ra các ví dụ cũng liên quan đến quy tắc này.

**Xem thêm**  
Quy tắc 15.3, Quy tắc 16.2, Quy tắc 16.3, Quy tắc 16.4, Quy tắc 16.5, Quy tắc 16.6

### Quy tắc 16.2: Một nhãn switch chỉ được sử dụng khi câu lệnh tổng hợp gần nhất bao quanh là thân của câu lệnh switch

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Lý do**  
Tiêu chuẩn cho phép một nhãn switch, tức là một nhãn case hoặc default, được đặt trước bất kỳ câu lệnh nào trong thân của câu lệnh switch, điều này có thể dẫn đến mã không có cấu trúc. Để ngăn chặn điều này, một nhãn switch chỉ được xuất hiện ở cấp độ ngoài cùng của câu lệnh tổng hợp tạo thành thân của câu lệnh switch.

**Ví dụ**

|  |
| --- |
|  |

switch ( x )

{

case 1: /\* Hợp lệ \*/

if ( flag )

{

case 2: /\* Không hợp lệ \*/

x = 1;

}

break;

default:

break;

}

**Xem them**

Quy tắc 16.1

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng khi lập trình C theo chuẩn MISRA.

### Quy tắc 16.3: Một câu lệnh break không điều kiện phải kết thúc mọi điều khoản switch

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Lý do**

Nếu một lập trình viên không kết thúc một điều khoản switch bằng câu lệnh break, thì luồng điều khiển sẽ "rơi" vào điều khoản switch tiếp theo hoặc, nếu không có điều khoản nào tiếp theo, sẽ tiếp tục vào câu lệnh sau switch. Mặc dù việc rơi vào điều khoản switch tiếp theo đôi khi có chủ ý, nhưng thường là một lỗi. Một điều khoản switch không kết thúc ở cuối câu lệnh switch có thể rơi vào bất kỳ điều khoản nào được thêm sau này.

Để đảm bảo phát hiện được các lỗi như vậy, câu lệnh cuối cùng trong mọi điều khoản switch phải là câu lệnh break, hoặc nếu điều khoản switch là một câu lệnh tổng hợp, câu lệnh cuối cùng trong câu lệnh tổng hợp phải là một câu lệnh break.

**Lưu ý**: Một điều khoản switch được định nghĩa là chứa ít nhất một câu lệnh. Hai nhãn liên tiếp, case hoặc default, không có câu lệnh xen giữa và do đó được phép bởi quy tắc này.

**Ví dụ**

|  |
| --- |
|  |

switch ( x )

{

case 0:

break; /\* Hợp lệ - break không điều kiện \*/

case 1: /\* Hợp lệ - rơi qua nhóm trống \*/

case 2:

break; /\* Hợp lệ \*/

case 4:

a = b; /\* Không hợp lệ - thiếu break \*/

case 5:

if ( a == b )

{

++a;

break; /\* Không hợp lệ - break có điều kiện \*/

}

default:

; /\* Không hợp lệ - default cũng phải có break \*/

}

**Xem them**

Quy tắc 16.1

### Quy tắc 16.4: Mọi câu lệnh switch phải có nhãn default

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**

Điều khoản switch sau nhãn default phải chứa, trước câu lệnh break kết thúc:

* Một câu lệnh, hoặc
* Một chú thích.

**Lý do**

Yêu cầu về một nhãn default là lập trình phòng ngừa. Bất kỳ câu lệnh nào sau nhãn default đều nhằm mục đích thực hiện hành động phù hợp. Nếu không có câu lệnh nào sau nhãn, chú thích có thể được sử dụng để giải thích lý do tại sao không thực hiện hành động cụ thể nào.

**Ví dụ**

|  |
| --- |
|  |

int16\_t x;

switch ( x )

{

case 0:

++x;

break;

case 1:

case 2:

break;

/\* Không hợp lệ - cần nhãn default \*/

}

int16\_t x;

switch ( x )

{

case 0:

++x;

break;

case 1:

case 2:

break;

default: /\* Hợp lệ - nhãn default có mặt \*/

errorflag = 1; /\* nên không rỗng nếu có thể \*/

break;

}

enum Colours

{ RED, GREEN, BLUE } colour;

switch ( colour )

{

case RED:

next = GREEN;

break;

case GREEN:

next = BLUE;

break;

case BLUE:

next = RED;

break;

/\* Không hợp lệ - không có nhãn default.

\* Mặc dù tất cả các giá trị của kiểu liệt kê đã được xử lý,

\* không có gì đảm bảo rằng biến colour có một trong những giá trị đó \*/

}

**Xem them**

Quy tắc 2.1, Quy tắc 16.1

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng khi lập trình C theo chuẩn MISRA.

### Quy tắc 16.5: Một nhãn default phải xuất hiện như nhãn đầu tiên hoặc cuối cùng của một câu lệnh switch

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Lý do**

Quy tắc này giúp dễ dàng xác định vị trí của nhãn default trong một câu lệnh switch.

**Ví dụ**

|  |
| --- |
|  |

switch ( x )

{

default: /\* Hợp lệ - nhãn default là nhãn đầu tiên \*/

case 0:

++x;

break;

case 1:

case 2:

break;

}

switch ( x )

{

case 0:

++x;

break;

default: /\* Không hợp lệ - nhãn default được trộn với các nhãn case \*/

x = 0;

break;

case 1:

case 2:

break;

}

switch ( x )

{

case 0:

++x;

break;

case 1:

case 2:

break;

default: /\* Hợp lệ - nhãn default là nhãn cuối cùng \*/

x = 0;

break;

}

**Xem them**

Quy tắc 15.7, Quy tắc 16.1

### Quy tắc 16.6: Mọi câu lệnh switch phải có ít nhất hai điều khoản switch

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Lý do**

Một câu lệnh switch với chỉ một đường dẫn là dư thừa và có thể là dấu hiệu của một lỗi lập trình.

**Ví dụ**

|  |
| --- |
|  |

switch ( x )

{

default: /\* Không hợp lệ - câu lệnh switch là dư thừa \*/

x = 0;

break;

}

switch ( y )

{

case 1:

default: /\* Không hợp lệ - câu lệnh switch là dư thừa \*/

y = 0;

break;

}

switch ( z )

{

case 1:

z = 2;

break;

default: /\* Hợp lệ \*/

z = 0;

break;

}

**Xem them**

Quy tắc 16.1

### Quy tắc 16.7: Biểu thức điều khiển của một câu lệnh switch không được có kiểu dữ liệu cơ bản là Boolean

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Lý do**

Tiêu chuẩn yêu cầu biểu thức điều khiển của một câu lệnh switch phải có kiểu số nguyên. Vì kiểu dữ liệu được sử dụng để biểu diễn các giá trị Boolean là kiểu số nguyên, nên có thể có câu lệnh switch được điều khiển bởi một biểu thức Boolean. Trong trường hợp này, cấu trúc if-else sẽ phù hợp hơn.

**Ví dụ**

|  |
| --- |
|  |

switch ( x == 0 ) /\* Không hợp lệ - kiểu cơ bản là Boolean \*/

{ /\* Trong trường hợp này, "if-else" sẽ hợp lý hơn \*/

case false:

y = x;

break;

default:

y = z;

break;

}

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng khi lập trình C theo chuẩn MISRA.

### 8.17 Hàm

**Quy tắc 17.1**: Các tính năng của <stdarg.h> không được sử dụng.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 45, 70–76], **C99** [Không xác định 81, 128–135]  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**

Không được sử dụng va\_list, va\_arg, va\_start, va\_end và, đối với C99, va\_copy.

**Lý do**

Tiêu chuẩn liệt kê nhiều trường hợp hành vi không xác định liên quan đến các tính năng của <stdarg.h>, bao gồm:

* Không sử dụng va\_end trước khi kết thúc hàm trong đó va\_start được sử dụng;
* Sử dụng va\_arg trong các hàm khác nhau trên cùng một va\_list;
* Kiểu của một đối số không tương thích với kiểu được chỉ định cho va\_arg.

**Ví dụ**

|  |
| --- |
|  |

#include <stdarg.h>

void h ( va\_list ap ) /\* Không hợp lệ \*/

{

double y;

y = va\_arg ( ap, double ); /\* Không hợp lệ \*/

}

void f ( uint16\_t n, ... )

{

uint32\_t x;

va\_list ap; /\* Không hợp lệ \*/

va\_start ( ap, n ); /\* Không hợp lệ \*/

x = va\_arg ( ap, uint32\_t ); /\* Không hợp lệ \*/

h ( ap );

/\* không xác định - ap không xác định do va\_arg được sử dụng trong h() \*/

x = va\_arg ( ap, uint32\_t ); /\* Không hợp lệ \*/

/\* không xác định - kết thúc mà không sử dụng va\_end() \*/

}

void g ( void )

{

/\* không xác định - không khớp kiểu uint32\_t:double khi f sử dụng va\_arg() \*/

f ( 1, 2.0, 3.0 );

}

**Quy tắc 17.2**: Các hàm không được gọi chính mình, dù trực tiếp hay gián tiếp.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Không quyết định được, Hệ thống  
**Áp dụng cho**: C90, C99

**Lý do**

Đệ quy mang nguy cơ vượt quá không gian ngăn xếp khả dụng, có thể dẫn đến thất bại nghiêm trọng. Trừ khi đệ quy được kiểm soát chặt chẽ, không thể xác định trước khi thực thi việc sử dụng ngăn xếp trường hợp xấu nhất sẽ như thế nào.

**Quy tắc 17.3**: Hàm không được khai báo một cách ngầm định.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 6, 22, 23]  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90

**Lý do**

Nếu một lời gọi hàm được thực hiện trong khi có một prototype, thì một ràng buộc đảm bảo rằng số lượng đối số khớp với số lượng tham số và mỗi đối số có thể được gán cho tham số tương ứng.

Nếu một hàm được khai báo một cách ngầm định, trình biên dịch C90 sẽ giả định rằng hàm có kiểu trả về là int. Vì một khai báo hàm ngầm định không cung cấp prototype, trình biên dịch sẽ không có thông tin về số lượng tham số hàm và kiểu của chúng. Các chuyển đổi kiểu không phù hợp có thể dẫn đến lỗi khi truyền đối số và gán giá trị trả về, cũng như các hành vi không xác định khác.

**Ví dụ**

Nếu hàm power được khai báo như sau:

|  |
| --- |
|  |

extern double power ( double d, int n );

nhưng khai báo không hiển thị trong đoạn mã sau thì sẽ xảy ra hành vi không xác định.

|  |
| --- |
|  |

void func ( void )

{

/\* Không hợp lệ - kiểu trả về và cả hai kiểu đối số đều sai \*/

double sq1 = power ( 1, 2.0 );

}

**Xem them**

Quy tắc 8.2, Quy tắc 8.4

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng khi lập trình C theo chuẩn MISRA.

### Quy tắc 17.4: Tất cả các đường dẫn thoát khỏi một hàm có kiểu trả về khác void phải có một câu lệnh return rõ ràng với một biểu thức

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 43], **C99** [Không xác định 82]  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Lý do**

Biểu thức được cung cấp cho câu lệnh return xác định giá trị mà hàm trả về. Nếu một hàm không phải kiểu void không trả về giá trị nhưng hàm gọi lại sử dụng giá trị trả về, hành vi sẽ không xác định. Điều này có thể tránh được bằng cách đảm bảo rằng trong một hàm không phải kiểu void:

* Mọi câu lệnh return đều có một biểu thức, và
* Luồng điều khiển không thể đến cuối hàm mà không gặp câu lệnh return.

**Lưu ý**: C99 ràng buộc mọi câu lệnh return trong hàm không phải kiểu void phải trả về một giá trị.

**Ví dụ**

|  |
| --- |
|  |

int32\_t absolute ( int32\_t v )

{

if ( v < 0 )

{

return v;

}

/\*

\* Không hợp lệ - luồng điều khiển có thể đến điểm này mà không

\* trả về giá trị

\*/

}

uint16\_t lookup ( uint16\_t v )

{

if ( ( v < V\_MIN ) || ( v > V\_MAX ) )

{

/\* Không hợp lệ - không trả về giá trị. Ràng buộc trong C99 \*/

return;

}

return table[ v ];

}

**Xem thêm**  
Quy tắc 15.5

### Quy tắc 17.5: Đối số hàm tương ứng với tham số được khai báo có kiểu mảng phải có số phần tử phù hợp

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Khuyến nghị  
**Phân tích**: Không quyết định được, Hệ thống  
**Áp dụng cho**: C90, C99

**Mở rộng**

Nếu một tham số được khai báo là mảng với kích thước xác định, đối số tương ứng trong mỗi lời gọi hàm nên trỏ đến một đối tượng có ít nhất số phần tử bằng với mảng.

**Lý do**

Việc sử dụng khai báo mảng cho tham số hàm xác định giao diện hàm rõ ràng hơn so với việc sử dụng con trỏ. Số phần tử tối thiểu mà hàm mong đợi được chỉ định rõ ràng, trong khi điều này không thể thực hiện được với con trỏ.

Một khai báo mảng tham số hàm không chỉ định kích thước được hiểu là hàm có thể xử lý mảng với bất kỳ kích thước nào. Trong các trường hợp đó, kích thước mảng dự kiến sẽ được truyền thông qua các cách khác, ví dụ như được truyền dưới dạng một tham số khác hoặc bằng cách kết thúc mảng với một giá trị dấu hiệu.

Việc sử dụng ràng buộc kích thước mảng được khuyến nghị vì nó cho phép kiểm tra giới hạn ngoài phạm vi trong thân hàm và các kiểm tra thêm khi truyền tham số. Trong C, việc truyền một mảng có kích thước không đúng cho một tham số có kích thước được chỉ định là hợp lệ, nhưng có thể dẫn đến hành vi không mong muốn.

**Ví dụ**

|  |
| --- |
|  |

/\*

\* Ý định là hàm không truy cập ngoài phạm vi

\* array1[0] .. array1[3]

\*/

void fn1 ( int32\_t array1[4] );

/\* Ý định là hàm xử lý các mảng có kích thước bất kỳ \*/

void fn2 ( int32\_t array2[] );

void fn ( int32\_t \*ptr )

{

int32\_t arr3[3] = { 1, 2, 3 };

int32\_t arr4[4] = { 0, 1, 2, 3 };

/\* Hợp lệ - kích thước mảng khớp với prototype \*/

fn1 ( arr4 );

/\* Không hợp lệ - kích thước mảng không khớp với prototype \*/

fn1 ( arr3 );

/\* Hợp lệ chỉ khi ptr trỏ đến ít nhất 4 phần tử \*/

fn1 ( ptr );

/\* Hợp lệ \*/

fn2 ( arr4 );

/\* Hợp lệ \*/

fn2 ( ptr );

}

**Xem thêm**  
Quy tắc 17.6

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng khi lập trình C theo chuẩn MISRA.



Dưới đây là bản dịch tiếng Việt:

### Quy tắc 17.6: Khai báo một tham số mảng không được chứa từ khóa static trong dấu ngoặc vuông [ ]

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C99** [Không xác định 71]  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C99

**Lý do**

Tiêu chuẩn ngôn ngữ C99 cung cấp một cơ chế để lập trình viên thông báo cho trình biên dịch rằng một tham số mảng chứa một số phần tử tối thiểu được chỉ định. Một số trình biên dịch có thể tận dụng thông tin này để tạo mã hiệu quả hơn cho một số loại bộ xử lý.

Nếu đảm bảo do lập trình viên đưa ra không được tuân thủ và số phần tử ít hơn số tối thiểu được chỉ định, hành vi sẽ không xác định.

Các bộ xử lý được sử dụng trong các ứng dụng nhúng điển hình khó có khả năng cung cấp các tiện ích cần thiết để tận dụng thông tin bổ sung do lập trình viên cung cấp. Rủi ro khi chương trình không đáp ứng số phần tử tối thiểu được đảm bảo lớn hơn bất kỳ lợi ích tiềm năng nào về hiệu suất.

**Ví dụ**

Không có cách sử dụng nào của tính năng ngôn ngữ C99 này mà tuân thủ quy tắc. Các ví dụ dưới đây cho thấy một số hành vi không xác định có thể phát sinh từ việc sử dụng nó.

|  |
| --- |
|  |

/\* Không hợp lệ - sử dụng từ khóa static trong khai báo mảng \*/

uint16\_t total ( uint16\_t n, uint16\_t a[ static 20 ] )

{

uint16\_t i;

uint16\_t sum = 0U;

/\* Hành vi không xác định nếu a có ít hơn 20 phần tử \*/

for ( i = 0U; i < n; ++i )

{

sum = sum + a[ i ];

}

return sum;

}

extern uint16\_t v1[ 10 ];

extern uint16\_t v2[ 20 ];

void g ( void )

{

uint16\_t x;

x = total ( 10U, v1 ); /\* Không xác định - v1 có 10 phần tử nhưng cần

\* ít nhất 20 \*/

x = total ( 20U, v2 ); /\* Xác định nhưng không hợp lệ \*/

}

**Xem thêm**  
Quy tắc 17.5

### Quy tắc 17.7: Giá trị trả về bởi một hàm có kiểu trả về khác void phải được sử dụng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Lý do**

Có thể gọi một hàm mà không sử dụng giá trị trả về, điều này có thể là lỗi. Nếu giá trị trả về của một hàm không được sử dụng rõ ràng, nó nên được ép kiểu sang void. Điều này có tác dụng sử dụng giá trị mà không vi phạm Quy tắc 2.2.

**Ví dụ**

|  |
| --- |
|  |

uint16\_t func ( uint16\_t para1 )

{

return para1;

}

int16\_t x;

void discarded ( uint16\_t para2 )

{

func ( para2 ); /\* Không hợp lệ - giá trị bị bỏ qua \*/

( void ) func ( para2 ); /\* Hợp lệ \*/

x = func ( para2 ); /\* Hợp lệ \*/

}

**Xem them**

Dir 4.7, Quy tắc 2.2

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng khi lập trình C theo chuẩn MISRA.

### Quy tắc 17.8: Tham số hàm không nên bị sửa đổi

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Khuyến nghị  
**Phân tích**: Không quyết định được, Hệ thống  
**Áp dụng cho**: C90, C99

**Lý do**

Tham số hàm hoạt động giống như một đối tượng có thời gian lưu trữ tự động. Mặc dù ngôn ngữ C cho phép các tham số được sửa đổi, việc sử dụng như vậy có thể gây nhầm lẫn và mâu thuẫn với mong đợi của lập trình viên. Sẽ ít gây nhầm lẫn hơn nếu sao chép tham số vào một đối tượng tự động và sửa đổi bản sao đó. Với trình biên dịch hiện đại, điều này thường sẽ không gây ra bất kỳ sự phạt nào về lưu trữ hoặc thời gian thực thi.

Những lập trình viên không quen với ngôn ngữ C, nhưng quen thuộc với các ngôn ngữ khác, có thể sửa đổi một tham số với hy vọng rằng các tác động của việc sửa đổi sẽ được cảm nhận trong hàm gọi.

**Ví dụ**

|  |
| --- |
|  |

int16\_t glob = 0;

void proc ( int16\_t para )

{

para = glob; /\* Không hợp lệ \*/

}

void f ( char \*p, char \*q )

{

p = q; /\* Không hợp lệ \*/

\*p = \*q; /\* Hợp lệ \*/

}

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng khi lập trình C theo chuẩn MISRA.

### 8.18 Con trỏ và mảng

**Quy tắc 18.1**: Một con trỏ kết quả từ phép toán trên một toán hạng con trỏ phải trỏ đến một phần tử của cùng mảng với toán hạng con trỏ đó.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 30], **C99** [Không xác định 43, 44, 46, 59]  
**Loại**: Bắt buộc  
**Phân tích**: Không quyết định được, Hệ thống  
**Áp dụng cho**: C90, C99

**Mở rộng**

Việc tạo một con trỏ trỏ tới phần tử vượt ra ngoài cuối mảng được định nghĩa rõ ràng bởi Tiêu chuẩn và được phép bởi quy tắc này. Việc giải tham chiếu một con trỏ trỏ tới phần tử vượt ra ngoài cuối mảng dẫn đến hành vi không xác định và bị cấm bởi quy tắc này.

Quy tắc này áp dụng cho tất cả các dạng chỉ số mảng:

* integer\_expression + pointer\_expression
* pointer\_expression + integer\_expression
* pointer\_expression - integer\_expression
* pointer\_expression += integer\_expression
* pointer\_expression -= integer\_expression
* ++pointer\_expression
* pointer\_expression++
* --pointer\_expression
* pointer\_expression--
* pointer\_expression[integer\_expression]
* integer\_expression[pointer\_expression]

**Lưu ý**: Một mảng con cũng là một mảng.

**Lưu ý**: Đối với mục đích của phép toán trên con trỏ, Tiêu chuẩn coi một đối tượng không phải là phần tử của mảng như thể nó là một mảng có một phần tử (C90 Mục 6.3.6, C99 Mục 6.5.6).

**Lý do**

Mặc dù một số trình biên dịch có thể xác định tại thời gian biên dịch rằng một ranh giới mảng đã bị vượt qua, thường không có kiểm tra nào được thực hiện tại thời gian chạy đối với chỉ số mảng không hợp lệ. Việc sử dụng chỉ số mảng không hợp lệ có thể dẫn đến hành vi sai lệch của chương trình.

Các giá trị chỉ số mảng được xác định tại thời gian chạy là mối quan tâm lớn nhất vì chúng không dễ dàng được kiểm tra bằng phân tích tĩnh hoặc xem xét thủ công. Mã mang tính chất lập trình phòng ngừa nên được cung cấp, nếu có thể và hợp lý, để kiểm tra các giá trị chỉ số này với các giá trị hợp lệ và thực hiện hành động thích hợp nếu cần thiết.

**Ví dụ**

|  |
| --- |
|  |

int32\_t f1 ( int32\_t \* const a1, int32\_t a2[ 10 ] )

{

int32\_t \*p = &a1[ 3 ]; /\* Hợp lệ/Không hợp lệ tùy thuộc vào giá trị của a1 \*/

return \*( a2 + 9 ); /\* Hợp lệ \*/

}

void f2 ( void )

{

int32\_t data = 0;

int32\_t b = 0;

int32\_t c[ 10 ] = { 0 };

int32\_t d[ 5 ][ 2 ] = { 0 }; /\* Mảng 5 phần tử của mảng 2 phần tử kiểu int32\_t \*/

int32\_t \*p1 = &c[ 0 ]; /\* Hợp lệ \*/

int32\_t \*p2 = &c[ 10 ]; /\* Hợp lệ - trỏ tới phần tử vượt ra ngoài cuối \*/

int32\_t \*p3 = &c[ 11 ]; /\* Không hợp lệ - trỏ tới hai phần tử vượt ra ngoài cuối \*/

data = \*p2; /\* Không hợp lệ - giải tham chiếu phần tử vượt ra ngoài cuối \*/

data = f1 ( &b, c );

data = f1 ( c, c );

p1++; /\* Hợp lệ \*/

c[ -1 ] = 0; /\* Không hợp lệ - vượt giới hạn mảng \*/

data = c[ 10 ]; /\* Không hợp lệ - giải tham chiếu phần tử vượt ra ngoài cuối \*/

data = \*( &data + 0 ); /\* Hợp lệ - C coi data như một mảng kích thước 1 \*/

d[ 3 ][ 1 ] = 0; /\* Hợp lệ \*/

data = \*( \*( d + 3 ) + 1 ); /\* Hợp lệ \*/

data = d[ 2 ][ 3 ]; /\* Không hợp lệ - vượt giới hạn nội bộ \*/

p1 = d[ 1 ]; /\* Hợp lệ \*/

data = p1[ 1 ]; /\* Hợp lệ - p1 trỏ tới một mảng kích thước 2 \*/

}

Ví dụ sau minh họa phép toán trên con trỏ áp dụng cho các thành viên của một cấu trúc. Vì mỗi thành viên là một đối tượng độc lập, quy tắc này ngăn cản việc sử dụng phép toán trên con trỏ để di chuyển từ thành viên này sang thành viên khác. Tuy nhiên, nó không ngăn cản phép toán trên con trỏ tới một thành viên với điều kiện con trỏ kết quả nằm trong phạm vi của đối tượng thành viên.

|  |
| --- |
|  |

struct

{

uint16\_t x;

uint16\_t y;

uint16\_t z;

uint16\_t a[ 10 ];

} s;

uint16\_t \*p;

void f3 ( void )

{

p = &s.x;

++p; /\* Hợp lệ - p trỏ tới một phần tử vượt ra ngoài s.x \*/

p[ 0 ] = 1; /\* Không hợp lệ - giải tham chiếu phần tử vượt ra ngoài s.x \*/

p[ 1 ] = 2; /\* Không hợp lệ \*/

p = &s.a[ 0 ]; /\* Hợp lệ \*/

p = p + 8; /\* Hợp lệ - p vẫn trỏ vào s.a \*/

p = p + 3; /\* Không hợp lệ - p trỏ nhiều hơn một phần tử vượt ra ngoài s.a \*/

}

**Xem thêm**  
Dir 4.1, Quy tắc 18.4

**Quy tắc 18.2**: Phép trừ giữa các con trỏ chỉ được áp dụng cho các con trỏ trỏ tới các phần tử của cùng mảng **C90** [Không xác định 31], **C99** [Không xác định 45].

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Không quyết định được, Hệ thống  
**Áp dụng cho**: C90, C99

**Lý do**

Quy tắc này áp dụng cho các biểu thức có dạng:

* pointer\_expression\_1 - pointer\_expression\_2

Hành vi không xác định sẽ xảy ra nếu pointer\_expression\_1 và pointer\_expression\_2 không trỏ tới các phần tử của cùng mảng hoặc phần tử một ngoài cuối mảng đó.

**Ví dụ**

|  |
| --- |
|  |

#include <stddef.h>

void f1 ( int32\_t \*ptr )

{

int32\_t a1[ 10 ];

int32\_t a2[ 10 ];

int32\_t \*p1 = &a1[ 1 ];

int32\_t \*p2 = &a2[ 10 ];

ptrdiff\_t diff;

diff = p1 - a1; /\* Hợp lệ \*/

diff = p2 - a2; /\* Hợp lệ \*/

diff = p1 - p2; /\* Không hợp lệ \*/

diff = ptr - p1; /\* Không hợp lệ \*/

}

**Xem them**

Dir 4.1, Quy tắc 18.4

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng khi lập trình C theo chuẩn MISRA.

### Quy tắc 18.3: Các toán tử quan hệ >, >=, < và <= không được áp dụng cho các đối tượng kiểu con trỏ trừ khi chúng trỏ vào cùng một đối tượng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 33], **C99** [Không xác định 50]  
**Loại**: Bắt buộc  
**Phân tích**: Không quyết định được, Hệ thống  
**Áp dụng cho**: C90, C99

**Lý do**

Việc cố gắng so sánh giữa các con trỏ sẽ dẫn đến hành vi không xác định nếu hai con trỏ không trỏ tới cùng một đối tượng.

**Lưu ý**: Việc trỏ tới phần tử tiếp theo sau cuối mảng là được phép, nhưng không được truy cập phần tử này.

**Ví dụ**

|  |
| --- |
|  |

void f1 ( void )

{

int32\_t a1[ 10 ];

int32\_t a2[ 10 ];

int32\_t \*p1 = a1;

if ( p1 < a1 ) /\* Hợp lệ \*/

{

}

if ( p1 < a2 ) /\* Không hợp lệ \*/

{

}

}

struct limits

{

int32\_t lwb;

int32\_t upb;

};

void f2 ( void )

{

struct limits limits\_1 = { 2, 5 };

struct limits limits\_2 = { 10, 5 };

if ( &limits\_1.lwb <= &limits\_1.upb ) /\* Hợp lệ \*/

{

}

if ( &limits\_1.lwb > &limits\_2.upb ) /\* Không hợp lệ \*/

{

}

}

**Xem them**

Dir 4.1

### Quy tắc 18.4: Các toán tử +, -, += và -= không nên được áp dụng cho biểu thức kiểu con trỏ

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Khuyến nghị  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Lý do**

Cú pháp chỉ số mảng, ptr[expr], là dạng ưu tiên hơn của phép toán trên con trỏ vì nó thường rõ ràng hơn và ít có khả năng gây lỗi hơn so với thao tác trên con trỏ. Mọi giá trị con trỏ được tính toán một cách tường minh đều có khả năng truy cập vào địa chỉ bộ nhớ không mong muốn hoặc không hợp lệ. Mặc dù hành vi này cũng có thể xảy ra với cú pháp chỉ số mảng, cú pháp chỉ số có thể giúp việc xem xét thủ công trở nên dễ dàng hơn.

Phép toán trên con trỏ trong C có thể gây nhầm lẫn cho người mới. Biểu thức ptr+1 có thể bị hiểu sai là phép cộng 1 vào địa chỉ trong ptr. Trên thực tế, địa chỉ mới phụ thuộc vào kích thước tính bằng byte của mục tiêu con trỏ. Sự hiểu sai này có thể dẫn đến hành vi không mong muốn nếu sizeof được áp dụng không đúng.

Tuy nhiên, khi sử dụng một cách thận trọng, việc thao tác trên con trỏ sử dụng ++ có thể được coi là tự nhiên hơn trong một số trường hợp, chẳng hạn như truy cập tuần tự các vị trí trong một kiểm tra bộ nhớ, nơi tiện lợi hơn khi coi không gian bộ nhớ như một tập hợp liên tiếp các vị trí và giới hạn địa chỉ có thể được xác định tại thời gian biên dịch.

**Ngoại lệ**  
Theo quy tắc 18.2, phép trừ giữa hai con trỏ được phép.

**Ví dụ**

|  |
| --- |
|  |

void fn1 ( void )

{

uint8\_t a[ 10 ];

uint8\_t \*ptr;

uint8\_t index = 0U;

index = index + 1U; /\* Hợp lệ - quy tắc chỉ áp dụng cho con trỏ \*/

a[ index ] = 0U; /\* Hợp lệ \*/

ptr = &a[ 5 ]; /\* Hợp lệ \*/

ptr = a;

ptr++; /\* Hợp lệ - toán tử tăng không phải + \*/

\*( ptr + 5 ) = 0U; /\* Không hợp lệ \*/

ptr[ 5 ] = 0U; /\* Hợp lệ \*/

}

void fn2 ( void )

{

uint8\_t array\_2\_2[ 2 ][ 2 ] = { { 1U, 2U }, { 4U, 5U } };

uint8\_t i = 0U;

uint8\_t j = 0U;

uint8\_t sum = 0U;

for ( i = 0U; i < 2U; i++ )

{

uint8\_t \*row = array\_2\_2[ i ];

for ( j = 0U; j < 2U; j++ )

{

sum += row[ j ]; /\* Hợp lệ \*/

}

}

}

void fn3 ( uint8\_t \*p1, uint8\_t p2[] )

{

p1++; /\* Hợp lệ \*/

p1 = p1 + 5; /\* Không hợp lệ \*/

p1[ 5 ] = 0U; /\* Hợp lệ \*/

p2++; /\* Hợp lệ \*/

p2 = p2 + 3; /\* Không hợp lệ \*/

p2[ 3 ] = 0U; /\* Hợp lệ \*/

}

uint8\_t a1[ 16 ];

uint8\_t a2[ 16 ];

uint8\_t data = 0U;

void fn4 ( void )

{

fn3 ( a1, a2 );

fn3 ( &data, &a2[ 4 ] );

}

**Xem thêm**

Quy tắc 18.1, Quy tắc 18.2

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng khi lập trình C theo chuẩn MISRA.

### Quy tắc 18.5: Khai báo không nên chứa nhiều hơn hai mức độ lồng nhau của con trỏ

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Khuyến nghị  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Không nên áp dụng nhiều hơn hai dấu khai báo con trỏ liên tiếp vào một kiểu. Bất kỳ tên typedef nào xuất hiện trong khai báo được coi như thể nó được thay thế bằng kiểu mà nó biểu thị.

**Lý do**  
Việc sử dụng hơn hai mức độ lồng nhau của con trỏ có thể làm giảm nghiêm trọng khả năng hiểu được hành vi của mã, do đó nên tránh.

**Ví dụ**

|  |
| --- |
|  |

typedef int8\_t \* INTPTR;

void function ( int8\_t \*\* arrPar[ ] ) /\* Không hợp lệ \*/

{

int8\_t \*\* obj2; /\* Hợp lệ \*/

int8\_t \*\*\* obj3; /\* Không hợp lệ \*/

INTPTR \* obj4; /\* Hợp lệ \*/

INTPTR \* const \* const obj5; /\* Không hợp lệ \*/

int8\_t \*\* arr[ 10 ]; /\* Hợp lệ \*/

int8\_t \*\* ( \*parr )[ 10 ]; /\* Hợp lệ \*/

int8\_t \* ( \*\*pparr )[ 10 ]; /\* Hợp lệ \*/

}

struct s

{

int8\_t \* s1; /\* Hợp lệ \*/

int8\_t \*\* s2; /\* Hợp lệ \*/

int8\_t \*\*\* s3; /\* Không hợp lệ \*/

};

struct s \* ps1; /\* Hợp lệ \*/

struct s \*\* ps2; /\* Hợp lệ \*/

struct s \*\*\* ps3; /\* Không hợp lệ \*/

int8\_t \*\* ( \*pfunc1 )( void ); /\* Hợp lệ \*/

int8\_t \*\* ( \*\*pfunc2 )( void ); /\* Hợp lệ \*/

int8\_t \*\* ( \*\*\*pfunc3 )( void ); /\* Không hợp lệ \*/

int8\_t \*\*\* ( \*\*pfunc4 )( void ); /\* Không hợp lệ \*/

**Lưu ý**:

* arrPar có kiểu là con trỏ tới con trỏ tới con trỏ tới int8\_t, vì tham số được khai báo với kiểu mảng sẽ được chuyển thành con trỏ tới phần tử đầu tiên của mảng — đây là ba mức độ và không hợp lệ.
* arr có kiểu là mảng của con trỏ tới con trỏ tới int8\_t — điều này là hợp lệ.
* parr có kiểu là con trỏ tới mảng của con trỏ tới con trỏ tới int8\_t — điều này là hợp lệ.
* pparr có kiểu là con trỏ tới con trỏ tới mảng của con trỏ tới int8\_t — điều này là hợp lệ.

### Quy tắc 18.6: Địa chỉ của một đối tượng có thời gian lưu trữ tự động không được sao chép sang một đối tượng tồn tại lâu hơn sau khi đối tượng đầu tiên không còn tồn tại

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 9, 26], **C99** [Không xác định 8, 9, 40]  
**Loại**: Bắt buộc  
**Phân tích**: Không quyết định được, Hệ thống  
**Áp dụng cho**: C90, C99

**Mở rộng**

Địa chỉ của một đối tượng có thể được sao chép bằng các phương thức:

* Gán;
* Các hàm di chuyển hoặc sao chép bộ nhớ.

**Lý do**

Địa chỉ của một đối tượng trở nên không xác định khi vòng đời của đối tượng đó kết thúc. Việc sử dụng một địa chỉ không xác định dẫn đến hành vi không xác định.

**Ví dụ**

|  |
| --- |
|  |

int8\_t \*func ( void )

{

int8\_t local\_auto;

return &local\_auto; /\* Không hợp lệ - &local\_auto trở nên không xác định

khi func trả về \*/

}

uint16\_t \*sp;

void g ( uint16\_t \*p )

{

sp = p; /\* Không hợp lệ - địa chỉ của tham số u của f

được sao chép vào sp tĩnh \*/

}

void f ( uint16\_t u )

{

g ( &u );

}

void h ( void )

{

static uint16\_t \*q;

uint16\_t x = 0u;

q = &x; /\* Không hợp lệ - &x được lưu trữ trong một đối tượng có

vòng đời dài hơn \*/

}

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng khi lập trình C theo chuẩn MISRA.

Quy tắc 18.7: Không được khai báo thành viên mảng linh hoạt

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C99** [Không xác định 59]  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Lý do**  
Thành viên mảng linh hoạt thường được sử dụng cùng với việc cấp phát bộ nhớ động, điều này bị cấm bởi Dir 4.12 và Quy tắc 21.3.

Sự xuất hiện của thành viên mảng linh hoạt làm thay đổi hành vi của toán tử sizeof theo cách mà lập trình viên có thể không mong đợi. Việc gán một cấu trúc có chứa thành viên mảng linh hoạt cho một cấu trúc khác cùng loại có thể không hoạt động như mong đợi vì nó chỉ sao chép các phần tử đến nhưng không bao gồm điểm bắt đầu của thành viên mảng linh hoạt.

**Ví dụ**

|  |
| --- |
|  |

#include <stdlib.h>

struct s

{

uint16\_t len;

uint32\_t data[]; /\* Không hợp lệ - thành viên mảng linh hoạt \*/

} str;

struct s \*copy ( struct s \*s1 )

{

struct s \*s2;

/\* Lược bỏ kiểm tra kết quả của malloc để ngắn gọn \*/

s2 = malloc ( sizeof ( struct s ) + ( s1->len \* sizeof ( uint32\_t ) ) );

\*s2 = \*s1; /\* Chỉ sao chép s1->len \*/

return s2;

}

**Xem thêm**  
Dir 4.12, Quy tắc 21.3

### Quy tắc 18.8: Không được sử dụng kiểu mảng có độ dài thay đổi

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C99** [Không xác định 21; Không xác định 69, 70]  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C99

**Lý do**  
Kiểu mảng có độ dài thay đổi được chỉ định khi kích thước của mảng được khai báo trong một khối hoặc nguyên mẫu hàm không phải là một biểu thức hằng số nguyên. Chúng thường được triển khai như các đối tượng có kích thước biến lưu trữ trên ngăn xếp. Do đó, việc sử dụng chúng có thể làm cho việc xác định trước lượng bộ nhớ cần thiết để dự trữ trên ngăn xếp trở nên không thể.

Nếu kích thước của mảng có độ dài thay đổi là âm hoặc bằng không, hành vi sẽ không xác định. Nếu một mảng có độ dài thay đổi được sử dụng trong ngữ cảnh cần tương thích với một kiểu mảng khác, có thể chính nó cũng là mảng có độ dài thay đổi, thì kích thước của các kiểu mảng phải giống nhau. Hơn nữa, tất cả kích thước phải được đánh giá thành các số nguyên dương. Nếu các yêu cầu này không được đáp ứng, hành vi sẽ không xác định.

Nếu một kiểu mảng có độ dài thay đổi được sử dụng làm toán hạng của toán tử sizeof, trong một số trường hợp không xác định liệu biểu thức kích thước mảng có được đánh giá hay không.

Mỗi trường hợp của kiểu mảng có độ dài thay đổi có kích thước cố định khi bắt đầu vòng đời của nó. Điều này dẫn đến hành vi có thể gây nhầm lẫn, ví dụ:

|  |
| --- |
|  |

void f ( void )

{

uint16\_t n = 5;

typedef uint16\_t Vector[n]; /\* Một kiểu mảng với 5 phần tử \*/

n = 7;

Vector a1; /\* Một kiểu mảng với 5 phần tử \*/

uint16\_t a2[n]; /\* Một kiểu mảng với 7 phần tử \*/

}

**Ví dụ**

Không có cách sử dụng nào của mảng có độ dài thay đổi tuân thủ quy tắc này. Các ví dụ cho thấy một số hành vi không xác định có thể phát sinh từ việc sử dụng chúng.

c

Sao chép mã

void f ( int16\_t n )

{

uint16\_t vla[n]; /\* Không hợp lệ - Không xác định nếu n <= 0 \*/

}

void g ( void )

{

f(0); /\* Không xác định \*/

f(-1); /\* Không xác định \*/

f(10); /\* Hợp lệ \*/

}

void h ( uint16\_t n, uint16\_t a[10][n] ) /\* Không hợp lệ \*/

{

uint16\_t (\*p)[20];

/\* Không xác định trừ khi n == 20: kiểu không tương thích nếu không \*/

p = a;

}

**Xem thêm**  
Quy tắc 13.6

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng khi lập trình C theo chuẩn MISRA.

### Bộ nhớ chồng lấp

**Quy tắc 19.1**: Một đối tượng không được gán hoặc sao chép vào một đối tượng khác có vùng nhớ chồng lấp.

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 34, 55], **C99** [Không xác định 51, 94]  
**Loại**: Bắt buộc  
**Phân tích**: Không quyết định được, Hệ thống  
**Áp dụng cho**: C90, C99

**Lý do**  
Hành vi không xác định xảy ra khi hai đối tượng được tạo ra có một phần vùng nhớ chồng lấp và một trong số đó được gán hoặc sao chép vào đối tượng kia.

**Ngoại lệ**  
Các trường hợp sau được phép vì hành vi được định nghĩa rõ ràng:

1. Việc gán giữa hai đối tượng có vùng nhớ chồng lấp hoàn toàn và có kiểu tương thích (không tính đến các bộ định tính kiểu).
2. Sao chép giữa các đối tượng có vùng nhớ chồng lấp một phần hoặc hoàn toàn bằng cách sử dụng hàm memmove của Thư viện Chuẩn.

**Ví dụ**

Ví dụ này cũng vi phạm Quy tắc 19.2 vì sử dụng union.

|  |
| --- |
|  |

void fn ( void )

{

union

{

int16\_t i;

int32\_t j;

} a = { 0 }, b = { 1 };

a.j = a.i; /\* Không hợp lệ \*/

a = b; /\* Hợp lệ - ngoại lệ 1 \*/

}

|  |
| --- |
|  |

#include <string.h>

int16\_t a[ 20 ];

void f ( void )

{

memcpy( &a[ 5 ], &a[ 4 ], 2u \* sizeof( a[ 0 ] ) ); /\* Không hợp lệ \*/

}

void g ( void )

{

int16\_t \*p = &a[ 0 ];

int16\_t \*q = &a[ 0 ];

\*p = \*q; /\* Hợp lệ - ngoại lệ 1 \*/

}

**Xem thêm**  
Quy tắc 19.2

**Quy tắc 19.2**: Từ khóa union không nên được sử dụng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 39, 40; Thực thi 27], **C99** [Không xác định 10; Không xác định 61, 62]  
**Loại**: Khuyến nghị  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Lý do**  
Một thành viên union có thể được ghi và cùng thành viên đó có thể được đọc lại theo cách định nghĩa rõ ràng.

Tuy nhiên, nếu một thành viên union được ghi và sau đó một thành viên khác được đọc lại, hành vi phụ thuộc vào kích thước tương đối của các thành viên:

* Nếu thành viên được đọc có kích thước lớn hơn thành viên được ghi thì giá trị là không xác định.
* Nếu không, giá trị là do triển khai xác định.

Tiêu chuẩn cho phép truy cập các byte của một thành viên union thông qua một thành viên khác có kiểu là mảng unsigned char. Tuy nhiên, vì có thể truy cập các byte với giá trị không xác định, union không nên được sử dụng.

Nếu quy tắc này không được tuân thủ, các loại hành vi cần được xác định là:

* **Padding**: bao nhiêu phần đệm được chèn vào cuối union.
* **Căn chỉnh**: các thành viên của bất kỳ cấu trúc nào trong union được căn chỉnh như thế nào.
* **Endianness**: byte có ý nghĩa cao nhất của một từ được lưu trữ tại địa chỉ bộ nhớ thấp nhất hay cao nhất.
* **Bit-order**: các bit được đánh số như thế nào trong các byte và các bit được gán cho các trường bit như thế nào.

**Ví dụ**

Trong ví dụ không hợp lệ này, một giá trị 16-bit được lưu vào union nhưng một giá trị 32-bit được đọc lại, dẫn đến một giá trị không xác định được trả về.

|  |
| --- |
|  |

uint32\_t zext ( uint16\_t s )

{

union

{

uint32\_t ul;

uint16\_t us;

} tmp;

tmp.us = s;

return tmp.ul; /\* Giá trị không xác định \*/

}

**Xem thêm**  
Quy tắc 19.1

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng khi lập trình C theo chuẩn MISRA.

### 8.20 Chỉ thị tiền xử lý

#### **Quy tắc 20.1**:

Các chỉ thị #include chỉ nên được đặt trước bởi các chỉ thị tiền xử lý hoặc nhận xét

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 56], **C99** [Không xác định 96, 97]  
**Loại**: Khuyến nghị  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Quy tắc này sẽ được áp dụng cho nội dung của tệp trước khi thực hiện tiền xử lý.

**Lý do**  
Để hỗ trợ khả năng đọc mã, tất cả các chỉ thị #include trong một tệp mã cụ thể nên được nhóm lại với nhau gần đầu tệp.

Ngoài ra, việc sử dụng #include để bao gồm một tệp tiêu đề chuẩn trong một khai báo hoặc định nghĩa, hoặc việc sử dụng một phần của Thư viện Chuẩn trước khi bao gồm tệp tiêu đề chuẩn liên quan dẫn đến hành vi không xác định.

**Ví dụ**

|  |
| --- |
|  |

/\* f.h \*/

xyz = 0;

/\* f.c \*/

int16\_t

#include "f.h" /\* Không hợp lệ \*/

/\* f1.c \*/

#define F1\_MACRO

#include "f1.h" /\* Hợp lệ \*/

#include "f2.h" /\* Hợp lệ \*/

int32\_t i = 0;

#include "f3.h" /\* Không hợp lệ \*/

#### **Quy tắc 20.2**:

Các ký tự ', ", \ và các chuỗi ký tự /\* hoặc // không được xuất hiện trong tên tệp tiêu đề

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 14], **C99** [Không xác định 31]  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Lý do**  
Hành vi là không xác định nếu:

* Các ký tự ', ", \ hoặc các chuỗi ký tự /\*, // được sử dụng giữa các dấu phân cách < và > trong một token tên tệp tiêu đề.
* Các ký tự ', \ hoặc các chuỗi ký tự /\*, // được sử dụng giữa các dấu phân cách " trong một token tên tệp tiêu đề.

**Lưu ý**: Mặc dù việc sử dụng ký tự \ dẫn đến hành vi không xác định, nhiều triển khai sẽ chấp nhận ký tự / thay thế.

**Ví dụ**

|  |
| --- |
|  |

#include "fi'le.h" /\* Không hợp lệ \*/

#### **Quy tắc 20.3**:

Chỉ thị #include phải được theo sau bởi một chuỗi <tên\_tệp> hoặc "tên\_tệp"

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 48], **C99** [Không xác định 85]  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Quy tắc này áp dụng sau khi đã thực hiện thay thế macro.

**Lý do**  
Hành vi là không xác định nếu một chỉ thị #include không sử dụng một trong các dạng sau:

* #include <tên\_tệp>
* #include "tên\_tệp"

**Ví dụ**

|  |
| --- |
|  |

#include "filename.h" /\* Hợp lệ \*/

#include <filename.h> /\* Hợp lệ \*/

#include another.h /\* Không hợp lệ \*/

#define HEADER "filename.h"

#include HEADER /\* Hợp lệ \*/

#define FILENAME file2.h

#include FILENAME /\* Không hợp lệ \*/

#define BASE "base"

#define EXT ".ext"

#include BASE EXT /\* Không hợp lệ - các chuỗi được nối sau khi tiền xử lý \*/

#include "./include/cpu.h" /\* Hợp lệ - tên tệp có thể bao gồm đường dẫn \*/

#### Quy tắc 20.4:

Một macro không được định nghĩa với tên giống từ khóa

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 56], **C99** [Không xác định 98]  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Quy tắc này áp dụng cho tất cả các từ khóa, bao gồm cả những từ khóa mở rộng ngôn ngữ.

**Lý do**  
Việc sử dụng macro để thay đổi ý nghĩa của từ khóa có thể gây nhầm lẫn. Hành vi sẽ không xác định nếu một tệp tiêu đề chuẩn được bao gồm trong khi một macro được định nghĩa với tên giống từ khóa.

**Ví dụ**

Ví dụ sau không hợp lệ vì nó thay đổi hành vi của từ khóa int. Việc bao gồm một tệp tiêu đề chuẩn trong sự hiện diện của macro này sẽ dẫn đến hành vi không xác định.

|  |
| --- |
|  |

#define int some\_other\_type

#include <stdlib.h>

Ví dụ sau cho thấy việc định nghĩa lại từ khóa while là không hợp lệ, nhưng việc định nghĩa một macro mở rộng ra các câu lệnh là hợp lệ.

|  |
| --- |
|  |

#define while( E ) for ( ; ( E ) ; ) /\* Không hợp lệ - định nghĩa lại while \*/

#define unless( E ) if ( ! ( E ) ) /\* Hợp lệ \*/

#define seq( S1, S2 ) do { \

S1; S2; } while ( false ) /\* Hợp lệ \*/

#define compound( S ) { S; } /\* Hợp lệ \*/

Ví dụ sau hợp lệ trong C90 nhưng không trong C99 vì inline không phải là từ khóa trong C90.

|  |
| --- |
|  |

/\* Bỏ inline nếu biên dịch cho C90 \*/

#define inline

**Xem thêm**  
Quy tắc 21.1

#### Quy tắc 20.5:

Không nên sử dụng #undef

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Khuyến nghị  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Lý do**  
Việc sử dụng #undef có thể làm mờ mịt việc xác định macro nào tồn tại tại một điểm cụ thể trong một đơn vị dịch chuyển.

**Ví dụ**

|  |
| --- |
|  |

#define QUALIFIER volatile

#undef QUALIFIER /\* Không hợp lệ \*/

void f ( QUALIFIER int32\_t p )

{

while ( p != 0 )

{

; /\* Chờ... \*/

}

}

### Quy tắc 20.6: Các token giống như một chỉ thị tiền xử lý không được xuất hiện trong một đối số macro

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 50], **C99** [Không xác định 87]  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Lý do**  
Một đối số chứa các chuỗi token mà nếu không sẽ hoạt động như các chỉ thị tiền xử lý dẫn đến hành vi không xác định.

**Ví dụ**

|  |
| --- |
|  |

#define M( A ) printf ( #A )

#include <stdio.h>

void main ( void )

{

M (

#ifdef SW /\* Không hợp lệ \*/

"Message 1"

#else /\* Không hợp lệ \*/

"Message 2"

#endif /\* Không hợp lệ \*/

);

}

Kết quả của đoạn mã trên có thể là:

arduino

Sao chép mã

#ifdef SW "Message 1" #else "Message 2" #endif

hoặc

arduino

Sao chép mã

"Message 2"

hoặc có thể dẫn đến hành vi khác.

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng khi lập trình C theo chuẩn MISRA.

### Quy tắc 20.7: Các biểu thức từ việc mở rộng tham số macro phải được đặt trong dấu ngoặc đơn

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**[Koenig 78–81]**  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Nếu việc mở rộng bất kỳ tham số macro nào tạo ra một token hoặc chuỗi token tạo thành một biểu thức, thì trong macro mở rộng đầy đủ, biểu thức đó phải:

* Là một biểu thức có dấu ngoặc đơn, hoặc
* Được đặt trong dấu ngoặc đơn.

**Lưu ý**: Quy tắc này không nhất thiết yêu cầu tất cả tham số macro phải có dấu ngoặc đơn; việc đặt dấu ngoặc đơn trong các đối số macro là chấp nhận được.

**Lý do**  
Nếu không sử dụng dấu ngoặc đơn, độ ưu tiên của toán tử có thể không cho ra kết quả mong muốn khi xảy ra thay thế macro. Nếu một tham số macro không được sử dụng như một biểu thức thì không cần phải có dấu ngoặc đơn vì không có toán tử nào liên quan.

**Ví dụ**

Trong ví dụ không hợp lệ sau:

|  |
| --- |
|  |

#define M1( x, y ) ( x \* y )

r = M1 ( 1 + 2, 3 + 4 );

Macro mở rộng thành:

c

Sao chép mã

r = ( 1 + 2 \* 3 + 4 );

Các biểu thức 1 + 2 và 3 + 4 được mở rộng từ tham số x và y tương ứng, nhưng không được đặt trong dấu ngoặc đơn. Giá trị của biểu thức kết quả là 11, trong khi kết quả mong đợi có thể là 21.

Mã có thể được viết lại để tuân thủ bằng cách đặt dấu ngoặc đơn cho các đối số macro hoặc viết phiên bản khác của macro chèn dấu ngoặc trong quá trình mở rộng, ví dụ:

|  |
| --- |
|  |

r = M1 ( ( 1 + 2 ), ( 3 + 4 ) ); /\* Hợp lệ \*/

#define M2( x, y ) ( ( x ) \* ( y ) )

r = M2 ( 1 + 2, 3 + 4 ); /\* Hợp lệ \*/

Ví dụ sau là hợp lệ vì việc mở rộng đầu tiên của x là toán hạng của toán tử ##, không tạo thành biểu thức. Việc mở rộng lần hai của x là một biểu thức được đặt trong dấu ngoặc đơn như yêu cầu.

|  |
| --- |
|  |

#define M3( x ) a ## x = ( x )

int16\_t M3 ( 0 );

Ví dụ sau hợp lệ vì việc mở rộng tham số M như một tên thành viên không tạo thành biểu thức. Việc mở rộng tham số S tạo thành một biểu thức có kiểu cấu trúc hoặc liên minh, cần có dấu ngoặc đơn.

|  |
| --- |
|  |

#define GET\_MEMBER( S, M ) ( S ).M

v = GET\_MEMBER ( s1, minval );

Ví dụ hợp lệ sau cho thấy rằng không phải lúc nào cũng cần thiết phải đặt dấu ngoặc đơn cho mỗi trường hợp của tham số, mặc dù đây thường là cách dễ nhất để tuân thủ quy tắc này.

|  |
| --- |
|  |

#define F( X ) G( X )

#define G( Y ) ( ( Y ) + 1 )

int16\_t x = F ( 2 );

Macro mở rộng đầy đủ là ( ( 2 ) + 1 ). Theo dõi lại qua quá trình mở rộng macro, giá trị 2 bắt nguồn từ việc mở rộng tham số Y trong macro G và từ tham số X trong macro F. Vì 2 được đặt trong dấu ngoặc đơn trong macro mở rộng đầy đủ, mã này là hợp lệ.

**Xem thêm**  
Dir 4.9

### Quy tắc 20.8: Biểu thức điều khiển của chỉ thị tiền xử lý #if hoặc #elif phải đánh giá ra 0 hoặc 1

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Quy tắc này không áp dụng cho các biểu thức điều khiển trong chỉ thị tiền xử lý không được đánh giá. Các biểu thức điều khiển không được đánh giá nếu chúng nằm trong mã bị loại trừ và không ảnh hưởng đến việc mã có bị loại trừ hay không.

**Lý do**  
Kiểu mạnh yêu cầu biểu thức điều khiển của chỉ thị tiền xử lý điều kiện phải có giá trị Boolean.

**Ví dụ**

|  |
| --- |
|  |

#define FALSE 0

#define TRUE 1

#if FALSE /\* Hợp lệ \*/

#endif

#if 10 /\* Không hợp lệ \*/

#endif

#if ! defined ( X ) /\* Hợp lệ \*/

#endif

#if A > B /\* Hợp lệ với điều kiện A và B là số \*/

#endif

**Xem thêm**  
Quy tắc 14.4

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng khi lập trình C theo chuẩn MISRA.

### Quy tắc 20.9: Tất cả các định danh được sử dụng trong biểu thức điều khiển của chỉ thị tiền xử lý #if hoặc #elif phải được #define trước khi đánh giá

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Ngoài việc sử dụng chỉ thị tiền xử lý #define, các định danh có thể được #define theo các cách khác, tùy thuộc vào định nghĩa của trình biên dịch. Ví dụ, một số trình biên dịch hỗ trợ:

* Sử dụng tùy chọn dòng lệnh của trình biên dịch, chẳng hạn như -D để cho phép định danh được định nghĩa trước khi dịch;
* Sử dụng các biến môi trường để đạt được hiệu quả tương tự;
* Các định danh được định nghĩa trước do trình biên dịch cung cấp.

**Lý do**  
Nếu cố gắng sử dụng một định danh macro trong chỉ thị tiền xử lý mà định danh đó chưa được định nghĩa, trình tiền xử lý sẽ giả định rằng nó có giá trị bằng không. Điều này có thể không phù hợp với mong đợi của lập trình viên.

**Ví dụ**

Các ví dụ sau giả định rằng macro M chưa được định nghĩa.

|  |
| --- |
|  |

#if M == 0 /\* Không hợp lệ \*/

/\* 'M' có mở rộng thành 0 hay chưa được định nghĩa? \*/

#endif

#if defined ( M ) /\* Hợp lệ - M không được đánh giá \*/

#if M == 0 /\* Hợp lệ - M đã biết là được định nghĩa \*/

/\* 'M' phải mở rộng thành 0. \*/

#endif

#endif

/\* Hợp lệ - B chỉ được đánh giá trong ( B == 0 ) nếu nó được định nghĩa \*/

#if defined ( B ) && ( B == 0 )

#endif

### Quy tắc 20.10: Các toán tử tiền xử lý # và ## không nên được sử dụng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 12; Không xác định 51, 52], **C99** [Không xác định 25; Không xác định 3, 88, 89]  
**Loại**: Khuyến nghị  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Lý do**  
Thứ tự đánh giá liên quan đến nhiều toán tử #, nhiều toán tử ## hoặc sự kết hợp của # và ## là không xác định. Trong một số trường hợp, không thể dự đoán được kết quả của việc mở rộng macro.

Việc sử dụng toán tử ## có thể dẫn đến mã khó hiểu.

**Lưu ý**: Quy tắc 1.3 bao gồm hành vi không xác định phát sinh nếu:

* Kết quả của toán tử # không phải là một chuỗi hợp lệ;
* Kết quả của toán tử ## không phải là một token tiền xử lý hợp lệ.

**Xem thêm**  
Quy tắc 20.11

### Quy tắc 20.11: Tham số macro ngay sau toán tử # không được đi ngay sau bởi toán tử ##

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 12], **C99** [Không xác định 25]  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Lý do**  
Thứ tự đánh giá liên quan đến nhiều toán tử #, nhiều toán tử ## hoặc sự kết hợp của # và ## là không xác định. Việc sử dụng # và ## không được khuyến khích theo Quy tắc 20.10. Đặc biệt, kết quả của toán tử # là một chuỗi và rất ít khả năng việc nối chuỗi này với bất kỳ token tiền xử lý nào khác sẽ tạo thành một token hợp lệ.

**Ví dụ**

|  |
| --- |
|  |

#define A( x ) #x /\* Hợp lệ \*/

#define B( x, y ) x ## y /\* Hợp lệ \*/

#define C( x, y ) #x ## y /\* Không hợp lệ \*/

**Xem thêm**  
Quy tắc 20.10

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng khi lập trình C theo chuẩn MISRA.

### Quy tắc 20.12: Một tham số macro được sử dụng như một toán hạng cho các toán tử # hoặc ##, và chính nó cũng được thay thế macro, chỉ nên được sử dụng như một toán hạng cho các toán tử này

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Lý do**  
Tham số macro được sử dụng như một toán hạng của toán tử # hoặc ## không được mở rộng trước khi được sử dụng. Tham số giống nhau xuất hiện ở nơi khác trong văn bản thay thế sẽ được mở rộng. Nếu tham số macro cũng được thay thế macro, việc sử dụng nó trong các ngữ cảnh hỗn hợp trong quá trình thay thế macro có thể không đáp ứng mong đợi của lập trình viên.

**Ví dụ**

Trong ví dụ không hợp lệ sau, tham số macro x được thay thế bằng AA, tiếp tục được thay thế macro khi không được sử dụng như toán hạng của ##.

|  |
| --- |
|  |

#define AA 0xffff

#define BB( x ) ( x ) + wow ## x /\* Không hợp lệ \*/

void f ( void )

{

int32\_t wowAA = 0;

/\* Mở rộng thành wowAA = ( 0xffff ) + wowAA; \*/

wowAA = BB ( AA );

}

Trong ví dụ hợp lệ sau, tham số macro X không bị thay thế macro thêm.

|  |
| --- |
|  |

int32\_t speed;

int32\_t speed\_scale;

int32\_t scaled\_speed;

#define SCALE( X ) ( ( X ) \* X ## \_scale )

/\* mở rộng thành scaled\_speed = ( ( speed ) \* speed\_scale ); \*/

scaled\_speed = SCALE ( speed );

### Quy tắc 20.13: Một dòng có token đầu tiên là # phải là một chỉ thị tiền xử lý hợp lệ

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Khoảng trắng được phép giữa # và các token tiền xử lý.

**Lý do**  
Một chỉ thị tiền xử lý có thể được sử dụng để loại trừ mã nguồn có điều kiện cho đến khi gặp chỉ thị #else, #elif hoặc #endif tương ứng. Một chỉ thị tiền xử lý sai hoặc không hợp lệ trong đoạn mã bị loại trừ có thể không được phát hiện bởi trình biên dịch, có thể dẫn đến việc loại trừ nhiều mã hơn so với dự định.

Yêu cầu tất cả các chỉ thị tiền xử lý phải có cú pháp hợp lệ, ngay cả khi chúng xuất hiện trong một khối mã bị loại trừ, đảm bảo điều này không xảy ra.

**Ví dụ**

Trong ví dụ sau, tất cả mã giữa các chỉ thị #ifndef và #endif có thể bị loại trừ nếu AAA được định nghĩa. Lập trình viên đã dự định gán AAA cho x, nhưng chỉ thị #else được nhập sai và không được trình biên dịch chẩn đoán.

|  |
| --- |
|  |

#define AAA 2

int32\_t foo ( void )

{

int32\_t x = 0;

#ifndef AAA

x = 1;

#else1 /\* Không hợp lệ \*/

x = AAA;

#endif

return x;

}

Ví dụ sau là hợp lệ vì văn bản #start xuất hiện trong chú thích không phải là một token.

|  |
| --- |
|  |

/\*

#start không phải là một token trong chú thích

\*/

### Quy tắc 20.14: Tất cả các chỉ thị #else, #elif và #endif phải nằm trong cùng tệp với chỉ thị #if, #ifdef hoặc #ifndef mà chúng liên quan đến

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Lý do**  
Sự nhầm lẫn có thể xảy ra khi các khối mã được bao gồm hoặc loại trừ bằng cách sử dụng các chỉ thị biên dịch điều kiện được phân tán qua nhiều tệp. Yêu cầu rằng một chỉ thị #if phải được kết thúc trong cùng một tệp làm giảm độ phức tạp trực quan của mã và giảm khả năng xảy ra lỗi trong quá trình bảo trì.

**Lưu ý**: Các chỉ thị #if có thể được sử dụng trong các tệp được bao gồm miễn là chúng được kết thúc trong cùng tệp đó.

**Ví dụ**

|  |
| --- |
|  |

/\* file1.c \*/

#ifdef A /\* Hợp lệ \*/

#include "file1.h"

#endif

/\* Kết thúc file1.c \*/

/\* file2.c \*/

#if 1 /\* Không hợp lệ \*/

#include "file2.h"

/\* Kết thúc file2.c \*/

/\* file1.h \*/

#if 1 /\* Hợp lệ \*/

#endif

/\* Kết thúc file1.h \*/

/\* file2.h \*/

#endif

/\* Kết thúc file2.h \*/

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng khi lập trình C theo chuẩn MISRA.

### Thư viện chuẩn

#### Quy tắc 21.1: #define và #undef không được sử dụng trên các định danh hoặc tên macro được dành riêng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 54, 57, 58, 62, 71], **C99** [Không xác định 93, 100, 101, 104, 108, 116, 118, 130]  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Quy tắc này áp dụng cho các trường hợp sau:

* Các định danh hoặc tên macro bắt đầu bằng dấu gạch dưới (\_);
* Các định danh trong phạm vi tệp được mô tả trong Phần 7, "Thư viện", của Tiêu chuẩn;
* Tên macro được mô tả trong Phần 7, "Thư viện", của Tiêu chuẩn và được định nghĩa trong tệp tiêu chuẩn.

Quy tắc này cũng cấm việc sử dụng #define hoặc #undef trên định danh defined vì điều này dẫn đến hành vi không xác định một cách rõ ràng.

**Lưu ý**: Macro NDEBUG không được định nghĩa trong tệp tiêu chuẩn và có thể được #define.

**Lý do**  
Các định danh và tên macro được dành riêng được thiết kế để sử dụng bởi trình triển khai. Việc loại bỏ hoặc thay đổi ý nghĩa của một macro được dành riêng có thể dẫn đến hành vi không xác định.

**Ví dụ**

|  |
| --- |
|  |

#undef \_\_LINE\_\_ /\* Không hợp lệ - bắt đầu bằng \_ \*/

#define \_ GUARD\_H 1 /\* Không hợp lệ - bắt đầu bằng \_ \*/

#undef \_BUILTIN\_sqrt /\* Không hợp lệ - trình triển khai có thể sử dụng \_BUILTIN\_sqrt cho các mục đích khác, ví dụ như tạo lệnh sqrt \*/

#define defined /\* Không hợp lệ - định danh được dành riêng \*/

#define errno my\_errno /\* Không hợp lệ - định danh thư viện \*/

#define isneg( x ) ( ( x ) < 0 ) /\* Hợp lệ - quy tắc không bao gồm các định hướng tương lai của thư viện \*/

**Xem thêm**  
Quy tắc 20.4

### Quy tắc 21.2: Một định danh hoặc tên macro được dành riêng không được khai báo

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 57, 58, 64, 71], **C99** [Không xác định 93, 100, 101, 104, 108, 116, 118, 130]  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Xem phần mở rộng của Quy tắc 21.1 để biết mô tả về các định danh và tên macro liên quan.

**Lý do**  
Trình triển khai được phép dựa vào các định danh được dành riêng để hoạt động như được mô tả trong Tiêu chuẩn và có thể xử lý chúng đặc biệt. Nếu các định danh được dành riêng được tái sử dụng, chương trình có thể có hành vi không xác định.

**Ví dụ**

Trong ví dụ không hợp lệ sau, hàm memcpy được khai báo một cách tường minh. Phương pháp tuân thủ là bao gồm <string.h>.

|  |
| --- |
|  |

/\*

\* Bao gồm <stddef.h> để định nghĩa size\_t

\*/

#include <stddef.h>

extern void \*memcpy ( void \*restrict s1, const void \*restrict s2, size\_t n );

Trình triển khai được phép cung cấp định nghĩa macro dạng hàm cho mỗi hàm Thư viện Chuẩn bên cạnh bản thân hàm thư viện. Tính năng này thường được sử dụng bởi các nhà viết trình biên dịch để tạo các thao tác inline hiệu quả hơn thay cho lệnh gọi hàm thư viện.

Ví dụ, đoạn <math.h> khai báo sqrt có thể được viết sử dụng macro dạng hàm như sau:

|  |
| --- |
|  |

extern double sqrt ( double x );

#define sqrt( x ) ( \_BUILTIN\_sqrt ( x ) )

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng khi lập trình C theo chuẩn MISRA.

### Quy tắc 21.3: Các hàm cấp phát và giải phóng bộ nhớ của <stdlib.h> không được sử dụng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 19; Không xác định 9, 91, 92; Thực thi 69], **C99** [Không xác định 39, 40; Không xác định 8, 9, 168–171; Thực thi J.3.12(35)]  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Các định danh calloc, malloc, realloc và free không được sử dụng và không có macro nào với tên này được mở rộng.

**Lý do**  
Việc sử dụng các hàm cấp phát và giải phóng bộ nhớ động do Thư viện Chuẩn cung cấp có thể dẫn đến hành vi không xác định, ví dụ:

* Bộ nhớ không được cấp phát động nhưng lại bị giải phóng;
* Con trỏ đến bộ nhớ đã giải phóng được sử dụng bằng bất kỳ cách nào;
* Truy cập bộ nhớ đã cấp phát trước khi gán giá trị cho nó.

**Lưu ý**: Quy tắc này là một trường hợp cụ thể của Dir 4.12.

**Xem thêm**  
Dir 4.12, Quy tắc 18.7, Quy tắc 22.1, Quy tắc 22.2

### Quy tắc 21.4: Tệp tiêu chuẩn <setjmp.h> không được sử dụng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 14; Không xác định 64–67], **C99** [Không xác định 32; Không xác định 118–121, 173]  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Không có các tiện ích nào được xác định là được cung cấp bởi <setjmp.h> được sử dụng.

**Lý do**  
setjmp và longjmp cho phép cơ chế gọi hàm thông thường bị bỏ qua. Việc sử dụng chúng có thể dẫn đến hành vi không xác định và không rõ ràng.

### Quy tắc 21.5: Tệp tiêu chuẩn <signal.h> không được sử dụng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 67–69; Thực thi 48–52], **C99** [Không xác định 122–127; Thực thi J.3.12(12)]  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Không có các tiện ích nào được xác định là được cung cấp bởi <signal.h> được sử dụng.

**Lý do**  
Việc xử lý tín hiệu chứa hành vi xác định và không xác định tùy theo trình triển khai.

### Quy tắc 21.6: Các hàm nhập/xuất Thư viện Chuẩn không được sử dụng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 2–5, 16–18; Không xác định 77–89; Thực thi 53–68], **C99** [Không xác định 3–6, 34–37; Không xác định 138–166, 186; Thực thi J.3.12(14–32)]  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Quy tắc này áp dụng cho các hàm được cung cấp bởi <stdio.h> và trong C99, các hàm tương đương sử dụng ký tự rộng được cung cấp trong Phần 7.24.2 và 7.24.3 của Tiêu chuẩn C99 thông qua <wchar.h>. Không được sử dụng bất kỳ định danh nào trong số này và không có macro nào với tên này được mở rộng.

**Lý do**  
Các luồng dữ liệu và I/O tập tin có các hành vi không xác định, không rõ ràng và phụ thuộc vào trình triển khai.

**Xem thêm**  
Quy tắc 22.1, Quy tắc 22.3, Quy tắc 22.4, Quy tắc 22.5, Quy tắc 22.6

Hy vọng phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng khi lập trình C theo chuẩn MISRA.

### Quy tắc 21.7: Các hàm atof, atoi, atol và atoll của <stdlib.h> không được sử dụng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 90], **C99** [Không xác định 113]  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Các định danh atof, atoi, atol và (chỉ với C99) atoll không được sử dụng và không có macro nào với tên này được mở rộng.

**Lý do**  
Các hàm này có hành vi không xác định khi chuỗi không thể được chuyển đổi.

### Quy tắc 21.8: Các hàm thư viện abort, exit, getenv và system của <stdlib.h> không được sử dụng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 93; Thực thi 70–73], **C99** [Không xác định 172, 174, 175; Thực thi J.3.12(36–38)]  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Các định danh abort, exit, getenv và system không được sử dụng và không có macro nào với tên này được mở rộng.

**Lý do**  
Các hàm này có hành vi không xác định và phụ thuộc vào trình triển khai.

### Quy tắc 21.9: Các hàm thư viện bsearch và qsort của <stdlib.h> không được sử dụng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 20, 21], **C99** [Không xác định 41, 42; Không xác định 176–178]  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Các định danh bsearch và qsort không được sử dụng và không có macro nào với tên này được mở rộng.

**Lý do**  
Nếu hàm so sánh không nhất quán khi so sánh các phần tử hoặc nếu nó thay đổi bất kỳ phần tử nào, hành vi sẽ không xác định. Hơn nữa, việc triển khai qsort thường sử dụng đệ quy, điều này có thể gây áp lực không xác định lên tài nguyên ngăn xếp, điều này đặc biệt quan trọng trong các hệ thống nhúng với kích thước ngăn xếp cố định, nhỏ.

### Quy tắc 21.10: Các hàm thời gian và ngày tháng của Thư viện Chuẩn không được sử dụng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90** [Không xác định 22; Không xác định 80, 97; Thực thi 75, 76], **C99** [Không xác định 43, 44; Không xác định 146, 154, 182; Thực thi J.3.12(39–42)]  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Không được sử dụng bất kỳ tiện ích nào được cung cấp bởi <time.h>. Trong C99, định danh wcsftime cũng không được sử dụng và không có macro nào với tên này được mở rộng.

**Lý do**  
Các hàm thời gian và ngày tháng có các hành vi không xác định, không rõ ràng và phụ thuộc vào trình triển khai.

### Quy tắc 21.11: Tệp tiêu chuẩn <tgmath.h> không được sử dụng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C99** [Không xác định 184, 185]  
**Loại**: Bắt buộc  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C99

**Mở rộng**  
Không được sử dụng bất kỳ tiện ích nào được cung cấp bởi <tgmath.h>.

**Lý do**  
Việc sử dụng các tiện ích của <tgmath.h> có thể dẫn đến hành vi không xác định.

**Ví dụ**

|  |
| --- |
|  |

#include <tgmath.h>

float f1, f2;

void f(void) {

f1 = sqrt(f2); /\* Không hợp lệ - sử dụng hàm sqrt tổng quát \*/

}

|  |
| --- |
|  |

#include <math.h>

float f1, f2;

void f(void) {

f1 = sqrtf(f2); /\* Hợp lệ - sử dụng phiên bản float của sqrt \*/

}

Hy vọng các phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng khi lập trình C theo chuẩn MISRA.

### Rule 21.12: Không sử dụng các tính năng xử lý ngoại lệ của <fenv.h>

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C99** [Không xác định 27, 28; Không xác định 109–111; Thực thi J.3.6(8)]  
**Loại**: Tư vấn  
**Phân tích**: Có thể quyết định được, Đơn vị Dịch chuyển Dịch duy nhất  
**Áp dụng cho**: C99

**Mở rộng**  
Không được sử dụng các định danh feclearexcept, fegetexceptflag, feraiseexcept, fesetexceptflag và fetestexcept, và không có macro nào với tên này được mở rộng. Các macro FE\_INEXACT, FE\_DIVBYZERO, FE\_UNDERFLOW, FE\_OVERFLOW, FE\_INVALID và FE\_ALL\_EXCEPT, cùng với bất kỳ macro ngoại lệ số học dấu phẩy động do triển khai định nghĩa nào, không được sử dụng.

**Lý do**  
Trong một số trường hợp, giá trị của các cờ trạng thái số học dấu phẩy động không được xác định và cố gắng truy cập vào chúng có thể dẫn đến hành vi không xác định. Thứ tự mà các ngoại lệ được kích hoạt bởi hàm feraiseexcept không được xác định và có thể làm cho chương trình không hoạt động đúng như thiết kế.

**Ví dụ**

|  |
| --- |
|  |

#include <fenv.h>

void f(float32\_t x, float32\_t y) {

float32\_t z;

feclearexcept(FE\_DIVBYZERO); /\* Không tuân thủ \*/

z = x / y;

if (fetestexcept(FE\_DIVBYZERO)) { /\* Không tuân thủ \*/

} else {

#pragma STDC FENV\_ACCESS ON

z = x \* y;

}

if (z > x) {

#pragma STDC FENV\_ACCESS OFF

if (fetestexcept(FE\_OVERFLOW)) { /\* Không tuân thủ \*/

}

}

}

### Tài nguyên

### Quy tắc 22.1: Tất cả tài nguyên được lấy động bằng các hàm Thư viện Chuẩn phải được giải phóng một cách rõ ràng

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Loại**: Bắt buộc  
**Phân tích**: Không thể quyết định được, Hệ thống  
**Áp dụng cho**: C90, C99

**Mở rộng**  
Các hàm Thư viện Chuẩn phân bổ tài nguyên gồm malloc, calloc, realloc và fopen.

**Lý do**  
Nếu tài nguyên không được giải phóng một cách rõ ràng, có khả năng xảy ra lỗi do cạn kiệt tài nguyên. Giải phóng tài nguyên càng sớm càng tốt giúp giảm khả năng xảy ra tình trạng cạn kiệt.

**Ví dụ**

|  |
| --- |
|  |

#include <stdlib.h>

int main(void) {

void \*b = malloc(40);

/\* Không tuân thủ - bộ nhớ động không được giải phóng \*/

return 1;

}

|  |
| --- |
|  |

#include <stdio.h>

int main(void) {

FILE \*fp = fopen("tmp", "r");

/\* Không tuân thủ - tệp không được đóng \*/

return 1;

}

Trong ví dụ không tuân thủ sau, tay cầm trên “tmp-1” bị mất khi “tmp-2” được mở.

|  |
| --- |
|  |

#include <stdio.h>

int main(void) {

FILE \*fp;

fp = fopen("tmp-1", "w");

fprintf(fp, "\*");

/\* Tệp "tmp-1" cần được đóng ở đây, nhưng luồng bị rò rỉ. \*/

fp = fopen("tmp-2", "w");

fprintf(fp, "!");

fclose(fp);

return 0;

}

Hy vọng các phần dịch này giúp bạn hiểu rõ hơn về các quy tắc và cách áp dụng trong lập trình C theo chuẩn MISRA.

**Tham khảo thêm**: Dir 4.12, Dir 4.13, Rule 21.3, Rule 21.6

### Quy tắc 22.2: Một khối bộ nhớ chỉ được giải phóng nếu nó được phân bổ bằng các hàm của Thư viện Chuẩn

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**C90 [Hành vi không xác định 92], C99 [Hành vi không xác định 169]**

**Danh mục**: Bắt buộc

**Phân tích**: Không xác định, Hệ thống

**Áp dụng cho**: C90, C99

**Mở rộng**: Các hàm Thư viện Chuẩn để phân bổ bộ nhớ bao gồm malloc, calloc, và realloc. Một khối bộ nhớ được giải phóng khi địa chỉ của nó được truyền vào free và có thể được giải phóng khi địa chỉ được truyền vào realloc. Sau khi đã giải phóng, một khối bộ nhớ không còn được coi là đã phân bổ và do đó không thể giải phóng lại sau đó.

**Lý do**: Giải phóng bộ nhớ không được phân bổ hoặc giải phóng lại cùng một vùng bộ nhớ đã phân bổ dẫn đến hành vi không xác định.

**Ví dụ**:

|  |
| --- |
|  |

#include <stdlib.h>

void fn (void) {

int32\_t a;

/\* Không tuân thủ - a không trỏ đến vùng nhớ đã phân bổ \*/

free(&a);

}

void g (void) {

char \*p = (char \*) malloc(512);

char \*q = p;

free(p);

/\* Không tuân thủ - khối bộ nhớ đã được giải phóng lần thứ hai \*/

free(q);

/\* Không tuân thủ - khối bộ nhớ có thể được giải phóng lần thứ ba \*/

p = (char \*) realloc(p, 1024);

}

### Quy tắc 22.3: Không được mở cùng một tệp để đọc và ghi đồng thời trên các luồng khác nhau

* **C90 [Thực hiện 61], C99 [Thực hiện J.3.12(22)]**
* **Danh mục**: Bắt buộc
* **Phân tích**: Không xác định, Hệ thống
* **Áp dụng cho**: C90, C99

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Mở rộng**: Quy tắc này áp dụng cho các tệp được mở bằng các hàm Thư viện Chuẩn. Nó cũng có thể áp dụng cho các tính năng tương tự được cung cấp bởi môi trường thực thi.

**Lý do**: Chuẩn không chỉ rõ hành vi nếu một tệp vừa được ghi vừa được đọc qua các luồng khác nhau.

**Lưu ý**: Việc mở một tệp nhiều lần để chỉ đọc là chấp nhận được.

**Ví dụ**:

|  |
| --- |
|  |

#include <stdio.h>

void fn (void) {

FILE \*fw = fopen("tmp", "r+"); /\* "r+" mở để đọc/ghi \*/

FILE \*fr = fopen("tmp", "r"); /\* Không tuân thủ \*/

}

**Tham khảo thêm**:

* **Rule 21.6**: Không sử dụng các hàm nhập/xuất chuẩn.

### Quy tắc 22.4: Không được phép ghi vào một luồng chỉ được mở để đọc

* **Danh mục**: Bắt buộc
* **Phân tích**: Không xác định, Hệ thống
* **Áp dụng cho**: C90, C99

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Lý do**: Chuẩn không chỉ định hành vi nếu cố gắng ghi vào một luồng chỉ mở để đọc. Do đó, việc ghi vào luồng như vậy được coi là không an toàn.

**Ví dụ**:

|  |
| --- |
|  |

#include <stdio.h>

void fn(void) {

FILE \*fp = fopen("tmp", "r");

(void) fprintf(fp, "What happens now?"); /\* Không tuân thủ \*/

(void) fclose(fp);

}

**Tham khảo thêm**:

* **Rule 21.6**: Không sử dụng các hàm nhập/xuất chuẩn.

### Quy tắc 22.5: Không được tham chiếu con trỏ tới đối tượng FILE

* **Danh mục**: Bắt buộc
* **Phân tích**: Không xác định, Hệ thống
* **Áp dụng cho**: C90, C99

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Mở rộng**: Không được phép tham chiếu trực tiếp hoặc gián tiếp con trỏ tới đối tượng FILE (ví dụ: thông qua gọi memcpy hoặc memcmp).

**Lý do**: Chuẩn (C90 Mục 7.9.3(6), C99 Mục 7.19.3(6)) nêu rằng địa chỉ của đối tượng FILE được sử dụng để điều khiển một luồng có thể quan trọng và bản sao của đối tượng có thể không cho kết quả giống nhau. Quy tắc này đảm bảo rằng không thể tạo bản sao như vậy.

**Ví dụ**:

|  |
| --- |
|  |

#include <stdio.h>

FILE \*pf1;

FILE \*pf2;

FILE f3;

pf2 = pf1; /\* Tuân thủ \*/

f3 = \*pf2; /\* Không tuân thủ \*/

Ví dụ tiếp theo giả định rằng FILE \* chỉ rõ một kiểu đầy đủ có thành viên pos:

|  |
| --- |
|  |

pf1->pos = 0; /\* Không tuân thủ \*/

**Tham khảo thêm**:

* **Rule 21.6**: Không sử dụng các hàm nhập/xuất chuẩn.

### Quy tắc 22.6: Giá trị của con trỏ tới FILE không được sử dụng sau khi luồng tương ứng đã đóng

* **C99 [Hành vi không xác định 140]**
* **Danh mục**: Bắt buộc
* **Phân tích**: Không xác định, Hệ thống
* **Áp dụng cho**: C90, C99

|  |  |
| --- | --- |
| Rule |  |
|  | |
| Phân loại | Bắt buộc |
| Phân tích | Quyết định được, Đơn vị Dịch thuật Đơn |
| Áp dụng | C90, C99 |

**Lý do**: Chuẩn nêu rằng giá trị của con trỏ FILE không xác định sau khi một luồng đã được đóng.

**Ví dụ**:`#include <stdio.h>

void fn(void) {

FILE \*fp;

void \*p;

fp = fopen("tmp", "w");

if (fp == NULL) {

error\_action();

}

fclose(fp);

fprintf(fp, "?"); /\* Không tuân thủ \*/

p = fp; /\* Không tuân thủ \*/

}

**Tham khảo thêm**:

* **Dir 4.13**: Yêu cầu việc xử lý tài nguyên được quản lý chính xác.
* **Rule 21.6**: Không sử dụng các hàm nhập/xuất chuẩn.

### Tóm tắt Phụ lục A: Các hướng dẫn

**Thực thi**

* **Dir 1.1 (Bắt buộc)**: Mọi hành vi do nhà phát triển quyết định mà chương trình phụ thuộc vào phải được ghi chép và hiểu rõ.

**Biên dịch và xây dựng**

* **Dir 2.1 (Bắt buộc)**: Tất cả các tệp nguồn phải được biên dịch mà không gặp bất kỳ lỗi biên dịch nào.

**Truy xuất yêu cầu**

* **Dir 3.1 (Bắt buộc)**: Tất cả mã phải có thể truy xuất được đến các yêu cầu đã được ghi chép.

**Thiết kế mã**

* **Dir 4.1 (Bắt buộc)**: Các lỗi thời gian chạy phải được giảm thiểu.
* **Dir 4.2 (Tư vấn)**: Mọi việc sử dụng ngôn ngữ lắp ráp nên được ghi chép.
* **Dir 4.3 (Bắt buộc)**: Ngôn ngữ lắp ráp phải được đóng gói và cách ly.
* **Dir 4.4 (Tư vấn)**: Không nên có phần mã nào bị “bình luận ra”.
* **Dir 4.5 (Tư vấn)**: Các định danh trong cùng một không gian tên với phạm vi hiển thị chồng chéo nên dễ phân biệt về mặt hình thức.
* **Dir 4.6 (Tư vấn)**: Nên sử dụng các typedef chỉ ra kích thước và dấu thay vì các kiểu số cơ bản.
* **Dir 4.7 (Bắt buộc)**: Nếu một hàm trả về thông tin lỗi, thì thông tin lỗi đó phải được kiểm tra.
* **Dir 4.8 (Tư vấn)**: Nếu một con trỏ tới một cấu trúc hoặc union không bao giờ được tham chiếu trong một đơn vị dịch, thì việc thực hiện đối tượng nên được ẩn.
* **Dir 4.9 (Tư vấn)**: Nên sử dụng hàm thay vì macro giống hàm khi chúng có thể thay thế cho nhau.
* **Dir 4.10 (Bắt buộc)**: Phải có biện pháp phòng ngừa để tránh nội dung của một tệp tiêu đề được bao gồm nhiều lần.
* **Dir 4.11 (Bắt buộc)**: Giá trị được truyền đến các hàm thư viện phải được kiểm tra tính hợp lệ.
* **Dir 4.12 (Bắt buộc)**: Không được sử dụng việc cấp phát bộ nhớ động.
* **Dir 4.13 (Tư vấn)**: Các hàm được thiết kế để thực hiện các thao tác trên một tài nguyên phải được gọi theo trình tự phù hợp.

**Môi trường C tiêu chuẩn**

* **Rule 1.1 (Bắt buộc)**: Chương trình không được vi phạm cú pháp C tiêu chuẩn và không được vượt quá giới hạn dịch của thực thi.
* **Rule 1.2 (Tư vấn)**: Không nên sử dụng các mở rộng ngôn ngữ.
* **Rule 1.3 (Bắt buộc)**: Không được có sự xuất hiện của hành vi không xác định hoặc hành vi không xác định nghiêm trọng.

**Mã không sử dụng**

* **Rule 2.1 (Bắt buộc)**: Dự án không được chứa mã không thể đạt tới.
* **Rule 2.2 (Bắt buộc)**: Không được có mã chết.
* **Rule 2.3 (Tư vấn)**: Dự án không nên chứa khai báo kiểu không sử dụng.
* **Rule 2.4 (Tư vấn)**: Dự án không nên chứa khai báo thẻ không sử dụng.
* **Rule 2.5 (Tư vấn)**: Dự án không nên chứa khai báo macro không sử dụng.
* **Rule 2.6 (Tư vấn)**: Một hàm không nên chứa các khai báo nhãn không sử dụng.
* **Rule 2.7 (Tư vấn)**: Không nên có tham số không sử dụng trong các hàm.

**Nhận xét**

* **Rule 3.1 (Bắt buộc)**: Các chuỗi ký tự /\* và // không được sử dụng trong một nhận xét.
* **Rule 3.2 (Bắt buộc)**: Không được sử dụng ghép dòng trong nhận xét kiểu //.

**Bộ ký tự và quy ước từ vựng**

* **Rule 4.1 (Bắt buộc)**: Các chuỗi thoát thập phân và thập lục phân phải được kết thúc.
* **Rule 4.2 (Tư vấn)**: Không nên sử dụng chuỗi ba ký tự (trigraphs).

**Định danh**

* **Rule 5.1 (Bắt buộc)**: Các định danh ngoài phải khác biệt.
* **Rule 5.2 (Bắt buộc)**: Các định danh khai báo trong cùng một phạm vi và không gian tên phải khác biệt.
* **Rule 5.3 (Bắt buộc)**: Một định danh khai báo trong một phạm vi bên trong không được che lấp một định danh đã khai báo trong phạm vi bên ngoài.
* **Rule 5.4 (Bắt buộc)**: Các định danh macro phải khác biệt.
* **Rule 5.5 (Bắt buộc)**: Các định danh phải khác với tên macro.
* **Rule 5.6 (Bắt buộc)**: Tên typedef phải là một định danh duy nhất.
* **Rule 5.7 (Bắt buộc)**: Tên thẻ phải là một định danh duy nhất.
* **Rule 5.8 (Bắt buộc)**: Các định danh định nghĩa đối tượng hoặc hàm có liên kết ngoài phải là duy nhất.
* **Rule 5.9 (Tư vấn)**: Các định danh định nghĩa đối tượng hoặc hàm có liên kết trong nên là duy nhất.

**Kiểu dữ liệu**

* **Rule 6.1 (Bắt buộc)**: Các trường bit chỉ được khai báo với kiểu phù hợp.
* **Rule 6.2 (Bắt buộc)**: Trường bit một bit có tên không được có kiểu dấu.

**Hằng số và ký tự**

* **Rule 7.1 (Bắt buộc)**: Không được sử dụng hằng số bát phân.
* **Rule 7.2 (Bắt buộc)**: Một hậu tố “u” hoặc “U” phải được áp dụng cho tất cả các hằng số số nguyên được biểu diễn dưới dạng kiểu không dấu.
* **Rule 7.3 (Bắt buộc)**: Ký tự viết thường “l” không được sử dụng trong hậu tố hằng số.
* **Rule 7.4 (Bắt buộc)**: Chuỗi ký tự không được gán cho đối tượng trừ khi kiểu của đối tượng là “con trỏ tới char được gán const”.