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

---

## 8 Quy tắc

### 8.1 Một môi trường C tiêu chuẩn

#### Quy tắc 1.1: 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.

- [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.

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

---

### 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.

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

---

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

- 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

- 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

### 8.2 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]

- 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

- [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

- 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

- 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

- 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

- 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

- 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;

}

```

### 8.3 Bình luận (Comments)

#### Quy tắc 3.1: Chuỗi ký tự /\* và // không được sử dụng trong một bình luận

- 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

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

---

#### Quy tắc 3.2: Không được sử dụng nối dòng trong các bình luận //

- 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

- 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.

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

---

##### 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

- 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:

---

### 8.5 Định danh

#### Quy tắc 5.1: Các định danh bên ngoài phải khác biệt

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

### 8.6 Các loại

#### Quy tắc 6.1: Các trường bit chỉ nên được khai báo với loại thích hợp

- 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

- 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.

### 8.7 Hằng số và hằng số số học

#### Quy tắc 7.1: Không được sử dụng hằng bát phân

- [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

- 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

- 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”

- 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

### 8.8 Khai báo và định nghĩa

#### Quy tắc 8.1: Các loại phải được chỉ định rõ ràng

- 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

- 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

- 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

- 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: Một đối tượng hoặc hàm có liên kết bên ngoài phải được khai báo một lần trong một và chỉ một tệp

- [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: Một định danh có liên kết bên ngoài phải có chính xác một định nghĩa bên ngoài

- 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: 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

- [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: 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

- 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

- 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

- 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 ràng

- 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

- 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

- 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 )

{

}

```

### 8.9 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

- 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

- 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

- 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

- 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 ràng

- 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

- 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

- 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

- 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

- 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

- 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

### 8.11 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

- 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

- \*\*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

- \*\*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

- \*\*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

- \*\*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

- \*\*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

- \*\*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

- \*\*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

- \*\*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

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

---

### Rule 12.1: The precedence of operators within expressions should be made explicit

Category: Advisory

Analysis: Decidable, Single Translation Unit

Applies to: C90, C99

\*\*Amplification\*\*

The following table is used in the definition of this rule.

| Description | Operator or Operand | Precedence |

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

| Primary | identifier, constant, string literal, (expression) | 16 (high) |

| Postfix | [], () (function call), ., ->, ++ (post-increment), -- (post-decrement), (), {} (C99: compound literal) | 15 |

| Unary | ++ (pre-increment), -- (pre-decrement), &, \*, +, -, ~, !, sizeof, defined (preprocessor) | 14 |

| Cast | () | 13 |

| Multiplicative | \*, /, % | 12 |

| Additive | +, - | 11 |

| Bitwise shift | <<, >> | 10 |

| Relational | <, >, <=, >= | 9 |

| Equality | ==, != | 8 |

| Bitwise AND | & | 7 |

| Bitwise XOR | ^ | 6 |

| Bitwise OR | | | 5 |

| Logical AND | && | 4 |

| Logical OR | || | 3 |

| Conditional | ?: | 2 |

| Assignment | =, \*=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |= | 1 |

| Comma | , | 0 (low) |

The precedences used in this table are chosen to allow a concise description of the rule. They are not necessarily the same as those that might be encountered in other descriptions of operator precedence.

For the purposes of this rule, the precedence of an expression is the precedence of the element (operand or operator) at the root of the parse tree for that expression.

For example: the parse tree for the expression `a << b + c` can be represented as:

```

<<

/ \

a +

/ \

b c

```

The element at the root of this parse tree is `<<` so the expression has precedence 10.

\*\*The following advice is given:\*\*

- The operand of the sizeof operator should be enclosed in parentheses;

- An expression whose precedence is in the range 2 to 12 should have parentheses around any operand that has both:

- Precedence of less than 13, and

- Precedence greater than the precedence of the expression.

\*\*Rationale\*\*

The C language has a relatively large number of operators and their relative precedences are not intuitive. This can lead less experienced programmers to make mistakes. Using parentheses to make operator precedence explicit removes the possibility that the programmer’s expectations are incorrect. It also makes the original programmer’s intention clear to reviewers or maintainers of the code.

It is recognized that overuse of parentheses can clutter the code and reduce its readability. This rule aims to achieve a compromise between code that is hard to understand because it contains either too many or too few parentheses.

\*\*Examples\*\*

The following example shows expressions with a unary or postfix operator whose operands are either primary-expressions or expressions whose top-level operators have precedence 15.

```c

a[i]->n; // Compliant - no need to write (a[i])->n

\*p++; // Compliant - no need to write \*(p++)

sizeof x + y; // Non-compliant - write either sizeof (x) + y or sizeof (x + y)

```

The following example shows expressions containing operators at the same precedence level. All of these are compliant but, depending on the types of a, b, and c, any expression with more than one operator may violate other rules.

```c

a + b;

a + b + c;

(a + b) + c;

a + (b + c);

a + b - c + d;

(a + b) - (c + d);

```

The following example shows a variety of mixed-operator expressions:

```c

x = f(a + b, c); // Compliant - no need to write f((a + b), c)

// Non-compliant

// Operands of conditional operator (precedence 2) are:

// == precedence 8 needs parentheses

// a precedence 16 does not need parentheses

// - precedence 11 needs parentheses

x = a == b ? a : a - b;

// Compliant

x = (a == b) ? a : (a - b);

```

### Rule 12.2: The right-hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left-hand operand

Category: Required

Analysis: Undecidable, System

Applies to: C90, C99

\*\*Rationale\*\*

If the right-hand operand is negative, or greater than or equal to the width of the left-hand operand, then the behavior is undefined. If, for example, the left-hand operand of a left-shift or right-shift is a 16-bit integer, then it is important to ensure that this is shifted only by a number in the range 0 to 15. See Section 8.10 for a description of essential type and the limitations on the essential types for the operands of shift operators.

There are various ways of ensuring this rule is followed. The simplest is for the right-hand operand to be a constant (whose value can then be statically checked). Use of an unsigned integer type will ensure that the operand is non-negative, so then only the upper limit needs to be checked (dynamically at runtime or by review). Otherwise, both limits will need to be checked.

\*\*Example\*\*

```c

u8a = u8a << 7; // Compliant

u8a = u8a << 8; // Non-compliant

u16a = (uint16\_t)u8a << 9; // Compliant

```

To assist in understanding the following examples, it should be noted that the essential type of `1u` is essentially unsigned char, whereas the essential type of `1UL` is essentially unsigned long.

```c

1u << 10u; // Non-compliant

(uint16\_t)1u << 10u; // Compliant

1UL << 10u; // Compliant

```

---

### Rule 12.3: The comma operator should not be used

Category: Advisory

Analysis: Decidable, Single Translation Unit

Applies to: C90, C99

\*\*Rationale\*\*

Use of the comma operator is generally detrimental to the readability of code, and the same effect can usually be achieved by other means.

\*\*Example\*\*

```c

f((1, 2), 3); // Non-compliant - how many parameters?

```

The following example is non-compliant with this rule and other rules:

```c

for (i = 0, p = &a[0]; i < N; ++i, ++p) {

// Non-compliant

}

```

---

### Rule 12.4: Evaluation of constant expressions should not lead to unsigned integer wrap-around

Category: Advisory

Analysis: Decidable, Single Translation Unit

Applies to: C90, C99

\*\*Amplification\*\*

This rule applies to expressions that satisfy the constraints for a constant expression, whether or not they appear in a context that requires a constant expression. If an expression is not evaluated, for example, because it appears in the right operand of a logical AND operator whose left operand is always false, then this rule does not apply.

\*\*Rationale\*\*

Unsigned integer expressions do not strictly overflow, but instead wrap-around. Although there may be good reasons to use modulo arithmetic at runtime, it is less likely for its use to be intentional at compile-time.

\*\*Example\*\*

The expression associated with a case label is required to be a constant expression. If an unsigned wrap-around occurs during evaluation of a case expression, it is likely to be unintentional. On a machine with a 16-bit `int` type, any value of `BASE` greater than or equal to 65024 would result in wrap-around in the following example:

```c

#define BASE 65024u

switch (x) {

case BASE + 0u:

f();

break;

case BASE + 1u:

g();

break;

case BASE + 512u: // Non-compliant - wraps to 0

h();

break;

}

```

The controlling expression of a `#if` or `#elif` preprocessor directive is required to be a constant expression.

```c

#if 1u + (0u - 10u) // Non-compliant as (0u - 10u) wraps

```

In this example, the expression `DELAY + WIDTH` has the value 70000 but this will wrap-around to 4464 on a machine with a 16-bit `int` type.

```c

#define DELAY 10000u

#define WIDTH 60000u

void fixed\_pulse(void) {

uint16\_t off\_time16 = DELAY + WIDTH; // Non-compliant

}

```

This rule does not apply to the expression `c + 1` in the following compliant example as it accesses an object and therefore does not satisfy the constraints for a constant expression:

```c

const uint16\_t c = 0xffffu;

void f(void) {

uint16\_t y = c + 1u; // Compliant

}

```

In the following example, the sub-expression `(0u - 1u)` leads to unsigned integer wrap-around. In the initialization of `x`, the sub-expression is not evaluated and the expression is therefore compliant. However, in the initialization of `y`, it may be evaluated and the expression is therefore non-compliant.

```c

bool\_t b;

void g(void) {

uint16\_t x = (0u == 0u) ? 0u : (0u - 1u); // Compliant

uint16\_t y = b ? 0u : (0u - 1u); // Non-compliant

}

```

---

### Rule 12.1 Examples (Continued)

```c

/\* Compliant

\* Operands of << operator (precedence 10) are:

\* a precedence 16 does not need parentheses

\* (E) precedence 16 already parenthesized

\*/

x = a << (b + c);

/\* Compliant

\* Operands of && operator (precedence 4) are:

\* a precedence 16 does not need parentheses

\* && precedence 4 does not need parentheses

\*/

if (a && b && c) {

}

/\* Compliant

\* Operands of && operator (precedence 4) are:

\* defined(X) precedence 14 does not need parentheses

\* (E) precedence 16 already parenthesized

\*/

#if defined(X) && ((X + Y) > Z)

#endif

/\* Compliant

\* Operands of && operator (precedence 4) are:

\* !defined(X) precedence 14 does not need parentheses

\* defined(Y) precedence 14 does not need parentheses

\* Operand of ! operator (precedence 14) is:

\* defined(X) precedence 14 does not need parentheses

\*/

#if !defined(X) && defined(Y)

#endif

/\* Compliant - parsed as (x = a), b \*/

x = a, b;

```

### Rule 13.1: Initializer lists shall not contain persistent side effects

\*\*Category:\*\* Required

\*\*Analysis:\*\* Undecidable, System

\*\*Applies to:\*\* C99

\*\*Rationale:\*\*

C90 constraints the initializers for automatic objects with aggregate types to contain only constant expressions. However, C99 permits automatic aggregate initializers to contain expressions that are evaluated at runtime. It also permits compound literals which behave as anonymous initialized objects. The order in which side effects occur during the evaluation of the expressions in an initializer list is unspecified and the behavior of the initialization is therefore unpredictable if those side effects are persistent.

\*\*Example:\*\*

```c

volatile uint16\_t v1;

void f (void) {

// Non-compliant - volatile access is persistent side effect

uint16\_t a[2] = { v1, 0 };

}

void g (uint16\_t x, uint16\_t y) {

// Compliant - no side effects

uint16\_t a[2] = { x + y, x - y };

}

uint16\_t x = 0u;

extern void p (uint16\_t a[2]);

void h (void) {

// Non-compliant - two side effects

p((uint16\_t[2]) { x++, x++ });

}

```

\*\*See also:\*\* Rule 13.2

### Rule 13.2: The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders

\*\*Category:\*\* Required

\*\*Analysis:\*\* Undecidable, System

\*\*Applies to:\*\* C90, C99

\*\*Amplification:\*\*

Between any two adjacent sequence points or within any full expression:

1. No object shall be modified more than once;

2. No object shall be both modified and read unless any such read of the object’s value contributes towards computing the value to be stored into the object;

3. There shall be no more than one modification access with volatile-qualified type;

4. There shall be no more than one read access with volatile-qualified type.

Note: An object might be accessed indirectly, by means of a pointer or a called function, as well as being accessed directly by the expression. This Amplification is intentionally stricter than the headline of the rule. As a result, expressions such as:

```c

x = x = 0;

```

are not permitted by this rule even though the value and the persistent side effects, provided that x is not volatile, are independent of the order of evaluation or side effects.

Sequence points are summarized in Annex C of both the C90 and C99 standards. The sequence points in C90 are a subset of those in C99. Full expressions are defined in Section 6.6 of the C90 standard and Section 6.8 of the C99 standard.

\*\*Rationale:\*\*

The Standard gives considerable flexibility to compilers when evaluating expressions. Most operators can have their operands evaluated in any order. The main exceptions are:

- Logical AND `&&` in which the second operand is evaluated only if the first operand evaluates to non-zero;

- Logical OR `||` in which the second operand is evaluated only if the first operand evaluates to zero;

- The conditional operator `?:` in which the first operand is always evaluated and then either the second or third operand is evaluated;

- The comma `,` operator in which the first operand is evaluated and then the second operand is evaluated.

Note: The presence of parentheses may alter the order in which operators are applied. However, this does not affect the order of evaluation of the lowest-level operands, which may be evaluated in any order. Many of the common instances of the unpredictable behavior associated with expression evaluation can be avoided by following the advice given by Rule 13.3 and Rule 13.4.

\*\*Examples:\*\*

When the COPY\_ELEMENT macro is invoked in this non-compliant example, `i` is read twice and modified twice. It is unspecified whether the order of operations on `i` is:

- Read, modify, read, modify, or

- Read, read, modify, modify.

```c

#define COPY\_ELEMENT(index) (a[(index)] = b[(index)])

COPY\_ELEMENT(i++);

```

In this non-compliant example, the order in which `v1` and `v2` are read is unspecified.

```c

extern volatile uint16\_t v1, v2;

uint16\_t t;

t = v1 + v2;

```

In this compliant example, `PORT` is read and modified.

```c

extern volatile uint8\_t PORT;

PORT = PORT & 0x80u;

```

The order of evaluation of function arguments is unspecified as is the order in which side effects occur as shown in this non-compliant example.

```c

uint16\_t i = 0;

/\*

\* Unspecified whether this call is equivalent to:

\* f(0, 0)

\* or f(0, 1)

\*/

f(i++, i);

```

The relative order of evaluation of a function designator and function arguments is unspecified. In this non-compliant example, if the call to `g` modifies `p` then it is unspecified whether the function designator `p->f` uses the value of `p` prior to the call of `g` or after it.

```c

p->f(g(&p));

```

\*\*See also:\*\* Dir 4.9, Rule 13.1, Rule 13.3, Rule 13.4

### Rule 13.3: A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator

\*\*Category:\*\* Advisory

\*\*Analysis:\*\* Decidable, Single Translation Unit

\*\*Applies to:\*\* C90, C99

\*\*Amplification:\*\*

A function call is considered to be a side effect for the purposes of this rule. All sub-expressions of the full expression are treated as if they were evaluated for the purposes of this rule, even if specified as not being evaluated by The Standard.

\*\*Rationale:\*\*

The use of increment and decrement operators in combination with other operators is not recommended because:

- It can significantly impair the readability of the code;

- It introduces additional side effects into a statement with the potential for undefined behavior (covered by Rule 13.2).

It is clearer to use these operations in isolation from any other operators.

\*\*Example:\*\*

The expression:

```c

u8a = u8b++;

```

is non-compliant. The non-compliant expression statement:

```c

u8a = ++u8b + u8c--;

```

is clearer when written as the following sequence:

```c

++u8b;

u8a = u8b + u8c;

u8c--;

```

The following are all compliant because the only side effect in each expression is caused by the increment or decrement operator:

```c

x++;

a[i]++;

b.x++;

c->x++;

++(\*p);

\*p++;

(\*p)++;

```

The following are all non-compliant because they contain a function call as well as an increment or decrement operator:

```c

if ((f() + --u8a) == 0u) {

}

g(u8b++);

```

The following are all non-compliant even though the sub-expression containing the increment or decrement operator or some other side effect is not evaluated:

```c

u8a = (1u == 1u) ? 0u : u8b++;

if (u8a++ == ((1u == 1u) ? 0u : f())) {

}

```

\*\*See also:\*\* Rule 13.2

### Rule 13.4: The result of an assignment operator should not be used

\*\*Category:\*\* Advisory

\*\*Analysis:\*\* Decidable, Single Translation Unit

\*\*Applies to:\*\* C90, C99

\*\*Amplification:\*\*

This rule applies even if the expression containing the assignment operator is not evaluated.

\*\*Rationale:\*\*

The use of assignment operators, simple or compound, in combination with other arithmetic operators is not recommended because:

- It can significantly impair the readability of the code;

- It introduces additional side effects into a statement making it more difficult to avoid the undefined behavior covered by Rule 13.2.

\*\*Example:\*\*

```c

x = y; /\* Compliant \*/

a[x] = a[x = y]; /\* Non-compliant - the value of x = y is used \*/

/\*

\* Non-compliant - value of bool\_var = false is used but

\* bool\_var == false was probably intended

\*/

if (bool\_var = false) {

}

/\* Non-compliant even though bool\_var = true isn't evaluated \*/

if ((0u == 0u) || (bool\_var = true)) {

}

/\* Non-compliant - value of x = f() is used \*/

if ((x = f()) != 0) {

}

/\* Non-compliant - value of b += c is used \*/

a[b += c] = a[b];

/\* Non-compliant - values of c = 0 and b = c = 0 are used \*/

a = b = c = 0;

```

\*\*See also:\*\* Rule 13.2

### Rule 13.5: The right-hand operand of a logical && or || operator shall not contain persistent side effects

\*\*Category:\*\* Required

\*\*Analysis:\*\* Undecidable, System

\*\*Applies to:\*\* C90, C99

\*\*Rationale:\*\*

The evaluation of the right-hand operand of the `&&` and `||` operators is conditional on the value of the left-hand operand. If the right-hand operand contains side effects, those side effects may or may not occur, which may be contrary to programmer expectations.

If evaluation of the right-hand operand would produce side effects that are not persistent at the point in the program where the expression occurs, then it does not matter whether the right-hand operand is evaluated or not. The term "persistent side effect" is defined in Appendix J.

\*\*Example:\*\*

```c

uint16\_t f(uint16\_t y) {

/\* These side effects are not persistent as seen by the caller \*/

uint16\_t temp = y;

temp = y + 0x8080U;

return temp;

}

```

In this example, the function `f` modifies `temp` in a way that is not visible outside the function, so these side effects are not considered persistent.

Here is a non-compliant example:

```c

uint16\_t i = 0;

if ( (i != 0) && (i++ == 1) ) { /\* Non-compliant - i++ has a persistent side effect \*/

/\* ... \*/

}

```

In the non-compliant example, `i++` has a persistent side effect that may or may not be executed depending on the evaluation of `i != 0`.

\*\*See also:\*\* Rule 13.2

### Rule 13.6: The operand of the sizeof operator shall not contain any expression which has potential side effects

\*\*Category:\*\* Mandatory

\*\*Analysis:\*\* Decidable, Single Translation Unit

\*\*Applies to:\*\* C90, C99

\*\*Amplification:\*\*

Any expressions appearing in the operand of a sizeof operator are not normally evaluated. This rule mandates that the evaluation of any such expression shall not contain side effects, whether or not it is actually evaluated. A function call is considered to be a side effect for the purposes of this rule.

\*\*Rationale:\*\*

The operand of a sizeof operator may be either an expression or may specify a type. If the operand contains an expression, a possible programming error is to expect that expression to be evaluated when it is actually not evaluated in most circumstances.

The C90 standard states that expressions appearing in the operand are not evaluated at run-time. In C99, expressions appearing in the operand are usually not evaluated at run-time. However, if the operand contains a variable-length array type, then the array size expression will be evaluated if necessary. If the result can be determined without evaluating the array size expression, then it is unspecified whether it is evaluated or not.

\*\*Exception:\*\*

An expression of the form `sizeof(V)`, where `V` is an lvalue with a volatile-qualified type that is not a variable-length array, is permitted.

\*\*Example:\*\*

```c

volatile int32\_t i;

int32\_t j;

size\_t s;

s = sizeof(j); /\* Compliant \*/

s = sizeof(j++); /\* Non-compliant \*/

s = sizeof(i); /\* Compliant - exception \*/

s = sizeof(int32\_t); /\* Compliant \*/

volatile uint32\_t v;

void f(int32\_t n) {

size\_t s;

s = sizeof(int32\_t[n]); /\* Compliant \*/

s = sizeof(int32\_t[n++]); /\* Non-compliant \*/

s = sizeof(void (\*[n])(int32\_t a[v])); /\* Non-compliant \*/

}

```

\*\*See also:\*\* Rule 18.8

### Rule 13.5: The right-hand operand of a logical && or || operator shall not contain persistent side effects

\*\*Category:\*\* Required

\*\*Analysis:\*\* Undecidable, System

\*\*Applies to:\*\* C90, C99

\*\*Rationale:\*\*

The evaluation of the right-hand operand of the `&&` and `||` operators is conditional on the value of the left-hand operand. If the right-hand operand contains side effects, those side effects may or may not occur, which may be contrary to programmer expectations.

If evaluation of the right-hand operand would produce side effects that are not persistent at the point in the program where the expression occurs, then it does not matter whether the right-hand operand is evaluated or not. The term "persistent side effect" is defined in Appendix J.

\*\*Example:\*\*

```c

uint16\_t h(uint16\_t y) {

static uint16\_t temp = 0;

/\* This side effect is persistent \*/

temp = y + temp;

return temp;

}

void g(void) {

/\* Compliant - f() has no persistent side effects \*/

if (ishigh && (a == f(x))) {

}

/\* Non-compliant - h() has a persistent side effect \*/

if (ishigh && (a == h(x))) {

}

}

volatile uint16\_t v;

uint16\_t x;

/\* Non-compliant - access to volatile v is persistent \*/

if ((x == 0u) || (v == 1u)) {

}

/\* Non-compliant if fp points to a function with persistent side effects \*/

(fp != NULL) && (\*fp)(0);

```

### Rule 13.4: The result of an assignment operator should not be used

\*\*Category:\*\* Advisory

\*\*Analysis:\*\* Decidable, Single Translation Unit

\*\*Applies to:\*\* C90, C99

\*\*Amplification:\*\*

This rule applies even if the expression containing the assignment operator is not evaluated.

\*\*Rationale:\*\*

The use of assignment operators, simple or compound, in combination with other arithmetic operators is not recommended because:

- It can significantly impair the readability of the code;

- It introduces additional side effects into a statement making it more difficult to avoid the undefined behavior covered by Rule 13.2.

\*\*Example:\*\*

```c

x = y; /\* Compliant \*/

a[x] = a[x = y]; /\* Non-compliant - the value of x = y is used \*/

/\*

\* Non-compliant - value of bool\_var = false is used but

\* bool\_var == false was probably intended

\*/

if (bool\_var = false) {

}

/\* Non-compliant even though bool\_var = true isn't evaluated \*/

if ((0u == 0u) || (bool\_var = true)) {

}

/\* Non-compliant - value of x = f() is used \*/

if ((x = f()) != 0) {

}

/\* Non-compliant - value of b += c is used \*/

a[b += c] = a[b];

/\* Non-compliant - values of c = 0 and b = c = 0 are used \*/

a = b = c = 0;

```

\*\*See also:\*\* Rule 13.2

### Rule 13.3: A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator

\*\*Category:\*\* Advisory

\*\*Analysis:\*\* Decidable, Single Translation Unit

\*\*Applies to:\*\* C90, C99

\*\*Amplification:\*\*

A function call is considered to be a side effect for the purposes of this rule. All sub-expressions of the full expression are treated as if they were evaluated for the purposes of this rule, even if specified as not being evaluated by The Standard.

\*\*Rationale:\*\*

The use of increment and decrement operators in combination with other operators is not recommended because:

- It can significantly impair the readability of the code;

- It introduces additional side effects into a statement with the potential for undefined behavior (covered by Rule 13.2).

It is clearer to use these operations in isolation from any other operators.

\*\*Example:\*\*

The expression:

```c

u8a = u8b++;

```

is non-compliant. The non-compliant expression statement:

```c

u8a = ++u8b + u8c--;

```

is clearer when written as the following sequence:

```c

++u8b;

u8a = u8b + u8c;

u8c--;

```

The following are all compliant because the only side effect in each expression is caused by the increment or decrement operator:

```c

x++;

a[i]++;

b.x++;

c->x++;

++(\*p);

\*p++;

(\*p)++;

```

The following are all non-compliant because they contain a function call as well as an increment or decrement operator:

```c

if ((f() + --u8a) == 0u) {

}

g(u8b++);

```

The following are all non-compliant even though the sub-expression containing the increment or decrement operator or some other side effect is not evaluated:

```c

u8a = (1u == 1u) ? 0u : u8b++;

if (u8a++ == ((1u == 1u) ? 0u : f())) {

}

```

\*\*See also:\*\* Rule 13.2

### Rule 14.1: A loop counter shall not have essentially floating type

\*\*Category:\*\* Required

\*\*Analysis:\*\* Undecidable, System

\*\*Applies to:\*\* C90, C99

\*\*Rationale:\*\*

When using a floating-point loop counter, accumulation of rounding errors may result in a mismatch between the expected and actual number of iterations. This can happen when a loop step that is not a power of the floating-point radix is rounded to a value that can be represented.

Even if a loop with a floating-point loop counter appears to behave correctly on one implementation, it may give a different number of iterations on another implementation.

\*\*Example:\*\*

In the following non-compliant example, the value of `counter` is unlikely to be 1000 at the end of the loop.

```c

uint32\_t counter = 0u;

for (float32\_t f = 0.0f; f < 1.0f; f += 0.001f) {

++counter;

}

```

The following compliant example uses an integer loop counter to guarantee 1000 iterations and uses it to generate `f` for use within the loop.

```c

float32\_t f;

for (uint32\_t counter = 0u; counter < 1000u; ++counter) {

f = (float32\_t)counter \* 0.001f;

}

```

The following while loop is non-compliant because `f` is being used as a loop counter.

```c

float32\_t f = 0.0f;

while (f < 1.0f) {

f += 0.001f;

}

```

The following while loop is compliant because `f` is not being used as a loop counter.

```c

float32\_t f;

uint32\_t u32a;

f = read\_float32();

do {

u32a = read\_u32();

/\* f does not change in the loop so cannot be a loop counter \*/

} while (((float32\_t)u32a - f) > 10.0f);

```

\*\*See also:\*\* Rule 14.2

---

### Rule 14.2: A for loop shall be well-formed

\*\*Category:\*\* Required

\*\*Analysis:\*\* Undecidable, System

\*\*Applies to:\*\* C90, C99

\*\*Amplification:\*\*

The three clauses of a for statement are the:

1. \*\*First clause\*\* which

- Shall be empty, or

- Shall assign a value to the loop counter, or

- Shall define and initialize the loop counter (C99).

2. \*\*Second clause\*\* which

- Shall be an expression that has no persistent side effects, and

- Shall use the loop counter and optionally loop control flags, and

- Shall not use any other object that is modified in the for loop body.

3. \*\*Third clause\*\* which

- Shall be an expression whose only persistent side effect is to modify the value of the loop counter, and

- Shall not use objects that are modified in the for loop body.

There shall only be one loop counter in a for loop, which shall not be modified in the for loop body. A loop control flag is defined as a single identifier denoting an object with essentially Boolean type that is used in the second clause.

The behavior of a for loop body includes the behavior of any functions called within that statement.

\*\*Rationale:\*\*

The for statement provides a general-purpose looping facility. Using a restricted form of loop makes code easier to review and to analyze.

\*\*Exception:\*\*

All three clauses may be empty, for example `for (;;)`, so as to allow for infinite loops.

\*\*Example:\*\*

In the following C99 example, `i` is the loop counter and `flag` is a loop control flag.

```c

bool\_t flag = false;

for (int16\_t i = 0; (i < 5) && !flag; i++) {

if (C) {

flag = true; /\* Compliant - allows early termination of loop \*/

}

i = i + 3; /\* Non-compliant - altering the loop counter \*/

}

```

\*\*See also:\*\* Rule 14.1, Rule 14.3, Rule 14.4

---

### Rule 14.3: Controlling expressions shall not be invariant

\*\*Category:\*\* Required

\*\*Analysis:\*\* Undecidable, System

\*\*Applies to:\*\* C90, C99

\*\*Amplification:\*\*

This rule applies to:

- Controlling expressions of `if`, `while`, `for`, `do ... while`, and `switch` statements;

- The first operand of the `?:` operator.

\*\*Rationale:\*\*

If a controlling expression has an invariant value, it is possible that there is a programming error. Any code that cannot be reached due to the presence of an invariant expression may be removed by the compiler. This might have the effect of removing defensive code, for instance, from the executable.

\*\*Exception:\*\*

1. Invariants that are used to create infinite loops are permitted.

2. A `do ... while` loop with an essentially Boolean controlling expression that evaluates to 0 is permitted.

\*\*Example:\*\*

```c

s8a = (u16a < 0u) ? 0 : 1; /\* Non-compliant - u16a always >= 0 \*/

if (u16a <= 0xffffu) {

/\* Non-compliant - always true \*/

}

if (2 > 3) {

/\* Non-compliant - always false \*/

}

for (s8a = 0; s8a < 130; ++s8a) {

/\* Non-compliant - always true \*/

}

if ((s8a < 10) && (s8a > 20)) {

/\* Non-compliant - always false \*/

}

if ((s8a < 10) || (s8a > 5)) {

/\* Non-compliant - always true \*/

}

while (s8a > 10) {

if (s8a > 5) {

/\* Non-compliant - s8a not volatile \*/

}

}

while (true) {

/\* Compliant by exception 1 \*/

}

do {

/\* Compliant by exception 2 \*/

} while (0u == 1u);

const uint8\_t numcyl = 4u;

/\*

\* Non-compliant - compiler is permitted to assume that numcyl always

\* has value 4

\*/

if (numcyl == 4u) {

}

const volatile uint8\_t numcyl\_cal = 4u;

/\*

\* Compliant - compiler assumes numcyl\_cal may be changed by

\* an external method, e.g., automotive calibration tool, even

\* though the program cannot modify its value

\*/

if (numcyl\_cal == 4u) {

}

```

### Rule 14.4: The controlling expression of an `if` statement and the controlling expression of an iteration statement shall have essentially Boolean type

\*\*Category:\*\* Required

\*\*Analysis:\*\* Decidable, Single Translation Unit

\*\*Applies to:\*\* C90, C99

\*\*Amplification:\*\*

The controlling expression of a `for` statement is optional. The rule does not require the expression to be present but does require it to have essentially Boolean type if it is present.

\*\*Rationale:\*\*

Strong typing requires the controlling expression of an `if` statement or iteration-statement to have essentially Boolean type.

\*\*Example:\*\*

```c

int32\_t \*p, \*q;

while (p) /\* Non-compliant - p is a pointer \*/

{

}

while (q != NULL) /\* Compliant \*/

{

}

while (true) /\* Compliant \*/

{

}

extern bool\_t flag;

while (flag) /\* Compliant \*/

{

}

int32\_t i;

if (i) /\* Non-compliant \*/

{

}

if (i != 0) /\* Compliant \*/

{

}

```

\*\*See also:\*\* Rule 14.2, Rule 20.8

---

### Rule 15.1: The `goto` statement should not be used

\*\*Category:\*\* Advisory

\*\*Analysis:\*\* Decidable, Single Translation Unit

\*\*Applies to:\*\* C90, C99

\*\*Rationale:\*\*

Unconstrained use of `goto` can lead to programs that are unstructured and extremely difficult to understand.

In some cases, a total ban on `goto` requires the introduction of flags to ensure correct control flow, and it is possible that these flags may themselves be less transparent than the `goto` they replace. Therefore, if this rule is not followed, the restricted use of `goto` is allowed where that use follows the guidance in Rule 15.2 and Rule 15.3.

\*\*See also:\*\* Rule 9.1, Rule 15.2, Rule 15.3, Rule 15.4

---

### Rule 15.2: The `goto` statement shall jump to a label declared later in the same function

\*\*Category:\*\* Required

\*\*Analysis:\*\* Decidable, Single Translation Unit

\*\*Applies to:\*\* C90, C99

\*\*Rationale:\*\*

Unconstrained use of `goto` can lead to programs that are unstructured and extremely difficult to understand. Restricting the use of `goto` so that “back” jumps are prohibited ensures that iteration only occurs if the iteration statements provided by the language are used, helping to minimize visual code complexity.

\*\*Example:\*\*

```c

void f(void) {

int32\_t j = 0;

L1:

++j;

if (10 == j) {

goto L2; /\* Compliant \*/

}

goto L1; /\* Non-compliant \*/

L2:

++j;

}

```

\*\*See also:\*\* Rule 15.1, Rule 15.3, Rule 15.4

---

### Rule 15.3: Any label referenced by a `goto` statement shall be declared in the same block, or in any block enclosing the `goto` statement

\*\*Category:\*\* Required

\*\*Analysis:\*\* Decidable, Single Translation Unit

\*\*Applies to:\*\* C90, C99

\*\*Amplification:\*\*

For the purposes of this rule, a `switch`-clause that does not consist of a compound statement is treated as if it were a block.

\*\*Rationale:\*\*

Unconstrained use of `goto` can lead to programs that are unstructured and extremely difficult to understand. Preventing jumps between blocks, or into nested blocks, helps to minimize visual code complexity.

\*\*Example:\*\*

```c

void example(void) {

int32\_t i = 0;

{

int32\_t j = 0;

label\_1:

j++;

if (j < 10) {

goto label\_1; /\* Compliant \*/

}

}

goto label\_2; /\* Non-compliant - jumps to a label in an inner block \*/

label\_2:

i++;

}

```

\*\*Note:\*\* C99 is more restrictive when variably modified types are used. An attempt to make a jump from outside the scope of an identifier with a variably modified type into such a scope results in a constraint violation.

\*\*See also:\*\* Rule 15.1, Rule 15.2, Rule 15.4