PRE00-C. Prefer inline or static functions to function-like macros

Macros are dangerous because their use resembles that of real functions, but they have different semantics. The inline function-specifier was introduced to the C programming language in the C99 standard. Inline functions should be preferred over macros when they can be used interchangeably. Making a function an inline function suggests that calls to the function be as fast as possible by using, for example, an alternative to the usual function call mechanism, such as *inline substitution*. (See also PRE31-C. Avoid side effects in arguments to unsafe macros, PRE01-C. Use parentheses within macros around parameter names, and PRE02-C. Macro replacement lists should be parenthesized.)

Inline substitution is not textual substitution, nor does it create a new function. For example, the expansion of a macro used within the body of the function uses the definition it had at the point the function body appeared, not where the function is called; and identifiers refer to the declarations in scope where the body occurs.

Arguably, a decision to inline a function is a low-level optimization detail that the compiler should make without programmer input. The use of inline functions should be evaluated on the basis of (a) how well they are supported by targeted compilers, (b) what (if any) impact they have on the performance characteristics of your system, and (c) portability concerns. Static functions are often as good as inline functions and are supported in C.

Noncompliant Code Example

In this noncompliant code example, the macro `CUBE()` has undefined behavior when passed an expression that contains side effects:

```
#define CUBE(X) ((X) * (X) * (X))

void func(void) {
  int i = 2;
  int a = 81 / CUBE(++i);
  /* ... */
}
```

For this example, the initialization for a expands to

```
int a = 81 / ((++i) * (++i) * (++i));
```

which is undefined (see EXP30-C. Do not depend on the order of evaluation for side effects).

Compliant Solution

When the macro definition is replaced by an inline function, the side effect is executed only once before the function is called:

```
inline int cube(int i) {
  return i * i * i;
}

void func(void) {
  int i = 2;
  int a = 81 / cube(++i);
  /* ... */
}
```

Noncompliant Code Example

In this noncompliant code example, the programmer has written a macro called `EXEC_BUMP()` to call a specified function and increment a global counter [Dewhurst 2002]. When the expansion of a macro is used within the body of a function, as in this example, identifiers refer to the declarations in scope where the body occurs. As a result, when the macro is called in the `aFunc()` function, it inadvertently increments a local counter with the same name as the global variable. Note that this example also violates DCL01-C. Do not reuse variable names in subscopes.

```
size_t count = 0;

#define EXEC_BUMP(func) (func(), ++count)

void g(void) {
  printf("Called g, count = %zu.\n", count);
}

void aFunc(void) {
  size_t count = 0;
  while (count++ < 10) {
    EXEC_BUMP(g);
  }
}
```

The result is that invoking `aFunc()` (incorrectly) prints out the following line five times:

```
Called g, count = 0.
```

Compliant Solution

In this compliant solution, the `EXEC_BUMP()` macro is replaced by the inline function `exec_bump()`. Invoking `aFunc()` now (correctly) prints the value of `count` ranging from 0 to 9:

```
size_t count = 0;

void g(void) {
  printf("Called g, count = %zu.\n", count);
}

typedef void (*exec_func)(void);
inline void exec_bump(exec_func f) {
  f();
  ++count;
}

void aFunc(void) {
  size_t count = 0;
  while (count++ < 10) {
    exec_bump(g);
  }
}
```

The use of the inline function binds the identifier `count` to the global variable when the function body is compiled. The name cannot be re-bound to a different variable (with the same name) when the function is called.

Noncompliant Code Example

Unlike functions, the execution of macros can interleave. Consequently, two macros that are harmless in isolation can cause undefined behavior when combined in the same expression. In this example, `F()` and `G()` both increment the global variable `operations`, which causes problems when the two macros are used together:

```
int operations = 0, calls_to_F = 0, calls_to_G = 0;

#define F(x) (++operations, ++calls_to_F, 2 * x)
#define G(x) (++operations, ++calls_to_G, x + 1)

void func(int x) {
  int y = F(x) + G(x);

}
```

The variable `operations` is both read and modified twice in the same expression, so it can receive the wrong value if, for example, the following ordering occurs:

```
read operations into register 0
read operations into register 1
increment register 0
increment register 1
store register 0 into operations
store register 1 into operations
```

This noncompliant code example also violates EXP30-C. Do not depend on the order of evaluation for side effects.

Compliant Solution

The execution of functions, including inline functions, cannot be interleaved, so problematic orderings are not possible:

```
int operations = 0, calls_to_F = 0, calls_to_G = 0;

inline int f(int x) {
  ++operations;
  ++calls_to_F;
  return 2 * x;
}

inline int g(int x) {
  ++operations;
  ++calls_to_G;
  return x + 1;
}


void func(int x) {
  int y = f(x) + g(x);

}
```

Platform-Specific Details

GNU C (and some other compilers) supported inline functions before they were added to the C Standard and, as a result, have significantly different semantics. Richard Kettlewell provides a good explanation of differences between the C99 and GNU C rules [Kettlewell 2003].

Exceptions

PRE00-C-EX1: Macros can be used to implement *local functions* (repetitive blocks of code that have access to automatic variables from the enclosing scope) that cannot be achieved with inline functions.

PRE00-C-EX2: Macros can be used for concatenating tokens or performing stringification. For example,

```
enum Color { Color_Red, Color_Green, Color_Blue };
static const struct {
  enum Color  color;
  const char *name;
} colors[] = {
#define COLOR(color)   { Color_ ## color, #color }
  COLOR(Red), COLOR(Green), COLOR(Blue)
};
```

calculates only one of the two expressions depending on the selector's value. See PRE05-C. Understand macro replacement when concatenating tokens or performing stringification for more information.

PRE00-C-EX3: Macros can be used to yield a compile-time constant. This is not always possible using inline functions, as shown by the following example:

```
#define ADD_M(a, b) ((a) + (b))
static inline int add_f(int a, int b) {
  return a + b;
}
```

In this example, the `ADD_M(3,4)` macro invocation yields a constant expression, but the `add_f(3,4)` function invocation does not.

PRE00-C-EX4: Macros can be used to implement type-generic functions that cannot be implemented in the C language without the aid of a mechanism such as C++ templates.

An example of the use of function-like macros to create type-generic functions is shown in MEM02-C. Immediately cast the result of a memory allocation function call into a pointer to the allocated type.

Type-generic macros may also be used, for example, to swap two variables of any type, provided they are of the same type.

PRE00-C-EX5: Macro parameters exhibit call-by-name semantics, whereas functions are call by value. Macros must be used in cases where call-by-name semantics are required.

Risk Assessment

Improper use of macros may result in undefined behavior.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
PRE00-C	Medium	Unlikely	Medium	**P4**	**L3**

Automated Detection

Tool	Version	Checker	Description
Astrée	19.04	**macro-function-like**	Fully checked
Axivion Bauhaus Suite	6.9.0	**CertC-PRE00**	
ECLAIR	1.2	**CC2.PRE00**	Fully implemented
Klocwork	2018	**MISRA.DEFINE.FUNC**	
LDRA tool suite	9.7.1	**340 S**	Enhanced enforcement
Parasoft C/C++test	10.4.2	**CERT_C-PRE00-a**	A function should be used in preference to a function-like macro
Polyspace Bug Finder	R2019b	CERT C: Rec. PRE00-C	Checks for use of function-like macro instead of function (rec. fully covered)
PRQA QA-C	9.7	**3453**	Fully implemented
RuleChecker	19.04	**macro-function-like**	Fully checked
SonarQube C/C++ Plugin	3.11	**S960**	

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Related Guidelines

SEI CERT C++ Coding Standard	VOID PRE00-CPP. Avoid defining macros
ISO/IEC TR 24772:2013	Pre-processor Directives [NMP]
MISRA C:2012	Directive 4.9 (advisory)

Bibliography

[Dewhurst 2002]	Gotcha #26, "`#define` Pseudofunctions"
[FSF 2005]	Section 5.34, "An Inline Function Is as Fast as a Macro"
[Kettlewell 2003]	

[Summit 2005]	Question 10.4