

# C static code analysis: Printf-style format strings should not lead to unexpected behavior at runtime

2 minutes

---

Because `printf` format strings are interpreted at runtime, rather than validated by the compiler, they can contain errors that lead to unexpected behavior or runtime errors. This rule statically validates the good behavior of `printf` formats.

The related rule {rule:cpp:S3457} is about errors that produce an unexpected string, while this rule is about errors that will create undefined behavior.

## Noncompliant Code Example

```
printf("%d", 1.2); // Noncompliant, an "int" is expected rather than a "double"
```

```
printf("%d %d", 1); // Noncompliant, the second argument is missing
```

```
printf("%0$d ", 1); // Noncompliant, arguments are numbered starting from 1
```

```
printf("%1$d %d", 1, 2); // Noncompliant, positional and non-positional arguments can not be mixed
```

```
printf("%*d", 1.1, 2); // Noncompliant, field width should be an integer
```

```
printf("ab\0cd"); // Noncompliant, format string contains null char
```

```
int x;  
printf("%+p", (void*)&x); // Noncompliant, flag "+" has undefined  
behavior with conversion specifier "p"  
printf("%vd", x); //Noncompliant, conversion specifier "v" is not  
valid
```

## Compliant Solution

```
printf("%f", 1.2); // Compliant, format is consistent with the  
corresponding argument  
printf("%d", 1); // Compliant, number of specifiers is consistent  
with number of arguments  
printf("%1$d ", 1); // Compliant, number of positional argument is  
consistent
```

## Exceptions

This rule will only work if the format string is provided as a string literal.

## See

- [CERT, FIO47-C.](#) - Use valid format strings