The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.8.6.4 The return statement

Constraints

return void type

A **return** statement with an expression shall not appear in a function whose return type is **void**. 1799

Commentary

base document

The base document did not define any semantics for this case (and simplifying automatic C source code generators was not considered to be sufficiently important for the committee to define any).

C++

6.6.3p3 *A return statement with an expression of type "cv **void**" can be used only in functions with a return type of cv void; the expression is evaluated just before the function returns to its caller.*

Source developed using a C++ translator may contain **return** statements with an expression returning a void type, which will cause a constraint violation if processed by a C translator.

```
1  void f(void)
2  {
3  return (void)4; /* constraint violation */
4                  // does not change the conformance status of a program
5  }
```

Other Languages

Many other languages support constructs called *procedure*s or *subroutine*s to take the role performed by function returning type **void**. The **return** statements appearing within these constructs are not allowed to include an expression.

return without expression

A **return** statement without an expression shall only appear in a function whose return type is **void**. 1800

Commentary

base document

The base document did not provide a mechanism enabling function declarations to explicitly specify that they did not return a value. It was common practice to omit the return type in the function definition to indicate that the function did not return a value. Unfortunately it was also common practice for developers to omit a return type and rely on the translator supplying an implicit **int** type. The type **void** was introduced into C by the C90 Standard.

type specifiers possible sets of

This constraint is new in C99 and ties in with the removal of support for *implicit int* in declarations.

function termination reaching }

The behavior that occurs after executing the last statement in a function definition, when it is not a **return** statement, is discussed elsewhere.

C90

This constraint is new in C99.

```
1  int f(void)
2  {
3  return; /* Not a constraint violation in C90. */
4  }
```

Other Languages

Many languages do not permit the expression in a **return** statement to be omitted when the containing function is defined to return a value.

Common Implementations

Some C90 implementations provided an option to switch on the diagnosing of additional constructs. The requirement specified in this constraint was often one of these additional constructs.

Coding Guidelines

This constraint is new in C99 and the majority of existing, C90, implementations do not issue a diagnostic for a violation of it. Any coding guideline documents that continue to be based on the C90 Standard (e.g., MISRA) might like to consider including a guideline recommendation along the lines of this constraint. The practice of not specifying an expression when it is known that the caller does not make use of the returned value also occurs in existing code, this issue is discussed elsewhere.

MISRA

function termination reaching }

Usage

The translated form of this book's benchmark programs contained 19 instances of a **return** statement without an expression appearing in a function whose return type was **void**.

Semantics

1801 A **return** statement terminates execution of the current function and returns control to its caller.

Commentary

The **return** statement is one of the two constructs that can be used to terminate execution of the current function (use of `longjmp` is relatively rare).

Other Languages

Some languages (e.g., Pascal) support nested functions and what are known as *non-local goto* statements. Within a body of a nested function it is possible to perform non-local goto to a label visible in the body of an outer function. Fortran supports what is known as an *alternative return*. This kind of return (it is only supported for subroutines, not functions) causes control to resume at some location other than the statement following the one that performed the call. The alternative locations at which execution resume are passed as arguments to the call.

Common Implementations

Many processors include a return instruction that is the inverse of the one used to perform a call. A return instruction has to reinstate the stack (the most common form used to implement function call and return) to the state it was in prior to the call. This is often simply a matter of resetting the stack pointer to its value prior to the call. The return address may be loaded into a register (and an indirect jump performed) or popped from the stack (many processors contain an instruction that pops an address off the stack and jumps to it).

Depending on the calling conventions used either the caller or callee will restore any registers whose contents were saved prior to the call. Implementations where the callee restores the registers are likely to translate a **return** statement into a branch to the end of the function,[1] reducing duplication of the common housekeeping code.

register function call housekeeping

It is unlikely that storage allocated to objects having variably modified types will be treated any differently than storage allocated to objects having other types (because, like them, it is allocated on the stack).

1802 A function may have any number of **return** statements.

Commentary

When the C language was first specified it was not uncommon for languages definitions to specify that a **return** statement ended the body of a function/procedure/subroutine.

C++

The C++ Standard does not explicitly specify this permission.

Other Languages

Some languages (e.g., Pascal) have no **return** statement. A function (or procedure/subroutine) returns when the flow of control reaches the end of its body.

Coding Guidelines

Some coding guideline documents specify that function definitions should contain a single **return** statement. Adhering to such a recommendation may involve using either a **goto** statement or additional flags (that prevent statements following the return decision point being executed). The cost of using these alternative constructs does not appear to be less than that associated with using multiple **return** statements.

iteration statement executed repeatedly

If a **return** statement with an expression is executed, the value of the expression is returned to the caller as the value of the function call expression. 1803

Commentary

C specifies that functions return values, not references. Implementations need to act as if temporary storage was used to hold the value being returned by a function. In an assignment statement this temporary object and the object being assigned to do not overlap.

footnote 136 1806

C++

The C++ Standard supports the use of expressions that do not return a value to the caller.

return void type 1799

Other Languages

In some languages (e.g., Fortran and Pascal) the last value assigned to an identifier, whose spelling is the same as that of the called function, is the result of a function call. Pascal does not support a **return** statement and execution of a function does not terminate until control reaches the end of that function. In Algol 68 the value returned by a function is the value of the last expression in its body (rather like the behavior of the `gcc` compound expression).

compound expression

Common Implementations

Values having scalar type are usually returned in a register. Those having a structure type are usually returned in an area of storage allocated by the caller (often passed as a hidden parameter to the called function).

---

return implicit cast

If the expression has a type different from the return type of the function in which it appears, the value is converted as if by assignment to an object having the return type of the function.[136)] 1804

Commentary

This situation can only occur if the expression has a scalar type. As the response to DR #094 pointed out, this *as if* assignment implies that the constraints given for the assignment operator also apply here.

assignment operator modifiable lvalue

C++

In the case of functions having a return type of *cv* **void** (6.6.3p3) the expression is not implicitly converted to that type. An explicit conversion is required.

Coding Guidelines

The guideline recommendations applicable here are the same as those that apply to any implicit conversions.

operand convert automatically

EXAMPLE In: 1805

```
struct s { double i; } f(void);
union {
        struct {
                int f1;
                struct s f2;
        } u1;
        struct {
                struct s f3;
                int f4;
        } u2;
} g;

struct s f(void)
```

```
{
        return g.u1.f2;
}

/* ... */
g.u2.f3 = f();
```

there is no undefined behavior, although there would be if the assignment were done directly (without using a function call to fetch the value).

Commentary

Neither is there any undefined behavior if the function `f` is defined as an inline function.

inline function

1806 136) The **return** statement is not an assignment.

footnote 136

Commentary

The difference between the **return** statement and the assignment operator is that in the former case the value is required to act as if it were held in temporary storage before it is assigned (rather like the difference between the `memcpy` and `memmove` library functions). In an assignment statement it is intended that implementations be able to perform the operation without the need of temporary storage. Possible differences in behavior can only occur when operating storage for the two operands overlaps.

C90

This footnote did not appear in the C90 Standard. It was added by the response to DR #001.

C++

This distinction also occurs in C++, but as a special case of a much larger issue involving the creation of temporary objects (for constructs not available in C).

A return statement can involve the construction and copy of a temporary object (12.2). 6.6.3p2

Temporaries of class type are created in various contexts: binding an rvalue to a reference (8.5.3), returning an rvalue (6.6.3), . . . 12.2p1

Common Implementations

Translators that inline function calls, returning a structure or union type, have to be careful about how they handle **return** statements. Mapping them to an assignment may not produce the same affect as a function call.

inline suggests fast calls

1807 The overlap restriction of subclause 6.5.16.1 does not apply to the case of function return.

Commentary

This is effectively a requirement on implementations to ensure that the behavior is well defined (not undefined as specified elsewhere).

assignment value overlaps object

1808 The representation of floating-point values may have wider range or precision and is determined by `FLT_EVAL_METHOD`.

Commentary

This sentence calls out behavior that is specified elsewhere as applying to the expression in a **return** statement.

This sentence was added by the response to DR #290.

floating operands evaluation format

1809 A cast may be used to remove this extra range and precision.

Commentary

return 1804
implicit cast

An implicit conversion is only performed on the value in a **return** statement if the type of this value is different from that of the return type.

This sentence was added by the response to DR #290.

References

1. J. W. Davidson, J. R. Rabung, and D. B. Whalley. Relating static and dynamic machine code measurements. Technical Report CS-89-03, Department of Computer Science, University of Virginia, July 13 1989.