The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.3.2.2 void

void expression

The (nonexistent) value of a *void expression* (an expression that has type **void**) shall not be used in any way, and implicit or explicit conversions (except to **void**) shall not be applied to such an expression. 740

Commentary

This defines the term *void expression*. It makes no sense to take an expression having a type that cannot represent any values and cast it to a type that can represent a range of values. As specified here, these requirements do not require a diagnostic to be issued. However, constraint requirements in the definition of the cast operator do require a translator to issue a diagnostic, if a violation occurs.

cast scalar or void type

C++

3.9.1p9 *An expression of type **void** shall be used only as an expression statement (6.2), as an operand of a comma expression (5.18), as a second or third operand of **?:** (5.16), as the operand of **typeid**, or as the expression in a return statement (6.6.3) for a function with the return type **void**.*

The C++ Standard explicitly enumerates the contexts in which a void expression can appear. The effect is to disallow the value of a void expression being used, or explicitly converted, as per the C wording.

The C++ Standard explicitly permits the use of a void expression in a context that is not supported in C:

```
extern void g(void);

void f(void)
{
return g(); /* Constraint violation. */
}
```

5.2.9p4 *Any expression can be explicitly converted to type "cv **void**."*

Thus, C++ supports the **void** casts allowed in C.

Other Languages

In many languages the assignment token is not an operator; it is a token that appears in an assignment statement. Casting away the result of an assignment is not possible because it does not have one. Also there is often a requirement that functions returning a value (those that don't return a value are often called procedures or subroutines) must have that value used.

Coding Guidelines

It is possible for a program to accidentally make use of a *void value*; for instance, if a function is defined with a **void** return type, but at the point of call in a different translation unit, it is declared to return a scalar type. The behavior in this case is undefined because of the mismatch between definition and declaration at the point of call. If the guideline recommendations on having a single point of declaration is followed this situation will not occur.

function call not compatible with definition

identifier ?? declared in one file

If an expression of any other type is evaluated as a void expression, its value or designator is discarded. 741

Commentary

A void expression can occur in an expression statement and as the left operand of the comma operator. According to the definition of a void expression, it has type **void**. In the example below the expression statement x = y has type **int**, not **void**. But in this context its value is still discarded. In the following:

void expression 740

```
int x, y;

void f(void)
{
x = y;
(void)(x = y);

x = (y++, y+1);
x = ((void)y++, y+1);
}
```

all of the statements are expressions, whose value is discarded after the assignment to x.

C90

If an expression of any other type occurs in a context where a void expression is required, its value or designator is discarded.

The wording in the C90 Standard begs the question, "When is a void expression required"?

C++

There are a number of contexts in which an expression is evaluated as a void expression in C. In two of these cases the C++ Standard specifies that lvalue-to-rvalue conversions are not applied: Clauses 5.18p1 left operand of the comma operator, and 6.2p1 the expression in an expression statement. The other context is an explicit cast:

*Any expression can be explicitly converted to type "`cv` **`void`**." The expression value is discarded.* 5.2.9p4

So in C++ there is no value to discard in these contexts. No other standards wording is required.

Other Languages

Most languages require that the value of an expression be used in some way, but then such languages do not usually regard assignment as an operator (which returns a value) or have the equivalent of the comma operator. In many cases the syntax does not support the use of an expression in a context where its value would be discarded.

Common Implementations

For processors that use registers to hold intermediate results during expression evaluation, it is a simple matter of ignoring register contents when there is no further use for its contents. For stack-based architectures the value being discarded has to be explicitly popped from the evaluation stack at some point. (As an optimization, some implementations only reset the evaluation stack when a backward jump or function **`return`** statement is encountered.)

Coding Guidelines

expression statement evaluated as void expression
comma operator left operand

All expression statements return a value that is not subsequently used, as does the left operand of the comma operator. However, all of the information needed for readers to evaluate the usage is visible at the point the statement or operator appears in the source.

In the case of a function returning a value that is not used, the intent may not be immediately visible to readers. Perhaps the function had not originally returned a value and a later modification changed this behavior. One way of indicating, to readers, that the return value is being intentionally ignored, is to use an explicit cast (the exception for library functions is given because the specification of these functions' return type rarely changes).

Cg 741.1

If a function, that does not belong to a standard library, returning a non-void type is evaluated as a void expression its result shall be explicitly cast to **`void`**.

Example

```
extern int f(void);

void g(void)
{
(void)f();
}
```

(A void expression is evaluated for its side effects.) 742

Commentary

redun-dant code

A void expression is not required to have any side effects. The issue of redundant code is discussed elsewhere.

C++

This observation is not made in the C++ Standard.

Example

In the following:

```
int i,
    j;

extern int f(void);

void g(void)
{
i+1;      /* expression 1 */
j=4;      /* expression 2 */
4+(i=3); /* expression 3 */
(void)f();
}
```

execution of *expression 1* will not change the state of the abstract machine. It contains no side effects. Execution of *expression 2* will change the state of the abstract machine, but it may not have any effect on subsequent calculations; or, if it does, those calculations may not have any effect on the output of the program, so is a redundant side effect. *expression 3* contains a side effect, but in a nested subexpression. The root operator in *expression 3* does not contain a side effect.

References