The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.3.2.1 Lvalues, arrays, and function designators

lvalue

An *lvalue* is an expression with an object type or an incomplete type other than **void**;[53)] 721

Commentary

This defines the term *lvalue*, pronounced *ell-value*. It is an important concept and this term is used by the more sophisticated developers. An lvalue can have its address taken.

The difference between evaluating, for instance, the expression `a[i]` as a value and as an lvalue is that in the former case the result is the value, while in the latter it is the address of the i'th element of the array `a`.

```
From: Dennis Ritchie
Newsgroups: comp.std.c
Subject: Re: C99 change of lvalue definition
Date: Sat, 27 Oct 2001 03:32:09 +0000
Organization: Bell Labs / Lucent Technologies
 . . .
> Part of the problem is that the term "lvalue" is being used for two
> different notions, one syntactic (a certain class of expressions) and
> the other semantic (some abstraction of the notion of "address").  The
> first step would be to remove this ambiguity by selecting distinct
> terms for these two notions, e.g., "lvalue-expression" for the
> syntactic notion and "lvalue" for the semantic notion.
>
> The syntactic notion could be defined by suitably amending the grammar
> appendix so that lvalue-expression is a non-terminal whose expansions
> are exactly the expressions that can appear on the left side of an
> assignment without causing a compile-time error.
>
  . . .
> Tom Payne
This is true.  To clean up the history a bit: The 1967 BCPL manual I
have uses the words "lvalue" and "rvalue" liberally; my copy does not
have a consolidated grammar, but the section about Simple Assignment
Commands describes the syntactical possibilities for things on the
left of := , namely an identifier (not a manifest constant), a vector
application, or an rv expression (equivalent to C *, or later BCPL
!).  However, "lvalue" in the text does indeed seem to be a semantic
notion, namely a bit pattern that refers to a cell in the abstract
machine.
The Richards and Whitbey-Strevens book (my printing seems post-1985,
but it's a reprint of the 1980 edition) does not seem to use the
terms lvalue or rvalue.  On the other hand, it does make even more
explicit syntactically (in its grammar) what can appear on the LHS of
the := of an assignment-command, namely an <lhse> or left-hand-side
expression. This is the syntactic lvalue.
On yet another hand, it only indirectly and by annotation says that
the operand of the address-of operator is restricted.  Nevertheless,
the textual description seems identical to the consolidated grammar's
<lhse>.
In any event, my observation that K&R 1 used syntactic means as the
underlying basis for answering the question "what can go on the left
of = or as the operand of ++, &, ...?" seems true.  In C89 and C99
there are no syntactical restrictions; the many that exist are
semantic, for better or worse.
```

```
Dennis
```

C90

*An lvalue is an expression (with an object type or an incomplete type other than **void**) that designates an object.31)*

The C90 Standard required that an lvalue designate an object. An implication of this requirement was that some constraint requirements could only be enforced during program execution (e.g., the left operand of an assignment operator must be an lvalue). The Committee intended that constraint requirements be enforceable during translation.

assignment operator modifiable lvalue

Technically this is a change of behavior between C99 and C90. But since few implementations enforced this requirement during program execution, the difference is unlikely to be noticed.

C++

An lvalue refers to an object or function. 3.10p2

Incomplete types, other than **void**, are object types in C++, so all C lvalues are also C++ lvalues.

object types

The C++ support for a function lvalue involves the use of some syntax that is not supported in C.

As another example, the function 3.10p3

```
int& f();
```

yields an lvalue, so the call `f()` is an lvalue expression.

Other Languages

While not being generic to programming languages, the concept of lvalue is very commonly seen in the specification of other languages.

---

722 if an lvalue does not designate an object when it is evaluated, the behavior is undefined.

Commentary

This can occur if a dereferenced pointer does not refer to an object, or perhaps it refers to an object whose lifetime has ended.

lifetime of object

C90

In the C90 Standard the definition of the term *lvalue* required that it designate an object. An expression could not be an lvalue unless it designated an object.

C++

In C++ the behavior is not always undefined:

Similarly, before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any lvalue which refers to the original object may be used but only in limited ways. Such an lvalue refers to allocated storage (3.7.3.2), and using the properties of the lvalue which do not depend on its value is well-defined. 3.8p6

Coding Guidelines

A guideline recommending that an lvalue always denote an object when it is evaluated is equivalent to a guideline recommending that programs not contain defects.

Example

```
1  extern int *p;
2
3  void f(void)
4  {
5  p[3]=3; /* Does p designate an object? */
6  }
```

---

particular type

When an object is said to have a particular type, the type is specified by the lvalue used to designate the object. 723

Commentary

This is not a definition of the term *particular type*, which is only used in two other places in the standard. In the case of objects with static and automatic storage duration, it is usually the declared type of the object. For objects with allocated storage duration, almost any type can be used. The term *effective type* was introduced in C99 to provide a more precise specification of an object's type.

effective type

C++

The situation in C++ is rather more complex:

1.8p1 *The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An object is a region of storage. [Note: A function is not an object, regardless of whether or not it occupies storage in the way that objects do.] An* `object` *is created by a* `definition` *(3.1), by a* `new-expression` *(5.3.4) or by the implementation (12.2) when needed. The properties of an object are determined when the object is created. An object can have a* `name` *(clause 3). An object has a* `storage duration` *(3.7) which influences its* `lifetime` *(3.8). An object has a* `type` *(3.9). The term* `object type` *refers to the type with which the object is created. Some objects are* `polymorphic` *(10.3); the implementation generates information associated with each such object that makes it possible to determine that object's type during program execution. For other objects, the interpretation of the values found therein is determined by the type of the* `expressions` *(clause 5) used to access them.*

Example

```
1  extern int ei;
2  extern void *pv;
3
4  void f(void)
5  {
6  ei = 2;
7  *(char *)pv = 'x';
8  }
```

---

modifiable lvalue

A *modifiable lvalue* is an lvalue that does not have array type, does not have an incomplete type, does not have a const-qualified type, and if it is a structure or union, does not have any member (including, recursively, any member or element of all contained aggregates or unions) with a const-qualified type. 724

Commentary

This defines the term *modifiable lvalue*; which is not commonly used by developers. Since most lvalues are modifiable, the term *lvalue* tends to be commonly used to denote those that are modifiable. There are a variety of different terms used to describe lvalues that are not modifiable lvalues, usually involving the phrase *const-qualified*. As the name suggests, a modifiable lvalue can be modified during program execution. All of the types listed in the C sentence are types whose lvalues are not intended to be modified, or for which there is insufficient information available to be able to modify them (an incomplete type).

The result of a cast is not an lvalue, so it is not possible to cast away a const-qualification. However, it is possible to cast away constness via a pointer type: footnote 85

```
1   struct {
2           const int m1;
3           /*
4            * The const qualifier applies to the array
5            * element, not to the array a_mem.
6            */
7           const int a_mem[3];
8           } glob;
9
10  const int * cp = &glob.m1;
11
12  void f(void)
13  {
14  *(int *)cp=2;
15  }
```

The term member applies to structure and union types, while element refers to array types.

C++

The term *modifiable lvalue* is used by the C++ Standard, but understanding what this term might mean requires joining together the definitions of the terms *lvalue* and *modifiable*:

> *An lvalue for an object is necessary in order to modify the object except that an rvalue of class type can also be used to modify its referent under certain circumstances.* 3.10p10

> *If an expression can be used to modify the object to which it refers, the expression is called modifiable.* 3.10p14

There does not appear to be any mechanism for modifying objects having an incomplete type.

> *Objects of array types cannot be modified, see 3.10.* 8.3.4p5

This is a case where an object of a given type cannot be modified and follows the C requirement.

> *. . . ; a const-qualified access path cannot be used to modify an object even if the object referenced is a non-const object and can be modified through some other access path.* 7.1.5.1p3

The C++ wording is based on access paths rather than the C method of enumerating the various cases. However, the final effect is the same.

Other Languages

Many languages only provide a single way of modifying an object through assignment. In these cases a general term describing modifiability is not required; the requirements can all be specified under assignment. Languages that support operators that can modify the value of an object, other than by assignment, sometimes define a term that serves a purpose similar to modifiable lvalue.

lvalue converted to value

Except when it is the operand of the **sizeof** operator, the unary **&** operator, the **++** operator, the **--** operator, or the left operand of the **.** operator or an assignment operator, an lvalue that does not have array type is converted to the value stored in the designated object (and is no longer an lvalue).

Commentary

The exceptions called out here apply either because information on the storage for an object is used or are situations where there is a possibility for the lvalue to be modified (in the case of the **&** and **.** operators this may occur in a subsequent operation). In the case of an array type, in most contexts, a reference to an object having this type is converted to a pointer to its first element (and so is always an lvalue). This converted lvalue is sometimes called an *rvalue*.

array 729 converted to pointer

rvalue 736

C++

Quite a long chain of deduction is needed to show that this requirement also applies in C++. The C++ Standard uses the term *rvalue* to refer to the particular value that an lvalue is converted into.

3.10p7 *Whenever an lvalue appears in a context where an rvalue is expected, the lvalue is converted to an rvalue;*

5p8 *Whenever an lvalue expression appears as an operand of an operator that expects an rvalue for that operand, the lvalue-to-rvalue (4.1), . . . standard conversions are applied to convert the expression to an rvalue.*

The C wording specifies that lvalues are converted unless they occur in specified contexts. The C++ wording specifies that lvalues are converted in a context where an rvalue is expected. Enumerating the cases where C++ expects an rvalue we find:

5.3.4p4 *The lvalue-to-rvalue (4.1), . . . standard conversions are not applied to the operand of* ***sizeof****.*

What is the behavior for the unary **&** operator?

4p5 *There are some contexts where certain conversions are suppressed. For example, the lvalue-to-rvalue conversion is not done on the operand of the unary* **&** *operator.*

However, this is a *Note:* and has no normative status. There is no mention of any conversions in 5.3.1p2-5, which deals with the unary **&** operator.

In the case of the postfix **++** and **--** operators we have:

5.2.6p1 *The operand shall be a modifiable lvalue. . . . The result is an rvalue.*

In the case of the prefix **++** and **--** operators we have:

5.3.2p1 *The operand shall be a modifiable lvalue. . . . The value is the new value of the operand; it is an lvalue.*

So for the postfix case, there is an lvalue-to-rvalue conversion, although this is never explicitly stated and in the prefix case there is no conversion.

The C case is more restrictive than C++, which requires a conforming implementation to successfully translate:

```
1  extern int i;
2
3  void f(void)
4  {
5  ++i = 4; // Well-formed
6           /* Constraint violation */
7  }
```

For the left operand of the `.` operator we have:

If E1 is an lvalue, then E1.E2 is an lvalue. 5.2.5p4

The left operand is not converted to an rvalue. For the left operand of an assignment operator we have:

All require a modifiable lvalue as their left operand, . . . ; the result is an lvalue. 5.17p1

The left operand is not converted to an rvalue. And finally for the array type:

An lvalue (3.10) of a non-function, non-array type T can be converted to an rvalue. 4.1p1

An lvalue having an array type cannot be converted to an rvalue (i.e., the C++ Standard contains no other wording specifying that an array can be converted to an rvalue).

In two cases the C++ Standard specifies that lvalue-to-rvalue conversions are not applied: Clause 5.18p1 left operand of the comma operator and Clause 6.2p1 the expression in an expression statement. In C the values would be discarded in both of these cases, so there is no change in behavior. In the following cases C++ performs a lvalue-to-rvalue conversion (however, the language construct is not relevant to C): Clause 8.2.8p3 Type identification; 5.2.9p4 static cast; 8.5.3p5 References.

Other Languages

The value that the lvalue is converted into is sometimes called an *rvalue* in other languages.

Coding Guidelines

The distinction between lvalue and rvalue and the circumstances in which the former is converted into the latter is something that people learning to write software for the first time often have problems with. But once the underlying concepts are understood, developers know how to distinguish the two contexts. The confusion that developers often get themselves into with pointers is the *pointer* or the *pointed-to object* being accessed is a separate issue and is discussed under the indirection operator.

unary * indirection

Usage

Usage information on the number of translation time references, in the source code, is given elsewhere (see Figure **??**, Figure **??**).

---

726 If the lvalue has qualified type, the value has the unqualified version of the type of the lvalue;

lvalue value is unqualified

Commentary

Type qualification only applies to objects, not to values. A pointer value can refer to an object having a qualified type. In this case it is not the pointer value that is qualified.

```
1  extern int glob;
2  extern const int *p_1;
3  extern       int * const p_2 = &glob; /* The value of p_2 cannot be modified. */
4
5  void f(void)
6  {
7  const int loc = 2;
8
9  p_1 = &loc; /* The value of p_1 can be modified. */
10 }
```

C++

The value being referred to in C is what C++ calls an *rvalue*.

4.1p1

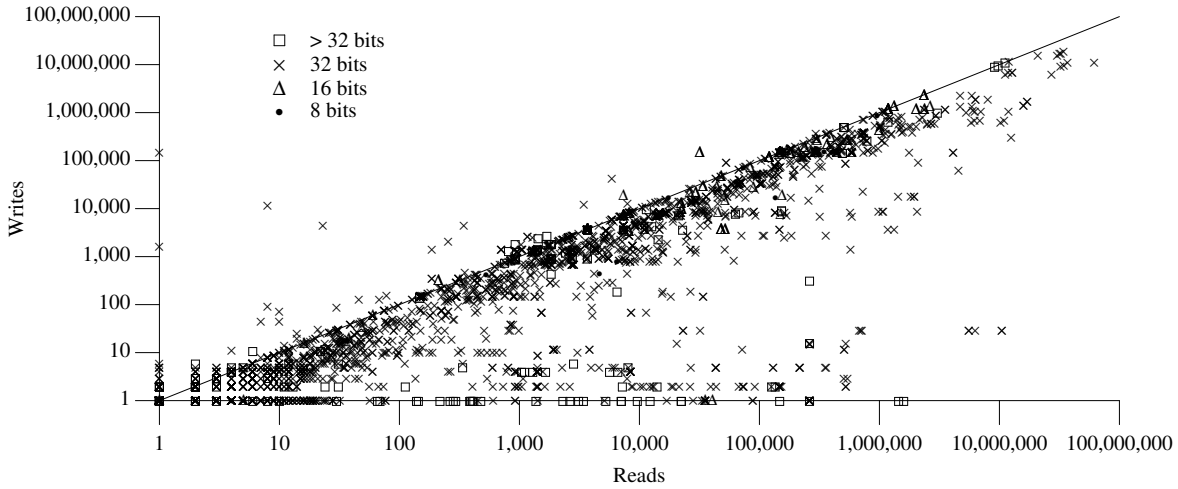

Figure 725.1: Execution-time counts of the number of reads and writes of the same object (declared in block or file scope, i.e., not allocated storage) for a subset of the MediaBench benchmarks; items above the diagonal indicate more writes than reads. Data kindly supplied by Caspi, based on his research.[1]

If T is a non-class type, the type of the rvalue is the cv-unqualified version of T. Otherwise, the type of the rvalue is T.[49)]

Footnote 49

49) In C++ class rvalues can have cv-qualified types (because they are objects). This differs from ISO C, in which non-lvalues never have cv-qualified types.

Class rvalues are objects having a structure or union type in C.

Other Languages

Some languages give string literals their equivalent of a **`const`** qualifier, thus making it possible for values to be qualified, as well as have a type.

---

otherwise, the value has the type of the lvalue. 727

Commentary

object representation
value representation
integer promotions

Accessing an lvalues value does not change its type, even if there are more bits in the object representation than in the value representation. Those values that have an integer type whose rank is less than that of **`int`** may be promoted to **`int`** or some other type as a result of being the operand of some operator. But values start off with the type of the lvalue from which they were obtained.

Other Languages

This statement is probably generic to all languages.

If the lvalue has an incomplete type and does not have array type, the behavior is undefined. 728

Commentary

In the case of array types the element type is known, which is the only information needed by a translator (to calculate the offset of the indexed element, given any index value).

footnote 109

As the standard points out elsewhere, an incomplete type may only be used when the size of an object of that type is not needed.

C++

An lvalue (3.10) of a non-function, non-array type T can be converted to an rvalue. If T is an incomplete type, a program that necessitates this conversion is ill-formed. 4.1p1

An lvalue or rvalue of type "array of N T" or "array of unknown bound of T" can be converted to an rvalue of type "pointer to T." 4.2p1

The C behavior in this case is undefined; in C++ the conversion is ill-formed and a diagnostic is required.

Common Implementations

It is very likely that an occurrence of this undefined behavior will be diagnosed by an implementation.

Coding Guidelines

It may surprise developers to find this case is not a constraint violation, but undefined behavior. While the standard specifies that the behavior is undefined, in this case, it is difficult to see how a translator could do anything other than issue a diagnostic and fail to translate (there are no meaningful semantics to give to such a construct). This is not to say that a low-quality translator will accept a program containing such a construct without issuing a diagnostic. Given the likely translator behavior, no guideline is recommended here.

Example

A reference to an array degenerates into a pointer to its first element:

incomplete array indexing

```
1  int ar[];
2
3  void f(void)
4  {
5  ar[2] = 3;
6  }
```

even though `ar` is incomplete at the time it is indexed. Below is a pointer to an incomplete type:

```
1  struct T *pt, *qt;
2
3  void g(void)
4  {
5  *pt=*qt;
6  }
```

Information on the members of the structure `T` is not needed when dealing with pointers to that structure.

---

729 Except when it is the operand of the **`sizeof`** operator or the unary **`&`** operator, or is a string literal used to initialize an array, an expression that has type "array of *type*" is converted to an expression with type "pointer to *type*" that points to the initial element of the array object and is not an lvalue.

array converted to pointer

Commentary

If arrays were converted to a pointer to their first element when appearing as the operand of the **`sizeof`** operator, it would be impossible to obtain their sizes.

A similar conversion occurs in the declaration of parameters having an array type. Almost at the drop of a hat objects having an array type are converted into a pointer to their first element. Thus, complete arrays, unlike structures, cannot be assigned or passed as parameters because an occurrence of the identifier, denoting the array object, is converted to a pointer to the first element of that object.

array type adjust to pointer to

wide string literal type of

String literals have type array of **char**, or array of wchar_t.

In many contexts applying the unary **&** operator to an operand that is an array index is redundant (rather like using the unary **+** operator). However, support for such usage can simplify the automatic generation of C source (by allowing objects having an array type to be treated like any other object type).

```
int a[10];

&a;    /* Type pointer to array of int */
&a[3]; /* Type pointer to int */
a[3];  /* Type int */
```

This sentence in the C99 Standard has relaxed some of the restrictions that were in the C90 Standard:

```
#include <stdio.h>

struct S {
         int a[2];
         };

struct S WG14_N813(void)
{
struct S x;

x.a[0] = x.a[1] = 1;
return x;
}

int main(void)
{
/*
 * The following would have been a constraint violation in C90
 * since the expression WG14_N813(a).a is not an lvalue (in C90 the
 * member selection operator is only an lvalue if its left operand
 * is an lvalue, and a function call is not an lvalue).
 */
printf("Answer is %d\n", WG14_N813().a[0]);
}
```

C90

*Except when it is the operand of the **sizeof** operator or the unary **&** operator, or is a character string literal used to initialize an array of character type, or is a wide string literal used to initialize an array with element type compatible with wchar_t, an lvalue that has type "array of type" is converted to an expression that has type "pointer to type" that points to the initial element of the array object and is not an lvalue.*

The C90 Standard says " . . . , an lvalue that has type "array of type" is converted . . . ", while the C99 Standard says " . . . , an expression that has type . . . ". It is possible to create a non-lvalue array. In these cases the behavior has changed. In C99 the expression (g?x:y).m1[1] is no longer a constraint violation (C90 footnote 50, "A conditional expression does not yield an lvalue").

In the following, C90 requires that the size of the pointer type be output, while C99 requires that the size of the array be output.

```
#include <stdio.h>

struct {
       int m1[2];
```

```
5           } x, y;
6   int g;
7
8   int main(void)
9   {
10  printf("size=%ld\n", sizeof((g?x:y).m1));
11  }
```

C++

The . . . , array-to-pointer (4.2), . . . standard conversions are not applied to the operand of ***sizeof****.* 5.3.3p4

An lvalue or rvalue of type "array of N T" or "array of unknown bound of T" can be converted to an rvalue of type "pointer to T." The result is a pointer to the first element of the array. 4.2p1

A string literal . . . can be converted . . . This conversion is considered only when there is an explicit appropriate pointer target type, and not when there is a general need to convert from an lvalue to an rvalue. 4.2p2

When is there an explicit appropriate pointer target type? Clause 5.2.1 Subscripting, requires that one of the operands have type *pointer to T*. A character string literal would thus be converted in this context.

If the left operand is not of class type, the expression is implicitly converted (clause 4) to the cv-unqualified type of the left operand. 5.17p3

After determining the type of each parameter, any parameter of type "array of T" or "function returning T" is adjusted to be "pointer to T" or "pointer to function returning T," respectively. 8.3.5p3

Clause 5.3.1p2 does not say anything about the conversion of the operand of the unary **&** operator. Given that this operator requires its operand to be an lvalue not converting an lvalue array to an rvalue in this context would be the expected behavior.

There may be other conversions, or lack of, that are specific to C++ constructs that are not supported in C.

Other Languages

Implicitly converting an array to a pointer to its first element is unique to C (and C++).

Common Implementations

Most implementations still follow the C90 behavior. `gcc`'s support for `(g?x:y).m1[1]` is a known extension to C90 that is not an extension in C99.

Coding Guidelines

The implicit conversion of array objects to a pointer to their first element is a great inconvenience in trying to formulate stronger type checking for arrays in C.

Inexperienced, in the C language, developers sometimes equate arrays and pointers much more closely than permitted by this requirement (which applies to uses in expressions, not declarations). For instance, in:

file_1.c
```
extern int *a;
```

file_2.c
```
extern int a[10];
```

the two declarations of a are sometimes incorrectly assumed by developers to be compatible. It is difficult to see what guideline recommendation would overcome incorrect developer assumptions (or poor training). If the guideline recommendation specifying a single point of declaration is followed, this problem will not occur.

identifier ?? declared in one file

function designator 732 converted to type

Unlike the function designator usage, developers are familiar with the fact that objects having an array type are implicitly converted to a pointer to their first element. Whether applying a unary **&** operator to an operand having an array type provides readers with a helpful visual cue or causes them to wonder about the intent of the author ("what is that redundant operator doing there?") is not known.

Example

```
static double a[5];

void f(double b[5])
{
double (*p)[5] = &a;
double **q = &b;    /* This looks suspicious, */

p = &b;             /* and so does this.      */
q = &a;
}
```

---

array object register storage class

If the array object has register storage class, the behavior is undefined. 730

Commentary

The conversion specified in the previous sentence is implicitly returning the address of the array object. This statement is making it clear what the behavior is in this case (no diagnostic is required). It is a constraint violation to explicitly take the address, using the **&** operator, of an object declared with the **register** storage class (even although in this case there is no explicit address-of operator).

unary & operand constraints footnote 83

C90

This behavior was not explicitly specified in the C90 Standard.

C++

7.1.1p3 *A **register** specifier has the same semantics as an **auto** specifier together with a hint to the implementation that the object so declared will be heavily used.*

Source developed using a C++ translator may contain declarations of array objects that include the **register** storage class. The behavior of programs containing such declarations will be undefined if processed by a C translator.

Coding Guidelines

Given that all but one uses of an array object defined with register storage class, result in undefined behavior (its appearance as the operand of the **sizeof** operator is defined), there appears to be no reason for specifying this storage class in a declaration. However, the usage is very rare and no guideline recommendation is made.

731 A *function designator* is an expression that has function type. function designator

Commentary

This defines the term *function designator*, a term that is not commonly used in developer discussions. The phrase *designates a function* is sometimes used.

C++

This terminology is not used in the C++ Standard.

Other Languages

Many languages do not treat functions as types. When they occur in an expression, it is because the name of the function is needed in a function call. A few languages do support function types and allow an expression to evaluate to a function type. These languages usually define some technical terminology to describe the occurrence.

732 Except when it is the operand of the **sizeof** operator[54] or the unary **&** operator, a function designator with type "function returning *type*" is converted to an expression that has type "pointer to function returning *type*". function designator converted to type

Commentary

Using the unary **&** operator with function types is redundant (as it also is for array types), but supporting such usage makes the automatic generation of C source code simpler (by allowing objects having pointer-to function type to be treated like any other object type). In:

```
1  extern void f(void);
2
3  void g(void)
4  {
5  void (*pf)(void) = f;
6
7  f();
8  pf();
9  }
```

the last two occurrences of `f` are converted to a pointer-to the function denoted by the declaration at file scope. In the first case this pointer is assigned to the object `pf`. In the second case the pointer value is operated on by the **()** operator, causing the pointed-to function to be called. The definition of `g` could, equivalently (and more obscurely), have been:

```
1  void g(void)
2  {
3  void (*pf)(void) = &f;
4
5  &f();
6  (*pf)();
7  }
```

C++

The . . . , and function-to-pointer (4.3) standard conversions are not applied to the operand of ***sizeof****.* 5.3.3p4

5.3.1p2

The result of the unary **&** *operator is a pointer to its operand. The operand shall be an lvalue or a* `qualified-id`*. In the first case, if the type of the expression is "T," the type of the result is "pointer to T."*

While this clause does not say anything about the conversion of the operand of the unary **&** operator, given that this operator returns a result whose type is "pointer to T", not converting it prior to the operator being applied would be the expected behavior. What are the function-to-pointer standard conversions?

4.3p1 *An lvalue of function type T can be converted to an rvalue of type "pointer to T."*

5p8 *Whenever an lvalue expression appears as an operand of an operator that expects an rvalue for that operand, . . . , or function-to-pointer (4.3) standard conversions are applied to convert the expression to an rvalue.*

In what contexts does an operator expect an rvalue that will cause a function-to-pointer standard conversion?

5.2.2p1 *For an ordinary function call, the postfix expression shall be either an lvalue that refers to a function (in which case the function-to-pointer standard conversion (4.3) is suppressed on the postfix expression), or it shall have pointer to function type.*

The suppression of the function-to-pointer conversion is a difference in specification from C, but the final behavior is the same.

5.16p2 *If either the second or the third operand has type (possibly cv-qualified)* **`void`***, then the . . . , and function-to-pointer (4.3) standard conversions are performed on the second and third operands,*

5.17p3 *If the left operand is not of class type, the expression is implicitly converted (clause 4) to the cv-unqualified type of the left operand.*

8.3.5p3 *After determining the type of each parameter, any parameter of type "array of T" or "function returning T" is adjusted to be "pointer to T" or "pointer to function returning T," respectively.*

This appears to cover all cases.

Other Languages

Those languages that support function types provide a variety of different mechanisms for indicating that the address of a function is being taken. Many of them do not contain an address-of operator, but do have an implicit conversion based on the types of the operands in the assignment.

Common Implementations

Most implementations generate code to call the function directly (an operation almost universally supported by host processors), not to load its address and indirectly call it.

Coding Guidelines

The fact that a function designator is implicitly converted to a pointer-to function type is not well-known to developers. From the practical point of view, this level of detail is unlikely to be of interest to them. Using the unary **&** operator allows developers to make their intentions explicit. The address is being taken and the parentheses have not been omitted by accident. Use of this operator may be redundant from the translators point of view, but from the readers point of view, it can be a useful cue to intent. In:

```
int f(void)
{ /* ... */ }

void g(void)
{
if (f)  /* Very suspicious. */
   ;
if (f())
   ;
if (&f) /* Odd, but the intent is clear. */
   ;
}
```

Cg 732.1 When the address of a function is assigned or compared against another value, the address-of operator shall be applied to the function designator.

Example

In the following, all the function calls are equivalent:

```
void f(void);

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

they all cause the function `f` to be called.

733 **Forward references:** address and indirection operators (6.5.3.2), assignment operators (6.5.16), common definitions **<stddef.h>** (7.17), initialization (6.7.8), postfix increment and decrement operators (6.5.2.4), prefix increment and decrement operators (6.5.3.1), the **sizeof** operator (6.5.3.4), structure and union members (6.5.2.3).

734 53) The name "lvalue" comes originally from the assignment expression **E1 = E2**, in which the left operand **E1** is required to be a (modifiable) lvalue. footnote 53

Commentary

Or *left value* of assignment. The *right value* being called the rvalue. This is translator writers' terminology that the C Standard committee has adopted.

C++

The C++ Standard does not provide a rationale for the origin of the term *lvalue*.

Other Languages

Many languages have terms denoting the concept of lvalue and rvalue. Some use these exact terms.

735 It is perhaps better considered as representing an object "locator value".

Commentary

Many languages only allow an object to be modified through assignment. C is much more flexible, allowing objects to be modified by operators other than assignment (in the case of the prefix increment/decrement operators the object being modified appears on the right). Happily, the term *locator* starts with the letter *l* and can be interpreted to have a meaning close to that required.

Coding Guidelines

This term is not in common usage by any C related groups or bodies. The term *lvalue* is best used. It is defined by the C Standard and is also used in other languages.

---

rvalue

What is sometimes called "rvalue" is in this International Standard described as the "value of an expression". 736

Commentary

The term *rvalue* only appears in this footnote. It is not generally used by C developers, while the term *value of expression* is often heard.

C++

The C++ Standard uses the term *rvalue* extensively, but the origin of the term is never explained.

3.10p1 *Every expression is either an lvalue or an rvalue.*

Other Languages

The term *rvalue* is used in other languages.

Coding Guidelines

The term *value of an expression* is generally used by developers. While the term *rvalue* is defined in the C++ Standard, its usage by developers in that language does not appear to be any greater than in C. There does not seem to be any benefit in trying to change this commonly used terminology.

An obvious example of an lvalue is an identifier of an object. 737

Commentary

string literal
static storage duration

A nonobvious lvalue is a string literal. An obvious example of an rvalue is an integer constant.

As a further example, if `E` is a unary expression that is a pointer to an object, `*E` is an lvalue that designates the object to which `E` points. 738

Commentary

And it may, or may not also be a modifiable lvalue.

---

footnote 54

54) Because this conversion does not occur, the operand of the **`sizeof`** operator remains a function designator and violates the constraint in 6.5.3.4. 739

Commentary

sizeof constraints

The committee could have specified a behavior that did not cause this constraint to be violated. However, the size of a function definition is open to several interpretations. Is it the amount of space occupied in the function image, or the number of bytes of machine code generated from the statements contained within the function definition? What about housekeeping information that might need to be kept on the stack during program execution? Whichever definition is chosen, it would require a translator to locate the translated source file containing the function definition. Such an operation is something that not only does not fit into C's separate compilation model, but could be impossible to implement (e.g., if the function definition had not yet been written). The Committee could have specified that the size need not be known until translation phase 8 (i.e., link-time) but did not see sufficient utility in supporting this functionality.

translation phase 8

C++

The C++ Standard does not specify that use of such a usage renders a program ill-formed:

> *The* **`sizeof`** *operator shall not be applied to an expression that has function or incomplete type, or to an enumeration type before all its enumerators have been declared, or to the parenthesized name of such types, or to an lvalue that designates a bit-field.* 5.3.3p1

References

1. E. Caspi. Empirical study of opportunities for bit-level specialization in word-based programs. Thesis (m.s.), University of California, Berkeley, 2000.