The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.5.3.2 Address and indirection operators

Constraints

unary & operand constraints

The operand of the unary **&** operator shall be either a function designator, the result of a **[]** or unary * operator, or an lvalue that designates an object that is not a bit-field and is not declared with the **register** storage-class specifier. 1088

Commentary

bit-field packed into

byte addressable unit

register storage-class

Bit-fields are permitted (intended even) to occupy part of a storage unit. Requiring bit addressing could be a huge burden on implementations. Very few processors support bit addressing and C is based on the byte being the basic unit of addressability.

The **register** storage-class specifier is only a hint to the translator. Taking the address of an object could effectively prevent a translator from keeping its value in a register. A harmless consequence, but the C Committee decided to make it a constraint violation.

C90

The words:

> *. . . , the result of a* ***[ ]*** *or unary * operator,*

are new in C99 and were added to cover the following case:

```
1  int a[10];
2
3  for (int *p = &a[0]; p < &a[10]; p++)
4      /* ... */
```

where C90 requires the operand to refer to an object. The expression a+10 exists, but does not refer to an object. In C90 the expression &a[10] is undefined behavior, while C99 defines the behavior.

C++

Like C90 the C++ Standard does not say anything explicit about the result of a **[]** or unary * operator. The C++ Standard does not explicitly exclude objects declared with the **register** storage-class specifier appearing as operands of the unary **&** operator. In fact, there is wording suggesting that such a usage is permitted:

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. [Note: the hint can be ignored and in most implementations it will be ignored if the address of the object is taken. —end note]*

Source developed using a C++ translator may contain occurrences of the unary **&** operator applied to an operand declared with the **register** storage-class specifier, which will cause a constraint violation if processed by a C translator.

```
1  void f(void)
2  {
3  register int a[10]; /* undefined behavior */
4                      // well-formed
5
6  &a[1]  /* constraint violation */
7         // well-formed
8       ;
9  }
```

Other Languages

Many languages that support pointers have no address operator (e.g., Pascal and Java, which has references, not pointers). In these languages, pointers can only point at objects returned by the memory-allocation functions. The address-of operator was introduced in Ada 95 (it was not in available in Ada 83). Many languages do not allow the address of a function to be taken.

Coding Guidelines

In itself, use of the address-of operator is relatively harmless. The problems occur subsequently when the value returned is used to access storage. The following are three, coding guideline related, consequences of being able to take the address of an object:

- It provides another mechanism for accessing the individual bytes of an object representation (a pointer to an object can be cast to a pointer to character type, enabling the individual bytes of an object representation to be accessed). pointer converted to pointer to character
- It is an alias for the object having that address.
- It provides a mechanism for accessing the storage allocated to an object after the lifetime of that object has terminated.

Assigning the address of an object potentially increases the scope over which that object can be accessed. When is it necessary to increase the scope of an object? What are the costs/benefits of referring to an object using its address rather than its name? (If a larger scope is needed, could an objects definition be moved to a scope where it is visible to all source code statements that need to refer to it?)

The parameter-passing mechanism in C is pass by value. What is often known as *pass by reference* is achieved, in C, by explicitly passing the address of an object. Different calls to a function having pass-by-reference arguments can involve different objects in different calls. Passing arguments, by reference, to functions is not a necessity; it is possible to pass information into and out of functions using file scope objects. function call preparing for

Assigning the address of an object creates an alias for that object. It then becomes possible to access the same object in more than one way. The use of aliases creates technical problems for translators (the behavior implied by the use of the **`restrict`** keyword was introduced into C99 to help get around this problem) and can require developers to use additional cognitive resources (they need to keep track of aliased objects). restrict intended use

A classification often implicitly made by developers is to categorize objects based on how they are accessed, the two categories being those accessed by the name they were declared with and those accessed via pointers. A consequence of using this classification is that developers overlook the possibility, within a sequence of statements, of a particular object being modified via both methods. When readers are aware of an object having two modes of reference (a name and a pointer dereference) is additional cognitive effort needed to comprehend the source? Your author knows of no research in on this subject. These coding guidelines discuss the aliasing issue purely from the oversight point of view (faults being introduced because of lack of information), because there is no known experimental evidence for any cognitive factors.

One way of reducing aliasing issues at the point of object access is to reduce the number of objects whose addresses are taken. Is it possible to specify a set of objects whose addresses should not be taken and what are the costs of having no alternatives for these cases? Is the cost worth the benefit? Restricting the operands of the address operator to be objects having block scope would limit the scope over which aliasing could occur. However, there are situations where the addresses of objects at file scope needs to be used, including:

- An argument to a function could be an object with block scope, or file scope; for instance, the `qsort` function might be called.
- In resource-constrained environments it may be decided not to use dynamic storage allocation. For instance, all of the required storage may be defined at file scope and pointers to objects within this storage used by the program.

- The return from a function call is sometimes a pointer to an object, holding information. It may simplify storage management if this is a pointer to an object at file scope.

The following guideline recommendation ensures that the storage allocated to an object is not accessed once the object's lifetime has terminated.

Cg 1088.1 The address of an object shall not be assigned to another object whose scope is greater than that of the object assigned.

Dev 1088.1 An object defined in block scope, having static storage duration, may have its address assigned to any other object.

function designator converted to type
array converted to pointer

A function designator can appear as the operand of the address-of operator. However, taking the address of a function is redundant. This issue is discussed elsewhere. Likewise for objects having an array type.

Example

In the following it is not possible to take the address of a or any of its elements.

```
1  register int a[3];
```

In fact this object is virtually useless (the identifier a can appear as the operand to the **sizeof** operator). If allocated memory is not permitted (we know the memory requirements of the following on program startup):

```
1  extern int *p;
2
3  void init(void)
4  {
5  static int p_obj[20];
6
7  p=&p_obj;
8  }
```

This provides pointers to objects, but hides those objects within a block scope. There is no pointer/identifier aliasing problem.

---

unary * operand has pointer type

The operand of the unary * operator shall have pointer type. 1089

Commentary

Depending on the context in which it occurs, there may be restrictions on the pointed-to type (because of the type of the result).

unary * 1098 result type

C++

5.3.1p1 *The unary * operator performs indirection: the expression to which it is applied shall be a pointer to an object type, or a pointer to a function type . . .*

C++ does not permit the unary * operator to be applied to an operand having a pointer to **void** type.

```
1  void *g_ptr;
2
3  void f(void)
4  {
5  &*g_ptr; /* DR #012 */
6           // DR #232
7  }
```

Other Languages

In some languages indirection is a postfix operator; for instance, Pascal uses the token ^ as a postfix operator.

Semantics

1090 The unary & operator yields the address of its operand.

unary & operator

Commentary

For operands with static storage duration, the value of the address operator may be a constant (objects having an array type also need to be indexed with a constant expression). There is no requirement that the address of an object be the same between different executions of the same program image (for objects with static storage duration) or different executions of the same function (for objects with automatic storage duration).

address constant

All external function references are resolved during translation phase 8. Any identifier denoting a function definition will have been resolved.

translation phase 8

The C99 Standard refers to this as the *address-of* operator.

footnote 79

C90

This sentence is new in C99 and summarizes what the unary & operator does.

C++

Like C90, the C++ Standard specifies a pointer to its operand (5.3.1p1). But later on (5.3.1p2) goes on to say: "In particular, the address of an object of type "*cv* T" is "pointer to *cv* T," with the same cv-qualifiers."

Other Languages

Many languages do not contain an address-of operator. Fortran 95 has an address assignment operator, =>. The left operand is assigned the address of the right operand.

Common Implementations

Early versions of K&R C treated p=&x as being equivalent to p&=x.[5]

In the case of constant addresses the value used in the program image is often calculated at link-time. For objects with automatic storage duration, their address is usually calculated by adding a known, at translation time, value (the offset of an object within its local storage area) to the value of the frame pointer for that function invocation. Addresses of elements, or members, of objects can be calculated using the base address of the object plus the offset of the corresponding subobject.

Having an object appear as the operand of the address-of operator causes many implementations to play safe and not attempt to perform some optimizations on that object. For instance, without sophisticated pointer analysis, it is not possible to know which object a pointer dereference will access. (Implementations often assume all objects that have had their address taken are possible candidates, others might use information on the pointed-to type to attempt to reduce the set of possible accessed objects.) This often results in no attempt being made to keep the values of such objects in registers.

Implementations' representation of addresses is discussed elsewhere.

pointer type describes a

1091 If the operand has type "*type*", the result has type "pointer to *type*".

Commentary

Although developers often refer to the address returned by the address-of operator, C does not have an *address* type.

1092 If the operand is the result of a unary * operator, neither that operator nor the & operator is evaluated and the result is as if both were omitted, except that the constraints on the operators still apply and the result is not an lvalue.

&*

Commentary

footnote 84 *&

The only effect of the operator pair &* is to remove any lvalueness from the underlying operand. The combination *& returns an lvalue if its operand is an lvalue. This specification is consistent with the behavior of the last operator applied controlling lvalue-ness. This case was added in C99 to cover a number of existing coding idioms; for instance:

```
#include <stddef.h>

void DR_076(void)
{
int *n = NULL;
int *p;

/*
 * The following case is most likely to occur when the
 * expression *n is a macro argument, or body of a macro.
 */
p = &*n;
/* ... */
}
```

C90

The responses to DR #012, DR #076, and DR #106 specified that the above constructs were constraint violations. However, no C90 implementations known to your author diagnosed occurrences of these constructs.

C++

This behavior is not specified in C++. Given that either operator could be overloaded by the developer to have a different meaning, such a specification would be out of place.

At the time of this writing a response to C++ DR #232 is being drafted (a note from the Oct 2003 WG21 meeting says: "We agreed that the approach in the standard seems okay: `p = 0; *p;` is not inherently an error. An lvalue-to-rvalue conversion would give it undefined behavior.").

```
void DR_232(void)
{
int *loc = 0;

if (&*loc == 0) /* no dereference of a null pointer, defined behavior */
                // probably not a dereference of a null pointer.
   ;

&*loc = 0; /* not an lvalue in C */
           // how should an implementation interpret the phrase must not (5.3.1p1)?
}
```

Common Implementations

Some C90 implementations did not optimize the operator pair &* into a no-op. In these implementations the behavior of the unary * operator was not altered by the subsequent address-of operator. C99 implementations are required to optimize away the operator pair &*.

Similarly, if the operand is the result of a `[]` operator, neither the `&` operator nor the unary * that is implied by the `[]` is evaluated and the result is as if the `&` operator were removed and the `[]` operator were changed to a + operator. 1093

Commentary

This case was added in C99 to cover a number of coding idioms; for instance:

```
void DR_076(void)
{
int a[10];
int *p;

/*
 * It is possible to point one past the end of an object.
 * For instance, we might want to loop over an object, using
 * this one past the end value.  Given the equivalence that
 * applies to the subscript operator the operand of & in the
 * following case is the result of a unary * operator.
 */
p = &a[10];

for (p = &a[0]; p < &a[10]; p++)
   /* ... */ ;
}
```

C90

This requirement was not explicitly specified in the C90 Standard. It was the subject of a DR #076 that was closed by adding this wording to the C99 Standard.

C++

This behavior is not specified in C++. Given that either operator could be overloaded by the developer to have a different meaning, such a specification would be out of place. The response to C++ DR #232 may specify the behavior for this case.

Common Implementations

This requirement describes how all known C90 implementations behave.

Coding Guidelines

The expression `&a[index]`, in the visible source code, could imply

- a lack of knowledge of C semantics (why wasn't `a+index` written?),
- that the developer is trying to make the intent explicit, and
- that the developer is adhering to a coding standard that recommends against the use of pointer arithmetic— the authors of such standards often view `(a+index)` as pointer arithmetic, but `a[index]` as an array index (the equivalence between these two forms being lost on them).

array subscript identical to

1094 Otherwise, the result is a pointer to the object or function designated by its operand.

Commentary

There is no difference between the use of objects having a pointer type and using the address-of operator. For instance, the result of the address-of operator could be assigned to an object having the appropriate pointer type, and that object used interchangeably with the value assigned to it.

Common Implementations

In most implementations a pointer refers to the actual address of an object or function.

pointer type describes a

1095 The unary * operator denotes indirection.

unary * indirection

Commentary

The terms *indirection* and *dereference* are both commonly used by developers.

C++

5.3.1p1 *The unary * operator performs indirection.*

Other Languages

Some languages (e.g., Pascal and Ada) use the postfix operator `^`. Other languages— Algol 68 and Fortran 95— implicitly perform the indirection operation. In this case, an occurrence of operand, having a pointer type, is dereferenced to return the value of the pointed-to object.

Coding Guidelines

Some coding guideline documents place a maximum limit on the number of simultaneous indirection operators that can be successively applied. The rationale being that deeply nested indirections can be difficult to comprehend. Is there any substance to this claim?

sequential nesting * sequential nesting ()

Expressions, such as `***p`, are similar to nested function calls in that they have to be comprehended in a right-to-left order. The issue of nested constructions in natural language is discussed in that earlier C sentence. At the time of this writing there is insufficient experimental evidence to enable a meaningful cost/benefit analysis to be performed and these coding guidelines say nothing more about this issue.

If sequences of unary * operators are needed in an expression, it is because an algorithm's data structures make the usage necessary. In practice, long sequences of indirections using the unary * operator are rare. Like the function call case, it may be possible to provide a visual form that provides a higher-level interpretation and hides the implementation's details of the successive indirections.

member selection

An explicit unary * operator is not the only way of specifying an indirection. Both the array subscript, `[]`, and member selection, `->`, binary operators imply an indirection. Developers rarely use the form `(*s).m` (`(&s)->m`), the form `s->m` (`s.m`) being much more obvious and natural. While the expression `s1->m1->m2->m3` is technically equivalent to `(*(*(*s1).m1).m2).m3`, it is comprehended in a left-to-right order.

Usage

SPEC benchmarks

A study by Mock, Das, Chambers, and Eggers[6] looked at how many different objects the same pointer dereference referred to during program execution (10 programs from the SPEC95 and SPEC2000 benchmarks were used). They found that in 90% to 100% of cases (average 98%) the set of objects pointed at, by a particular pointer dereference, contained one item. They also performed a static analysis of the source using a variety of algorithms for deducing points-to sets. On average (geometric mean) the static points to sets were 3.3 larger than the dynamic points to sets.

If the operand points to a function, the result is a function designator; 1096

Commentary

The operand could be an object, with some pointer to function type, or it could be an identifier denoting a function that has been implicitly converted to a pointer to function type. This result is equivalent to the original function designator. Depending on the context in which it occurs this function designator may be converted to a pointer to function type.

function designator function designator converted to type

C++

The C++ Standard also specifies (5.3.1p1) that this result is an lvalue. This difference is only significant for reference types, which are not supported by C.

Other Languages

Those languages that support some form of pointers to functions usually only provide a mechanism for, indirect, calls to the designated value. Operators for obtaining the function designator independent of a call are rarely provided. Some languages (e.g., Algol 88, Lisp) provide a mechanism for defining anonymous functions in an expression context, which can be assigned to objects and subsequently called.

Common Implementations

For most implementations the result is an address of a storage location. Whether there is a function definition (translated machine code) at that address is not usually relevant until an attempt is made to call the designated function (using the result).

Coding Guidelines

Because of the implicit conversions a translator is required to perform, the unary * operator is not required to cause the designated function to be called. There are a number of situations that can cause such usage to appear in source code: the token sequence may be in automatically generated source code, or the sequence may occur in developer-written source via arguments passed to macros, or developers may apply it to objects having a pointer to function type because they are unaware of the implicit conversions that need to be performed.

Example

```
1  extern void f(void);
2  extern void (*p_f)(void);
3
4  void g(void)
5  {
6  f();
7  (*f)();
8  (*******f)();
9  (*p_f)();
10 }
```

1097 if it points to an object, the result is an lvalue designating the object.

Commentary

The indirection operator produces a result that allows the pointed-to object to be treated like an anonymous object. The result can appear in the same places that an identifier (defined to be an object of the same type) can appear. The resulting lvalue might not be a modifiable lvalue. There may already be an identifier that refers to the same object. If two or more different access paths to an object exist, it is said to be *aliased*.

modifiable lvalue
object
aliased

Common Implementations

Some processors (usually CISC) have instructions that treat their operand as an indirect reference. For instance, an indirect load instruction obtains its value from the storage location pointed to by the storage location that is the operand of the instruction.

1098 If the operand has type "pointer to *type*", the result has type "*type*".

unary *
result type

Commentary

The indirection operator removes one level of pointer from the operand's type. The operand is required to have pointer type. In many contexts the result type of a pointer to function type will be implicitly converted back to a pointer type.

1089 unary *
operand has pointer type
function designator
converted to type

1099 If an invalid value has been assigned to the pointer, the behavior of the unary * operator is undefined.[84)]

Commentary

footnote 84

pointer to void converted to/from

The standard does not provide an all-encompassing definition of what an *invalid value* is. The footnote gives some examples. An invalid value has to be created before it can be assigned and this may involve a conversion operation. Those pointer conversions for which the standard defines the behavior do not create invalid values. So the original creation of the invalid value, prior to assignment, must also involves undefined behavior.

object initial value indeterminate

If no value has been assigned to an object, it has an indeterminate value.

array subscript identical to

The equivalence between the array access operator and the indirection operator means that the behavior of what is commonly known as an *out of bounds array access* is specified here.

C++

The C++ Standard does not explicitly state the behavior for this situation.

Common Implementations

For most implementations the undefined behavior is decided by the behavior of the processor executing the program. The root cause of applying the indirection operator to an invalid valid is often a fault in a program and implementations that perform runtime checks sometimes issue a diagnostic when such an event occurs. (Some vendors have concluded that their customers would not accept the high performance penalty incurred in performing this check, and they don't include it in their implementation.) The result can often be manipulated independently of whether there is an object at that storage location, although some processors do perform a few checks.

pointer cause undefined behavior

Techniques for detecting the dereferencing of invalid pointer values usually incur a significant runtime overhead[1,3,4,7,8] (programs often execute at least a factor of 10 times slower). A recent implementation developed by Dhurjati and Adve[2] reported performance overheads in the range 12% to 69%.

Coding Guidelines

guidelines not faults

This usage corresponds to a fault in the program and these coding guidelines are not intended to recommend against the use of constructs that are obviously faults.

Forward references: storage-class specifiers (6.7.1), structure and union specifiers (6.7.2.1). 1100

References

1. T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, 1994.
2. D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceeding of the 28th International Conference on Software Engineering*, pages 162–171, 2006.
3. D. M. Jones. The Model C Implementation. Knowledge Software Ltd, 1992.
4. R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In M. Kamkar and D. Byers, editors, *Third International Workshop on Automated Debugging*. Linkoping University Electronic Press, 1997.
5. B. W. Kernighan. Programming in C-A tutorial. Technical Report ???, Bell Laboratories, Aug. ???
6. M. Mock, M. Das, C. Chambers, and S. J. Eggers. Dynamic points-to sets: A comparison with static analysis and potential applications in program understanding and optimization. Technical Report UW CSE Technical Report 01-03-01, University of Washington, Mar. 2001.
7. Y. Oiwa, T. Sekiguichi, E. Sumii, and A. Yonezawa. Fail-safe ANSI-C compiler: An approach to making C programs secure. In M. Okada, B. Pierce, A. Scedrov, H. Tokuda, and A. Yonezawa, editors, *Software Security – Theories and Systems*, pages 133–153. Springer-Verlag, Apr. 2003.
8. J. L. Steffen. Adding run-time checking to the portable C compiler. *Software–Practice and Experience*, 22(4):305–316, 1992.