Unary * Operator
Randy Meyers
N697 J11/97-060
May 27, 1997

1. Introduction

During the last year, the committee has been making some unfortunate interpretations of the Standard dealing with the indirection operator, the address operator, and the index operator. In particular, the committee has decided that the following idiom is not conforming:

```
int *p, a[10];

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

because **`a[10]`** is not an lvalue for an object. Given that idiom appears in lots of code and runs without problems everywhere, it does not seem reasonable to call it not conforming.

Originally, this paper was going to suggest wording to fix this problem. That turned out to be harder than I thought, and I attempted to withdraw the paper. Tom Plum requested that I at least write down a description of the problem promote further discussion.

2. Why This Should Work

`&a[10]` is identical to **`&(*(a + 10))`**

The Standard guarantees that you can form a pointer to one past any object. Thus all implementations must be able to represent **`a + 10`**, although some implementations must work hard to meet this requirement (for example, never allocate an object at the end of a segment boundary).

In terms of the abstract semantics, the unary **`*`** operator takes a pointer rvalue and turns it into an lvalue. This process does not further stress an implementation. In terms of real computer architecture, the unary **`*`** operator itself does not cause any code to be generated until you know how the result of the unary **`*`** operator is going to be used. The code generated by a unary **`*`** that is the operand of an unary **`&`** operator potentially looks very different than the code for a unary **`*`** that is the target of an assignment or the code for a unary **`*`** whose value is actually fetched.

In particular, the code for **`&*p`** looks just like the code for some pointer **`p`**, although the code for either expression might be in some cases a materialized value in a register versus just an operand specifier in an instruction. The important thing is that there is no fetch that occurs for **`*p`** that is reversed for **`&*p`**. Thus, any implementation that can represent **`p`**, can also represent **`&*p`**.

3. The Interpretation

The committee has been interpreting **`&a[10]`** above as the following.

By Section 6.3.2.1 (Array subscripting), paragraph 2, the above expression is identical to
`&(*(a + 10))`

By Section 6.3.6 (Additive Operators), paragraph 8, **`a + 10`** is a valid pointer value that points one past the end of the array.

Section 6.3.6, paragraph 8 (again) makes **`*(a + 10)`** undefined:

> Unless both the pointer operand and the result point to elements of the same array object, or the pointer operand points one past the last element of an array object and the result points to an element of the same array object, the behavior is undefined if the result is used as an operand of the unary * operator.

In the above expression, the pointer operand of the **`+`** operator points within the array object and the result of the **`+`** operator points one past the array object. Thus, the behavior is undefined.

(Part of the problem is that the above rule fails to special case a unary **`*`** that is the operand of a unary **`&`**.)

Section 6.3.3 (Address and indirection operators), paragraph 4 says:

> The unary * operator denotes indirection. If the operand points to a function, the result is a function designator; if it points to an object, the result is an lvalue designating the object. If the operand has type ``pointer to type," the result has type ``type." If an invalid value has been assigned to the pointer, the behavior of the unary * operator is undefined.

This presents another problem: In **`*(a + 10)`**, **`a +10`** is a valid pointer value, but it is not a pointer to an object.

Section 6.3.3, paragraph 1 (constraints) says:

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

The final problem: **`*(a + 10)`** is not an lvalue that designates an object. Note that "designates an object" is redundant since Section 6.2.2.1 defines lvalues as an expression that designates an object.

It appears allowing **`&(*(a + 10))`** requires a new type of lvalue that corresponds to pointers to one past an object. Such lvalues could not be fetched (undergo the lvalue to rvalue conversion from Section 6.2.2.1 paragraph 2) or be used to store a value.

Try your hand at improved wording.