Document Number: J16/00-0011
WG21 N1234
Date: 7 March, 2000
Project: PL C++
Reply to: William M Miller
wmm@fastdial.net

Proposed Resolution for Core Issue 73

I. The Problem

The definition of comparing two pointers for equality is found in 5.10¶1:

> Pointers to objects or functions of the same type (after pointer conversions) can be compared for equality. Two pointers of the same type compare equal if and only if they are both null, both point to the same object or function, or both point one past the end of the same array.

What does it mean for two pointers to "point to the same object?" At least three situations have been mentioned where the answer to this question is less than obvious.

The first scenario that was raised is the case where storage allocation results in the "one-past-the-end" of an array having the same address as an object that is not part of the array. For instance, in the following code, if the implementation allocates no padding between arrays `a` and `b`, is it permissible for the function to return the value `true`?

```
struct {
     int a[1];
     int b[1];
} x;

bool f() {
     return a+1 == b;
}
```

According to one interpretation of the wording above, the answer is "no" – the expression `a+1` cannot be considered to "point to" `b[0]`.

The second problem mentioned for the current wording is that `void*` pointers do not point to **objects**, so they either cannot be compared or can never compare equal. (There can be no objects of type `void`.)

The third problem arises from the fact that different objects can occupy the same storage location at different times. For example,

```
struct B { };
struct D1: B { };
struct D2: B { };
void f() {
     B* bp1 = new D1;
     B* bp2 = new (bp1) D2;
     return bp1 == bp2;
}
```

The Standard gives restrictions in section 3.8 about the operations that are permitted in such cases. The use in this example is not one of those forbidden in 3.8, so it is presumably allowed. However, since `bp1` and `bp2` evidently "point to" different objects, the wording in 5.10 requires that they not compare equal.

II. Analysis

The reason these questions are important is that we want, for efficiency reasons, to permit implementations to behave in certain ways that are not allowed by the interpretations cited above. In particular, an efficient implementation will simply compare the values (addresses) in the pointers rather than keeping track of which object is associated with each. We would like to ensure that such an implementation is conformant.

II.A. Alternative Interpretations

Each of the objections to the current wording mentioned above is the result of a particular interpretation of how the wording applies. In each case, however, an alternative interpretation exists that allows a conforming implementation to exhibit the desired behavior.

In the first case (an object fortuitously appearing at an address not necessarily associated with it), the issue presupposes a certain definition of the phrase "points to." However, the Standard does not define this term. Furthermore, the Standard allows a pointer to receive its value in essentially arbitrary fashion (via conversion from an integral type, for instance) and assumes that such a pointer value validly "points to" an object occurring at that address. Given this degree of liberty, it would seem that the only definition of "points to" that is consistent with the Standard is the operational definition: if there is an object of the correct type at the address indicated by the value of the pointer, the pointer must be said to "point to" that object. With this definition, then, `a+1` in the example above does indeed "point to" `b[0]`.

The second case (a `void*` pointer) is much less convincing in light of the wording of 3.9.2¶4:

> Objects of cv-qualified (3.9.3) or cv-unqualified type `void*` (pointer to void), can be used to **point to** objects of unknown type. A `void*` shall be able to hold any object pointer. *[emphasis added]*

At least in this context, `void*` pointers can be described as "pointing to" objects.

As for the third issue, a closer examination of section 3.8 reveals that a pointer to an object after its lifetime has ended may validly be used only "before the storage which the object occupied is reused or released" (3.8¶5) or if the "reuse" is by creation of an object of the same type at that location (3.8¶7). In the latter case, the "pointer that pointed to the original object … will automatically refer to the new object," and there is no conflict with the current wording of pointer comparison. In the example, however, the reuse is by creation of an object of a different type and is thus covered by neither of the allowed cases. The behavior of this code is not explicitly described by the Standard and is therefore undefined; an implementation is completely unconstrained in such cases. Implementations are, therefore, free to behave in the desired fashion in comparing pointers to different generations of objects residing sequentially in a single storage location.

Although these interpretations are defensible, not everyone will find them compelling. The next section suggests an alternative approach that addresses the problem more directly.

II.B. Levels of Abstraction

The issue in all three of these cases can be linked to the use of the word **object**. (Although the original question was phrased in terms of the definition of the phrase "points to," it can be just as well approached from the perspective that "one-past-the-end" of an array is not an object.)

This divergence between the specification and implementation of pointer comparison can be viewed as a difference in level of abstraction – objects are at a higher level of abstraction than addresses. This distinction is treated explicitly elsewhere in the Standard. Section 1.7 deals with the C++ memory model and defines the term *address*. However, there is no mention of "object" in that section; discussion of objects properly belongs in section 1.8, dealing with the object model. The problems cited above result from the fact that the specification of pointer comparison is phrased in terms of the object model, while implementations work at the level of the memory model.

The obvious way to fix these problems is to say in 5.10 that the result of comparing two pointers yields `true` if and only if the values of the pointers address the same byte. The difficulty with this approach is that there is nothing currently in the Standard saying that the values of pointers are, in fact, addresses! Instead, the undefined notion of "pointing to" is used to describe the values of pointers (which is where the discussion started):

III. Proposed Changes

The following wording is intended to clarify three points that are, at best, implicit in the current wording of the Standard:

- The values of pointers are addresses

- Pointers "point to" objects if there is an object of the appropriate type at the address contained in the pointer

- The equality of pointer values is determined by comparing the addresses they represent

In 3.9.2¶3, add the following wording immediately preceding, "The value representation of pointer types is implementation-defined."

> A valid value of an object pointer type represents either the addresses of a byte in memory (1.7) or a null pointer (4.10). If an object of type `T` is located at an address `A`, a pointer of type *cv* `T*` whose value is the address `A` is said to *point to* that object, regardless of how the value was obtained. [*Note:* for instance, the address one past the end of an array (5.7) would be considered to point to an unrelated object of the array's element type that might be located at that address.] The value representation…

In 5.10¶1, change the sentence beginning, "Two pointers of the same type…" to read

> Two pointers of the same type compare equal if and only if they are both null, both point to the same function, or both represent the same address (3.9.2).

IV. Issues

One concern that might be raised over this proposal is whether it causes problems for architectures in which a single address might be represented by a number of different bit patterns. For example, a pointer might be composed of a base and an offset, where different combinations of base and offset could designate the same location in memory. If pointers to the same byte are required to compare equal, and if two such pointers can have different representations, would this proposal require an implementation on such an architecture to normalize pointers before comparing them? If so, that would be a major performance penalty for such architectures.

In fact, this proposal imposes no restrictions on such architectures beyond those required by the current wording. Conforming implementations are already required to normalize pointers within a single "top-level" object in order to support features such as ordering comparisons and `offsetof` – that is, there must be a single base used for pointers within a given complete object. However, the question is whether this proposal would penalize an implementation that used different bases for distinct objects.

The specific case that might cause a problem is when two pointers whose base addresses reflect different objects have offsets that effectively cause them to designate the same byte in the overall address space. However, for this situation to arise, it must be the case that one or both of the pointer values is the result of an address calculation that exceeded the bounds of the object associated with the base component of the address. According to section 5.7, the behavior in such cases is undefined. An implementation is thus free to do

an efficient comparison of the two pointers and treat them as unequal, even though they in fact designate the same effective address.

V. Acknowledgments

Clark Nelson of Intel reviewed an earlier version of this paper and made a number of valuable suggestions.