Access to Objects
WG14/N640
X3J11/97-003

Bill Homer
Cray Research, Inc.
655F Lone Oak Drive
Eagan, MN 55121

January 2, 1997

Abstract

A paragraph in section 6.3 (Expressions) limits the types of lvalues through which the value of an object shall be accessed. The current language does not appear to be appropriate for objects with no declared type, such as those allocated by one of the library memory management functions. New language is proposed to remedy this.

1 Introduction

Paragraph 7 of section 6.3 currently reads:

> An object shall have its stored value accessed only by an lvalue expression that has one of the following types:
>
> - a type compatible with the declared type of the object,
> - a qualified version of a type compatible with the declared type of the object,
> - a type that is the signed or unsigned type corresponding to the declared type of the object,
> - a type that is the signed or unsigned type corresponding to a qualified version of the declared type of the object,
> - an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or
> - a character type.

This paragraph appears to imply that only lvalues with a character type shall be used to access the value of an object without a declared type. Such objects include those allocated with the library memory management functions, `malloc`, `calloc`, and `realloc`. It seems unlikely that this was the intent.

It is proposed to change the specification to sanction a common usage: The type of an object with no declared type is determined by the type of the lvalue used to store a value in the object. This is analogous to the semantics of a union, except that there is no declared list of possible types.

Two additional usages are also addressed in the proposed change. First, if `memcpy` or `memmove` is used to copy a value from one object to another, it seems reasonable that the type of the source object should become the effective type of the target object. Copying values between objects using lvalues with type array of a character type should be treated similarly. Second, it is possible for an object to acquire a value that neither comes from another lvalue nor is otherwise associated with a particular type. This is true of an object allocated with `calloc`, an object modified with `memset`, or a more exotic object such as a read-only memory mapped register. In this case, there is no basis for restricting the type of lvalue that can be used to access the value of the object.

2 Changes to the Standard

Modify section 6.3, paragraph 7, to read:

> The *effective type* of an object for an access to its stored value is the declared type of the object, if it has one. If a value is stored into an object having no declared type through an lvalue having a type that is not a character type, then the type of the lvalue becomes the effective type of the object for that access and for subsequent accesses that do not modify the stored value. If a value is copied into an object having no declared type using `memcpy` or `memmove`, or is copied as an array of character type, then the effective type of the modified object for that access and for subsequent accesses that do not modify the value is the effective type of the object from which the value is copied, if it has one. For all other accesses to an object having no declared type, the effective type of the object is simply the type of the lvalue used for the access.
>
> An object shall have its stored value accessed only by an lvalue expression that has one of the following types:
>
> - a type compatible with the effective type of the object,
> - a qualified version of a type compatible with the effective type of the object,
> - a type that is the signed or unsigned type corresponding to the effective type of the object,
> - a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,
> - an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or
> - a character type.