Fat Pointers

WG14/N312

X3511/93-059

Michael Meissner

Open Software Foundation
11 Cambridge Center
Cambridge, MA. 02142
meissner@osf.org

This is an alternate proposal to the Cray Research 'Arrays of Variable Length' proposal that merges the Fat pointer proposal from USL with variable length arrays. The main difference between the two is that Fat Pointers encode the array information within an extended pointer (usually called a dope vector in the programming language literature), while the Cray proposal deals with passing variably sized arrays. The portion of the Cray proposal that deals with creating variably sized arrays is identical to the current proposal.

3.1.2.4 Storage Duration of Objects

An Object whose identifier is declared with no linkage and without the storage-class specifier static has automatic storage duration. Storage is guaranteed to be reserved for a new instance of such an object on each normal entry into the block with which it is associated. If the block with which the object is associated is entered by a jump from outside the block to a labeled statement in the block or in an enclosed block, then the storage is guaranteed to be reserved provided the object does not have a variable length array type. If the object is variably qualified and the block is entered by a jump to a labeled statement, then the behavior is undefined. If an initialization is specified for the value stored in the object, it is performed on each normal entry, but not if the block is entered by a jump to a labeled statement. Storage for the object is no longer guaranteed to be reserved when execution of the block ends in any way. (Entering an enclosed block suspends but does not end execution of the enclosing block. Calling a function suspends but does not end execution of the block containing the call.) The value of a pointer that referred to an object with automatic storage duration that is no longer guaranteed to be reserved is indeterminate.

Forward references: variably qualified (3.5), variable length array (3.5.4.2).

3.2.2.3 Pointers

A pointer to void may be converted to or from a pointer to any incomplete or object type. A pointer to any incomplete or object type may be converted to a pointer to void and back again; the result shall compare equal to the original pointer.

For any qualifier q, a pointer to a non-q-qualified type may be converted to a pointer to the q-qualified version of the type; the values stored in the original and converted pointers shall compare equal.

An integral constant expression with the value 0, or such an expression cast to type void *, is called a null pointer constant. If a null pointer constant is assigned to or compared for equality to a pointer, the constant is converted to a pointer of that type. Such a pointer, call a null pointer, is guaranteed to compare unequal to a pointer to any object or function. Two null pointers, converted

hrough possibly different sequences of casts to pointer types, shall ompare equal.

An array may be converted to a fat pointer. The base pointer to he array and all bounds specified by *, are stored in the fat ointer. Nonarray types, except for a null pointer may not be onverted to a fat pointer, since the bounds would not be known. A ull pointer converted to a fat pointer stores 0 for each of the ounds, and a null pointer into the base pointer to the variable ength array. A fat pointer converted to any other pointer to object r incomplete type, or integer uses the base pointer to the variable ength array , discarding the array bounds.

orward references: cast operators (3.3.4), equality operators (3.3.9), imple assignment
3.3.16.1), fat pointers (3.5.4.2).

.3.3.4 The sizeof operator

emantics

When applied to an operand that has array type, the result is the otal number of bytes in the array. For variable length array types he result is not a constant expression and is computed at program xecution time.

When applied to an operand that has fat pointer type, the result is if a null pointer was assigned to the pointer, or the total number f bytes in the array pointed to by the pointer. If the fat pointer as never assigned to, the result is undefined. The result is not a onstant expression, and is computed at program execution time.

orward reference: variable length array (3.5.4.2), fat pointer 3.5.4.2).

.4 Constant Expressions

emantics

An integral constant expression shall have integral type and shall nly have operands that are integer constants, enumeration constants, haracter constants, sizeof expressions whose operand does not have a ariable length array type, and floating constants that are the mmediate operands of casts. An arithmetic constant expression shall ave arithmetic type and shall only have operands that are integer onstants, floating constants, enumeration constants, character onstants, and sizeof expressions whose operand does not have a ariable length array or fat pointer type.

.5 Declarations

emantics

If the sequence of specifiers in a declarator contains a variable ength array type, the type specified by the declarator is said to be ariably qualified.

orward reference: variable length array (3.5.4.2).

.5.2 Type Specifiers

onstraints

ly identifiers with automatic storage duration can have a ably qualified type.

Only ordinary identifiers (as defined in 3.1.2.3) with automatic torage duration may be declared with a variable length array type.

.5.2.1 Structure and Union Specifiers

Constraints

A structure or union shall not contain a member with a variable ength array type.

Forward reference: specifiers for variably qualified types are escribed in 3.5.4.2.

.5.4.2 Array declarators

Constraints

The [and] shall delimit an expression or *. If [and] delimit an expression (which specifies the size of an array), it shall be an integral type. If the expression is a constant expression then it hall have a value greater than zero.

ntics

If, in the declaration "T D1," D1 has the form

D[assignment-expressionopt]

or

D[*]

nd the type specified for ident in the declaration "T D" is "derived-eclarator-type-list T," then the type specified for ident in "T D" s "derived-declarator-type-list array of T." If the size xpression is not present the array type is an incomplete type. If he size expression is a constant expression, the array type is a ixed length array type. Otherwise, the size expression (which may ontain side effects), is evaluated at program execution time, shall valuate to a value greater than zero, and the array type is a ariable length array type.

If any sequence of array declarators, contain *, all of the array eclarators in the sequence are treated as a fat pointer to a equence of variable length arrays. The fat pointer includes a base ointer to the start of the array, and for each bound specified by *, he corresponding array bound.

For two fixed length array types to b e compatible, both shall have ompatible element types and if both size specifiers are present, the ifiers shall have the same constant value. For two array types e compatible when at least one is a variable length array or fat ointer type, both shall have compatible element types. Furthermore, f two types are required to be compatible then the dimension size

pecifier of a variable length array type must evaluate, at program
xecution time, as equal to the value of the other dimension specifier.
t is undefined behavior if these corresponding dimension size
pecifiers produce unequal values at execution time.