Arrays of Variable Length

X3J11
VLA Subcommittee
Technical Report
Draft 1.0
X3J16/94-007
WG21/N0464
WG14/N317
X3J11/94-001

Foreward

NCEG, at its initial meeting in May 1989, identified support for variable length arrays as one of its focus areas and organized a subgroup to produce a technical report. NCEG became technical subcommittee X3J11.1 under its parent committee X3J11 (the committee that defined the C Standard). On December 7, 1993, X3J11.1 voted to forward the following proposal up to the parent committee with the recommendation that it be included in the final Technical Report (TR).

A previous version of this draft (X3J11.1/93-007) was distributed for review to various professional organizations including X3J11, WG14, X3J16, WG21, X3T2, and X3J3. Received comments were presented to X3J11.1 and the appropriate changes were made and approved in a subsequent revision (X3J11.1/93-043).

The inability to declare arrays whose size is known only at runtime is often cited as a primary deterrent to using C as a numerical computing language. Arrays of this nature are implemented in the current GNU-C and Cray Research C compilers. The adoption of arrays whose size is known only at runtime was proposed to committee X3J11 but was dismissed as having too many far-reaching implications. Eventual adoption of some standard notion of runtime arrays is considered crucial for C's acceptance as a major player in the numerical computing world. This paper describes an implementation of variable length arrays which Cray Research has chosen for its Standard C Compiler.

People who have made valuable contributions to this document are:

Analog Devices, Marc Hoffman
Analog Devices, Kevin Leary
AT&T Bell Laboratories, Dennis Ritchie
Control Data Corporation, Azar Hashemi
Convex Corporation, Austin Curley
Convex Corporation, Bill Torkelsom
Cray Research, Incorporated, Dave Becker
Cray Research, Incorporated, Steve Collins
Cray Research, Incorporated, John Dawson
Cray Research, Incorporated, Mike Holly
Cray Research, Incorporated, Bill Homer
Cray Research, Incorporated, David Knaak
Cray Research, Incorporated, Mark Pagel
Cray Research, Incorporated, Kent Zoya
DEC Professional, Rex Jaeschke
Digital Equipment Corporation, Randy Meyers
Farance Inc, Frank Farance
Free Software Foundation, Richard Stallman
Hewlett Packard, John Kwan
IBM, Fred Tydeman
Keaton Consulting, David Keaton
Kendall Square Research, Tim Peters
Lawrence Livermore National Laboratory, Linda Stanberry
Open Software Foundation, Mike Meissner
SunPro, David Hough
SunPro, Bob Jervis
Thinking Machines Corporation, James L. Frankel
Unix System Laboratories, David Prosser

Arrays of Variable Length

X3J16/94-0077
WG21/N0464

Tom MacDonald
Cray Research, Inc.
655F Lone Oak Drive
Eagan, MN 55121
tam@cray.com

Introduction

This is Cray Research's proposal for Variable Length Array (VLA) types. Insertion of this new array type into the existing C Standard requires an examination of each section of the existing Standard with necessary elaboration for those sections that are affected. Accordingly, the numbering, titles and general format of each of the following several paragraphs match those found in the current C Standard. The ISO section numbers are listed first, followed by section numbers in parentheses that correspond to the former ANSI standard X3.159-1989. Section numbers from both document numbers are used to help those that still have the old document follow this document.

If a paragraph is meant to address definitive issues relating to *variable length arrays,* then an overview of these issues appears followed by a brief rationale section detailing Cray Research's current approach. Therefore, this proposal describes each feature as edits to the current C Standard with rationale interspersed appropriately.

6.1.2.4 (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 storage is guaranteed to be reserved provided the object does not have a variable length array type. If the object is variably modified 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 modified* (6.5), (3.5), *variable length array* (6.5.4.2), (3.5.4.2).

6.3 (3.3) Expressions

6.3.3.4 (3.3.3.4) The **`sizeof`** operator

Semantics

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

Forward reference: *variable length array* (6.5.4.2), (3.5.4.2).

Example 1

```
int i = 2, j = 3, k, koo;
int *p = &i;

int func(int n) {
     char b[*p==n ? k = j++ : n+j]; /* side effects OK */
     return (sizeof(b));            /* runtime expression */
}

main() {
   koo = func(10);   /* runtime sizeof; koo == 13 */
}
```

ISSUE

Overview:

The **sizeof** operator could previously be used in any constant expression (except those associated with preprocessor directives such as **#if**). With this extension to the language, such is no longer the case since runtime code must be generated to calculate the size of a *variable length array* type.

Rationale:

The notion of ''size'' is an important part of such operations as pointer increment, subscripting, and pointer difference. Although the **sizeof** operator will now produce a value computed at runtime, there still exists a consistency when applied to the previously mentioned operations. Furthermore, it can still be used as an operand to the **malloc** function and to compute the number of elements in an array.

6.3.6 (3.3.6) Additive Operators

Rationale:

The description of pointer arithmetic does not require modification. Pointer arithmetic is still well defined with pointers to *variable length array* types as shown by the following example.

Example 2

```
{
 int n = 4, m = 3;
 int a[n][m];
 int (*p)[m] = a;    /* p == &a[0]      */
 p += 1;             /* p == &a[1]      */
 (*p)[2] = 99;       /* a[1][2] == 99 */
 n = p - a;          /* n == 1          */
}
```

If array **a** in the above example is declared to be a fixed length array, and pointer **p** is declared to be a pointer to a fixed length array that points to **a**, the computed results are still the same.

6.4 (3.4) Constant Expressions

Semantics

An *integral constant expression* shall have integral type and shall only have operands that are integer constants, enumeration constants, character constants, **`sizeof`** expressions whose operand does not have a *variable length array* type, and floating constants that are the immediate operands of casts. An *arithmetic constant expression* shall have arithmetic type and shall only have operands that are integer constants, floating constants, enumeration constants, character constants, and **`sizeof`** expressions whose operand does not have a *variable length array* type.

6.5 (3.5) Declarations

Semantics

If the sequence of specifiers in a declarator contains a *variable length array* type, the type specified by the declarator is said to be *variably modified.*

Forward reference: *variable length array* (6.5.4.2), (3.5.4.2).

6.5.2 (3.5.2) Type Specifiers

Constraints

Only identifiers with block scope or function prototype scope can have a *variably modified* type. Only ordinary identifiers (as defined in **6.1.2.3 (3.1.2.3)**) without linkage may be declared with a *variable length array* type. If an identifier is declared to be an object with static storage duration, it shall not have a *variable length array* type.

Forward reference: *variable length array* (6.5.4.2), (3.5.4.2).

ISSUES

Overview:

This section discusses three issues about declarations containing *variable length array* specifiers (i.e., VLAs). It is important to note that there is a distinction made between *variable length array* types and *variably modified* types (e.g., a pointer to a *variable length array*). First, all declarations of *variably modified* types must be declared at either block scope or function prototype scope. This means that file scope identifiers cannot be declared with a *variably modified* type of any fashion. Second, array objects declared with either the **`static`** or **`extern`** storage class specifiers cannot be declared with a *variable length array* type. However, block scope pointers declared with the **`static`** storage class specifier can be declared as pointers to *variable length array* types. Finally, if the identifier that is being declared has a *variable length array* type (as opposed to a pointer to an array), then it must be an ordinary identifier. This eliminates structure and union members.

Example 3

```
extern int n;
typedef int T[n];                     /* Error - file scope identifier */
struct { int (*z)[n]; };              /* Error - file scope identifier */
int A[n];                             /* Error - file scope identifier */
static int (*p1)[n];                  /* Error - file scope identifier */
extern int (*p2)[n];                  /* Error - file scope identifier */
int B[100];                           /* OK - file scope but not a VLA */

void func(int m, int C[m][m]) {       /* OK - prototype VLA */
   typedef int VLA[m][m];             /* OK - block scope typedef VLA */
   struct { int (*y)[n];              /* OK - block scope member pointer to VLA */
            int z[n];    };           /* Error - z is not an ordinary identifier */
   int D[m];                          /* OK - auto VLA */
   static int E[m];                   /* Error - static block scope VLA */
   extern int F[m];                   /* Error - F has linkage and is a VLA */
   int (*s)[m];                       /* OK - auto pointer to VLA */
   extern int (*r)[m];                /* Error - r has linkage and is a pointer to VLA */
   static int (*q)[m] = &B;           /* OK - static block scope pointer to VLA */
}
```

Rationale:

Restricting *variable length array* declarators to identifiers with automatic storage duration is natural since ''variableness'' at file scope requires some notion of parameterized typing.

6.5.2.1 (3.5.2.1) Structure and Union Specifiers

Constraints

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

Forward reference: *variable length array* (6.5.4.2), (3.5.4.2).

ISSUES

Overview:

This issue involves the declaration of members of structures or unions with *variably modified* types. A structure member can be declared to be a pointer to a VLA, but not a VLA. To allow this engenders a host of problems such as treatment when passing such objects (or even pointers to such objects) as parameters. In addition, the **offsetof** macro would require extended meaning for examples similar to the following.

Example 4

```
int n = 8;
main() {
  struct tag {
     int m1;
     int m2[n];          /* not allowed by this proposal */
     int m3;
  };
  int i;
  i = offsetof(struct tag, m1); /* OK */
  i = offsetof(struct tag, m2); /* OK */
  i = offsetof(struct tag, m3); /* undefined behavior? */
}
```

Since the expansion of **offsetof** is implementation specific, it seems impractical to guarantee any behavior for members which might follow a *variable length array* type member. Finally, any structure or union containing a *variable length array* type must not be declared at file scope due to its variable nature. This forces every user to re-specify the structure type inside each function, and raises type compatibility questions. Allowing formal arguments to be structures with *variable length array* type members (as well as pointers to same) was problematic and, since functions could not return them either, the utility of the whole exercise centered around stack allocated structures with variable length members. Objects of this sort are narrowly focused and thus the decision was made to disallow the *variable length array* type within structures and unions.

If all *ordinary identifiers* found in the arbitrary size expressions of a given member were compelled to resolve to other members of the same structure or union, then some utility could be found in allowing structures and unions to have *variably modified* members. Such members would become self-contained "fat pointers" of sorts. The following example demonstrates this concept:

Example 5

```
struct tag {          /* not allowed by this proposal */
   int n;             /* initialized to 5 */
   int m2[n];         /* uses member n to complete size   */
} x = { 5 };
```

The possibility of resolving the size of *variable length array* members amongst other members could be explored. There were two votes taken at NCEG meeting #5 in Norwood, MA, which asked the following questions:

Q: Should there be some way to declare a VLA member
Yes: 11 No: 1 Undecided: 4

Q: In favor of type shown in Example 4? 3
Q: In favor of type shown in Example 5? 6
Q: Undecided? 7

There is considerable sentiment for adding VLA members but it appears the implications of each approach needs to be explored before the committee can reach consensus. This proposal has elected to pursue VLAs outside of structures and unions. A future proposal may attempt to resolve the VLA issue for members.

6.5.4 (3.5.4) Declarators

6.5.4.2 (3.5.4.2) Array declarators

Constraints

The `[` and `]` may 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 shall have a value greater than zero.

Semantics

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

`D[`*assignment-expression*$_{opt}$`]`

or

`D[*]`

and the type specified for *ident* in the declaration "`T D`" is "*derived-declarator-type-list T*," then the type specified for *ident* in "`T D`" is "*derived-declarator-type-list* array of *T*." If the size expression is not present the array type is an incomplete type. If `*` is used instead of a size expression, the array type is a *variable length array* type of unspecified size, which can only be used in declarations with function prototype scope. If the size expression is an integer constant expression and the element type has a fixed size, the array type is a *fixed length array* type. Otherwise, the array type is a *variable length array* type. If the size expression is not a constant expression, it is evaluated at program execution time, it may contain side effects, and shall evaluate to a value greater than zero. The size of a *variable length array* type shall not change until the execution of the block containing the declaration has ended.

For two array types to be compatible, both shall have compatible element types, and if both size specifiers are present and integer constant expressions, then both size specifiers shall have the same constant value. If either size specifier is variable, the two sizes must evaluate, at program execution time, to equal values. If the two array types are used in a context which requires then to be compatible, it is undefined behavior if the two size specifiers evaluate to unequal values at execution time.

Example 6

```
extern int n;
extern int m;

main() {
  int a[n][6][m];
  int (*p)[4][n+1];
  int c[n][n][6][m];
  int (*r)[n][n][n+1];
  p = a;        /* Error - not compatible because 4 != 6 */
  r = c;        /* compatible - assume n == 6 and m == n+1 at execution time */
}
```

Example 7

```
int dim4 = 4;
int dim6 = 6;

void func() {
   int (*q)[dim4][8][dim6];
   int (*r)[dim6][8][1];
   r = q;     /* compatible, but undefined behavior at runtime */
}
```

ISSUES

Overview:

An issue raised by this section concerns type compatibility and the fact that *variably modified* types are not fully specified until execution time. Is that when compatibility checks should occur? Should the checks be mandatory? This kind of execution time checking is considered to be a quality of implementation issue. Finally, the presence of possible side-effects in type declarations is new. Since side effects include volatile variables there seems to be no compelling reason to forbid side effects in *variable length array* size specifiers.

Rationale:

The language chosen by Cray Research reflects a ''path of least resistance'' approach. The ''undefined behavior'' paradigm is invoked if things do not match up at runtime as they normally would for *fixed length arrays.* Prototype side-effects are handled with a syntactic option (see use of `[*]` in section **6.5.4.3 (3.5.4.3)** below) and by inhibiting expression evaluation until function definition time.

6.5.4.3 (3.5.4.3) Function Declarators (Including Prototypes)

Semantics

For each parameter declared with *variable length array* type, the type used for compatibility comparisons is the one that results from conversion to a pointer type, as in **6.7.1 (3.7.1)**. In a function prototype declarator that is not part of a function definition, the syntax `[*]` may be used to specify a *variable length array* type.

Example 8

```
/* Following prototype has a variably modified parameter */

void addscalar(int n, int m, double a[n][n*m+300], double x);

double A[4][308];

main() {
   addscalar(4, 2, A, 3.14);
}

void addscalar(int n, int m, double a[n][n*m+300], double x) {

   int i, j, k=n*m+300;

   for (i=0; i<n; i++)
      for (j=0; j<k; j++)
         a[i][j] += x;      /* a is a pointer to a VLA of size: n*m+300 */
}
```

The following are all compatible function prototype declarators.

Example 9

```
void matrix_mult(int n, int m, double a[n][m], double b[m][n], double c[n][n]);

void matrix_mult(int n, int m, double a[*][*], double b[*][*], double c[*][*]);

void matrix_mult(int n, int m, double a[ ][*], double b[ ][*], double c[ ][*]);

void matrix_mult(int n, int m, double a[ ][m], double b[ ][n], double c[ ][n]);
```

The size expression can involve identifiers declared at file scope. The following example shows how this can be done.

Example 10

```
extern int n;

void f(double a[][n], int n) { ... }  /* file scope n used */

extern int m;

void g(double b[][n*m]) { ... }
```

The following example is also acceptable. The parameter **n** is used to complete the VLA size expression.

Example 11

```
extern int n;

void g(int n, int a[][n]) {  /* OK */

    ...

}
```

ISSUES

Overview:

Currently, programmers do not need to concern themselves with the order in which formal parameters are specified, and one common programming style is to declare the most important parameters first. Consider the following prototype definition:

Example 12

```
/* prototype declaration for old-style definition */

void f(double a[*][*], int n);

void f(a, n)
   int n;
   double a[n][n];
{
   ...
}
```

The order in which the names are specified in the parameter list does not depend on the order of the parameter declarations themselves. However, because of the lexical ordering rules in Standard C, the accompanying prototype declaration is not allowed. All identifiers used in VLA size expressions must be declared prior to use (as is the case with the usage of all identifiers).

Example 13

```
void f(double a[n][n], int n) {  /* Error - n declared after use */
   ...
}
```

With Standard C's lexical ordering rules the declaration of **a** would force **n** to be undefined or captured by an outside declaration. The possibility of allowing the scope of parameter **n** to extend to the beginning of the parameter-type-list was explored (relaxed lexical ordering). This allows the size of parameter **a** to be defined in terms of parameter **n.** However, such a change to the lexical ordering rules is not considered to be in the "Spirit of C." This is an unforeseen side-effect of Standard C prototype syntax. The full proposal for relaxed lexical ordering semantics is in **Appendix B**.

An example of a way to declare parameters in any order but avoid lexical ordering issues is the following:

Example 14

```
void g(void *ap, int n) {
   double (*a)[n] = ap;

   ... a[1][2] ...

}
```

In this case the parameter **ap** is assigned to a local pointer that is declared to be a pointer to a VLA.

Another issue concerns function prototype declarators whose parameters have *variable length array* types. Since the dimension size specifiers of *variable length array* types in prototype declarators which are not definitions are resolved, two passes are required over the parameter type list in order to diagnose undeclared identifiers.

```
/* prototype declarator whose parameter type remains incomplete */
/* Error - x and y are never declared */

void f1(double a[x][y], int n);
```

Rationale:

This issue concerns prototype declarators that contain parameters with *variable length array* types that are never completed. A previous version of this document allowed this behavior. The motivation was to not force a a diagnostic because the actual array sizes were never needed. A vote that was taken at NCEG meeting #5 in Norwood, MA, asked the following question:

Q: Which notation is preferred to denote a VLA parameter (in a prototype)?

Arbitrary **[x]** (where **x** can include undeclared names) : 0
[*] : 11
Undecided : 3

Since the **[*]** syntax seems to be the most widely accepted this proposal has been changed to reflect this sentiment. Also, the presence of these diagnostics will help uncover more programmer errors.

6.5.6 (3.5.6) Type definitions

Constraints

Typedef declarations which specify a *variably modified* type shall have block scope. The array size specified by the *variable length array* type shall be evaluated at the time the type definition is declared and not at the time it is used as a type specifier in an actual declarator.

Example 15

```
main() {
  extern void func();
  func(20);
}

void func(int n) {
  typedef int A[n];    /* OK - because declared in block scope */
  A a;
  A *p;
  p = &a;
}
```

Example 16

```
int n;
typedef int A[n];       /* Error - because declared at file scope    */
```

Example 17

```
void func(int n) {
     typedef int A[n]; /* A is n ints with n evaluated now */
     n += 1;
     {
         A a;            /* a is n ints - n without += 1 */
         int b[n];       /* a and b are different sizes */
         for (i = 1; i < n; i++)
            a[i-1] = b[i];
     }
  }
```

ISSUE

Overview:

The question arises whether the non-constant expression which engenders the *variable length array* type should be evaluated at the time the type definition itself was declared or each time the type definition is invoked for some object declaration.

Rationale:

If the evaluation were to take place each time the typedef name is used, then a single type definition could yield *variable length array* types involving many different dimension sizes. This possibility seemed to violate the spirit of type definitions and Cray Research decided to force evaluation of the expression at the time the type definition itself is declared.

6.5.7 (3.5.7) Initialization

Constraints

The type of the entity to be initialized shall be an object type or an array of unknown size, but not a *variable length array* type.

Example 18

```
int n;
main() {
   int a[n] = {1, 2};         /* Error */
   int (*p)[n] = &a;          /* OK - pointer is scalar of fixed size */
}
```

6.6 (3.6) Statements

Semantics

A *full expression* is an expression that is not part of another expression. Each of the following is a full expression: ***a variably modified declarator;*** an initializer; the expression in an expression statement; the controlling expression of a selection statement (**if** or **switch**); the controlling expression of a **while** or **do** statement; each of the three (optional) expressions of a **for** statement; the (optional) expression in a **return** statement. The end of a full expression is a sequence point.

6.6.2 (3.6.2) Compound Statement, or Block

Semantics

A *compound statement* (also called a *block*) allows a set of statements to be grouped into one syntactic unit, which may have its own set of declarations and initializations (as discussed in **6.1.2.4 (3.1.2.4)**). The initializers of objects that have automatic storage duration ***and variably qualified declarators*** are evaluated and the values are stored in the objects in the order their declarators appear in the translation unit.

Example 19

```
{
   int n = 3, m = 4;
   int a[n++][n++];         /* order of evaluation is undefined */
   int b[m++], c[m++];      /* order of evaluation is defined */
}
```

Rationale:

Since sequence points control the order of evaluation of expressions with side effects, it seems reasonable to add a sequence point after every *variably modified* declarator. This specifies the behavior of declarations such as:

```
int x[n++];
int y[n++];
```

in that the side effects must be evaluated at the end of the declarator. However, the behavior of a declaration such as:

```
int z[n++][n++];
```

is not specified because there is no guaranteed order of evaluation until the declarator is complete. This seems consistent with an expression such as:

```
z[n++][n++]
```

in which there is no guaranteed order of evaluation either.

6.6.4.2 (3.6.4.2) Switch statement

Constraints

The controlling expression shall not cause a block to be entered by a jump from outside the block to a statement that follows a **case** or **default** label in the block (or an enclosed block) if it contains the declaration of a *variably modified* object or *variably modified* typedef name.

Example 20

```
int i = 0;
main() {
    int n = 10;
    switch (n) {        /* Error - bypasses declaration of a[n] */
        int a[n];
        case 10:
            a[0] = 1;
            break;
        case 20:
            a[0] = 2;
            break;
    }
    switch (i) {        /* OK - declaration of b[n] is not bypassed */
        case 0:
            {
                int b[n];
                b[2] = 4;
            }
            break;
        case 1:
            break;
    }
}
```

ISSUE

Overview:

The concern here is that a **switch** will bypass evaluation of the arbitrary integer expressions associated with a *variably modified* type. In the case of a simple pointer that is *variably modified* this could cause incorrect pointer arithmetic. In the case of a *variable length array* type this is even more serious in that the *variably modified* object itself is not even allocated.

Rationale:

Disallowing a jump passed the declaration of a *variably modified* identifier in this fashion seems reasonable and a compile time diagnostic is issued.

6.6.6.1 (3.6.6.1) Goto statements

Constraints

The identifier in a **goto** statement shall name a label located somewhere in the enclosing function. A **goto** statement shall not cause a block to be entered by a jump from outside the block to a labeled statement in the block (or an enclosed block) if it contains the declaration of a *variably modified* object or *variably modified* typedef name.

Example 21

```
void func(int n) {
     int j = 4;
     goto lab3;          /* Error - going INTO scope of variable length array */
     {
        double a[n];
        a[j] = 4.4;
lab3:
        a[j] = 3.3;
        goto lab4;       /* OK - going WITHIN scope of variable length array */
        a[j] = 5.5;
lab4:
        a[j] = 6.6;
     }
     goto lab4;          /* Error - going INTO scope of variable length array */
     return;
}
```

ISSUE

Overview:

The concern here is that a **goto** will bypass the evaluation of the arbitrary integer expressions associated with a *variably modified* type. In the case of a simple pointer that is *variably modified* this causes wrong pointer arithmetic. In the case of a *variable length array* object, the object itself would not even be allocated.

Rationale:

Disallowing a jump passed the declaration of a *variably modified* identifier in this fashion seems reasonable and a compile time diagnostic is issued.

6.7.1 (3.7.1) Function Definitions

On entry to the function all size expressions of *variably modified* parameters are evaluated.

7.6.2.1 (4.6.2.1) The `longjmp` function

Description

If a **`longjmp`** function invocation causes the termination of a function or block in which *variable length array* objects are still allocated, then the behavior is undefined.

ISSUE

Overview:

The **`longjmp`** function that returns control back to the point of the **`setjmp`** invocation might cause memory associated with a *variable length array* object to be squandered. Consider the following example:

Example 22

```
#include <setjmp.h>
void g(int n);
void h(int n);
int n = 6;

void f() {
   int x[n];           /* OK - f is not terminated */
   setjmp (buf);
   g(n);
}

void g(int n) {
   int a[n];           /* undefined - a is still allocated */
   h(n);
}

void h(int n) {
   int b[n];           /* undefined - b is still allocated */
   longjmp(buf,2);     /* causes undefined behavior */
}
```

In this example the function **`h`** might cause storage to be lost for the *variable length array* object **`a`**, declared in function **`g`** (whose existence is unknown to **`h`**). Additional storage could be lost for *variable length array* object **`b`** declared in function **`h`**. In this case an implementation knows that a **`longjmp`** is being performed in the presence of a *variable length array* object whose storage needs to be managed accordingly. However, since **`longjmp`** is a function, it can be invoked through a "pointer to function" and thus **`h`** would have no knowledge of the **`longjmp`** invocation occurring.

Rationale:

The Cray Research implementation ensures that any *variable length array* type objects are placed on the existing stack first and heap storage is used only if room cannot be found on the stack. Thus, unless there is excessive stack allocation, no space will be wasted because only vacant space on the stack was used initially.

A LANGUAGE SYNTAX SUMMARY

A.1.2.2 Declarations

(6.5.4), (3.5.4) direct-declarator:

identifier

(*declarator***)**

direct-declarator **[***assignment-expression*$_{opt}$**]**

direct-declarator **[***$_{opt}$**]**

direct-declarator **(***parameter-type-list***)**

direct-declarator **(***identifier-list*$_{opt}$**)**

(6.5.5), (3.5.5) direct-abstract-declarator:

(*abstract-declarator***)**

direct-abstract-declarator **[***assignment-expression*$_{opt}$**]**

direct-abstract-declarator **[***$_{opt}$**]**

direct-abstract-declarator **(***parameter-type-list*$_{opt}$**)**

B Relaxed Lexical Ordering Semantics

The following is a description of changes necessary to implement the relaxed lexical ordering rules for *variable length array* types declared in function prototypes. The relaxed lexical ordering rules allow the size expressions of these *variable length array* declarations to contain variables declared lexically after the arrays (but still inside the same prototype). The relaxed lexical ordering rules permit parameters to be declared in any order. This change to the lexical ordering rules was debated at great length and finally rejected because it's not in the "Spirit of C." It is included in this appendix to help others wishing to do research in this area.

6.5.4.3 (3.5.4.3) Function Declarators (Including Prototypes)

Semantics

For each parameter declared with *variable length array* type, the type used for compatibility comparisons is the one that results from conversion to a pointer type, as in **6.7.1 (3.7.1)**. In a function prototype declarator that is not part of a function definition, the syntax `[*]` may be used to specify a *variable length array* type. In a parameter-type-list, if an identifier is both declared as a parameter and appears in an array size expression, then the scope of that identifier is extended to the beginning of the parameter-type-list (rather than beginning just after the completion of its declarator). If the size of a *variable length array* depends upon the size of itself, or another variable length array declared in the same parameter-type-list, then the behavior is undefined.

Example 23

```
/* Following prototype has a variably modified parameter */

void addscalar(int n, int m, double a[n][n*m + 300], double x);

double A[4][308];

main() {
   addscalar(4, 2, A, 3.14);
}

void addscalar(int n, int m, double a[n][n*m + 300], double x) {

   int i,j,k=n*m+300;

   for (i=0; i<n; i++)
      for (j=0; j<k; j++)
         a[i][j] += x;      /* a is pointer to a VLA of size: n*m+300 */
}
```

Example 24

```
/* The following are compatible function prototype declarators: */

void matrix_mult(int n, int m, double a[n][m], double b[m][n], double c[n][n]);

void matrix_mult(int n, int m, double a[*][*], double b[*][*], double c[*][*]);

void matrix_mult(int n, int m, double a[ ][*], double b[ ][*], double c[ ][*]);

void matrix_mult(int n, int m, double a[ ][m], double b[ ][n], double c[ ][n]);
```

Since the order in which parameters are specified is unimportant, the following compatible function prototypes are functionally equivalent to the above four prototypes in that the values of parameters **n** and **m** are used to complete the *variable length array* modifiers.

```
void matrix_mult(double a[n][m], double b[m][n], double c[n][n], int n, int m);

void matrix_mult(double a[*][*], double b[*][*], double c[*][*], int n, int m);

void matrix_mult(double a[ ][*], double b[ ][*], double c[ ][*], int n, int m);

void matrix_mult(double a[ ][m], double b[ ][n], double c[ ][n], int n, int m);
```

The following atypical program shows that the meaning of this currently conforming program has changed. The parameter **n** is used in the array size expression instead of the enumeration constant **n**. This change is made to eliminate any surprising behavior if an enumeration constant **n** is, say, present in a header file and changes the meaning of an intended VLA declaration. For example, this quietly changes the meaning of the following program.

Example 25

```
enum { n = 5 };

/* parameter n used instead of enum constant n */
void f1(int a[][n], int n) {}
```

Without this change, the size used by parameter **a** is the enumeration constant and not the intended parameter.

The following atypical program is not currently standard conforming because the parameter **x** declared in function **func** is a VLA. Again the scope of parameter **t** is extended to the beginning of the parameter list.

Example 26

```
enum {r=3, s=4, t=5};
float a[6][6];

void func(int, float [][*], int);

main() {
   func(4, a, 2);   /* need ptr to 6 element array */
}

/* parameter t used instead of enum constant t */

void func(int n, float x[][n+t], int t) {
   printf("%s\n",
      sizeof(*x)==sizeof(float [6]) ? "OK" : "Compiler error");
   return;
}
```

The following function declarator contains an error because the array sizes of parameters `p` and `q` depend upon the size of a VLA declared within the parameter-type-list.

Example 27

```
/* Error - p and q use VLA sizes of parameters */

int func(int (*p)[sizeof(*q)], int (*q)[sizeof(*p)]) { ... }
```

In the following function prototypes, the parameter `n` is used to complete the variable length array types. The file scope variable `n` is never referenced.

Example 28

```
extern int n;          /* file scope variable is never referenced!! */

void f(double a[][n], int n) { ... }

extern int m;          /* file scope variable */

void g(double b[][n*m], int n) { ... }
```

The following example is also acceptable. The parameter `n` is used to complete the VLA size expression.

Example 29

```
extern int n;

void g(int n, int a[][n]) {  /* OK */

    ...

}
```

ISSUES

Overview:

By far the most controversial issue involves the "lexical ordering problem" that is presented when prototypes with *variably modified* parameters are used and the size expression involves an identifier that is not visible. Currently, programmers do not need to concern themselves with the order in which formal parameters are specified, and one common programming style is to declare the most important parameters first. Consider the following prototype definition:

Example 30

```
/* prototype declaration for old-style definition */

void f(double a[*][*], int n);

void f(a, n)
    int n;
    double a[n][n];
{
    ...
}
```

The order in which the names are specified in the parameter list does not depend on the order of the parameter declarations themselves. The accompanying prototype declaration is compatible with this definition and thus it seems appropriate to allow the following prototype definition in lieu of the previous old-style definition.

Example 31

```
void f(double a[n][n], int n) {
    ...
}
```

With current lexical ordering rules the declaration of `a` would force `n` to be undefined or captured by an outside declaration; but in no case would the parameter `n` be used as the programmer clearly intends. This is an unforeseen side-effect of Standard C prototype syntax. The following vote that was taken at NCEG meeting #5 in Norwood, MA, about the importance of preserving lexical ordering.

In favor of preservation of lexical ordering? 4
In favor of more relaxed rules? 4
Undecided? 7

At X3J11.1 meeting #1 in Cupertino, CA, a proposal was made to require a permutation such that all identifiers in size expressions must associate with a visible definition, and that no two permutations result in different associations. This same vote was taken again:

In favor of preservation of lexical ordering? 7
In favor of more relaxed ordering? 7
Undecided? 2

This approach introduced its own set of issues and did not significantly affect the vote.

An example of a way to declare parameters in any order but avoid lexical ordering issues is the following:

```
void g(void *ap, int n) {
   double (*a)[n] = ap;

   ... a[1][2] ...

}
```

In this case the parameter **ap** is assigned to a local pointer that is declared to be a pointer to a VLA.

Another issue concerns function prototype declarators whose parameters have *variable length array* types. Since the dimension size specifiers of *variable length array* types in prototype declarators which are not definitions are resolved, two passes are required over the parameter type list in order to diagnose undeclared identifiers.

```
/* prototype declarator whose parameter type remains incomplete */
/* Error - x and y are never declared */

void f1(double a[x][y], int n);
```

Rationale:

By assigning all identifiers in VLA size expressions a ***universal type*** that is compatible with any type, the parsing may proceed to the matching **]** token. A possible solution to the "lexical ordering problem" then involves tokenizing, syntactically analyzing, and marking where the *variably modified* expressions are found within dimension specifiers, until the **)** that terminates the prototype is found. A second scan of the these *variably modified* expressions associates identifiers with ***universal types*** to visible declarations thereby completing the VLA types. If during the second scan it is determined that an identifier used in a VLA size expression is still not visible, a diagnostic message is produced.

Another issue concerns prototype declarators that contain parameters with *variable length array* types that are never completed. A previous version of this document allowed this behavior. The motivation was to not force the implementation to make two passes over the prototype just to issue diagnostics. A vote that was taken at NCEG meeting #5 in Norwood, MA, asked the following question:

Q: Which notation is preferred to denote a VLA parameter (in a prototype)?

Arbitrary **[x]** (where **x** can include undeclared names) : 0
[*] : 11
Undecided : 3

Since the **[*]** syntax seems to be the most widely accepted this proposal has been changed to reflect this sentiment. Also, the presence of these diagnostics will help uncover more programmer errors.