Arrays of Variable Length
WG14/N637 (X3J11/96-101)

Tom MacDonald

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

30 December 1996

Introduction

The inability to declare arrays whose size is known only at execution time is often cited as a primary deterrent to using C as a numerical computing language. Eventual adoption of some standard notion of execution time arrays is considered crucial for C's acceptance as a major player in the numerical computing world. This paper is based on an implementation of variable length arrays chosen by Cray Research for its Standard C Compiler, but this specification contains several enhancements recommended by committee reviews.

This document describes a proposal for Variable Length Array (VLA) types. The size of a VLA cannot necessarily be determined at translation time. Rather, its size is usually determined at execution time. Insertion of this new array type into the C Standard requires an examination of each section of the Draft C 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 **WG14/N559 (**a.k.a. **X3J11/96-023) C9X Draft 6**.

Paragraphs address definitive issues relating to VLAs. If new words are added to a paragraph already in the Draft C Standard, a different font is used to highlight them. Overviews of the issues are given after the sections, followed by a brief rationale that explains why certain decisions were reached. Therefore, this proposal describes each feature as edits to C9X Draft 6 with rationale interspersed appropriately.

Changes

The changes in this document are diff-marked against the previous version. There are some wording changes as a result of action items from the Toronto meeting. The words ‘‘If the size of the array is not needed, it is unspecified whether the size expression is evaluated’’ were added to **6.5.4.2**. There was some concern about the implications of the word *needed* but in this case it shouldn't be a problem because the most liberal intepretation is permitted. Essentially it is unwise to count on side-effects in VLA size expressions. However, sometimes a function call is needed for this purpose and should not be unilaterally dismissed. This change also eliminated the need for the last sentence in the `sizeof` section that indicates VLA expressions do not always need to be evaluated.

Section **6.5** was changed to read ‘‘nested sequence of declarators in a full declarator’’ to correctly indicate where array types appear in a type. This sentence was moved to section **6.5.4** to be with the ‘‘full declarator’’ definition.

The constrain in **6.5.2.1** was turned into a footnote.

Evaluation of section **6.5.6** did not yield any insights into how to improve the wording of the second sentence in the semantics part.

6.1.2.4 Storage durations 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), *variable length array* (6.5.4.2).

6.1.2.6 Compatible type and composite type

A *composite type* can be constructed from two types that are compatible; it is a type that is compatible with both of the two types and satisfies the following conditions:

— If one type is an array of ***known*** constant size, the composite type is an array of that size; ***otherwise, if one type is a*** *variable length array* ***the composite type is that type.***

Forward references: *variable length array* (6.5.4.2).

6.3 Expressions

6.3.3.4 The `sizeof` operator

Semantics

The **`sizeof`** operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type. The size is determined from the type of the operand. The result is an integer. ***If the type of the operand is not a*** *variable length array* ***type, the operand is not evaluated, and the result is an integer constant.*** *

Forward reference: declarators (6.5.4), *variable length array* (6.5.4.2).

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);              /* execution time expression */
}

main() {
   koo = func(10);                /* execution time 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, this is no longer the case since execution time 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 can now produce a value computed at execution time, there still exists a consistency when applied to the previously mentioned operations. Furthermore, it can still be used in an argument to the **malloc** function and to compute the number of elements in an array.

6.3.6 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 an array of known constant size, and pointer **p** is declared to be a pointer to an array of the same known constant size that points to **a**, the computed results are still the same.

6.4 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*** *variable length array* ***type or a parenthesized name of such a type,*** and floating constants that are the immediate operands of casts. Cast operators in an integral constant expression shall only convert arithmetic types to integral types, except as part of an operand to the **`sizeof`** operator.

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*** *variable length array* ***type or a parenthesized name of such a type.*** Cast operators in an arithmetic constant expression shall only convert arithmetic types to arithmetic types, except as part of an operand to the **`sizeof`** operator.

Note: footnote 58 states that the size of an array must be a constant expression.

6.5 Declarations

6.5.2 Type specifiers *

Constraints

Only ordinary identifiers (as defined in **6.1.2.3**) with block scope or function prototype scope and without linkage can have a *variably modified* 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: *variably modified* (6.5.4), *variable length array* (6.5.4.2).

ISSUES

Overview:

This section discusses three issues about declarations containing *variable length array* specifiers. 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 a *variable length 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 t { 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 */
                                   /* array size m evaluated now */
   struct tag {
      int (*y)[n];                 /* Error - y is not an ordinary identifier */
      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 Structure and union specifiers

Add the following footnote:

A structure or union shall not contain a *variably modified* member because member names are not ordinary identifiers (as defined in **6.1.2.3**).

ISSUES

Overview:

This issue involves the declaration of members of structures or unions with *variably modified* types. Allowing structure members to have a VLA type introduces a host of problems, such as the treatment when passing these objects (or even pointers to these objects) as parameters. In addition, the semantics of the `offsetof` macro would need to be extended 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? */
}
```

Because the expansion of **offsetof** is implementation specific, it seems impractical to guarantee any behavior for members which 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) is 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 Declarators

Semantics

Each declarator declares one identifier, and asserts that when an operand of the same form as the declarator appears in an expression, it designates a function or object with the scope, storage duration, and type indicated by the declaration specifiers.

A** full declarator* ***is a declarator that is not part of another declarator. The end of a full declarator is a sequence point. If the nested sequence of declarators in a full declarator contains a variable length array type, the type specified by the full declarator is said to be *variably modified.*

In the following subsections, consider a declaration

```
T D1
```

where `T` contains the declaration specifiers that specify a type *T* (such as `int`) and `D1` is a declarator that contains an identifier *ident.* The type specified for the identifier *ident* in the various forms of declarator is described inductively using this notation.

If, in the declaration ‘‘`T D1`,’’ `D1` has the form

identifier

then the type specified for *ident* is *T.*

If, in the declaration ‘‘`T D1`,’’ `D1` has the form

```
( D )
```

then *ident* has the type specified by the declaration ‘‘`T D`.’’ Thus, a declarator in parentheses is identical to the unparenthesized declarator, but the binding of complex declarators may be altered by parentheses.

6.5.4.2 Array declarators

Constraints

The `[` and `]` may delimit an expression or `*`. If `[` and `]` delimit an expression (which specifies the size of an array), it shall have 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* 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 known constant size, the array type is not a*** *variable length array* ***type. Otherwise, the array type is a*** *variable length array* ***type. If the size expression is not a constant expression, and it is evaluated at program execution time, it shall evaluate to a value greater than zero. If the size of the array is not needed, it is unspecified whether the size expression is evaluated. The size of each instance of a*** *variable length array* ***type does***

not change during its lifetime.

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 the two array types are used in a context which requires them to be compatible, it is undefined behavior if the two size specifiers evaluate to unequal values.***

Example 6

```
extern int n;
extern int m;

func() {
   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, but undefined behavior unless n == 6 and m == n+1 */
}
```

Example 7

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

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

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 may include accessing volatile objects there seems to be no compelling reason to forbid side effects in *variable length array* size specifiers.

Rationale:

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

6.5.4.3 Function declarators (including prototypes)

Semantics

A parameter type list specifies the types of, and may declare identifiers for, the parameters of the function. ***A declared parameter that is a member of the parameter type list that is not part of a function definition, may use the `[*]` notation in its sequence of declarator specifiers to specify a*** *variable length array* ***type.***

Example 8

```
/* The 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 declaration and function 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 following example shows how to declare parameters in any order and avoid lexical ordering issues.

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 rules for parameters inside function prototype declarators with *variable length array* types that do not use the **[*]** syntax are the same whether or not the function declarator is part of a definition, any identifier used to specify the size of these *variable length array* types needs to be visible.

```
/* prototype declarator with an incorrect parameter type */
/* Error - x and y are never declared */

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

This is the primary reason behind providing the **[*]** syntax.

```
void f1(double a[*][*]);  /* OK */
```

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 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 Type definitions

Constraints

If a typedef name specifies a *variably modified* type then it shall have block scope.

Semantics

In a declaration whose storage-class specifier is **typedef**, each declarator defines an identifier to be a typedef name that specifies the type specified for the identifier in the way described in **6.5.4**. ***Any array size expressions associated with*** *variable length array* ***declarators shall be evaluated with the typedef name at the beginning of its scope upon each normal entry to the block.***

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 determines the size of 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 a decision was made to force evaluation of the expression at the time the type definition itself is declared.

6.5.7 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.***

Remove footnote 78 from 6.5.7.

Example 18

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

6.6 Statements

Semantics

A *full expression* is an expression that is not part of another expression. Each of the following is a full expression: 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 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**). The initializers of objects that have automatic storage duration, ***and the*** *variable length array* ***declarators of ordinary identifiers with block scope*** 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++];      /* undefined behavior */
   int b[m++], c[m++];   /* well defined behavior */
}
```

Rationale:

Since sequence points cause the side effects of previous evaluations to be complete, it seems reasonable to add a sequence point after every *variably modified* full declarator (see **6.5.4** for full declarator definition). 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 full 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 The `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 that block 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 required.

6.6.6.1 The `goto` statement

Constraints

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 that block 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 incorrect 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 required.

6.7.1 Function definitions

On entry to the function ***all size expressions of its*** *variably modified* ***parameters are evaluated, and*** the value of each argument expression shall be converted to the type of its corresponding parameter, as if by assignment to the parameter.

7.6.2.1 The `longjmp` function

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>
jmp_buf buf;
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];          /* a may remain allocated */
   h(n);
}

void h(int n) {
   int b[n];          /* b may remain allocated */
   longjmp(buf,2);    /* might cause some memory to be lost */
}
```

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:

Essentially this behavior is allowed in a program (just like a **`longjmp`** invocation that bypasses a call to the **`free`** routine). These semantics do ***not*** prevent storage from being lost by the execution environment in a conforming program.

A Language Syntax Summary

A.1.2.2 Declarations

(6.5.4) 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) direct-abstract-declarator:

(*abstract-declarator* **)**

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

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

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