Extending C with Arrays of Variable Length

Dr. Harry H. Cheng, Assistant Professor
Director of Integration Engineering Laboratory
Department of Mechanical & Aeronautical Engineering
and Graduate Group in Computer Science
University of California
Davis, CA 95616
Email: hhcheng@ucdavis.edu
World Wide Web: http://iel.ucdavis.edu

Abstract

Extending C with arrays of variable length is critical in evolving C as a primary scientific programming language. Arrays of variable length whose size is known only at program execution time are implemented in the C^H programming language. C^H is designed to be a superset of C. This paper describes the current implementation of arrays of variable length in the C^H programming language. It also makes comparison studies of variable length arrays in C^H with those implemented in GNU C compiler gcc and Cray Research standard C compiler SCC as well as other alternate proposals.

Keywords

C, variable length array, deferred-shape array, assumed-shape array, pointer to assumed-shape array.

1 Introduction

Arrays in C are intimately tied with pointers. Treating array variables as pointers in C is very elegant for system programming; it is one of C's major strengths. But, numerically oriented applications in science and engineering were not the original design goal of C as it is reflected in an assumption that "C provides rectangular multi-dimensional arrays, although in practice they are much less used than arrays of pointers" [1]. This assumption is true only when C is used for system programming because multi-dimensional arrays are essential for applications in science and engineering and they are not less frequently used than arrays of pointers at all. Because C was not designed for numerical computing, handling of multi-dimensional arrays in C is cumbersome in many situations. For example, in contrary to the fame of C for its conciseness and clarity, passing arrays of variable length to functions in C is neither intuitive nor easy to understand; it is very complicated [2]. Arrays of variable length are available in FORTRAN since its earlist days [3]. Scientific programmers with prior FORTRAN experiences are often disappointed at C's inability to handle arrays of variable length.

Adding variable length arrays (VLA) to C is a critical step in evolving C as a leading programming language for applications in science and engineering. This paper describes the current implementation of variable length arrays — deferred-shape arrays, assumed-shape arrays, and pointers to assumed-shape array — in the C^H programming language. C^H is designed to be a superset of C. Various extensions of C^H over C have been described in [4-9]. Although adding assumed-shape arrays to C is a conservative enhancement to the language, addition of a new feature into the standard needs a careful examination of its potential impact on the language as a whole. The new feature should be a natural extension to C, namely, in the so-called spirit of C; it must not break all currently existing codes. Therefore, the ISO section numbers [10], with which the C standard appears to be impacted, are listed under the section numbers in this paper. Arrays whose size is known only at runtime have also been implemented in the current GNU C and Cray Research's Standard C compilers gcc and SCC, respectively. These arrays of variable length have been proposed to the ANSI C Standard Committee X3J11 for consideration as possible new extensions to C [11, 12]. In addition, a conceptual fat pointer to variable length arrays has been proposed by Ritchie [13]. This paper will also compare the implementation of variable length arrays in the C$^{\mathrm{H}}$ programming language with these alternative approaches.

2 Storage Duration and Declaration of Arrays

(6.1.2.5)
An *array* consists of elements that extend in one or more dimensions to represent columns, planes, cubes, etc. The number of dimensions in an array is referred to as the *rank* of the array, the number of elements in a dimension is called the *extent* of the array in that dimension. The *shape* of an array is a vector where each element of the vector is the extent in the corresponding dimension of the array. The *size* of an array is the number of bytes used to store the total number of elements of the array.

2.1 Storage Duration of Objects

(6.1.2.4)
Storage duration determines the lifetime of an object. An array declared with external or internal linkage, or with the storage-class specifier `static` has *static storage duration.* For such an array, its storage is reserved and its stored value for each element is initialized once only. Each element of the array exists and retains its last-stored value throughout the execution of the entire program. The shape of the array with static storage duration has to be resolved before the execution of the function `main()`. Therefore, each extent of an array definition with static storage duration shall be an integral constant expression with a value greater than zero as shown in the following sample program.

```
int n = 5;
int a[4][5], aa[3] = {1,2,3};
extern int b[6][7], c[8], d[][9], e[]; /* d and e are incomplete */
/* complete shape for d and e in the following definition */
int  b[6][7], c[8], d[4][9], e[10], ee[2][3] = {1,2,3,4,5,6};
main(){
  static int s[4], ss[2+3] ={1,2,3,4,5};
  extern int a[4][5];
  extern int b[6][7], c[8], d[][9], e[];
}
```

An array declared with no linkage and without the storage class specifier `static` within a function or nested function has *automatic storage duration.* Storage is guaranteed to be reserved for a new instance of such an array on each normal entry into the block with which it is associated. If an initialization is specified for the array with automatic storage duration, it is performed on each normal entry. Storage for the array is no longer guaranteed to be reserved when execution of the enlosing block ends in any way including by means of `goto`, `continue`, `break`, and `return` statements. For example, arrays in the following code fragment have automatic storage duration.

```
int n = 5;
void funct1(){
  int m = 6;
  int a[4][5], aa[3] = {1,n,m};      /* n==5, m==6 */
  void funct2(){
    int s[4], ss[2+3] ={1,2,3,n,m}; /* n==5, m==6 */
  }
}
```

In this sample code, the shape of an array is completely specified by constant integral expressions for each extent. Since memory space for an array of automatic storage duration is allocated at execution time upon the entry to the block within which it is declared. It is desirable that the length of the array can be different when the block or function is invoked each time. An array whose size is determined at program execution time is called a *variable length array* (VLA).

2.2 Declaration of Arrays

(6.5.4) (6.5.4.2)

Variables can be declared in the form of

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 delimiters [and] may delimit an expression or : for declaration of arrays.

If D1 has the form

D[$assignment\text{-}expression_{opt}$]

and the type specified for identifier in the declaration "T D1" is "*derived-declarator-type-list T*," then the type specified for *ident* is "*derived-declarator-type-list array of T.*" If the size is not present, the array type is an incomplete type as shown in the previous sample code. If delimiters [and] delimit an expression that specifies the size of an array, it shall be an integral type. If it is a constant integral expression, it shall have a value greater than zero. If the size expression of an array is not a constant expression, it is evaluated at program execution time and shall evaluate to a value greater than zero and the array type is a *deferred-shape array* type. If the size expression is an integral constant expression and the element type has a fixed size, the array type is a *fixed-length array* type.

If D1 has the form

D[:]

and the type specified for identifier in the declaration "T D1" is "*derived-declarator-type-list T*," then the type specified for *ident* is "*derived-declarator-type-list assumed-shape array of T.*" The array is called *assumed-shape array* type. The shape of the array is assumed at program execution time. The *variable length array* type includes assumed-shape and deferred-shape arrays as well as pointers to array of assumed-shape. The following example will clarify the concepts of these various array definitions.

```
void funct(int a[:][:], (*b)[:], c[], n, m){
/* a: assumed-shape array */
/* b: pointer to array of assumed-shape */
/* c: incomplete array completed by function call */
  int d[4][5];     /* d: fixed-length array */
  int e[n][m];     /* e: deferred-shape array */
  int (*f)[:];     /* f: pointer to array of assumed-shape */
  extern int g[];  /* g: incomplete array completed by external linkage */
  int h[] = {1,2}; /* h: incomplete array completed by initialization */
}
```

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

Details related to variable length arrays will be discussed in the following sections.

3 Deferred-Shape Arrays

3.1 Constraints and Semantics

(6.5.2) (6.3.6) (6.6.2)

The size of a deferred-shape array type is obtained at program execution time and the value of the size shall be greater than zero. The size of a deferred-shape array type shall not change until the execution of the block containing the declaration has ended. Therefore, at least one of the size expression is a non-constant integral expression for deferred-shape arrays. The variables used in the size expression must be declared beforehand. For example, arrays a, b, c, d, and e in the following code fragment are valid declaration of deferred-shape arrays whereas arrays f, g, and h are not.

```
int N1;
extern int N;
void funct1(int n, m){
  int i = 8*n;
  int j = 0, k = -9;
  int a[i][4];                  /* OK */
  int b[3][m];                  /* OK: mix fixed-extent with deferred-extent */
  int c[n*m][n];                /* OK */
  int d[funct2(n)][3*funct2(i)]; /* OK */
  int e[N][N1*n];               /* OK */
  int f[M];          /* ERROR: M has not been defined yet */
  int g[j], gg[0];   /* ERROR: zero size */
  int h[k], hh[-9];  /* ERROR: negative size */
}
int funct2(int i)
{ return i*i;}
int N, M;            /* define N and M */
```

Deferred-shape arrays shall be declared in block scope such as variables inside functions and nested functions. Arrays declared with the `static` storage class specifier in block scope shall not be declared as deferred-shape arrays. The behavior for declarations of deferred-shape arrays with file or program scope is undefined. For example,

```
#include <stdio.h>
void funct1(int n, m){
  int funct2(int n, i){
    int a[n][i];          /* OK */
    int b[n];             /* OK */
    return n+m;
  }
  int b[funct2(n,m)][printf("%d\n",n)];  /* OK */
}
extern int n;
int a[n][n];              /* UNDEFINED: not block scope */
static int b[n][n];       /* UNDEFINED: not block scope */
extern int c[n][n];       /* UNDEFINED: not block scope */
int d[2+3][90];           /* OK */
void funct3(int i){
  extern int a[n][n];     /* UNDEFINED: a has linkage */
  static int b[n][n];     /* ERROR: b is static identifier */
  int c[i+3][abs(i)];     /* OK */
}
```

The initializers of objects that have static storage duration are evaluated and the results are stored to objects at compilation time. But, the initializers of objects with automatic storage duration and size expression of deferred-shape arrays are evaluated and values are stored in the object at program execution time. For example,

```
#include <stdio.h>
int n = 4;                /* compile time n==4 */
main(){
  int m = 5;              /* runtime m == 5 */
  int a[n++][n++];        /* order of evaluation is undefined */
```

```
  int b[n++], c[n++];    /* order of evaluation is undefined */
  int d[n++]; int e[n++];/* order of evaluation is defined */
  printf("%d %d %d", n--, b[n--], c[n--]); /* order of evaluation is undefined */
}
```

Since the size of a deferred-shape array is unknown until the execution time. The size of the deferred-shape array often time is different at each invocation. Therefore, the deferred-shape array shall not be initialized. For example,

```
void funct1(int n){
  int a[3] = {1,2,3};              /* OK */
  int b[ ] = {1,2,3};              /* OK */
  int c[2][3] = {{1,2,3},{4,5,6}}; /* OK */
  int d[ ][3] = {{1,2,3},{4,5,6}}; /* OK */
  int e[n] = {1,2};                /* ERROR: initialization */
  int f[n][n] = {1,2,4,5};         /* ERROR: initialization */
}
```

Pointers to deferred-shape array shall not be declared. For example,

```
void funct(int n){
  int (*p1)[3];      /* OK: pointer to fixed-length array */
  int (*p2)[n];      /* ERROR: pointer to deferred-shape array */
  int (*p3)[n][3];   /* ERROR: pointer to deferred-shape array */
  int (*p3)[3][n];   /* ERROR: pointer to deferred-shape array */
}
```

Deferred-shape arrays shall not be declared at the function prototype scope. For example,

```
void funct1(int n, a[n]); /* ERROR: deferred-shape array a as function parameter */
void funct2(int a[n], n); /* ERROR: n is undefined in a[n] */
                          /* ERROR: deferred-shape array a as function parameter */
void funct1(int n, a[n])  /* ERROR: deferred-shape array a as function parameter */
{ }
void funct2(int a[n], n)  /* ERROR: n is undefined in a[n] */
{ }                       /* ERROR: deferred-shape array a as function parameter */
```

Deferred-shape shall not mix with incomplete array type. For example,

```
int n;
int a[][n] = {{1,2},{3,4}}; /* ERROR: initialization */
void funct(int n, b[][n]);  /* ERROR: function prototype scope */
extern c[][n];              /* ERROR: static storage duration */
```

For two array types to be compatible, both shall have compatible element types and the same shape. For example,

```
void funct1(int (*p)[4]])
{int i = sizeof(p);}        /* i == 4 */
void funct2(int p[3][4])
{int i = sizeof(p);}        /* i == 4 */
void funct3(int p[ ][4])
{int i = sizeof(p);}        /* i == 4 */
void funct4(int n)
{
  int i = 3, j = 4;
  int (*p)[4];
  int a[i][j];
```

```
  int b[j][j];
  int c[i][i];
  p = a; funct1(a); funct2(a); funct3(a); /* compatible */
  p = b; funct1(b); funct2(b); funct3(b); /* compatible */
  p = c; funct1(c); funct2(c); funct3(c); /* incompatible */
}
```

3.2 Deferred-Shape Arrays Related to Switch Statement

(6.6) (6.6.4.2)
The controlling expression for a `switch` statement 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 if it contains the declaration of a deferred-shape array. Otherwise, the memory for the deferred-shape array within the block will not be allocated. For example,

```
int i;
main(){
  int n = 10;
  switch (n){
    int a[n]; /* ERROR: bypass declaration of a[n] */
    case 10:
      a[0] = 1;
      break;
    case 20:
      a[1] = 2;
      break;
    case 30:
      {
        int b[n]; /* OK */
        b[1] = 90;
      }
      break;
  }
}
```

3.3 Deferred-Shape Arrays Related to Goto Statement

(6.6.4.2)
Similarly, the identifier in a `goto` statement shall name a label located somewhere in the enclosing or its calling 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 if it contains the declaration of a deferred-shape array. For example,

```
void funct(int n){
  int i;
label1:
  if(n>10)
    goto label2;      /* ERROR: bypass declaration of a[n] */
  {
    int a[n];
    a[i] = 8;
label2:
    a[i] = 9;
    goto label1;      /* OK */
label3:
    a[i] = 10;
```

```
      goto label2:        /* OK */
    }

  void funct1(int m){
    void funct2(int r){
      if(r)
        goto label4;      /* OK */
      else
        goto label5;      /* ERROR: bypass declaration of b[m] */
    }
  label4:
    {
      int b[m];
  label5:
      a[0] = 9;
      goto label5;       /* OK */
    }
  }
```

When a goto statement transfers the program execution flow from a nested function to its parent function, it shall terminate execution of the active function invocation. All dynamically allocated memory including those for deferred-shape arrays shall be deallocated and the previous calling environment shall be restored. The function that called the function containing the goto statement once again becomes the active function. If the label named in the goto statement is not in the now-active function, the deactivation of the current function and activation of its parent function continue. Eventually, the function containing the label of the goto statement will be active and control flow will be transferred to the statement with the proper label. For example,

```
void funct1(int n){
  local void funct2(int n);
  local void funct3(int n);
  int a[n];
label:
  funct2(n);
  void funct2(int n){
    int b[n];
    funct3(n);
  }
  void funct3(int n){
    int c[n];
    goto label; /* b[n] and c[n] will be deallocated*/
  }
}
```

In this example, memory allocated for deferred-shape arrays b[n] and c[n] will be deallocated when the control flow is transferred from function funct3() to function funct1() through function funct2(). If *label* and *goto label* statements in the above example were replaced by functions setjmp(buf) and longjmp(buf), respectively, the memory of deferred-shape arrays may not be deallocated. With nested functions, functions setjmp(buf) and longjmp(buf) may become good candidates for obsolete features.

3.4 Deferred-Shape Arrays as Members of Structures and Unions

(6.5.2.1) (7.1.6)

Not only ordinary identifiers, but also members of structures and unions, can be declared as deferred-shape arrays. But, structures and unions with members of deferred-shape arrays shall be declared with automatic storage duration. The behavior for declaration of structures and unions with members of deferred-shape array at file or program scope is undefined. Structures declared with the `static` storage specifier in block scope shall not be declared with members of deferred-shape arrays.

Like `sizeof`, `offsetof` is also a built-in operator. If a structure has no member of deferred-shape array, the operation `offsetof(`*type, member-designator)* evaluates to an integral constant value that has type `size_t`, the value of which is the offset in bytes, to the structure member (designated by *member-designator*), from the beginning of the structure (designated by *type*). If the structure contains a member of deferred-shape array, the result is not a constant expression and is computed at program execution time. Because of variable length of deferred-shape arrays, given

`static` *type* t;

the expression `&(t.`*member-designator)* will not evaluate to an address constant if the structure contains a deferred-shape array.

Structures and unions shall not be defined at the function prototype scope. Structures and unions with members of deferred-shape array can be declared at the function prototype scope of nested functions. For example,

```
int n;
struct tag{
  int m;
  int a[n];            /* UNDEFINED: not block scope for tag1 */
  int b[m];            /* UNDEFINED: not block scope for tag1 */
};
void funct1(int m){
  int l;
  static struct tag{
    int m;
    int a[n];          /* ERROR: static block scope for tag1 */
    int b[m];          /* ERROR: static block scope for tag1 */
  };
  struct tag1{         /* structure shared by
                          funct1(), funct2(), and funct3() */
    int r=2*m;         /* initialization of member r */
    int a[n][m][l];                 /* OK */
    int q=2+l, q2;     /* initialization of member q */
    int b[r][q];                    /* OK */
  };
  void funct2(){
     struct tag1 s1;                /* OK */
     int i;
     i = offsetof(struct tag1, r); /* OK: runtime offsetof() */
     i = offsetof(struct tag1, a); /* OK: runtime offsetof() */
     i = offsetof(struct tag1, b); /* OK: runtime offsetof() */
  }
  /* structure with deferred-shape array as function arg */
  void funct3(struct tag1 s){
    int i, j;
    struct tag1 s1;                 /* OK */
    for(i=0; i<s.r; i++)
      for(j=0; j<s.q; j++)
        s1.b[i][j] = s.b[i][j];
  }
  struct tag2{
    int a[2][3];
    int b[4][5];
  };
```

```
    l = offsetof(struct tag1, r);    /* OK: runtime offsetof() */
    l = offsetof(struct tag1, a);    /* OK: runtime offsetof() */
    l = offsetof(struct tag1, b);    /* OK: runtime offsetof() */
    l = offsetof(struct tag2, a);    /* OK: compile time offsetof() */
    l = offsetof(struct tag2, b);    /* OK: compile time offsetof() */
  }
```

3.5 Sizeof

(6.3.3.4)

When the built-in `sizeof` operator is applied to an operand that has array type, the result is the total number of bytes allocated for storing the elements of the array. For deferred-shape array, the result is not a constant expression and is computed at program execution time. For example,

```
int funct(int n, m){
  int i;
  int a[3][4];
  int b[n][m];
  int c[sizeof(a)];         /* c is fixed-length array */
  int d[sizeof(b)];         /* d is deferred-shape array */
  i = sizeof(a);            /* compile time sizeof(a) is 48 */
  j = sizeof(b);            /* runtime sizeof(b) is nxmx4 */
  return j;
}
```

When the `sizeof` operand is applied to an operand that has structure or union type, the result is the total number of bytes in such object, including internal and trailing padding. If any member of a structure or union is a deferred-shape array, the result is not a constant expression and it is computed at program execution time. For example,

```
int n
int funct1(int m){
  int l;
  struct tag1{
    int a[2][3];
  };
  struct tag2{
    int r;
    int a[4][5];
    int b[n][m][l][r];
  };
  int i;
  struct tag1 s1;
  struct tag2 s2;
  void funct2(struct tag1 s1, struct tag2 s2){
    int i;
    i = sizeof(s1);         /* compile time sizeof() */
    i = sizeof(s1.a);       /* compile time sizeof() */
    i = sizeof(s2.a);       /* compile time sizeof() */
    i = sizeof(s2);         /* runtime sizeof() */
    i = sizeof(s2.b);       /* runtime sizeof() */
  }
  i = sizeof(struct tag1); /* compile time sizeof() */
  i = sizeof(s1);          /* compile time sizeof() */
  i = sizeof(s1.a);        /* compile time sizeof() */
  i = sizeof(s2.a);        /* compile time sizeof() */
```

```
  i = sizeof(struct tag2); /* runtime sizeof() */
  i = sizeof(s2);          /* runtime sizeof() */
  i = sizeof(s2.b);        /* runtime sizeof() */
}
```

3.6 Typedef

(6.5.6)

Typedef declarations that specify an aggregate type with a deferred-shape array shall have block scope. The behavior for typedef declaration with deferred-shape arrays in file or program scope is undefined. The deferred-shape of the array shall be evaluated not at the time the type definition is declared, but at the time it is used as a type specifier in an actual declarator. For example,

```
int n = 5;
typedef int A[n];                  /* UNDEFINED: not block scope */
typedef struct tag{int aa[n]} TAG1;/* UNDEFINED: not block scope */
main(){
  int n;
  typedef int B[n];                /* OK */
  B bb;                            /* OK: int bb[n] */
  B *cc;                           /* ERROR: int (*cc)[n] */
}
void funct(int m){
  typedef int A[m];                /* m is not stored in A */
  typedef struct tag {
    int b[m];                      /* store m in b[m] */
    struct tag *prev;
    struct tag *next;
} TAG1;
A d;                               /* store m in d */
m++;                               /* increment m */
{
   A a;             /* a[] has one more element than d[] */
   TAG1 s;          /* s.b[] has one more element than d[] */
   int c[m];        /* c[] has one more element than d[] */
}
}
funct(6);           /* ==> d[6], a[7], s.b[7], c[7] */
```

3.7 Other Data Types and Pointer Arithmetic

Deferred-shape arrays of different data type can be declared in the same manner as fixed-length arrays. For example,

```
void funct(int n){
  char c[n], *cp[n];
  int *ip[n][n];
  float f[n], **fp[n][n];
  double d[n], *dp[n][n];
  complex z[n], *zp[n][n];
}
```

The pointer arithmetic related to fixed-length arrays is still valid for deferred-shape arrays. For example,

```
void funct(int n, m){
  int i, j;
```

```
    int a[n][m];
    a[i][j] = 90;
    *(a[i]+j) = 90;          /* a[i][j] = 90 */
    *(*(a+i)+j) = 90;        /* a[i][j] = 90 */
    *(&a[0][0]+i*m+j) = 90; /* a[i][j] = 90 */
    *((int *)a[i]+j) = 90;  /* a[i][j] = 90 */
    *((int *)(a+i)+j) = 90; /* a[i][j] = 90 */
    *((int *)a+i*m+j) = 90; /* a[i][j] = 90 */
    i = a[n-1] - a[n-2];    /* i == m */
  }
```

4 Assumed-Shape Arrays

4.1 Constraints and Semantics

(6.3.6) (6.5.2) (6.6.2)

Assumed-shape arrays shall be declared at the function prototype scope or in a typedef declaration. The assumed-shape array is a formal argument which takes the shape of the actual argument passed to it. That is, the arrays for actual and formal arguments have the same rank and the same extent in each dimension. The shape of assumed-shape arrays cannot be determined until execution time. The rank of an assumed-shape array is equal to the number of colons in the assumed-shape specification. For example,

```
  void funct(int [:], [:][:])      / * OK */
  void funct(int dummy1[:], dummy2[:][:]) / * OK */
  void funct(int a[:], b[:][:])    / * OK */
  int A[:];                        /* ERROR: not function prototype scope */
  static int B[:][:];              /* ERROR: not function prototype scope  */
  extern C[:][:];                  /* ERROR: not function prototype scope  */
  void funct(int a[:], b[:][:]){   / * OK */
    int c[:][:];                   /* ERROR: not function prototype scope   */
    extern int A[:];               /* ERROR: not function prototype scope   */
    void funct2(int a[:], b[:][:]){/* OK */
       int c[:][:];                /* ERROR: not function prototype scope   */
    }
    funct2(a, b);                  /* OK */
  }
```

Assumed-shape array may also appear in a typedef declaration. For example,

```
  typedef int A[:];
  A a;                /* ERROR: not function prototype scope */
  void funct(A a);    /* OK */
```

Only variables of fixed-length, deferred-shape, or assumed-shape array type can be used as an actual argument of a formal argument of assumed-shape array type in function parameters. A pointer or pointer to array, which does not have the complete shape information, shall not be used as an actual argument of a formal argument of assumed-shape array type. For example,

```
  funct1(int a[:][:]){
    int n=a[1][1], m = a[1][2];
    int b[3][4];
    int c[n][m];
    int *p1, (*p2)[4], (*p3)[:];
    void funct3(int a[:][:]);
    void funct2(int a[:][:])
    { }
```

```
    funct2(a);  funct3(a);  /* OK a is assumed-shape array */
    funct2(b);  funct3(b);  /* OK b is fixed-length array */
    funct2(c);  funct3(c);  /* OK c is deferred-shape array */
    funct2(p1); funct3(p1); /* ERROR: p1 is pointer */
    funct2(p2); funct3(p2); /* ERROR: p2 is pointer to array of fixed-length */
    funct2(p3); funct3(p3); /* ERROR: p3 is pointer to array of assumed-shape */
  }
  void funct3(int a[:][:])
  { }
```

Although complete arrays can be extracted from a pointer to array, they shall not be used as actual arguments of an assumed-shape array. For example,

```
  void funct1(int a[3]);
  void funct2(int a[5][7]);
  void funct11(int a[:]);
  void funct22(int a[:][:]);
  void funct3(int p2[][5][7]){
    int a[5][3];
    int (*p1)[3];
    p1 = a;
    funct1(p1[0]);    /* OK: passed a[0][0], ..., a[0][2] */
    funct1(p1[1]);    /* OK: passed a[1][0], ..., a[1][2] */
    funct1(*(p1+1)); /* OK: passed a[1][0], ..., a[1][2] */
    funct1(a[4]);     /* OK: passed a[4][0], ..., a[4][2] */
    funct1(p1+1);     /* OK: p1+1 is a pointer to array of 3 ints */
    funct2(p2[1]);    /* OK: passed p2[1][0][0], ..., p2[1][4][6] */

    funct11(p1[0]);   /* ERROR: passing array a[0][0], ..., a[0][2]  */
    funct11(p1[1]);   /* ERROR: passing array a[1][0], ..., a[1][2] */
    funct11(*(p1+1));/* ERROR: passing array a[1][0], ..., a[1][2] */
    funct11(a[4]);    /* ERROR: passing array a[4][0], ..., a[4][2] */
    funct11(p1+1);    /* ERROR: p1+1 is a pointer to array of 3 floats */
    funct22(p2[1]);   /* ERROR: passing array p2[1][0][0], ..., p2[1][4][6] */
  }
```

where `p1[0]`, `p1[1]`, `*(p1+1)`, and `a[4]` are arrays with size of 12 bytes, `p2[1]` is an array of 140 bytes, and `p+1` is a pointer to array with size of 4 bytes.

Assumed-shape array shall not mix with fixed-length or incomplete array type. For example,

```
  void funct(int a[:][3]); /* ERROR: mix assumed-shape with fixed-length */
  void funct(int a[3][:]); /* ERROR: mix assumed-shape with fixed-length */
  void funct(int a[n][:]); /* ERROR: mix assumed-shape with deferred-shape */
  void funct(int a[:][n]); /* ERROR: mix assumed-shape with deferred-shape */
  void funct(int a[ ][:]); /* ERROR: mix assumed-shape with incomplete */
```

If the operand of a polymorphic operation or function is an element of an assumed-shape array, the data type of the result and operation depend on the data type of the formal argument. However, if the formal and actual data types of an argument are different, but they are compatible, the operand will be cast to an operand with data type of the formal argument before operation takes place. If an element is used as an lvalue, the rvalue is cast to the data type of the actual argument if they are different. In other words, elements of the actual array are coerced to the data type of the assumed-shape array at program execution time when they are fetched whereas they are coerced to data type of the actual argument when they are stored. For example,

```
  float A[3] = {1, 2};
```

```
complex Z[3] = {complex(1,0), complex(2,0)};
void funct(float a[:], complex z[:]){
  a[2] = a[0] + a[1];      /* addition of floats */
  z[2] = z[0] + z[1];      /* addition of complexs */
}
funct(Z, A);               /* A[2]==3.0, Z[2]=3.0+i0.0 */
```

If the formal argument is an assumed shape, the actual argument can also be an assumed-shape array. For example,

```
void funct2(complex aa[:], b[:][:], (*c)[6], d[][6], e[4][6]){
  aa[1] = b[1][2];
}
void funct1(complex a[:], b[:][:]){
  if(real(a[1]) == 0)
  funct2(a,b,b,b,b);       /* a and b are assumed-shape arrays */
}
main(){
  complex  A[2], B[4][6];
  funct1(A,B);             /* A and B are fixed-length arrays */
}
```

When the function `funct2()` is invoked by the function call of `funct2(a,b,b,b,b)`, the memory allocated for array `A` in the main routine is used by the assumed-shape array `a` in the function `funct1()`, and subsequently it is passed to the assumed-shape array `aa` in the function `funct2()`. An assumed-shape array can also be used as the actual argument of a pointer to fixed-length array in a function. In the above example, the memory allocated for array `B` in the main routine is used as `b` in the function `funct1()` and as `b, c, d, e` in the function `funct2()`. Different identifiers `a` and `aa` are used for the same array object allocated at the declaration of array `A`. But, the same identifier `b` has been used in both functions `funct1()` and `funct2()` for the array object `B`. This shows that the names of identifiers are irrelevant to argument association of functions.

4.2 Sizeof

(6.3.3.4)
When the operand of **sizeof**() operation is an assumed-shape array type, the result is the total number of bytes used to store elements of the array computed at program execution time. Furthermore, since arrays of different data types can be passed to assumed-shape arrays, the size of an element of an assumed-shape array will also be computed at program execution time. For example,

```
int funct(complex z[:]){
  int i, numOfElement;
  numOfElement = sizeof(z)/sizeof(z[0]);
  return numOfElement; /* sizeof(z)=80, sizeof(z[0])=4 */
}
main(){
  int num;
  float a[20];
  num = funct(a);      /* num == 20 */
}
```

4.3 Other Data Types and Pointer Arithmetic

Assumed-shape arrays of other data types are handled in the same manner as assumed-shape arrays of ints. For example, the following statement declares that variables `a, b`, and `c` are assumed-shape complex arrays of rank one, two, and three, respectively.

```
int funct(complex a[:], b[:][:], c[:][:][:]);
```

Assumed-shape arrays of different data types can be handled in the same manner. For example, in the following code fragment

```
char *cc[10]; float **ff[2][4]; double ***dd[3][5][7];
int funct(char *c[:]; float **f[:][:], double ***d[:][:][:]);
funct(cc, ff, dd);
```

the function prototype

```
int funct(char *c[:], float **f[:][:], double ***d[:][:][:]);
```

defines variables `c, f`, and `d` as the rank-one assumed-shape array of pointer to char, rank-two assumed-shape array of double pointer to float, rank-three assumed-shape array of triple pointer to double, respectively. Arrays `cc, ff`, and `dd` are passed to the formal assumed-shape arrays `c, f`, and `d` in the function `funct()`, respectively. The assumed-shape arrays in a function are handled in the same manner as fixed-length arrays. For example,

```
  void funct(complex z1[:], z2[:][:]){
    complex z, *zp, **zp2;
    zptr = z1;              /* the address of the array */
    zptr = &z1[2];          /* the address of the third element */
    /* z2[2][1] -= 1; z1[1] = z2[2][1] + z1[2]; z1[2] += 1; */
    z1[1] = --z2[2][1]+z1[2]++;
    z = *z1;                /* z = z1[0] */
    z = *(z1+5);            /* z = z1[5] */
    z = **z2;               /* z = z2[0][0] */
    zp = z2[2];             /* zp = &z2[2][0] */
    zp2 = (complex **)z2;
    /* z2[1][1] = z2[1][3] + z2[2][3] - z2[2][4]; */
    zp2[1][1] = z2[1][3]+ *(*(z2+2)+3) - *(4+*(z2+2));
    /* z2[2][3] = z2[1][3] + z2[2][3] */
    *(*(z2+2)+3) = z2[1][3]+  *(3+*(z2+2));
}
```

5 Pointers to Array of Assumed-Shape

5.1 Declaration

(6.1.2.5) (6.5.4.1) (6.5.5)

A *pointer type* may be derived from a function type, and object type, or an incomplete type, called the *reference type*. A pointer type describes an object whose value provides a reference to an entity of the reference type. A pointer type derived from the reference type *T* is sometimes called "pointer to *T*." The construction of a pointer type from a referenced type is called "pointer type construction."

If, in the declaration "T D1" described in section 2.2, D1 has the form

* *type-qualifier-list*$_{opt}$ 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* pointer to *T*." For each type qualifier in the list, *ident* is a so-qualified pointer.

Pointers to array of fixed-length is declared as

T (*D)[*assignment-expression*]

where T contains the declaration specifiers that specify a type and the assignment expression is an integral constant expression. For example,

```
int (*p1)[3];      /* p1 is pointer to array of 3 ints */
int *(*p2)[3];     /* p2 is pointer to array of 3 pointer to int */
int (*p3)[3][4];   /* p3 is pointer to 3x4 array of ints */
int *(*p4)[3][4];  /* p4 is pointer to 3x4 array of pointer to int */
```

Pointers to array of assumed-shape are declared as
T (*D)[:]
where T contains the declaration specifiers that specify a type. For example,

```
int (*p1)[:];      /* OK */
int (*p2)[:][:];   /* OK */
int *(*p3)[:][:];  /* OK */
int n = 8;
int (*p4)[3][:];   /* ERROR: mix fixed-length with assumed-shape */
int (*p5)[:][3];   /* ERROR: mix fixed-length with assumed-shape */
int (*p6)[n][:];   /* ERROR: mix deferred-shape with assumed-shape */
int (*p7)[:][n];   /* ERROR: mix deferred-shape with assumed-shape */
int (*p8)[ ][:];   /* ERROR: mix deferred-shape with incomplete type */
```

where `p1` is pointer to array of assumed-shape of rank 1 with int type, `p2` is pointer to array of assumed-shape of rank 2 with int type, and `p3` is pointer to array of assumed-shape of rank 2 with pointer to int type.

The shape of the array pointed to by a pointer to assumed-shape array is determined at program execution time. A pointer to assumed-shape array is sometimes called a *fat pointer* since it can store more information than a pointer to object of scalar type or a pointer to array of fixed-length at program execution time.

5.2 Constraints and Semantics

(6.2.2.3) (6.5.4.1)

Except for pointer to assumed-shape array type, 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, except pointer to assumed-shape array 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 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*. 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, called a *null pointer*, is guaranteed to compare unequal to a pointer to any object or function.

Two null pointers, converted through possibly different sequences of casts to pointer types, shall compare equal.

When a null pointer is converted to a pointer to array of assumed-shape, a null pointer is installed at the base pointer of the assumed-shape array and the bounds of the assumed-shape array are undefined.

An array, including fixed-length array, deferred-shape array, and assumed-shape array, may be converted to a pointer to assumed-shape array. A pointer to array of fixed-length or pointer to array of assumed-shape may also be converted to a pointer to assumed-shape array. The base pointer to array and all bounds are stored in the pointer to assumed-shape array. All any other pointer types that do not have the array shape information shall not be converted to a pointer to array of assumed-shape. For example,

```
void funct(int a[:][:], p1[2][4], (*p2)[4], p3[][4], n, m){
  int *p;
  int b[3][4];
  int c[n][m];
  int (*p4)[4];
  int (*p5)[:];
  int (*p6)[:];
  p6 = NULL;
```

```
  p6 = a;         /* OK: a is an assumed-shape array */
  p6 = b;         /* OK: b is a fixed-length array */
  p6 = c;         /* OK: c is a deferred-shape array */
  p6 = p1;        /* OK: p1 is a pointer to array of fixed-length */
  p6 = p2;        /* OK: p2 is a pointer to array of fixed-length */
  p6 = p3;        /* OK: p3 is a pointer to array of fixed-length */
  p6 = p4;        /* OK: p4 is a pointer to array of fixed-length */
  p6 = p5;        /* OK: p5 is a pointer to array of assumed-shape */
  p4 = p;         /* WARNING: array bounds do not match */
  p6 = p;         /* ERROR: p is not array type */
}
```

For two pointer types to be compatible, both shall be identically qualified and both shall be pointers to compatible types. For two pointers to fixed-length array to be compatible, both shapes of array pointed to by the pointer shall be the same. For two pointers to assumed-shape array to be compatible, both ranks of array pointed to by the pointer shall be the same and the shapes shall evaluate to the same value at program execution time. For example,

```
void funct(int a[:], b[:][:][:], (*p1)[4][5], p2[3], n, m){
  int c[3][4][5];
  int d[n][m][m];
  int (*p3)[4][5];
  int (*p4)[:];
  p4 = a;         /* ERROR: incompatible, wrong rank  */
  p4 = b;         /* ERROR: incompatible, wrong rank  */
  p4 = p1;        /* ERROR: incompatible, wrong rank  */
  p4 = p2;        /* ERROR: incompatible, wrong rank  */
  p4 = c;         /* ERROR: incompatible, wrong rank  */
  p4 = d;         /* ERROR: incompatible, wrong rank  */
  p3 = p4;        /* WARNING: incompatible, wrong rank */
}
```

When a pointer to array of assumed-shape is converted to any other pointer to object or to a scalar value, only the base pointer to assumed-shape array is used.

```
char c, *cp;
int i, *ip;
float f, *fp;
int (*ap)[4];
int (*p)[:];
c  = (char) p;        /* OK */
cp = (char *) p;      /* OK */
i  = (int) p;         /* OK */
ip = (int*) p;        /* OK */
ip = p;               /* OK */
f  = (float) p;       /* OK */
fp = (float*) p;      /* OK */
ap = p;               /* OK */
```

5.3 Function Prototype Scope

(6.5.4.3)

A pointer to assumed-shape array can be used as an argument parameter of a function to pass arrays of different size to the function. For example,

```
void funct(int (*)[:]);
```

```
void funct(int (*dummy)[:]);
void funct(int (*p)[:]);
int a[3][4], b[4][3];
int (*p1)[:];
funct(a,3,4);       /* passing fixed-length array a[3][4] */
funct(b,3,4);       /* passing fixed-length array b[4][3] */
p1 = a;
funct(p1,3,4);      /* passing fixed-length array a[3][4] */
funct(NULL,0,0);    /* passing NULL */
void funct(int (*p)[:], n, m){
  int i, j;
  int a[n][m];
  if(p == NULL)
    return;
  for(i=0; i<n; i++)
    for(i=0; i<m; i++)
      a[i][j] = p[i][j];
}
```

Arrays of deferred-shape and assumed-shape can also be passed to a pointer to array of assumed-shape. For example,

```
void funct1(int a[:][:], n, m){
  int b[n][m];
  void funct2(int (*p)[:])
  {
    int i, j;
    int c[n][m];
    if(p == NULL)
      return;
    for(i=0; i<n; i++)
      for(i=0; i<m; i++)
        c[i][j] = p[i][j];
  }
  funct2(a);  /* a is an assumed-shape array */
  funct2(b);  /* b is a deferred-shape array */
}
```

5.4 Typedef

(6.5.6)

The assumed-shape array and pointer to assumed-shape array are handled in the same manner as the fixed-length array and pointer to fixed-length array in typedef declarations. For example,

```
typedef int A[5];
typedef int B[:];
A a;               /* OK: int a[5] */
A *ap;             /* OK: int (*ap)[5] */
B b;               /* ERROR: not function prototype scope for 'int b[:]' */
B *bp;             /* OK: int (*bp)[:] */
/* void funct(int a[5], (*ap)[5], int b[:], (*bp)[:]); */
void funct(A a, *ap, B b, *bp); /* OK */
```

where `a` is an assumed-shape array and `ap` is a pointer to assumed-shape array.

5.5 Arrays Allocated by Dynamic Memory Allocation Functions

(6.3.4)
Arrays can be dynamically allocated as shown in the following example.

```
funct(int n, int m){
  double a[n][m];
  double (*p1)[:] = a;   /* OK p[i][j] = a[i][j] */
  double (*p2)[:] = (double [n][m])malloc(sizeof(double)*n*m); /* OK */
  double (*p3)[:] = (double [ ][m])malloc(sizeof(double)*n*m); /* OK */
  double (*p4)[:] = (double [][m])malloc(sizeof(double)*n*m);  /* OK */
  /* The following is an ERROR, pointer to deferred-shape array is not allowed */
  double (*p5)[:] = (double (*)[m])malloc(sizeof(double)*n*m); /* ERROR */
}
```

where `a` is a deferred-shape array, and `p1, p2`, and `p3` are pointers to assumed-shape array of double data type. All memories pointed to by `p1, p2, p3`, and `p4` are dynamically allocated. But, memories for `p2, p3,` and `p4` are obtained explicitly by the memory allocation function `malloc()`.

5.6 Similarities between Pointers to Fixed-Length Array and Pointers to Assumed-Shape Array

A pointer to array of assumed-shape behaves very much like a pointer to array of fixed-length. For example, as a pointer, it should be pointed to an object before its elements can be referenced. Some other points that may be not so straightforward at the first sight will be clarified in this section.

5.6.1 Static and Automatic Storage Duration

Unlike deferred-shape arrays, there is no restriction on the scope where a pointer to array of assumed-shape can be declared. It can be declared with either static storage duration or automatic storage duration. For example,

```
int (*p1)[:];
extern int (*p2)[:];
static int (*p3)[:];
main(){
  int (*p4)[:];
  static (*p5)[:];
  extern int (*p1)[:];
}
```

5.6.2 Initialization

A pointer to array of assumed-shape can be initialized at both compilation and program execution time. For example,

```
int a[3][4];
int (*p1)[:] = NULL;        /* runtime initialization */
extern int (*p2)[:];
static int (*p3)[:] = NULL; /* runtime initialization */
main(){
  int b[3][4];
  int (*p4)[:] = NULL;      /* compile time initialization */
  int (*p5)[:] = b;         /* compile time initialization */
  int (*p6)[:] = p1;        /* compile time initialization */
  static (*p7)[:] = a;      /* runtime initialization */
  static (*p8)[:] = p1;     /* runtime initialization */
```

```
  static (*p8)[:] = b;      /* ERROR: b is variable of auto class */
}
```

5.6.3 Members of Structures and Unions

Not only ordinary identifiers, but also members of structures and unions, can be declared as pointer to array of assumed-shape. For example,

```
struct tag1{
  int (*p1)[3];    /* pointer to fixed-length array */
  int (*p2)[:];    /* pointer to assumed-shape array */
};
main(){
  struct tag2{
    int (*p1)[3];  /* pointer to fixed-length array */
    int (*p2)[:];  /* pointer to assumed-shape array */
  };
}
```

where the structure `tag1` has static storage duration and structure `tag2` has automatic storage duration.

5.6.4 Sizeof

(6.3.3.4)

The size of a pointer to array of assumed-shape is the same as the size of a pointer to array of fixed-length. The size of a pointer to array of assumed-shape is the same as the size of the pointer to the data type of the array, which is evaluated at compile time. For example,

```
int (*a)[5];
int (*p1)[:];
main(){
  int (*b)[5];
  int (*p2)[:];
  void funct(int (*p3)[:], (*p4)[:][:])
  {
    int i;
    i = sizeof(a);  /* i == 4 */
    i = sizeof(b);  /* i == 4 */
    i = sizeof(p1); /* i == 4 */
    i = sizeof(p2); /* i == 4 */
    i = sizeof(p3); /* i == 4 */
    i = sizeof(p4); /* i == 4 */
  }
}
```

5.6.5 Other Data Types and Pointer Arithmetic

Pointers to array of assumed-shape of different data type can be declared in the same manner as pointers to array of assumed-shape of int. For example,

```
void funct(int n){
  char    (*cp1)[:], *(*cp2)[:], **(*cp3)[:];
  int     (*ip1)[:], *(*ip2)[:], **(*ip3)[:];
  float   (*fp1)[:], *(*fp2)[:], **(*fp3)[:];
  double  (*dp1)[:], *(*dp2)[:], **(*dp3)[:];
  complex (*zp1)[:], *(*zp2)[:], **(*zp3)[:];
}
```

The pointer arithmetic related to pointers to array of fixed-length is still valid for pointers to array of assumed-shape. For example,

```
main(){
  int i=2, j=3;
  int n=4, m=5;
  int a[4][5];
  int (*p)[:]
  p = a;
  p[i][j] = 90;           /* a[i][j] = 90 */
  *(p[i]+j) = 90;         /* a[i][j] = 90 */
  *(*(p+i)+j) = 90;       /* a[i][j] = 90 */
  *(&p[0][0]+i*m+j) = 90; /* a[i][j] = 90 */
  *((int *)p[i]+j) = 90;  /* a[i][j] = 90 */
  *((int *)(p+i)+j) = 90; /* a[i][j] = 90 */
  *((int *)p+i*m+j) = 90; /* a[i][j] = 90 */
  i = p[n-1] - p[n-2];    /* i == m */
}
```

6 Rationales

This section provides some rationales for features of VLAs implemented in the C^H programming language. The differences between VLAs in C^H and VLAs in other alternate proposals will be highlighted.

6.1 Deferred-Shape Arrays

Stallman and MacDonald pioneered the work on extending C with arrays of variable length [11, 12]. Many features of deferred-shape arrays described in section 3 are also implemented in GNU C compiler gcc and Cray Research standard C compiler SCC. Some linguistic features presented in section 3 are identical to MacDonald's proposal [12]. However, there are some differences between VLAs in C^H and VLAs in gcc and SCC. One of major differences is the behavior of the deferred-shape in the function prototype scope. Deferred-shape arrays are prohibited in the function prototype scope in C^H. Arrays of variable length are passed to functions by assumed-shape arrays or pointers to assumed-shape array in C^H.

The syntax of Stallman's proposal, based upon his implementation of GNU C compiler gcc, can be illustrated by passing two nxm arrays `a` and `b` and a nxr array `c` to a function as follows:

```
float a[3][5], b[3][5], c[3,7];
void funct(int n; int m; int r;
           float a[n][m], float b[n][m], float c[n][r],
           int n, int m, int r);
funct(a, b, c, 3, 5, 7);
```

In his proposal, each forward parameter declaration is followed by a semicolon. The presence of a following semicolon is to distinguish forward parameter declarations from real ones. Each parameter must have one and only one real declaration. The parameters for dimension of an array are defined before they are used in the array definition. The main drawbacks of this syntax are that the dimension parameters of an array are declared twice at the same lexical level and the number of arguments of the calling function is different from that in the function definition. The redundant declaration and unmatched number of arguments seem to be in conflict with the spirit of C.

By far the biggest controversy of MacDonald's proposal, based upon his implementation of Cray Research standard C compiler SCC, is the lexical ordering in declaration of arrays in the function definition and prototypes. MacDonald proposed two formats for passing arrays of variable length to functions. The syntax of the proposal can be illustrated by passing three arrays of `a[n][m]`, `b[n][m]`, and `c[n][r]` to a function. The first syntax is as follows:

```
float a[3][5], b[3][5], c[3,7];
void funct(int n, int m, int r, float a[n][m], float b[n][m], float c[n][r]);
funct(a, b, c, 3, 5, 7);
```

It should be pointed out that the same syntax has also been implemented in GNU gcc. However, to ease the type checking for a program with files compiled separately, it is further specified in MacDonald's proposal that the parameter in the argument for a dimension of an array can be replaced by the symbol "$*$" in function prototypes and the first size expression can be incomplete type. For example, a function prototype can be declared as follows:

```
void funct(int n, int m, int r, float a[][m], float b[*][*], float c[][*]);
```

Using the symbol "$*$" in an array dimension argument can overcome the problem when the parameters such as `n` and `m` are also predefined constants.

To design a language suitable for scientific numerical computing, one cannot ignore a large body of experienced FORTRAN programmers and well-crafted FORTRAN programs. One common programming style in FORTRAN is to place the array definition first, then followed by dimension parameters in a function subroutine. Therefore, in order to minimize the effort of porting FORTRAN programs and documents to C, it is desirable not to restrict the order of the declarations for the array definition and its dimension parameters. To this end, an alternative array syntax is also included in MacDonald's proposal, which can be demonstrated as follows:

```
float a[3][5], b[3][5], c[3,7];
void funct(float a[n][m], float b[n][m], float c[n][r], int n, int m, int r);
funct(a, b, c, 3, 5, 7);
```

where the scope of parameters n, m, and r is extended to the beginning of the parameter list. The implementation of this backward reference for array parameters is possible. In fact, even for a C implementation with a single pass, this backward reference can be achieved by only reprocessing the parameter list of a function definition after the enclosing parenthesis for the function arguments — namely, two-passes over the parameter list are required in order to diagnose the undeclared identifiers. However, this lexical reordering is not considered in the spirit of C [14]. According to the C standard, an identifier should be declared before it can be used in an expression. Therefore, in general, only one pass is needed to parse a C program. The major technique problem for this lexical ordering appears to be that when the parameters, such as `n` and `m`, for dimensions of an array happen to be predefined constants at the lexical level higher than the parameter level of a function, in this scenario the existing C code will break as shown in the following example.

```
  enum { n = 5 };
  void funct(int a[n], int n){ }
```

According to the C standard, the above code is equivalent to the following code.

```
  enum { n = 5 };
  void funct(int a[5], int n){ }
```

However, according to MacDonald's proposal, the parameter n in the function prototype scope is extended to the beginning of the parameter list for the argument `a[n]`. The second format has been withdrawn from MacDonald's recent proposal after years of heated debate among users, designers, and implementors of C [14].

Another major difference between this proposal and all other previous VLA proposal is that this proposal permitting deferred-shape arrays as members of structures and unions whereas others do not. A vote was taken at NCEG meeting #5 in Norwood, MA, which asked the following question: should there are some way to declare a VLA member? Eleven people voted yes, one no, and three undecided. Clearly, there is considerable sentiment for permitting VLA member in structures and unions. This proposal reflects this sentiment. However, there is no portable means to share the type with other functions without introducing nested functions if deferred-shape arrays are restricted to variables with automatic storage duration. VLAs within nested functions are implemented in C^H. In this proposal, a `goto` statement in nested functions behaves like that in the Pascal language [16]. Memory allocated at program execution time for deferred-shape arrays will be automatically deallocated when the program exits the function either through a `return` statement or through a notorious `goto` statement.

In McDonald's proposal, the deferred-shape of the array shall be evaluated at the time the type definition *typedef* is declared and not at the time it is used as a type specifier in an actual declarator. Our implementation experience indicates that the deferred-shape of array obtained in an actual declarator is more consistent when all three different VLAs are implemented.

One minor difference between the deferred-shape arrays in this proposal and MacDonald's proposal is that the order of evaluations in the declaration statement such as

```
int n;
int a[n++], b[n++];
```

is defined in MacDonald's proposal. The order of evaluation in the above example is undefined according to this proposal. It seems reasonable to leave the order of evaluation undefined in light of the fact that the order of evaluation of the following similar statement is also undefined according to the C standard.

```
printf("%d %d \n", n++, a[n++], b[n++]);
```

It should be pointed out that there is a technical reason for prohibiting deferred-shape array at block scope with static storage duration. For the following example,

```
void funct(int n){
  static int b[n];
  ...
}
```

if the memory for the deferred-shape array `a[n]` was allocated at compile time, the environment for variable `n` would not be available at the point where array `a[n]` was declared. If the memory for the deferred-shape array `a[n]` was allocated at program execution time each time when the function is invoked, it violates "staticness" of the C standard. But, the restriction on automatic storage duration of deferred-shape arrays can be relaxed as was implemented in the C^H programming environment. For example, the following code is valid in the C^H programming environment.

```
extern int n
int a[n][n];
int n = 90;
int m = n;
int (*p)[:] = a;
static int b[m][m];
main(){
  int i;
  for(i = 0; i<n; i++)
    b[i][i] = a[i][i];
}
```

where the memory for the deferred-shape arrays `a[n]` and `b[m]` are allocated at program execution time, The initialization of variables `n`, `m`, and `p` are performed at compile time. In the C^H programming environment, the program starts execution of function `main()` after execution of all executable statements. It is an error to modify the size expression of the deferred-shape. For example,

```
int n=8, m=9;
extern int a[n];
int b[n];
int a[m];          /* ERROR: modify deferred-shape */
extern int a[8];   /* ERROR: modify deferred-shape */
extern int a[2*n]; /* ERROR: modify deferred-shape */
```

```
main(){
  extern int b[m]; /* ERROR: modify deferred-shape */
}
```

However, even when restriction on automatic storage duration of deferred-shape arrays are relaxed, structures and unions with members of deferred-shape arrays cannot be effectively used because variables and members of structures and unions with static storage duration shall be initialized only once, regardless it is performed at compile time or program execution time. Therefore, nested functions appear to be the most effective way to take advantage of the dynamic nature of deferred-shape arrays in members of structures and unions.

6.2 Pointers to Array of Assumed-Shape

Pointers to VLA are implemented in Cray Research standard C compiler as shown in the following example.

```
void funct(int *p1, int *p2, int n, int m, int r){
  int (*A)[n] = p1;
  int (*B)[m][r] = p2;
  A[0][0] = B[0][0][0];
}
int a[3][4], b[5][6][7];
funct((int *)a, (int *)b, 4, 6, 7);
```

where `A` and `B` are declared as pointers to VLA. The values of pointer to int passed as parameters `p1` and `p2` are assigned to `A` and `B`, respectively. As one can see that parameters `n, m`, and `r` are needed for casting from other pointers to pointers to VLA.

Ritchie was the first who proposed using fat pointers as a means of VLA in C [13]. The construct `[?]` is used to declare a pointer to VLA in Ritchie's proposal. The shape information is automatically stored in association with pointers, which is why they are sometimes called *fat pointers*. Ritchie's proposal can be illustrated by the following code example,

```
void funct(int *p1, int *p2, int n, int m, int r){
  int (*A)[?] = (int (*)[n])p1;
  int (*B)[?][?] = (int (*)[m][r])p2;
  A[0][0] = B[0][0][0];
}
int a[3][4], b[5][6][7];
funct((int *)a, (int *)b, 4, 6, 7);
```

where fat pointers `A` and `B` are declared without integral size expressions for array bounds. Ritchie's proposal only contains fat pointers.

Prosser extended Ritchie's proposal with automatical binding of the address of any array with a fat pointer [13] as shown in the following example.

```
void funct(int (*A)[?], B[][?]){
  A[0][0] = B[0][0];
}
int a[3][4], b[5][6][7];
funct(a, b);
```

where both A and B are fat pointers. In addition, Prosser called for integration of automatic variable length arrays proposed by MacDonald and fat pointers proposed by Ritchie [13]. Since fat pointers were not implemented and tested, many important details about fat pointers were not provided in Ritchie and Prosser's proposal. Meissner [15] provided more specifications for fat pointers. For example, he defined that the size of a fat pointer is 0 if a null pointer was assigned to the fat pointer or the size is the total number of bytes in the array pointed to by the pointer, computed at program execution time.

Pointers to assumed-shape array implemented in $\mathrm{C^H}$ are quite similar to fat pointers. However, there are some differences. Unlike fat pointer proposals, pointers to assumed-shape arrays described in this paper are based upon an actual implementation of the $\mathrm{C^H}$ programming language. Linguistic features are spelled out based upon implementation and application experience. One of most important contributions of this paper is that pointers to array of assumed-shape are treated in the same manner as pointers to array of fixed-length and they are integrated with deferred-shape arrays and assumed-shape arrays. For example, pointers to assumed-shape array can be declared in any scope with either static or automatic storage duration, as members of structures and unions, and within nested functions. The size of a pointer to array of assumed-shape is handled in the same manner as the size of a pointer to function. The size of a pointer to array of assumed-shape is the same as that of a pointer to array of fixed-length, which is different from Meissner's proposal. Unlike Meissner's proposal, when a null pointer is assigned to a pointer to assumed-shape array, the bounds of the array is not specifically specified in this proposal. In fact, the last-stored values for bounds are not changed in the implementation of the $\mathrm{C^H}$ programming language. It should be pointed out that, like pointers to array of fixed-length, the new `restrict` qualifier currently proposed by Homer from Cray Research [19] may be applied to pointers to array of assumed-shape to specify that the object pointed to by the pointer is not aliased with other objects as shown in the following example,

```
void funct(int (* restrict A)[:], (* restrict B)[:], n, m){
  int i, j, a[n][m];
  int (* restrict p)[:] = a;
  for(i=0; i<n; i++)
    for(j=0; j<n; j++)
      A[i][j] = p[i][j]+B[i][j]; /* vectorization or parallelization */
}
int a[3][4], b[3][4];
funct(a, b, 3, 4);
```

Another important difference between pointers to array of assumed-shape and fat pointers is the syntax. The unspecified extents of the pointer to VLA, fat pointer, and pointer to array of assumed-shape are specified as `[n]`, `[?]`, `[:]`, respectively. If in the function prototype scope, it may also be specified as `[*]` in MacDonald's proposal of pointer to VLA. Using the colon symbol ":", additional features such as passing a segment of arrays with adjustable lower bounds for each dimension, as they are defined in Fortran 90 [17], can be extended in the future.

6.3 Assumed-Shape Arrays

Both assumed-shape array and pointer to array of assumed-shape can be used to pass VLAs to functions. However, there are some important differences between assumed-shape arrays and pointers to array of assumed-shape. First, when the `sizeof` operand is applied to an assumed-shape array, the result is the total number of the bytes used to store elements of the actual array, which is evaluated at program execution time. The size of a pointer to array of assumed-shape is the same as the size of the pointer to the data type of the array, which is evaluated at compile time.

Second, assumed-shape arrays are designed for anti-aliasing of unrestricted pointers, because only variables of array type can be passed to an assumed-shape array in the argument of a function. The arguments for assumed-shape arrays of a function behave as if they were completely copied in when the function is called and completely copied out when the control of the program is returned to the calling function. The hidden aliasing associated with pointers can be illustrated by the following program.

```
void add(complex zz1[], zz2[], zz3[], int n){
/* zz1[n] = zz2[n] + zz3[n] */
  int i;
  for(i=0; i<n; i++)
    zz1[i] = zz2[i]+zz3[i];  /* or *zz1++ = *zz2++ + *zz3++ */
}
main(){
  complex z1[11], z2[11], z3[11];
```

```
    add(z1, z2, z3, 10);       /* no aliases */
    add(&z1[1], z1, z2, 10);   /* with aliases */
  }
```

where function `add()` appears to add the corresponding elements of two one-dimensional arrays `zz2` and `zz3` together, and store the results into array `zz1`. This is precisely what happens when the function `add()` is called by `add(z1, z2, z3, 10)` without any aliases. The resulting array `z1` contains the sum of arrays `z2` and `z3`. The loop inside the function `add()`, therefore, appears an ideal candidate for vectorization and parallelization [2]. However, when the function is called by `add(&z1[1], z1, z2, 10)` with aliases. The effective result in the main routine is the recursive addition operation `z1[i+1] = z[i]+z2[i]`. The loop inside the function `add()` must be executed in scalar mode this time. Because there is no guarantee that arrays passed to function `add()` are disjoint, therefore, the compiler must produce the scalar code for the loop inside the function `add()`. However, this is not the case for functions using the assumed-shape arrays. For example, in the following program,

```
void vadd(complex zz1[:], zz2[:], zz3[:], int n){
/* zz1[n] = zz2[n] + zz3[n] */
  int i;
  for(i=0; i<n; i++)
    zz1[i] = zz2[i]+zz3[i]; /* vectorization or parallelization */
 }
main(){
  complex z1[11], z2[11], z3[11];
  vadd(z1, z2, z3, 10);     /* no aliases */
}
```

because arrays `a`, `b`, and `c` in the function `vadd()` are complete arrays, the for-loop for array addition can be executed in vector or parallel mode if the vector pipelined architecture and multiple CPUs are available in a computer system. A program using assumed-shape arrays will have a great potential for performance optimization using vector and parallel processing capabilities of modern computers. Hidden aliasing associated with pointers in C is one of the most difficult problems to be solved. Because only complete arrays not just pointers can be passed to assumed-shape arrays, there are more information for compilers to generate an optimized code. However, using assumed-shape arrays alone cannot solve all world's problems. For example, if the statement

```
zz1[i] = zz2[i] + zz3[i];
```

inside the function `vadd()` of the above program is replaced by the statement

```
zz1[i+1] = zz2[i] + zz3[i];
```

the function call of `vadd(z1, z2, z3, 10)` will become the recursive operation

```
zz1[i+1] = zz2[i] + zz3[i]
```

as if it was performed in the main routine because of the aliases. Therefore, in this case, the compiler must produce the scalar code for the loop inside the function `vadd()`. Note that the function call of `vadd(&z1[1], z2, z3, 10)` would be illegal since a pointer cannot be passed to an assumed-shape array.

Finally, assumed-shape arrays allow association of the actual arrays of different data type from the formal array in a function call, which transcends a long-standing tradition in scientific programming. The application of passing arrays of different data type can be illustrated by the following example.

```
#include <stdio.h>
main(){
  int i;
  /* coefficients for (x^2+1)(x^2-4) */
  float coeff[5] = {4, 0, -3, 0, 1};
  float froot[4];
```

```
    complex zroot[4];
    /* void zroots(complex [:], int, complex [:], int); */

    zroots(coeff, 4, zroot, 0);
    zroots(coeff, 4, froot, 0);
    for(i=0; i<4; i++)
      printf("%0.1f ", zroot[i]);
    printf("\n");
    for(i=0; i<4; i++)
      printf("%0.1f ", froot[i]);
    printf("\n");
  }
```

where function `zroots()` is a system function. It can compute all roots of a complex polynomial equation. The first argument is a complex array that contains the coefficients of the polynomial. The second integral argument indicates the order of the polynomial. The third argument is a complex array that returns the roots of the polynomial. The last argument is a boolean number; if it is true, a more sophisticated algorithm will be used. For the function call of `zroots(p, 4, froot, 0)`, the assumed-shape array `coeff` of 5 floats for coefficients of the polynomial $(x^2+1)(x^2-4) = x^4 - 3x^2 - 4$ is coerced to a complex array implicitly. The result of a complex array of 4 elements is also coerced to an assumed-shape array `froot` of 4 floats. The function `zroots()` can be used to obtain either real roots or complex roots, which is not possible in any other existing typed computer programming language. The output from the above code is shown below.

complex(0.0,1.0) complex(0.0,-1.0) complex(-2.0,0.0) complex(2.0,0.0)
NaN NaN -2.0 2.0

It should be pointed out that, unlike in FORTRAN, when a complex number is cast to real number in the $\mathrm{C^H}$ programming language, if the imaginary part of the complex number is identically zero, the result is the real part of the complex number; otherwise, the result is a NaN [5].

7 Conclusions

The implementation of arrays of variable length in the $\mathrm{C^H}$ programming language within the framework of C has been presented in this paper. Arrays of variable length consist of deferred-shape arrays, assumed-shape arrays, and pointers to array of assumed-shape. VLAs described in this paper retain the brevity of C. Deferred-shape arrays with automatic storage duration in block scope and static storage duration in file or program scope can be declared. They can be declared within nested functions and as members of structures and unions. Assumed-shape arrays can be used to pass multi-dimensional arrays of variable length to functions. They are useful for overcoming the hidden aliasing problems associated with unrestricted pointers. This anti-aliasing feature is especially meaningful in the arena of vector and parallel programming, the future of scientific computing. The other significant feature of the assumed-shape array in $\mathrm{C^H}$ is that it can interface with arrays of different data types, which is not feasible in any other currently existing typed computer programming language. Pointers to array of assumed-shape are the most flexible. They can be declared at any scope with either static storage duration or automatic storage duration. They are treated in the same manner as pointers to array of fixed-length. All code fragments presented in this paper, except for code related to switch statement, goto statement, and structures, which are currently under implementation, are in their original text forms as they have been tested in a $\mathrm{C^H}$ programming environment. Therefore, implementation of variable length arrays with their linguistic features described in this paper within the paradigm of the C programming language is practical and feasible.

8 Acknowledgements

This work was supported in part by the National Science Foundation as a Research Initiation Award under Grant DMII-9309207, by the University of California, Davis through a faculty research grant, an undergraduate instructional improvement grant, and a junior faculty research fellowship, and by Motorola, Inc.

9 References

1. Kernighan, B. W. and Ritchie, D. M., *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, first edition, 1978; second edition, 1988.

2. Cheng, H. H., Passing Arrays to Functions under the Programming Paradigm of C, NCEG, X3J11.1/93-041, September 18, 1993.

3. ANSI, *ANSI Standard X3.9-1978, Programming Language FORTRAN* (revision of ANSI X2.9-1966), American National Standards Institute, Inc., NY, 1978.

4. Cheng, H. H., Scientific Computing in the C^H Programming Language, *Scientific Programming*, vol. 2, No. 3, Fall 1993, pp. 49-75.

5. Cheng, H. H., Handling of Complex Numbers in the C^H Programming Language, *Scientific Programming*, vol. 2, No. 3, Fall 1993, pp. 77-106.

6. Cheng, H. H., Programming with Dual Numbers and Its Applications in Mechanisms Design, *Engineering with Computers,* Vol. 10, No. 4, 1994, pp. 212-229.

7. Cheng, H. H., Adding References and Nested Functions to C for Modular and Parallel Scientific Programming, NCEG, X3J11.1/93-44, 1993.

8. Cheng, H. H., Extending C and FORTRAN for Design Automation, *Proceedings of ASME Design Automation Conference*, Minneapolis, MN, Sept. 11-14, 1994, Vol. 1, pp. 93-103; also *ASME Journal of Mechanical Design* (accepted).

9. Cheng, H. H., Extending C with Arrays of Variable Length, WG14/N348, X3J11/94-033, June 6, 1994.

10. ISO/IEC, *Programming Languages - C*, 9899:1990E, ISO, Geneva, Switzerland.

11. Stallman, R., Variable Length Array Parameters, NCEG X3J11.1/92-016, March 17, 1992.

12. MacDonald, T., Arrays of Variable Length, NCEG, X3J11.1/93-007, 1993.

13. Prosser, D. F., Providing Automatic Variable Length Arrays, NCEG, X3J11.1/92-035, May 5, 1992.

14. NCEG, The Great VLA Debate, NCEG X3J11.1/93-015, April 1, 1993.

15. Meissner, M., Fat Pointers, NCEG, X3J11.1/93, December 6, 1993.

16. ANSI, *ANSI/IEEE Standard 770 X3.97-1983, IEEE Standard Pascal Programming Language*, IEEE, Inc., NJ, 1983.

17. ISO/IEC, *Information Technology, Programming Languages - FORTRAN*, 1539:1991E, ISO, Geneva, Switzerland.

18. Cheng, H. H., Vector Pipelining, Chaining, and Speed on the IBM 3090 and Cray X-MP, *IEEE Computer*, Vol. 22, No. 9, September, 1989, pp. 31-46.

19. Homer, B., Restricted Pointers in C, NCEG, X3J11.1/93-006, 1993.