;login:

THE MAGAZINE OF USENIX & SAGE

October 2001 • Volume 26 • Number 6

inside:

PROGRAMMING

Variable Length Arrays

By Glen McCluskey

USENIX & SAGE

The Advanced Computing Systems Association &
The System Administrators Guild

variable length arrays

by Glen McCluskey

Glen McCluskey is a consultant with 20 years of experience and has focused on programming languages since 1988. He specializes in Java and C++ performance, testing, and technical documenta-tion areas.

glenm@glenmccl.com

We've been looking at some of the features added to C9X, the recent standards update to C. In this column we'll consider the use of variable length arrays (VLAs).

Some Basics

Suppose that you need to allocate an array in your program, but when you're writing the program, you don't know how long the array should be. What do you do in such a case? An obvious answer is to use malloc() and dynamic allocation. This approach will certainly work, but has a couple of problems. One is that you need to worry about freeing up the storage when you're done with it to avoid memory leaks, and another is that dynamic allocation for multidimensional arrays gets complicated.

C9X offers another approach, the use of VLAs. Here's an example of what such usage looks like:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    if (argc != 2) {
        fprintf(stderr, "Missing numeric argument\n");
        return 1;
    }

    int n = atoi(argv[1]);

    int x[n];

    printf("sizeof = %d\n", sizeof(x));

    return 0;
}
```

A numeric value representing an array length is passed to the program, and an array of this length is allocated. When the array goes out of scope, its storage is automatically reclaimed.

The size of the array is calculated at run time. For example, if you specify an argument of 10, and the size of an int on your machine is 4, then 40 will be printed.

The array is of fixed size once it's allocated, but its size is not fixed until the flow of control passes the declaration.

Variable Length Arrays as Function Arguments

Here's another example of how you can use VLAs, passing them as function arguments:

```
#include <stdio.h>

typedef void (*fp)(int, int[*][*]);

void f(int, int[*][*]);

int main()
{
    int n = 2;
```

```
    int x[n][n];

    x[0][0] = 1;
    x[0][1] = 2;
    x[1][0] = 3;
    x[1][1] = 4;

    fp fptr = &f;

    (*fptr)(n, x);
    fptr(n, x);

    return 0;
}

void f(int n, int x[n][n])
{

    printf("0,0 = %d\n", x[0][0]);
    printf("0,1 = %d\n", x[0][1]);
    printf("1,0 = %d\n", x[1][0]);
    printf("1,1 = %d\n", x[1][1]);
}
```

In this example a 2 x 2 VLA is created and then passed as an argument to a function. The called function is declared before use, along with a function pointer typedef. Note how the [*] notation is used to specify VLA parameters.

Pointer Arithmetic

In the first example above, we showed how sizeof() returns a dynamic value, known only at run time. This same consideration also applies to other calculations, such as pointer arithmetic. Consider the following example:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    if (argc!=2) {
        fprintf(stderr, "Missing numeric argument\n");
        return 1;
    }
    int n = atoi(argv[1]);
    int arr[10][n];
    int (*p)[n] = arr;
    arr[4][n-1] = 37;

    p += 4;

    printf("%d\n", p[0][n-1]);

    return 0;
}
```

The VLA arr becomes a 10 x n array, with n set at run time. We initialize a pointer to the array, store a value at [4][n-1] in the array, and then increment the pointer by 4. In this situation, saying p += 4 means that four rows of the array should be skipped, but the length of a row (the number of columns) is not known to the compiler and must be dynamically evaluated.

The variable p in this example uses what is known as a "variably modified type." The line

```
int (*p)[n] = arr;
```

declares p to be of variably modified type and then initializes it with arr. p is a pointer to an array of n integers, and the initialization sets p to point at the VLA. [n] is part of the variably modified type.

VLAs are a subset of variably modified types. Such types must be declared at block or function prototype scope. So, in this example:

```
int n = 5;
//int (*p)[n];
void f()
{
      int x[n][n];
      int (*p)[n] = x;
}
```

uncommenting the global declaration will trigger a compile error.

Restrictions on Variable Length Arrays

There are some things you can't do with VLAs. One of them is to use {} initializers, like this:

```
void f(int n)
{
      int x[n] = {1, 2, 3};   /* can't do this */
}
```

One problem with allowing this usage is that the value of n is not known to the compiler, so it's impossible to determine whether too many initializer values have been specified.

Another thing you can't do is to allocate a VLA using global or static storage:

```
int n = 3;
int x[n];
void f()
{
      static int y[n];    /* can't do this */
}
```

There's no way at compile time to determine how big these arrays will be.

A third example concerns the use of sizeof, like this:

```
void f()
{
      int n = 3;
      int x[n];
      int y = 5;

      switch (y) {
         case sizeof(n):
            break;
         case sizeof(x):  /* can't do this */
            break;
```

```
    }
}
```

The usage in the second case label is invalid because the size of x is not known to the compiler.

Finally, it's illegal to jump around the declaration of a variable length array:

```
void f()
{
    int n = 3;
    int y;
    goto lab;      /* can't jump around decl below */

    int x[n];

lab:

    y = x[0];
}
```

Static and Restrict

There's another interesting aspect of VLAs that ties in with performance and optimization. When you're specifying variable array parameters to a function, you can use the static and restrict keywords:

```
#include <stdio.h>

double f(int n, double x[static n])
{
    double sum = 0.0;

    for (int i = 0; i < n; i++)
        sum += x[i];

    return sum;
}

int main()
{
    int n = 10;
    double x[n];
    for (int i = 0; i < n; i++)
        x[i] = (double)(i + 1);

    double sum = f(n, x);

    printf("%g\n", sum);

    return 0;
}
```

Using static tells the compiler that the underlying pointer used to hold the VLA argument (1) is not NULL, (2) points to elements of double type, and (3) points to at least n elements which are guaranteed to be available.

This information can be used to initiate loads or prefetches of the arrays that are accessed within the function. Another example uses both static and restrict:

```
#include <stdio.h>
```

```
void f(int n, double x[static restrict n],
       double y[static restrict n])
{
       for (int i = 0; i < n; i++)
           x[i] += y[i];
}

int main()
{
       int n = 10;
       double x[n];
       double y[n];
       for (int i = 0; i < n; i++) {
           x[i] = (double)(i + 1);
           y[i] = (double)(i + 100);
       }

       f(n, x, y);

       for (int i = 0; i < n; i++)
           printf("%d %g\n", i, x[i]);

       return 0;
}
```

In this example, the array parameters to f() are guaranteed to be (1) non-NULL, (2) of type double, (3) at least of length n, and (4) unique and non-overlapping. Such information can be used to generate optimized code.

Variable length arrays are especially useful in numerical programming, and also in situations where you don't know the array size at compile time, and you don't want to deal with all the complications of dynamic allocation.