;login:

THE MAGAZINE OF USENIX & SAGE
July 2001 • Volume 26 • Number 4

inside:

FLEXIBLE ARRAY MEMBERS AND DESIGNATORS IN C9X
by Glen McCluskey

USENIX & SAGE
The Advanced Computing Systems Association &
The System Administrators Guild

flexible array members and designators in C9X

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 documentation areas.

<glenm@glenmccl.com>

We've started looking at new features in C9X, the recent standards update to C. In this column we'll look at a couple of C9X features that you can use to declare and initialize structures.

Flexible Array Members

If you've been around C for a long time, you might have encountered the following sort of usage, which goes by the name "struct hack":

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

struct A {
    int a;
    int b[1];
};

int main()
{
    const int N = 10;
    int i;

    struct A* p = malloc(sizeof(struct A) +
        sizeof(int) * (N - 1));

    p->a = 100;
    for (i = 0; i < N; i++)
        p->b[i] = i;

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

    return 0;
}
```

The idea here is that you have a struct, and one of the struct members is an array, and the array is of variable length. So you put it at the end of the struct, and you allocate extra space for the struct. So the memory for the struct object has some extra space tacked on to it, for the elements of the array.

This usage represents undefined behavior, even though it will work much of the time. With C9X, however, the usage has been legitimized, and goes by the name "flexible array member." Such a member must be the last in the struct, and the length is omitted within the []. The corresponding C9X version of the example above would be:

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

struct A {
    int a;
    int b[];
};

int main() {
    const int N = 10;
    int i;

    struct A* p = malloc(sizeof(struct A) +
        sizeof(int) * N);

    p->a = 100;
```

```
    for (i = 0; i < N; i++)
        p->b[i] = i;
    printf("%d %d %d\n", p->a, p->b[0], p->b[N-1]);
    return 0;
}
```

No actual array space is allocated within the struct object, so we allocate extra space for N elements instead of N-1. However, the size of the object takes into account the alignment restrictions on the array. For example, with this code:

```
#include <stdio.h>

struct A {
    char a;
};

struct B {
    char a;
    int b[];
};

int main() {
    printf("%d\n", sizeof(struct A));
    printf("%d\n", sizeof(struct B));
    return 0;
}
```

the size of A will typically be 1, and the size of B will be 4.

There are some restrictions on the use of flexible array members. For example, you cannot have an array of objects, if the object type includes a flexible array member:

```
struct A {
    int a;
    int b[];
};

struct A data[10];
```

Nor can an object of such a type be used as a member in the middle of another type:

```
struct A {
    int a;
    int b[];
};

struct B {
    struct A a;
    double b;
};
```

In both cases, the flexible array member causes problems with object layout, because its total size is not known.

Designators

Suppose that you're initializing a struct object, and your code looks like this:

```
#include <stdio.h>

struct A {
    int x;
    int y;
};
```

```
struct B {
    struct A a;
    double b;
};
struct B obj = {37, 47, 12.34};
int main() {
    printf("%d %d %g\n", obj.a.x, obj.a.y, obj.b);
    return 0;
}
```

Even in this simple example, the initialization of the B object is a little confusing. The first two values, 37 and 47, get assigned to the A sub-object within B, and then the 12.34 value gets assigned to the second field of B.

Designators can be used to make such initialization much more clear. Here's an equivalent program to the one above:

```
#include <stdio.h>
struct A {
    int x;
    int y;
};
struct B {
    struct A a;
    double b;
};
struct B obj = {
    .a = {
        .x = 37,
        .y = 47
    },
    .b = 12.34
};
int main() {
    printf("%d %d %g\n", obj.a.x, obj.a.y, obj.b);
    return 0;
}
```

The notation:

```
.x = value
```

is used to initialize a specific field; the fields need not be given in order, and you can omit fields.

You can also use designators to specify particular elements in array initialization, like this:

```
#include <stdio.h>
struct A {
    int x;
    int y;
};
struct A data[] = {
    [100].x = 37, [100].y = 47
};
```

```
int main()
{
    printf("%d %d\n", data[0].x, data[0].y);
    printf("%d %d\n", data[100].x, data[100].y);
    return 0;
}
```

In this example, the designator notation is not only easier to read, but it allows you to jump ahead to initialize a specific array element, without having to specify zeros for all the previous elements. Another example of initializing sparse arrays in this way looks like this:

```
#include <stdio.h>
#define N 10000
int data[] = {[0] = 1, [N/2] = 2, [N-1] = 3};
int main() {
    printf("%d %d %d\n", data[0], data[N/2], data[N-1]);
    return 0;
}
```

You can also use designators in union initialization. The existing C rules say that an initializer for a union is applied to the first member of the union. For example, in this code:

```
#include <stdio.h>
union U {
    char a;
    double b;
};
union U obj = {12.34};
int main() {
    printf("%g\n", obj.b);
    return 0;
}
```

the double initializer value is applied to the `char` member, and you will get garbage when you later try to access the double value. With designators, however, this problem is easily solved:

```
#include <stdio.h>
union U {
    char a;
    double b;
};
union U obj = {.b = 12.34};
int main()
{
    printf("%g\n", obj.b);
    return 0;
}
```

Using a designator, you can initialize any member of a union.

You can use designators and flexible array members in your programming to make your job easier, and your code easier to read and maintain.