;login:

THE MAGAZINE OF USENIX & SAGE
June 2001 • Volume 26 • Number 3

inside:

PROGRAMMING
Declarations in C9X

by Glen McCluskey

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

Declarations 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>

Last time, we started looking at some of the new language features in C9X, the recent update to C. In this column we'll look at C9X declarations and how they have changed from what you are familiar with.

Function Declarations

If you've used C for a long time, you might remember that the language was once much looser about function declarations. You didn't have to declare functions before use; for example, you could say:

```
void f()
{
    g(37);
}
```

without complaint. If you didn't specify a return type for a function:

```
f()
{
}
```

it would be assumed to be int, like this:

```
int f()
{
}
```

And you didn't have to use a return statement, or maybe the return statement didn't match the return type:

```
int f()
{
    return;
}
```

These examples are invalid in C9X; you must declare functions before use, you cannot default the return type of a function, the return type must match return statements, and a return statement is required for a function with a non-void return type.

You are still allowed to declare functions without a prototype:

```
void f();

void g()
{
    f(37, 47);
}
```

but in the C9X standard this usage is marked as obsolescent. Note also that:

```
void f();
```

is not the same as:

```
void f(void);
```

or:

```
void f() {}
```

The first of these is a non-prototype function declaration, with parameter information unspecified, while the latter two declarations specify that the function has no parameters.

These tighter rules are not arbitrary; they lead to better code. Consider, for example, this program:

```
#include <stdio.h>

void f();

int main()
{
    f(37);

    return 0;
}

void f(int a, int b)
{
    printf("%d %d\n", a, b);
}
```

We use an old-style declaration of f(), then call the function with a single argument. The function actually requires two arguments, so when the printf() is encountered, one of the parameters has a garbage value.

The same consideration applies to return statements, if, for example, you fail to return a value from a function declared to return an int.

Inline Functions

With C9X you can define a function using the inline keyword, like this:

```
inline void f() {...}
```

inline is a hint to the compiler that it should optimize calls to the function, perhaps by expanding them in the context of the caller. There is no requirement that a compiler actually observe this hint.

Inline functions are quite subtle in a couple of areas, illustrated by the following example, composed of two translation units:

```
// file #1

void f()
{
}

void g(void);

int main()
{
    f();
    g();

    return 0;
}

// file #2

inline void f()
{
}

void g()
{
    f();
}
```

The inline definition of f() has external linkage. To give the function internal linkage, we would use static inline void f().

This definition is not an external definition, and because f() is actually called, an external definition is required somewhere within the program. This definition is found in the first translation unit.

A program can contain both an inline and an external definition of a function. The program cannot rely on the inline definition being used, even if it is clearly visible, as in the case above where g() calls f().

The inline and external definitions are considered distinct, and a program's behavior should not depend on which is actually called. As another example of this idea, consider:

```
// file #1

const int* f()
{
    static const int x = 0;
    return &x;
}

// file #2

inline const int* f()
{
    static const int x = 0;
    return &x;
}

int main()
{
    return f() == f();
}
```

Given these definitions of f(), it is not necessarily the case that main() always returns 1. Different f()s may be called, containing different static objects. An example of where this might happen would be with a compiler that applies some heuristic about how big inline expansions are allowed to get, with the external definition of the function used in cases where the inline expansion would be too large.

The C inline model is similar to that of C++, but differs in that (1) if a function is declared inline in one translation unit, it need not be declared so in all translation units, and (2) all definitions of an inline function need not be the same, but the program's behavior should not depend on whether an external definition or an inline version of a function is called.

Mixed Declarations and Code

In C9X you can intersperse declarations and code, a feature also found in C++. It's no longer necessary to place all declarations at the beginning of a block. Here's an example that counts the number of "A" characters in a file, using this coding style:

```
#include <stdio.h>

int main(int argc, char* argv[])
```

```
{
    if (argv[1] == NULL) {
        fprintf(stderr, "Missing input file\n");
        return 1;
    }

    FILE* fp = fopen(argv[1], "r");
    if (fp == NULL) {
        fprintf(stderr, "Cannot open input file\n");
        return 1;
    }

    int cnt = 0;
    int c;

    while ((c = getc(fp)) != EOF) {
        if (c == 'A')
            cnt++;
    }

    fclose(fp);

    printf("count = %d\n", cnt);

    return 0;
}
```

This style is in many ways more natural than the older approach of grouping all declarations at the top of a block. Each declaration is introduced as needed, in a meaningful context, and there's less temptation to recycle declarations, by using a variable in multiple ways within a function.

The scope of each identifier declared in this way is from its point of declaration to the end of the block.

Declarations in for Statements

You can also use declarations within for statements, like this:

```
#include <stdio.h>

int main()
{
    for (int i = 1, j = 100; i <= 10; i++, j—)
        printf("%d %d\n", i, j);

    return 0;
}
```

This is obviously a convenient way to specify loop indices. If you use this approach, a common coding style is no longer valid:

```
int main()
{
    int max = 10;

    for (int i = 1; i <= max; i++) {
        if (i == 5)
            break;
    }
    if (i > max)
        ; // loop completed normally
    return 0;
}
```

The scope of the loop index goes to the end of the for statement, so the index name is unknown outside. If you want to code this way, you need to continue declaring the loop variable before the loop.

Here's another (legal) example of for statements, one that demonstrates the subtleties of scoping:

```
void f()
{
    int i = 37;

    for (int i = 47; i <= 100; i++) {
        int i = 57;
    }
}
```

The for statement introduces a new block, and the statement body is also a distinct block. To make the blocks more visible, we could write the code like this:

```
void f()
{
    int i = 37;

    {
        for (int i = 47; i <= 100; i++) {
            int i = 57;
        }
    }
}
```

The three declarations of i are all valid, because each occurs in a new scope. Each declaration hides variables of the same name in outer scopes.

The new declaration features promote better programming and more natural documentation. They are natural to use and reduce programming errors.