;login:

THE MAGAZINE OF USENIX & SAGE
October 2002 volume 27 • number 5

inside:

USENIX & SAGE

The Advanced Computing Systems Association &
The System Administrators Guild

C99 and compatibility

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

In previous columns, we've looked at some of the new features in C99, the standards update to C. In this presentation we'll discuss compatibility and look at issues with mixing C89 (the previous C standard) and C99 code. We'll also look at compatibility between C99 and C++.

C99 and C89

Let's start by stating what is probably obvious: if you use new C99 features in your C programming, you should not expect your programs to compile with an older C89 compiler. Here's an example:

```
#include <stdio.h>

struct A {
      int x;
      int y;
};

struct A a = {
      .y = 37,
      .x = 47
};

int main()
{
      printf("%d\n", a.x);
}
```

This program uses the designator feature of C99. When I compile the program as C89, the result is:

```
"e1a.c", line 9: error: expected an expression
      .y = 37,
      ^
"e1a.c", line 10: error: expected an expression
      .x = 47
      ^
2 errors detected in the compilation of "e1a.c".
```

Another example is the use of interspersed declarations and statements:

```
#include <stdio.h>

int main()
{
      int x;
      x = 37;
      int y;
      y = 47;
      printf("%d %d\n", x, y);
}
```

This feature was borrowed from C++ and added to C99.

You can't use C99 features with a C89 compiler, but what about going the other way? What happens if you try to compile a C89 program with a C99 compiler?

For example, consider the following program:

```
#include <stdio.h>

int main()
{
      static x = 37;
      x = g(x);
      printf(""%d\n", x);
}

int g(int x)
{
      return x + 10;
}
```

This usage is legal C89, but not C99. The declaration:

```
static x = 37;
```

leaves off the type (int), and the statement

```
x = g(x);
```

calls an undeclared function. The C99 standard tightened up both of these areas. Requiring that a function be declared before use catches a certain class of errors, such as passing a wrong argument type.

Another example of valid C89 usage that is invalid C99 concerns the use of keywords. For example, restrict is a C99 keyword, so this program is no longer legal:

```
#include <stdio.h>

int restrict = 37;

int main()
{
      printf("%d\n", restrict);
}
```

Other new keywords include inline, _Bool, _Complex, and _Imaginary. There are also many new library functions, which may conflict with existing functions in C89 programs.

A third example is failure to specify a return value:

```
int f()
{
      return;
}

int main()
{
}
```

This is valid C89 but invalid C99.

Beyond a few areas like this, C89 programs should work with a C99 compiler.

C99 and C++

The C++ language was designed on a C base, and in the early days there was emphasis on trying to keep C++ compatible with C, so that C programs could be compiled as C++ code. Since that time, C and C++ have both diverged and converged, and compatibility between them is a complicated issue.

The first point is similar to what we said above about using C99 features with a C89 compiler. There are a great many C++ features that have no equivalent in C99. One example is the C++ template feature:

```
#include <cstdio>

using namespace std;

template <class T> T min(T a, T b)
{
      return a < b ? a : b;
}

int main()
{
      int x = min(37, 47);

      printf("%d\n", x);
}
```

This usage has long been part of C++ but is unknown in C. Another example is function overloading:

```
#include <cstdio>

using namespace std;

void f(int i)
{
      printf("f(int) called\n");
}

void f(double d)
{
      printf("f(double) called\n");
}

int main()
{
      f(37);
}
```

The specific f() to call is determined based on the argument type. Again, there's no C equivalent to this feature.

Just as there are C++ features not known to C, there are C99 features not part of C++. For example, C99 mandates a long long type:

```
#include <stdio.h>

long long x = 0xffffffffffffffffull;

int main()
{
      printf("%llu\n", x);
}
```

Many C++ compilers allow this feature, but if I compile the code using strict conformance compiler options, the result is:

```
"e4a.c", line 3: error: the type "long long" is nonstandard
      long long x = 0xffffffffffffffffull;
         ^
"e4a.c", line 3: error: the type "long long" is nonstandard
      long long x = 0xffffffffffffffffull;
         ^
2 errors detected in the compilation of "e4a.c".
```

Another example is the C99 predefined identifier feature, used to obtain the name of a function at compile time:

```
#include <stdio.h>

void f()
{
      printf("%s\n", __func__);
}

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

C and C++ have diverged over the years, but they've also converged in some areas. For example, the following code uses a declaration within a for loop, and is now both legal C (C99) and C++:

```
#include <stdio.h>
int main()
{
        for (int i = 1; i <= 10; i++)
            printf("%d\n", i);
}
```

Likewise, this code uses //-style comments, an idea C99 borrowed from C++:

```
// This is an example of C++-style comments.

int main()
{
}
```

Another area of incompatibility between C and C++ concerns features that are part of both languages, but which have different semantics. For example, both C and C++ support wide character types, but in C, wchar_t is a typedef defined in a header file, whereas in C++ it is a keyword. Based on this difference, the following code is valid C99, but not C++:

```
int wchar_t = 37;

int printf(const char*, ...);

int main()
{
        printf("%d\n", wchar_t);
}
```

No header file is included in this program, so it's perfectly okay to use wchar_t as an identifier, assuming this is a C99 program. If it's a C++ program, then wchar_t is a keyword, and the program is invalid.

An additional example of different semantics concerns file statics:

```
#include <stdio.h>

static int x = 37;

int main()
{
        printf("%d\n", x);
}
```

This code is legal C and C++, but the C++ usage is deprecated, that is, there is a possibility that the code will not be valid at some future time. The preferred C++ usage is unnamed namespaces:

```
#include <cstdio>

using namespace std;

namespace {
        int x = 37;
}

int main()
{
        printf("%d\n", x);
}
```

Whether this approach is really better than file statics obviously depends on your particular biases.

Conclusions

Suppose that you are concerned about compatibility in a practical way. You might have a large body of C89 code that you are thinking of migrating to C99. Or you might have some C code that you want to compile as C++. What should you do?

It seems likely that current C compilers will be upgraded to incorporate C99 features, and C99 is mostly compatible with existing C89 code. C99 provides some attractive new features that you might want to use. But if you care about compatibility with C++, it's not at all clear if and when C++ will incorporate C99 features. And it seems very unlikely that C will ever adopt many of the distinctive C++ features such as templates.

If you have a body of C code that you compile with a C++ compiler, some of the C99 features will help with compatibility: for example, support for C++-style comments and for mixing declarations and code.

Beyond these basic observations, there is really no alternative to sitting down and identifying the underlying differences between C and C++ and specifying some coding standards for use in your project. For example, if you want to use wide characters in your C application and compile the result with a C++ compiler, then you need to know that C treats wchar_t as a typedef'd type defined in a header, whereas C++ treats it as a keyword.

One Web page that discusses C/C++ differences can be found at *http://david.tribble.com/text/cdiffs.htm*.