;login:

THE MAGAZINE OF USENIX & SAGE
February 2002 • Volume 27 • Number 1

inside:

PROGRAMMING

NEW PREPROCESSOR FEATURES IN C9X
BY GLEN MCCLUSKEY

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

new preprocessor features 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 been looking at some of the new features added to C9X, the standards update to C. In this column we'll look at how the preprocessor has changed for C9X.

Line-Oriented Comments

C++ and Java have always had //-style comments, where a comment goes from // to the end of the line. C9X now has this comment style as well. Such comments are straightforward to use, and you only need to remember that comments neither nest nor apply within string literals.

For example, in this code:

```
/*//*/ int main()
{
}
```

the // does not introduce a comment, and the code is a valid C program.

And in this example:

```
#include <stdio.h>

int main()
{
      char* s = "//testing";
      printf("%s\n", s);

      int a = 20;
      int b = 5;
      int c = a /**// b;
      printf("%d\n"" c);
}
```

the output is:

```
//testing
4
```

The // within the literal is not a comment, and /**// is equivalent to /.

Alternative Spellings and Digraphs

When you write C programs, especially if you live in the United States, you assume that the full ASCII character set is available to write your code. But this is a tricky area. ISO/IEC 646, the international equivalent of ASCII, allows local variations in some of the characters that C uses. These characters have the appropriate numeric value, but they won't necessarily display as the characters you might expect from looking in a C handbook like Kernighan & Ritchie.

Earlier versions of C defined trigraphs, sequences of three characters used to represent a character: for example, ??< to replace {. So if the ASCII value for { has some local variant meaning, you can use ??< instead.

But trigraphs are not the most readable, and in C9X, another attempt is made to tackle this readability problem by using digraphs, two-character sequences instead — for example, <% to replace {. To judge for yourself whether digraphs are successful, here's a short program written the usual way:

```
#include <stdio.h>
#define N 5

int vec[N];

int main()
{
      for (int i = 0; i < N; i++)
          vec[i] = i + 1;
      for (int i = 0; i < N - 1; i++) {
          for (int j = i + 1; j < N; j++) {
              if (vec[i] < vec[j]) {
                  int t = vec[i];
                  vec[i] = vec[j];
                  vec[j] = t;
              }
          }
      }

      for (int i = 0; i < N; i++)
          printf("%d\n", vec[i]);
}
```

And here it is using digraphs:

```
%:include <stdio.h>
%:define N 5

int vec<:N:>;

int main()
<%
      for (int i = 0; i < N; i++)
          vec<:i:> = i + 1;
      for (int i = 0; i < N - 1; i++) <%
          for (int j = i + 1; j < N; j++) <%
              if (vec<:i:> < vec<:j:>) <%
                  int t = vec<:i:>;
                  vec<:i:> = vec<:j:>;
                  vec<:j:> = t;
              %>
          %>
      %>

      for (int i = 0; i < N; i++)
          printf("%d\n", vec<:i:>);
%>
```

The second program is still readable, though its syntax is more cluttered. It's more portable in the display sense, that is, there is less likelihood that some of the characters will display oddly in a particular local environment.

Note that trigraphs are expanded within string literals, but digraphs are not.

Another C9X feature that aids readability and handling of local character-set variations is the <iso646.h> header, which defines macros for some common operators. For example, you can use or_eq instead of ||=. Here's a short example, showing two ways of writing the same assignment expression:

```
#include <stdio.h>
#include <iso646.h>

int main()
{
      int a = 37;
      int b = 47;
      int c;

      c = ~(a ^ b);
      printf("%x\n", c);

      c = compl(a xor b);
      printf("%x\n", c);
}
```

Macro Expansion

C has long had a mechanism to define functions with a variable number of arguments, but no corresponding feature for macros. C9X adds this feature, and here's what it looks like:

```
#include <stdio.h>

#define f(a, b, ...) #__VA_ARGS__

int main()
{
      printf("%s\n", f(37, 47, 57, 67));
}
```

The output of this program is:

```
57, 67
```

__VA_ARGS__ is a special preprocessor identifier, and it is replaced with the tokens of the variable arguments to the macro, in this example, 57 and 67. Prepending # to __VA_ARGS__ turns it into a string.

This feature is quite useful, for example, in writing debugging macros:

```
#include <stdio.h>

#define debug(...) fprintf(stderr, __VA_ARGS__)

void f(int x)
{
      debug("x = %d\n", x);
}

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

debug() uses a variable argument list and passes it to the fprintf() function, which also takes a variable number of arguments.

Another new feature is the ability to omit an argument. For example, if you run this program:

```
#include <stdio.h>

#define f(a, b) a##b

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

the result is:

```
3747

47
```

Preprocessor Arithmetic

intmax_t and uintmax_t are new integer types defined in <stdint.h>. They are typedefs that specify the maximum-width integer type available on your local system and are at least 64 bits.

C9X requires that preprocessor arithmetic for #if and #else be done using these types. Here's an example of what this requirement means in practice:

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

#if UINTMAX_MAX > 0xffffffful

char* s = "UINTMAX_MAX greater than 32 bits";

#else

char* s = "UINTMAX_MAX not greater than 32 bits";

#endif

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

If preprocessor arithmetic is not performed using maximum-width types, then the #if comparison in this example cannot be made accurately.

The features we've described above are all useful in writing more expressive and portable programs and in debugging applications that you've written.