;login:

THE MAGAZINE OF USENIX & SAGE
June 2002 volume 27 • number 3

inside:

PROGRAMMING

New I/O Features in C9X

by Glen McCluskey

USENIX & SAGE

The Advanced Computing Systems Association &
The System Administrators Guild

new I/O 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 presenting some of the new features in C9X, the standards update to C. In this column we'll discuss I/O features added to the library. We'll start by looking at printf specifiers, and then go on to consider several new I/O functions.

Printf **and New Types**

C9X adds four types to C: _Bool, wchar_t, long long, and _Complex. How do you print values of these types? _Bool has no printf specifier, and so to print a value of the type, you need to say:

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

int main()
{
      _Bool b = true;

      printf("%s\n", b == true ? "true" : "false");
}
```

Alternatively, you can treat a _Bool as an integer, with values 0/1.

The wide character type, wchar_t, is output using the printf %lc specifier or functions like fputwc. Here's an example:

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

int main()
{
      wchar_t c1 = L'\u1234';

      FILE* fp = fopen("test", "wb");
      fprintf(fp, "%lc", c1);
      fclose(fp);

      fp = fopen("test", "rb");
      wchar_t c2 = fgetwc(fp);
      fclose(fp);

      if (c1 != c2)
          printf("c1 != c2\n");

      fp = fopen("test", "rb");
      int c;
      while ((c = getc(fp)) != EOF)
          printf("%x ", c);
      printf("\n");
      fclose(fp);
}
```

Wide characters have an encoding, used to convert them to or from a sequence of bytes. For example, the wide character L'\u1234' is encoded as the three bytes:

```
e1 88 b4
```

The long long type is formatted using the %lld specifier, like this:

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

```
int main()
{
      long long x = LLONG_MIN;

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

The _Complex type has no specifier. Instead, you use the creall and cimagl functions to extract the real and imaginary parts of the complex number. An example:

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

int main()
{
      _Complex long double c = 37.0L + 47.0L * I;

      printf("%Lg + %Lg*I\n", creall(c), cimagl(c));
}
```

The output is:

```
37 + 47*I
```

and the %Lg specifier is used to format long doubles.

Other Printf Specifiers

Another group of printf specifiers is used to handle situations where an integral type is expressed as a typedef, and the underlying type could be signed or unsigned int, long, or long long; size_t is an example. The %u notation specifies an unsigned type, and the z modifier (i.e., %zu) indicates that the type has size_t width, based on local system settings. Here's how you print a size_t value:

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

int main()
{
      size_t x = ~0u;
      printf("%zu\n", x);
}
```

A similar approach is used for the intmax_t types defined in <stdint.h> with the j modifier for %d:

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

int main()
{
      intmax_t x = INTMAX_MAX;
      printf("%jd\n", x);
}
```

The output on my Linux system is:

```
9223372036854775807
```

A third example is the t modifier for the ptrdiff_t type:

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

```
int main()
{
	char a;
	char b;
	char c;
	char d;
	ptrdiff_t x = &d - &a;

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

Here are a couple of other examples of new specifiers. %hh converts the corresponding printf argument to character width, and then formats the value as an integer. For example, the output of this program:

```
#include <stdio.h>

typedef unsigned char UINT8;

int main()
{
	UINT8 a = 100;
	UINT8 b = 200;

	printf("%u\n", a + b);
	printf("%hhu\n", a + b);
}
```

is:

```
300

44
```

In both cases, a + b has a value of 300, passed to printf as an argument. But in the second case, the argument is converted to an unsigned character, and thus has the value 44 (300 mod 256). The %hh specifier is useful for working with short integers, for example types like int8_t defined in <stdint.h>:

```
typedef signed char int8_t;
```

A final example uses the %a specifier to format hexadecimal floating constants:

```
#include <stdio.h>

int main()
{
	float f = 16320;

	printf("%a\n", f);
}
```

The output of this program is:

```
0xf.fp+10
```

In other words:

```
(15 + 15/16) * 2^10 = 16320
```

Scanf Specifiers

Many of the same specifiers used in printf are available in scanf. For example, this program is the inverse of the one just above:

```
#include <stdio.h>

int main()
{
	double d;

	sscanf("0xf.fp+10", "%la", &d);

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

The output of the program is:

```
16320
```

The Snprintf Function

Snprintf is a function much like sprintf, but with the ability to specify a maximum buffer width. Here's an example of snprintf:

```
#include <stdio.h>

void f()
{
	char buf[8];

	//sprintf(buf, "testing %d", 1234);
	//printf("%s\n", buf);

	snprintf(buf, sizeof buf, "testing %d", 1234);
	printf("%s\n", buf);
}
int main()
{
	f();
}
```

When I run this program with the sprintf call uncommented, the result is a segmentation violation, due to buffer overflow. snprintf avoids this problem by allowing you to specify the buffer width.

This particular problem is a major source of security holes: for example, manipulating the amount of buffer overflow such that a stack frame gets overwritten.

Vfprintf

vfprintf, and the related functions vfscanf, vsnprintf, vsprintf, and vsscanf, allow you to pass a variable argument list to the function. Here's an example that defines an error-reporting mechanism:

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

void report_error(const char* file, int line, char* format, ...)
{
	va_list args;
```

```
        va_start(args, format);

        fprintf(stderr, "Error at file %s, line %d: ", file, line);

        vfprintf(stderr, format, args);

        va_end(args);
}
int main()
{
        int x = 37;
        int y = 47;

        if (x < y) {
            report_error(__FILE__, __LINE__,
                "x < y (x=%d y=%d)\n", x, y);
        }
}
```

In this example, I have a report_error function, and I want to pass it a file and line, and also a printf format and a variable number of arguments to be used with the format. Inside report_error, I can set up a variable argument list, and further pass it to the vfprintf function.

The result of running this program is:

```
Error at file vf1.c, line 23: x < y (x=37 y=47)
```

The features we've described above are all useful in writing more portable and secure programs, and in working with new C9X types.