;login:

THE MAGAZINE OF USENIX & SAGE
December 2001 • Volume 26 • Number 8

inside:

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

some new numeric programming features in c9x

We've been looking at some of the changes in the C9X revision of the C language standard. In this column, we'll consider several new features in the numeric programming area.

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

New Integer Types

Suppose that you're doing some C programming and you need to work with 32-bit integers. Which C type should you use for this? You could say:

```
int x;
```

but there's no guarantee that int is 32 bits. With old PCs or embedded applications, it might be 16, and on a high-end workstation or a supercomputer, 64 or 128. A standard way of dealing with this problem is to define a header file with typedefs in it, like this:

```
typedef long INT32;
```

and then use INT32 everywhere.

In C9X, this mechanism has been formalized through the stdint.h header file. Here's a simple example:

```
#include <stdint.h>
#include <stdio.h>
#define N 100

int32_t vector[N];

int main()
{
      for (int i = 0; i < N; i++)
          vector[i] = 0x7fffffff;

      printf("vector[59] = %d\n", vector[59]);

      return 0;
}
```

int32_t is a signed integer type of exactly 32 bits. We can go further in using stdint.h types in this example, and come up with the following:

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

#define N 100

int32_t vector[N];

int main()
{
      for (uint_fast16_t i = 0; i < N; i++)
          vector[i] = INT32_MAX;

      printf("vector[59] = %d\n", vector[59]);

      return 0;
}
```

uint_fast16 is another typedef, specifying an unsigned integer type of at least 16 bits, that is the fastest for your local hardware. The idea is that you know you need at least a 16-bit unsigned type, and you let the system pick the best one for you.

INT32_MAX is a macro that gives the maximum value for a signed 32-bit type.

Another type in stdint.h is intptr_t, a type that is guaranteed to hold a void* pointer, such that you can convert a void* to intptr_t and back, without any loss of information. Here's an example of how you would use intptr_t:

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

int main()
{
    void* p1 = (void*)0x12345678;

    intptr_t saveptr = (intptr_t)p1;

    void* p2 = (void*)saveptr;
    printf("p2 = %lx\n", p2);

    return 0;
}
```

Working with Integer Types

There's another header file, inttypes.h, that provides some utilities for working with the integer types described above. To illustrate these utilities, here's an example that shows how you can use the intmax_t type, a type that specifies the maximum-size integer available on your machine:

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

int main()
{
    intmax_t val = INTMAX_MAX;

    printf("val = %" PRIdMAX "\n", val);

    return 0;
}
```

inttypes.h includes stdint.h, so you don't have to explicitly include it.

The program defines a variable of type intmax_t and sets it to the maximum value. The value is then printed. PRIdMAX is a string defined in inttypes.h that specifies the appropriate printf format for integer ("d") variables of maximum width ("MAX"). On my machine, given that intmax_t is long long, this format is "lld". The three juxtaposed strings in the printf statement are concatenated. Note that the standard requires that intmax_t be at least 64 bits.

Given this ability to specify the printf format in a portable way, we can go back and modify a previous example a little more:

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

#define N 100
```

```
int32_t vector[N];

int main()
{
      for (uint_fast16_t i = 0; i < N; i++)
          vector[i] = INT32_MAX;

      printf("vector[59] = %" PRId32 "\n", vector[59]);

      return 0;
}
```

PRId32 is used to format 32-bit decimal integers.

Another utility offered in inttypes.h is one that lets you do integer division with intmax_t values, obtaining both the quotient and remainder in one operation. Here's an example:

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

int main()
{
      intmax_t num = 987654321;
      intmax_t denom = 123456789;
      imaxdiv_t res = imaxdiv(num, denom);

      printf("quotient = %" PRIdMAX "\n", res.quot);
      printf("remainder = %" PRIdMAX "\n", res.rem);

      return 0;
}
```

There's also a function for taking the absolute value.

inttypes.h also specifies a group of functions that you use to convert strings to intmax_t values. Here's a demo program that shows one of these functions:

```
#include <ctype.h>
#include <inttypes.h>
#include <stdio.h>

int main()

{

      char* input = "1,  123456789123456789,  37,-987654321  ,0,59";
      char* currptr = input;
      char* endptr;

      for (;;) {
          while (*currptr &&
          !(isdigit(*currptr) || *currptr == '-'))
              currptr++;

           if (!*currptr)
              break;
          intmax_t val = strtoimax(currptr, &endptr, 10);
          printf("val = %" PRIdMAX "\n", val); currptr = endptr;
      }

      return 0;
}
```

strtoimax is a function that parses an input string, converts it to an intmax_t value, and returns an updated string pointer so that you can step through the string.

You can specify the number base to strtoimax, as this example shows:

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

int main()
{
      intmax_t val = strtoimax("11111111111111111111", 0, 2);

      printf("val = %" PRIdMAX "\n", val);

      return 0;
}
```

The output from the program is 1048575.

Type Generic Math

Standard math functions like cos typically accept an argument of type double. There are times when you'd like to operate on floats, for space or speed reasons, or on long doubles, to get extra precision. And you might want to use complex types as well, given that C9X supports complex arithmetic.

There are new functions in the standard for working with float and long double and complex types. For example:

cosf	cosine function for float
cosl	cosine function for long double
ccosl	cosine function for complex long double

In addition to these functions, there is a facility defined in tgmath.h, that lets you use a single function (cos) for all these cases, with the "right thing" automatically done for you by the compiler and library; that is, the right function is called based on the argument type. Here's an example that illustrates how this works:

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

int main()
{
      float f = 0.123456;
      double d = 0.234567;
      long double ld = 0.345678;
      complex long double cld = 0.456789;

      if (cos(f) != cos(f))
          printf("cosf error\n");

      if (cos(d) != cos(d))
          printf("cos error\n");

      if (cosl(ld) != cos(ld))
          printf("cosl error\n");

      if (ccosl(cld) != cos(cld))
          printf("ccosl error\n");

      return 0;
}
```

Another way you can look at what's happening with generic dispatching is shown in this demo:

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

int main()
{
        printf("%d\n", sizeof(cos((float)0)));
        printf("%d\n", sizeof(cos((double)0)));
        printf("%d\n", sizeof(cos((long double)0)));

        printf("%d\n", sizeof(cos((complex float)0)));
        printf("%d\n", sizeof(cos((complex double)0)));
        printf("%d\n", sizeof(cos((complex long double)0)));

        return 0;
}
```

On my machine, the sizes of the results of the cos() calls range from 4 (float) to 24 (complex long double), indicating that different versions of the cos function are indeed being called.

Here's another example, using sqrt:

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

int main()
{
        complex double d = 37.0 + 47.0 * I;
        complex double s1 = sqrt(d);
        complex double s2 = csqrt(d);

        printf("%g %g\n", creal(s1), cimag(s1));
        printf("%g %g\n", creal(s2), cimag(s2));

        return 0;
}
```

When you run this program, the result is:

```
6.9576    3.3776

6.9576    3.3776
```

The type generic feature is similar to C++ function overloading and allows for portable code to be written. You can use all of the features we've described above in this way, to write efficient code that's easy to move from one machine and compiler to another.