;login:

THE MAGAZINE OF USENIX & SAGE

April 2001 • volume 26 • number 2

inside:

PROGRAMMING

Using New Types and Values in C9X

USENIX & SAGE

The Advanced Computing Systems Association &
The System Administrators Guild

using new types and values in C9X

by Glen McCluskey

Glen McCluskey is a consultant with 15 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>

If you've followed the development of the C language over time, you're probably aware that there have been some additions to the original ANSI standard for the language. One of these updates was recently approved, and goes by names such as C99, ISO/IEC 9899, and C9X. In the next few columns we'll look at some of the features added to the language, and we'll use the term "C9X" to refer to the new features.

In this column we'll look at several new types added to C9X, and new ways of expressing values of particular types.

Bool

Bool is an integer type used to hold the values 0/1. It represents an obvious way to store false/true values. The keyword _Bool has been added to the language, so you say:

```
_Bool b = 1;
```

or:

```
#include <stdbool.h>

_Bool b = true;
```

Note that _Bool is not a macro defined in a header file but an actual keyword in C9X that is fully incorporated into the C type system. The macros false, true, and bool are defined in stdbool.h as 0, 1, and _Bool, respectively.

What happens when you try to use _Bool in expressions, like this?

```
#include <stdio.h>

int main()
{
    _Bool b = 1;
    b = b + 37;

    printf("%d\n", (int)b);

    return 0;
}
```

The Bool value is converted to an int, and then 37 is added to it. The result, 38, is non-zero, and so the value 1 is stored back into b. This same approach is used for conversions to Bool from other types, for example:

```
char* p;
_Bool b;

...

b = p;
```

This is equivalent to:

```
b = (p != 0);
```

and b gets a value of 0 if p is null, otherwise 1.

You can use Bool in bit fields, like this:

```
struct A {
    _Bool b : 1;
};
```

If you use Bool values in your program and you have, say, an array of values like this:

```
_Bool b[1000];
```

will the values each be stored using only one bit? The answer depends on the compiler you have. For example, a particular compiler might use a whole byte for each value.

Long Long

Long long is another new integer type, guaranteed to hold at least 64 bits. For example, you can say:

```
long long x = 123456789012345LL;
```

or:

```
unsigned long long x = 0xffffffffffffffffull;
```

You print long long values using the ll modifier of printf():

```
printf("%llu\n", 0xffffffffffffffffull);
```

The output is:

```
18446744073709551615
```

Complex and Imaginary

The keywords _Complex and _Imaginary are used in combination with floating types to specify complex data types. There are a total of six such types:

```
_Complex float

_Complex double

_Complex long double

_Imaginary float

_Imaginary double

_Imaginary long double
```

The header file <complex.h> specifies a macro I, that has the value of the imaginary unit, that is, the square root of –1. You initialize a complex value by saying:

```
_Complex float x = 37.0 + 47.0 * I;
```

The imaginary unit has the usual properties, for example:

```
I * I == -1
```

Here's a small program that initializes and multiplies two complex numbers:

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

int main()
{
        _Complex float c1 = 37.0 + 47.0 * I;
        _Complex float c2 = 57.0 + 67.0 * I;
        _Complex float c3 = c1 * c2;

        printf("%g %g\n", crealf(c3), cimagf(c3));

        return 0;
}
```

crealf() and cimagf() are functions used to obtain the real and imaginary parts of a complex number. When this program is run, the output is:

```
–1040 5158
```

In other words, when you multiply:

```
(37 + 47i) * (57 + 67i)
```

you have:

```
37 * 57 + 37 * 67i + 47i * 57 + 47i * 67i
```

or:

```
–1040 + 5158i
```

You can convert complex values to floating values and vice versa. For example, if you say:

```
_Complex double c = 77.0 + 87.0 * I;
double d;

d = c;
```

then d gets the value 77.0, and the imaginary part is discarded.

The C9X standard also specifies functions that operate on complex values, for example catanh() for complex arc hyperbolic tangent.

Hexadecimal Floating Constants

We've looked at several new data types that C9X provides. There are also new features for expressing values of specific types. One of these is the ability to express hex floating constants. For example, you can say:

```
float f = 0xf.fp+10f;
```

which has the value:

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

and:

```
double d = 0x8.0p-3;
```

with the value:

```
(8 + 0) * 2^-3 = 1
```

You print hexadecimal floats using the a specifier in printf():

```
printf("%a\n", 59.0);
```

The output here is:

```
0x1.d8p+5
```

Hex floating literals offer a natural way of expressing certain kinds of floating constants.

Compound Literals

Another new way you can specify values is through the use of compound literals. A compound literal looks like this:

```
(type){value1,value2,...}
```

Here are several examples of compound literals:

```
(int){37}

(int[]){1,2,3}

struct Point {int x, y;};
typedef struct Point Point;
...
(Point){100,200}

(_Complex long double){a + b * I}
```

The programming value of these literals is obvious; you don't have to explicitly name a temporary and initialize it. Here's an example that contrasts the old and new approaches:

```
#include <stdio.h>

struct Point {
      int x;
      int y;
};
      typedef struct Point Point;

      void f(Point p)
{
          printf("%d %d\n", p.x, p.y);
}

int main()
{
      // old way

          Point p = {100,200};
      f(p);

      // new way

          f((Point){100,200});

      return 0;
}
```

Compound literals are not constants and, if used within a function, can be initialized using non-constant expressions. And you can take the address of a literal, like this:

```
Point* ptr = &(Point){x + f(), y + g()};
```

When you use compound literals, it's important to note that each literal creates only a single object in a given scope. For example, in this code:

```
int i;

for (i = 1; i <= 10; i++) {
      Point* ptr = &(Point){i,i+10};
}
```

there's only one literal object, initialized at each loop iteration.

Compound literals give a hint of what some of the most important features in C9X are about – the ability to express a program more naturally. For example, it's quite possible to implement `long long` and complex data types using libraries, but it's more natural to include these features in the language itself.