(NeXT Tip #37a) unsigned int

Christopher Lane (*lane[at]CAMIS.Stanford.EDU*)
Wed, 7 Jul 1993 15:23:31 -0700 (PDT)

Most of the time, when we need an integer in a C program, we just declare a variable of type 'int' or 'char'. These are shorthand for 'signed int' & 'signed char' -- so let's take a look at the alternative, 'unsigned'.

As you expect, an 'unsigned int' can only be a positive number and has a maximum value that is one bit larger (2 * INT_MAX + 1) since conceptually the storage for the sign information can now be used for extending the value.

Does declaring an 'int' to be 'unsigned' mean that I'll get compiler or runtime warnings when/if an attempt is made to set the variable to a negative value? No, not at all. In fact this code fragment:

```
void function()
{
unsigned int x = -1;
...
}
```

as odd as it may seem to someone reading it, will compile and run just fine with no complaints, even with -Wall/lint. It's perfectly legal 'C'. Why use 'unsigned' integer and character types in 'C' if the additional type information doesn't provide greater safety? There are several reasons:

1) As a way of commenting your code to let others know your expectations. For example, most loop incrementing variables shouldn't go negative:

```
unsigned int i;

for(i = 0; i < LIMIT; i++) {
...
}
```

2) You've a algorithm that will generate values greater than <type>_MAX and need to use an unsigned number since it can contain larger positive values. However, if you're really playing in the number space between <type>_MAX and U<type>_MAX, you should consider moving to a larger numeric type, e.g. upgrade
>From 'char' to 'short', 'short' to 'int', etc.

3) You need to control 'sign extension'.

What does 'sign extension' mean? When a signed integer type is assigned to a larger integer type (signed or unsigned) the 'sign' of the number is extended. Thus, if you have a 'signed char' whose value is -1 (0xff), and you assign it to a 'signed int':

```
signed char a = -1;
signed int b;

b = a;
```

The value of the signed int (b) will be -1 (0xffffffff), not 255 (0x000000ff aka UCHAR_MAX). The sign of the number is preserved through sign extension.

If you don't care about the sign of the number -- you're just dealing with bytes from a file, for example -- and you use 'char' instead of 'unsigned char', and subsequently assign that to a larger type, e.g a 'short', then you may end up with unexpected results. The two ways to fix this are either to use an 'unsigned char' to begin with or mask the value on assignment:

```
b = a & SCHAR_MAX; // SCHAR_MAX = maximum signed character value
```

which is probably less elegant and more common in older 'C' code before the 'char' type could be declared as being 'unsigned'.

Here's the result of assigning both a signed and unsigned 'char' type, whose value is -1, to an signed and unsigned 'short' type:

short int = char; signed short int unsigned short int
signed char -1 65535 (USHRT_MAX)
unsigned char 255 (UCHAR_MAX) 255 (UCHAR_MAX)

Sign extension occurs in both cases involving the signed source type regardless of whether it's being assigned to a signed type or not. If we look at this same table for the assignment of a signed and unsigned 'short int' to a signed and unsigned 'int', we see similar results:

int = short int; signed int unsigned int
signed short int -1 4294967295 (UINT_MAX)
unsigned short int 65535 (USHRT_MAX) 65535 (USHRT_MAX)

Can we assume this behavior whenever we assign from a 'smaller' type to a 'larger' type? Well, sort of -- it breaks down a little when we repeat this table for 'int' and 'long int':

long int = int; signed long int unsigned long int
signed int -1 4294967295 (ULONG_MAX)
unsigned int -1 4294967295 (UINT_MAX)

What happened? The trick is that we aren't assigning a shorter type to a longer type -- on most of the 32-bit architectures we deal with, the 'int' and 'long int' types are the same size and so this table represents what you'd expect to see when assigning between signed and unsigned types of the same size. When we move to the next larger type, 'long long int', the results are again what we would expect from sign extension between a small and large type.

Although we normally just see 'int' or 'char' in programs but there are more than thirty different ways to declare an integer (e.g. 'static unsigned long int') in 'C'. Usually you don't need to worry about all the variations but occasionally they do come in useful.

- Christopher

PS: The 'UINT_MAX', etc. symbols are availble from the ANSI <limits.h> file.