



CS107 Lecture 4 Extras

Bits and Bytes Wrap-up, Bitwise Operators

Reading: Bryant & O'Hallaron, Ch. 2.1

Casting Between Signed and Unsigned

What happens at the byte level when we cast between variable types?

The bit patterns remain the same, but **their interpretations are dictated by the type.**

```
int v = -12345;  
unsigned int uv = v;  
printf("v = %d, uv = %u\n", v, uv);
```

Output: **v = -12345, uv = 4294954951.**

Why the difference?

-12345 in two's complement binary is **0b1111111111111111111100111111000111.**

When treated as an unsigned, inherently nonnegative number, **-12345** is **all magnitude.**

Casting Between Signed and Unsigned

C-style casts can be used to on-the-fly reinterpret a value under a different type system, as with:

```
int v = -12345;  
printf("v = %d, uv = %u\n", v, (unsigned int) v);
```

You can also append a **U** to a numeric literal to treat it as an unsigned, as with:

-12345U

Here, the **-12345** is evaluated for its 32-bit, **signed int** bit pattern, but because of that **U**, its bits are then interpreted as an **unsigned int** instead.

Extending Bit Representations

What happens when we initialize a variable of type **int** from a **short**?

```
short s = 4, t = -4;  
int i = s, j = t;
```

Though it's uncommon, C allows it and defines clear rules for how the conversion works, choosing a behavior that's **efficient** for the hardware and **intuitive** for the programmer.

In this case, the easiest thing to preserve the **signed value** is to **sign extend** the bit pattern of the smaller integer to fill the extra bits of the larger one.

				0000 0000	0000 0100	s
				1111 1111	1111 1100	t
0000 0000	0000 0000	0000 0000	0000 0000	0000 0100		i
1111 1111	1111 1111	1111 1111	1111 1100			j

i and **j** are each four-byte quantities that evaluate to 4 and -4, respectively.

By **sign extending** the two-byte representations—that is, by replicating the sign bit to fill up the extra space, we preserve both **magnitude** and **sign**.

Extending Bit Representations

What happens when we initialize an **unsigned int** from an **unsigned short**?

```
unsigned short us = 0b1111111111110010;  
unsigned int ui = us;
```

This is simpler, because we can always fill the extra bits with zeroes. In an unsigned world, leading zero bits leave the value alone.

				us
1111 1111		1111 0010		
0000 0000	0000 0000	1111 1111	1111 0010	ui

Sign and zero extension aren't unique to **shorts**.

longs can be initialized from **ints**, **shorts**, or **chars**.

ints and **shorts** can be initialized from **chars**.

Truncating Bit Representations

What about the opposite? What happens when we initialize a **short** from an **int**?

```
int i = 50000, j = 100000, k = -32769;  
short s = i, t = j, v = k;
```

C can't wedge four bytes of data into two, so it **discards**—or rather, **truncates**—the most significant bits and **retains** the least significant ones.

0000 0000	0000 0000	1100 0011	0101 0000	i	
0000 0000	0000 0001	1000 0110	1010 0000	j	
1111 1111	1111 1111	0111 1111	1111 1111	k	
		1100 0011	0101 0000	s	s is -15536
		1000 0110	1010 0000	t	t is -31072
		0111 1111	1111 1111	v	v is 32767

Comparisons Between Different Types

Note: When comparing signed and unsigned integers. **C will implicitly cast** the signed value to **unsigned** and evaluate the expression assuming both values are **unsigned** and therefore **nonnegative**.

expression	comparison type?	evaluates to?	mathematically accurate?
<code>0 == 0U</code>			

Comparisons Between Different Types

Note: When comparing signed and unsigned integers. **C will implicitly cast** the signed value to **unsigned** and evaluate the expression assuming both values are **unsigned** and therefore **nonnegative**.

expression	comparison type?	evaluates to?	mathematically accurate?
<code>0 == 0U</code>	unsigned	true	yes
<code>-1 < 0</code>	signed	true	yes
<code>-1 < 0U</code>	unsigned	false	nope
<code>2147483647 > -2147483648</code>	signed	true	yes
<code>2147483647U > -2147483648</code>	unsigned	false	nada
<code>-1 > -2</code>	signed	true	yes
<code>(unsigned long)-1 > -2</code>	unsigned	true	yes