CSE 351 Lecture 5 – Integers II

Casting in C

The data type of a variable or expression in C determine the data's behavior – their interpreted values as well as the behavior and validity of different operators.

Type casting (or just "*casting*") in C is the conversion of data of one data type into a different data type. Conversions can include changes in bit-widths (*e.g.* `short` to `int`), changes in interpretations (*e.g.* `int` to `unsigned int`, `long int` to `char*`), or actual changes in representations (*e.g.* `int` to `float`).

An *implicit cast* is done automatically by the compiler when there is a type mismatch in an assignment statement or argument assignment, but there is a well-defined conversion between the two types. Implicit casts will sometimes raise compiler warnings, while incompatible type mismatches will cause compiler errors. An *explicit cast* can be performed by the programmer by prepending the new data type in parentheses to the expression: `(data_type)expression`. Explicit casts can be used to suppress compiler warnings for implicit casts or to forcibly cause changes in interpretation or representation.

Signed vs. Unsigned Integers

Signed and unsigned integers are just two different possible *interpretations* of data – nothing is actually changed in the way that the data is being stored. For example, the byte $0\text{x}80$ can be interpreted as the value 128 (unsigned) or -128 (signed), but the stored data is still $0\text{b}10000000$. Literals/constants can be given an unsigned data type by appending a '`u`' (or '`U`') to the end (*e.g.* `unsigned x = 100u;`).

Arithmetic on fixed-width binary numbers is performed using ***modular arithmetic*** – all bits of the result past the end (*i.e.* to the left) of the fixed-width are dropped. Subtraction is performed via addition (*i.e.* $x - y = x + (-y)$) using the Two's Complement negation technique: $-\boldsymbol{x} = \sim\boldsymbol{x} + \mathbf{1}$.

We define the constants UMin, UMax, TMin, and TMax to be the minimum (Min) and maximum (Max) *values* that a w-bit numeral can take when interpreted as either unsigned (U) or signed (T). These values will always have specific encodings:

Encoding:	$0\text{b}00\ldots0$	$0\text{b}01\ldots1$	$0\text{b}10\ldots0$	$0\text{b}11\ldots1$
Unsigned value:	0 (UMin)	$2^{w-1}-1$	2^{w-1}	$2^{w}-1$ (UMax)
Signed value:	0	$2^{w-1}-1$ (TMax)	-2^{w-1} (TMin)	-1

Casting between signed and unsigned doesn't change the representation of the data, just the interpretation of the data's value and the effect of certain operators. When you mix signed and unsigned values in an expression, the signed values are implicitly cast to unsigned so all operators will default to their unsigned behaviors.

Integer Representation Limitations

Arithmetic overflow occurs when the result of a calculation can't be represented in the current encoding scheme (*i.e.* it lies outside of the representable range of values), resulting in an incorrect value. We differentiate between *unsigned* overflow (*i.e.* result lies outside of [UMin, UMax]) and *signed* overflow (*i.e.* result lies outside of [TMin, TMax]). An indicator of signed overflow is if you add two numbers with the same sign (*i.e.* positive + positive or negative + negative) and the result has the opposite sign.

When you cast from a shorter integer type to a longer type, we must *extend* the old data. ***Zero extension*** pads unsigned data with extra zeros on the left, which preserves the value of the numeral. ***Sign extension*** pads signed data using copies of the most significant bit to preserve the sign and value.

Zero extension: `0b0111 → 0b00000111, 0b1111 → 0b00001111`
Sign extension: `0b0111 → 0b00000111, 0b1111 → 0b11111111`

Bit Shifting

The *shift operators* shift a bit vector in a specified direction by a specified amount (n). This will cause n bits of the original bit vector to "fall off" the end and be lost, while the vacated bits will be filled by either all zeros or all ones, depending on the original vector and the type of shift:

- A ***left shift*** (`x<<n`) will fill with zeros on the right.
- A ***logical right shift*** (`x>>n`, `x` is unsigned) will fill with zeros on the left.
- An ***arithmetic right shift*** (`x>>n`, `x` is signed) will fill with copies of the most significant bit on the left.

Examples for `char x = 0xFF;`:

```
                x<<3 → 0b11111000  // left shift
                x>>3 → 0b11111111  // arithmetic right shift
(unsigned char)x>>3 → 0b00011111  // logical right shift
```

Due to the weighting of the different bit positions for integers, shifting can also be interpreted as either multiplying (left shift) or dividing (right shift) by powers of 2. For example, `char x = 5; char y = x<<2;` results in `y` having the value $0\text{b}00010100 = 20 = 5 \times 2^2$. This means that left shifting can cause overflow!