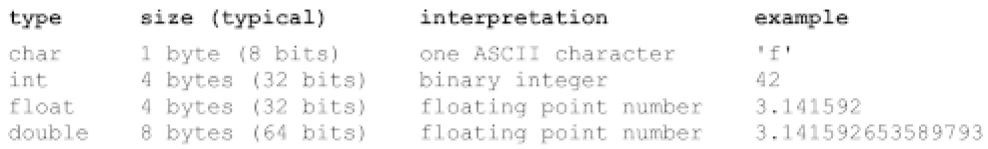
Overflow and Underflow

coursera.org/learn/programming-fundamentals/supplement/BOTAW/overflow-and-underflow

The figure below shows us some of the basic types supported in C and their sizes.



type	size (typical)	interpretation	example
char	1 byte (8 bits)	one ASCII character	'f'
int	4 bytes (32 bits)	binary integer	42
float	4 bytes (32 bits)	floating point number	3.141592
double	8 bytes (64 bits)	floating point number	3.141592653589793

The fact that each type has a set size creates a limit on the smallest and largest possible number that can be stored in a variable of a particular type. For example, a short is typically 16 bits, meaning it can express exactly 2^16 possible values. If these values are split between positive and negative numbers, then the largest possible number that can be stored in a short is 0111111111111111, or 32767.

What happens if you try to add 1 to this number? Adding 1 yields an unsurprising 1000000000000000. The bit pattern is expected. But the interpretation of a signed short with this bit pattern is -32768, which could be surprising (the very popular xkcd comic, illustrates this principle nicely: http://xkcd.com/571/). If the short were unsigned, the same bit pattern 1000000000000000 would be interpreted as an unsurprising 32768.

This odd behavior is an example of overflow: an operation results in a number that is too large to be represented by the result type of the operation. The opposite effect is called underflow in which an operation results in a number that is too small to be represented by the result type of the operation. Overflow is a natural consequence of the size limitations of types.

Note that overflow (and underflow) are actions that occur during a specific operation. It is correct to say "Performing a 16-bit signed addition of 32767 + 1 results in overflow." It is not correct to say "-32768 overflowed." The number -32768 by itself is perfectly fine. The problem of overflow (or underflow) happens when you get -32768 as your answer for 32767 + 1. The operation does not have to be a "math" operation to exhibit overflow. Assignment of a larger type to a smaller type can result in overflow as well. Consider the following code:

```
short int s;

int x = 99999;

s = x;

printf("%d\n", s);
```

In this code, the assignment of x (which is a 32-bit int) to s (which is a 16-bit short int) overflows—the truncation performed in the type conversion discards non-zero bits. This code will print out -31073, which would be quite unexpected to a person who does not understand overflow.

Whether overflow is a problem for the correctness of your program is context-specific. Clocks, for example, experience overflow twice a day without problems. (That 12:59 is followed by 1:00 is the intended behavior). As a programmer, realize that your choice of type determines the upper and lower limits of each variable, and you are responsible for knowing the possibility and impact of overflow for each of these choices.