Types

- a type defines the set of values that an object can have and the operations that can be performed with the object
- see https://en.cppreference.com/w/c/language/arithmetic_types
  - use the *Equivalent type* column when specifying integer types
- the integer types behave similarly to Java
  - but their exact size is not specified by the standard
- the floating-point types behave similarly to Java

<limits.h>

- contains constants related to the limits of the numeric types
  - https://en.cppreference.com/w/c/types/limits
- C standard imposes only three constraints on integer sizes
  - the size of every object is an integer multiple of the size of char
  - each integer type must support a minimum range of specified values
  - smaller types cannot be wider than larger types

Unsigned integers

- unsigned integers have ranges that start at zero and their maximum value is greater than that of the corresponding signed type
  - why?
- have the simplest binary representation

Binary representation

- example: 8-bit unsigned char
  - sum the columns to get the final value of 107

0	**1**	**1**	**0**	**1**	**0**	**1**	**1**	
0×2^7	1×2^6	1×2^5	0×2^4	1×2^3	0×2^2	1×2^1	1×2^0	
0	64	32	0	8	0	2	1	**= 107**

The C standard has no way to specify binary numbers.

- gcc implements an extension for this purpose

```c
#include <stdio.h>

int main(void) {
    unsigned char c = 0b01101011;   // gcc extension
    printf("%c\n", c);              // print as char
    printf("%u\n", c);              // print as integer
}
```

Binary representation

- for an N bit unsigned binary number, the maximum value is $2^N - 1$
  - proof is by induction

Signed integers

- signed integers have ranges that have a most negative value and a most positive value
  - one bit is needed to indicate if the number is
- the C standard does not specify how such numbers should be represented
  - most architectures use two's-complement representation, and it is planned that a future C standard will support only two's-complement representation

Two's-complement representation

- when sign bit is equal to 1, its weight is equal to $-(2^{N-1})$ for an N bit number

0	**1**	**1**	**0**	**1**	**0**	**1**	**1**	
-0×2^7	1×2^6	1×2^5	0×2^4	1×2^3	0×2^2	1×2^1	1×2^0	
0	64	32	0	8	0	2	1	**= 107**

Two's-complement representation

- example: 8-bit signed char
  - sum the columns to get the final value of -21

1	**1**	**1**	**0**	**1**	**0**	**1**	**1**	
-1×2^7	1×2^6	1×2^5	0×2^4	1×2^3	0×2^2	1×2^1	1×2^0	
-128	64	32	0	8	0	2	1	**= -21**

Two's-complement representation

- for almost any integer value x, the two's-complement representation of $-x$ can be found by flipping the bits of x and adding 1

0	0	0	0	0	0	0	1	= 1
1	1	1	1	1	1	1	0	flip bits
1	1	1	1	1	1	1	1	add 1
-1×2^7	1×2^6	1×2^5	1×2^4	1×2^3	1×2^2	1×2^1	1×2^0	
-128	64	32	16	8	4	2	1	= -1

0	**0**	**1**	**1**	**1**	**0**	**0**	**0**	**= 56**
1	**1**	**0**	**0**	**0**	**1**	**1**	**1**	**flip bits**
1	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**add 1**
-0×2^7	1×2^6	0×2^5	0×2^4	1×2^3	0×2^2	0×2^1	0×2^0	
-128	64	0	0	8	0	0	0	**= -56**

1	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**= -56**
0	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**flip bits**
0	**0**	**1**	**1**	**1**	**0**	**0**	**0**	**add 1**
-0×2^7	0×2^6	1×2^5	1×2^4	1×2^3	0×2^2	0×2^1	0×2^0	
0	0	32	16	8	0	0	0	**= 56**

Two's-complement representation

- the range of values is asymmetric around zero
- for an N bit number, the range of values is -2^{N-1} to $2^{N-1} - 1$
- this means that signed integers are susceptible to difficult to detect errors
  - **`-x`** may not exist
  - **`abs(x)`** may not exist
  - **`-1 * x`** may not exist
  - **`x / -1`** may not exist
    - the above are all result in undefined behavior in C

```c
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    signed char c = SCHAR_MIN;     // requires limits.h
    printf("%d\n", c);

     signed char x = -c;
    printf("%d\n", x);

     x = abs(c);                             // requires stdlib.h
    printf("%d\n", x);

     x = -1 * c;
    printf("%d\n", x);

     x = x / -1;
    printf("%d\n", x);
}
```

Floating-point representation

- floating-point values behave similarly to how they behave in Java
- the special floating-point values representing infinity and NaN are defined in **`<math.h>`**
- **`<math.h>`** also declares most of the standard library mathematical functions
  - **`abs`** (integer absolute value) is defined in **`<stdlib.h>`**
    - see https://en.cppreference.com/w/c/numeric/math

```c
#include <math.h>
#include <stdio.h>

// gcc -std=c99 mathdemo.c -o mathdemo -lm

int main(void) {
    double x = sqrt(2.0);     // square root
    printf("%lf\n", x);

    x = pow(x, 2);            // power
    printf("%lf\n", x);

    x = fmax(1.2, 5.7);       // max of two values
    printf("%lf\n", x);
}
```

`printf` for floating-point types

- the `%f` conversion specifier converts a floating-point value to a character string using decimal notation
- the default precision is 6 digits to the right of the decimal place
  - can be changed by including `.`*`int-val`* before the conversion specifier where *`int-val`* is the desired precision (number of digits after the decimal point)

```c
#include <math.h>
#include <stdio.h>

// gcc -std=c99 mathdemo2.c -o mathdemo2 -lm

int main(void) {
    double x = sqrt(2.0);
    printf("%.12f\n", x);

    x = pow(x, 2);
    printf("%.2f\n", x);

    x = fmax(1.2, 5.7);
    printf("%.1f\n", x);
}
```

`-lm`

- compiling a C source code file produces an object file
  - made up of machine code
- if a source code file contains a function call to a function defined in a different file or library, then a program called the *linker* links the object file to the object file or library that contains the function definition
- gcc's **`-l`** (ell, not one) option instructs the linker to link to the specified library
  - **`m`** indicates the standard math library