Basic C Types

Integer Types

Integer types in C represent whole numbers. There are several different variants of integer types (for more information see <limits.h>):

Type	Size (minimum)	Range (minimum)
char	8 bits	signed: -128 to 127 unsigned: 0 to 255
short unsigned short	16 bits	signed: -32768 to 32767 unsigned: 0 to 65535
int unsigned int	32 bits	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long unsigned long	32 bits	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long long (C99)	64 bits	signed: −9223372036854775808 to 9223372036854775807 unsigned: 0 to 18446744073709551615

Note the word **minimum**: Integer sizes are platform dependent

Beyond these minimums, don't make assumptions about the width or ranges of integer types. Historical note: integers haven't always had minimum width 32 bits: they used to be 16 bit!

Architectural Data models and C

Type	Data model			
	LP32	ILP32	P64	LP64
(unsigned) short	16 bits	16 bits	16 bits	16 bits
(unsigned) int	16 bits	32 bits	32 bits	32 bits
(unsigned) long	32 bits	32 bits	32 bits	64 bits
(unsigned) long long	64 bits	64 bits	64 bits	64 bits
all pointers	32 bits	32 bits	64 bits	64 bits

- 32-bit systems:
  - **LP32** or **2/4/4** (int is 16-bit, long and pointer are 32-bit): Win16 API
  - **ILP32** or **4/4/4** (int, long and pointer are 32-bit): Win32 API, Unix, Linux and MAC OS X
- 64-bit systems:
  - **P64** or **4/4/8** (int and long are 32-bits and pointer is 64-bit): Win64 API
  - **LP64** or **4/8/8** (int is 32-bit, long and pointer are 64-bit): Unix, Linux and MAC OS X

Types Information

The C operator `sizeof()` can be applied to any type or value in C to determine its width in bytes
`sizeof()` is used extensively in portable C programs

All we can be sure of is that:

- sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)
- sizeof(char) == 1
- sizeof(long) == sizeof(pointer) == native address bus width (in bytes)

Integers may be signed or unsigned. All integer types are signed by default, except for char - whether char is signed or unsigned is platform-dependent

Signed types: two's complement
Unsigned types: minimum value = 0; maximum value = $2^{size} - 1$

Note that (signed and unsigned) arithmetic **silently wraps round on overflow**, eg. exceeding the max value wraps silently to the min value

`stdint.h`

Sizes and ranges of integer types can vary by platform in C

C99 introduced the header `<stdint.h>` which provides exact width integer types. New advice: use these when you care about width, and use `int` when you don't care:

Type	Size	Range	Printf Spec
uint8_t	8 bits	0 to 255	%u
uint16_t	16 bits	0 to 65535	%u
uint32_t	32 bits	0 to 4294967295	%lu
uint64_t	64 bits	0 to 18446744073709551615	%llu
int8_t	8 bits	-128 to 127	%d
int16_t	16 bits	-32768 to 32767	%d
int32_t	32 bits	-2147483648 to 2147483647	%ld
int64_t	64 bits	-9223372036854775808 to 9223372036854775807	%lld

Floating Point Types

Follow IEEE 754

Type	Size	Range	Precision (decimal)
`float`	32 bits	1.2e-38 to 3.4e+38	6 places
`double`	64 bits	2.3e-308 to 1.7e+308	15 places
`long double`	128 bits	3.4e-4932 to 1.1e+4932	34 places

LONG DOUBLE: 80 bits, not 128 bits!

Some numbers can be represented in floating point exactly: an integer that fits in the mantissa, or a binary fraction such as 0.25

Most decimal fractions (e.g. 0.1) cannot be represented exactly as binary fractions - they are recurring binary fractions

This means that **it is rare to get exact answers from floating point arithmetic**

You should assume that all floating pointer values are approximations.

Floating Point Inaccuracies

Hence: equality often fails

Instead of `x == y`, write `fabs(x-y) < EPSILON`, where `fabs()` comes from `<math.h>`

`#define EPSILON` to a suitable value

Note that when using `<math.h>`, you have to add `-lm` to the gcc link command, in Make and CB terms that means setting `LDLIBS = -lm`

Associativity and commutativity don’t always apply

The limited precision (especially of float) can easily result in a loss of the least significant parts of numbers. eg. adding a big and a small number together

Addition and subtraction are the largest contributors to floating point error, especially when the numbers are very different in magnitude

Multiplication and division generally preserve relative error (but can take us outside the floating point range)