Data types

- what kinds of data can be naturally modeled in the language?
- what kinds of operations are naturally supported on each?
- what does syntax/operator set look like for each type?
- what options are there for internal representations?
- for “specialty” languages, the data types often revolve around the specialization (e.g. strings in bash)
- primitive types: core data types, not implemented in terms of collections or groupings of other types
- composite types: collections or groupings of other types
- user defined types: can user create/name their own types?

Primitive types

- Typical primitive types are integers, real numbers, characters, booleans
- Might be a variety of different integer and real types, supporting different ranges of values or different precision
- Ordinal types: have a finite set of possible values, often subranges of integers or characters, or enumerated value sets
- Strings can be a primitive type in some languages (where they’re not modeled as a collection of characters)

Integer types

- Generally a finite range of possible values for each integer type (e.g. `-32768` to `+32767` for a short, `-2147483648` to `2147483647` for an int, etc)
- host of numerical operations usually supported, +, -, *, etc, often directly using underlying hardware operations (might also include operations like bit shifts, bitwise and, or, xor)
- often support implicit conversions to/from reals, sometimes to/from strings
- Ranges often based on the assumption of a two's complement representation using a fixed number of bits, e.g. with N bits can represent `-2^(N-1)` to `+(2^(N-1))-1`

Compilers and optimizations

- Compiler often recognizes more efficient ways to implement certain optimizations
- e.g. `16*x` might be implemented by shifting `x` left 4 bits
- e.g. `x = 37.5;` might actually store the `37.5` using an integer whose bit pattern matches that for float `37.5`, and moving that “integer” into x’s memory space

Real numbers

- Common representation uses sign bit, fixed number of bits for exponent, fixed number of bits for precision
- `34.75` is `32 + 2 + 1/2 + 1/4`
- abstractly, bit pattern would be `100010.11`, but might be thought of as `1.0001011 x 2^5`
- For 16 bit floats, perhaps use 1 bit for sign, 4 bits for exponent, 11 bits for mantissa
- Thus `0 0101 10001011000`
- Or, treating the mantissa 1 as implicit, use 11 bits for mantissa to get 12 bits of precisions

Real numbers continued

- Usual host of math operations typically supported
- Some might be implemented directly in hardware, e.g. by a floating point unit, others in software
- Compiler responsible for identifying which is available and which to use, as well as any optimizations

Rational numbers

- while often treated as if they were primitive types, rationals often represented as two integers (e.g. in a struct, class, or array) with one part for numerator, one for denominator
- operations include usual math (+, -, *, etc) but typically implemented in software, not hardware
- need to consider whether stored in simplified form or to include a simplification operation, including handling of positive/negative numerator/denominator

Complex numbers

- as with rationals, often represented as a composite with values for real and imaginary components
- as with rationals, operations typically implemented in software, not hardware

“Big” integers and reals

- data types like `bignum` sometimes supported for arbitrary-length numbers
- actually represented as a composite type, e.g. an array or linked list, representing the number in chunks
- requires software implementation of operations (+, -, etc) to match the composite structure

Booleans

- may or may not be its own named type (e.g. bool)
- representations of true, false
- often tied to representation of a core type (C: 0 is false, anything else is true. Lisp: nil is false, anything else is true)
- typical operations: assignment, equality tests, and, or, not
- often supports conversion to/from matching core type (e.g. ints with C)
- theoretically implementation could be single bit, but typically actually uses a byte

Characters

- what characters can be used? often tied to underlying representation: ascii, ebcdid, unicode, etc (fixed vs variable sized character reps?)
- syntax for a char often meant to be distinct from string syntax, e.g. `‘x’`, `#\x`, etc
- operations typically include assignment, equality tests
- testing subsets types? (`isspace`, `isalpha`, `isdigit`, etc)
- conversions? (`toupper`, `tolower`, `char-upcase`, `char-downcase`)
- translation between integer code and associated character? e.g. `(char)(37)`, `(int)(‘x’)`, `(code-char 37)`, `(char-code #\x)`
- ordering comparisons (based on character code? a fixed “agreed” ordering? customizable?)

Data type implementations in C

- C (and hence C++) allows us to lookup the number of bytes needed to store an item of a given datatype using the `sizeof` operator, e.g. `sizeof(short)`, `sizeof(int)`, `sizeof(char)`, `sizeof(double)`, etc
- This also works on user defined types, e.g.

```
typedef struct { int i; float f } MyStructType;
printf("%d\n", sizeof(MyStructType));
```

Internal structure of types

- C also allows us to use the & operator to look up the memory address of items, including elements within an array and fields within a struct, e.g.
- Let’s display the memory address (in hex) of the start of a struct and each of its fields

```
MyStructType s; // had int field i and float field f
printf(“%p, %p, %p\n”, &s, &(s.i), &(s.f));
```

Internal structure of types

- Finally, we can use type casting to display the bit patterns used to store variables, fields, and elements of interest, e.g. typecast something to either a pointer or unsigned integer of the same size, then print it in hex
- Later we'll examine use of unions to investigate deeper
- Together, `sizeof`, `&`, and type casting allow us to investigate the internal storage and data representation of C types in great detail

Example: investigate a struct

```
#include <cstdio>
struct Stype { int i; float f; }
union Cheat { Stype s; int* p; } // both are 8 bytes
int main () {
   Cheat data; // can hold an int* or an Stype
   data.s.i = 25; // store Stype data
   data.S.f = 5.375;
   printf(“%p\n”, (void*)(data.p)); // look at as a ptr
}
// prints 0x40ac000000000019, 4 bytes are f, 4 bytes are i
```