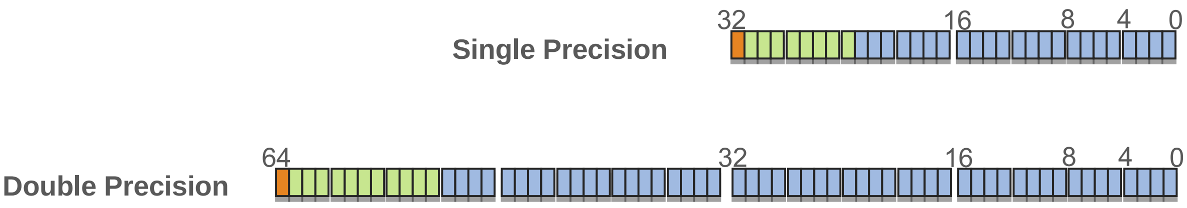
CS 430
Spring 2019

Mike Lam, Professor



Data Types

Type Systems

- Type system
  - Rules about valid types, type compatibility, and how data values can be used
- Benefits of a robust type system
  - Earlier error detection
  - Better documentation
  - Increased modularization

Data Types

- Data type: collection of values and associated operations
  - Descriptor: collection of a variable's attributes, including its type
- Primitive data types
  - Integer, floating-point, complex, decimal, boolean, character
- User-defined data types
  - Structured: arrays, tuples, maps, records, unions
  - Ordinal: enumerations, subranges
  - Pointers and references

Data Types

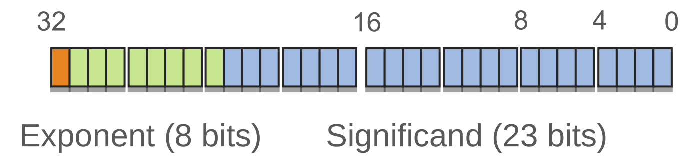
- Primitive data types
  - Integer: signed vs. unsigned, two's complement, arbitrary sizes
    - Tradeoff: storage/speed vs. range
  - Floating-point: IEEE standard (sign bit, exponent, significand), precision, rounding error
    - Tradeoff: storage/speed vs. accuracy, precision vs. range
  - Complex: pairs of floats (real and imaginary)
  - Decimal: binary coded decimal
  - Boolean: 0 (false) or 1 (true); usually byte-sized
  - Character: ASCII, Unicode, UTF-8, and UTF-16 (variable-length), UTF-32 (fixed-length)

IEEE Floating Point

- – Sign bit (s)
- – Exponent (e)
- – Significand (m)

Value:
$(-1)^s \cdot m \cdot 2^e$

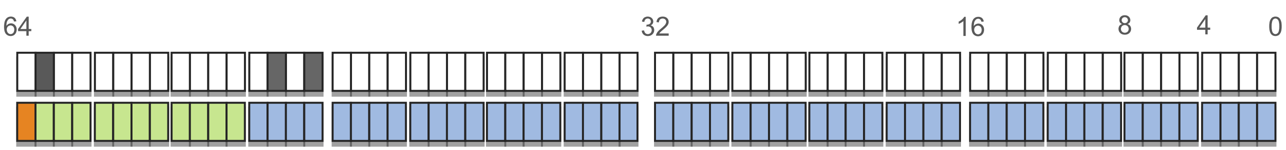
Single Precision



Double Precision

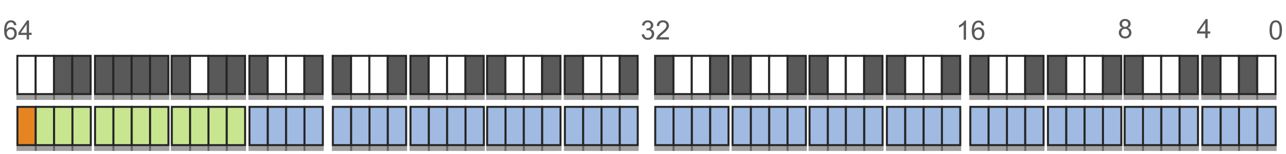
Representing 2.625:

0x4005000000000000



Representing 0.1:

0x3FB999999999999A



User-Defined Data Types

- Structured
  - Arrays and lists: indexed sequences of elements
  - Tuples: fixed-length sequence of elements
  - Associative arrays: mapping from keys to values (often w/ hashing)
  - Records: (name, type) pairs, dot notation, a.k.a. "structs"
  - Unions: different types at runtime, tag/discriminant, safety issues
- Ordinal (value <=> integer mapping)
  - Booleans and characters
  - Enumerations: subset of constants
  - Subranges: contiguous subsequence of another ordinal type

Data Types

- Product vs. sum types
  - Product types: cross product of other types
    - Like a struct in C
  - Sum types: union of other types
    - Like a union in C
  - Haskell examples:

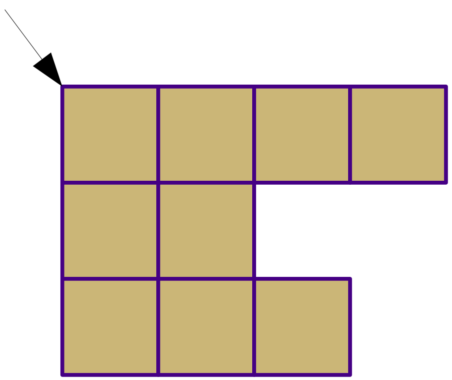
```
data P = P Float Int          -- product type

data S = F Float              -- sum type
       | I Int

data Point2D = Point Float Float

data Point = Point2D Float Float
           | Point3D Float Float Float
```

Arrays and Lists

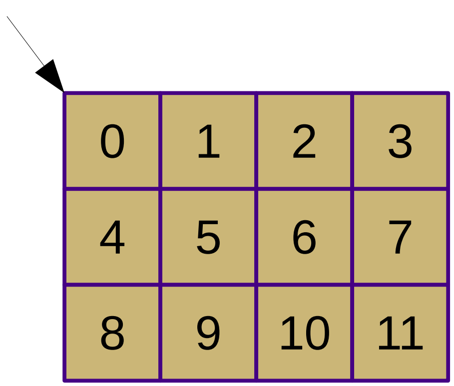
- Arrays
  - Usually homogeneous (with fixed element width)
  - Usually fixed-length
  - Usually static or fixed stack/heap-dynamic
  - Calculating index offsets: `base + index * (element_size)`
- Lists
  - Sometimes heterogeneous
  - Usually variable-length
  - Usually stack-dynamic or heap-dynamic
  - In functional languages: usually defined as `head:tail`

Multidimensional Arrays

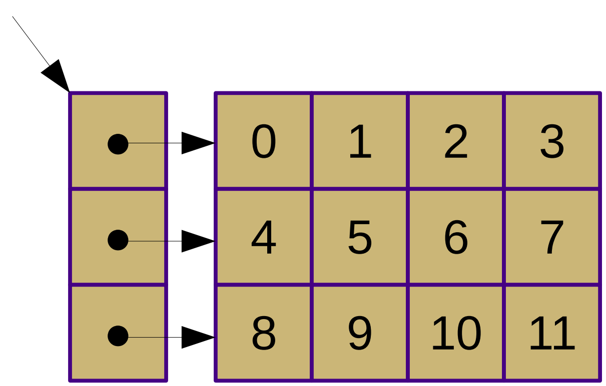
- Multidimensional arrays
  - True multidimensional vs. array-of-arrays
  - Row-major vs. column-major
  - Rectangular vs. jagged
  - Calculating index offsets



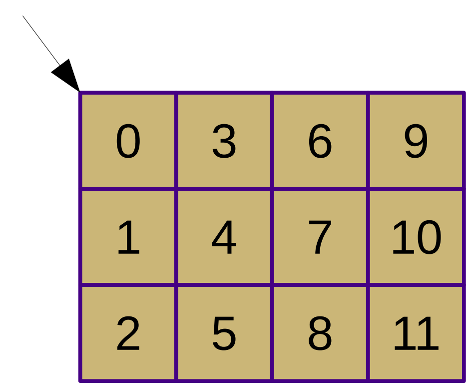
Ragged



Row-major



Row-major arrray-of-arrays



Column-major

Character Strings

- Strings are often stored as arrays of characters
- Common operations: length calculation, concatenation, slicing, pattern matching
- Questions:
  - Should the language provide special support?
  - Should string length be static or dynamic?
    - How should the length be tracked?
  - Should strings be immutable?
- Tradeoffs: speed vs. convenience
- Buffer/length overruns are a common source of security vulnerabilities

Subtypes

- A subtype is a constrained version of an existing type
  - Values of the subtype can often be used in place of the original, but not vice versa
  - E.g., in Ada: `subtype Small_Int is Integer range 0..100;`

Pointers and References

- Pointer: memory address or **null / nil / 0**
  - Example of a nullable type
- Reference: object or value in memory
  - Often can be nullable
  - Different semantics than pointers
  - Strictly safer than pointers
- Implementation
  - Allocation/initialization
  - Dereferencing
  - Arithmetic (allowed for pointers, not references)

Pointers and References

- Design issues
  - Scope and lifetime of pointer and associated value
  - Type restrictions (must match? void* allowed?)
  - Language support (pointers, references, or both?)
- Problems
  - Dangling pointer: value has been deallocated but pointer remains
    - Dereferencing pointer is invalid (might segfault; might not!)
    - Debuggers (e.g., gdb) can help
  - Memory leaks: value is still allocated but no longer accessible
    - In CPL: lost heap-dynamic variables
    - Memory remains allocated; analysis tools (e.g., valgrind) can help

Historical approaches

- Tombstones
  - Extra level of indirection: new access pointer for each object
  - External pointers only point to tombstones
  - When deallocated, tombstone is set to null
  - Causes null pointer dereference if ever used
- Locks and keys
  - Pointers are stored as (key, address) pairs
  - Heap variables store key field as well
  - Pointer key and value key are compared for every reference
  - If the keys do not match, the access is invalid

Downside: high overhead!

Garbage Collection

- Reference counters
  - Track # of references to an object
  - Deallocate object when counter hits zero
- Mark-and-sweep
  - Pause the application (sometimes unnecessary)
  - Initialize indicators for all memory cells to "unmarked"
  - Mark reachable heap memory cells by recursively following pointers from stack and static memory
  - Deallocate unmarked cells
  - Improvements:
    - Generational collection
    - Incremental collection