Data Types

- A data type defines a collection of data values and a set of predefined operations on those values
- Some languages allow user to define additional types
- Useful for error detection through type checking
- Also useful as a documentation tool
- An implementation (compiler or interpreter, plus run-time environment) needs descriptors for data types for type checking, access, allocation, deallocation, etc.

Primitive Data Types

- Primitive Data Types: not defined in terms of other types
- Numeric Types
- Boolean Types
- Character Types

Numeric Types

- Integer: signed vs unsigned, range/size, arbitrary length support
- Floating-point: Most use IEEE Floating-Point Standard 754 format: double is 1 sign bit, 11-bit exponent, 53-bit mantissa
- Floating-point values have both range and precision
- Binary Coded Decimals (each digit may take one byte or a half byte)
- Boolean: some languages have an Boolean type, others allow other numeric values (0 is false, nonzero true)
- Character types: numeric but value indicates character based on some mapping. Most common is 8-bit using ASCII, but "wide" unicode characters are becoming more common.

Character Strings

- Abstraction of sequence of characters
- Issues:
  - Primitive type or a character array, or supported through external libraries?
  - Static vs. dynamic length?
- Common operations:
  - assignment
  - individual character access
  - concatenation
  - substring reference
  - comparison
  - pattern matching

Character Strings

- C and C++ use character arrays (null-terminated) to store strings: unsafe because there are no bound checks
- C++ also have a string class
- Java supports String and StringBuffer (why?). C# and Ruby are similar.
- Python strings are immutable
- Pattern matching are often specified by regular expressions

Character Strings

Three different options for length:

- Static length: length is static and set when the string is created. Suitable for immutable strings, but least flexible.
- Limited dynamic length: allow dynamic length up to some fixed maximum (e.g. C). Somewhat more flexible.
- Dynamic length: C++ string class, etc. Maximum flexibility, requires overhead.

Character Strings

Implementation of descriptor:

- Static: type name, length, and address (compile-time)
- Limited Dynamic: type name, maximum length, current length, address (run-time)
- Dynamic: type name, current length, address (run-time)

Common dynamic length string implementations:

- Dynamic arrays
- Ropes

Enumeration Types

- Allows a related set of constants to be grouped into a type
- e.g. enum in C/C++/C#
- Allows for type checking
- Issues:
  - Is it a separate type, or are conversions to/from integers implicit?
  - Can a constant appear in more than one type?
  - Value and type checking
  - Any other types that can be coerced to an enumeration type?
  - Operations (e.g. arithmetic, bitwise?)

Arrays

- Homogeneous aggregate of data elements, individually identified by position.
- Most languages require elements to be of the same type, but can get around with pointers, inheritance, etc.
- Issues:
  - Subscript types
  - Reference range checking
  - When are subscript ranges bound
  - Allocation time
  - Ragged or rectangular multi-dimensional arrays
  - Initialization?
  - Slices?

Array Indices/Subscripts

- A list of subscripts is used to identify a particular elements.
- Some languages use `[]`, others use `()`. Possible overlap with function calls (e.g. Ada)
- Element type and subscript type are separate
- Some languages allow lower and upper bounds on array subscripts
- Some languages (e.g. Perl) allow negative subscripts—reference from end of array

Subscript Ranges and Storage Binding

The range of subscript values and storage may be determined at compile time or run time.

- Static: subscript ranges and storage are both bound at compile time. Most efficient, but least flexible.
- Fixed stack-dynamic: subscript range is statically bound, allocation done at declaration elaboration time (e.g. local fixed-size arrays). Space efficient (can be reused), require some allocation/deallocation time.
- Fixed heap-dynamic: both are bound at run time, but once the storage is bound the size is fixed. More flexible, but also more overhead
- Heap-dynamic: both are bound at run time, but they can change. Most flexible but also most overhead.

Array Operations

- Some languages allow operations on arrays as a whole (e.g. assignment, concatenation, comparison, etc.)
- Comparison can be based on object or equivalence
- Some languages allow "slices" of arrays which are subsections of the arrays. (e.g. subarray, column, row, every second element of a row). May or may not allocate extra storage.

Multidimensional Arrays

- Rectangular: each row must have the same number of columns
- Jagged: length of each row may be different
- Jagged arrays are usually implemented as arrays of arrays

Implementation

- Descriptor contains type name, element type, index type, lower and upper bounds for indices, and address.
- Some parts are needed at compile time, some at run time
- Bounds are needed only for range checks
- Multidimensional rectangular arrays: usually row major order (notable exception Fortran)
- Address calculations for multidimensional arrays

Associative Arrays

- Unordered collection of data elements indexed by keys
- In some languages associative arrays are native in the language (e.g. Python, Perl, Ruby)
- In some languages it is supported by standard libraries (map and unordered map in C++)
- Implemented by hash tables, binary trees, ...

Record Types

- An aggregate of data elements, identified by field names
- Each element is accessed through offsets from the beginning of structure.
- Issues:
  - how to refer to fields
  - elliptical references?

Record Types

- References to individual fields usually require the name of the record and the name of the fields (. notation in C++, OF in COBOL)
- Nested records usually require a fully qualified reference: `a.b.c.d`
- Some languages (e.g. COBOL) allows elliptical references `d OF a` as long as it is unambiguous. Difficult to read
- Many of the modern languages are block-structured and allows for new local scopes to be created
- Descriptor contains type name, starting address, as well as the name, type, and offset for each field (compile time)
- Memory layout is straightforward

List Types

- Functional programming languages often have native support for lists
- Lists can typically be nested
- Common list operations: car, cdr, cons, list

Union Types

- A variable that may store different type values at different times during run time
- Allows for same memory location to be reused
- Free unions: no type checking performed
- Discriminated union: a tag is added internally to perform type checking (e.g. Ada)
- Space has to be allocated for the largest variant

Pointer Types

- Pointers store memory addresses, with a special `nil` value which is invalid
- Allow for indirect addressing and dynamic storage
- Can be used to access anonymous variables (e.g. those obtained from the heap)
- Issues:
  - scope and lifetime
  - lifetime of heap-dynamic variable
  - restrictions on what type of value they can point to
  - for dynamic storage, indirect addressing, or both?
  - support pointers at all?

Pointer

- Common operations: assignment, dereferencing, allocate/deallocate
- C/C++: pointer arithmetic, array-pointer equivalence, `void *`
- Problems:
  - Dangling pointers
  - Lost heap-dynamic variables (memory leak)

  Removing explicit deallocation may help

Reference Types

- Like a pointer, but a reference type refers to an object or a value in memory
- No need to dereference, no possibility to change address
- One use: pass by reference
- Java does not have pointers, but have references. Need to remember that assignments do not necessarily make a new copy of an object (aliasing)
- Can be implemented by a pointer
- Safer than pointers

Pointer Implementation

- Represented by whatever the machine uses to address memory
- To solve dangling pointer problem:
  - Tombstones: each heap-dynamic variable is itself a pointer to the "real" heap-dynamic variable.
  - Locks-and-keys: each pointer has an address and a key value that is given at time of allocation and copied by assignment operators. Each heap-dynamic variable also has a lock value matching the key.

    Access checks to see if lock and key match.

Heap Management

- If all objects are of the same size, it is easier. Different sizes: may lead to fragmentation
- When to deallocate (implicit):
  - Reference count: small cost for each access/allocation/deallocation, reclaim immediately. Circular structures?
  - Garbage collection: at certain times, an algorithm marks all accessible memory locations. Inaccessible memory locations are deallocated. Run time can be unpredictable and possibly costly.

Type Checking

- Ensures that the operands of an operator/function are of compatible types
- Compatible: either legal, or can be implicitly converted according to the language specification
- Automatic conversion: coercion
- Coercion: convenience vs. error checking?
- Type error: application of an operator/function to an operand of an incompatible type
- Dynamic type checking is needed if type binding is done dynamically
- Difficult in some cases (e.g. union in C/C++)

Type Equivalence

- Type equivalence: one type can be substituted for another, without coercion
- Name type equivalence: only for the same declaration or same type name. Easy to check, more restrictive
- Structure type equivalence: if two variables have identical structures. Harder to check (e.g. circular references), more flexible (too flexible?)
- Some languages use a combination, including support for "subtypes"
- In C++, struct, enum and union use name type equivalence, but arrays use structural type equivalence (sort of)