Composite Types in Detail

- Records (Structures)
- Variants (Unions)
- Arrays
- Strings
- Sets
- Lists
- Pointers and Recursive Types
- Streams (including files, I/O)

Records/Structures

- Records/structures aggregate heterogeneous data types to be stored and manipulated together

- Records typically declared as distinct types (e.g. type `element` in Scott), particularly for name equivalence

- ML does structural equivalence for records quite well, so no explicit type declaration is necessary:

```ml
val copper = { name: "Cu", atomic_number = 29, atomic_weight = 63.546 };
val gold = { name: "Au", atomic_weight = 196.96655, atomic_number = 79 };

(* if you really have to...in this case you will have name equivalence *)
datatype element = element of {atomic_number : int, atomic_weight : real, name : string};
```
Record Fields

• Orthogonality: most languages allow records within records; Fortran 90 and Common Lisp do not

• Accessing fields
  – Dot notation (C, Java, Pascal): record.nested.field
  – Other delimiters (Fortran 90: %): record%field
    • And remember no nested records in Fortran 90
  – Inverted notation: field first
    • Cobol, Algol 68: field of nested of record
    • ML: #field ( #nested record )
    • Common Lisp: (field record)
      – Again, no nested records here

Records and Memory Layout

• Typically contiguous, in the declared field order
  – Alignment issues: the classic time vs. space tradeoff
  – One way to optimize is to rearrange the fields internally, but this will be an issue in systems programming where record layouts are supposed to mirror low-level bit fields

• Pascal: explicit packed keyword to designate preference to save space rather than time

• Memory layout also affects record comparison and assignment
  – Assignment: bit-for-bit transfer (block copy)
  – Comparison: bit-for-bit compare (block compare), maybe, but…how about the holes due to field alignment?
    • Can be tricky — so Pascal and C ditch the whole idea
Scoping Records

- Multiple activities on the fields of the same record can result in lengthy code, particularly for “deep” records

- Pascal: `with` keyword sets up a local scope for a particular record; identifiers for fields can then be addressed “by themselves”
  - How to manipulate two records of the same type within a `with` block?
  - Standalone variables with the same name as record fields

- C/C++: instead of `with`, setup arbitrary pointers whose base type is the record to be manipulated; `->` notation accesses the fields with similar brevity

- Object-oriented languages: functions/methods within classes setup this scope for member/instance variables of those classes

Variants/Unions

- Two or more alternative fields or collections of fields, only one of which is valid at any given time
  - in other words, “memory recycling”

- Tag/discriminant: record field that determines how to interpret the shared memory
  - Tag can be named or anonymous

- Fortran, Algol: `equivalence` — memory sharing for multiple variables, not just within a record

- “Integration” — does the variant section have a name of its own (in C, it does — see Scott p. 359)
Things to Consider with Variants/Unions

- Semantic checks galore!
  - When to use which alternative representation
    - Check the tag/discriminant?
    - What if the tag is anonymous?
      - Algol 68: assignments change “which” alternative is current
    - Compile time or runtime? Sometimes it is impossible to do compile time — implies additional runtime code
  - What about initialization — say we change the tag value on the fly?
- Due to these issues, latest descendants of certain languages have dropped variants/unions: Modula-3, Java
- Primary use today: systems or low-level programming, where bit fields may be interpreted in more than one way
  - Low-level color may be a 4-byte red/green/blue/alpha structure or a single unique 32-bit integer

Arrays

- Homogeneous mappings from a domain of index values to a range of component or element values
- Basic version restricts the index to integers (traceable to origins of how arrays are implemented) — C and descendants even restrict the index range to always start at zero
- Other languages stretch a bit to allow subranges or any enumerated or discrete type for the index
- Most generalized version can map any type to any other type — these associative arrays are sufficiently different in terms of implementation that they are named differently, usually maps (Java, C++)
Array Elements

• Array elements originally scalar (early Fortrans, BASICs)
• Generally today, an array element can be of any type (including other arrays)
• Access by subscript, delimited by parentheses (Fortran, Ada) or square brackets (Pascal, C/C++, Java)
  – Square bracket advantage: avoids overloading of parentheses as delimiters for function arguments
  – …unless you explicitly want to think of array accesses as a form of function call! (Ada)
• Arrays are typically declared with a fixed size
• Arrays of arrays are multidimensional — i.e. a single basic array counts as one dimension (geometric interpretation)
• Pointers and arrays in C/C++ — a special case

Array Slices and Operations

• Array slices are subsets of a larger array; heavily supported in Fortran 90, supported for 1-dimensional arrays in Ada only, largely unsupported in other languages

• Array operations: most of the time, purely access and assignment only (essentially, *get* and *put*)
  – Ada and Fortran 90 allow comparisons and operations over every element of an array (Ada: boolean operators; Fortran 90: arithmetic, mathematical functions)
  – Other languages require explicit looping over elements of an array
  – Array *iterators* in newer languages simplify things somewhat, though they may never reach the succinctness of Fortran 90 array addition:

\[ C = A + B \]
Dimensions, Bounds, Allocation

- Binding again! Array implementations can be distinguished by their “life cycles,” and life cycle implies binding
- For arrays: when to bind to memory, and when to bind their shape (size, dimensions, size per dimension)
  - *global lifetime, static shape*: available throughout life of program, at a fixed shape known at compile time
  - *local lifetime, static shape*: allocated at runtime (local variable), at a fixed shape known at compile time
  - *local lifetime, shape bound at elaboration time*: allocated at runtime, at a fixed shape once allocated, but that shape is not known until runtime
  - *arbitrary lifetime, shape bound at elaboration time*: essentially a heap-allocated array; fixed shape, but known only at runtime
  - *arbitrary lifetime, dynamic shape*: heap-allocated array whose shape can change at any time

Arrays and Memory

- The “first array” — contiguous locations in memory
- For arrays of composite types, memory alignment is an issue just as with records
  - Pascal packed keyword applies to both arrays and records, to similar effect
- How about multidimensional arrays?
  - *Row-major vs. column-major*: is the next element in memory the next array element along the same dimension, or the equivalent array element along the next dimension?
  - Key difference: easier to view a multidimensional array as nested arrays of arrays, since the nesting remains contiguous in memory
  - Performance issues: depending on how array elements are iterated, one arrangement may favor another
- Alternative representation: instead of contiguous blocks of the element type, an array can be contiguous *pointers* to blocks of that type
Arrays and Memory 2: Calculations

• The general approach for contiguous layout, given a known start location in memory, is an accumulated sum of products of indices along each dimension and the size of the element type; pointer-based layout is similar, using the size of a pointer instead of the size of the element type

• But — can we perform this at compile-time or runtime? In other words, can we pre-calculate the memory offset of an array element at compile time?
  – In other words, can A[i, j, k] be automatically compiled as some memory[offset]?
  – Depends on knowledge of array shape
  – Also depends on how array is allocated (statically, on the stack, on the heap) — this determines knowledge of the start location

Arrays and Memory 3: Bounds Checking

• Accesses outside of array bounds are always semantic errors — static vs. dynamic depends on the language

• C/C++ — because arrays and pointers are essentially variations on the same theme, must wait for “Segmentation fault” or similar-sounding errors to realize that you are out-of-bounds…and sometimes you never find out

• In other languages (such as Java), an array “knows” how long it is; more generally, an array “knows” its shape
  – Shape is stored in a dope vector for the array
  – Still, does not completely eliminate dynamic semantic checks
Strings

- Structurally, strings are one-dimensional arrays of characters
  - In many languages, that’s all they are (Pascal, C)
  - Other languages have a specific string type (Java, C++, ML)
  - Either way, even in languages where strings are “just character arrays,”
    they frequently get special handling (Pascal: string literals, C: complete
    string library)

- Strings are a special case, and a *sufficiently* special case:
  - Broad applicability — virtually every program needs to manipulate strings
  - Strings have a stable, consistent structure (one-dimensional array of bytes,
    pre-Unicode; 16 bits per character for Unicode)
    - Makes implementation of string manipulation easier than implementing the
      generalized array manipulation case

String Operations

- String literals: specifying string values directly in code
  - Support for “escape characters” when content is not typable: \n, \t

- String functions — note how they are conceptually generalizable
to arrays of any type, but are difficult to implement that way
  - String length
  - String concatenation
  - String comparisons

- C: string functions are thin wrappers on generalized character
  pointer manipulation
- Java: String is a distinct, full-fledged class
- Other languages are somewhere in between
Sets

- Unordered collection of values for a particular element type, with values appearing only once in the collection
  - Key operations are the same as sets in math: union, intersection, difference

- In the huge, general case, sets are specialized arrays (or maps)
  - Remove access by order
  - Check for duplicate elements

- In special cases, sets can be implemented as a *bit field* and thus can be very fast
  - 1 bit per element in the type’s domain
  - zero if that element is in the set, one if not
  - a zero bit field is the empty set

Pointers and Recursive Types

- Recursive types are types whose definitions include references to themselves (or to types that eventually reference back in a cycle)

- Useful for many data structures: linked lists, trees, graphs

- Ultimately, type recursion leads to some form of reference
  - Implicit reference: Lisp, ML, Clu, Java
  - Explicit reference = *pointer* type: C, Pascal, Ada

- Pointers are *not* addresses
  - Addresses are direct locations in memory
  - Pointers are references to a separate object without being the object itself
    - they *may* be addresses, but not necessarily
Pointers and Allocation

- Pointers can be:
  - a mechanism for heap allocation of objects
  - a mechanism for referring to any other value in a program (individual variables, record fields, array elements)
    - implies availability of an “reference to” operator (C/C++: &; ML: ref)
  - Depending on the language, pointers may be exclusively for heap use; sometimes both
    - Pascal, Ada 83, Modula-3: heap only
    - PL/I, Algol 68, C, C++, Ada 95: heap or reference

- When pointers refer to the heap, the natural follow-up question is how to reclaim heap space
  - Automatic reclamation (“garbage collection“): Lisp, ML, Modula-3, Ada, Smalltalk, Java

Recursive Type Implementation

- Reference model
  - All symbols are references, but can be accessed without additional syntax
  - Thus, recursive types just restate the type
  - An assignment copies a reference, not its value

- Value model
  - Special pointer type explicitly differentiates a reference
  - Recursive types are defined through pointers to that type
  - Subfields/elements of the recursive type must be dereferenced through the pointer using specific syntax (Pascal: ^; C: *, ->)
  - Assignments always copy values
    - Note how this is still consistent with pointer types: assigning one pointer to another does copy that pointer’s value into the other variable
Pointers and Arrays in C

- C is unique in its interchangeable approach to arrays and pointers

- An array is a pointer; multidimensional arrays are $n$-level pointers to pointers

- Dereferencing an array is the same as “moving” a pointer down the array an index number of times

- Pointer arithmetic allows numerical manipulation of pointers; must take into account the size of the pointer’s base type

- When pointers are involved, array layout becomes an issue

Pointer Management

- Dangling references — pointers whose referents are no longer bound (heap deallocation, call stack)
  - Converse of dangling references are memory leaks — referents that are bound in memory but have nothing pointing to them

- Multiple techniques for helping with dangling references
  - Tombstones
  - Lock and key
  - In C++, destructor methods: explicit clean-up code

- Garbage collection techniques: attempting to minimize the harm caused by dangling references and memory leaks
  - Reference counts
  - Mark-and-sweep (heap traversal)
Lists

• Another collection type of a specific base element type, internally implemented as a “lower level” collection such as a specific array or a linked list

• Presented for recursive access as a head and a tail (Lisp, ML); otherwise, not much different from array-like collections
  – Head has base element type
  – Tail is another list

• Operations for list concatenation
• Special literal for the empty list
• Iteration loops are applicable

Streams

• Arrays with “state”
  – “Current element,” whether or not we are at the end of a stream
  – Explicit traversal “up” or “down” the array

• Two directions
  – Input stream — stream from which we get values
  – Output stream — stream to which we put values

• Originated with the specific needs of file I/O, but since generalized (Java, C++)
  – Files remain, now modeled as streams of various types (bytes, characters, strings, entire objects)
  – Blocks of memory (byte arrays, strings) can be viewed as streams
  – Database constructs: tables, query results