Data Types

CSIT 313

Fall 2015

J.W. Benham

Basic Definition

A **data type** is a collection of data values and a set of pre-defined operations on those values

- Original Fortran – integers, reals (floating point numbers) and arrays
- COBOL introduced a structured programmer-defined data type and decimal type
- ALGOL 68 included only a few basic types and some structure-defining operations
- Most languages today include Abstract Data Types
- Functional languages (such as Lisp) feature *lists* as a basic type.

Type System Basics (2)

- Uses of the type system
  - Error detection
  - Program documentation
  - Program modularization
- Variable descriptors
  - Collection of attributes of a variable
  - If all attributes are static – descriptor maintained by compiler, usually as part of symbol table
  - Dynamic attributes – descriptor needed during execution
  - Used for creating code for allocation and deallocation

Primitive Types

- Numeric types
  - Integer
  - Floating point
  - Complex
  - Decimal
- Boolean
- Character
  - ASCII
  - Unicode (UCS-2 and UCS-4 or UTF-32)

Character Strings

- Design Issues
  - Array of characters or primitive type?
  - Static of dynamic length?
- Strings in various languages
  - C and C++: Strings are arrays of characters terminated by the null character (ASCII 0)
  - Java: String class for immutable strings, StringBuilder class for mutable strings
  - Python: Strings are a primitive type – operations to extract chars (strings of length one)
  - Perl, JavaScript, Ruby, PHP strings include built-in pattern matching
- String Length: Static, limited dynamic, and dynamic

String Descriptors

- Static: type name, length, address of first character (needed at compile time)
  Needed only at compile type
- Limited dynamic: fields needed for fixed maximum length as well as current length
  Needed at run time
- Dynamic: only needs a field for current length
  Needed at run time

Memory Allocation for Strings

- Static and limited dynamic length: no special dynamic allocation needed
  - For limited dynamic, just allocate space for maximum length string
- Dynamic length strings
  - Linked list
  - Arrays of *pointers* to individual characters allocated in heap
  - Use adjacent storage cells

Evaluation of String Types

- Essential to writability
- Strings are character arrays can be cumbersome to process
- Adding a primitive string type (or as part of a standard library) does not significantly increase complexity of language or make it significantly more difficult to compile
- Operations such as concatenation and pattern matching are essential
- Dynamic length strings are most flexible, but require significant overhead for their implementation

User-Defined Ordinal Types

- **Ordinal type**: Range of values can be associated with a set of positive integers
  - Examples: integer, boolean, character
- Enumeration types
- Design issues for enumeration types:
  - Is an enumeration constant allowed to appear in more than one type definition. How is type of an occurrence checked?
  - Are enumeration types coerced to integer?
  - Are any other types coerced to an enumeration type?

Enumeration Types in Pascal, C, and C++

- Coerced to integer when put in integer context
- C++ allows conversion of enumeration values to any numeric type, but no other type can be coerced to an enumeration type
- C++ enumeration constants can appear in only one enumeration type in a given referencing environment

Enumeration Types in Ada, Java, and C#

- **Ada**: permits **overloaded literals**
  - Must be able to determine type from context of appearance
  - No coercion to integers – range of operations and range of values restricted
- **Java:** implicitly subclasses of a class **Enum**
  - No coercion to any other type
  - No assignment of any other type to enumeration type variable
  - Method **ordinal** returns numeric value
- **C#:** Enumeration types never coerced to integer
  - Operations restricted to those that make sense

Enumeration Types in ML and F#

- ML – use **datatype** declaration
  - List of values with or operation (|)
- F# - use **type** declaration
  - List of value with or (|) – also required before first value
- None of the recent scripting languages include enumeration types

Evaluation of Enumerated Types

- Enhance readability and reliability
- Reliability:
  - Ada, C#, F#, and Java do not allow arithmetic operations on enumeration types
  - They also do not allow assignment of a value outside the defined range
  - C treats enumeration types as integers and provides neither of these advantages
  - C++ allows assignment of numeric values, but they must be cast to enumeration type – and there is run-time range checking

Subrange Types

- Contiguous subsequence of an ordinal types
- Compiler must generate range-checking code for every assignment to subrange variable
- Enhancement to readability and reliability
- Of contemporary languages, only Ada has subrange types

Array Types

- Homogeneous aggregate of data elements
  - Each individual element identified by its position in the aggregate
- In many languages all elements must be of same type
  - All pointers and references must point to or refer to objects of the same type
  - C, C++, Java, Ada, C#
- In other languages, variables (and, hence, array elements) are type-less references to objects or data values
  - Still homogeneous, since array elements are all of same type

Design Issues for Arrays

- What types are legal for subscripts?
- Are subscript values range checked?
- When are subscript ranges bound?
- When does array allocation occur?
- Are multidimensional or ragged arrays (or both) permitted?
- Can arrays be initialized when their storage is allocated?
- What kinds of array slices (if any) are permitted?

Array Indices

- Array elements referenced by (1) name of array and (2) **selector**
  - Selector comprises one or more **subscripts** or **indices**
  - Selectors can be static (all subscripts constant) or dynamic
  - Selection operation as a **finite mapping**
  - Array subscripts typically enclosed in square brackets or parentheses
  - Permissible subscript types

Subscript Bindings and Array Categories

- Binding of subscript type to array variable usually static
  - Subscript value ranges can be dynamically bound

Array Categories

- Static
- Fixed stack-dynamic
- Stack-dynamic
- Fixed heap-dynamic
- Heap-dynamic

Examples of Array Categories

- Arrays in C/C++
  - **static** modifier (static)
  - without static (fixed stack-dynamic)
  - **malloc** and **free** (in C) or **new** and **delete** (C++) can be used to create fixed heap-dynamic arrays
- Ada allows stack dynamic – can get length from input and create array with that many elements
- Java: all non-generic arrays are fixed heap-dynamic

Examples of Array Categories (2)

- **C#:**
  - Fixed heap dynamic arrays
  - Generic heap dynamic arrays using **List** class
- **Perl:**
  - Grow array using **push** or **unshift** operations
  - Array length is largest subscript + 1
- **JavaScript:** Similar to Perl
  - Negative subscripts not allowed
  - Sparse arrays

Array Initialization

- In some languages, one can initialize arrays at same time storage is allocated
- Fortran 95 and later: denote initialization values in list of literals delimited by parens and slashes
- C family (C, C++, Java, C#) – length determined by compiler (init list in curly braces.)
- C /C++ also permit initializations of strings as in:
  - **char** aName[] = "joe" //
  - **char** ***threeNames[] = { "Larry", "Moe", "Curly"}
- Java array of strings

  String[] names = {"Larry", "Moe", "Shemp"}

Array Initialization (continued)

- Ada has two ways of specifying initialization values:
  - Arr1: **array**(1,6) **of** Integer := (2, 3, 2, 5, 2, 7);
  - Arr2: **array**(1,6) **of** Integer := (2=> 3, 4=>5, 6=>7,
    others => 3);

Array operations

- Operate on array as a unit
  - Assignment, catenation, comparison (for equality)
- C-based languages (C, C++, Java, C#) do not provide any array operations
- Ada allows array assignments and catenation (&)
  - Right side of assignment may be an aggregate value
- Python arrays (called lists) support assignment, catenation (+), element membership (**in**), two different comparison operations (**is** and **==**)
- Ruby arrays are also references to objects
  - Equality operation (==)
  - Catenation via an **Array** method

Array Operations (2)

- Fortran (95 and later): **elemental** operations
  - Library of functions for various vector and matrix operations
- F# **Array** module
- APL – a powerful array-based language
  - Overloaded binary operators
  - Unary operators for vectors and matrices
  - Special operators that take other operators as parameters
    - Example: inner product operator (specified by period)

Array Slices

- An array **slice** is some substructure of array
  - Example: row or column of a matrix
  - Not a new type – mechanism for referencing part of array as a unit
- Slices in Python
- Slices in Perl

Array Type Implementation

- One-dimensional arrays
- Multi-dimensional arrays (that are not arrays of arrays)
- Row major and column major order

Associative Arrays

- Unordered collection of data elements that are indexed by *keys*
  - Number of keys = number of data elements
- Perl hashes
- Python dictionaries
- Ruby – keys can be any object (not just strings)
- PHP has normal & associative arrays
- Lua tables

Record Types

- A **record** is an aggregate of data elements that are identified by names and accessed through offsets from the beginning of the aggregate
  - COBOL records (level numbers)
  - Ada records (nested records)
  - C/C++/C# - **struct** data type
    - ML and F# also include structs

Record Types (2)

- References to record fields
  - COBOL syntax
  - Dot notation (Ada, C, C++)
  - Fully qualified and elliptical references
- Implementation
  - Offset address – relative to beginning of record – associated with each field
  - Compile time descriptor – none needed at run time

Tuple Types

- Similar to records, but without named elements
- Python has an immutable tuple type
  - Access tuple elements with an index
  - Catenation (+) and remove (**del**) operations
- ML tuples
  - Creation and access to element
  - Defining new tuple types
- F# tuples
  - Create by assigning value – list of expressions in parentheses – to a name in a **let** statement
  - Assignment of tuple elements to **tuple pattern** in let statement

List Types

- First supported in LISP
- Included in all functional languages
  - Recently also included in some imperative languages
- LISP lists delimited by parentheses with only spaces between elements
  - Example: **(W X Y Z)**
  - Nested lists possible: **(W (X Y) Z)** denotes a list with three elements: W, the list (X Y), and Z.

List Types (continued)

- List operations in LISP and Scheme
  - CAR and CDR
  - CONS and LIST
- Lists and list operations in ML
  - In square brackets, separated by commas
  - Operations **hd** and **tl** basically
  - Infix operation (**::**) in place of CONS
- Python lists
  - List comprehensions

Union Types

- May store values of different types during different points in program execution
- Design issues
  - Should type checking be required
  - Should unions be embedded in records
- Discriminated and Free Unions

Pointer and Reference Types

- Pointer type – values are memory addresses
  - Special value **nil** (or **null**)
  - Indirect addressing
  - Manage access to locations in dynamically allocated storage (i.e., storage in **heap**)

Pointer Design Issues

- What are the scope and lifetime of a pointer variable?
- What is the lifetime of the heap-dynamic variable (the value referenced by pointer)?
- Are pointers restricted by the type of value they can point to?
- Are pointers used for dynamic storage management, indirect addressing, or both?
- Should the language support pointer types, reference types, or both?

Pointer Operations

- Assignment: Assign an address to a pointer variable.
  - If language permits indirect addressing for variables that are not heap-dynamic , it requires an explicit operation or built-in subprogram to get address
  - If pointers are used only for dynamic storage management (heap-dynamic variables), then the allocation mechanism will serves to initialize pointer variable
- Dereferencing: get value at address that pointer variable references
  - Implicit dereferencing
  - Explicit dereferencing

Pointer Problems

- **Dangling pointer** (dangling reference) contains address of heap-dynamic variable that has been de-allocated
  - This could well be the address of memory location that has since been re-allocated for a different purpose (even if type is the same)
- **A lost heap-dynamic variable** is an allocated heap-dynamic variable that is no longer accessible to program
  - Garbage
  - Memory leakage

Solutions to Pointer Problems

- Dangling Pointers
  - Tombstones
  - Locks and keys
  - Take de-allocation out of the hands of programmers
- Heap Management/Garbage Collection
  - Reference counters
  - Mark-sweep