Data Types

COSC337

by Matt Evett

adopted from slides by Robert Sebesta

Data Types

- Evolution of Data Types:
  - FORTRAN I (1956) - INTEGER, REAL, arrays…
  - COBOL allowed users to define precision
  - ALGOL (1968) provided a few basic types, and allowed user to form new aggregate types.
  - Ada (1983) - User can create a unique type for every category of variables in the problem space and have the system enforce the types

Descriptors

- Def: A descriptor is the collection of the attributes of a variable
  - If all attributes are static, descriptors are required only at compilation time (usually stored within compiler's symbol table)
  - If dynamic, this information must be stored in memory (Lisp does this via *property lists*) and used by run-time system.
- Data type is part of a descriptor.

Primitive Data Types:

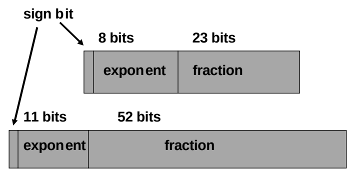
- Primitive Data Types are those types not defined in terms of other data types.
- Integer
  - Almost always an exact reflection of the hardware, so the mapping is trivial
  - There may be as many as eight different integer types in a language
    - Ex: int, long, char, byte

Floating Point

- Model real numbers, but only approximately
- Languages for scientific use support at least two floating-point types
- Usually the representation matches the hardware's, but not always; some languages allow accuracy specs in code
  - e.g. (Ada)
    - type SPEED is digits 7 range 0.0..1000.0;
    - type VOLTAGE is delta 0.1 range -12.0..24.0;

Internal Representation of Floats

- IEEE Floating-Point Standard 754



Decimal

- Common in systems supporting financial applications
- Store a fixed number of decimal digits (coded)
  - Advantage: accuracy
  - Disadvantages: limited range, wastes memory

Boolean

- Could be implemented as bits, but often as bytes
- Advantage: readability

Strings

- Values are sequences of characters
- Design issues:
  - Is it a primitive type or just a special kind of array?
  - Is the length of objects static or dynamic?
- Operations:
  - Assignment
  - Comparison (=, >, etc.)
  - Catenation (or "concatenation")
  - Substring reference
  - Pattern matching (find matching substrings, etc.)

Examples of Strings

- Pascal
  - Not primitive; assignment and comparison only (of packed arrays)
- Ada, FORTRAN 77, FORTRAN 90 and BASIC
  - Somewhat primitive
  - Assignment, comparison, catenation, substring reference
  - FORTRAN has an intrinsic for pattern matching
  - Ada provides catenation (N := N1 & N2) and substrings (N(2..4))

Strings in C

- C
  - Not primitive
  - Use char arrays and a library of functions that provide operations
- C++
  - Provides C-style strings
  - Use of STL class provides "primitive" strings

More String Examples

- SNOBOL4 (a string manipulation language)
  - Primitive
  - Many operations, including elaborate pattern matching
- Perl
  - Patterns are defined in terms of regular expressions
    - A very powerful facility! Similar to Unix's *grep*
    - e.g: /[A-Za-z][A-Za-z\d]+/
- Java
  - String class primitive

String Length Encoding

- String Length Options:
- Static - FORTRAN 77, Ada, COBOL
  - e.g. (FORTRAN 90)
    CHARACTER (LEN = 15) NAME;
  - NAME knows its own length
- Limited Dynamic Length - C and C++ actual length is indicated implicitly by a null character delimiter
- Dynamic - SNOBOL4, Perl, C++ and Java String classes

Utility (of string types):

- Aid to writability
- As a primitive type with static length, they are inexpensive to provide--why not have them?
- Dynamic length is nice, but is it worth the expense?
  - An additional pointer indirection.

Implementing Strings

- Static strings - compile-time descriptor
- Limited dynamic strings - may need a run-time descriptor for current & max length
  - But not in C and C++. Of course there's no index checking provided!
- Dynamic strings - need run-time descriptor; allocation/deallocation is the biggest implementation problem

Ordinal Types

- Def: *ordinal type* = range of possible values can be easily associated with the set of positive integers
  - Enumeration
  - Subrange

Enumerations

- The user enumerates all of the possible values, which are symbolic constants.
  - Design Issue: Should a symbolic constant be allowed to be in more than one enumeration type?
    - No in C, C++, Pascal, because they're implicitly converted into integers.
- Ex (Pascal):

```
type WATER_TEMP = ( Frigid, Cold, Warm, Hot);
var temp : WATER_TEMP; …
if temp > Warm ...
```

Enumerations in Languages

- Pascal
  - cannot reuse constants; they can be used for array subscripts, for variables, case selectors; NO input or output; can be compared.
    - Predecessor and successor functions, loops
- Ada
  - constants can be reused (overloaded literals); disambiguate with context or type_name ' (one of them); can be used as in Pascal; CAN be input and output

Enumeration in Languages (2)

- C and C++
  - like Pascal, except they can be input and output as integers
- Java does not include an enumeration type

Utility (of enumerations)

- Aid to readability–e.g. no need to code a color as a number
- Aid to reliability–e.g. compiler can check operations and ranges of values
  - E.g. Don't have to worry about bad indices:
    - int dailyHighTemp[MONDAY] vs.
    - int dailyHighTemp[1]

Subrange Type

- An ordinal type representing an ordered contiguous subsequence of another ordinal type
- Examples:
  - Pascal
    - Subrange types behave as their parent types; can be used as for variables and array indices

    e.g. type pos = 0 .. MAXINT;

Examples of Subrange Types

- Ada
  - Subtypes are not new types, just constrained existing types (so they are compatible); can be used as in Pascal, plus case constants
  - Ex:

  subtype POS_TYPE is INTEGER range 0 ..INTEGER'LAST;

Utility of subrange types:

- Aid to readability
- Aid to reliability - restricted ranges add error detection

Implementating user-defined ordinal types

- Enumeration types are implemented as integers
- Subrange types are the parent types with code inserted (by the compiler) to restrict assignments to subrange variables

Arrays

- A homogeneous aggregate of data elements in which an individual element is identified by its position in the aggregate, relative to the first.
- Design Issues:
  - What types are legal for subscripts?
  - Are subscripting expressions in element references range checked?
  - When are subscript ranges bound?
  - When does allocation take place?
  - What is the maximum number of subscripts?
  - Can array objects be initialized?
  - Are any kind of slices allowed?

Indexing

- Def: mapping from indices to elements

  map(array_name, index_value_list) : an element
- Syntax
  - FORTRAN, PL/I, Ada use parentheses
  - Most others use brackets
- Subscript Types:
  - FORTRAN, C - int only
  - Pascal - any ordinal type (int, boolean, char, enum)
  - Ada - int or enum (includes boolean and char)
  - Java - integer types only

Categories of Arrays

- (based on subscript binding and binding to storage)
  - Static
  - Fixed stack dynamic
  - Stack-dynamic
  - Heap-dynamic

Static and Fixed Stack

- Static
  - range of subscripts and storage bindings are static
  - e.g. FORTRAN 77, some arrays in Ada, static and global C arrays
  - Advantage: execution efficiency (no allocation or deallocation)
- Fixed stack dynamic
  - range of subscripts is statically bound, but storage is bound at elaboration time
  - e.g. Pascal locals and, C locals that are not static
  - Advantage: space efficiency

Stack-Dynamic

- Range and storage are dynamic, but fixed from then on for the variable's lifetime
- e.g. Ada declare blocks:

```
declare
  STUFF : array (1..N) of FLOAT;
  begin
  ...
  end;
```

- Advantage: flexibility - size need not be known until the array is about to be used

Heap-dynamic

- Subscript range and storage bindings are dynamic and not fixed
- e.g. (FORTRAN 90)

  INTEGER, ALLOCATABLE, ARRAY (:,:) :: MAT

  (Declares MAT to be a dynamic 2-dim array)

  ALLOCATE (MAT (10, NUMBER_OF_COLS))

  (Allocates MAT to have 10 rows and NUMBER_OF_COLS columns)

  DEALLOCATE MAT

  (Deallocates MAT's storage)
- APL & Perl: arrays grow and shrink as needed
- In Java, all arrays are objects (heap-dynamic)

Subscripts

- Number of subscripts
  - FORTRAN I allowed up to three
  - FORTRAN 77 allows up to seven
  - C, C++, and Java allow just one, but elements can be arrays
  - Others - no limit

Array Initialization

- Usually just a list of values that are put in the array in the order in which the array elements are stored in memory
- Examples:
  - FORTRAN - uses the DATA statement, or put the values in / ... / on the declaration
  - C and C++ - put the values in braces; can let the compiler count them
    - e.g. int stuff [] = {2, 4, 6, 8};
  - Ada - positions for the values can be specified

    SCORE : array (1..14, 1..2) := (1 => (24, 10), 2 => (10, 7), 3 =>(12, 30), others => (0, 0));

Slices

- **1. FORTRAN 90**
- `INTEGER MAT (1 : 4, 1 : 4)`
- `MAT(1 : 4, 1)` **- the first column**
- `MAT(2, 1 : 4)` **- the second row**
- **2. Ada - single-dimensioned arrays only**
- `LIST(4..10)`

Implementing Arrays

- **- Access function maps subscript expressions to**
- **an address in the array**
- **- Row major (by rows) or column major order (by**
- **columns)**

Associative Arrays

- **- An *associative array* is an unordered collection of**
- **data elements that are indexed by an equal**
- **number of values called *keys***
- **- *Design Issues*:**
- **1. What is th eform of references to elements?**
- **2. Is the size static or dynamic?**

Chapter 5

5

- Structure and Operations in Perl

- Names begin with `%`
- Literals are delimited by parentheses
  e.g.,

```
%hi_temps = ("Monday" => 77,
             "Tuesday" => 79,…);
```

- Subscripting is done using braces and keys
  e.g.,
  `$hi_temps{"Wednesday"} = 83;`

- Elements can be removed with `delete`
  e.g.,

```
delete $hi_temps{"Tuesday"};
```

Records

A *record* is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names

Chapter 5

Record Definition Syntax

- COBOL uses level numbers to show nested records; others use recursive definitions

Record Field References

1. COBOL
field_name `OF` record_name_1 `OF` ... `OF` record_name_n

2. Others (dot notation)
record_name_1.record_name_2. ... .record_name_n.field_name

***Fully qualified references* must include all record names**

***Elliptical references* allow leaving out record names as long as the reference is unambiguous**

Pascal and Modula-2 provide a `with` clause to abbreviate references

Record Operations

Chapter 5

***Record Operations* (continued)**

2. Initialization
- Allowed in Ada, using an aggregate constant

3. Comparison
- In Ada, = and /=; one operand can be an aggregate constant

4. `MOVE CORRESPONDING`
- In COBOL - it moves all fields in the source record to fields with the same names in the destination record

Comparing records and arrays

1. Access to array elements is much slower than access to record fields, because subscripts are dynamic (field names are static)
2. Dynamic subscripts could be used with record field access, but it would disallow type checking and it would be much slower

Unions

Chapter 5

Design Issues for unions:

1. What kind of type checking, if any, must be done?

2. Should unions be integrated with records?

Examples:

1. FORTRAN - with `EQUIVALENCE`

2. Algol 68 - discriminated unions
- Use a hidden tag to maintain the current type
- Tag is implicitly set by assignment
- References are legal only in conformity clauses (see book example p. 231)
- This runtime type selection is a safe method of accessing union objects

3. Pascal - both discriminated and nondiscriminated unions

```
e.g. type intreal =
       record tagg : Boolean of
```

Chapter 5

Problem with Pascal's design: type checking is ineffective

Reasons:

a. User can create inconsistent unions (because the tag can be individually assigned)

```
var blurb : intreal;
        x : real;
blurb.tagg := true;   { it is an integer }
blurb.blint := 47;    { ok }
blurb.tagg := false;  { it is a real }
x := blurb.blreal;    { assigns an integer
                        to a real }
```

b. The tag is optional!
- Now, only the declaration and the second and last assignments are required to cause trouble

4. Ada - discriminated unions

- Reasons they are safer than Pascal & Modula-2:

Chapter 5

5. C and C++ - free unions (no tags)
- Not part of their records
- No type checking of references

6. Java has neither records nor unions

***Evaluation* - potentially unsafe in most languages (not Ada)**

Sets

A *set* is a type whose variables can store unordered collections of distinct values from some ordinal type

Design Issue:

What is the maximum number of elements in any set base type?

***Examples*:**
1. Pascal

Chapter 5

Examples (continued)

2. Modula-2 and Modula-3
   - Additional operations: `INCL`, `EXCL`, `/` (symmetric set difference (elements in one but not both operands))

3. Ada - does not include sets, but defines `in` as set membership operator for all enumeration types

4. Java includes a class for set operations

Evaluation

- If a language does not have sets, they must be simulated, either with enumerated types or with arrays

- Arrays are more flexible than sets, but have much slower operations

Chapter 5

Pointers

A *pointer type* is a type in which the range of values consists of memory addresses and a special value, nil (or null)

Uses:

1. Addressing flexibility
2. Dynamic storage management

Design Issues:

1. What is the scope and lifetime of pointer variables?
2. What is the lifetime of heap-dynamic variables?
3. Are pointers restricted to pointing at a particular type?
4. Are pointers used for dynamic storage management, indirect addressing, or both?
5. Should a language support pointer types, reference types, or both?

Chapter 5

Problems with pointers:

1. Dangling pointers (dangerous)
   - A pointer points to a heap-dynamic variable that has been deallocated

   - Creating one:
     a. Allocate a heap-dynamic variable and set a pointer to point at it
     b. Set a second pointer to the value of the first pointer
     c. Deallocate the heap-dynamic variable, using the first pointer

2. Lost Heap-Dynamic Variables (wasteful)
   - A heap-dynamic variable that is no longer referenced by any program pointer

   - Creating one:
     a. Pointer `p1` is set to point to a newly created heap-dynamic variable

5

Examples:

1. *Pascal:* used for dynamic storage management only
   - Explicit dereferencing

   - Dangling pointers are possible (`dispose`)

   - Dangling objects are also possible

2. *Ada:* a little better than Pascal and Modula-2
   - Some dangling pointers are disallowed because dynamic objects can be automatically deallocated at the end of pointer's scope

   - All pointers are initialized to null

   - Similar dangling object problem (but rarely happens)

Chapter 5

3. *C and C++*
   - Used for dynamic storage management and addressing

   - Explicit dereferencing and address-of operator

   - Can do address arithmetic in restricted forms

   - Domain type need not be fixed (`void *`)

   e.g.
```
float stuff[100];
float *p;
p = stuff;
```

`*(p+5)` is equivalent to `stuff[5]` and `p[5]`
`*(p+i)` is equivalent to `stuff[i]` and `p[i]`

   - `void *` - can point to any type and can be type checked (cannot be dereferenced)

Chapter 5

4. FORTRAN 90 Pointers
   - Can point to heap and non-heap variables

   - Implicit dereferencing

   - Special assignment operator for non-dereferenced references

   e.g.
```
REAL, POINTER :: ptr  (POINTER is an
                        attribute)
```
`ptr => target` (where `target` is either a pointer or a non-pointer with the `TARGET` attribute))

   - The TARGET attribute is assigned in the declaration, as in:

```
INTEGER, TARGET :: NODE
```

Chapter 5

5. C++ Reference Types
   - Constant pointers that are implicitly dereferenced
   - Used for parameters
     - Advantages of both pass-by-reference and pass-by-value

6. Java - Only references
   - No pointer arithmetic
   - Can only point at objects (which are all on the heap)
   - No explicit deallocator (garbage collection is used)
     - Means there can be no dangling references
   - Dereferencing is always implicit

Evaluation of pointers:

1. **Dangling pointers and dangling objects are problems, as is heap management**