6. Data Types

6.1 Introduction

- Evolution of Data Types
  - FORTRAN I (1957) - INTEGER, REAL, arrays
  - …
  - Ada (1983) - User can create a unique type for every category of variables in the problem space and have the system enforce the types
- A descriptor is the collection of the attributes of a variable
- Design Issues for all data types
  - What is the syntax of references to variables?
  - What operations are defined and how are they specified?

6.2 Primitive Data Types

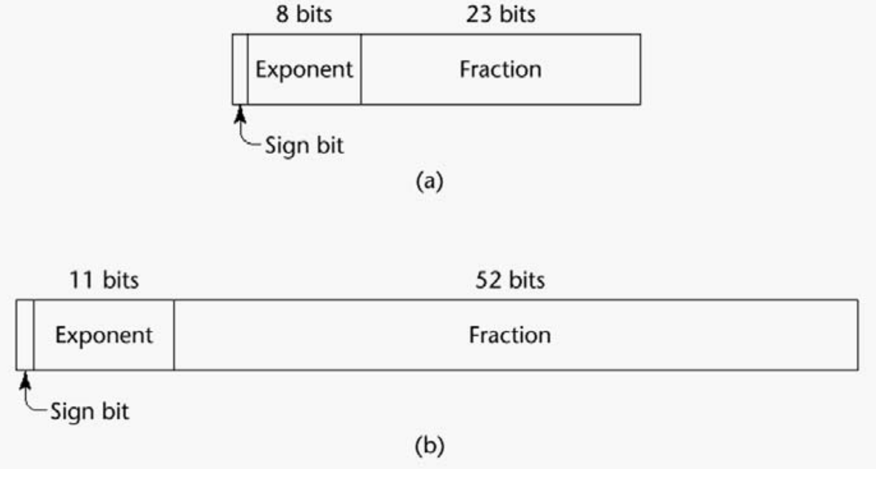
- Those not defined in terms of other data types
- Integer
  - Almost always an exact reflection of the hardware, so the mapping is trivial
  - There may be as many as eight different integer types in a language
- Floating Point
  - Model real numbers, but only as approximations
  - Languages for scientific use support at least two floating-point types; sometimes more
  - Usually exactly like the hardware, 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;
  - See book for representation of floating point (p. 223)

6.2 Primitive Data Types (continued)

- representation of floating point



6.2 Primitive Data Types (continued)

- Decimal
  - For business applications (money)
  - 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
- Character Types
  - ASCII character set
  - Unicode

6.3 Character String Types

- 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
  - Substring reference
  - Pattern matching

6.3 Character String Types (continued)

- Examples
  - Pascal
    - Not primitive; assignment and comparison only (of packed arrays)
  - Ada, FORTRAN 90, and BASIC
    - Somewhat primitive
    - Assignment, comparison, catenation, substring reference
    - FORTRAN has an intrinsic for pattern matching
  - Ada
    - N := N1 & N2 (catenation)
    - N(2..4) (substring reference)
  - C and C++
    - Not primitive
    - Use char arrays and a library of functions that provide operations
  - SNOBOL4 (a string manipulation language)
    - Primitive
    - Many operations, including elaborate pattern matching

6.3 Character String Types (continued)

  - Perl and JavaScript
    - Patterns are defined in terms of regular expressions
    - A very powerful facility!
    - e.g.,
      /[A-Za-z][A-Za-z\d]+/
  - Java - String class (not arrays of char)
    - Objects are immutable
    - StringBuffer is a class for changeable string objects
- String Length Options
  - Static - FORTRAN 77, Ada, COBOL
    e.g. (FORTRAN 90)

    CHARACTER (LEN = 15) NAME;
  - Limited Dynamic Length - C and C++ actual length is indicated by a null character
  - Dynamic - SNOBOL4, Perl, JavaScript

6.3 Character String Types (continued)

- Evaluation (of character 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?
- Implementation
  - Static length - compile-time descriptor
  - Limited dynamic length - may need a run-time descriptor for length (but not in C and C++)
  - Dynamic length - need run-time descriptor; allocation/deallocation is the biggest implementation problem

6.4 User-Defined Ordinal Types

- An ordinal type is one in which the range of possible values can be easily associated with the set of positive integers
- Enumeration Types
  - One in which 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 type definition?
  - Examples
    - Pascal - cannot reuse constants; they can be used for array subscripts, for variables, case selectors; NO input or output; can be compared
    - 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
    - C and C++ - like Pascal, except they can be input and output as integers
    - Java does not include an enumeration type, but provides the Enumeration interface

6.4 User-Defined Ordinal Types

- Evaluation (of enumeration types):
  - Aid to readability - e.g. no need to code a color as a number
  - Aid to reliability - e.g. compiler can check:
    - operations (don't allow colors to be added)
    - ranges of values (if you allow 7 colors and code them as the integers, 1..7, 9 will be a legal integer (and thus a legal color))
- Subrange Type
  - An ordered contiguous subsequence of an ordinal type
  - Design Issue: How can they be used?

6.4 User-Defined Ordinal Types

- Examples
  - Pascal
    - Subrange types behave as their parent types; can be used as for variables and array indices

      e.g. type pos = 0 .. MAXINT;
  - Ada
    - Subtypes are not new types, just constrained existing types (so they are compatible); can be used as in Pascal, plus case constants

      e.g.

      subtype POS_TYPE is INTEGER range 0 ..INTEGER'LAST;
- Evaluation of subrange types
  - Aid to readability
  - Reliability - restricted ranges add error detection
  - Implementation of 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

6.5 Arrays

- An array is an aggregate of homogeneous data elements in which an individual element is identified by its position in the aggregate, relative to the first element
- 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 is a mapping from indices to elements
  - map(array_name, index_value_list) → an element

6.5 Arrays (continued)

- Index Syntax
  - FORTRAN, PL/I, Ada use parentheses
  - Most other languages use brackets
- Subscript Types:
  - FORTRAN, C - integer only
  - Pascal - any ordinal type (integer, boolean, char, enum)
  - Ada - integer or enum (includes boolean and char)
  - Java - integer types only
- Four Categories of Arrays (based on subscript binding and binding to storage)
- Static - range of subscripts and storage bindings are static
  - e.g. FORTRAN 77, some arrays in Ada
  - Advantage: execution efficiency (no allocation or deallocation)

6.5 Arrays (continued)

- Fixed stack dynamic - range of subscripts is statically bound, but storage is bound at elaboration time
  - e.g. Most Java 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

6.5 Arrays (continued)

- 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)
```

  - In APL, Perl, and JavaScript, arrays grow and shrink as needed
  - In Java, all arrays are objects (heap-dynamic)

6.5 Arrays (continued)

- Number of subscripts
  - FORTRAN I allowed up to three
  - FORTRAN 77 allows up to seven
  - 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 e.g. SCORE : array (1..14, 1..2) := (1 => (24, 10), 2 => (10, 7), 3 =>(12, 30), others => (0, 0));
    - Pascal does not allow array initialization

6.5 Arrays (continued)

- Array Operations
  - APL - many, see book (p. 240-241)
  - ADA
    - Assignment; RHS can be an aggregate constant or an array name
    - Catenation; for all single-dimensioned arrays
    - Relational operators (= and /= only)
  - FORTRAN 90
    - Intrinsics (subprograms) for a wide variety of array operations (e.g., matrix multiplication, vector dot product)
- Slices
  - A slice is some substructure of an array; nothing more than a referencing mechanism
  - Slices are only useful in languages that have array operations

6.5 Arrays (continued)

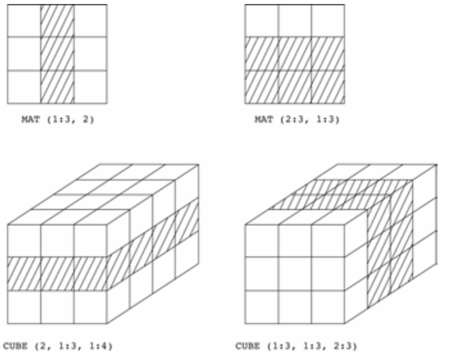
- Slice Examples:
  - FORTRAN 90

    INTEGER MAT (1 : 4, 1 : 4)

    MAT(1 : 4, 1) - the first column

    MAT(2, 1 : 4) - the second row
  - Ada - single-dimensioned arrays only

    LIST(4..10)



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

6.6 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
  - What is the form of references to elements?
  - Is the size static or dynamic?
- 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"};

6.7 Records

- A record is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names
- Design Issues
  - What is the form of references?
  - What unit operations are defined?
- Record Definition Syntax
  - COBOL uses level numbers to show nested records; others use recursive definitions
- Record Field References
  - COBOL
    field_name OF record_name_1 OF ... OF record_name_n
  - Others (dot notation)
    record_name_1.record_name_2. ... .record_name_n.field_name
- Fully qualified references must include all record names

6.7 Records (continued)

- Elliptical references allow leaving out record names as long as the reference is unambiguous
- Pascal provides a with clause to abbreviate references
- Record Operations
  - Assignment
    - Pascal, Ada, and C allow it if the types are identical
    - In Ada, the RHS can be an aggregate constant
  - Initialization
    - Allowed in Ada, using an aggregate constant
  - Comparison
    - In Ada, = and /=; one operand can be an aggregate constant
  - `MOVE CORRESPONDING`
    - In COBOL - it moves all fields in the source record to fields with the same names in the destination record

6.7 Records (continued)

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

6.8 Unions

- A union is a type whose variables are allowed to store different type values at different times during execution
- Design Issues for unions
  - What kind of type checking, if any, must be done?
  - Should unions be integrated with records?
- Examples:
- FORTRAN - with EQUIVALENCE
  - No type checking
- Pascal - both discriminated and nondiscriminated unions, e.g.

```
type intreal =  record
tagg : Boolean of
    true : (blint : integer);
    false : (blreal : real);
end;
```

6.8 Unions (continued)

- Problem with Pascal's design: type checking is ineffective
- Reasons why Pascal's unions cannot be type checked effectively:
  - 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}
```

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

6.8 Unions (continued)

- Ada - discriminated unions
  - Reasons they are safer than Pascal:
    - Tag must be present
    - It is impossible for the user to create an inconsistent union (because tag cannot be assigned by itself--All assignments to the union must include the tag value, because they are aggregate values)
- C and C++ - free unions (no tags)
  - Not part of their records
  - No type checking of references
- Java has neither records nor unions
- Evaluation - potentially unsafe in most languages (not Ada)

6.9 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
  - Pascal
    - No maximum size in the language definition (not portable, poor writability if max is too small)
    - Operations: in, union (+), intersection (*), difference (-), =, <>, superset (>=), subset (<=)

6.9 Sets (continued)

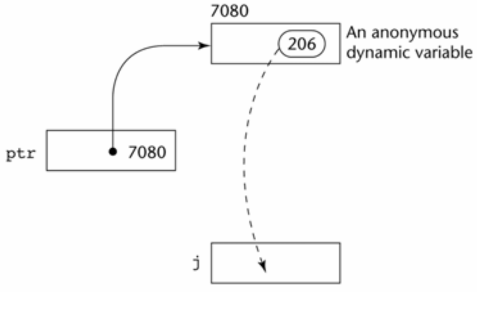
  - Ada - does not include sets, but defines in as set membership operator for all enumeration types
  - Java includes a class for set operations
- Evaluation
  - If a language does not have sets, they must be simulated, either with enumerated types or with arrays
  - Arrays are more flexible than sets, but have much slower set operations
- Implementation
  - Usually stored as bit strings and use logical operations for the set operations

6.10 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
  - Addressing flexibility
  - Dynamic storage management
- Design Issues
  - What is the scope and lifetime of pointer variables?
  - What is the lifetime of heap-dynamic variables?
  - Are pointers restricted to pointing at a particular type?
  - Are pointers used for dynamic storage management, indirect addressing, or both?
  - Should a language support pointer types, reference types, or both?

6.10 Pointers (continued)

- Fundamental Pointer Operations:
  - Assignment of an address to a pointer
  - References (explicit versus implicit dereferencing)
  - The assignment operation j = *ptr



6.10 Pointers (continued)

- Problems with pointers
  - Dangling pointers (dangerous)
    - A pointer points to a heap-dynamic variable that has been deallocated
    - Creating one (with explicit deallocation):
      - Allocate a heap-dynamic variable and set a pointer to point at it
      - Set a second pointer to the value of the first pointer
      - Deallocate the heap-dynamic variable, using the first pointer
  - Lost Heap-Dynamic Variables (wasteful)
    - A heap-dynamic variable that is no longer referenced by any program pointer
    - Creating one:
      - Pointer p1 is set to point to a newly created heap-dynamic variable
      - p1 is later set to point to another newly created heap-dynamic variable
    - The process of losing heap-dynamic variables is called memory leakage

6.10 Pointers (continued)

- Examples:
- Pascal: used for dynamic storage management only
  - Explicit dereferencing (postfix ^)
  - Dangling pointers are possible (dispose)
  - Dangling objects are also possible
- Ada: a little better than Pascal
  - Some dangling pointers are disallowed because dynamic objects can be automatically deallocated at the end of pointer's type scope
  - All pointers are initialized to null
  - Similar dangling object problem (but rarely happens, because explicit deallocation is rarely done)

6.10 Pointers (continued)

- 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]
      (Implicit scaling)
```

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

6.10 Pointers (continued)

- FORTRAN 90 Pointers
  - Can point to heap and non-heap variables
  - Implicit dereferencing
  - Pointers can only point to variables that have the TARGET attribute
    - The TARGET attribute is assigned in the declaration, as in: `INTEGER, TARGET :: NODE`
  - A special assignment operator is used 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))
```

This sets ptr to have the same value as target

6.10 Pointers (continued)

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

6.10 Pointers (continued)

- Evaluation of pointers
  - Dangling pointers and dangling objects are problems, as is heap management
  - Pointers are like goto's
    - they widen the range of cells that can be accessed by a variable
  - Pointers or references are necessary for dynamic data structures
    - so we can't design a language without them

7. Expressions and Assignment Statements