Chapter 6 : DATA TYPES

- Introduction
- Primitive Data Types
- Character String Types
- User-Defined Ordinal Types
- Array Types
- Associative Arrays
- Record Types
- Union Types
- Pointer Types

Introduction

- A data type defines a collection of *data objects* and a set of *predefined operations* on those objects
- 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

Introduction

- Design issues for all data types:
  1. What is the syntax of references to variables?
  2. What operations are defined and how are they specified?

Primitive Data Types

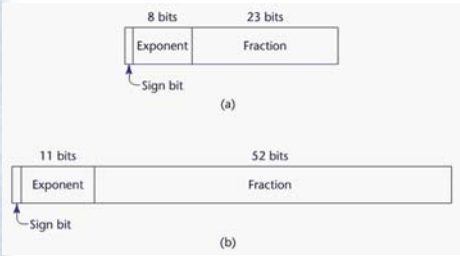
- Those not defined in terms of other data types

1. 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

Primitive Data Types

2. 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

IEEE Floating Point Formats



Primitive Data Types

3. Decimal
   - For business applications (money)
   - Store a fixed number of decimal digits (coded)
   - Advantage: accuracy
   - Disadvantages: limited range, wastes memory

Primitive Data Types

4. Boolean
   - Could be implemented as bits, but often as bytes
   - Advantage: readability
5. Character
   - Stored as numeric codings (e.g., ASCII, Unicode)

Character String Types

- Values are sequences of characters
- Design issues:
  1. Is it a primitive type or just a special kind of array?
  2. Is the length of objects static or dynamic?

Character String Types

- Operations:
  - Assignment
  - Comparison (=, >, etc.)
  - Catenation
  - Substring reference
  - Pattern matching

Character String Types

- 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

  e.g. (Ada)

  `N := N1 & N2` (catenation)

  `N(2..4)` (substring reference)

Character String Types

- 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

Character String Types

- 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 cannot be changed (immutable)
  - **`StringBuffer`** is a class for changeable string objects

Character String Types

- String Length Options:
  1. Static - FORTRAN 77, Ada, COBOL

     e.g. (FORTRAN 90)

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

Character String Types

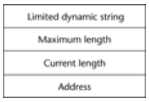
- Evaluation
  - Aid to writability
  - As a primitive type with static length, they are inexpensive to provide--why not have them?
  - Dynamic length is nice, but is it worth the expense?

Character String Types

- 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

Character String Types

Compile-time descriptor for static strings



Run-time descriptor for limited dynamic strings

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

User-Defined Ordinal Types

1. 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?

User-Defined Ordinal Types

- 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); distinguish with context or type_name ‘ (one of them); 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

User-Defined Ordinal Types

- Evaluation (of enumeration types):
  a. Aid to readability--e.g. no need to code a color as a number
  b. Aid to reliability--e.g. compiler can check:
     i. operations (don’t allow colors to be added)
     ii. ranges of values (if you allow 7 colors and code them as the integers, 1..7, then 9 will be a legal integer (and thus a legal color))

User-Defined Ordinal Types

2. Subrange Type
   - An ordered contiguous subsequence of an ordinal type

- Design Issue: How can they be used?

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;`**

User-Defined Ordinal Types

- Examples of Subrange Types (continued)
  - 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;
```

User-Defined Ordinal Types

- Evaluation of subrange types:
  - Aid to readability
  - Reliability - restricted ranges add error detection

User-Defined Ordinal Types

- 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

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.

Arrays

- Design Issues:
  1. What types are legal for subscripts?
  2. Are subscripting expressions in element references range checked?
  3. When are subscript ranges bound?
  4. When does allocation take place?
  5. What is the maximum number of subscripts?
  6. Can array objects be initialized?
  7. Are any kind of slices allowed?

Arrays

- Indexing is a mapping from indices to elements

map(array_name, index_value_list) → an element

- Index Syntax
  - FORTRAN, PL/I, Ada use parentheses
  - Most other languages use brackets

Arrays

- 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

Arrays

- Categories of arrays (based on subscript binding and binding to storage)
  1. Static - range of subscripts and storage bindings are static

     e.g. FORTRAN 77, some arrays in Ada
  - Advantage: execution efficiency (no allocation or deallocation)

Arrays

2. Fixed stack dynamic - range of subscripts is statically bound, but storage is bound at declaration elaboration time
   - e.g. Most Java locals, and C locals that are not `static`
   - Advantage: space efficiency

Arrays

3. 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

Arrays

4. Fixed heap-dynamic - subscript range and storage bindings are dynamic, get fixed after storage allocation (in contrast to declaration elaboration time)

Arrays

5. 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)

Arrays

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

Arrays

- 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

Arrays

- Examples of array initialization:
  1. FORTRAN - uses the DATA statement, or put the values in `/ ... /` on the declaration
  2. C and C++ - put the values in braces; can let the compiler count them

     e.g.

```
int stuff [] = {2, 4, 6, 8};
```


Arrays

- Examples of array initialization:
  3. 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));
```

  4. Pascal does not allow array initialization

Arrays

- Array Operations
  1. APL - many, see book (p. 253)
  2. Ada
     - Assignment; RHS can be an aggregate constant or an array name
     - Catenation; for all single-dimensioned arrays
     - Relational operators (= and /= only)
  3. FORTRAN 90
     - Intrinsics (subprograms) for a wide variety of array operations (e.g., matrix multiplication, vector dot product)

Arrays

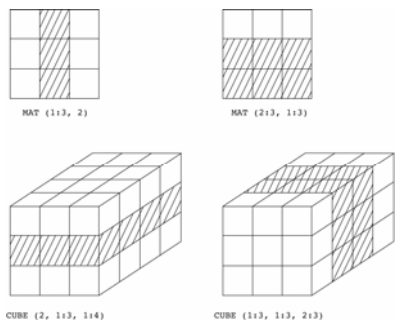
- Slices
  - A slice is some substructure of an array; nothing more than a referencing mechanism
  - Slices are only useful in languages that have array operations

Arrays

- Slice Examples:
  1. FORTRAN 90

```
INTEGER MAT (1:4, 1:4)
  MAT(1:4, 1)  - the first column
  MAT(2, 1:4)  - the second row
```

Example Slices in FORTRAN 90



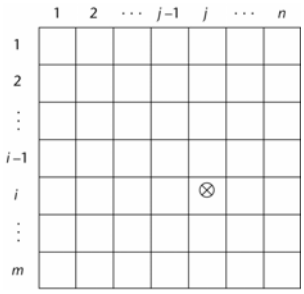
START-> 7/26/06 Arrays

- Slice Examples:
  2. Ada - single-dimensioned arrays only
  `LIST(4..10)`

Arrays

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

Locating an Element



Compile-Time Descriptors

Array
Element type
Index type
Index lower bound
Index upper bound
Address

Single-dimensioned array

Multidimensioned array
Element type
Index type
Number of dimensions
Index range 1
⋮
Index range *n*
Address

Multi-dimensional array

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 the form of references to elements?
  2. Is the size static or dynamic?

Associative Arrays

- 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**

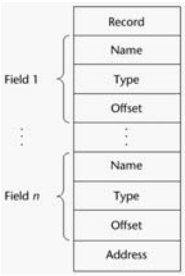
  e.g.,

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


Records

- A record is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names
- Design Issues:
  1. What is the form of references?
  2. What unit operations are defined?

Records



A compile-time descriptor for a record

Records

- Record Operations
  1. Assignment
  - Pascal, Ada, and C allow it if the types are identical
  - In Ada, the RHS can be an aggregate constant
  2. Initialization
  - Allowed in Ada, using an aggregate constant

Records

- Record Operations (continued)
  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

Records

- 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

- A union is a type whose variables are allowed to store different type values at different times during execution
- Design Issues for unions:
  1. What kind of type checking, if any, must be done?
  2. Should unions be integrated with records?

Unions

- Examples:
  1. FORTRAN - with **EQUIVALENCE**
  – No type checking
  2. Pascal - both discriminated and nondiscriminated unions

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

  – Problem with Pascal's design: type checking is ineffective

Unions

- Reasons why Pascal's unions cannot be type checked effectively:
  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 real }
```

Unions

- Examples (continued):
  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?

Pointers

- Problems with pointers:
  1. Dangling pointers (dangerous)
  - A pointer points to a heap-dynamic variable that has been deallocated
  - Creating one (with explicit deallocation):
    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

Pointers

- Problems with pointers (continued):
  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
    b. p1 is later set to point to another newly created heap-dynamic variable
  - The process of losing heap-dynamic variables is called memory leakage

Pointers

- Examples:
  1. Pascal: used for dynamic storage management only
  - Explicit dereferencing (postfix `^`)
  - Dangling pointers are possible (**`dispose`**)
  - Dangling objects are also possible

Pointers

- Examples (continued):
  2. 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)

Pointers

- Examples (continued):
  3. C and C++
  - Used for dynamic storage management and addressing
  - Explicit dereferencing and address-of operator
  - Domain type need not be fixed (`void *`)
  - `void *` - Can point to any type and can be type checked (cannot be dereferenced)

Pointers

3. C and C++ (continued)
- Can do address arithmetic in restricted forms, 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)

Pointers

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

Pointers

- Examples (continued):
  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

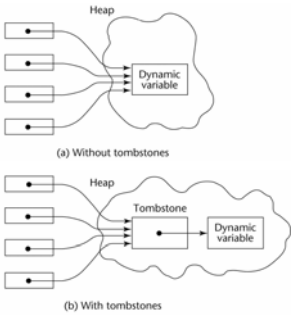
Pointers

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

Pointers

- Representation of pointers and references
  - Large computers use single values
  - Intel microprocessors use segment and offset
- Dangling pointer problem
  1. Tombstone: extra heap cell that is a pointer to the heap-dynamic variable
  - The actual pointer variable points only at tombstones
  - When heap-dynamic variable deallocated, tombstone remains but set to nil

Implementing Dynamic Variables



Pointers

- Dangling pointer problem (continued)
  2. Locks and keys: Pointer values are represented as (key, address) pairs
  - Heap-dynamic variables are represented as variable plus cell for integer lock value
  - When heap-dynamic variable allocated, lock value is created and placed in lock cell (system controlled) and key cell (user program controlled) of pointer

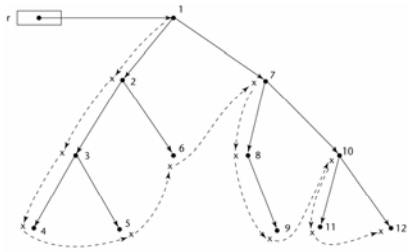
Pointers

- Heap management
  - Single-size cells vs. variable-size cells
  - Reference counters (eager approach) vs. garbage collection (lazy approach)
  1. Reference counters: maintain a counter in every cell that store the number of pointers currently pointing at the cell
  - Disadvantages: space required, execution time required, complications for cells connected circularly

Pointers

- Heap management
  2. Garbage collection: allocate and disconnect until all available cells allocated; then begin gathering all garbage
  - Every heap cell has an extra bit used by collection algorithm
  - All cells initially set to garbage
  - All pointers traced into heap, and reachable cells marked as not garbage
  - All garbage cells returned to list of available cells

Marking Algorithm



Dashed lines show the order of node_marking

Pointers

- Heap management (continued)
  2. Garbage collection (continued):
  - Disadvantages: when you need it most, it works worst (takes most time when program needs most of cells in heap)

☺ Some emerging 'ism's of the new economy ☺