Evolution of Data Types:

FORTRAN I (1956) - 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

Def: A *descriptor* is the collection of the attributes of a variable

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)

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. 199)

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 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?

Operations:

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

e.g. (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

- Perl
  - 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)

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

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

Examples:

- *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

Ordinal Types (user defined)

An *ordinal type* is one in which the range of possible values can be easily associated with the set of positive integers

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?

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

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 operations and ranges of values

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

Design Issue: How can they be used?

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 Enumeration 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;
```

Evaluation of enumeration 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

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:

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?

Indexing is a mapping from indices to elements

map(array_name, index_value_list) Æ 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

Four 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)

2. *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

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

4. *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, arrays grow and shrink as needed
- In Java, all arrays are objects (heap-dynamic)

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:

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};

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));
```

Array Initialization (continued)

4. Pascal and Modula-2 do not allow array initialization

Array Operations

1. APL - many, see book (p. 216-217)

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)

Slices

A slice is some substructure of an array; nothing more than a referencing mechanism

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

2. 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)

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?

- *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 name

Design Issues:

1. What is the form of references?
2. What unit operations are defined?

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

1. Assignment
   - Pascal, Ada, and C allow it if the types are identical
   - In Ada, the RHS can be an aggregate constant

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

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?

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 true : (blint :
         integer); false : (blreal : real);
       end;
   ```

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: a. Tag must be present
     b. 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)

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

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

Implementation

- Usually stored as bit strings and use logical operations for the set operations

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?

Fundamental Pointer Operations:

1. Assignment of an address to a pointer
2. References (explicit versus implicit dereferencing)

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
     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*

\

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)

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)

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

- There is a special assignment when dereferencing is not wanted

  e.g.,

  pointer => target

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
2. Pointers are like goto's--they widen the range of cells that can be accessed by a variable

3. Pointers are necessary--so we can't design a language without them