Chapter 5 - Data Types

- Data types and the characteristics of the common primitive data types (integer, floating-point, etc.)
- Designs of enumeration and subrange types
- Structured data types: arrays, records, and unions.
- Set types
- Pointers

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

Evolution of Data Types

- FORTRAN I (1956) - INTEGER, REAL, arrays, …
  - dynamic structures are all modeled with arrays.
- COBOL - accuracy of decimal data values and structured data type for records of information.
- PL/I - many data types
- ALGOL 68 - a few basic types and few flexible structure defining operators, i.e., user-defined data 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 --> abstract data types
  - use of a type is separated from the representation and set of operations on objects of that type.
  - All of the types provided by a HL PL are abstract dt.

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

Character String Types

Values are sequences of characters

Useful in labeling output, doing input/output, maniupulating 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

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

e.g. (Ada)

- `N := N1 & N2` (catenation)
- `N(2:4)` (substring reference)

But what about assigning and comparing operands of different lengths?

Pattern matching provided as library functions rather than an operation of the language.

- **C and C++**
  - Not primitive
  - terminated with `null`, i.e., zero
    - Library functions that construct strings often supply the null character.
  - Use `char` arrays and a library of functions (string.h) that provide operations

E.g., `char *str = “apples”;`

`strcpy, strcat, strcmp, strlen`

- **SNOBOL4** (a string manipulation lang.)
  - 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

   In F90: CHARACTER (LEN = 15) NAME;
2. *Limited Dynamic Length* - C & C++

   actual length (0..max) is indicated by a null character
3. *Dynamic* - SNOBOL4, Perl

   + no maximum, flexible

   - overhead,i.e.,dynamic de/allocations

Evaluation (of character string types):

- Aid to writability
- As a primitive type with static length, they are inexpensive to provide--*why not have them?*

(What if they are not primitive type?)

- Dynamic length is nice, but is it worth the expense?

Implementation:

Implemented in software or supported directly in hardware

- *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; de/allocation is the biggest implementation problem.
  - linked list + heap: extra storage
  - adjacent storage: frequent moves

Static String
Length
Address

Limited Dynamic String
Maximum Length
Current Length
Address

Dynamic String
Current Length
Address

Ordinal Types

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

In Pascal, the primitive ordinal types are integer, char, and Boolean.

User-defined ordinal types:

- enumeration
- subrange

1. Enumeration Type

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

Common operations

predecessor, successor, position in the list of values, and value for a given position number.

Evaluation (of enumeration types):

1. Aid to readability--e.g. no need to code a color as a number.
2. Aid to reliability--e.g. compiler can check (except in C or C++, think why?)
3. operations and ranges of values

2. Subrange Type

is an ordered contiguous subsequence of an ordinal type.

- Introduced by Pascal.

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;`

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.
  - first value is usually reprepresented as 0.
  - In C and C++, treated exactly like integers.
- Subrange types are the parent types with code inserted (by the compiler) to restrict assignments to subrange variables

  (i.e., range checks in every assignment.)

Arrays

is an aggregate of homogeneous data

- elements of primitive/structured type.
- 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?

Arrays and Indexes(Subscripts)

- *Indexing* is a mapping from indices to elements:

map(array_name, index_value_list) → an element

- Implicit lower subscript bounds in some languages.

Syntax

- FORTRAN, PL/I, Ada use parentheses
  - [ ] were not available at that time!
  - can be easily confused with subprogram calls
    - if an array declaration is missing, compiler cannot detect the error in FORTRAN:
      - checks if it is an array or a locally defined subroutine,
      - else it should be an external subroutine to be linked later! (remember separate compilation)
- Most others use brackets

Types?

element type and subscript type

Subscript Types:

- FORTRAN, C - int only
- Pascal - any ordinal type (int, boolean, char, enum)
- 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

+ execution efficiency (no de/allocation)

2. *Fixed stack dynamic* - range of subscripts is statically bound, but storage is bound at elaboration time

e.g. Pascal locals

C locals that are not static

+ space efficiency (two procedures can share the same large array)

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;
```

+ 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 - *Implementation?*

FORTRAN 90:

```
INTEGER,ALLOCATABLE,ARRAY(:,:):: MAT
ALLOCATE (MAT (10, NUMBER_OF_COLS))
DEALLOCATE MAT
```

APL & Perl: arrays grow and shrink as needed

Java: all arrays are objects (heap-dynamic)

See the comformant arrays in Pascal.

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
4. **Pascal** - does not allow array initialization

Array Operations

An array operation is one that operates on an array as a unit.

1. APL - many, see book (chp.13 or 3rd ed.)

2. Ada

- assignment; RHS can be an aggregate constant or an array name
- catenation; for all single-dim arrays
- relational operators (= and /= only)

3. FORTRAN 90

- elemental operations: operations between pairs of array elements
  - assignment, arithmetic, relational, and logical operators.
- intrinsics or library functions for a wide variety of array operations (e.g., matrix multiplication and transpose, vector dot product)

Slices

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

Design Issue: syntax.

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)
```

Evaluation

- Arrays have been implemented in every imperative language.
  - Since their introduction in FORTRAN I, few improvements were made: ordinal types as subscript types and dynamic arrays.

Implementation of Arrays

- Requires more compile-time effort than implementing primitive types
  - The code to allow accessing of array elements are generated at compile time.
  - This code is executed at runtime to produce element addresses.

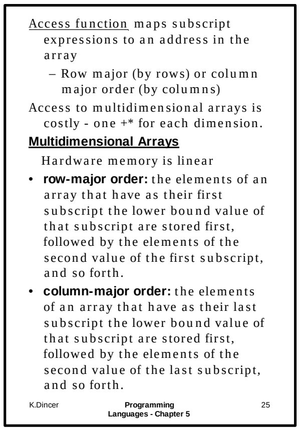
A single-dimensioned array is a list of adjacent memory cells.

`address(list[k]) = . . .`

- if element type & array are statically bound,constant part is precomputed
- if the base is not known until runtime, then . . .

A compile time descriptor is kept to construct an access function.

- If runtime checking of index ranges is not done and the attributes are all static, . . .
- If runtime checking is done . . .
- If the subscript ranges of a particular array type are static, . . .
- If any of the descriptor entries are dynamically bound, . . .



Access function maps subscript expressions to an address in the array

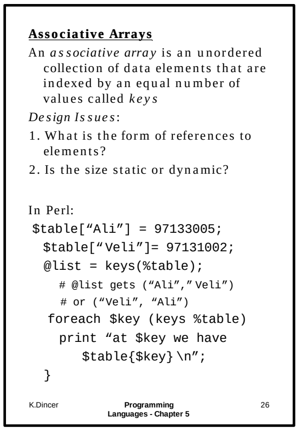
– Row major (by rows) or column major order (by columns)

Access to multidimensional arrays is costly - one +* for each dimension.

Multidimensional Arrays

Hardware memory is linear

- **row-major order:** the elements of an array that have as their first subscript the lower bound value of that subscript are stored first, followed by the elements of the second value of the first subscript, and so forth.
- **column-major order:** the elements of an array that have as their last subscript the lower bound value of that subscript are stored first, followed by the elements of the second value of the last subscript, and so forth.



Associative Arrays

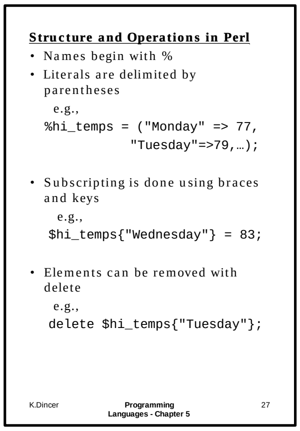
An *associative array* is an unordered collection of data elements that are indexed by an equal number of values called *keys*

Design Issues:

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

In Perl:

```
$table[“Ali”] = 97133005;
  $table[“Veli”]= 97131002;
  @list = keys(%table);
     # @list gets (“Ali”,” Veli”)
     # or (“Veli”, “Ali”)
   foreach $key (keys %table)
     print “at $key we have
         $table{$key} \n”;
  }
```



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

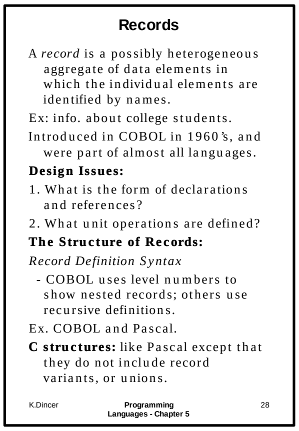
  e.g.,

```
$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.

Ex: info. about college students.

Introduced in COBOL in 1960's, and were part of almost all languages.

Design Issues:

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

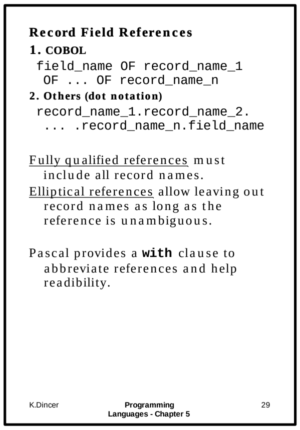
The Structure of Records:

Record Definition Syntax

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

Ex. COBOL and Pascal.

C structures: like Pascal except that they do not include record variants, or unions.



Record Field References

1. COBOL

```
field_name OF record_name_1
 OF ... OF record_name_n
```

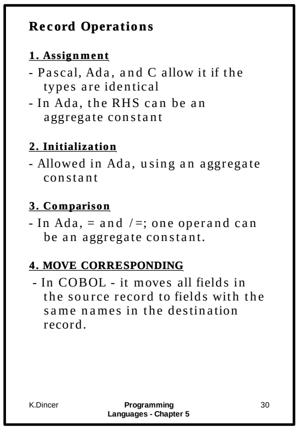
2. Others (dot notation)

```
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 provides a `with` clause to abbreviate references and help readibility.



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

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.

Evaluation.

1. Valuable data types
2. Straightforward design and safe usage.
3. Except elliptical references they increase readability.

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.

Implementation

Fields are stored in adjacent memory locations, an offset address is associated with each field.

See Figure.

Unions

A *union* is a type whose variables are allowed to store different type values at different times during execution.

Ex: a table of constants for a compiler

Design Issues for unions:

1. What kind of type checking, if any, must be done?
2. Should unions be integrated with (embedded in) records?

Examples:

1. FORTRAN - with EQUIVALENCE

```
INTEGER X
REAL  Y
EQUIVALENCE (X, Y)
```

specifies that both X and Y are to cohabit the same storage location, i.e., they are aliases.

2. Algol 68 - discriminated unions

- Use a hidden tag (discriminant) to maintain the current type
- Tag is implicitly set by assignment
- References are legal only in conformity clauses (see book example p. 231, 4th Ed., p.224, 3rd Ed.)
- This static type checking and *runtime type selection (and so runtime type detection)* is a safe method of accessing union objects.

3. Pascal - both discriminated (called *record variant*) & nondiscriminated unions. See Ex. in book.

e.g.

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

Problem with Pascal's design:

- unsafe - 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; {is an integer}
 blurb.blint := 47;   { ok }
 blurb.tagg := false;{ is a real }
 x := blurb.blreal; {assigns an
                 integer to a real }
```

b. The tag is optional! - Free union

Now, only the declaration and the second and last assignments are required to cause trouble.

Variant records in Pascal are often used to get around some of the restrictions in the language. Ex: pointer arithmetic.

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)

Implementation - see Figure.

Set Types

A *set* is a type whose variables can store unordered collections of distinct values from some ordinal type, called its *base 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
  - usually sets are implemented as bit strings that fit into a single machine word - not portable!
  - poor writability if max is too small
- Operations: intersection (*), union (+), difference (-), =, <>, superset (>=), subset (<=), in.

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, user must write code to simulate them, 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.

Ex: ['a'..'p'] ⇒ 16-bit machine word, a **1** representing a present element. Membership, union, etc. is done in one machine instruction.

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 - indirect addressing
2. Dynamic storage management.

Dynamic Variables: variables that are dynamically allocated from the heap (often have no identifiers)

Anonymous Variables: variables without names.

Some Remarks on Pointers

- not structured types.
- not scalar variables.
- add writability to the language. (compare a binary tree implementation in C and Fortran)

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)

PL/I is the first language that included pointer variables.

Assignment.

Sets a pointer variable to the address of some object.

If pointers are used

- only to manage dynamic storage,

  the allocation mechanism serves to initialize the pointer variable.
- for indirect addressing to non-dynamic variables,

  address of a variable must be fetched and assigned to pointer variable explicitly.

Explicit operator or built-in subprograms are used in both cases.

Interpretation of occurrence of a pointer variable in an expression:

- as an address: As in nonpointer variables, it refers to the contents of the memory cell to which the variable is bound
- as an indirect reference: dereference the pointer to obtain the value.

Dereferencing.

takes a reference through one level of indirection.

Two types:

1. Implicit dereferencing.

   In Algol 68 and F90.
2. Explicit dereferencing.

   ^ in Pascal and * in C/C++

Pointers to Records:

In C and C++:

- (*p).name
- p->name (-> operator combines dereferencing and field reference)

In Pascal:

- p^.name

In Ada:

- p.name (implicit derefencing)

Heap Management:

C: built-in subprogram, alloc/free

C++: operators, new and delete

Problems with Pointers:

1. Type Checking

Domain type: the type of object to which a pointer can point.

Ex. PL/I is not restricted to single domain type, most modern languages are.

2. Dangling Pointers (dangerous)

A pointer points to a heap-dynamic variable that has been deallocated.

Why dangerous?

Creating one - (Method 1)

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

Creating one - (Method 2)

Declare a pointer variable in a larger scope than that of the object to which it points.

This extends the lifetime of the pointer past the lifetime of the object to which it points.

3. Lost Objects (Lost Heap-Dynamic Variables) (wasteful)

A heap-dynamic variable that can no longer be 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*.

4. Examples

- **Pascal***:* used for dynamic storage management only
  - Explicit dereferencing
  - Dangling pointers are possible (dispose)
  - Dangling objects are also possible

Dangling pointer problem exist in all languages with explicit deallocation.

Alternatives for implementing explicit deallocation in Pascal:

- ignore dispose
- exclude dispose in language definition
- deallocate by leaving dangling pointers behind.
- Implement dispose completely and correctly.

- **C and C++.** Used for dynamic storage management and addressing.
  - – Explicit dereferencing and address-of operator
  - – 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]`

– Domain type need not be fixed (void *)

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

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.

Implementation of Pointers:

In most computers, pointers are single values stored in either two- or four-byte memory cells.

In PCs with Intel chips, addresses are of two parts: a segment and an offset.

Associated pointer variables use a pair of 16-bit words.