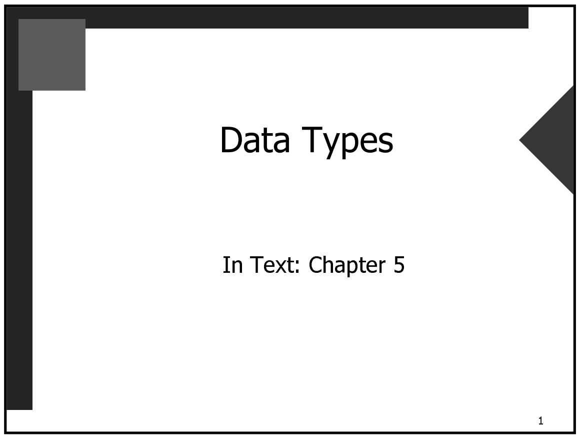
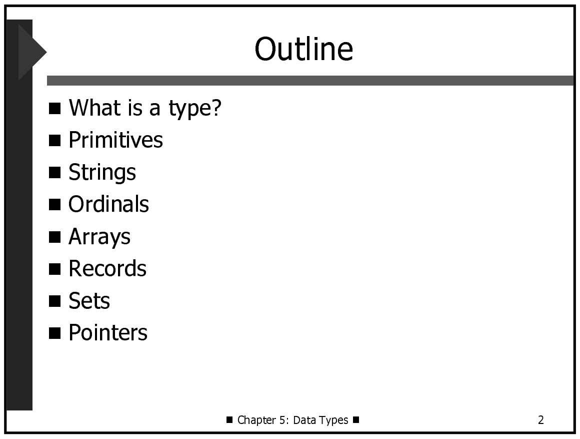


Data Types

In Text: Chapter 5

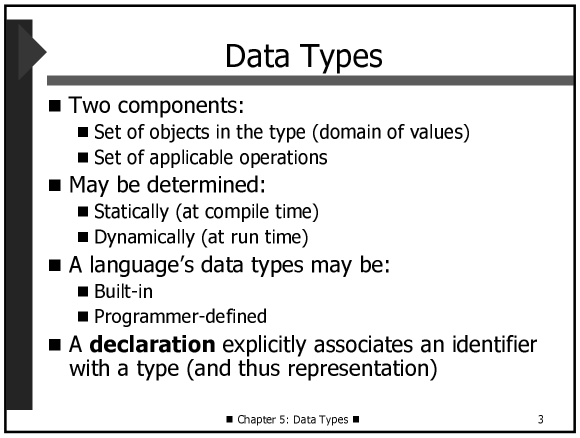
1



Outline

- What is a type?
- Primitives
- Strings
- Ordinals
- Arrays
- Records
- Sets
- Pointers

■ Chapter 5: Data Types ■ 2



Data Types

- Two components:
  - Set of objects in the type (domain of values)
  - Set of applicable operations
- May be determined:
  - Statically (at compile time)
  - Dynamically (at run time)
- A language's data types may be:
  - Built-in
  - Programmer-defined
- A **declaration** explicitly associates an identifier with a type (and thus representation)

■ Chapter 5: Data Types ■ 3

Design Issues for All Data Types

- How is the domain of values specified?
- What operations are defined and how are they specified?
- What is the syntax of references to variables?

Primitive Data Types

- A **primitive** type is one that is not defined in terms of other data types
- Typical primitives include:
  - Boolean
  - Character
  - Integral type(s)
  - Fixed point type(s)
  - Floating point type(s)

Boolean

- Used for logical decisions/conditions
- Could be implemented as a bit, but usually as a byte
- Advantage: readability

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
- Each such integer type usually maps to a different representation supported by the machine

Fixed Point (Decimal) Types

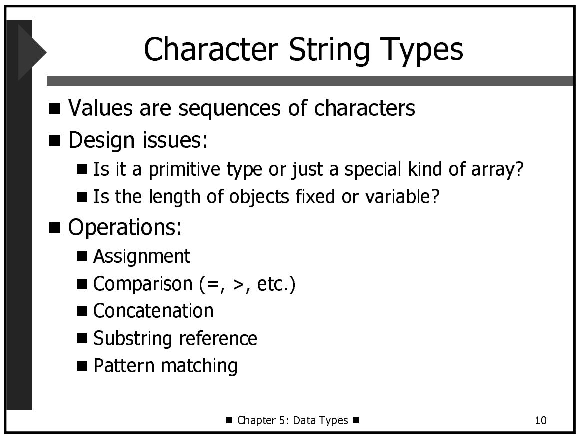
- Originated with business applications (money)
- Store a fixed number of decimal digits (coded)
- Advantage: accuracy
- Disadvantages: limited range, wastes memory

Floating Point Types

- Model real numbers, but only as approximations
- In any particular numeric domain, the **model numbers** are the set of exactly representable numbers
- 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):

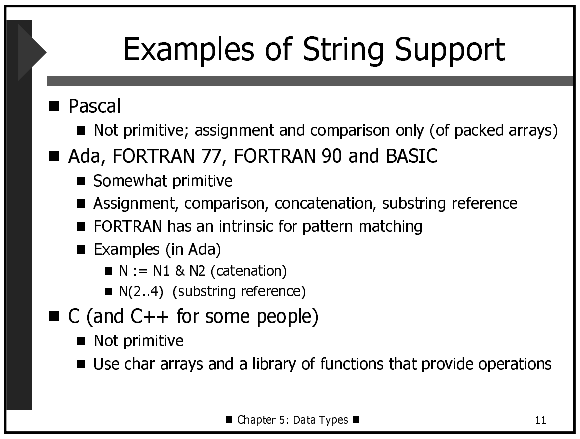
```
  type Speed is
      digits 7 range 0.0..1000.0;
   type Voltage is
      delta 0.1 range -12.0..24.0;
```

- See book for representation (p. 199)



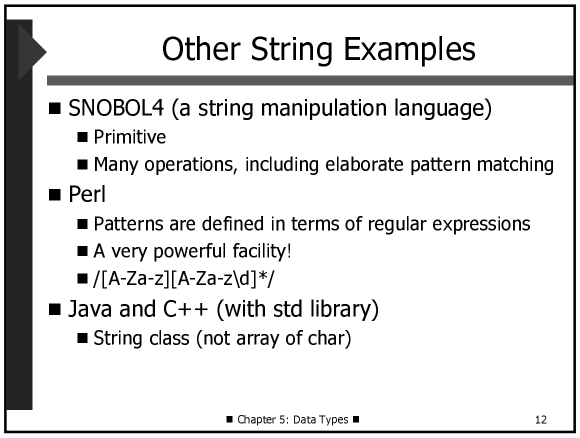
Character String Types

- Values are sequences of characters
- Design issues:
  - Is it a primitive type or just a special kind of array?
  - Is the length of objects fixed or variable?
- Operations:
  - Assignment
  - Comparison (=, >, etc.)
  - Concatenation
  - Substring reference
  - Pattern matching



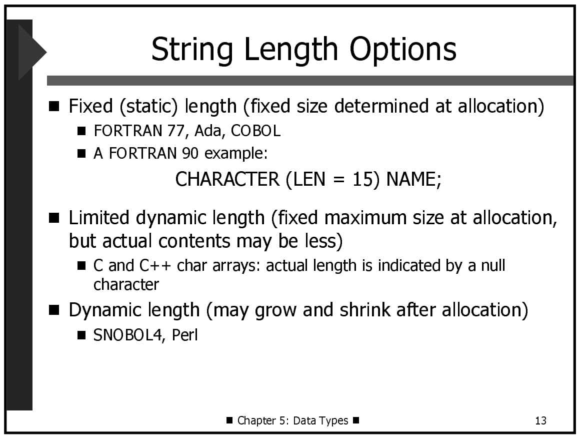
Examples of String Support

- Pascal
  - Not primitive; assignment and comparison only (of packed arrays)
- Ada, FORTRAN 77, FORTRAN 90 and BASIC
  - Somewhat primitive
  - Assignment, comparison, concatenation, substring reference
  - FORTRAN has an intrinsic for pattern matching
  - Examples (in Ada)
    - N := N1 & N2 (catenation)
    - N(2..4) (substring reference)
- C (and C++ for some people)
  - Not primitive
  - Use char arrays and a library of functions that provide operations



Other 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!
  - /[A-Za-z][A-Za-z\d]*/
- Java and C++ (with std library)
  - String class (not array of char)



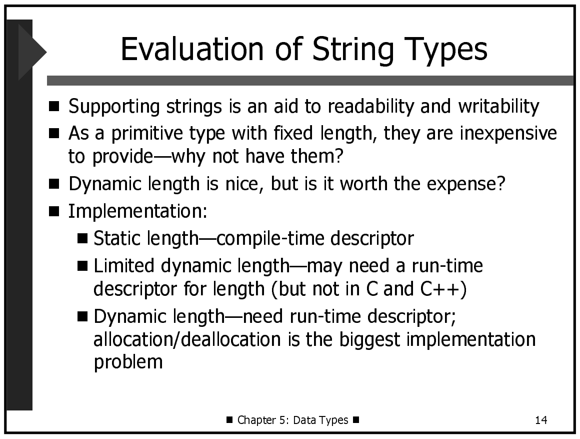
String Length Options

- Fixed (static) length (fixed size determined at allocation)
  - FORTRAN 77, Ada, COBOL
  - A FORTRAN 90 example:

```
CHARACTER (LEN = 15) NAME;
```

- Limited dynamic length (fixed maximum size at allocation, but actual contents may be less)
  - C and C++ char arrays: actual length is indicated by a null character
- Dynamic length (may grow and shrink after allocation)
  - SNOBOL4, Perl

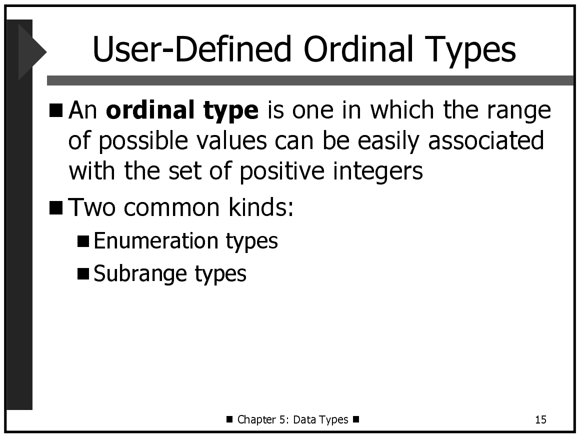
■ Chapter 5: Data Types ■ 13



Evaluation of String Types

- Supporting strings is an aid to readability and writability
- As a primitive type with fixed 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

■ Chapter 5: Data Types ■ 14



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
- Two common kinds:
  - Enumeration types
  - Subrange types

■ Chapter 5: Data Types ■ 15

Enumeration Types

- 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); can be used as in Pascal; input and output supported
  - C and C++—like Pascal, except they can be input and output as integers
  - Java does not include an enumeration type

Subrange Types

- An ordered, contiguous subsequence of another 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

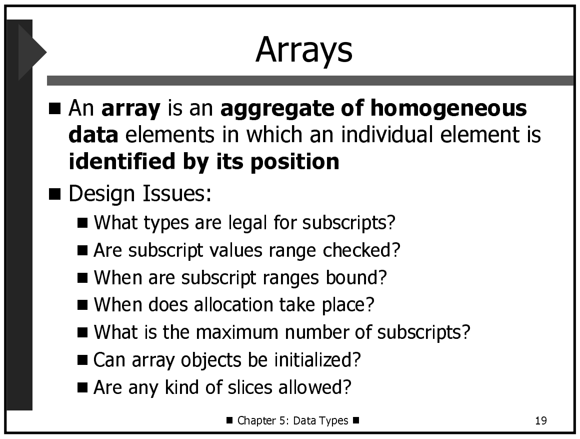
```
        type pos = 0 .. MAXINT;
```

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

```
    subtype Pos_Type is
        Integer range 0 ..Integer'Last;
```

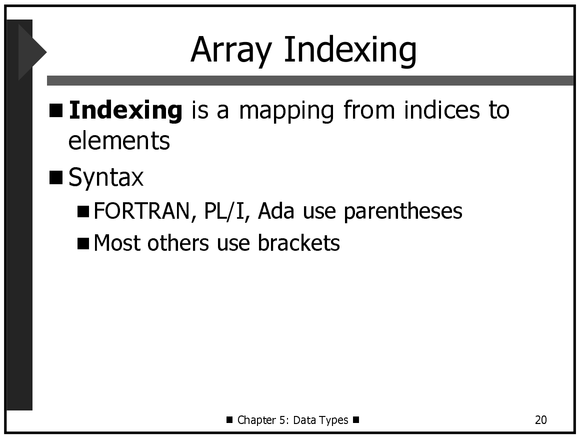
Evaluation of Ordinal Types

- Aid readability and writeability
- Improve reliability—restricted ranges add error detection abilitiy
- Implementation of user-defined ordinal types:
  - Enumeration types are implemented as integers
  - Subrange types are the parent types; code may be inserted (by the compiler) to restrict assignments to subrange variables



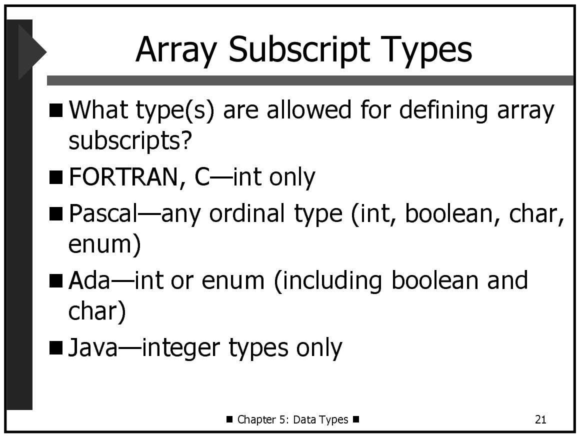
Arrays

- An **array** is an **aggregate of homogeneous data** elements in which an individual element is **identified by its position**
- Design Issues:
  - What types are legal for subscripts?
  - Are subscript values 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?



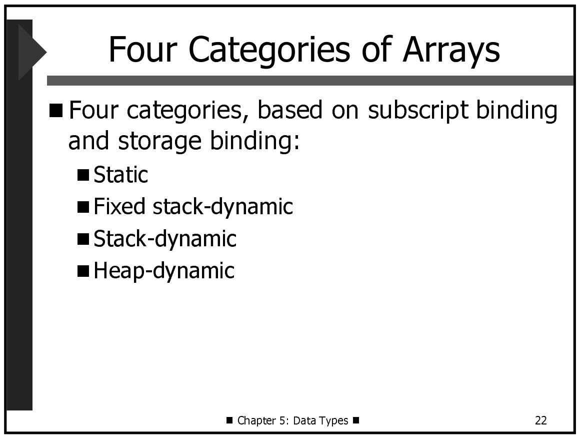
Array Indexing

- **Indexing** is a mapping from indices to elements
- Syntax
  - FORTRAN, PL/I, Ada use parentheses
  - Most others use brackets



Array Subscript Types

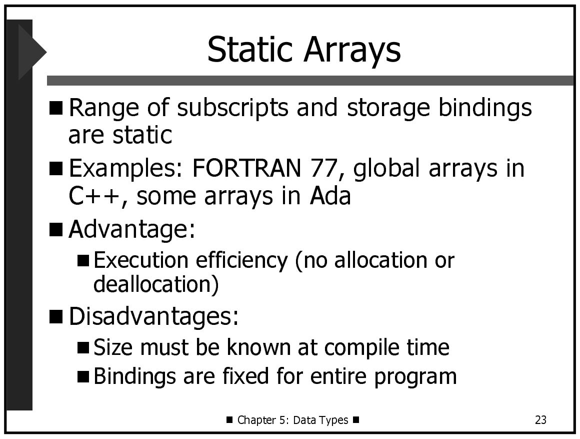
- What type(s) are allowed for defining array subscripts?
- FORTRAN, C—int only
- Pascal—any ordinal type (int, boolean, char, enum)
- Ada—int or enum (including boolean and char)
- Java—integer types only



Four Categories of Arrays

- Four categories, based on subscript binding and storage binding:
  - Static
  - Fixed stack-dynamic
  - Stack-dynamic
  - Heap-dynamic

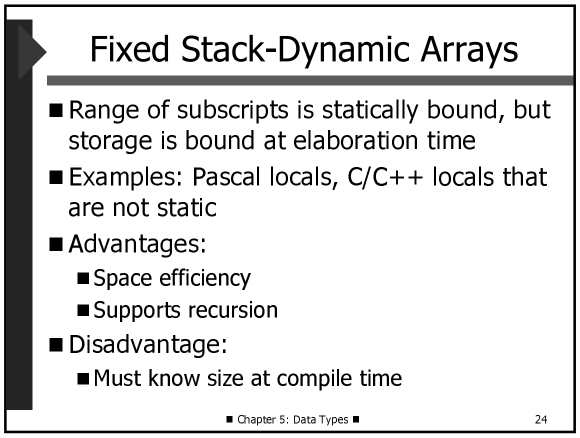
■ Chapter 5: Data Types ■ 22



Static Arrays

- Range of subscripts and storage bindings are static
- Examples: FORTRAN 77, global arrays in C++, some arrays in Ada
- Advantage:
  - Execution efficiency (no allocation or deallocation)
- Disadvantages:
  - Size must be known at compile time
  - Bindings are fixed for entire program

■ Chapter 5: Data Types ■ 23



Fixed Stack-Dynamic Arrays

- Range of subscripts is statically bound, but storage is bound at elaboration time
- Examples: Pascal locals, C/C++ locals that are not static
- Advantages:
  - Space efficiency
  - Supports recursion
- Disadvantage:
  - Must know size at compile time

■ Chapter 5: Data Types ■ 24

Stack-Dynamic Arrays

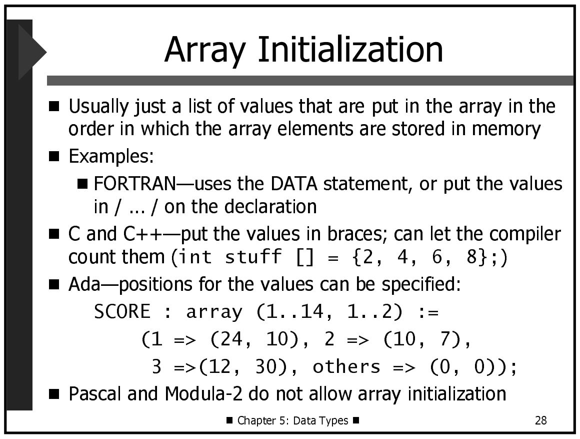
- Range and storage are dynamic, but fixed from then on for the variable's lifetime
- Examples: Ada locals in a procedure or block
- Advantage:
  - Flexibility—size need not be known until the array is about to be used
- Disadvantage:
  - Once created, array size is fixed for lifetime

Heap-Dynamic Arrays

- Subscript range and storage bindings are dynamic and not fixed
- Examples: FORTRAN 90, APL, Perl
- Advantage:
  - Ultimate in flexibility
- Disadvantages:
  - More space required
  - Run-time overhead

Number of Array 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



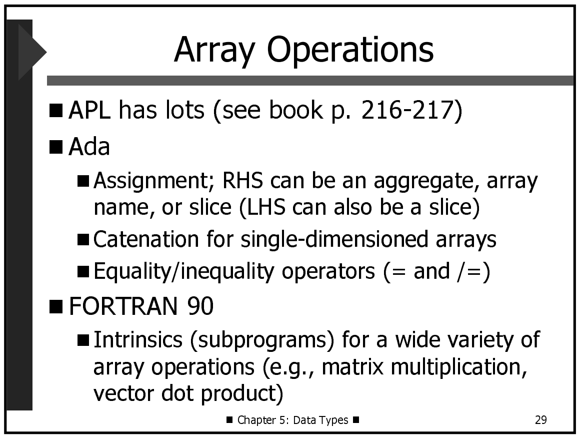
Array Initialization

- 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 (`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));
```

- Pascal and Modula-2 do not allow array initialization

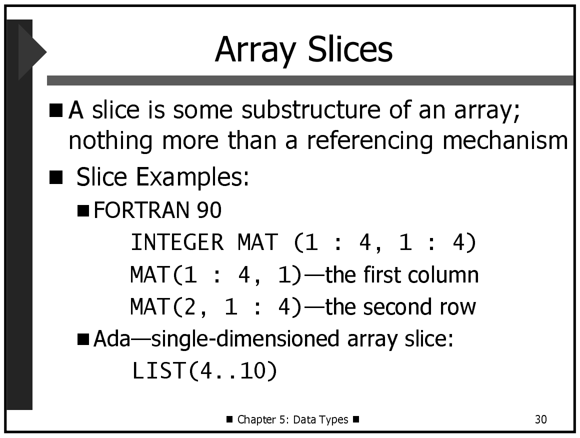
Chapter 5: Data Types 28



Array Operations

- APL has lots (see book p. 216-217)
- Ada
  - Assignment; RHS can be an aggregate, array name, or slice (LHS can also be a slice)
  - Catenation for single-dimensioned arrays
  - Equality/inequality operators (= and /=)
- FORTRAN 90
  - Intrinsics (subprograms) for a wide variety of array operations (e.g., matrix multiplication, vector dot product)

Chapter 5: Data Types 29



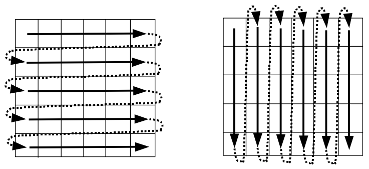
Array Slices

- A slice is some substructure of an array; nothing more than a referencing mechanism
- 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 array slice:
    - `LIST(4..10)`

Chapter 5: Data Types 30

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:
  - What is the form of references to elements?
  - Is the size static or dynamic?

Perl Associative Arrays

- Names begin with %
- Literals are delimited by parentheses

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

- Subscripting is done using braces and keys

```
$hi_temps{"Wednesday"} = 83;
```

- Elements can be removed with delete

```
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:
  - 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

In C:

```
typedef struct {
    int    field1;
    float field2;
    . . .
} MyRecord;
```

In Ada:

```
type MyRecord is record
    field1 : Integer;
    field2 : Float;
    . . .
end record;
```

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
- **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

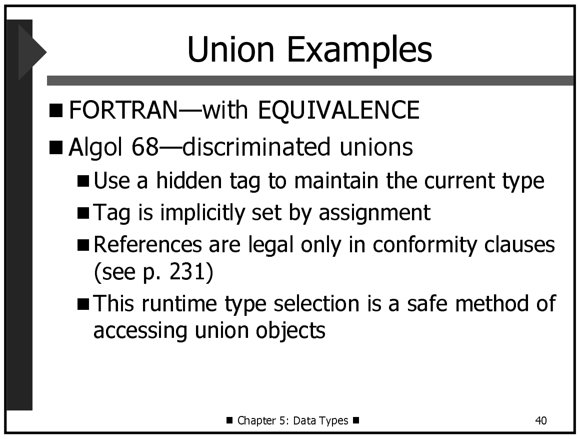
- Assignment
  - Pascal, Ada, and C++ allow it if the types are identical
  - In Ada, the RHS can be an aggregate
- Initialization
  - Allowed in Ada, using an aggregate
- Comparison
  - In Ada, = and /=; one operand can be an aggregate
- `MOVE CORRESPONDING`
  - In COBOL - it moves all fields in the source record to fields with the same names in the destination record

Comparing Records and Arrays

- Access to array elements is 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 be much slower

Unions

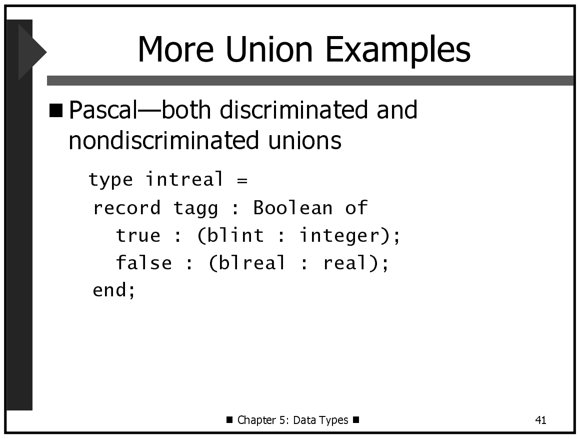
- A **union** is a type whose variables are allowed to store different type values at different times during execution
- Design Issues for unions:
  - What kind of type checking, if any, must be done?
  - Should unions be integrated with records?



Union Examples

- FORTRAN—with EQUIVALENCE
- 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 p. 231)
  - This runtime type selection is a safe method of accessing union objects

■ Chapter 5: Data Types ■ 40

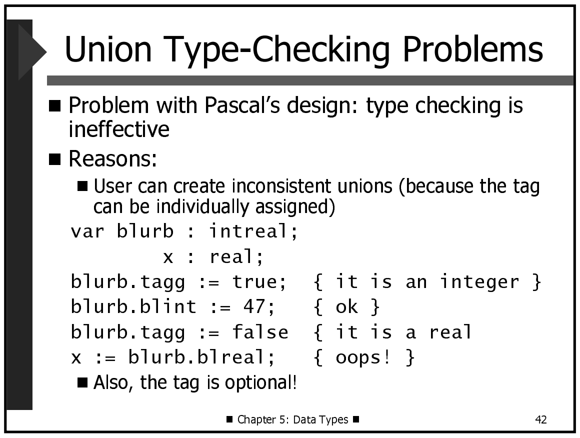


More Union Examples

- Pascal—both discriminated and nondiscriminated unions

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

■ Chapter 5: Data Types ■ 41



Union Type-Checking Problems

- Problem with Pascal's design: type checking is ineffective
- Reasons:
  - 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;   { oops! }
```

  - Also, the tag is optional!

■ Chapter 5: Data Types ■ 42

Ada Discriminated Unions

- Reasons they are safer than Pascal & Modula-2:
  - Tag must be present
  - All assignments to the union must include the tag value—tag cannot be assigned by itself
  - It is impossible for the user to create an inconsistent union

Free Unions

- C and C++ have free unions (no tags)
- Not part of their records
- No type checking of references
- Java has neither records nor unions

Evaluation of Unions

- Useful
- Potentially unsafe in most languages
- Ada, Algol 68 provide safe versions

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 the base type of a set?

Set Examples

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

Set Examples (cont.)

- Modula-2 and Modula-3
  - Additional operations: INCL, EXCL, / (symmetric set difference (elements in one but not both operands))
- Ada—does not include sets, but defines in as set membership operator for all enumeration types and subrange expressions
- Java includes a class for set operations

Evaluation of Sets

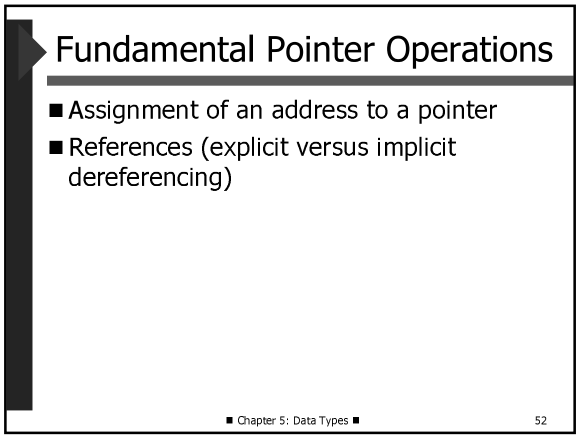
- If a language does not have sets, they must be simulated, either with enumerated types or with arrays
- Arrays are more flexible than sets, but have much slower operations
- Set implementation:
  - Usually stored as bit strings and use logical operations for the set operations

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:
  - Addressing flexibility
  - Dynamic storage management

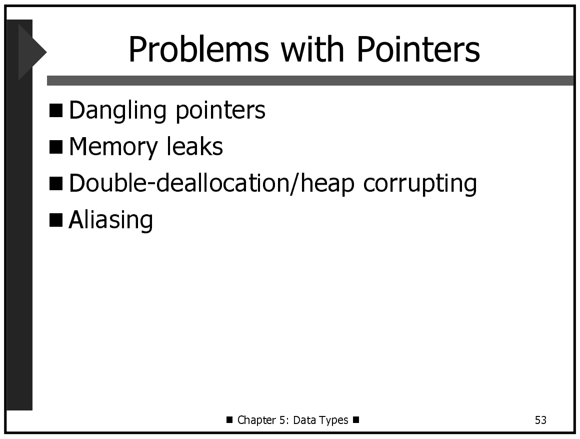
Pointer Design Issues

- 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?



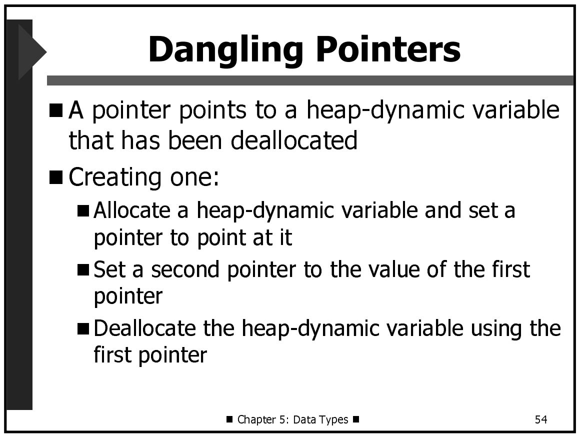
Fundamental Pointer Operations

- Assignment of an address to a pointer
- References (explicit versus implicit dereferencing)



Problems with Pointers

- Dangling pointers
- Memory leaks
- Double-deallocation/heap corrupting
- Aliasing



Dangling Pointers

- A pointer points to a heap-dynamic variable that has been deallocated
- Creating one:
  - 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

Memory Leaks

- Lost heap-dynamic variables (garbage):
- 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

Double Deallocation

- When a heap-dynamic object is explicitly deallocated twice
- Usually corrupts the data structure(s) the run-time system uses to maintain free memory blocks (the heap)
- A special case of dangling pointers

Aliasing

- When two pointers refer to the same address
- Pointers need not be in the same program unit
- Changes made through one pointer affect the behavior of code referencing the other pointer
- When unintended, may cause unpredictability, loss of locality of reasoning

Pointer Examples

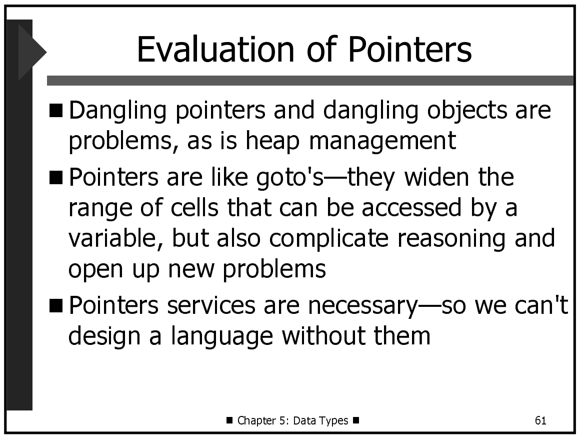
- Pascal: used for dynamic storage management only
  - Explicit dereferencing
  - Dangling pointers and memory leaks are possible
- Ada: a little better than Pascal and Modula-2
  - Implicit dereferencing
  - All pointers are initialized to null
  - Similar dangling pointer and memory leak problems for typical implementations

Pointer Examples (cont.)

- C and C++: for both 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 *)
- C++ reference types
  - Constant pointers that are implicitly dereferenced
  - Typically used for parameters
  - Advantages of both pass-by-reference and pass-by-value

Pointer Examples (cont.)

- FORTRAN 90 Pointers
  - Can point to heap and non-heap variables
  - Implicit dereferencing
  - Special assignment operator for non-dereferenced references
- 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

- 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, but also complicate reasoning and open up new problems
- Pointers services are necessary—so we can't design a language without them

■ Chapter 5: Data Types ■ 61