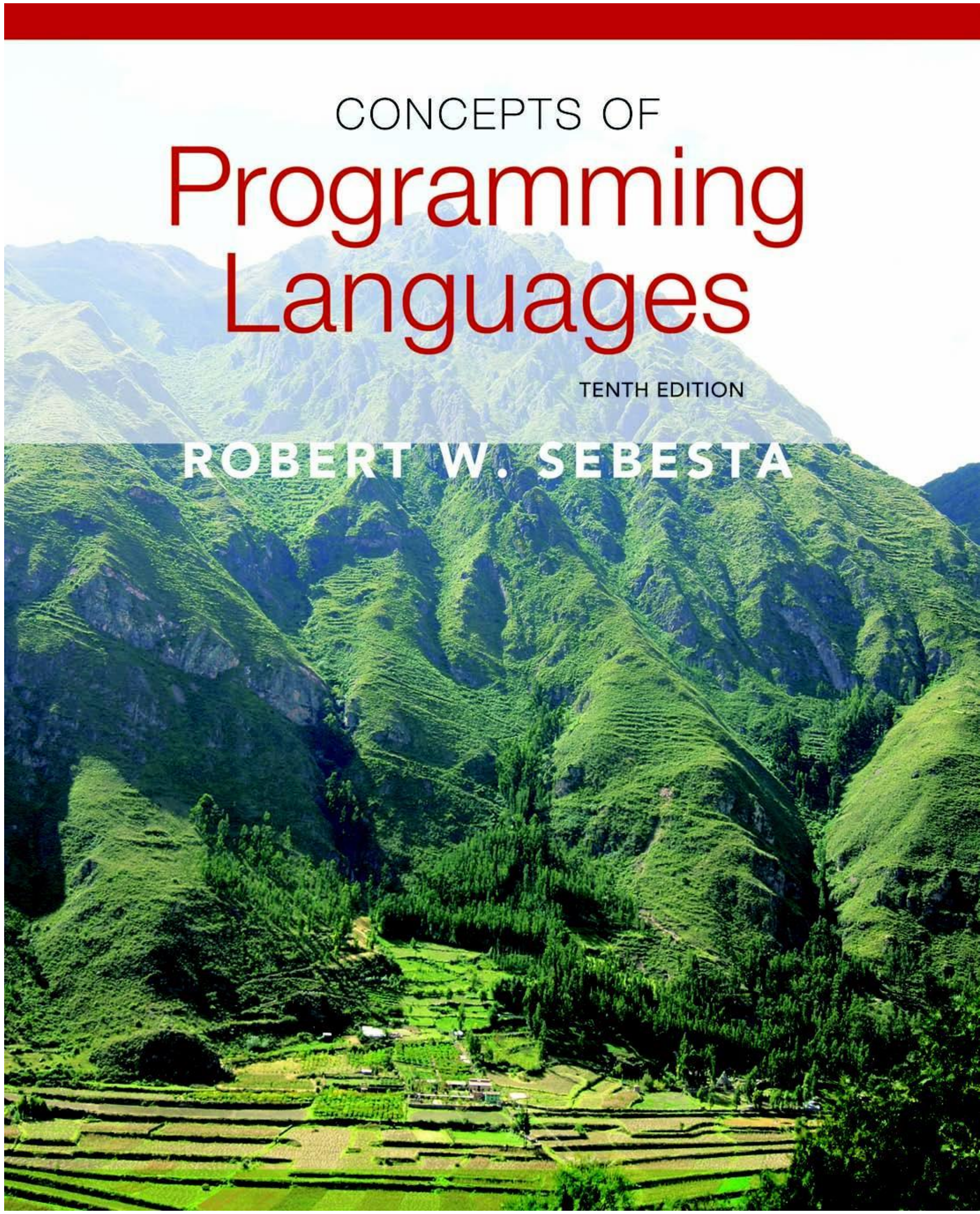
Chapter 6

Data Types



Chapter 6 Topics

- Introduction
- Primitive Data Types
- Character String Types
- User-Defined Ordinal Types
- Array Types
- Associative Arrays
- Record Types
- Tuple Types
- List Types
- Union Types
- Pointer and Reference Types
- Type Checking
- Strong Typing
- Type Equivalence
- Theory and Data Types

Introduction

- A *data type* defines a collection of data objects and a set of predefined operations on those objects
- A *descriptor* is the collection of the attributes of a variable
- An *object* represents an instance of a user-defined (abstract data) type
- One design issue for all data types: What operations are defined and how are they specified?

Data Types

- A data type defines
  - a collection of data objects, and
  - a set of predefined operations on the objects

    type: integer

    operations: +, -, *, /, %, ^

- Evolution of Data Types
  - Early days:
    - all programming problems had to be modeled using only a few data types
    - FORTRAN I (1957) provides INTEGER, REAL, arrays
  - Nowadays:
    - Users can define abstract data types (representation + operations)

Data Types

- Primitive Types
- Strings
- Records
- Unions
- Arrays
- Associative Arrays
- Sets
- Pointers

Primitive Data Types

- Almost all programming languages provide a set of *primitive data types*
- Primitive data types: Those not defined in terms of other data types
- Some primitive data types are merely reflections of the hardware
- Others require only a little non-hardware support for their implementation

Primitive Data Types

- Those not defined in terms of other data types
  - Numeric types
    - Integer
    - Floating point
    - decimal
  - Boolean types
  - Character types

Primitive Data Types: Integer

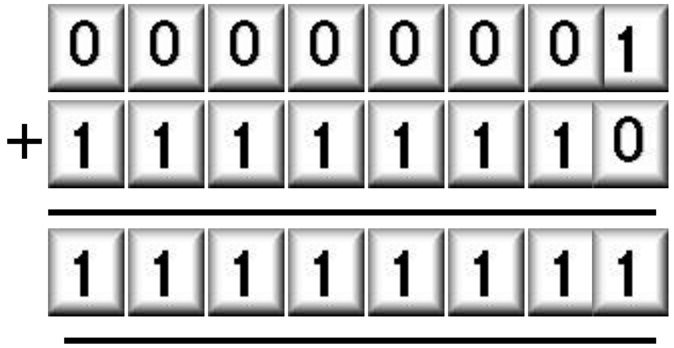
- Almost always an exact reflection of the hardware so the mapping is trivial
- There may be as many as eight different integer types in a language
- Java's signed integer sizes: **`byte, short, int, long`**

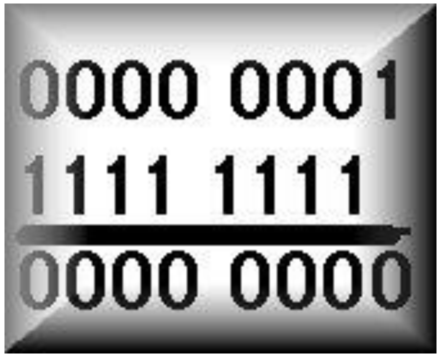
Representing Negative Integers

1 + (-1) = ?

Ones complement, 8 bits

- +1 is 0000 0001
- -1 is 1111 1110
- If we use natural method of summation we get sum 1111 1111



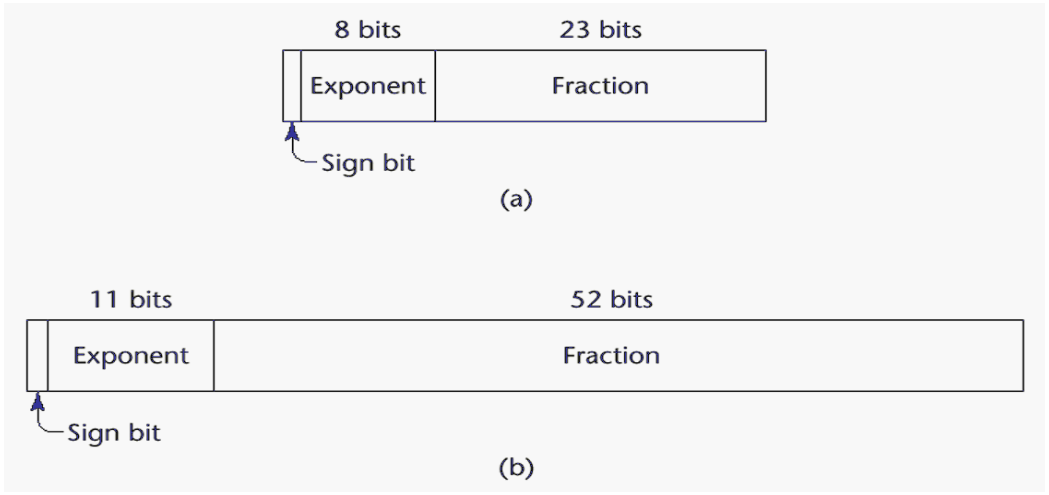


Twos complement, 8 bits

- +1 is 0000 0001
- -1 is 1111 1111
- If we use the natural method we get sum 0000 0000 (and carry 1 which we disregard)

Primitive Data Types: Floating Point

- Model real numbers, but only as approximations
- Languages for scientific use support at least two floating-point types (e.g., **`float`** and **`double`**; sometimes more
- Usually exactly like the hardware, but not always
- IEEE Floating-Point Standard 754



Floating Point

- Floating Point
  - Approximate real numbers
    - Note: even 0.1 cannot be represented exactly by a finite number of of binary digits!
    - Loss of accuracy when performing arithmetic operation
  - Languages for scientific use support at least two floating-point types; sometimes more

$$1.63245 \text{ x } 10^5$$

  - Precision: accuracy of the fractional part
  - Range: combination of range of fraction & exponent
  - Most machines use IEEE Floating Point Standard 754 format

Floating Point Puzzle

```
int x = 1;

float f = 0.1;

double d = 0.1;
```

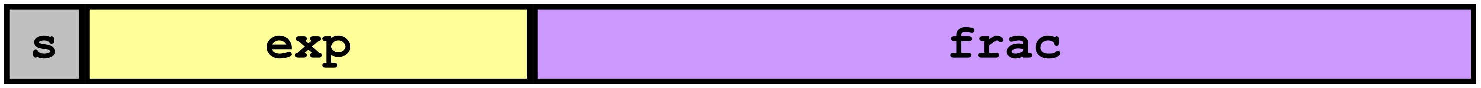
True or False?

• `x == (int)(float) x`	**True**
• `x == (int)(double) x`	**True**
• `f == (float)(double) f`	**True**
• `d == (float) d`	**False**
• `f == -(-f);`	**True**
• `d > f`	**False**
• `-f > -d`	**False**
• `f > d`	**True**
• `-d > -f`	**True**
• `d == f`	**False**
• `(d+f)-d == f`	**True**

Floating Point Representation

- Numerical Form
  - $-1^{s}\ M\ 2^{E}$
    - Sign bit *s* determines whether number is negative or positive
    - Significand *M* normally a fractional value in range [1.0,2.0).
    - Exponent *E* weights value by power of two

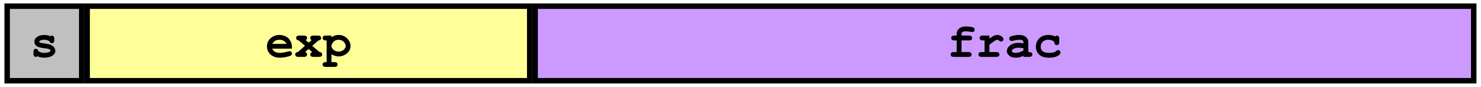
- Encoding



  - MSB is sign bit
  - `exp` field encodes *E*
  - `frac` field encodes *M*

Floating Point Representation

- Encoding



  - MSB is sign bit
  - `exp` field encodes *E*
  - `frac` field encodes *M*

- Sizes
  - Single precision: 8 `exp` bits, 23 `frac` bits
    - 32 bits total
  - Double precision: 11 `exp` bits, 52 `frac` bits
    - 64 bits total
  - Extended precision: 15 `exp` bits, 63 `frac` bits
    - Only found in Intel-compatible machines
    - Stored in 80 bits
      - 1 bit wasted

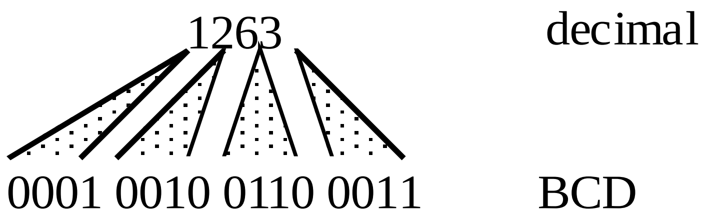
Primitive Data Types: Complex

- Some languages support a complex type, e.g., C99, Fortran, and Python
- Each value consists of two floats, the real part and the imaginary part
- Literal form (in Python):

  `(7 + 3j)`, where `7` is the real part and `3` is the imaginary part

Decimal Types

- For business applications ($$$) – e.g., COBOL
- Store a fixed number of decimal digits, with the decimal point at a fixed position in the value
- Advantage
  - can precisely store decimal values
- Disadvantages
  - Range of values is restricted because no exponents are allowed
  - Representation in memory is wasteful
    - Representation is called binary coded decimal (BCD)



Boolean Types

- Could be implemented as bits, but often as bytes
- Introduced in ALGOL 60
- Included in most general-purpose languages designed since 1960
- Ansi C (1989)
  - all operands with nonzero values are considered true, and zero is considered false
- Advantage: readability

Character Types

- Characters are stored in computers as numeric codings
- Traditionally use 8-bit code ASCII, which uses 0 to 127 to code 128 different characters
- ISO 8859-1 also use 8-bit character code, but allows 256 different characters
  - Used by Ada
- 16-bit character set named Unicode (UCS-2)
  - Includes Cyrillic alphabet used in Serbia, and Thai digits
  - First 128 characters are identical to ASCII
  - used by Java and C#
- 32-bit Unicode (UCS-4)
  - Supported by Fortran, starting with 2003

Character String Types

- Values consist of sequences of characters
- Design issues:
  - Is it a primitive type or just a special kind of character array?
  - 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
  - Fortran 90
    - Somewhat primitive; operations include assignment, comparison, catenation, substring reference, and pattern matching

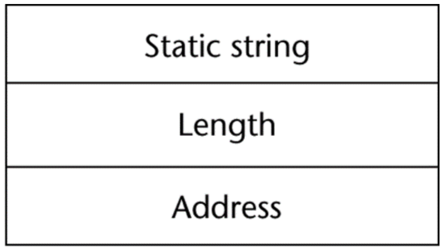
Character Strings

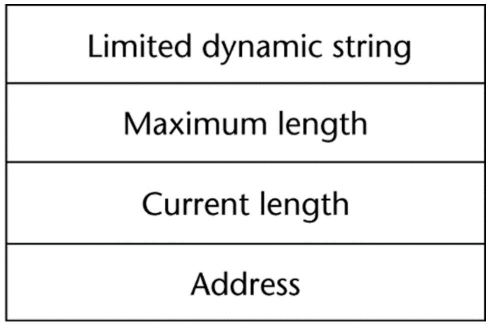
- Examples
  - Ada

    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, JavaScript, Ruby, and PHP
    - Patterns are defined in terms of regular expressions; a very powerful facility
  - Java
    - String class (not arrays of char); Objects are immutable
    - StringBuffer is a class for changeable string objects

Character Strings

- String Length
  - 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
- Evaluation (of character string types)
  - Aid to writability
  - As a primitive type with static length, they are inexpensive to provide
  - Dynamic length is nice, but is it worth the expense?
- Implementation





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
- Examples of primitive ordinal types in Java
  - **`integer`**
  - **`char`**
  - **`boolean`**

Ordinal Data Types

- Range of possible values can be easily associated with the set of positive integers

- Enumeration types
  - user enumerates all the possible values, which are symbolic constants

    enum days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
  - Design Issue:
    - Should a symbolic constant be allowed to be in more than one type definition?
    - Type checking
      - Are enumerated types coerced to integer?
      - Are any other types coerced to an enumerated type?

Enumeration Types

- All possible values, which are named constants, are provided in the definition (user enumerates all the possible values, which are symbolic constants)
- C# example

```
enum days {mon, tue, wed, thu, fri, sat, sun};
```

- Design issues
  - Is an enumeration constant allowed to appear in more than one type definition, and if so, how is the type of an occurrence of that constant checked?
  - Are enumeration values coerced to integer?
  - Any other type coerced to an enumeration type?

Enumeration Data Types

- Examples
  - Pascal
    - cannot reuse constants; can be used for array subscripts, for variables, case selectors; can be compared
  - Ada
    - constants can be reused (overloaded literals); disambiguate with context or type_name'(one of them) (e.g, Integer'Last)
  - C and C++
    - enumeration values are coerced into integers when they are put in integer context
  - Java
    - Java 4.0 and previous versions do not include an enumeration type, but provides the Enumeration interface
    - Java 5.0 includes enumeration type
    - can implement them as classes

```
class colors {
    public final int red = 0;
    public final int blue = 1;
}
```

Java enum

A Java Enum is a special Java type used to define collections of constants. More precisely, a Java enum type is a special kind of Java class. An enum can contain constants, methods etc. Java enums were added in Java 5.

```
public enum Level {
    HIGH,
    MEDIUM,
    LOW
}

Level level = Level.HIGH;
```

Java enum

You can add fields to a Java enum. Thus, each constant enum value gets these fields. The field values must be supplied to the constructor of the enum when defining the constants. Here is an example:

```
public enum Level {
    HIGH  (3),  //calls constructor with value 3
    MEDIUM(2),  //calls constructor with value 2
    LOW   (1)   //calls constructor with value 1
    ; // semicolon needed when fields / methods follow


    private final int levelCode;

    public Level(int levelCode) {
        this.levelCode = levelCode;
    }
}
```

Subrange Data Types

- An ordered contiguous subsequence of an ordinal type
  - e.g., 12..14 is a subrange of integer 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;

```
type Days is (mon, tue, wed, thu, fri, sat, sun);
subtype Weekdays is Days range mon..fri;
subtype Index is Integer range 1..100;

Day1: Days;
Day2: Weekday;
Day2 := Day1;
```

- Evaluation
  - Aid to readability - restricted ranges add error detection

Implementation of 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 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?

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: A(3)
  - most other languages use brackets: A[3]
- 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

- Number of subscripts (dimensions)
  - 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

      ```
      Integer List(3)
      Data List /0, 5, 5/
      ```

    - 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 does not allow array initialization

Arrays

- Array Operations
  - Ada
    - Assignment; RHS can be an aggregate constant or an array name
    - Catenation between single-dimensioned arrays
  - FORTRAN 95
    - Includes a number of array operations called elementals because they are operations between pairs of array elements
      - E.g., add (+) operator between two arrays results in an array of the sums of element pairs of the two arrays
  - Slices
    - A slice is some substructure of an array
    - 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)
```

Arrays

- Implementation of Arrays
  - Access function maps subscript expressions to an address in the array
  - Single-dimensioned array

    address(list[k])
      = address(list[lower_bound])
          + (k-1)*element_size
      = (address[lower_bound] – element_size)
          + (k * element_size)

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

  - Multi-dimensional arrays
    - Row major order: 3, 4, 7, 6, 2, 5, 1, 3, 8
    - Column major order 3, 6, 1, 4, 2, 3, 7, 5, 8

$$\begin{matrix} 3 & 4 & 7 \\ 6 & 2 & 5 \\ 1 & 3 & 8 \end{matrix}$$

Subscript Binding and Array Categories

- *Static*: subscript ranges are statically bound and storage allocation is static (before run-time)
  - Advantage: efficiency (no dynamic allocation)
- *Fixed stack-dynamic*: subscript ranges are statically bound, but the allocation is done at declaration time
  - Advantage: space efficiency

Subscript Binding and Array Categories (continued)

- *Stack-dynamic*: subscript ranges are dynamically bound and the storage allocation is dynamic (done at run-time)
  - Advantage: flexibility (the size of an array need not be known until the array is to be used)
- *Fixed heap-dynamic*: similar to fixed stack-dynamic: storage binding is dynamic but fixed after allocation (i.e., binding is done when requested and storage is allocated from heap, not stack)

Subscript Binding and Array Categories (continued)

- Heap-dynamic: binding of subscript ranges and storage allocation is dynamic and can change any number of times
  - Advantage: flexibility (arrays can grow or shrink during program execution)

Subscript Binding and Array Categories (continued)

- C and C++ arrays that include `static` modifier are static
- C and C++ arrays without `static` modifier are fixed stack-dynamic
- C and C++ provide fixed heap-dynamic arrays
- C# includes a second array class `ArrayList` that provides fixed heap-dynamic
- Perl, JavaScript, Python, and Ruby support heap-dynamic arrays

Array Initialization

- Some language allow initialization at the time of storage allocation
  - C, C++, Java, C# example

```
int list [] = {4, 5, 7, 83}
```

  - Character strings in C and C++

```
char name [] = "freddie";
```

  - Arrays of strings in C and C++

```
char *names [] = {"Bob", "Jake", "Joe"];
```

  - Java initialization of String objects

```
String[] names = {"Bob", "Jake", "Joe"};
```

Heterogeneous Arrays

- A *heterogeneous array* is one in which the elements need not be of the same type
- Supported by Perl, Python, JavaScript, and Ruby

Array Initialization

- C-based languages
  - `int list [] = {1, 3, 5, 7}`
  - `char *names [] = {"Mike", "Fred", "Mary Lou"};`
- Ada

  -
    ```
    List : array (1..5) of Integer :=
        (1 => 17, 3 => 34, others => 0);
    ```
- Python
  - List comprehensions

    ```
    list = [x ** 2 for x in range(12) if x % 3 ==
     0]
    ```

    puts `[0, 9, 36, 81]` in `list`

Arrays Operations

- APL provides the most powerful array processing operations for vectors and matrixes as well as unary operators (for example, to reverse column elements)
- Ada allows array assignment but also catenation
- Python's array assignments, but they are only reference changes. Python also supports array catenation and element membership operations
- Ruby also provides array catenation
- Fortran provides *elemental* operations because they are between pairs of array elements
  - For example, + operator between two arrays results in an array of the sums of the element pairs of the two arrays

Rectangular and Jagged Arrays

- A rectangular array is a multi-dimensioned array in which all of the rows have the same number of elements and all columns have the same number of elements
- A jagged matrix has rows with varying number of elements
  - Possible when multi-dimensioned arrays actually appear as arrays of arrays
- C, C++, and Java support jagged arrays
- Fortran, Ada, and C# support rectangular arrays (C# also supports jagged 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

Slice Examples

- Python

```
vector = [2, 4, 6, 8, 10, 12, 14, 16]
mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

`vector (3:6)` is a three-element array

`mat[0][0:2]` is the first and second element of the first row of `mat`

- Ruby supports slices with the `slice` method

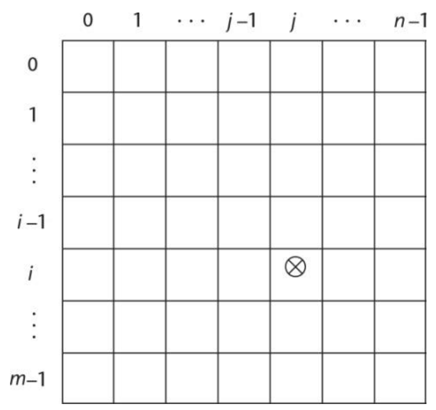
`list.slice(2, 2)` returns the third and fourth elements of `list`

Implementation of Arrays

- Access function maps subscript expressions to an address in the array
- Access function for single-dimensioned arrays:

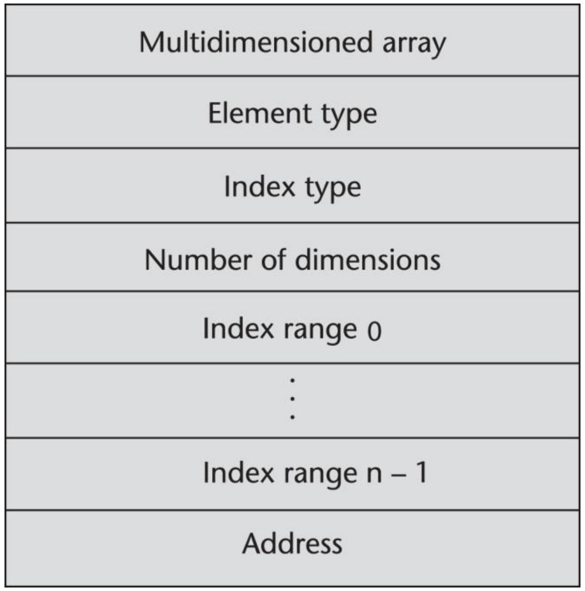
address(list[k]) = address (list[lower_bound])

+ ((k-lower_bound) * element_size)



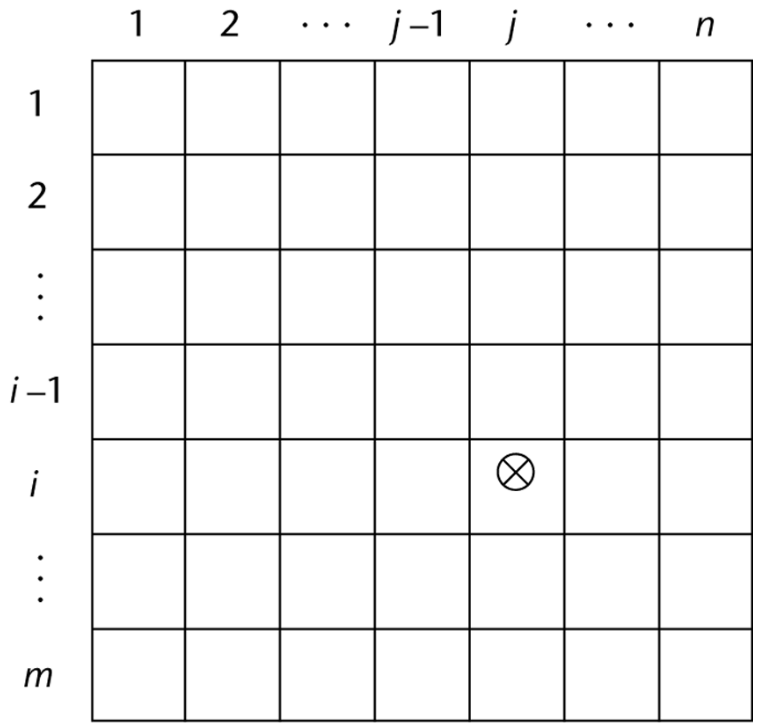
Accessing Multi-dimensioned Arrays

- Two common ways:
  - Row major order (by rows) – used in most languages
  - Column major order (by columns) – used in Fortran
  - A compile-time descriptor for a multidimensional array

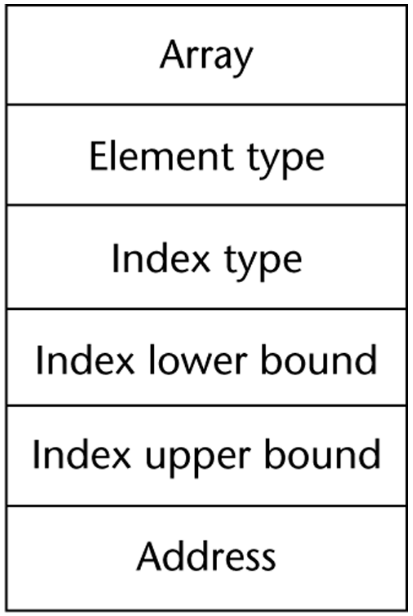


Locating an Element in a Multi-dimensioned Array

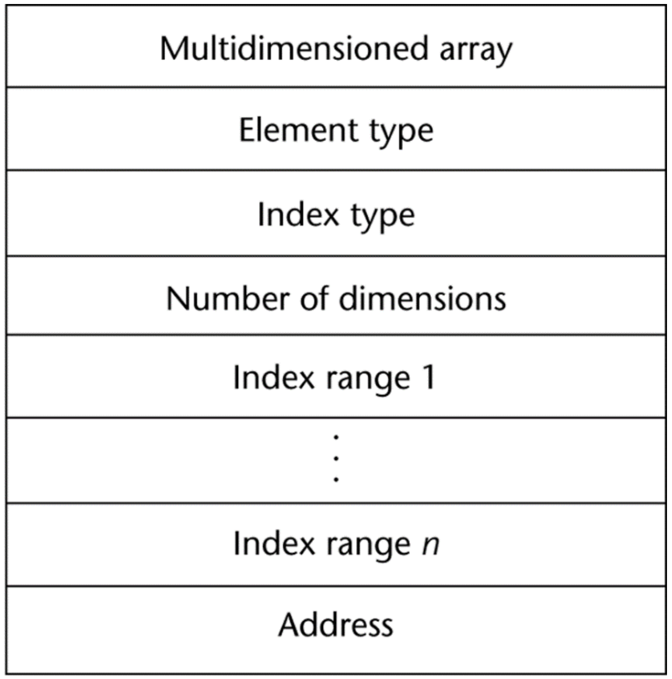
- General format
  Location (a[I,j]) = address of a [row_lb,col_lb] + (((I - row_lb) * n) + (j - col_lb)) * element_size



Compile-Time Descriptors



Single-dimensioned array



Multidimensional array

Associative Arrays

- An *associative array* is an unordered collection of data elements that are indexed by an equal number of values called *keys*
  - User-defined keys must be stored
- Design issues:

  - What is the form of references to elements?

  - Is the size static or dynamic?
- Built-in type in Perl, Python, Ruby, and Lua
  - In Lua, they are supported by tables

Associative Arrays

- Structure and Operations in Perl
  - Names begin with %
  - Literals are delimited by parentheses
  - %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"};

Record Types

- A *record* is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names
- Design issues:
  - What is the syntactic form of references to the field?
  - Are elliptical references allowed

Records

- Record Definition Syntax
  - COBOL uses level numbers to show nested records; others use recursive definitions
  - COBOL

```
01      EMPLOYEE-RECORD.
        02      EMPLOYEE-NAME.
                     05  FIRST            PICTURE IS X(20).
                     05  MIDDLE           PICTURE IS X(10).
                     05 LAST              PICTURE IS X(20).
        02      HOURLY-RATE               PICTURE IS 99V99.
```

Level numbers (01,02,05) indicate their relative values in the hierarchical structure of the record

PICTURE clause show the formats of the field storage locations

X(20): 20 alphanumeric characters
99V99: four decimal digits with decimal point in the middle

Records

- Ada:

```
Type Employee_Name_Type is record
    First: String (1..20);
    Middle: String (1..10);
    Last: String (1..20);
end record;
type Employee_Record_Type is record
    Employee_Name: Employee_Name_Type;
    Hourly_Rate: Float;
end record;

Employee_Record: Employee_Record_Type;
```

Records

- References to Record Fields
- COBOL field references

  field_name OF record_name_1 OF … OF record_name_n

  e.g. MIDDLE OF EMPLOYEE-NAME OF EMPLOYEE_RECORD

- Fully qualified references must include all intermediate record names
- Elliptical references allow leaving out record names as long as the reference is unambiguous
  - e.g., the following are equivalent:

    FIRST,   FIRST OF EMPLOYEE-NAME, FIRST OF EMPLOYEE-RECORD

Records

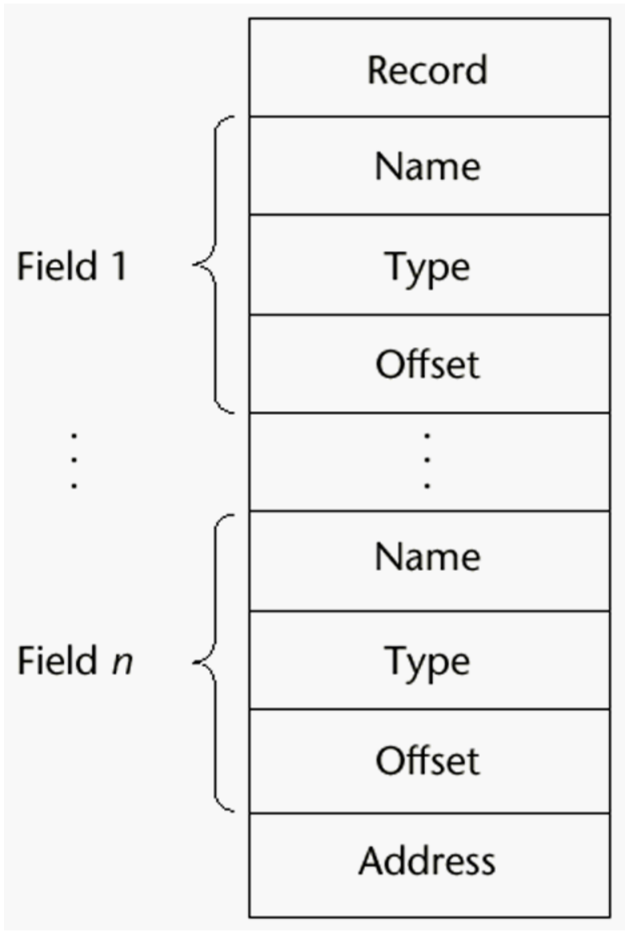
- Operations
  - Assignment
    - Pascal, Ada, and C allow it if the types are identical
      - In Ada, the RHS can be an aggregate constant
  - Initialization
    - Allowed in Ada, using an aggregate constant
  - Comparison
    - In Ada, = and /=; one operand can be an aggregate constant
  - MOVE CORRESPONDING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

Comparing Records to Arrays

- Records are used when collection of data values is heterogeneous
- 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

Implementation of Record Type

Offset address relative to the beginning of the records is associated with each field



Tuple Types

- A tuple is a data type that is similar to a record, except that the elements are not named
- Used in Python, ML, and F# to allow functions to return multiple values
  - Python
    - Closely related to its lists, but immutable
    - Create with a tuple literal

      `myTuple = (3, 5.8, 'apple')`

      Referenced with subscripts (begin at `1`)

    Catenation with + and deleted with **`del`**

Tuple Types (continued)

- ML

```
val myTuple = (3, 5.8, 'apple');
```

- Access as follows:

  `#1(myTuple)` is the first element

- A new tuple type can be defined

```
type intReal = int * real;
```

- F#

```
let tup = (3, 5, 7)
```

`let a, b, c = tup` This assigns a tuple to a tuple pattern `(a, b, c)`

List Types

- Lists in LISP and Scheme are delimited by parentheses and use no commas

  `(A B C D)` and `(A (B C) D)`
- Data and code have the same form

  As data, `(A B C)` is literally what it is

  As code, `(A B C)` is the function `A` applied to the parameters `B` and `C`
- The interpreter needs to know which a list is, so if it is data, we quote it with an apostrophe

  `′(A B C)` is data

List Types (continued)

- List Operations in Scheme
  - `CAR` returns the first element of its list parameter

    `(CAR ′(A B C))` returns `A`
  - `CDR` returns the remainder of its list parameter after the first element has been removed

    `(CDR ′(A B C))` returns `(B C)`
  - `CONS` puts its first parameter into its second parameter, a list, to make a new list

    `(CONS ′A (B C))` returns `(A B C)`
  - `LIST` returns a new list of its parameters

    `(LIST ′A ′B ′(C D))` returns `(A B (C D))`

List Types (continued)

- List Operations in ML
  - Lists are written in brackets and the elements are separated by commas
  - List elements must be of the same type
  - The Scheme `CONS` function is a binary operator in ML, `::`

    `3 :: [5, 7, 9]` evaluates to `[3, 5, 7, 9]`
  - The Scheme `CAR` and `CDR` functions are named `hd` and `tl`, respectively

List Types (continued)

- F# Lists
  - Like those of ML, except elements are separated by semicolons and `hd` and `tl` are methods of the `List` class
- Python Lists
  - The list data type also serves as Python’s arrays
  - Unlike Scheme, Common LISP, ML, and F#, Python’s lists are mutable
  - Elements can be of any type
  - Create a list with an assignment

```
myList = [3, 5.8, "grape"]
```

List Types (continued)

- Python Lists (continued)
  - List elements are referenced with subscripting, with indices beginning at zero

    `x = myList[1]`   Sets `x` to `5.8`
  - List elements can be deleted with `del`

    `del myList[1]`
  - List Comprehensions – derived from set notation

    `[x * x for x in range(6) if x % 3 == 0]`

    `range(7)` creates `[0, 1, 2, 3, 4, 5, 6]`

    Constructed list: `[0, 9, 36]`

List Types (continued)

- Haskell's List Comprehensions
  - The original

```
[n * n | n <- [1..10]]
```

- F#'s List Comprehensions

```
let myArray = [|for i in 1 .. 5 -> [i * i) |]
```

- Both C# and Java supports lists through their generic heap-dynamic collection classes, `List` and `ArrayList`, respectively

Unions Types

- 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

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

    - Problem with Pascal’s design: type checking is ineffective

Unions

- Example (Pascal)…
  - 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

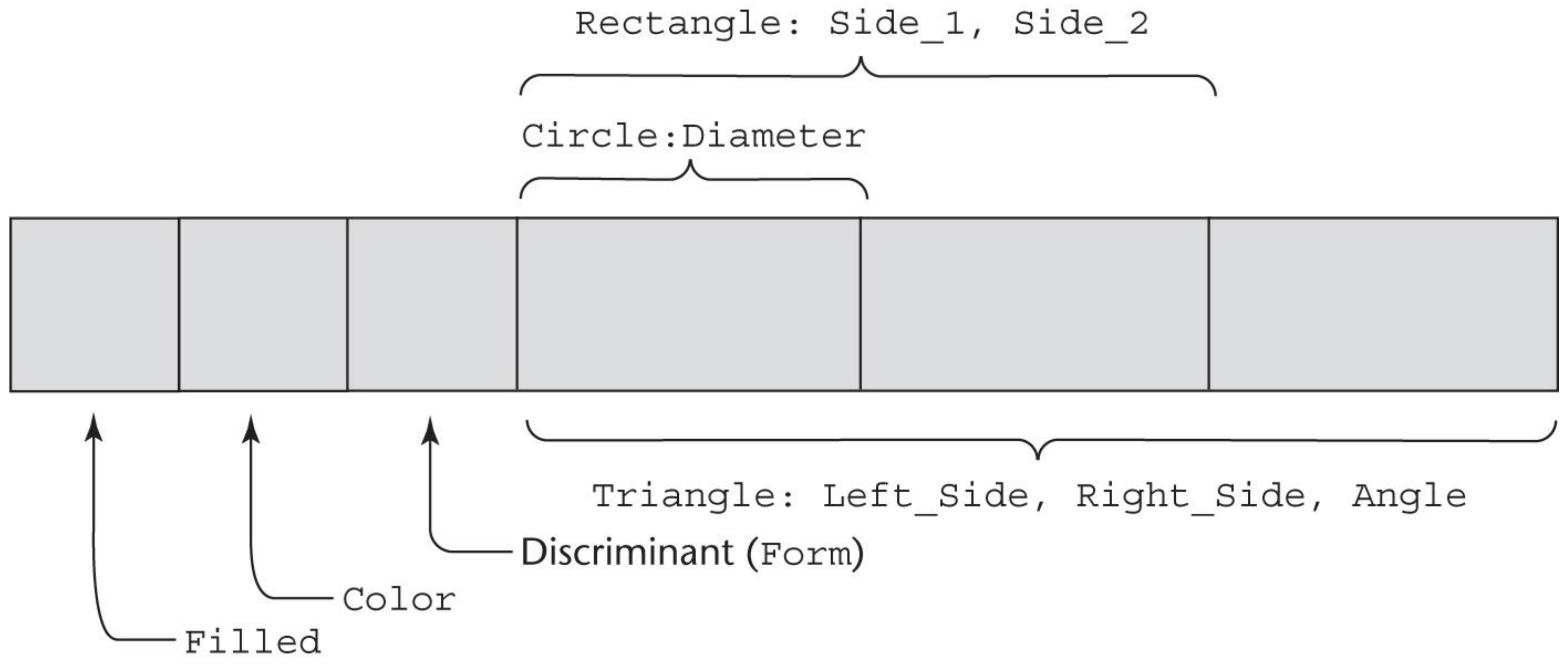
Unions

- Examples…
  - 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

Ada Union Types

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
   Filled: Boolean;
   Color: Colors;
   case Form is
      when Circle => Diameter: Float;
      when Triangle =>
            Leftside, Rightside: Integer;
            Angle: Float;
      when Rectangle => Side1, Side2: Integer;
   end case;
end record;
```

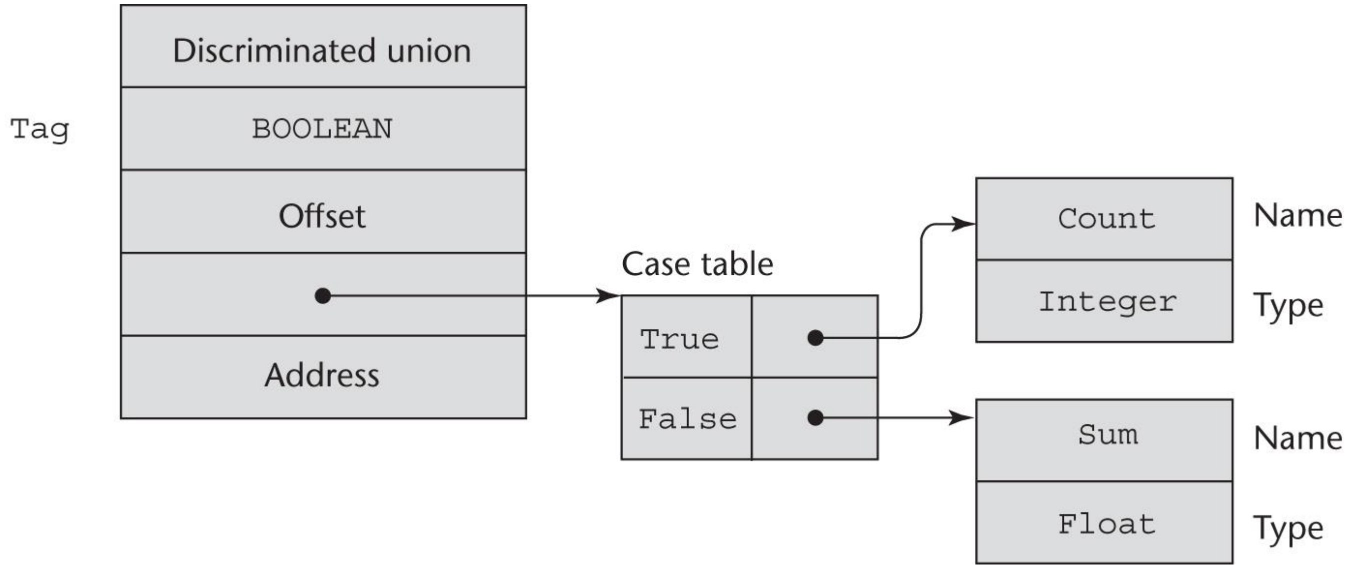
Ada Union Type Illustrated


Rectangle: Side_1, Side_2
Circle:Diameter
Triangle: Left_Side, Right_Side, Angle
Discriminant (Form)
Color
Filled

A discriminated union of three shape variables

Implementation of Unions

```
type Node (Tag : Boolean) is
  record
    case Tag is
      when True => Count : Integer;
      when False => Sum : Float;
    end case;
  end record;
```



Evaluation of Unions

- Free unions are unsafe
  - Do not allow type checking
- Java and C# do not support unions
  - Reflective of growing concerns for safety in programming language
- Ada's descriminated unions are safe

Sets

- 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?
- Example
  - Pascal
    - No maximum size in the language definition
      (not portable, poor writability if max is too small)
    - Operations: in, union (+), intersection (*), difference (-), =, <>, superset (>=), subset (<=)
  - Ada
    - does not include sets, but defines in as set membership operator for all enumeration types
  - Java
    - includes a class for set operations

Sets

- 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

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?
- Fundamental Pointer Operations:
  - Assignment of an address to a pointer
  - References (explicit versus implicit dereferencing)

Pointers

- 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*

Pointers

- 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)

Pointers

- Examples…
  - 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 *)

```
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)

Pointers

- Examples…
  - 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

      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

Pointers

- Examples…
  - 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

Pointers

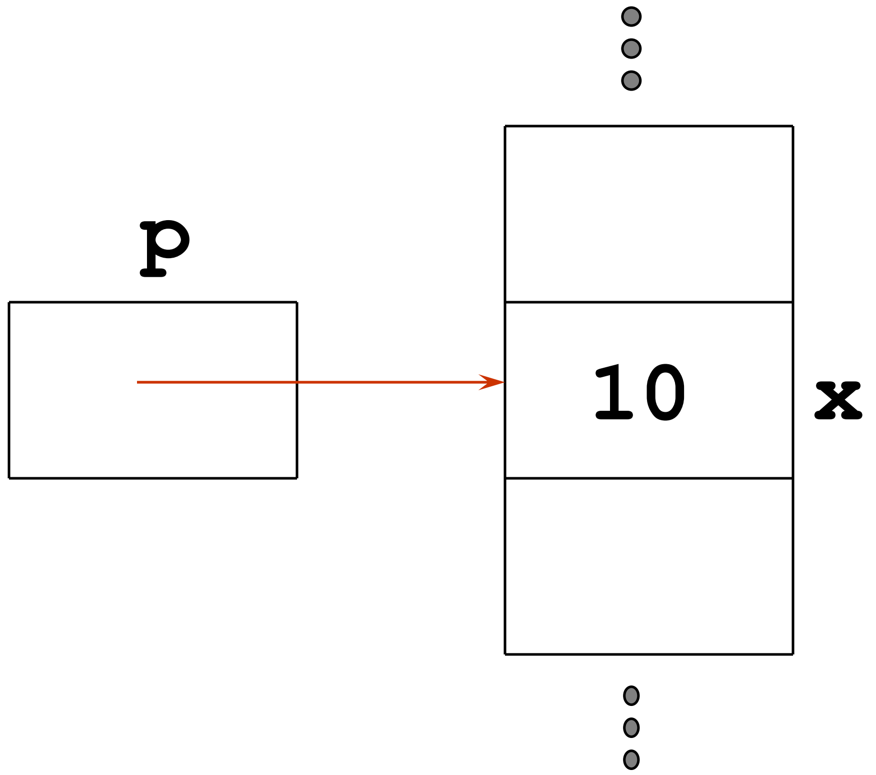
- Evaluation
  - Dangling pointers and dangling objects are problems, as is heap management
  - Pointers are like goto's--they widen the range of cells that can be accessed by a variable
  - Pointers or references are necessary for dynamic data structures--so we can't design a language without them

Pointers

- A pointer is a variable holding an address value

```
int x = 10;
int *p;

p = &x;
```



p contains the address of **x** in memory.

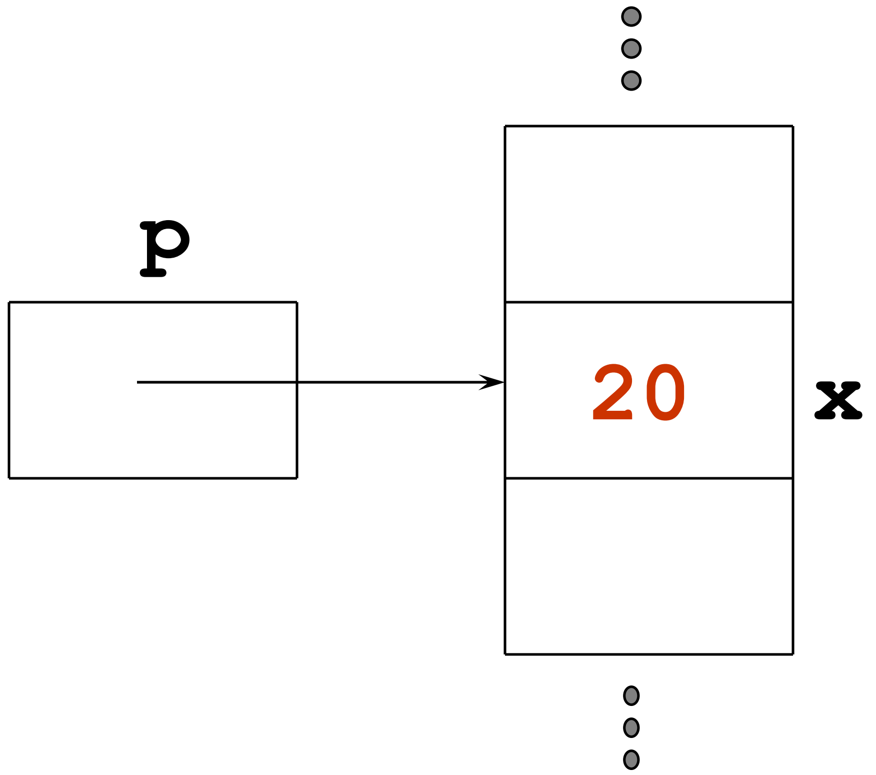
Pointers

- A pointer is a variable holding an address value

```
int x = 10;
int *p;

p = &x;

*p = 20;
```



`*p` refers to the value stored in x.

Pointers

```
int x = 10;
int *p;
```

Declares a pointer to an integer

```
p = &x;
```

`&` is **address** operator gets address of x

```
*p = 20;
```

`*` **dereference** operator gets value at `p`

Pointers

- Pointers are designed for two kinds of uses
  - Provide a method for indirect addressing

    (see example on the previous slides)
  - Provide a method of dynamic storage management

    ```
    int *ip = new int[100];
    ```

- Pointer dereferencing
  - Implicit: dereferenced automatically
    - In Fortran 90, pointers have no associated storage until it is allocated or associated by pointer assignment

      ```
      REAL, POINTER :: var
      ALLOCATE (var)
      var = var + 2.3
      ```

    (no special symbol needed to dereference)

  - Explicit: In C++, use dereference operator (*)

Problems with Pointers

- Dangling pointers (dangerous)
  - points to deallocated memory

```
int *p;
void trouble () {
    int x;
    *p = &x;
    return;
}
main() {
    trouble();
}
```

- Lost Heap-Dynamic Variables

```
int *p = new int[10];        /* p points to anonymous variable */
int y;
p = &y;                      /* space for anonymous variable lost */
```

Solutions to Dangling Pointer Problem

- Tombstones
  - Every heap-dynamic variable includes a special cell, called a tombstone, that is itself a pointer to the heap-dynamic variable
  - Actual pointer points only at tombstones and never to heap dynamic variables
  - When heap-dynamic variable is deallocated, tombstone remains but set to nil
  - This prevents pointer from ever pointing to a deallocated variable
  - Any reference to any pointer that points to nil tombstone can be detected as an error
  - Problem: costly in both time and space
    - Every access to heap-dynamic variable through a tombstone requires one more level of indirection, which consumes an additional machine cycle on most computers

Solutions to Dangling Pointer Problem

- Locks-and-keys approach
  - Pointer values are represented as ordered pairs (key,address)
  - Heap-dynamic variables are represented as storage for variable plus a header cell that stores an integer lock value
  - When heap-dynamic variable is allocated, a lock value is created and placed both in the lock cell (of heap-dynamic variable) and key cell (of pointer)
  - Every access to the dereferenced pointer compares key value of pointer to lock value of heap-dynamic variable
  - When heap-dynamic variable is deallocated, its lock value is cleared to an illegal lock value
  - When dangling pointer is dereferenced, its address value is still intact, but its key value no longer match the lock
- Leave deallocation to the runtime system
  - Garbage collection in Java

Type Checking

Generalize the concept of operands and operators to include subprograms and assignments

- *Type checking* is the activity of ensuring that the operands of an operator are of compatible types
- A *compatible type* is one that is either legal for the operator, or is allowed under language rules to be implicitly converted, by compiler-generated code, to a legal type.
- This automatic conversion is called a *coercion*.
- A *type error* is the application of an operator to an operand of an inappropriate type
- Note:

  If all type bindings are static, nearly all checking can be static

  If type bindings are dynamic, type checking must be dynamic

Strong Typing

A programming language is *strongly typed* if

- type errors are always detected
- There is strict enforcement of type rules with no exceptions.
- All types are known at compile time, i.e. are statically bound.
- With variables that can store values of more than one type, incorrect type usage can be detected at run-time.
- Strong typing catches more errors at compile time than weak typing, resulting in fewer run-time exceptions.

Which languages have strong typing?

- Fortran 77 isn't because it doesn't check parameters and because of variable equivalence statements.
- The languages Ada, Java, and Haskell are strongly typed.
- Pascal is (almost) strongly typed, but variant records screw it up.
- C and C++ are sometimes described as strongly typed, but are perhaps better described as weakly typed because parameter type checking can be avoided and unions are not type checked
- Coercion rules strongly affect strong typing—they can weaken it considerably (C++ versus Ada)

Type Compatibility

Type compatibility by name means the two variables have compatible types if they are in either the same declaration or in declarations that use the same type name

- Easy to implement but highly restrictive:
  - Subranges of integer types aren't compatible with integer types
  - Formal parameters must be the same type as their corresponding actual parameters (Pascal)

Type compatibility by structure means that two variables have compatible types if their types have identical structures

- More flexible, but harder to implement

Type Compatibility

Consider the problem of two structured types.

Suppose they are circularly defined

- Are two record types compatible if they are structurally the same but use different field names?
- Are two array types compatible if they are the same except that the subscripts are different? (e.g. [1..10] and [-5..4])
- Are two enumeration types compatible if their components are spelled differently?

With structural type compatibility, you cannot differentiate between types of the same structure (e.g. different units of speed, both float)

Type Compatibility Language examples

Pascal: usually structure, but in some cases name is used (formal parameters)

C: structure, except for records

Ada: restricted form of name

- Derived types allow types with the same structure to be different
- Anonymous types are all unique, even in:

```
A, B : array (1..10) of INTEGER:
```

Summary

- The data types of a language are a large part of what determines that language’s style and usefulness
- The primitive data types of most imperative languages include numeric, character, and Boolean types
- The user-defined enumeration and subrange types are convenient and add to the readability and reliability of programs
- Arrays and records are included in most languages
- Pointers are used for addressing flexibility and to control dynamic storage management