Topics: Data Types

- Primitive Data Types
- Character String Types
- Enumeration Types
- Array Types
- Associative Arrays

Note: Today's office hours changed to 12:00-1:00pm

Terms

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

Primitive Data Types

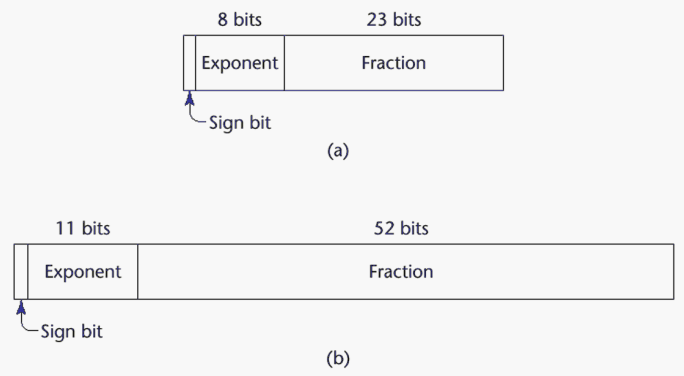
- Almost all programming languages provide a set of *primitive data types*
- Primitive data types: Those not defined in terms of other data types
- Some primitive data types are merely reflections of the hardware
- Others require only a little non-hardware support for their implementation

Primitive Data Types: 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`**

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



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

Primitive Data Types: Decimal

- For business applications (money)
  - Essential to COBOL
  - C# offers a decimal data type
- Store a fixed number of decimal digits, in coded form (BCD)
- *Advantage*: accuracy
- *Disadvantages*: limited range, wastes memory

Primitive Data Types: Boolean

- Simplest of all
- Range of values: two elements, one for "true" and one for "false"
- Could be implemented as bits, but often as bytes
  - Advantage: readability

Primitive Data Types: Character

- Stored as numeric codings
- Most commonly used coding: ASCII
- An alternative, 16-bit coding: Unicode (UCS-2)
  - Includes characters from most natural languages
  - Originally used in Java
  - Now supported by many languages
- 32-bit Unicode (UCS-4)
  - Supported by Fortran, starting with 2003

Character String Types

- Values are sequences of characters
- Design issues:
  - Is it a primitive type or just a special kind of array?
  - Should the length of strings be static or dynamic?

Character String Types – Operations

- Typical operations:
  - Assignment/copying
  - Comparison (=, >, etc.)
  - Catenation
  - Substring reference
  - Pattern matching

Character String Type in Certain Languages

- 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
- Fortran and Python
  - Primitive type with assignment and several operations
- Java (and C#, Ruby, and Swift)
  - Primitive via the `String` class
- Perl, JavaScript, Ruby, and PHP
  - Provide built-in pattern matching, using regular expressions

Character String Length Options

- *Static Length*: COBOL, Java's `String` class
- *Limited Dynamic Length*: C and C++
  - In these languages, a special character is used to indicate the end of a string's characters, rather than maintaining the length
- *Dynamic Length* (no maximum): SNOBOL4, Perl, JavaScript

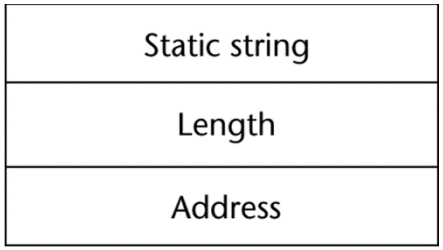
Character String Type Evaluation

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

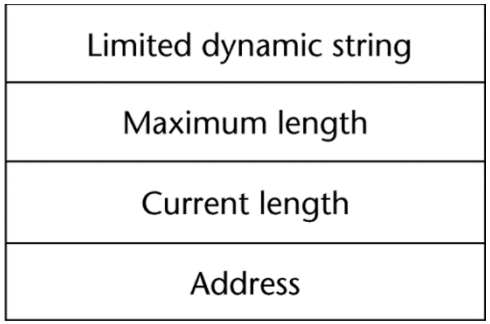
Character String Implementation

- 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

Compile- and Run-Time Descriptors



Compile-time descriptor for static strings



Run-time descriptor for limited dynamic strings

Enumeration Types

- All possible values, which are named constants, are provided in the definition
- 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?

Evaluation of Enumerated Type

- Aid to readability, e.g., no need to code a color as a number
- Aid to reliability, e.g., compiler can check:
  - operations (don't allow colors to be added)
  - No enumeration variable can be assigned a value outside its defined range
  - C#, F#, Swift, and Java 5.0 provide better support for enumeration than C++ because enumeration type variables in these languages are not coerced into integer types

Array Types

- An array is a homogeneous aggregate of data elements in which an individual element is identified by its position in the aggregate, relative to the first element.

Array 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?
- Are ragged or rectangular multidimensional arrays allowed, or both?
- What is the maximum number of subscripts?
- Can array objects be initialized?
- Are any kind of slices supported?

Array Indexing

- *Indexing* (or subscripting) is a mapping from indices to elements

  array_name (index_value_list) → an element
- Index Syntax
  - Fortran and Ada use parentheses
    - Ada explicitly uses parentheses to show uniformity between array references and function calls because both are *mappings*
  - Most other languages use brackets

Arrays Index (Subscript) Types

- FORTRAN, C: integer only
- Java: integer types only
- Index range checking
  - C, C++, Perl, and Fortran do not specify range checking
  - Java, ML, C# specify range checking

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 elaboration time
  - Advantage: space efficiency

Subscript Binding and Array Categories (continued)

- *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
- Perl, JavaScript, Python, and Ruby support heap-dynamic arrays

Array Initialization

- Some language allow initialization at the time of storage allocation
  - C, C++, Java, Swift, and C#
  - 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"};
```

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

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

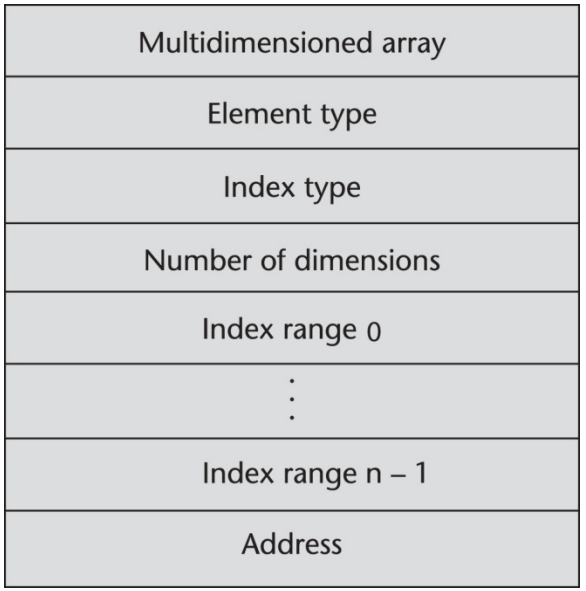
- Access function maps subscript expressions to an address in the array
- Access function for single-dimensioned arrays:

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

  address(list[k]) = address (list[0])
  + k * element_size

Accessing Multi-dimensioned Arrays

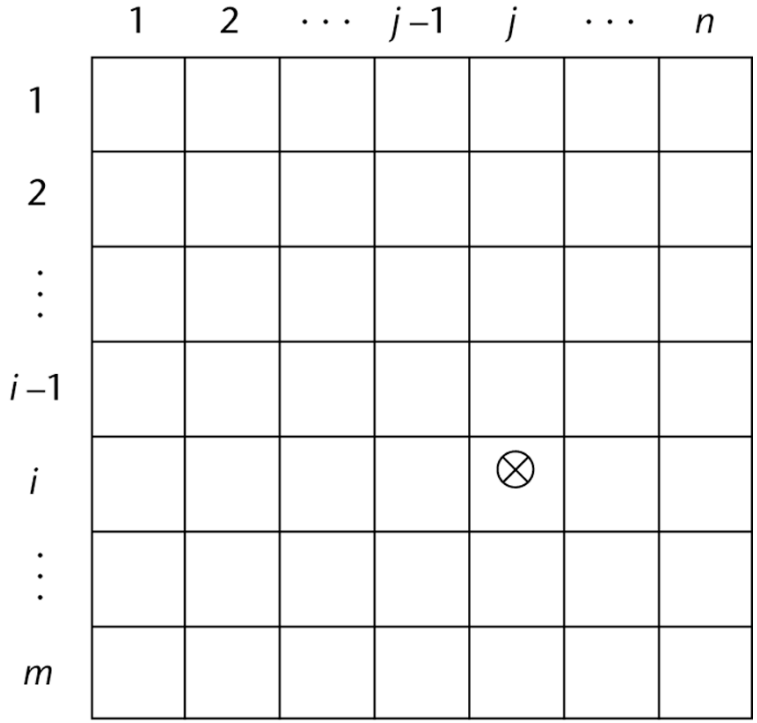
- Two common ways:
  - Row major order (by rows) – used in most languages
  - Column major order (by columns) – used in Fortran
  - A compile-time descriptor for a multidimensional array



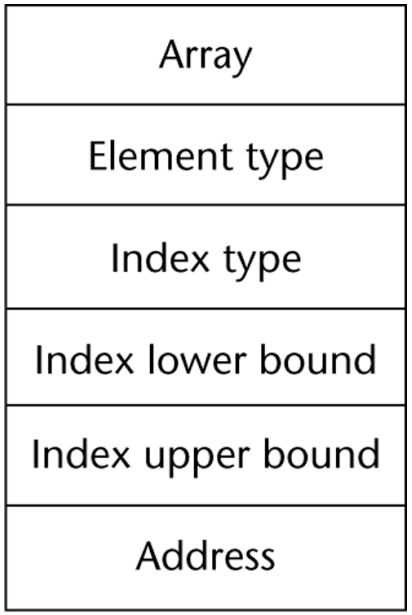
Locating an Element in a Multi-dimensioned Array

- General format

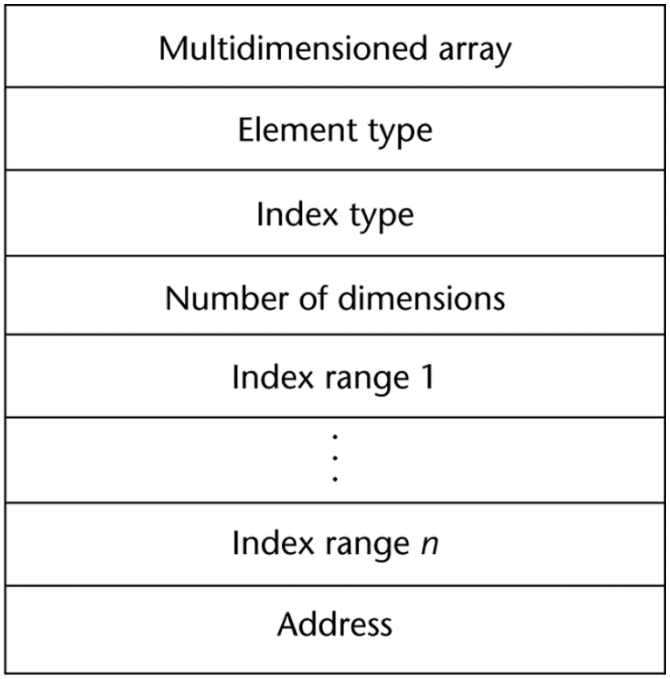
  Location (a[I,j]) = address of a [row_lb,col_lb] + (((I – row_lb) * n) + (j – col_lb)) * element_size



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 Swift

Associative Arrays in Perl

- Names begin with `%`; literals are delimited by parentheses

```
%hi_temps = ("Mon" => 77, "Tue" => 79, "Wed" =>
  65, …);
```

- Subscripting is done using braces and keys

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

  - Elements can be removed with **`delete`**

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