# COS 301

# Programming Languages

**Sebesta Chapter 6**

Data Types

## Topics

## What is a Type?

- A *type* is a collection of values and operations on those values.
- Example: Integer type has values ..., -2, -1, 0, 1, 2, ... and operations +, -, *, /, <, ...
- The Boolean type has values true and false and operations AND, OR, NOT
- We can generally distinguish 3 levels of typing:
  - Types that correspond to machine level types
  - Types supplied as part of a language
  - Programmer-defined 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?

## Evolution

- FORTRAN: Arrays, reals, ints
- COBOL: allowed programmer to specify accuracy; provided records
- Algol 68: few basic types + structure defining mechanisms (user defined types)
- 1980's: Abstract data types
- Evolved into objects (first developed in 1960's)

## Types in Early Languages

- In the early languages, Fortran, Algol, Cobol, all of the types were built in.
  - If you needed a type to represent colors, you could use integers; but what does it mean to multiply two colors?
  - If you needed a type to represent days of the week, you could use integers, but what is (Mon + Fri) / Tuesday?
- The purpose of types in programming languages is to provide ways of effectively modeling a problem and its solution.

## Levels of Abstraction

- Ultimately every computable problem has to be expressed as a string of 1's and 0's; likewise the solution is also thus represented
- Assembly languages were invented to provide mnemonics (such as SUB and LOAD) and human-readable numbers to replace strings of 1's and 0's
- HLLs were invented to provide a virtual machine that hides the real machine with its registers and native machine types
  - Increasingly sophisticated typing schemes have been invented to map human abstractions such as colors, days, documents, weather systems in language that corresponds to the human abstraction rather than to any machine, virtual or otherwise
  - OO programming just another level of typing abstraction

## 5.1 Type Errors

- Machine data carries no type information.
- Machines contain bit strings. Example:
- 0100 0000 0101 1000 0000 0000 0000 0000

## Bit Strings

- 0100 0000 0101 1000 0000 0000 0000 0000
- Can be
  - The floating point number 3.375
  - The 32-bit integer 1,079,508,992
  - Two 16-bit integers 16472 and 0
  - Four ASCII characters: @ X NUL NUL
- What else?
- What about 1111 1111?

## Type Errors

- A *type error* is any error that arises because an operation is attempted on a data type for which it is undefined.
- Type errors are easy to make in assembly language programming.
- HLLs help to reduce the number of type errors.
  - As the level of abstraction rises, type errors tend to decrease
- A *type system* provides a basis for detecting type errors.

## 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 in C, Ada, Java

| Type | C | Ada | Java |
|---|---|---|---|
| **Byte** | char | | byte |
| **Integer** | short, int, long | Integer, Natural, Positive | short, int, long |
| **Float** | float, double, extended double | Float, Decimal | float, double |
| **Char** | char | Character | char |
| **Bool** | | Boolean | boolean |

Notes:
1. C char and integer types can be signed or unsigned
2. Ada provides Natural and Positive as subclasses of integer
3. All Java integers are signed
4. Ada provides floating and fixed point types with programmer-specified precision

## 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
  - C and C++ support signed and unsigned `byte, short, int, long`
    - Signed = 2's complement
  - Java's supports signed integer sizes: `byte, short, int, long`
- C/C++ mapping to machine data types can cause problems
  - What do relational operators mean when we have signed on one side and unsigned on the other?

## Primitive Data Types: Integer

- When integers are mapped onto machine types, they have a limited range but arithmetic is very efficient
- Starting with LISP, some languages support a conceptually unlimited integer
  `984375928418932092850984590384590835`
- Starting with LISP, some languages support a conceptually unlimited integer
- Modern languages include Python, Haskell, Smalltalk
  - A language implementation may use a machine representation for small integers and then switch to a less efficient representation when needed

## Integers mapped to machine types

- Most languages do not specify the number of bits associated with any of the basic types – this is implementation dependent
  - Java is an exception: byte = 8, short=16, int = 32, long=64
  - Ada allows programmer to specify size, will raise exception at compile time if too many bits
- Many languages fail to generate an exception when overflow occurs.
  - This is an example of the conflict between run-time efficiency and type safety. What is the cost imposed by generating exceptions on overflow?

## Overflow and Wraparound

- In most languages, the numeric types are finite in size and correspond to a particular word size.
- So a + b may overflow the finite range.
  - $\exists$ a,b,c | a + (b + c) $\neq$ (a + b) + c
- Due to the peculiarities of the most common machine representation for signed integers (2's complement), an overflow of positive numbers results in a sum with a negative sign
- Also in some C-like languages, the equality and relational operators produce an int, not a Boolean

## Primitive Data Types: Floating Point

- Model real numbers, but only as approximations
  - There are significant problems with floating point computation
  - Machine representations generally only represent sums of powers of two
  - Apparently easy and tractable numbers such as 0.1 have no precise binary representations
- 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
  - Some older scientific data is only accessible through software simulation of old hardware because formats have changed and older numbers are no longer precisely representable
- Current standard is IEEE Floating-Point Standard 754

## IEEE Floats

- Uses "binamal": $1.011_2 = 2.375$
- Because there are only two digits in binary, IEEE formats normalize numbers so that a 1 in front of the binamal point so that every number other than 0 will start with a 1.
- The leading 1 is NOT stored in the number.
- This provides an extra bit of precision and is sometimes referred to as a HIDDEN BIT.

| Sign Bit | Bias-127 Exponent E 8-bits | Unsigned Fraction F 23 bits |
|---|---|---|
| | | |

```
Val = (-1)^Sign * 1.F*2^(E-127) if E <> 0
```

## IEEE 64-bit Format (double)

- Range of float is approximately $\pm 10^{38}$ with 6-7 digits of precision for 32 bits
- $\pm 10^{308}$ with 14-15 digits of precision for 64 bits

| Sign Bit | Bias-1023 Exponent E 11-bits | Unsigned Fraction F 52 bits |
|---|---|---|
| | | |

```
Val = (-1)^Sign * 1.F*2^(E-1023) if E <> 0
```

## 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

- Usually for business applications (money)
  - Essential to COBOL
  - C# offers a decimal data type
  - Almost any DBMS offers decimal data type for storage
- Stores a fixed number of decimal digits, in binary coded decimal (BCD)
- *Advantage*: accuracy – more precise than IEEE floats
- *Disadvantages*: limited range, uses more memory and much more CPU time than floats
- Some hardware has direct support

## Primitive Data Types: Boolean

- Simplest of all types
- Range of values: two elements, one for "true" and one for "false"
- Could be implemented as bits, but usually at least bytes
  - Advantage: readability
- C (until 1999) did not have a Boolean type
  - Booleans were represented as ints
- Many languages cast freely between Booleans and other types
  - 0 = false, anything else = true
  - "" = false, non-empty string = true

## Booleans

- Booleans are problematic in some languages that support special values to indicate missing or uninitialized data
  - Null
  - Empty
  - Missing
- What is false && null?

## Booleans in PHP

- PHP is a bit tricky because it coerces types freely and uses a rather strange internal representation for Booleans
  - 1 = TRUE        "" = false

```
<?php
echo "The value of TRUE is->",TRUE,"<-\n";
echo "The value of FALSE is->",FALSE,"<-\n";
?>
Output:
The value of TRUE is->1<-
The value of FALSE is-><-
```

## Primitive Data Types: Character

- Characters are stored as numeric codings
- American Standard Code for Information Interchange (ASCII) was a long time standard
- Extended Binary Coded Decimal Interchange Code (EBCDIC) was used by IBM mainframes
- Problem: ASCII is a 7 bit code and EBCDIC an 8-bit code
  - There are more than 128 characters that we might want to use

## ASCII Characters

- Languages that use ASCII implicitly normally use 8 bits to represent each character
- The meaning of the upper 128 characters varies with the operating system or other software
- Often the ISO 8859 encoding is used to represent characters used in European languages

## Unicode

- Unicode is a system designed to transcend character encodings - it is not simply an expansion of ASCII code
- To understand Unicode you must also understand UCS (Universal Character Set) or ISO 10646
- UCS is like a giant alphabet (32 bits) designed to encode any human character known
  - And some that aren't human – it includes a "private use area" that has been used for Klingon characters among other things
- Unicode provides algorithms for encoding UCS characters

## Recommended Reading - Unicode

- An excellent and concise introduction:

  The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)

  At

  http://www.joelonsoftware.com/articles/Unicode.html

## From unicode.org

- See http://www.unicode.org/standard/WhatIsUnicode.html

  Unicode provides a unique number for every character, no matter what the platform, no matter what the program, no matter what the language. The Unicode Standard has been adopted by such industry leaders as Apple, HP, IBM, JustSystems, Microsoft, Oracle, SAP, Sun, Sybase, Unisys and many others. Unicode is required by modern standards such as XML, Java, ECMAScript (JavaScript), LDAP, CORBA 3.0, WML, etc., and is the official way to implement ISO/IEC 10646. It is supported in many operating systems, all modern browsers, and many other products. The emergence of the Unicode Standard, and the availability of tools supporting it, are among the most significant recent global software technology trends.

## Unicode

- Unicode can be implemented with different character encodings
  - Most common are UTF-8, UTF-16 (UCS-2), and UTF-32 (UCS-4)
  - UTF 8 is a variable length encoding that conveniently encodes ASCII in single bytes
- Unicode has been adopted by many modern languages and nearly all popular operating systems
  - Java, XML, .NET framework, Python, Ruby etc.
- For a Unicode tutorial, see

http://www.unicode.org/notes/tn23/tn23-1.html

## Character String Types

- String 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 and copying
  - Comparison (=, >, etc.)
  - Catenation
  - Substring reference
  - Pattern matching

## String Libraries

- As string processing has become increasingly important library support has become much more extensive
- Concatenation, substring extraction, case conversion, searching, pattern matching, regular expressions, substring replacement, ….

## Example: PHP string functions 1

- addcslashes — Quote string with slashes in a C style
- addslashes — Quote string with slashes
- bin2hex — Convert binary data into hexadecimal representation
- chop — Alias of rtrim
- chr — Return a specific character
- chunk_split — Split a string into smaller chunks
- convert_cyr_string — Convert from one Cyrillic character set to another
- convert_uudecode — Decode a uuencoded string
- convert_uuencode — Uuencode a string
- count_chars — Return information about characters used in a string
- crc32 — Calculates the crc32 polynomial of a string
- crypt — One-way string encryption (hashing)
- echo — Output one or more strings
- explode — Split a string by string
- fprintf — Write a formatted string to a stream
- get_html_translation_table — Returns the translation table used by htmlspecialchars and htmlentities
- hebrev — Convert logical Hebrew text to visual text
- hebrevc — Convert logical Hebrew text to visual text with newline conversion
- html_entity_decode — Convert all HTML entities to their applicable characters
- htmlentities — Convert all applicable characters to HTML entities

## Example: PHP string functions 2

- html_entity_decode — Convert all HTML entities to their applicable characters
- htmlentities — Convert all applicable characters to HTML entities
- htmlspecialchars_decode — Convert special HTML entities back to characters
- htmlspecialchars — Convert special characters to HTML entities
- implode — Join array elements with a string
- join — Alias of implode
- lcfirst — Make a string's first character lowercase
- levenshtein — Calculate Levenshtein distance between two strings
- localeconv — Get numeric formatting information
- ltrim — Strip whitespace (or other characters) from the beginning of a string
- md5_file — Calculates the md5 hash of a given file
- md5 — Calculate the md5 hash of a string
- metaphone — Calculate the metaphone key of a string
- money_format — Formats a number as a currency string
- nl_langinfo — Query language and locale information
- nl2br — Inserts HTML line breaks before all newlines in a string
- number_format — Format a number with grouped thousands
- ord — Return ASCII value of character
- parse_str — Parses the string into variables

## Example: PHP string functions 3

- print — Output a string
- printf — Output a formatted string
- quoted_printable_decode — Convert a quoted-printable string to an 8 bit string
- quoted_printable_encode — Convert a 8 bit string to a quoted-printable string
- quotemeta — Quote meta characters
- rtrim — Strip whitespace (or other characters) from the end of a string
- setlocale — Set locale information
- sha1_file — Calculate the sha1 hash of a file
- sha1 — Calculate the sha1 hash of a string
- similar_text — Calculate the similarity between two strings
- soundex — Calculate the soundex key of a string
- sprintf — Return a formatted string
- sscanf — Parses input from a string according to a format
- str_getcsv — Parse a CSV string into an array
- str_ireplace — Case-insensitive version of str_replace.
- str_pad — Pad a string to a certain length with another string
- str_repeat — Repeat a string
- str_replace — Replace all occurrences of the search string with the replacement str_rot13 — Perform the rot13 transform on a string
- str_shuffle — Randomly shuffles a string

## Example: PHP string functions 4

- str_split — Convert a string to an array
- str_word_count — Return information about words used in a string
- strcasecmp — Binary safe case-insensitive string comparison
- strchr — Alias of strstr
- strcmp — Binary safe string comparison
- strcoll — Locale based string comparison
- strcspn — Find length of initial segment not matching mask
- strip_tags — Strip HTML and PHP tags from a string
- stripcslashes — Un-quote string quoted with addcslashes
- stripos — Find position of first occurrence of a case-insensitive string
- stripslashes — Un-quotes a quoted string
- stristr — Case-insensitive strstr
- strlen — Get string length
- strnatcasecmp — Case insensitive string comparisons using a "natural order" algorithm
- strnatcmp — String comparisons using a "natural order" algorithm
- strncasecmp — Binary safe case-insensitive string comparison of the first n characters
- strncmp — Binary safe string comparison of the first n characters

## Example: PHP string functions 5

- strpbrk — Search a string for any of a set of characters
- strpos — Find position of first occurrence of a string
- strrchr — Find the last occurrence of a character in a string
- strrev — Reverse a string
- strripos — Find position of last occurrence of a case-insensitive string in a string
- strrpos — Find position of last occurrence of a char in a string
- strspn — Finds the length of the first segment of a string consisting entirely of characters contained within a given mask.
- strstr — Find first occurrence of a string
- strtok — Tokenize string
- strtolower — Make a string lowercase
- strtoupper — Make a string uppercase
- strtr — Translate certain characters
- substr_compare — Binary safe comparison of 2 strings from an offset, up to length characters
- substr_count — Count the number of substring occurrences
- substr_replace — Replace text within a portion of a string
- substr — Return part of a string
- trim — Strip whitespace (or other characters) from the beginning and end of a stringstrncmp — Binary safe string comparison of the first n characters * ucfirst — Make a string's first character uppercase
- ucwords — Uppercase the first character of each word in a string
- vfprintf — Write a formatted string to a stream
- vprintf — Output a formatted string
- vsprintf — Return a formatted string
- wordwrap — Wraps a string to a given number of characters

## Character String Type in Various Languages

- C and C++
  - Not primitive
  - Use **`char`** arrays and a library of functions that provide operations
- With a non-primitive type simple variable assignment can't be used

```
char line[MAXLINE];
char filename[20];
char *p;
if(argc==2) strcpy(filename, argv[1]);
```

- Note that C does not check bounds in strcpy
- In the example above we have an opening for a buffer overflow attack - copying a string from a command line
- C++ does provide a string class that is more sophisticated than standard C strings

## Character String Type in Various Languages

- SNOBOL4 (a string manipulation language)
  - Primitive
  - Many operations, including elaborate pattern matching
- Fortran and Python
  - Primitive type with assignment and several operations
- Java
  - Primitive via the `String` class
- Perl, JavaScript, Ruby, and PHP
  - Provide built-in pattern matching, using regular expressions
  - Extensive libraries

## Character String Length Options

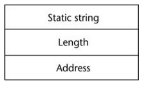
- Static: Fixed length set when string is created
  - COBOL, Java's `String` class, .NET String class etc.
- *Limited Dynamic Length*: C and C++
  - In these languages, a delimiter is used to indicate the end of a string's characters, rather than maintaining the length
- *Dynamic* (no maximum): SNOBOL4, Perl, JavaScript
  - Expensive computationally (garbage collection)
- Ada supports all three string length options
- Most DBMS provide three string types:

| | |
|---|---|
| – Char | fixed |
| – Varchar(n) | vary to max specified |
| – Text or BLOB | unlimited |

## Character String Implementation

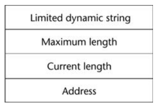
- Strings are rarely supported in hardware
- Static length: compile-time descriptor
- Limited dynamic length: may need a run-time descriptor for length (but not in C and C++)
- Dynamic length: need run-time descriptor; allocation/de-allocation is the biggest implementation problem

## Compile- and Run-Time Descriptors



| |
|---|
| Static string |
| Length |
| Address |

Compile-time descriptor for static strings



| |
|---|
| Limited dynamic string |
| Maximum length |
| Current length |
| Address |

Run-time descriptor for limited dynamic strings

## 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`
- User-defined ordinal types fall into 2 groups:
  - Enumerations
  - Subranges

## 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};
```

- Pascal example (with subranges)

```
Type
   Days = (monday,tuesday,wednesday,thursday,
           friday, saturday,sunday);
    WorkDays = monday .. friday;
    WeekEnd = Saturday .. Sunday;
```

## Enumeration Types

- First appeared in Pascal and C
- Pascal-like languages allow array subscripting by enumerations

```
var schedule : array[Monday..Saturday] of string;
var beerprice : array[Budweiser..Guinness] of real;
```

- Primary purpose of enumerations is enhance readability of code
- Some languages treat enums as integers and perform implicit conversions
- Others such as Java or Ada have strict type-checking and require explicit conversions

## Enumeration Types

- Interestingly are not supported by any of the major modern scripting languages:
  - Perl, Javascript, PHP, Python, Ruby, Lua
  - Added to Java in version 5.0 (after 10 years)
- 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?

    ```
    for (day = Sunday; day <= Saturday; day++)
    ```

  - Any other type coerced to an enumeration type?

    ```
    day = monday * 2;
    ```

## Why use Enumerated Types?

- 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
  - Ada, C#, and Java 5.0 provide better support for enumeration than C++ because enumeration type variables in these languages are not coerced into integer types

## Subrange Types

- A subranges is an ordered contiguous subsequence of an ordinal type
  - Example: 12..18 is a subrange of integer type
- Ada's design

```
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;
```

## Why Subranges?

- Aid to readability
  - Make it clear to the readers that variables of subrange can store only certain range of values
- Reliability
  - Assigning a value to a subrange variable that is outside the specified range is detected as an error

## Implementation of User-Defined Ordinal Types

- Enumeration types are usually implemented as integers
  - The main issue is how well the compiler hides the implementation
- Subrange types are implemented like the parent types with code inserted (by the compiler) to restrict assignments to subrange variables

## Array Types

- Aside from character strings, arrays are the most widely used non-primitive data type
  - Especially when we consider that strings are usually really arrays
- Classical Definition:
  - An array is an aggregate of homogeneous data elements in which an individual element is identified by its position in the aggregate, relative to the first element.

## 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?
- 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, PL/I, Ada, Basic, Pascal use parentheses
    - Ada explicitly uses parentheses to show uniformity between array references and function calls because both are *mappings*
  - Many other languages use brackets

## Arrays Index (Subscript) Types

- FORTRAN, C: integer only
- Ada, Pascal : any ordinal type, including integer, integer subranges, enumerations, Boolean and characters
- Java: integer types only
- Index range checking
  - A clear conflict between safety and efficiency
  - Lack of bounds checking allows buffer overflow attacks
- C, C++, Perl, and Fortran do not specify range checking
  - Java, ML, C# specify range checking
  - In Ada, the default is to require bounds checks but it can be turned off

## Perl prefix chars

- Arrays are declared in Perl with @ but indexed elements are scalars so references to elements use $

```
@friends = ("Rachel", "Monica", "Phoebe", "Chandler",
  "Joey", "Ross");
# prints "Phoebe"
print $friends[2];
# prints "Joey"
print $friends[-2];
```

## Implicit lower bounds

- In all C descendants and many other curly-brace languages the implicit lower bound of any array in any dimension is 0
- Fortran implicit base is 1
- Pascal-like languages and many BASICs allow arbitrary lower bounds
- Some Basics provide an Option Base statement to set the implicit base

## Subscript Binding and Array Categories

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

## Subscript Binding and Array Categories

- *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

- 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

- C and C++
  - Arrays declared outisde function bodies or that include `static` modifier are static
  - C and C++ arrays in function bodies and 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

## Sparse Arrays

- A few languages such as Javascript support sparse arrays where subscripts need not be contiguous
- A family of database management systems starting with M is based on sparse matrices
  - You can regard any database system almost as simply a programming language with persistent storage
- Currently represented by Intersystems Cachè

## Array Initialization

- Initialization in source code
  - 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"};
```

## Array Initialization

- Ada

```
Primary : array(Red .. Violet) of Boolean =
  (True, False, False, True, False);
```

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

```
$fruits = array (
 "fruits"  => array("a" => "orange", "b" => "banana", "c"
  => "apple"),
 "numbers" => array(1, 2, 3, 4, 5, 6),
 "holes"   => array("first", 5 => "second", "third"));
```

## Python List Comprehensions

- This list feature first appeared in Haskell but Smalltalk has a more general approach where a block of code can be passed to any iterator
- A function is applied to each element of an array and a new array is constructed

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

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

the conditional filters the results of range(12)

## Autmatic Array Initialization

- Some languages will pre-initialize arrays; e.g., Java and most BASICs:
  - Numeric values set to 0
  - Characters to \0 or \u0000
  - Booleans to false
  - Objects to null pointers
- Relying on automatic initialization can be a dangerous programming practice

## Array Operations

- Array operations work on the array as a single object
  - Assignment
  - Catenation (concatenation)
  - Equality / Inequality
  - Array slicing

  - C/C++/C# : none
  - Java: assignment
  - Ada: assignment, catenation
  - Python: numerous operations but assignment is reference only
- The difference between deep and shallow copy becomes important in array assignment
  - Deep copy: a separate copy where all elements are copied as well
  - Shallow copy: copy reference only

## Array Operations – Implied Iterators

- Fortran 95 has “elemental” array operations
  - Ex: C = A + B results in array C with the some of each element in A and B
  - Assignment, arithmetic, relational and logical operators
- APL provides the most powerful array processing operations for vectors and matrixes as well as unary operators (for example, to reverse column elements)
- Some APL operators take other operators as arguments

## 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)
- Subscripting expressions vary:

```
arr[3][7]      arr[3,7]
```

## Type signatures

In C

```
float x[3][5];
```

type of x: float[ ][ ]

type of x[1]: float[ ]

type of x[1][2]: float

## Arrays in Dynamically Typed Languages

- Most languages with dynamic typing allow arrays to have elements of different types
  - The array itself is implemented as an array of pointers
  - Many of these languages have dynamic array sizing
  - Many of these languages have built in support for one-dimensional arrays called lists
  - Recursive arrays can be created in some languages, where an array includes itself as an element

## 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

- Fortran 95

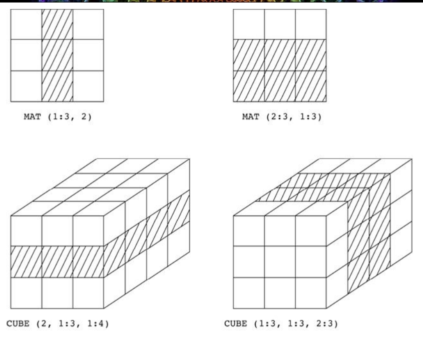
```
Integer, Dimension (10) :: Vector
Integer, Dimension (3, 3) :: Mat
Integer, Dimension (3, 3) :: Cube
```

`Vector (3:6)` is a four element array

- Ruby supports slices with the `slice` method

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

## Slice Examples in Fortran 95



## Python Lists and Slices

- Example from Python:

  ***B = [33, 55, 'hello','R2D2']***
- Elements can be accessed with subscripts; B[0] = 33
- A *slice* is a contiguous series of entries:
  - Ex: B[1:2]  B[1:]  B[:2]  B[-2:]
- Since strings are treated as character arrays slicing is very useful for string operations
- + is used as the concatenation operator

## Implementation of Arrays

- Requires a lot more compile-time effort than implementing scalar variables and primitive types
  - To access an element of an array we need an access function that maps subscript expressions to an address in the array
  - This code has to support as many array dimensions as the language allows

## Vectors

- Access function for single-dimensioned arrays:
  - k is the desired element
  - lb is the lower bound (0 for C-like languages)

  **`addr(list[k]) = addr(list[lb]) + ((k-lb) * eltsize)`**
- These operations are performed at runtime
- Many computer architectures have indirect addressing modes that can perform part or all of vector element address computation

## Array Storage Order

- 2-D arrays of scalar types can be stored in either row-major or column-major order
  - In row-major order the elements of the first row are stored contiguously, followed by the 2nd row, etc.
  - In column-major order elements of the first column are stored contiguously, followed the 2nd, etc.
- C and most other languages use row-major order
- Notable exceptions are Fortran and Matlab
- For higher dimensions, row-major = first to last and column major = last to first

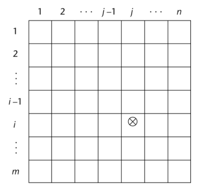
## Array Storage Order

- Why does this matter?
  - Significant difference in memory access time if elements are accessed in the wrong order
  - Essential to know for mixed-language programming
- Calculation of element addresses
  - Row-Major: Addr(c[i][j]) = addr(C[0][0]) + e (ni +j)
    - Where e is element size and n number of elts in row
  - Col-Major: Addr(c[i][j]) = addr(C[0][0]) + e (mj +i)
    - Where e is element size and m number of elts in col

## Locating an Element in an n-dimensioned Array

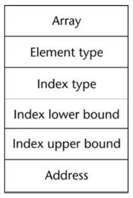
•General format

```
Addr(a[i,j]) = addr of a[row_lb,col_lb]
+ (((i - row_lb) * n) + (j - col_lb)) * elt_size
```

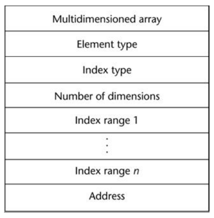


•For each additional dimension we need one more addition and one more multiplication

## Compile-Time Descriptors (Dope Vectors)



Single-dimensioned array



Multi-dimensional 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
  - Also called dictionary or hash
- Design issues:
  - What is the form of references to elements?
  - Is the size static or dynamic?
- Built-in type in Perl, Python, PHP, Ruby, and Lua
- Many other languages provide class with similar functionality
  - .NET has a variety of collection classes
  - Smalltalk has dictionaries

## Associative Arrays in Perl

- Called “hashes” in Perl because elements are stored and retrieved with hash functions
  - 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"};
    ```
  - Clear array with {}

    ```
    $hi_temps = {};
    ```
- Size is dynamic

## Associative Arrays in PHP

- All arrays in PHP are inherently associative, but associative arrays are also indexed numerically
- PHP arrays are therefore ordered collections
  - No special naming conventions
  - `$hi_temps = Array("Mon"=>77,"Tue"=>79,“Wed”=>65, …);`
  - Subscripting is done using square braces and keys or indices

    ```
    $hi_temps["Wed“] = 83;
    $hi_temps[2] = 83;
    ```
  - Elements can be added with []

    ```
    $hi_temps[] = 99;
    ```
- Size is dynamic
- PHP has a very rich set of array functions
- In web form processing, the query string is available as an array $_GET[] and form post values are $_POST[]

## Implementing Associative Arrays

- Perl uses a hash function for fast lookups but is optimized for fast reorganization
  - Uses a 32 bit hash value for each entry but only a few bits are used for small arrays
  - To double the array size use one more bit and move half the existing entries
- PHP also uses a hash function but stores arrays in a linked list for easy traversal
  - An array with both associative and numeric indices can develop gaps in the numeric sequence

## Record Types

- A *record* is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names
  - Record types are closely related to both relational and hierarchical databases
- The main design issue is the syntactic form of references to the fields

## Record Types

- Used first in Cobol and PL/I but absent from Fortran, Algol 60
- In contrast to an array, structure elements are accessed by name rather than index
- Common to Pascal-like, C-like languages
  - Called struct in C, C++, C#; “record” in most other languages
- Part of all major imperative and OO languages except pre-1990 Fortran
  - Records have evolved to classes in OO languages
  - A record is simply an OO class with only instance variables or properties and no methods
  - Deliberately omitted from Java for this reason

## C Example

```
struct employeeType {
    int id;
    char name[25];
   int age;
   float salary;
   char dept;
};
struct employeeType employee;
...
employee.age = 45;
```

- Fields are usually allocated in a contiguous block of memory, but actual memory layout is compiler dependent
- Minimum memory allocation not guaranteed

## Definition of Records in COBOL

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

```
01 EMP-REC.
   02 EMP-NAME.
      05 FIRST PIC X(20).
      05 MID   PIC X(10).
      05 LAST  PIC X(20).
   02 HOURLY-RATE PIC 99V99.
```

- COBOL layouts have **levels,** from level 01 to level 49.
- Level 01 is a special case, and is reserved for the record level; the 01 level is the *name of the record*.
- Levels from 02 to 49 are all "equal" (level 02 is no more significant than level 03), but there is a hierarchy to the structure.
- Anyfield listed in a lower level (higher number) is subordinate to a field or group in a higher level (lower number). For example, FIRST and LAST in the example above belong to, the group EMP-NAME

## Definition of Records in Ada

- Record structures are indicated in an orthogonal way

```
type Emp_Name is record
    First: String (1..20);
    Mid: String (1..10);
    Last: String (1..20);
end record;
type Emp_Rec is record
    name: Emp_Name;
    Hourly_Rate: Float;
end record;
```

## References to Records

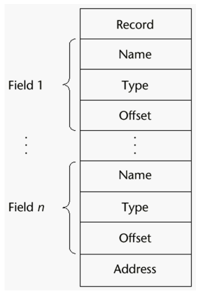
- Record field references
  1. COBOL
  field_name OF record_name_1 OF ... OF record_name_n
  2. Others (dot notation)
  record_name_1.record_name_2. ... record_name_n.field_name
- Fully qualified references must include all record names
- Elliptical references allow leaving out record names as long as the reference is unambiguous, for example in COBOL
  `FIRST`, `FIRST OF EMP-NAME`, and `FIRST OF EMP-REC` are elliptical references to the employee's first name

## Operations on Records

- Assignment is very common if the types are identical
  - But what does identical mean?
- Ada allows record comparison for equality or inequality
- Ada records can be initialized with aggregate literals
- COBOL provides `MOVE CORRESPONDING`
  - Copies a field of the source record to the corresponding field in the target record
  - Records need not be identical in structure

## Implementation of Record Types

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



## Union Types

- A *union* is a type whose variables are allowed to store different type values at different times during execution
- Design issues
  - Should type checking be required?
  - Should unions be embedded in records?

## Union Types

- The C union type pokes a large hole in static typing (large even for C with its relatively weak type system)

```
union { int a; float p} u;
u.a = 1;
x += u.p;
```

- Union types were developed because memory was a very scarce resource
- In the union type, memory is shared between the members of the union
  - But only one interpretation is correct depending on previous assignment

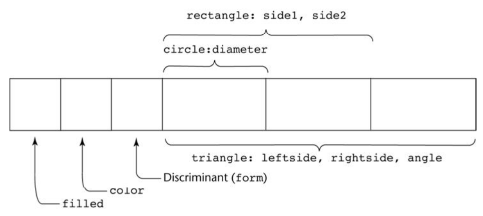
## Discriminated vs. Free Unions

- Fortran, C, and C++ provide union constructs in which there is no language support for type checking; the union in these languages is called *free union*
- Type checking of unions require that each union include a type indicator called a *discriminant*
  - Supported by Ada, Pascal

## 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



A discriminated union of three shape variables

## Unions

- Free unions are unsafe
  - Major hole in static typing
- Designed when memory was very expensive
- Little or no reason to use these structures today
  - Physical memory is much cheaper today than in the past
  - OS provides virtual memory that can provide a memory space many times the size of actual physical memory
  - Java and C# do not support unions
  - Ada's discriminated unions are safe but why use them?