COSC252: Programming Languages:

Basic Semantics: Data Types…

Jeremy Bolton, PhD
Asst Teaching Professor

GEORGETOWN UNIVERSITY

Common Types and Design Concerns

- Primitive Data Types
- Character String Types
- Enumeration Types
- Array Types
- Record Types
- List Types
- Union Types
- Pointer and Reference Types
- Type Checking
- Strong Typing
- Type Equivalence
- Why data types?

GEORGETOWN UNIVERSITY
Copyright © 2015 Pearson. All rights reserved.

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?

What is Data?

- Data – information, facts, details, substance of conveyance.
  - Generally stored or communicated via some media
  - Information Theory: Information is the *encoding* of data.
    - Digital or electric encoding
    - A digital sequence

- How do types help?

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`**

Binary Representations

- Data is encoded into computers as binary string of various lengths
  - Each bit (latch in hardware) can store a 0 or 1
  - A byte is an 8 bit sequence
  - Each unique binary string can be used to represent a different integer.
  - However, binary strings can be used to represent various data.

- EG: binary strings as integers (polynomial expansion)

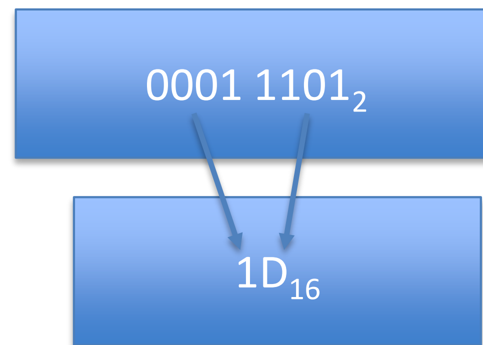
  $0001\ 1101_2$

  $2^7(0)+ 2^6(0)+ 2^5(0)+ 2^4(1) + 2^3(1)+ 2^2(1)+ 2^1(0)+ 2^0(1) = 29$

- Hexi-decimal Representation

  $16^1(1)+ 16^1(13) = 29$

  $1D_{16}$



GEORGETOWN UNIVERSITY

Data Types

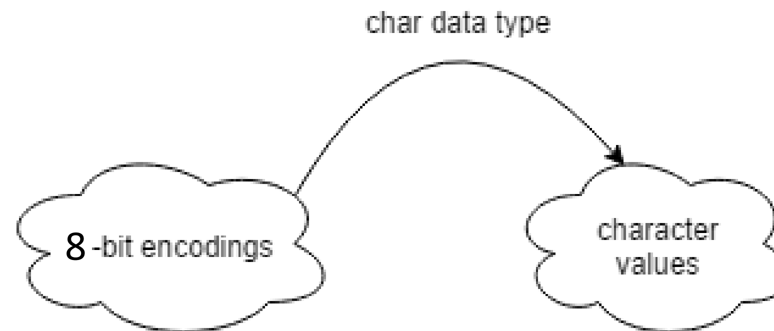
- Binary strings interpreted differently based on data type
  - Data type can be seen as a set of all possible values of some category; or more generally a data type is a mapping from set of all valid binary strings (of some length) to a value.

- Example: chars
  - Assumes 8-bit

EG:

$0100\ 0100 \quad \rightarrow \quad 'D'$

$0000\ 1010 \quad \rightarrow \quad '\backslash\text{n}'$



Integer Representation

- unsigned ints
  - Unsigned ints are generally represented using the standard polynomial expansion of binary sequences.

- ints are generally stored in “2’s complement” which allows for the representation of negative integers and allows for an efficient implementation of arithmetic.
  - Example (assuming 4-byte representation)
  - Conversion to two’s complement of negative int
    1. List binary representation of positive value.
    2. Invert binary digits
    3. Add 1
  - Example: What is two’s complement representation of -67?

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0100\ 0011,\ 0x0043$$

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011\ 1100,\ 0xFFBC$$

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011\ 1101,\ 0xFFBD$$

$0x0043$

GEORGETOWN UNIVERSITY

Memory requirements of common primitive data types (c++)

- Each data type may have different encoding schemes, as noted previously.
- Different data types may also have notably different lengths, or memory requirements.

Type Name	Bytes	Other Names	Range of Values
int	4	signed	–2,147,483,648 to 2,147,483,647
unsigned int	4	unsigned	0 to 4,294,967,295
short	2	short int, signed short int	–32,768 to 32,767
unsigned short	2	unsigned short int	0 to 65,535
long	4	long int, signed long int	–2,147,483,648 to 2,147,483,647
unsigned long	4	unsigned long int	0 to 4,294,967,295
long long	8	none (but equivalent to __int64)	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
short	2	short int, signed short int	–32,768 to 32,767
bool	1	none	false or true
char	1	none	–128 to 127 by default 0 to 255 when compiled by using

GEORGETOWN UNIVERSITY

Low-Level Operations on Data

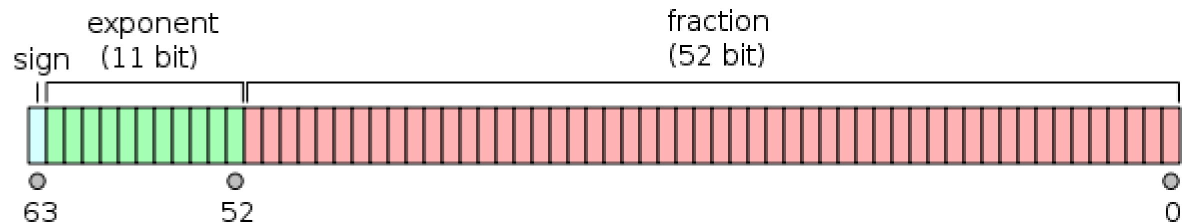
- Binary Operations
  - Bit-wise Logical OR: |
  - Bit-wise Logical AND: &
  - Bit-wise Logical NOT: ~
  - Bit-wise Logical XOR: ^
  - Bit-wise shift left: <<
  - Bit-wise shift right: >>

```
  0110 0001 1101 0100
| 1011 1000 1110 0100
  -------------------
  1111  1001 1111 0100

  0110 0001 1101 0100
& 1011 1000 1110 0100
  -------------------
  0010 0000 1100 0100

  0011 1000 1110 0100 << 2
  ------------------------
  1110  0011 1001 0000
```

Conceptually: Directly operates on binary sequence rather than "value"

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
- IEEE Floating-Point Standard 754



$$(-1)^{sign}\left(1 + \sum_{i=1}^{52} b_{52-i} 2^{-i}\right) 2^{e-1023}$$

GEORGETOWN UNIVERSITY

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
  - Why?

Challenge

- Design a BCD data type (for currency)
  - Remember: a data type is simply a mapping from a domain (binary sequences of a specific length) to a range (value or interpretation)

10.5

120.10

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
  - C# and JavaScript also support Unicode
- 32-bit Unicode (UCS-4)
  - Supported by Fortran, starting with 2003

GEORGETOWN UNIVERSITY

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 Type in Certain Languages

- C and C++
  - Not primitive
  - Use `char` arrays and a library of functions that provide operations
- Fortran and Python
  - Primitive type with assignment and several operations
- Java
  - Primitive via the `String` class
- Perl and Ruby

  - Provide built-in pattern matching, using regular expressions

Character String Length Options

- Static: Java's `String` class
  - **`immutable`**
- *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
  - **mutable**
- *Dynamic* (no maximum): SNOBOL4, Perl, JavaScript

Character String Type Evaluation

- Considerations:
  - 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

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

GEORGETOWN UNIVERSITY

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# 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.

GEORGETOWN UNIVERSITY

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?

GEORGETOWN UNIVERSITY

Array Indexing

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

  array_name (index_value_list) $\rightarrow$ 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

GEORGETOWN UNIVERSITY

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)

- *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)

- 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"};`
- 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)
- Python's array assignments, but they are only reference changes. Python also supports array catenation and element membership operations
- Ruby also provides array catenation

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
- F# and C# support rectangular arrays and jagged arrays

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

Multidimensioned array
Element type
Index type
Number of dimensions
Index range 0
⋮
Index range n – 1
Address

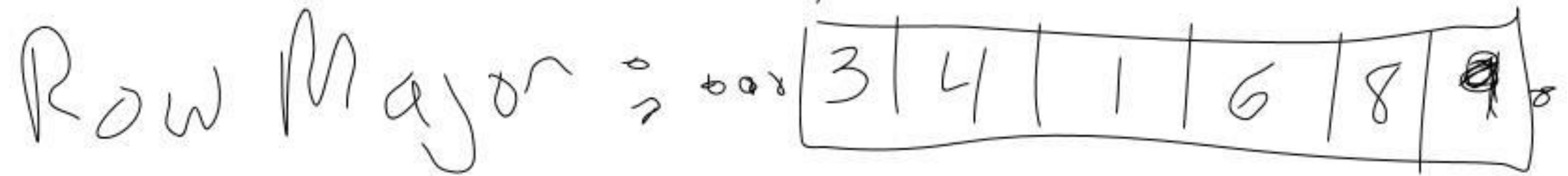
GEORGETOWN UNIVERSITY

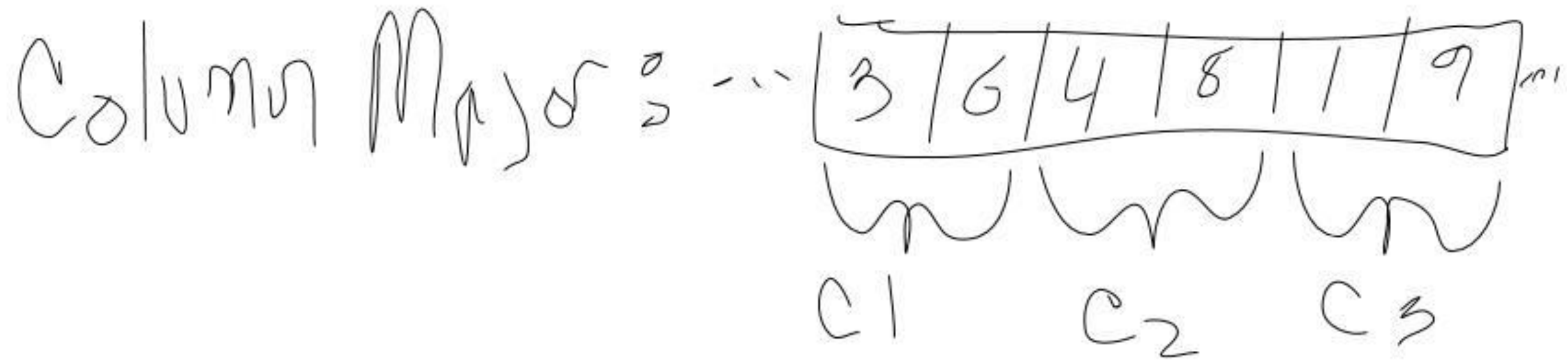
Row Major vs Col. Major

$$\begin{bmatrix} 3 & 4 & 1 \\ 6 & 8 & 9 \end{bmatrix}$$

Memory

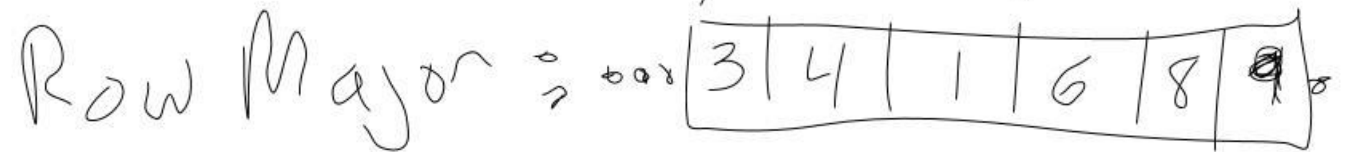
r1 r2

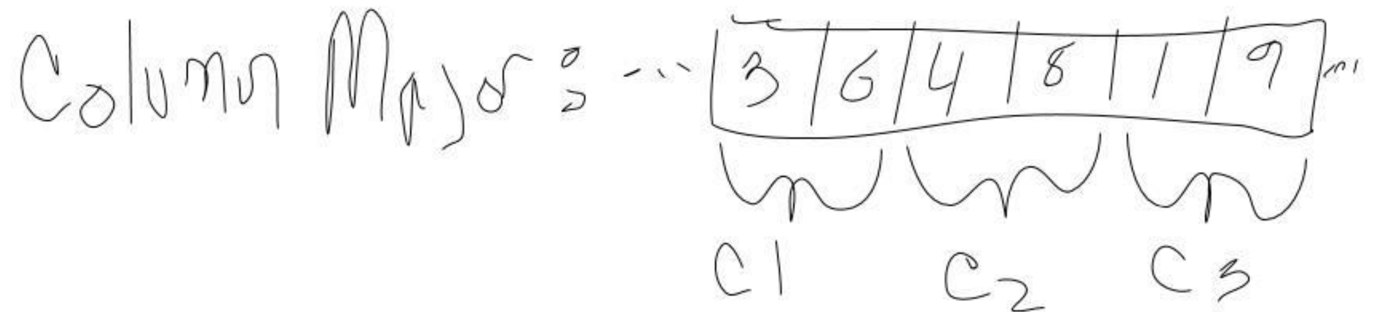




Row Major vs Col. Major

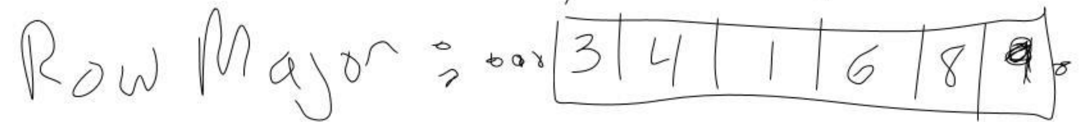
- Traversals may be slowed if multi-dimensional arrays are traversed poorly.
  - Delays can be caused by memory paging delays, caching delays, and potentially indexing arithmetic.



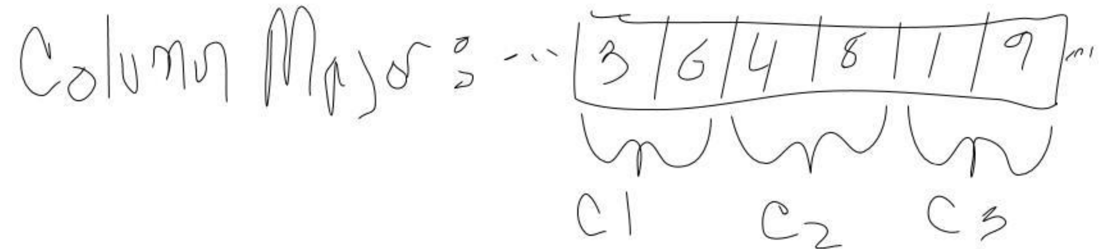


Code Example: Row Major vs Col. Major

```
// Efficient Traversal for Row -Major

for (int i = 0;i< numRows; i++)
        for (int j = 0; j < numCols; j++)
                c[i][j] = i + j;
```




```
// Efficient Traversal for Col-Major

for (int j = 0;j< numCols; j++)
        for (int i = 0; i < numRows; i++)
                c[i][j] = i + j;
```

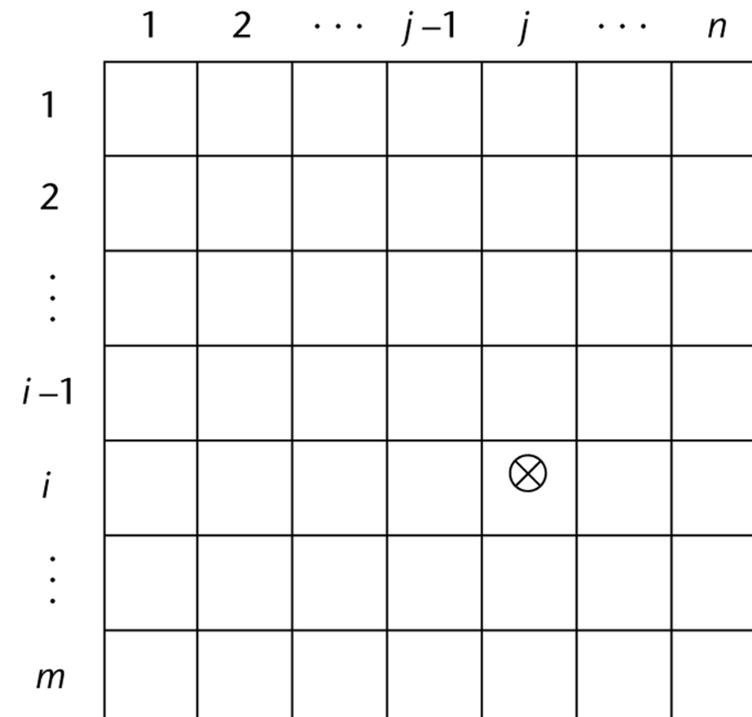


Locating an Element in a Multi-dimensioned Array

- General format (Assuming Row Major storage)
  - Example c++ (assumes indexing starts at 0, though image shows 1)

    array[i][j]

    * (array + i *numCols+ j)



GEORGETOWN UNIVERSITY

Compile-Time Descriptors

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

Single-dimensioned array

Multidimensioned array
Element type
Index type
Number of dimensions
Index range 1
⋮
Index range *n*
Address

Multidimensional array

GEORGETOWN UNIVERSITY

Record Types

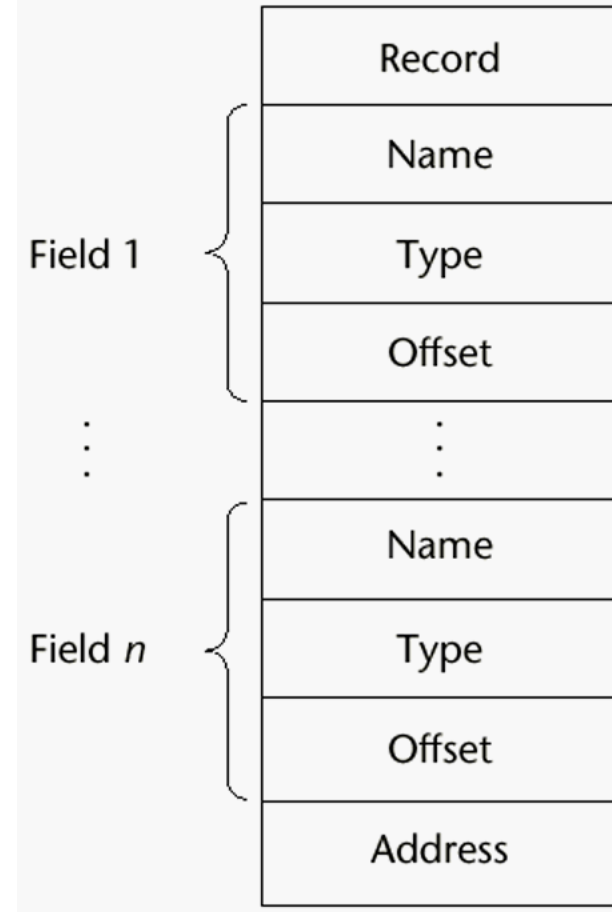
- A *record* is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names
- Design issues:
  - What is the syntactic form of references to the field?
  - Are elliptical references allowed

Evaluation and Comparison 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



List Types

- 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(12)` creates `[0, 1, 2, 3, 4, 5, 6]`

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

Unions Types

- A *union* is a type whose variables are allowed to store different type values at different times during execution
- Design issue
  - Should type checking be required?
  - How is allocation size determined?

Discriminated vs. Free Unions

- 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 ML, Haskell, and F#

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

Pointer and Reference Types

- A *pointer* type variable has a range of values that consists of memory addresses and a special value, *nil*
- Provide the power of indirect addressing
- Provide a way to manage dynamic memory
- A pointer can be used to access a location in the area where storage is dynamically created (usually called a *heap*)

Design Issues of Pointers

- What are the scope of and lifetime of a pointer variable?
- What is the lifetime of a heap-dynamic variable?
- Are pointers restricted as to the type of value to which they can point?
- Are pointers used for dynamic storage management, indirect addressing, or both?
- Should the language support pointer types, reference types, or both?

Pointer Operations

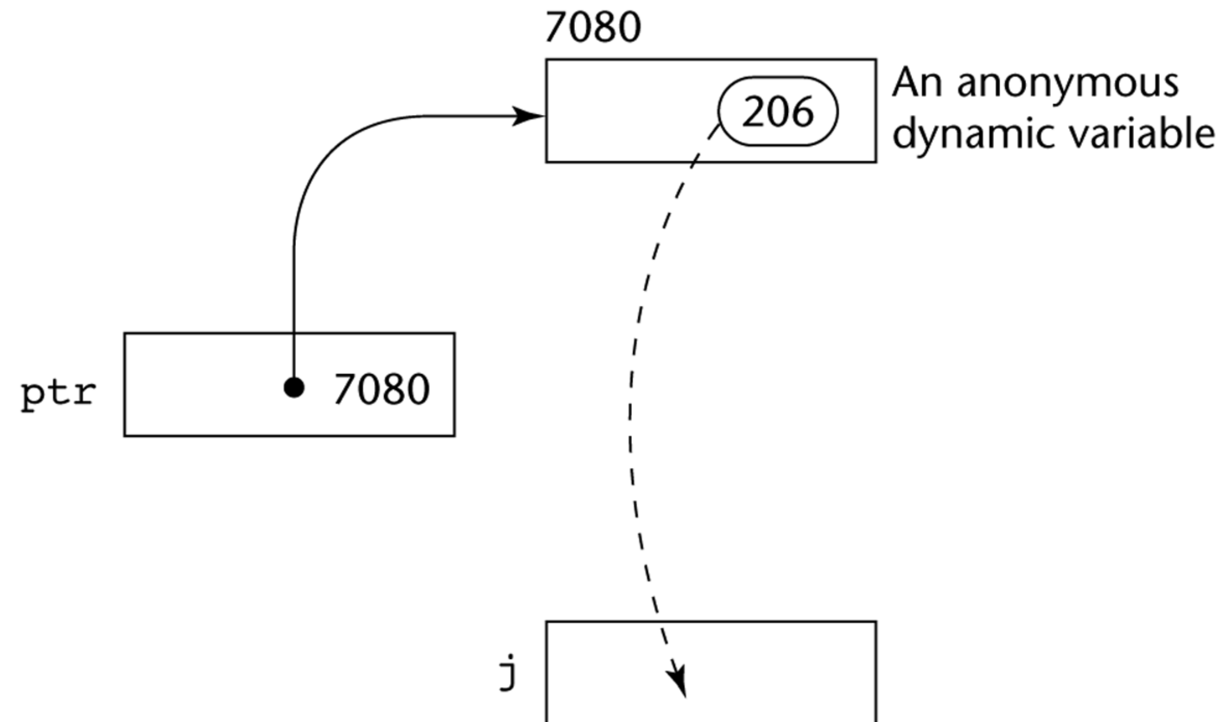
- Two fundamental operations: assignment and dereferencing
- Assignment is used to set a pointer variable's value to some useful address
- Dereferencing yields the value stored at the location represented by the pointer's value
  - Dereferencing can be explicit or implicit
  - C++ uses an explicit operation via *

    `j = *ptr`

    sets j to the value located at `ptr`

GEORGETOWN UNIVERSITY

Pointer Assignment Illustrated



The assignment operation j = *ptr

Pointer Arithmetic in C and C++

```
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]`

Problems with Pointers

- Dangling pointers (dangerous)
  - A pointer points to a heap-dynamic variable that has been deallocated
- Lost heap-dynamic variable (memory leak)
  - An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage*)
    - Pointer `p1` is set to point to a newly created heap-dynamic variable
    - Pointer `p1` is later set to point to another newly created heap-dynamic variable
    - The process of losing heap-dynamic variables is called *memory leakage*

Dangling Pointer "Solutions"

- *Tombstone*: extra heap cell that is a pointer to the heap-dynamic variable
  - The actual pointer variable points only at tombstones
  - When heap-dynamic variable de-allocated, tombstone remains but set to nil
  - Costly in time and space

. *Locks-and-keys*: Pointer values are represented as (key, address) pairs
  - Heap-dynamic variables are represented as variable plus cell for integer lock value
  - When heap-dynamic variable allocated, lock value is created and placed in lock cell and key cell of pointer

Heap Management (garbage collection)

- Strategies for memory leak
- A very complex run-time process
- Two approaches to reclaim garbage
  - Reference counters (*eager approach*): reclamation is gradual
  - Mark-sweep (*lazy approach*): reclamation occurs when the list of variable space becomes empty

Reference Counter

- Reference counters: maintain a counter in every cell that store the number of pointers currently pointing at the cell
  - *Disadvantages*: space required, execution time required,
  - *Advantage*: it is intrinsically incremental, so significant delays in the application execution are avoided

GEORGETOWN UNIVERSITY

Mark-Sweep

- The run-time system allocates storage cells as requested and disconnects pointers from cells as necessary; mark-sweep then begins
  - Every heap cell has an extra bit used by collection algorithm
  - All cells initially set to garbage
  - All pointers traced into heap, and reachable cells marked as not garbage
  - All garbage cells returned to list of available cells
  - Disadvantages: in its original form, it was done too infrequently. When done, it caused significant delays in application execution. Contemporary mark-sweep algorithms avoid this by doing it more often—called incremental mark-sweep

Type Checking (Segue to Expr Eval)

- *Type checking* is the activity of ensuring that the operands of an operator are of compatible types
  - Generalize the concept of operands and operators to include subprograms and assignments
- 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

Type Checking (continued)

- If all type bindings are static, nearly all type checking can be static
- If type bindings are dynamic, type checking must be dynamic
- A programming language is *strongly typed* if type errors are always detected
- Advantage of strong typing: allows the detection of the misuses of variables that result in type errors

Strong Typing

- Language examples:
  - C and C++ are not: parameter type checking can be avoided; unions are not type checked
  - Java and C# are, almost (because of explicit type casting)
- Coercion rules strongly affect strong typing--they can weaken it considerably (C++)

GEORGETOWN UNIVERSITY