Programming Languages
Lecture10 – Types

남 범 석

bnam@skku.edu

Data Types

- Intuitive notion of what types are:
  - Denotational point of view
    - A set of values from a "domain"
  - Constructive point of view
    - Either a small collection of built-in types
      (integer, character, boolean, real, etc.) or
    - A composite type created by constructor
      (record, array, set, etc.)
  - Abstraction-based point of view
    - Collection of well-defined operations that can be applied to objects of that type

What Are Types Good For?

- Provide implicit context
  - Make sure that certain meaningless operations do not occur
    - a + b : use integer addition, if types of both are integer
- Limit the set of operations
  - Prevent programmers from using semantically invalid operations (e.g., add a character to a record)
  - Type checking cannot prevent all meaningless operations
    - It catches enough of them to be useful

Type System

- Type system consists of
  - A mechanism to define types and their language constructs
  - A set of rules for type equivalence, compatibility, inference

- Notions in type system
  - Type equivalence
    - When are the types of two values the same?
  - Type compatibility
    - When can a value of type A be used in a context that expects type B?
  - Type inference
    - What is the type of an expression, given the types of the operands?

Type Checking – Strong vs. Weak

- Strong typing
  - Prevents you from applying an inappropriate operation to data
  - Strongly typed languages
    - Ada, Java
- Weak typing
  - Weak typed languages
    - C, C++
    - C89 is more strongly typed than its predecessor dialects, but less strongly typed than Pascal

Type Checking – Static vs. Dynamic

- Static typing
  - Means that all the type checking can be done at compile time
  - Statically typed languages
    - e.g., Ada, Pascal
    - In practice, most type-checking can be done at compile time,
      - But not 100% can be done at compile time – needs dynamic checking (e.g., array index range check)

- Dynamic typing
  - Dynamic type checking
    - Lisp, Smalltalk, and most scripting languages (e.g., Python and Ruby) perform type checking at run time (but strongly typed)
    - Languages with dynamic scoping are generally dynamically typed

Type Conversion

- Static typing expects a specific type for many contexts
  - a = expr --- expr should be the same type of a
  - a + b --- + requires both a and b are integers or reals
  - foo(a, b) --- a and b should be the same types of foo's formal parameters
- Cases in type conversion
  - Structurally equivalent
    - No conversion code is needed
  - Two types have common values (subrange, signed/unsigned)
    - Dynamic check is needed to avoid semantic errors
  - Structurally different
    - Convert may result in loss of precision, overflow
      - Conversions among int, unsigned, float, double
    - Many processors provide conversion instructions

Type Coercion

- When an expression of one type is used in a context where a different type is expected, one normally gets a type error

- But what about

```
int a;   float b, c;
         ...
c = a + b;
```

- If languages allow different types than the expected one, implicit type conversion (called type coercion in this case) should take place

Type Coercion (cont'd)

- Many languages coerce an expression to be of the proper type
  - Coercion can be based just on types of operands, or expected type from surrounding context as well

- Fortran
  - All based on operand type
- C
  - all floats in expressions become doubles
  - short int and char become int in expressions
  - if necessary, precision is removed when assigning to l-value

Records and Variants

- Records (= structures)
  - Usually laid out contiguously
  - Possible holes for alignment reasons

  - Smart compilers may rearrange fields to minimize holes
    - But C compilers promise not to rearrange
  - Kernel programs sometimes assume a particular layout
    - To handle memory-mapped control registers for a device

Records and Variants

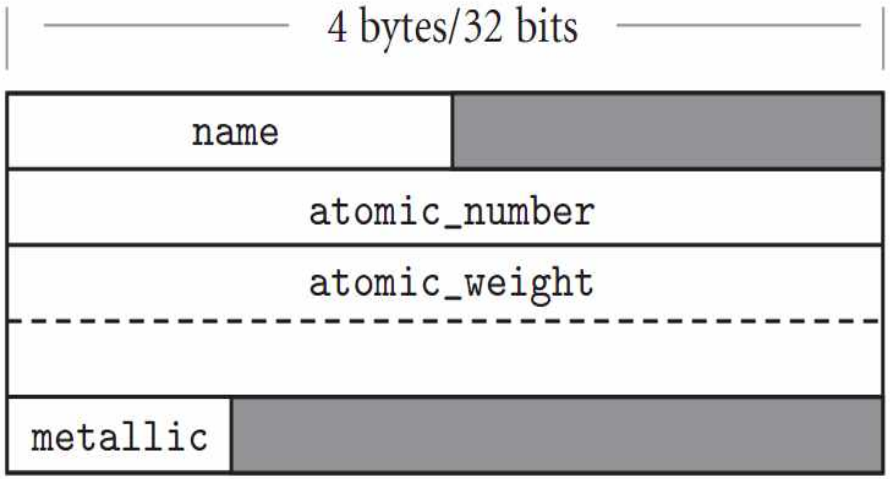
- Variant records (= unions)
  - Overlay space
  - Cause problems for type checking

- Main usage patterns
  - Same bytes interpreted in different ways at different times
  - Alternative sets of fields within a record
    - Common fields + various other fields

Structure Memory Layout

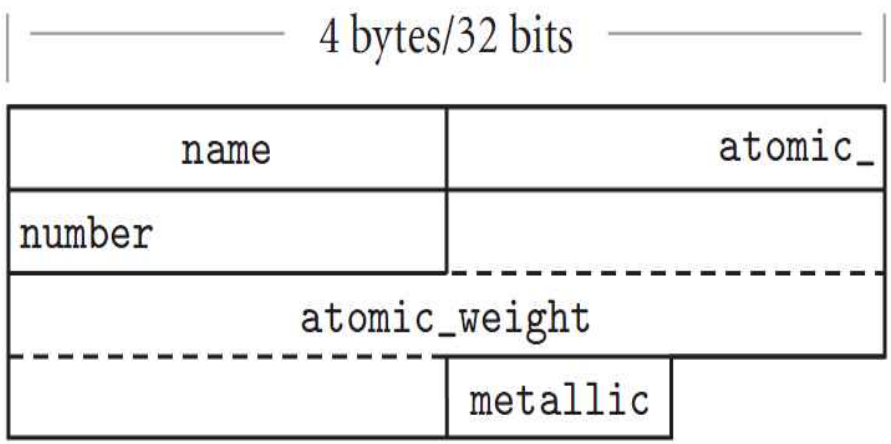
- Holes due to alignment
  - If not aligned, multiple instructions needed to read a field
  - For speed, aligning fields is needed

```
struct Atom {
    short   name;
    int     atomic_number;
    double atomic_weight;
    char    metalic;
}
```

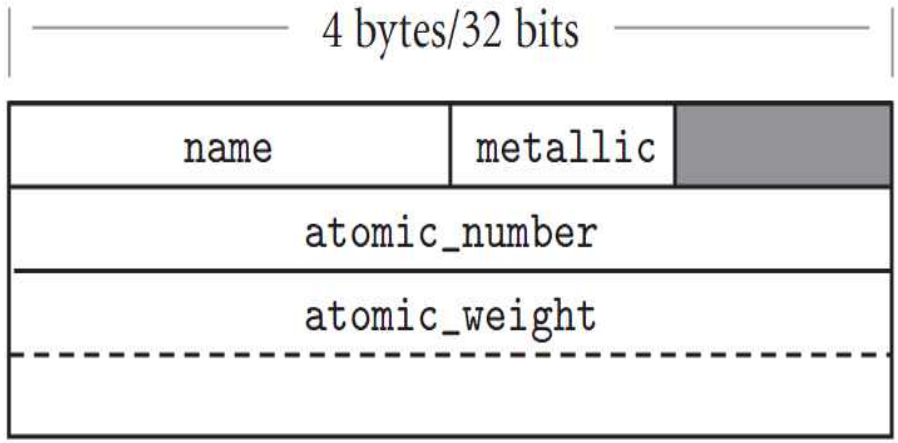


Structure Memory Layout (cont'd)

- Packed record
  - Allows the compiler to optimize for the space
    - May require multiple instructions to read non-aligned field
- Reordered (sorted) fields
  - Can minimize the space due to holes



Packed record



Reordered record

Arrays

- Most common and important composite data types

- Homogeneous elements, unlike records
  - Fortran77 requires element type be scalar
  - Elements can be any type (Fortran90, etc.)

- A mapping from an index type to a component or element type
  - Fortran requires index type be integer
  - Many languages allow index to be any discrete type (integers, Booleans, characters -- countable)

Dope Vectors

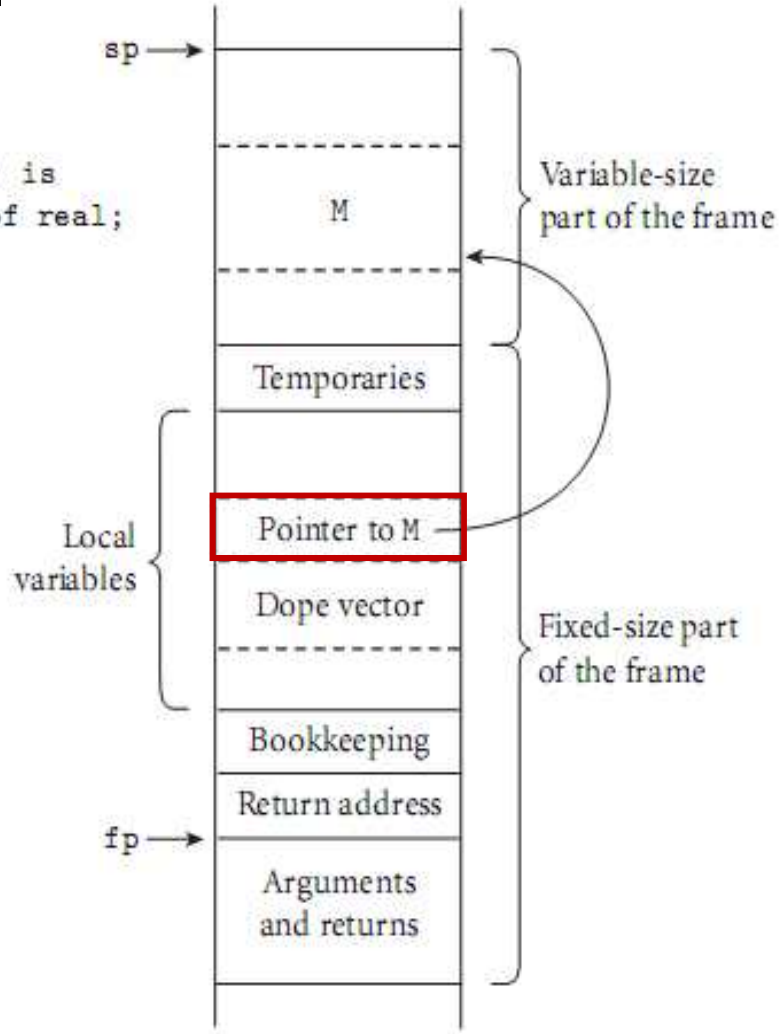
- A dope vector contains
  - Lower bound and size of each dimension
  - Upper bound (redundant) but useful to avoid computation in dynamic bound check
- If dimensions and their sizes of an array are static
  - The compiler can lookup symbol table and generate code to calculated the addresses (no need of dope vectors)
- If dimensions and their sizes are not statically-known
  - These are dynamic shape arrays
  - The compiler generates address calculation code to include the dope vector lookup

Dynamic Shape Arrays in Stack Frame

- Additional indirection is used

```
-- Ada:
procedure foo (size : integer) is
M : array (1..size, 1..size) of real;
...
begin
    ...
end foo;
```

```
// C99:
void foo(int size) {
    double M[size][size];
    ...
}
```



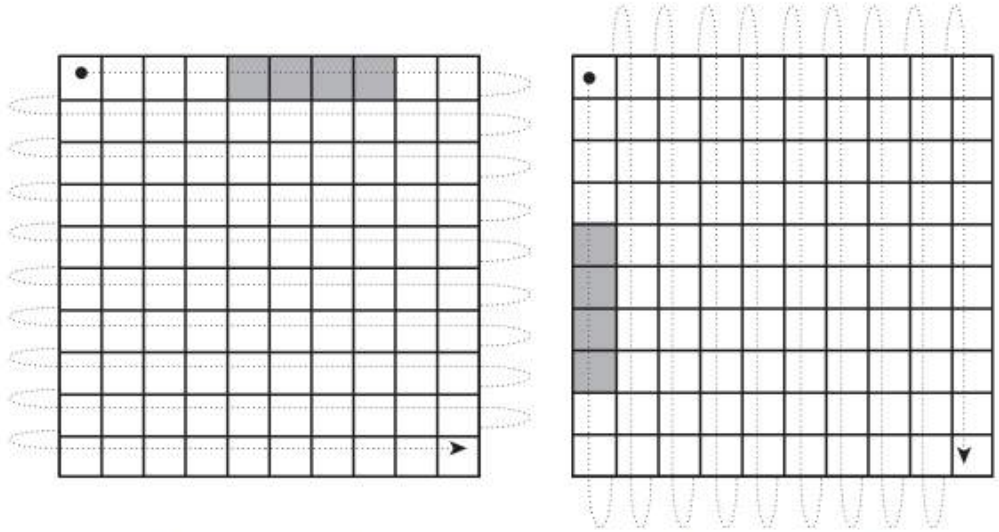
Dynamic Shape Arrays in Heap

- Fully dynamic shape arrays
  - Can change their shapes arbitrary points of a program
  - Need to accommodate these arrays in the heap
  - Examples
    - Variable length strings (Java, C#)
      - String s = “short”; ... s = s + “ but sweet”;
    - Dynamically resizable arrays
      - Vector class, ArrayList class in C++, Java, C# libraries

- Fully dynamic shape arrays with local lifetime
  - Space reclamation code is needed

Memory Layouts of Arrays

- Contiguous elements
- Multidimensional arrays
  - Column major -- only in Fortran
  - Row major -- used by everybody else
    - array [a..b, c..d] is the same as array [a..b] of array [c..d]



Row-major order

Column-major order

Memory Layouts of Arrays (cont'd)

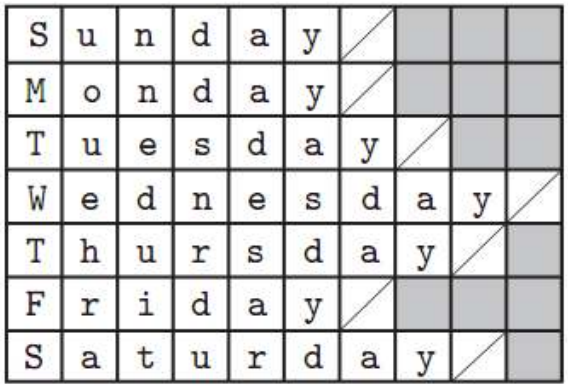
- Row pointers
  - An option in C (with pointers), but all arrays in Java
  - Allows rows to be put anywhere
  - Nice for ragged arrays whose rows are of different lengths
    - e.g., an array of strings
  - Requires extra space for the pointers

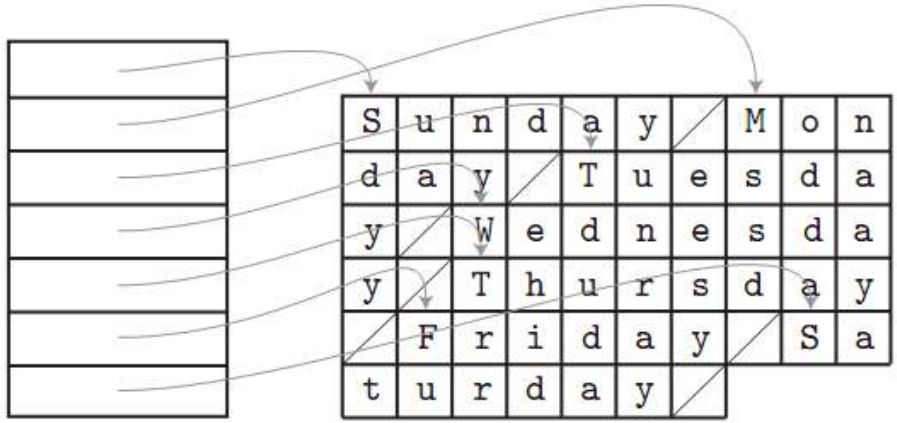
Row-Pointer Layout for Arrays

- Can save space for ragged arrays
- But need an extra pointer for a row

```
char days[][10] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```

```
char *days[] = {
    "Sunday", "Monday", "Tuesday",
    "Wednesday", "Thursday",
    "Friday", "Saturday"
};
...
days[2][3] == 's';  /* in Tuesday */
```





Address Calculations for Arrays

A : array [L1..U1] of array [L2..U2] of array [L3..U3] of elem_type;

D1 = U1-L1+1

D2 = U2-L2+1

D3 = U3-L3+1

Number of composing elements in each dimension

S3 = size of elem_type

S2 = D3 * S3

S1 = D2 * S2

Size of each dimension in bytes

Address of A[i][j][k] = address_of_A
+ (i – L1) * S1 + (j-L2) * S2 + (k-L3) * S3

= (i * S1) + (j * S2) + (k * S3)
+ address_of_A - [(L1 * S1) + (L2 * S2) + (L3 * S3)]

Compile-time constant
for static shape arrays

Strings

- Strings are really just arrays of characters

- Dynamic sizing is often allowed by language designers
  - Variable-length strings are fundamental to many applications
    - C++, Java, C#: string is a built-in class
    - ML, Lisp: string is a chain of blocks (linked list of chars)

Pointers And Recursive Types

- Pointers serve two purposes:
  - Efficient access to elaborated objects (as in C)
  - Dynamic creation of linked data structures – recursive type
    - (in conjunction with a heap storage manager)

- Several languages (e.g. Pascal) restrict pointers only to access objects in the heap

Pointers and Arrays in C

- Types are compatible

  int *a == int a[ ]

  int **a == int *a[ ]

- BUT equivalences don't always hold
  - Specifically, a declaration allocates an array if it specifies a size for the first dimension
  - Otherwise it allocates a pointer

```
int  **a,  int *a[ ]        // pointer to pointer to int
int  *a[n]                  // n-element array of row pointers
int  a[n][m]                // 2-d array
```

Pointers and Arrays in C

- Compiler must know the size of what you point to
  - So the followings are not valid:

```
int a[ ][ ]          // bad
int (*a)[ ]          // bad
```

  - C declaration rule:
    - ( ), [ ] – highest precedence, left-to-right associativity
    - * – lower precedence, right-to-left associativity

```
int *a[n]        // n-element array of pointers to integer
int (*a)[n]      // a pointer to n-element array of integers
```

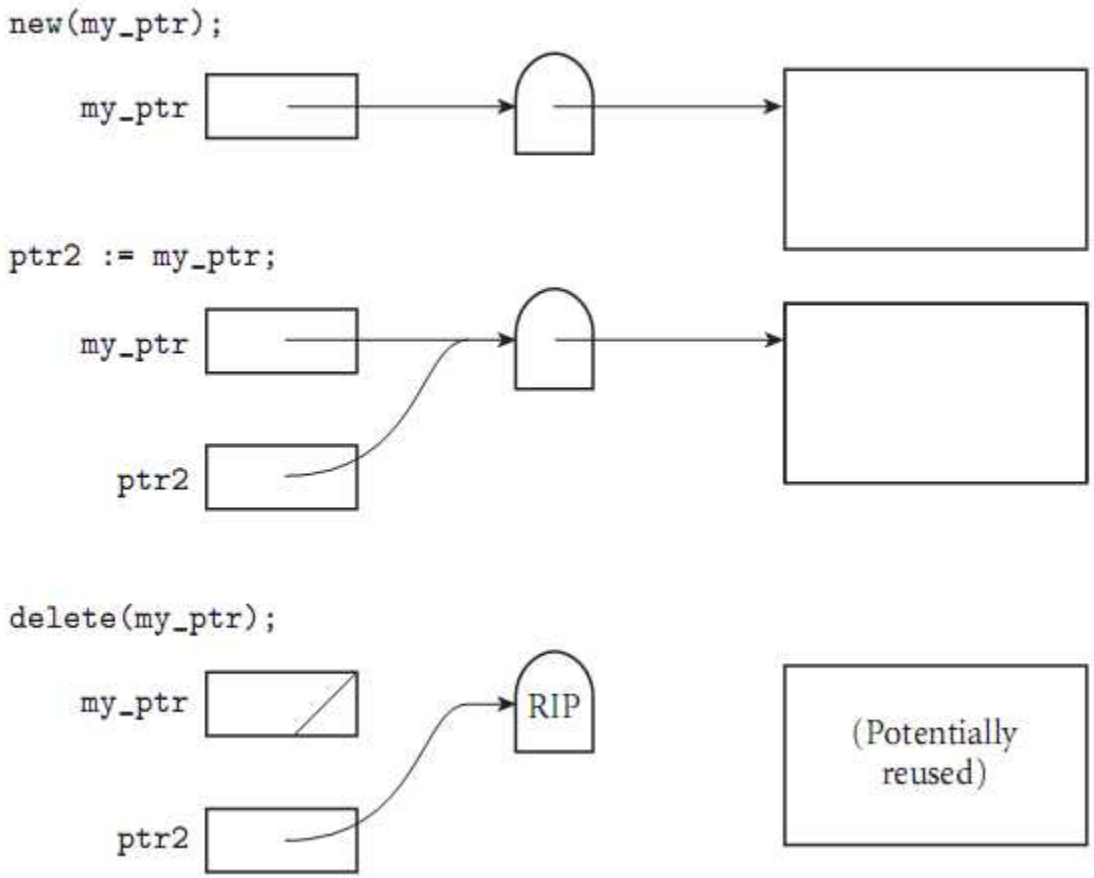
Dangling References

- Dangling reference is
  - A live pointer that no longer points to a valid object

- Dangling pointer problems are often due to
  - Explicit deallocation of heap objects
    - Only in languages that have explicit deallocation
  - Implicit deallocation of elaborated objects

Tombstones – dangling pointer

- Extra indirection data to mark the validity of objects



Locks and Keys - dangling pointer

- Key – a word added to every pointer
- Lock – a word added to every heap object
- To be a valid pointer, its lock and key should match