Data Types

CSE 307 – Principles of Programming Languages

Stony Brook University

http://www.cs.stonybrook.edu/~cse307

Data Types

- We all have developed an intuitive notion of what types are; what's behind the intuition?
  - collection of values from a "*domain*" (the denotational approach)
  - internal *structure* of data, described down to the level of a small set of fundamental types (the structural approach)
  - *equivalence* class of objects (the implementer's approach)
  - collection of well-defined *operations* that can be applied to objects of that type (the abstraction approach)

Type Systems

- Computers are naturally untyped: binary
- Encoding by a type is necessary to store data:
  - as integer: -1, -396, 2, 51, 539
  - as float: -3.168, 384.0, 1.234e5
  - as Strings: "SBCS" (ASCII, Unicode UTF-16, etc.)
- We associate types with:
  - Expressions
  - Objects (anything that can have a name)
    - Type checking can also be done with user-defined types:

speed = 100 *miles*/*hour*

time = 2 *hour*

distance = speed * time (*miles*)

distance + 5 *miles* (ok!)

distance + 5 *hours* (bad!)

Type Systems

- What has a type?
  - things that have values:
    - constants
    - variables
    - fields
    - parameters
    - subroutines
    - objects
  - A name (identifier) might have a type, but refer to an object of a different (compatible type):

```
double a = 1;
Person p = new Student("John");
```

Type Systems

- A *type system* consists of:
  - (1) a mechanism to define types and associate them with certain language constructs, and
  - (2) a set of rules for *type equivalence*, *type compatibility*, and *type inference*

Type Systems

- What are types good for?
  - implicit context for operators ("+" is concatenation for Strings vs. integer summation for integers, etc.)
  - *type checking* - make sure that certain meaningless operations do not occur
    - A violation of the compatibility rules is known as a *type clash*
    - Type checking cannot prevent all meaningless operations
      - It catches enough of them to be useful
- Polymorphism results when the compiler finds that it doesn't need to know certain things

Type Systems

- STRONG TYPING means that the language prevents you from applying an operation to data on which it is not appropriate:
  - unlike types cause type errors
- WEAK TYPING: unlike types cause conversions

Type Systems

- STATIC TYPING means that the compiler can do all the checking at compile time:
  - types are computed and checked at compile time
- DYNAMIN TYPING: types wait until runtime

Type Systems

- Java is strongly typed, with a non-trivial mix of things that can be checked statically and things that have to be checked dynamically (for instance, for dynamic binding):

```
String a = 1;                  //compile-time error
int i = 10.0;                  //compile-time error
Student s = (Student)(new Object());// runtime
```

- Python is strong dynamic typed:

```
a = 1;
b = "2";
a + b          run-time error
```

- Perl is weak dynamic typed:

```
$a = 1
$b = "2"
$a + $b                no error.
```

Type Systems

- There is a trade-off here:
  - Strong-static: verbose code (everything is typed), errors at compile time (cheap)
  - Strong-dynamic: less writing, errors at runtime
  - Weak-dynamic: the least code writing, potential errors at runtime, approximations in many cases

Type Systems

- ***Duck typing*** is concerned with establishing the suitability of an object for some purpose
  - JavaScript uses duck dynamic typing

```
var Duck = function() {
  this.quack = function()
{alert('Quaaaaaack!');};
  return this;
};
var inTheForest = function(object) {
  object.quack();
};
var donald = new Duck();
inTheForest(donald);
```

Type Systems

- ORTHOGONALITY:
  - A collection of features is ***orthogonal*** in a programming language if there are <u>no restrictions on the ways in which the features can be combined</u>
  - For example:
    - Prolog is more orthogonal than ML (because it allows arrays of elements of different types, for instance)
      - It also allows input and output parameters in relations (any combination)
  - Orthogonality is nice primarily because it makes a language easy to understand, easy to use, and easy to reason about

What do we mean by type?

- Three main schools of thought:
  - ***Denotational***: a type is a shorthand for a set of values (e.g., the byte domain is: {0, 1, 2, ... 255})
    - Some are simple (set of integers)
    - Some are complex (set of functions from variables to values)
    - Everything in the program is computing values in an appropriate set
  - ***Constructive***: a type is built out of components:
    - **`int, real, string,`**
    - **`record, tuple, map.`**
  - ***Abstraction***: a type is what it does:
    - OO thinking

Type Checking

- A ***type system*** has rules for:
  - *type equivalence*: when are the types of two values the same?
    - ***Structural equivalence***: two types are the same if they consist of the same components
  - *type compatibility*: when can a value of type A be used in a context that expects type B?
    - type compatibility is the one of most concern to programmers
  - *type inference*: what is the type of an expression, given the types of the operands?

```
a : int        b : int
----------------------
      a + b : int
```

Types and Equality Testing

- What should **`a == b`** do?
  - Are they the same object?
  - Bitwise-identical?
- Languages can have different equality operators:
  - Ex. Java's **`==`** vs **`equals`**

Type Casts

- Two casts: converting and non-converting
  - ***Converting cast***: changes the meaning of the object type in question
    - cast of **`double`** to **`int`** in Java
  - ***Non-converting casts***: means to **interpret** the bits as the same type

    `Person p = new Student();` // implicit non-converting

    `Student s = (Student)p;` // explicit non-converting cast
- *Type coercion*: May need to perform a runtime semantic check
  - Example: Java references:

```
Object o = "...";
String s = (String) o;
```

    // maybe after **`if(o instanceOf String)`**…

Type Checking

- ***Structural equivalence***: most languages agree that the **format of a declaration should not matter**:

```
struct { int b, a; }
```

is the same as the type:

```
struct {
        int a;
        int b;
}
```

  - To determine if two types are structurally equivalent, a compiler can expand their definitions by replacing any embedded type names with their respective definitions, recursively, until nothing is left but a long string of type constructors, field names, and built-in types.

Type Checking

- ***Name equivalence*:**

```
TYPE new_type = old_type;
```

new_type is said to be an alias for **old_type**.

- aliases to the same type

```
TYPE human = person;
```

A language in which aliased types are considered equivalent is said to have ***loose name equivalence***

- there are times when aliased types should probably **Not** be the same:

```
TYPE celsius_temp = REAL,
     fahrenheit_temp = REAL;
VAR  c : celsius_temp,
     f : fahrenheit_temp;
f := c; (* this should probably be an error *)
```

A language in which aliased types are considered distinct is said to have ***strict name equivalence***

Type Checking

- Parametrized generic data structures:

```
TYPE stack_element = INTEGER; (* alias *)

MODULE stack;
IMPORT stack_element;
EXPORT push, pop;
...
PROCEDURE push(elem : stack_element);
...
PROCEDURE pop() : stack_element;
...
```

Classification of Types

- Types can be discrete (countable/finite in implementation):
  - **`boolean`**:
    - in C, 0 or not 0
  - **integer types:**
    - different precisions (or even multiple precision)
    - different signedness
    - Why do we define required precision? Leave it up to implementer
  - **floating point numbers:**
    - only numbers with denominators that are a power of 10 can be represented precisely
  - **decimal types:**
    - allow precise representation of decimals
    - useful for money: Visual Studio .NET:

```
decimal myMoney = 300.5m;
```

Classification of Types

- **character**
  - often another way of designating an 8 or 16 or 32 bit integer
  - Ascii, Unicode (UTF-16, UTF-8), BIG-5, Shift-JIS, latin-1
- **subrange numbers**
  - Subset of a type (`for i in range(1:10)`)
  - Constraint logic programming: `X in 1..100`
- **rational types:**
  - represent ratios precisely
- **complex numbers**

Classification of Types

- Types can be composite :
  - **records (unions)**
  - **arrays**
    - **Strings** (most languages represent Strings like arrays)
      - list of characters: null-terminated
      - With length + get characters
  - **sets**
  - **pointers**
  - **lists**
  - **files**
  - **functions, classes**, etc.

Record Types

- A record consists of a number of fields:
  - Each has its own type:

```
struct MyStruct {
    boolean ok;
    int bar;
};
MyStruct foo;
```

- There is a way to access the field:

  `foo.bar;` <- C, C++, Java style.

  `bar of foo` <- Cobol/Algol style

  `person.name` <- F-logic *path expressions*

Record Types

- When a language has value semantics, it's possible to assign the entire record in one ***path expression***:

  ```
  a.b.c.d.e = 1; // even used in query languages like XPath
  ```

- With statement: accessing a deeply nested field can take a while. Some languages (JS) allow a with statement:

  ```
  with a.b.c.d {
      e = 1;
      f = 2;
  }
  ```

- Variant records (a and b take up the same memory, saves memory, but usually unsafe, tagging can make safe again):

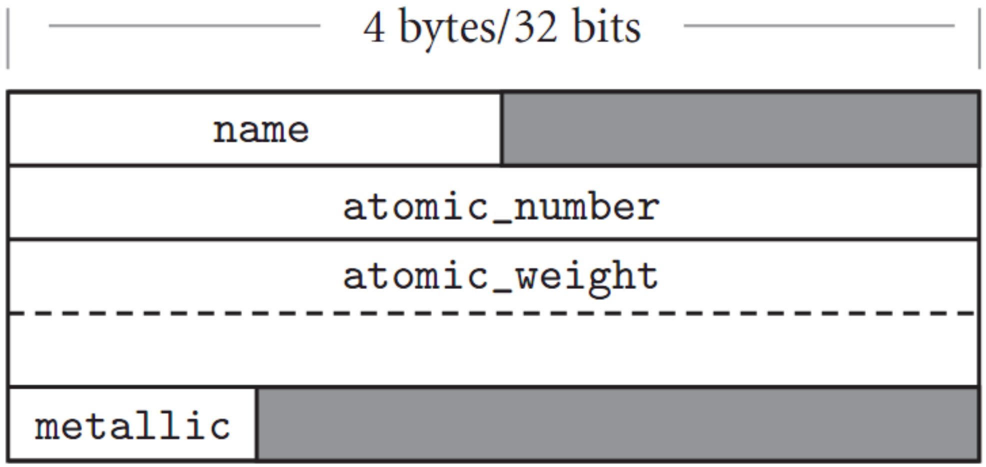
  ```
  union {
      int a;
      float b;
  }
  ```

Record Types

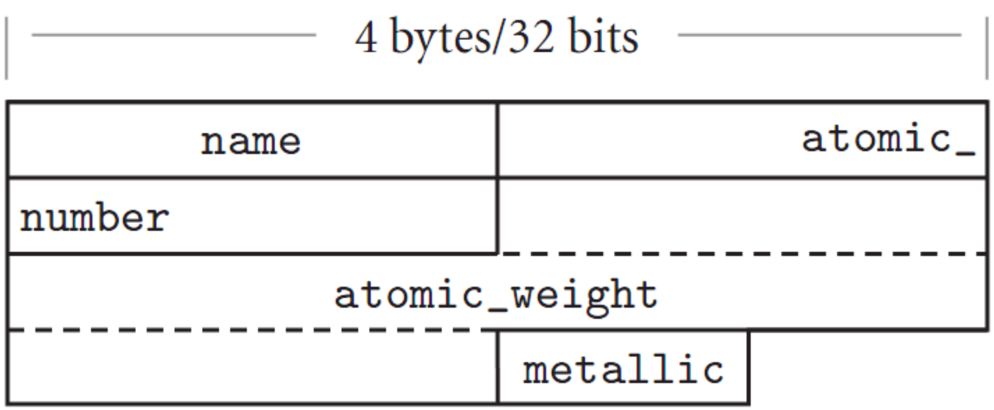
- ***Records***: multiple properties in one complex type
  - **usually laid out contiguously**
  - possible holes for alignment reasons
  - smart compilers may rearrange fields to minimize holes (C compilers promise not to)
    - See next slide for an example of *memory layout*
  - implementation problems are caused by records containing dynamic arrays

Record Types

- Memory layout

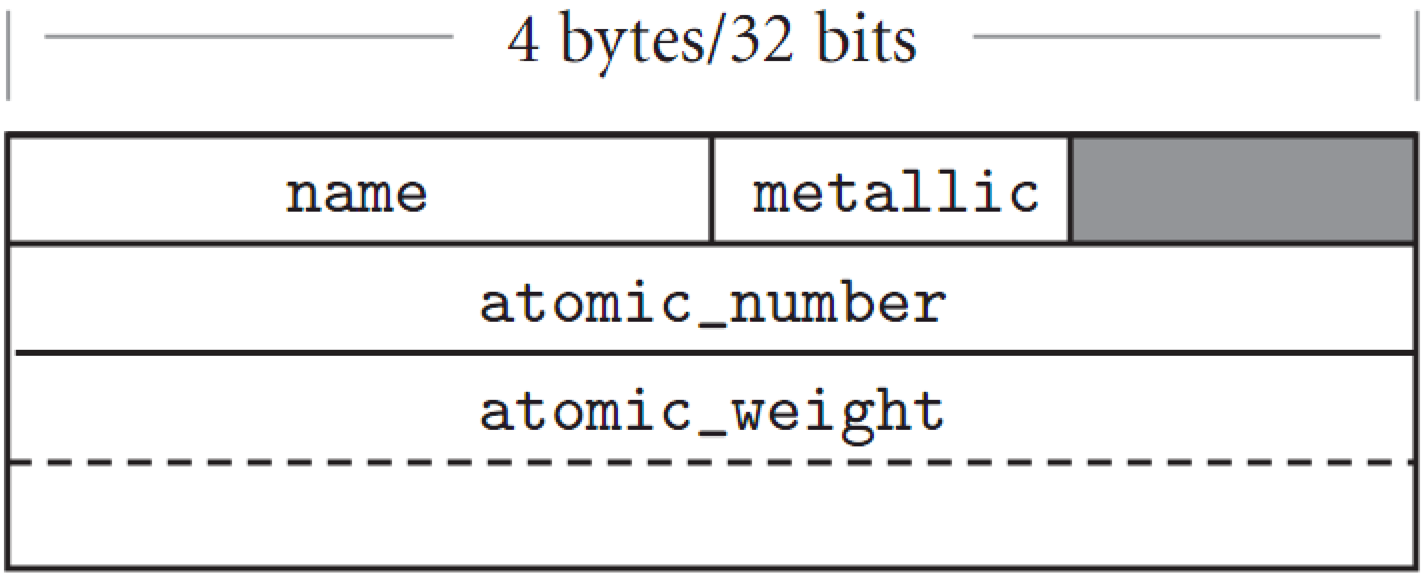


- memory layout for **packed** element records



Record Types

- Rearranging record fields to minimize holes and keep fields optimally addressable:



Record Types

- ***Unions*** (a.k.a., ***variant records***):
  - **overlay space**
  - cause problems for type checking
  - Lack of tags means you don't know what is there
    - Ability to change tag and then access fields hardly better: can make fields "uninitialized" when tag is changed (requires extensive run-time support) –
  - Memory layout for unions example:

Arrays

- Arrays are the most common and important composite data types
- Unlike records, which group related fields of disparate types, arrays are usually homogeneous
- Semantically, they can be thought of as a mapping from an index type to a component or element type

Arrays

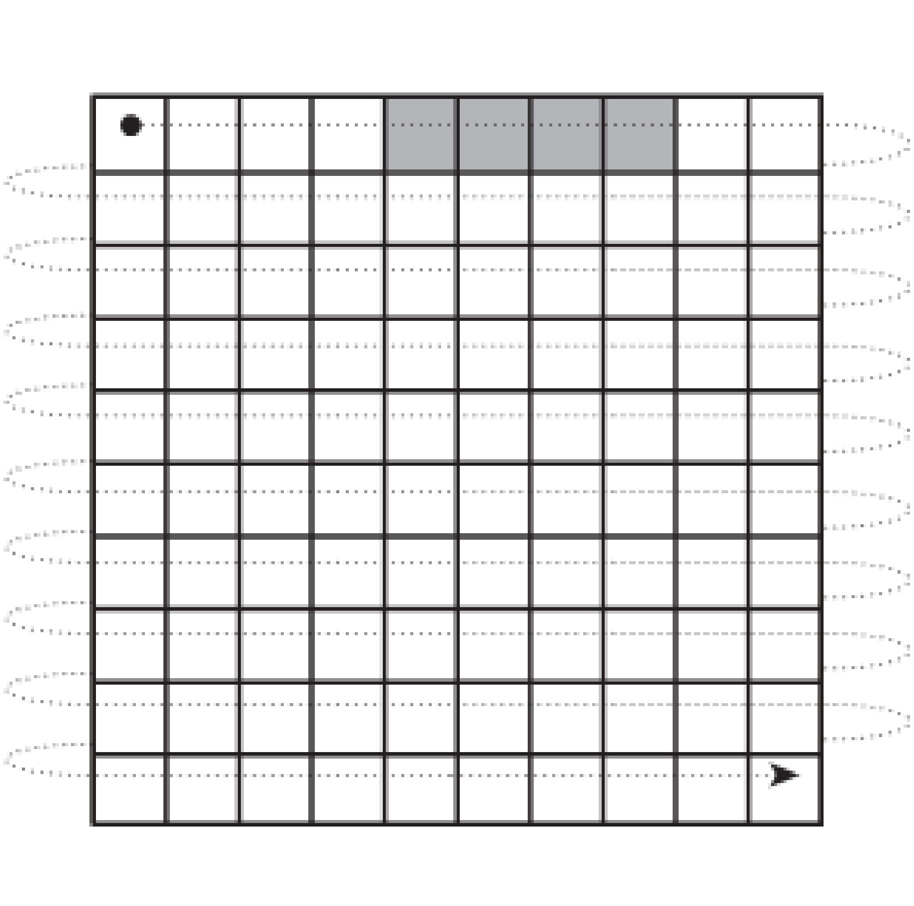
- Arrays = areas of memory of the same type.
  - Stored consecutively.
    - Element access (read & write) = O(1)
  - Possible layouts of memory:
    - Row-major and Column-major:
      - storing multidimensional arrays in linear memory
      - Example: `int A[2][3] = { {1, 2, 3}, {4, 5, 6} };`
        - *Row-major*: A is laid out contiguously in linear memory as: `123456`

          `offset = row * NUMCOLS + column`
        - *Column-major*: A is laid: `142536`

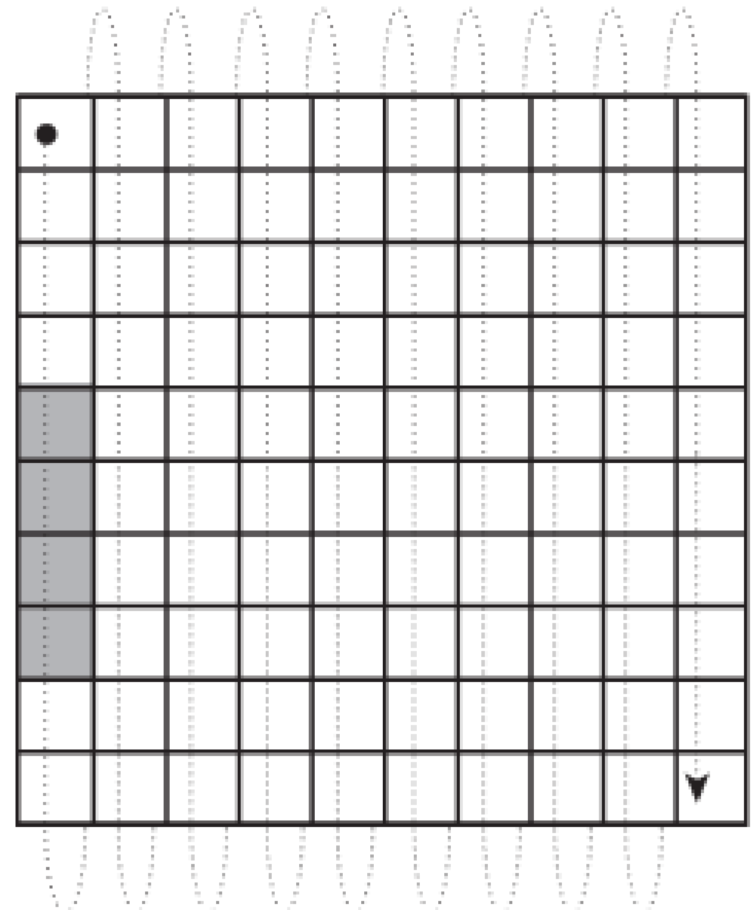
          `offset = row + column*NUMROWS`
      - Row-major order is used in C, PL/I, Python and others.
      - Column-major order is used in Fortran, MATLAB, GNU Octave, R, Rasdaman, X10 and Scilab.
    - Row pointers: Java (next)

Arrays

- Row-major and Column-major:



Row-major order

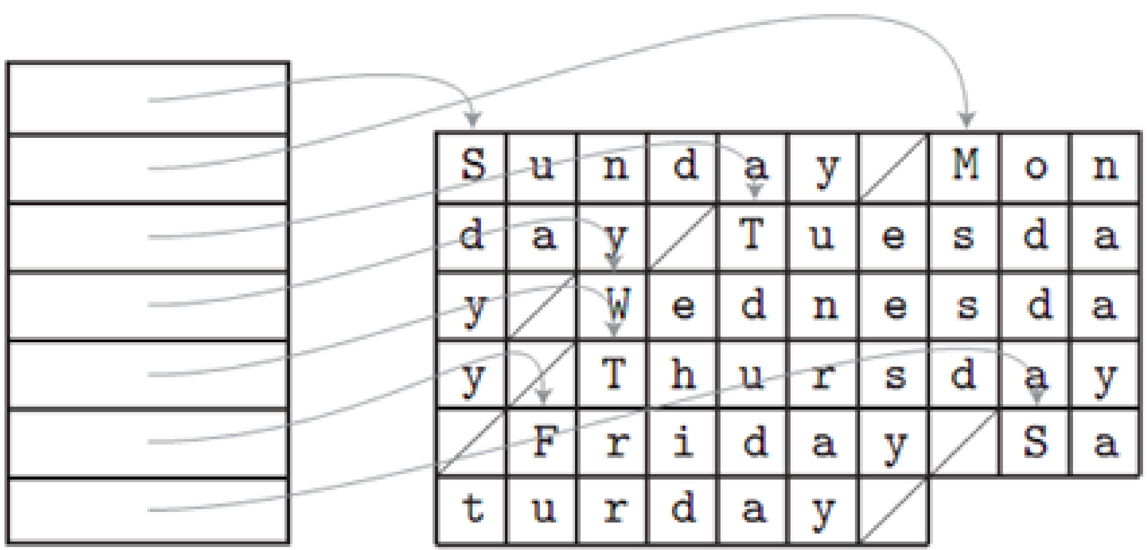


Column-major order

Arrays

- Row pointers:

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



Arrays

- Row pointers:
  - allows rows to be put anywhere - nice for big arrays on machines with segmentation problems
  - avoids multiplication
  - nice for matrices whose rows are of different lengths
    - e.g. an array of strings
  - requires extra space for the pointers

Arrays

- Allocation of arrays in memory:
  - global lifetime, static shape — If the shape of an array is known at compile time, and if the array can exist throughout the execution of the program, then the compiler can allocate space for the array in **static global memory**
  - local lifetime, static shape — If the shape of the array is known at compile time, but the array should not exist throughout the execution of the program, then space can be allocated in the subroutine's stack frame at run time (**stack allocation**)
  - local lifetime, shape bound at elaboration time (**heap allocation**)

Arrays

- Row-major generalizes to higher dimensions, so a 2×3×4 array looks like:

int A[2][3][4] = {{{1,2,3,4}, {5,6,7,8}, {9,10,11,12}}, {{13,14,15,16}, {17,18,19,20}, {21,22,23,24}}};

is laid out in linear memory as: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

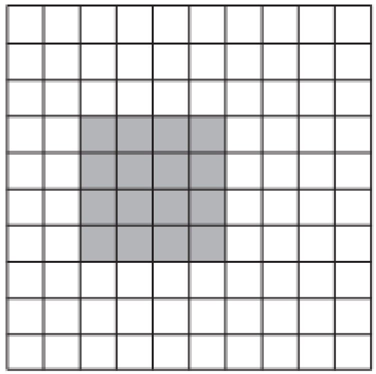
- Efficiency issues due to caching
  - Can effect behavior of algorithms
- **Row/Column major require dimension to be part of the type.**

Arrays

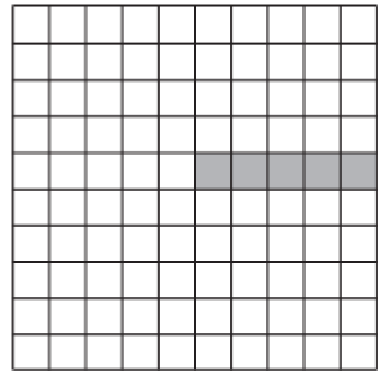
- *Indexing* is a special operator, since it can be used as an l-value
  - In languages that let you overload operators, often need two variants:
    - __getindex__ and __setindex__

Arrays Operations

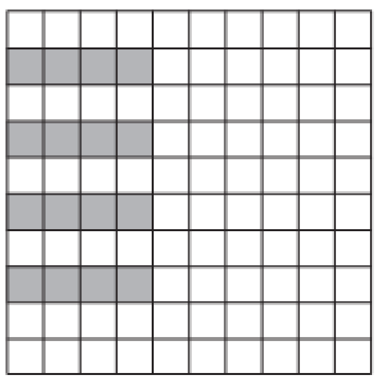
- Array slices (sections) in Fortran90



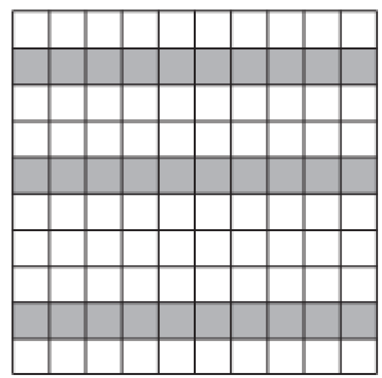
`matrix(3:6, 4:7)`



`matrix(6:, 5)`



`matrix(:4, 2:8:2)`



`matrix(:, (/2, 5, 9/))`

Can you do them in python? Yes.

Arrays Operations

- Array slices in python:
  - **`a[start:end]`** # items start through **`end-1`**
    - The **`end`** value represents the first value that is not in the selected slice
  - **`a[start:]`** # items start through the rest of the array
  - **`a[:end]`** # items from the beginning through end-1
  - **`a[:]`** # a copy of the whole array
- There is also the step value, which can be used with any of the above:
  - **`a[start:end:step]`** # start through not past end, by step
- The start or end may be a negative number, which means it counts from the end of the array instead of the beginning
  - **`a[-1]`** # last item in the array
  - **`a[-2:]`** # last two items in the array
  - **`a[:-2]`** # everything except the last two items

Strings

- In some languages, strings are really just arrays of characters
- In others, they are often special-cased, to give them flexibility (like polymorphism or dynamic sizing) that is not available for arrays in general
  - It's easier to provide these things for strings than for arrays in general because strings are one-dimensional and (more importantly) non-circular

Sets

- Set: contains **distinct** elements without order.
  - Pascal supported sets of any discrete type, and provides union, intersection, and difference operations:

```
var A, B, C : set of char;
D, E : set of weekday;
...
A := B + C;
(* union; A := {x | x is in B or x is in C} *)
A := B * C;
(* intersection; A := {x | x is in B and x is in C} *)
A := B - C;
(* difference; A := {x | x is in B and x is not in C}*)
```

Sets

- Ways to implement sets:
  - ***Hash Maps*** (keys without values or the value is the same with the key)
  - When we know # of values, can assign each value a bit in a ***bit vector***
    - Things like intersection, union, membership, etc. can be implemented efficiently with bitwise logical instructions
- ***Bag***s: allows the same element to be contained inside it multiple times.

Maps/Dictionaries

- Maps keys to values
- Multimap: Maps keys to set of values
- Can be implemented in the same way as sets.

Lists

- Prolog-style ***linked lists*** (same with SML) vs. Python-style Array lists:
  - Prolog: matching against lists
    - **`Head`**
    - **`Tail`**

    Can match more complex patterns than SML:

    `[a,1,X|T].`

Representation of Lists in Prolog

- List is handled as binary tree in Prolog

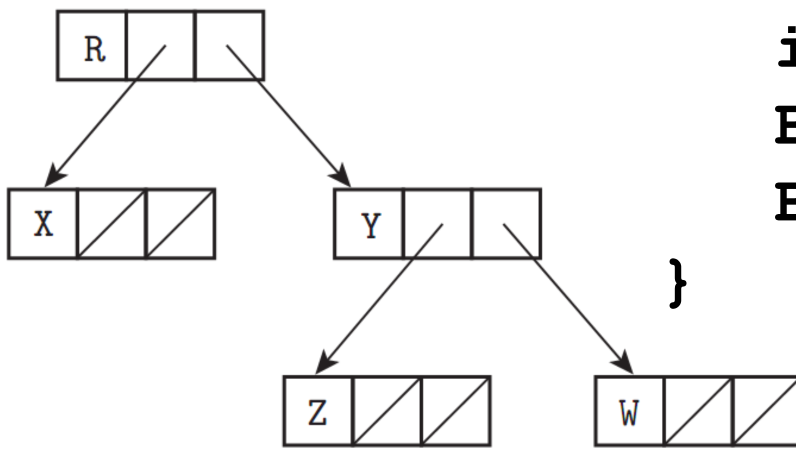
```
[Head | Tail] = .(Head,Tail)
```

  - Where **`Head`** is any number of atoms and **`Tail`** is a list
  - Example:

```
[a,b,c] = .(a,.(b,.(c,[])))
= [a|[b,c]] = [a|[b|[c]]] =
= [a|[b|[c|[]]]] =
= [a,b|[c|nil]]] =
= [a|.(b,.(c,[]))]
```

Pointers/Reference Types

- ***Pointers*** serve two purposes:
  - dynamic creation of linked data structures, in conjunction with a heap storage manager
  - efficient (and sometimes intuitive) access to elaborated objects (as in C)
    - Recursive types – like trees:




```
class BinTree {
    int value;
    BinTree left;
    BinTree right;
}
```

- Several languages (e.g. Pascal, Ada 83) restrict pointers to accessing things in the heap

Pointers/Reference Types

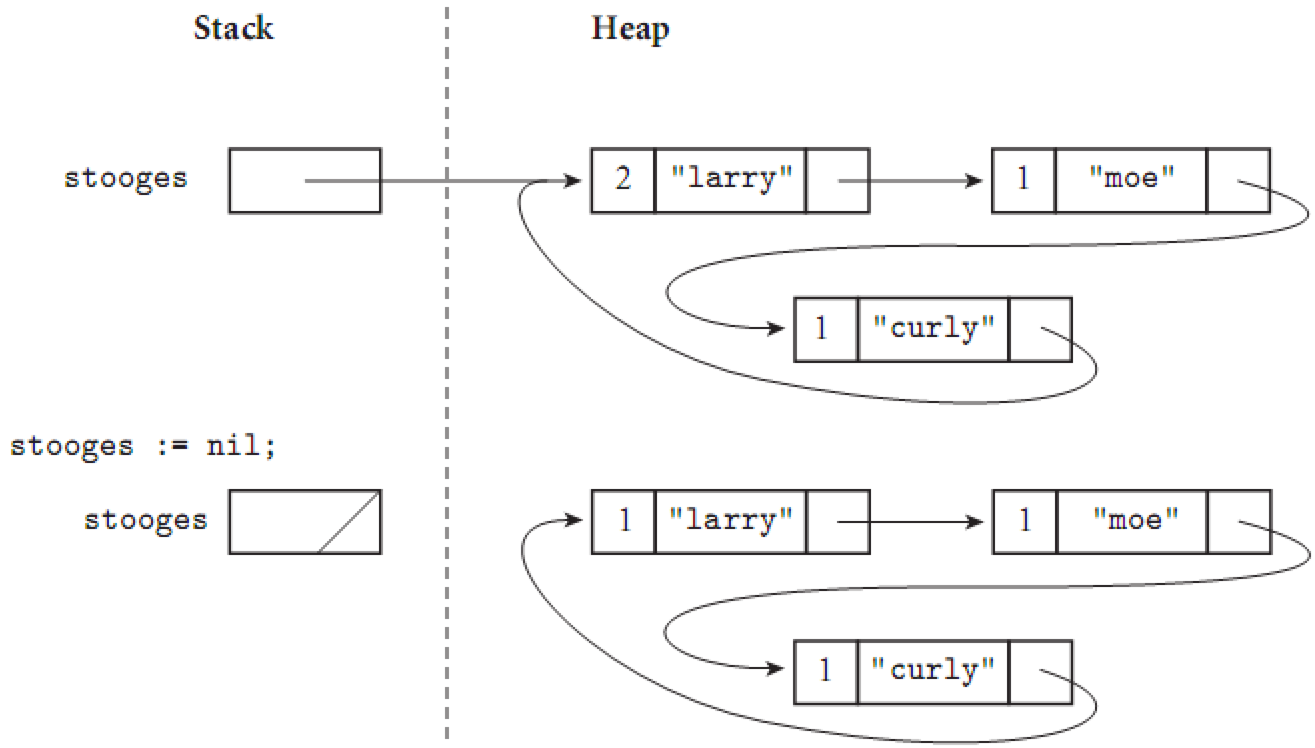
- Pointers tend to allow pointer arithmetic: **`foo += 1`**
  - Only useful when in an array
    - Leave the bounds of your array, and you can have security holes
  - Problem: Can point to something that isn't a **`BinTree`**, or even out of memory.
- In Java, references are assigned an object, and don't allow pointer arithmetic.
  - Can be **`NULL`**.

Pointers/Reference Types

- Problems with garbage collection
  - many languages leave it up to the programmer to design without garbage creation - this is VERY hard
  - others arrange for automatic garbage collection
    - reference counting
    - does not work for circular structures
    - works great for strings

Pointers/Reference Types

- Garbage collection with ***reference counts***



  - The list shown here cannot be found via any program variable, but because it is circular, every cell contains a nonzero count.

Pointers/Reference Types

- ***Mark-and-sweep***:
  - The collector walks through the heap, tentatively marking every block as "**useless**"
  - Beginning with all pointers outside the heap, the collector recursively explores all linked data structures in the program, marking each newly discovered block as "**useful**"
    - When it encounters a block that is already marked as "**useful**," the collector knows it has reached the block over some previous path, and returns without recursing.
  - The collector again walks through the heap, moving every block that is still marked "**useless**" to the free list.

Files and Input/Output

- Input/output (I/O) facilities allow a program to communicate with the outside world
  - interactive I/O and I/O with files
  - Interactive I/O generally implies communication with human users or physical devices
- Files generally refer to off-line storage implemented by the operating system
  - Files may be further categorized into
    - ***temporary files*** exist for the duration of a single program run
    - ***persistent files*** exist before and after a program runs