



# Types

Chapters 7, 8

# Data Types

- Types provide implicit context for operations
- C: `a + b`
  - integer/floating point addition
- Pascal: `new p`
  - allocate right size
- C: `new my_type()`
  - allocate right size
  - call right constructor

# Data Types

- Boolean
  - true/false; one byte, sometimes one bit
  - C: integers, true = non-0, false = 0
- Character
  - one byte – ASCII
  - two bytes - Unicode
- Numeric
  - integers, reals
  - complex: C, Fortran, Scheme (pair of floats)
  - rational: Scheme (pair of integers)
- Discrete (or ordinal)
  - integers, Booleans, characters
  - countable, well-defined predecessor/successor
- Scalar (or simple): discrete, rational, real, complex

# Data Types

- Enumeration

- introduced in Pascal:

```
type weekday = (sun, mon, tue, wed, thu, fri, sat);
```

- ordered: mon < tue; can index an array

- Subrange: type test\_score = 0..100

- Composite (non-scalar)

Can contains more than one thing

- Records (struct) – collection of fields
  - Arrays – most common; map from index to elements
    - strings = arrays of characters
  - Sets – powerset of base type non-repeat elements
  - Pointers – references to objects; recursive data types
  - Lists – sequence; no map; recursive definition, fundamental in functional programming
  - Files – like arrays but with current position

# Type checking

- *Type equivalence*: two types are the same
- *Type compatibility*: a type can be used in a context
- *Type inference*: deduce the type from components
- *Type clash*: violation of type rules

# Type Systems

- *Strongly typed* language
  - prohibits any application of an operation to an object that is not intended to support that operation
- *Statically typed* language
  - strongly typed
  - at compile time – good performance
  - C, C++, Java
  - C: more strongly typed with each new version
- *Dynamically typed* language
  - at run time - ease of programming
  - Scheme, Lisp, Smalltalk – strongly typed
  - Scripting: Python, Ruby – strongly typed

# Type Checking: Equivalence

- *Structural equivalence*
  - same components put together in the same way
  - C, Algol-68, Modula-3, ML
- *Name equivalence*
  - lexical occurrence
  - each definition is a new type长得一样但名字不一样 所以是new type
  - Java, C#, Pascal

# Type Checking: Equivalence

- Structural equivalence:
  - format should not matter

```
type R1 = record  
    a, b : integer  
end;
```

形式不重要  
只要内容一样

```
type R1 = record  
    a : integer;  
    b : integer;  
end;
```

- What about order? (most languages consider it equivalent)

```
type R3 = record  
    b : integer;  
    a : integer  
end;
```

内容顺序也可改变

# Type Checking: Equivalence

- Structural equivalence: problem

```
type student = record
    name, address : string
    age : integer
type school = record
    name, address : string
    age : integer
x : student;
y : school;
...
x := y; -- is this an error?
```

- compiler says it's okay
- programmer most likely says it's an error

# Type Checking: Equivalence

- Name equivalence
  - Distinct definitions mean distinct types
  - If the programmer takes the time to write two type definitions, then they are different types
  - Aliases

```
type new_type = old_type (* Algol syntax *)
typedef old_type new_type /* C syntax */
```

- Are aliases the same or different types?
- Different: *strict name equivalence*
- Same: *loose name equivalence*

# Type Checking: Equivalence

- Strict name equiv.:
  - `blink` different from `alink`
  - `p, q` – same type; `r, u` – same type
- Loose name equiv.:
  - `blink, alink` – same type
  - `p, q` – same type; `r, s, u` – same type
- Structural equiv.:
  - `p, q, r, s, t, u` – same type

```
type cell = ...           -- whatever
type alink = pointer to cell
type blink = alink        -- alias
p, q : pointer to cell
r : alink
s : blink
t : pointer to cell
u : alink
```

why in loose name equiv and  
strict name equiv `t` is different  
from `p, q`?

# Type Checking: Conversion

- *Type conversion (cast)*: **explicit** conversion

```
r = (float) n;
```

- *Type coercion*: **implicit** conversion

- very useful
- weakens type security
- dynamically typed languages: very important
- statically typed languages: wide variety
  - C: arrays and pointers intermixed
  - C++: programmer-defined coercion to and from existing types to a new type (class)

# Type Checking: Compatibility

- *Type compatibility*

- more important than equivalence
- most of the time we need compatibility
- assignment: RHS compatible with LHS
- operation: operands types compatible with a common type that supports the operation
- subroutine call: arguments types compatible with formal parameters types

# Type Checking: Inference

- *Type inference*
  - infer expression type from components
  - $\text{int} + \text{int} \Rightarrow \text{int}$
  - $\text{float} + \text{float} \Rightarrow \text{float}$
  - subranges cause complications
    - type Atype = 0..20; Btype = 10..20;
    - var a : Atype; b : Btype;
    - What is the type of  $a + b$  ? 10..40

# Type Checking: Inference

- Type inference
  - declarations: type inferred from the context

- C#: var

```
var i = 123;  
// equiv. to:  
int i = 123;
```

```
var map = new Dictionary<string, int>();  
// equiv. to:  
Dictionary<string, int> map = new  
Dictionary<string, int>();
```

# Type Checking: Inference

- C++: `auto`

```
auto reduce = [] (list<int> L, int f(int, int), int s) {  
    for (auto e : L) { s = f(e, s); }  
    return s; 所以最后auto是int  
};  
...  
int sum = reduce(my_list, [](int a, int b){return a+b;}, 0);  
int prod = reduce(my_list, [](int a, int b){return a*b;}, 1);
```

- the `auto` keyword allows to omit the type:

```
int (*reduce) (list<int>, int (*)(int, int), int)  
= [] (list<int> L, int f(int, int), int s) {  
    for (auto e : L) { s = f(e, s); }  
    return s;  
};
```

# Type Checking: Inference

- C++: decltype
  - match the type of an existing expression
  - the type of sum depends on the types of A and B under the coercion rules of C++
  - both `int` gives `int`
  - one is `double` gives `double`

```
template <typename A, typename B>
...
A a;
B b;
decltype(a + b) sum;
```

# Polymorphism

- *Polymorphism* (polymorphous = multiform)
- code works with multiple types
  - must have common characteristics a+b可以有不同的implementation但本质要和原版的一致
- parametric polymorphism: take a type as parameter
  - explicit parametric polymorphism (*generics*, C++: *templates*)  
appears in statically typed languages
  - implemented at compile time
- subtype polymorphism: code works with subtypes
  - object-oriented languages subclass is a subtype
- combination (subtype + parametric polymorphism)
  - container classes
  - List<T>, Stack<T>; T instantiated later

assume List and Stack are subclass from some more generic class  
As we can see, we can use it combine with T → parametric polymorphism

# Polymorphism

- *Implicit:*

- Scheme:

(define SOME\_NAME (\_\_\_\_\_)) → You define name for those thing after name

因为a和b会take any argument, 但他们不立刻compare type  
直到后面function里面需要compare时候 才会compare.  
所以这要implicit polymorphism

(define min (lambda (a b) (if (< a b) a b)))

- applied to arguments of any type to which it can be applied
  - disadvantage: checked dynamically

- *Explicit: generics*

- C++: *templates*

- checked statically

it will run faster, because it avoid type checks

- Generics in object-oriented programming

- parametrize entire class
  - *container*

# Polymorphism

- Ada example: overloading (left) vs generics (right)

```
function min(x, y : integer)
    return integer is
begin
    if x < y then return x;
    else return y;
    end if;
end min;

function min(x, y : long_float)
    return long_float is
begin
    if x < y then return x;
    else return y;
    end if;
end min;
```

```
generic
    type T is private;
    with function "<"(x, y : T) return Boolean;
function min(x, y : T) return T;
begin
    if x < y then return x;
    else return y;
    end if;
end min;

function int_min is new min(integer, "<");
function real_min is new min(long_float, "<");
function string_min is new min(string, "<");
function date_min is new min(date, date_precedes);
```

- C++ example

```
template<class item, int max_items = 100>
class queue {
    item items[max_items];
    int next_free, next_full, num_items;
public:
    queue() : next_free(0), next_full(0), num_items(0) { }
    bool enqueue(const item& it) {
        if (num_items == max_items) return false;
        ++num_items; items[next_free] = it;
        next_free = (next_free + 1) % max_items;
        return true;
    }
    bool dequeue(item* it) {
        if (num_items == 0) return false;
        --num_items; *it = items[next_full];
        next_full = (next_full + 1) % max_items;
        return true;
    }
};

...
queue<process> ready_list;
queue<int, 50> int_queue;
```

# Arrays

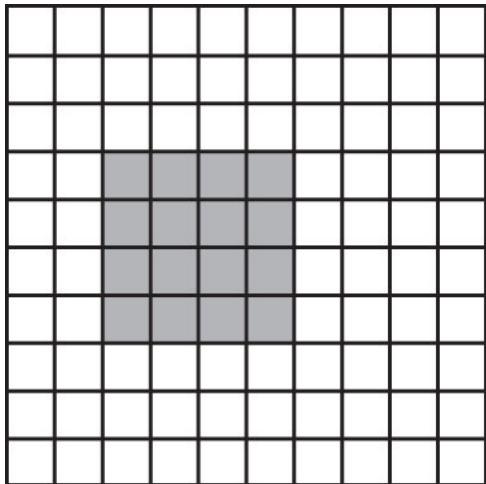
- Arrays
  - the most important composite data type
  - semantically, map: index type → element type
- Homogenous data
- Index type
  - usually discrete type: int, char, enum, subranges of those
  - non-discrete type: *associative array, dictionary, map*
    - implemented using hash tables or search trees
- Dense – most positions non-zero
  - sparse arrays – stored using linked structures

they are not stored in Array data structure

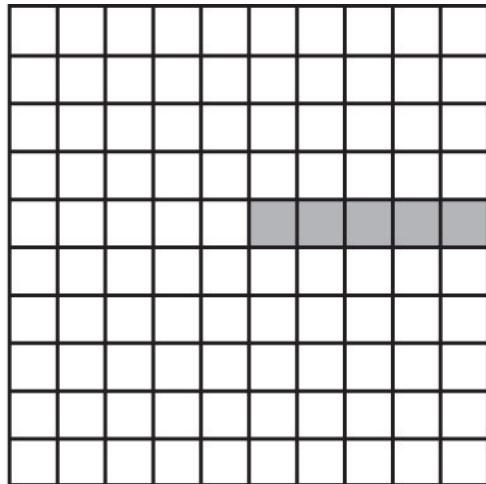
# Arrays

- Slices

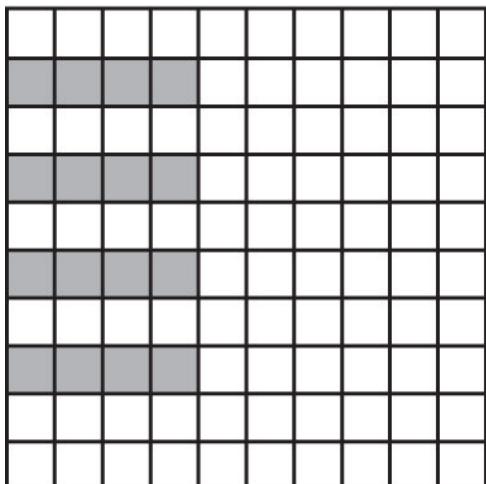
Matrix( column start : column end, row start : row end)



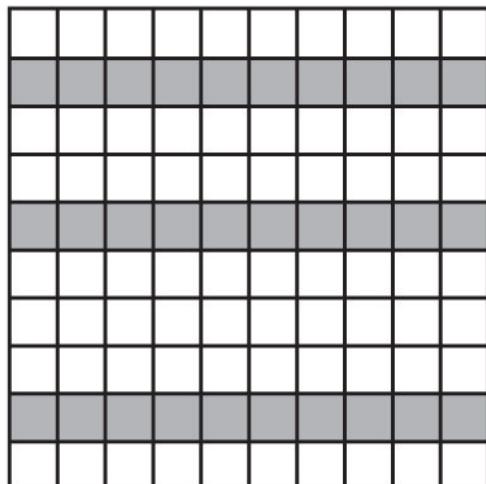
matrix(3:6, 4:7)  
column from 3 to 6    row from 4 to 7



matrix(6:, 5)  
column from 6 to end (inward)    row 5



matrix(:4, 2:8:2)



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

# Arrays: Dimensions, Bounds, Allocation

- Static allocation:

- array with lifetime the entire program
- shape known at compile time

如果要变size就用heap  
如果size固定就要用stack

- Stack allocation:

- array with lifetime inside subroutine
- shape known at compile time

- Heap / stack allocation

- dynamically allocated arrays
- *dope vector*: holds shape information at run time
- compiled languages need the number of dimensions
- shape known at elaboration time – can allocate on stack
- shape changes during execution: allocated on heap

因为知道shape  
所以好allocate  
memory 用stack

# Arrays

You know the shape of T

- Example: C dynamic local array

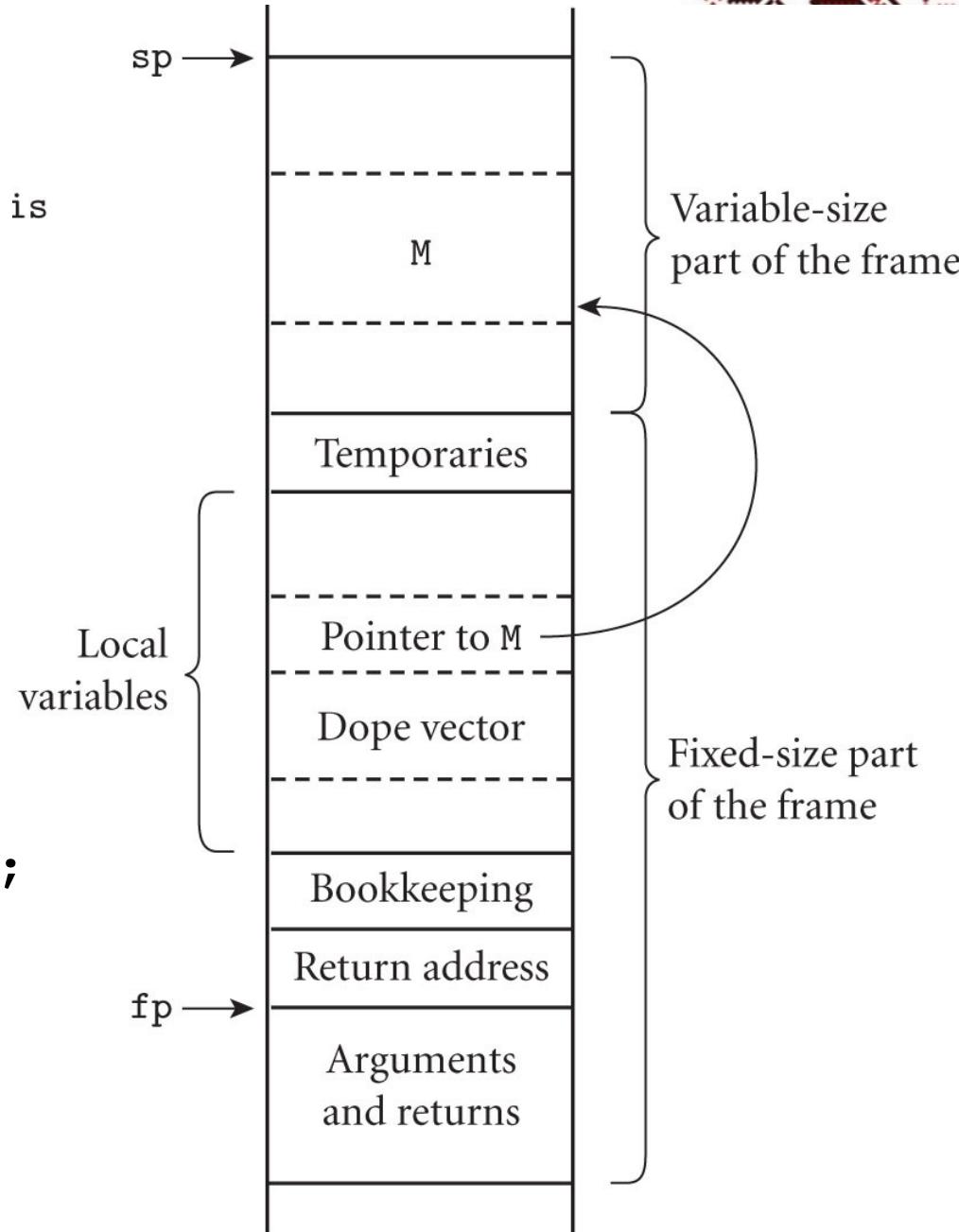
```
void square(int n, double M[n][n]) {  
    double T[n][n];  
    for (int i = 0; i < n; i++) { // copy product to T  
        for (int j = 0; j < n; j++) {  
            double s = 0;  
            for (int k = 0; k < n; k++)  
                s += M[i][k] * M[k][j];  
            T[i][j] = s;  
        }  
    }  
    for (int i = 0; i < n; i++) { // copy T back to M  
        for (int j = 0; j < n; j++)  
            M[i][j] = T[i][j];  
    }  
}
```

# Arrays

- Shape known at elaboration time
  - can be allocated on stack
  - in the variable-size part

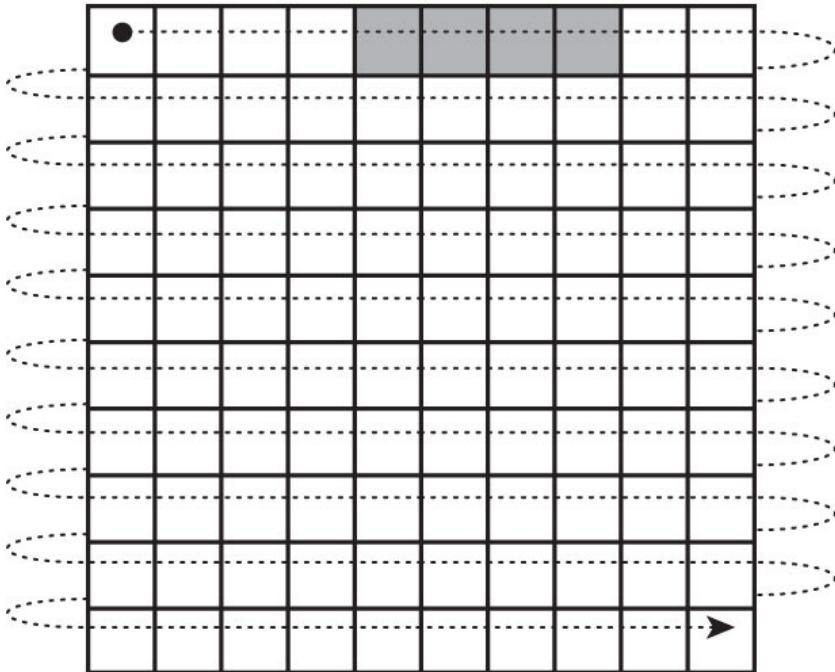
- Example:

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

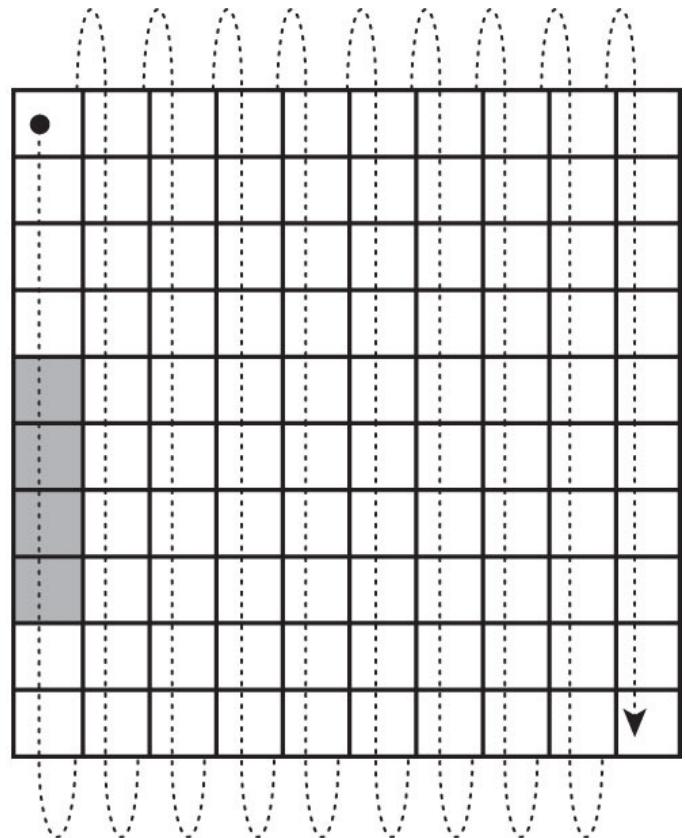


# Arrays

- Memory layout
  - column major order – Fortran
  - row major order – everybody else



Row-major order



Column-major order

# Arrays

- Memory layout
- Contiguous allocation
  - consecutive locations in memory:  $A[2,4]$ ,  $A[2,5]$
  - consecutive rows adjacent in memory
- Row pointers
  - consecutive rows anywhere in memory
  - extra memory for pointers
  - rows can have different lengths (ragged array)
  - can construct an array from existing rows without copying
- C, C++, C# - allow both
- Java – only row-pointer for all arrays

# Arrays

- Example: C – array of strings
  - true two-dimensional array

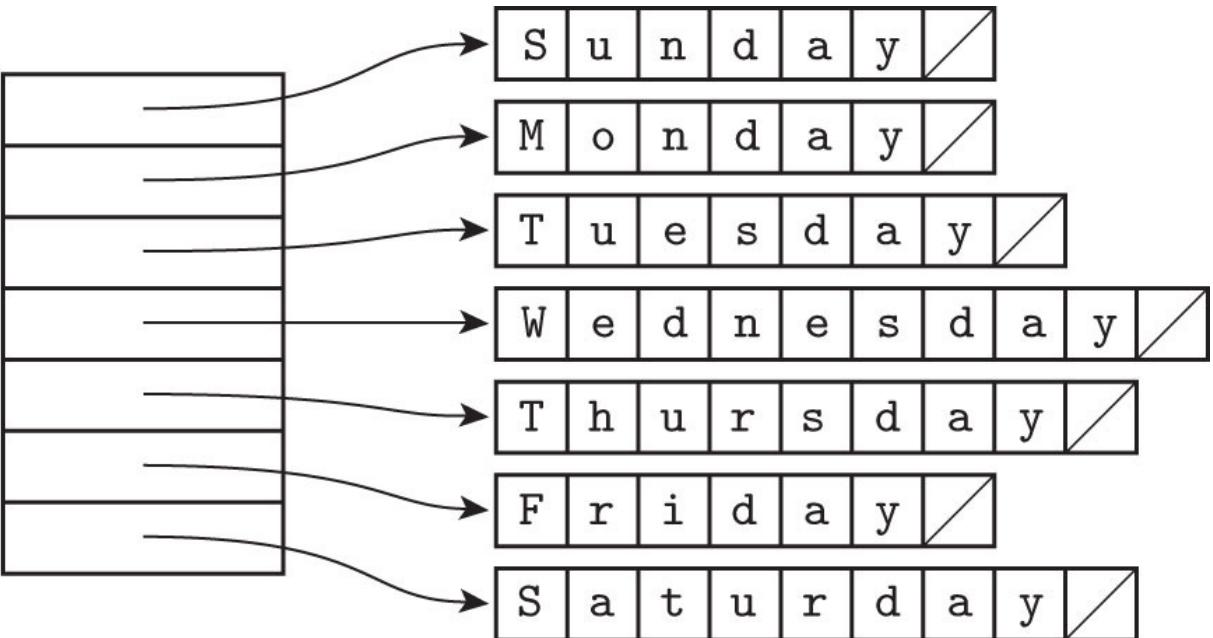
```
char days[] [10] = {  
    "Sunday", "Monday", "Tuesday",  
    "Wednesday", "Thursday",  
    "Friday", "Saturday"  
};  
...  
days[2][3] == 's'; /* in Tuesday */
```

S	u	n	d	a	y	/			
M	o	n	d	a	y	/			
T	u	e	s	d	a	y	/		
W	e	d	n	e	s	d	a	y	/
T	h	u	r	s	d	a	y	/	
F	r	i	d	a	y	/			
S	a	t	u	r	d	a	y	/	

# Arrays

- Example: C – array of strings
  - array of pointers

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



# Arrays

- Address calculation

A : array [L<sub>1</sub>..U<sub>1</sub>] of array [L<sub>2</sub>..U<sub>2</sub>] of array  
[L<sub>3</sub>..U<sub>3</sub>] of elem\_type;

$$S_3 = \text{size of elem\_type}$$

$$S_2 = (U_3 - L_3 + 1) \times S_3$$

$$S_1 = (U_2 - L_2 + 1) \times S_2$$

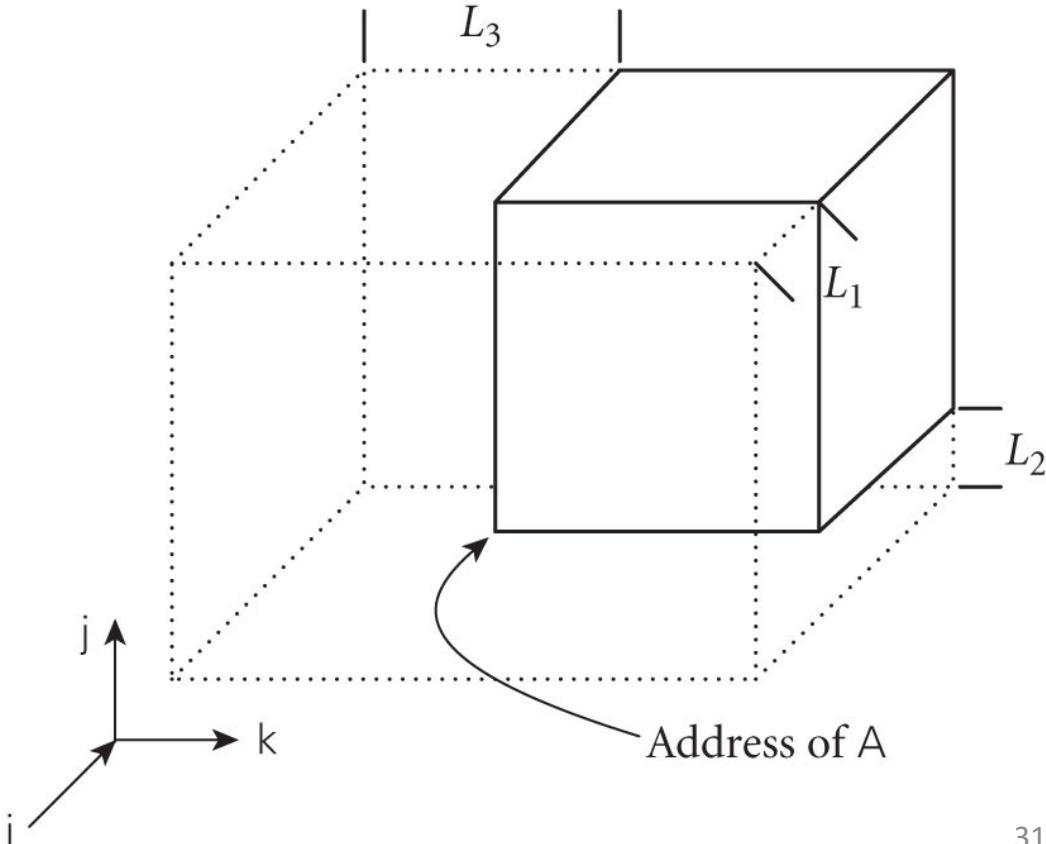
address of A [ i , j , k ]

= address of A

$$+ (i - L_1) \times S_1$$

$$+ (j - L_2) \times S_2$$

$$+ (k - L_3) \times S_3$$



# Arrays

- Faster address calculation

address of  $A[i, j, k]$

$$= \text{address of } A + (i - L_1) \times S_1 + (j - L_2) \times S_2 + (k - L_3) \times S_3$$

- Fewer operations

- $C = [(L_1 \times S_1) + (L_2 \times S_2) + (L_3 \times S_3)]$

- $C$  – known at compile time

address of  $A[i, j, k]$

$$= \text{address of } A + (i \times S_1) + (j \times S_2) + (k \times S_3) - C$$

# Sets

- *Set*: unordered collection of an arbitrary number of distinct values of a common type
- Implementation
  - *characteristic array* – one bit for each value (small base type)
    - efficient operations – bitwise op
  - general implementation: hash tables, trees, etc.
  - Python, Swift – built-in sets
  - Others use dictionaries, hashes, maps

```
X = set(['a', 'b', 'c', 'd']) # set constructor
Y = {'c', 'd', 'e', 'f'}       # set literal
U = X | Y                     # union
I = X & Y                     # intersection
D = X - Y                     # difference
O = X ^ Y                     # symmetric diff.
'c' in I                      # membership
```

# Pointers and Recursive Types

- *Pointer*

- a variable whose value is a reference to some object
- not needed with a reference model of variables
- needed with a value model of variables
- efficient access to complicated objects

- *Recursive type*

- objects contain references to other objects
- can create dynamic data structures

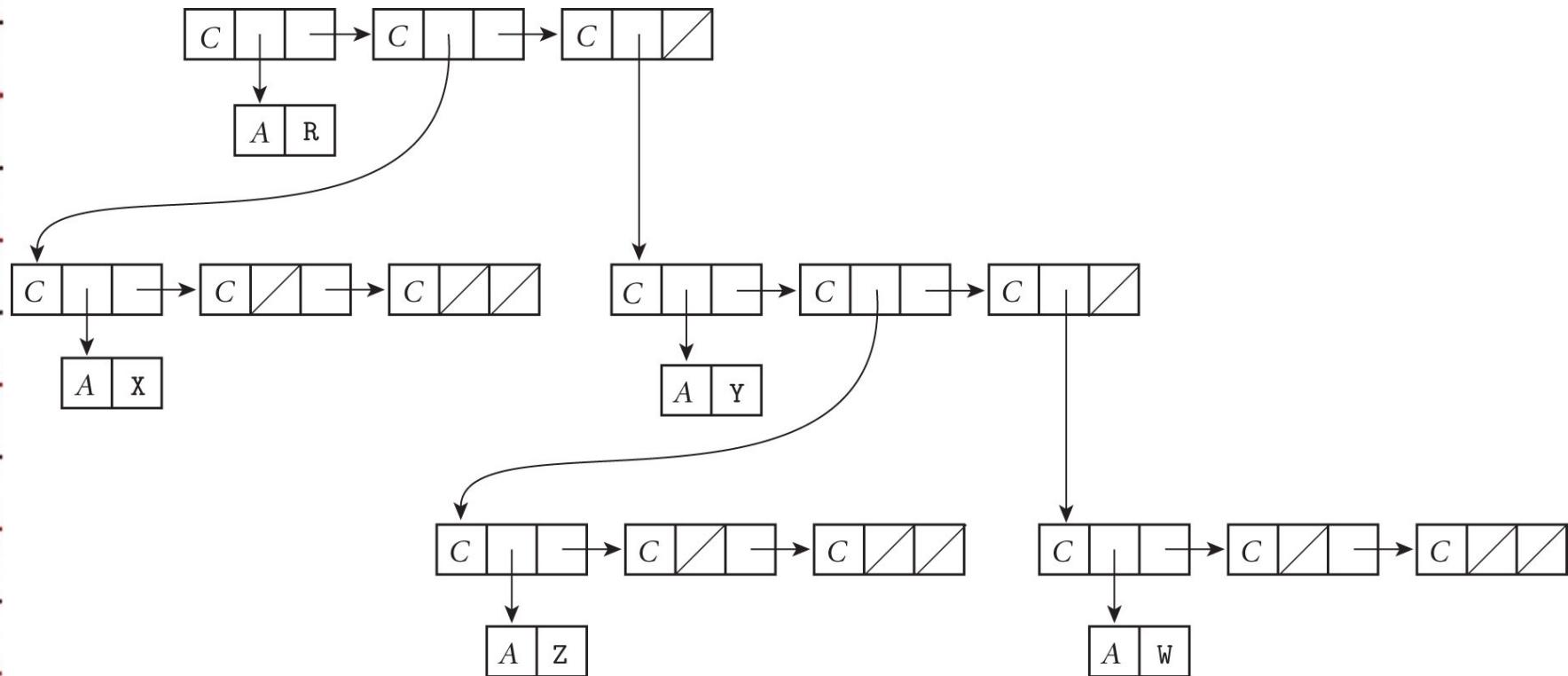
- Pointer  $\neq$  address

- pointer = high-level concept; reference to object
- address = low-level concept; location in memory
- pointers are implemented as addresses

# Pointers and Recursive Types

- Reference model example: Tree in Scheme
- Two types of objects: (1) `cons` cells (2) atoms

```
' (#\R (#\X ()()) (#\Y (#\Z ()()) (#\W ()())))
```



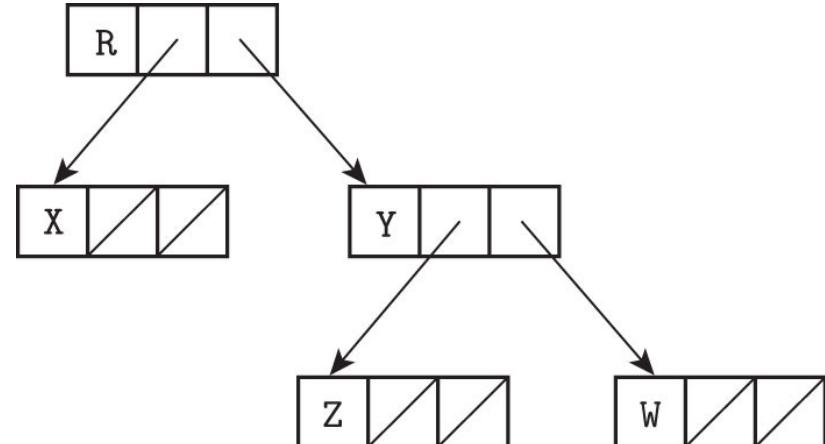
# Pointers and Recursive Types

- Value model example: Tree in C

```
struct chr_tree {  
    struct chr_tree *left, *right;  
    char val;  
};  
my_ptr = malloc(sizeof(struct chr_tree));
```

- C++, Java, C# – type safe

```
my_ptr = new chr_tree(arg_list);  
(*my_ptr).val = 'X';  
my_ptr->val = 'X';
```



# Pointers and Recursive Types

- *Dangling reference*

- live pointer that no longer points to a valid object
- Example: caused by local variable after subroutine return:

```
int i = 3;
int *p = &i;
...
void foo(){ int n = 5; p = &n; }
...
cout << *p;           // prints 3
foo();
...
cout << *p; // undefined behavior: n is no
              longer live
```

# Pointers and Recursive Types

- Dangling reference
  - Example: caused by manual deallocation:

```
int *p = new int;  
*p = 3;  
...  
cout << *p;      // prints 3  
delete p;  
...  
cout << *p; // undefined behavior: *p has been  
              reclaimed
```

- Problem: a dangling reference can write to memory that is part of a different object; it may even interfere with bookkeeping, corrupting the stack or heap

# Garbage collection

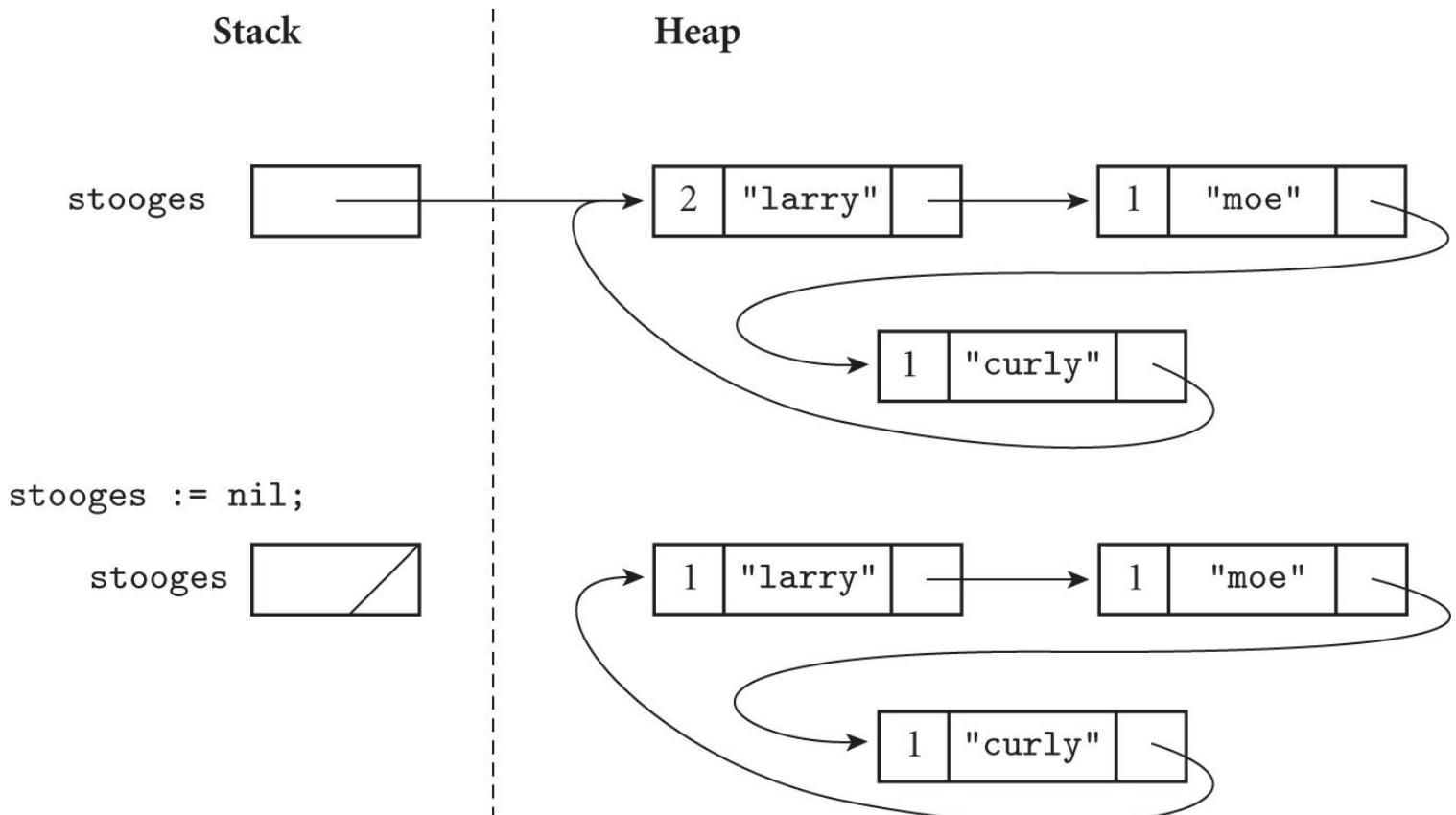
- *Garbage collection*
  - automatic reclamation of memory
  - slower than manual (`delete`)
  - difficult to implement
  - eliminates the need to check for dangling references
  - very convenient for programmers
  - essential for functional languages
  - increasingly popular in imperative languages
    - Java, C#

# Garbage collection

- *Reference counts*
  - object no longer useful when no pointers to it exist
  - store *reference count* for each object
    - initially set to 1
    - update when assigning pointers
    - update on subroutine return
    - when 0, reclaim object

# Garbage collection

- Reference counts
  - count  $\neq 0$  does not necessarily mean useful (circular lists)



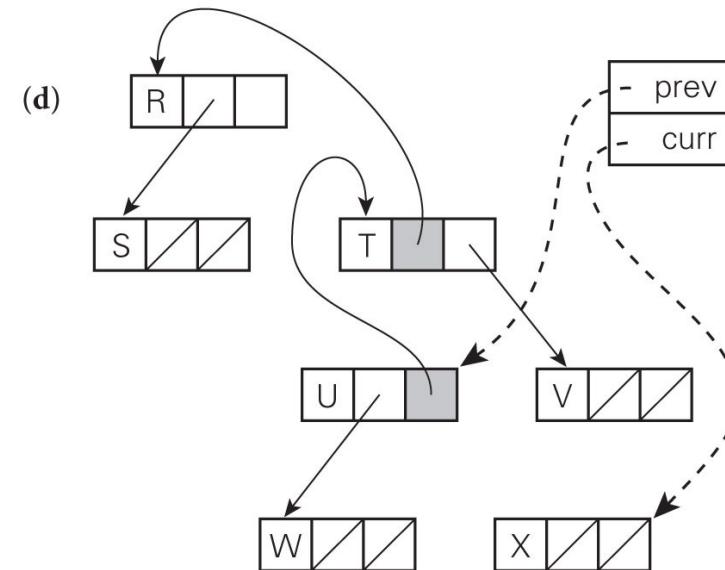
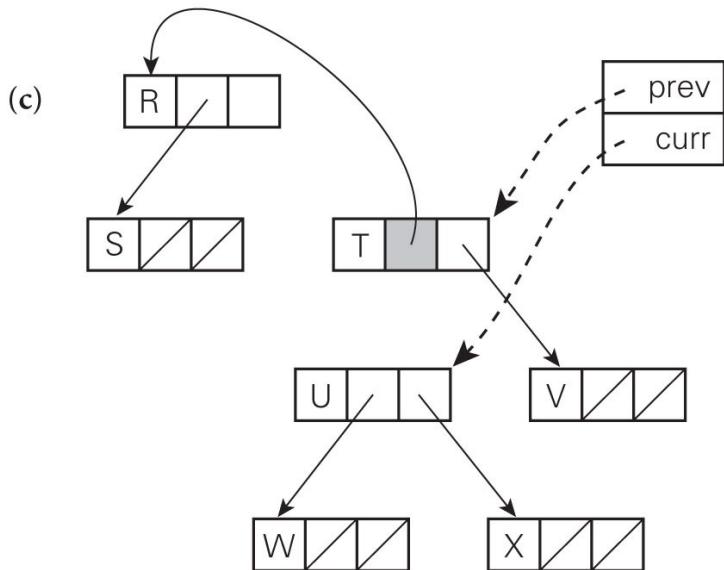
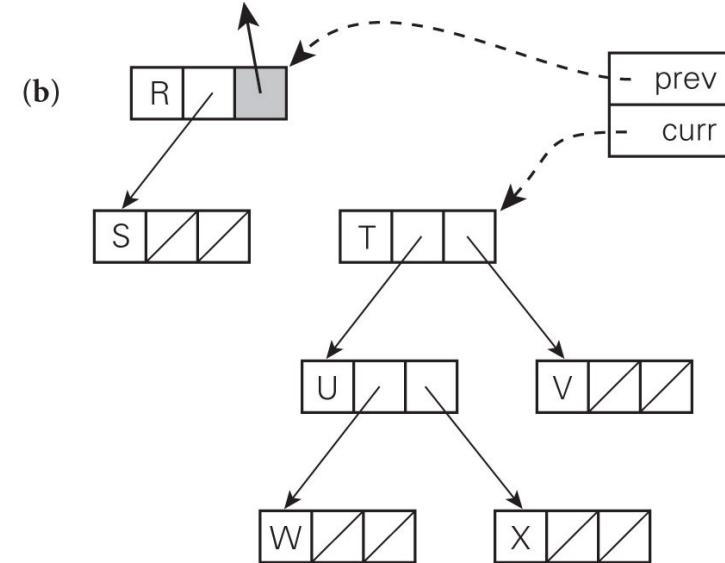
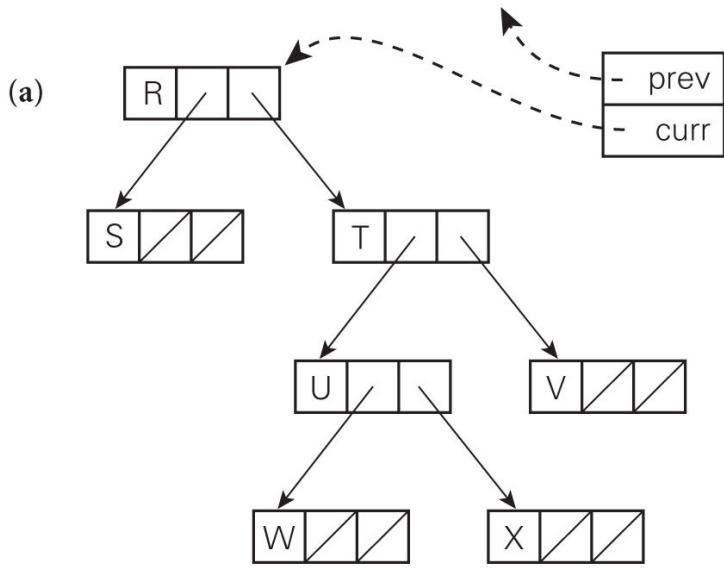
# Garbage collection

- Smart pointers in C++
- `unique_ptr`
  - one object only
- `shared_ptr`
  - implements a reference count
- `weak_ptr`
  - does not affect counts; for, e.g., circular structures

# Garbage collection

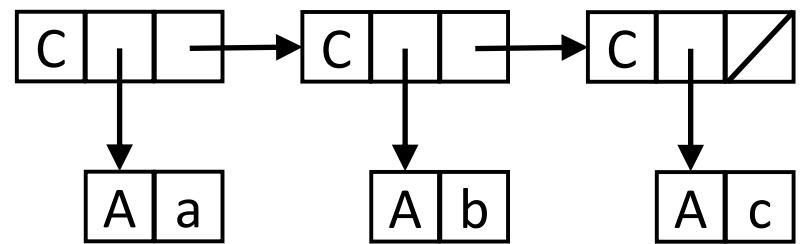
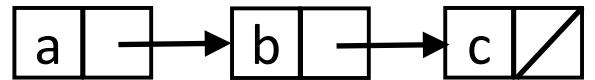
- *Tracing collection*
  - object useful if reachable via chain of valid pointers from outside the heap
  - *Mark-and-sweep*
    - (1) mark entire heap as “useless”
    - (2) starting from outside heap, recursively mark as “useful”
    - (3) move “useless” block from heap to free list
  - Step (2) requires a potentially very large stack
  - Without stack: *pointer reversal* (next slide)
- *Stop-and-copy*: defragmentation
  - use half of heap; copy useful data compactly to the other one

# Pointer reversal



# Lists

- *List*: empty or (head + tail)
- essential in functional and logic programming (recursive)
- used also in imperative languages
- *Homogeneous* (same type): ML
- *Heterogeneous*: Scheme



# Lists

- Scheme:

- `' ( . . . )` prevents evaluation; also `(quote ( . . . ))`

`(+ 1 2) ⇒ 3`

`' (+ 1 2) ⇒ ' (+ 1 2)`

`(cons 'a '(b)) ⇒ ' (a b)`

`(car ' (a b)) ⇒ a`

`(car ' ()) ⇒ error`

`(cdr ' (a b c)) ⇒ ' (b c)`

`(cdr ' (a)) ⇒ ' ()`

`(cdr ' ()) ⇒ error`

`(append ' (a b) ' (c d)) ⇒ ' (a b c d)`

# Lists

- *List comprehension*

- adapted from traditional math set notation:

$$\{i \times i \mid i \in \{1, \dots, 10\} \wedge i \bmod 2 = 1\}$$

- Example: Python

```
[i*i for i in range(1, 10) if i % 2 == 1]  
⇒ [1, 9, 25, 49, 81]
```