Types

CMPSC 461
Programming Language Concepts
Penn State University
Fall 2016

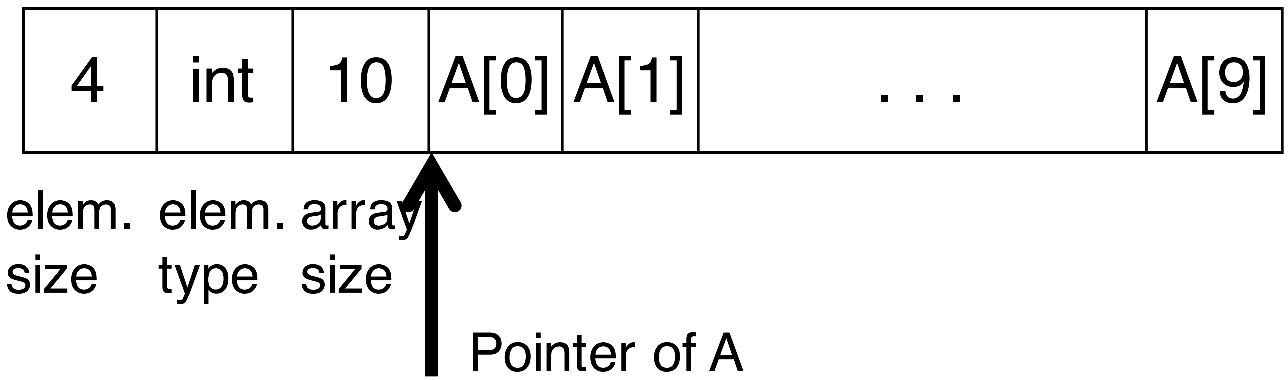
Kinds of Types

Primitive

Constructed

- Products
- Unions
- ***Arrays***

Dope Vectors (one example)



Address of A[i]? A + 4*I

Bound check? 0 <= i < 10

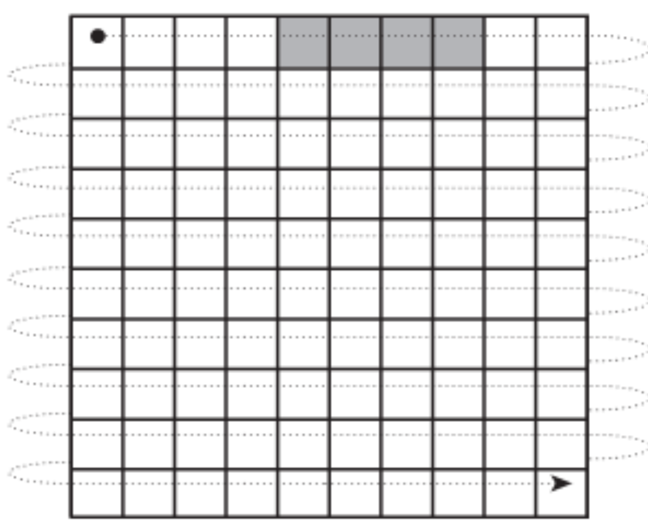
Benefit: the array may change dynamically

Memory Layout

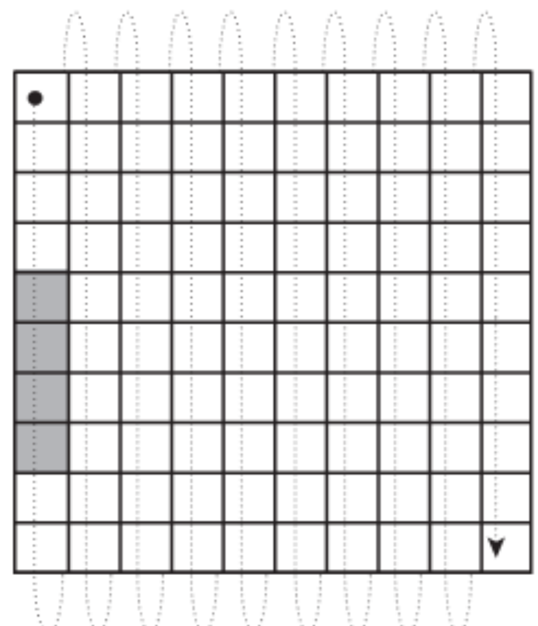
One-dimensional arrays

Two-dimensional arrays

```
int[][] A = new int[10][100]
```



Row-major order

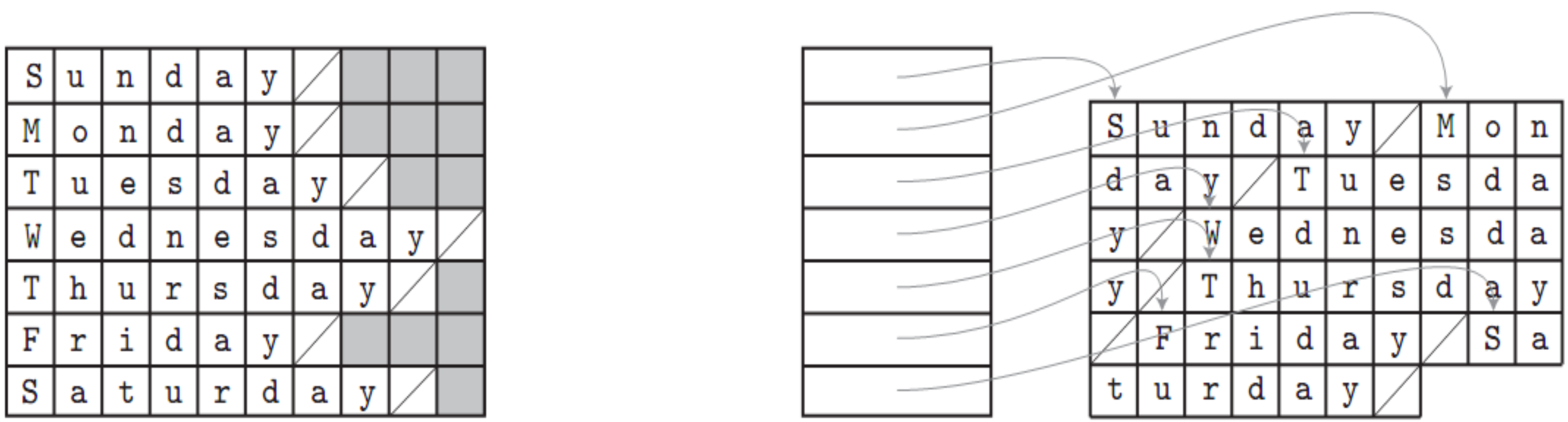


Column-major order

Memory Layout

Row-Pointer Layout

```
int[][] B = new int[10][]
B[0] = new int[100]
B[1] = new int[50]
```



Address Calculation (Row major)

4	int	10	100	A[0][0]	A[0][1]	. . .	A[9][99]
elem. size	elem. type	rows	colu-mns				

Pointer of A

Address of A[i][j]? $A + 4(i*100+j)$

Bound check? $0 <= i < 10,\ 0 <= j < 100$

Address Calculation (Column major)

4	int	10	100	A[0][0]	A[1][0]	. . .	A[9][99]
elem. size	elem. type	rows	colu-mns				

Pointer of A

Address of A[i][j]? $A + 4(j*10+i)$

Bound check? $0 <= i < 10, 0 <= j < 100$

Pointers vs. References

Pointers: `int *p;`

References: `int &p;`

Value Model vs. Reference Model: A = B

- Value model: the value of B is copied to A
- Reference model: A is an alias of B (same memory)
- Java: primitive types follow value model; objects follow reference model

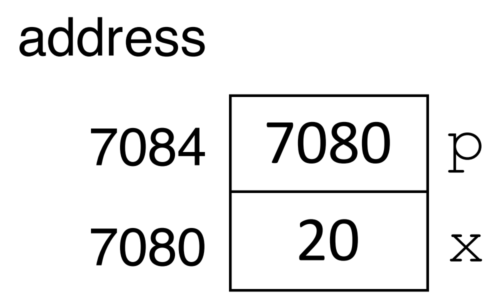
Pointers

What are they?

- A set of memory addresses and operations on them

Values: legal addresses, and a special value, `nil`

```
int x=20;
int* p = &x;
```



Pointers

Operations: assignment, dereferencing, arithmetic

```
int x=20;
int* p = &x;
int y=*p;
```

```
int a[3] = {1,2,3};
int x = *(a+1) //same as a[1]
```

Uses

- Indirect addressing (access arbitrary address)
- Manage dynamic storage (heap)

The Nil Pointer Problem

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. ... This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."

C.A.R. Hoare, 2009

How to avoid it?

The Nil Pointer Problem

Use sum types

Haskell

```
data Maybe Int = Nothing | Just Int
-- given p has type Maybe Int
case p of
  Nothing -> .. nil value ..
  Just i -> .. valid value ..
```

SML

```
datatype int option = NONE | SOME of int
-- given p has type int option
case p of
  None => .. Nil value ..
| SOME i => .. Valid value ..
```

C Pointers and Arrays

Pointers and arrays are related

- `int *a  ==  int a[]`
- `int **a ==  int *a[]`

But equivalences don’t always hold

- `int a[10]` also allocates memory for the array
- 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]` – 2D array

Dangling References

Problem: a pointer to a deallocated address

- Explicit deallocation

```
int *pt1 = new int[5];
Int *pt2 = pt1;
delete pt1; // pt2 is dangling
```

- Implicit deallocation

```
int *pt=null;
void foo () {
  int x = 5;
  pt = &x;
}
foo(); // pt is dangling
```

Solutions

Don’t allow user control (Java)

Safety algorithms: detect dangling pointers

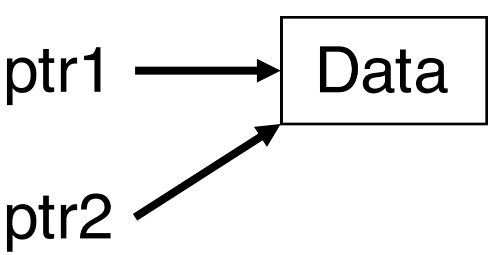
- Tombstones
- Locks & Keys

Tombstones

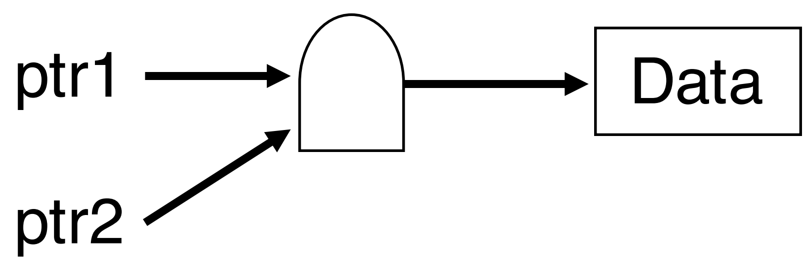
Each heap variable is given another memory location (tombstone)

The tombstone points to the heap variable

All pointers to the variable point to the tombstone



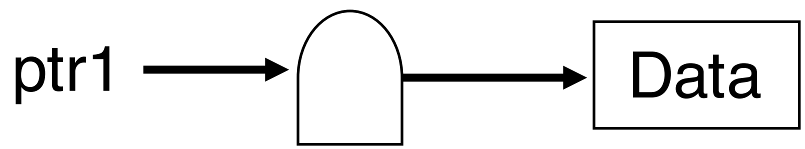
Without tombstone



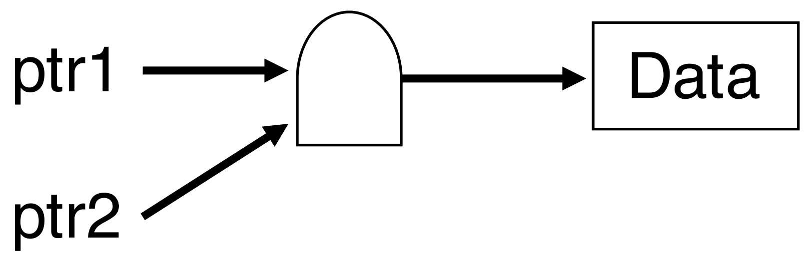
With tombstone

Tombstones

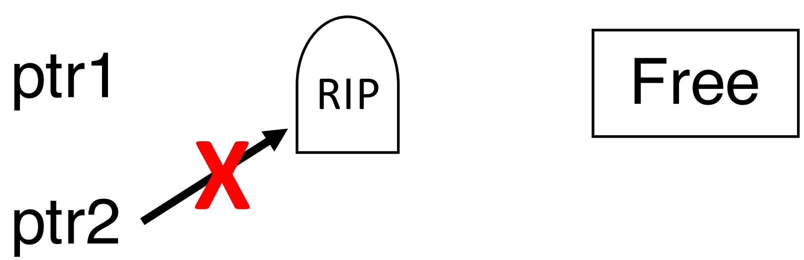
`new(ptr1)`



`ptr2 = ptr1`



`delete(ptr1)`



Cost: extra memory space, extra memory access

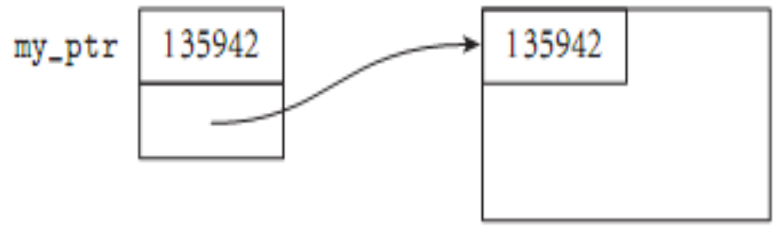
Locks & Keys

When a heap variable is allocated

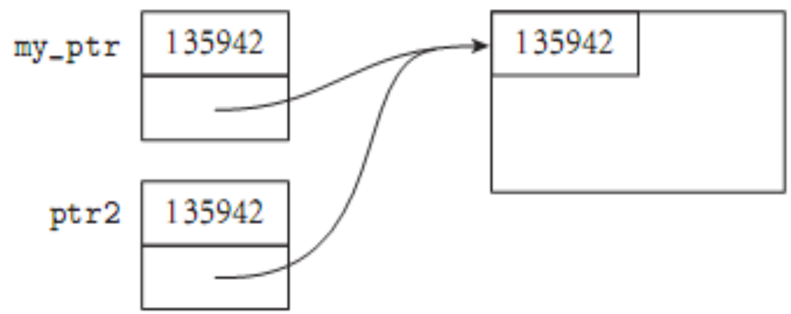
- Allocate storage for the value
- Allocate an integer which holds a ***lock value***
- Return a pair of ***key value*** and address
- Key value is set of lock value

Locks & Keys

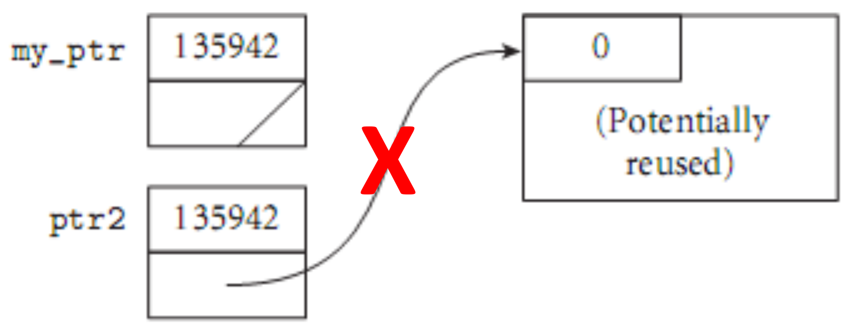
```
new(my_ptr)
```




```
ptr2 = my_ptr
```




```
delete(my_ptr)
```



References

Restricted pointers: cannot be used as value or operated in any way

Not directly visible to the programmer

No explicit data type

```
double r=2.3;
double& s=r;//s,r share memory
double *p = &r; //p has value: address of r
s += 1; *p += 1;
```

References

Uses

- ~~Indirect addressing (access arbitrary address)~~
- Manage dynamic storage (heap)

Alias of existing variables