Programming Paradigms

Composite Types (Part 2)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Summer 2023

Quiz: Pointer Warm-Up

What does the following C code print?

```
int *iptr = (int *) 0x1005;
char *cptr = (char *) 0x1005;

void *a = iptr+3;
void *b = cptr+3;

printf("%p %p\n", a, b);
```

Quiz: Pointer Warm-Up

What does the following C code print?

```
int *iptr = (int *) 0x1005;
char *cptr = (char *) 0x1005;

void *a = iptr+3;
void *b = cptr+3;

printf("%p %p\n", a, b);
```

Result: `0x1011 0x1008`

Quiz: Pointer Warm-Up

What does the following C code print?

```
int *iptr = (int *) 0x1005;
char *cptr = (char *) 0x1005;

void *a = iptr+3;
void *b = cptr+3;

printf("%p %p\n", a, b);
```

Two pointers initialized with hexadecimal numbers.

Result: `0x1011 0x1008`

Quiz: Pointer Warm-Up

What does the following C code print?

```
int *iptr = (int *) 0x1005;
char *cptr = (char *) 0x1005;

void *a = iptr+3;
void *b = cptr+3;

printf("%p %p\n", a, b);
```

Adding `3*size(t)` to each pointer, where `t` is the type the pointer refers to.

Result: `0x1011 0x1008`

Overview

- **Records**
- **Arrays**
- **Pointers and Recursive Types** ←
  - Operations on Pointers
  - Pointers and Arrays in C
  - Dangling References
  - Garbage Collection

Motivation

- **Most programs handle complex data**
- **“Linked” data structures to represent them**
  - Lists
  - Trees
  - Graphs
- **Often: Want reference to objects of same type**

Pointers and Recursive Types

- **Pointer: Reference to location of memory object**
    - Essentially, an address
- **Recursive type: Composite type with reference to objects of the same type**

Reference vs. Value Model

PLs with reference model of variables

- No need for explicit pointers
- Fields simply refer to object of same (or other) type

PLs with value model of variables

- Need explicit pointers to refer to objects
- Otherwise, would always copy the entire memory object

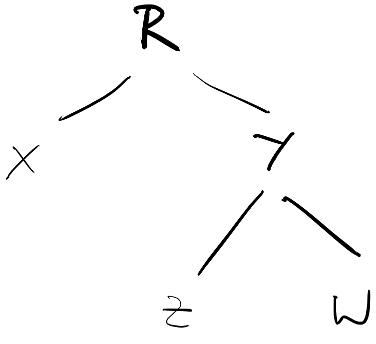
Example: Tree in OCaml

```
type chr_tree =
    Empty |
    Node of char * chr_tree * chr_tree;;
```

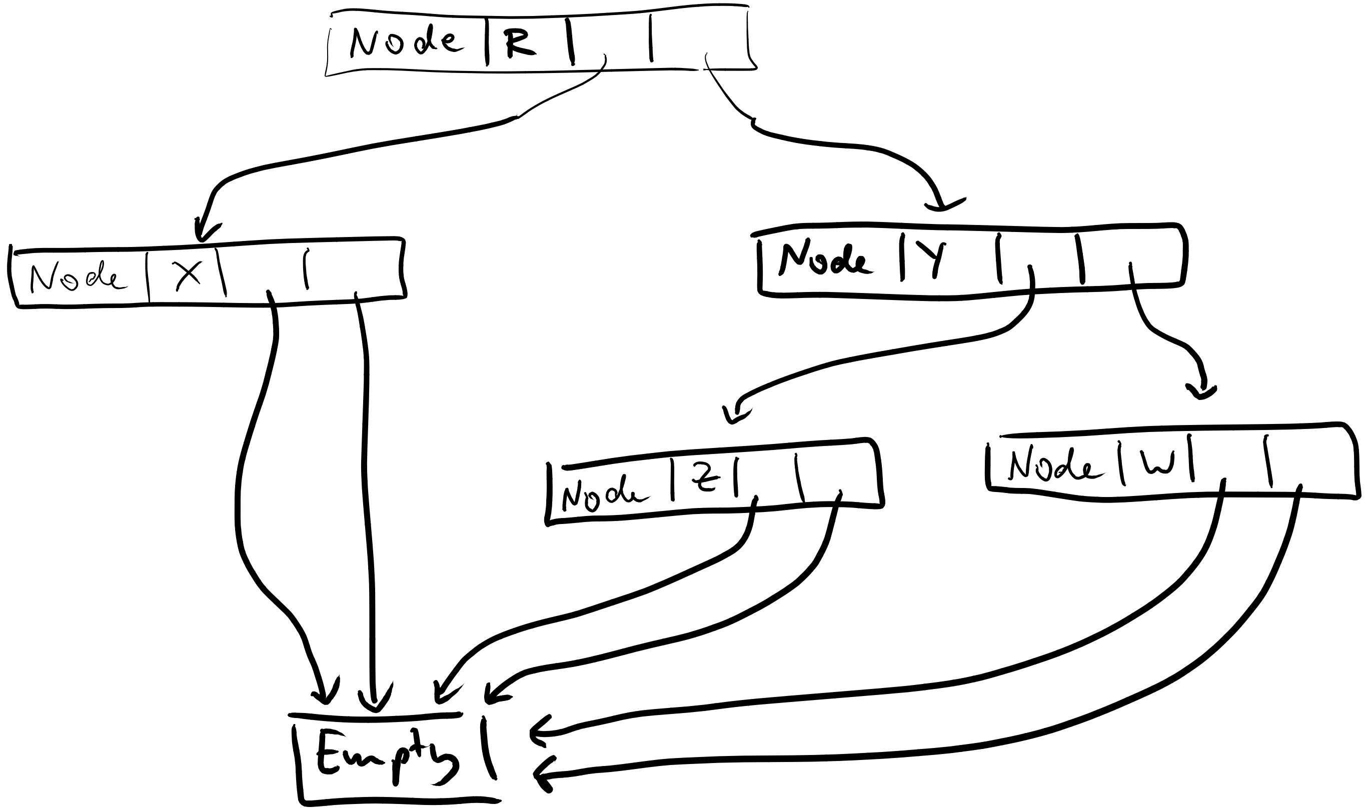
Tuple type with fields separated by `*`

Example: Ocaml

Conceptually:



In memory:

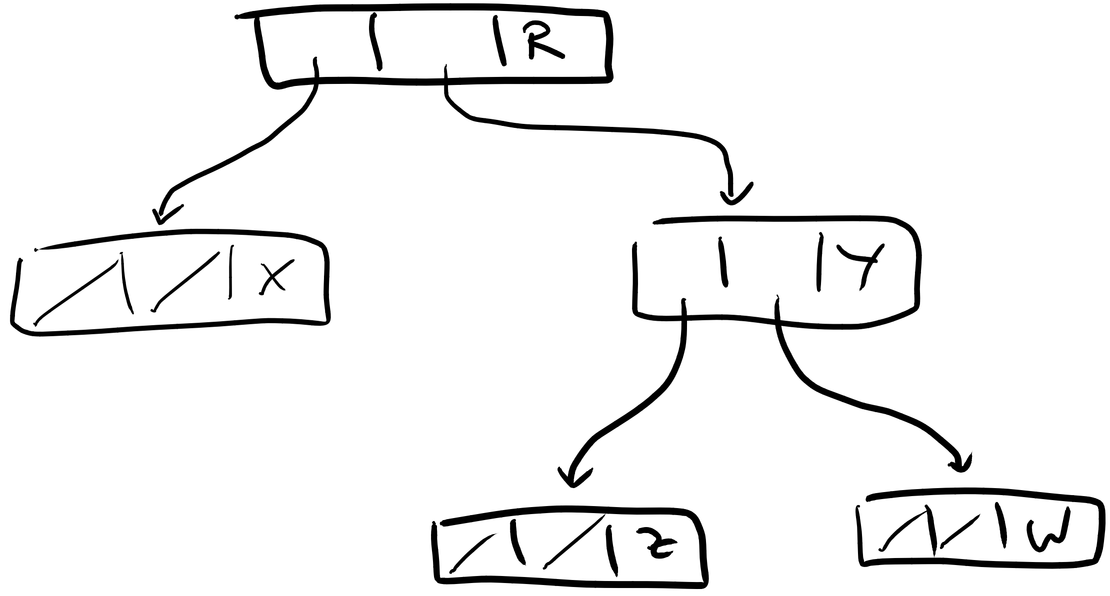


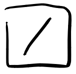
Example: Tree in C

```
struct chr_tree {
  struct chr_tree *left, *right;
  char var;
};
```

Pointers to objects of type `struct chr_tree`

Example: C



 mean null pointer

Operations on Pointers

- Creation
- Allocation
- Dereference
- Deallocation

Operations on Pointers

- Creation
- Allocation
- Dereference
- Deallocation

Handled differently in different PLs

Creating Pointers

- **Implicit** when calling a constructor
- **Built-in function** that allocates heap memory and returns reference to it
- **Address-of operator**

Creating Pointers

- **Implicit** when calling a constructor
- **Built-in function** that allocates heap memory and returns reference to it
- **Address-of operator**

Example (C++):

```
my_ptr = new chr_tree(/* ... */);
```

Creating Pointers

- **Implicit** when calling a constructor
- **Built-in function** that allocates heap memory and returns reference to it
- **Address-of operator**

Example (C):

```
my_ptr = malloc(sizeof(struct chr_tree));
```

Creating Pointers

- **Implicit** when calling a constructor
- **Built-in function** that allocates heap memory and returns reference to it
- **Address-of operator**

Example (C):

```
int n = 3;
my_ptr = &n;
```

Allocating Memory

- **Pointer itself is only an address**
- **Need sufficient memory to hold the object it refers to**
- **Memory allocation**
  - Implicit on some PLs (e.g., OCaml, Java)
  - Explicit in other PLs (e.g., C)

Allocating Memory

- **Pointer itself is only an address**
- **Need sufficient memory to hold the object it refers to**
- **Memory allocation**
  - ☐ Implicit on some PLs (e.g., OCaml, Java)
  - ☐ Explicit in other PLs (e.g., C)

Example (OCaml):

```
let t = Node(’R’, Empty, Empty);;
```

Allocating Memory

- **Pointer itself is only an address**
- **Need sufficient memory to hold the object it refers to**
- **Memory allocation**
  - Implicit on some PLs (e.g., OCaml, Java)
  - Explicit in other PLs (e.g., C)

Example (C):

```
my_ptr = malloc(sizeof(struct chr_tree));
// fill object with content
```

Dereferencing a Pointer

Access memory object a pointer refers to

- Access entire object
    - Dereferencing operator

- Access fields of record that the pointer refers to
    - Right-arrow notation
    - Dot notation:

      Implicit dereferencing

Dereferencing a Pointer

Access memory object a pointer refers to

- Access entire object
  - ☐ Dereferencing operator

Example (Pascal):

```
my_ptr^.val := 'X';
```

Example (C):

```
(*my_ptr).val = 'X';
```

- Access fields of record that the pointer refers to
  - ☐ Right-arrow notation
  - ☐ Dot notation:

    Implicit dereferencing

Dereferencing a Pointer

Access memory object a pointer refers to

- Access entire object
  - Dereferencing operator
- Access fields of record that the pointer refers to
  - Right-arrow notation →
  - Dot notation:

    Implicit dereferencing

Example (C):

```
my_ptr->val = 'X';
```

Dereferencing a Pointer

Access memory object a pointer refers to

- Access entire object
  - Dereferencing operator
- Access fields of record that the pointer refers to
  - Right-arrow notation
  - Dot notation:

    Implicit dereferencing

Example (Ada):

```
T : chr_tree;
P : chr_tree_ptr;
...
T.val := 'X';
P.val := 'Y';
```

Deallocation

- **Memory must be reclaimed at some point**
  - Otherwise: Memory leak and, eventually, out-of-memory
- **Explicit deallocation by programmer**
  - E.g., C, C++, Rust
- **Implicit deallocation by runtime: Garbage collection**
  - E.g., Java, C#, Python

Deallocation: Example

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *line = NULL;
    size_t size = 0;
    for(;;) {
        /* read line from stdin;
           implicitly allocates memory */
        getline(&line, &size, stdin);
        // ...

        line = NULL;
    }
    return 0;
}
```

Deallocation: Example

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *line = NULL;
    size_t size = 0;
    for(;;) {
        /* read line from stdin;
           implicitly allocates memory */
        getline(&line, &size, stdin);
        // ...

        line = NULL;
    }
    return 0;
}
```

Memory leak:
Each iteration
allocates memory
that gets never freed.

Deallocation: Example

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *line = NULL;
    size_t size = 0;
    for(;;) {
        /* read line from stdin;
           implicitly allocates memory */
        getline(&line, &size, stdin);
        // ...
        free(line);
        line = NULL;
    }
    return 0;
}
```

Fix: Free memory in each iteration

Quiz: Memory Leak

How many bytes of memory are leaked when executing the following code?

Assumption: ints occupy four bytes

```
int *c;
for (int i = 3; i <= 8; i += 2) {
    c = malloc(sizeof(int));
    if ((i-1) % 3 == 0) {
        free(c);
    }
}
```

Quiz: Memory Leak

How many bytes of memory are leaked when executing the following code?

Assumption: ints occupy four bytes

```
int *c;
for (int i = 3; i <= 8; i += 2) {
    c = malloc(sizeof(int));
    if ((i-1) % 3 == 0) {
        free(c);
    }
}
```

Answer: 8 bytes

Quiz: Memory Leak

How many bytes of memory are leaked when executing the following code?

Assumption: ints occupy four bytes

```
int *c;
for (int i = 3; i <= 8; i += 2) {
    c = malloc(sizeof(int));
    if ((i-1) % 3 == 0) {
        free(c);
    }
}
```

Iterations with `i` being 3, 5, and 7

Answer: 8 bytes

Quiz: Memory Leak

How many bytes of memory are leaked when executing the following code?

Assumption: ints occupy four bytes

```
int *c;
for (int i = 3; i <= 8; i += 2) {
    c = malloc(sizeof(int));
    if ((i-1) % 3 == 0) {
        free(c);
    }
}
```

Iterations with `i` being 3, 5, and 7

Memory freed only when `i` is 7

Answer: 8 bytes

Overview

- **Records**
- **Arrays**
- **Pointers and Recursive Types**
    - Operations on Pointers
    - Pointers and Arrays in C
    - Dangling References
    - Garbage Collection

Pointers and Arrays in C

- Closely linked language constructs
- Example

```
int n;
int *a;
int b[5] = {1,2,3,4,5};

a = b;
n = a[3];
n = *(a+3);
n = b[3];
n = *(b+3);
```

Pointers and Arrays in C

- Closely linked language constructs
- Example

```
int n;
int *a;
int b[5] = {1,2,3,4,5};

a = b;
n = a[3];
n = *(a+3);
n = b[3];
n = *(b+3);
```

Pointer to the initial element of `b`

Pointers and Arrays in C

- Closely linked language constructs
- Example

```
int n;
int *a;
int b[5] = {1,2,3,4,5};

a = b;
n = a[3];
n = *(a+3);
n = b[3];
n = *(b+3);
```

All store 4 into `n`

Array Access = Pointer Arithmetic

- **Subscript operator `[]` defined in terms of pointer arithmetic:**

  `E1[E2]` means `(*((E1)+(E2)))`

  - For any expressions `E1` and `E2`

- **E.g., `arr[3]` is equivalent to `3[arr]`**

More Pointer Arithmetic

Arithmetic operations beyond addition

- Subtraction
  - Get distance between two elements:

    `p1 - p2` where both are pointers to elements in the same array
- Comparison
  - Check if one element is at higher index than another:

    `p1 > p2`
- All scaled according to type of pointer

Difference: Allocation

Main difference between arrays and pointers

- Arrays are implicitely allocated:

  `int arr[10];` allocates space for ten ints

- Pointers must be explicitedly allocated:

  `int *arr;` does not allocate anything

Overview

- **Records**
- **Arrays**
- **Pointers and Recursive Types**
    - Operations on Pointers
    - Pointers and Arrays in C
    - Dangling References ←
    - Garbage Collection

Dangling References

- **Dangling reference: Live pointer that no longer points to a valid object**
- **Dual problem to memory leaks**
- **Created when**
  - Pointer to stack object escapes to surrounding context
  - Heap object is explicitely deallocated, but pointer lives on
- **Behavior of dereferencing: Undefined**

Quiz: Dangling References

At which line(s) does this C code use a dangling reference?

```
 1  char *foo() {
 2      char *cp = malloc(sizeof(char));
 3      return cp;
 4  }
 5  int main(void) {
 6      char *csp = malloc(5 * sizeof(char));
 7      csp[0] = ’a’;
 8      csp[2] = *foo();
 9      csp[4] = ’c’;
10      free(csp);
11      printf("%c %c %c\n", csp[0], csp[2], csp[4]);
12  }
```

Quiz: Dangling References

At which line(s) does this C code use a dangling reference?

```
 1  char *foo() {
 2      char *cp = malloc(sizeof(char));
 3      return cp;
 4  }
 5  int main(void) {
 6      char *csp = malloc(5 * sizeof(char));
 7      csp[0] = 'a';
 8      csp[2] = *foo();
 9      csp[4] = 'c';
10      free(csp);
11      printf("%c %c %c\n", csp[0], csp[2], csp[4]);
12  }
```

Dangling references

(because `free` was called)

Overview

- **Records**
- **Arrays**
- **Pointers and Recursive Types**
  - Operations on Pointers
  - Pointers and Arrays in C
  - Dangling References
  - Garbage Collection ←

Garbage Collection

- **Memory deallocation managed by PL implementation**
  - Avoids dangling references
  - Programmer can focus on other aspects of the code
- **Common in “managed languages”, e.g., Java, Python, JavaScript**

Reference Counts

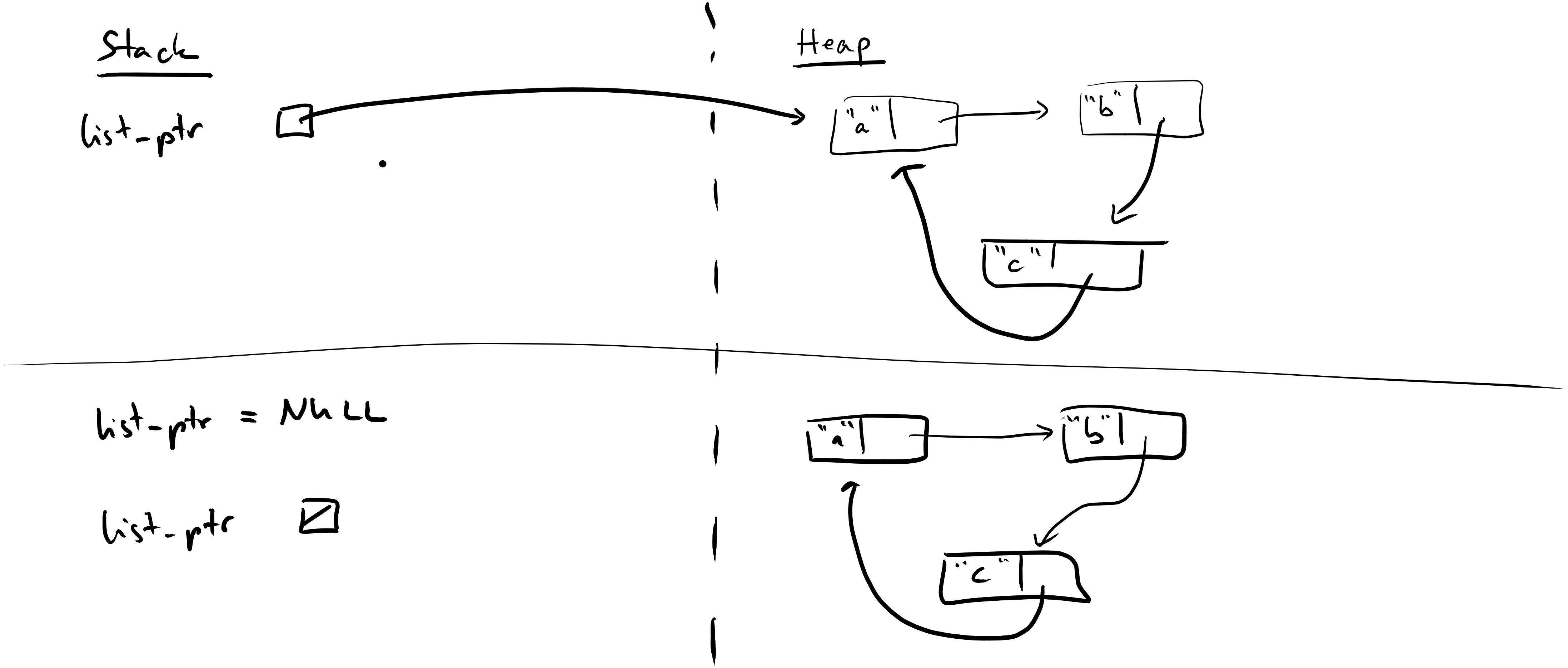
How to implement garbage collection?

- One counter for each memory object
- Increment when new pointer to object created
- Decrement when pointer gets destroyed
    - E.g., for pointers to local variables, on function return
- Deallocate “useless” objects, i.e., with reference count zero

Circular Dependencies

- **Problem of naive implementation: Circular data structures**
  - Memory object may be “useless” despite having references pointing to it

Example: Circular Data Structure



Circular Dependencies

- **Problem of naive implementation: Circular data structures**
  - Memory object may be “useless” despite having references pointing to it
- **Better approach**
  - Object o is “useless” unless there is a chain of valid pointers
    - from a name with an active binding
    - to o

Mark and Sweep

Algorithm to identify useless blocks

- Walk heap and mark every block as useless
- Start from external references (i.e., names in program) and mark every reachable block as useful
- Move all useless blocks to free list
    - Free list: Data structure to maintain free heap space

Optimizations and Other Algorithms

- **Various improvements of simple mark and sweep**
  - Pointer reversal: Traversal without a stack of visited blocks
  - Stop-and-copy: Prevent fragmentation
  - Generational garbage collection: Maintain older and newer memory objects in separate subheaps

Overview

- **Records**
- **Arrays**
- **Pointers and Recursive Types**
  - Operations on Pointers
  - Pointers and Arrays in C
  - Dangling References
  - Garbage Collection ✓