Programming Paradigms

Composite Types (Part 4)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Summer 2020

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**

```c
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       cp[0] = ’b’;
4       return cp;
5   }
6   int main(void) {
7       char *csp = malloc(3 * sizeof(char));
8       csp[0] = ’a’;
9       csp[1] = *foo();
10      csp[2] = ’c’;
11      free(csp);
12      printf("%c %c %c\n", csp[0], csp[1], csp[2]);
13  }
```

Please vote via Ilias.

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       cp[0] = 'b';
4       return cp;
5   }
6   int main(void) {
7       char *csp = malloc(3 * sizeof(char));
8       csp[0] = 'a';
9       csp[1] = *foo();
10      csp[2] = 'c';
11      free(csp);
12      printf("%c %c %c\n", csp[0], csp[1], csp[2]);
13  }
```

Dangling references (because `free` was called)

Please vote via Ilias.

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

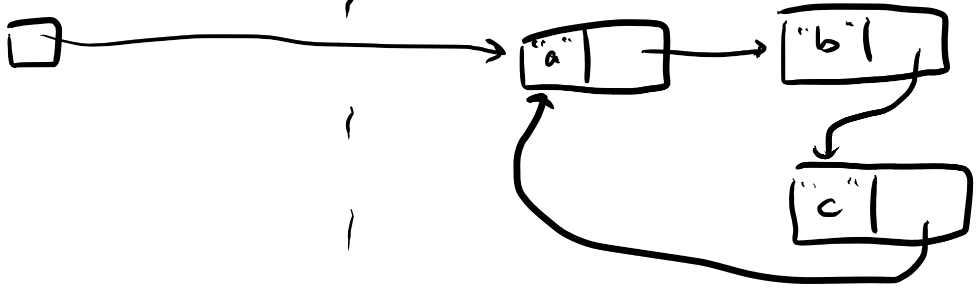
Circular Dependencies

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

Example: Circular Data structure

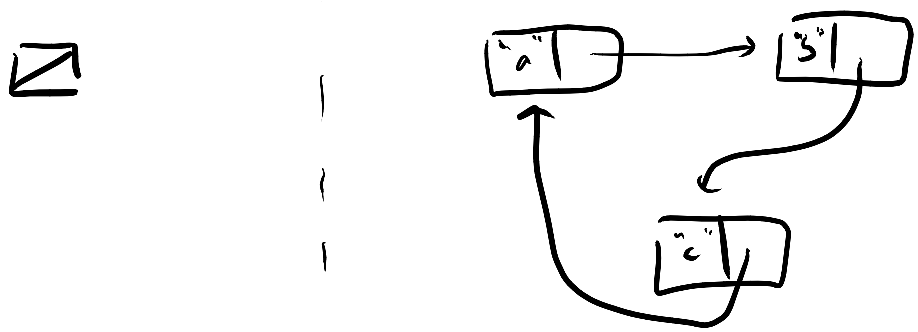
Stack | Heap

list_ptr



list_ptr = NULL

list_ptr



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 a chain of valid pointers from something that has a name to o exists

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