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

```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);
```
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);
```

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:

  \[ E_1[E_2] \text{ means } (*((E_1)+(E_2))) \]

  - For any expressions \( E_1 \) and \( E_2 \)

- E.g., \( arr[3] \) is equivalent to \( 3[arr] \)
More Pointer Arithmetic

Arithmetic operations beyond addition

- **Subtraction**
  - Get distance between two elements:
    \[ p_1 - p_2 \]
    where both are pointers to elements in the same array

- **Comparison**
  - Check if one element is at higher index than another:
    \[ p_1 > p_2 \]

- All scaled according to type of pointer
**Difference: Allocation**

**Main difference between arrays and pointers**

- **Arrays are implicitly allocated:**
  ```c
  int arr[10]; allocates space for ten ints
  ```

- **Pointers must be explicitly allocated:**
  ```c
  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 explicitly deallocated, but pointer lives on
- **Behavior of dereferencing**: Undefined
Quiz: Dangling References

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

```c
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 }
```
Quiz: Dangling References

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

```c
char *foo() {
    char *cp = malloc(sizeof(char));
    cp[0] = 'b';
    return cp;
}

int main(void) {
    char *csp = malloc(3 * sizeof(char));
    csp[0] = 'a';
    csp[1] = *foo();
    csp[2] = 'c';
    free(csp);
    printf("%c %c %c\n", csp[0], csp[1], csp[2]);
}
```

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

Stack

\[ \text{list-\_ptr} \]

Heap

\[ \text{list-\_ptr} = \text{NULL} \]

\[ \text{list-\_ptr} \]
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
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