Programming Paradigms

Lecture 11: Composite Types (Part 2)

Prof. Dr. Michael Pradel
Software Lab, University of Stuttgart
Winter 2019/2020
Wake-up Exercise

What does the following C code print?

```c
char *cptr = (char *) 0x1000;
int *iptr = (int *) 0x1000;

void *a = cptr+4;
void *b = iptr+4;

printf("%p %p\n", a, b);
```
Wake-up Exercise

What does the following C code print?

```c
char *cptr = (char*) 0x1000;
int *iptr = (int*) 0x1000;

void *a = cptr + 4;
void *b = iptr + 4;

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

**Result:** 0x1004 0x1010
Wake-up Exercise

What does the following C code print?

```c
char *cptr = (char *) 0x1000;
int *iptr = (int *) 0x1000;
void *a = cptr + 4;
void *b = iptr + 4;
printf("%p %p\n", a, b);
```

Result: 0x1004 0x1010

Two pointers initialized with hexadecimal numbers.
What does the following C code print?

```c
char *cptr = (char *) 0x1000;
int *iptr = (int *) 0x1000;
void *a = cptr +4;
void *b = iptr +4;
printf("%p %p\n", a, b);
```

Adding 4*size(t) to each pointer, where t is the type the pointer refers to.

**Result:** 0x1004 0x1010
Overview

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

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

Tuple type with fields separated by *
Example: OCaml

Conceptually:

```
R
  
X
  
Z
```

In memory:

```
Node [R]
  
Node [X]
  
Node [Z]
```

```
Node [Y]
  
Node [W]
```

Empty
Example: Tree in C

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

Pointers to objects of type

```
struct chr_tree
```
Example: C

\[ \text{\textbullet} \quad \text{means null pointer} \]
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++):**

```cpp
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):

```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):

```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):

```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):

```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

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

Example (Pascal):
my_ptr^ .val := 'X';

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 (C):

```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

```c
#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

```c
#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

```c
#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

```c
int *c;
for (int i = 0; i < 5; i += 2) {
    c = malloc(sizeof(int));
    if (i % 4 == 0) {
        free(c);
    }
}
```

https://ilias3.uni-stuttgart.de/vote/0ZT9
Quiz: Memory Leak

How many bytes of memory are leaked when executing the following code?
Assumption: ints occupy four bytes

```c
int *c;
for (int i = 0; i < 5; i += 2) {
    c = malloc(sizeof(int));
    if (i % 4 == 0) {
        free(c);
    }
}
```

Answer: 4 bytes

https://ilias3.uni-stuttgart.de/vote/0ZT9
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);
```
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:
    \[ p1 - p2 \]
    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
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
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]);
}
```
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 Structures

Stack

$\text{list- ptr}$

Heap

$\text{list- ptr = NULL}$

$\text{list- ptr}$

![Diagram showing a stack and a heap with linked list nodes labeled 'a', 'b', and 'c'.]
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
Overview

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