Programming Paradigms

Composite Types (Part 3)
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);
```
Quiz

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
Quiz

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);
```

Result: \texttt{0x1004 0x1010}

Adding \(4 \times \text{size}(t)\) to each pointer, where \(t\) is the type the pointer refers to.
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

<table>
<thead>
<tr>
<th>PLs with reference model of variables</th>
<th>PLs with value model of variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>▪ No need for explicit pointers</td>
<td>▪ Need explicit pointers to refer to objects</td>
</tr>
<tr>
<td>▪ Fields simply refer to object of same (or other) type</td>
<td>▪ Otherwise, would always copy the entire memory object</td>
</tr>
</tbody>
</table>
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:

\[ R \]

\[ X \quad Y \quad Z \quad W \]

In memory:

Node \( R \)

Node \( X \)

Node \( Z \)

Node \( Y \)

Node \( W \)

Empty
Example: Tree in C

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

Pointers to objects of type `struct chr_tree`
Example: C

□ 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):
```
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):

```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);
        // ...
        free(line);
        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);
    }
}
```

Please vote via Ilias.
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

Please vote via Ilias.