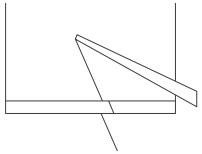
Bring ideas to life
VIA University College



C Memory Management, Pointer and Data Types

ESW1

At the end of this session, you should

- Understand memory management
- Be able to explain the purpose of a `struct` and how to declare and use it
- Understand the purpose of `typedef`
- Use structs in a Linked List example

Exercises

- Exercise 4.1 – Wait with the `my_strdup(...)`
- REMEMBER TO USE TEST DRIVEN DEVELOPMENT!

Structures (`struct`) in C

- Structures in C group related data of different types (objects in Java)
- `struct` defines a type that can be used to create variables
- Member access '.' (Like Java)

- A kind of simple classes with only public fields and without functions

```
struct student {
        int student_number;
        char* student_name;
};

int main () {
        struct student me;
        struct student you;

        me.student_number = 123;
        me.student_name = strdup("Julia");

        you.student_number = 247;
        you.student_name = strdup("John");
        …
```

Dynamic memory allocation (stdlib.h)

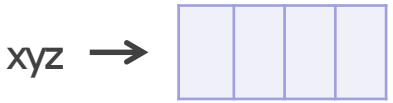
General form of memory allocation:

```
pointer = (type*) malloc (sizeof(type)); // malloc returns void*
```

```
char* p = (char*) malloc(sizeof(char) * 8);



struct my_struct {
   int x, y;
};

struct my_struct* xyz;
xyz = (struct my_struct*) malloc(sizeof(struct
my_struct));
```

Allocates an array of 8 chars in memory/heap and makes p point to the first char

Allocates a block of two integers (2 bytes each) as a struct and makes xyz point to it



Freeing Dynamic Memory

In C dynamic memory must be deallocated after use - Remember there is No Garbage Collector

```
void free(void* ptr);
```

```
struct my_struct {
   int x, y;
};

struct my_struct* xyz;
xyz = (struct my_struct*) malloc(sizeof(struct my_struct));

// Use the dynamic struct here

free(xyz);  // Dealocate the memory again after use!!
```

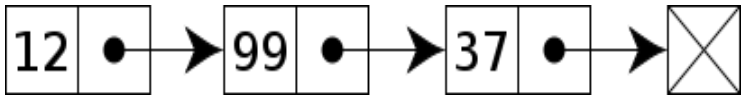
Pointers to structs

- We *rarely* transfer structs to functions – prefer pointers to structs instead (call by reference)(more efficient)
- Functions need the address (&) of a struct variable to be able to change it's elements
- Member access through pointer "->" is used when the struct is referenced by a pointer

```
int input_next_student (struct student* s)
{
   …
   …
   s->student_number = 25;
   s->student_name = strdup("Roman");
   …
   return 1; /*all went well*/
}

int main(void)
{
   struct student me;

   if (input_next_student (&me)) {
      …
   }
 …
```

Self-referential structures

- Linked lists




```
struct my_linked_list {
        int id;
        struct my_linked_list* next;
};

int main () {

  struct my_linked_list* the_list;

  the_list = (struct my_linked_list*)
        malloc(sizeof(struct my_linked_list));

  the_list->ID = 12;
  the_list->next = NULL;
```

typedef

Makes the code easier to read

```
typedef real_type defined_type
```

```
typedef int int_t; // Declare an integer type

typedef struct my_linked_list {      // Declare a struct type
        int id;
        struct my_linked_list* next;
}  my_linked_list_t;

int_t my_int;
my_linked_list_t my_list;     // Create a variable of the type
```

typedef

Without `typedefs`

```
struct my_linked_list {
        int id;
        struct my_linked_list* next;
};

int main(void) {

  struct my_linked_list* the_list;
  the_list = (struct my_linked_list *) malloc(sizeof(struct my_linked_list));
  the_list->id= 12;
  the_list->next = NULL;
```

With `typedefs`

```
typedef struct my_linked_list* my_linked_list_ptr_t;

typedef struct my_linked_list {
  int id;
  my_linked_list_ptr_t next;
} my_linked_list_t;

int main(void) {
  my_linked_list_ptr_t the_list;
  the_list = malloc(sizeof(my_linked_list_t));
  the_list->id= 12;
  the_list->next = NULL;
```

Exercises

- Design and implement the my_strdup(...) from Exercise 4.1
- Exercise 4.2

- REMEMBER TO USE TEST DRIVEN DEVELOPMENT!