PMC Lecture 03

Gianmaria Romano

1 October 2025

Appendix A

The C programming language

A.1 Pointers

A pointer is a particular variable that contains the memory address to another variable of the same type, eventually allowing to retrieve its value as well through dedicated commands.
Example: The following code, after initializing two pointers and two integers, sets p_1 to point towards a's memory address and p_2 to point towards b's memory address:

```c
#include <stdio.h>

int main() {
    int *p1, *p2; // Initialize pointers p1 and p2.

    int a = 6;
    int b = 3;

    p1 = &a; // p1 now points at the address where a is stored.
    p2 = &b; // p2 now points at the address where b is stored.

    return 0;
}
```

A.1.1 The void pointer

Commonly used for memory allocation, the void pointer is a special pointer that can be used to specify that a function can use any type of pointer, therefore allowing to easily convert pointer types.

A.2 Memory allocation

Memory in C consists of a static memory, which is used by the stack for executing a function, and a dynamic memory, which is used by the heap to handle

dynamic variables whose size may not be known at runtime.
In particular, dynamic memory allocation is typically performed using the malloc() function, which reserves a memory buffer and provides a void pointer towards the address of the allocated space, although, in some cases, it might be more convenient to use functions such as realloc(), which tries to modify the size of the allocated buffer, or calloc(), which is used to reserve a memory buffer for a collection, such as an array.
On the other hand, it is possible to free up memory space by calling the free() function on a pointer towards an allocated buffer.
N.B.: Whenever there is not enough space to perform memory allocation, the allocation functions will return a NULL pointer to state that there was an error during the allocation.
Generally speaking, it is possible to see how much memory is needed to store a certain variable using the sizeof() function.
Example: The following code tries to allocate an integer in the memory space:

```c
#include <stdio.h>
#include <stdlib.h>

void func1(int *ptr);

int main() {
    int *p = malloc(sizeof(int));
    if (p == NULL) {
        printf("It is not possible to allocate space! \n");
        return 0;
    }

    *p = 30;
    printf("p = %d \n", *p); // Print 30.

    func1(p); // Modify the variable indicated by p.

    printf("p = %d \n", *p); // Print 13.

    return 0;
}

void func1(int *ptr) {
    *ptr = 13;
}
```

In particular, if the NULL pointer is returned, meaning that there is not enough space for the allocation, the code will return an error message.
Otherwise, it will change the value of the variable indicated by the pointer and call the func1() function to modify it.

A.3 Structured data

Declared using the "struct" keyword, a structure contains variables of different types that can be accessed using the .field command on a single instance.
Example: The following code defines a structure for articles that contains information about their code and price:

```c
#include <stdio.h>

struct Article {
    int articleID;
    float price;
}; // The semicolon is used because this is a declaration.

int main() {
    struct Article a1 = {467, 12.5}; // Direct initialization.

    struct Article a2; // Simple declaration.
    a2.articleID = 189; // Assignment by access.
    a2.price = 34.99;

    printf("a1 - Code: %d, Price: %0.2f \n", a1.articleID, a1.price);
    printf("a2 - Code: %d, Price: %0.2f \n", a2.articleID, a2.price);

    return 0;
}
```

A.3.1 Pointers to structured data

Similarly for variables, it is possible to assign pointers to a struct and give it a memory address, which can also be used to efficiently access its data.

A.3.2 Changing names using the typedef command

Generally speaking, it is possible to declare structured data more efficiently by exploiting an alias that is created using the "typedef" keyword.
Example: The following code defines a structure for books, which is then invoked at runtime using an alias:

```c
#include <stdio.h>

typedef struct Book {
    int pages;
    char title[50];
    float price;
} book; // This alias introduces a new type "book" representing Book structures.

int main() {
    book book1 = {354, "Fondamenti di Programmazione", 42.50};

    return 0;
}
```

A.3.3 Union structures

The "union" keyword allows to create a structure containing variables of different types that are all located at the same memory address, although, at any given time, only one of these variables can contain a value.
For this reason, a union structure always requires a memory space that is large enough to store a value for its biggest variable.
Example: The following code creates a union containing an integer and a character:

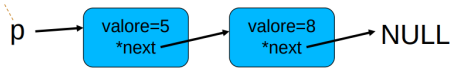
```c
#include <stdio.h>
#include <stadlib.h>

union MyData {
    int i;
    char c;
} mydata;

// An int requires 4 bytes of memory space while a char only requires 1 byte.
// MyData therefore requires 4 bytes to contain all its variables.

int main() {
    printf("Allocated memory: %d\n", sizeof(mydata));
    return 0;
}
```

Since integers take up more memory space than characters, the union will need a memory space of the same size as an integer to store its possible value.

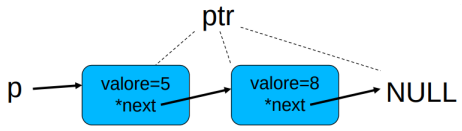
A.4 Linked lists

A limitation of arrays lies in the fact that they are static, meaning that each instance has a fixed size and stores its element contiguously in the memory.
On the other hand, a linked list is a dynamic collection, meaning that its size can change over time and, generally speaking, its elements are not stored contiguously in the memory.
For this reason, the elements of a list always feature a pointer towards the next element, with the exception of the last element, which instead uses a NULL pointer to indicate the end of the list.
Most particularly, list pointers come in handy to perform the following operations:

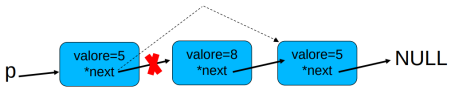
- **Inserting elements:** assuming for simplicity that the value is added at the end of the list, the last node has to point towards the new element, which instead will now feature the NULL pointer to represent the end of the list.



- **Printing elements:** a list can be printed by declaring a new pointer and moving it across the array to print its elements one by one.



- **Deleting elements:** if the first element has to be deleted, it is sufficient to set the first pointer towards the second element.
  From a more general point of view, deleting an element in the list requires moving up until the preceding element and updating its pointer to the element after the node to delete, which ends up having no references in the list.



N.B.: The pointer towards the first element of the list should never be modified as it allows to access the entire list.