Recap : Memory Allocation, Pointers, Structs, Typedefs

K08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Κώστας Χατζηκοκολάκης

Computer memory

- Computers have memory, a device that allows storing and retrieving data
- Each **byte** of memory has an **address**
  - unique number associated with this byte
- Programs need to **allocate**, **deallocate**, **read** and **write** on memory
- In C the programmer has **direct access** to the memory
  - the only complicated part, in an otherwise very simplistic language

Allocating memory

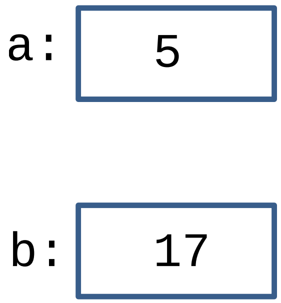
C programs allocate memory in 2 ways:

- **Automatic**
  - by declaring variables
- **Manual**
  - by calling `malloc`

Allocating memory via variables

- Space for two `int`s is allocated the moment `foo` is **called**
- The values `5`, `17` are copied in the allocated memory

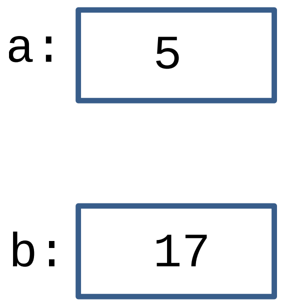
```
void foo() {
    int a = 5;
    int b = 17;
}

int main() {
    foo();
}
```



Parameters

- Parameters are essentially just local variables
- Only difference: the **argument** provided by the caller is copied

```
void foo(int a) {
    int b = 17;
}

int main() {
    foo(5);
}
```



Address-of operator `&`

- To see where a variable is stored, use the **address-of** operator `&`
- We can print it in hex

```
printf("Memory address of a in hex: %p \n", &a);
```

```
void foo(int a) {
    int b = 17;

    printf("foo &a: %p\n", &a);
    printf("foo &b: %p\n", &b);
}
```

Deallocating a variable's memory

- A variable's memory is **deallocated** when the function call **returns**
- Deallocation simply means that such memory can be given to some **other variable**

```
void foo() {
    int a = 5;
    printf("foo &a: %p\n", &a);
}

void bar() {
    int a = 17;
    printf("bar &a: %p\n", &a);
}

int main() {
    foo();
    bar();
}
```

Deallocating a variable's memory

- Here, `foo` has **not returned** yet when `bar` is called
- Will we get the same result?

```c
void bar() {
    int a = 17;
    printf("bar &a: %p\n", &a);
}

void foo() {
    int a = 5;
    printf("foo &a: %p\n", &a);
    bar();
}

int main() {
    foo();
}
```

Global variables

They remain allocated until the program finishes

```c
int global = 5;

void foo() {
    printf("foo  &global: %p\n", &global);
}

int main() {
    printf("main &global: %p\n", &global);
    foo();
    printf("main &global: %p\n", &global);
}
```

Pointers

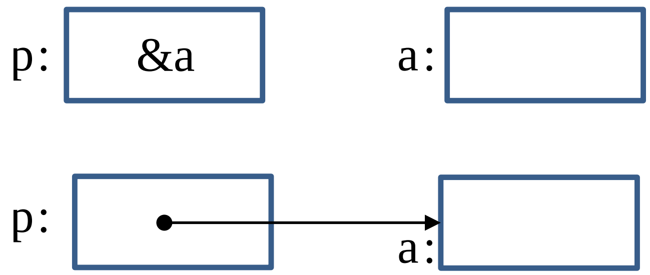
- Pointers are just variables, nothing special
- They are allocated/deallocated the same way as all variables are
  - Their **content** has **nothing** to do with allocation/deallocation

```
void foo() {
    int* p;
}
```

Pointer content

- In a pointer we store **memory addresses**, e.g. the address of a variable
- Nothing special happens; we just store a **number** in a variable
  - we just think of `p` as “pointing to” `a`

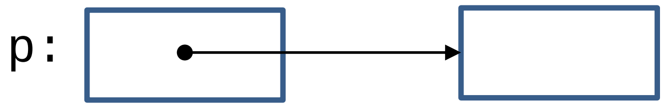
```
void foo() {
    int a;
    int* p = &a;

    printf("&a: %p\n", &a);
    printf("&p: %p\n", &p);
    printf(" p: %p\n",  p);
}
```



Manual allocation

- Done by calling `malloc(size)` (actually easier to understand)
- Returns the **address** of the allocated memory
  - we need to **store** such address (in a pointer)

```
int* p = malloc(sizeof(int));
```

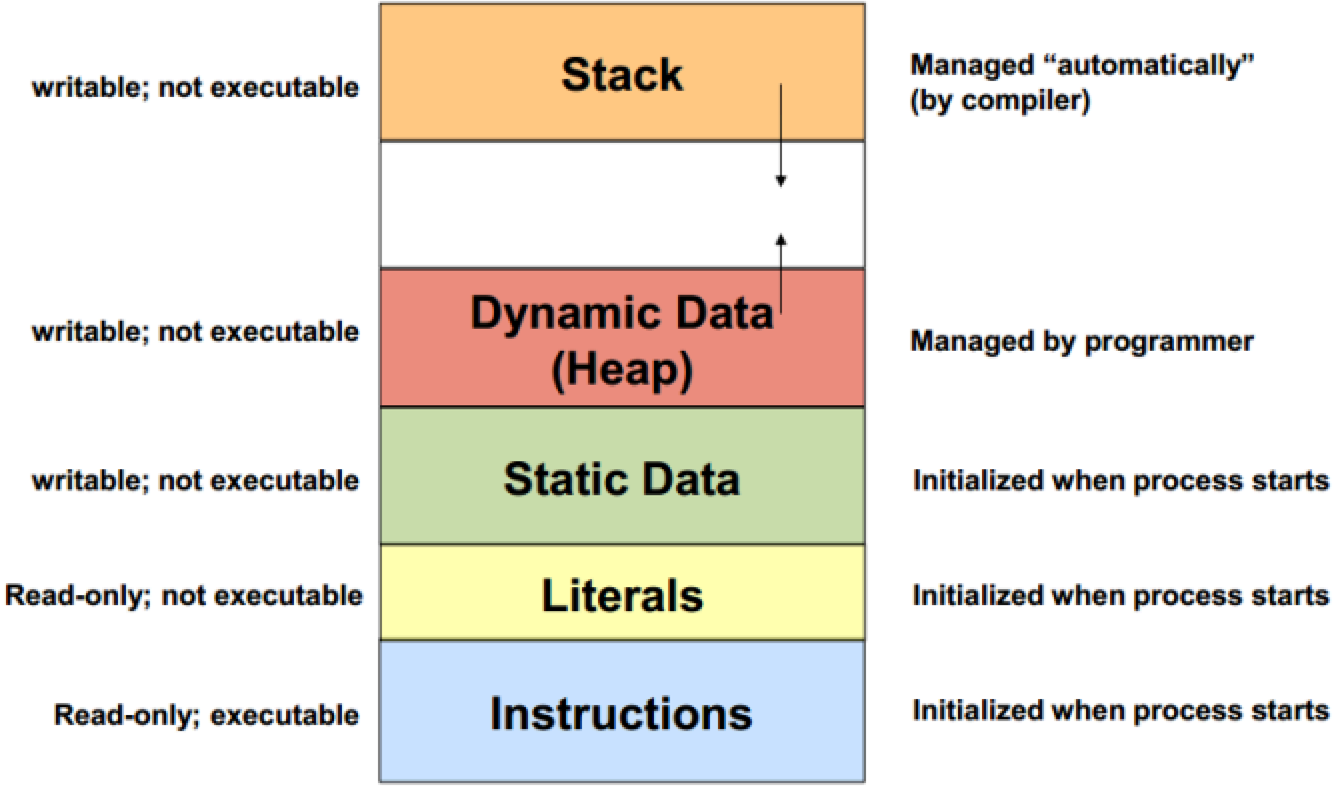


Manual allocation

- The allocated memory is **not** the address of **any** variable
- In fact, the allocated memory is “far” from all variables
  - variables are allocated in the **stack**
  - `malloc` allocates memory in the **heap**
  - just fancy names for two different areas of memory

```c
int* a = malloc(sizeof(int));

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

Program Memory



Manual deallocation

- Call `free(address)`, for some `address` previously returned by `malloc`
  - typically stored in a pointer
- `free`d memory can be reused (this is what "deallocated" means)

```
int* p1 = malloc(sizeof(int));
int* p2 = malloc(sizeof(int));

free(p2);
int* p3 = malloc(sizeof(int));

printf("p1: %p\n", p1);
printf("p2: %p\n", p2);
printf("p3: %p\n", p3);
```

Remember

1. C never looks at the **content** of a variable when deallocating its memory

```c
void foo() {
    int* p = malloc(sizeof(int));
}
```

2. `free` does not modify the content of any variable

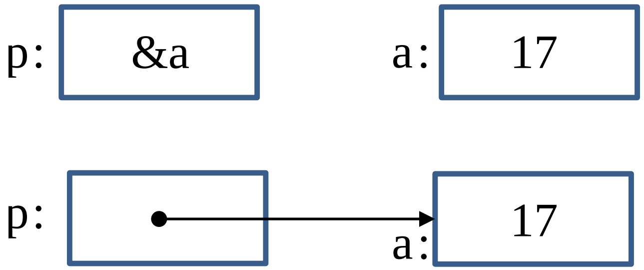
```c
int* p = malloc(sizeof(int));

printf("p: %p\n", p);
free(p);
printf("p: %p\n", p);

p = NULL;       // καλή πρακτική μετά το free
```

Accessing memory via pointers, operator `*`

When reading or writing to a variable:

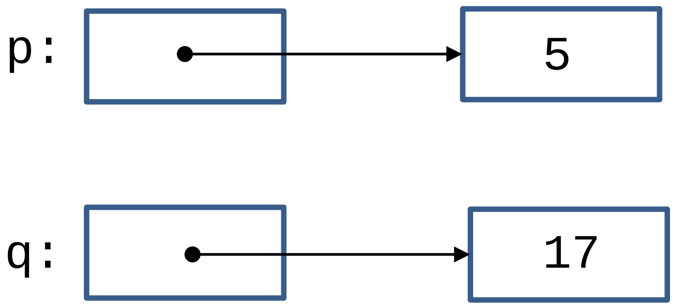
- `a`, `p`, read/write to the memory **allocated** for `a`, `p`
- `*p` reads/writes to the memory **stored** in `p`

```
int a;
int* p;

 p = &a;    // στη μνήμη που δεσμεύτηκε για το p, γράψε τον αριθμό &a
*p = 16;    // στη μνήμη που περιέχει το p (δηλαδή στην &a), γράψε το

 a = *p + 1;
```

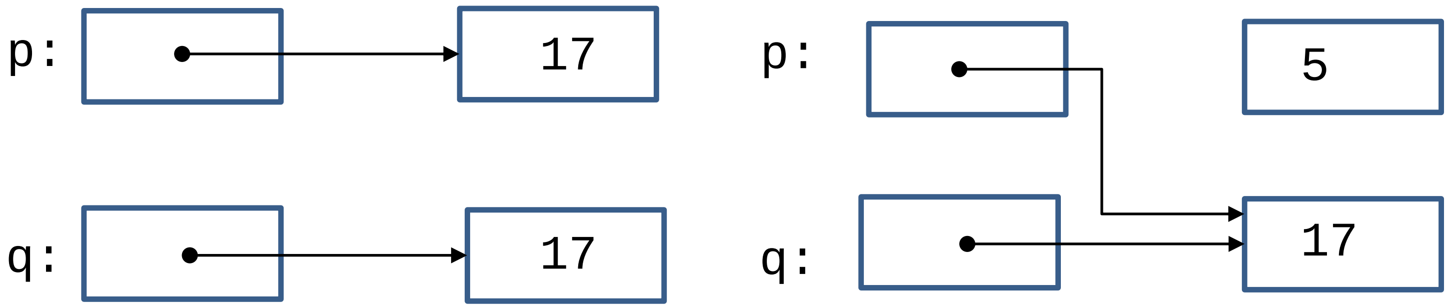


Pointer quiz

- We have this situation:

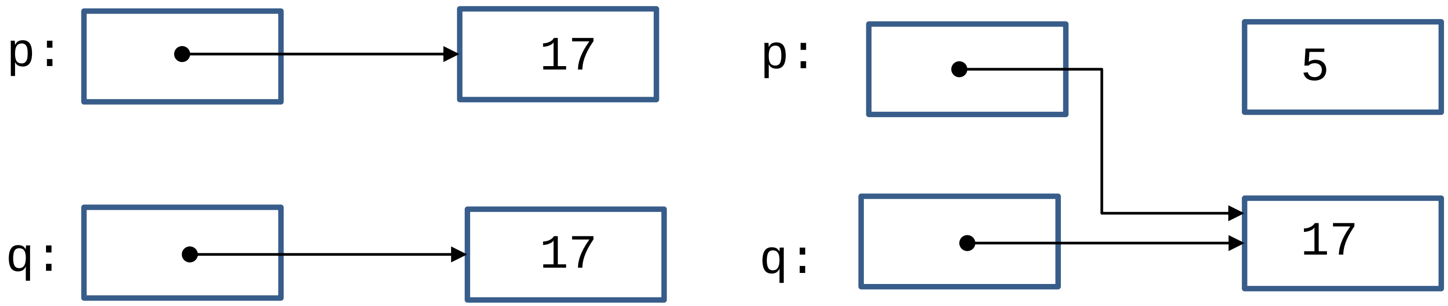


- Which commands produce each of the following?



Pointer quiz

- Which commands produce each of the following?




```
*p = *q;    // αριστερό
 p =  q;    // δεξί
```

Pointers as function arguments

- Nothing special happens at all
  - We just receive a **number** as an argument
- This is very useful for accessing memory outside the function

```
void foo(int a, int* p) {
    *p = a;
}

int main() {
    int a = 1;
    foo(52, &a);
}
```

Swap

Will this work?

```
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int a = 1;
    int b = 5;
    swap(a, b);
}
```

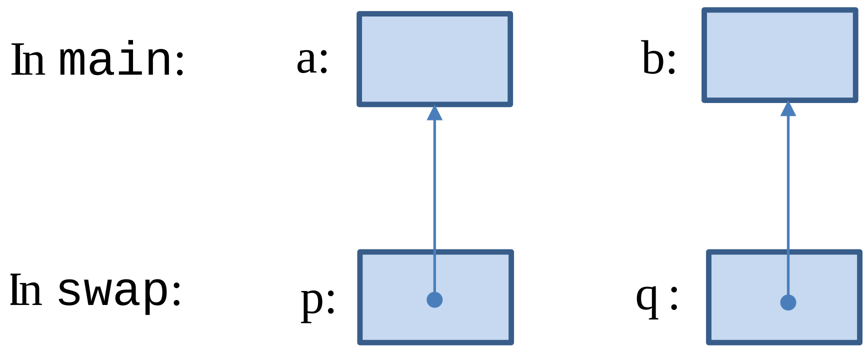
Swap

Will this work?

```
void swap(int* p, int* q) {
    int* temp = p;
    p = q;
    q = temp;
}

int main() {
    int a = 1;
    int b = 5;
    swap(&a, &b);
}
```

Swap, correct

```
void swap(int* p, int* q) {
    int temp = *p;
    *p = *q;
    *q = temp;
}

int main() {
    int a = 1;
    int b = 5;
    swap(&a, &b);
}
```



Returning pointers

- Again, nothing special happens, we just **return a number**

```c
int* foo() {
    int* p = malloc(sizeof(int));
    *p = 42;
    return p;
}

int main() {
    int* p = foo();
    printf("content of p: %d\n", *p);
    free(p);
}
```

Dangling Pointers

- A pointer `p` containing **deallocated** memory is dangerous!
  - it's not our memory anymore
  - using `*p` has **undefined** behaviour (typically it makes your PC explode)

- Think about deallocation rules **before returning a pointer**

```
int* foo() {
    int a = 63;
    int* p = &a;
    return p;       // πού δείχνει ο p;
}

int* foo() {
    int* p = malloc(sizeof(int));
    *p = 42;
    free(p);
    return p;       // πού δείχνει ο p;
}
```

Structs

- A simple way of storing several pieces of data together
- Useful for creating custom **types**
- A struct has **members**, each member has a **name**

```
struct point_2d {                       // ένα σημείο στον δισδιάστατο χώρο
    float x;
    float y;
};

int main() {
    struct point_2d point;              // μία μεταβλητή!
    point.x = 1.2;                      // έχει αρκετό χώρο
    point.y = 0.4;                      // για 2 floats
}
```

Structs, allocation

- Nothing special, just like any other type

```c
void foo() {
    // θα αποδεσμευθεί στο τέλος της κλήσης της foo
    struct point_2d point;

    // θα αποδεσμευθεί όταν κάνουμε free
    struct point_2d* p = malloc(sizeof(struct point_2d));
}
```

Structs, pointers

- When `p` is a **pointer to a struct**:
  - `p->member` is just a **synonym** for `(*p).member`

```
void foo(struct point_2d* p) {
   (*p).x = -1.2;
   p->y = 0.4;

    // Μπορούμε να αντιγράφουμε και ολόκληρο το struct!
    struct point_2d point = *p;
    point.x = point.y * 2;
    *p = point;
}
```

typedef

- Simply gives a **new name** to an existing type

```
typedef int Intetzer;       // English style
typedef int Integker;       // Γκρικ στάιλ

int main() {
    Intetzer a = 1;
    Integker b = 2;
    a = b;                  // και τα δύο είναι απλά ints
}
```

typedef, common uses

Simplify structs

```
struct point_2d {
    float x;
    float y;
};
typedef struct point_2d Point2d;

int main() {
    Point2d point;  // δε χρειάζεται το "struct point_2d"
}
```

Even simpler:

```
typedef struct {
    float x;
    float y;
} Point2d;
```

typedef, common uses

“Hide” pointers

```c
// list.h
struct list {
    ...
};
typedef struct list* List;

List list_create();
void list_destroy(List list);
```

```c
// main.c
#include "list.h"

int main() {
    List list = list_create();       // ποιος "pointer";
    list_destroy(list);
}
```

Function pointers

- Receive a **function** as argument
- A `typedef` is **highly** recommended

```
// Για μια συνάρτηση σαν αυτή
int foo(int a) {
    ...
}

// Δηλώνουμε τον τύπο ως εξής (το foo αλλάζει σε (*TypeName))
typedef int (*MyFunc)(int a);

int main() {
    // Και μετά μπορούμε να αποθηκεύουμε το "foo" σε μια μεταβλητή f
    MyFunc f = foo;
    f(40);      // το ίδιο με foo(40)
}
```

Function pointers

```c
typedef int (*MyFunc)(int a);

int foo1(int a) {
    return a + 1;
}
int foo2(int a) {
    return 2*a;
}

int bar(MyFunc f) {
    printf("f(0) = %d\n", f(0));
}

int main() {
    bar(foo1);
    bar(foo2);
}
```

Void pointers

- All pointers are just numbers!
- A variable with type `void*` can store **any pointer**

```
int* int_p;
float* float_p;
Point2d* point_p;
MyFunc func_p;

void* p;

p = int_p;
p = float_p;
p = point_p;
p = func_p;

int_p = p;
float_p = p;
point_p = p;
func_p = p;
```

Generic functions

- `void*` allows to define operations on data of **any type**

```
// Ανταλλάσσει τα περιεχόμενα των p,q, μεγέθους size το καθένα
void swap(void* p, void* q, int size) {
    void* temp = malloc(size);   // allocate size bytes
    memcpy(temp, p, size);       // αντιγραφή size bytes από το p στο
    memcpy(p, q, size);
    memcpy(q, temp, size);
    free(temp);
}

int main() {
    int a = 1;
    int b = 5;
    swap(&a, &b, sizeof(int));

    float c = 4.3;
    float d = 1.2;
    swap(&c, &d, sizeof(float));
}
```

Generic functions

Combine with **function pointers** for full power!

```
typedef void* Pointer;                    // απλούστερο

// Δείκτης σε συνάρτηση που συγκρίνει 2 στοιχεία a και b και επιστρέφ
// < 0  αν a <  b
//   0  αν a == b
// > 0  αν a >  b

typedef int (*CompareFunc)(Pointer a, Pointer b);

Pointer max(Pointer a, Pointer b, CompareFunc comp) {
    if(comp(a, b) > 0)
        return a;
    else
        return b;
}
```

Generic functions

```
#include <string.h>

int compare_ints(Pointer a, Pointer b) {
    int* ia = a;
    int* ib = b;
    return *ia - *ib;
}

int compare_strings(Pointer a, Pointer b) {
    return strcmp(a, b);
}

int main() {
    int a1 = 1;
    int a2 = 5;
    int* max_a = max(&a1, &a2, compare_ints);

    char* s1 = "zzz";
    char* s2 = "aaa";
    char* max_s = max(s1, s2, compare_strings);

    printf("max of a1,a2: %d\n", *max_a);
    printf("max of s1,s2: %s\n",  max_s);
}
```

Readings

- Π. Σταματόπουλος, *Σημειώσεις Εισαγωγής στον Προγραμματισμό*.