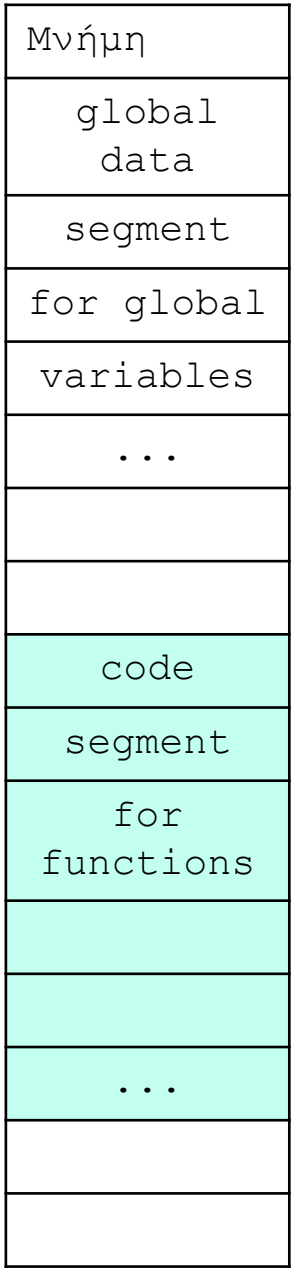
# HY255 Εργαστήριο Λογισμικού - L08

## Dynamic Memory
## (Στόχος και Χρήση)

HY255 Εργαστήριο Λογισμικού
Άνοιξη 2021
Άγγελος Μπίλας

# Μνήμη προγράμματος (έως τώρα)

- Ποια είναι η εικόνα της μνήμης ενός προγράμματος όταν αρχίζει να εκτελείται;
  - Global data segment
  - Code segment
- Και τα δύο αυτά τμήματα έχουν συγκεκριμένο και γνωστό μέγεθος όταν γίνεται compile το πρόγραμμα
  - Οι τύποι καθορίζουν το μέγεθος κάθε μεταβλητής
  - Ο κώδικας είναι συγκεκριμένος και δεν αλλάζει
  - Το μέγεθος αυτό δεν αλλάζει κατά την εκτέλεση
- Οπότε, κατά την εκτέλεση το προγράμματος και τα δύο αυτά segments είναι σταθερά σε μέγεθος, δεν προστίθεται και δεν αφαιρείται τίποτε, μόνο αλλάζουν τα περιεχόμενα των μεταβλητών
- Πότε ελευθερώνεται αυτή η μνήμη; Απλά όταν τερματίσει το πρόγραμμα.
- Άρα μέχρι τώρα η εικόνα της μνήμης του προγράμματος είναι ιδιαίτερα απλή και η απλότητα είναι καλή, όταν μας εξυπηρετεί…



# Μπορούμε να γράψουμε όλα τα προγράμματα έτσι;

- Ναι, αλλά δεν θα είναι πολύ αποτελεσματικά (κατά κάποια έννοια κόστους)
- Ας σκεφτούμε ένα πρόγραμμα που πρέπει να πολλαπλασιάσει δύο πίνακες που διαβάζει από το stdin
- Αλλά δεν γνωρίζει το μέγεθος των πινάκων που του δίνουμε κάθε φορά
- Πως μπορεί να γραφτεί ένα τέτοιο πρόγραμμα; Γίνεται να δεσμεύσει τον χώρο που χρειάζεται μόνο με global (ή και local) μεταβλητές;
- Αν το σκεφτούμε αυτό, θα καταλήξουμε γρήγορα ότι δεν αρκούν οι global/local μεταβλητές
  - Αν δηλώσουμε δύο arrays συγκεκριμένου μεγέθους (είτε μικρά είτε μεγάλα) δεν θα είναι ποτέ ακριβώς αυτό που θέλουμε
  - Δεν μπορούμε να δηλώσουμε δυναμικά νέες μεταβλητές, κ.ο.κ.
- Ένα πρόγραμμα πρέπει λοιπόν να έχει την δυνατότητα να πάρει μνήμη αφού έχει ξεκινήσει την εκτέλεσή του
- Αυτή η δυνατότητα ονομάζεται δυναμική μνήμη
- Για να έχουμε αυτή τη δυνατότητα χρειαζόμαστε δύο μηχανισμούς
  - (a) Να μπορούμε να πάρουμε μνήμη από το σύστημα → malloc/free
  - (b) Να μπορούμε να ονομάσουμε τηη μνήμη → pointers

# malloc/free

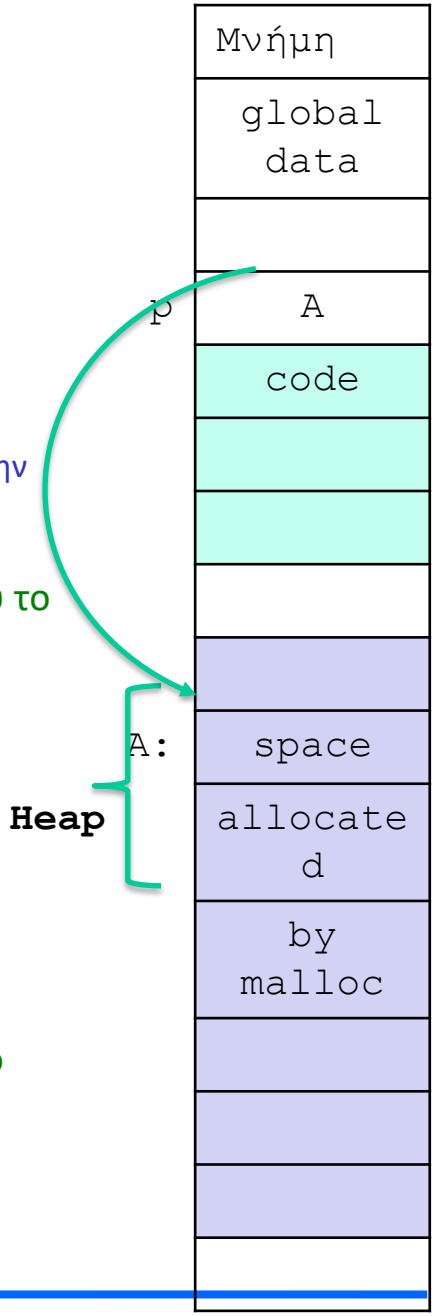
- Χρήση malloc/free, e.g.

```
char *p;
p = malloc (size);
free(p);
```

- Η malloc δεσμεύει και επιστρέφει ένα block μνήμης, μεγέθους τουλάχιστον size bytes
  - Δεν γνωρίζουμε που βρίσκεται το block αυτό
  - Η μνήμη που επιστρέφει η malloc είναι συνεχόμενη
  - Αν η malloc δεν βρει μνήμη, επιστρέφει NULL
- Η free επιστρέφει την μνήμη που δείχνει το p
  - Το p πρέπει αναγκαστικά να είναι μια τιμή που μας έδωσε η malloc
  - Π.χ. στο παραπάνω παράδειγμα δεν μπορούμε να πούμε free(p+2) και να ελευθερώσουμε την μνήμη μετά τα 2 πρώτα bytes
  - Μετά το free απαγορεύεται να χρησιμοποιήσουμε την μνήμη (που δείχνει το p)
- Όλη η μνήμη που χειρίζεται η malloc/free βρίσκεται σε ένα νέο τμήμα της μνήμης που το ονομάζουμε heap
- Πάντα ελέγχουμε την τιμή επιστροφής της malloc για null

```
p = malloc (size);
if (p==NULL) {
        printf("error - out of memory\n");
        ...
}
```

- Το module της C για δυναμική μνήμη διαθέτει και μερικές ακόμη συναρτήσεις
  - calloc
  - realloc
- Η malloc/free είναι ουσιαστικά ένα module που μας δίνει η C στην standard lib και το οποίο υλοποιεί/χειρίζεται το ένα heap του κάθε προγράμματος για να παρέχει δυναμικά μνήμη στο πρόγραμμα



# Χρήση των pointers σαν ονόματα για δυναμική μνήμη

- Μια μεταβλητή τύπου pointer έχει σαν περιεχόμενα μια διεύθυνση (εξ' ορισμού) που πρέπει να δείχνει σε ένα "σωστό" κομμάτι μνήμης, που δηλαδή ανήκει στο πρόγραμμα
- Τέτοια κομμάτια μνήμης μέχρι τώρα είναι για μας οι global μεταβλητές (και ο κώδικας αλλά ας τον ξεχάσουμε για τώρα)
- Άρα μέχρι τώρα οι pointers μπορούν να δείχνουν μόνο σε άλλες μεταβλητές του προγράμματός μας
  - Σε αυτή την χρήση των pointers μπορούμε να δείξουμε σε τμήματα της μνήμης που μπορούμε να ονομάσουμε και μέσω άλλων μεταβλητών
- Με την δυναμική μνήμη οι pointers έχουν μια ακόμη χρήση: Είναι ο μόνος τρόπος να χρησιμοποιήσουμε την μνήμη που παίρνουμε από το σύστημα κατά την εκτέλεση του προγράμματος

# Παράδειγμα στη χρήση δυναμικής μνήμης

- Πως χρησιμοποιούμε τους pointers σε linked lists
- Πως δηλώνουμε έναν κόμβο της λίστας

```
struct node {
    int value; /* value stored in node */
    struct node *next;
}
struct node *head = NULL;
```

- (Παρένθεση: named structs)
  - Είδαμε ότι μια μεταβλητή s τύπου struct δηλώνεται σαν
  - struct {int v;} s;
  - Τι γίνεται αν το struct πρέπει να περιέχει έναν pointer σε struct ίδιου τύπου;

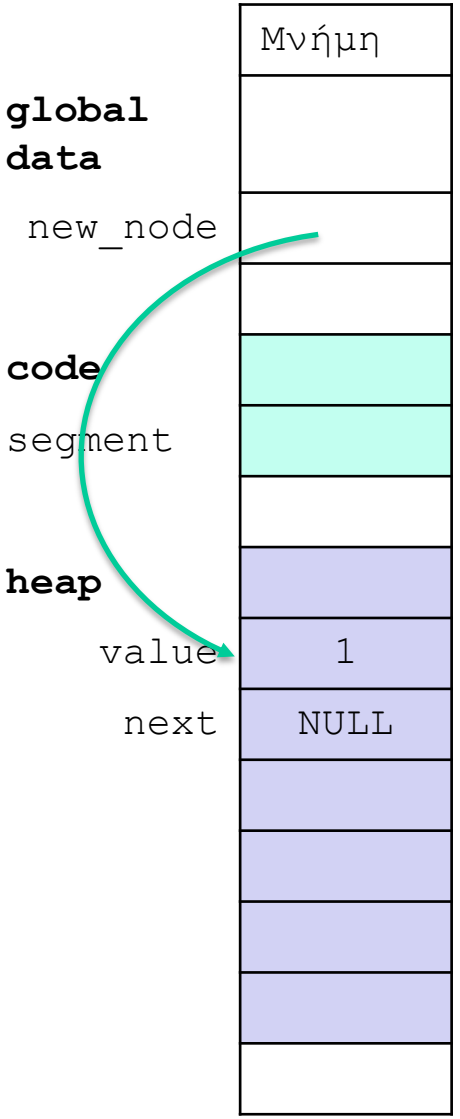
```
struct {
        int v;
        struct ... *next;
} s;
```

  - Οι … θα πρέπει να είναι συντακτικά οι ίδιες {}, αλλά αυτό είναι αναδρομικός ορισμός
  - Για να λυθεί αυτό το θεματάκι, η C επιτρέπει τα structs να έχουν ονόματα που μπορούμε να τα χρησιμοποιήσουμε για να αναφερθούμε στο struct. Οπότε μπορούμε να γράψουμε:

```
struct nodeS {
        int v;
        struct nodeS *next;
} s;
```

# Allocate έναν κόμβο της λίστας

```
struct node *new_node = malloc (sizeof (struct node));
if (new_node == NULL) {
      printf("error – out of memory\n");
      ...
}
(*new_node).value = 1;
(*new_node).next = NULL;
or (syntactic sugar)
new_node->value = 1;
new_node->next = NULL;
```

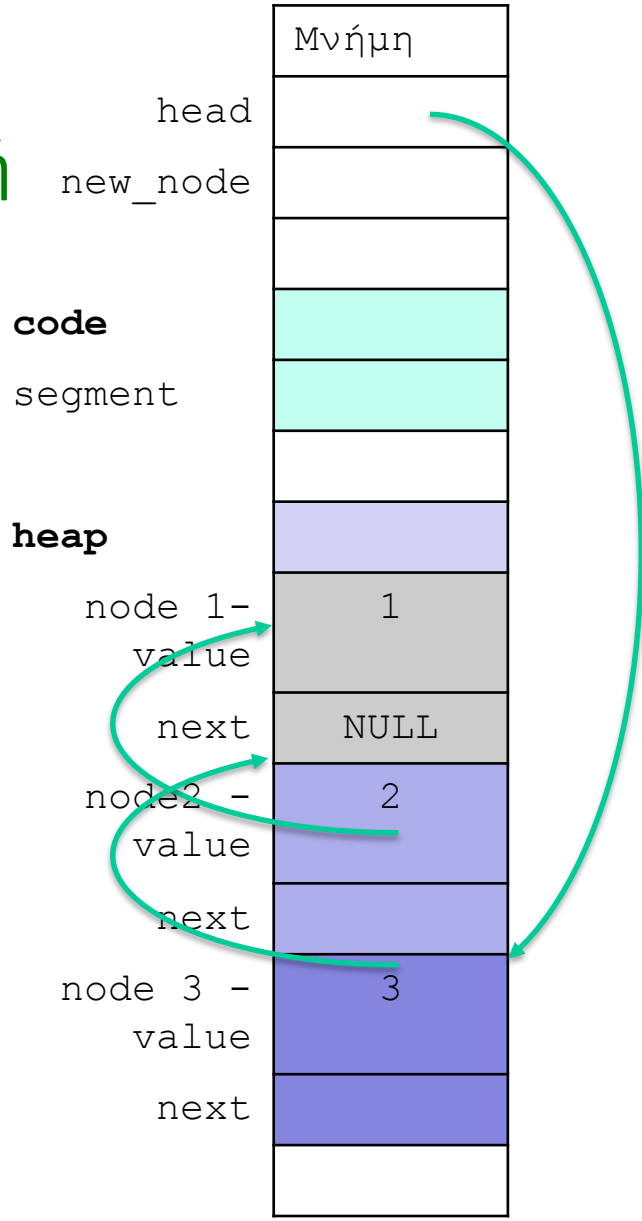


# Insert ένα στοιχείο στην αρχή της λίστας

- Έστω η μεταβλητή head που δείχνει στην αρχή της λίστας

```
struct node *head = NULL;
new_node->next = head;
head = new_node;
```

- Αν εισάγουμε 2 ακόμη κόμβους θα έχουμε τελικά το διπλανό σχήμα



# Συνάρτηση για insert

- Έστω τώρα ότι θέλουμε να γράψουμε μια συνάρτηση που να κάνει το insert

```
void insert(struct node *list, int value){
    struct node *n;
    n = (struct node *) malloc(sizeof(struct node));
    if (n==NULL) {...}
    n->value = value;
    n->next = list;
→   list = n;
}
```

  - και να την καλέσουμε με μια λίστα struct node *L, π.χ. σαν insert(L,1);
- Τι θα κάνει αυτή η συνάρτηση όταν εκτελεστεί και ειδικά η εντολή →;
  - Θα αλλάξει μια τοπική μεταβλητή της συνάρτησης (την παράμετρο), αλλά ΔΕΝ θα αλλάξει την μεταβλητή L που πέρασε αυτός που την κάλεσε
- Για να το πετύχουμε αυτό χρειάζεται λίγο περισσότερη προσοχή, π.χ.

```
void insert(struct node **list, int value){
  struct node *n;
  malloc(sizeof(...));
  if (n==NULL) {...}
  n->value = value;
→ n->next = *list;
  *list = n;
}
```

με κλήση: `insert(&L, v);`

OR

```
struct node *insert(struct node *list, int value){
  struct node *n;
  n = (struct node *) malloc(sizeof(...));
  if (n==NULL) {...}
  n->value = value;
  n->next = list;
  list = n;
  return list;
}
```

με κλήση: `L = insert(L, v);`

# Search/delete an element

- Το search είναι σχετικά απλό:

```
struct node *search(struct node *list, value v){
        struct node *p;
        for(p=list; p!=NULL; p=p->next){
                if (p->value == v) return p;
        }
        return NULL;
}
```

- Το delete θέλει λίγο περισσότερο προσοχή ώστε
  - Να κρατήσουμε τον σωστό pointer στο στοιχείο που θέλουμε να αφαιρέσουμε και να τον προσαρμόσουμε στο επόμενο στοιχείο της λίστας
  - Να ελευθερώσουμε την μνήμη του κόμβου στο σωστό σημείο του κώδικα ώστε να μην χρειαστεί να διαβάσουμε τα περιεχόμενα του κόμβου μετά το free
  - Να φροντίσουμε την περίπτωση που αφαιρούμε το πρώτο στοιχείο της λίστας

```
struct node *delete(struct node *list, value v){
        struct node *p, *head=list, *prev=NULL;
        for(p=head; p!=NULL; p=p->next){
                if (p->value == v) {
                        if (!prev) head = p->next;
                        else prev->next = p->next;
                        free(p);
                        return head;
                }
                prev = p;
        }
        return head;
}
```

- Βλέπουμε ότι ο κώδικας με pointers γρήγορα γίνεται περίπλοκος και θέλει προσοχή για το που δείχνουν οι pointers και πως τους χρησιμοποιούμε

# Interfaces και ”συμβόλαια” δυναμικής μνήμης

- Η εισαγωγή της δυναμικής μνήμης δημιουργεί σημαντική πολυπλοκότητα στα interfaces των modules
- Πλέον κάθε module πρέπει να καθορίζει στο interface του ποιανού ευθύνη είναι να δεσμεύει και να ελευθερώνει μνήμη για τα αντικείμενα που είναι γνωστά εκτός του module
  - Ένα παράδειγμα αυτού θα δούμε στην 3η άσκησή μας
- Στην πραγματικότητα και ειδικά σε critical συστήματα, η χρήση της δυναμικής μνήμης δημιουργεί
  - Πολυπλοκότητα στον σχεδιασμό και στην υλοποίηση – σίγουρα θα το δείτε στις ασκήσεις
  - Μεγάλη αβεβαιότητα – τι γίνεται αν κάποια στιγμή κατά την εκτέλεση δεν βρούμε την μνήμη που ζητήσαμε;
  - Προβλήματα απόδοσης – θα μιλήσουμε αργότερα
- Η δυναμική μνήμη είναι ένα από τα (λίγα) στοιχεία που κάνουν τον σχεδιασμό (μεγάλων) συστημάτων ιδιαίτερη πρόκληση! Άλλα στοιχεία είναι:
  - Η χρήση των pointers/layout των δεδομένων στη μνήμη
  - Ο μεγάλος αριθμός από interfaces
  - Ο παραλληλισμός/concurrency
  - Τα πρώτα τρία θα τα αντιμετωπίσουμε στο HY255 (αλλά όχι τον παραλληλισμό/concurrency)

# Reading

- King Ch. 17