C: Από τη Θεωρία στην Εφαρμογή

Κεφάλαιο 14ο

Διαχείριση Μνήμης και Δομές Δεδομένων

Γ. Σ. Τσελίκης - Ν. Δ. Τσελίκας

Διαχείριση Μνήμης

- Η διαχείριση της μνήμης αφορά κυρίως τις διαδικασίες **δέσμευσης** και **αποδέσμευσης** μνήμης
- Η δέσμευση μνήμης μπορεί να γίνει με δύο τρόπους:
  - είτε **στατικά**
  - είτε **δυναμικά**
- Στα προγράμματα μέχρι τώρα, έχουμε χρησιμοποιήσει μόνο τον στατικό τρόπο δέσμευσης μνήμης

Κατανομή Μνήμης

- Ο μεταγλωττιστής χωρίζει τη διαθέσιμη μνήμη του υπολογιστή στα ακόλουθα τέσσερα τμήματα:

  1) Στο *τμήμα κώδικα (code segment)*, το οποίο χρησιμοποιείται για την αποθήκευση του μεταγλωττισμένου κώδικα του προγράμματος

  2) Στο *τμήμα δεδομένων (data segment)*, το οποίο χρησιμοποιείται για την αποθήκευση των καθολικών και static μεταβλητών του προγράμματος. Αν και εξαρτάται από την υλοποίηση, σε αυτό το τμήμα συνήθως αποθηκεύονται και τα κυριολεκτικά αλφαριθμητικά (συνήθως σε περιοχή μνήμης που είναι μόνο για διάβασμα)

  3) Στο τμήμα που ονομάζεται *στοίβα (stack segment)*, το οποίο χρησιμοποιείται για την αποθήκευση των δεδομένων των συναρτήσεων (π.χ. τοπικές μεταβλητές)

  4) Στο τμήμα που ονομάζεται *σωρός (heap)*, το οποίο χρησιμοποιείται για δυναμική δέσμευση μνήμης

Στατική Δέσμευση Μνήμης (Ι)

- Με τη στατική δέσμευση, η μνήμη δεσμεύεται από τη στοίβα
- Το μέγεθος της μνήμης που θα δεσμευτεί πρέπει να είναι γνωστό πριν την εκτέλεση του προγράμματος και δεν μπορεί να αλλάξει κατά τη διάρκεια εκτέλεσής του
- Π.χ. όταν δημιουργούμε ένα πρόγραμμα για την αποθήκευση των βαθμών 500 φοιτητών, τότε γνωρίζουμε ότι πρέπει να δεσμεύσουμε μνήμη για 500 φοιτητές και όχι για 300 ή 600 (δηλ. το πρόγραμμα καθορίζει ακριβώς το μέγεθος της απαιτούμενης μνήμης)
- Άρα, με την εντολή: `float grades[500];` δεσμεύουμε με στατικό τρόπο από τη μνήμη του υπολογιστή `500*4 = 2000` bytes
- Το μέγεθος του πίνακα `grades` δεν μπορεί να αλλάξει κατά τη διάρκεια του προγράμματος
- Ο μοναδικός τρόπος για να αλλάξει το μέγεθος του πίνακα είναι να ξαναγραφεί το πρόγραμμα και να γίνει νέα μεταγλώττιση

Στατική Δέσμευση Μνήμης (ΙΙ)

Τι συμβαίνει όταν γίνεται κλήση μίας συνάρτησης???

- Όταν καλείται μία συνάρτηση, δεσμεύεται ένα τμήμα της στοίβας *(stack frame)* για να αποθηκευτούν τα δεδομένα της συνάρτησης
- Π.χ., όταν καλείται μία συνάρτηση που επιστρέφει τιμή, δέχεται παραμέτρους και χρησιμοποιεί τοπικές μεταβλητές, δεσμεύεται μνήμη στη στοίβα για να αποθηκευτούν τα παρακάτω δεδομένα:
  - Οι τιμές των παραμέτρων
  - Οι τοπικές μεταβλητές
  - Η τιμή επιστροφής
  - Η διεύθυνση της εντολής που θα εκτελεστεί μετά τον τερματισμό της συνάρτησης

Στατική Δέσμευση Μνήμης (III)

Τι συμβαίνει όταν τερματίζει μία συνάρτηση???

- Όταν τερματίσει η συνάρτηση (έστω η συνάρτηση του περιγράφηκε προηγουμένως) συμβαίνουν τα ακόλουθα:
  - Αν η τιμή επιστροφής της συνάρτησης εκχωρείται σε κάποια μεταβλητή, εξάγεται από τη στοίβα και αποθηκεύεται σε αυτήν
  - Η διεύθυνση της επόμενης εντολής εξάγεται επίσης από τη στοίβα, ώστε το πρόγραμμα να συνεχίσει με την εκτέλεσή της
  - Το τμήμα της στοίβας αποδεσμεύεται, άρα στις θέσεις μνήμης των τοπικών μεταβλητών και παραμέτρων της συνάρτησης μπορεί να εγγραφούν νέα δεδομένα

Παράδειγμα

Παράδειγμα

```
#include <stdio.h>

void test(int i, int j);

int main()
{
        float a[1000], b[10];

        /* Κλήσεις της test(). */
        return 0;
}

void test(int i, int j)
{
        int arr[200];
        ...
}
```

- Σε κάθε κλήση της `test()` δεσμεύονται 808 bytes από τη στοίβα, για να αποθηκευτούν οι τιμές των παραμέτρων `i` και `j` και των στοιχείων του πίνακα `arr`
- Όταν τερματίζει η εκτέλεση της `test()`, η μνήμη αποδεσμεύεται
- Παρομοίως, η μνήμη των 4040 bytes που έχει δεσμευτεί για τις τοπικές μεταβλητές της `main()` αποδεσμεύεται, όταν τερματιστεί η εκτέλεση του προγράμματος

Παρατηρήσεις

Αν δεν υπάρχει διαθέσιμος χώρος στη στοίβα για την αποθήκευση των δεδομένων μίας συνάρτησης, η εκτέλεση του προγράμματος **θα τερματιστεί ανώμαλα** και είναι πιθανό να εμφανιστεί το μήνυμα `"Stack overflow"` *(υπερχείλιση στοίβας)*

- Κάτι τέτοιο μπορεί να συμβεί αν μία συνάρτηση δεσμεύει αρκετή μνήμη και καλεί άλλες ένθετες συναρτήσεις, οι οποίες επίσης δεσμεύουν αρκετή μνήμη

- Μία **αναδρομική συνάρτηση** που **καλεί πολλές φορές τον εαυτό της** μπορεί επίσης να οδηγήσει σε εξάντληση της διαθέσιμης μνήμης της στοίβας

Δυναμική Δέσμευση Μνήμης (Ι)

- Η **δυναμική δέσμευση μνήμης** χρησιμοποιείται συνήθως όταν ο προγραμματιστής **δεν γνωρίζει εκ των προτέρων** το μέγεθος της μνήμης που χρειάζεται να δεσμεύσει, αφού το μέγεθος της μνήμης που δεσμεύεται με δυναμικό τρόπο **μπορεί να μεταβληθεί κατά τη διάρκεια του προγράμματος**
- Π.χ. όταν θέλουμε να δημιουργήσουμε ένα πρόγραμμα, το οποίο πρέπει να αποθηκεύει τους βαθμούς φοιτητών, αλλά **δεν γνωρίζουμε τον αριθμό των φοιτητών** τότε **δεν πρέπει** να δεσμεύσουμε μνήμη **με στατικό τρόπο**
- Αν δηλαδή, δηλώσουμε: `float grades[500];`

  τότε δεσμεύουμε με στατικό τρόπο μνήμη για αποθήκευση 500 βαθμών φοιτητών
- Όμως, αν οι φοιτητές **είναι περισσότεροι από** 500, τότε **δεν δεσμεύεται μνήμη για τους βαθμούς όλων των φοιτητών αλλά μόνο** για τους 500 πρώτους
- Επίσης, αν οι φοιτητές **είναι τελικά λιγότεροι από** 500, τότε έχουμε **σπατάλη μνήμης**, αφού **δεσμεύεται περισσότερη από την απαιτούμενη μνήμη**

Δυναμική Δέσμευση Μνήμης (ΙΙ)

- Με τη δυναμική δέσμευση, η μνήμη δεσμεύεται από τον **σωρό *(heap)*** **δυναμικά**, δηλαδή κατά τη διάρκεια εκτέλεσης του προγράμματος

- Όπως είπαμε, σε αντίθεση με τη στατική δέσμευση, το μέγεθος της μνήμης που θα δεσμευτεί **δεν χρειάζεται** να είναι γνωστό πριν την εκτέλεση του προγράμματος

- Επίσης, το μέγεθος της μνήμης που δεσμεύεται με δυναμικό τρόπο **μπορεί** να μεγαλώσει ή να μικρύνει, ανάλογα με τις απαιτήσεις του προγράμματος

- Τα μεγέθη της στοίβας και του σωρού είναι πεπερασμένα, ωστόσο το μέγεθος του **σωρού** είναι συνήθως **πολύ μεγαλύτερο** από το προκαθορισμένο μέγεθος της **στοίβας**

Δυναμική Δέσμευση Μνήμης (III)

- Π.χ. με την παρακάτω δήλωση πίνακα:

```
int arr[1000000];
```

το πρόγραμμα μπορεί να μην βρει την απαιτούμενη μνήμη στη στοίβα και να μην εκτελεστεί, ενώ με τις εντολές:

```
int *arr;
arr = (int*)malloc(1000000 * sizeof(int));
```

το πρόγραμμα θα βρει την απαιτούμενη μνήμη στον σωρό και θα εκτελεστεί χωρίς πρόβλημα

- Η μνήμη που δεσμεύεται με δυναμικό τρόπο πρέπει να αποδεσμεύεται, όταν πλέον δεν χρειάζεται (θα δούμε σε λίγο τη συνάρτηση `free()`)

Η συνάρτηση `malloc()`

- Η συνάρτηση `malloc()` χρησιμοποιείται για τη δυναμική δέσμευση ενός συγκεκριμένου μεγέθους μνήμης
- Το πρωτότυπό της δηλώνεται στο αρχείο `stdlib.h` και είναι το ακόλουθο:

```
void *malloc(size_t size);
```

- Η παράμετρος `size` δηλώνει πόσα bytes μνήμης θα δεσμευτούν
  - Αν η δέσμευση της μνήμης είναι επιτυχημένη, τότε η συνάρτηση επιστρέφει έναν δείκτη προς το πρώτο byte που δεσμεύτηκε, δηλαδή στην αρχή της δεσμευμένης μνήμης
  - Αν η δέσμευση της μνήμης αποτύχει, τότε η συνάρτηση επιστρέφει την τιμή `NULL`
- Η συνάρτηση `malloc()` επιστρέφει έναν δείκτη σε τύπο `void`, που σημαίνει ότι στη δεσμευμένη μνήμη μπορεί να αποθηκευτεί οποιοσδήποτε τύπος δεδομένων
- Παρόλα αυτά, προτείνεται να γίνεται προσαρμογή του τύπου δεδομένων που θα αποθηκευτούν στη μνήμη, ώστε να φαίνεται ξεκάθαρα ο τύπος των δεδομένων που θα περιέχει η μνήμη αυτή

Παράδειγμα χρήσης της malloc() (I)

```
int *ptr;
ptr = (int*) malloc(100);
```

- Το (int*) πριν από την κλήση της malloc() δεν είναι υποχρεωτικό να υπάρχει, αλλά προστίθεται για να φαίνεται ξεκάθαρα ότι η μνήμη που θα δεσμευτεί θα χρησιμοποιηθεί **για την αποθήκευση ακεραίων αριθμών** τύπου int
- Αν η δέσμευση της μνήμης είναι επιτυχημένη, τότε θα δεσμευτούν 100 bytes μνήμης και ο δείκτης ptr θα δείχνει στην αρχή αυτής της μνήμης
- Άρα, αφού κάθε ακέραιος απαιτεί 4 bytes, θα μπορούν να αποθηκευτούν **συνολικά 25 ακέραιοι** σε αυτή τη μνήμη
- Αντί για 100, θα μπορούσαμε (και μάλιστα συνηθίζεται) να γράψουμε 25*sizeof(int)
- Αν η δέσμευση αποτύχει, τότε η τιμή του ptr θα είναι ίση με NULL

Παράδειγμα χρήσης της malloc() (II)

```
char *ptr;
ptr = (char*) malloc(n+1);
```

- Με την παραπάνω δήλωση δεσμεύουμε δυναμικά μνήμη για την αποθήκευση n χαρακτήρων
- Η επιπλέον θέση (παρατηρείστε το n+1) δεσμεύεται για την αποθήκευση του τερματικού χαρακτήρα ('\0')

Παράδειγμα χρήσης της malloc() (III)

```
struct student *ptr;
ptr = (struct student*) malloc(100 * sizeof(student));
```

- Το (struct student*) πριν από την κλήση της malloc() δηλώνει ρητά ότι η μνήμη που θα δεσμευτεί θα χρησιμοποιηθεί για την αποθήκευση 100 δομών τύπου student
- Τα bytes μνήμης που θα δεσμευτούν θα είναι ίσα με το γινόμενο του 100 επί το μέγεθος της μνήμης που καταλαμβάνει μία τέτοια δομή (τύπου student)
- Ο δείκτης ptr θα δείχνει στην αρχή αυτής της μνήμης

Παρατηρήσεις (Ι)

- Εκτός από τη `malloc()` οι συναρτήσεις βιβλιοθήκης `realloc()` και `calloc()` χρησιμοποιούνται για δυναμική δέσμευση μνήμης, όμως η `malloc()` είναι η πιο συνηθισμένη από τις τρεις
- Τον δείκτη που επιστρέφει η συνάρτηση `malloc()` μπορούμε να τον χειριστούμε είτε σαν δείκτη είτε σαν πίνακα
- Δεδομένου ότι το όνομα ενός πίνακα είναι δείκτης στο πρώτο του στοιχείο, μπορούμε να θεωρήσουμε ότι η `malloc()` δημιουργεί έναν μονοδιάστατο πίνακα που έχει ως όνομα το όνομα του δείκτη
- Π.χ. αν δεσμεύσουμε μνήμη για 1000 ακέραιους

```
int *ptr;
ptr = (int*) malloc(1000*sizeof(int));
```

και θέλουμε να αποθηκεύσουμε στον πρώτο ακέραιο αυτής της μνήμης την τιμή 13, θα μπορούσαμε να γράψουμε:

- είτε `*ptr = 13;`
- είτε `ptr[0] = 13;`

Παρατηρήσεις (II)

Να ελέγχετε πάντα αν η τιμή επιστροφής της `malloc()` είναι διαφορετική από `NULL`

- Αν είναι `NULL` και επιχειρήσετε να χρησιμοποιήσετε τον δείκτη, το πρόγραμμα θα τερματιστεί με ανώμαλο τρόπο

- Για τον καθορισμό του μεγέθους της μνήμης να χρησιμοποιείτε τον τελεστή `sizeof`, ώστε το πρόγραμμα να μπορεί να εκτελείται σε διαφορετικούς υπολογιστές
- Π.χ. ο τύπος `short int` σε έναν υπολογιστή μπορεί να δεσμεύει `2` bytes και σε κάποιον άλλο `4` bytes

- Όταν δεν χρειάζεστε άλλο τη δεσμευμένη μνήμη, πρέπει να την απελευθερώνετε, ώστε να μπορεί το λειτουργικό σύστημα να τη διαθέσει σε άλλα προγράμματα
- Η αποδέσμευση της μνήμης γίνεται με τη συνάρτηση `free()`
- Ο γενικός κανόνας είναι ότι η μνήμη που δεσμεύεται με κάθε κλήση της συνάρτησης `malloc()` πρέπει να αποδεσμεύεται με αντίστοιχη κλήση της συνάρτησης `free()`

Παρατηρήσεις (III)

- Η προσαρμογή (type casting) του τύπου επιστροφής της `malloc()` δεν είναι υποχρεωτική
- Δηλαδή, π.χ αντί για :

```
int *ptr;
ptr = (int*) malloc(100);
```

Θα μπορούσαμε να γράψουμε:

```
int *ptr;
ptr = malloc(100);
```

- Αν και η πρώτη σύνταξη είναι πιο δυσνόητη, προτείνεται γιατί:

α) φαίνεται ξεκάθαρα ο τύπος των δεδομένων που θα περιέχει η μνήμη και δεν χρειάζεται ο αναγνώστης του κώδικα να ανατρέξει στη δήλωση της μεταβλητής `ptr` για να το θυμηθεί

β) Θα μπορεί το πρόγραμμά σας να μεταγλωττιστεί σε κάποιον άλλο μεταγλωττιστή που μπορεί να απαιτεί type casting (π.χ. σε C++ μεταγλωττιστή)

Η συνάρτηση free()

- Η συνάρτηση free() χρησιμοποιείται για την απελευθέρωση μνήμης που δεσμεύτηκε δυναμικά
- Το πρωτότυπό της δηλώνεται στο αρχείο stdlib.h και είναι το ακόλουθο:

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

- Η παράμετρος ptr είναι δείκτης στη μνήμη που δεσμεύτηκε με κάποια συνάρτηση δυναμικής δέσμευσης μνήμης, π.χ. malloc()

ΠΡΟΣΟΧΗ: Αν ο δείκτης-παράμετρος της free() δεν δείχνει σε δυναμικά δεσμευμένη μνήμη, θα δημιουργηθεί πρόβλημα στην εκτέλεση του προγράμματος

- Π.χ. το διπλανό πρόγραμμα δεν θα εκτελεστεί σωστά, γιατί ο δείκτης ptr έχει απλώς οριστεί και δεν δείχνει σε δεσμευμένη μνήμη...

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        int* ptr;
        free(ptr);
        return 0;
}
```

Παρατηρήσεις

- Η `free()` χρησιμοποιείται για την αποδέσμευση δυναμικής μνήμης και όχι στατικής

- Η απόπειρα αποδέσμευσης μνήμης που έχει ήδη αποδεσμευτεί είναι επίσης λανθασμένη ενέργεια

Π.χ. ο παρακάτω κώδικας δεν είναι σωστός, αφού η δεύτερη κλήση της `free()` προσπαθεί να αποδεσμεύσει μία μνήμη που έχει ήδη αποδεσμευτεί με την πρώτη κλήση

```
int *ptr = (int *) malloc(100*sizeof(int));
if(ptr != NULL)
{
        free(ptr);
        free(ptr);
}
```

Όταν δεν χρειάζεστε άλλο τη δεσμευμένη μνήμη, μην ξεχνάτε να καλείτε την `free()` για να την απελευθερώσετε

Η συνάρτηση memcpy()

- Η συνάρτηση memcpy() χρησιμοποιείται για την αντιγραφή οποιουδήποτε τύπου δεδομένων από μία περιοχή μνήμης σε μία άλλη
- Το πρωτότυπό της δηλώνεται στο αρχείο string.h και είναι το ακόλουθο:

```
void *memcpy(void *dest, const void *src, size_t size);
```

- Η συνάρτηση memcpy() αντιγράφει size bytes από την περιοχή μνήμης στην οποία δείχνει ο δείκτης src στην περιοχή μνήμης στην οποία δείχνει ο δείκτης dest
- Αν οι περιοχές μνημών επικαλύπτονται, η συμπεριφορά της συνάρτησης είναι ακαθόριστη

Παράδειγμα

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
	char *arr;
	/* Δυναμική δέσμευση μνήμης ίση με το μήκος του αλφαριθμητικού "ABCDE" και μίας επιπλέον οκτάδας για τον χαρακτήρα '\0'. */
	arr = (char *) malloc(6);
	if(arr != NULL)
	{
		memcpy(arr, "ABCDE", 6);
		printf("%s\n", arr);
		free(arr);
	}
	return 0;
}
```

Παρατηρήσεις

- Όταν η `memcpy()` χρησιμοποιείται για την αντιγραφή αλφαριθμητικών μοιάζει με τη `strcpy()` με την εξής όμως διαφορά:
  - Με την `strcpy()` η αντιγραφή αλφαριθμητικών ολοκληρώνεται όταν συναντηθεί ο τερματικός χαρακτήρας (`'\0'`)
  - Αντίθετα, η `memcpy()` δεν σταματάει την αντιγραφή ακόμα κι αν συναντήσει τον τερματικό χαρακτήρα

Π.χ. αν έχουμε:

```
char str2[6], str1[] = {'a', 'b', 'c', '\0', 'd', 'e'};
```

και γράψουμε: `memcpy(str2, str1, 6);`

το περιεχόμενο του πίνακα `str2` θα γίνει το ίδιο με του `str1`

Αντίθετα, αν γράψουμε: `strcpy(str2, str1);`

το περιεχόμενο του `str2` θα γίνει ίσο με `{'a', 'b', 'c', '\0'}`

Η συνάρτηση memmove()

- Η συνάρτηση **memmove()** είναι παρόμοια με την **memcpy()**, με τη διαφορά ότι η **memmove()** **εξασφαλίζει τη σωστή αντιγραφή** των δεδομένων, **ακόμα και αν οι δύο περιοχές μνήμης επικαλύπτονται**

- Επειδή ακριβώς η **memcpy()** δεν κάνει αυτόν τον έλεγχο, εκτελείται πιο γρήγορα από τη **memmove()**

Παρατηρήσεις

- Κατά τη χρήση της `memcpy()` ή της `memmove()`, για τη μνήμη προορισμού **πρέπει** να έχουν δεσμευτεί τουλάχιστον `size` bytes, σε διαφορετική περίπτωση, τα πλεονάζοντα bytes θα εγγραφούν σε **μη δεσμευμένη μνήμη**, η οποία μπορεί να χρησιμοποιείται για άλλους σκοπούς

- Π.χ. η επόμενη αντιγραφή **δεν είναι σωστή**, γιατί το μέγεθος της μνήμης προορισμού είναι 3 bytes, ενώ τα bytes που θα αντιγραφούν είναι 6

```
char str1[3];
char str2[] = "abcde";
memcpy(str1, str2, sizeof(str2));
```

- Η συνάρτηση `memcpy()` είναι πολύ **χρήσιμη**, γιατί συνήθως υλοποιείται με τέτοιο τρόπο ώστε η αντιγραφή μεγάλου όγκου δεδομένων από μία περιοχή μνήμης σε μία άλλη να ολοκληρώνεται πιο γρήγορα από ότι με επαναληπτικούς βρόχους

- Για παράδειγμα, αν θέλετε να αντιγράψετε τα περιεχόμενα ενός πίνακα `100000` ακεραίων σε έναν άλλον πίνακα, τότε να χρησιμοποιήσετε τη `memcpy()` και όχι `for` βρόχο, γιατί το πιθανότερο είναι **η αντιγραφή να ολοκληρωθεί πιο γρήγορα**

Παράδειγμα

- Γράψτε ένα πρόγραμμα το οποίο να ορίζει δύο πίνακες 100000 ακεραίων, να θέτει τις τιμές από 1 έως 100000 στα στοιχεία του πρώτου πίνακα και να αντιγράφει τις τιμές των στοιχείων του στον δεύτερο πίνακα με χρήση της συνάρτησης **memcpy()**.

  Το πρόγραμμα να εμφανίζει τα περιεχόμενα του δεύτερου πίνακα και να τερματίζει.

```
#include <stdio.h>
#include <string.h>
int main()
{
        int i,arr1[100000],arr2[100000];

        for(i = 0; i < 100000; i++)
                arr1[i] = i+1;

        memcpy(arr2,arr1,sizeof(arr1));

        for(i = 0; i < 100000; i++)
                printf("%d\n",arr2[i]);
        return 0;
}
```

Εναλλακτικά:

```
for(i = 0; i < 100000; i++)
        arr2[i] = arr1[i];
```

Η συνάρτηση memcmp()

- Η συνάρτηση **memcmp()** χρησιμοποιείται για **τη σύγκριση οποιουδήποτε τύπου δεδομένων** που περιέχονται σε μία περιοχή μνήμης με τα δεδομένα που περιέχονται σε μία άλλη περιοχή μνήμης

- Το πρωτότυπό της δηλώνεται στο αρχείο **string.h** και είναι το ακόλουθο:

```
int memcmp(const void *ptr1, const void *ptr2, size_t size);
```

- Η συνάρτηση **memcmp()** συγκρίνει **size** bytes από την περιοχή μνήμης στην οποία δείχνει ο δείκτης **ptr1** με τα αντίστοιχα bytes που περιέχονται στην περιοχή μνήμης στην οποία δείχνει ο δείκτης **ptr2**

- Αν οι δύο περιοχές μνήμης **περιέχουν τα ίδια δεδομένα**, τότε η συνάρτηση **memcmp()** επιστρέφει **0**, αλλιώς μια μη μηδενική τιμή

Παρατηρήσεις

- Όταν η `memcmp()` χρησιμοποιείται για τη σύγκριση αλφαριθμητικών μοιάζει με τη `strcmp()` με την εξής όμως διαφορά:
  - Με την `strcmp()` η σύγκριση αλφαριθμητικών ολοκληρώνεται όταν συναντηθεί ο τερματικός χαρακτήρας (`'\0'`)
  - Αντίθετα, η `memcmp()` δεν σταματάει τη σύγκριση όταν συναντηθεί ο τερματικός χαρακτήρας (`'\0'`)

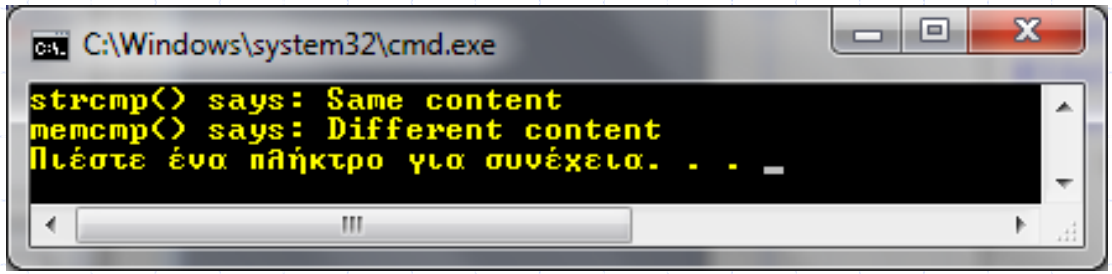
Παράδειγμα (Ι)

- Ποια είναι η έξοδος του προγράμματος???

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main()
{
	char str1[] = {'a', 'b', 'c', '\0', 'd', 'e'};
	char str2[] = {'a', 'b', 'c', '\0', 'd', 'f'};

	if(strcmp(str1, str2) == 0)
		printf("strcmp() says: Same content\n");
	else
		printf("strcmp() says: Different content\n");

	if(memcmp(str1, str2, sizeof(str1)) == 0)
		printf("memcmp() says: Same content\n");
	else
		printf("memcmp() says: Different content\n");
	return 0;
}
```



Παράδειγμα (ΙΙ)

- Γράψτε ένα πρόγραμμα το οποίο να ορίζει δύο πίνακες 100000 ακεραίων, να θέτει τις τιμές από 1 έως 100000 στα στοιχεία τους και να συγκρίνει τις τιμές των στοιχείων τους με χρήση της συνάρτησης **memcmp()**.

  Το πρόγραμμα να εμφανίζει ένα διαγνωστικό μήνυμα για το αποτέλεσμα της σύγκρισης.

```c
#include <stdio.h>
#include <string.h>
int main()
{
	int i,arr1[100000],arr2[100000];

	for(i = 0; i < 100000; i++)
		arr1[i] = arr2[i] = i+1;

	if(memcmp(arr1,arr2,sizeof(arr1)) == 0)
		printf("The same content\n");
	else
		printf("Different content\n");

	return 0;
}
```

Εναλλακτικά:

```c
for(i = 0; i < 100000; i++)
	if(arr2[i] != arr1[i])
	{
	  printf("Different content\n");
	  break;
	}
```

Στατικές Δομές Δεδομένων

- Οι **δομές δεδομένων** χρησιμοποιούνται για την αποθήκευση και επεξεργασία πληθώρας δεδομένων με εύκολο και γρήγορο τρόπο
- Π.χ. ο **πίνακας** είναι μία **δομή δεδομένων**, ο οποίος χρησιμοποιείται για την αποθήκευση δεδομένων ίδιου τύπου
- Παρομοίως, οι **δομές** (**structs**) και οι **ενώσεις** (**unions**) είναι **δομές δεδομένων**, οι οποίες χρησιμοποιούνται για την αποθήκευση δεδομένων οποιουδήποτε τύπου
- **Αυτές οι δομές δεδομένων ονομάζονται στατικές, γιατί το μέγεθος της μνήμης που έχει δεσμευτεί για αυτές δεν μπορεί να αλλάξει κατά την εκτέλεση του προγράμματος**
- Π.χ. όταν δηλώνεται έναν πίνακας με μία συγκεκριμένη διάσταση (π.χ. `int arr[100]`), η διάστασή του, δηλαδή το 100, δεν μπορεί να αλλάξει κατά την εκτέλεση του προγράμματος

Δυναμικές Δομές Δεδομένων

- Υπάρχουν όμως περιπτώσεις που αντί να χρησιμοποιήσουμε μία **στατική** δομή δεδομένων, όπως είναι ο πίνακας, να είναι πιο αποδοτικό να χρησιμοποιήσουμε μία **δυναμική δομή δεδομένων**

- Αντίθετα με τη στατική δομή, **το μέγεθος** μίας **δυναμικής δομής** δεδομένων **μπορεί να αυξομειώνεται κατά την εκτέλεση του προγράμματος** με τη δέσμευση και την αποδέσμευση αντίστοιχης μνήμης

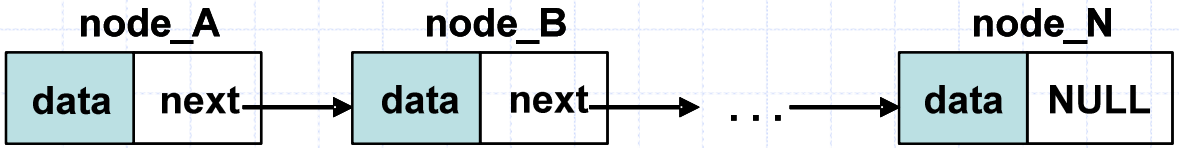
- Τα παραδείγματα των δυναμικών δομών δεδομένων που θα περιγράψουμε είναι η **απλά συνδεδεμένη λίστα**, η **ουρά** και η **στοίβα**

- Μία **δυναμική δομή δεδομένων** αποτελείται από ένα ή περισσότερα **συνδεδεμένα στοιχεία**, τα οποία συνήθως ονομάζονται **κόμβοι**

- Κάθε **κόμβος** συνήθως αντιπροσωπεύεται από μία **δομή**, η οποία περιέχει – συν τοις άλλοις – κι ένα πεδίο που είναι δείκτης στον επόμενο κόμβο

Απλά συνδεδεμένη λίστα

- Η **πιο συνηθισμένη δυναμική δομή δεδομένων** είναι μία **απλά συνδεδεμένη λίστα**
- Στο σχήμα φαίνεται ότι κάθε κόμβος μίας τέτοιας λίστας περιέχει τα **δεδομένα του κόμβου** (π.χ. το πεδίο `data`) και **έναν δείκτη** (π.χ. το πεδίο `next`) που «δείχνει» στον επόμενο κόμβο



- Ο **πρώτος κόμβος** της λίστας ονομάζεται **κεφαλή (head)** της λίστας και ο τελευταίος κόμβος ονομάζεται **ουρά (tail)**
- Το πεδίο-δείκτης του τελευταίου κόμβου **πρέπει** να έχει την τιμή `NULL`, ώστε να προσδιορίζεται το τέλος της λίστας
- Ο χειρισμός μίας απλά συνδεδεμένης λίστας γίνεται συνήθως με τη χρήση **δύο δεικτών**, με τον πρώτο να δείχνει στη διεύθυνση μνήμης της **κεφαλής** της λίστας και τον δεύτερο στη διεύθυνση μνήμης της **ουράς** της λίστας

Εισαγωγή κόμβου σε απλά συνδεδεμένη λίστα (Ι)

- Για να εισάγουμε έναν νέο κόμβο στη λίστα, εξετάζουμε τις ακόλουθες περιπτώσεις:

  1) Αν η λίστα είναι κενή (δηλ. δεν περιέχει κανέναν κόμβο) τότε ο κόμβος εισάγεται στη λίστα και αποτελεί ταυτόχρονα την κεφαλή και την ουρά της λίστας, ενώ η τιμή του δείκτη του γίνεται `NULL` , αφού δεν υπάρχει επόμενος κόμβος

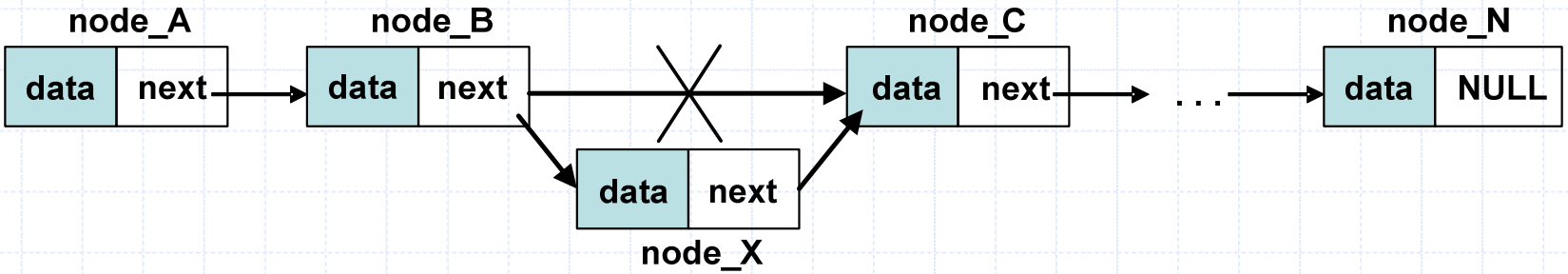
  2) Αν η λίστα δεν είναι κενή τότε εξετάζουμε τις ακόλουθες υπο-περιπτώσεις:

     2α) Αν επιθυμούμε ο νέος κόμβος να τοποθετηθεί στην αρχή της λίστας, τότε ο νέος κόμβος γίνεται η νέα κεφαλή της λίστας και ο δείκτης του δείχνει στην παλιά κεφαλή, που τώρα γίνεται ο δεύτερος κόμβος της λίστας

Εισαγωγή κόμβου σε απλά συνδεδεμένη λίστα (ΙΙ)

2β) Αν επιθυμούμε ο νέος κόμβος να τοποθετηθεί στο **τέλος** της λίστας, τότε ο νέος κόμβος γίνεται η **νέα ουρά** της λίστας και η τιμή του δείκτη του γίνεται **NULL**, ενώ ο κόμβος που ήταν προηγουμένως η ουρά της λίστας γίνεται ο προτελευταίος κόμβος της λίστας με την τιμή του δείκτη του να αλλάζει από **NULL** και να δείχνει στον νέο κόμβο

2γ) Για να εισάγουμε έναν νέο κόμβο **μετά** από έναν ενδιάμεσο κόμβο μίας λίστας, τότε κάνουμε τον δείκτη αυτού του κόμβου να δείχνει στον νέο κόμβο και τον δείκτη του νέου κόμβου να δείχνει στον κόμβο που έδειχνε ο ενδιάμεσος κόμβος (όπως φαίνεται στο σχήμα, με τον κόμβο X να εισάγεται μεταξύ των κόμβων B και C)



Διαγραφή κόμβου από απλά συνδεδεμένη λίστα (Ι)

- Για να διαγράψουμε έναν κόμβο από μία λίστα, εξετάζουμε τις ακόλουθες περιπτώσεις:

  1) Αν επιθυμούμε να διαγράψουμε τον κόμβο που είναι η **αρχή** της λίστας, τότε εξετάζουμε τις ακόλουθες υπο-περιπτώσεις:

     1α) Αν υπάρχει επόμενος κόμβος, τότε αυτός ο κόμβος γίνεται η **νέα κεφαλή** της λίστας

     1β) Αν δεν υπάρχει επόμενος κόμβος, τότε η λίστα γίνεται **κενή**

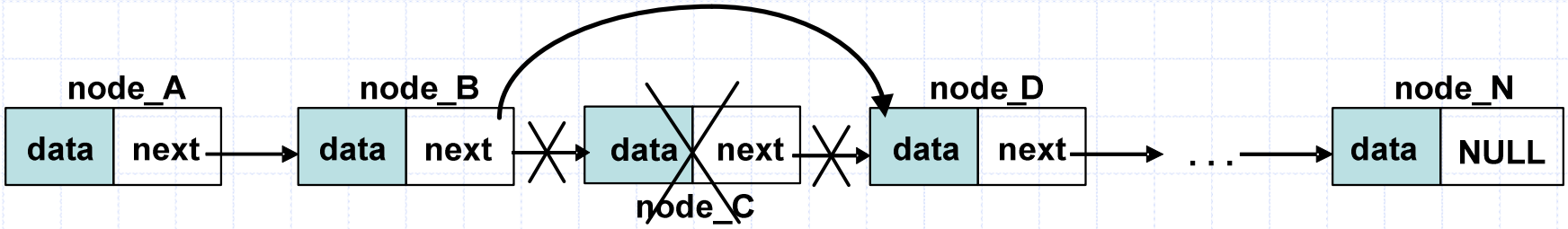
  2) Αν επιθυμούμε να διαγράψουμε τον **τελευταίο** κόμβο της λίστας, τότε εξετάζουμε τις ακόλουθες υπο-περιπτώσεις:

     2α) Αν υπάρχει προηγούμενος κόμβος, τότε αυτός ο κόμβος γίνεται η **νέα ουρά** της λίστας και η τιμή του δείκτη του γίνεται **NULL**

     2β) Αν δεν υπάρχει προηγούμενος κόμβος, τότε η λίστα γίνεται **κενή**

Διαγραφή κόμβου από απλά συνδεδεμένη λίστα (ΙΙ)

3) Για να διαγράψουμε ένα κόμβο που βρίσκεται ανάμεσα σε δύο κόμβους μίας λίστας, τότε κάνουμε τον δείκτη του προηγούμενου κόμβου από αυτόν που θέλουμε να διαγράψουμε να δείχνει στον επόμενο κόμβο από αυτόν που θέλουμε να διαγράψουμε και αποδεσμεύουμε τη μνήμη που καταλαμβάνει (όπως φαίνεται στο σχήμα, όπου διαγράφεται ο κόμβος C)



Παρατηρήσεις

- Το **βασικό μειονέκτημα** μίας λίστας, σε σχέση με τους πίνακες, είναι ότι για να βρούμε κάποιο στοιχείο της λίστας **πρέπει να διατρέξουμε <u>όλη τη λίστα</u>** ξεκινώντας από την αρχή της, ενώ με τους πίνακες μπορούμε να έχουμε **άμεση πρόσβαση** στο επιθυμητό στοιχείο

Στοίβα (stack)

- Η **στοίβα** (**stack**) αποτελεί μία ειδική περίπτωση λίστας, με τους ακόλουθους περιορισμούς:

  1) Η εισαγωγή ενός νέου κόμβου γίνεται **μόνο** στην **αρχή** της στοίβας, δηλαδή, κάθε νέος κόμβος στη στοίβα γίνεται η νέα κεφαλή της στοίβας

  2) Ο **μόνος** κόμβος που επιτρέπεται να διαγράψουμε από τη στοίβα είναι η **κεφαλή** της στοίβας

- Μία τέτοια στοίβα ονομάζεται **LIFO** (**L**ast **I**n **F**irst **O**ut), με την έννοια ότι ο κόμβος που **εισάγεται τελευταίος εξάγεται πρώτος**

Ουρά (queue)

- Η **ουρά** (**queue**) αποτελεί μία ειδική περίπτωση λίστας, με τους ακόλουθους περιορισμούς:

  1) Η εισαγωγή ενός νέου κόμβου γίνεται **μόνο** στο **τέλος** της ουράς, δηλαδή, κάθε νέος κόμβος γίνεται η νέα «ουρά» της ουράς

  2) Ο **μόνος** κόμβος που επιτρέπεται να διαγράψουμε από την ουρά είναι η **κεφαλή** της ουράς

- Μία τέτοια ουρά ονομάζεται **FIFO** (**F**irst **I**n **F**irst **O**ut), με την έννοια ότι ο κόμβος που **εισάγεται πρώτος εξάγεται και πρώτος**