

Διάλεξη 13η: Δυναμική Διαχείριση Μνήμης

Τμήμα Επιστήμης Υπολογιστών, Πανεπιστήμιο Κρήτης

Εισαγωγή στην Επιστήμη Υπολογιστών



Δυναμική Διαχείριση Μνήμης

- Μέχρι τώρα βλέπουμε στατική ανάθεση και δέσμευση μνήμης
 - Ζητούσαμε τη μέγιστη μνήμη που μπορεί να χρειαζόταν το πρόγραμμα

```
#define MAX_SIZE 10000  
int array[MAX_SIZE];
```

- Μειονέκτημα: Υπάρχουν εφαρμογές όπου το `MAX_SIZE` είναι άγνωστο
 - Αναγκαζόμαστε να δηλώνουμε μεγάλες τιμές και να χάνουμε σε μνήμη και σε ταχύτητα
- Δυναμική δέσμευση μνήμης (dynamic memory allocation)
 - Αίτηση στο λειτουργικό σύστημα να παραχωρήσει μνήμη στο πρόγραμμα
 - Ειδοποίηση στο λειτουργικό ότι δεν χρειαζόμαστε πια ένα μέρος μνήμης
 - Ό,τι δεσμεύεται πρέπει να αποδεσμεύεται



Κατηγορίες Μνήμης του Προγράμματος

- Μνήμη όπου αποθηκεύεται ο κώδικας του προγράμματος
- Μνήμη όπου αποθηκεύονται οι μεταβλητές που δεσμεύονται με στατική (καθολικές) ή αυτόματη (τοπικές) διαχείριση μνήμης
 - Αυτή η μνήμη ονομάζεται *στοίβα* (stack)
- Μνήμη όπου αποθηκεύονται οι μεταβλητές που δεσμεύονται με δυναμική διαχείριση μνήμης
 - Αυτή η μνήμη ονομάζεται *σωρός* (heap)



Συναρτήσεις Δυναμικής Διαχείρισης Μνήμης

- Βρίσκονται στη βιβλιοθήκη `stdlib.h`

Συναρτήσεις διαχείρισης μνήμης

```
void *malloc(size_t num_bytes);  
void *calloc(size_t num_elements, size_t elt_size);  
void *realloc(void *ptr, size_t size);  
void free(void *ptr);
```



Η συνάρτηση `malloc`

- Η συνάρτηση δέσμευσης `malloc`

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

- Δεσμεύει `num_bytes` αριθμό από διαδοχικά bytes και επιστρέφει ένα δείκτη στην αρχή της δεσμευμένης θέσης μνήμης
- Επιστρέφει `NULL` αν δεν μπορεί να δεσμεύσει άλλη μνήμη

```
int *x, *y;  
x = (int *) malloc(sizeof(int));  
y = (int *) malloc(100 * sizeof(int));
```



Η συνάρτηση `calloc`

- Η συνάρτηση δέσμευσης `calloc`

```
void *calloc(size_t num_elements, size_t elt_size);
```

- Δεσμεύει `num_elements` αριθμό από διαδοχικά αντικείμενα το καθένα μεγέθους `elt_size` και επιστρέφει ένα δείκτη στην αρχή της δεσμευμένης μνήμης
- Επιστρέφει NULL αν δεν μπορεί να δεσμεύσει άλλη μνήμη
- Αρχικοποιεί τα περιεχόμενα της δεσμευμένης μνήμης στο 0
- Παρόμοια με την `malloc`
 - Διαφορετική κλήση, παίρνει αριθμό στοιχείων και το μέγεθος ενός στοιχείου
 - Αρχικοποιεί τα περιεχόμενα με 0

```
int *x, *y;  
x = (int *) calloc(1, sizeof(int));  
y = (int *) calloc(100, sizeof(int));
```



Η συνάρτηση `realloc`

- Η συνάρτηση δέσμευσης `realloc`

```
void *realloc(void *ptr, size_t size);
```

- Επεκτείνει τη δεσμευμένη μνήμη
- Ο δείκτης `ptr` πρέπει να δείχνει σε μνήμη ήδη δεσμευμένη με την `malloc` ή `calloc`

```
ptr = (int *) malloc (100 * sizeof(int));  
ptr = (int *) realloc(ptr, 200 * sizeof(int));
```

- Τα προηγούμενα περιεχόμενα της δεσμευμένης μνήμης διατηρούνται (τουλάχιστον όσα χωρούν στη νέα)
- Η νέα μνήμη μπορεί να είναι σε άλλη διεύθυνση αν δεν υπήρχε αρκετός συνεχόμενος χώρος σε ελεύθερη μνήμη στο τέλος της υπάρχουσας
 - Σε αυτή την περίπτωση γίνεται μεταφορά των δεδομένων και αποδεσμεύεται η μνήμη `ptr` σαν να γίνεται `free(ptr)`



Η συνάρτηση `free`

- Η συνάρτηση αποδέσμευσης `free`
 - Αποδεσμεύει τη μνήμη που έχει δεσμευτεί με μία από τις προηγούμενες κλήσεις στην `malloc`, `calloc` ή `realloc`
 - Προσοχή: Δεν μπορούμε να αποδεσμεύσουμε πάνω από μια φορά την ίδια μνήμη

```
ptr = (int *) malloc (100 * sizeof(int));  
free(ptr); // OK  
free(ptr); // error!
```



Ο τύπος `void *`

- Γενικός τύπος δείκτη σε μια διεύθυνση μνήμης
 - Ο τύπος του δείκτη δεν έχει πληροφορία για το τί βρίσκεται σε αυτή τη διεύθυνση
 - Σημαίνει “δείκτης που δείχνει σε δεδομένα που δεν χρειάζεται να ξέρουμε τι είναι”
- Ο γενικός δείκτης επιτρέπει συναρτήσεις που επιστρέφουν χώρο στη μνήμη χωρίς να προσδιορίζουν τον τύπο των δεδομένων που θα αποθηκευτούν, π.χ., `malloc`
- Προσοχή: είναι τελείως διαφορετικό από το `void`
 - `void f()` - η συνάρτηση δεν επιστρέφει τίποτα
 - `void *f()` - η συνάρτηση επιστρέφει μια διεύθυνση μνήμης χωρίς να ξέρουμε τι τύπος δεδομένων περιέχεται εκεί



Δυναμικοί Πίνακες

- Πίνακες με μέγεθος που καθορίζεται κατά την εκτέλεση του προγράμματος
- Καλύτερη προσαρμογή στις απαιτήσεις του προγράμματος χωρίς περιορισμό

Παράδειγμα

```
int *create_array(int elements) {  
    int i, *x = (int *) malloc(elements * sizeof(int));  
  
    for(i = 0; i < elements; i++) {  
        x[i] = i;  
    }  
    return x;  
}
```



Πιθανά Λάθη: Αποτυχία Δέσμευσης

- Αν δεν υπάρχει αρκετή μνήμη οι συναρτήσεις δέσμευσης επιστρέφουν **NULL**
- Κάνετε πάντα έλεγχο για αυτή την περίπτωση
- Είναι ευθύνη του προγραμματιστή και αρχή σωστού προγραμματισμού
- Αν αποτύχει η δέσμευση μνήμης μπορούμε να διακόψουμε την εκτέλεση του προγράμματος αν χρειάζεται

check-malloc.c

```
#include<stdio.h>
int main(void) {
    int * x;

    x = (int *) malloc(1000 * sizeof(int));
    if (x == NULL) {
        printf("Not enough memory\n");
        exit(1);
    }
    x[0] = 42;
}
```



Πιθανά Λάθη: Αποτυχία Δέσμευσης (2)

- Αντί να ελέγχουμε κάθε φορά, μπορούμε να γράψουμε δική μας `malloc`

check-malloc2.c

```
#include<stdio.h>

void *malloc_or_fail(size_t size) {
    void *ptr = malloc(size);
    if (ptr == NULL) {
        printf("Not enough memory\n");
        exit(1);
    }
    return ptr;
}

int main() {
    int *x;

    x = (int *) malloc_or_fail(1000 * sizeof(int));
    x[0] = 42;
}
```



Πιθανά Λάθη: Διαρροή Μνήμης

- Πρέπει να αποδεσμεύουμε την μνήμη που δεν χρειαζόμαστε πλέον με τη συνάρτηση **free**
- Δεν ελευθερώνουμε ποτέ μνήμη που δεν έχει αποκτηθεί δυναμικά
- Αν το πρόγραμμα δεν ελευθερώσει όλη τη μνήμη που δεσμεύει, έχει *διαρροή μνήμης* (memory leak)

Διαρροή

```
int use_array(void) {  
    int *x = (int *) malloc(10 * sizeof(int));  
}
```



Πιθανά Λάθη: Διαρροή Μνήμης

- Η διαρροή μνήμης μπορεί να εξαντλήσει την ελεύθερη μνήμη
 - Όταν τελειώνει η ελεύθερη μνήμη του υπολογιστή το λειτουργικό αποθηκεύει κομμάτια μνήμης στο δίσκο προσωρινά για να ελευθερώσει χώρο
 - Το λειτουργικό επαναφέρει τα περιεχόμενα από το δίσκο όταν χρειαστούν, πιθανώς αποθηκεύοντας άλλα για να κάνει χώρο
 - Αν γεμίσει η μνήμη θα φτάσει να γράφει και να διαβάζει το δίσκο για κάθε πρόσβαση
 - Αυτό λέγεται *thrashing*
 - Ακόμη και η διαρροή μερικών bytes μπορεί να προκαλέσει *thrashing* μετά από ώρες, μέρες ή χρόνια
- Προσοχή: Αποδεσμεύουμε αμέσως τη μνήμη που δεν χρησιμοποιούμε πλέον με τη συνάρτηση **free**



Πιθανά Λάθη: Χρήση Αποδεσμευμένης Μνήμης

- Η αποδέσμευση μνήμης την απελευθερώνει για άλλη χρήση
- Δεν ανήκει πλέον στο πρόγραμμα
- Η πρόσβαση σε απελευθερωμένη μνήμη ισοδυναμεί με χρήση μη δεσμευμένης μνήμης
 - Segmentation fault
 - Γράφουμε τυχαία σε ό,τι άλλες μεταβλητές τυχαίνει να επαναδεσμεύσουν την ίδια μνήμη (ανιχνεύεται δύσκολα)



Πιθανά Λάθη: Αποδέσμευση άλλης μνήμης

- Η συνάρτηση **free** αποδεσμεύει μόνο μνήμη που δεσμεύτηκε δυναμικά
- Δεν μπορούμε να αποδεσμεύσουμε άλλη μνήμη, ακόμη κι αν ανήκει στο πρόγραμμα
 - Τοπικές, καθολικές, στατικές, μεταβλητές, δείκτες σε συναρτήσεις, κλπ.
- Δεν ορίζεται συμπεριφορά της **free** για διευθύνσεις που δεν δεσμεύτηκαν δυναμικά
- Ανάλογα με την υλοποίηση, η αποδέσμευση άλλων διευθύνσεων μπορεί να προκαλέσει λάθος στο λειτουργικό σύστημα



Δείκτες σε δείκτες

- Ένας δείκτης μπορεί να δείχνει σε οποιοδήποτε τύπο δεδομένων
 - Ακόμη και δείκτη
 - Κ.Ο.Κ.

```
char ch;  
char *pch;  
char **ppch;  
char ***pppch;
```

- Η **ch** περιέχει ένα χαρακτήρα
- Η **pch** περιέχει δείκτη σε χαρακτήρα
 - Ισοδύναμο με πίνακα χαρακτήρων με άγνωστο μέγεθος
- Η **ppch** περιέχει δείκτη σε πίνακες χαρακτήρων
 - Ισοδύναμο με πίνακα με άγνωστο μέγεθος από πίνακες χαρακτήρων άγνωστου μεγέθους, ή πίνακα με strings
- Η **pppch** περιέχει δείκτη σε πίνακα από πίνακες με strings
- Κ.λ.π.



- Με τις μεταβλητές:

```
char ch;  
char *pch = &ch;  
char **ppch = &pch;  
char ***pprch = &ppch;
```

- Η εντολή `*pch = 'a'` γράφει 'a' στο `ch`
- Η εντολή `**ppch = 'a'` γράφει 'a' στο `ch`
- Η εντολή `**pprch = 'a'` είναι λάθος



Παράδειγμα

ptptr.c

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

int main(void) {
    int lines = 5;
    int cols = 10;
    int i, **arr;

    arr = (int **) malloc(lines * sizeof(int *));
    if(arr == NULL) {
        printf("Cannot allocate row pointers.\n");
        exit(0);
    }

    for(i = 0; i < lines; i++) {
        arr[i] = (int *) malloc(cols * sizeof(int));
        if(arr[i] == NULL) {
            printf("Cannot allocate row[%d]\n", i);
            exit(0);
        }

        printf("%d %p %ld", i,
               (void *)arr[i],
               (long)arr[i]);
        if(i > 0) {
            printf(" %d\n",
                   (int)(arr[i] - arr[i - 1]));
        } else {
            printf("\n");
        }
    }
    /* free the memory */
    for (i = 0; i < lines; i++) {
        free (arr[i]);
    }
    free(arr);
    return 0;
}
```

Δείκτες και πίνακες

- Ομοιότητες και διαφορές μεταξύ των δεικτών σε δείκτες και τους δισδιάστατους πίνακες
- Δισδιάστατοι πίνακες
 - `int a[10][20];`
 - Η έκφραση `a[i][j]` είναι συντακτικά σωστή
 - Ο πίνακας δεσμεύει 10×20 ακέραιους στη μνήμη
 - Η διεύθυνση του `a[i][j]` είναι `a + i*20 + j`, ή
 - η διεύθυνση του `a[i][j]` είναι `(char*)a + (i * 20 + j) * sizeof(int)`



Δείκτες και πίνακες (2)

- Δείκτες σε δείκτες

- `int **b;`
- Η έκφραση `b[i][j]` είναι συντακτικά σωστή
- Για δέσμευση 10 δεικτών χρειάζεται

```
b = (int**)malloc(10 * sizeof(int*))
```

- Για δέσμευση μνήμης για 20 ακέραιους σε κάθε “γραμμή” χρειάζεται

```
for(i = 0; i < 10; i++) {  
    b[i] = (int*)malloc(20 * sizeof(int))  
}
```

- Η διεύθυνση του `b[i][j]` είναι `b[i] + j`, ή
- η διεύθυνση του `b[i][j]` είναι `(char*)b[i] + j * sizeof(int)`, ή
- Πριν τη χρήση του `b` ως πίνακα πρέπει να έχει γίνει σωστή αρχικοποίηση για τους δείκτες `b[i]` και δέσμευση μνήμης

