Προγραμματισμός Υπολογιστών με C++

ΔΕΙΚΤΕΣ ΚΑΙ ΔΙΕΥΘΥΝΣΕΙΣ

Γεώργιος Παπαϊωάννου (2013-14)
gepap@aueb.gr

Περιεχόμενο Παρουσίασης

- Περιγραφή:
  - Εισαγωγή στους δείκτες
  - Το μοντέλο της μνήμης, σωρός και στοίβα
  - Αναφορές
  - Δείκτες και πίνακες
  - Δέσμευση και αποδέσμευση μνήμης

- Τελευταία ενημέρωση: Οκτώβριος 2015

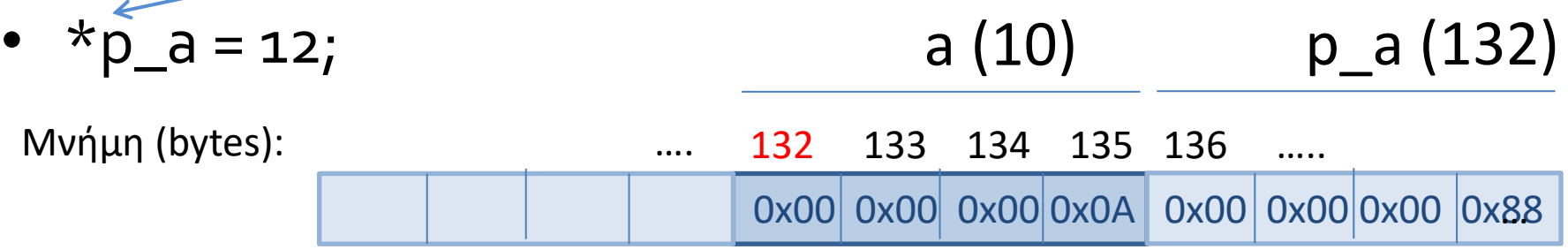
Τι Είναι ένας Δείκτης;

- Στη C++, έχουμε πρόσβαση στη διεύθυνση μιας μεταβλητής, την οποία και μπορούμε να αποθηκεύσουμε ως μεταβλητή κι αυτή (δείκτης):
- int a = 10;
- int * p_a = &a;
- *p_a = 12;

&: Η διεύθυνση στη μνήμη μιας μεταβλητής

int *: Μεταβλητή τύπου "Δείκτη" σε int (διεύθυνση μιας ποσότητας τύπου int)

*p_a: Το περιεχόμενο μιας διεύθυνσης, ή αλλιώς τα δεδομένα που «δείχνει» ο δείκτης



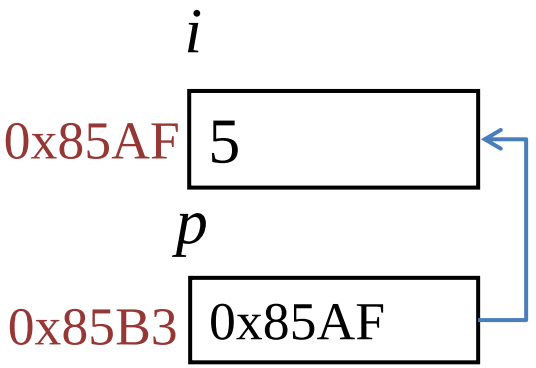
Γιατί Χρειάζομαι τους Δείκτες;

- Για τους ίδιους λόγους που στη Java χρειάζομαι τις αναφορές (είναι το ίδιο πράγμα!). Ποιοι είναι αυτοί;
- Για να έχω τυχαία προσπέλαση στη μνήμη
- Για να έχω ορατότητα σε κοινόχρηστα δεδομένα χωρίς να τα αντιγράφω συνέχεια
- Για να διαχειρίζομαι αντικείμενα και συναρτήσεις σε χαμηλό επίπεδο (π.χ. δείκτες σε συναρτήσεις!)

Fun with Pointers!

```
int i = 5;
cout << i;      // Τυπώνει 5.
cout << &i;     // Τυπώνει τη διεύθυνση της i: 0x85AF.

int* p = &i;    // Δείκτης σε ακέραιο.
cout << p;      // Τυπώνει 0x85AF.
cout << *p;     // Τυπώνει 5.

*p = 6;          // Αλλάζει το i.
cout << i;       // Τυπώνει 6.

int j = 10;
p = &j;         // Τώρα το p δείχνει στο j.
cout << *p;     // Τυπώνει 10.
```

p ⟶ i



p ⟶ j

Πράξεις με Δείκτες

- Ο δείκτες περιέχουν διευθύνσεις και άρα είναι φυσικοί αριθμοί!
- Άρα μπορώ να κάνω:
  - Βασικές πράξεις μεταξύ αριθμών και δεικτών ή μεταξύ δεικτών (όποιες έχουν νόημα): +, -, +=, -=, ++, --
  - Συγκρίσεις δεικτών (δηλ. συγκρίσεις διευθύνσεων)
- Ο τύπος της μεταβλητής που δείχνει ένας pointer καθορίζει και το increment των αριθμητικών πράξεων:

```
int a = 10;
int * p_a = &a;
p_a += 1;  // ΔΕΝ ανεβαίνει κατά ένα byte, αλλά κατά ένα int (4 bytes)
```

Ο Άκυρος Δείκτης

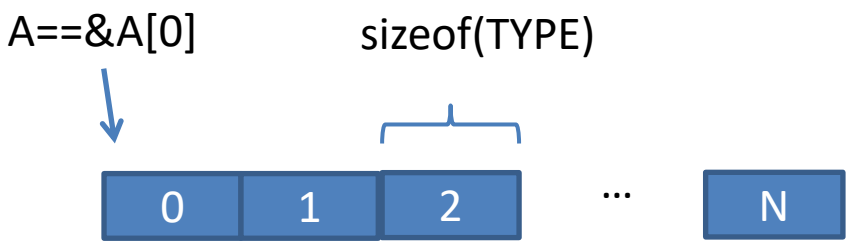
- Η άκυρη τιμή δείκτη είναι η μηδενική διεύθυνση:

```
0
'\0'
NULL
nullptr (C++11)
```

```
float * a = NULL;
```

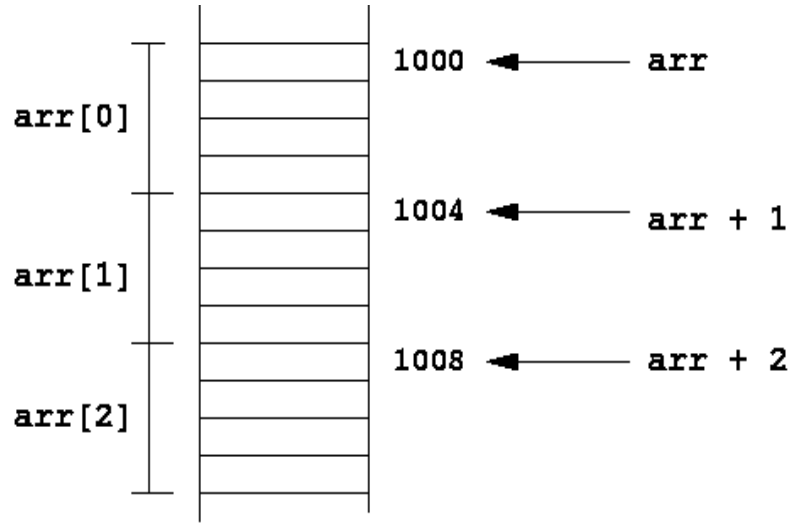
Πίνακες και Δείκτες

- Οι πίνακες είναι συνεχόμενες θέσεις μνήμης
- Ένας πίνακας A, N στοιχείων ξεκινά από μια διεύθυνση βάσης (base pointer): &A[0] == A (η διεύθυνση του πρώτου στοιχείου)
- Εκτείνεται για N X sizeof(TYPE) bytes
- Ο τελεστής [i] είναι ισοδύναμος με «το περιεχόμενο στη θέση i»,δηλαδή:
- A[i] = *(A+i)



Πίνακες και Δείκτες - Παραδείγματα

```
int arr[10];
bool b = (arr[2] == * (arr+2));
      // true


arr = arr + 1;
```



Αυτό δεν είναι επιτρεπτό γιατί εδώ
Ο arr είναι δηλωμένος ως σταθερός (arr[10]), άρα ναι μεν έχω πρόσβαση στη διεύθυνση αρχής, αλλά δε μπορώ να την αλλάξω

Δείκτες και Πίνακες Χαρακτήρων

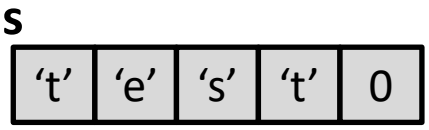
```
char buffer[80] = "σκουλικομερμιγκότρυπα";
char *sub = buffer+8;
cout << sub;   // "μερμιγκότρυπα"
```


```
for (int i = 0; i<=6; i++)
      *(buffer+8+i) = *(buffer+16+i);
       // buffer[8+i] = buffer[16+i];
cout << buffer;  // "σκουλικοτρυπα"
```

Άλλες Παραλλαγές Αρχικοποίησης Πινάκων

```
int arr[] = {1, 2, 3};          // Στατικός πίνακας.
int* arr = new int[3];          // Δυναμικός πίνακας.
int* arr = {1, 2, 3};           // Δεν επιτρέπεται.
int arr[] = new int[3];         // Δεν επιτρέπεται.

char s[] = "test";              // Στατικός πίνακας χαρακτήρων.
const char* s = "test";         // OK: const
char* s = "test";               // Δεν επιτρέπεται
```

Λεπτομέρεια: Δείκτες σε Σταθερή Μνήμη




```
char s[] = "test";
```

- Στην πρώτη σειρά, η συμβολοσειρά αποθηκεύεται σε έναν πίνακα που εκχωρείται γι αυτό το σκοπό και που έχει ακριβώς το μέγεθος της.
- Μπορώ να έχω πρόσβαση και να αλλάξω τα περιεχόμενα του s
- Δε μπορώ να μεταβάλλω την τιμή του s (const char *)

sizeof και Δείκτες

- Η sizeof μας δίνει το μέγεθος μιας μεταβλητής στη μνήμη
- Άρα για δείκτες μας δίνει πόσα bytes καταλαμβάνει η αναπαράσταση μιας διεύθυνσης:
  - Για 32bit αρχιτεκτονικές είναι 4
  - Για 64bit αρχιτεκτονικές είναι 8
- Παρατήρηση:

```
char s[] = "test";
```

- sizeof(s) == **5** // Το μέγεθος ενός σταθερού πίνακα

sizeof και Δείκτες

float a[4];

sizeof(a) / sizeof(float) ← 4:

- ο αριθμός των στοιχείων του πίνακα προκύπτει από:

Bytes που καταλαμβάνει ο πίνακας / bytes του κάθε στοιχείου

Ερωτήσεις

- Πως μπορώ να φτιάξω έναν πίνακα από δείκτες;
- Πως μπορώ να έχω πρόσβαση στα στοιχεία του;
- Ποια χρήση έχει κάτι τέτοιο;

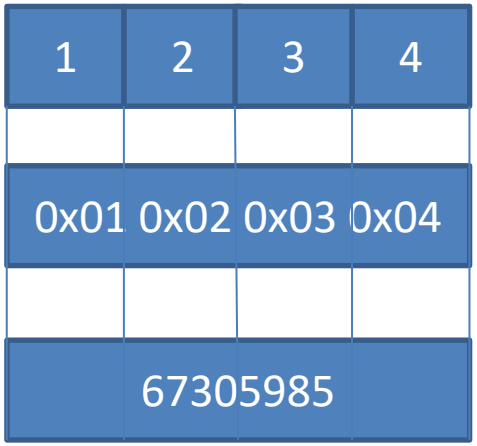
Ο Τύπος void*

- Συχνά, χρειάζεται να περνάμε δεδομένα σε περιοχές μνήμης χωρίς να ξέρουμε τον ακριβή τύπο τους
- Πότε;
  - Όταν διαβάζουμε δεδομένα που τα προσπερνάμε ή τα μεταβιβάζουμε χωρίς να μας απασχολεί το περιεχόμενό τους (π.χ.network package data)
  - Όταν θέλουμε να ορίσουμε παραμέτρους σε συναρτήσεις ή μεθόδους χωρίς να ξέρουμε τον ακριβή τύπο που θα περαστεί, αλλά μπορούμε να τον συμπεράνουμε εσωτερικά

    (π.χ. void ThreadProcess(void *pData))

Χρησιμοποιώντας το Δείκτη σε void

- Επειδή δεν έχει ακριβή τύπο το void *, οποιαδήποτε πράξη πάνω σε έναν τέτοιο δείκτη γίνεται με μονάδα το byte
- Τη μνήμη που δείχνει ένας δείκτης μπορώ να την καλουπώσω σε οτιδήποτε!
- char a[4] = {1,2,3,4};
- void * p_a = a;
- int b = *((int*)p_a); // copy
- // b == 67305985



Τι διαφορές Εντόπισα σε Σχέση με τη Java;

C++	Java
Ο δείκτης είναι ισοδύναμος με τη διεύθυνση στη μνήμη μιας μεταβλητής και χρησιμεύει ως αναφορά σε αυτή, διαχωρίζοντάς τη από το περιεχόμενό της	Υπάρχει μόνο η έννοια της «αναφοράς» σε αντικείμενα, που ισοδυναμεί με ένα δείκτη σε αντικείμενα στη C++
Αν δεν ξέρουμε τον τύπο ενός ορίσματος όταν δηλώνουμε μια συνάρτηση/μέθοδο, περνάμε απλά μια διεύθυνση μεταβλητής: void *	Αν δεν ξέρουμε τον τύπο ενός ορίσματος όταν δηλώνουμε μια μέθοδο, περνάμε απλά μια αναφορά σε Object

Μεταβλητές και Μνήμη

- Κάθε πρόγραμμα όταν εκτελείται, δεσμεύει μνήμη σε δύο φάσεις:
  - Στατικά (με την εκκίνηση)
  - Δυναμικά (κατά την εκτέλεσή του)
- Στην αρχικοποίηση του προγράμματος δεσμεύεται κατευθείαν ο χώρος που είναι γνωστός από πριν:
  - Χώρος για τον δυαδικό κώδικα μηχανής
  - Χώρος για στατικές μεταβλητές
  - Χώρος για την υπογραφή των κλάσεων κλπ...
- Όλες οι μεταβλητές δεσμεύουν χώρο δυναμικά!

Χώροι Δέσμευσης Μνήμης

- Κάθε πρόγραμμα δεσμεύει 5 τύπους χώρου μνήμης:
- Χώρος εκτελέσιμου δυαδικού κώδικα μηχανής (στατικός). Λέγεται code segment ή text segment (παρότι δεν είναι κείμενο!)
- Αρχικοποιημένος χώρος δεδομένων (data segment). Περιέχει στατικές μεταβλητές, υπογραφές κλασεων, global μεταβλητές (στατικός)
- Χώρος μη αρχικοποιημένων δεδομένων (BSS)
- Σωρός (heap), για δυναμική δέσμευση μνήμης (π.χ. με new)
- Στοίβα (call stack) για τις αυτόματες (τοπικές) μεταβλητές και τις παραμέτρους κλήσης συναρτήσεων

Το Εκτελέσιμο Αρχείο και η Μνήμη

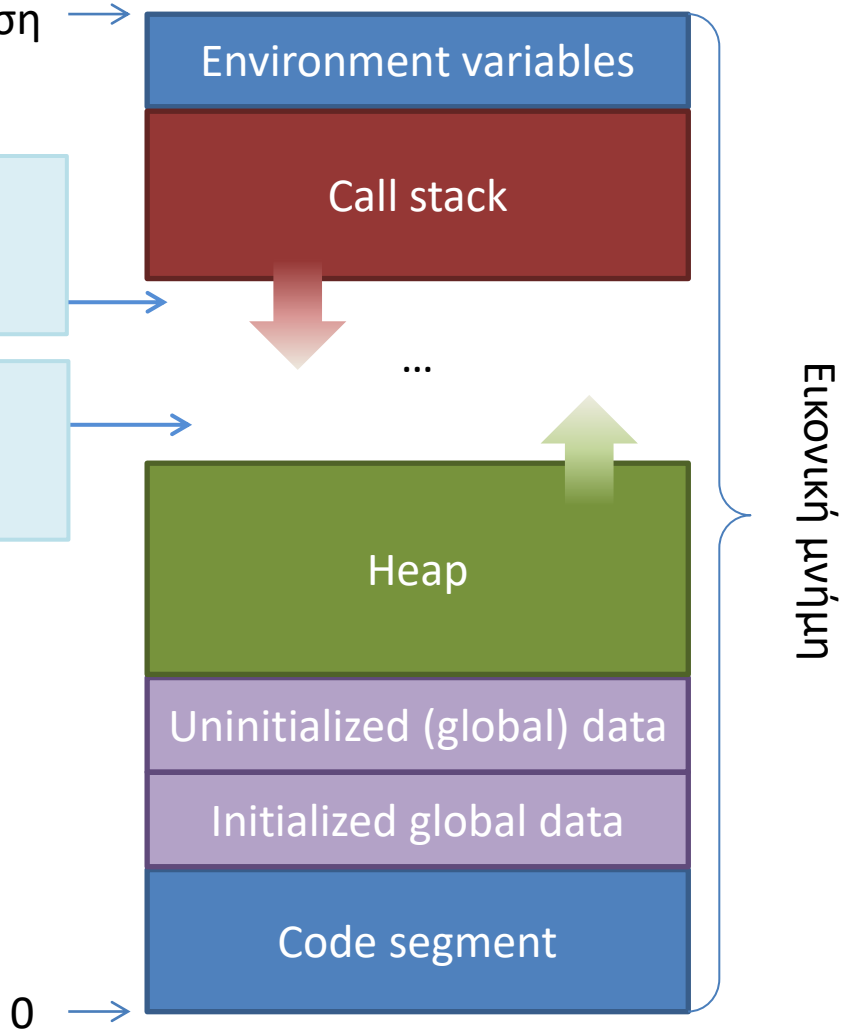
- Ένα εκτελέσιμο αρχείο περιλαμβάνει, εκ κατασκευής από το μεταγλωττιστή και συνδέτη, όλη τη δυαδική πληροφορία του code segment και data segment
- Η δυαδική αναπαράσταση φορτώνεται στο χώρο (virtual memory space) που εκχωρείται από το λειτουργικό, ως έχει
- Η δυναμική περιοχή μνήμης δεν αποθηκεύεται στο αρχείο προφανώς, αλλά καθορίζεται και δεσμεύεται καθώς εκτελείται ο κώδικας

Οργάνωση της Μνήμης μιας Διεργασίας

Μέγιστη εικονική διεύθυνση
(κάτι μεγάλο, π.χ. 2^{39})

Η στοίβα μεγαλώνει προς τα κάτω (μικρότερες διευθύνσεις – x86 αρχιτεκτονικές)

Ο σωρός δεσμεύεται κι αυξάνει προς τα πάνω (προς μεγαλύτερες διευθύνσεις μνήμης)



Η Στοίβα Κλήσεων Συναρτήσεων

- Κάθε φορά που καλείται μια συνάρτηση (ή μέθοδος), ακόμα και η main(), χρησιμοποιείται χώρος στην περιοχή της στοίβας κλήσεων συναρτήσεων (call stack ή σκέτο stack)
- Ο χώρος μνήμης της στοίβας χρησιμοποιείται αποκλειστικά γι αυτή τη χρήση
- Έχει πολύ απλό μηχανισμό (στοίβας…) για τη δέσμευση και αποδέσμευση (rewind) χώρου
- Αποθηκεύονται:
  - Όλες οι τοπικές μεταβλητές μιας συνάρτησης
  - Οι παράμετροι που περάστηκαν
  - Το σημείο εξόδου και επαναφοράς στην καλούσα συνάρτηση (θέση μνήμης)

Τι διαφορές Εντόπισα σε Σχέση με τη Java;

C++

Ο κώδικάς μου παράγει εκτελέσιμο που φορτώνεται όπως στο παραπάνω σχήμα και εκτελείται απευθείας πάνω στο λειτουργικό σύστημα

Java

Ο κώδικάς μου είναι στην ουσία τα «δεδομένα» ενός άλλου προγράμματος (JVM), γραμμένου σε C++ (:D) που αυτό τρέχει στο λειτουργικό μου σύστημα (*)

Η Java υλοποιεί στο χώρο μνήμης δεδομένων της, heap, stacks και code segments για τα threads των Java προγραμμάτων που εκτελούνται

(*) http://www.artima.com/insidejvm/ed2/jvm.html

Η Στοίβα – Παράδειγμα (1)

```
static int s_f = 3.1f;

float Foo(float a, float b) {
    float c = a+b;
    return c*c;
}

int main() {
    float data[] = {.1f, .2f};
    data[0] = Foo(data[1], s_f);
    cout << data[0];
    return 0;
}
```

Μεταγλώττιση, σύνδεση και φόρτωμα

Μετά το φόρτωμα του εκτελέσιμου

exe path:= “c:\temp\test.exe”
args = “test.exe”

...

s_f := 3.1
cout

- main() machine code
- Foo() machine code
- osteam::<<() machine code

Η Στοίβα – Παράδειγμα (2)

```
static int s_f = 3.1f;

float Foo(float a, float b) {
    float c = a+b;
    return c*c;
}

int main() {
    float data[] = {.1f, .2f};
    data[0] = Foo(data[1], s_f);
    cout << data[0];
    return 0;
}
```

Έναρξη εκτέλεσης

exe path:= “c:\temp\test.exe”
args = “test.exe”

...

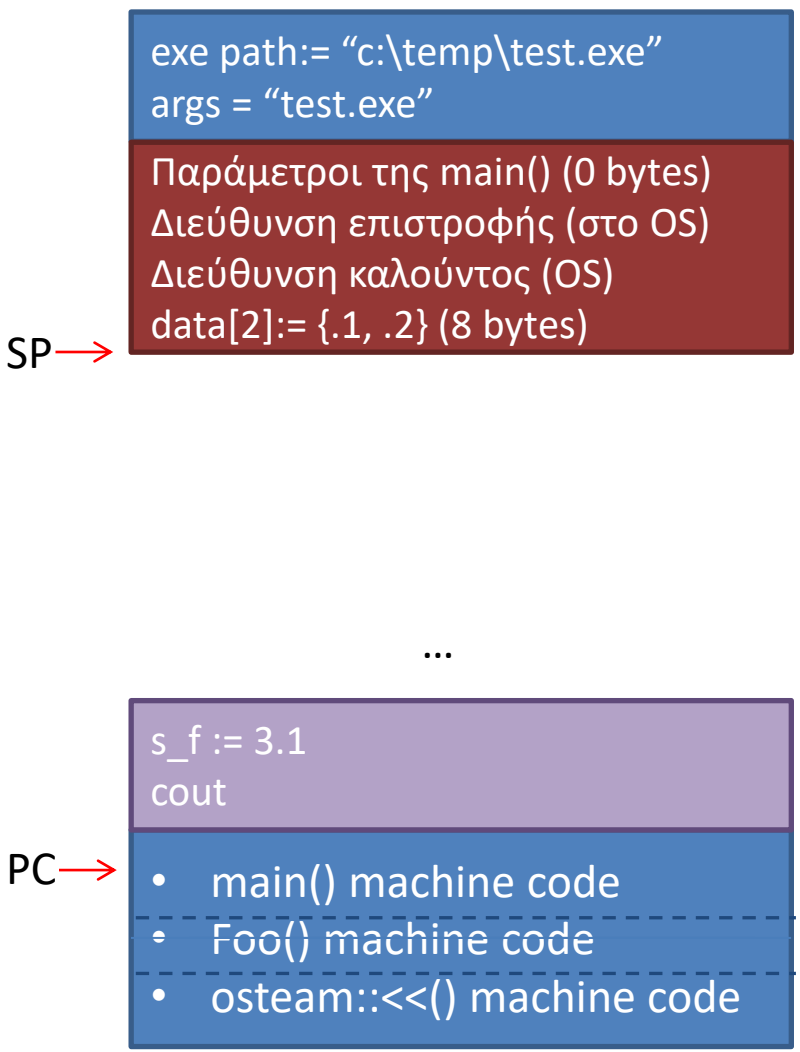
PC →

s_f := 3.1
cout

- main() machine code
- ~~Foo() machine code~~
- osteam::<<() machine code

Η Στοίβα – Παράδειγμα (3)

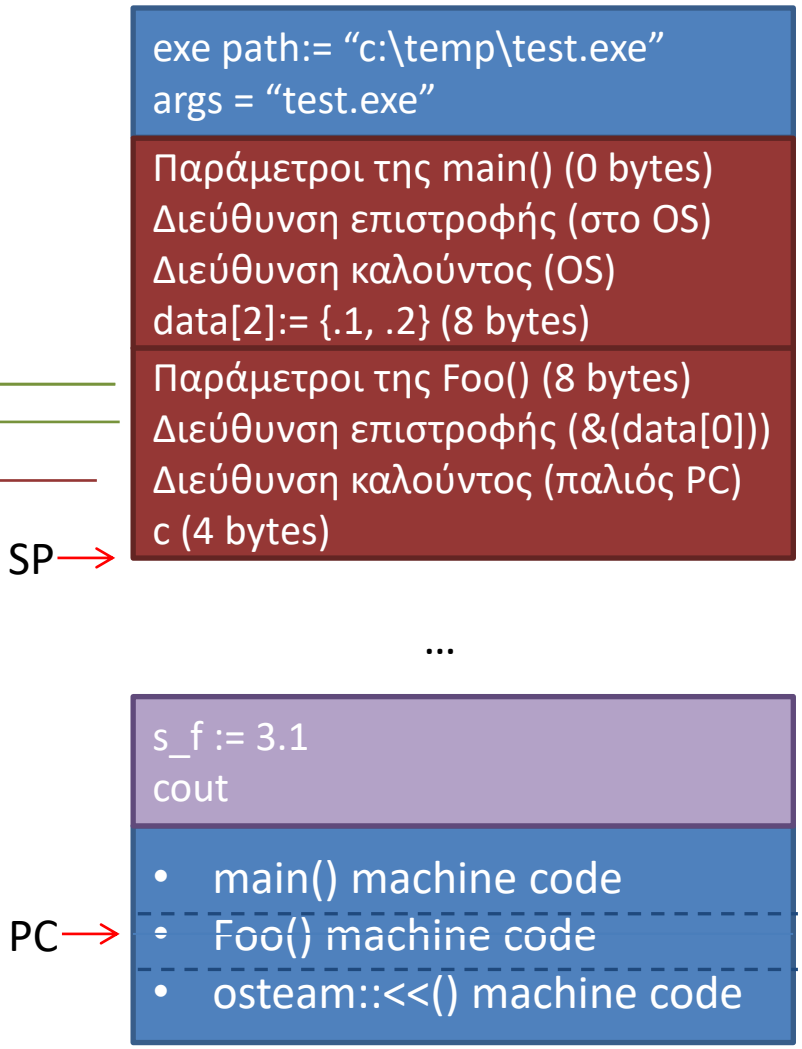
```
static int s_f = 3.1f;

float Foo(float a, float b) {
    float c = a+b;
    return c*c;
}

int main() {
    float data[] = {.1f, .2f};
    data[0] = Foo(data[1], s_f);
    cout << data[0];
    return 0;
}
```



Η Στοίβα – Παράδειγμα (4)

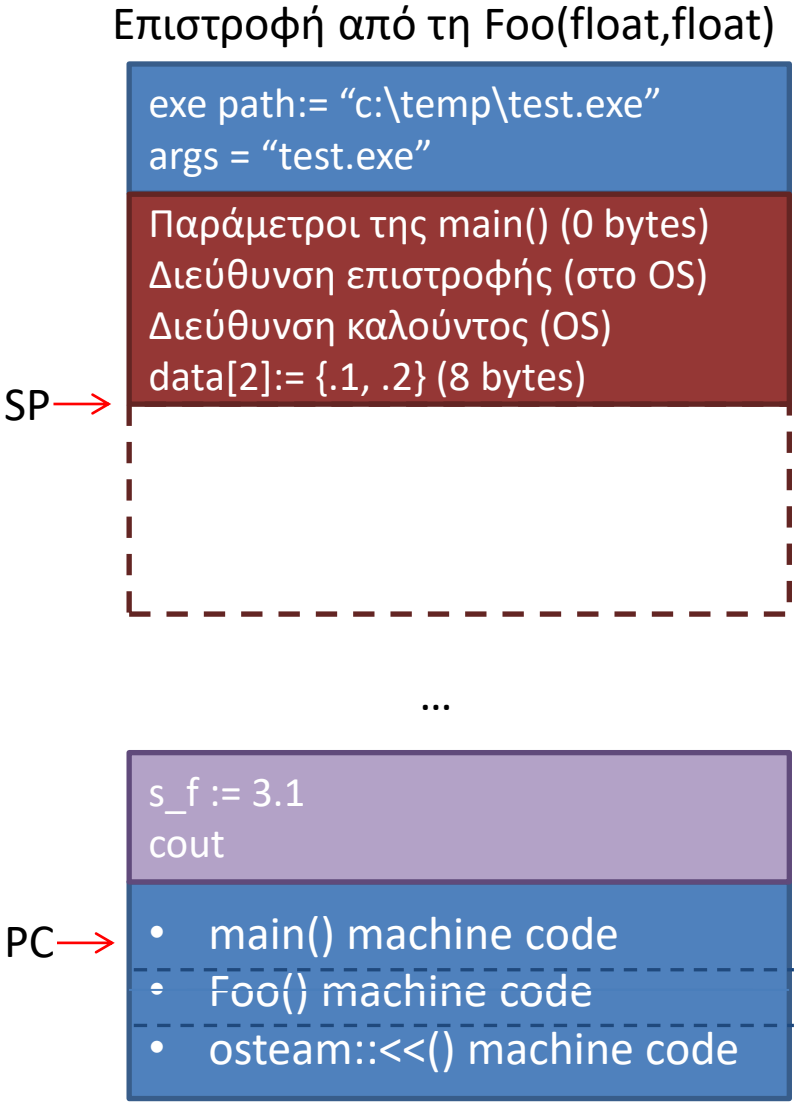
```
static int s_f = 3.1f;

float Foo(float a, float b) {
    float c = a+b;
    return c*c;
}

int main() {
    float data[] = {.1f, .2f};
    data[0] = Foo(data[1], s_f);
    cout << data[0];
    return 0;
}
```

Κλήση της Foo(float,floar)



Η Στοίβα – Παράδειγμα (3)

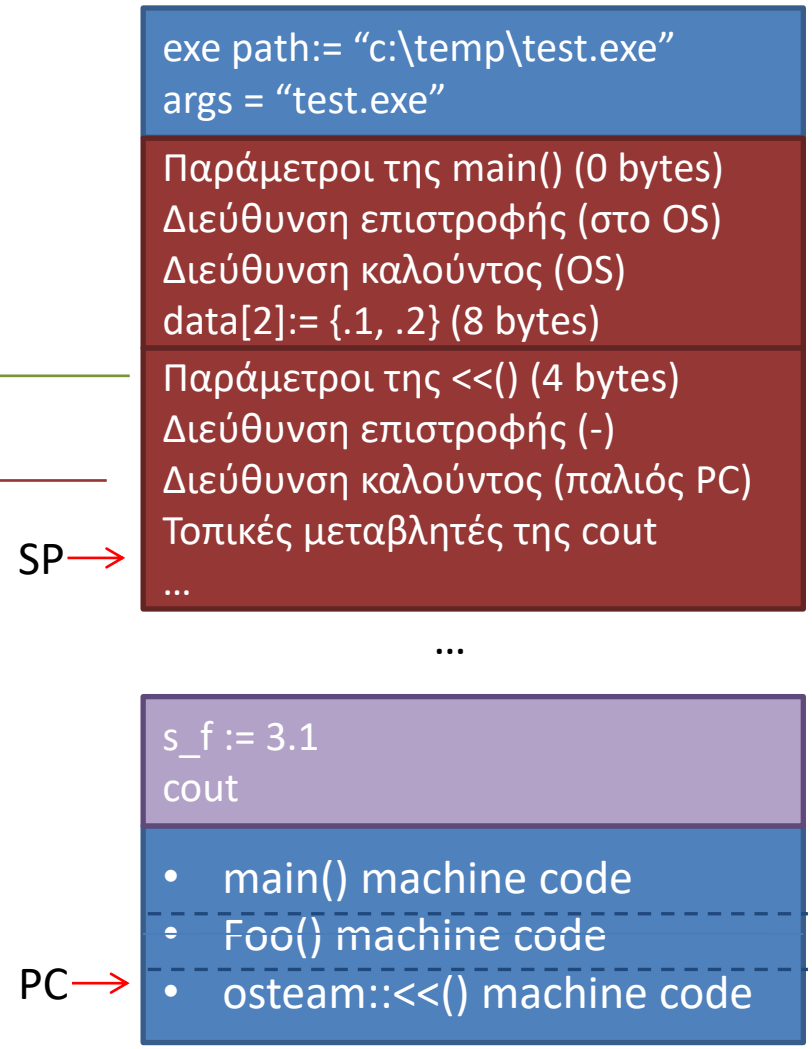
```
static int s_f = 3.1f;

float Foo(float a, float b) {
    float c = a+b;
    return c*c;
}

int main() {
    float data[] = {.1f, .2f};
    data[0] = Foo(data[1], s_f);
    cout << data[0];
    return 0;
}
```



Η Στοίβα – Παράδειγμα (6)

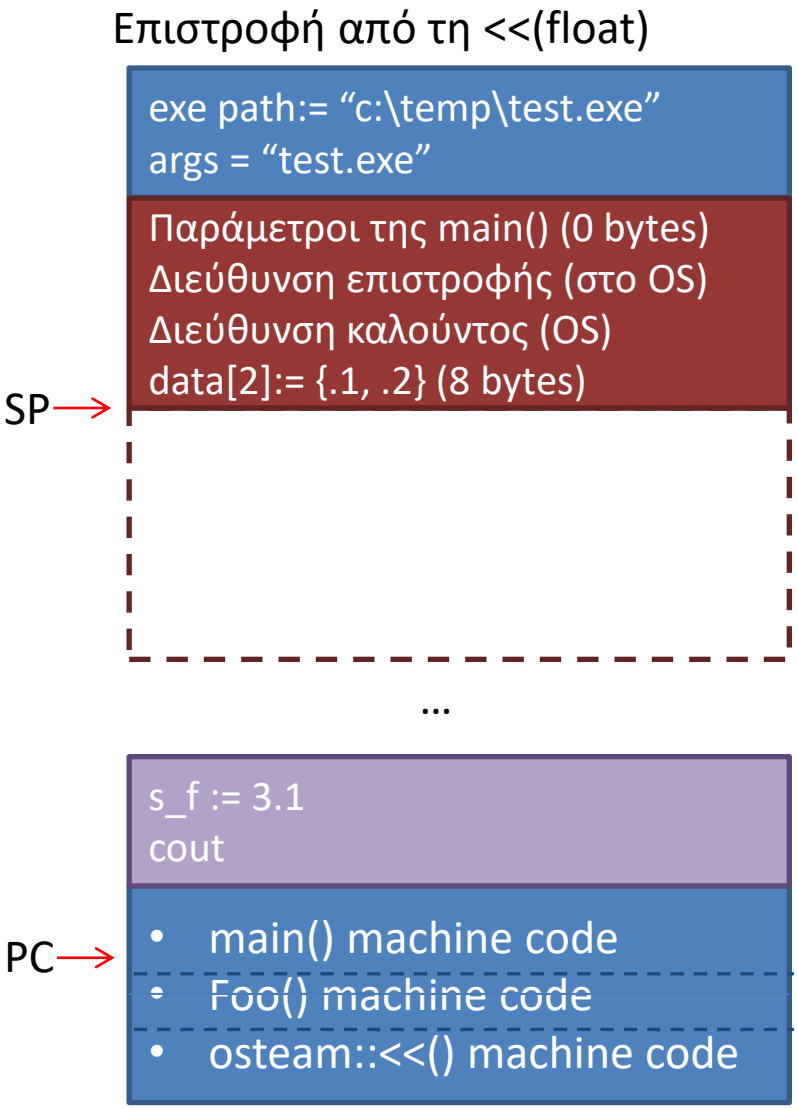
```
static int s_f = 3.1f;

float Foo(float a, float b) {
    float c = a+b;
    return c*c;
}

int main() {
    float data[] = {.1f, .2f};
    data[0] = Foo(data[1], s_f);
    cout << data[0];
    return 0;
}
```

Κλήση της ostream::<<(float)



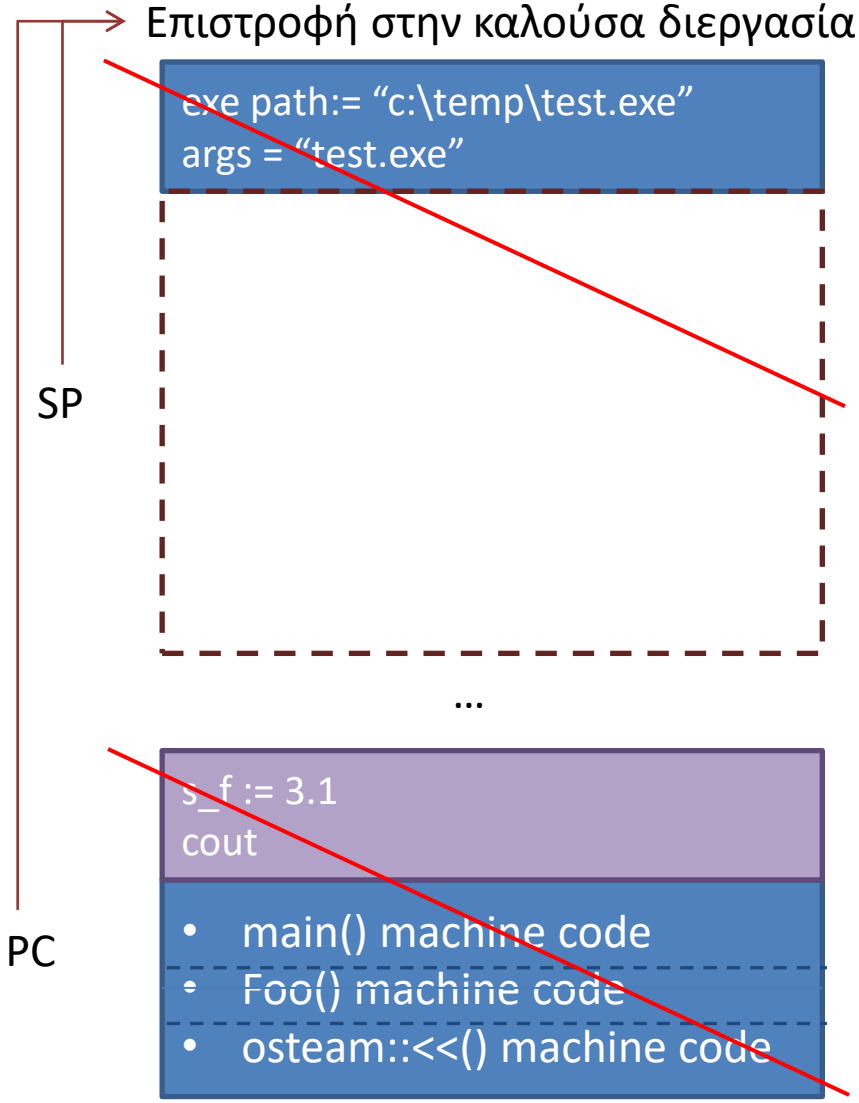
Η Στοίβα – Παράδειγμα (3)

```
static int s_f = 3.1f;

float Foo(float a, float b) {
    float c = a+b;
    return c*c;
}

int main() {
    float data[] = {.1f, .2f};
    data[0] = Foo(data[1], s_f);
    cout << data[0];
    return 0;
}
```



Η Στοίβα – Παράδειγμα (3)

```
static int s_f = 3.1f;

float Foo(float a, float b) {
    float c = a+b;
    return c*c;
}

int main() {
    float data[] = {.1f, .2f};
    data[0] = Foo(data[1], s_f);
    cout << data[0];
    return 0;
}
```



Αρχικοποίηση Μεταβλητών και Πεδίων

- Προσοχή: Για λόγους ταχύτητας, στη C++ και στη C δεν αποδίδονται τυπικές τιμές κατά τη δήλωση μεταβλητών.
- Εμείς είμαστε υπεύθυνοι για την απόδοση αρχικής τιμής

```
int c;
cout << c;  //  3435973836 (0xcccccccc) : σκουπίδια
```

- Γιατί;
  - Αν η μεταβλητή θα υποδεχθεί ένα αποτέλεσμα ή μία ανάθεση αργότερα δεν υπάρχει λόγος να εκτελεστεί η αρχικοποίηση της σε μια τιμή που ποτέ δε θα χρησιμοποιηθεί

Δυναμική Εκχώρηση Μνήμης - Λόγοι

- Είδαμε ότι οι τοπικές μεταβλητές δεσμεύονται σε κάθε stack frame κλήσης μιας συνάρτησης. Όταν η συνάρτηση επιστρέφει, τα δεδομένα αυτά χάνονται
- Συχνά δεν είμαστε σε θέση να προσδιορίσουμε από την αρχή το μέγεθος ή και τον ακριβή τύπο πινάκων και δεδομένων (βλ. πίνακες και κληρονομικότητα)
- Θέλουμε να δεσμεύσουμε μεταβλητές σε χώρο που να έχουν πρόσβαση πολλές συναρτήσεις/μέθοδοι/νήματα
- Θέλουμε να δημιουργήσουμε μη συνεκτικά δεδομένα στη μνήμη (συνδεδεμένες λίστες, όχι πίνακες)
- Θέλουμε να δεσμεύσουμε μεγάλη ποσότητα μνήμης, που δε γίνεται να μπει σε ένα frame (stack overflow)
- …

Ο Σωρός

- Είναι το διάστημα μνήμης που το πρόγραμμά μας μπορεί να δεσμεύσει χώρο για δεδομένα δυναμικά
- Είναι ξεχωριστός από το χώρο των αυτόματων μεταβλητών (call stack)
- Η δέσμευση χώρου γίνεται με την εντολή new στη C++ (όπως και στη Java)
- Τη μνήμη που δεσμεύουμε, **πρέπει να την αποδεσμεύουμε εμείς** με την εντολή delete – ΔΕΝ υπάρχει garbage collection στη C++ (*)

(*) Υπάρχει υποστήριξη μέσω της STL (π.χ. shared_ptr) για δείκτες που καταστρέφονται μόνοι τους, αλλά αυτό έχει πολυπλοκότητα και κόστος (όπως και στη Java!)

Η Εντολές new και delete

```
float* fp = new float; // Δημιουργία νέου χώρου
                       // αποθήκευσης για μία τιμή float
*fp = 2.0F;            // Ανάθεση τιμής float στο
                       // περιεχόμενο του fp
cout << fp << endl     // Τυπώνει 0xFF10 (διεύθυνση)
     << *fp << endl;   // Τυπώνει 2 (τιμή)
delete fp;             // Αποδέσμευση του χώρου στον
                       // οποίο δείχνει ο fp
```

- Σε αυτή την περίπτωση ο δείκτης δεν δείχνει στο χώρο μιας άλλης μεταβλητής, αλλά σε χώρο που δημιουργήσαμε με new

Τι Κάνουν οι new και delete;

- Για απλούς τύπους (float, int κλπ), δεσμεύουν και αποδεσμεύουν μνήμη
- Για κλάσεις (με αυτή τη σειρά):
  - new:
    - Δεσμεύει μνήμη, όση χρειάζεται για να αποθηκευτεί ένα στιγμιότυπο
    - Καλεί τον κατασκευαστή της κλάσης
  - delete:
    - Καλεί τον καταστροφέα της κλάσης
    - Αποδεσμεύει τη μνήμη που είχε εκχωρηθεί

Δέσμευση Πινάκων

- Για να δεσμεύσουμε μια ακολουθία από ίδιου τύπου δεδομένα χρησιμοποιούμε τον τελεστή new[]:

  float *array = new float[num];

- Και για να διαγράψουμε τον πίνακα στοιχείων αυτόν χρησιμοποιούμε την delete[]:

  delete[] array;

Δείκτες και Πίνακες - Παραδείγματα

```
int a[5];                // Στατική καταχώριση χώρου για πίνακα.
int* b = new int[5];     // Δυναμική καταχώριση χώρου για πίνακα.
a[4] = 10;
b[4] = 10;
*a = 2;                  // Ισοδύναμο του a[0] = 2.
*b = 2;                  // Ισοδύναμο του b[0] = 2.
*(a + 1) = 4;            // Ισοδύναμο του a[1] = 4.
*(b + 1) = 4;            // Ισοδύναμο του b[1] = 4.
b = a;                   // Δεν είναι λάθος, χάσαμε όμως τη διεύθυνση του
                         // δυναμικού πίνακα. Πώς θα κάνουμε delete[];
int* c = a;              // Τα ονόματα πινάκων είναι ουσιαστικά δείκτες.
a = b;                   // Τα ονόματα πινάκων είναι δείκτες που δεν
                         // επιτρέπεται να δείξουν αλλού.
```

new/delete – Παράδειγμα (1)

```
char * CreateData(int num)
{
  char * data = new char[num];
  for (int i=0; i<num; i++)
    data[i] = '\0';
  return data;
}

 int main() {
    char * str = CreateData(10);
    str[0] = 'A';
    cout << str;
    delete[] str;
    return 0;
 }
```

Stack

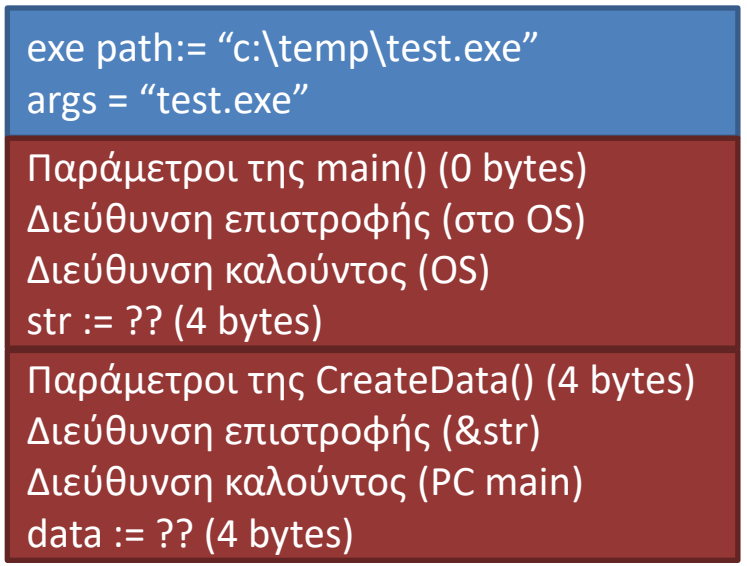
exe path:= “c:\temp\test.exe”
args = “test.exe”

Παράμετροι της main() (0 bytes)
Διεύθυνση επιστροφής (στο OS)
Διεύθυνση καλούντος (OS)
char *:= ?? (4 bytes)

new/delete – Παράδειγμα (2)

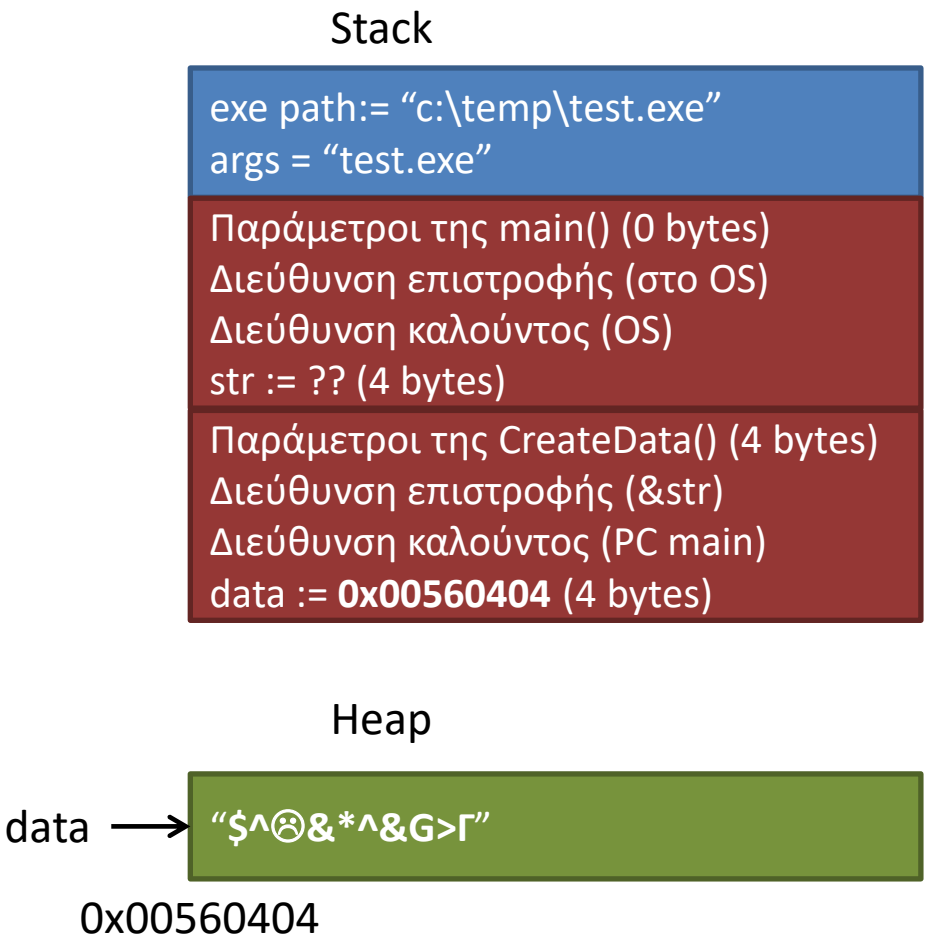
```
char * CreateData(int num)
{
  char * data = new char[num];
  for (int i=0; i<num; i++)
    data[i] = '\0';
  return data;
}

 int main() {
    char * str = CreateData(10);
    str[0] = 'A';
    cout << str;
    delete[] str;
    return 0;
 }
```

Stack



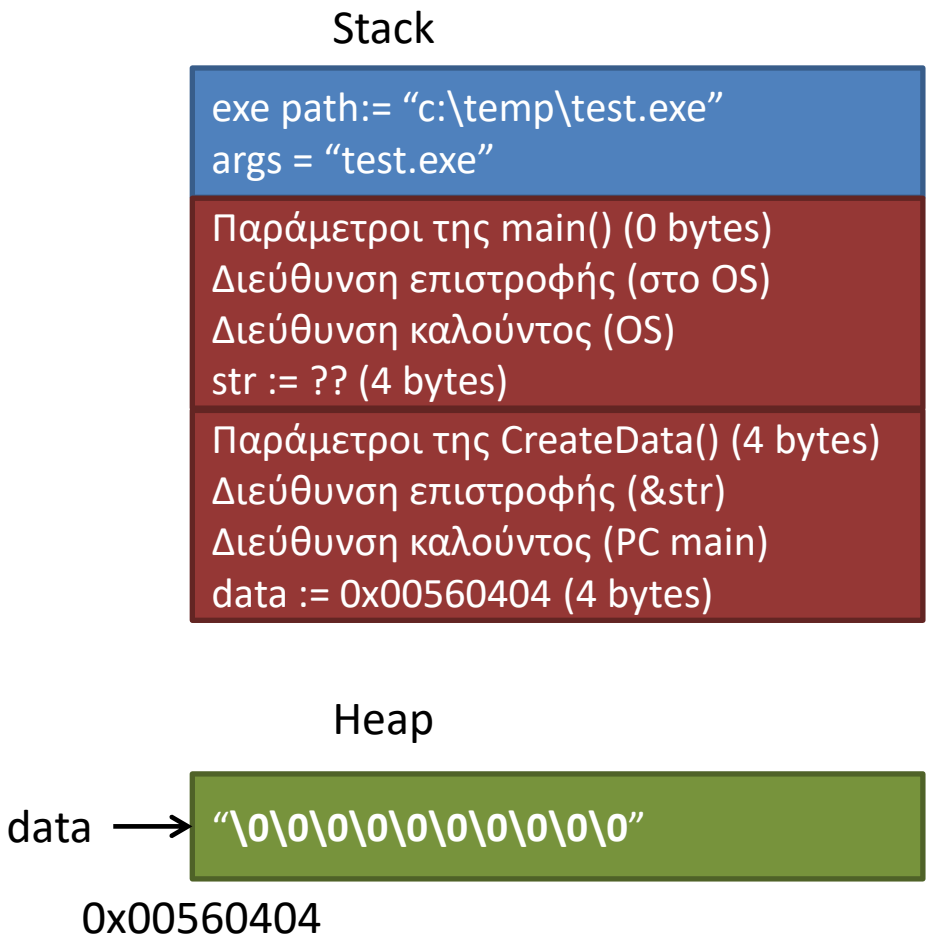
new/delete – Παράδειγμα (3)

```
char * CreateData(int num)
{
  char * data = new char[num];
  for (int i=0; i<num; i++)
    data[i] = '\0';
  return data;
}

 int main() {
    char * str = CreateData(10);
    str[0] = 'A';
    cout << str;
    delete[] str;
    return 0;
 }
```



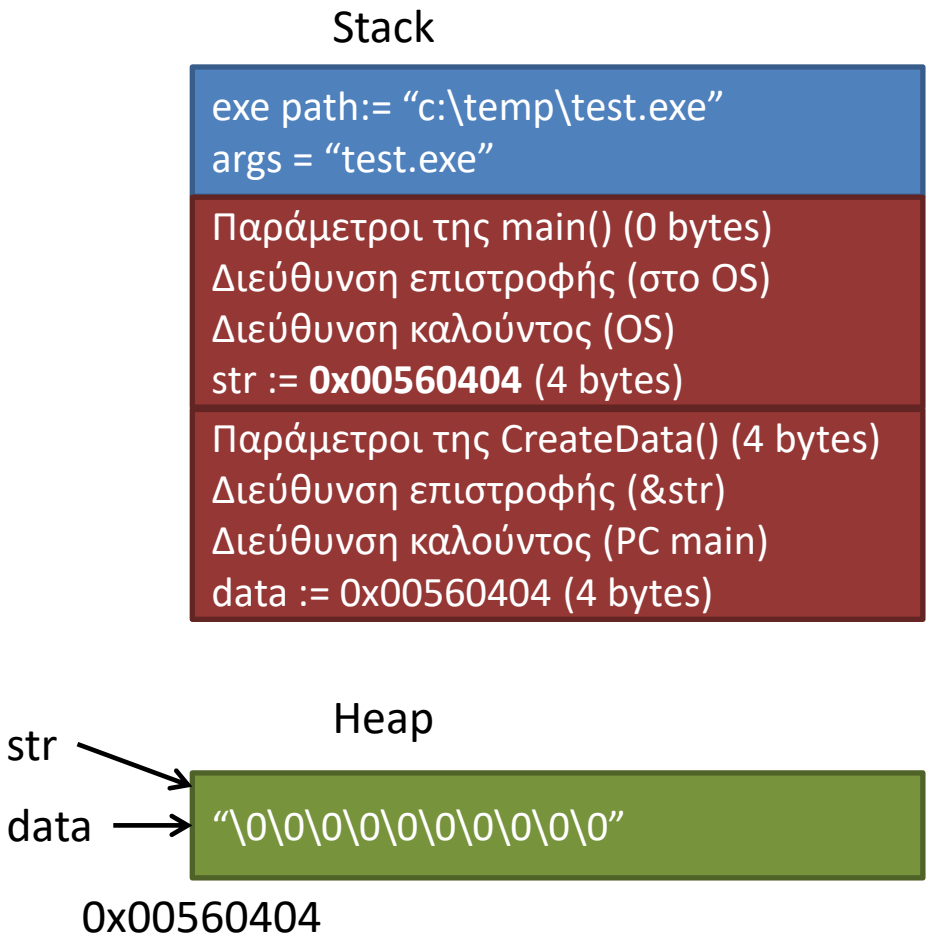
new/delete – Παράδειγμα (4)

```
char * CreateData(int num)
{
  char * data = new char[num];
  for (int i=0; i<num; i++)
    data[i] = '\0';
  return data;
}

 int main() {
    char * str = CreateData(10);
    str[0] = 'A';
    cout << str;
    delete[] str;
    return 0;
 }
```



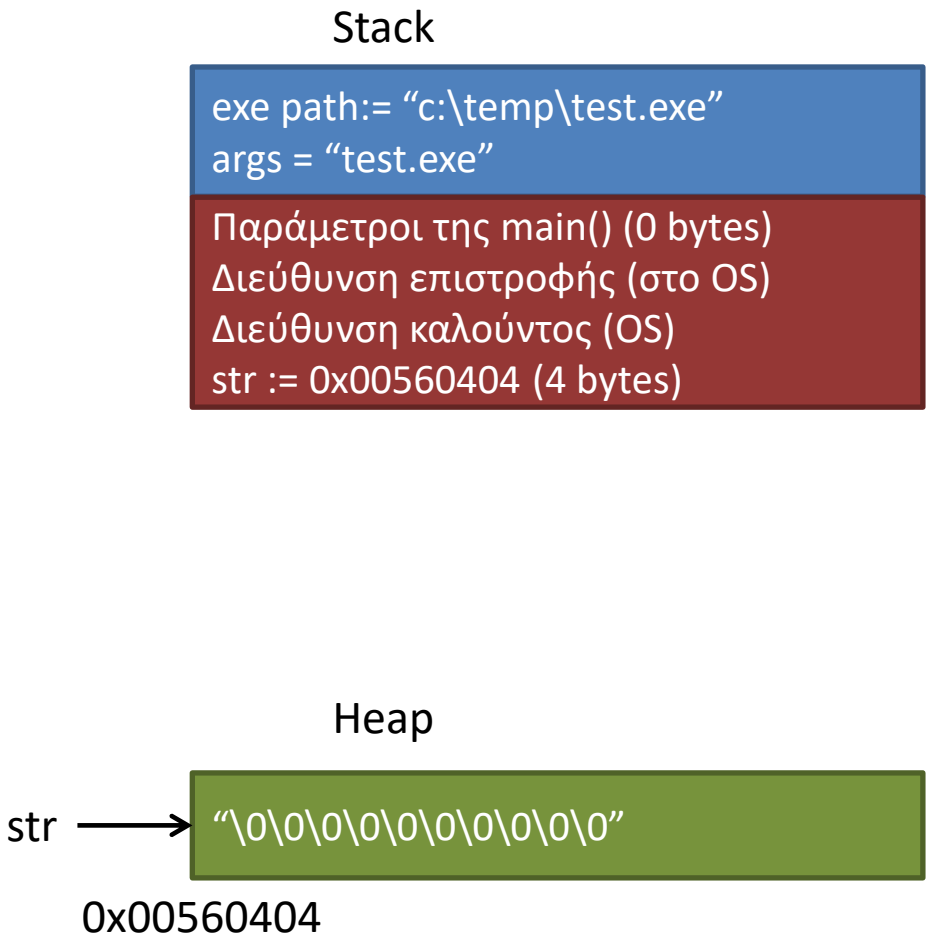
new/delete – Παράδειγμα (5)

```
char * CreateData(int num)
{
  char * data = new char[num];
  for (int i=0; i<num; i++)
    data[i] = '\0';
  return data;
}

 int main() {
    char * str = CreateData(10);
    str[0] = 'A';
    cout << str;
    delete[] str;
    return 0;
 }
```



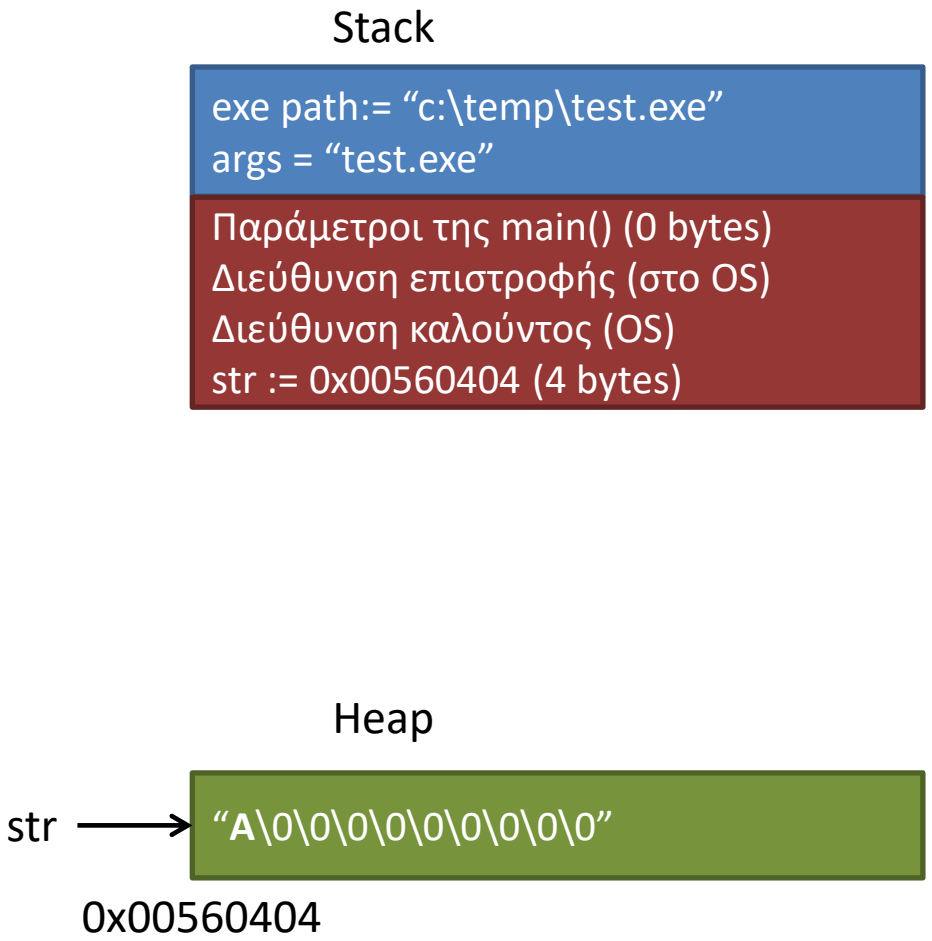
new/delete – Παράδειγμα (6)

```
char * CreateData(int num)
{
  char * data = new char[num];
  for (int i=0; i<num; i++)
    data[i] = '\0';
  return data;
}

 int main() {
    char * str = CreateData(10);
    str[0] = 'A';
    cout << str;
    delete[] str;
    return 0;
 }
```



new/delete – Παράδειγμα (7)

```
char * CreateData(int num)
{
  char * data = new char[num];
  for (int i=0; i<num; i++)
    data[i] = '\0';
  return data;
}

 int main() {
    char * str = CreateData(10);
    str[0] = 'A';
    cout << str;
    delete[] str;
    return 0;
 }
```



new/delete – Παράδειγμα (8)

```
char * CreateData(int num)
{
  char * data = new char[num];
  for (int i=0; i<num; i++)
    data[i] = '\0';
  return data;
}

 int main() {
    char * str = CreateData(10);
    str[0] = 'A';
    cout << str;
    delete[] str;
    return 0;
 }
```

Stack

exe path:= “c:\temp\test.exe”
args = “test.exe”

Παράμετροι της main() (0 bytes)
Διεύθυνση επιστροφής (στο OS)
Διεύθυνση καλούντος (OS)
str := **????** (4 bytes)

Heap

Δεν ξεχνάμε να καλέσουμε τη delete, αλλιώς η μνήμη δε θα αποδεσμευθεί! (Memory leak)

Παρατηρήσεις

- Η μνήμη των αυτόματων μεταβλητών, δηλαδή των τοπικών μεταβλητών μέσα στο σώμα μιας συνάρτησης ή μεθόδου, διαγράφεται μόνη της με την επιστροφή της συνάρτησης (επιστρέφει ο Stack Pointer στην παλιά του τιμή)
- Η μνήμη που δεσμεύουμε στο σωρό παραμένει δεσμευμένη και μετά την επιστροφή της συνάρτησης

Ένα πιο Σύνθετο Παράδειγμα με Δυν. Πίνακες

```
#include <iostream>
using namespace std;

int main() {

   unsigned howMany; cin >> howMany;
   int* storage = new int[howMany];

   for(unsigned i = 0; i < howMany; i++) {
      cin >> storage[i];
   }
   for(unsigned i = 0; i < howMany; i++) {
      cout << i << ": " << storage[i] << endl;
   }
   delete[] storage;
}
```

Δείκτες σε Σταθερές

```
✓ const int i2 = 10;
X int* p2 = &i2;          // Θα μπορούσα να αλλάξω το i2
                          // μέσω του p2, με *p2 = 20.
✓ const int* p3 = &i2;    // OK. Υπόσχομαι να μην αλλάξω
                          // αυτό στο οποίο δείχνει ο p3.
✓ cout << *p3;            // OK. Τυπώνει 10.
X *p3 = 20;               // Δεν επιτρέπεται. Δείκτης σε const.
✓ int  i1 = 10;           // Το i1 δεν είναι σταθερά.
✓ const int* p4 = &i1;    // Επιτρέπεται, αλλά δεν μπορώ να
                          // αλλάξω το i1 μέσω του p4.
✓ i1 = 20;                // OK.
X *p4 = 20;               // Δεν επιτρέπεται.
```

Σταθεροί Δείκτες

```
✓ int j1 = 10, j2 = 20;
✓ int* const p5 = &j1;  // Σταθερός δείκτης στο j1.
✓ *p5 = 30;             // OK, αλλάζει το j1.
X p5 = &j2;             // Ο p5 δεν μπορεί να δείξει αλλού.

✓ const int* const  p6 = &j1; // Σταθερός δείκτης σε
                              // σταθερά.
X p6 = &j2;                    // Ο p6 είναι σταθερός: δεν
                               // μπορεί να δείξει αλλού.
X *p6 = 30;                    // Ο p6 δείχνει σε σταθερά:
                              // δεν μπορεί να αλλάξει τα
                              // περιεχόμενα της θέσης
                              // μνήμης στην οποία δείχνει.
```

Δέσμευση Μνήμης στη C

- Η αντίστοιχη δέσμευση / αποδέσμευση μνήμης μπορεί να γίνει με τις συναρτήσεις της C:
  - void * malloc(num_bytes) και
  - void free(void *):
- float * data = (float*) malloc(10*sizeof(float));
- free (data);

- ΠΟΤΕ δεν αναμιγνύουμε τη new με την free και την malloc με την delete (είναι σφάλμα)

Σημείωση- Μέγεθος του Δείκτη

- Ένας δείκτης (διεύθυνση), ως μεταβλητή απαιτεί κάποιο χώρο αποθήκευσης, π.χ. 32bits
- Το μέγεθος του δείκτη μπορούμε να το πάρουμε κι αυτό με τη sizeof: π.χ. sizeof(float*)
- ΔΕΝ πρέπει να υποθέτουμε μόνοι μας ποτέ πόσο είναι το μέγεθος του pointer γιατί αλλάζει μεταξύ αρχιτεκτονικών (π.χ. στη x64 είναι 64bits)