Προγραμματισμός II (ECE116)

#10 δείκτες σε συναρτήσεις / μετα-συναρτήσεις (function pointers / meta-functions)

Σπύρος Λάλης
lalis@uth.gr

Συνάρτηση

- Ομάδα εντολών που γράφουμε ξεχωριστά
  - ως υπο-πρόγραμμα (subroutine)
- Για καλύτερη / πιο καθαρή δόμηση του κώδικα
- Για να εκτελούμε αυτές τις εντολές πολλές φορές, **χωρίς να τις ξαναγράφουμε** κάθε φορά από την αρχή στο πρόγραμμα μας
- Για να μπορεί αυτή η λειτουργικότητα να χρησιμοποιηθεί (και) **από άλλα προγράμματα**
- Μια συνάρτηση συνήθως έχει **παραμέτρους**
  - έτσι ώστε μια «βασική» διαδικασία υπολογισμού / επεξεργασίας να μπορεί να **διαμορφωθεί** με ευέλικτο τρόπο
  - αυξάνοντας σημαντικά το εύρος χρήσης

Μετα-συνάρτηση

- Συνάρτηση που δέχεται ως παράμετρο ... **κώδικα!**

- Στην C: συνάρτηση που δέχεται ως μια από τις παραμέτρους της έναν **δείκτη σε συνάρτηση**

- Πρέπει να οριστεί ρητά ο **τύπος** της συνάρτησης που θα δοθεί ως παράμετρος στην μετα-συνάρτηση
  (α) ο **τύπος των παραμέτρων** που δέχεται η συνάρτηση που θα περάσουμε ως παράμετρο
  (β) ο **τύπος της τιμής** που επιστρέφει η συνάρτηση που θα περάσουμε ως παράμετρο

Γιατί να έχουμε μετα-συναρτήσεις;

- Μια «βασική» διαδικασία μπορεί να «εξειδικεύεται» για να υποστηρίξει διαφορετική λειτουργικότητα

Συμβατική προσέγγιση:

- Η βασική διαδικασία υλοποιείται **πολλές φορές**
- Κάθε φορά μέσα σε μια ξεχωριστή συνάρτηση για κάθε εξειδικευμένη λειτουργικότητα

Εναλλακτική προσέγγιση:

- Η βασική διαδικασία υλοποιείται **μια φορά**
- Ως **μετα-συνάρτηση**
- Έτσι ώστε να **δέχεται ως παράμετρο** την διαδικασία για κάθε εξειδικευμένη λειτουργία

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

/* comparator function for objects in array */
int comp(const void *a, const void *b) {
    return (*(int *)a - *(int *)b);
}

int main() {
    int a[] = {5, 2, 3, 1, 4};
    int i, n = sizeof(a) / sizeof(a[0]);

    /* sort the array */
    qsort(a, n, sizeof(a[0]), comp);

    for (i = 0; i < nofelems; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");

   return 0;
}
```

συμβατική προσέγγιση

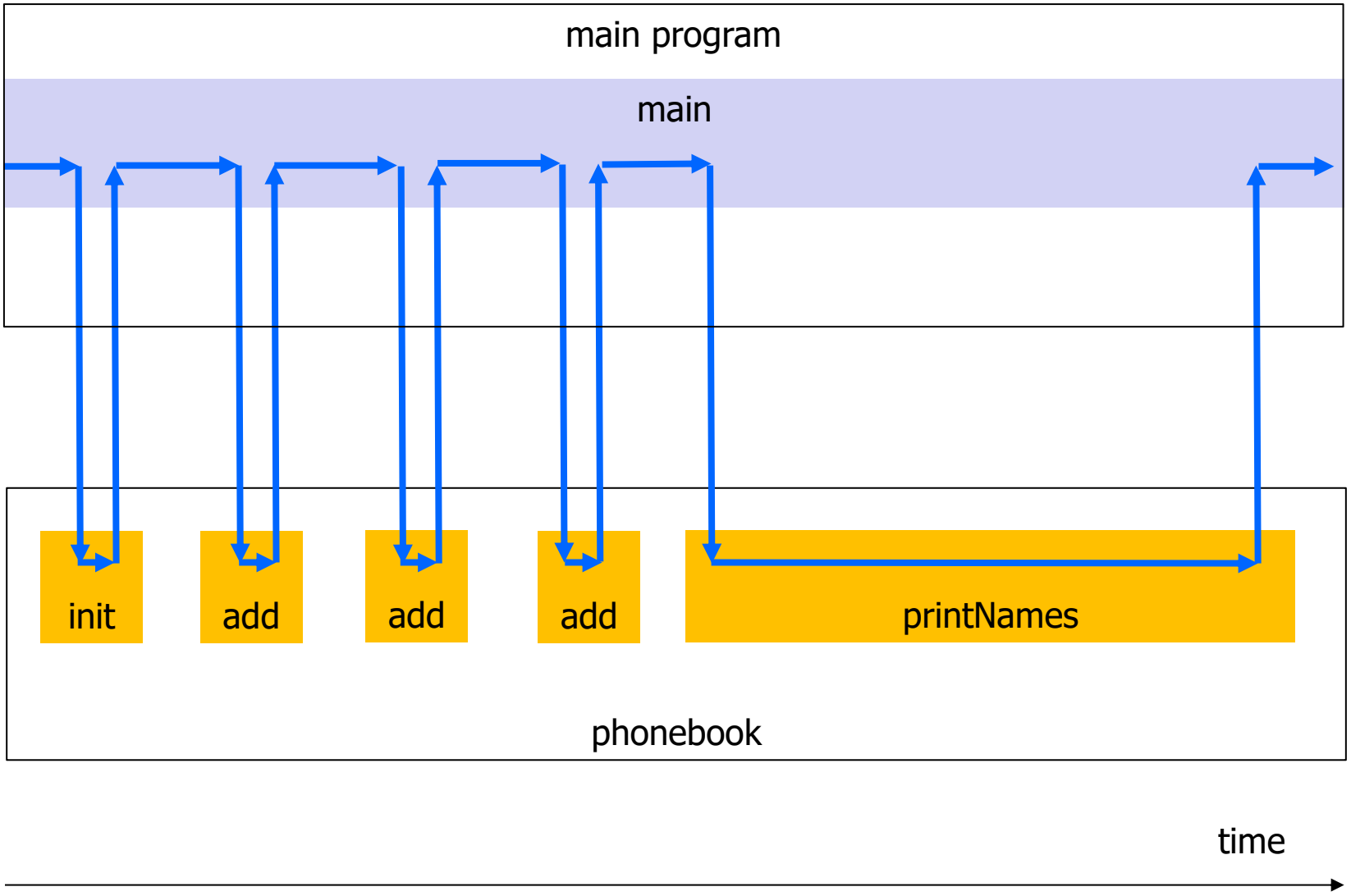
```
typedef struct {
  char name[64];
  char phone[64];
} entry_t;

typedef struct {
  entry_t* entries; /* dynamic table holding the entries */
  int size;         /* current size of the table */
} phonebook_t;

void phonebook_init(phonebook_t *pb);
int phonebook_add(phonebook_t *pb, const entry_t *e);
int phonebook_find(const phonebook_t *pb,
                   const char *name, entry_t *e);
int phonebook_rmv(phonebook_t *pb, const char *name);
void phonebook_clear(phonebook_t *pb);

/* print functions */
void phonebook_printNames(phonebook_t *pb);
void phonebook_printPhoneNumbers(phonebook_t *pb);
```

```
void phonebook_printNames(phonebook_t *pb) {
  int i;

  for(i=0; i < pb->size; i++) {
    printf("Name: %s\n", pb->entries[i].name);
  }
}
```

βασικός κώδικας διάσχισης της δομής

```
void phonebook_printPhoneNumbers(phonebook_t *pb) {
  int i;

  for(i=0; i < pb>size; i++) {
    printf("Name: %s\n", pb->entries[i].phone);
  }
}
```

```
int main(int argc, char *argv[]) {
  entry_t e;
  phonebook_t pb;

  phonebook_init(&pb);
  for (…) {
    …
    phonebook_add(&pb, &e);
  }

  phonebook_printNames(&pb);
  phonebook_printPhoneNumbers(&pb);

  …

}
```


main program
main
init
add
add
add
printNames
phonebook
time

προσέγγιση με μετα-συνάρτηση διάσχισης

```
typedef struct {
  char name[64];
  char phone[64];
} entry_t;

typedef struct {
  entry_t* entries; /* dynamic table holding the entries */
  int size;         /* current size of the table */
} phonebook_t;

void phonebook_init(phonebook_t *pb);
int phonebook_add(phonebook_t *pb, const entry_t *e);
int phonebook_find(const phonebook_t *pb,
                   const char *name, entry_t *e);
int phonebook_rmv(phonebook_t *pb, const char *name);
void phonebook_clear(phonebook_t *pb);


/* meta-function for generic traversal */
void phonebook_traverse(phonebook_t *pb,
                        void (*handle)(entry_t *));
```

δείκτης σε συνάρτηση που (α) δέχεται ως παράμετρο ένα δείκτη σε `entry_t` και (β) επιστρέφει `void`

```
void phonebook_traverse(phonebook_t *pb,
                        void (*handle)(entry_t *)) {
  int i;

  for(i=0; i < pb->size; i++) {
    handle(&pb->entries[i]);
  }
}
```

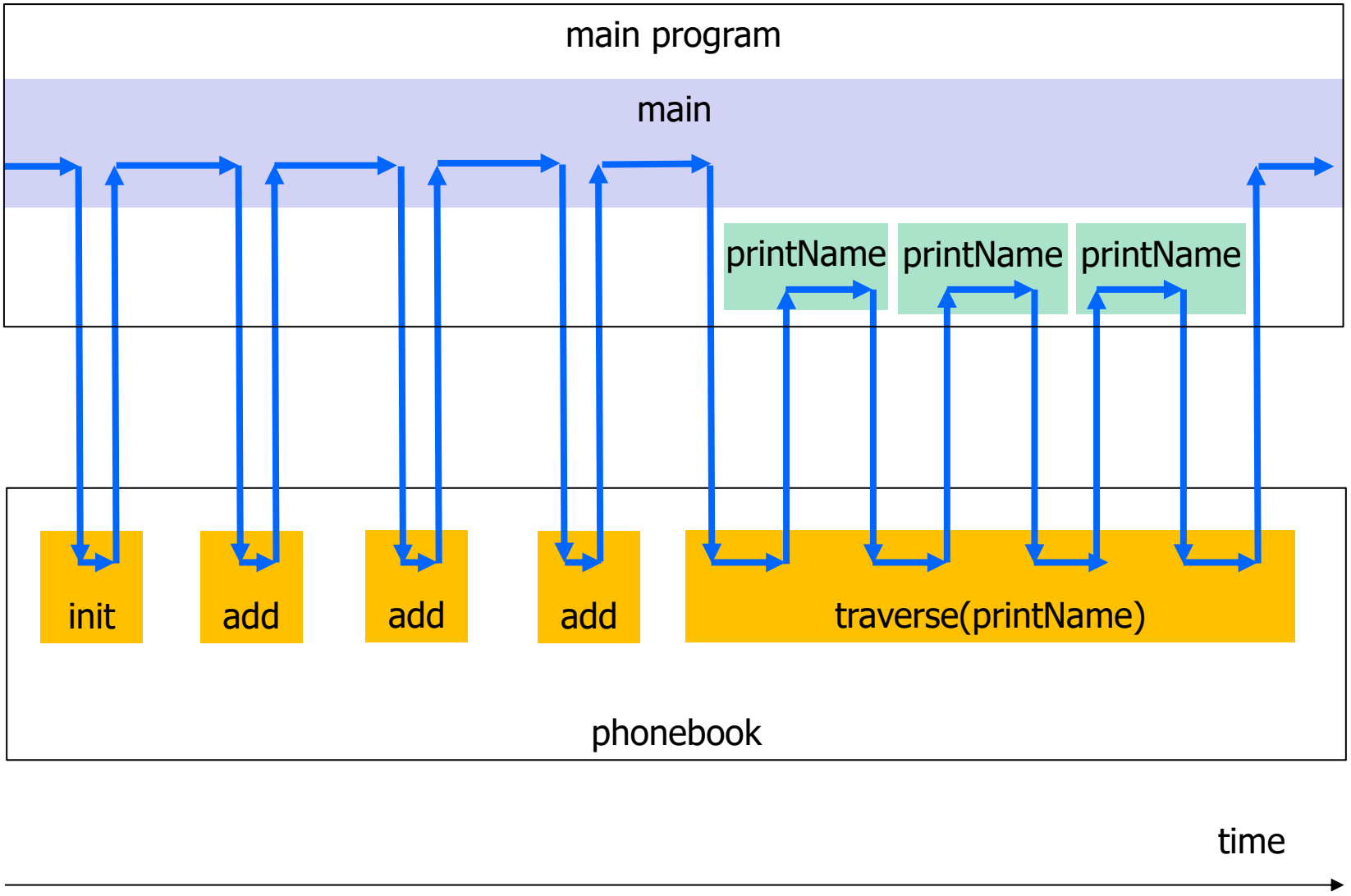
```
void printNames(entry_t *e) {
  printf("Name: %s\n", e->name);
}
void printPhoneNumber(entry_t *e) {
  printf("Phone: %s\n", e->phone);
}
void printFull(entry_t *e) {
  printf("Name: %s, Phone: %s\n", e->name, e->phone);
}

int main(int argc, char *argv[]) {
  entry_t e;
  phonebook_t pb;

  phonebook_init(&pb);
  for (…) {
    …
    phonebook_add(&pb, &e);
  }

  phonebook_traverse(&pb, printNames);
  phonebook_traverse(&pb, printPhoneNumber);
  phonebook_traverse(&pb, printFull);
  …

}
```

Ροή εκτέλεσης

- Τι γίνεται με την ροή εκτέλεσης του προγράμματος όταν χρησιμοποιούνται συναρτήσεις ως παράμετροι άλλων συναρτήσεων;

- Στην ουσία: μια συμβατική κλήση συνάρτησης Β μέσα από άλλη συνάρτηση Α

- Η διαφορά είναι ότι στον κώδικα της συνάρτησης Α **δεν** προσδιορίζεται ποια συνάρτηση Β θα κληθεί κατά την διάρκεια της εκτέλεσης
  - σε αντίθεση με τις «καρφωτές» κλήσεις συνάρτησης


main program
main
printName
printName
printName
init
add
add
add
traverse(printName)
phonebook
time

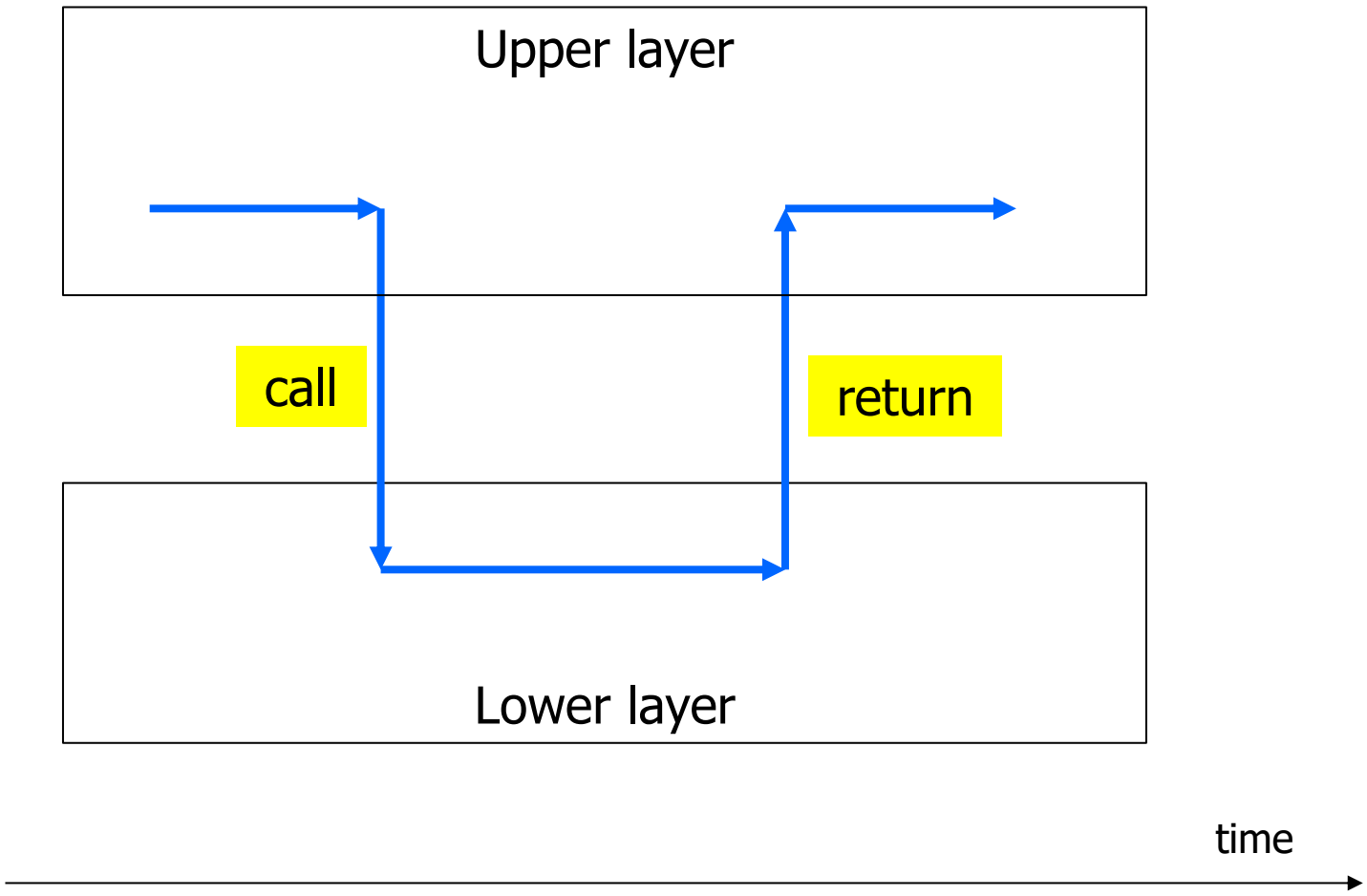
Αντίστροφες κλήσεις – up calls

- Έστω ότι έχουμε δύο επίπεδα λογισμικού
  - «πάνω» επίπεδο: κώδικας που γράφουμε τώρα, π.χ. εφαρμογή
  - «κάτω» επίπεδο: κώδικας που ήδη υπάρχει, π.χ. βιβλιοθήκη

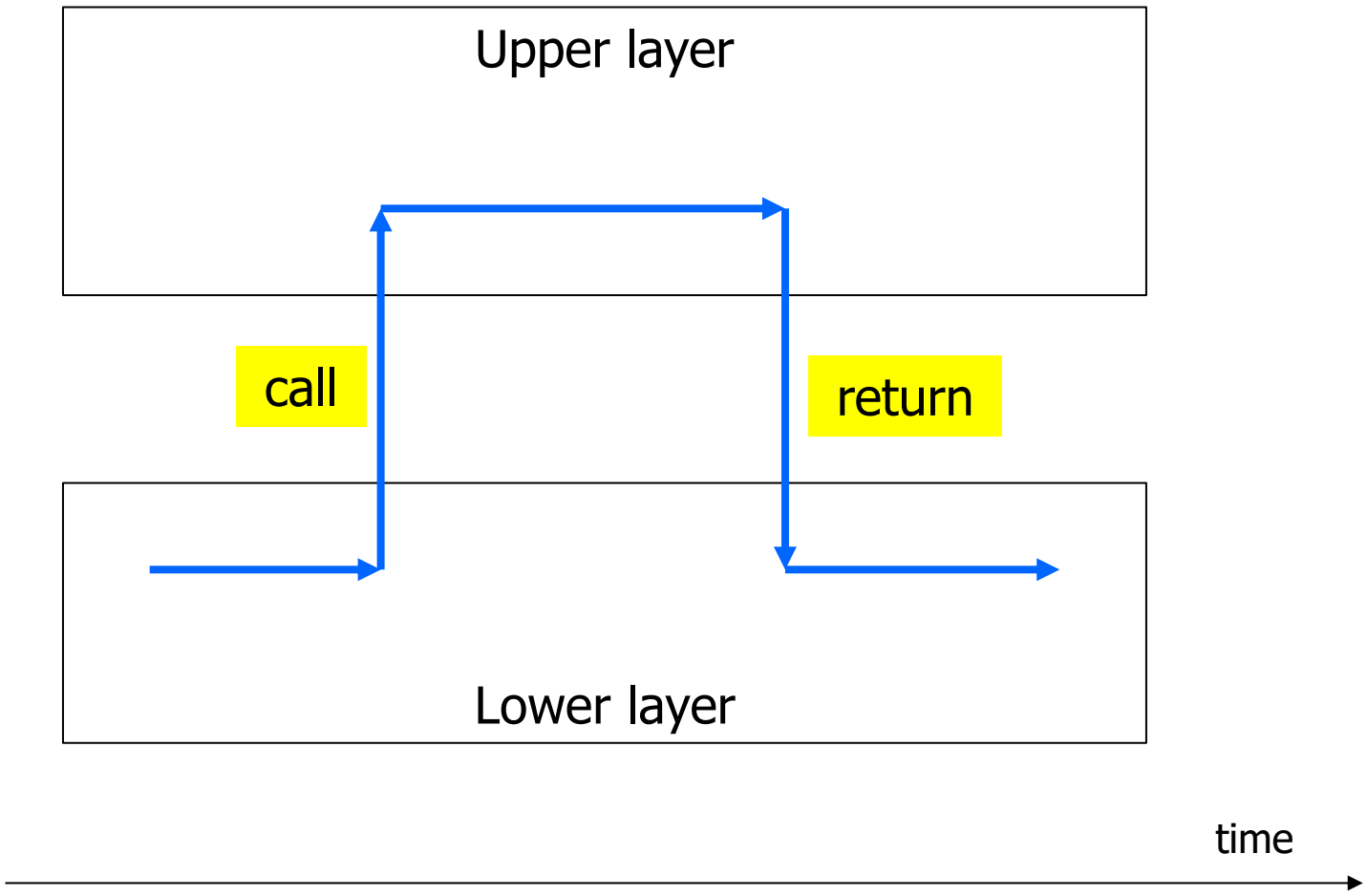
- Συμβατική/κλασική ροή εκτέλεσης: **down calls**
  - η κλήση γίνεται από πάνω προς το κάτω

- Με δείκτες σε συναρτήσεις, η συμβατική ροή εκτέλεσης αντιστρέφεται: **up-calls / call-backs**
  - η κλήση γίνεται από κάτω προς τα πάνω

Down call



Up call



Προγραμματισμός με γεγονότα

- Το πάνω επίπεδο λογισμικού δίνει/περνάει μια συνάρτηση στο κάτω επίπεδο λογισμικού
- Γίνεται **εγκατάσταση** ενός **χειριστή**

- Κάθε φορά μου λαμβάνει χώρα ένα γεγονός στο κάτω επίπεδο, γίνεται (αντίστροφη) κλήση του χειριστή
- Για να υπάρξει ενημέρωση και να μπορεί να γίνει χειρισμός του γεγονότος (και) στο πάνω επίπεδο

Event handling