Δείκτες σε συναρτήσεις

Συνάρτηση

- Ομάδα εντολών που γράφουμε ξεχωριστά για να υλοποιήσουμε μια συγκεκριμένη λειτουργία
  - για καλύτερη / πιο καθαρή δόμηση του κώδικα
  - για να μπορούμε να τις εκτελούμε πολλές φορές χωρίς να πρέπει να επαναλαμβάνουμε κάθε φορά τις ίδες εντολές

- Μπορεί να δέχεται παραμέτρους
  - παραμετροποιημένος κώδικας
  - δεν χρειάζεται να γράφουμε (πρακτικά) τον ίδιο κώδικα πολλές φορές μέσα στα προγράμματα μας
  - επαναχρησιμοποίηση μιας γενικής/βασικής διαδικασίας έχοντας παράλληλα την ευελιξία για μικρο-προσαρμογές μέσω των παραμέτρων

Μετα-συνάρτηση

- Μια από τις παραμέτρους που δέχεται η συνάρτηση είναι ... **κώδικας**

- Ένας τρόπος (στην C ο μόνος) να περάσουμε κώδικα ως παράμετρο μιας συνάρτησης είναι να οριστεί κατάλληλη παράμετρος με **τύπο συνάρτησης**

- Αυτός ορίζεται μέσω των παραμέτρων που δέχεται και του αποτελέσματος που επιστρέφει η συνάρτηση

Γιατί να έχουμε μετα-συναρτήσεις;

- Μια συνάρτηση μπορεί να έχει διαφορετικές, **ανεξάρτητες** διαδικασίες / «διαστάσεις»
  - π.χ., διάσχιση και εκτύπωση των περιεχομένων μιας πολύπλοκης δομής δεδομένων
- Μπορεί να είναι χρήσιμο αυτές οι διαδικασίες να υλοποιηθούν **ξεχωριστά**
  - π.χ., διαδικασία διάσχισης / διαδικασία εκτύπωσης
- Πως μπορούμε να το πετύχουμε αυτό;
- Υλοποιούμε την βασική διαδικασία έτσι ώστε να δέχεται τις υπόλοιπες διαδικασίες ως παραμέτρους
  - π.χ., η συνάρτηση που υλοποιεί την διάσχιση της δομής **δέχεται ως παράμετρο την συνάρτηση** που υλοποιεί την εκτύπωση των περιεχομένων κάθε στοιχείου της δομής

```
struct list {
  int v;
  struct list *nxt;
};

struct list *list_init();
int list_hasElement(struct list *root, int v);
struct list *list_insert(struct list *root, int v);
struct list *list_remove(struct list *root, int v);
void list_destroy(struct list *root);

void list_print(struct list *root);
```

structure-level traversal code

```
void list_print(struct list *root) {
  struct list *curr;

  for(curr=root; curr != NULL; curr=curr->nxt) {
    printf("%d\n",curr->v);
  }
}
```

node-level print code

```c
struct list {
  int v;
  struct list *nxt;
};

struct list *list_init();
int list_hasElement(struct list *root, int v);
struct list *list_insert(struct list *root, int v);
struct list *list_remove(struct list *root, int v);
void list_destroy(struct list *root);

void list_traverse(struct list *root,
                   void (*handle)(struct list *));
```

```
void list_traverse(struct list *root,
                   void (*handle)(struct list *)) {
  struct list *curr;

  for(curr=root; curr != NULL; curr=curr->nxt) {
    handle(cur);
  }
}
```

```
void printFun(struct list *l) {
  printf("%d\n",l->v);
}
```

```
struct list *root = list_init();
struct list *root = list_insert(root, 1);
struct list *root = list_insert(root, 2);
struct list *root = list_insert(root, 3);
list_traverse(root, printFun);
```

Υλοποίηση δομών με πλήρη ανεξαρτησία ως προς το περιεχόμενο των στοιχείων

- Οι βασικές λειτουργίες των πολύπλοκων δομών δεδομένων μπορεί να υλοποιηθούν **μια φορά**, ανεξάρτητα από το περιεχόμενο των στοιχείων
- Η υλοποίηση αφορά την οργάνωση της αποθήκευσης των στοιχείων στην μνήμη (όχι το περιεχόμενο τους)
  - π.χ., πίνακας, λίστα, δέντρο κλπ
- Κάποιες λειτουργίες χρειάζονται πρόβαση στα περιεχόμενα των στοιχείων
  - αναζήτηση: απαιτείται έλεγχος ισότητας / ισοδυναμίας ανάμεσα σε δύο στοιχεία
  - ταξινόμηση: απαιτείται σύγκριση ανάμεσα σε δύο στοιχεία
- Κατάλληλη αφαίρεση μέσω συναρτήσεων που δίνονται ως παράμετροι από την εφαρμογή

element comparison function
(-1 smaller, 0 equal, 1 bigger)

```
typedef struct setstruct *set_t;

set_t set_init(int (*cmp)(void *, void*));
int set_find(set_t s, void *elem);
set_t set_add(set_t s, void *elem);
set_t set_rmv(set_t s, void *elem);
void set_traverse(set_t s, void (*handle)(void *));
void set_destroy(set_t s);
```

element handling function
(whatever the application wants
to do with every element)

```
struct student {
  char name[64];
  unsigned short courses_taken;
  unsigned short courses_passed;
  unsigned short courses_left;
  float gpa;
};

int cmpStudents(void *sptr1, void *sptr2) {
  struct student *s1 = (struct student *)sptr1;
  struct student *s2 = (struct student *)sptr2;
  return(strcmp(s1->name,s2->name));
}

set_t students = set_init(cmpStudents);
…
```

```
void printStudent(void *sptr) {
  struct student *s = (struct student *)sptr;
  printf("%s\t", s->name);
  printf("%d\t", s->courses_left);
  printf("%f\n", s->gpa);
}

printf("Name\tCourses Left\tGPA\n");
set_traverse(students, printStudent);
```

```
int counter; // global

void countStudent(void *sptr) {
  counter++;
}

counter = 0;
set_traverse(students, countStudent);
printf("nof students is %d\n",counter);
```

Μια πιο καθαρή λύση για την διάσχιση

- Η διάσχιση της δομής χωρίς έλεγχο από την εφαρμογή δεν είναι ιδιαίτερα ευέλικτη

- Μπορεί να δοθεί περισσότερος έλεγχος στην εφαρμομή, μέσα από μια διαφορετική προσέγγιση

- Ξεχωριστές λειτουργίες για την έναρξη/αρχικοποίηση της διαδικασίας διάσχισης της δομής, και για την επιστροφή του επόμενου στοιχείου

```
void set_traverse_init(set_t s);
void *set_traverse_nxt(set_t s);

void print_students() {
  struct student *s

  printf("Name\tCourses Left\tGPA\n");

  set_traverse_init(students);
  while (1) {
    s = (struct student *)set_traverse_nxt(students);
    if (s == NULL) break;
    printf("%s\t", s->name);
    printf("%d\t", s->courses_left);
    printf("%f\n", s->gpa);
  }
}
```

```
void set_traverse_init(set_t s);
void *set_traverse_nxt(set_t s);

void count_students() {
  int counter;
  struct student *s

  counter = 0;
  set_traverse_init(students);
  while (1) {
    s = (struct student *)set_traverse_nxt(students);
    if (s == NULL) break;
    counter++;
  }
  printf("nof students is %d\n",counter);
}
```

Ροή εκτέλεσης

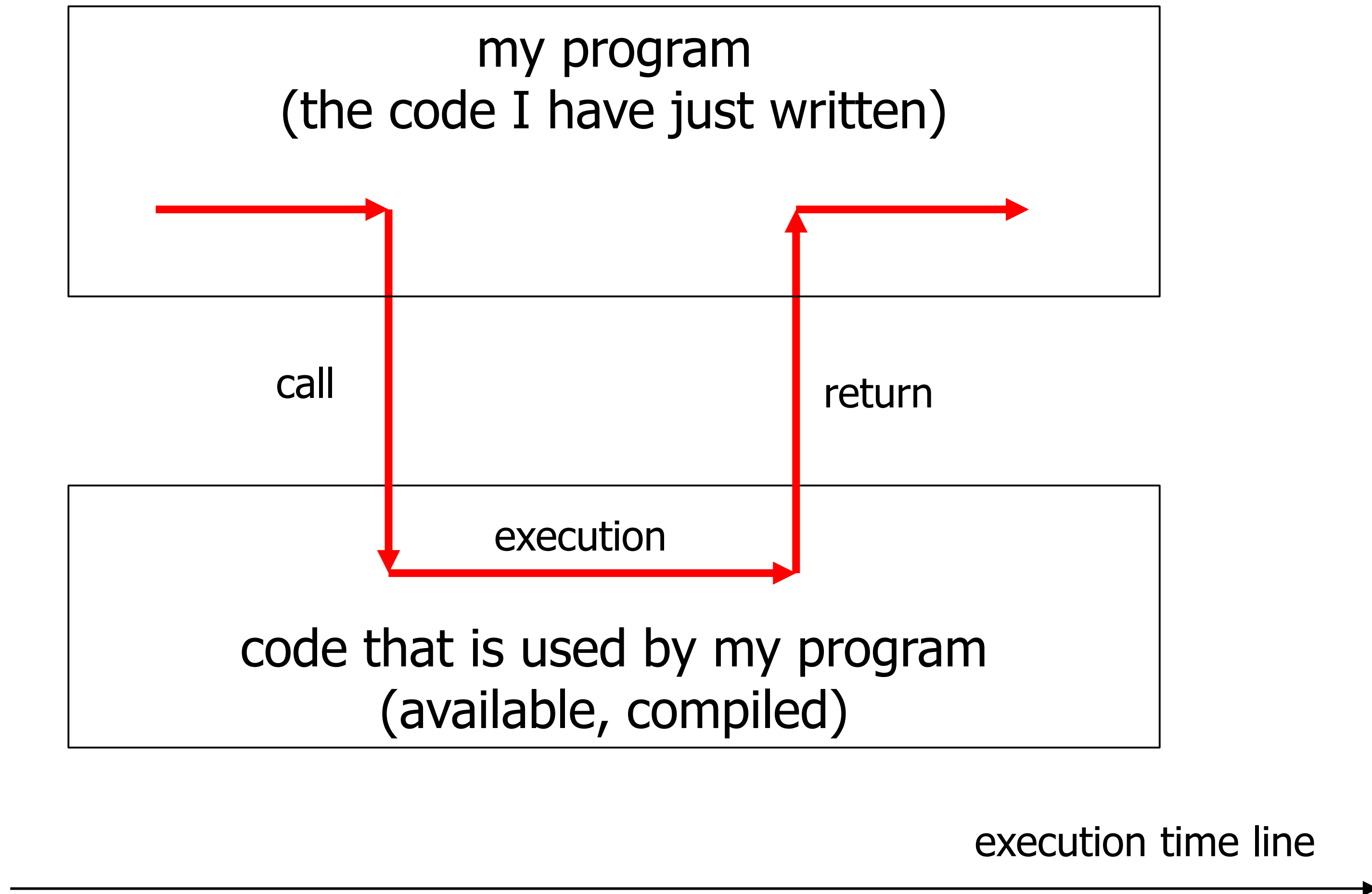
- Τι γίνεται με την ροή εκτέλεσης του προγράμματος όταν χρησιμοποιούνται συναρτήσεις ως παράμετροι άλλων συναρτήσεων;
- Μια κλήση συνάρτησης μέσα από άλλη συνάρτηση
  - όπως ακριβώς με τις «καρφωτές» κλήσεις συναρτήσεων

- Όμως, η καλούσα συνάρτηση / καλών κώδικας **δεν γνωρίζει** ποιά συνάρτηση / ποιός κώδικας θα κληθεί κατά την διάρκεια της εκτέλεσης
- Ένας τρόπος ευέλικτης **επέκτασης** λογισμικού: κώδικας που γράφουμε «τώρα» μπορεί να καλέσει κώδικα που **θα** γραφτεί κάποια στιγμή «στο μέλλον» (πιθανώς από άλλους)

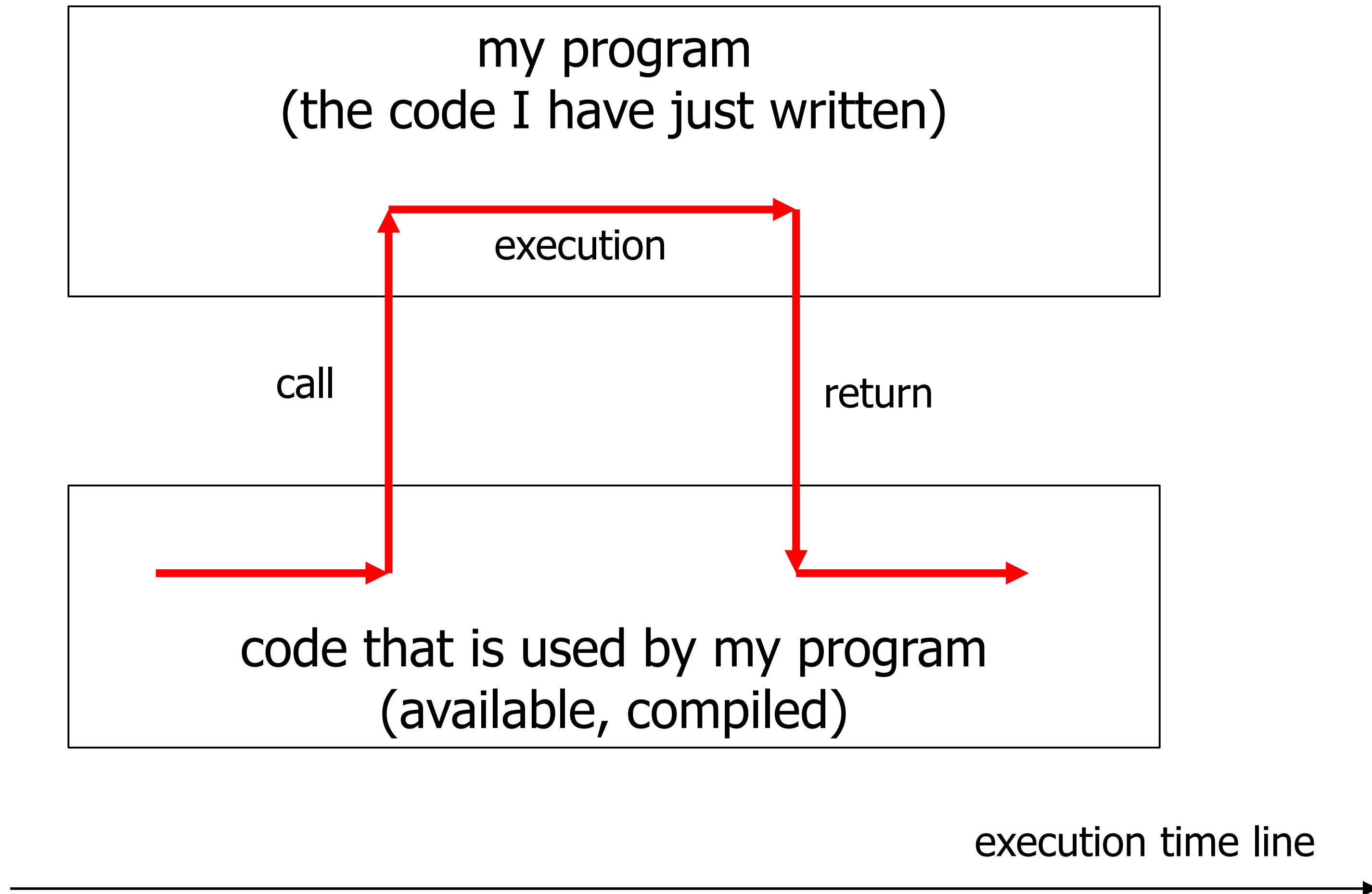
Αντίστροφες κλήσεις – up calls

- Η συνηθισμένη/κλασική ροή εκτέλεσης είναι το λογισμικό που βρίσκεται σε ψηλότερο «επίπεδο» (ο κώδικας γράφουμε τώρα) να καλεί το λογισμικό που βρίσκεται σε «χαμηλότερο» επίπεδο (κώδικας που έχει ήδη γραφτεί – και μεταφραστεί – την ώρα που γράφουμε τον παραπάνω κώδικα)
  - down calls
- Με δείκτες σε συναρτήσεις, αυτή η ροή εκτέλεσης μπορεί να **αντιστραφεί!**
- Το λογισμικό που βρίσκεται σε πιο χαμηλό επίπεδο μπορεί να καλέσει λογισμικό που βρίσκεται σε πιο ψηλό επίπεδο
  - up calls

Down call



Up call



Πολυμορφικές δομές δεδομένων

Δομές με ετερογενή στοιχεία

- Μια δομή μπορεί να χρησιμοποιηθεί για την διαχείριση στοιχείων **διαφορετικού** τύπου
- Πως μπορεί να υλοποιηθούν οι βασικές λειτουργίες της δομής **ανεξάρτητα** από τον τύπο των στοιχείων;
- Ορίζεται ένας **βασικός τύπος** με την απαραίτητη πληροφορία για την διαχείρηση της δομής, με βάση τον οποίο υλοποιούνται όλες οι λειτουργίες
- Ο τύπος των στοιχείων που προστίθενται στην δομή ορίζονται ως **επεκτάσεις** του βασικού τύπου
- Πως γνωρίζουμε τον τύπο του στοιχείου που επιστρέφεται π.χ. σε μια αναζήτηση;
- Πρέπει να προβλεφθεί επιπλέον πληροφορία τύπου (type information) που καθοδηγεί το type casting

```
struct listbase {
  int key;  /* for search/sort functionality */
  int type; /* type code for data extensions */
  struct listbase *nxt; /* implement a linked list */
};

struct listbase *root;

void list_init();
void list_add(int v, int type, struct listbase *l);
struct listbase * list_find(int v);
void list_remove(int v);
void list_destroy();
```

```
#define TYPE1 1
#define TYPE2 2

struct ext1 {
  struct listbase base;
  int data; /* data on top of extension */
};

struct ext2 {
  struct listbase base;
  char data[16]; /* data on top of extension */
};
```

```
…

for (i=0; i<N; i++) {
  if (i%2 == 0) {
    e1 = (struct ext1 *) malloc(sizeof(struct ext1));
    e1->data = i;
    list_add(i,TYPE1,(struct listbase *)e1);
  }
  else {
    e2 = (struct ext2 *) malloc(sizeof(struct ext2));
    sprintf(e2->data,"%d",i);
    list_add(i,TYPE2,(struct listbase *)e2);
  }
}
```

```
…

struct listbase *l;

for (i=N-1; i>=0; i--) {
  l = list_find(id);
  if (l != NULL) {
    switch (l->type) {
      case TYPE1: {
        printf("%d\n",((struct ext1 *)l)->data);
        break;
      }
      case TYPE2: {
        printf("%s\n",((struct ext2 *)l)->data);
        break;
      }
    }
  }
}
```

Επεκτασιμότητα στο λογισμικό

- Η χρήση αντίστροφων κλήσεων σε συνδυασμό με τις πολυμορφικές δομές είναι από τους πιο βασικούς μηχανισμούς **επεκτασιμότητας**
- Κώδικας που γράφεται «τώρα» μπορεί να καλέσει κώδικα που θα γραφτεί «αργότερα»
- Κώδικας που γράφεται «τώρα» μπορεί να διαχειριστεί ετερογενή στοιχεία ο πραγματικός τύπος των οποίων θα προσδιοριστεί «αργότερα»
- Παρόμοια λογική ανάπτυξης υποστηρίζεται, με πιο «καθαρό» τρόπο, από τις αντικειμενοστραφείς (object-oriented) γλώσσες προγραμματισμού

lalis@inf.uth.gr