Προγραμματισμός Ι (ECE115)

#13 δομές – ενώσεις – απαριθμήσεις (structs – unions – enums)

Σπύρος Λάλης
lalis@uth.gr

Σύνθετοι τύποι δεδομένων

- Συχνά τα δεδομένα ενός προγράμματος δεν είναι απλά κάποιοι μεμονωμένοι αριθμοί ή χαρακτήρες
- Ούτε απλοί πίνακες από βασικούς τύπους δεδομένων

- Μπορούμε να **ομαδοποιήσουμε** περισσότερα δεδομένα διαφορετικού τύπου ως **μια οντότητα**

- Ο προγραμματιστής μπορεί να ορίσει **δικούς** του, απεριόριστα **πολύπλοκους** τύπους δεδομένων
  - με βάση πιο απλούς/βασικούς τύπους δεδομένων

Δομές
(structs)

Δομή `struct`

- Η **δομή** ομαδοποιεί πολλά ξεχωριστά δεδομένα (πιθανώς διαφορετικού τύπου) σε μια οντότητα
- Τα επιμέρους στοιχεία της δομής ονομάζονται **πεδία**
- Το μέγεθος της δομής καθορίζεται **αυτόματα** (από τον μεταφραστή) με βάση τον χώρο που χρειάζεται για την αποθήκευση τιμών των πεδίων
- Η μνήμη μιας δομής δεσμεύεται **μονοκόμματα**
- Τα πεδία αποθηκεύονται στη μνήμη με τη σειρά που δηλώθηκαν στη δομή
- Αλλά **δεν** καταλαμβάνουν απαραίτητα συνεχόμενες θέσεις στη μνήμη – το αποφασίζει ο **μεταφραστής!**
  - είναι προγραμματιστικό **λάθος** να γίνονται υποθέσεις

```
struct xxx {
   /* δήλωση πεδίων */
};
…
struct xxx var1, var2;
```

όνομα τύπου δομής

ονόματα μεταβλητών

```
struct xxx {
   /* δήλωση πεδίων */
} var1;
…
struct xxx var2;
```

όνομα τύπου δομής

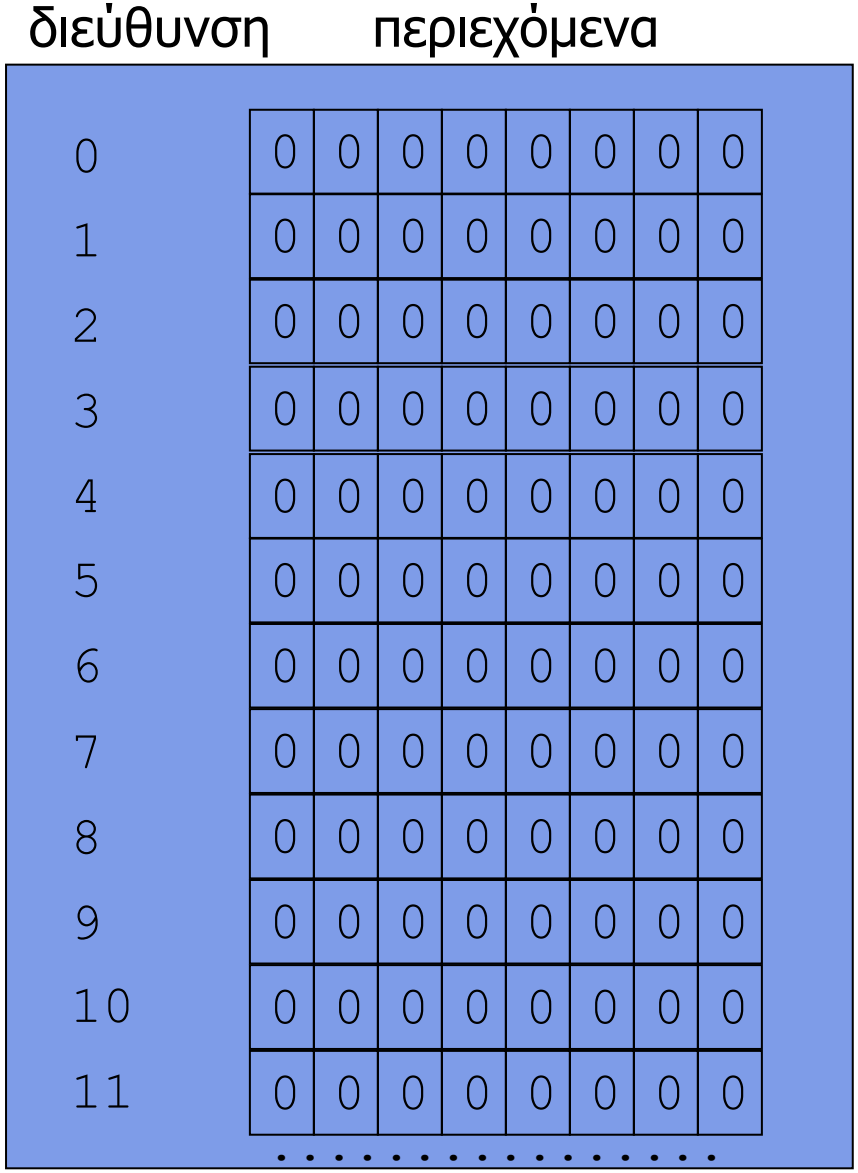
ονόματα μεταβλητών

```
struct {
   /* δήλωση πεδίων */
} var1, var2;
```

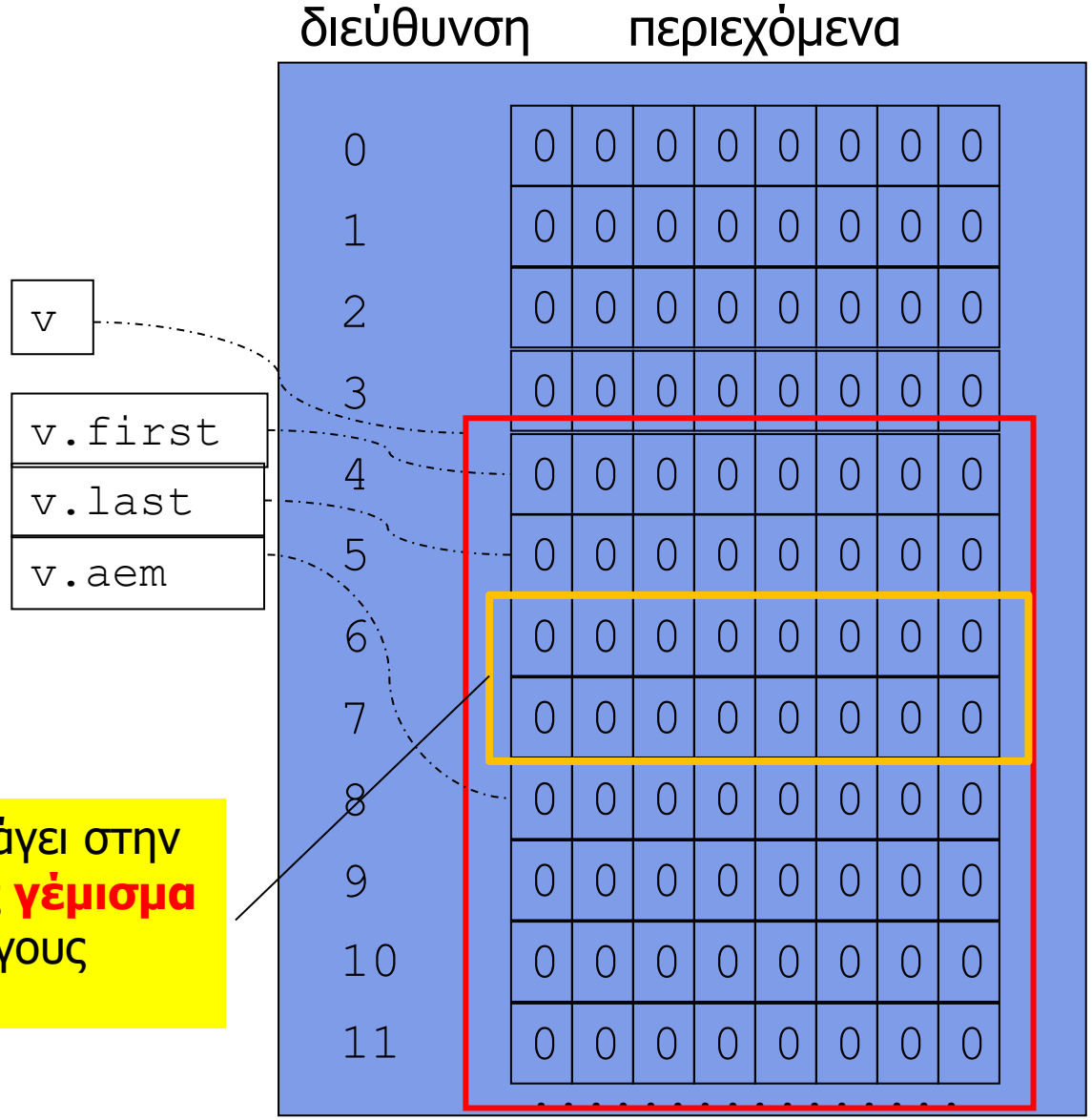
ανώνυμος τύπος δομής

ονόματα μεταβλητών

```
struct student {
  char first;
  char last;
  int aem;
};
```

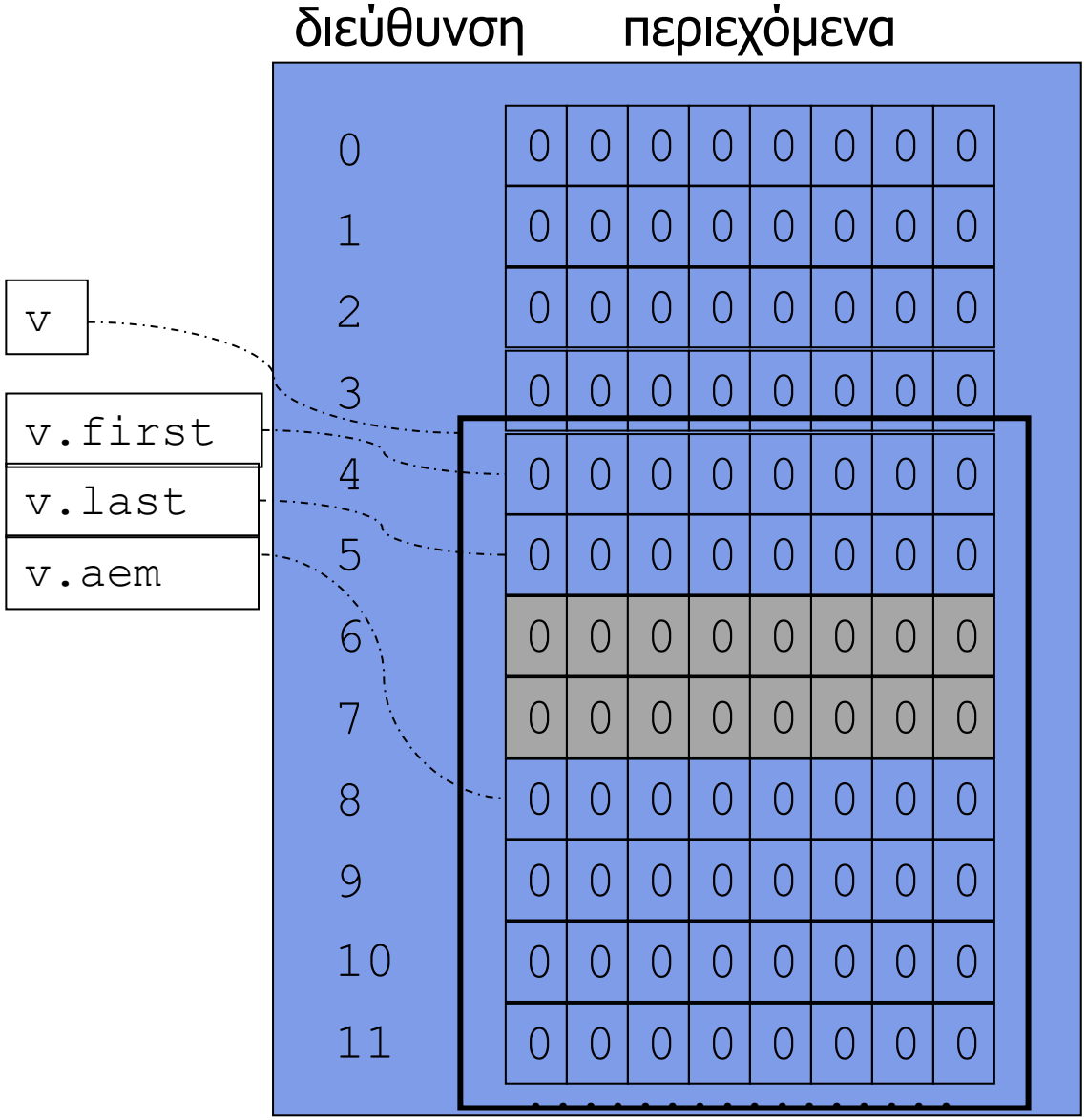


```
struct student {
   char first;
   char last;
   int aem;
};

struct student v;
```

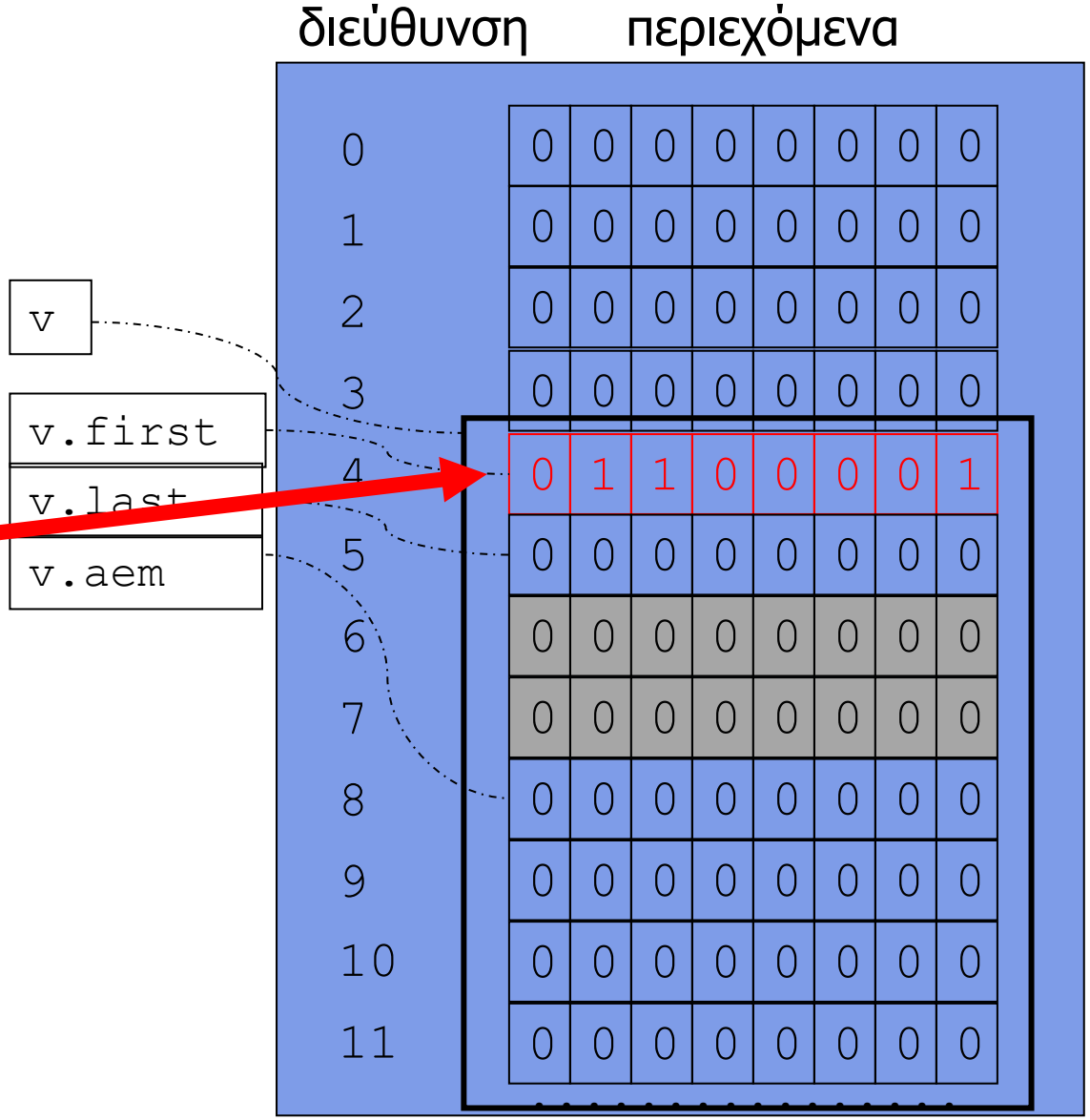


bytes που μπορεί να εισάγει στην δομή ο μεταφραστής ως **γέμισμα** (filler bytes), π.χ. για λόγους απόδοσης

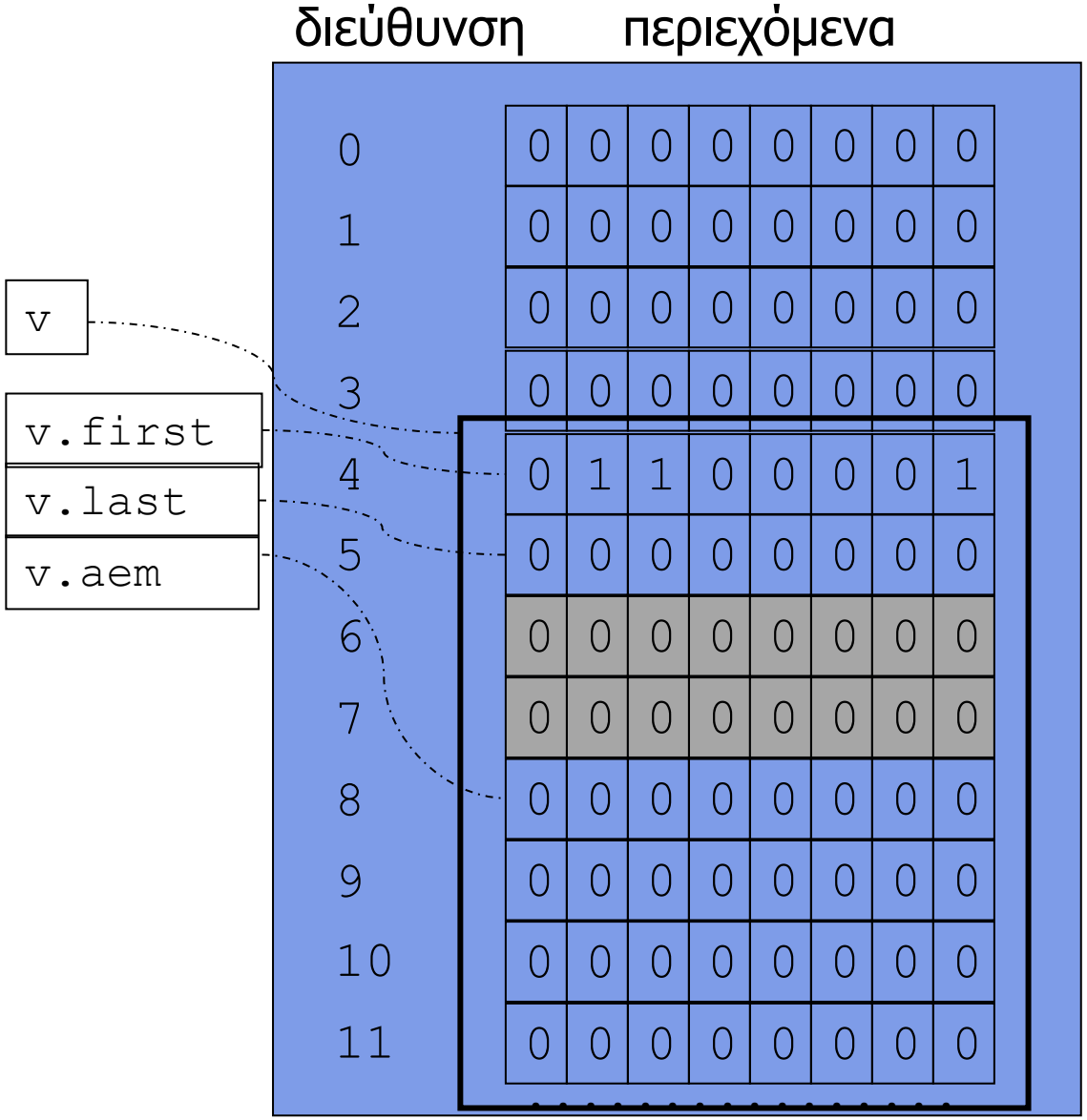
```
struct student {
  char first;
  char last;
  int aem;
};

struct student v;
```



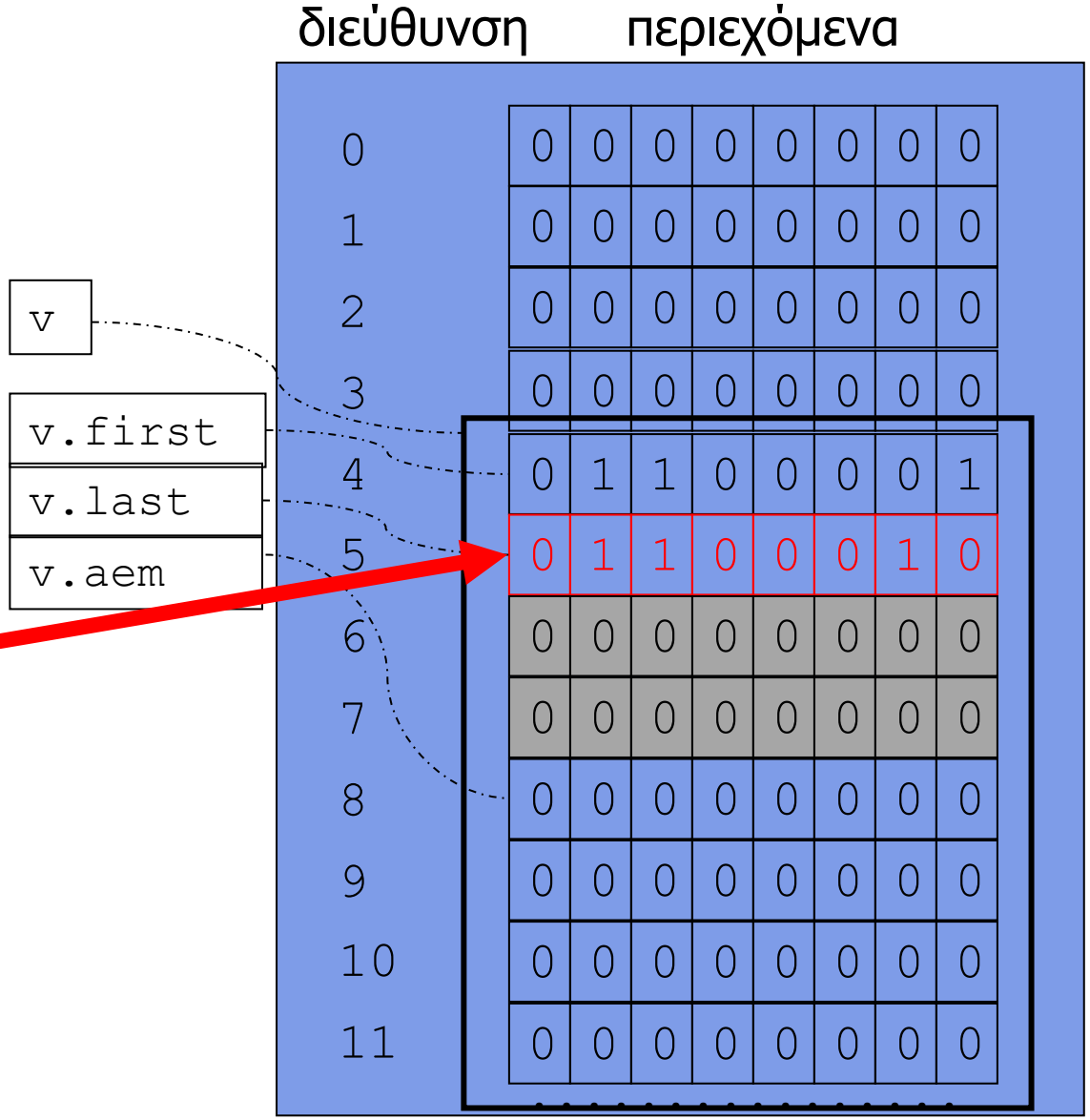
```
struct student {
   char first;
   char last;
   int aem;
};

struct student v;

v.first = 'a';
```



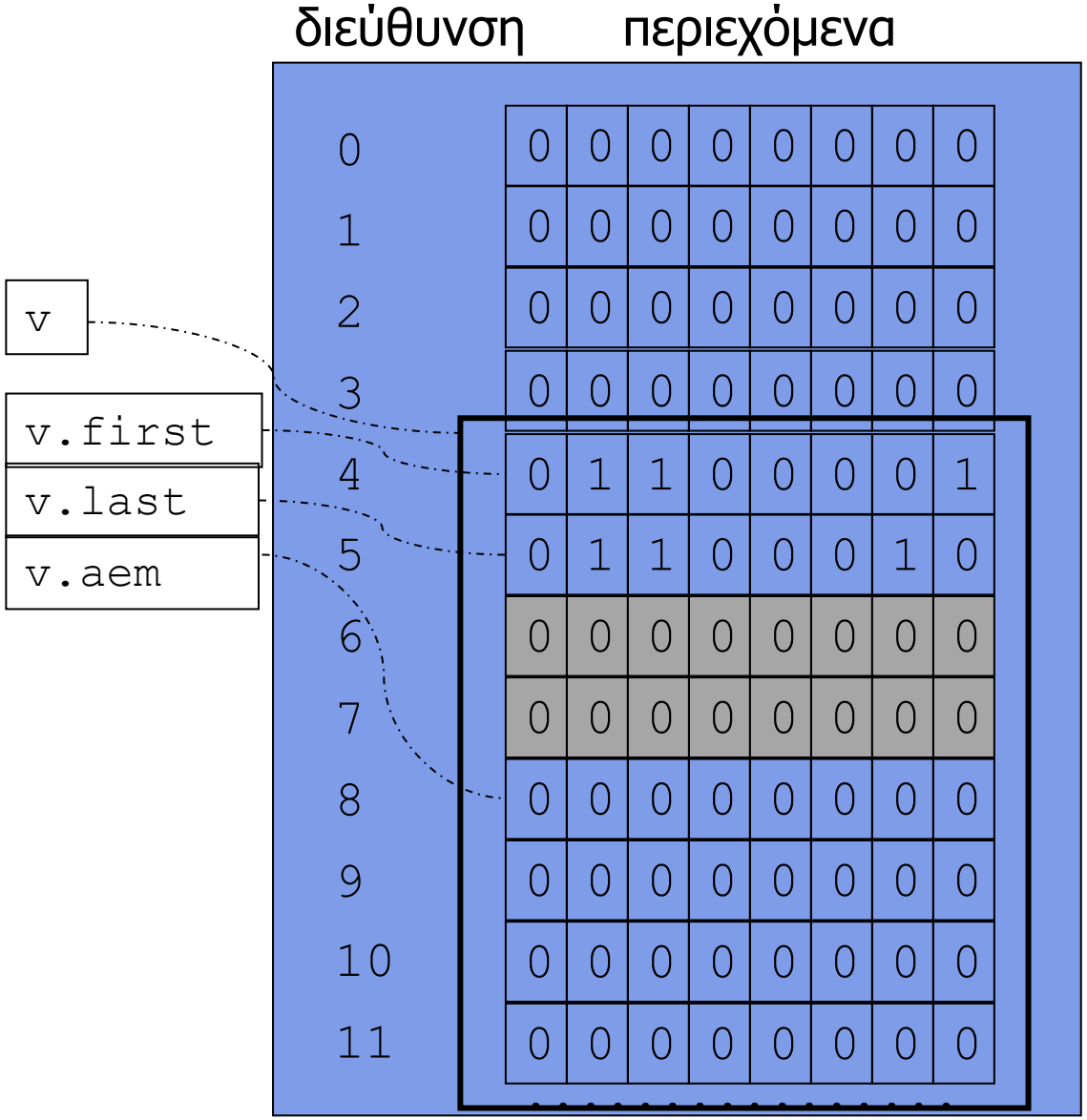
```
struct student {
  char first;
  char last;
  int aem;
};

struct student v;

v.first = 'a';
```



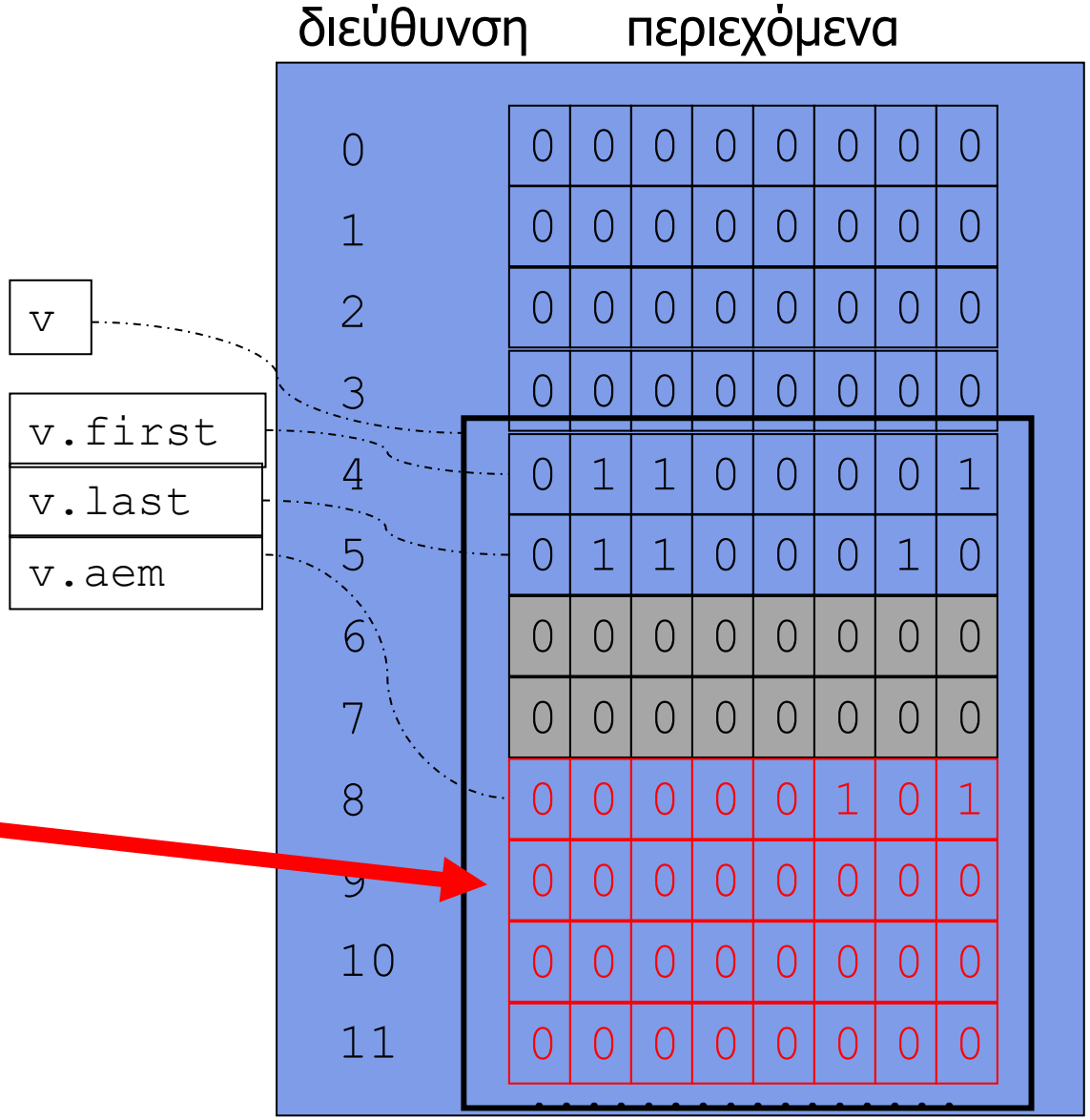
```
struct student {
   char first;
   char last;
   int aem;
};

struct student v;

v.first = 'a';

v.last = 'b';
```



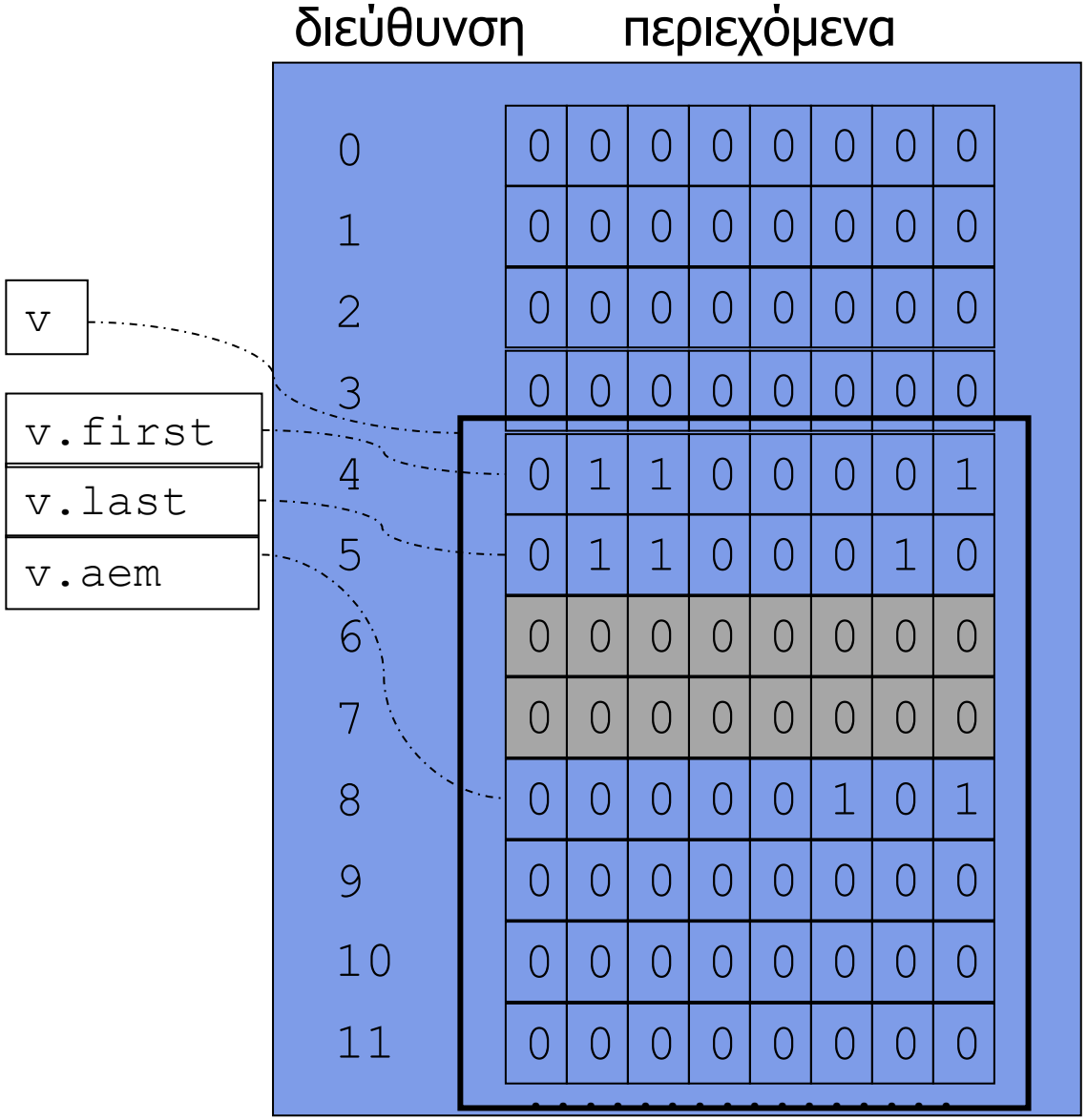
```
struct student {
  char first;
  char last;
  int aem;
};

struct student v;

v.first = 'a';

v.last = 'b';
```



```
struct student {
  char first;
  char last;
  int aem;
};

struct student v;

v.first = 'a';

v.last = 'b';

v.aem = 5;
```



```
struct student {
  char first;
  char last;
  int aem;
};

struct student v;

v.first = 'a';

v.last = 'b';

v.aem = 5;
```



Προσπέλαση δομών

- Αναφορά στα πεδία μιας μεταβλητής `struct` γίνεται μέσω του τελεστή '**.**'
  - τοποθετείται **ανάμεσα** στο όνομα της μεταβλητής που αντιστοιχεί στη δομή και το όνομα του πεδίου
- Η ανάθεση τιμών σε μεταβλητές `struct` μπορεί να γίνει επιλεκτικά, πεδίο-προς-πεδίο
- Μπορεί να γίνει και σε **συνολικό** επίπεδο δομής
  - όπως για κανονικές μεταβλητές βασικών τύπων
- Αν δύο μεταβλητές έχουν τον ίδιο τύπο `struct T` τότε μπορεί να γίνει απ' ευθείας **ανάθεση** της τιμής της μιας μεταβλητής στην άλλη
  - γίνεται αντιγραφή ολόκληρης της περιοχής της μνήμης (δεν γίνεται αντιγραφή πεδίο προς πεδίο)

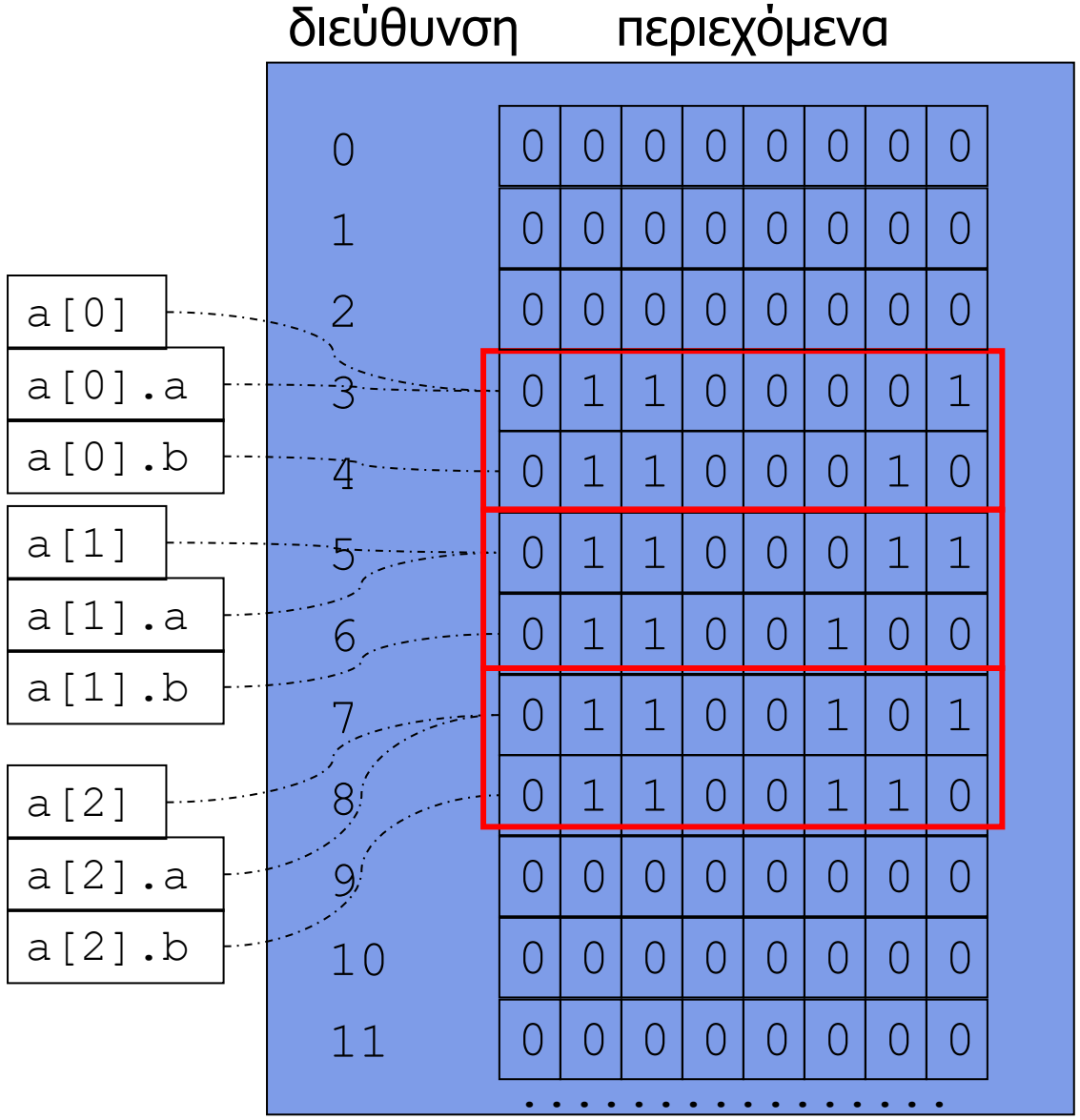
```c
struct date {
  int day;   /* αριθμός ημέρας: 1..31 */
  int month; /* αριθμός μήνα: 1..12 */
  int year;  /* αριθμός έτους */
};

int main(int argc, char *argv[]) {
  struct date d1, d2; /* μεταβλητές struct date */

  d1.day = 25;
  d1.month = 12;
  d1.year = 2019;
  printf("d1: %d/%d/%d\n", d1.day, d1.month, d1.year);

  d2 = d1;
  printf("d2: %d/%d/%d\n", d2.day, d2.month, d2.year);

  d2.year++;
  printf("d2: %d/%d/%d\n", d2.day, d2.month, d2.year);

  return(0);
}
```

Πίνακες από δομές

- Μπορεί να οριστούν **πίνακες** από `struct`
  - όπως και για τους βασικούς τύπους δεδομένων
- Δεσμεύεται ένας **συνεχόμενος** χώρος στην μνήμη για την αποθήκευση **όλων** των στοιχείων του πίνακα

- Η πρόσβαση στα στοιχεία του πίνακα γίνεται με τον ίδιο / συμβατικό τρόπο που ήδη γνωρίζουμε
  - προσδιορισμός θέσης του στοιχείου στον πίνακα
  - μέσα στα επιτρεπτά όρια
- Από τη στιγμή που προσδιοριστεί ένα συγκεκριμένο στοιχείο του πίνακα, πρόσβαση στα επιμέρους πεδία του `struct` γίνεται κανονικά, μέσω του τελεστή '**.**'

```
struct s {
  char a, b;
};
…
struct s a[3];

a[0].a = 'a';
a[0].b = 'b';

a[1].a  ='c';
a[1].b = 'd';

a[2].a = 'e';
a[2].b = 'f';
```



Δείκτες σε δομές

- Η έννοια του δείκτη εφαρμόζεται **και** σε σύνθετους τύπους δεδομένων, π.χ. `pointer-to-struct`
- Πρόσβαση στην δομή που δείχνει ένας δείκτης μπορεί να γίνει «συμβατικά», με εφαρμογή του τελεστή `*`
- Πρόσβαση στα επιμέρους πεδία της δομής που δείχνει ένας δείκτης μπορεί να γίνει μέσω του τελεστή **`->`**
- Ισχύουν οι παρατηρήσεις που έχουν ήδη γίνει για τους δείκτες σε βασικούς τύπους δεδομένων
  - μια μεταβλητή `array-of-T` είναι στην ουσία ένας σταθερός δείκτης στο πρώτο στοιχείο του πίνακα
  - ένας `pointer-of-T` μπορεί να θεωρηθεί ως η αρχή ενός πίνακα από στοιχεία τύπου `T`
  - αριθμητική με δείκτες

```c
struct date {
  int day, month, year;
};

int main(int argc, char *argv[]) {
  struct date d[2], *p;

  d[0].day = 1;
  d[0].month = 1;
  d[0].year = 2019;

  p = d;                /* p points to d[0] */

  (*p).month = 12;      /* d[0].month is 12 */
  p->year++;            /* d[0].year is 2020 */
  printf("d[0]: %d/%d/%d\n", d[0].day, d[0].month, d[0].year);

  d[1] = *p;            /* d[1] = d[0] */

  p++;                  /* p points to d[1] */

  (*p).month = 1;       /* d[1].month is 1 */
  p->year++;            /* d[1].year is 2021 */
  printf("d[1]: %d/%d/%d\n", d[1].day, d[1].month, d[1].year);

  return(0);
}
```

Δομές ως παράμετροι συναρτήσεων

- Ισχύουν όλες οι παρατηρήσεις που έχουν ήδη γίνει για τους βασικούς τύπους δεδομένων
- Το πέρασμα παραμέτρων γίνεται **καθ' αποτίμηση**
  - ως πραγματική παράμετρος αντιγράφεται (στην στοίβα) (ολόκληρο) το **περιεχόμενο** της δομής
- Οι όποιες αλλαγές γίνονται από τον κώδικα της συνάρτησης στις τιμές των τυπικών παραμέτρων **δεν** είναι ορατές στο εξωτερικό περιβάλλον που κάλεσε την συνάρτηση
- Αν θέλουμε η συνάρτηση να αλλάξει τα περιεχόμενα μιας εξωτερικής μεταβλητής που είναι δομή, ως παράμετρος πρέπει να περαστεί η **διεύθυνση** της
  - η αντίστοιχη τυπική παράμετρος πρέπει να είναι δείκτης

```c
struct date {
  int day, month, year;
};

void incday0(struct date d) {
  d.day++;
}

int main(int argc, char *argv[]) {
  struct date d;

  d.day = 25;
  d.month = 12;
  d.year = 2019;
  printf("d: %d/%d/%d\n", d.day, d.month, d.year);

  incday0(d);
  printf("d: %d/%d/%d\n", d.day, d.month, d.year);

  return(0);
}
```

```c
struct date {
  int day, month, year;
};

void incday1(struct date *d) {
  d->day++;
}

int main(int argc, char *argv[]) {
  struct date d;

  d.day = 25;
  d.month = 12;
  d.year = 2019;
  printf("d: %d/%d/%d\n", d.day, d.month, d.year);

  incday1(&d);
  printf("d: %d/%d/%d\n", d.day, d.month, d.year);

  return(0);
}
```

Σύγκριση ανάμεσα σε δομές

- **Δεν** υποστηρίζεται η σύγκριση ανάμεσα σε `struct`
  - ακόμα και αν έχουν τον ίδιο / συμβατό τύπο
- Ο μεταφραστής **δεν** γνωρίζει την **σημασία** που δίνει ο προγραμματιστής στα επιμέρους πεδία της δομής
  - η «προφανής» σύγκριση byte προς byte δεν είναι πάντα αυτό που θέλουμε και μπορεί να δώσει λάθος αποτελέσματα
- Ο προγραμματιστής πρέπει να υλοποιήσει **δικές του** μεθόδους σύγκρισης
  - με βάση την «σημασία» των επιμέρους πεδίων μιας δομής

- Αντιστοιχία: σύγκριση strings (πίνακες χαρακτήρων)
  - γίνεται μέσω της συνάρτησης `strcmp`

```
struct date {
  int day;
  int month;
  int year;
};

int datecmp(struct date d1, struct date d2) {
  if (d1.year < d2.year)
    return(-1);
  else if (d1.year > d2.year)
    return(1);
  else if (d1.month < d2.month)
    return(-1);
  else if (d1.month > d2.month)
    return(1);
  else if (d1.day < d2.day)
    return(-1);
  else if (d1.day > d2.day)
    return(1);
  else
    return(0);
}
```

Βάση δεδομένων στην μνήμη χρησιμοποιώντας πίνακα από δομές

Παράδειγμα: τηλεφωνικός κατάλογος

Ζητούμενο

- Επιθυμούμε να διαχειριστούμε τα περιεχόμενα ενός (απλού) τηλεφωνικού καταλόγου
- Λειτουργίες: προσθήκη, απομάκρυνση, αναζήτηση

Προσέγγιση

- Ομαδοποιούμε τα δεδομένα κάθε «εγγραφής» μέσω μιας κατάλληλης **δομής**
- Κρατάμε τις εισαγωγές ως ένα **πίνακα από δομές**
- Οι διάφορες λειτουργίες πρέπει να υλοποιηθούν σύμφωνα με τις **εσωτερικές συμβάσεις διαχείρισης** των στοιχείων του πίνακα

Ελεύθερα / υπό-χρήση στοιχεία

- Το μέγεθος του πίνακα εγγραφών προσδιορίζεται **στατικά** την ώρα της μετάφρασης
  - πρέπει να αποφασίσουμε εκ των προτέρων πόσες εγγραφές θα μπορεί να διαχειριστεί το πρόγραμμα

- Οι εγγραφές αποθηκεύονται στον πίνακα **σταδιακά**
  - κάθε φορά που ο χρήστης ζητά να προστεθεί μια εγγραφή
  - στην αρχή, δεν υπάρχει καμία εγγραφή

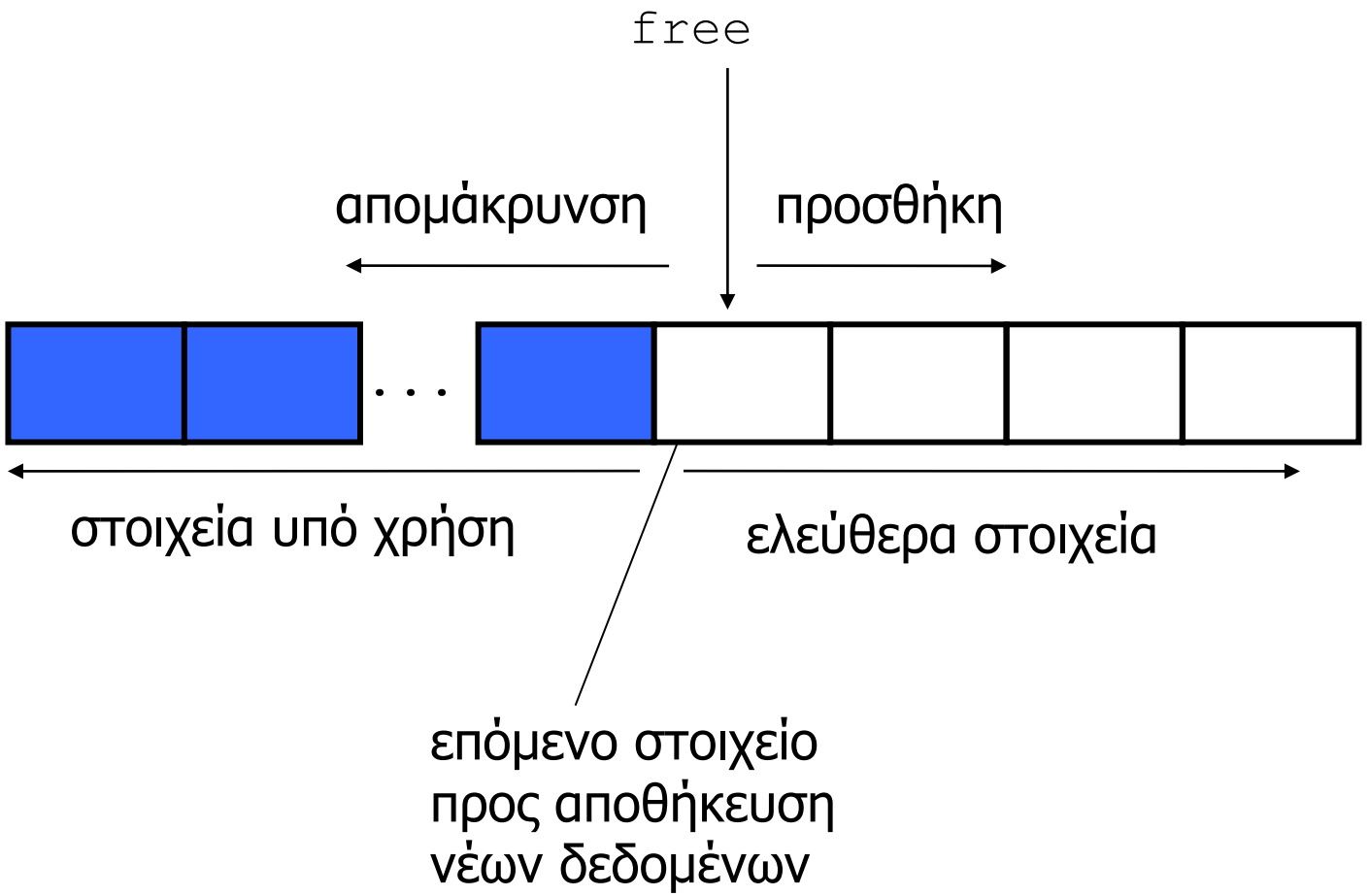
- Ανά πάσα στιγμή, κάποια στοιχεία του πίνακα μπορεί να **χρησιμοποιούνται** για την αποθήκευση εγγραφών ενώ κάποια άλλα να είναι **ελεύθερα**

```
struct entry {
  char name[64];   /* όνομα ως αλφαριθμητικό */
  char phone[32];  /* τηλέφωνο ως αλφαριθμητικό */
};

struct entry entries[MAX_ENTRIES];
```

Προσέγγιση Α

- Όλα τα στοιχεία που χρησιμοποιούνται για την αποθήκευση εγγραφών, βρίσκονται σε **συνεχόμενες** θέσεις του πίνακα
  - π.χ., στην αρχή του πίνακα
- Συνεπώς, και όλα τα ελεύθερα στοιχεία βρίσκονται επίσης σε συνεχόμενες θέσεις του πίνακα
  - π.χ., στο τέλος του πίνακα
- Χρειαζόμαστε μια επιπλέον **μεταβλητή** για να καταγράφουμε το σημείο/θέση διαχωρισμού
- Η λειτουργία της αναζήτησης δεν χρειάζεται να ελέγχει όλα τα στοιχεία του πίνακα
- Όμως στην περίπτωση διαγραφής μιας εγγραφής απαιτεί αντιγραφή/μετακίνηση των εγγραφών

Προσέγγιση Α

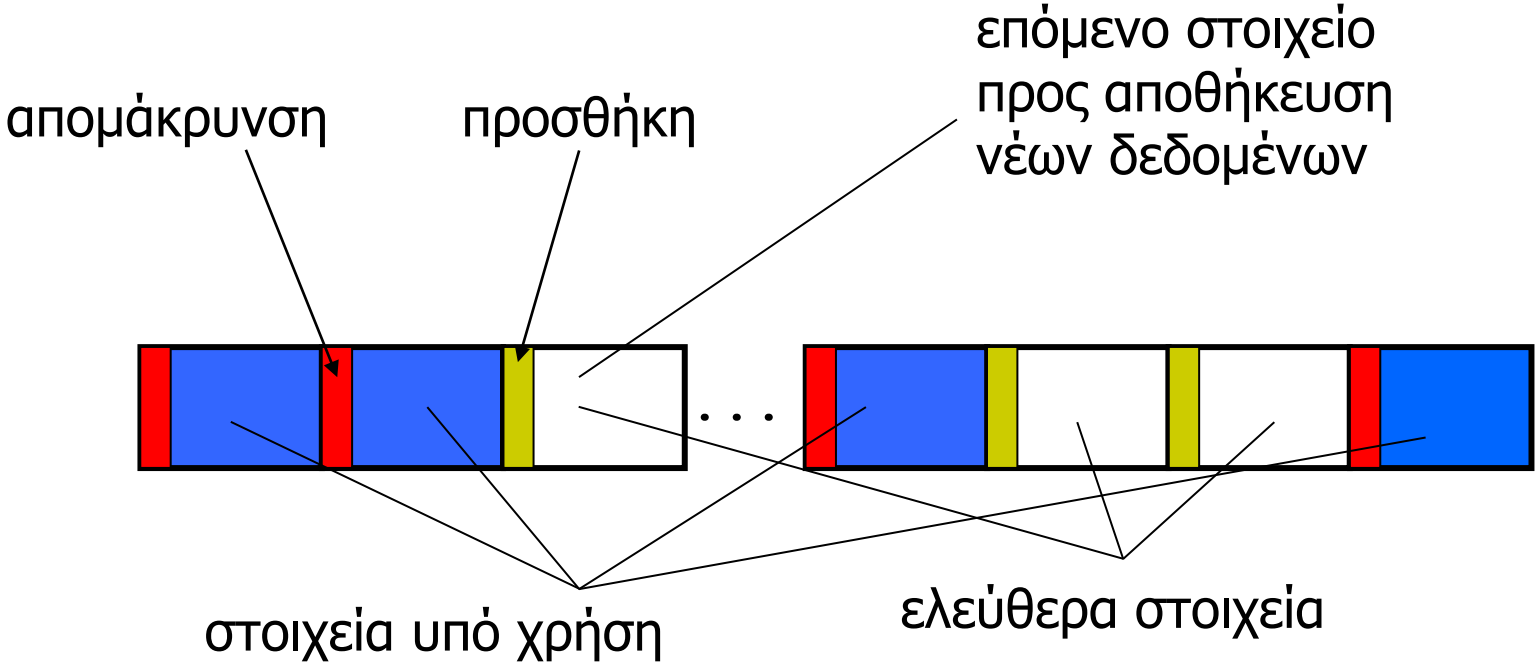


Προσέγγιση Β

- Κάθε στοιχείο του πίνακα έχει ένα **επιπλέον πεδίο** μέσω του οποίου σημειώνεται **ρητά** κατά πόσο αυτό είναι υπό χρήση ή ελεύθερο
- Τα υπό χρήση και τα ελεύθερα στοιχεία του πίνακα μπορεί να βρίσκονται **διάσπαρτα** μέσα στον πίνακα
  - σε μη συνεχόμενες θέσεις

- Αποφεύγεται η αντιγραφή/μετακίνηση των εγγραφών δεδομένων κατά την διαγραφή μιας εγγραφής
- Όμως η λειτουργία της αναζήτησης πρέπει να ελέγχει όλα τα στοιχεία του πίνακα
- Στο παράδειγμα, υιοθετούμε αυτή την προσέγγιση

Προσέγγιση Β



```
struct entry {
  char used;        /* 1 υπό χρήση, 0 ελεύθερο */
  char name[64];    /* όνομα ως αλφαριθμητικό */
  char phone[32];   /* τηλέφωνο ως αλφαριθμητικό */
};

struct entry entries[MAX_ENTRIES];
```

Δόμηση προγράμματος

- Ορίζουμε τις συναρτήσεις για τις βασικές λειτουργίες
  - αρχικά, μόνο τις επικεφαλίδες
  - χωρίς υλοποίηση ή με κενή υλοποίηση
- Γράφουμε την συνάρτηση διαλόγου με τον χρήστη
  - συνήθως η main
  - καλεί τις συναρτήσεις που υλοποιούν τις λειτουργίες
- Υλοποιούμε σταδιακά τις επιμέρους συναρτήσεις
  - βασική παράμετρος κάθε συνάρτησης είναι ο πίνακας εγγραφών (ή ένας δείκτης σε εγγραφή)

```
void phonebook_init(struct entry *entries, int n);
/* αρχικοποιεί τον κατάλογο (ως κενό/άδειο) */

int phonebook_find(const struct entry *entries, int n,
                   const char *name, char *phone);
/* αναζήτηση με το όνομα name και αντιγραφή τηλεφώνου
   στο phone, επιστρέφει 1 για επιτυχία, 0 για αποτυχία */

int phonebook_rmv(struct entry *entries, int n,
                  const char *name);
/* απομάκρυνση της εγγραφής με το όνομα name,
   επιστρέφει 1 για επιτυχία, 0 για αποτυχία */

int phonebook_add(struct entry *entries, int n,
                  const char *name, const char *phone);
/* προσθήκη εγγραφής με όνομα name και τηλέφωνο phone,
   επιστρέφει 1 αν καταχωρηθεί μια νέα εγγραφή,
   -1 για αλλαγή τηλεφώνου μιας υπάρχουσας εγγραφής,
   0 για αποτυχία (δεν υπάρχει χώρος αποθήκευσης */
```

```
void phonebook_init(struct entry *entries, int n) {
  int i;

  for (i=0; i<n; i++) {
    entries[i].used = 0;
  }

}
```

Αναζήτηση

- Η αναζήτηση εγγραφών μέσα στον πίνακα εγγραφών απαιτείται για πολλές διαφορετικές λειτουργίες
  - ρητή αναζήτηση από τον χρήστη
  - αναζήτηση κατά την εισαγωγή
  - αναζήτηση κατά την απομάκρυνση

- Υλοποιούμε την λειτουργία της αναζήτηση μόνο **μια φορά** ως ξεχωριστή συνάρτηση
  - με κατάλληλες παραμέτρους και τιμή επιστροφής
  - ώστε να μπορεί να κληθεί από τις `phonebook_find()`, `phonebook_rmv()` και `phonebook_add()`

<N: θέση στοιχείου που περιέχει την εγγραφή
N: δεν βρέθηκε εγγραφή

```
int find0(const struct entry *entries, int n,
          const char *name){
  int i;

  for (i=0; (i<n) && (!entries[i].used ||
                      strcmp(entries[i].name,name)); i++);

  return(i);
}
```

```
int phonebook_find(const struct entry *entries, int n,
                   const char *name, char *phone) {
  int pos;

  pos = find0(entries, n, name);

  if (pos == n) {
    return(0);
  }
  else {
    strcpy(phone, entries[pos].phone);
    return(1);
  }

}
```

```
int phonebook_rmv(struct entry *entries, int n,
                  const char name[]) {
  int pos;

  pos = find0(entries, n, name);

  if (pos < n) {
    entries[pos].used = 0;
    return(1);
  }
  else {
    return(0);
  }

}
```

```
int phonebook_add(struct entry *entries, int n,
                  const char *name, const char *phone) {
  int pos;

  pos = find0(entries, n, name);

  if (pos < n) {
    strcpy(entries[pos].phone, phone);
    return(-1);  /* existing entry replaced */
  }

  for (pos=0; (pos<n) && (entries[pos].used); pos++);
  if (pos == n) {
    return(0);  /* no free space */
  }

  strcpy(entries[pos].name, name);
  strcpy(entries[pos].phone, phone);
  entries[pos].used = 1;
  return(1); /* new entry added */
}
```

Πιο γρήγορη αναζήτηση

- Για πιο γρήγορη αναζήτηση, μπορούμε να αλλάξουμε υλοποίηση και να υιοθετήσουμε την προσέγγιση Α
  - όμως έχει πιο πολύπλοκη λειτουργία σβησίματος εγγραφών

- Η ταχύτητα της αναζήτησης μπορεί να αυξηθεί ακόμα περισσότερο, αν οι εγγραφές αποθηκεύονται στον πίνακα με αύξουσα/φθίνουσα λεξικογραφική σειρά
  - γιατί;

- Όμως τότε γίνεται πιο πολύπλοκη και η λειτουργία της προσθήκης εγγραφών
  - γιατί;

Συμβολικοί τύποι

Συμβολικοί τύποι

- Με την **`typedef`** μπορεί να οριστούν ονόματα για **συμβολικούς τύπους** δεδομένων
  - με βάση άλλους βασικούς ή σύνθετους τύπους
- Το συντακτικό είναι παρόμοιο με την δήλωση μεταβλητών ενός τύπου `T`, βάζοντας το πρόθεμα `typedef` στην αρχή της δήλωσης
  - `typedef T myNameForT;`
- Η εισαγωγή συμβολικών τύπων γίνεται συνήθως για να βελτιωθεί η αναγνωσιμότητα του κώδικα
- Ο προγραμματιστής μπορεί να αναφέρεται και να χρησιμοποιεί μεταβλητές/αντικείμενα συμβολικού τύπου χωρίς να πρέπει να γνωρίζει τον πραγματικό (πιθανώς σύνθετο) τύπο τους

Τηλεφωνικός κατάλογος (ξανά)

- Ορίζουμε τον ίδιο τον τηλεφωνικό κατάλογο ως ξεχωριστό, συμβολικό τύπο δεδομένων
- Αλλάζουμε τις συναρτήσεις που υλοποιούν τις λειτουργίες έτσι ώστε να παίρνουν ως βασική παράμετρο ένα αντικείμενο τέτοιου τύπου

Τι κερδίσαμε;

- Ο τύπος του τηλεφωνικού καταλόγου και οι λειτουργίες του μπορεί να χρησιμοποιηθούν **χωρίς** να μας απασχολεί η υλοποίηση
- Μπορούμε να αλλάξουμε την υλοποίηση (του τύπου και των λειτουργιών) **χωρίς** να γίνει καμία αλλαγή στις επικεφαλίδες των συναρτήσεων και την σημασία των παραμέτρων τους

```c
#define MAX_ENTRIES 1000

struct entry {
  char used;        /* 1 υπό χρήση, 0 ελεύθερο */
  char name[64];    /* όνομα ως αλφαριθμητικό */
  char phone[32];   /* τηλέφωνο ως αλφαριθμητικό */
};

struct phonebook {
  struct entry entries[MAX_ENTRIES];
};

typedef struct phonebook phonebookT;
```

```
#define MAX_ENTRIES 1000

struct entry {
  char name[64];   /* όνομα ως αλφαριθμητικό */
  char phone[32];  /* τηλέφωνο ως αλφαριθμητικό */
};

struct phonebook {
  struct entry entries[MAX_ENTRIES];
  int nofentries;
};

typedef struct phonebook phonebookT;
```

```
void phonebook_init(phonebookT *pb);
/* αρχικοποιεί τον κατάλογο (ως κενό/άδειο) */

int phonebook_find(const phonebookT *pb,
                   const char *name, char *phone);
/* αναζήτηση με το όνομα name και αντιγραφή τηλεφώνου
   στο phone, επιστρέφει 1 για επιτυχία, 0 για αποτυχία */

int phonebook_rmv(phonebookT *pb,
                  const char *name);
/* απομάκρυνση της εγγραφής με το όνομα name,
   επιστρέφει 1 για επιτυχία, 0 για αποτυχία */

int phonebook_add(phonebookT *pb,
                  const char *name, const char *phone);
/* προσθήκη εγγραφής με όνομα name και τηλέφωνο phone,
   επιστρέφει 1 αν καταχωρηθεί μια νέα εγγραφή,
   -1 για αλλαγή τηλεφώνου μιας υπάρχουσας εγγραφής,
   0 για αποτυχία (δεν υπάρχει χώρος αποθήκευσης */
```

Αφηρημένοι τύποι δεδομένων

- Με την χρήση συμβολικών τύπων μπορεί να υλοποιηθούν **αφηρημένοι τύποι δεδομένων**
  - abstract data types
- Τύποι για τους οποίους ορίζονται συγκεκριμένες λειτουργίες που μπορεί να χρησιμοποιηθούν από τρίτους **χωρίς** αυτοί να πρέπει να γνωρίζουν ή να κατανοούν την «εσωτερική» υλοποίηση
  - όπως ήδη γίνεται για τους βασικούς τύπους δεδομένων
- Η καλή/σωστή **αφαίρεση** (ενίοτε με συνειδητή **απόκρυψη** της υλοποίησης) είναι συχνά απολύτως βασική προϋπόθεση για να μπορούν **άλλοι** να χρησιμοποιήσουν αυτό που φτιάχνεις εσύ ...

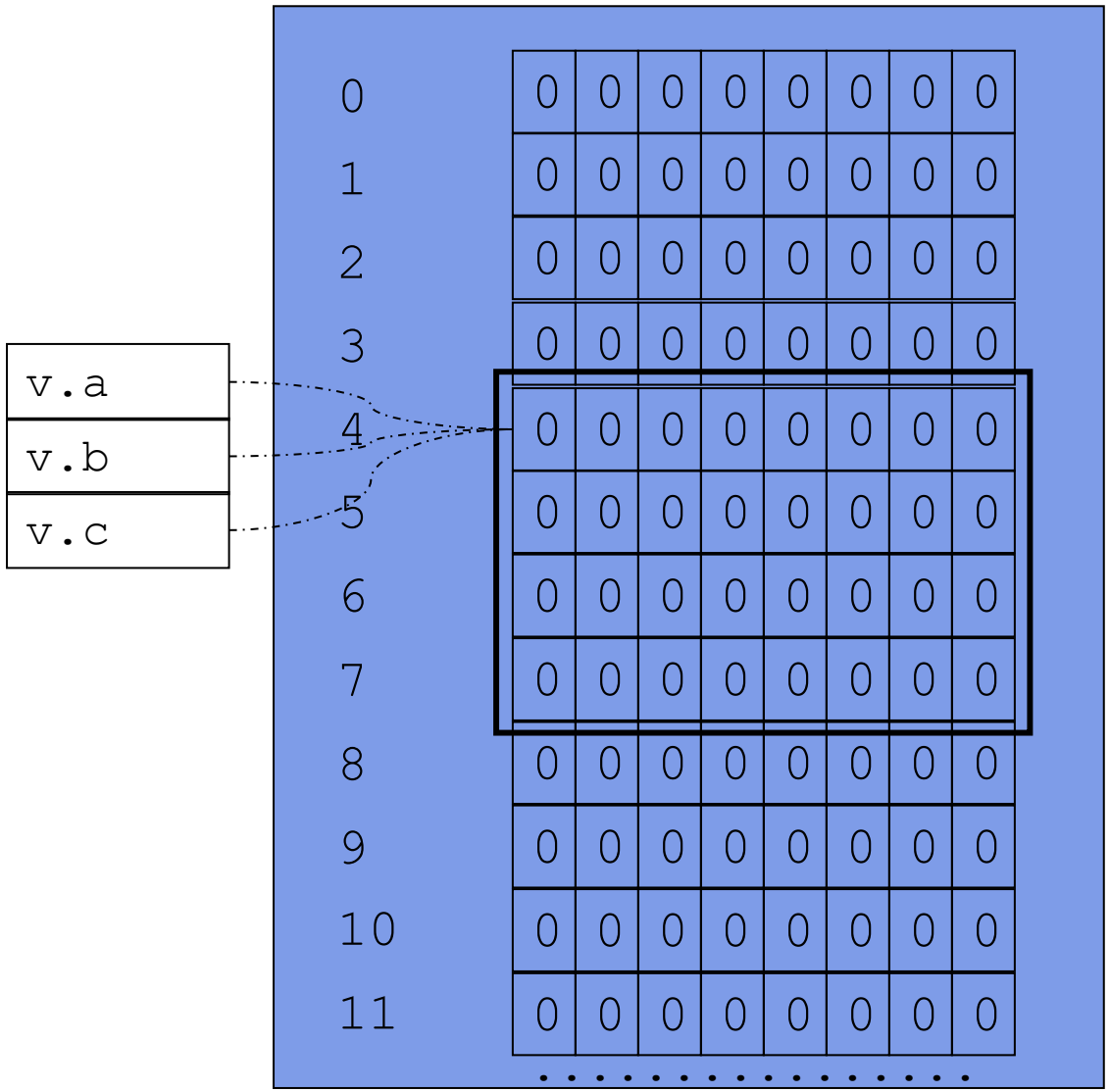
```
struct date {
  …
};

/* αφηρημένος τύπος date_t */
typedef struct date date_t;

/* λειτουργίες πάνω στον τύπο date_t */
void dateinit(date_t *d, int day, int month, int year);
void dateinc(date_t *d);
void datedec(date_t *d);
int datecmp(const date_t *d1, const date_t *d2);
```

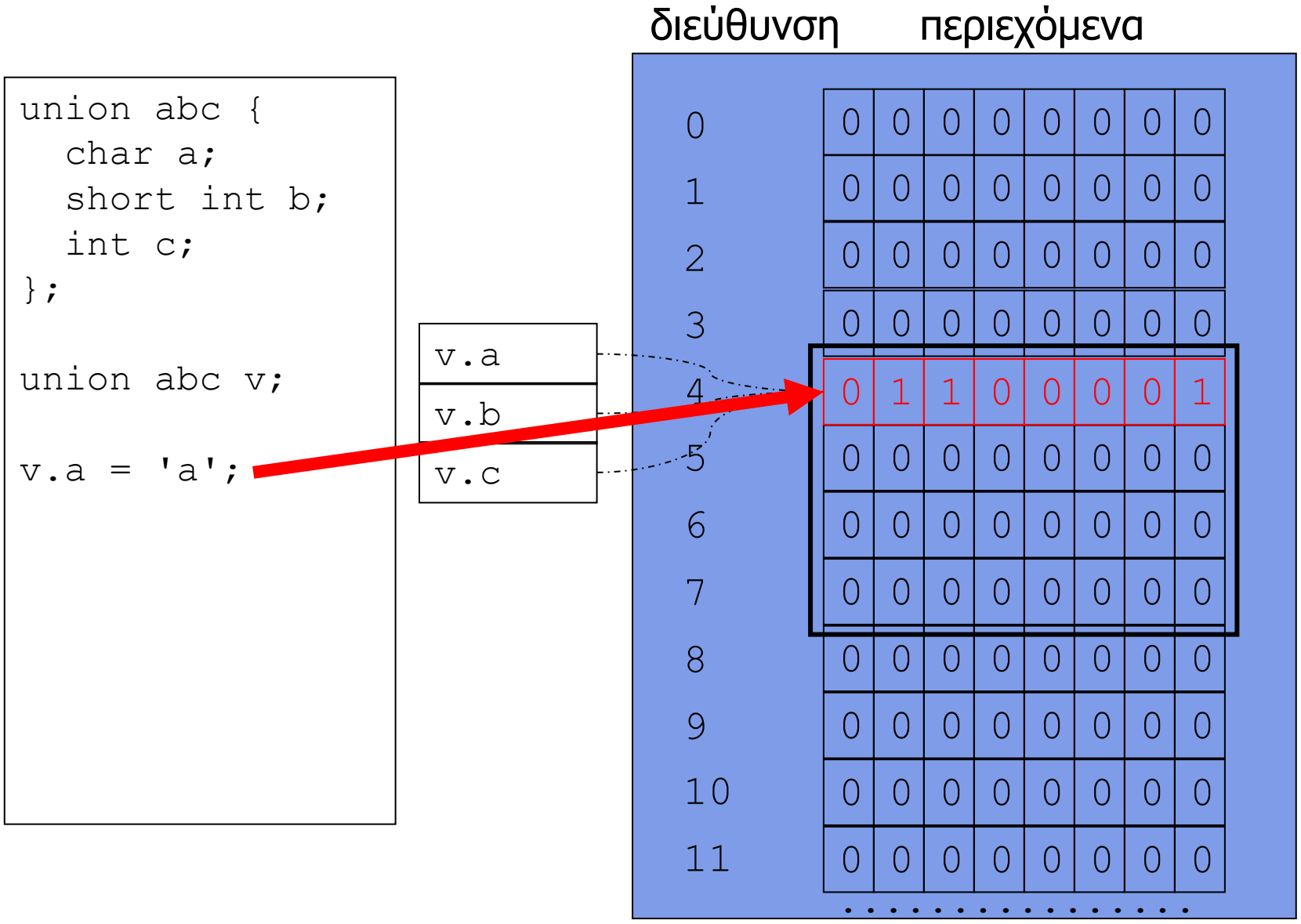
Ενώσεις
(unions)

Ένωση `union`

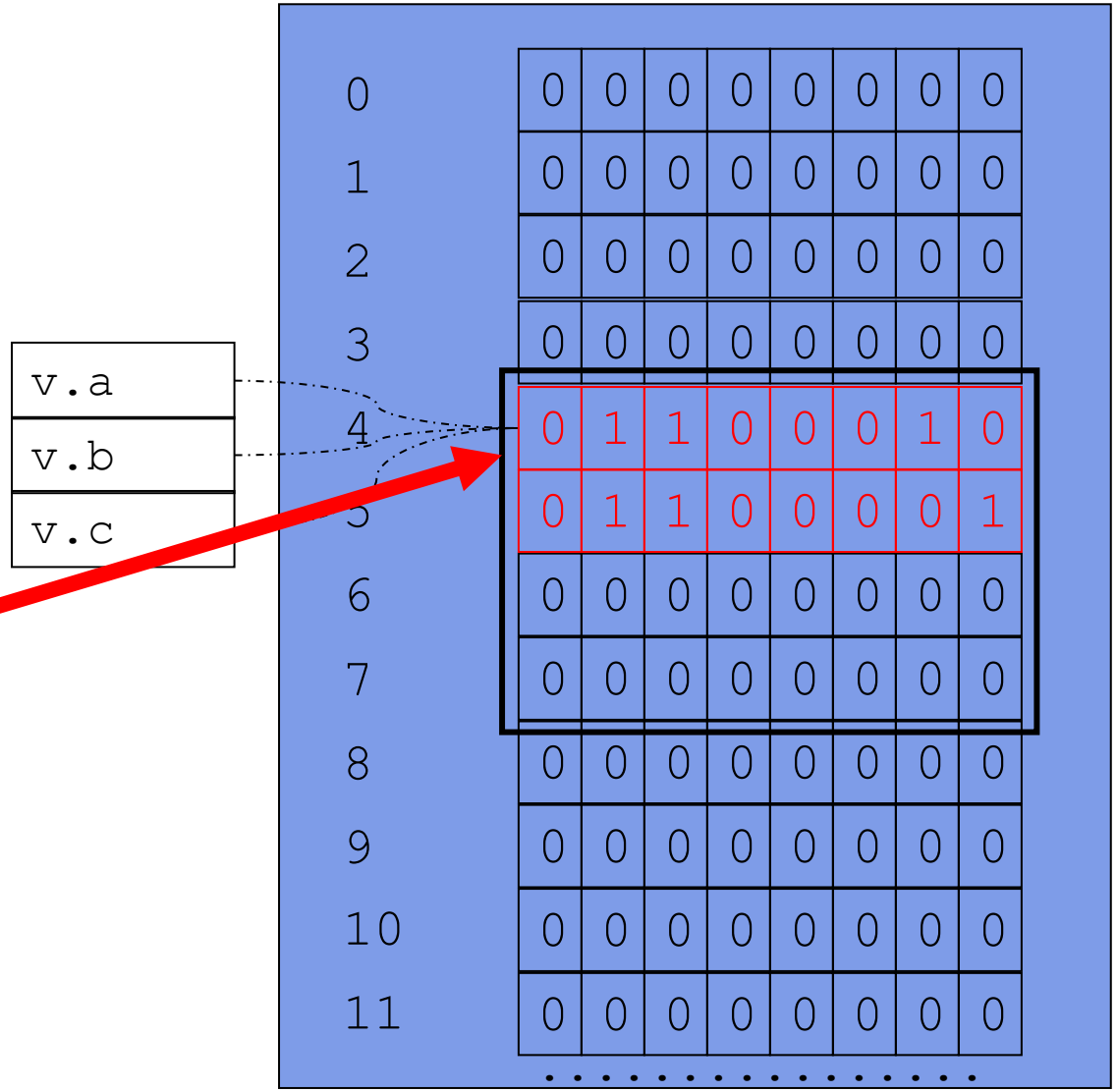
- Ένα διαφορετικό είδος σύνθετου τύπου
- Η ένωση **ομαδοποιεί** δεδομένα διαφορετικών τύπων
- Τα επιμέρους της ένωσης πεδία έχουν **κοινή** μνήμη
  - σε αντίθεση με τα πεδία μιας δομής
  - το μέγεθος της ένωσης καθορίζεται έτσι ώστε να μπορεί να αποθηκευτεί το **μεγαλύτερο** πεδίο της
- Ο προγραμματιστής είναι αποκλειστικά **υπεύθυνος** για την σωστή χρήση των περιεχομένων μιας ένωσης
  - δεν μπορεί να διαπιστωθεί η εγκυρότητα των τιμών των πεδίων, π.χ. σε πιο πεδίο έγινε η τελευταία ανάθεση τιμής
- Συνήθως, η ένωση χρησιμοποιείται **σε συνδυασμό** με την δομή – ένα πεδίο της δομής καθορίζει το **πως** χρησιμοποιείται η ένωση ανά πάσα χρονική στιγμή

```
union abc {
   char a;
   short int b;
   int c;
};

union abc v;
```

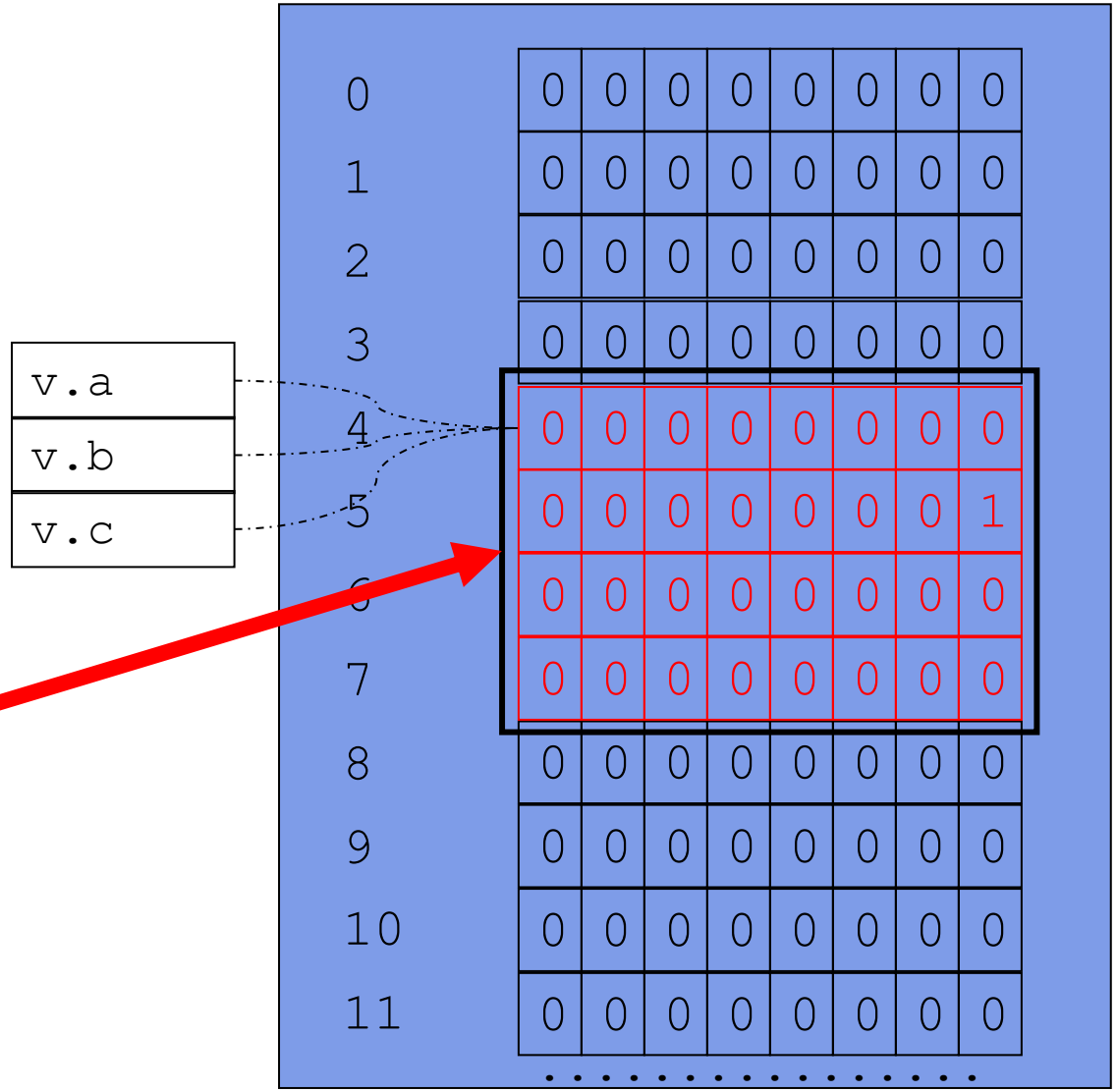



διεύθυνση
περιεχόμενα
union abc {
char a;
short int b;
int c;
};
union abc v;
v.a = 'a';
v.a
v.b
v.c
0
1
2
3
4
5
6
7
8
9
10
11
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

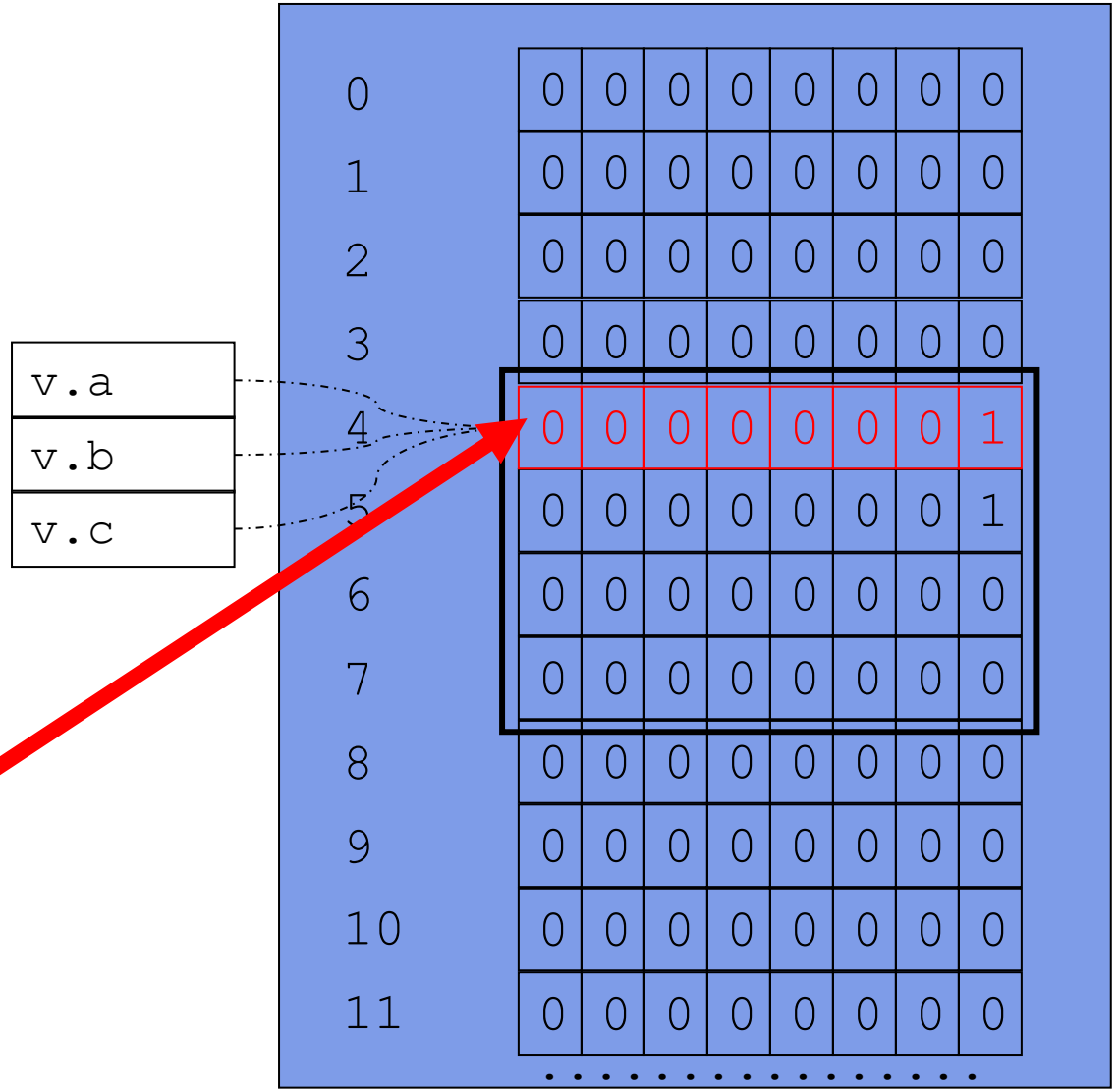
```
union abc {
   char a;
   short int b;
   int c;
};

union abc v;

v.a = 'a';

v.b = 0x6162;
```



```
union abc {
   char a;
   short int b;
   int c;
};

union abc v;

v.a = 'a';

v.b = 0x6162;

v.c = 256;
```



```
union abc {
   char a;
   short int b;
   int c;
};

union abc v;

v.a = 'a';

v.b = 0x6162;

v.c = 256;

v.a = 0x01;
```



Ένωση ως παράμετρος σε συνάρτηση

- Μια συνάρτηση μπορεί έχει ως παράμετρο μια ένωση
  - όπως μπορεί να έχει ως παράμετρο μια δομή
- Στην ουσία, επιτρέπει την συνάρτηση να δέχεται ως πραγματική παράμετρο τιμές **διαφορετικού** τύπου
  - αναλόγως με τα (εναλλακτικά) πεδία της ένωσης
- Η συνάρτηση πρέπει να γνωρίζει πως να ερμηνεύσει τα περιεχόμενα της ένωσης
  - π.χ., με βάση την τιμή μιας άλλης παραμέτρου
- Ο προγραμματιστής είναι αποκλειστικά **υπεύθυνος** να περάσει ως παραμέτρους τις σωστές τιμές κατά την κλήση της συνάρτησης

```c
/* ένωση αποθήκευσης τιμής ως int ή double */

union num {
  int i;       /* τιμή int */
  double d;    /* τιμή double */
};

void printNum(int type, union num n) {
  if (type)
    printf("%lf\n", n.d);
  else
    printf("%d\n", n.i);
}

int main(int argc, char *argv[]) {

  printNum(0, (union num)1234);

  printNum(1, (union num)12.34);

  return(0);
}
```

```c
/* δομή αποθήκευσης τιμής ως int ή double με ένωση */

struct num {
  int type;     /* 0 αν ισχύει val.i, 1 για val.d */
  union {
    int i;      /* τιμή int */
    double d;   /* τιμή double */
  } val;
};

void printNum(struct num n) {
  if (n.type)
    printf("%lf\n", n.val.d);
  else
    printf("%d\n", n.val.i);
}

int main(int argc, char *argv[]) {
  struct num n;

  n.val.i = 1234;
  n.type = 0;
  printNum(n);

  n.val.d = 12.34;
  n.type = 1;
  printNum(n);

  return(0);
}
```

Μέγεθος σύνθετων τύπων

Μέγεθος σύνθετων τύπων

- Η **`sizeof()`** έχει εφαρμογή και σε σύνθετους τύπους που ορίζει ο προγραμματιστής
  - όπως για τους βασικούς τύπους δεδομένων
- Επιστρέφει το **μέγεθος** (σε bytes) που καταλαμβάνει η μεταβλητή / ο σύνθετος τύπος
  - **δεν** επιστρέφει απαραίτητα την ίδια τιμή για διαφορετικούς μεταφραστές ή διαφορετικές αρχιτεκτονικές επεξεργαστών
  - μπορεί να υπάρχουν διαφορές στον τρόπο δέσμευσης μνήμης δομών ή/και το μέγεθος των βασικών τύπων
- Μπορούν να γραφτούν **παραμετρικές** εκφράσεις ως προς το μέγεθος των τύπων (σύνθετων και μη) που χρησιμοποιεί το πρόγραμμα
  - χωρίς ο προγραμματιστής να πρέπει να γνωρίζει τα μεγέθη των δεδομένων ή/και να τα / «καρφώσει» στον κώδικα του

```c
#include <stdio.h>

struct s {
  char c;
  int i;
  double d;
};

union u {
  char c;
  int i;
  double d;
};

int main(int argc, char *argv[]) {
  printf("struct s: %ld\n", sizeof(struct s));
  printf("union u:  %ld\n", sizeof(union u));
  return(0);
}
```

Προσδιορισμός μεγέθους πεδίων `int`

- Για τα πεδία μιας δομής που ορίζονται ως `int` μπορεί (προαιρετικά) να οριστεί ο **αριθμός των bits** που χρειάζονται για την αποθήκευση των τιμών που θα ανατεθούν σε αυτά
  - αν το πεδίο τιμών είναι γνωστό εκ των προτέρων
- Η κωδικοποίηση και ερμηνεία των bits είναι **αποκλειστική** ευθύνη του προγραμματιστή

- Ο μεταφραστής μπορεί να εκμεταλλευτεί αυτή την πληροφορία για να δεσμεύσει **λιγότερη** μνήμη για την αποθήκευση της δομής
  - σημαντικό σε συστήματα με περιορισμένη μνήμη
  - το αν και πως θα γίνει αυτό, εξαρτάται από τον μεταφραστή

```c
#include <stdio.h>

struct date {
  unsigned int day;
  unsigned int month;
  unsigned int year;
};

struct c_date {
  unsigned int day:5;    /* 31 διαφορετικές τιμές */
  unsigned int month:4;  /* 12 διαφορετικές τιμές */
  unsigned int year;     /* δεν υπάρχει περιορισμός*/
};

int main(int argc, char *argv[]) {
  printf("date: %ld\n", sizeof(struct date));
  printf("c_date:  %ld\n", sizeof(struct c_date));
  return(0);
}
```

Απαριθμήσεις (enums)

Απαριθμήσεις `enum`

- Αντιστοίχιση **διαδοχικών ακέραιων** τιμών σε μια σειρά από ονόματα (για αναγνωσιμότητα)
  - χωρίς να ενδιαφέρει (πάντα) η τιμή που δίνεται σε αυτά
- Πρακτικό όταν μια μεταβλητή μπορεί να λάβει μόνο περιορισμένες/συγκεκριμένες τιμές
  - ιδίως όταν δεν έχει παίζει ρόλο η «απόλυτη» τιμή της
- Τα ονόματα αντιστοιχίζονται σε διαδοχικές τιμές που **αυξάνουν** με την σειρά που έχουν δοθεί τα ονόματα
- Η αρίθμηση αρχίζει «αυτομάτως» από το `0`
  - μπορεί να προσδιοριστεί συγκεκριμένη τιμή για κάποιο όνομα
  - τα υπόλοιπα ονόματα θα αντιστοιχηθούν σε τιμές ανάλογα με την θέση τους
- Εναλλακτικά: ορισμός σταθερών μέσω `#define`

```
enum weekdays {Mon=1, Tue, Wed, Thu, Fri, Sat, Sun};

enum weekdays d;

for (d = Mon; d <= Sun; d++) {
  …
}
```