Δομές Δεδομένων

Εισαγωγή

Στη γλώσσα C μπορούμε, εκτός από τους τύπους που ήδη υπάρχουν, int, float, double, char, να επινοήσουμε και χρησιμοποιήσουμε καινούριους, δικούς μας τύπους δεδομένων με συνδυασμό των ήδη υπαρχόντων. Αυτούς τους λέμε ***παράγωγους τύπους*** (derived types) και τους ορίζουμε με τη βοήθεια των λεγόμενων ***δομών δεδομένων*** (data structures). Οι πίνακες που έχουμε ήδη εξετάσει μπορούν να θεωρηθούν και αυτοί δομές δεδομένων, μόνο που συνδυάζουν στοιχεία του ίδιου τύπου και έτσι δε χρειάστηκε να ορίσουμε ένα νέο τύπο. Όπως θα δούμε όμως, μπορούμε να συνδυάσουμε και στοιχεία διαφορετικού τύπου σε μία ενιαία οντότητα αρκεί να καθορίσουμε με σαφή και ακριβή τρόπο τη δομή της. Εκτός από την προφανή ευελιξία που επιτρέπει η ύπαρξη ενός τέτοιου προγραμματιστικού εργαλείου, η ύπαρξη των δομών δεδομένων έχει μεγάλη σημασία επειδή αποτελεί τη γέφυρα με τον αντικειμενοστρεφή προγραμματισμό. Πράγματι, ο αντικειμενοστρεφής προγραμματισμός βασίζεται στον ορισμό αντικειμένων με συγκεκριμένες ιδιότητες που άρα είναι καλό να μπορούμε να τις ομαδοποιήσουμε σε μία ενιαία οντότητα, ασχέτως τύπου αυτών. Οι δομές δεδομένων της C είναι παρόμοιες με τις κλάσεις της C++ και της java, αν και κάπως πιο περιορισμένων δυνατοτήτων.

Ορισμός και συζήτηση της έννοιας

Δομή δεδομένων ονομάζουμε ένα σύνολο από μεταβλητές διαφορετικών εν γένει τύπων τις οποίες ομαδοποιούμε έτσι ώστε να αναφερόμαστε σε αυτές με ένα κοινό όνομα. Συνήθως, οι δομές δεδομένων ορίζονται έτσι ώστε οι μεταβλητές που τις αποτελούν να συνδέονται μεταξύ τους με κάποια σχέση ή να έχουν κάποια κοινά σημεία που δικαιολογούν την ομαδοποίησή τους. Έτσι, αν ένα πρόγραμμα επεξεργάζεται τα στοιχεία διαφόρων προσώπων, τότε μπορούμε να ορίσουμε μεταβλητές όπως Ονοματεπώνυμο, διεύθυνση, ηλικία, τηλέφωνο, του κατάλληλου τύπου η κάθε μία, τις οποίες μπορούμε να ομαδοποιήσουμε σε μία δομή δεδομένων. Αυτές οι μεταβλητές είναι τα ***μέλη*** της δομής αυτής. Οι μεταβλητές αυτές έχουν όνομα, τύπο και μέγεθος, όπως και κάθε μεταβλητή που έχουμε δει μέχρι τώρα. Η δομή σαν σύνολο έχει ένα όνομα που λέγεται ***ετικέτα*** (label) της δομής και με το οποίο μπορούμε να αναφερόμαστε σε αυτήν όπως ακριβώς αναφερόμαστε και σε μία απλή μεταβλητή.

Τρόπος δήλωσης και ορισμού

Όπως είπαμε, μία δομή δεδομένων έχει όνομα και μέλη, το καθένα από τα οποία μπορεί να είναι απλή μεταβλητή ή και πίνακας, άρα έχει όνομα, τύπο και μέγεθος. Για να χρησιμοποιήσουμε μία δομή δεδομένων και να έχουμε πρόσβαση στα μέλη της πρέπει πρώτα να ορίσουμε με ακρίβεια την κατασκευή της. Αυτό το κάνουμε με τη δήλωση της δομής που γίνεται με τον εξής τρόπο:

```
struct ετικέτα_δομής
{
     ορισμός πρώτου μέλους; /* με ερωτηματικό σα να είναι δήλωση μεταβλητής! */
     ορισμός δεύτερου μέλους;
     .....
     ορισμός νυοστού μέλους;
};   /* και εδώ ερωτηματικό στο τέλος! */
```

Στο παράδειγμα που προαναφέρθηκε, με τα στοιχεία προσώπων, μπορούμε να ορίσουμε μία δομή της μορφής:

```
struct person {
     char name[30];
     char address [40];
     int age;
     char phone[15];
} ;
```

Με το παραπάνω παράδειγμα, ουσιαστικά ορίσαμε ένα νέο τύπο δεδομένων με το όνομα person, αλλά δεν έχουμε ορίσει ακόμη μία νέα μεταβλητή αυτού του τύπου, δεσμεύοντας μνήμη για αυτήν ή και αποδίδοντας αρχικές τιμές στα μέλη της. Είναι σα να λέμε ότι εκτός από τους τύπους int, float, char κλπ που ξέραμε ήδη, προστέθηκε άλλος ένας, ειδικά για το συγκεκριμένο πρόγραμμα, που λέγεται person και περιμένει να τον χρησιμοποιήσουμε! Για να το κάνουμε αυτό πρέπει να δηλώσουμε και ορίσουμε επίσης και μεταβλητές αυτού του τύπου. Υπάρχουν διάφοροι τρόποι για να το κάνουμε:

1. Κατά την ίδια τη δήλωση της δομής, π.χ.

```
struct person {
     char name[30];
     char address[40];
     int age;
     char phone[15];
} Maria, Costas, Eleni, Andreas;
```

Στην παραπάνω δήλωση της δομής person έχουμε συγχρόνως και δήλωση των μεταβλητών Maria, Costas, Eleni και Andreas. Αυτές είναι τύπου struct person, όπως ακριβώς άλλες μεταβλητές που είχαμε δει μέχρι τώρα ήταν τύπου int, float, char.

Στη δήλωση μιας μεταβλητής με αυτό τον τρόπο μπορούμε ακόμη και να αποδώσουμε αρχική τιμή βάζοντας τις τιμές των μελών σε άγκιστρα, όπως κάνουμε και με τους πίνακες:

```
struct person {
     char name[30];
     char address[40];
     int age;
     char phone[15];
} Mitsos = {"Mitsos Mitsopoulos", "Stadiou 40", 35, "2105555555"};
```

Να σημειωθεί ότι δεν επιτρέπεται απ' ευθείας απόδοση τιμών στο κάθε μέλος.

2. Γράφοντας struct ετικέτα σαν ένα ακόμη όνομα τύπου.
Ένας άλλος τρόπος είναι να χρησιμοποιήσουμε την ετικέτα της δομής όπως χρησιμοποιούμε και τα ονόματα τύπων που ξέραμε μέχρι τώρα, μόνο που τώρα πρέπει να βάλουμε μπροστά και τη λέξη κλειδί struct, π.χ πρώτα δηλώνουμε τη δομή person,

```
struct person {
     char name[30];
     char address [40];
     int age;
     char phone[15];
} ;
```

και σε επόμενες γραμμές του προγράμματος ορίζουμε μεταβλητές τύπου person ως εξής:

```
struct person Maria, Costas, Eleni, Andreas;
```

Και πάλι, μπορούμε να αποδώσουμε αρχικές τιμές κατά τη δήλωση των μεταβλητών:

```
struct person Mitsos = {"Mitsos Mitsopoulos", "Stadiou 40", 35,
"2105555555"};
```

Προφανώς ο δεύτερος τρόπος για δήλωση μεταβλητών και απόδοση αρχικών τιμών είναι πιο ευέλικτος γιατί αρκεί να δηλώσουμε μια φορά τη δομή και μετά χρησιμοποιούμε μόνο την ετικέτα της.

3. Χρήση typedef.

Υπάρχει και τρίτος τρόπος που συντομεύει ακόμη πιο πολύ την αναφορά σε μια δομή παραλείποντας τη λέξη struct. Αυτό μπορεί να γίνει δίνοντας ένα ψευδώνυμο στη δομή πράγμα που κάνουμε με τη βοήθεια της δεσμευμένης λέξης typedef. Συγκεκριμένα, μπορούμε να γράψουμε το εξής:

```
typedef struct person someone;
```

το οποίο σημαίνει ότι η λέξη someone αναφέρεται σε δομές δεδομένων τύπου person και μπορούμε κάλλιστα να γράψουμε

```
someone Maria, Costas, Eleni, Andreas;
```

και

```
someone Mitsos = {"Mitsos Mitsopoulos", "Stadiou 40", 35, "2105555555"};
```

Αξίζει να σημειωθεί ότι την typedef μπορούμε να χρησιμοποιήσουμε ακόμη και για τους υπάρχοντες τύπους, π.χ. να γράψουμε

```
typedef float real;
```

και στο εξής μέσα στο πρόγραμμά μας θα μπορούμε να δηλώνουμε πραγματικές μεταβλητές απλής ακρίβειας με τη λέξη real αντί για τη float. Δηλαδή, γενικά μπορούμε να αποδώσουμε ένα νέο όνομα σε έναν ήδη υπάρχοντα τύπο με τον εξής τρόπο:

```
typedef τύπος νέο_όνομα;
```

Η δήλωση typedef δεν δημιουργεί νέο τύπο αλλά μόνο ένα νέο *όνομα* για έναν ήδη υπάρχοντα τύπο. Είναι πολύ χρήσιμη όχι μόνο γιατί επιτρέπει να χρησιμοποιούμε ευνόητα μνημονικά ονόματα για τις δομές, αλλά και επιτρέπει πιο εύκολη συντήρηση ενός προγράμματος. Π.χ. αν υπάρχει ενδεχόμενο να πρέπει αλλάζουμε από float σε double όταν μεταφέρουμε το πρόγραμμα σε άλλη πλατφόρμα μπορούμε να ορίσουμε typdef float real; σε ένα αρχείο επικεφαλίδας και να αλλάξουμε μόνο αυτό σε double στη νέα πλατφόρμα.

Όσον αφορά τον πρώτο τρόπο δήλωσης και απόδοσης αρχικών τιμών, μπορεί κανείς να τον χρησιμοποιήσει και χωρίς ετικέτα δηλαδή να γράψει

```
struct {
     char name[30];
     char address[40];
     int age;
     char phone[15];
} Mitsos = {"Mitsos Mitsopoulos", "Stadiou 40", 35, "2105555555"};
```

αλλά τότε δε θα μπορεί να ξαναχρησιμοποιήσει τη δομή αφού δεν έχει τρόπο αναφοράς σε αυτή. Άρα, δεν είναι ιδιαίτερα χρήσιμη πρακτική.

Συχνά, είναι πρακτικό οι δομές να δηλώνονται "εξωτερικά" σε σχέση με τις συναρτήσεις και γι' αυτό είναι καλό να δημιουργούμε ένα αρχείο επικεφαλίδας όπου συγκεντρώνουμε τις δηλώσεις των δομών δεδομένων και τους ορισμούς τύπων με την typedef και να περιλαμβάνουμε αυτό το αρχείο όπου χρειαστεί. Έτσι, μπορούμε να δηλώσουμε μια φορά κάποιες χρήσιμες δομές στο πρόγραμμά μας, που να μπορούν να χρησιμοποιηθούν σε όσες συναρτήσεις θέλουμε.

Κλείνοντας σημειώνουμε ότι ένα μέλος μπορεί να έχει ίδιο όνομα με μια κανονική μεταβλητή (δηλαδή όχι μέλος) ή ακόμη και με το μέλος μιας άλλης δομής αφού τα ονόματα διακρίνονται μέσω του πλαισίου αναφοράς (context), αλλά για λόγους καλού στυλ το αποφεύγουμε αν δεν υπάρχει στενή σχέση ανάμεσα στα αναπαριστώμενα αντικείμενα.

Στις δομές επιτρέπεται να γίνουν οι εξής πράξεις:

- απόδοση τιμής στα μέλη τους,
- αντιγραφή σαν σύνολο,
- λήψη διεύθυνσης και
- προσπέλαση ενός μέλους.

Αυτά παρουσιάζονται στη συνέχεια.

Επεξεργασία αντιτύπων των δομών
Αφού δηλώσαμε μία δομή και ορίσαμε διάφορα ***αντίτυπα*** αυτής (δηλαδή μεταβλητές αυτού του τύπου) θέλουμε να επεξεργαστούμε και πρώτα απ' όλα να αποδίδουμε διάφορες τιμές στα μέλη των μεταβλητών αυτών. Την αναφορά στα μέλη την επιτυγχάνουμε με τους ***τελεστές μέλους***, και συγκεκριμένα με τον ***τελεστή τελείας*** και με τον ***τελεστή βέλους***, αναλόγως αν χρησιμοποιούμε την ίδια τη μεταβλητή-αντίτυπο δομής ή ένα δείκτη που αναφέρεται σε αυτή. Αν αναφερόμαστε στην ίδια τη μεταβλητή μπορούμε να γράψουμε:

```
όνομα_αντίτυπου.όνομα_μέλους
```

Έτσι για το αντίτυπο Mitsos της δομής person, μπορούμε να αναφερθούμε στην ηλικία γράφοντας

```
Mitsos.age
```

και μάλιστα μπορούμε να αποδώσουμε τιμή γράφοντας

```
Mitsos.age = 35;
```

όπως θα κάναμε και με απλές μεταβλητές που γνωρίσαμε ως εδώ.

Αλλά όπως και με τους άλλους τύπους μεταβλητών έτσι και με τις δομές δεδομένων μπορούμε να ορίσουμε δείκτες σε αυτές. Π.χ. γράφοντας

```
struct person *mitsakos;
```

ορίσαμε ένα δείκτη με το όνομα mitsakos που “δείχνει” σε μία μεταβλητή τύπου person ή, με άλλα λόγια, ένα αντίτυπο της δομής δεδομένων τύπου person. Μπορούμε κάλλιστα να πούμε

```
mitsakos = &Mitsos;
```

δηλαδή να αποδώσουμε τη διεύθυνση της μεταβλητής Mitsos στο δείκτη mitsakos. Μετά μπορούμε να αναφερθούμε στα μέλη του, π.χ. στην ηλικία με τον τελεστή τελείας ως εξής:

```
(*mitsakos).age = 35;
```

Χρησιμοποιήσαμε παρενθέσεις επειδή ο τελεστής τελείας έχει πιο υψηλή προτεραιότητα από τον τελεστή περιεχομένου. Οι τελεστές μέλους μαζί με τις παρενθέσεις των συναρτήσεων και τις αγκύλες των πινάκων έχουν την πιο υψηλή προτεραιότητα από όλους τους τελεστές. Π.χ. **++p.x** *αυξάνει το* **x** *και όχι το* **p***! Αν θέλουμε να αλλάξουμε προτεραιότητα χρησιμοποιούμε παρενθέσεις.*

Οι δείκτες σε δομές χρησιμοποιούνται πολύ συχνά και έτσι επινοήθηκε και ένας ειδικός τελεστής μέλους για αυτή την περίπτωση, ώστε να απαλλαγούμε από τις παρενθέσεις. Αυτός είναι ο τελεστής βέλους και, στο συγκεκριμένο παράδειγμα, χρησιμοποιείται ως εξής:

```
mitsakos->age = 35;
```

Το σύμβολο του τελεστή βέλους είναι ένα “πλην” ακολουθούμενο από ένα σύμβολο ανισότητας “μεγαλύτερο από”.

Οι δομές, εκτός από αρχικές τιμές με τους τρόπους που είδαμε, μπορούν να πάρουν διάφορες τιμές κατά την εκτέλεση του προγράμματος με τους εξής τρόπους:

- αποδίδοντας τιμή σε κάποιο από τα μέλη τους με τη βοήθεια των τελεστών μελών, όπως είδαμε στα τελευταία παραδείγματα.

- αντιγράφοντας μια δομή σε μια άλλη. Είναι επιτρεπτό να χρησιμοποιήσουμε τον τελεστή αντικατάστασης = όπως κάνουμε και με τις απλές μεταβλητές. Έτσι, μπορούμε να γράψουμε

  ```
  struct person Kitsos;
  ```

  και πιο κάτω να γράψουμε:

  ```
  Kitsos = Mitsos;
  ```

  οπότε όλα τα μέλη θα αντιγραφούν ένα-ένα από το δεξί στο αριστερό μέλος σα να είχαμε γράψει

  ```
  Kitsos.name = Mitsos.name;
  Kitsos.address = Mitsos.address;
  Kitsos.age = Mitsos.age;
  Kitsos.phone = Mitsos.phone;
  ```

- Μέσω συναρτήσεων. Αυτό σημαίνει ότι μπορούμε να ορίσουμε
  - συναρτήσεις που επιιστρέφουν δομές,
  - συναρτήσεις που επιστρέφουν δείκτες σε δομές,
  - συναρτήσεις με παραμέτρους μέλη δομών, ολόκληρες δομές ή δείκτες σε δομές

  και οποιονδήποτε συνδυασμό των παραπάνω. Ειδικά για το θέμα των παραμέτρων πρέπει να πούμε ότι *αντίθετα από τους πίνακες, οι δομές περνάνε μέσω τιμής και όχι μέσω διεύθυνσης*. Αν θέλουμε να περάσουμε ένα αντίτυπο δομής μέσω διεύθυνσης πρέπει να το δηλώσουμε αναλόγως στη δήλωση και τον ορισμό της συνάρτησης, όπως κάνουμε και με τις απλές μεταβλητές. Αν μια μεγάλη δομή πρόκειται να περαστεί σε μια συνάρτηση είναι πιο αποτελεσματικό να περαστεί με δείκτη γιατί το πέρασμα μέσω τιμής απαιτεί την αντιγραφή ολόκληρης της δομής. Αν η συνάρτηση καλείται συχνά αυτό θα έχει επιπτώσεις στην ταχύτητα εκτέλεσης.

Τα μέλη των δομών χρησιμοποιούνται όπως και τα ονόματα των απλών μεταβλητών. Έτσι, αν σε μια συνάρτηση περνάει ο δείκτης mitsakos και μέσα σε αυτή τη συνάρτηση καλείται η scanf για να διαβάσει τα μέλη του, τότε θα γράψουμε

```
scanf("%s", mitsakos->name);
scanf("%s", mitsakos->address);
scanf("%d", &mitsakos->age);
scanf("%s", mitsakos->phone);
```

Στις παραπάνω εντολές, προσέξτε πού μπαίνει και πού δεν μπαίνει ο τελεστής διεύθυνσης (&)! Η συνάρτηση scanf παίρνει διευθύνσεις μεταβλητών, γι' αυτό μπροστά από το όρισμα μπαίνει ο τελεστής διεύθυνσης &. Οι μεταβλητές mitsakos->name, mitsakos->address και mitsakos->phone είναι πίνακες χαρακτήρων και έτσι είναι συνώνυμα της διεύθυνσης του πρώτου στοιχείου τους. Γι' αυτό δεν γράψαμε το & στις παραπάνω εντολές. Η μεταβλητή mitsakos->age είναι ακέραια και θέλει το & όπως θα το ήθελε κάθε άλλη ακέραια μεταβλητή. Δε χρειάζονται παρενθέσεις επειδή ο τελεστής βέλους, ->, έχει μεγαλύτερη προτεραιότητα από το & οπότε είναι το ίδιο σα να γράφαμε `&(mitsakos->age)`.

Περαιτέρω χρήση των δομών
Οι δομές μπορούν να χρησιμοποιηθούν ως εξής:

- Για να ορίσουμε αντίτυπά τους που λειτουργούν ακριβώς όπως οι απλές μεταβλητές, δηλαδή να τους αποδίδουμε τιμές, να τα αντιγράφουμε, να τα περνάμε ως παραμέτρους σε συναρτήσεις ή να ορίζουμε δείκτες που να αποθηκεύουν τις διευθύνσεις τους.
- Ως τύποι συναρτήσεων, δηλαδή να ορίσουμε συναρτήσεις που επιστρέφουν αντίτυπα δομών

ή δείκτες σε αυτά.

- Για τον ορισμό πινάκων δομών. Μπορούμε να ορίσουμε πίνακες των οποίων τα στοιχεία είναι αντίτυπα δομών ή δεικτών σε αυτά. Π.χ. μπορούμε να ορίσουμε έναν πίνακα

```
struct person myfriends[] = {Costas, Maria, Eleni, Andreas, Mitsos};
```

Σε αυτό το παράδειγμα `myfriends[0].name` υποδηλώνει το μέλος name του πρώτου στοιχείου του πίνακα myfriends, ενώ αν γράφαμε `myfriends.name[0]` θα σήμαινε ένα αντίτυπο δομής που θα λεγόταν myfriends και θα είχε ως μέλος έναν πίνακα name από τον οποίο και ανακτούμε το πρώτο στοιχείο.

- Για τον ορισμό μελών άλλων δομών ορίζοντας ένθετες δομές (nested structures). Για παράδειγμα μπορώ να ορίσω μια δομή για ημερομηνίες με μέλη μέρα, μήνα και έτος

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

και μετά να ορίσω μια βελτιωμένη έκδοση της δομής person ως εξής:

```
struct person {
     char name[30];
     char address [40];
     char phone[15];
     struct date birthday;
};
```

Τότε μπορώ να αναφερθώ στο έτος γέννησης ενός αντίτυπου της δομής ως εξής:

```
Mitsos.birthday.year
```

Το μέλος μιας δομής μπορεί να είναι και δείκτης σε μια άλλη δομή.

- Για τον ορισμό ***αυτοαναφορικών δομών***. Δηλαδή, μπορούμε να ορίσουμε μία δομή που περιλαμβάνει ένα ή περισσότερα μέλη-δείκτες *του ιδίου τύπου με τον εαυτό της*. Ένα αντίτυπο μιας δομής δε μπορεί να περιέχει ένα άλλο αντίτυπο της ίδιας, αλλά δεν υπάρχει κανένα πρόβλημα αν απλώς δείχνει σε ένα άλλο αντίτυπο. Αυτό επιτρέπει ένα πλήθος από ενδιαφέρουσες κατασκευές ώστε αξίζει να εξεταστούν ιδιαίτερα.

Αυτοαναφορικές δομές

Οι αυτοαναφερόμενες ή αυτοαναφορικές δομές, δηλαδή οι δομές που δείχνουν στον εαυτό τους λέγονται και ***δυναμικές*** δομές επειδή η μνήμη που καταλαμβάνουν τα αντίτυπά τους δεν είναι στατικά ορισμένη αλλά μπορεί να μεταβάλλεται “δυναμικά” στη διάρκεια εκτέλεσης του προγράμματος, δηλαδή υφίσταται ***δυναμική κατανομή*** (dynamic allocation) ανάλογα με τις ανάγκες της στιγμής. Οι εφαρμογές των αυτοαναφορικών δομών είναι ενδιαφέρουσες επειδή δείχνουν τη χρησιμότητα των δεικτών και των δομών δεδομένων μέσα από τη συνδυασμένη χρήση τους.

Παράδειγμα αυτοαναφορικής δομής [Σεφερίδης 1995]

```
struct element {
     int value;
     struct element *link;
};
```

Στην παραπάνω δομή διακρίνουμε το *ωφέλιμο φορτίο* (payload) που εν προκειμένω είναι το μέλος value και το δείκτη (εδώ, το μέλος link) που χρησιμεύει στη διασύνδεση με ένα άλλο αντίτυπο της δομής. Ο δείκτης αυτός μπορεί να μη δείχνει πουθενά αλλά να είναι ίσος με μηδέν, οπότε λέμε ότι έχει την τιμή NULL που είναι μια συμβολική σταθερά ορισμένη ίση με 0 στο stdio.h (το μηδέν δε θεωρείται έγκυρη διεύθυνση δεδομένων και γι' αυτό χρησιμοποιείται όταν δε θέλουμε ένας δείκτης να δείχνει κάπου συγκεκριμένα).

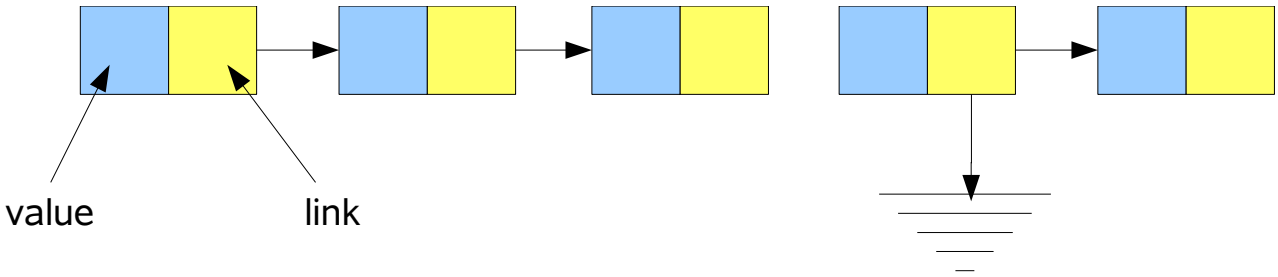
Με την αυτοαναφορά μπορούμε να συνδέσουμε πολλά αντίτυπα της ίδιας δομής το ένα με το άλλο κάνοντας πολύπλοκες κατασκευές που μας παρέχουν μεγάλη ευελιξία στο χειρισμό μαζικών δεδομένων. Η τιμή NULL χρησιμεύει ιδιαίτερα στο να δείχνει το τέλος μιας αυτοαναφερόμενης δομής, δηλαδή ότι δεν υπάρχει άλλο στοιχείο. Η πιο απλή περίπτωση κατασκευών αυτού του είδους είναι αυτή της ***γραμμικής συνδεδεμένης λίστας*** (linear linked list). Βάσει του παραπάνω παραδείγματος μπορούμε να ορίσουμε [Σεφερίδης 1995]

```
struct element one, two, three;
```

και μετά

```
one.link = &two; two.link = &three; three.link = NULL;
```

φτιάχνοντας μία “αλυσίδα” από στοιχεία που καθένα συνδέεται με ένα άλλο, έτσι ώστε προκύπτει μια *διάταξη* με αρχή ή ***κεφαλή*** (head) και τέλος ή ***ουρά*** (tail). Η λίστα του παραδείγματος απεικονίζεται στο παρακάτω σχήμα. Παρατηρείστε ότι κατά τη γραφική απεικόνιση τέτοιων δομών, συνηθίζεται για τους δείκτες με τιμή NULL να χρησιμοποιείται το σύμβολο της γείωσης.



Μία λίστα μπορούμε να τη χειριστούμε ως εξής:

- δημιουργία της λίστας
- εισαγωγή στοιχείων
- διαγραφή στοιχείων
- εμφάνιση στοιχείων
- απαρίθμηση στοιχείων
- συγχώνευση στοιχείων από δύο λίστες.

Πριν μιλήσουμε αναλυτικά για αυτά, μία παρένθεση για τη δυναμική διαχείριση μνήμης. Έχουμε αναφέρει και στο κεφάλαιο για τους πίνακες τις συναρτήσεις malloc και free και τον τελεστή sizeof. Ο τελεστής sizeof μπορεί να πάρει ως όρισμα είτε έναν τύπο (π.χ. int, float) ή ακόμη και μία μεταβλητή, ένα στοιχείο πίνακα κλπ και μας επιτρέπει το μέγεθος της μνήμης που του αντιστοιχεί σε bytes. Η συνάρτηση malloc μπορεί να δεσμεύσει μία περιοχή μνήμης με το μέγεθος που θα της πούμε και να επιστρέψει ένα δείκτη που να δείχνει στην αρχή αυτής της περιοχής. Η συνάρτηση free παίρνει σαν όρισμα το δείκτη μιας δεσμευμένης περιοχής μνήμης και την αποδεσμεύει.

Δημιουργία
Για να κατασκευάσουμε μία γ. σ. λ. ορίζουμε την κεφαλή και την ουρά της. Με τη συνάρτηση malloc και τον τελεστή sizeof δεσμεύουμε μνήμη για την κεφαλή και την ουρά και βάζουμε την κεφαλή να δείχνει στην ουρά και την ουρά στο NULL. Π.χ.

```
struct element *header;
```

```
header = malloc( sizeof( element ) );
header->link = malloc( sizeof( element ) );
header->link->link = NULL;
```

Εισαγωγή στοιχείων
Για να εισάγουμε ένα στοιχείο σε μία λίστα το παρεμβάλλουμε ανάμεσα στην κεφαλή και το αμέσως επόμενο, ακολουθώντας τα εξής βήματα:

α) δεσμεύουμε μνήμη για ένα ακόμη αντίτυπο της δομής, έστω X

β) το βάζουμε να δείχνει εκεί όπου δείχνει και η κεφαλή

γ) βάζουμε την κεφαλή να δείχνει στο X, π.χ.

```
struct element *new_element
new_element = malloc( sizeof( element ) );
new_element->link = header->link;
header->link = new_element;
```

Διαγραφή στοιχείων
Για να διαγράψουμε ένα στοιχείο X από μία λίστα γνωρίζοντας το προηγούμενο αυτού, ακολουθούμε τα εξής βήματα:

α) βάζουμε το προηγούμενο να δείχνει εκεί όπου δείχνει το X(δηλαδή το παρακάμπτουμε)

β) με τη συνάρτηση free αποδεσμεύουμε τη μνήμη που κατείχε το X, π.χ.

```
if(current->link != NULL) {
     previous->link = current->link;
     free(current);
}
```

Εμφάνιση και απαρίθμηση στοιχείων
Ξεκινώντας από ένα στοιχείο της λίστας ελέγχουμε αν δείχνει στο 0 και όσο δεν ισχύει αυτό μπορούμε να εμφανίζουμε και απαριθμούμε τα μέλη του, π.χ.

```
kount = 0;
while(current->link != NULL) {
     printf("\nvalue number %d = %d", kount++, current->value);
     current = current->link;
}
```

Συγχώνευση στοιχείων από δύο λίστες
Η πιο απλή περίπτωση είναι να κάνουμε το τελευταίο στοιχείο πριν την ουρά της μίας να δείχνει στο πρώτο στοιχείο μετά την κεφαλή της άλλης και να απελευθερώσουμε το χώρο από την ουρά και την κεφαλή που περίσσεψαν, π.χ.

```
while(current->link->link != NULL) ;
struct *element tail1 = current->link;
current->link = head2->link;
free(tail1);
free(head2);
```

Αυτό είναι ένα άλλο ένα είδος “παράκαμψης” όπως εκείνο στην περίπτωση διαγραφής στοιχείου από λίστα.

Αυτή η κατασκευή μπορεί, όπως και οι πίνακες, να αποθηκεύσει πολλά στοιχεία αλλά πλεονεκτεί σε όρους ταχύτητας, ευελιξίας και οικονομίας μνήμης. Η διαδικασία για την ένθεση ή διαγραφή στοιχείων απαιτεί την εναλλαγή δεικτών και διαρκεί τον ίδιο (μικρό) χρόνο ανεξάρτητα από το μέγεθος της λίστας. Έτσι, το μέγεθος της λίστας μπορεί να αυξομειωθεί ανάλογα με τις ανάγκες του χρόνου εκτέλεσης, ενώ δε χρειάζεται αποθήκευση των δεδομένων σε συνεχόμενες θέσεις της μνήμης όπως γίνεται με τους πίνακες και άρα δε χρειάζεται να σπαταληθεί υπολογιστικός χρόνος σε αντιγραφή στοιχείων ενώ η καταναλισκόμενη μνήμη περιορίζεται κάθε φορά στην απολύτως αναγκαία.

Μειονέκτημα της γραμμικής λίστας είναι ότι για να εμφανίσουμε ή χειριστούμε ένα στοιχείο που βρίσκεται οπουδήποτε μέσα στη λίστα, πρέπει να τη διατρέξουμε και να "αναγνωρίσουμε" αυτοό το στοιχείο, π.χ. από την τιμή ενός μέλους του που είναι το "κλειδί" αναζήτησης, ενώ στους πίνακες μπορούμε να ανακτήσουμε απευθείας κάθε στοιχείο τους με τη βοήθεια του αριθμοδείκτη τους. Γι' αυτό επιδιώκουμε να χρησιμοποιούμε συνδυασμούς πινάκων και δυναμικών δομών που συνδυάζουν, ανάλογα και με τη φύση του προβλήματος, τα πλεονεκτήματα και των δύο τρόπων αποθήκευσης δεδομένων.

Το παραπάνω είδος λίστας υλοποιεί μία στοίβα που υπακούει στην αρχή LIFO γιατί αν θέλουμε να προσπελάσουμε τα στοιχεία της το πρώτο στο οποίο έχουμε πρόσβαση είναι αυτό που βρίσκεται στην κορυφή (head->link) δηλαδή αυτό που μπήκε τελευταίο.

Άλλες κατασκευές παρόμοιου είδους είναι οι ουρές (αρχή FIFO), οι διπλές λίστες και τα δέντρα.

Ενώσεις και απαριθμητές

Εκτός από δομές δεδομένων μπορούμε να ορίσουμε και άλλες δύο κατηγορίες τύπων.

Ο τρόπος δήλωσης που εφαρμόσαμε για τις δομές χρησιμοποιείται με παρόμοιο τρόπο και για τη δήλωση μιας οντότητας που λέγεται ***ένωση δεδομένων***:

```
union ετικέτα {
      τύπος όνομα_μέλους;
      τύπος όνομα_μέλους;
      ...
}
```

Η ένωση δεν πρέπει να συγχέεται με τη δομή γιατί *περιέχει ένα μόνο δεδομένο κάθε φορά* που χρησιμοποιείται, μόνο που αυτό μπορεί να έχει διαφορετικό τύπο κάθε φορά, ανάλογα με τους τύπους που έχουμε προσδιορίσει στις δηλώσεις των μελών του. Π.χ.

```
union example {
      int i;
      float f;
      double d;
      char c;
};
union example var;
var.i = 1; var.f = 1.1; var.d = 3.141; var.c = 'a';
```

Μία μεταβλητή τύπου που έχει οριστεί μέσω ένωσης καταλαμβάνει μία διεύθυνση μνήμης και χώρο τόσο όσος απαιτείται για να αποθηκεύσει το μεγαλύτερο από τους τύπους των μελών της. Έτσι, η ίδια διεύθυνση μνήμης επιτρέπεται να έχει πολλαπλή ερμηνεία, κάτι χρήσιμο για συσκευές με περιορισμένη μνήμη.

Μπορούν να οριστούν ένθετες ενώσεις, δείκτες σε ενώσεις και να μεταβιβαστούν δείκτες ενώσεων σε συναρτήσεις. Οι ενώσεις μπορούν να εμφανίζονται μέσα σε δομές και πίνακες και το αντίστροφο.

Όπως και με τις δομές, μπορούμε να αποδώσουμε τιμή, να αντιγράψουμε σε άλλη ένωση, να πάρουμε τη διεύθυνση και να προσπελάσουμε ένα μέλος. Αρχική τιμή μιας ένωσης μπορεί να είναι μόνο του τύπου που είναι και το πρώτο κατά σειρά μέλος της.

Προσοχή χρειάζεται στο εξής: ο τύπος που ανακτάται πρέπει να είναι ο τελευταίος που αποθηκεύτηκε. Είναι ευθύνη του προγραμματιστή να παρακολουθεί ποιος τύπος είναι αποθηκευμένος σε μία ένωση σε κάθε στάδιο του προγράμματος.

Οι ***τύποι απαρίθμησης*** ορίζονται αντιστοιχώντας ένα διακριτό σύνολο σταθερών με συμβολικά ονόματα σε μεταβλητές που παίρνουν τιμές μέσα από το σύνολο αυτό. Ορισμός:

```
enum ετικέτα {μέλος1, μέλος2, ... μέλοςN };
```

Τα μέλη είναι οι *απαριθμητές* και η ετικέτα υποδηλώνει συλλογικά τον τύπο δεδομένων που απαρτίζουν οι απαριθμητές. Κάθε απαριθμητής αντιστοιχεί σε μία ακέραια τιμή από το 0 μέχρι το N-1 που αποτελεί την εσωτερική αναπαράσταση των απαριθμητών ενώ τα ονόματά τους είναι συμβολικά για τη διευκόλυνση του προγραμματιστή. Π.χ.

```
enum days {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday};
enum days today, tomorrow;
today = Sunday;
tomorrow = today+1;
```

Μπορούν να αποδοθούν και διαφορετικές *ακέραιες* τιμές από αυτές που αποδίδονται αυτόματα, π.χ.

```
enum days {Sunday=1, Monday=2, Tuesday=3, Wednesday=4, Thursday=5,
Friday=6, Saturday=7};
```

Μια ματιά στον αντικειμενοστρεφή προγραμματισμό: κλάσεις

Μια έννοια εντελώς ανάλογη με τις δομές δεδομένων στη C είναι αυτή της ***κλάσης*** (class) που χρησιμοποιείται στη C++. Μια κλάση μπορεί να περιέχει δηλώσεις μεταβλητών και πρότυπα συναρτήσεων που χειρίζονται αυτές τις μεταβλητές και λέγονται ***μέθοδοι.*** Επιπλέον, τα μέλη μιας κλάσης χωρίζονται σε ***ιδιωτικά*** (private), ***δημόσια*** (public) και ***προστατευμένα*** (protected). Τα ιδιωτικά μέλη είναι ορατά μόνο από τα άλλα μέλη της κλάσης όπου ανήκουν. Τα δημόσια είναι προσπελάσιμα από οπουδήποτε εντός και εκτός κλάσης. Τα προστατευμένα είναι ορατά από τα μέλη της κλάσης τους και όλων των κλάσεων που *παράγονται* από αυτή. Μία κλάση μπορεί να παράγεται από άλλη είτε με ***σύνθεση*** (να έχει ως μέλος την άλλη κλάση) είτε με ***κληρονομικότητα*** (να έχει όλα τα μέλη μιας γενικότερης και πιο αφηρημένης κλάσης, καθώς και άλλα καινούρια που την καθιστούν πιο συγκεκριμένη). Παράδειγμα δήλωσης κλάσης [Loudon 2003]:

```
class Account {
public:
     Account (double b); /* "κατασκευαστής" για τη δέσμευση απαραίτητης μνήμης */
     void deposit(double amt);
     void withdraw(double amt);
     void getBalance( ) const;
private:
     double balance;
};
```