Προγραμματισμός Υπολογιστών C++

Δομές & Ενώσεις

Δομές

- Η δομή είναι μία συλλογή πεδίων, που χρησιμοποιούνται για την ομαδοποίηση πληροφορίας που περιγράφει μία λογική οντότητα

Π.χ. μία δομή μπορεί να περιέχει πληροφορίες για μία εταιρεία, όπως την επωνυμία της, το έτος ίδρυσης, το Α.Φ.Μ, τη διεύθυνσή της, το αντικείμενο εργασιών της, τον αριθμό των υπαλλήλων της, στοιχεία επικοινωνίας και άλλα δεδομένα ή ακόμα και συναρτήσεις που να διαχειρίζονται τα αποθηκευμένα δεδομένα της

Δήλωση Δομής (1)

- Η δήλωση μίας δομής αρχίζει με τη δεσμευμένη λέξη `struct` και στη γενική της μορφή έχει την ακόλουθη σύνταξη:

```
struct ετικέτα_δομής
{
        Δηλώσεις πεδίων;
        Δηλώσεις συναρτήσεων;
} λίστα_μεταβλητών;
```

- Μία `struct` δήλωση δημιουργεί έναν τύπο, ο οποίος ορίζεται από τον προγραμματιστή
- Αν και η `ετικέτα_δομής` είναι προαιρετική, εμείς θα ονοματίζουμε κάθε τύπο δομής που δημιουργούμε, ώστε να μπορούμε να χρησιμοποιούμε την ετικέτα, όποτε θέλουμε, για να δηλώνουμε αντίστοιχες μεταβλητές
- Τα πεδία ή μέλη της δομής, χρησιμοποιούνται όπως και οι μεταβλητές του αντίστοιχου τύπου
- Μία δομή μπορεί να αποτελείται από πεδία διαφορετικού τύπου. Η λογική συσχέτισή τους είναι ότι περιέχουν την πληροφορία που απαιτείται για την περιγραφή μίας συγκεκριμένης οντότητας
- Όπως φαίνεται στη δήλωση, μπορούμε προαιρετικά να δηλώσουμε μία `λίστα_μεταβλητών` αυτού του τύπου

Δήλωση Δομής (2)

- Μία δομή μπορεί να περιέχει συναρτήσεις. Για απλότητα, θα χρησιμοποιήσουμε δομές που περιέχουν μόνο μεταβλητές και όχι συναρτήσεις, γιατί, όπως θα δούμε στο Κ.17, ο τύπος που συνηθίζεται να επιλέγεται για την προσθήκη συναρτήσεων είναι κλάση και όχι δομή
- Η συνήθης πρακτική είναι οι δηλώσεις δομών μαζί με άλλες δηλώσεις, όπως πρωτότυπα συναρτήσεων και μακροεντολές, να αποθηκεύονται σε ένα ξεχωριστό αρχείο επικεφαλίδας το οποίο να συμπεριλαμβάνεται με την οδηγία `#include` όπου χρειάζεται
- Για απλότητα, στα επόμενα παραδείγματα θα δηλώνουμε τον κάθε τύπο δομής με καθολική εμβέλεια στην αρχή του αρχείου, ώστε όλες οι συναρτήσεις να μπορούν να τον χρησιμοποιήσουν

Παράδειγμα Δήλωσης

- Για παράδειγμα, για να αποθηκεύσουμε πληροφορία για μία εταιρεία θα μπορούσαμε να δηλώσουμε τον τύπο Company:

```
struct Company
{
        string name;
        int start_year;
        int field;
        int tax_num;
        int num_empl;
        string addr;
        float balance;
};
```

- Το πρώτο γράμμα της ετικέτας συνηθίζεται να είναι κεφαλαίο
- Αν και τα πεδία του ίδιου τύπου επιτρέπεται να δηλώνονται στην ίδια γραμμή, προτιμώ να τα δηλώνω ξεχωριστά, ώστε να φαίνεται πιο εύκολα η αντιστοίχιση με την πληροφορία που θέλουμε να αποθηκεύσουμε
- Τα πεδία μίας δομής αποθηκεύονται σε θέσεις μνήμης που αυξάνονται, με το πρώτο να αρχίζει από τη διεύθυνση της δομής
- Η δήλωση μίας δομής που δεν ακολουθείται από λίστα μεταβλητών, όπως εδώ της Company, δεν προκαλεί δέσμευση μνήμης. Απλά, περιγράφει τη μορφή της δομής

Δήλωση Μεταβλητών

- Αν η δομή έχει ετικέτα, μπορούμε να τη χρησιμοποιήσουμε για δηλώσεις μεταβλητών. Για παράδειγμα: `Company c1, c2;`
- Για να είναι ξεκάθαρο, η δήλωση της δομής `Company` ορίζει έναν τύπο δεδομένων με αυτό το όνομα, δεν προκαλεί δέσμευση μνήμης και ούτε η `Company` είναι μεταβλητή
- Η `Company` απλά περιγράφει τη μορφή που θα έχουν οι μεταβλητές της δομής όταν αυτές δηλωθούν
- Οι μεταβλητές `c1` και `c2` δηλώνονται σαν δομές τύπου `Company`, η κάθε μία έχει τα δικά της πεδία και ο μεταγλωττιστής δεσμεύει μνήμη για αυτές
- Εναλλακτικά, μπορούμε να δηλώσουμε μεταβλητές στη `λίστα_μεταβλητών`. Π.χ.

```
struct Book
{
        string title;
        int year;
        float price;
} b1, b2;
```

- Οι μεταβλητές `b1` και `b2` δηλώνονται σαν δομές τύπου `Book`

Παρατηρήσεις (1)

- Όπως με κάθε μεταβλητή, όταν δηλώνεται μία δομή ο μεταγλωττιστής δεσμεύει μνήμη για την αποθήκευση των πεδίων της με τη σειρά που εμφανίζονται. Π.χ. το επόμενο πρόγραμμα εμφανίζει πόσες οκτάδες δεσμεύτηκαν για τη δομή **d**:

```
#include <iostream>
struct Date
{
        int day;
        int month;
        int year;
};
int main()
{
        Date d;

        std::cout << sizeof(d) << '\n';
        return 0;
}
```

- Αφού η **d** είναι δομή τύπου **Date**, αποτελείται από τρία ακέραια πεδία, άρα η μνήμη που δεσμεύτηκε είναι 3*4 = 12 οκτάδες. Αν και σε αυτό το παράδειγμα το μέγεθος της δεσμευμένης μνήμης είναι ίσο με το άθροισμα της μνήμης που δεσμεύουν οι τύποι των πεδίων, μπορεί να υπάρχουν περιπτώσεις που να είναι μεγαλύτερο. Για παράδειγμα, αν αλλάξουμε τον τύπο του πεδίου **day** από **int** σε **char**, το πιθανότερο είναι ότι το πρόγραμμα θα εμφανίσει πάλι 12 και όχι 9, όπως θα ήταν το αναμενόμενο
- Αυτό μπορεί να συμβεί, αν ο μεταγλωττιστής, για πιο γρήγορη πρόσβαση στα πεδία της δομής, απαιτεί κάθε πεδίο να αποθηκεύεται σε μία διεύθυνση που να είναι πολλαπλάσια κάποιου αριθμού (συνήθως του 4). Αν υποθέσουμε ότι το πεδίο **month** πρέπει να αποθηκευτεί σε μία διεύθυνση πολλαπλάσια του 4, ο μεταγλωττιστής θα δεσμεύσει τρεις πρόσθετες οκτάδες αμέσως μετά το πεδίο **day**
- Για τον υπολογισμό της μνήμης που καταλαμβάνει μία δομή να χρησιμοποιείτε πάντα τον τελεστή **sizeof** και να **μην** προσθέτετε τη μνήμη που δεσμεύουν ξεχωριστά τα πεδία της

Αρχικοποίηση Πεδίων Δομής (1)

- Ο πιο συνηθισμένος τρόπος για να προσπελάσουμε ένα πεδίο μίας δομής είναι να γράψουμε το όνομά της, να προσθέσουμε τον τελεστή τελεία (.) και μετά το όνομα του πεδίου
- Το επόμενο πρόγραμμα αρχικοποιεί και εμφανίζει τις τιμές των πεδίων της **b**

```
#include <iostream>
#include <string>

struct Book
{
        std::string title;
        int year;
        float price;
};
int main()
{
        Book b;

        b.title = "Literature";
        b.year = 2023;
        b.price = 10.85;
        std::cout << b.title << ' '  << b.year << ' ' << b.price <<'\n';
        return 0;
}
```

- Εκτός από τον τελεστή . θα δούμε στη συνέχεια ότι μπορούμε να χρησιμοποιήσουμε και τον τελεστή -> για να προσπελάσουμε τα πεδία μίας μεταβλητής δομής

Αρχικοποίηση Πεδίων Δομής (2)

- Ένας εναλλακτικός τρόπος απόδοσης αρχικών τιμών στα πεδία μίας μεταβλητής δομής είναι μαζί με τη δήλωσή της. Σε αυτή την περίπτωση, τη δήλωση της δομής την ακολουθεί μία λίστα τιμών που περικλείεται σε άγκιστρα και πρέπει να ταιριάζει με τη σειρά δήλωσης των πεδίων, καθώς και με τους τύπους τους. Οι τιμές της λίστας χωρίζονται με κόμμα και δεν πρέπει να είναι περισσότερες από τα πεδία. Π.χ. με τη δήλωση:

  ```
  Book b = {"Literature", 2023, 10.85};
  ```

  η τιμή του `b.title` γίνεται `Literature` το `b.year` ίσο με 2023 και το `b.price` ίσο με `10.85`

- Όπως τα στοιχεία ενός πίνακα, έτσι και τα πεδία μίας δομής που δεν αρχικοποιούνται αποκτούν μηδενικές τιμές. Π.χ. με τη δήλωση:

  ```
  Book b = {"Literature"};
  ```

  η τιμή του `b.title` γίνεται `Literature` και τα `b.year` και `b.price` ίσα με `0`

- Αν οι {} είναι κενές, τα πεδία αρχικοποιούνται με τις προεπιλεγμένες τιμές των τύπων τους. Π.χ. με τη δήλωση:

  ```
  Book b = {};
  ```

  η τιμή του `b.title` γίνεται `""` και τα `b.year` και `b.price` ίσα με `0`

Αρχικοποίηση Πεδίων Δομής (3)

- Επίσης, μία μεταβλητή που δηλώνεται στη `λίστα_μεταβλητών` μπορεί ταυτόχρονα να αρχικοποιηθεί. Π.χ.

```
struct Book
{
        string title;
        int year;
        float price;
} b = {"Literature", 2023, 10.8};
```

- Με την C++11, μπορούμε να παραλείψουμε το = για τον καθορισμό της αρχικής τιμής. Π.χ.

```
Book b1{"Literature", 2023, 10.85};
Book b2{}; /* Τα μέλη της δομής αρχικοποιούνται με τις προεπιλεγμένες
αρχικές τιμές σύμφωνα με τον τύπο τους. Δηλαδή, τα αριθμητικά πεδία με
0 και το title με "". */
```

Δείκτης σε Πεδίο Δομής

- Όπως χρησιμοποιούμε έναν δείκτη σε μία μεταβλητή, μπορούμε να τον χρησιμοποιήσουμε και σε ένα πεδίο μίας δομής. Π.χ. ο επόμενος κώδικας χρησιμοποιεί δείκτες για να εμφανίσει τα πεδία της δομής **b**:

```
int main()
{
        string *p1;
        int *p2;
        float *p3;
        Book b = {"Literature", 2023, 10.8};

        p1 = &b.title;
        p2 = &b.year;
        p3 = &b.price;

        cout << *p1 << ' ' << *p2 << ' ' << *p3 << '\n';
        return 0;
}
```

- Για να δείξει κάποιος δείκτης στο πεδίο μίας δομής πρέπει ο τύπος του να είναι συμβατός με τον τύπο του αντίστοιχου πεδίου. Π.χ. αφού ο τύπος του πεδίου **year** είναι **int**, ο δείκτης **p2** δηλώνεται σαν **int***

Λειτουργίες μεταξύ Δομών

- Αν και δεν μπορούμε να χρησιμοποιήσουμε τον τελεστή = για να αντιγράψουμε έναν πίνακα σε έναν άλλο, μπορούμε να τον χρησιμοποιήσουμε για να αντιγράψουμε μία δομή σε μία άλλη
- Βέβαια, οι δομές πρέπει να είναι ίδιου τύπου. Π.χ.

```
struct Student
{
        int code;
        float grd;
};

int main()
{
        Student s1, s2;
        s1.code = 1234;
        s1.grd = 6.7;
        s2 = s1; // Αντιγραφή δομής.
        ...
}
```

Παρατηρήσεις (1)

- Με την εντολή:

```
s2 = s1;
```

οι τιμές των πεδίων της δομής `s1` αντιγράφονται στα αντίστοιχα πεδία της δομής `s2`

Δηλαδή, η παραπάνω εντολή είναι ισοδύναμη με:

```
s2.code = s1.code;
s2.grd = s1.grd;
```

- Αν οι `s1` και `s2` δεν ήταν μεταβλητές του ίδιου τύπου δομής, η εντολή `s2 = s1;` δεν θα μεταγλωττιζόταν, ακόμα κι αν τα δύο διαφορετικά πρότυπα δομής περιείχαν τα ίδια ακριβώς πεδία και σε αριθμό και σε τύπο δεδομένων

Παρατηρήσεις (2)

- Σημειώστε, επίσης, ότι αν μία δομή περιέχει πίνακα (π.χ. `name`), αν και, όπως γνωρίζετε, δεν μπορούμε να γράψουμε

```
s2.name = s1.name;
```

με την αντιγραφή της δομής (π.χ. `s2 = s1`) αντιγράφεται και ο πίνακας της `s1` στην `s2`

- Επίσης, αν η δομή περιέχει πεδίο δείκτη, μετά την αντιγραφή και οι δύο δείκτες θα δείχνουν στο ίδιο σημείο, γεγονός που μπορεί να αποβεί πολύ επικίνδυνο

Παρατηρήσεις (3)

- Εκτός από την ανάθεση καμία άλλη ενέργεια δεν επιτρέπεται μεταξύ δομών

- Π.χ. οι τελεστές == και != δεν μπορούν να χρησιμοποιηθούν για τον έλεγχο της ισότητας δύο δομών

  Δηλαδή, δεν επιτρέπεται να γράψετε:

```
if(s1 == s2)
```

ή

```
if(s1 != s2)
```

- Αν πρέπει να γίνει έλεγχος αν δύο δομές περιέχουν τα ίδια ακριβώς δεδομένα, πρέπει αναγκαστικά να λάβει χώρα σύγκριση όλων των πεδίων των δομών ένα προς ένα

  Δηλαδή:

```
if((s1.code == s2.code) && (s1.grd == s2.grd))
```

Παράδειγμα Δομής που Περιέχει Πίνακες

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>
#include <cstring>

struct Student
{
        char name[50];
        float grd[2];
};

int main()
{
        Student s1, s2;
        strcpy(s1.name, "somebody");
        s1.grd[0] = 8.5;
        s1.grd[1] = 7.5;
        std::cout << s1.name << ' ' << s1.name[0] << ' ' <<
*s1.name << '\n';
        s2 = s1;
        std::cout << s2.name << ' ' << s2.grd[0] << ' ' <<
s2.grd[1] << '\n';
        return 0;
}
```

Παράδειγμα Δομής που Περιέχει Πίνακες

Έξοδος:
```
somebody s s
somebody 8.5 7.5
```

Παράδειγμα Δομής που Περιέχει Δείκτες

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>
#include <cstring>

struct Student
{
        const char *name;
        float *avg_grd;
};

int main()
{
        float grd = 8.5;
        Student s1, s2;

        s1.name = "somebody";
        s1.avg_grd = &grd;
        std::cout << s1.name+3 << ' ' << *s1.avg_grd << '\n';

        s2.name = "else";
        s2 = s1;
        grd = 3.4;
        std::cout << s2.name << ' ' << *s2.avg_grd << '\n';
        return 0;
}
```

Παράδειγμα Δομής που Περιέχει Δείκτες

- Με την εντολή `s1.name = "somebody";` ο μεταγλωττιστής αρχικά δεσμεύει μνήμη για να αποθηκεύσει το αλφαριθμητικό `"somebody"` και στη συνέχεια ο δείκτης `name` δείχνει στην αρχή αυτής της μνήμης. Με την εντολή `s1.avg_grd = &grd;` ο δείκτης `avg_grd` δείχνει στη διεύθυνση της `grd`. Για να προσπελάσουμε το περιεχόμενο της μνήμης που δείχνει ένα πεδίο δείκτης, ο τελεστής `*` πρέπει να προηγείται του ονόματος της δομής. Επομένως, το πρόγραμμα εμφανίζει: `ebody 8.5`

- Με την `s2 = s1;` οι δείκτες της `s2` δείχνουν εκεί που δείχνουν οι δείκτες της `s1`. Επομένως, το πρόγραμμα εμφανίζει: `somebody 3.4` (προσοχή, όχι `else`). Ο λόγος που έβαλα αυτή την εντολή είναι για να σας επιστήσω την προσοχή όταν γίνεται αντιγραφή δομής που περιέχει δείκτη. Η τρέχουσα τιμή του δείκτη της δομής στην οποία γίνεται η εκχώρηση (π.χ. `s2`) θα χαθεί. Προσοχή, αυτό μπορεί να αποτελέσει σοβαρό πρόβλημα

- Προσοχή στην αντιγραφή δομής όταν περιέχει δείκτη

Ένθετες Δομές

- Μία δομή μπορεί να περιέχει μία ή περισσότερες δομές, οι οποίες ονομάζονται **ένθετες δομές**

- Μία ένθετη δομή πρέπει να **ορίζεται πριν από τον ορισμό της δομής στην οποία περιέχεται**, αλλιώς ο μεταγλωττιστής θα εμφανίσει μήνυμα λάθους

- Για να προσπελάσουμε τα πεδία μίας δομής που περιέχεται μέσα σε μία άλλη δομή **χρησιμοποιούμε δύο φορές** τον τελεστή τελεία (.)

- **Την πρώτη φορά** ανάμεσα στο όνομα της εξωτερικής και της ένθετης δομής και **τη δεύτερη φορά** ανάμεσα στο όνομα της ένθετης δομής και το όνομα του πεδίου που μας ενδιαφέρει (το οποίο ανήκει στην ένθετη δομή)

Παράδειγμα Δομής που Περιέχει Δομή

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>
#include <string>

struct Date
{
            int day;
            int month;
            int year;
};
struct Product /* Αφού ο τύπος Date έχει οριστεί, μπορεί να χρησιμοποιηθεί για να δηλώσουμε ένθετες δομές. */
{
            std::string name;
            double price;
            Date s_date;
            Date e_date;
};
int main()
{
            Product prod;

            prod.name = "product";
            prod.s_date.day = 1;
            prod.s_date.month = 9;
            prod.s_date.year = 2023;

            prod.e_date.day = 1;
            prod.e_date.month = 9;
            prod.e_date.year = 2025;

            prod.price = 7.5;
            std::cout << "The product's life is " << prod.e_date.year - prod.s_date.year << " years\n";
            return 0;
}
```

Παράδειγμα Δομής που Περιέχει Δομή

Έξοδος: `The product's life is 2 years`

Πεδία Δομής με Μέγεθος bit (1)

- Ένα πεδίο δομής μπορεί να περιέχει πεδία, των οποίων το μέγεθος να δηλώνεται σαν ένας συγκεκριμένος αριθμός από bits
- Ένα τέτοιο πεδίο ονομάζεται **πεδίο bit** και δηλώνεται με τον ακόλουθο τρόπο:

```
τύπος_δεδομένων όνομα_πεδίου_bit : αριθμός_bits;
```

- Τα bit πεδία μπορούν να χρησιμοποιηθούν όπως και τα απλά πεδία μίας δομής
- Ο τύπος ενός bit πεδίου πρέπει να είναι ακέραιος
- Επειδή η μνήμη για ένα bit πεδίο δεν δεσμεύεται όπως με τις συνηθισμένες μεταβλητές, δεν επιτρέπεται να εφαρμοστεί ο τελεστής διεύθυνσης & σε αυτό

Πεδία Δομής με Μέγεθος bit (2)

Παράδειγμα:

```
struct Person
{
        unsigned int sex : 1;
        unsigned int married : 1;
        unsigned int children : 4;
};
```

Μέγεθος bit: 1
Εύρος τιμών: 0 – 1

Μέγεθος bit: 4
Εύρος τιμών: 0 – 15

Πεδία Δομής με Μέγεθος bit (3)

- Για την αποθήκευση των τιμών του προηγούμενου παραδείγματος απαιτούνται συνολικά: `1 + 1 + 4 = 6` bits
- Αφού ο τύπος δεδομένων των πεδίων είναι `unsigned int`, τότε ο μεταγλωττιστής **δεσμεύει** `4` byte μνήμης, και `4×8-6 = 26` bits **μένουν αχρησιμοποίητα**
- Το **κύριο πλεονέκτημα** της χρήσης των πεδίων bit είναι **η εξοικονόμηση μνήμης**
- Π.χ., στο προηγούμενο πρόγραμμα ο μεταγλωττιστής δεσμεύει τέσσερις οκτάδες αντί για δώδεκα που θα δέσμευε αν δεν χρησιμοποιούσαμε bit πεδία
- Επομένως, αφού για τα στοιχεία ενός ανθρώπου εξοικονομούμε οκτώ οκτάδες, αν έπρεπε να αποθηκεύσουμε τα στοιχεία `200.000` ανθρώπων θα εξοικονομούσαμε `1.600.000` οκτάδες

Δείκτης σε Δομή

- Ένας δείκτης σε μία δομή χρησιμοποιείται με τον ίδιο τρόπο όπως κάθε άλλος δείκτης. Π.χ.

```
#include <iostream>
#include <string>

struct Product
{
        std::string name;
        float grd;
};
int main()
{
        Student *ptr, stud;

        ptr = &stud;
        (*ptr).name = "somebody";
        (*ptr).grd = 6.7;
        std::cout << stud.name << ' ' << stud.grd << '\n';
        return 0;
}
```

- Το πρόγραμμα εμφανίζει somebody 6.7. Η μεταβλητή ptr δηλώνεται σαν δείκτης σε δομή τύπου Student. Με την εντολή ptr = &stud; ο ptr δείχνει στη διεύθυνση μνήμης της δομής stud. Συγκεκριμένα, δείχνει στη διεύθυνση μνήμης του πρώτου πεδίου της δομής. Αφού ο ptr δείχνει στη διεύθυνση μνήμης της δομής stud, το *ptr είναι ισοδύναμο με τη δομή stud και με τον τελεστή . μπορούμε να αποκτήσουμε πρόσβαση στα πεδία της δομής. Η έκφραση *ptr πρέπει να περικλείεται σε παρενθέσεις, γιατί ο τελεστής . έχει μεγαλύτερη προτεραιότητα από τον τελεστή *

Ο Τελεστής ->

- Ένας εναλλακτικός τρόπος για να αποκτήσουμε πρόσβαση στα πεδία μίας δομής με χρήση δείκτη είναι χρησιμοποιώντας τον τελεστή -> αντί του τελεστή τελεία (.). Για παράδειγμα, δείτε πώς μπορούμε να αλλάξουμε το προηγούμενο πρόγραμμα:

```
int main() // Χρήση δείκτη για την προσπέλαση των πεδίων.
{
        Student *ptr, stud;

        ptr = &stud;
        ptr->name = "somebody";
        ptr->grd = 6.7;
        std::cout << ptr->name << ' ' << ptr->grd << '\n';
        return 0;
}
```

- Η έκφραση `ptr->name` είναι ισοδύναμη με `(*ptr).name` και η έκφραση `ptr->grd` ισοδύναμη με `(*ptr).grd`
- Μεταξύ των δύο τρόπων χρήσης του δείκτη για πρόσβαση στα πεδία μίας δομής, θεωρώ ότι είναι πιο εκφραστικό και απλό να χρησιμοποιούμε τον τελεστή ->

Πίνακας Δομών

- Ένας πίνακας δομών είναι ένας πίνακας με στοιχεία δομές

- Συνήθως, ένας πίνακας δομών χρησιμοποιείται σε εφαρμογές που απαιτείται αποθήκευση πληροφορίας για πολλές οντότητες, όπως μία εφαρμογή καταχώρησης των εμπορευμάτων μίας αποθήκης, των φοιτητών μίας σχολής ή των υπαλλήλων μίας εταιρείας

- Ουσιαστικά, ένας πίνακας δομών μπορεί να χρησιμοποιηθεί σαν μία απλή βάση δεδομένων. Π.χ. με την εντολή:

```
Student stud[100];
```

η μεταβλητή `stud` δηλώνεται σαν ένας πίνακας 100 στοιχείων, όπου κάθε στοιχείο του είναι μία δομή τύπου `Student`

Αρχικοποίηση Πίνακα Δομών (1)

- Ένας πίνακας δομών μπορεί να αρχικοποιηθεί μαζί με τη δήλωσή του. Π.χ. για τη δομή:

```
struct Student
{
        string name;
        int code;
        float grd;
};
```

με την παρακάτω αρχικοποίηση:

```
struct Student stud[3] = {{"nick stergiou", 150, 7.3},
                          {"john nikas", 160, 5.8},
                           {"peter karras", 170, 6.7}};
```

η τιμή του πεδίου `stud[0].name` γίνεται `"nick stergiou"`

η τιμή του πεδίου `stud[1].code` γίνεται `160`

η τιμή του πεδίου `stud[2].grd` γίνεται `6.7`

- Όπως συνήθως, αν παραληφθεί η διάσταση στις αγκύλες `[]`, ο μεταγλωττιστής θα υπολογίσει τον αριθμό των στοιχείων

Αρχικοποίηση Πίνακα Δομών (2)

- Επίσης, μπορούμε να δηλώσουμε και να αρχικοποιήσουμε έναν πίνακα δομών μαζί με τη δήλωση της δομής. Π.χ:

```
struct Student
{
   string name;
   int code;
   float grd;
} stud[] = {{"nick stergiou", 150, 7.3},
            {"john nikas", 160, 5.8},
                  {"peter karras", 170, 6.7}};
```

- Και με τους 2 τρόπους αρχικοποίησης, τα εσωτερικά άγκιστρα δεν είναι απαραίτητα, ωστόσο αν εισάγονται ξεχωρίζει η αρχικοποίηση της κάθε δομής

Παρατηρήσεις

- Όπως και με έναν απλό πίνακα, μπορούμε να χρησιμοποιήσουμε σημειογραφία δείκτη για να προσπελάσουμε τα στοιχεία του

  Π.χ. για την προηγούμενη δομή `stud` (τύπου `Student`) αφού το όνομα ενός πίνακα είναι δείκτης στη διεύθυνση του 1ου στοιχείου του, τότε:
  - το `*stud` είναι ισοδύναμο με `stud[0]`
  - το `*(stud+1)` είναι ισοδύναμο με `stud[1]`
  - το `*(stud+2)` είναι ισοδύναμο με `stud[2]` κ.ο.κ.

- Άρα, αν θέλουμε να προσπελάσουμε το πεδίο `grd` του τρίτου φοιτητή, οι εκφράσεις `stud[2].grd` και `(*(stud + 2)).grd` είναι ισοδύναμες (οι παρενθέσεις στη δεύτερη περίπτωση είναι απαραίτητες λόγω των προτεραιοτήτων)

- Όπως και στην περίπτωση των απλών πινάκων, είναι πιο βολικό να χρησιμοποιούμε τη θέση του στοιχείου και όχι σημειογραφία δείκτη, ώστε ο κώδικας να είναι πιο απλός και ευανάγνωστος

Παράδειγμα

- Το παρακάτω πρόγραμμα χρησιμοποιεί έναν επαναληπτικό βρόχο, για να αποθηκεύσει τα στοιχεία 100 φοιτητών σε έναν πίνακα δομών τύπου **Student**

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

const int SIZE = 100;

struct Student
{
   string name;
   int code;
   float grd;
};
int main()
{
          int i;
          Student stud[SIZE];
          for(i = 0; i < SIZE; i++)
          {
                    cout << "\nEnter name: ";
                    getline(cin, stud[i].name);

                    cout << "Enter code: ";
                    cin >> stud[i].code;

                    cout << "Enter grade: ";
                    cin >> stud[i].grd;

                  cout << stud[i].name << ' ' << stud[i].code << ' ' << stud[i].grd << '\n';
                    cin.get(); /* Διαβάζουμε τον χαρακτήρα αλλαγής γραμμής που αποθηκεύτηκε
στο cin μετά την εισαγωγή του βαθμού. */
          }
          return 0;
}
```

Παράδειγμα

- Και όπως ξέρουμε αντί για έναν απλό πίνακα μπορούμε να χρησιμοποιήσουμε ένα **vector** αντικείμενο. Π.χ.

```
int main()
{
        vector<Student> stud(SIZE);
        ... // Ο υπόλοιπος κώδικας είναι ίδιος.
}
```

Συνάρτηση με Παράμετρο Δομή

- Μία δομή μπορεί να μεταβιβαστεί σε μία συνάρτηση όπως οποιαδήποτε άλλη μεταβλητή, δηλαδή **είτε η ίδια η δομή είτε η διεύθυνση μνήμης της δομής**, ενώ επίσης, μία συνάρτηση μπορεί να επιστρέφει δομή
- Υπενθυμίζεται ότι, όταν μεταβιβάζεται **η τιμή** μιας παραμέτρου σε συνάρτηση, τότε στη συνάρτηση διοχετεύονται **αντίγραφα** των παραμέτρων του προγράμματος που την καλεί, επομένως, οποιαδήποτε αλλαγή γίνει στις τιμές των πεδίων της δομής μέσα στη συνάρτηση **δεν επηρεάζει** τις αντίστοιχες τιμές των πεδίων της δομής που διοχετεύθηκε στη συνάρτηση, γιατί οι τυχόν αλλαγές γίνονται σε αντίγραφό της
- **Αντίθετα**, σε περίπτωση που μεταβιβάζεται **η διεύθυνση** της παραμέτρου σε συνάρτηση, στη συνάρτηση διοχετεύονται **οι διευθύνσεις μνήμης** των παραμέτρων του προγράμματος που την καλεί και όχι αντίγραφά τους, όπως προηγουμένως, επομένως, αφού η συνάρτηση έχει πρόσβαση στη διεύθυνση της δομής του προγράμματος που την κάλεσε, τότε **μπορεί να μεταβάλλει** τις τιμές των πεδίων της

Παράδειγμα Μεταβίβασης Δομής

```
#include <iostream>
#include <string>

struct Student
{
   std::string name;
   int code;
   float grd;
};

Student test(Student s);

int main()
{
          Student st = {"somebody", 20, 5};
          st = test(st); /* Σύμφωνα με τη δήλωση της συνάρτησης, πρέπει να κληθεί με όρισμα
μία δομή τύπου Student. */
          std::cout << st.name << ' ' << st.code << ' ' << st.grd << '\n';
          return 0;
}

Student test(Student s)
{
          s.name = "new";
          s.code = 30;
          s.grd = 7;
          return s;
}
```

- Όταν καλείται η `test()` οι τιμές των πεδίων της `st` αντιγράφονται στα αντίστοιχα πεδία της `s`. Αφού οι μεταβλητές `s` και `st` αποθηκεύονται σε διαφορετικές θέσεις μνήμης, οποιαδήποτε αλλαγή γίνει στα πεδία της `s` δεν επηρεάζει τις αντίστοιχες τιμές των πεδίων της `st`. Μετά, η δομή που επιστρέφει η `test()` αντιγράφεται στην `st`. Έτσι, το πρόγραμμα εμφανίζει:
  `new 30 7`

Παράδειγμα Μεταβίβασης Διεύθυνσης Δομής

- Αντίθετα, αν μεταβιβάσουμε τη διεύθυνση μνήμης της δομής η συνάρτηση μπορεί να αλλάξει τις τιμές των πεδίων της. Για παράδειγμα, ας τροποποιήσουμε την `test()`, ώστε να μπορεί να αλλάξει τις τιμές των πεδίων της δομής `st`

```
void test(Student *ptr);

int main()
{
        Student st = {"somebody", 20, 5};
        test(&st);
        std::cout << st.name << ' ' << st.code << ' ' << st.grd << '\n';
        return 0;
}

void test(Student *ptr) /* Τώρα, μπορούμε να αλλάξουμε τα πεδία της
δομής. */
{
        ptr->name = "new";
        ptr->code = 30;
        ptr->grd = 7;
}
```

- Όταν καλείται η `test()` έχουμε `ptr = &st`. Άρα, αφού ο `ptr` δείχνει στη διεύθυνση μνήμης της `st`, η συνάρτηση μπορεί να αλλάξει τις τιμές των πεδίων της. Επομένως, ο κώδικας εμφανίζει: `new 30 7`

Παρατηρήσεις

- Όταν μία δομή μεταβιβάζεται σε μία συνάρτηση γίνεται αντιγραφή των πεδίων της στα πεδία της αντίστοιχης παραμέτρου της συνάρτησης. Σημειώστε ότι αν το μέγεθος της δομής είναι μεγάλο ή αν η συνάρτηση καλείται αρκετές φορές, αυτή η διαδικασία αντιγραφής μπορεί να προσθέσει κάποια χρονική καθυστέρηση στην εκτέλεση του προγράμματος

- Αντίθετα, όταν μεταβιβάζεται η διεύθυνση της δομής, μέσω δείκτη ή αναφοράς, δεν γίνεται αντιγραφή πεδίων

- Για καλύτερη απόδοση, να μεταβιβάζετε σε μία συνάρτηση τη διεύθυνση της δομής, ακόμα και αν η συνάρτηση δεν πρόκειται να αλλάξει τις τιμές των πεδίων της

- Αν δεν θέλετε η συνάρτηση να μπορεί να αλλάξει τις τιμές των πεδίων της δομής, να δηλώνετε τον δείκτη ή την αναφορά ως `const`. Π.χ. `void test(const Student *ptr);`

Ενώσεις (Unions)

- Όπως και η δομή, μία **ένωση** αποτελείται από ένα ή περισσότερα πεδία που μπορεί να έχουν διαφορετικούς τύπους
- Οι ιδιότητες της ένωσης είναι παρόμοιες με αυτές των δομών με τις ίδιες λειτουργίες να επιτρέπονται όπως και στις δομές
- Η διαφορά τους είναι ότι τα πεδία μίας δομής αποθηκεύονται σε διαφορετικές διευθύνσεις μνήμης, ενώ τα πεδία μίας ένωσης αποθηκεύονται στην ίδια διεύθυνση μνήμης
- Αφού ο μεταγλωττιστής δεν δεσμεύει ξεχωριστή μνήμη για κάθε πεδίο, ο κύριος λόγος χρήσης μίας ένωσης είναι η εξοικονόμηση μνήμης, όπως για παράδειγμα σε εφαρμογές ενσωματωμένων συστημάτων, όπου η μνήμη είναι περιορισμένη

Δήλωση Ένωσης

- Μία ένωση δηλώνεται όπως και μία δομή, με τη διαφορά ότι, αντί της λέξης `struct` χρησιμοποιείται η λέξη `union`

- Όταν δηλώνεται μία ένωση, ο μεταγλωττιστής δεσμεύει μνήμη για την αποθήκευση του **μεγαλύτερου** πεδίου της, και όχι για όλα τα πεδία της ένωσης

- Επομένως, τα πεδία μίας ένωσης αποθηκεύονται σε έναν **κοινό** χώρο μνήμης και όχι σε ξεχωριστή μνήμη, όπως στην περίπτωση μίας δομής

Παράδειγμα

- Στο επόμενο πρόγραμμα η μνήμη που δεσμεύεται για τη μεταβλητή s είναι οκτώ οκτάδες, αφού το μεγαλύτερο πεδίο της είναι τύπου double

```
#include <iostream>

union Sample
{
  char ch;
  int i;
  double d;
};

int main()
{
        Sample s;
        std::cout << sizeof(s) << '\n';
        return 0;
}
```

Πρόσβαση Πεδίων Ένωσης

- Τα πεδία μίας ένωσης μπορούμε να τα προσπελάσουμε με τους ίδιους τρόπους που προσπελαύνουμε τα πεδία μίας δομής

- Ωστόσο, επειδή όλα τα πεδία αποθηκεύονται στην **ίδια** μνήμη, μόνο το **τελευταίο** πεδίο στο οποίο εκχωρήθηκε μία τιμή έχει έγκυρη τιμή

- Όταν γίνεται ανάθεση τιμής σε ένα πεδίο μίας ένωσης, η τιμή που τελευταία είχε αποθηκευτεί σε κάποιο άλλο πεδίο υπερεγγράφεται

Παράδειγμα

- Το επόμενο πρόγραμμα θέτει μία τιμή σε κάποιο πεδίο της ένωσης **s** και εμφανίζει τις τιμές των υπολοίπων πεδίων

```
#include <iostream>

union Sample
{
   char ch;
   int i;
   double d;
};

int main()
{
          Sample s;

          s.ch = 'a';
          std::cout << s.ch << ' ' << s.i << ' ' << s.d << '\n';

          s.i = 64;
          std::cout << s.ch << ' ' << s.i << ' ' << s.d << '\n';

          s.d = 12.48;
          std::cout << s.ch << ' ' << s.i << ' ' << s.d << '\n';
          return 0;
}
```

- Αρχικά, το πρόγραμμα εμφανίζει **a** και χωρίς νόημα τιμές για τα **s.i** και **s.d**. Μετά, εκχωρείται η τιμή **64** στο πεδίο **i**. Αφού αυτή η τιμή αποθηκεύεται στον κοινό χώρο μνήμης, η τιμή του **s.ch** υπερεγγράφεται. Άρα, εμφανίζεται **64** και χωρίς νόημα τιμές για τα **s.ch** και **s.d**. Τέλος, εκχωρείται η τιμή **12.48** στο πεδίο **d**. Η τιμή του **s.i** υπερεγγράφεται και το πρόγραμμα εμφανίζει **12.48** και χωρίς νόημα τιμές για τα **s.ch** και **s.i**