Προγραμματισμός Υπολογιστών C++

Συναρτήσεις

Συναρτήσεις

- Μία **συνάρτηση** είναι ένα ανεξάρτητο τμήμα κώδικα που έχει όνομα
- Όταν κληθεί εκτελείται ο κώδικάς της και όταν τελειώσει μπορεί **προαιρετικά** να επιστρέψει μία τιμή
- Η συγγραφή προγραμμάτων με χρήση συναρτήσεων που εκτελούν ανεξάρτητες εργασίες αποτελεί τη βάση του λεγόμενου **δομημένου προγραμματισμού**
- Με τη χρήση συναρτήσεων ένα πρόγραμμα χωρίζεται σε μικρότερα τμήματα, άρα ο κώδικας αναπτύσσεται, διαβάζεται, τροποποιείται και ελέγχεται πιο εύκολα
- Επίσης, επειδή μία συνάρτηση μπορεί να κληθεί όσες φορές θέλουμε μέσα σε ένα πρόγραμμα, αποφεύγουμε τη συγγραφή του ίδιου κώδικα και έτσι μειώνεται το μέγεθος του προγράμματος
- Γενικά, η χρήση συναρτήσεων (δεν αναφέρομαι σε μικρά και απλά προγράμματα) είναι απαραίτητη για την καλύτερη οργάνωση και διαχείριση του προγράμματος
- Μέχρι τώρα, η μοναδική συνάρτηση που έχουμε γράψει είναι η συνάρτηση `main()`, ενώ τώρα θα μάθετε να γράφετε δικές σας συναρτήσεις και να τις χρησιμοποιείτε στα προγράμματά σας

Δήλωση Συνάρτησης

- Η **δήλωση** (ή αλλιώς **πρωτότυπο**) μίας συνάρτησης καθορίζει το **όνομα** της συνάρτησης, τον **τύπο επιστροφής** της και **μία λίστα παραμέτρων**

- Η γενική περίπτωση δήλωσης μίας συνάρτησης έχει την παρακάτω μορφή:

```
τύπος_επιστροφής όνομα_συνάρτησης(λίστα_παραμέτρων);
```

- Το `όνομα_συνάρτησης` πρέπει να είναι μοναδικό μέσα στο πρόγραμμα, δηλαδή να μην υπάρχει άλλη μεταβλητή ή συνάρτηση με το ίδιο όνομα

- Η δήλωση της συνάρτησης πρέπει να τελειώνει πάντοτε με το ελληνικό ερωτηματικό ;

- Το πρωτότυπο της συνάρτησης ενημερώνει τον μεταγλωττιστή για τον τύπο των παραμέτρων, τον αριθμό τους, τον τύπο επιστροφής, ώστε να μπορεί να ελέγξει αν ο τρόπος που καλείται στο πρόγραμμα είναι σύμφωνα με αυτό

- Προσπαθήστε να επιλέγετε περιγραφικά ονόματα για τις συναρτήσεις σας, ώστε αυτός που διαβάζει τον κώδικα της συνάρτησης να αντιλαμβάνεται γρήγορα τον σκοπό της

Πού γράφουμε τη Δήλωση (Πρωτότυπο) μίας Συνάρτησης;

- Συνήθως, οι δηλώσεις των συναρτήσεων περιέχονται σε διαφορετικό αρχείο από τον κώδικα τους. Για παράδειγμα, οι δηλώσεις των συναρτήσεων βιβλιοθήκης περιέχονται σε διάφορα αρχεία επικεφαλίδας

- Όταν θέλουμε να χρησιμοποιήσουμε κάποια συνάρτηση που η δήλωσή της περιέχεται σε ξεχωριστό αρχείο, χρησιμοποιούμε την οδηγία `#include` για να συμπεριλάβουμε το αρχείο που περιέχει τη δήλωσή της. Π.χ. για να χρησιμοποιήσουμε την `sqrt()` συμπεριλαμβάνουμε το αρχείο `cmath`

- Όπως θα δούμε στη συνέχεια, αν η συνάρτηση οριστεί πριν από το σημείο κλήσης της μπορούμε να παραλείψουμε το πρωτότυπό της. Σε αυτή την περίπτωση, ο ορισμός παρέχει στον μεταγλωττιστή την πληροφορία που χρειάζεται για να χειριστεί την κλήση της συνάρτησης

- Γενικά, αυτό που συνηθίζεται είναι οι δηλώσεις συναρτήσεων να τοποθετούνται σε ένα αρχείο επικεφαλίδας, το οποίο να συμπεριλαμβάνεται με την οδηγία `#include` σε όποιο αρχείο κώδικα την χρειάζεται

- Για απλότητα, θα δηλώνουμε τις συναρτήσεις που γράφουμε στο ίδιο αρχείο με τη `main()`

Επιστροφή Συνάρτησης

- Μία συνάρτηση μπορεί να επιστρέψει το πολύ μία τιμή
- Ο τύπος_επιστροφής μίας συνάρτησης καθορίζει τον τύπο της τιμής επιστροφής
- Μία συνάρτηση μπορεί να επιστρέψει οποιοδήποτε τύπο δεδομένων, όπως char, int, double, ... ή δείκτη σε κάποιον τύπο
- Μοναδικός περιορισμός είναι ότι ο τύπος επιστροφής δεν επιτρέπεται να είναι πίνακας ή συνάρτηση. Επιτρέπεται, όμως, να επιστρέψει δείκτη σε πίνακα ή συνάρτηση
- Ο τύπος επιστροφής void δηλώνει ότι η συνάρτηση δεν επιστρέφει κάποια τιμή
- Με την C++11 η συνάρτηση μπορεί να επιστρέψει μία λίστα τιμών μέσα σε άγκιστρα. Η λίστα χρησιμοποιείται για την αρχικοποίηση της προσωρινής μεταβλητής που χρησιμοποιείται για τιμή επιστροφής. Π.χ.

```
vector<int> f()
{
        return {4, 5, 6};
}
int main()
{
       vector<int> v = f(); /* Το v αρχικοποιείται με τις τιμές 4,
5, και 6. */
}
```

Παράμετροι Συνάρτησης

- Μία συνάρτηση μπορεί **προαιρετικά** να δέχεται μία λίστα παραμέτρων (`λίστα_παραμέτρων`), που χωρίζονται μεταξύ τους με κόμμα (`,`)

- Κάθε παράμετρος χαρακτηρίζεται από τον τύπο της

- Η παράμετρος μίας συνάρτησης είναι ουσιαστικά μία μεταβλητή της συνάρτησης, η οποία θα αρχικοποιηθεί με μία τιμή, όταν θα γίνει η κλήση της

- Αν η συνάρτηση **δεν δέχεται παραμέτρους** χρησιμοποιούμε κενές παρενθέσεις `()`

Παραδείγματα Δήλωσης Συναρτήσεων

```
void show(); /* Δήλωση μίας συνάρτησης με όνομα show, η
οποία δεν δέχεται παραμέτρους και δεν επιστρέφει κάτι. */

double show(char ch, int a, float b); /* Δήλωση μίας
συνάρτησης με όνομα show, η οποία δέχεται ένα χαρακτήρα,
μία ακέραια και μία πραγματική παράμετρο τύπου float και
επιστρέφει μία πραγματική τιμή τύπου double. */

double *show(int *p1, int *p2); /* Δήλωση μίας συνάρτησης
με όνομα show, η οποία δέχεται σαν παραμέτρους δύο δείκτες
σε ακεραίους και επιστρέφει έναν δείκτη σε τύπο double.*/
```

Παρατηρήσεις

- Σημειώστε ότι τα ονόματα των παραμέτρων δεν είναι υποχρεωτικά, αρκεί ο τύπος τους, π.χ., θα μπορούσαμε να γράψουμε:

```
double show(char, int, float);
```

- Η συνήθης προτίμηση είναι να προστίθενται τα ονόματα, ώστε αυτός που απλά διαβάζει το πρωτότυπο της συνάρτησης ή σκοπεύει να χρησιμοποιήσει τη συνάρτηση να παίρνει μία ιδέα για την πληροφορία που της διοχετεύεται, με ποια σειρά και τον σκοπό της κάθε παραμέτρου

- Ο τύπος της κάθε παραμέτρου πρέπει να καθορίζεται ακόμα και αν όλοι οι παράμετροι έχουν τον ίδιο τύπο. Π.χ. είναι **λάθος** να γράψουμε

```
void test(int a, b, c);
```

αντί για:

```
void test(int a, int b, int c);
```

Ορισμός Συνάρτησης (1)

- Ο **ορισμός** μίας συνάρτησης (function definition) περιέχει τον κώδικα της συνάρτησης. Η γενική μορφή του ορισμού μίας συνάρτησης είναι:

```
τύπος_επιστροφής όνομα_συνάρτησης(λίστα_παραμέτρων)
{
        /* Σώμα Συνάρτησης */
}
```

- Ο ορισμός μίας συνάρτησης δεν επιτρέπεται να μοιράζεται σε διαφορετικά αρχεία. Επίσης, κάθε συνάρτηση πρέπει να έχει ένα μοναδικό ορισμό

- Η πρώτη γραμμή πρέπει να ταιριάζει με τη δήλωση της συνάρτησης, αλλά δεν χρειάζεται να είναι ίδια. Για παράδειγμα, τα ονόματα των παραμέτρων δεν χρειάζεται να είναι τα ίδια με τα ονόματα στη δήλωση της συνάρτησης. Επίσης, σημειώστε ότι δεν προστίθεται το ερωτηματικό ; στο τέλος της

- Ο **κώδικας** ή, αλλιώς, το **σώμα** της συνάρτησης είναι ότι περιέχεται ανάμεσα στα άγκιστρα. Το σώμα της συνάρτησης μπορεί να είναι άδειο

- Ο κώδικας μίας συνάρτησης εκτελείται μόνο όταν αυτή κληθεί από κάποιο σημείο του προγράμματος

- Η εκτέλεση μίας συνάρτησης τερματίζει αν κληθεί μία εντολή τερματισμού (π.χ. `return`) ή όταν εκτελεστεί η τελευταία εντολή της

Παράδειγμα Χρήσης Συνάρτησης

```
#include <iostream>

void test(); /* Πρωτότυπο συνάρτησης.*/

int main(void)
{
   test(); /* Κλήση συνάρτησης. */
   return 0;
}

void test()/* Ορισμός συνάρτησης. */
{
   /* Σώμα συνάρτησης. */
   std::cout << "In\n";
}
```

Το πρόγραμμα εμφανίζει `In`

Δήλωση Συνάρτησης σε Ξεχωριστό Αρχείο (1)

- Για να <u>δηλώσετε μία συνάρτηση σε ξεχωριστό αρχείο</u>:
  - Δημιουργήστε ένα αρχείο, π.χ., `prototype.h`
  - Αντιγράψτε το πρωτότυπο (τη δήλωση) μέσα στο αρχείο αυτό
  - Αφαιρέστε τη δήλωση απ' το αρχείο του πηγαίου κώδικα όπου την είχατε έως τώρα
  - Χρησιμοποιήστε την οδηγία `#include` για να συμπεριλάβετε το αρχείο `prototype.h` όπως παρακάτω:

Έστω ότι το αρχείο έχει όνομα `program.cpp`

```
#include <iostream>
#include "prototype.h"

int main()
{
     ...
}
void test()
{
     ...
}
```

Το αρχείο `prototype.h` περιέχει μόνο τη δήλωση (πρωτότυπο) της `test()` δηλ.:

```
void test();
```

Παρατηρήσεις

- Αν ο ορισμός της συνάρτησης γίνει πριν από το πρώτο σημείο κλήσης της, η δήλωσή της μπορεί να παραλειφθεί. Π.χ.

```
#include <iostream>

void test()
{
   std::cout << "In\n";
}

int main()
{
   test();
   return 0;
}
```

- Αν και τα προγράμματά μας είναι μικρά και θα μπορούσαμε να ορίζουμε τις συναρτήσεις μας πριν από την `main()`, στη γενική περίπτωση μία τέτοια πρακτική δεν είναι πάντα εφαρμόσιμη
- Για παράδειγμα, σε μεγάλες εφαρμογές που ο κώδικας μοιράζεται σε πολλά αρχεία, η συνάρτηση που θέλουμε να καλέσουμε σε ένα αρχείο μπορεί να ορίζεται σε ένα διαφορετικό αρχείο. Τότε, η ύπαρξη πρωτοτύπου είναι υποχρεωτική, ώστε να γνωρίζει ο μεταγλωττιστής τον τρόπο που θα την καλέσει
- Επιπλέον, αν υπάρχουν πολλές συναρτήσεις η τοποθέτησή τους σε σωστή σειρά ώστε κάθε μία από αυτές να ορίζεται πριν κληθεί από κάποια άλλη απαιτεί χρόνο ή μπορεί να είναι και αδύνατη
- Επίσης, η ύπαρξη πρωτοτύπων επιτρέπει σε κάποιον που τα διαβάζει να αποκτήσει γρήγορα μια γενική εικόνα της λειτουργικότητας των συναρτήσεων
- Άρα, μαθαίνουμε να χρησιμοποιούμε πρωτότυπα

Σύνθετες Δηλώσεις (1)

- Αφού είδαμε πώς ορίζουμε συναρτήσεις, ας κάνουμε μία παρένθεση για να μάθουμε πώς αναλύουμε σύνθετες δηλώσεις
- Αναλύουμε κάθε δήλωση πάντα «από μέσα προς τα έξω»
- Βρίσκουμε το όνομα της μεταβλητής και, αν εσωκλείεται σε παρενθέσεις (), αρχίζουμε την ανάλυση από εκεί και συνεχίζουμε με τον επόμενο τελεστή, όπου η προτεραιότητα από την υψηλότερη στη χαμηλότερη είναι:
  - α) οι μεταθεματικοί τελεστές, δηλ. οι παρενθέσεις () που υποδεικνύουν συνάρτηση και οι αγκύλες [] που υποδεικνύουν πίνακα
  - β) ο προθεματικός τελεστής * υποδεικνύει «δείκτη σε...»

Σύνθετες Δηλώσεις (2)

- Π.χ., έστω η δήλωση: `int *p[5];`

  Το όνομα **p** δεν περικλείεται σε παρενθέσεις κι επειδή οι **[]** προηγούνται του * πάμε δεξιά και έχουμε ότι το **p** είναι **«πίνακας πέντε…»**, πάμε αριστερά στο * και η δήλωση γίνεται **«πίνακας πέντε δεικτών σε…»**, προσθέτουμε και τον τύπο και καταλήγουμε στη δήλωση: **«πίνακας πέντε δεικτών σε ακεραίους»**

- Άλλο ένα, έστω η δήλωση: `int (*p)[5];`

  Τώρα το **p** περικλείεται σε παρενθέσεις, κι επειδή υπάρχει μέσα στην παρένθεση το * έχουμε ότι το **p** είναι **«δείκτης σε…»**, πάμε δεξιά τώρα στις **[]** και η δήλωση γίνεται **«δείκτης σε πίνακα πέντε…»**, πάμε και αριστερά και καταλήγουμε στη δήλωση: **«δείκτης σε πίνακα πέντε ακεραίων»**

- Άντε κι άλλο ένα, έστω η δήλωση: `int *(*p)(int);`

  Ξεκινάμε όμοια με πριν από την παρένθεση ότι το **p** είναι **«δείκτης σε…»**, οι **()** προηγούνται του *, άρα πάμε δεξιά και έχουμε **«δείκτης σε συνάρτηση με όρισμα ακέραιο…»**, πάμε αριστερά στον τύπο επιστροφής και καταλήγουμε στη δήλωση **«δείκτης σε συνάρτηση με όρισμα ακέραιο που επιστρέφει δείκτη σε ακέραιο»**

Η Εντολή return (1)

- Η εντολή return χρησιμοποιείται για τον άμεσο τερματισμό μίας συνάρτησης, δηλ. αν η εκτέλεση του κώδικα της συνάρτησης φτάσει σε μία εντολή return, τότε η συνάρτηση τερματίζεται
- Άρα, αν εκτελεστεί η εντολή return μέσα στη συνάρτηση main(), τότε το πρόγραμμα τερματίζει. Π.χ.

```
#include <iostream>
int main()
{
   int num;
   while(1)
   {
         std::cout << "Enter number: ";
         std::cin >> num;

         if(num == 2)
                  return 0; // Τερματισμός προγράμματος.
         else /* Το else δε χρειάζεται, απλά για ευκολία στο παράδειγμα. */
                  std::cout << num << '\n';
   }
   return 0;
}
```

- Το παραπάνω πρόγραμμα τερματίζει, αν ο χρήστης εισάγει την τιμή 2, αλλιώς εμφανίζει την εισαγόμενη τιμή (παρατηρήστε ότι η τελευταία return δεν θα εκτελεστεί ποτέ, αφού μόνο η πρώτη return είναι αυτή που μπορεί να τερματίσει το πρόγραμμα, λόγω του ατέρμονου βρόχου)
- Η τιμή που επιστρέφει η main() δηλώνει τον τρόπο τερματισμού του προγράμματος. Η τιμή 0 δηλώνει τον ομαλό τερματισμό του προγράμματος, ενώ μία μη μηδενική τιμή δηλώνει συνήθως μία εσφαλμένη συνθήκη τερματισμού

Η Εντολή return (2)

- *Αν η συνάρτηση δεν έχει οριστεί να επιστρέφει κάποια τιμή (δηλ. αν ο επιστρεφόμενος τύπος της είναι* void*), τότε – για να τερματίσουμε άμεσα σε κάποιο σημείο τη συνάρτηση - γράφουμε απλά* return;
- Επίσης, σε αυτή την περίπτωση, η return στο τέλος της συνάρτησης (πριν το άγκιστρο «κλεισίματος» }) δεν είναι απαραίτητη, αφού η συνάρτηση θα επιστρέψει αυτόματα, δεδομένου ότι τελειώνει το «σώμα» της
- *Αν, όμως, η συνάρτηση έχει οριστεί να επιστρέφει κάποια τιμή, τότε η εντολή* return **πρέπει να ακολουθείται** *από κάποια τιμή. Αυτή η τιμή επιστρέφεται στο σημείο κλήσης της συνάρτησης. Η καλούσα συνάρτηση είναι ελεύθερη να χρησιμοποιήσει ή να αγνοήσει την επιστρεφόμενη τιμή. Π.χ.*

```
int f(int a, int b)
{
   if(a == b)
         return 0;
   cout << (a+b)/2.0;
   return 1;
}
```

- *Ο τύπος της τιμής που επιστρέφεται πρέπει να* **ταιριάζει** *με τον τύπο που ορίστηκε να επιστρέφει η συνάρτηση στη δήλωσή της. Αν δεν ταιριάζουν, ο μεταγλωττιστής θα μετατρέψει την επιστρεφόμενη τιμή, αν αυτό είναι δυνατό, στον τύπο επιστροφής της συνάρτησης. Π.χ:*

```
int test()
{
   return 4.9;
}
```

Αφού η **test()** *έχει δηλωθεί να επιστρέφει* int*, η τιμή επιστροφής θα μετατραπεί σε* int*. Επομένως, θα επιστραφεί η τιμή* **4**

Κλήση Συνάρτησης

- Όταν καλείται μία συνάρτηση, η εκτέλεση του προγράμματος συνεχίζει με την εκτέλεση του κώδικα της συνάρτησης
- Όταν τερματίζεται η συνάρτηση, η εκτέλεση του προγράμματος **επιστρέφει στο σημείο κλήσης** της συνάρτησης και συνεχίζει με την εκτέλεση της επόμενης εντολής
- Μία συνάρτηση μπορεί να κληθεί **όσες φορές είναι απαραίτητο** για τους σκοπούς του προγράμματος
- Όταν γίνεται η κλήση μίας συνάρτησης, ο μεταγλωττιστής **δεσμεύει μνήμη** για να αποθηκεύσει τις μεταβλητές που δηλώνονται στη λίστα παραμέτρων της συνάρτησης, καθώς και αυτές που δηλώνονται μέσα στο σώμα της
- Συνήθως, αυτή η μνήμη **δεσμεύεται** από ένα συγκεκριμένο τμήμα μνήμης που παρέχει το λειτουργικό σύστημα στο πρόγραμμα και ονομάζεται **στοίβα** (**stack**), χωρίς αυτό να είναι υποχρεωτικό
- Η **αποδέσμευση** αυτής της μνήμης **γίνεται αυτόματα** όταν τερματιστεί η εκτέλεση της συνάρτησης

Παρατηρήσεις

- Όταν χρησιμοποιείτε μία συνάρτηση βιβλιοθήκης, θα πρέπει να συμπεριλάβετε το αρχείο που περιέχει τη δήλωσή της με την οδηγία `#include`

- Για παράδειγμα, για να χρησιμοποιηθεί η συνάρτηση `strlen()` (δηλ. για να μπορέσετε να καλέσετε τη συγκεκριμένη συνάρτηση σε ένα πρόγραμμά σας), πρέπει να προστεθεί το αρχείο `cstring` με την οδηγία:

```
#include <cstring>
```

αλλιώς ο μεταγλωττιστής θα εμφανίσει μήνυμα λάθους για αδήλωτη συνάρτηση

Κλήση Συνάρτησης χωρίς Παραμέτρους

- Η κλήση μίας συνάρτησης που δεν δέχεται παραμέτρους γίνεται γράφοντας το όνομά της ακολουθούμενο από κενές παρενθέσεις **()**
- Στη συνάρτηση δεν μεταβιβάζεται πληροφορία
- Στο επόμενο παράδειγμα, η καλούσα συνάρτηση, δηλαδή η **main()**, καλεί δύο φορές τη συνάρτηση **test()**

```
#include <iostream>
void test();
int main()
{
  std::cout << "Call_1 ";
  test(); /* Κλήση συνάρτησης. Οι παρενθέσεις είναι κενές, γιατί
  η συνάρτηση δεν δέχεται παραμέτρους. */
  std::cout << "Call_2 ";
  test(); // Δεύτερη κλήση.
  return 0;
}
void test()
{
  // Σώμα συνάρτησης.
  for(int i = 0; i < 2; i++)
        std::cout << "In ";
}
```

Το πρόγραμμα εμφανίζει **Call_1 In In Call_2 In In**

Παράδειγμα Κλήσης Συνάρτησης Χωρίς Παραμέτρους που Επιστρέφει Τιμή

```
#include <iostream>

int test();

int main()
{
  int sum;

  sum = test(); /* Κλήση συνάρτησης. Η τιμή που
  επιστρέφει η test() αποθηκεύεται στη sum. */
  std::cout << sum << '\n';
  return 0;
}

int test()
{
  int i = 10, j = 20;
  return i+j;
}
```

Το πρόγραμμα εμφανίζει 30

Κλήση Συνάρτησης με Παραμέτρους (1)

- Η κλήση μίας συνάρτησης που δέχεται παραμέτρους σημαίνει ότι στη συνάρτηση μεταβιβάζεται πληροφορία μέσω των ορισμάτων της
- Η διαφορά μεταξύ παραμέτρου και ορίσματος είναι ότι ο όρος παράμετρος αναφέρεται στις μεταβλητές που εμφανίζονται στον ορισμό της συνάρτησης, ενώ ο όρος όρισμα αναφέρεται στις εκφράσεις που εμφανίζονται στην κλήση της συνάρτησης. Π.χ.

```
#include <iostream>

int test(int x, int y);

int main()
{
  int sum, a = 10, b = 20;

  sum = test(a, b); /* Οι μεταβλητές a και b είναι τα ορίσματα της
  συνάρτησης. */
  std::cout << sum << '\n';
  return 0;
}

int test(int x, int y) /* Οι μεταβλητές x και y είναι οι παράμετροι της
  συνάρτησης. */
{
  return x+y;
}
```

Κλήση Συνάρτησης με Παραμέτρους (2)

- Η κλήση της συνάρτησης γίνεται γράφοντας **το όνομά της** και μέσα σε παρενθέσεις **τη λίστα ορισμάτων**

- Ένα όρισμα μπορεί να είναι μία οποιαδήποτε έγκυρη έκφραση, όπως μία σταθερά, μία μεταβλητή, μία μαθηματική ή λογική έκφραση ή ακόμη και μία άλλη συνάρτηση με τιμή επιστροφής

- Το **πλήθος** των ορισμάτων και οι **τύποι** τους <u>πρέπει να ταιριάζουν</u> με τη δήλωση της συνάρτησης

- Αν το πλήθος είναι μικρότερο, ο μεταγλωττιστής θα εμφανίσει μήνυμα λάθους

- Αν οι τύποι των ορισμάτων δεν ταιριάζουν, ο μεταγλωττιστής θα προσπαθήσει να μετατρέψει τους τύπους των ασύμβατων ορισμάτων στους τύπους των αντίστοιχων παραμέτρων

- Αν τα καταφέρει, μπορεί να εμφανίσει μήνυμα προειδοποίησης (warning) για να ενημερώσει τον προγραμματιστή για τη μετατροπή. Αν δεν τα καταφέρει, θα εμφανίσει μήνυμα λάθους

- Όταν καλείται μία συνάρτηση οι τιμές της λίστας των ορισμάτων εκχωρούνται **μία-προς-μία** στις παραμέτρους της συνάρτησης

Επεξήγηση Παραδείγματος

- Ας επανέλθουμε στο πρόγραμμα. Όταν εκτελείται, ο μεταγλωττιστής δεσμεύει οκτώ θέσεις μνήμης (όπως έχουμε πει, θεωρούμε ότι ο `int` τύπος χρειάζεται τέσσερις οκτάδες) για την αποθήκευση των τιμών των **a** και **b**

- Όταν καλείται η `test()`, ο μεταγλωττιστής δεσμεύει άλλες οκτώ θέσεις για την αποθήκευση των τιμών των **x** και **y**

- Στη συνέχεια, αντιγράφει τις τιμές των ορισμάτων **a** και **b** (δηλ. 10 και 20) στις αντίστοιχες θέσεις μνήμης των παραμέτρων **x** και **y**

- Ουσιαστικά, κάθε παράμετρος μίας συνάρτησης είναι μία μεταβλητή που δημιουργείται όταν καλείται η συνάρτηση. Η τιμή με την οποία αρχικοποιείται είναι η τιμή του αντίστοιχου ορίσματος

- Όταν τερματιστεί η εκτέλεση της συνάρτησης `test()` γίνεται αυτόματη αποδέσμευση της μνήμης που είχε δεσμευτεί για τις παραμέτρους **x** και **y**

Κλήση Μέσω Τιμής

- **Αφού οι διευθύνσεις μνήμης των `x` και `y` είναι διαφορετικές από τις διευθύνσεις μνήμης των `a` και `b`, οποιαδήποτε αλλαγή γίνει στις τιμές των `x` και `y` δεν επηρεάζει τις τιμές των `a` και `b`**
- **Αυτός ο τρόπος μεταβίβασης τιμών ονομάζεται κλήση κατ'αξία ή κλήση μέσω τιμής (*call by value*)**
- **Δηλαδή, η καλούμενη συνάρτηση δέχεται τις τιμές των ορισμάτων της σε αντίγραφα των πρωτότυπων μεταβλητών. Επομένως, μπορεί να αλλάξει μόνο την τιμή του αντιγράφου της και όχι της πρωτότυπης μεταβλητής**
- **Όσον αφορά την απόδοση, το κόστος αυτού του τρόπου μεταβίβασης είναι η μνήμη και ο χρόνος που χρειάζεται για την αντιγραφή των ορισμάτων**
- **Όπως θα δούμε στη συνέχεια, εξαίρεση σε αυτόν τον κανόνα αποτελούν οι πίνακες. Όταν ένας πίνακας μεταβιβάζεται σε μία συνάρτηση, δεν δημιουργείται αντίγραφο του πίνακα, αλλά μεταβιβάζεται η διεύθυνση του πρώτου του στοιχείου. Η συνάρτηση μπορεί να αλλάξει την τιμή οποιουδήποτε στοιχείου του πίνακα**

Παράδειγμα

- Πριν συνεχίσουμε στην επόμενη ενότητα, ας κάνουμε μία άσκηση. Βρείτε τα λάθη στο παρακάτω πρόγραμμα:

```
#include <iostream> // Πρόγραμμα με λάθη μεταγλώττισης.

void f(int a, int b);

int main()
{
  int i, j, k;

  k = f(i, &j);
  return 0;
}

void f(int a, int b)
{
  if(a > 0)
       return b;
}
```

Παράδειγμα

Απάντηση. Η κλήση της `f()` είναι λάθος, γιατί ο τύπος της έκφρασης `&j` είναι δείκτης σε ακέραιο, και όχι ακέραιος όπως θα έπρεπε να είναι σύμφωνα με το πρωτότυπο. Η ανάθεση τιμής στο `k` είναι λάθος, γιατί η `f()` δεν επιστρέφει τιμή. Στην `f()`, η εντολή `return b;` είναι λάθος, γιατί η `f()` δεν επιστρέφει τιμή.

Αλλαγή Πρωτότυπων Τιμών

- Αν επιθυμούμε μία συνάρτηση να μπορεί να αλλάξει την τιμή κάποιου ορίσματος, πρέπει να μεταβιβάσουμε στη συνάρτηση τη διεύθυνση μνήμης του ορίσματος και όχι την τιμή του

- Επομένως, αφού η συνάρτηση θα έχει πρόσβαση στη διεύθυνση μνήμης του ορίσματος, θα μπορεί να μεταβάλλει την τιμή του

- Η αντίστοιχη παράμετρος πρέπει να δηλωθεί σαν δείκτης και η συνάρτηση μπορεί να χρησιμοποιήσει τον δείκτη για να αποκτήσει πρόσβαση στην μεταβλητή

Παράδειγμα

```
#include <iostream>

void test(int *p, int a);

int main()
{
  int i = 10, j = 20;

  test(&i, j);
  std::cout << i << ' ' << j << '\n';
  return 0;
}

void test(int *p, int a)
{
  *p = 30;
  a = 40;
}
```

Αφού ο τύπος της έκφρασης &i είναι δείκτης σε ακέραιο, η κλήση της `test()` *ταιριάζει με το πρωτότυπο της συνάρτησης. Όταν γίνεται η κλήση της έχουμε p = &i. Αφού ο δείκτης p δείχνει στο i, η* `test()` *μπορεί να αλλάξει την τιμή του i. Ειδικότερα, το *p είναι ισοδύναμο με το i και η εντολή *p = 30; αλλάζει την τιμή του i από 10 σε 30. Όπως φαίνεται, όταν γίνεται κλήση μίας συνάρτησης, επιτρέπεται να γίνει συνδυασμός στη μεταβίβαση των ορισμάτων, δηλαδή, για κάποια ορίσματα να διοχετευθούν οι διευθύνσεις τους και για κάποια άλλα οι τιμές τους. Έτσι, η τιμή του j δεν αλλάζει και το πρόγραμμα εμφανίζει 30 και 20*

Παρατηρήσεις

- Αφού μία συνάρτηση δεν μπορεί να επιστρέψει περισσότερες από μία τιμές, η μεταβίβαση των διευθύνσεων των ορισμάτων αποτελεί έναν ευέλικτο τρόπο για την αλλαγή των τιμών τους. Αυτή είναι μία μεγάλη χρησιμότητα των δεικτών

- Όσον αφορά την απόδοση, όταν μεταβιβάζουμε απλές μεταβλητές (π.χ. `int`) τις μεταβιβάζουμε όπως έχουμε δει, ενώ όταν μεταβιβάζουμε μεγάλες οντότητες (π.χ. ένα vector που περιέχει χιλιάδες στοιχεία) είναι πιο αποδοτικό να μεταβιβάσουμε την διεύθυνση μνήμης της οντότητας, ώστε να αποφύγουμε τον χρόνο και τη μνήμη που απαιτείται για την αντιγραφή της οντότητας

- Στο Κ.16 θα μιλήσουμε για έναν εναλλακτικό τρόπο για να μεταβιβάσουμε την διεύθυνση, μέσω αναφοράς (*call by reference*)

- Αν θέλουμε μόνο να χρησιμοποιήσουμε την τιμή του ορίσματος και να αποτρέψουμε τυχόν αλλαγές κάνουμε τον δείκτη `const`. Π.χ.

```
void test(const int *p)
{
  if(*p == 10) // Σωστό.
       *p = 30; // Λάθος.
  ...
}
```

Παράδειγμα

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>

void test(int *arg);
int var = 100;

int main()
{
  int *ptr, i = 30;

  ptr = &i;
  test(ptr);
  std::cout << *ptr << '\n';
  return 0;
}

void test(int *arg)
{
  arg = &var;
}
```

Παράδειγμα

Απάντηση. Επειδή η `test()` δέχεται σαν όρισμα την τιμή του `ptr` και όχι τη διεύθυνσή του, δεν μπορεί να αλλάξει την τιμή του. Άρα, οποιαδήποτε μεταβολή στην τιμή του `arg` δεν επηρεάζει την τιμή του `ptr` και το πρόγραμμα εμφανίζει 30. Για την εμβέλεια της `var` θα μιλήσουμε σε λίγο. Αν θέλουμε μία συνάρτηση να μπορεί να αλλάξει την τιμή ενός δείκτη μεταβιβάζουμε στη συνάρτηση όρισμα δείκτη σε δείκτη. Δείτε το επόμενο παράδειγμα

Παράδειγμα Μεταβίβασης Δείκτη σε Δείκτη

Αν θέλουμε μία συνάρτηση να μπορεί να αλλάξει την τιμή ενός δείκτη μεταβιβάζουμε στη συνάρτηση όρισμα δείκτη σε δείκτη. Π.χ.

```
#include <iostream>

void test(int **arg);
int var = 100;

int main()
{
  int *ptr, i = 30;

  ptr = &i;
  test(&ptr); /* Η τιμή του &ptr είναι η διεύθυνση μνήμης του ptr, ο οποίος με τη
  σειρά του δείχνει στη διεύθυνση του i. Άρα, ο τύπος του ορίσματος είναι δείκτης
  σε δείκτη σε ακέραιο και συμφωνεί με την int** δήλωση της συνάρτησης. */
  std::cout << *ptr << '\n';
  return 0;
}

void test(int **arg)
{
  *arg = &var;
}
```

Αφού στην `test()` μεταβιβάζεται η διεύθυνση της μεταβλητής `ptr`, η `test()` μπορεί να αλλάξει την τιμή της. Συγκεκριμένα, όταν καλείται η test() έχουμε arg = &ptr, άρα *arg = ptr. Επομένως, η έκφραση *arg = &var ισοδυναμεί με ptr = &var, δηλαδή, η τιμή του `ptr` αλλάζει και δείχνει στη διεύθυνση της `var`. Άρα, το πρόγραμμα εμφανίζει `100`

Παράδειγμα (1)

- Δημιουργήστε μία συνάρτηση που να δέχεται σαν παράμετρο έναν ακέραιο και να επιστρέφει το τετράγωνο του αριθμού και άλλη μία που να επιστρέφει τον κύβο του αριθμού. Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει έναν ακέραιο και να εμφανίζει το άθροισμα του τετραγώνου και του κύβου του αριθμού με χρήση των συναρτήσεων

Παράδειγμα (1)

```
#include <iostream>

int square(int a);
int cube(int a);

int main()
{
  int i, j, k;

  std::cout << "Enter number: ";
  std::cin >> i;

  j = square(i);
  k = cube(i);
  std::cout << j+k << '\n'; /* Θα μπορούσαμε να μην χρησιμοποιήσουμε τις
  μεταβλητές j, k και να γράψουμε κατευθείαν cout << square(i)+cube(i); */
  return 0;
}

int square(int a)
{
   return a*a;
}

int square(int a)
{
   return a*a*a;
}
```

Παράδειγμα (2)

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>

int f(int a);

int main()
{
  int i = 10;

  std::cout << f(3-f(2*f(i+1))) << '\n';
  return 0;
}

int f(int a)
{
   return a+1;
}
```

Παράδειγμα (2)

- Απάντηση. Κάθε φορά που καλείται η `f()` επιστρέφει την τιμή του ορίσματος αυξημένη κατά ένα. Οι κλήσεις της `f()` εκτελούνται από μέσα προς τα έξω. Άρα, η πρώτη κλήση επιστρέφει την τιμή `12` και το όρισμα της δεύτερης κλήσης είναι η τιμή `24`. Η δεύτερη κλήση επιστρέφει την τιμή `25` και το όρισμα της τρίτης κλήσης είναι η τιμή `-22`. Επομένως, το πρόγραμμα θα εμφανίσει `-21`

Παράδειγμα (3)

- Δημιουργήστε μία συνάρτηση που να δέχεται σαν παραμέτρους έναν ακέραιο και έναν χαρακτήρα και να εμφανίζει τον χαρακτήρα τόσες φορές όσες και η τιμή του ακεραίου. Επίσης, η συνάρτηση με χρήση της εντολής `switch` να επιστρέφει τον ίδιο τον χαρακτήρα αν αυτός είναι `'a'`, `'b'` ή `'c'`, αλλιώς τον επόμενο χαρακτήρα στο ASCII σύνολο. Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει έναν ακέραιο και έναν χαρακτήρα, να καλεί τη συνάρτηση και να εμφανίζει την τιμή επιστροφής

Παράδειγμα (3)

```
#include <iostream>
using std::cout;
using std::cin;

char show(int num, char c);

int main()
{
   char c;
   int i;

   cout << "Enter character: ";
   cin >> c;
   cout << "Enter number: ";
   cin >> i;

   c = show(i, c);
   cout << '\n' << c << '\n'; /* Θα μπορούσαμε να μην καλέσουμε σε ξεχωριστή εντολή τη
   show() και στη θέση του c να γράψουμε show(i, c) */
   return 0;
}

char show(int num, char c)
{
   for(int i = 0; i < num; i++)
          cout << c;

   switch(c)
   {
          case 'a':
          case 'b':
          case 'c':
          return c;

          default:
          return c+1;
   }
}
```

Παράδειγμα (4)

- Δημιουργήστε μία συνάρτηση που να δέχεται σαν παραμέτρους δύο δείκτες σε πραγματικούς και να αντιμεταθέτει τα περιεχόμενά τους. Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει δύο πραγματικούς και να τους αντιμεταθέτει με χρήση της συνάρτησης

Παράδειγμα (4)

```
#include <iostream>

void swap(float *ptr1, float *ptr2);

int main()
{
  float i, j;

  std::cout << "Enter numbers: ";
  std::cin >> i >> j;

  swap(&i, &j);
  std::cout << i << ' ' << j << '\n';
  return 0;
}

void swap(float *ptr1, float *ptr2)
{
  float m;

  m = *ptr1; // Ισοδύναμο με m = i;
  *ptr1 = *ptr2; // Ισοδύναμο με i = j;
  *ptr2 = m; // Ισοδύναμο με j = m;
}
```

Εμβέλεια Μεταβλητών

- **Εμβέλεια** μίας μεταβλητής καλείται το τμήμα του προγράμματος, στο οποίο η μεταβλητή είναι **προσβάσιμη**, ή αλλιώς λέμε ότι **είναι «ορατή»** από τα υπόλοιπα τμήματα του προγράμματος

- Η εμβέλεια μίας μεταβλητής εξαρτάται από το σημείο δήλωσής της μέσα στο πρόγραμμα

- Τα διαφορετικά είδη μεταβλητών βάσει της εμβέλειάς των είναι:
  - Οι **τοπικές** μεταβλητές (**local** variables)
  - Οι **καθολικές** μεταβλητές (**global** variables)

Τοπικές Μεταβλητές

- Μία μεταβλητή που δηλώνεται στο σώμα μίας συνάρτησης ονομάζεται τοπική
- Η εμβέλειά της περιορίζεται μέσα στη συνάρτηση από το σημείο της δήλωσής της μέχρι το τέλος του τμήματος που περικλείει τη δήλωση και ονομάζεται εμβέλεια τμήματος. Αυτό σημαίνει ότι οι υπόλοιπες συναρτήσεις του προγράμματος δεν έχουν πρόσβαση σε αυτή
- Αφού μία τοπική μεταβλητή δεν είναι ορατή έξω από τη συνάρτηση στην οποία δηλώνεται, μπορούμε να δηλώσουμε μεταβλητές με το ίδιο όνομα σε άλλες συναρτήσεις
- Σημειώστε ότι οι παράμετροι που εμφανίζονται στη δήλωση μίας συνάρτησης θεωρούνται και αυτές τοπικές μεταβλητές της συνάρτησης
- Κάθε φορά που καλείται μία συνάρτηση ο μεταγλωττιστής δεσμεύει μνήμη για να δημιουργήσει τις τοπικές μεταβλητές της. Αυτή η μνήμη αποδεσμεύεται αυτόματα όταν τελειώσει η εκτέλεση της συνάρτησης, άρα, μία τοπική μεταβλητή δεν διατηρεί την τιμή της μεταξύ διαδοχικών κλήσεων

Παράδειγμα

- Στο επόμενο πρόγραμμα, η τοπική μεταβλητή i που δηλώνεται στη main() είναι διαφορετική από την τοπική μεταβλητή i που δηλώνεται στην test(), παρόλο που έχουν το ίδιο όνομα

```
#include <iostream>

void test();

int main()
{
   int i = 10;

   test();
   std::cout << "I_main = " << i << '\n';
   return 0;
}

void test()
{
   int i = 200;
   std::cout << "I_test = " << i << '\n';
}
```

Αφού είναι διαφορετικές μεταβλητές, η κάθε μία έχει τη δική της τιμή και το πρόγραμμα εμφανίζει: I_test = 200 *και* I_main = 10*. Τι θα συνέβαινε αν δεν δηλώναμε την* i *σαν* int *στην* test() *και γράφαμε* i = 200*;*

Αφού οι δύο μεταβλητές i *είναι ασυσχέτιστες μεταξύ τους, ο μεταγλωττιστής θα εμφάνιζε μήνυμα λάθους ότι η μεταβλητή* i *στην* test() *δεν έχει δηλωθεί.*

Δήλωση Μεταβλητών μέσα σε Τμήμα

- Επιτρέπεται να δηλώσουμε μεταβλητές μετά το { μίας σύνθετης εντολής (π.χ. if)

- Η εμβέλεια μίας μεταβλητής που δηλώνεται σε ένα τμήμα εκτείνεται μέχρι το αντίστοιχο }, άρα δεν σχετίζεται με τυχόν ομώνυμες μεταβλητές έξω από το τμήμα

- Ο μεταγλωττιστής δεσμεύει μνήμη για αυτήν όταν εισέρχεται στο τμήμα και την αποδεσμεύει όταν εξέρχεται. Π.χ.

```
#include <iostream>
int main()
{
  int i = 20, num = 10;

  if(num == 10)
  {
        int i; // Δέσμευση μνήμης για τη νέα i.
        i = 50;
  } // Αποδέσμευση μνήμης.
  std::cout << i << '\n';
  return 0;
}
```

Επειδή η i που δηλώνεται στην if εντολή δεν έχει καμία σχέση με την πρώτη i, το πρόγραμμα εμφανίζει 20. Ένας λόγος για να δηλώσουμε μεταβλητές μέσα σε ένα τμήμα και όχι στην αρχή της συνάρτησης είναι ότι θα δεσμευτεί μνήμη για αυτές μόνο αν χρειαστεί. Αυτό μπορεί να είναι χρήσιμο για ανάπτυξη εφαρμογών σε συστήματα με περιορισμένη μνήμη

Στατικές Μεταβλητές

- Όπως συζητήσαμε, η μνήμη που δεσμεύεται για μία τοπική μεταβλητή αποδεσμεύεται όταν τελειώσει η εκτέλεση της συνάρτησης
- Επομένως, αν κληθεί πάλι η συνάρτηση, δεν υπάρχει καμία εγγύηση ότι θα έχει διατηρήσει την τιμή της
- Αν θέλουμε μία τοπική μεταβλητή να διατηρεί την τιμή της, πρέπει να δηλωθεί με την λέξη `static`
- Η μνήμη για μία `static` μεταβλητή δεσμεύεται μόνο την πρώτη φορά που θα κληθεί η συνάρτηση. Αυτή η μνήμη δεν αποδεσμεύεται όταν τερματιστεί η συνάρτηση, αλλά όταν τερματιστεί το πρόγραμμα
- Άρα, μία στατική μεταβλητή διατηρεί την τελευταία τιμή της
- Μία στατική μεταβλητή αρχικοποιείται μόνο την πρώτη φορά που καλείται η συνάρτηση. Αν δεν της ανατεθεί κάποια τιμή, αρχικοποιείται με `0`. Στις επόμενες κλήσεις της συνάρτησης, η μεταβλητή διατηρεί την τελευταία τιμή της και δεν αρχικοποιείται πάλι

Παράδειγμα

- **Ποια είναι η έξοδος του παρακάτω προγράμματος;**

```
#include <iostream>

void test();

int main()
{
  test();
  test();
  test();
  return 0;
}

void test()
{
  static int i = 100, arr[1];
  int j = 0;

  i++;
  arr[0] += 2;
  j++;
  std::cout << i << ' ' << arr[0] << ' ' << j << '\n';
}
```

Παράδειγμα

- **Απάντηση. Στην πρώτη κλήση της `test()` η τιμή του i γίνεται 101. Αφού το i έχει δηλωθεί σαν στατική μεταβλητή, η τιμή του διατηρείται και με την επόμενη κλήση της `test()` γίνεται 102. Παρομοίως, με την τρίτη κλήση γίνεται 103. Αφού τα στοιχεία ενός στατικού πίνακα αρχικοποιούνται εξ'ορισμού με 0, η τιμή του arr[0] γίνεται 2, 4 και 6, αντίστοιχα. Αντίθετα, η j είναι μία αυτόματη μεταβλητή που δεν διατηρεί την τιμή της ανάμεσα στις κλήσεις της `test()`. Επομένως, το πρόγραμμα εμφανίζει:**

  101 2 1
  102 4 1
  103 6 1

Παράδειγμα

- Αφού η μνήμη μίας απλής τοπικής μεταβλητής αποδεσμεύεται, μία συνάρτηση **δεν πρέπει** να επιστρέφει τη διεύθυνσή της, π.χ.

```
#include <iostream>

int *test();

int main()
{
  int *ptr, j;

  ptr = test();
  std::cout << *ptr << '\n';

  j = *ptr;
  std::cout << j << '\n';
  return 0;
}

int *test()
{
  int i = 10;
  return &i; /* Σοβαρό λάθος, μην το
  κάνετε. */
}
```

Όταν καλείται η test(), ο μεταγλωττιστής δεσμεύει μνήμη για την τοπική μεταβλητή i. Η εντολή return &i; επιστρέφει τη διεύθυνση μνήμης της.

Όμως, αυτή η μνήμη αποδεσμεύεται όταν τερματίζει η εκτέλεση της συνάρτησης.

Επομένως, μπορεί να αποθηκευτούν νέα δεδομένα σε αυτή τη μνήμη και να χαθεί η τιμή 10.

Άρα, το πρόγραμμα αντί για τις τιμές 10 και 10 μπορεί να εμφανίσει 10 και μία άλλη τυχαία τιμή ή ακόμα και δύο τυχαίες τιμές.

Η συμπεριφορά του προγράμματος είναι **απρόβλεπτη**

Παρατηρήσεις

- Αν, στο προηγούμενο παράδειγμα, η τοπική μεταβλητή `i` είχε δηλωθεί ως `static`, η συνάρτηση θα μπορούσε να επιστρέφει τη διεύθυνση της `i`, δεδομένου ότι η μνήμη που έχει δεσμευτεί για μία `static` μεταβλητή δεν αποδεσμεύεται όταν τερματιστεί η συνάρτηση

- Προσοχή Λοιπόν!!!

  Μην επιστρέφετε τη διεύθυνση μίας τοπικής μεταβλητής, εκτός αν έχει δηλωθεί ως `static`

Καθολικές Μεταβλητές

- Μία μεταβλητή που δηλώνεται έξω από οποιαδήποτε συνάρτηση και τύπους με εμβέλεια τμήματος, όπως απαρίθμηση, κλάση και χώρος ονομάτων, ονομάζεται καθολική (global) μεταβλητή
- Η εμβέλειά της εκτείνεται από το σημείο της δήλωσής της μέχρι το τέλος του αρχείου στο οποίο δηλώνεται. Επομένως, όλες οι συναρτήσεις που ορίζονται μετά από το σημείο δήλωσής της έχουν πρόσβαση σε αυτήν
- Μία καθολική μεταβλητή έχει μόνιμη διάρκεια ζωής και διατηρεί την τελευταία τιμή που της εκχωρήθηκε. Δεσμεύεται μνήμη για αυτήν όταν αρχίζει το πρόγραμμα και αποδεσμεύεται όταν τερματίσει η εκτέλεσή του

Παρατηρήσεις

- Όταν η ίδια μεταβλητή χρησιμοποιείται σε πολλές συναρτήσεις, πολλοί προγραμματιστές συνηθίζουν να τη δηλώνουν σαν καθολική, αντί να τη μεταβιβάζουν σαν όρισμα στις κλήσεις των συναρτήσεων

- Ουσιαστικά, μία καθολική μεταβλητή μπορεί να χρησιμοποιηθεί σαν εναλλακτικός τρόπος για τη μεταβίβαση πληροφορίας σε μία συνάρτηση

- Για καλύτερο έλεγχο και κατανόηση του προγράμματος, είναι πολύ βοηθητικό το όνομα που θα επιλέξετε για μία καθολική μεταβλητή να περιγράφει τον ρόλο της. Για παράδειγμα, μην χρησιμοποιείτε ονόματα τα οποία συνήθως δίνονται σε τοπικές μεταβλητές (π.χ. `i`)

- Εξ'ορισμού, ο μεταγλωττιστής αρχικοποιεί μία καθολική μεταβλητή ή τα μέλη ενός σύνθετου τύπου (π.χ. πίνακας) με `0`

- Αν και η χρήση καθολικών μεταβλητών μπορεί να φαίνεται ότι διευκολύνει την δουλειά του προγραμματιστή, μάλλον είναι καλύτερα να αποφεύγετε τη χρήση τους ή τουλάχιστον τη συχνή χρήση τους. Ένας λόγος είναι ότι δυσκολεύει η ανίχνευση προβληματικών καταστάσεων. Για παράδειγμα, όταν πολλές συναρτήσεις έχουν πρόσβαση στην ίδια μεταβλητή και κάποια από αυτές της εκχωρήσει μία ανεπιθύμητη τιμή, πρέπει να βρούμε και να ελέγξουμε όλες τις συναρτήσεις που την χρησιμοποιούν για να εντοπιστεί η «ένοχη» συνάρτηση

Παράδειγμα

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>

void add();
void sub();

int glob = 0;/* Αν και ο μεταγλωττιστής την αρχικοποιεί με 0, προτιμώ να το κάνω ξεκάθαρα. */

int main()
{
   add();
   glob += 2;
   sub();
   std::cout << glob << '\n';
   return 0;
}

void add()
{
   glob++;
}

void sub()
{
   glob--;
}
```

- Όλες οι συναρτήσεις έχουν πρόσβαση στην καθολική μεταβλητή `glob`, αφού η δήλωσή της γίνεται πριν από τους ορισμούς τους. Το πρόγραμμα εμφανίζει 2

Παρατηρήσεις

- Μία τοπική μεταβλητή μπορεί να έχει το ίδιο όνομα με μία καθολική. Ο κανόνας εμβέλειας λέει ότι όταν μία μεταβλητή σε ένα τμήμα κώδικα ονοματίζεται το ίδιο με μία άλλη που είναι ήδη ορατή, η νέα δήλωση «κρύβει» την παλιά στο συγκεκριμένο τμήμα

- Π.χ. στο παρακάτω πρόγραμμα η τοπική μεταβλητή **a** που δηλώνεται στην **test()** είναι διαφορετική από την καθολική μεταβλητή **a**. Το πρόγραμμα εμφανίζει 100

```
#include <iostream>

void test();

int a = 100;

int main()
{
  test();
  std::cout << a << '\n';
  return 0;
}
void test()
{
  int a;
  a = 2000;
}
```

Εξωτερικές Καθολικές Μεταβλητές (extern)

- Μία καθολική μεταβλητή μπορεί να γίνει ορατή σε περισσότερα από ένα αρχεία

- Αυτό είναι πολύ χρήσιμο, ιδιαίτερα στις περιπτώσεις μεγάλων προγραμμάτων, που είναι πολύ πιθανό διαφορετικά αρχεία να χρειάζεται να χρησιμοποιήσουν την ίδια μεταβλητή

- Η λέξη `extern` είναι ένα προσδιοριστικό κατηγορίας αποθήκευσης, η οποία επιτρέπει μία μεταβλητή να μπορεί να γίνει ορατή σε διαφορετικά αρχεία

- Συνήθως, μία τέτοια μεταβλητή δηλώνεται σαν καθολική στο αρχείο που χρησιμοποιείται περισσότερο και στα υπόλοιπα αναφέρεται ως `extern`

- Π.χ. η εντολή: `extern int size;`

  ενημερώνει τον μεταγλωττιστή ότι η ακέραια μεταβλητή με όνομα `size` δεν ορίζεται σε αυτό το αρχείο, αλλά σε κάποιο άλλο αρχείο του προγράμματος

- Η `size` μπορεί να χρησιμοποιηθεί και, αν χρειαστεί, να τροποποιηθεί η τιμή της σε οποιοδήποτε από τα αρχεία στα οποία εμφανίζεται σαν `extern`

Συνάρτηση με Παράμετρο Πίνακα

- Όταν η παράμετρος μίας συνάρτησης είναι ένας μονοδιάστατος πίνακας, γράφουμε το όνομα του πίνακα ακολουθούμενο από κενές αγκύλες. Π.χ. `void test(int arr[]);`
- Όταν μεταβιβάζεται ένας πίνακας σε μία συνάρτηση, γράφουμε μόνο το όνομα του πίνακα, χωρίς τις αγκύλες. Π.χ. `test(arr);`
- Όταν το όνομα ενός πίνακα μεταβιβάζεται σε μία συνάρτηση, χρησιμοποιείται σαν δείκτης. Ουσιαστικά, μεταβιβάζεται ένας δείκτης στο πρώτο του στοιχείο και όχι ένα αντίγραφο του πίνακα. Επομένως, αφού δεν γίνεται αντιγραφή του πίνακα, ο χρόνος που απαιτείται για να μεταβιβαστεί ένας πίνακας σε μία συνάρτηση δεν εξαρτάται από το μέγεθός του
- Να θυμόμαστε λοιπόν, όταν μία απλή μεταβλητή μεταβιβάζεται σε μία συνάρτηση, η συνάρτηση εργάζεται με αντίγραφό της. Όμως, όταν μεταβιβάζεται πίνακας, δεν δημιουργείται αντίγραφό του, η συνάρτηση εργάζεται με τον πρωτότυπο πίνακα
- Και γιατί λήφθηκε αυτή η απόφαση; Για καλύτερη απόδοση, για να γλιτώσουμε την μνήμη και τον χρόνο που απαιτείται για την αντιγραφή όλων των στοιχείων του. Απλά, μεταβιβάζεται ο δείκτης στο πρώτο του στοιχείο και η συνάρτηση μπορεί να προσπελάσει όποιο στοιχείο επιθυμεί

Παράδειγμα

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>

void test(int arr[]);

int main()
{
  int i, a[5] = {10, 20, 30, 40, 50};

  test(a); // Ισοδύναμο με test(&a[0]);
  for(i = 0; i < 5; i++)
       std::cout << a[i] << ' ';
  return 0;
}

void test(int arr[])
{
  arr[0] = arr[1] = 0;
}
```

Παράδειγμα

- *Απάντηση.* Σημειώστε ότι στην `test()` θα μπορούσαμε να χρησιμοποιήσουμε σαν όνομα παραμέτρου το όνομα `a` αντί για `arr`, αφού, ήδη γνωρίζετε ότι, οι τοπικές μεταβλητές διαφορετικών συναρτήσεων δεν σχετίζονται μεταξύ τους, ακόμα και αν έχουν το ίδιο όνομα. Όταν καλείται η `test()` έχουμε `arr = a = &a[0]`. Επομένως, οι εντολές `arr[0] = 0;` και `arr[1] = 0;` αλλάζουν τις τιμές των δύο πρώτων στοιχείων του πίνακα και το πρόγραμμα εμφανίζει: `0 0 30 40 50`

Παρατηρήσεις (1)

- **Μία παράμετρος πίνακας μπορεί επίσης να δηλωθεί σαν δείκτης. Για παράδειγμα, οι δηλώσεις:**

  `void test(int arr[]);` **και** `void test(int *arr);`

  είναι ισοδύναμες. Ο μεταγλωττιστής τις χειρίζεται με τον ίδιο τρόπο και μεταβιβάζει στη συνάρτηση τον δείκτη. Άρα, η δεύτερη δήλωση είναι πιο ακριβής, αφού δηλώνει ξεκάθαρα ότι στη συνάρτηση μεταβιβάζεται δείκτης και όχι αντίγραφο του πίνακα. Σαν προτίμηση, προτιμώ τον πρώτο τρόπο, ώστε να φαίνεται ξεκάθαρα η πρόθεση του προγραμματιστή να μεταβιβάσει στη συνάρτηση πίνακα. Με τη δεύτερη δήλωση ο αναγνώστης του κώδικα δεν μπορεί να καταλάβει αν η συνάρτηση θα δεχτεί ένα πλήθος τιμών ή έναν δείκτη σε μία μόνο τιμή

- **Για ασφάλεια, αν δεν θέλουμε μία συνάρτηση να μπορεί να αλλάξει τις τιμές των στοιχείων του πίνακα, δηλώνουμε την παράμετρο σαν `const`. Π.χ. με την παρακάτω δήλωση η συνάρτηση `test()` μπορεί να προσπελάσει τα στοιχεία του πίνακα `arr`, αλλά δεν μπορεί να αλλάξει τις τιμές τους**

```
void test(const int arr[]);
```

Παρατηρήσεις (2)

- Εναλλακτικά, θα μπορούσαμε να χρησιμοποιήσουμε σημειογραφία δείκτη για να προσπελάσουμε τα στοιχεία του πίνακα. Π.χ:

```
void test(int arr[])
{
  *arr = 0; // Ισοδύναμο με arr[0] = 0.
  arr++;
  *arr = 0;
}
```

- Αυτό που έχει ενδιαφέρον είναι η εντολή `arr++`. Μπορούμε να αλλάξουμε την τιμή μίας μεταβλητής πίνακα; Όχι βέβαια, όπως έχουμε πει στο Κ.8, όταν το όνομα ενός πίνακα χρησιμοποιείται σαν δείκτης είναι `const` δείκτης. Επομένως, γιατί ο μεταγλωττιστής επιτρέπει την παραπάνω εντολή; Γιατί, όπως είπαμε, η παράμετρος `arr` μπορεί να δηλώνεται σαν πίνακας, αλλά στην πραγματικότητα είναι δείκτης. Άρα, μπορούμε να του εκχωρήσουμε μία νέα τιμή

Προσδιορισμός Τέλους Πίνακα (1)

- Επειδή λοιπόν στη συνάρτηση μεταβιβάζεται ο δείκτης, δεν παίζει κανένα ρόλο αν μέσα στις αγκύλες προσθέσουμε το μήκος του πίνακα

  (π.χ. `void test(int arr[5])`)
- Ο μεταγλωττιστής το αγνοεί, δεν πρόκειται να ελέγξει αν ο πίνακας έχει πράγματι το μήκος που δηλώνεται (π.χ. 5)
- Στην πραγματικότητα, αυτό που μεταβιβάζεται είναι ένας δείκτης στο πρώτο στοιχείο ενός πίνακα αγνώστου μήκους
- Αυτός είναι ο λόγος που οι αγκύλες συνηθίζεται να είναι κενές, ώστε να μην δημιουργείται η εσφαλμένη εντύπωση ότι ο μεταγλωττιστής απαιτεί ο πίνακας που μεταβιβάζεται να έχει ένα συγκεκριμένο πλήθος στοιχείων. Όχι, ο μεταγλωττιστής μεταγλωττίζει το πρόγραμμα ανεξάρτητα από το μήκος του πίνακα

Προσδιορισμός Τέλους Πίνακα (2)

- Ένας εύκολος τρόπος για να μάθει η συνάρτηση τον αριθμό των στοιχείων του πίνακα είναι να το μεταβιβάσουμε σαν όρισμα. Π.χ:

```
void test(int arr[], int size);
```

  και για να την καλέσουμε να γράψουμε: `test(a, 5);`

- Εναλλακτικά, μπορούμε να δηλώσουμε μία σταθερά όπως,

  `const int SIZE = 5;` και να τη χρησιμοποιούμε όπου χρειάζεται, αντί να μεταβιβάζουμε ένα πρόσθετο όρισμα

- Σημειώστε ότι αν και μπορούμε να χρησιμοποιήσουμε τον τελεστή `sizeof` για να υπολογίσουμε το μέγεθος ενός πίνακα, δεν μπορούμε να τον χρησιμοποιήσουμε για να υπολογίσουμε το μέγεθος ενός πίνακα που δηλώνεται σαν παράμετρος σε μία συνάρτηση. Αφού, όπως είπαμε, στη συνάρτηση μεταβιβάζεται δείκτης, ο τελεστής `sizeof` υπολογίζει το μέγεθος της μνήμης που δεσμεύει μία μεταβλητή δείκτης (π.χ. 4) και όχι τη μνήμη που δεσμεύει ο πίνακας

Μεταβίβαση Τμήματος Πίνακα

- Σε μία συνάρτηση μπορούμε να μεταβιβάσουμε ένα τμήμα του πίνακα. Για παράδειγμα, ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>

void test(int ptr[]);

int main()
{
  int i, arr[6] = {1, 2, 3, 4, 5, 6};

  test(arr+3); // Εναλλακτικά, test(&arr[3]).
  for(i = 0; i < 6; i++)
        std::cout << arr[i] << ' ';
  return 0;
}

void test(int ptr[])
{
  int i, tmp[3] = {10, 20, 30};

  for(i = 0; i < 3; i++)
        ptr[i] = tmp[i];

  *ptr = *(ptr-1);
}
```

Μεταβίβαση Τμήματος Πίνακα

- Απάντηση. Με την κλήση της test() έχουμε ptr = arr+3, δηλαδή, μεταβιβάζουμε στην test() το τμήμα του πίνακα arr από το τέταρτο στοιχείο και μετά. Αφού χρησιμοποιούμε τον δείκτη ptr σαν πίνακα, το ptr[0] αντιστοιχεί στο arr[3], το ptr[1] στο arr[4] και το ptr[2] στο arr[5]. Επομένως, με τον βρόχο, οι τιμές των arr[3], arr[4] και arr[5] γίνονται 10, 20 και 30, αντίστοιχα. Αφού ο ptr δείχνει στο arr[3], η εντολή *ptr = *(ptr-1); είναι ισοδύναμη με arr[3] = arr[2]; Άρα, το πρόγραμμα εμφανίζει: 1 2 3 3 20 30

Μεταβίβαση vector Αντικειμένου

- Και βέβαια, σε μία συνάρτηση μπορούμε να μεταβιβάσουμε ένα vector αντικείμενο. Για παράδειγμα, δείτε την παρακάτω άσκηση.

  Δημιουργήστε μία συνάρτηση που να δέχεται σαν παραμέτρους ένα vector αντικείμενο με βαθμούς φοιτητών και δύο βαθμούς (π.χ. a και b) και να επιστρέφει τον μέσο όρο των βαθμών που ανήκουν στο [a, b]. Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει τον αριθμό των φοιτητών, να δημιουργεί ένα vector αντικείμενο, να διαβάζει τους βαθμούς τους και να τους καταχωρεί σε αυτό. Μετά, να διαβάζει τους βαθμούς a και b, να καλεί την συνάρτηση και να εμφανίζει τον μέσο όρο. Το πρόγραμμα να υποχρεώνει τον χρήστη η τιμή του a να είναι μικρότερη ή ίση του b.

Παράδειγμα (1)

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

float avg_arr(const vector<float>& v, float min, float max);

int main()
{
   int i, num;
   float a, b, k;

   cout << "Enter number of students: ";
   cin >> num;

   vector<float> grd_v(num);
   for(i = 0; i < num; i++)
   {
          cout << "Enter grade: ";
          cin >> grd_v[i];
   }
   do
   {
          cout << "Enter min and max grades: ";
          cin >> a >> b;
   } while(a > b);

   k = avg_arr(grd_v, a, b);
   if(k == -1)
          cout << "None grade in the indicated set\n";
   else
          cout << "Avg = " << k << '\n';
   return 0;
}
```

Παράδειγμα (1)

```
float avg_arr(const vector<float>& v, float min, float max)
{
  int i, cnt = 0;
  float sum = 0;

  for(i = 0; i < v.size(); i++)
  {
       if(v[i] >= min && v[i] <= max)
       {
              cnt++;
              sum += v[i];
       }
  }
  if(cnt == 0)
       return -1; // Ειδική τιμή.
  else
       return sum/cnt;
}
```

Παράδειγμα (2)

- Δημιουργήστε μία συνάρτηση η οποία να δέχεται σαν παραμέτρους δύο αλφαριθμητικά και με χρήση δεικτών να επιστρέφει 0 αν είναι ίδια ή τη διαφορά των πρώτων χαρακτήρων τους που δεν είναι ίδιοι. Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει δύο αλφαριθμητικά μέχρι 100 χαρακτήρες και να εμφανίζει το αποτέλεσμα της σύγκρισής τους με χρήση της συνάρτησης

Παράδειγμα (2)

```
#include <iostream>
using std::cout;
using std::cin;

int str_cmp(const char *str1, const char *str2);

int main()
{
   char str1[100], str2[100];
   int i;

   cout << "Enter first text: ";
   cin.getline(str1, sizeof(str1));

   cout << "Enter second text: ";
   cin.getline(str2, sizeof(str2));

   i = str_cmp(str1, str2);
   if(i == 0)
          cout << str1 << " = " << str2 << '\n';
   else if(i < 0)
          cout << str1 << " < " << str2 << '\n';
   else
          cout << str1 << " > " << str2 << '\n';
   return 0;
}
int str_cmp(const char *str1, const char *str2)
{
   while(*s1 == *s2)
   {
          if(*s1 == '\0')
                    return 0;
          s1++;
          s2++;
   }
   return *s1-*s2;
}
```

Παράδειγμα (3)

- Δημιουργήστε μία συνάρτηση που να δέχεται σαν παράμετρο έναν πίνακα και να ελέγχει αν περιέχει τιμές που να επαναλαμβάνονται. Αν ναι, η συνάρτηση να επιστρέφει έναν δείκτη στο στοιχείο που επαναλαμβάνεται τις περισσότερες φορές, αλλιώς, να επιστρέφει `nullptr`. Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει 100 ακεραίους, να τους αποθηκεύει σε έναν πίνακα, να καλεί τη συνάρτηση και να χρησιμοποιεί την τιμή επιστροφής της για να εμφανίσει την τιμή του στοιχείου με τις περισσότερες εμφανίσεις. (Σημ. Αν υπάρχουν παραπάνω από ένα στοιχεία με τον ίδιο αριθμό μέγιστων εμφανίσεων, το πρόγραμμα να εμφανίζει το πρώτο από αυτά)

Παράδειγμα (3)

```
#include <iostream>

int *find(int arr[]);
const int SIZE = 100;

int main()
{
  int *ptr, i, arr[SIZE];

  for(i = 0; i < SIZE; i++)
  {
        std::cout << "Enter number: ";
        std::cin >> arr[i];
  }
  ptr = find(arr);
  if(ptr == nullptr)
        std::cout << "No duplicated
  value is found\n";
  else
        std::cout << "The number " <<
  *ptr <<  " appears the most times\n";
  return 0;
}
```

```
int *find(int arr[])
{
  int i, j, cnt, max, pos;

  max = 0;
  for(i = 0; i < SIZE; i++)
  {
        cnt = 0;
        for(j = i+1; j < SIZE; j++)
        {
              if(arr[i] == arr[j])
                    cnt++;
        }
        if(cnt > max)
        {
              max = cnt;
              pos = i;
        }
  }
  if(max == 0)
        return nullptr;
  else
        return arr+pos;
}
```

Δήλωση Συνάρτησης με Παράμετρο Διδιάστατο Πίνακα

- Ο πιο συνηθισμένος τρόπος για να δηλώσουμε μία συνάρτηση που να δέχεται σαν παράμετρο έναν διδιάστατο πίνακα είναι να γράψουμε το όνομα του πίνακα ακολουθούμενο από τις διαστάσεις του μέσα σε αγκύλες
- Για παράδειγμα, η `test()` δέχεται σαν παράμετρο έναν διδιάστατο πίνακα ακεραίων με 5 γραμμές και 10 στήλες:

```
void test(int arr[5][10]);
```

- Επειδή, όπως είδαμε στο Κ.7, ο μεταγλωττιστής δεν χρειάζεται να γνωρίζει τον αριθμό των γραμμών για να υπολογίσει τη θέση μνήμης ενός στοιχείου, μπορούμε να παραλείψουμε την τιμή της πρώτης διάστασης. Αν την δηλώσετε, ο μεταγλωττιστής θα την αγνοήσει. Π.χ: `void test(int arr[][10]);`
- Πρέπει, όμως, να δηλώσουμε τον αριθμό των στηλών. Στη γενική περίπτωση, όταν μεταβιβάζεται ένας πολυδιάστατος πίνακας σε μία συνάρτηση η πρώτη διάσταση μπορεί να παραλειφθεί. Οι άλλες πρέπει να δηλωθούν και να είναι ίδιες με αυτές στη δήλωση του πίνακα
- Όπως και στην περίπτωση του μονοδιάστατου πίνακα, για να μεταβιβάσουμε έναν διδιάστατο πίνακα σαν όρισμα συνάρτησης, γράφουμε μόνο το όνομα του πίνακα, χωρίς τις αγκύλες

Παρατηρήσεις (1)

- Αφού, όπως είπαμε, η C++ χειρίζεται έναν διδιάστατο πίνακα σαν έναν πίνακα μονοδιάστατων πινάκων, στην πραγματικότητα, ο μεταγλωττιστής μεταφράζει τον διδιάστατο πίνακα σε έναν **«δείκτη σε πίνακα»**. Επομένως, θα μπορούσαμε ισοδύναμα να γράψουμε:

```
void test(int (*arr)[10]);
```

- Επειδή οι αγκύλες `[]` έχουν μεγαλύτερη προτεραιότητα από το `*`, οι παρενθέσεις είναι απαραίτητες (αλλιώς, ο `arr` θα μεταφραστεί ως πίνακας `10` δεικτών σε ακεραίους αντί για δείκτης σε πίνακα `10` ακεραίων)

- Αν και αυτή η δήλωση είναι πιο ακριβής, προτιμώ την προηγούμενη δήλωση και να παραλείπω την πρώτη διάσταση, ώστε να φαίνεται ξεκάθαρα ότι η παράμετρος είναι διδιάστατος πίνακας

- Το γράφω πάλι, ο μεταγλωττιστής μεταφράζει τον διδιάστατο πίνακα σε δείκτη σε πίνακα, όχι σε δείκτη σε δείκτη, όπως ίσως να νομίζατε. Δηλαδή, είναι λάθος να γράψετε:

```
void test(int **arr);
```

Παρατηρήσεις (2)

- *Ας συνοψίσουμε πώς μεταφράζει ο μεταγλωττιστής ορίσματα που είναι πίνακες:*

Μονοδιάστατος: `arr[10]` μεταφράζεται σε δείκτη `*arr`

Διδιάστατος: `arr[][20]` μεταφράζεται σε δείκτη σε πίνακα `(*arr)[20]`

Πίνακας δεικτών: `*arr[50]` μεταφράζεται σε δείκτη σε δείκτη `**arr`

Παράδειγμα

- **Δημιουργήστε μία συνάρτηση που να δέχεται σαν παράμετρο έναν 3×4 διδιάστατο πίνακα και να επιστρέφει έναν μονοδιάστατο πίνακα όπου το κάθε στοιχείο του θα είναι ίσο με το άθροισμα των στοιχείων της αντίστοιχης γραμμής του διδιάστατου πίνακα, καθώς και έναν δεύτερο μονοδιάστατο πίνακα όπου το κάθε στοιχείο του θα είναι ίσο με το άθροισμα των στοιχείων της αντίστοιχης στήλης του διδιάστατου πίνακα. Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει 12 ακεραίους, να τους αποθηκεύει σε έναν διδιάστατο πίνακα 3×4 και να εμφανίζει το άθροισμα των στοιχείων της κάθε γραμμής και στήλης του με χρήση της συνάρτησης**

Παράδειγμα

```
#include <iostream>
using std::cout;
using std::cin;

const int ROWS = 3;
const int COLS = 4;

void find_sums(int arr1[][COLS], int arr2[], int arr3[]);

int main()
{
  int i, j, arr1[ROWS][COLS], arr2[ROWS], arr3[COLS];

  for(i = 0; i < ROWS; i++)
        for(j = 0; j < COLS; j++)
        {
                cout << "arr1[" << i << "][" << j << "] = ";
                cin >> arr1[i][j];
        }

  find_sums(arr1, arr2, arr3);
  for(i = 0; i < ROWS; i++)
        cout << "sum_line_" << i << " = " << arr2[i] << '\n';
  for(i = 0; i < COLS; i++)
        cout << "sum_col_" << i << " = " << arr3[i] << '\n';
  return 0;
}
```

Παράδειγμα

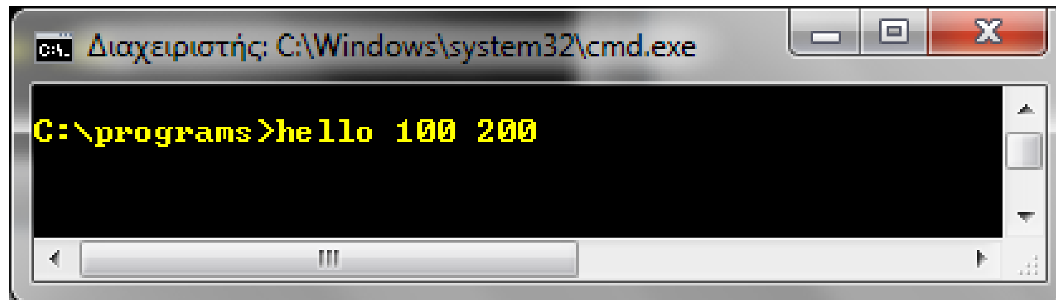
```
void find_sums(int arr1[][COLS], int arr2[], int arr3[])
{
  int i, j, sum;

  for(i = 0; i < ROWS; i++)
  {
        sum = 0;
        for(j = 0; j < COLS; j++)
              sum += arr1[i][j];
        arr2[i] = sum;
  }
  for(i = 0; i < COLS; i++)
  {
        sum = 0;
        for(j = 0; j < ROWS; j++)
              sum += arr1[j][i];
        arr3[i] = sum;
  }
}
```

Σχόλια: *Αφού μία συνάρτηση δεν μπορεί να επιστρέψει πίνακα, οι πίνακες δηλώνονται στη* `main()` *και μεταβιβάζονται στη συνάρτηση*

Διοχέτευση Πληροφορίας στη Συνάρτηση `main()` (1)

- Όταν εκτελούμε ένα πρόγραμμα από τη γραμμή εντολών (command line) μπορούμε να του μεταβιβάσουμε πληροφορία

- Για παράδειγμα, ας υποθέσουμε ότι το εκτελέσιμο αρχείο `hello.exe` είναι αποθηκευμένο στον φάκελο `C:\programs`. Τότε, αν γράψουμε στη γραμμή εντολών: `C:\programs>hello 100 200` θα εκτελεστεί το πρόγραμμα `hello` και στη `main()` θα διοχετευθούν οι τιμές `100` και `200`



Διοχέτευση Πληροφορίας στη Συνάρτηση `main()` (2)

- Όμως, για να είναι σε θέση η συνάρτηση `main()` ενός προγράμματος να πάρει τις τιμές των παραμέτρων από τη γραμμή εντολών **θα πρέπει** να έχει δηλωθεί ως:

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

Αν και μπορείτε να χρησιμοποιήσετε όποια ονόματα επιθυμείτε, η τυπική επιλογή είναι τα `argc` (*argument count*) και `argv` (*argument vector*). Και επειδή, όπως είπαμε, ο μεταγλωττιστής μεταφράζει μία παράμετρο πίνακα σε δείκτη πολλοί προτιμούν, αντί για `*argv[]`, να γράφουν `**argv`

α) Η παράμετρος `argc` είναι ένας ακέραιος που δηλώνει το πλήθος των ορισμάτων της γραμμής εντολών, συμπεριλαμβανομένου του ονόματος του προγράμματος. Για παράδειγμα, στην προηγούμενη γραμμή εντολών, η τιμή του `argc` είναι ίση με 3. Τα ορίσματα πρέπει να διαχωρίζονται μεταξύ τους με κενά, ώστε να ξεχωρίζουν οι τιμές

Διοχέτευση Πληροφορίας στη Συνάρτηση `main()` (3)

β) Η παράμετρος **argv** είναι ένας πίνακας δεικτών στα ορίσματα της γραμμής εντολών, τα οποία αποθηκεύονται σαν αλφαριθμητικά C-μορφής

Ο δείκτης **argv[0]** δείχνει στο όνομα του προγράμματος, ενώ οι υπόλοιποι δείκτες μέχρι τον **argv[argc-1]** δείχνουν στα υπόλοιπα ορίσματα. Το τελευταίο στοιχείο του πίνακα **argv** είναι το **argv[argc]**, του οποίου η τιμή είναι πάντα ίση με **nullptr**

Στο παράδειγμά μας, τα ορίσματα **hello**, **100** και **200** μεταβιβάζονται στην **main()** σαν αλφαριθμητικά. Ο δείκτης **argv[0]** δείχνει στο αλφαριθμητικό **"hello"**, ο **argv[1]** στο αλφαριθμητικό **"100"** και ο **argv[2]** στο **"200"**. Η τιμή του **argv[3]** είναι ίση με **nullptr**

Παράδειγμα

- Το επόμενο πρόγραμμα ελέγχει αν ο χρήστης εισήγαγε το σωστό όνομα χρήστη και συνθηματικό στη γραμμή εντολών. Θεωρήστε ότι αυτά είναι τα **user** και **pswd**, αντίστοιχα

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

int main(int argc, char *argv[])
{
  if(argc == 1)
        std::cout << "Error: missing user name and password\n";
  else if(argc == 2)
        std::cout << "Error: missing password\n";
  else if(argc == 3)
  {
        if(strcmp(argv[1], "user") == 0 && strcmp(argv[2], "pswd") == 0)
                std::cout << "Valid user. The program " << argv[0] << " will
  be executed ...\n";
        else
                std::cout << "Wrong input\n";
  }
  else
        std::cout << "Error: too many parameters\n";
  return 0;
}
```

- Οι **if** συνθήκες ελέγχουν την τιμή της παραμέτρου **argc**. Αν η τιμή της είναι 3, το πρόγραμμα ελέγχει αν ο χρήστης εισήγαγε έγκυρο όνομα χρήστη και συνθηματικό. Αν δεν είναι 3, το πρόγραμμα εμφανίζει ανάλογο μήνυμα

Συναρτήσεις με Μεταβλητό Αριθμό Παραμέτρων

- Μία συνάρτηση μπορεί να δεχτεί μεταβλητό αριθμό παραμέτρων

- Ένας τρόπος δήλωσης μίας τέτοιας συνάρτησης, που προέρχεται από τη C, είναι να γράψουμε πρώτα τις **σταθερές** παραμέτρους, δηλαδή, αυτές που πρέπει να υπάρχουν πάντα στην κλήση της συνάρτησης, και στο τέλος τα αποσιωπητικά `...`

- Π.χ. η συνάρτηση: `void test(int num, char *str, ...);`
  είναι μία συνάρτηση που δέχεται δύο σταθερές παραμέτρους (μία ακέραια μεταβλητή και έναν δείκτη σε χαρακτήρα) και μπορεί να ακολουθήσει ένας μεταβλητός αριθμός παραμέτρων

- Μία συνάρτηση που δέχεται έναν μεταβλητό αριθμό παραμέτρων πρέπει να έχει **τουλάχιστον μία σταθερή** παράμετρο

- Για να καλέσουμε μία τέτοια συνάρτηση, πρώτα γράφουμε τις τιμές των σταθερών ορισμάτων και μετά τις τιμές των προαιρετικών ορισμάτων. Για παράδειγμα, μία κλήση της `test()` θα μπορούσε να είναι:

```
test(3, "keimeno", 5, 8.9, "sample");
```

- **Οι σταθερές παράμετροι** αυτής έχουν τιμές `3` και `"keimeno"` αντίστοιχα

- Οι τύποι δεδομένων **των προαιρετικών ορισμάτων** αυτής της συνάρτησης είναι `int`, `double` και `char*` με τιμές `5`, `8.9` και `"sample"`, αντίστοιχα

Αναδρομικές Συναρτήσεις

- Μία συνάρτηση μπορεί να καλεί μέσα στο σώμα της οποιαδήποτε άλλη συνάρτηση, ακόμα και τον εαυτό της

- Μία συνάρτηση που μέσα στο σώμα της **καλεί τον εαυτό της** ονομάζεται **αναδρομική συνάρτηση**

- Για να εξηγήσουμε πως λειτουργεί μία αναδρομική συνάρτηση θα εξετάσουμε τη συνάρτηση `show()` του επόμενου παραδείγματος

- Η συνάρτηση `show()` βλέπουμε ότι είναι αναδρομική, αφού μέσα στο σώμα της καλεί τον εαυτό της

Παράδειγμα

```
#include <iostream>

void show(int num);

int main()
{
   int i;

   std::cout << "Enter number: ";
   std::cin >> i;
   show(i);
   return 0;
}

void show(int num)
{
   if(num > 1)
         show(num-1);

   std::cout << "val = " << num << '\n';
}
```

Επεξήγηση Παραδείγματος

- Για να δούμε πώς λειτουργεί η αναδρομή, ας υποθέσουμε ότι ο χρήστης εισάγει έναν αριθμό μεγαλύτερο από το 1, π.χ. το 3, ώστε να κληθεί πάλι η show()
  - α) Στην πρώτη κλήση της show(), αφού num = 3 > 1, η show() καλεί τον εαυτό της με όρισμα num-1 = 3-1 = 2 και η cout δεν θα εκτελεστεί. Αφού δεν τερματίζεται η εκτέλεση της show(), η μνήμη που δεσμεύτηκε για τη μεταβλητή num με τιμή 3, καθώς και πρόσθετη πληροφορία που σχετίζεται με την κλήση της συνάρτησης δεν αποδεσμεύεται
  - β) Στη δεύτερη κλήση της show() δεσμεύεται νέα μνήμη για τη num. Αφού num = 2 > 1, η show() καλεί πάλι τον εαυτό της με όρισμα num-1 = 2-1 = 1. Παρόμοια με πριν, η cout δεν θα εκτελεστεί και η μνήμη που δεσμεύτηκε για την νέα num με τιμή 2 δεν αποδεσμεύεται
  - γ) Όπως και πριν, στην τρίτη κλήση της show() δεσμεύεται νέα μνήμη για τη num. Τώρα όμως δεν θα γίνει νέα κλήση της show(), γιατί η τιμή της num δεν είναι μεγαλύτερη από το 1. Άρα, θα εκτελεστεί η cout και θα εμφανιστεί val = 1. Επίσης, αποδεσμεύεται η μνήμη για τη συγκεκριμένη num

- Στη συνέχεια, όλες οι ανεκτέλεστες cout θα εκτελεστούν διαδοχικά, ξεκινώντας από την τελευταία. Σε κάθε τερματισμό της show(), η μνήμη που είχε δεσμευτεί για την αντίστοιχη num και γενικότερα για τη συγκεκριμένη κλήση της συνάρτησης αποδεσμεύεται. Επομένως, το πρόγραμμα θα εμφανίσει:

```
val = 1
val = 2
val = 3
```

Παρατηρήσεις (1)

- Μία αναδρομική συνάρτηση πρέπει να περιέχει μία **συνθήκη τερματισμού**, αλλιώς θα εκτελείται συνεχώς με αποτέλεσμα την εξάντληση των πόρων του υπολογιστή. Στο προηγούμενο παράδειγμα, αυτή η συνθήκη ήταν η εντολή: `if(num > 1)`

- Κάθε φορά που μία αναδρομική συνάρτηση καλεί τον εαυτό της δεσμεύεται νέα μνήμη από τη στοίβα για τις αυτόματες μεταβλητές της. Για τις στατικές μεταβλητές δεν δεσμεύεται νέα μνήμη, δηλαδή, όλες οι κλήσεις διαχειρίζονται τις ίδιες στατικές μεταβλητές. Η πληροφορία για το ποιο τμήμα του κώδικα δεν εκτελέστηκε αποθηκεύεται και αυτή στη στοίβα

- Όμως, το μέγεθος της στοίβας μπορεί να μην είναι αρκετά μεγάλο για να αποθηκευτεί η πληροφορία που σχετίζεται με την κάθε κλήση της συνάρτησης. Για παράδειγμα, αν ο χρήστης εισάγει μία μεγάλη τιμή, π.χ. `100000`, είναι πολύ πιθανό να τερματιστεί η εκτέλεση του προγράμματος και να εμφανιστεί το μήνυμα "Stack overflow", που σημαίνει ότι δεν υπάρχει άλλος διαθέσιμος χώρος στη στοίβα

- Να προσέχετε όταν χρησιμοποιείτε μία αναδρομική συνάρτηση, γιατί, αν καλεί πολλές φορές τον εαυτό της, ο χρόνος εκτέλεσής της μπορεί να γίνει υπερβολικά μεγάλος, καθώς και οι διαδοχικές κλήσεις της μπορεί να οδηγήσουν στην εξάντληση της μνήμης της στοίβας

Παρατηρήσεις (2)

- Συνήθως, η αναδρομή χρησιμοποιείται στην ανάπτυξη μαθηματικών αλγορίθμων ή σε λειτουργίες σε δομές δεδομένων
- Γενικά, όταν μία επιμέρους εργασία είναι μία μικρότερη έκδοση της αρχικής εργασίας που πρέπει να πραγματοποιηθεί είναι υποψήφια να υλοποιηθεί με χρήση αναδρομικής συνάρτησης
- Ωστόσο, αν μπορείτε να γράψετε ισοδύναμο κώδικα χωρίς μεγάλη δυσκολία, είναι μάλλον καλύτερη ιδέα. Για παράδειγμα, να χρησιμοποιήσετε στη θέση της κάποιον επαναληπτικό βρόχο
- Αν και ο αναδρομικός κώδικας μπορεί να διαβάζεται και να γράφεται πιο εύκολα, η απόδοση του βρόχου μάλλον θα είναι καλύτερη αφού αποφεύγονται οι διαδοχικές κλήσεις της συνάρτησης και οι δεσμεύσεις μνήμης στη στοίβα

Παράδειγμα (1)

- Ποια είναι η έξοδος του παρακάτω προγράμματος;

```
#include <iostream>

int unknown(int arr[], int num);

int main()
{
  int arr[] = {10, 20, 30, 40};
  std::cout << unknown(arr, 4) << '\n';
  return 0;
}

int unknown(int arr[], int num)
{
  if(num == 1)
        return arr[0];
  else
        return arr[num-1] + unknown(arr, num-1);
}
```

Παράδειγμα (1)

- Απάντηση: Όταν καλείται η unknown() επιστρέφει:

  arr[4-1 = 3] + unknown(arr, 4-1 = 3) =

  arr[3] + (arr[3-1] + unknown(arr, 3-1)) =

  arr[3] + arr[2] + (arr[2-1] + unknown(arr, 2-1)) =

  arr[3] + arr[2] + arr[1] + unknown(arr, 1)

  Η τελευταία κλήση της unknown(arr, 1) επιστρέφει arr[0], γιατί num = 1. Επομένως, η τελική τιμή επιστροφής είναι: arr[3]+arr[2]+arr[1]+arr[0], δηλαδή, η συνάρτηση υπολογίζει με αναδρομικό τρόπο το άθροισμα των στοιχείων ενός πίνακα και το πρόγραμμα εμφανίζει 100.

Παράδειγμα (2)

- Δημιουργήστε μία αναδρομική συνάρτηση που να δέχεται σαν παράμετρο έναν ακέραιο αριθμό **n** και να επιστρέφει το παραγοντικό του, χρησιμοποιώντας τον τύπο **n! = n*(n-1)!**. Το παραγοντικό ενός ακεραίου **n**, όπου **n >= 1**, είναι το γινόμενο των ακεραίων από το **1** μέχρι και το **n**, δηλαδή **1*2*3*...*n**. Το παραγοντικό του **0** είναι **1** (**0!=1**). Επειδή τα παραγοντικά αριθμών αυξάνονται πολύ γρήγορα και μπορεί να ξεπεραστεί το εύρος του μεγαλύτερου ακεραίου τύπου μην εισάγετε μεγάλο ακέραιο

Παράδειγμα (2)

```
#include <iostream>

unsigned long long int fact(int num);

int main()
{
  int num;
  do
  {
        std::cout << "Enter a positive integer: ";
        std::cin >> num;
  } while(num < 0);

  std::cout << "Factorial of " << num << " is " << fact(num) << '\n';
  return 0;
}
unsigned long long int fact(int num)
{
  if((num == 0) || (num == 1))
        return 1;
  else
        return num * fact(num-1);
}
```

- Σχόλια: Παρατηρήστε ότι για μεγάλες τιμές της `num` οι κλήσεις της `fact()` αυξάνουν σημαντικά και επομένως και ο χρόνος υπολογισμού της τελικής τιμής. Σε αυτή την περίπτωση, η λύση με τον `for` βρόχο στην αντίστοιχη άσκηση του βιβλίου στο Κ.6 υπολογίζει το παραγοντικό του αριθμού πιο γρήγορα