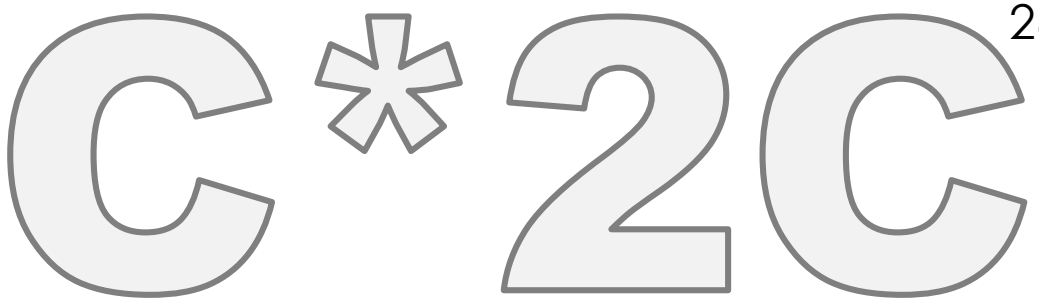
Εισαγωγή στην Πληροφορική & στον Προγραμματισμό

Αρχές Προγραμματισμού Η/Υ (με τη γλώσσα C)

Διάλεξη #7

26 Απριλίου 2024



Παναγιώτης Παύλου

c-programming-24@allos.gr

Από την $\mathbb{C}^*$ στην $\mathbb{C}$

Αξιοποιώντας τη θεωρία…

Παραδείγματα και ένα πρόβλημα…

```c
#include <stdio.h>

void printOranges(int oranges) {
    printf("Exw %d portokalia\n",
           oranges);
    return;
}

int main() {
    printOranges(5);
    return 0;
}
```

```c
#include <stdio.h>
#include <math.h>

double ypoteinousa(double a, double b) {
    return sqrt( a*a + b*b );
}

void proveUnitCycle() {
    double f = M_PI / 4;
    printf("%lf\n", ypoteinousa(
                        sin(f),
                        cos(f)
                    )
           );
}

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

Δηλώσεις συναρτήσεων

Όπως φαίνεται και στα προηγούμενα παραδείγματα είναι «παράλογη» η τοποθέτηση των συναρτήσεων αφού πρώτα εμφανίζονται οι πιο «εξειδικευμένες» συναρτήσεις και μετά οι γενικότερες και στο τέλος η… αρχική (η main)!

Επίσης σε περίπτωση όπου δύο συναρτήσεις χρειάζονται η μία να καλέσει την άλλη για να ολοκληρωθούν, τότε δεν γίνεται να γραφτούν και οι δύο… πρώτες!

Γι'αυτό τον λόγο υπάρχει και η δυνατότητα απλής **δήλωσης** (**declaration**) των συναρτήσεων, όπου γίνεται περιγραφή της συνάρτησης χωρίς να δίνεται ο κώδικάς της. Η δήλωση γίνεται όπως ο ορισμός, αλλά χωρίς το σώμα των εντολών ενώ αντί για άγκιστρα απλά μπαίνει ο τερματισμός εντολής `;` π.χ.

```
double inverse(double x);
```

Επίσης το όνομα των παραμέτρων μπορεί να παραληφθεί.

Μετά από τη δήλωση της, μια συνάρτηση μπορεί να κληθεί οπουδήποτε. Θα πρέπει όμως σε οποιοδήποτε σημείο του κώδικα από εκεί και κάτω να δοθεί και ο ορισμός.

Παράδειγμα

```c
#include <stdio.h>
#include <math.h>

double ypoteinousa(
        double a, double b
) {
    return sqrt( a*a + b*b );
}

void proveUnitCycle() {
    double f = M_PI / 4;
    printf("%lf\n",
        ypoteinousa(
            sin(f),
            cos(f)
        )
    );
}

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

```c
#include <stdio.h>
#include <math.h>

void proveUnitCycle();
double ypoteinousa(double a, double b);

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

void proveUnitCycle() {
    double f = M_PI / 4;
    printf("%lf\n", ypoteinousa( sin(f), cos(f) ));
}

double ypoteinousa(double a, double b) {
    return sqrt( a*a + b*b );
}
```

Άλλες διαφορές

Σε όσα είδαμε μέχρι εδώ υπάρχουν και οι παρακάτω «κρυφές» διαφορές:

- Οι **μη αρχικοποιημένες μεταβλητές** δεν έχουν πλέον την ειδική τιμή undefined, αλλά κάποια «τυχαία» τιμή, οπότε και πάλι πρέπει να τις αρχικοποιούμε.
- Επειδή οι τύποι δεδομένων είναι συγκεκριμένοι, πλέον είναι πιο εύκολο να προκύψει **υπερχείλιση** και πλέον δεν υπάρχει κάποια ειδική τιμή (όπως η Infinity) που να την υποδεικνύει.
- Ο τελεστής % εφαρμόζεται μόνο σε ακέραιες μεταβλητές, ενώ για τις πραγματικές υπάρχει η συνάρτηση fmod.

Σταθερές 1: Preprocessor (`#define`)

Η **`#define`** αντικαθιστά ένα λεκτικό με την παράσταση που το ακολουθεί σε όλη την έκταση του κώδικα από την ψευδοεντολή και κάτω, ακριβώς όπως ένας editor θα έκανε find/replace (εύρεση/αντικατάσταση). Π.χ.

```
#define THIS_YEAR 2021
```

Το όνομα (identifier) συνηθίζεται να γράφεται με κεφαλαίους χαρακτήρες.

Επίσης παρέχεται και μία άλλη παραλλαγή όπου το όνομα ακολουθείται από παρενθέσεις με παραμέτρους όπως οι συναρτήσεις. Αυτά τα «κατασκευάσματα» ονομάζονται macros ή μακροεντολές και δεν είναι συναρτήσεις. Π.χ.

```
#define PRODUCT(x,y) ((x)*(y))
```

Σε οποιοδήποτε σημείο του κώδικα πλέον γραφεί `PRODUCT(...,...)` τότε γίνεται η παραπάνω αντικατάσταση. Όμως με τα macros αυτά **δεν** θα ασχοληθούμε

Σταθερές 2: Λέξη κλειδί const

Η **const** , λειτουργεί όπως και στην C* με τη διαφορά ότι ανάμεσα σε αυτή και το όνομα της μεταβλητής γράφεται και ο τύπος δεδομένων. Π.χ.

```
const double MY_PI = 3.14;
```

Ουσιαστικά η **const** δεν παράγει σταθερές, αλλά μεταβλητές που δεν μπορεί να αλλαχθεί η τιμή τους (read only). Δηλαδή μία **const** αποθηκεύεται στη μνήμη του υπολογιστή, ενώ η προηγούμενη μέθοδος **(#define**) απλά αντικαθιστά τις τιμές μέσα στο κείμενο του κώδικα πριν από το build, οπότε είναι πιο αποδοτικός ο κώδικας.

Προσοχή! Η λέξη κλειδί **const** δεν είναι διαθέσιμη σε παλαιότερες εκδόσεις της C.

Και για τους δύο παραπάνω λόγους, **προτείνεται η χρήση της #define όπου είναι δυνατόν**. Σε πιο προχωρημένα θέματα, μπορεί να ταιριάζει η **const** καλύτερα από την **#define**.

Παράδειγμα

```
#include <stdio.h>

#define BEGIN {
#define END }

#define ONEpTHREE 1+3

#define PROD(x,y) ((x)*(y))

int main()
BEGIN
    printf("Result: %d\n", PROD(ONEpTHREE, 2+4));
    // Προσοχή! Η παραπάνω παράσταση γίνεται ((1+3)*(2+4)) πριν το build
    return 0;
END
```

Λογικές ποσότητες

Οι επεξεργαστές επεξεργάζονται την πληροφορία πάντα σε ομάδες από bits.

Όμως μια boolean μεταβλητή ουσιαστικά χρειάζεται μόνο ένα bit για την αποθήκεσή της. Έτσι οι επεξεργαστές δεν έχουν εντολές (της γλώσσας μηχανής) που να καλύπτουν τέτοιες λειτουργίες σε όλο το απαιτούμενο εύρος δυνατοτήτων.

Γι'αυτό και η C δεν έχει εγγενή υποστήριξη τύπου δεδομένων bool.

Έτσι στην C τυπικά το αληθές αντιστοιχεί στον αριθμό **1** και το ψευδές στο **0**

Γι'αυτό οι όροι **αληθές**, **true** και **1** θα χρησιμοποιούνται εναλλάξιμα και ομοίως οι όροι **ψευδές**, **false** και **0**.

Στην πράξη όμως ο επεξεργαστής θεωρεί το `0` ψευδές και κάθε τι μη μηδενικό αληθές.

Υπάρχουν δύο προσεγγίσεις στην υποστήριξη λογικών ποσοτήτων στη C:

1. Χρήση του τύπου δεδομένων `bool` και τις τιμές `true` ή `false` βάσει του νεότερου προτύπου της C (το C99) το οποίο απαιτεί να γίνει include το αρχείο stdbool.h **(προτείνεται)**
2. Ορισμός των `true` και `false` σε `1` και `0` αντίστοιχα με `#define` και χρήση ακέραιου τύπου δεδομένων πχ του unsinged char που γίνεται `#define` ως `bool` (λειτουργεί ανεξαρτήτως έκδοσης της C)

Ανεξαρτήτως της προσέγγισης το αποτέλεσμα είναι το ίδιο. Υπάρχουν λογικές μεταβλητές διαθέσιμες για τον προγραμματιστή.

Παράδειγμα

```
#include <stdbool.h>

int main() {
    bool isRed = true;
    bool isBad = false;
    return 0;
}
```

```
#define false 0
#define true 1
#define bool unsigned char

int main() {
    bool isRed = true;
    bool isBad = false;
    return 0;
}
```

Ο τριαδικός τελεστής ?:

Κάποιες φορές χρειάζεται απόφαση για μία τιμή που θα χρησιμοποιηθεί σε κάποια παράσταση, οπότε δεν συμφέρει να γραφτεί μία if και θα βόλευε κάπως το if να γραφεί «μέσα» στην παράσταση. Π.χ.

```
if (x > 100) {
    hlp = 2.1;
} else {
    hlp = 3.85;
}
a = hlp * sqrt(x);
```

```
a = (x>100?2.1:3.85)
    * sqrt(x);
```

```
if (x > 0) {
    hlp = x;
} else {
    hlp = -x;
}
a = hlp * M_PI;
```

```
a = (x>0?x:-x)*M_PI;
```

```
if (x > 10) {
    hlp = x;
} else {
    hlp = 0;
}
a = hlp + x;
```

```
a = (x>10?x:0)+x;
```

Ο τριαδικός τελεστής δέχεται μία συνθήκη, την τιμή που θα αντικαταστήσει την παράσταση εάν η συνθήκη είναι αληθής και την τιμή που την αντικαταστήσει εάν η συνθήκη είναι ψευδής.

Στατικές μεταβλητές

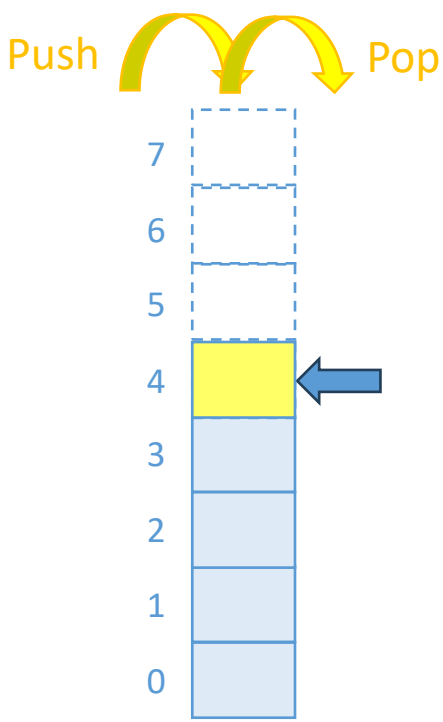
Μία νέα εμβέλεια που μας παρέχει η C είναι η δήλωση μεταβλητής ως στατική με τη λέξη κλειδί `static`. Η επίδραση της «διευκρίνισης» static εξαρτάται από το αν η μεταβλητή είναι global ή local.

Για τις τοπικές μεταβλητές η επίδραση είναι η εξής: Μετά το τέλος του block στο οποίο είναι δηλωμένη η μεταβλητή, αυτή δεν διαγράφεται από τη μνήμη, παρότι δεν είναι προσβάσιμη από τον κώδικα. Έτσι όταν η εκτέλεση του κώδικα επιστρέψει στο ίδιο block η μεταβλητή είναι και πάλι προσβάσιμη και περιέχει την προηγούμενη τιμή της. Φυσικά όταν δίνεται αρχική τιμή σε αυτή κατά τη δήλωσή της, η τιμή αποδίδεται μόνο στην πρώτη εκτέλεση της συνάρτησης (ή του block).

Για τις global μεταβλητές η δήλωσή της ως static, απλά περιορίζει την εμβέλειά τους στο αρχείο στο οποίο δηλώνονται, κάνοντάς τες ασφαλείς ως προς «συνωνυμίες» με global μεταβλητές άλλων αρχείων.

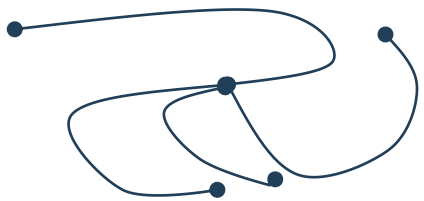
Stack

Το **Stack** είναι μία δομή δεδομένων η οποία λειτουργεί όπως μια στοίβα (το λέει και το όνομά του). Ένα stack έχει σκοπό να στοιβάζει δεδομένα και να τα επιστρέφει με την αντίστροφη σειρά που δόθηκαν (δομή τύπου LIFO). Η «τοποθέτηση» ενός δεδομένου ονομάζεται **push** και η ανάκτησή του **pop**.



Ένας τρόπος να υλοποιηθεί ένα stack είναι με τη βοήθεια ενός πίνακα και μιας μεταβλητής η οποία θα μας ενημερώνει πόσα «δεδομένα» έχουν τοποθετηθεί στο stack.

Παράδειγμα εφαρμογής είναι μια μηχανή CNC η οποία ξεκινά μία εργασία και πρέπει να κατεργαστεί μία περιοχή στην οποία δεν την καλύπτει «μονοκονδυλιά», αλλά την καλύπτει σε κλάδους όπως στο σχήμα.



Κάθε φορά τοποθετεί το σημείο της πιο πρόσφατης διακλάδωσης ώστε να επιστρέψει σε αυτό για να πάρει τον επόμενο κλάδο της διαδρομής.

Πρακτικά ζητήματα

smProject – Γιατί σας βοηθά να παραδώσετε σωστότερο αποτέλεσμα

Χρήση smProject

Η κάθε εργασία, εκτός από την εκφώνησή της, θα συνοδεύεται και από ένα CLion project το οποίο έχει σαν σκοπό να σας βοηθά να παραδίδετε πιο σωστές εργασίες, επειδή:

1. Περιλαμβάνει κάποιες δοκιμές για κάποια συνηθισμένα σφάλματα πάνω στα ζητούμενα. Έτσι θα έχετε την ευκαιρία να δείτε μόνοι σας εάν ο κώδικάς σας έχει κάποιο τέτοιο σφάλμα.
   Η λίστα των ελέγχων είναι ενδεικτική και όχι εξαντλητική, έτσι εάν πετύχουν όλοι οι έλεγχοι, πιθανώς να εξακολουθούν να υπάρχουν αρκετά και σοβαρά λογικά σφάλματα στον κώδικά σας. Και τα σταματημένα ρολόγια 2 φορές τη μέρα δείχνουν τη σωστή ώρα.
2. Έχει έτοιμες τις δηλώσεις των συναρτήσεων που ζητούνται, οπότε έχετε μία ευκολία παραπάνω. Σωστό όνομα συνάρτησης, σωστές παραμέτρους και τύπο δεδομένων του αποτελέσματος.
3. Έχει οριοθετημένο τον κώδικα με ένα σχόλιο στην αρχή και ένα στο τέλος. Εφόσον εσείς γράψετε όλο τον κώδικά σας ανάμεσα σε αυτά τα σχόλια, τότε σε συνδυασμό με το αναβαθμισμένο σύστημα υποβολής, εάν δεν έχετε αντιγράψει ολόκληρο τον κώδικα θα εμφανιστεί μία ειδοποίηση γι'αυτό.

Από τα παραπάνω είναι προφανές ότι για κάθε άσκηση θα πρέπει να κατεβάζετε το αντίστοιχο smProject αυτής της άσκησης και σε αυτό το project να γράφετε τον κώδικα.

Για να είναι δυνατή η λειτουργία του smProject, χρειάζεται η παραδοχή ότι αντί της `main`, η κύρια συνάρτηση του προγράμματος θα είναι η `smMain` με την ίδια ακριβώς σύνταξη που έχει η `main`.

Στην επόμενη διαφάνεια θα δείτε πως, όταν ανοίξετε το smProject στο CLion, θα επιλέγετε την εκτέλεση της κανονικής λειτουργίας ή των δοκιμαστικών ελέγχων.

Κανονική εκτέλεση και δοκιμές

Στην γραμμή εκτέλεσης εμφανίζονται πλέον δύο **στόχοι**.

- **smProject** , που αφορά την κανονική εκτέλεση του κώδικα
- **RunTests** , που αφορά την εκτέλεση των δοκιμών

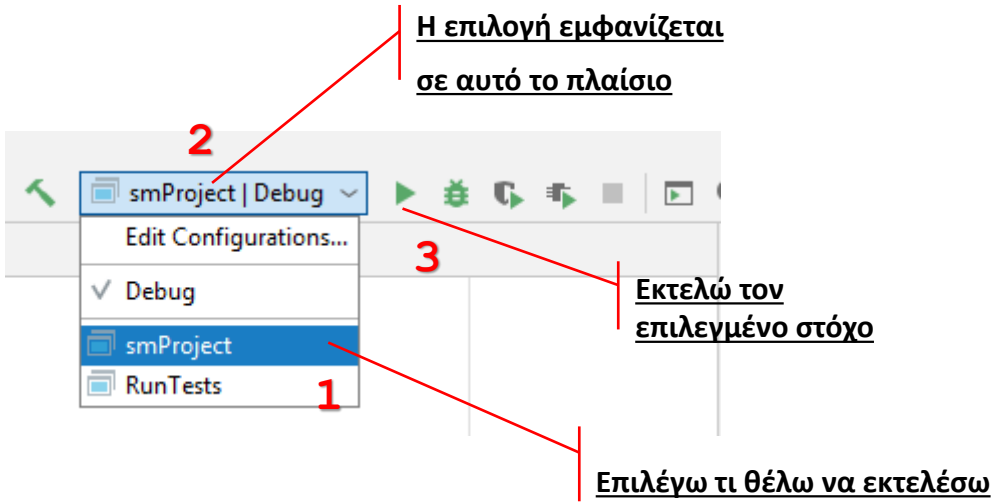
Θα πρέπει λοιπόν ο χρήστης, ανάλογα με την επιθυμία του, να επιλέξει τον αντίστοιχο στόχο.

Αφού έχει γίνει η επιλογή αυτή τα υπόλοιπα κουμπιά της γραμμής λειτουργούν κατά τα γνωστά.

Ως συνήθης πρακτική προτείνεται να γίνονται τα πάντα σε κανονική εκτέλεση μέχρι να θεωρήσει ο προγραμματιστής ότι ο κώδικάς είναι έτοιμος.

Κατόπιν να γίνεται αλλαγή του στόχου σε RunTests για να βεβαιωθεί ότι όλα εκτελούνται καλά.

Ενδεικτικό αποτέλεσμα ελέγχων φαίνεται δίπλα. *Το **ζητούμενο** είναι να μην υπάρχει η λέξη FAILED, να εμφανίζεται μόνο το Ok και η επιστρεφόμενη τιμή να είναι 0.*




```
TESTING MODE!


Test ARC_NORMAL_VALUE_TESTS :
  VALUE_1_2 FAILED : arc(1.2) returns unexpected result!
ARC_NORMAL_VALUE_TESTS FAILED!
Test aktina_0 : Ok
Test aktina_PI : Ok
Process finished with exit code 1
```

Ερωτήσεις?

- Διαβάστε τις σημειώσεις, διαβάστε τις διαφάνειες και δείτε τα videos **πριν** ρωτήσετε
- **Συμβουλευτείτε** τη σελίδα ερωταποκρίσεων του μαθήματος

  https://qna.c-programming.allos.gr
- **Στείλτε** τις ερωτήσεις σας πριν και μετά το μάθημα στο

  c-programming-24@allos.gr

- Εάν έχετε **πρόβλημα** με κάποιο κώδικα στείλτε τον κώδικα ως κείμενο με copy/paste. Εάν θεωρείτε ότι επιπλέον βοηθά και ένα στιγμιότυπο οθόνης, είναι καλοδεχούμενο.
- Επαναλαμβάνουμε : Μην στείλετε ποτέ κώδικα ως εικόνα μας είναι παντελώς άχρηστος!


Καλή Ανάσταση!
Καλό Πάσχα!