HY255 Εργαστήριο Λογισμικού - L07

(a) Writing Large Programs
(b) Makefiles

HY255 Εργαστήριο Λογισμικού
Άνοιξη 2021
Άγγελος Μπίλας

Ο κώδικας τείνει να αυξάνεται…

- Σήμερα, πολλά και πολύ χρήσιμα συστήματα περιλαμβάνουν πολύ κώδικα
- Πως οργανώνουμε τον κώδικα σε αρχεία;
- Αν βάλουμε όλο τον κώδικα σε ένα αρχείο
  - Είναι εξαιρετικά αργό το compilation
  - Οποιαδήποτε, έστω και μικρή αλλαγή, χρειάζεται να γίνει recompile όλος ο κώδικας
  - Αν θέλουμε να κάνουμε μια αλλαγή, δύσκολο να βρούμε το κατάλληλο σημείο σε ένα τεράστιο αρχείο
  - Δύσκολο να επανα-χρησιμοποιήσουμε τμήματα του κώδικα
- Οπότε πρέπει και συνήθως χωρίζουμε τον κώδικά μας σε πολλαπλά αρχεία
- Πως το κάνουμε αυτό;
- Ας θυμηθούμε ότι στις πιο πολλές γλώσσες: Πριν από κάθε χρήση ενός ονόματος πρέπει να υπάρχει μια δήλωση του ονόματος
- Τι συμβαίνει λοιπόν αν ένα όνομα πρέπει να χρησιμοποιηθεί σε περισσότερα από ένα σημεία (αρχεία) του προγράμματος;
  - Ας εξετάσουμε πρώτα τις συναρτήσεις και μετά τις μεταβλητές

Συναρτήσεις: <u>Δηλώσεις</u> και <u>Ορισμοί</u>

- Τι συμβαίνει αν πρέπει να χρησιμοποιήσουμε μια συνάρτηση σε δύο αρχεία f1.c και f2.c. Πιθανώς, έχουμε δύο επιλογές:
  - Αν τοποθετήσουμε την συνάρτηση μόνο στο αρχείο f1.c τότε αν την καλέσουμε στο f2.c θα μας πει ο compiler ότι δεν γνωρίζει το όνομα της συνάρτησης. (Γιατί ο compiler παραπονιέται αν δεν έχει δει την συνάρτηση πριν από μια κλήση;)
  - Αν τοποθετήσουμε την συνάρτηση και στα δύο αρχεία, κάποια στιγμή κατά την διαδικασία της μετάφρασης (στο linking) θα μας πει (ο linker) ότι έχουμε δύο συναρτήσεις με το ίδιο όνομα
- Πρέπει λοιπόν να έχουμε την δυνατότητα να γράψουμε την συνάρτηση μόνο μια φορά, σε ένα αρχείο και στο άλλο αρχείο να δηλώσουμε την ύπαρξή της, χωρίς να γράψουμε πάλι τον κώδικά της
- Για αυτό τον λόγο, υπάρχουν τα <u>prototypes</u> των συναρτήσεων – Ένα prototype μας λέει ποιες είναι οι παράμετροι και η τιμή επιστροφής μιας συνάρτησης, αλλά δεν περιλαμβάνει τον κώδικα της συνάρτησης. Επομένως το prototype μας λέει πως χρησιμοποιείται η συνάρτηση, αλλά όχι τι κάνει όταν εκτελείται.
- Έστω ότι
  - <u>Δήλωση</u> μιας συνάρτησης είναι ο τρόπος που μπορεί να χρησιμοποιηθεί
  - <u>Ορισμός</u> μιας συνάρτησης είναι το τι κάνει ο κώδικάς της (το πως εκτελείται)
- Τότε
  - Ο κώδικας της συνάρτησης αποτελεί <u>δήλωση</u> και ορισμό της συνάρτησης
  - Το prototype αποτελεί μόνο <u>δήλωση</u> (και όχι ορισμό)
- Οπότε
  - (1) Σε κάθε αρχείο που πρέπει να καλέσουμε μια συνάρτηση, φροντίζουμε πριν την πρώτη κλήση να υπάρχει μια <u>δήλωση</u> (prototype) στο αρχείο (άρα και στο f1.c και στο f2.c)
  - (2) Σε ένα μόνο αρχείο πρέπει να υπάρχει ο <u>ορισμός</u> της συνάρτησης (το σώμα με τον κώδικα), άρα είτε στο f1.c είτε στο f2.c (θα δούμε παρακάτω πιο προτιμούμε)

Μεταβλητές: Δηλώσεις και Ορισμοί

- Αφού εξετάσαμε την περίπτωση των συναρτήσεων, το ίδιο ακριβώς ισχύει και για τις μεταβλητές, αλλά με λίγο διαφορετικούς μηχανισμούς
- Τι συμβαίνει λοιπόν αν θέλουμε να χρησιμοποιήσουμε την ίδια μεταβλητή x σε δύο αρχεία f1.c και f2.c του προγράμματος μας; Πιθανώς έχουμε πάλι τις ίδιες δύο επιλογές:
  - Αν δηλώσουμε το x στο f1.c και το χρησιμοποιήσουμε στο f2.c, ο compiler θα μας πει στο f2.c ότι είναι άγνωστο το x
  - Αν δηλώσουμε την μεταβλητή και στα δύο αρχεία, σε επόμενο στάδιο της μετάφρασης (στο linking) θα μας πει ότι έχουμε δύο αντικείμενα με το ίδιο όνομα x
  - Αυτό συμβαίνει γιατί η δήλωση μια μεταβλητής κρατάει χώρο στη μνήμη. Άρα σε αυτή την περίπτωση το θα κρατήσουμε κάποιο χώρο κατά την μετάφραση του f1.c και θα του δώσουμε το όνομα x και μετά κατά την μετάφραση του f2.c θα κάνουμε το ίδιο και τελικά ο linker δεν θα γνωρίζει ποια διεύθυνση να χρησιμοποιήσει όταν παράγει το εκτελέσιμο a.out.
- Μπορούμε λοιπόν να πούμε ότι το πρόβλημα που έχουμε είναι ότι οι δηλώσεις μεταβλητών όπως τις γνωρίζουμε κάνουν δύο δουλειές:
  - Δήλωση: Μας λένε πως να χρησιμοποιήσουμε τις μεταβλητή (τι πράξεις μπορούμε να κάνουμε)
  - Ορισμός: Κρατούν χώρο για την μεταβλητή στη μνήμη
- Για να διαχωρίσουμε αυτές τις δύο δουλειές υπάρχει στη C ένα νέο keyword, το extern
  - int x; → Δηλώνει και ορίζει την μεταβλητή x
  - extern int x; → Δηλώνει μόνο (χωρίς να ορίζει) την μεταβλητή x
- Μπορούμε πλέον να εφαρμόσουμε τον ίδιο κανόνα με τις συναρτήσεις:
  - (1) Σε κάθε αρχείο που πρέπει να χρησιμοποιήσουμε μια μεταβλητή, φροντίζουμε πριν την πρώτη κλήση να υπάρχει μια δήλωση (extern int x) στο αρχείο (άρα και στο f1.c και στο f2.c)
  - (2) Σε ένα μόνο αρχείο πρέπει να υπάρχει ο ορισμός της μεταβλητής (int x), άρα είτε στο f1.c είτε στο f2.c (θα δούμε παρακάτω πιο προτιμούμε)

Παρατηρήσεις

- Τι γίνεται αν για κάποιο x έχουμε δύο “extern int x” και κανένα ”int x”
  - Ο compiler θα γνωρίζει πως να χρησιμοποιήσει την μεταβλητή, αλλά δεν θα δεσμεύσεις χώρο στη μνήμη (αφού δεν υπάρχει ορισμός). Οπότε ο linker δεν θα μπορεί να παράγει το a.out
- Έχει νόημα το extern για local variables;
  - Όχι ιδιαίτερα… Το ίδιο local int x σε περισσότερες από μια συναρτήσεις αναφέρεται σε διαφορετικές μεταβλητές.
- Τι συμβαίνει αν θέλουμε να κάνουμε define το ίδιο όνομα σε περισσότερα από ένα αρχεία;
  - Το #define ισχύει μέχρι τέλος του τρέχοντος αρχείου
  - Άρα θα πρέπει κάθε αρχείο να περιέχει το δικό του #define για το ίδιο όνομα
- Μέχρι τώρα δεν αναφερθήκαμε σε include (.h) αρχεία – Το πρόβλημα (και η λύση) του χωρίσματος ενός προγράμματος σε πολλά αρχεία δεν σχετίζεται με την χρήση (ή όχι) των .h αρχείων. Όλα όσα είπαμε αναφέρονται σε .c αρχεία που περιέχουν τον εκτελέσιμο κώδικα του προγράμματος.
  - Τα header .h files τα χρησιμοποιούμε απλά για να μην γράφουμε τα ίδια πράγματα πολλές φορές, οπότε είναι ”βοήθεια” όχι “απαίτηση”

Συνολικά: Δηλώσεις και Ορισμοί

- Οι συναρτήσεις και οι μεταβλητές έχουν δηλώσεις και ορισμούς
- Πριν από κάθε χρήση πρέπει να υπάρχει μια δήλωση
- Σε όλο το πρόγραμμά μας πρέπει να υπάρχει ένας μόνο ορισμός

Εξάσκηση

- Έστω το πρόγραμμα f.c

```
int x;
void fa(int a){ x = a; return; }
void fb(int b){ fa(0); x=b; return; }
int main(void){ fa(0); fb(0); return; }
```

- Χωρίστε το πρόγραμμα αυτό σε 2 αρχεία f1.c και f2.c ώστε το f1.c να περιέχει την συνάρτηση fa και το f2.c την fb
- Βήματα:
  - Εξετάζουμε <u>κάθε</u> όνομα
  - Φροντίζουμε να έχουμε μια δήλωση πριν από την πρώτη χρήση σε κάθε αρχείο
  - Προσθέτουμε έναν ορισμό σε ένα αρχείο

Χρήση include (.h) files

- Αν τώρα σε κάθε .c αρχείο του προγράμματός μας βάζουμε δηλώσεις για τις μεταβλητές και συναρτήσεις που χρειαζόμαστε, τότε θα καταλήξουμε στην αρχή του αρχείου να έχουμε μια μεγάλη λίστα από δηλώσεις
- Αυτό κάνει το πρόγραμμα δυσνόητο και δημιουργεί ευκαιρίες για λάθη, αφού π.χ. μια αλλαγή σε έναν τύπο μιας μεταβλητής πρέπει να γίνει σε πολλά σημεία (σε όλα τα σημεία που υπάρχει δήλωση)
- Αντί αυτού λοιπόν, προτιμούμε να βάζουμε τις <u>δηλώσεις</u> (όχι τους ορισμούς) σε ένα χωριστό αρχείο που κατά σύμβαση έχει κατάληξη .h
- Στη συνέχεια μπορούμε να κάνουμε αυτό το .h αρχείο include όπου χρειάζεται, ώστε να μην γράφουμε μόνοι μας όλες τις δηλώσεις
  - Αν π.χ. αλλάξει μια δήλωση μεταβλητής, πρέπει να αλλάξουμε μόνο το αρχείο .h και τον ένα ορισμό της μεταβλητής
- Μπορούμε σε ένα .h αρχείο να βάλουμε <u>ορισμούς</u>;
  - Όχι: Αν γίνει το .h include σε δύο αρχεία του προγράμματος, το πρόγραμμα μας δεν θα κάνει compile (build)
  - Κατά συνέπεια, ένα .h αρχείο δεν έχει νόημα να περιέχει τον ορισμό (το σώμα) μιας συνάρτησης
- Η χρήση λοιπόν των header files δεν είναι απαραίτητη στην γλώσσα C αλλά είναι ένα βολικό εργαλείο/μηχανισμός για να δομούμε σωστά και με ευκολο-νόητο τρόπο τα προγράμματά μας

Χωρισμός ενός προγράμματος σε αρχεία

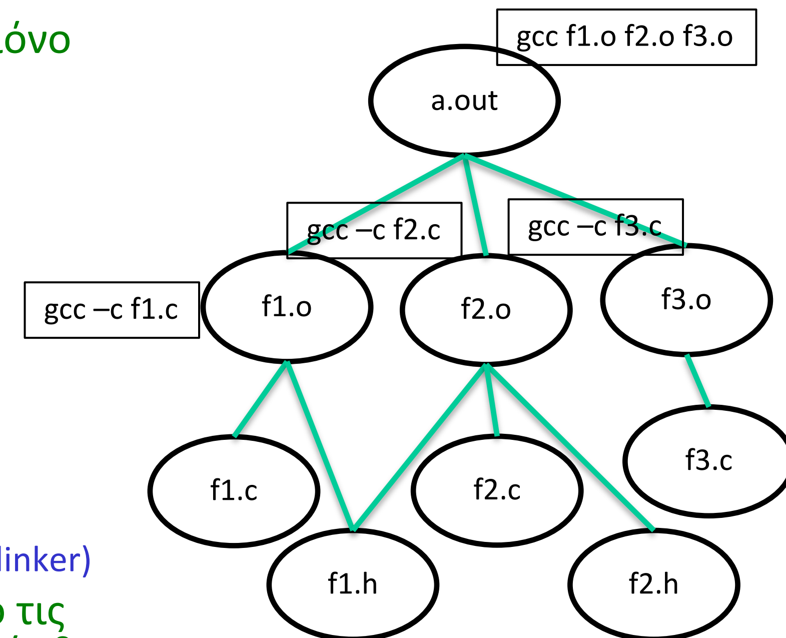
- Οι κανόνες που μπορούμε να εφαρμόσουμε είναι σχετικά απλοί:
- (1) Βάζουμε κάθε σύνολο από σχετικές συναρτήσεις και μεταβλητές στο ίδιο .c αρχείο
- (2) Δημιουργούμε ένα header file που περιέχει μόνο δηλώσεις για τις συναρτήσεις/μεταβλητές/#defines που μπορεί να χρησιμοποιηθούν σε άλλα αρχεία
  - Ενδεχομένως αντί για ένα .h αρχείο για ένα .c να χρειαστούμε περισσότερα ή ένα .h για περισσότερα .c αρχεία
- (3) Κάνουμε include το .h στα .c αρχεία που χρειάζονται τις αντίστοιχες μεταβλητές και συναρτήσεις
- (4) Βάζουμε την συνάρτηση main μόνη της σε ένα αρχείο που του δίνουμε το όνομα του προγράμματος. Στο αρχείο αυτό βάζουμε άλλες συναρτήσεις μόνο αν δεν χρησιμοποιούνται κάπου αλλού

Makefiles

- Έστω ότι έχουμε ένα πρόγραμμα που αποτελείται από πολλά αρχεία
- Αν κάθε φορά που παράγουμε το εκτελέσιμο πρέπει να κάνουμε compile όλα τα αρχεία, είναι πολύ ακριβό
  - Μεγάλα συστήματα όπως το Linux μπορεί να χρειάζονται ώρες για ένα full compilation
- Χρειαζόμαστε επομένως έναν τρόπο να κάνουμε compile μόνο τα αρχεία που το χρειάζονται
  - Incremental compilation
- Για να το πετύχουμε αυτό πρέπει να γνωρίζουμε τις εξαρτήσεις μεταξύ των αρχείων του προγράμματός μας
- Αν αλλάξει κάποιο αρχείο που περιέχει μια δήλωση ή ένα ορισμό, θα πρέπει να γίνουν compile όλα τα αρχεία που χρησιμοποιούν αυτή τη δήλωση ή τον ορισμό (και ελπίζουμε μόνο αυτά ώστε να γίνει η λιγότερη δυνατή δουλειά)
- Πως το πετυχαίνουμε αυτό;

Εξαρτήσεις αρχείων

- Τα αρχεία ενός προγράμματος έχουν εξαρτήσεις που μπορούν να περιγραφούν με ένα γράφο εξαρτήσεων, π.χ.
- Έστω ένα πρόγραμμα που αποτελείται από τα αρχεία f1.c, f2.c, f3.c, f1.h, f2.h με τις εξαρτήσεις που εκφράζονται από τον διπλανό γράφο
  - Π.χ. το f1.o χρειάζεται το f1.c που κάνει include το f1.h
- Αν έχουμε αυτόν τον γράφο μπορούμε να κάνουμε compile μόνο ότι χρειάζεται για να παράγουμε το εκτελέσιμο
- Αν αλλάξει κάποιο αρχείο, πρέπει να ακολουθήσουμε τα μονοπάτια από το αντίστοιχο αρχείο μέχρι την ρίζα και να παράγουμε πάλι τα αρχεία σε αυτά τα μονοπάτια
- Πως παράγουμε το κάθε αρχείο;
- Σε κάθε κόμβο του δέντρου σημειώνουμε την εντολή που χρειάζεται για να παραχθεί το αντίστοιχο αρχείο
  - Π.χ. Το f1.o μπορεί να παραχθεί από το f1.c με την εντολή
  - gcc –ansi –Wall –pedantic –c f1.c
  - Αντίστοιχα για όλα τα f1.o αρχεία
  - Το a.out παράγεται με την εντολή: gcc f1.o f2.o f3.o (καλεί μόνο τον linker)
- Όποτε αλλάζει ένα αρχείο αρκεί να εκτελούμε μηχανικά μόνο τις εντολές που βρίσκονται σε όλα τα paths από τον αντίστοιχο κόμβο μέχρι την ρίζα
  - Πχ. Αν αλλάξει το f1.h θα πρέπει να εκτέλεσουμε
  - gcc –ansi –Wall –pedantic –c f1.c
  - gcc –ansi –Wall –pedantic –c f2.c
  - gcc f1.o f2.o f3.o
  - Αλλά δεν χρειάζεται να εκτελέσουμε την εντολή: gcc –ansi –Wall –pedantic –c f3.c



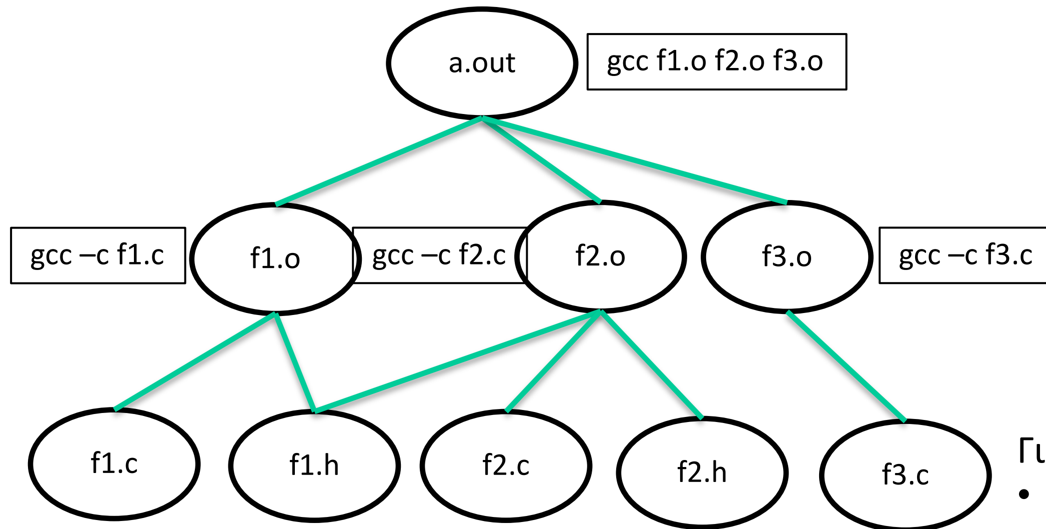
make

- Το make είναι ένα πρόγραμμα που μας διευκολύνει αυτοματοποιώντας αυτή τη διαδικασία
- Το make παίρνει σαν input έναν γράφο εξαρτήσεων με τα αντίστοιχα annotations εντολών σε κάθε κόμβο
- Όταν αλλάξει ένα αρχείο, εκτελεί τις εντολές που βρίσκονται στα αντίστοιχα paths
- Η περιγραφή του γράφου των εξαρτήσεων γίνεται με την μορφή κειμένου σε ένα αρχείο που συνήθως το ονομάζουμε Makefile (κατά σύμβαση)
- Το Makefile περιγράφει έναν-έναν τους κόμβους του δέντρου ως εξής

```
node: dependencies          π.χ.    f1.o:    f1.c f1.h
TAB     command                     TAB      gcc –ansi –Wall –pedantic –c f1.c
```

- Κατά σύμβαση, ο πρώτος κόμβος στο Makefile είναι ο τελικός στόχος (π.χ. η ρίζα a.out)
- Το make διαβάζει το Makefile, σχηματίζει τον γράφο των εξαρτήσεων και κοιτάζει τις ημερομηνίες των αρχείων
- Αν ένα αρχείο μιας εξάρτησης έχει μεταγενέστερη ημερομηνία από τον κόμβο θεωρείται ότι έχει αλλάξει και ότι πρέπει να ξαναδημιουργηθεί ο κόμβος με την εντολή που δώσαμε εμείς
- Έτσι το make εκτελεί βήμα βήμα μόνο τις εντολές που είναι απαραίτητες για να δημιουργηθεί πάλι το τελικό εκτελέσιμο a.out
- Το make και τα Makefiles δεν έχουν σχέση με την C. Είναι ένα εργαλείο που περιγράφει γράφους εξαρτήσεων και χρησιμοποιείται περισσότερο από οτιδήποτε άλλο για incremental compilation σε πολύπλοκα συστήματα
- Σε πολλές περιπτώσεις μπορούμε να παράγουμε τα Makefiles αυτόματα από τον κώδικα του προγράμματός μας (αλλά πρέπει να ακολουθούμε ορισμένες συμβάσεις με την χρήση των header files)

Γράφος εξαρτήσεων και Makefile



Για να κάνουμε compile το a.out:

- Τοποθετούμε το Makefile στο directory με το πρόγραμμα
- Κάθε φορά που κάνουμε edit κάποιο αρχείο
- Στο ίδιο directory γράφουμε απλά make στο prompt
  $make

Makefile

```
a.out: f1.o f2.o f3.o
TAB    gcc f1.o f2.o f3.o

f1.o: f1.c f1.h
TAB    gcc –ansi –Wall –pedantic –c f1.c

f2.o: f2.c f1.h f2.h
TAB    gcc –ansi –Wall –pedantic –c f2.c

f2.o: f3.c
TAB    gcc –ansi –Wall –pedantic –c f3.c
```

Reading

- King Ch. 10, 15, 18.2 (extern)
- More on make: Chapter 5, (online) Programming with GNU Software. Gary V. Vaughan, Akim Demaille, Paul Scott, Bruce Korb, and Richard Meeking. Edition 2, 2002. [pdf]
- Everything on make: https://www.gnu.org/software/make/manual/make.pdf