HY255 Εργαστήριο Λογισμικού - L06

(a) Modules
(b) Assignment 2 – string module

HY255 Εργαστήριο Λογισμικού
Άνοιξη 2021
Άγγελος Μπίλας

Modules

- Ένα module είναι ένα σύνολο από ”services/υπηρεσίες“ που μπορούν να χρησιμοποιήσουν άλλα μέρη ενός προγράμματος, π.χ.
  - Ένα σύνολο από συναρτήσεις
  - Ένα σύνολο από δηλώσεις ή δεδομένα
  - Ορισμοί τύπων
  - Συνδυασμός των παραπάνω
- Κάθε module έχει
  - Ένα interface που περιγράφει πως κάποιος μπορεί να χρησιμοποιήσει το module και τις υπηρεσίες του
  - Μία ή περισσότερες υλοποιήσεις που υλοποιεί με τον επιθυμητό τρόπο τις υπηρεσίες αυτές
- Το interface ενός module είναι ένα .h αρχείο ενώ η υλοποίηση είναι ένα σύνολο από .h και .c αρχεία

Γιατί modules?

- Υλοποιούμε ένα αυτόνομο κομμάτι κώδικα που μπορούν να χρησιμοποιούν άλλοι (ή και εμείς) κατανοώντας μόνο το interface (που είναι μικρό) χωρίς να τους απασχολεί η υλοποίηση (που ενδεχομένως είναι πολύπλοκη)
- Αυτές οι αρχές του modularity και reusability είναι εξαιρετικά σημαντικές για να μειώσουμε τον χρόνο/προσπάθεια σχεδιασμού συστημάτων που πλέον κυριαρχεί σε θέματα κόστους

Παράδειγμα: stack module

stack.h interface

```
void make_empty(void); /* create an empty stack */

int is_empty(void); /* return true if stack is
                       empty */

void push(int i);   /* place i on top of stack */

int pop(void);      /* return and remove element
                       from top of stack */
```

Implementation: stack.c

```
#include <stdio.h>
#include <stdlib.h>
#include “stack.h”
#define STACK_SZ 100;
...... int  contents[STACK_SZ];
...... int top = 0;
void make_empty(void){
  top=0;
}
int is_empty(void){
  return (top==0);
}
.... int is_full(void){
  return (top==STACK_SZ);
}
void push(int i){
  if (is_full()){
    printf(“Error in push – stack full\n”);
    exit(EXIT_FAILURE);
  }
  contents[top++] = i;
  return;
}
int pop(void){
  if (is_empty()) {
    printf(“Error in pop – stack empty\n”);
    exit(EXIT_FAILURE);
  }
  return contents[--top];
}
```

- Σε ένα module, είναι σημαντικό να ”αποκαλύψουμε” στον εξωτερικό κόσμο ακριβώς όση πληροφορία χρειάζεται: ούτε περισσότερη, ούτε λιγότερη.
- Αυτή η πληροφορία περιέχεται πάντα στο interface και είναι απαραίτητη για την χρήση του module από άλλα τμήματα του προγράμματος.
- Στο συγκεκριμένο παράδειγμα π.χ. είναι κρυμμένο από τον χρήστη το γεγονός ότι η στοίβα υλοποιείται με ένα array contents και μια μεταβλητή top που δείχνει στη στοίβα του array.
- Επίσης, για λόγους απλά του παραδείγματος είναι κρυμμένη και η συνάρτηση is_full();

1. Information hiding in modules

- Πως εξασφαλίζουμε όμως ότι οι πληροφορίες που θέλουμε να "κρύψουμε" δεν είναι προσβάσιμες στον χρήστη του module;
- Σίγουρα δεν συμπεριλαμβάνουμε αυτές τις πληροφορίες στο interface. Ωστόσο, αυτό δεν είναι αρκετό. Αν κάποιος χρήστης γνωρίζει η μαντέψει το όνομα της μεταβλητής top, τότε μπορεί να την χρησιμοποιήσει στο πρόγραμμά του.
- Για τον σκοπό αυτό η C διαθέτει ένα ειδικό keword για τις δηλώσεις:
- static: ορίζει ότι η αντίστοιχη δήλωση είναι ορατή μόνο στο αρχείο που περιέχεται και όχι σε άλλα αρχεία του προγράμματος
- Οπότε στο διπλανό πρόγραμμα τα σημεία "..." πρέπει να οριστούν ως static
- Γενικά, οποιεσδήποτε δηλώσεις δεν εμφανίζονται στο interface θα πρέπει να οριστούν σαν static
- Note: static για τοπικές μεταβλητές σημαίνει κάτι εντελώς διαφορετικό που δεν θα μας απασχολήσει και δεν χρειάζεται / πρέπει να πολύ-χρησιμοποιούμε

2. Interface-Implementation separation and reuse

- Έστω ότι γράφουμε ένα πρόγραμμα p.c με μια μόνο συνάρτηση main που χρησιμοποιεί το stack module

```
main.c
#include <stdio.h>
#include "stack.h"
void main(void){
     int t;
     make_empty();
     push(1);
     t=pop();
     return 0;
}
```

- 1. Τι χρειαζόμαστε για να παράγουμε το main.o ;
  - Το interface stack.h πρέπει να είναι διαθέσιμο για τον pre-processor για να εκτελεστεί η εντολή: gcc –Wall –pedantic –ansi –c p.c
  - Δεν χρειάζεται κάτι άλλο για να παράγουμε το main.o
- 2. Τι χρειαζόμαστε για να παράγουμε το a.out ;
  - Το main.o και το stack.o ώστε να εκτελεστεί η εντολή gcc main.o p.o που ουσιστικά εκτελεί τον linker
- 3. Τι χρειάζεται επομένως όποιος γράφει το πρόγραμμα p.c ;
  - Όποιος γράφει το P.c, τελικά χρειάζεται μόνο τα stack.h και stack.o και ΔΕΝ χρειάζεται να δει ποτέ το stack.c.
  - Επομένως, ένα module μπορεί να χρησιμοποιηθεί δίνοντας μόνο το interface και την υλοποίησης σε object/εκτελέσιμο κώδικα και όχι source.
  - Ο σχεδιαστής του module μπορεί να παράγει μια φορά το stack.o και να δίνει αυτό σε όλους τους χρήστες (μαζί με το stack.h)
  - gcc –Wall –pedantic –ansi -o stack.c
- 4. Τι γίνεται αν αλλάξει κάτι στην υλοποίηση του module;
  - Αν ο σχεδιαστής του module αργότερα αλλάξει κάτι στην υλοποίηση ή κάνει μια διαφορετική/καλύτερη υλοποίηση χωρίς να αλλάξει το interface, τότε το πρόγραμμα p θα χρειαστεί μόνο το νέο stack.o και δεν θα χρειαστεί να αλλάξει τίποτε στον source κώδικα του p.c

Nested #include's

- Τι γίνεται αν καθώς θέλουμε να χρησιμοποιήσουμε ένα interface κάποιου module "πέσουμε" στην εξής περίπτωση
- Έστω δύο modules f1 (f1.h, f1.c) και f2 (f2.h, f2.c) που το f1 χρειάζεται το f2 και το f2 χρειάζεται το f1
- Σε αυτή την περίπτωση, θα πρέπει να είναι γραμμένα σαν:

```
f1.h
#include "f2.h"
...
```

```
f2.h
#include "f1.h"
...
```

- Αν τώρα εμείς γράψουμε ένα πρόγραμμα p.c που χρειάζεται και τα δύο modules τότε μάλλον πρέπει να γράψουμε:

```
p.c
#include "f1.h"
#include "f2.h"
...
```

- Τι θα συμβεί στην περίπτωση αυτή κατά το compilation του p.c, στο στάδιο του cpp;
- Κυκλικές εξαρτήσεις των f1.h και f2.h που δεν τελειώνουν ποτέ...
- Ουσιαστικά, δεν έχουμε τρόπο να πούμε στον cpp ότι θέλουμε να συμπεριλάβει μια φορά το κάθε interface
- Για να το πετύχουμε αυτό χρησιμοποιούμε ένα trick

```
f1.h
#ifndef _F1_H_
#define _F1_H_

#include "f2.h"
...
#endif
```

```
f2.h
#ifndef _F2_H_
#define  _F2_H_

#include "f1.h"
...
#endif
```

- Οπότε την πρώτη φορά που θα γίνει include το f1.h ή το f2.h θα κάνει define το αντίστοιχο όνομα και τις επόμενες φορές δεν θα γίνει πάλι include

Χρήση #ifdef σε stack.h

stack.h interface

```
#ifdef _STACK_H_                    /* ensure module interface is
#define _STACK_H_                   defined once */

void make_empty(void);              /* create an empty stack */
int is_empty(void);                 /* return true if stack is empty */
void push(int i);                   /* place i on top of stack */
int pop(void);                      /* return and remove element from
                                    top of stack */

#endif
```

Modules vs. libraries

- Μια βιβλιοθήκη είναι ουσιαστικά ένα σύνολο από ένα ή περισσότερα modules
- Επομένως για να χρησιμοποιήσουμε μια βιβλιοθήκη, ουσιαστικά τα modules που περιέχει, χρειαζόμαστε τα αντίστοιχα interfaces και υλοποιήσεις
- Η C μας δίνει έτοιμα διάφορα modules
- Τα interfaces τους είναι τα αντίστοιχα .h files, π.χ. stdio.h, stdlib.h, assert.h, math.h, etc
- Οι υλοποιήσεις των περισσότερων συμπεριλαμβάνονται όλες σε μια βιβλιοθήκη με εκτελέσιμο κώδικα, την standard βιβλιοθήκη της C, stdlib που είναι το αρχείο libc.a (μαζί με τον compiler)
- Φυσικά υπάρχουν και άλλα modules που δεν έρχονται με την γλώσσα, π.χ. για πρωτόκολλα δικτύου (sockets), threads, κ.ο.κ.

Assignment 2: string module [https://www.csd.uoc.gr/~hy255/as2]

- Γράψτε ένα module που επεξεργάζεται strings, π.χ.
  - copy ένα string σε ένα άλλο
  - compare δύο strings
  - Οι συναρτήσεις που περιέχει το module δίνονται από την άσκηση και είναι ίδιες με τις συναρτήσεις της αντίστοιχης βιβλιοθήκης της C
  - Οπότε με man funname μπορείτε να δείτε τον ορισμό της κάθε συνάρτησης
- Στόχος η κατανόηση των βασικών τύπων και κυρίως των arrays, pointers, και της σχέσης τους
- Για να το πετύχει αυτό η άσκηση ζητάει το module να έχει δύο υλοποιήσεις του ίδιου interface mystring.h
  - 1. mystrarr.c που θα χρησιμοποιεί μεταβλητές τύπου array και όχι μεταβλητές τύπου pointer
  - 2. mystrptr.c που θα χρησιμοποιεί μεταβλητές τύπου pointer και όχι μεταβλητές τύπου array
- Η άσκηση δεν χρειάζεται την χρήση δυναμικής μνήμης (όπως και οι αντίστοιχες συναρτήσεις της C) – θα το κατανοήσετε στην πορεία
- Στην υλοποίησή σας θα χρησιμοποιήσετε και το assert module της C που έχει interface το assert.h
  - #include <assert.h>
  - Περιέχει μια μόνο συνάρτηση (macro), assert(cond) που σταματάει την εκτέλεση του προγράμματος αν η συνθήκη cond είναι false, π.χ. assert (x >= 0);
  - Θα χρησιμοποιήσουμε σε αυτό το σημείο την assert για όλα τα λάθη που μπορεί να συμβούν, π.χ. για τον έλεγχο null pointers. Αργότερα θα εξηγήσουμε ποια είναι η σωστή και ακριβής χρήση της assert
  - Μπορείτε να δείτε την υλοποίησης της assert μέσα στο αρχείο /usr/include/assert.h (σε αυτή την περίπτωση το module assert παρέχει σαν υπηρεσία τον ορισμό του macro assert και τίποτε άλλο)
- Το πρόγραμμά σας δεν θα πρέπει να έχει warnings οπότε οι τύποι θα πρέπει να ταιριάζουν ακριβώς
  - Αν υπάρχει κάποιο warning θα πρέπει να εξηγήσετε γιατί δεν γίνεται διαφορετικά

Reading - Εργασία

- King Ch. 18.2 (static), 19.{1,2}
- Διαβάστε την εκφώνηση για την Άσκηση 2
- Υλοποιήστε την Άσκηση 2