Tema 1 Multi-platform Development

- Dată publicare: **10.03.2021**
- **Deadline**: **21.03.2021, ora 23:55**
- **Deadline hard**: **28.03.2021, ora 23:55**

Scopul temei

- Recapitularea lucrului cu funcțiile din biblioteca standard C:
  - lucrul cu fișiere
  - alocare dinamică de memorie
  - folosirea pointerilor
- Încapsularea datelor într-o formă abstractă pentru o structură de date
- Realizarea unui Makefile pentru platformele Linux (folosind `gcc`) și Windows (folosind `cl`)

Dezvoltarea temei

Dezvoltarea trebuie făcută exclusiv pe mașinile virtuale SO.

Nu rulați testele *"local"* (pe calculatoarele voastre sau în mașinile voastre virtuale). Veți avea diferențe față de vmchecker, iar echipa de SO nu va depana testele care merg "local", dar pe vmchecker nu merg. Pe vmchecker sunt aceleași mașinile virtuale ca cele de pe wiki.

Este încurajat ca lucrul la tema să se desfășoare folosind `git`. Indicați în README link-ul către repository dacă ați folosit git. **Asigurați-vă că responsabilii de teme au drepturi de citire asupra repo-ului vostru**.

Ca să vă creați un repo de gitlab în instanța facultății: în repository-ul so-assignments de pe Github se află un script Bash care vă ajută să vă creați un repository privat pe instanța de Gitlab a facultății, unde aveți la dispoziție 5 repository-uri private utile pentru teme. Urmăriți indicațiile din README și de pe wiki-ul SO.

Motivul pentru care încurajăm acest lucru este că responsabilii de teme se pot uita mai rapid pe Gitlab la temele voastre pentru a vă ajuta în cazul în care întâmpinați probleme/bug-uri. Este mai ușor să primiți suport în rezolvarea problemelor implementării voastre dacă le oferiți responsabililor de teme acces la codul sursă pe Gitlab.

Crash-course practic de git puteți găsi aici: git-immersion

Testele publice care se rulează pe vmchecker se găsesc pe repo-ul so-assignments de pe Github:

```
student@so:~$ git clone https://github.com/systems-cs-pub-ro/
so-assignments.git
student@so:~$ cd so-assignments/1-multi
```

În repository-ul de pe Github se vor găsi și scheletele pentru temele viitoare, care vor fi actualizate și se vor putea descărca pe viitor folosind comanda:

```
student@so:~$ git pull
```

Tot prin comanda de mai sus se pot obține toate actualizările făcute în cadrul temei 1.

Enunț

Să se implementeze **în C** un mini preprocesor pentru fișiere conținând cod sursă C. Preprocesarea este o etapă premergătoare compilării efective a fișierului ce conține cod sursă. Preprocesorul va analiza fișierul de intrare, conținând cod sursa C și va scrie, la consolă sau într-un fișier de iesire, rezultatul preprocesării fișierelor de intrare.

Rezolvarea temei presupune implementarea unui subset al directivelor de preprocesare specifice limbajului C: `#define`, `#include`, `#if`, `#elseif` (sub forma `#elif`), `#else`, `#endif`, `#ifdef`, `#ifndef`, `#undef`. Sintaxa și descrierea acestora sunt prezentate în tabelul de mai jos.

Directiva	Descrierea directivei
`#define <SYMBOL> <MAPPING>`	Stochează o asociere între <SYMBOL> și <MAPPING>. Toate aparițiile lui <SYMBOL> în fișierul cu codul sursă vor fi înlocuite cu <MAPPING> (vezi exemplul de mai jos).
`#if <COND>` / `#elif <COND>` / `#else` / `#endif`	Se verifică secvențial dacă <COND> se evaluează la un literal întreg diferit de 0. În caz afirmativ, în fișierul rezultat se vor procesa și adauga doar liniile de cod specifice primului bloc a cărui condiție a fost validată.
`#ifdef <SYMBOL>` / `#ifndef <SYMBOL>` / `#else` / `#endif`	Se verifică dacă <SYMBOL> a fost sau nu definit anterior.
`#include “HEADER”`	Realizeaza preprocesarea fisierului indicat de “HEADER” si adauga liniile de cod preprocesat in fisierul de iesire.

Executabilul rezultat se va numi `so-cpp` și va avea următoarea semnătură:

`so-cpp [-D <SYMBOL>[=<MAPPING>]] [-I <DIR>] [<INFILE>] [ [-o] <OUTFILE>]`

Semnificația argumentelor este următoarea:

- `-D <SYMBOL>[=<MAPPING>]` sau `-D<SYMBOL>[=<MAPPING>]`: va defini simbolul cu numele

<SYMBOL> și valoarea <MAPPING>; dacă <MAPPING> lipsește, <SYMBOL> va primi valoarea șirului vid (””). <SYMBOL> poate fi lipit de -D sau nu.
- -I <DIR> sau -I<DIR>: va adăuga un director în care se vor căuta fișiere incluse de codul sursă folosind directive #include
- -o <OUTFILE> sau -o<OUTFILE>: va scrie output-ul preprocesat în fișierul <OUTFILE>
- <INFILE>: specifică un fișier din care se va citi codul sursă pentru; dacă parametrul lipsește, codul sursă va fi obținut de la consolă (stdin)

Atenție: argumentele specificate cu modificatorii -D și -I pot fi folosiți de mai multe ori; de fiecare dată vor adăuga o nouă definiție, respectiv vor apenda un nou path. Fișierele de intrare și de ieșire pot fi definite o singură dată! Pentru mai multe detalii, puteți consulta pagina de manual a preprocesorului C.

Exemplu de fișier de intrare, conținând cod sursă C:

```
#define VAR0 1

int main(int argc, char **argv)
{
    int y = VAR0 + 1;
    printf("VAR0 = %d\n", VAR0);

    return 0;
}
```

În urma preproceseării se va obține fișierul:

```
int main(int argc, char **argv)
{
    int y = 1 + 1;
    printf("VAR0 = %d\n", 1);

    return 0;
}
```

Se observă că în exemplul de mai sus, nu toate aparițiile șirului de caractere VAR0 au fost înlocuite cu 1: aparițiile într-un context de literal șir de caractere ale unui simbol introdus prin directiva #define nu trebuie înlocuite.

În vederea obținerii functionalității specifice unui preprocesor, se recomandă implementarea unei structuri de date de tip HashMap. Aceasta va fi folosită pentru a stoca asocieri de tipul <SYMBOL, MAPPING>.

În urma preprocesării fișierului de intrare, trebuie ca într-o structură de date de tip HashMap să existe o singură asociere între două șiruri de caractere, anume <“VAR0”, “1”>.

Precizări generale

Indicațiile și precizările generale pentru teme sunt valabile și aici. Vă rugăm să le parcurgeți și să țineți cont de ele înainte de a vă apuca de temă și respectiv înainte de submisia finală.

- Directiva `#define` trebuie implementată astfel încât să suporte:
  - `#define`-uri simple (precum cel din exemplu)
  - `#define`-uri care folosesc în componența lor alte `#define`-uri
  - `#define`-uri de tip multilinie (respectând aceeași sintaxa ca în C)
- Nu trebuie implementat suport pentru `#define`-uri parametrizate (de tip funcție)
- Pentru directivele de tip `#if <COND>` / `#elif <COND>`, **nu** este necesar să existe suport pentru operații aritmetice, de tipul `#if 2 + 3`. Însa, este necesar sa existe suport pentru utilizarea altor `#define`-uri pe post de `<COND>`.
- Directiva `#include` va suporta doar includerea de fisiere header oferite in scheletul temei. Astfel, nu trebuie implementat suport pentru includerea fișierelor din sistem de tipul `#include <stdio.h>` sau `#include <stdlib.h>`
- Fișierele header pot fi incluse recursiv (un fișier header poate include alt fișier header)
- Fișierele importate folosind directive `#include` trebuiesc căutate în directorul în care se află fișierul de input, sau în directorul curent, în cazul în care codul de input este specificat la consolă. Dacă nu este găsit în directorul fișierului, el este căutat în toate directoarele specificate folosind parametrul `-I`, în ordinea în care acestea au fost specificate în linia de comandă.
- În cazul în care un fișier inclus nu este găsit în niciun director, programul va iesi cu eroare.
- Pentru a împărți un șir în tokeni, vă recomandăm să folosiți următoarele delimitatoare: `\t []{}<>=+-*/%!&|^.,:;()\`.
- Executabilul generat va purta numele **`so-cpp`** pe Linux și **`so-cpp.exe`** pe Windows.
- Pentru Windows, compilarea se va realiza din PowerShell, iar rularea se va face folosind Cygwin.
- Makefile-ul pentru Windows trebuie să compileze sursele utilizând flag-ul **/MD**
- Dimensiunea maximă a unei linii din fisierul de cod sursa este de **256** de caractere.
- Buffer-ul folosit pentru citirea liniilor poate fi declarat cu dimensiune statică.
- **Verificați valorile întoarse de funcțiile `malloc/calloc/realloc` (în funcție de implementarea aleasă)**. În cazul în care una dintre aceste funcții eșuează, trebuie întors codul de eroare 12 (este codul de eroare pentru `ENOMEM`). Acest cod de eroare trebuie propagat și returnat până la ieșirea din program. Valoarea erorii este pozitivă.
  - exemplu: din `main` se apelează `f1`, iar `f1` apelează `f2`: dacă eroarea apare în momentul apelului unui `malloc` în funcția `f2`, atunci codul de eroare (valorea 12) va fi întors în `f1`, din `f1` va trebui întors tot 12, iar din main se va ieși cu același cod de eroare.
- **Nu aveți voie să apelați un alt preprocesor (folosind `system`, exec, sau orice altă metodă)** pentru a implementa functionalitatea cerută.

Precizări VMChecker

Arhiva temei va fi încărcată de două ori pe vmchecker (Linux și Windows). Arhiva trimisă trebuie să fie aceeași pe ambele platforme (se vor compara cele două arhive trimise, în caz că va exista vreo diferență între cele două încărcări, tema va fi punctată cu 0).

Insistăm, dacă mesajul cu roșu nu a fost clar: arhiva care se trimite pe vmchecker trebuie să fie **identică** pe ambele platforme. Puteți folosi `md5sum` sau `sha1sum` (sau comenzi similare) asupra arhivelor voastre dacă ați dezvoltat în locuri diferite.

Temele trimise pe o singură platformă sau cu arhive diferite nu vor fi punctate și vor fi notate cu 0.

Arhivele trebuie să conțină sursele temei, README și două fișiere Makefile care conțin target-urile `build` și `clean`:

- **Linux:** Fișierul Makefile se va numi `GNUmakefile`.
  - **ATENȚIE**: GNUmakefile (cu `m` mic).
- **Windows:** Fișierul Makefile se va numi `Makefile`.
- Regula de `build` trebuie să fie cea principală (executată atunci când se dă `make` fără parametrii)
- Pentru a documenta realizarea temei, vă recomandăm să folosiți template-ul de aici

Executabilul rezultat din operația de compilare și linking se va numi `so-cpp` pe Linux și `so-cpp.exe` pe Windows.

Punctare

- Tema va fi punctată cu minimul punctajelor obținute pe cele două platforme. Nu aveți voie să folosiți directive de preprocesare de forma:

```
#ifdef __linux__
[...]
#ifdef _WIN32
[...]
```

Cu alte cuvinte: exact același cod trebuie să ruleze pe ambele platforme. Tema **NU TREBUIE** să conțină surse specifice unuia sau altuia dintre sistemele de operare Linux/Windows. Veți avea două fișiere makefile (`GNUmakefile` pentru Linux și `Makefile` pentru Windows, cum e precizat mai sus) iar checker-ul va ști, în funcție de sistemul lui de operare, ce makefile să folosească.

Nota mai poate fi modificată prin depunctări suplimentare:

- Lista generală de depunctări
- `-2` implementare netransparentă a structurii de date de tip HashMap; HashMap-ul ar trebui să fie abstractizat cu un singur obiect (în C: structură de date), iar operațiile pe HashMap trebuie făcute pe obiectul respectiv. Puteți folosi definiții proprii pentru elementele HashMap-ului.
- `-4` alocare statică HashMap
- se pot scădea oricâte puncte pentru teme care conțin erori grave/vizibile de coding style sau de funcționare care pot să nu fie pe lista generală de depunctări

Testul 0 din cadrul checker-ului temei verifică automat coding style-ul surselor voastre folosind stilul de coding din kernelul Linux. Acest test valorează **5 puncte** din totalul de 100. Pentru mai multe informații despre un cod de calitate citiți pagina de recomandări.

Pentru investigarea problemelor de tip *Segmentation Fault* sau comportament incorect al aplicației la

unul din teste, pentru debugging, se recomandă folosirea gdb în Linux.

Una dintre depunctări este pentru leak-uri de memorie. În Linux pentru identificarea lor puteți folosi utilitarul valgrind.

Pentru instalarea `gdb` și `valgrind`, pe o distribuție Ubuntu se poate folosi comanda:

```
student@so:~$ sudo apt-get install gdb valgrind
```

Pentru debugging și detectarea leak-urilor de memorie este necesar să ștergeți toate optimizările de la flag-urile de compilare (e.g. `-O3`) și trebuie să compilați doar cu flag-urile `-Wall -g` (sau cele care mai activează alte warning-uri, e.g. `-Wextra`).

Nu trebuie la fiecare eroare considerată fatală să eliberați fiecare pointer alocat dinamic. În cadrul corecturii temei principala verificare pentru memory leaks va fi pe o funcționare corecta/normală, fără input invalid. Rețineți că memory leak-ul apare atunci când programul vostru nu poate returna sistemului de operare memoria folosită! Concentrați-vă pe folosirea `valgrind` pe teste care trec și care dau input valid într-o primă fază.

Precizări Makefile

Makefile-ul trebuie să respecte următoarea structură: pentru fiecare fișier `.c` generat trebuie să se obțină un fișier obiect. La final trebuie să faceți linkarea între sursa principală (să zicem `main.c` din care se obține `main.o`) și celelalte fișiere obiect obținute din celelalte surse ale voastre.

Porniți de la exemplele de Makefile atât pentru Linux cât și pentru Windows oferite în laboratorul 1. Un alt exemplu puteți găsi aici.

Nu uitați: Makefile-ul pentru Windows trebuie să compileze toate sursele voastre utilizând flag-ul **/MD**.

Denumirile lor trebuie să fie:

- **Linux:** Fișierul Makefile se va numi `GNUmakefile`.
  - **ATENȚIE**: GNUmakefile (cu `m` mic).
- **Windows:** Fișierul Makefile se va numi `Makefile`.

Resurse necesare realizării temei

Pentru a clona repo-ul și a accesa resursele temei 1:

```
student@so-vm:~$ git clone https://github.com/systems-cs-pub-ro/
so-assignments.git
student@so-vm:~$ cd so-assignments
student@so-vm:~/so-assignments$ cd 1-multi
```

- Referințe utile:
  - ANSI C reference
  - Data Structures Visualization

FAQ

- **Q:** Tema 1 se poate face în C++?
  - **A:** Nu.
- **Q:** Se pot folosi directive de preprocesare de tipul `#define`?
  - **A:** Da. Singurele directive de preprocesare interzise sunt cele care introduc cod condițional în funcție de OS-ul folosit (e.g. `ifdef __linux__`)
- **Q:** Pentru citire/scriere din fișier/consolă putem folosi `freopen`?
  - **A:** Da, e ok. Puteți folosi orice funcție din categoria `fopen`, `fread`, `fwrite`, `fclose`.
- **Q:** Se poate folosi `realloc`?
  - **A:** Da.
- **Q:** Se pot folosi funcțiile `fgets`, `fscanf`, `printf`, `fprintf`?
  - **A:** Da. Atenție să nu folosiți `gets`!
- **Q:** Pe Windows, folosind `cl.exe` nu mi se compilează același cod care mi se compila pe Linux. De ce?
  - **A:** Cel mai probabil cauza este următoarea: pe Linux este folosit C99 ca standard la `gcc`, care printre altele acceptă să declari variabile în mijlocul codului. Pe Windows, compilatorul `cl` folosește standardul C89, care forțează declararea variabilelor **doar** la început (un exemplu de problema).
- **Q:** Văd că pentru coding style iau 0, ce pot face în această situație?
  - **A:** Descărcați cu `wget` checkpatch.pl de aici, îl puneți în PATH și apoi rulați checker-ul de Linux (pașii sunt mai jos). Alternativ, vă puteți folosi de acest wrapper peste checkpatch.pl a verifica sursele folosind criteriile considerate în evaluarea temelor.

```
student@so:~$ wget https://raw.githubusercontent.com/torvalds/linux/master/
scripts/checkpatch.pl
student@so:~$ export PATH=$PATH:/path/to/dir/with-checkpatch
student@so:~$ cd /path/to/lin/checker && ./run_all.sh
```

Suport, întrebări și clarificări

Pentru întrebări sau nelămuriri legate de temă folosiți forumul temei. Recomandăm să căutați eventuale întrebări și în arhiva listei de discuții, poate veți găsi ceea ce căutați până veți primi un răspuns din partea noastră.

Orice intrebare pe mailing list e recomandat să conțină o descriere cât mai clară a eventualei probleme. Întrebări de forma: “Nu merge X. De ce?” fără o descriere mai amănunțită vor primi un răspuns mai greu.

ATENȚIE să nu postați imagini cu părți din soluția voastră pe forumul pus la dispoziție sau orice alt

canal public de comunicație. Dacă veți face acest lucru, vă asumați răspunderea dacă veți primi copiat pe temă.