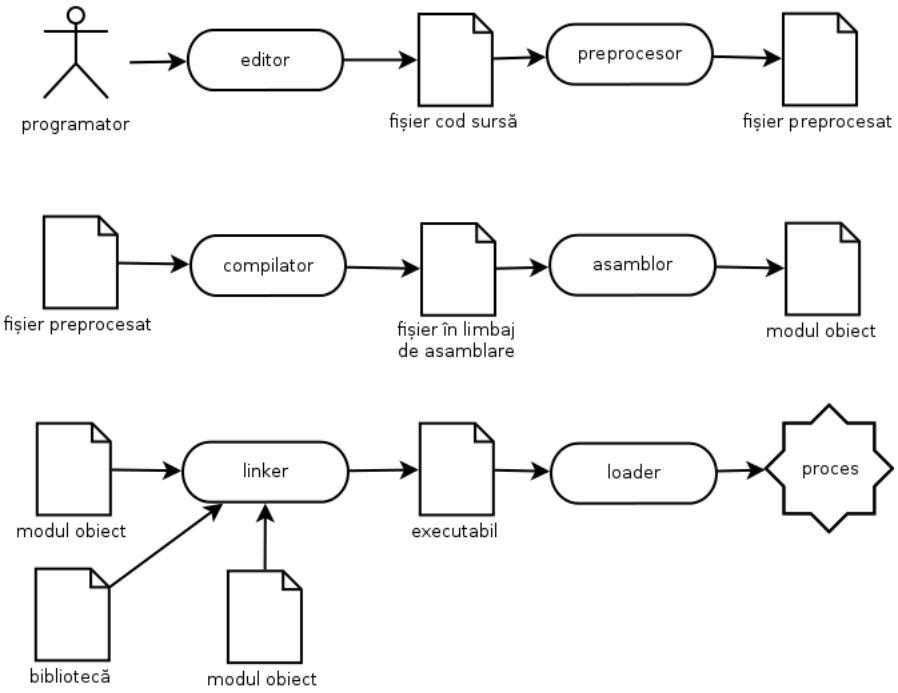
Laborator 03: Compilare

Etapele prin care trece un program scris în C din momentul în care este scris până când este rulat ca un proces sunt, in ordine:

- preprocesare
- compilare
- asamblare
- link editare

În imaginea de mai jos sunt reprezentate si detaliate aceste etape:



Preprocesare

În cadrul primei etape, cea de `preprocesare` au loc următoarele acțiuni plecând de la fișierul cod sursă:

- eliminarea comentariilor
- expandarea directivelor care încep cu simbolul #
  - înlocuirea valorilor corespunzătoare pentru `#define ...`
  - includerea conținutului fișierelor date ca parametru directivei `#include`

Compilare

În etapa de `compilare` au loc următoarele subetape:

- analiza lexicală - verificarea limbajului
- analiza sintactică - verificarea ordinii cuvintelor (`;` vine la finalul asignării unei variabile)
- analiza semantică - determinarea sensului codului scris (determinarea contextului variabilelor)
- generarea de cod în limbaj de asamblare care este o formă human-readable a ce ajunge procesorul să execute efectiv

Assembly / Asamblare

Assembly / Asamblare este penultima etapă a procesului de compilare în sens larg. În urma finalizării acestei etape rezultatul va fi crearea a unul sau mai multe fișiere obiect. Fișierele obiect pot conține mai multe lucruri, printre care:

1. nume de simboluri
2. constante folosite în cadrul programului
3. cod compilat
4. valori de tip import/export care vor fi „rezolvate” în etapa de linking

Pentru a crea un fișier obiect în cazul în care se utilizează compilatorul `gcc` se folosește opțiunea `-c` așa cum puteți vedea și în secțiunea Invocarea linker-ului de mai jos.

Linking / Legare

Linking / Legare este ultima etapă a procesului de compilare în sens larg. La finalul acestei etape va rezulta un fișier executabil prin unificarea(„legarea”) mai multor fișiere obiect care pot avea la bază limbaje de programare de nivel înalt diferite; tot ceea ce contează este ca fișierele obiect să fie create în mod corespunzător pentru ca linker-ul să le poată „interpreta”.

Pentru a obține un fișier executabil din fișiere obiect, linker-ul realizează următoarele acțiuni:

1. rezolvarea simbolurilor (*symbol resolution*): localizarea simbolurilor nedefinite ale unui fișier obiect în alte fișiere obiect
2. unificarea secțiunilor: unificarea secțiunilor de același tip din diferite fișiere obiect într-o singură secțiune în fișierul executabil
3. stabilirea adreselor secțiunilor și simbolurilor (*address binding*): după unificare se pot stabili adresele efective ale simbolurilor în cadrul fișierului executabil
4. relocarea simbolurilor (*relocation*): odată stabilite adresele simbolurilor, trebuie actualizate, în executabil, instrucțiunile și datele care referă adresele acelor simboluri
5. stabilirea unui punct de intrare în program (*entry point*): adică adresa primei instrucțiuni ce va fi executată

Invocarea linker-ului

Linker-ul este, în general, invocat de utilitarul de compilare (`gcc`, `clang`, `cl`). Astfel, invocarea linker-ului este transparentă utilizatorului. În cazuri specifice, precum crearea unei imagini de kernel sau imagini pentru sisteme încorporate, utilizatorul va invoca direct linkerul.

Dacă avem un fișier `app.c` cod sursă C, vom folosi compilatorul pentru a obține fișierul obiect `app.o`:

```
gcc -c -o app.o app.c
```

Apoi pentru a obține fișierul executabil `app` din fișierul obiect `app.o`, folosim tot utilitarul `gcc`:

```
gcc -o app app.o
```

În spate, `gcc` va invoca linker-ul și va construi executabilul `app`. Linker-ul va face legătura și cu biblioteca standard C (libc).

Procesul de linking va funcționa doar dacă fișierul `app.c` are definită funcția `main()`, funcția principală a programului. Fișierele linkate trebuie să aibă o singură funcție `main()` pentru a putea obține un executabil.

Dacă avem mai multe fișiere sursă C, invocăm compilatorul pentru fiecare fișier și apoi linker-ul:

```
gcc -c -o helpers.o helpers.c
gcc -c -o app.o app.c
gcc -o app app.o helpers.o
```

Ultima comandă este comanda de linking, care leagă fișierele obiect `app.o` și `helpers.o` în fișierul executabil `app`.

În cazul fișierelor sursă C++, vom folosi comanda `g++`:

```
g++ -c -o helpers.o helpers.cpp
g++ -c -o app.o app.cpp
g++ -o app app.o helpers.o
```

Putem folosi și comanda `gcc` pentru linking, cu precizarea linkării cu biblioteca standard C++ (libc++):

```
gcc -o app app.o helpers.o -lstdc++
```

Utilitarul de linkare este, în Linux, `ld` și este invocat în mod transparent de `gcc` sau `g++`. Pentru a vedea cum este invocat linker-ul, folosim opțiunea `-v` a utilitarului `gcc`, care va avea un rezultat asemănător cu:

```
/usr/lib/gcc/x86_64-linux-gnu/7/collect2 -plugin /usr/lib/gcc/x86_64-linux-gnu/7/liblto_plugin.so
-plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/7/lto-wrapper -plugin-opt=-fresolution=/tmp/ccwnf5NM.res
-plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lc
-plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s --build-id --eh-frame-hdr -m elf_i386 --hash-style=gnu
--as-needed -dynamic-linker /lib/ld-linux.so.2 -z relro -o hello
/usr/lib/gcc/x86_64-linux-gnu/7/../../../i386-linux-gnu/crt1.o
/usr/lib/gcc/x86_64-linux-gnu/7/../../../i386-linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-gnu/7/32/crtbegin.o
-L/usr/lib/gcc/x86_64-linux-gnu/7/32 -L/usr/lib/gcc/x86_64-linux-gnu/7/../../../i386-linux-gnu
-L/usr/lib/gcc/x86_64-linux-gnu/7/../../../../lib32 -L/lib/i386-linux-gnu -L/lib/../lib32 -L/usr/lib/i386-linux-gnu
-L/usr/lib/../lib32 -L/usr/lib/gcc/x86_64-linux-gnu/7 -L/usr/lib/gcc/x86_64-linux-gnu/7/../../../i386-linux-gnu
-L/usr/lib/gcc/x86_64-linux-gnu/7/../../.. -L/lib/i386-linux-gnu -L/usr/lib/i386-linux-gnu hello.o -lgcc --push-state
--as-needed -lgcc_s --pop-state -lc -lgcc --push-state --as-needed -lgcc_s --pop-state
/usr/lib/gcc/x86_64-linux-gnu/7/32/crtend.o /usr/lib/gcc/x86_64-linux-gnu/7/../../../i386-linux-gnu/crtn.o
COLLECT_GCC_OPTIONS='-no-pie' '-m32' '-v' '-o' 'hello' '-mtune=generic' '-march=i686'
```

Utilitarul `collect2` este, de fapt, un wrapper peste utilitarul `ld`. Rezultatul rulării comeznii este unul complex. O invocare "manuală" a comenzii `ld` ar avea forma:

```
ld -dynamic-linker /lib/ld-linux.so.2 -m elf_i386 -o app /usr/lib32/crt1.o /usr/lib32/crti.o app.o helpers.o -lc /usr/lib32/crtn.o
```

Argumentele comenzii de mai sus au semnificația:

- `-dynamic-linker /lib/ld-linux.so.2`: precizează loaderul / linkerul dinamic folosit pentru încărcarea executabilului dinamic
- `-m elf_i386`: se linkează fișiere pentru arhitectura x86 (32 de biți, i386)
- `/usr/lib32/crt1.o`, `/usr/lib32/crti.o`, `/usr/lib32/crtn.o`: reprezintă biblioteca de runtime C (`crt` - *C runtime*) care oferă suportul necesar pentru a putea încărca executabilul
- `-lc`: se linkează biblioteca standard C (libc)

Inspectarea fișierelor

Pentru a urmări procesul de linking, folosim utilitare de analiză statică precum `nm`, `objdump`, `readelf`.

Folosim utilitarul `nm` pentru a afișa simbolurile dintr-un fișier obiect sau un fișier executabil:

```
$ nm hello.o
00000000 T main
         U puts

$ nm hello
```

```
0804a01c B __bss_start
0804a01c b completed.7283
0804a014 D __data_start
0804a014 W data_start
08048370 t deregister_tm_clones
08048350 T _dl_relocate_static_pie
080483f0 t __do_global_dtors_aux
08049f10 t __do_global_dtors_aux_fini_array_entry
0804a018 D __dso_handle
08049f14 d _DYNAMIC
0804a01c D _edata
0804a020 B _end
080484c4 T _fini
080484d8 R _fp_hw
08048420 t frame_dummy
08049f0c t __frame_dummy_init_array_entry
0804861c r __FRAME_END__
0804a000 d _GLOBAL_OFFSET_TABLE_
         w __gmon_start__
080484f0 r __GNU_EH_FRAME_HDR
080482a8 T _init
08049f10 t __init_array_end
08049f0c t __init_array_start
080484dc R _IO_stdin_used
080484c0 T __libc_csu_fini
08048460 T __libc_csu_init
         U __libc_start_main@@GLIBC_2.0
08048426 T main
         U puts@@GLIBC_2.0
080483b0 t register_tm_clones
08048310 T _start
0804a01c D __TMC_END__
08048360 T __x86.get_pc_thunk.bx
```

Comanda `nm` afișează trei coloane:

- adresa simbolului
- secțiunea și tipul unde se găsește simbolul
- numele simbolului

Un simbol este numele unei variabile globale sau a unei funcții. Este folosit de linker pentru a face conexiunile între diferite module obiect. Simbolurile nu sunt necesare pentru executabile, de aceea executabilele pot fi stripped.

Adresa simbolului este, de fapt, offsetul în cadrul unei secțiuni pentru fișierele obiect. Și este adresa efectivă pentru executabile.

A doua coloana precizează secțiunea și tipul simbolului. Dacă este vorba de majusculă, atunci simbolul este exportat, este un simbol ce poate fi folosit de un alt modul. Dacă este vorba de literă mică, atunci simbolul nu este exportat, este propriu modulului obiect, nefolosibil în alte module. Astfel:

- `d`: simbolul este în zona de date inițializate (`.data`), neexportat
- `D`: simbolul este în zona de date inițializate (`.data`), exportat
- `t`: simbolul este în zona de cod (`.text`), neexportat
- `T`: simbolul este în zona de cod (`.text`), exportat
- `r`: simbolul este în zona de date read-only (`.rodata`), neexportat
- `R`: simbolul este în zona de date read-only (`.rodata`), exportat
- `b`: simbolul este în zona de date neinițializate (`.bss`), neexportat
- `B`: simbolul este în zona de date neinițializate (`.bss`), exportat
- `U`: simbolul este nedefinit (este folosit în modulul curent, dar este definit în alt modul)

Alte informații se găsesc în pagina de manual a utilitarul `nm`.

Cu ajutorul comenzii `objdump` dezasamblăm codul fișierelor obiect și a fișierelor executabile. Putem vedea, astfel, codul în limbaj de asamblare și funcționarea modulelor.

Comanda `readelf` este folosită pentru inspectarea fișierelor obiect sau executabile. Cu ajutorul comenzii `readelf` putem să vedem headerul fișierelor. O informație importantă în headerul fișierelor executabile o reprezintă entry pointul, adresa primei instrucțiuni executate:

```
$ readelf -h hello
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048310
  Start of program headers:          52 (bytes into file)
  Start of section headers:          8076 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         35
  Section header string table index: 34
```

Cu ajutorul comenzii `readelf` putem vedea secțiunile unui executabil / fișier obiect:

```
$ readelf -S hello
There are 35 section headers, starting at offset 0x1f8c:
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048154 000154 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048168 000168 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            08048188 000188 000024 00   A  0   0  4
[...]
```

Tot cu ajutorul comenzii `readelf` putem lista (*dump*) conținutul unei anumite secțiuni:

```
$ readelf -x .rodata hello

Hex dump of section '.rodata':
  0x080484d8 03000000 01000200 48656c6c 6f2c2057 ........Hello, W
  0x080484e8 6f726c64 2100                        orld!.
```

Majoritatea compilatoarelor oferă opțiunea de a genera și un fișier cu programul scris în limbaj de asamblare.

În cazul compilatorului `gcc` este de ajuns să adăugați flag-ul `-S` și vă va genera un fișier `*.s` cu codul aferent. În arhiva de TODO aveți un exemplu de trecere a unui program foarte simplu `hello.c` prin cele patru faze. Îl puteți testa pe un sistem Unix/Linux și pe un sistem Windows cu suport de MinGW.

```
$ make
cc  -E -o hello.i hello.c
cc -Wall -S -o hello.s hello.i
cc  -c -o hello.o hello.s
cc  -o hello hello.o

$ ls
Makefile  hello  hello.c  hello.i  hello.o  hello.s

$ ./hello
Hello, World!

$ tail -10 hello.i


# 5 "hello.c"
int main(void)
{
 puts("Hello, World!");

 return 0;
}

$ cat hello.s
    .file   "hello.c"
    .section    .rodata
.LC0:
    .string "Hello, World!"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    $.LC0, %edi
    call    puts
    movl    $0, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (Debian 5.2.1-17) 5.2.1 20150911"
    .section    .note.GNU-stack,"",@progbits

$ file hello.o
hello.o: ELF 64-bit LSB relocatable, x86-64, [...]

$ file hello
hello: ELF 64-bit LSB executable, x86-64, [...]

$ objdump -d hello.o

hello.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   bf 00 00 00 00          mov    $0x0,%edi
   9:   e8 00 00 00 00          callq  e <main+0xe>
   e:   b8 00 00 00 00          mov    $0x0,%eax
  13:   5d                      pop    %rbp
  14:   c3                      retq
```

Pentru a genera sintaxa intel pe 32 de biți, se pot folosi aceste opțiuni:

```
cc -Wall -m32 -S -masm=intel  -o hello.s hello.i
```

Dacă programele scrise în limbaje de nivel înalt ajung să fie portate ușor pentru procesoare diferite (arm, powerpc, x86, etc.), cele scrise în limbaj de asamblare sunt implementări specifice unei anumite arhitecturi. Limbaje de nivel înalt reprezintă o formă mai abstractă de rezolvare a unei probleme, din punctul de vedere al unui procesor, motiv pentru care și acestea trebuie traduse în limbaj de asamblare în cele din urmă, pentru a se putea ajunge la un binar care poate fi rulat. Mai multe detalii în laboratoarele următoare.

Exerciții

În cadrul laboratoarelor vom folosi repository-ul de Git de IOCLA: https://github.com/systems-cs-pub-ro/iocla [https://github.com/systems-cs-pub-ro/iocla]. Repository-ul este clonat pe desktopul mașinii virtuale. Pentru a îl actualiza, folosiți comanda `git pull origin master` din interiorul directorului în care se află repository-ul (`~/Desktop/iocla`). Recomandarea este să îl actualizați cât mai frecvent, înainte să începeți lucrul, pentru a vă asigura că aveți versiunea cea mai recentă. Dacă doriți să descărcați repository-ul în altă locație, folosiți comanda `git clone https://github.com/systems-cs-pub-ro/iocla ${target}` Pentru mai multe informații despre folosirea utilitarului `git`, urmați ghidul de la Git Immersion [https://gitimmersion.com/].

Cele mai multe dintre exerciții se desfășoară pe o arhitectură x86 (32 de biți, i386). Pentru a putea compila / linka pe 32 de biți atunci când sistemul vostru este pe 64 de biți, aveți nevoie de pachete specifice. Pe o distribuție Debian / Ubuntu, instalați pachetele folosind comanda:

```
sudo apt install gcc-multilib libc6-dev-i386
```

Pentru exersarea informațiilor legate de linking, parcurgem mai multe exerciții. În cea mai mare parte, acestea sunt dedicate observării procesului de linking, cele marcate cu sufixul `-tut` sau `-obs`. Unele exerciții necesită modificări pentru a repara probleme legate de linking, cele marcate cu sufixul `-fix`, altele au drept scop exersarea unor noțiuni (cele marcate cu sufixul `-diy`) sau dezvoltarea / completarea unor fișiere (cele marcate cu sufixul `-dev`). Fiecare exercițiu se găsește într-un director indexat; cele mai multe fișiere cod sursă și fișiere `Makefile` sunt deja prezente.

00. Folosirea variabilelor

Accesăm directorul `00-vars-obs/`. Vrem să urmărim folosirea variabilelor globale, exportate și neexportate.

În fișierul `hidden.c` avem variabila statică (neexportată) `hidden_value`. Variabila este modificată și citită cu ajutorul unor funcții neexportate: `init()`, `get()`, `set()`.

În fișierul `plain.c` avem variabila exportată `age`. Aceasta poate fi modificată și citită direct.

Aceste variabile sunt folosite direct (`age`) sau indirect (`hidden_value`) în fișierul `main.c`. Pentru folosirea lor, se declară funcțiile și variabilele în fișierul `ops.h`. Declararea unei funcții se face prin precizarea antetului; declararea unei variabile se face prin prefixarea cu `extern`.

Compilați și rulați programul obținut pe baza fișierelor de mai sus.

01. Linkarea unui singur fișier

Accesăm directorul `01-one-tut/`. Vrem să urmărim comenzile de linkare pentru un singur fișier cod sursă C. Fișierul sursă este `hello.c`.

În cele trei subdirectoare, se găsesc fișierele de suport pentru următoarele scenarii:

- `a-dynamic/`: crearea unui fișier executabil dinamic
- `b-static/`: crearea unui fișier executabil static
- `c-standalone/`: creare unui fișier executabil standalone, fără biblioteca standard C

În fiecare subdirector folosim comanda `make` pentru a compila fișierul executabil `hello`. Folosim comanda `file hello` pentru a urmări daca fișierul este compilat dinamic sau static.

În fișierele `Makefile`, comanda de linkare folosește `gcc`. Este comentată o comandă echivalentă care folosește direct `ld`. **Pentru a urmări folosirea directă a `ld`, putem comenta comanda `gcc` și decomenta comanda `ld`. Folosim iarăși comanda `file hello`.**

În cazul `c-standalone/`, pentru că nu folosim biblioteca standard C sau bibliotecă runtime C, trebuie să înlocuim funcționalitățile acestora. Funcționalitățile sunt înlocuite în fișierul `start.asm` și `puts.asm`. Aceste fișiere implementează, respectiv, funcția / simbolul `_start` și funcția `puts`. Funcția / simbolul `_start` este, în mod implicit, entry pointul unui program executabil. Funcția `_start` este responsabilă pentru apelul funcției `main` și încheierea programului. Pentru că nu există bibliotecă standard, aceste două fișiere sunt scrise în limbaj de asamblare și folosesc apeluri de sistem.

Adăugați, în fișierul `Makefile` din directorul `c-standalone/`, o comandă care folosește explicit `ld` pentru linkare.

Extra: Accesați directorul `01-one-diy/`. Vrem să compilăm și linkăm fișierele cod sursă din fiecare subdirector, asemănător cu ceea ce am făcut anterior. Copiați fișierele `Makefile` și actualizați-le în fiecare subdirector pentru a obține fișierul executabil.

02. Linkarea mai multor fișiere

Accesăm directorul `02-multiple-tut/`. Vrem să urmărim comenzile de linkare din fișiere multiple cod sursă C: `main.c`, `add.c`, `sub.c`.

La fel ca în exercițiile de mai sus, sunt trei subdirectoare pentru trei scenarii diferite:

- `a-no-header/`: declararea funcțiilor externe se face direct în fișierul sursă C (`main.c`)
- `b-header/`: declararea funcțiilor externe se face într-un fișier header separat (`ops.h`)
- `c-lib/`: declararea funcțiilor externe se face într-un fișier header separat, iar linkarea se face folosind o bibliotecă statică

În fiecare subdirector folosim comanda `make` pentru a compila fișierul executabil `main`.

Extra: Accesați directorul `02-multiple-diy/`. Vrem să compilăm și linkăm fișierele cod sursă din fiecare subdirector, asemănător cu ceea ce am făcut anterior. Copiați fișierele `Makefile` și actualizați-le în fiecare subdirector pentru a obține fișierul executabil.

03. Repararea entry pointului

Accesați directorul `03-entry-fix/`. Vrem să urmărim probleme de definire a funcției `main()`.

Accesați subdirectorul `a-c/`. Rulați comanda `make`, interpretați eroarea întâlnită și rezolvați-o prin editarea fișierului `hello.c`.

Accesați subdirectorul `b-asm/`. Rulați comanda `make`, interpretați eroarea întâlnită și rezolvați-o prin editarea fișierului `hello.asm`.

În subdirectoarele `c-extra-nolibc/` și `d-extra-libc/` veți găsi soluții care nu modifică codul sursă al `hello.c`. Aceste soluții modifică, în schimb, sistemul de build pentru a folosi altă funcție, diferită de `main()`, ca prima funcție a programului.

Extra: Accesați directorul `03-entry-2-fix/`. Rulați comanda `make`, interpretați eroarea întâlnită și rezolvați-o prin editarea fișierului `hello.c`.

04. Folosire simboluri (variabile și funcții)

Accesați directorul `04-var-func-fix/`. Rulați comanda `make` și rulați executabilul obținut. Interpretați erorile/eroarea întâlnite/întâlnită și rezolvați-le/rezolvați-o prin editarea fișierelor sursă.

05. Reparare problemă cu bibliotecă

Accesați directorul `05-lib-fix/`. Rulați comanda `make`, interpretați eroarea întâlnită și rezolvați-o prin editarea fișierului `Makefile`. Urmăriți fișierul `Makefile` din directorul `02-multiple-tut/c-lib/`.

06. Linkare fișier obiect (fără fișier cod sursă)

Accesați directorul `06-obj-link-dev/`. Fișierul `shop.o` expune o interfață (funcții și variabile) care permite afișarea unor mesaje. Editați fișierul `main.c` pentru a apela corespunzător interfața expusă și pentru a afișa mesajele:

```
price is 21
quantity is 42
```

Explorați interfața și conținutul funcțiilor din fisierul `shop.o` folosind `nm` și `objdump`.

Bonus. Utilizare cod python în C

În cadrul acestui exercițiu veți vedea un exemplu de ceea ce se poate face în urma legării unui anumit tip de fișiere obiect, și anume biblioteci; pentru acest exercițiu este vorba de biblioteca

```
python$(PYTHON_VERSION)
```

, unde **PYTHON_VERSION** poate să fie diferit în funcție de soluția propusă de fiecare. Pentru a putea să compilați surse veți avea nevoie de versiunea de dezvoltare pentru python; pentru instalare folosiți comanda de mai jos:

```
sudo apt-get install python$(PYTHON_VERSION)-dev
```

Înlocuiți $(PYTHON_VERSION) cu versiunea pe care o doriți(**3.8 sau mai recentă**)

Accesați directorul `bonus-c-python`. Fișierul main.c are un exemplu de cum se execută o funcție simplă de afișare a unui mesaj scrisă într-un modul python separat. Plecând de la exemplul prezentat, creați o funcție în modulul numit `my_module.py` aflat în directorul `python-modules`, care primește doi parametri reprezentând două șiruri de caractere și întoarce **poziția primei apariții a celui de-al doilea șir în cadrul primului**, dacă al doilea șir este un subșir al primului șir și **-1** în caz contrar.

Dacă funcția creată este denumită **subsir** atunci **subsir('123456789', '89')** va întoarce 7 iar **subsir('123', '4')** va întoarce -1. Practic semnătura este c

Rezultatul funcției scrisă în python va fi preluat în codul C și se va afișa un mesaj corespunzător. Urmăriți comentariile cu **TODO** din fișierele `main.c` și `my_module.py`.

Puteți să urmăriți și exemplele de aici [https://www.codeproject.com/Articles/820116/Embedding-Python-program-in-a-C-Cplusplus-code] și/sau aici [https://www.xmodulo.com/embed-python-code-in-c.html] pentru a vedea cum să preluați rezultatul funcției scrisă în python. De asemenea puteți consulta documentația de aici [https://docs.python.org/3/c-api/long.html] pentru a vedea cum să faceți conversia rezultatului la un tip de date din C.

Atenție la versiunea de python pe care o folosiți; nu este recomandată o anumită versiune însă trebuie să aveți în vedere că în funcție de soluția voastră este posibil să fie nevoie să folosiți versiune specifică. Makefile-ul folosește versiunea **3.9**.

Soluții

Soluțiile pentru exerciții sunt disponibile aici [https://elf.cs.pub.ro/asm/res/laboratoare/lab-03-sol.zip].