

Laborator 01 - Introducere

Scop

- introducerea în tematica laboratorului
- familiarizarea cu mediul și uneltele folosite în cadrul laboratorului

Cuvinte cheie

- programare de sistem, C, compilare, depanare, biblioteci
- gcc, make, gdb
- cl, nmake, Visual Studio

Materiale ajutătoare

- [lab01-slides.pdf](#)
- [lab01-refcard.pdf](#)
- [Visual Studio Tutorials](#)
- [Video Introducere](#)

Nice to read

- TLPI - Chapter 3, System Programming Concepts
- WSP4 - Chapter 1, Getting started with Windows

Desfășurarea laboratorului

Laboratorul de Sisteme de Operare este unul de [programare de sistem](#) având drept scop aprofundarea conceptelor prezentate la curs și prezentarea interfețelor de programare oferite de sistemele de operare (system API). Un laborator va prezenta un anumit set de concepte și va conține următoarele activități:

- prezentare teoretică
- parcurgerea exercițiilor rezolvate
- rezolvarea exercițiilor propuse

Pentru o desfășurare cât mai bună a laboratorului și o **înțelegere deplină** a conceptelor vă recomandăm să parcurgeți **conținutul laboratorului** de acasă. De asemenea, pentru consolidarea cunoștințelor folosiți **suportul de laborator** prezentat în paragraful următor.

Suport de laborator

- adăugați ca bookmark secțiunea [Resurse](#)
- Linux
 - [The Linux Programming Interface](#) - TLPI
- Windows
 - [Windows System Programming 4th Edition](#) - WSP4
- General
 - [lista de discuții](#)
 - canalul de IRC dedicat cursului #cs_so, de pe serverul [freenode](#).

Prezentare

Pentru a oferi o arie de cuprindere cât mai largă, laboratoarele au ca suport familiile de sisteme de operare Unix și Windows. Instanțele de sisteme de operare din familiile de mai sus alese pentru acest laborator sunt GNU/Linux, respectiv Windows 2003 Server.

În cadrul acestui laborator introductiv va fi prezentat mediul de lucru care va fi folosit în cadrul laboratorului de Sisteme de Operare cât și în rezolvarea temelor de casă.

Laboratorul folosește ca suport de programare limbajul C/C++. Pentru GNU/Linux se va folosi suita de compilatoare GCC, iar pentru Windows compilatorul Microsoft pentru C/C++ `cl`. De asemenea, pentru compilarea incrementală a surselor se vor folosi GNU make (Linux), respectiv `nmake` (Windows). Exceptând apelurile de bibliotecă standard, API-ul folosit va fi [POSIX](#), respectiv [Win32](#).

Linux

GCC

GCC este suita de compilatoare implicită pe majoritatea distribuțiilor Linux. Pentru mai multe detalii despre proiectul GCC apăsați pe butonul Show (de acum înainte secțiunile suplimentare vor fi ascunse folosind astfel de butoane).

În cadrul laboratoarelor de Sisteme de Operare ne vom concentra asupra facilităților oferite de compilator pentru limbajul C/C++. GCC suportă standardele ANSI C, ISO C99, POSIX, dar și multe extensii folosite care nu sunt incluse în niciunul din standarde; unele din aceste extensii vor fi prezentate în secțiunile ce urmează.

Utilizare GCC

Vom folosi pentru exemplificare un program simplu care tipărește la ieșirea standard un șir de caractere.

[hello.c](#)

```
#include <stdio.h>

int main(void)
{
    <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("S0, ... hello
world!\n");

    return 0;
}
```

GCC folosește pentru compilarea de programe C/C++ comanda `gcc`, respectiv `g++`. O invocare tipică este pentru compilarea unui program dintr-un singur fișier sursă, în cazul nostru `hello.c`.

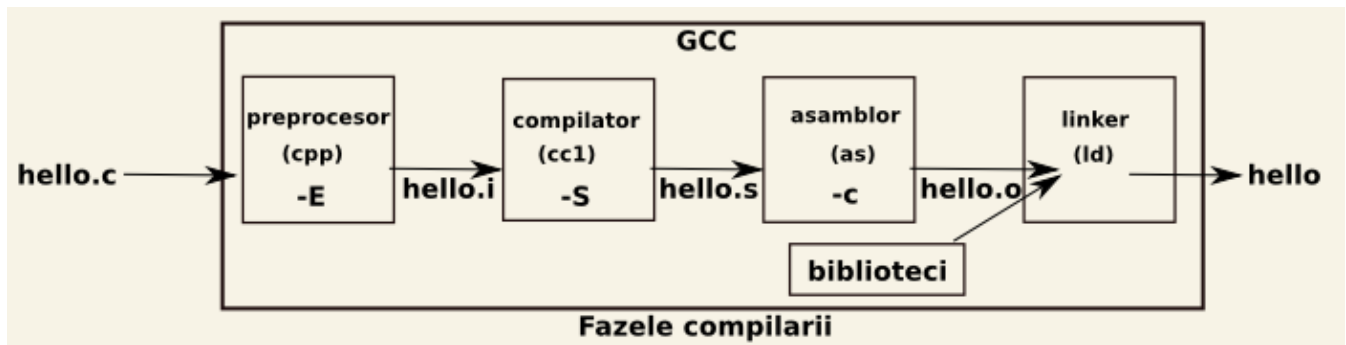
<pre>so@spook\$ ls hello.c so@spook\$ gcc hello.c so@spook\$ ls a.out hello.c so@spook\$./a.out S0, ... hello world!</pre>	<pre>so@spook\$ ls hello.c so@spook\$ gcc hello.c -o hello so@spook\$ ls hello hello.c so@spook\$./hello S0, ... hello world!</pre>
---	--

Așadar, comanda `gcc hello.c` a fost folosită pentru compilarea fișierului sursă `hello.c`. Rezultatul a fost obținerea fișierului executabil `a.out` (nume implicit utilizat de `gcc`). Dacă se dorește obținerea unui executabil cu un alt nume se poate folosi opțiunea `-o`.

În mod similar se poate folosi `g++` pentru compilarea unui program sursă C++.

Fazele compilării

Compilarea se referă la obținerea unui fișier executabil dintr-un fișier sursă. După cum am văzut în paragraful anterior comanda gcc a dus la obținerea fișierului executabil hello din fișierul sursă hello.c. Intern, gcc trece prin mai multe faze de prelucrare a fișierului sursă până la obținerea executabilului. Aceste faze sunt evidențiate în diagrama de mai jos:



Opțiuni

Implicit, la o invocare a comenzii gcc/g++ se obține din fișierul sursă un executabil. Folosind diverse opțiuni, putem opri compilarea la una din fazele intermediare astfel:

- -E - se realizează doar preprocesarea fișierului sursă
 - gcc -E hello.c ? va genera fișierul preprocesat pe care, implicit, îl va afișa la ieșirea standard.
- -S - se realizează inclusiv faza de compilare
 - gcc -S hello.c ? va genera fișierul în limbaj de asamblare hello.s
- -c - se realizează inclusiv faza de asamblare
 - gcc -c hello.c ? va genera fișierul obiect hello.o

Opțiunile de mai sus pot fi combinate cu -o pentru a specifica fișierul de ieșire.

Preprocesarea

Preprocesarea presupune înlocuirea directivelor de preprocesare din fișierul sursă C. Directivele de preprocesare încep cu #. Printre cele mai folosite sunt:

- #include ? pentru includerea fișierelor header într-un alt fișier.
- #define și #undef ? pentru definirea, respectiv anularea definirii de macrouri.
- #if, #ifdef, #ifndef, #else, #elif, #endif, pentru compilarea condiționată.
 - utile pentru comentarea bucăților mari de cod. Pentru a comenta toată funcția do_evil_things de mai jos nu putem folosi comentarii de tip C, ca în exemplul din dreapta, întrucât limbajul C nu permite comentariile imbricate. În astfel de cazuri se poate folosi directiva #if ca în exemplul din stânga.

<pre>#if 0 int do_evil_things(context_t *ctx) { int go_drink; /* set student mode ON :) */ ctx->go_drink = NO; } #endif</pre>	<pre>/* int do_evil_things(context_t *ctx) { int go_drink; /* set student mode ON :) */ ctx->go_drink = NO; } */</pre>
--	---

- utile pentru evitarea includerii de mai multe ori a unui fișier header, tehnică numită [include guard](#). Astfel, în exemplul de mai jos, dacă fișierul este inclus, a doua oară simbolul _STRING_H este deja definit de la prima includere, iar a doua operație de includere nu va avea nici un efect.

```
#ifndef _STRING_H
#define _STRING_H      1

__BEGIN_DECLS

/* Get size_t and NULL from <stddef.h>. */
#define __need_size_t
#define __need_NULL

/*
 * string related defines
 */

#endif /* string.h */
```

- `__FILE__`, `__LINE__`, `__func__` sunt înlocuite cu numele fișierului, linia curentă în fișier și numele funcției
- operatorul `#` este folosit pentru a înlocui o variabilă transmisă unui macro cu numele acesteia.

<pre>#include <stdio.h> #define show_var(a) printf("Variable %s has\nvalue %d\n", #a, a); int main(void) { int teh_var = 42; show_var(teh_var); return 0; }</pre>	<pre>so@spook\$ gcc -o show show.c so@spook\$ ls show show.c so@spook\$./show Variable teh_var has value 42</pre>
---	--

- operatorul `##` (token paste) este folosit pentru concatenarea între un argument al macrodefiniției și un alt șir de caractere sau între două argumente ale macrodefiniției.

Depanarea folosind directive de preprocesare

De multe ori, un dezvoltator va dori să poată activa sau dezactiva foarte facil afișarea de mesaje suplimentare (de informare sau de debug) în sursele sale.

Compilarea

Compilarea este faza în care din fișierul preprocesat se obține un fișier în limbaj de asamblare.

```
so@spook$ ls
hello.c
so@spook$ gcc -S hello.c
so@spook$ ls
hello.c hello.s
```

În continuare sunt prezentate, în stânga, fișierul sursă C, `hello.c` iar în dreapta fișierul în limbaj de asamblare corespunzător `hello.s`.

<pre>#include <stdio.h> int main(void) { printf("SO, ... hello world!\n"); return 0; }</pre>	<pre>.file "hello.c" .section .rodata .LC0: .string "SO, ... hello world!" .text .globl main .type main, @function main: pushl %ebp movl %esp, %ebp andl \$-16, %esp subl , %esp movl \$.LC0, (%esp) call puts movl so@spook\$ ls hello.c so@spook\$ gcc -c hello.c so@spook\$ ls hello.c hello.o , %eax leave ret .size main, .-main .ident "GCC: (Ubuntu 4.4.1-4ubuntu9) 4.4.1" .section .note.GNU-stack,"",@progbits</pre>
--	---

Asamblarea

Asamblarea este faza în care codul scris în limbaj de asamblare este tradus în *cod mașină* reprezentând codificarea binară a instrucțiunilor programului inițial. Fișierul obținut poartă numele de fișier *cod obiect*, se obține folosind opțiunea `-c` a compilatorului și are extensia `.o`.

```
void f(void);

/*
 * no definition for f here
 */

int main(void)
{
    f();
    return 0;
}
```

Editarea de legături

Pentru obținerea unui fișier executabil este necesară rezolvarea diverselor simboluri prezente în fișierul obiect. Această operație poartă denumirea de *editare de legături*, *link-editare*, *linking* sau *legare*.

<pre>void f(void); void f(void) { } int main(void) { f(); return 0; }</pre>	<pre>so@spook\$ ls sample.c so@spook\$ gcc -c -o sample sample.c so@spook\$ ls sample.c sample.o so@spook\$ gcc -o sample sample.c /tmp/cc0VreJg.o: In function `main': sample.c:(.text+0x7): undefined reference to `f' collect2: ld returned 1 exit status</pre>
---	--

<pre>so@spook\$ ls sample.c so@spook\$ gcc -c -o sample sample.c so@spook\$ ls sample.c sample.o so@spook\$ gcc -o sample sample.c so@spook\$ ls sample sample.c sample.o</pre>	<pre>#include <stdio.h> int main(void){{:laboratoare:makefile_rule.png}} { int min = 10, max = 20, midpoint; /* midpoint = min+(max-min)/2; */ midpoint = min + (max - min) >> 1; printf("The middle of interval [%d, %d] is %d\n", min, max, midpoint); return 0; }</pre>
---	--

Observăm că în partea stângă deși am obținut fișierul obiect `sample.o`, linkerul nu poate obține fișierul executabil întrucât nu găsește definiția funcției `f`. În partea dreaptă totul decurge normal, definiția funcției `f` fiind inclusă în fișierul sursă.

Activarea avertismentelor

În mod implicit, o rulare a `gcc` oferă puține avertismente utilizatorului. Pentru a activa afișarea de avertismente se folosesc opțiunile de tip `-W` cu sintaxa `-Woptiune-avertisment`. `optiune-avertisment` poate lua mai multe valori posibile printre care `return-type`, `switch`, `unused-variable`, `uninitialized`, `implicit`, `all`. Folosirea opțiunii `-Wall` înseamnă afișarea tuturor avertismentelor care pot cauza inconsistențe la rulare.

Considerăm ca fiind indispensabilă folosirea opțiunii `-Wall` pentru a putea detecta încă din momentul compilării posibilele erori. O cauză importantă a aparițiilor acestor erori o constituie sintaxa foarte permisivă a limbajului C. Sperăm ca exemplul de mai jos să justifice utilitatea folosirii opțiunii `-Wall`:

<p>middle.c</p> <pre>so@spook\$ ls middle.c so@spook\$ gcc -o middle middle.c so@spook\$./middle Middle of interval [10, 20] is 10</pre>	<pre>so@spook\$ gcc -Wall -o middle middle.c middle.c: In function ?main?: middle.c:8: warning: suggest parentheses around ?+? inside ?>>? #include <stdio.h> #include "util.h" int main(void) { f1(); f2(); return 0; }</pre>
---	--

La prima rulare, rezultatul nu e nici pe departe cel așteptat. Eroarea poate fi detectată ușor dacă includem și opțiunea `-Wall` la compilare. (operatorul `+` are prioritate în fața operatorului `>>`)

Opțiuni utile

Compilarea din mai multe fișiere

Exemplele de până acum tratează programe scrise într-un singur fișier sursă. În realitate, aplicațiile sunt complexe și scrierea întregului cod într-un singur fișier îl face greu de menținut și greu de extins. În acest sens aplicația este scrisă în mai multe fișiere sursă denumite module. Un modul conține, în mod obișnuit, funcții care îndeplinesc un rol comun.

Următoarele fișiere sunt folosite ca suport pentru a exemplifica modul de compilare a unui program provenind din mai multe fișiere sursă:

<pre> main.c #ifdef _UTIL_H #define _UTIL_H 1 void f1 (void); void f2 (void); #endif </pre>	<pre> util.h #include <stdio.h> #include "util.h" void f1(void) { printf("Current file name is %s\n", __FILE__); } </pre>
---	--

<pre> f1.c #include <stdio.h> #include "util.h" void f2(void) { printf("Current line %d in file %s\n", __LINE__, __FILE__); } </pre>	<pre> f2.c so@spook\$ ls f1.c f2.c main.c util.h so@spook\$ gcc -Wall main.c f1.c f2.c -o main so@spook\$ ls f1.c f2.c main main.c util.h so@spook\$./main Current file name f1.c Current line 8 in file f2.c </pre>
---	---

În programul de mai sus se apelează, respectiv, funcțiile `f1` și `f2` în funcția `main` pentru a afișa diverse informații. Pentru compilarea acestora se transmit toate fișierele C ca argumente către `gcc`:

```

so@spook$ ls
f1.c f2.c main.c util.h
so@spook$ gcc -Wall -c f1.c
so@spook$ gcc -Wall -c f2.c
so@spook$ gcc -Wall -c main.c
so@spook$ ls
f1.c f1.o f2.c f2.o main.c main.o util.h
so@spook$ gcc -o main main.o f1.o f2.o
so@spook$ ls
f1.c f1.o f2.c f2.o main main.c main.o util.h
so@spook$ ./main
Current file name f1.c
Current line 8 in file f2.c

```

Executabilul a fost denumit `main`; pentru acest lucru s-a folosit opțiunea `-o`.

Se observă folosirea fișierului header `util.h` pentru declararea funcțiilor `f1` și `f2`. Declararea unei funcții se realizează prin precizarea antetului. Fișierul header este inclus în fișierul `main.c` pentru ca acesta să aibă cunoștință de formatul de apel al funcțiilor `f1` și `f2`. Funcțiile `f1` și `f2` sunt definite, respectiv, în fișierele `f1.c` și `f2.c`. Codul acestora este integrat în executabil în momentul link-editării.

În general, pentru obținerea unui executabil din surse multiple se obișnuiește compilarea fiecărei surse până la modul obiect și apoi link-editarea acestora:

```
so@spook$ ls -l /usr/lib/libm.*
-rw-r--r-- 1 root root 496218 2010-01-03 15:19 /usr/lib/libm.a
lrwxrwxrwx 1 root root    14 2010-01-14 12:17 /usr/lib/libm.so -> /lib/libm.so.6
```

Se observă obținerea executabilului `main` prin legarea modulelor obiect. Această abordare are avantajul eficienței. Dacă se modifică fișierul sursă `f2.c` atunci doar acesta va trebui compilat și refăcută link-editarea. Dacă s-ar fi obținut un executabil direct din surse atunci s-ar fi compilat toate cele trei fișiere și apoi refăcută link-editarea. Timpul consumat ar fi mult mai [mare](#), în special în perioada de dezvoltare când fazele de compilare sunt dese și se dorește compilarea doar a fișierelor sursă modificate.

Scăderea timpului de dezvoltare prin compilarea numai a surselor care au fost modificate este motivația de bază pentru existența utilităților de automatizare precum `make` sau `nmake`.

Biblioteci în Linux

O bibliotecă este o colecție de funcții precompilate. În momentul în care un program are nevoie de o funcție, linker-ul va apela respectiva funcție din bibliotecă. Numele fișierului reprezentând biblioteca trebuie să aibă prefixul **lib**:

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

int main(void)
{
    double x = 1000.0;
    <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("Cubic root for %g is %g\n",
    x, cbrt(x));
    return 0;
}
```

Biblioteca matematică este denumită `libm.a` sau `libm.so`. În Linux bibliotecile sunt de două tipuri:

- **statice** - au, de obicei, extensia `.a`
- **partajate** - au extensia `.so`

Legarea se face folosind opțiunea `-l` transmisă comenzii `gcc`. Astfel, dacă se dorește folosirea unor funcții din `math.h`, trebuie legată biblioteca matematică:

<p>cbrt.c</p> <pre>so@spook\$ ls cbrt.c so@spook\$ gcc -Wall -o cbrt cbrt.c /tmp/ccwvm1zq.o: In function `main': cbrt.c:(.text+0x1b): undefined reference to `cbrt' collect2: ld returned 1 exit status so@spook\$ gcc -Wall -o cbrt -lm cbrt.c so@spook\$./cbrt Cubic root for 1000 is 10</pre>	<pre>so@spook\$ gcc -Wall -c f1.c so@spook\$ gcc -Wall -c f2.c</pre>
---	--

Se observă că, în primă fază, nu s-a rezolvat simbolul `cbrt`. După legarea bibliotecii matematice, programul s-a compilat și a rulat fără probleme.

Crearea unei biblioteci statice

Pentru crearea de biblioteci vom folosi fișierele din secțiunea [Compilarea din mai multe fișiere](#). Vom include modulele obiect rezultate din fișierele sursă `f1.c` și `f2.c` într-o bibliotecă pe care o vom folosi ulterior pentru obținerea executabilului final.

Primul pas constă în obținerea modulelor obiect asociate:

```
so@spook$ ar rc libintro.a f1.o f2.o
so@spook$ gcc -Wall main.c -o main -lintro
```



```
/usr/bin/ld: cannot find -lintro
collect2: ld returned 1 exit status
```

O bibliotecă statică este o arhivă ce conține fișiere obiect creată cu ajutorul utilitarului [ar](#) (interpretați parametrii rc).

<pre>so@spook\$ gcc -Wall main.c -o main -lintro -L. so@spook\$./main Current file name is f1.c Current line 5 in file f2.c</pre>	<pre>so@spook\$ gcc -fPIC -c f1.c so@spook\$ gcc -fPIC -c f2.c so@spook\$ gcc -shared f1.o f2.o -o libintro_shared.so so@spook\$ gcc -Wall main.c -o main -lintro_shared -L. so@spook\$./main ./main: error while loading shared libraries: libintro_shared.so: cannot open shared object file: No such file or directory</pre>
--	--

Atenție: -lintro trebuie să fie după specificarea sursei și a fișierului executabil

Linker-ul returnează eroare precizând că nu găsește biblioteca `libintro`. Aceasta deoarece linker-ul nu a fost configurat să caute și în directorul curent. Pentru aceasta se folosește opțiunea **-L**, urmată de directorul în care trebuie căutată biblioteca (în cazul nostru este vorba de directorul curent).

Dacă biblioteca se numește `libnume.a`, atunci ea va fi referită cu `-lnume`

Crearea unei biblioteci partajate

Spre deosebire de o bibliotecă statică despre care am văzut că nu este nimic altceva decât o arhivă de fișiere obiect, o bibliotecă partajată este ea însăși un fișier obiect. Crearea unei biblioteci partajate se realizează prin intermediul linker-ului. Opțiunea **-shared** indică compilatorului să creeze un obiect partajat și nu un fișier executabil. Este, de asemenea, indicată folosirea opțiunii **-fPIC** la crearea fișierelor obiect.

```
so@spook$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
so@spook$ ./main
Current file name is f1.c
Current line 5 in file f2.c
```

La rularea executabilului se poate observa că nu se poate încărca biblioteca partajată. Cauza este deosebirea dintre bibliotecile statice și bibliotecile partajate. În cazul bibliotecilor statice codul funcției de bibliotecă este copiat în codul executabil la link-editare. De partea cealaltă, în cazul bibliotecilor partajate, codul este încărcat în memorie în momentul rulării.

Astfel, în momentul rulării unui program, loader-ul (programul responsabil cu încărcarea programului în memorie), trebuie să știe unde să caute biblioteca partajată pentru a o încărca în memorie în cazul în care aceasta nu a fost încărcată deja. Loader-ul folosește câteva căi predefinite (`/lib`, `/usr/lib`, etc) și de asemenea locații definite în variabila de mediu **LD_LIBRARY_PATH**:

```
so@spook$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:. ./main
Fișierul curent este f1.c
Va aflați la linia 5 din fișierul f2.c
so@spook$ ./main
./main: error while loading shared libraries: libintro_shared.so:
cannot open shared object file: No such file or directory
```

În exemplul de mai sus variabilei de mediu `LD_LIBRARY_PATH` i-a fost adăugată calea către directorul curent rezultând în posibilitatea rulării programului. `LD_LIBRARY_PATH` va rămâne modificată cât timp va rula consola curentă. Pentru a face o modificare a unei variabile de mediu doar pentru o instanță a unui program se face atribuirea noii valori înaintea comenzii de execuție:

```
all:
    gcc -Wall hello.c -o hello
clean:
    rm -f hello
```

GNU Make

Make este un utilitar care permite automatizarea și eficientizarea sarcinilor. În mod particular este folosit pentru automatizarea compilării programelor. După cum s-a precizat, pentru obținerea unui executabil provenind din mai multe surse este ineficientă compilarea de fiecare dată a fiecărui fișier și apoi link-editarea. Se compilează fiecare fișier separat, iar la o modificare se va recompila doar fișierul modificat.

Exemplu simplu de Makefile

Utilitarul [make](#) folosește un fișier de configurare denumit `Makefile`. Un astfel de fișier conține reguli și comenzi de automatizare.

<pre> Makefile so@spook\$ make gcc -Wall hello.c -o hello so@spook\$./hello S0, ... hello world! </pre>	<pre> so@spook\$ make clean rm -f hello so@spook\$ make all gcc -Wall hello.c -o hello </pre>	<pre> so@spook\$ mv Makefile Makefile.ex1 so@spook\$ make make: *** No targets specified and no makefile found. Stop. so@spook\$ make -f Makefile.ex1 gcc -Wall hello.c -o hello so@spook\$ make -f Makefile.ex1 clean rm -f hello </pre>
--	---	---

Exemplul prezentat mai sus conține două reguli: `all` și `clean`. La rularea comenzii **make** se execută prima regulă din `Makefile` (în cazul de față `all`, nu contează în mod special denumirea). Comanda executată este `gcc -Wall hello.c -o hello`. Se poate preciza explicit ce regulă să se execute prin transmiterea ca argument comenzii `make`. (comanda **make clean** pentru a șterge executabilul `hello` și comanda **make all** pentru a obține din nou acel executabil).

În mod implicit, GNU Make caută, în ordine, fișierele `GNUmakefile`, `Makefile`, `makefile` și le analizează. Pentru a preciza ce fișier `Makefile` trebuie analizat, se folosește opțiunea `-f`. Astfel, în exemplul de mai jos, folosim fișierul `Makefile.ex1`:

```

all: hello

hello: hello.o
    gcc hello.o -o hello

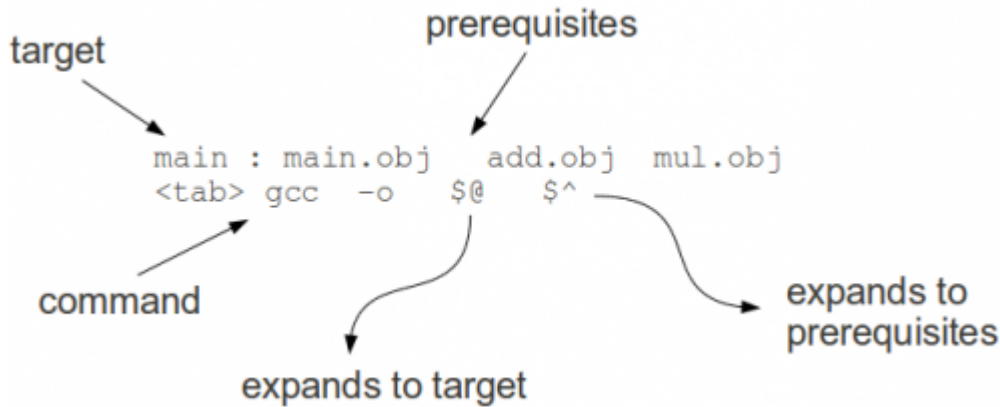
hello.o: hello.c
    gcc -Wall -c hello.c

clean:
    rm -f *.o *~ hello

```

Sintaxa unei reguli

În continuare este prezentată sintaxa unei reguli dintr-un fișier `Makefile`:



- **target** - este, de obicei, fișierul care se va obține prin rularea comenzii `command`. După cum s-a observat și din exemplul anterior, poate să fie o țintă virtuală care nu are asociat un fișier.
- **prerequisites** - reprezintă dependențele necesare pentru a urmări regula; de obicei sunt fișiere necesare pentru obținerea țintei.
- - reprezintă caracterul `tab` și trebuie neapărat folosit înaintea precizării comenzii.
- **command** - o listă de comenzi (niciuna, una, oricâte) rulate în momentul în care se trece la obținerea țintei.

Un exemplu indicat pentru un fișier `Makefile` este:

[Makefile.ex2](#)

```
so@spook$ make -f Makefile.ex2
gcc -Wall -c hello.c
gcc hello.o -o hello
```

Se observă prezența regulii `all` care va fi executată implicit.

- **all** are ca dependență `hello` și nu execută nicio comandă;
- **hello** are ca dependență `hello.o` și realizează link-editarea fișierului `hello.o`;
- **hello.o** are ca dependență `hello.c` și realizează compilarea și asamblarea fișierului `hello.c`.

Pentru obținerea executabilului se folosește comanda:

```
so@spook$ make -f Makefile.ex2
make: Nothing to be done for 'all'.
```

Funcționarea unui fișier `Makefile`

Pentru obținerea unui target trebuie satisfăcute dependențele (prerequisites) acestuia. Astfel, pentru obținerea targetului implicit (primul target), în cazul nostru `all`:

- pentru obținerea target-ului `all` trebuie obținut target-ul `hello`, care este un nume de executabil
- pentru obținerea target-ului `hello` trebuie obținut target-ul `hello.o`
- pentru obținerea target-ului `hello.o` trebuie obținut `hello.c`; acest fișier există deja, și cum acesta nu apare la rândul lui ca target în `Makefile`, nu mai trebuie obținut
- drept urmare se rulează comanda asociată obținerii `hello.o`; aceasta este `gcc -Wall -c hello.c`
- rularea comenzii duce la obținerea target-ului `hello.o`, care este folosit ca dependență pentru `hello`
- se rulează comanda `gcc hello.o -o hello` pentru obținerea executabilului `hello`
- `hello` este folosit ca dependență pentru `all`; acesta nu are asociată nici o comandă deci este automat obținut.

De remarcat este faptul că un target nu trebuie să aibă neapărat numele fișierului care se obține. Se recomandă, însă, acest lucru pentru înțelegerea mai ușoară a fișierului `Makefile`, și pentru a beneficia de faptul că `make` utilizează timpul de modificare al fișierelor pentru a decide când nu trebuie să facă nimic.

Acest format al fișierului `Makefile` are avantajul eficientizării procesului de compilare. Astfel, după ce s-a obținut executabilul `hello` conform fișierului `Makefile` anterior, o nouă rulare a `make` nu va genera nimic:

```
CC = gcc
CFLAGS = -Wall -g
```

```

all: hello

hello: hello.o
    $(CC) $^ -o $@

hello.o: hello.c
    $(CC) $(CFLAGS) -c $<

.PHONY: clean
clean:
    rm -f *.o *~ hello

```

Folosirea variabilelor

Un fișier Makefile permite folosirea de variabile. Astfel, un exemplu uzual de fișier Makefile este:

[Makefile.ex3](#)

```
main.o: main.c
```

În exemplul de mai sus au fost definite variabilele CC și CFLAGS. Variabila CC reprezintă compilatorul folosit, iar variabila CFLAGS reprezintă opțiunile (flag-urile) de compilare utilizate; în cazul de față sunt afișarea avertismentelor și compilarea cu suport de depanare. Referirea unei variabile se realizează prin intermediul construcției \$(VAR_NAME). Astfel, \$(CC) se înlocuiește cu gcc, iar \$(CFLAGS) se înlocuiește cu -Wall -g.

Variabile predefinite folosite sunt:

- \$@ se expandează la numele target-ului.
- \$^ se expandează la lista de cerințe.
- \$< se expandează la prima cerință.

Pentru mai multe detalii despre variabile consultați pagina info [1] sau manualul online [2]

Folosirea regulilor implicite

De foarte multe ori nu este nevoie să se precizeze comanda care trebuie rulată; aceasta poate fi detectată implicit.

<pre>so@spook\$(CC) \$(CFLAGS) -c -o \$@ \$<</pre>	<pre> CC = gcc CFLAGS = -Wall -g all: hello hello: hello.o hello.o: hello.c .PHONY: clean clean: rm -f *.o *~ hello </pre>
---	--

Astfel, fișierul Makefile.ex2 de mai sus poate fi simplificat, folosind reguli implicite, ca mai jos:

<pre> Makefile.ex4 so@spook\$ make -f Makefile.ex4 gcc -Wall -g -c -o hello.o hello.c gcc hello.o -o hello CC = gcc CFLAGS = -Wall -g all: hello hello: hello.o .PHONY: clean clean: rm -f *.o *~ hello </pre>	<pre> Makefile.ex5 so@spook\$ make -f Makefile.ex5 gcc -Wall -g -c -o hello.o hello.c gcc hello.o -o hello so@spook\$ ls hello.c so@spook\$ make hello cc hello.c -o hello </pre>
--	---

De remarcat faptul că dacă avem un singur fișier sursă nici nu trebuie să existe un fișier Makefile pentru a obține executabilul dorit.

```

CC = gcc                # compilatorul folosit
CFLAGS = -Wall -g      # optiunile pentru compilare
LDLIBS = -lelf         # optiunile pentru linking

# creeaza executabilele client si server
all: client server

# leaga modulele client.o user.o sock.o in executabilul client
client: client.o user.o sock.o log.o

# leaga modulele server.o cli_handler.o sock.o in executabilul server
server: server.o cli_handler.o sock.o log.o

# compileaza fisierul client.c in modulul obiect client.o
client.o: client.c sock.h user.h log.h

# compileaza fisierul user.c in modulul obiect user.o
user.o: user.c user.h

# compileaza fisierul sock.c in modulul obiect sock.o
sock.o: sock.c sock.h

# compileaza fisierul server.c in modulul obiect server.o
server.o: server.c cli_handler.h sock.h log.h

# compileaza fisierul cli_handler.c in modulul obiect cli_handler.o
cli_handler.o: cli_handler.c cli_handler.h

# compileaza fisierul log.c in modulul obiect log.o
log.o: log.c log.h

.PHONY: clean

clean:
    rm -fr *~ *.o server client

```

Pentru mai multe detalii despre reguli implicite consultați pagina info [3] sau manualul online [4].

Folosind toate facilitățile de până acum, ne propunem compilarea unui executabil client și a unui executabil server.

Fișierele folosite sunt:

- executabilul server depinde de fișierele C - server.c, sock.c, cli_handler.c, log.c, sock.h, cli_handler.h, log.h;
- executabilul client depinde de fișierele C - client.c, sock.c, user.c, log.c, sock.h, user.h, log.h;

Dorim, așadar, obținerea executabilelor client și server pentru rularea celor două entități. Structura fișierului Makefile

este prezentată mai jos:

[Makefile.ex6](#)

```
so@spook$ make -f Makefile.ex6
gcc -Wall -g -c -o client.o client.c
gcc -Wall -g -c -o user.o user.c
gcc -Wall -g -c -o sock.o sock.c
gcc -Wall -g -c -o log.o log.c
gcc client.o user.o sock.o log.o -lefence -o client
gcc -Wall -g -c -o server.o server.c
gcc -Wall -g -c -o cli_handler.o cli_handler.c
gcc server.o cli_handler.o sock.o log.o -lefence -o server
```

Pentru obținerea executabilelor server și client se folosește:

```
#include <stdio.h>

int main(void)
{
    <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("Hello, world!\n");
    return 0;
}
```

Regulile implicite intră în vigoare și se obțin, pe rând, fișierele obiect și fișierele executabile. Variabila LDLIBS este folosită pentru a preciza bibliotecile cu care se face link-editarea pentru obținerea executabilului.

Depanarea programelor

Există câteva unelte GNU care pot fi folosite atunci când nu reușim să facem un program să ne asculte. **gdb**, acronimul de la "Gnu Debugger" este probabil cel mai util dintre ele, dar există și altele, cum ar fi ElectricFence, gprof sau mtrace. gdb este prezentat pe scurt [aici](#).

Windows

Compilatorul Microsoft cl.exe

Soluția folosită pentru platforma Windows în cadrul acestui laborator este `cl.exe`, compilatorul Microsoft pentru C/C++. Recomandăm instalarea Microsoft Visual C++ Express 2008 (9.0) (versiunea Professional a Visual C++ este disponibilă gratuit în cadrul MSDNAA). Programele C/C++ pot fi compilate prin intermediul interfeței grafice sau în linie de comandă. În cele ce urmează vom prezenta compilarea folosind linia de comandă. În Windows fișierele cod obiect au extensia `*.obj`.

<pre>hello.c cl hello.c</pre>	<pre>\$ cl /? /* list of options for compiler */ \$ link /? /* list of options for linker */ cl /Fomyobj.obj /c mysrc.c</pre>
-------------------------------	---

Se vor prezenta mai jos o serie de opțiuni uzuale:

- **/Wall** - enable all warnings
- **/LIBPATH:** - această opțiune indică linker-ului să caute și în directorul dir bibliotecile pe care trebuie să le folosească programul; opțiunea se folosește după `/link`
- **/I** - caută și în acest director fișierele incluse prin directiva `include`
- **/c** - se va face numai compilarea, adică se va omite etapa de link-editare.
- **/D** - definirea unui macro de la compilare

<p>Opțiuni privind optimizarea codului:</p> <ul style="list-style-type: none"> - /O1 minimize space - /O2 maximize speed - /Os favor code space - /Ot favor code speed - /Od disable optimizations (default) - /Og enable global optimization 	<p>Setarea numelui pentru diferite fișiere de ieșire:</p> <ul style="list-style-type: none"> - /Fo nume fișier obiect - /Fa nume fișier în cod de asamblare - /Fp nume fișier header precompilat - /Fe nume fișier executabil
---	---

Exemple:

- Creare fișier obiect `myobj.obj` din sursa `mysrc.c`: `cl /Famyasm.asm /FA /c mysrc.c`
- Creare fișier `myasm.asm` în cod asamblor din sursa `mysrc.c`: `>lib /out:<nume.lib> <lista fișiere obiecte>`

Lista completă de opțiuni o puteți găsi [aici](#)

Biblioteci în Windows

Crearea unor biblioteci statice

Pentru a crea biblioteci statice se folosește comanda `lib`

```
# obținem fișierul obiect f1.obj din sursa f1.c
>cl /c f1.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42 for 80x86
Copyright (C) Microsoft Corporation. All rights reserved.

f1.c

#obținem fișierul f2.obj din sursa f2.c
>cl /c f2.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42 for 80x86
Copyright (C) Microsoft Corporation. All rights reserved.

f2.c

>cl /c main.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42 for 80x86
Copyright (C) Microsoft Corporation. All rights reserved.

main.c

#obținem biblioteca statică intro.lib din f1.obj și f2.obj
>lib /out:intro.lib f1.obj f2.obj
Microsoft (R) Library Manager Version 8.00.50727.42
Copyright (C) Microsoft Corporation. All rights reserved.

#intro.lib este compilat împreună cu main.obj pentru a obține main.exe
>cl main.obj intro.lib
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42 for 80x86
Copyright (C) Microsoft Corporation. All rights reserved.

Microsoft (R) Incremental Linker Version 8.00.50727.42
Copyright (C) Microsoft Corporation. All rights reserved.

/out:main.exe
```

```
main.obj
intro.lib
```

Vom considera exemplul folosit pentru crearea de biblioteci în Linux (main.c, util.h, f1.c, f2.c):

```
#include <stdio.h>

#include "funs.h"

#define DLL_IMPORTS

int main(void)
{
    f1();
    f2();

    return 0;
}
```

Pentru obținerea unei biblioteci statice folosim comanda `lib`. Argumentul `/out`: precizează numele bibliotecii statice de ieșire. Biblioteca are de obicei extensia `*.lib`. Pentru obținerea executabilului se folosește `cl` care primește ca argumente fișierele obiect și bibliotecile care conțin funcțiile dorite.

Crearea unor biblioteci partajate

Bibliotecile partajate din Linux au ca echivalent bibliotecile DLL (Dynamic Link Library) în Windows. Crearea unei biblioteci partajate pe Windows este mai complicată decât pe Linux. Pe de o parte pentru că în afara bibliotecii partajate (`dll`), mai trebuie creată o bibliotecă de import (`lib`). Pe de altă parte, legarea bibliotecii partajate presupune exportarea explicită a simbolurilor (funcții, variabile) care vor fi folosite.

Pentru precizarea simbolurilor care vor fi exportate de bibliotecă se folosesc identificatori predefiniți:

- **`__declspec(dllexport)`**, este folosit pentru a importa o funcție dintr-o bibliotecă.
- **`__declspec(dllimport)`**, este folosit pentru a exporta o funcție dintr-o bibliotecă.

Exemplul de mai jos prezintă trei programe: două dintre ele vor fi legate într-o bibliotecă partajată, iar celălalt conține codul de utilizare a funcțiilor exportate.

<pre>main.c 1 #ifndef FUNCS_H #define FUNCS_H 1 #ifdef DLL_IMPORTS #define DLL_DECLSPEC __declspec(dllimport) #else #define DLL_DECLSPEC __declspec(dllexport) #endif DLL_DECLSPEC void f1 (void); DLL_DECLSPEC void f2 (void); #endif</pre>	<pre>funs.h #include <stdio.h> #include "funs.h" void f1(void) { printf("Current file name is %s\n", __FILE__); }</pre>
---	---

<pre> f1.c #include <stdio.h> #include "funs.h" void f2(void) { printf("Current line %d in file %s\n", __LINE__, __FILE__); } </pre>	<pre> f2.c >cl /LD f1.obj f2.obj Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42 for 80x86 Copyright (C) Microsoft Corporation. All rights reserved. Microsoft (R) Incremental Linker Version 8.00.50727.42 Copyright (C) Microsoft Corporation. All rights reserved. /out:f1.dll /dll /implib:f1.lib f1.obj f2.obj Creating library f1.lib and object f1.exp >cl main.obj f1.lib Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42 for 80x86 Copyright (C) Microsoft Corporation. All rights reserved. Microsoft (R) Incremental Linker Version 8.00.50727.42 Copyright (C) Microsoft Corporation. All rights reserved. /out:main.exe main.obj f1.lib </pre>
--	---

Așadar, pentru crearea bibliotecii partajate și utilizarea acesteia de către programul main parcurgem următorii pași:

- f1.c va exporta funcția f1() folosind **__declspec(dllexport)**
- f2.c va exporta funcția f2() folosind **__declspec(dllexport)**
- main.c va importa funcțiile f1() și f2() folosind **__declspec(dllimport)**
- după obținerea fișierelor obiect f1.obj și f2.obj acestea vor fi folosite la crearea bibliotecii partajate folosind opțiunea /LD a comenzii cl.
- în final legăm main.obj cu biblioteca partajată și obținem main.exe

```
>link /nologo /dll /out:intro.dll /implib:intro.lib f1.obj f2.obj
    Creating library intro.lib and object intro.exp
```

```
>link /nologo /out:main.exe main.obj intro.lib
```

```
>main.exe
Current file name is f1.c
Current line 6 in file f2.c
```

Alternativ, biblioteca poate fi obținută cu ajutorul comenzii link:

```
OBJ_LIST = parser.tab.obj parser.yy.obj
CFLAGS = /nologo /W4 /EHsc /Za
```

```
EXE_NAMES = CUseParser.exe UseParser.exe DisplayStructure.exe
```

```
all : $(EXE_NAMES)
```

```
CUseParser.exe : CUseParser.obj $(OBJ_LIST)
```

```
$(CPP) $(CFLAGS) /Fe$@ $**

UseParser.exe : UseParser.obj $(OBJ_LIST)
$(CPP) $(CFLAGS) /Fe$@ $**

DisplayStructure.exe : DisplayStructure.obj $(OBJ_LIST)
$(CPP) $(CFLAGS) /Fe$@ $**

clean : exe_clean obj_clean

obj_clean :
del *.obj

exe_clean :
del $(EXE_NAMES)
```

Nmake

Nmake este utilitarul folosit pentru compilare incrementală pe Windows. Nmake are o sintaxă foarte asemănătoare cu Make. Un exemplu simplu de makefile este cel atașat parser-ului de la tema 1:

Makefile

```
win\VS Tutorial\Debug>"Hello World.exe"
```

Nmake oferă următoarele variabile speciale:

Macro	Semnificație
\$@	numele țintei curente
\$*	numele țintei curente mai puțin extensia
\$**	toate dependențele unei ținte
\$?	toate dependențele mai vechi decât ținta

Exerciții

În rezolvarea laboratorului folosiți arhiva de sarcini [lab01-tasks.zip](#)

Tastați CTRL+ALT+Insert pentru a vă loga pe Windows

Windows

1. (3 puncte) Utilizare Visual Studio

1. (1 punct) Compilare și rulare

- Intrați în directorul win/VS Tutorial și dați dublu-click pe fișierul *.sln pentru a deschide proiectul Visual Studio.
- Pentru a compila proiectul:
 - Build Build Solution (F7)
- Pentru a rula proiectul:
 - Debug Start Without Debugging (Ctrl + F5)
- Rulați executabilul din linia de comandă:
 - Porniți command prompt-ul de Visual Studio: Tools\Visual Studio Command Prompt (sau folosiți linkul de pe Desktop)
 - Navigați până ajungeți în folderul **Debug** din directorul rădăcină al proiectului
 - Acum puteți rula executabilul:> cl hello.c

2. (1 punct) Creare proiect nou:

- File New Project
 - Project types Win32
 - Visual Studio installed templates Win32 Console Application
 - Application Settings Console Application, Empty Project
- Adăugați fișier-ul VS Tutorial/debug.c la proiect
 - Source file Right Click Add Existing Item
- Compilați
- Pentru a vedea prima eroare: F8
- Modificați antetul funcției f astfel încât să întoarcă int

3. (1 punct) Debugging

- Compilați și rulați proiectul anterior. Programul ar trebui să afișeze valoarea salvată în variabila bug.
 - Adăugați un breakpoint la funcția f:
 - Click la linia cu definiția funcției f (linia 6), apoi F9
 - Rulați în mediul de debug: F5
 - Setați watch-uri pentru variabilele a, b, c, bug:
 - Debug QuickWatch Add
 - Rulați step-by-step programul:
 - Debug Step Over (F10)
 - Observați valoarea variabile bug în momentul în care apare problema, cât și mesajele afișate în fereastra Output
 - Remediați problema și rulați din nou programul
4. Mai multe informații utile despre Visual Studio găsiți [aici](#).
2. (**4 puncte**) Makefiles
- **Atenție** Acest set de exerciții se rulează din command-shell-ul de Visual Studio (**nu** cmd.exe).
 - Găsiți link la acesta pe Desktop sau accesând Tools Visual Studio Command Prompt
1. (**1 punct**) Compilarea unui singur fișier
- Intrați în directorul win/1-hello/.
 - Folosind **cl** obțineți și rulați executabilul hello.
 - > hello.exe
 - > cl /Fehello_win hello.c
 - Folosind **cl** obțineți și rulați executabilul hello_win.
 - > hello_win.exe
 - > nmake
 - Rămâneți în directorul win/1-hello/.
 - Analizați fișierul Makefile.
 - Folosind nmake obțineți și rulați executabilul hello.
 - > hello.exe
 - \$ gcc hello.c
2. (**1 punct**) Compilarea din mai multe surse
- Intrați în directorul win/2-debug/.
 - Analizați fișierele main.c și add.c. (Hint: type main.c)
 - Completați fișierul **Makefile.ndbg** astfel încât:
 - să obțineți obiecte din sursele main.c și add.c.
 - să obțineți executabilul main.exe din obiectele creat.
 - Completați fișierul **Makefile.dbg** astfel încât:
 - să compilați cu simbolul DEBUG__ definit.
 - să obțineți obiecte din sursele main.c și add.c și executabilul main.exe (ca la subpunctul precedent)
 - **Hints**
 - Înlocuiți "# TODO ?" cu regulile necesare
 - Folosiți **cl** /? pentru a determina cum se definește un macro în linia de comandă.
 - Revedeți secțiunea **cl**.
3. (**2 puncte**)
1. Creare bibliotecă statică
- Intrați în directorul win/3-bounds/
 - Analizați fișierele bounds.c, min.c și max.c
 - Completați fișierul **Makefile.static** astfel încât:
 - La rularea nmake bounds_static.lib să se creeze biblioteca statică **bounds_static.lib**. Biblioteca va conține fișierele obiect asociate fișierelor min.c și max.c
 - La rularea comenzii nmake să se creeze fișierul executabil bounds_static.exe obținut din legarea fișierului obiect corespunzător fișierului bounds.c cu biblioteca bounds_static.lib
 - **Hints**
 - Revedeți secțiunea [crearea unei biblioteci statice](#)
 - Chemați asistentul înainte de a trece la subpunctul următor.
2. Creare bibliotecă dinamică
- Rămâneți în directorul win/3-bounds/
 - Completați fișierul **Makefile.dynamic** astfel încât:
 - La rularea nmake bounds_dynamic.dll să se creeze biblioteca dinamică **bounds_dynamic.dll**. Biblioteca va conține fișierele obiect asociate fișierelor min.c și max.c
 - La rularea comenzii nmake să se creeze fișierul executabil bounds_dynamic.exe obținut prin legarea fișierului obiect corespunzător fișierului bounds.c cu biblioteca bounds_dynamic.dll
 - **Hints**
 - Revedeți secțiunea [crearea unei biblioteci dinamice](#)
 - Nu uitați să adăugați **__declspec(dllimport)** și **__declspec(dllexport)** la **antetele** funcțiilor
 - La crearea executabilului final aveți grijă să vă legați cu fișierul ".lib", nu ".dll"
 - Dacă folosiți cl pentru a crea biblioteca dinamică, puteți folosi '/Fe' pentru a seta numele bibliotecii

Linux

1. (**3 puncte**) Fișiere 'make'
 1. (**1 punct**) Compilarea unui singur fișier
 - Intrați în directorul `lin/1-hello/`.
 - Analizați fișierul `hello.c`.
 - Folosind [gcc](#) obțineți și rulați executabilul `a.out`.
 - `$./a.out`
 - `$ gcc -o hello hello.c`
 - Folosind [gcc](#) obțineți executabilul `hello`.
 - `$./hello`
 - `$./hello`
 2. (**2 puncte**) Creare biblioteci
 1. Creare bibliotecă statică
 - Intrați în directorul `lin/2-lib/`
 - Completați fișierul `Makefile_static` astfel încât:
 - La rularea comenzii `make libhexdump_static` să creeze bibliotecă statică `libhexdump_static.a`. Bibliotecă va conține fișierele obiect asociate fișierelor `hexdump.c` și `sample.c`
 - La rularea comenzii `make` să creeze executabilul `main_static` obținut din legarea fișierului obiect corespunzător lui `main.c` cu bibliotecă `libhexdump_static.a`.
 - **Hints**
 - Folosiți ar pentru a crea bibliotecă statică
 - Revedeți secțiunea crearea unei [biblioteci statice](#).
 2. Creare bibliotecă dinamică
 - Rămâneți în directorul `lin/2-lib/`
 - Completați fișierul `Makefile_dynamic` reguli astfel încât:
 - La rularea comenzii `make libhexdump_dynamic` să creeze bibliotecă dinamică `libhexdump_dynamic.so`. Bibliotecă va conține fișierele obiect asociate fișierelor `hexdump.c` și `sample.c`
 - La rularea comenzii `make` pe lângă executabilul `main_dynamic` obținut anterior să se creeze și executabilul `main_dynamic` obținut din legarea fișierului obiect corespunzător lui `main.c` cu bibliotecă partajată `libhexdump_dynamic.so`.
 - **Hints**
 - Revedeți secțiunea despre crearea unei [biblioteci dinamice](#).

BONUS

1. **1 so karma** Compilare din mai multe surse, opțiuni la compilare.
 - Intrați în directorul `lin/3-ops/`.
 - Analizați fișierele `ops.c`, `mul.c` și `add.c`
 - Fișierul `ops.c`, se folosește de funcțiile definite în `mul.c` și `add.c` pentru a realiza operații de adunare și înmulțire simple.
 - Creați fișierul `Makefile`, astfel încât:
 - obțineți fișierele obiect din sursele `mul.o`, `add.o` și `ops.o`.
 - obțineți executabilul `ops` din obiectele create.
 - **Observați** rezultatul obținut pentru sumă și înmulțire. Este corect? Rezolvați.
 - Rămâneți în directorul `lin/3-ops/`
 - La compilarea fișierului `ops.c` definiți simbolul `HAVE_MATH`.
 - Obțineți și rulați executabilul `ops`
 - **Hints**
 - Revedeți secțiunea despre [compilarea mai multor fișiere](#).
 - Pentru definirea simbolurilor în linia de comandă se folosește opțiunea `-D`.
 - Pentru a folosi funcția `pow` trebuie să includeți fișierul `math.h` și să legați bibliotecă `libm`.
 - Legarea unei biblioteci se face folosind opțiunea `-l`.
2. **1 so karma** Utilizare `gdb`
 - Intrați în directorul `lin/4-gdb/`
 - Analizați fișierul `fault.c`
 - Completați fișierul `Makefile` astfel încât la rularea comenzii `make` să se obțină fișierul executabil `fault`.
 - Compilați
 - Folosiți [gdb](#) pentru a determina cauza erorilor din fișierul `fault.c`
 - **Hints**
 - Citiți secțiunea [GDB](#)
 - Folosiți opțiunea `-g` pentru a compila sursa cu simbolurile de debug incluse.
 - Folosiți comanda `print` pentru a printa valorile variabilelor când faceți depanarea.

3. **1 so karma** Editare de legături
 - Intrați în directorul `lin/5-linker/`
 - Analizați fișierele `main.c` și `str.c`.
 - De ce nu obținem o eroare de compilare?
 - Rulați programul `main` și explicați rezultatele.

EXTRA

- [JNI](#)

Soluții

[lab01-sol.zip](#)

Resurse utile

- Linux
 1. [GCC online documentation](#)
 2. [The C Preprocessor](#)
 3. [The GNU C Library](#)
 4. [Program Library HOWTO](#)
 5. [GNU make manual](#)
 6. [GDB documentation](#)
- Windows
 1. [Visual C++ Express](#)
 2. [Nmake tool](#)
 3. [Nmake Macros](#)
 4. [Dynamic link library](#)
 5. [Creating and using DDL's](#)
 6. [Dynamic libraries](#)

From:

<http://elf.cs.pub.ro/so/wiki/> - **Sisteme de Operare**

Permanent link:

<http://elf.cs.pub.ro/so/wiki/laboratoare/laborator-01>

Last update: **2011/02/27 23:34**