USO
just crunch it

Cursul 9

9

Compilarea şi execuția programelor

15 decembrie 2008

Motto

There are only two kinds of programming languages: those people always bitch about and those nobody uses.

Bjarne Stroustrup

"Regression testing"? What's that? If it compiles, it is good, if it boots up it is perfect.

Linus Torvalds

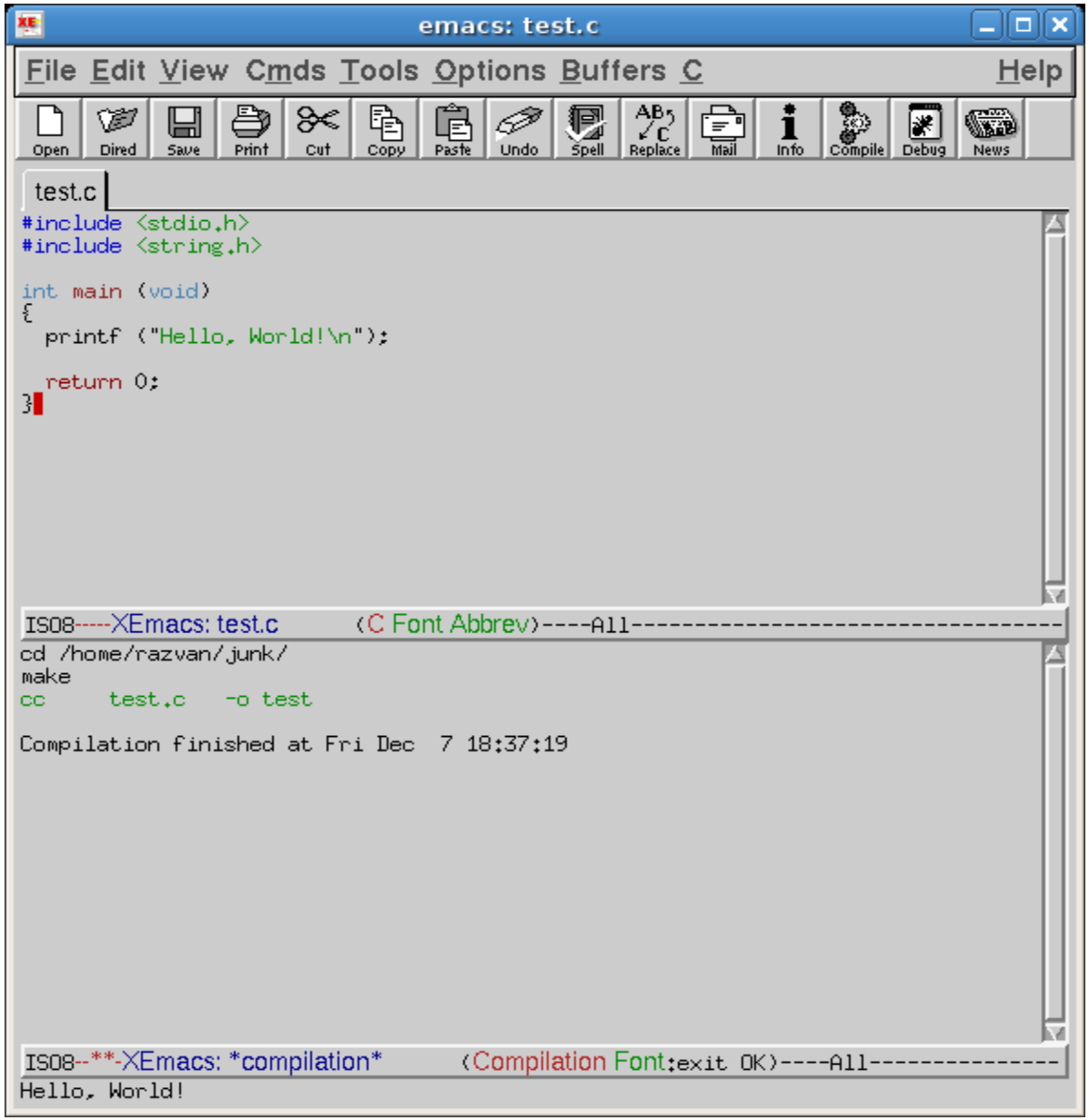
Programe

- Ce este un program?
  - Fişier executabil binar (conținutul este **cod maşină**)
- Care este faza inițială în care se găseşte un program?
  - Fişier **cod sursă** – fişier text ce conține implementarea programului într-un **limbaj de programare**
    - C - app.c, app.h
    - C++ - app.cpp, app.cxx, app.C, app.hpp
    - Java - Application.java
    - Perl, Python, Lisp, etc.
  - Se mai numeşte simplu **sursă** (sursă C, sursă Java, etc.)
  - **Scriere de cod sursă** sau scriere de cod
- La început programele erau scrise direct în cod maşină (binar) (cartele perforate)
  - Anevoios
  - Dificil de întreținut

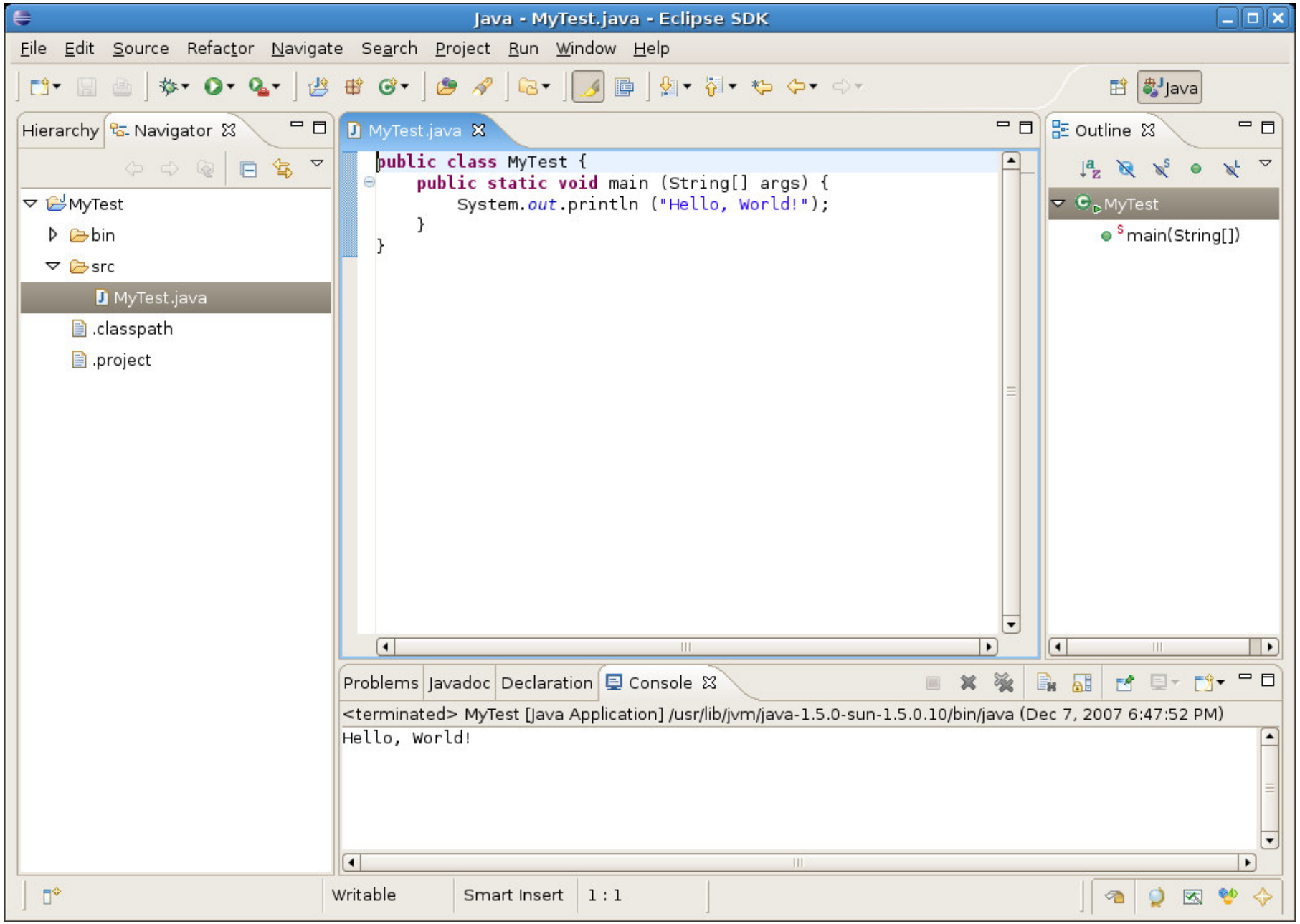
Editoare

- Scrierea de fişiere text (în particular cod sursă)
- Editoare
  - simple – one tool for the job
    - vim, Emacs, nano, joe
    - notepad++, notepad2, ultraedit, Crimson Editor
  - integrate (IDE)
    - Visual Studio 2008 (varianta Visual Studio Express Edition este free)
    - Eclipse, NetBeans
    - Emacs, Kdevelop
- Facilități
  - syntax higlighting, auto indentation, utilitare pentru debugging integrate
  - code folding, code completion (autocompletion)

XEmacs


emacs: test.c
File Edit View Cmds Tools Options Buffers C Help
Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail Info Compile Debug News
test.c
#include <stdio.h>
#include <string.h>
int main (void)
{
printf ("Hello, World!\n");
return 0;
}
ISO8----XEmacs: test.c (C Font Abbrev)----All----------
cd /home/razvan/junk/
make
cc test.c -o test
Compilation finished at Fri Dec 7 18:37:19
ISO8--**-XEmacs: *compilation* (Compilation Font:exit OK)----All----------
Hello, World!

Eclipse


Java - MyTest.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Java
Hierarchy
Navigator
MyTest
bin
src
MyTest.java
.classpath
.project
MyTest.java
public class MyTest {
public static void main (String[] args) {
System.out.println ("Hello, World!");
}
}
Outline
MyTest
main(String[])
Problems
Javadoc
Declaration
Console
<terminated> MyTest [Java Application] /usr/lib/jvm/java-1.5.0-sun-1.5.0.10/bin/java (Dec 7, 2007 6:47:52 PM)
Hello, World!
Writable
Smart Insert
1 : 1

Compilare şi interpretare

- Un fişier cod sursă poate fi **compilat** sau **interpretat**

- Care este deosebirea între compilare şi interpretare?
  - Compilare: codul sursă este translatat de un program denumit **compilator** în cod maşină, după care poate fi executat
  - Interpretare: un program este executat direct din cod sursă prin intermediul unui **interpretor**

- Orice limbaj poate fi compilat sau interpretat
  - limbaj (proiectare) != implementarea unui limbaj (compilator, interpretor)

- Multe limbaje au atât compilatoare, cât şi translatoare: C, Lisp
  - GCC (GNU Compiler Collection) - compilator de C, C++, Ada, Fortran
  - MSVC (Microsoft Visual C) - compilator de C, C++
  - CommonLisp - interpretor de Lisp

Compilare şi interpretare (2)

- Limbaje tradițional compilate
  - C, C++, Objective-C, Pascal, Java, Ada
- Limbaj tradițional interpretate (limbaje de scripting)
  - Perl, PHP, Python, Lisp, Ruby, shell scripting (bash, csh, ksh)
- Avantaje/dezavantaje
  - Interpretare
    - Sunt, în general, mai uşor de înțeles de programator
    - Debugging facil
    - Execuție lentă
  - Compilare
    - Debugging greoi (folosirea unui depanator – debugger)
    - Viteză mare de execuție

Compilare

- Un compilator este un translator
  - Translatează codul sursă în cod obiect; de la un limbaj de nivel înalt în cod maşină
- Un compilator va genera de obicei un executabil

```
F:\code>cl simple.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 14.00.50727.42 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.
simple.c
Microsoft (R) Incremental Linker Version 8.00.50727.42
Copyright (C) Microsoft Corporation.  All rights reserved.
/out:simple.exe
simple.obj

razvan@asgard:~/code$ ls
main.c
razvan@asgard:~/code$ gcc main.c
razvan@asgard:~/code$ ls
a.out  main.c
razvan@asgard:~/code$ file a.out
a.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.2.0, dynamically linked
    (uses shared libs), not stripped
```

Fazele compilării

- Componentă/utilitar specializat pentru fiecare fază

- Preprocesare: **cpp**
  - înlocuirea directivelor de preprocesare (#include, #define)
  - nu este specific tuturor limbajelor
- Compilare efectivă: **gcc**
  - translatarea codului sursă (preprocesat) în **limbaj de asamblare**
- Asamblare: **as**
  - translaterea fişierului în limbaj de asamblare într-un modul obiect (cod maşină)
- Linking: **ld**
  - legarea mai multor module obiect şi biblioteci într-un modul de sine stătător (de obicei un fişier executabil)

Fişier cod sursă

- Componenta vizibilă programatorului
- Declarații de variabile, funcții, clase etc.
- Sintaxa impusă de limbajul respectiv
- De ce i se spune **cod** sursă?
  - pentru că nu este limbaj natural; nu este uşor inteligibil
- Ce înseamnă **coding style**?
  - folosirea unui anumit tip de aranjare a codului sursă în fişier
  - denumirea variabilelor
  - utilizarea comentariilor
  - coding style bun -> uşor de înțeles de alți programatori, uşor de menținut
  - stiluri diferite, principii comune
    - indentare
    - cod relaxat şi lizibil – folosire spații, linii libere
    - denumiri intuitive ale funcțiilor şi variabilelor
    - modularitate (o funcție nouă pentru o sarcină nouă; nu se recomandă funcții main kilometrice)
    - documentarea surselor (comentare)

Preprocesare

- Fişier cod sursă

```
razvan@asgard:~/code$ cat simple.c
#include <stdio.h>

#define ITERATIONS      10
#define INITIAL         1
#define HEADER_STRING   "Number sum: "

int main (void)
{
    int i;
    int sum;

    printf (HEADER_STRING);
    for (i = INITIAL; i < INITIAL + ITERATIONS;
    i++)
            sum += i;

    printf ("%d\n", sum);
    return 0;
}
```

- Fişier preprocesat

```
razvan@asgard:~/code$ gcc -E -o simple.i simple.c
razvan@asgard:~/code$ cat simple.i
# 1 "simple.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "simple.c"
# 1 "/usr/include/stdio.h" 1 3 4
.............

int main (void)
{
        int i;
        int sum;

        printf ("Number sum: ");
        for (i = 1; i < 1 + 10; i++)
                sum += i;

        printf ("%d\n", sum);
        return 0;
}
```

Compilare

- Ține cont de **sintaxa** şi **semantica** programului
  - sintaxă: programul respectă un set de reguli (o anumită gramatică)
    - operatorul '+' poate avea doi operanzi sau poate avea unul singur
  - semantică: instrucțiunile din program trebuie să aibă sens
    - se adună un număr cu un alt număr, nu un număr cu un şir de caractere

- Program incorect din punct de vedere sintactic sau semantic -> **erori la compilare**:

```
razvan@asgard:~/code$ gcc main.c
main.c: In function `main':
main.c:6: error: `i' undeclared (first use in this function)
main.c:6: error: (Each undeclared identifier is reported only once
main.c:6: error: for each function it appears in.)
```

Compilare (2)

- **Fişier cod sursă**

```
#include <stdio.h>

#define ITERATIONS      10
#define INITIAL         1
#define HEADER_STRING   "Number sum: "

int main (void)
{
    int i;
    int sum;

    sum = 0;
    printf (HEADER_STRING);
    for (i = INITIAL; i < INITIAL +
    ITERATIONS; i++)
           sum += i;

    printf ("%d\n", sum);

    return 0;
}
```

- **Fişier după compilare**

```
razvan@asgard:~/code$ gcc -S simple.c
razvan@asgard:~/code$ cat simple.s
        .file   "simple.c"
        .section        .rodata
.LC0:
        .string "Number sum: "
.LC1:
        .string "%d\n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        andl    $-16, %esp
        …
        movl    $0, -8(%ebp)
        movl    $.LC0, (%esp)
        call    printf
        …
```

Compilare (3)

- Translatarea fişierului cod sursă (eventual preprocesat) în fişier în limbaj de asamblare
- Cunoştințe despre arhitectura sistemului de calcul.
- Cross-compiling
  - compilarea unui fişier pentru a fi folosit pe un sistem diferit de cel local
- Optimizări (îmbunătățiri) în faza compilării
  - cod care se execută mai rapid, care să consume mai puțină memorie, mai puțină putere

```
razvan@asgard:~/code$ gcc -Wall -O -o simple simple.c
razvan@asgard:~/code$ gcc -Wall -O2 -o simple simple.c
F:\code>cl /Ox simple.c
```

Compilare (4)

- Se recomandă (insistent) folosirea opțiunii **-Wall**
- Opțiunea –Wall previne neajunsuri în sintaxa C
  - folosirea operatorului "=" în loc de "=="

```
#include <stdio.h>

int main()

{
int i = 0;

if (i = 0)
     printf("Best C program ever!");
else
     printf ("I have failed!!");
}
```

```
~$ gcc -o uso main.c

~$ ./uso

I have failed!!

~$ gcc -Wall main.c -o uso

main.c:7: warning: suggest parentheses
around assignment used as truth value
```

```
#include <stdio.h>

int main()

{
int i = 0;

if ( i == 0 )
     printf("Best C program ever!");
else
     printf ("I have failed!!");
}
```

```
~$ gcc -Wall -o uso main.c

~$ ./uso

Best C program ever!
```

Limbaj de asamblare

- Formă de reprezentare a instrucțiinilor procesorului
  - foloseşte **mnemonici** (`movl, cmpl, addl`)

```
movl    $0, %eax    ; stocheaza valoarea 0 in eax
```

- Extensia .asm (Windows) sau .s (Unix)
- O instrucțiune în limbaj de asamblare
  - operatori (add, mul, cmp, mov)
  - operanzi puşi la dispoziție de procesor (registre, locații de memorie)

- Sintaxe de limbaj de asamblare
  - sintaxa Intel

```
mov     eax, 0          ; stocheaza valoarea 0 in eax
cmp     [ebp-4], 10     ; compara valoarea 10 cu ceea ce se
                        ; gaseste la adresa ebp-4
```

  - sintaxa AT&T

```
movl    $0, %eax
cmpl    $10, -4(%ebp)
```

Folosire limbaj de asamblare

- Construire unor module “pure”
- Integrare în programe C – **inline assembly**

```
__asm__ ("movl %0,r9\n\tmovl %1,r10\n\tcall _foo"
      : /* no outputs */
      : "g" (from), "g" (to)
      : "r9", "r10");
```

- Limbaj depedent de arhitectură
  - greu extensibil
  - dificil de menținut

- Când folosim limbajul de asamblare?
  - limbajul de nivel înalt nu ne oferă un suport pentru anumite instrucțiuni ale procesorului
  - numite părți din cod trebuie optimizate de “mână”
    - profiling

Asamblare

```
razvan@asgard:~/code$ as -o simple.o simple.s
razvan@asgard:~/code$ objdump --disassemble simple.o
simple.o:     file format elf32-i386

Disassembly of section .text:
00000000 <main>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 18                sub    $0x18,%esp
   6:   83 e4 f0                and    $0xfffffff0,%esp
   9:   b8 00 00 00 00          mov    $0x0,%eax
   e:   29 c4                   sub    %eax,%esp
  10:   c7 45 f8 00 00 00 00    movl   $0x0,0xfffffff8(%ebp)
  17:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
  1e:   e8 fc ff ff ff          call   1f <main+0x1f>
  23:   c7 45 fc 01 00 00 00    movl   $0x1,0xfffffffc(%ebp)
  …
  4f:   e8 fc ff ff ff          call   50 <main+0x50>
  54:   b8 00 00 00 00          mov    $0x0,%eax
  59:   c9                      leave
  5a:   c3                      ret
```

Asamblare (2)

- Translatarea fişierului în limbaj de asamblare în fişier cod obiect
- Instrucțiunile în limbaj de asamblare se traduc una câte una în echivalentul lor binar (cod maşină)
- Asamblor
- Exemplu rulare as/msvc

```
razvan@asgard:~/code$ as -o simple.o simple.s
F:\code>cl /Fasimple.asm simple.c
```

Fişiere cod obiect

- Extensia .obj (Windows) sau .o (Unix)
- Conțin, codificate, variabilele (zona de date) şi instrucțiunile programului (zona de cod)
- **objdump** – investigarea unui modul obiect (**dezasamblarea**)

```
razvan@asgard:~/code$ objdump -s simple.o
simple.o:     file format elf32-i386
Contents of section .text:
 0000 5589e583 ec1883e4 f0b80000 000029c4  U.............).
 0010 c745f800 000000c7 04240000 0000e8fc  .E.......$......
 0020 ffffffc7 45fc0100 0000837d fc0a7e02  ....E......}..~.
[...]
Contents of section .rodata:
 0000 4e756d62 65722073 756d3a20 0025640a  Number sum: .%d.
 0010 00                                   .
 …
```

- **nm** – identificarea simbolurilor dintr-un modul obiect

```
razvan@asgard:~/code$ nm simple.o
00000000 T main
         U print
```

Biblioteci

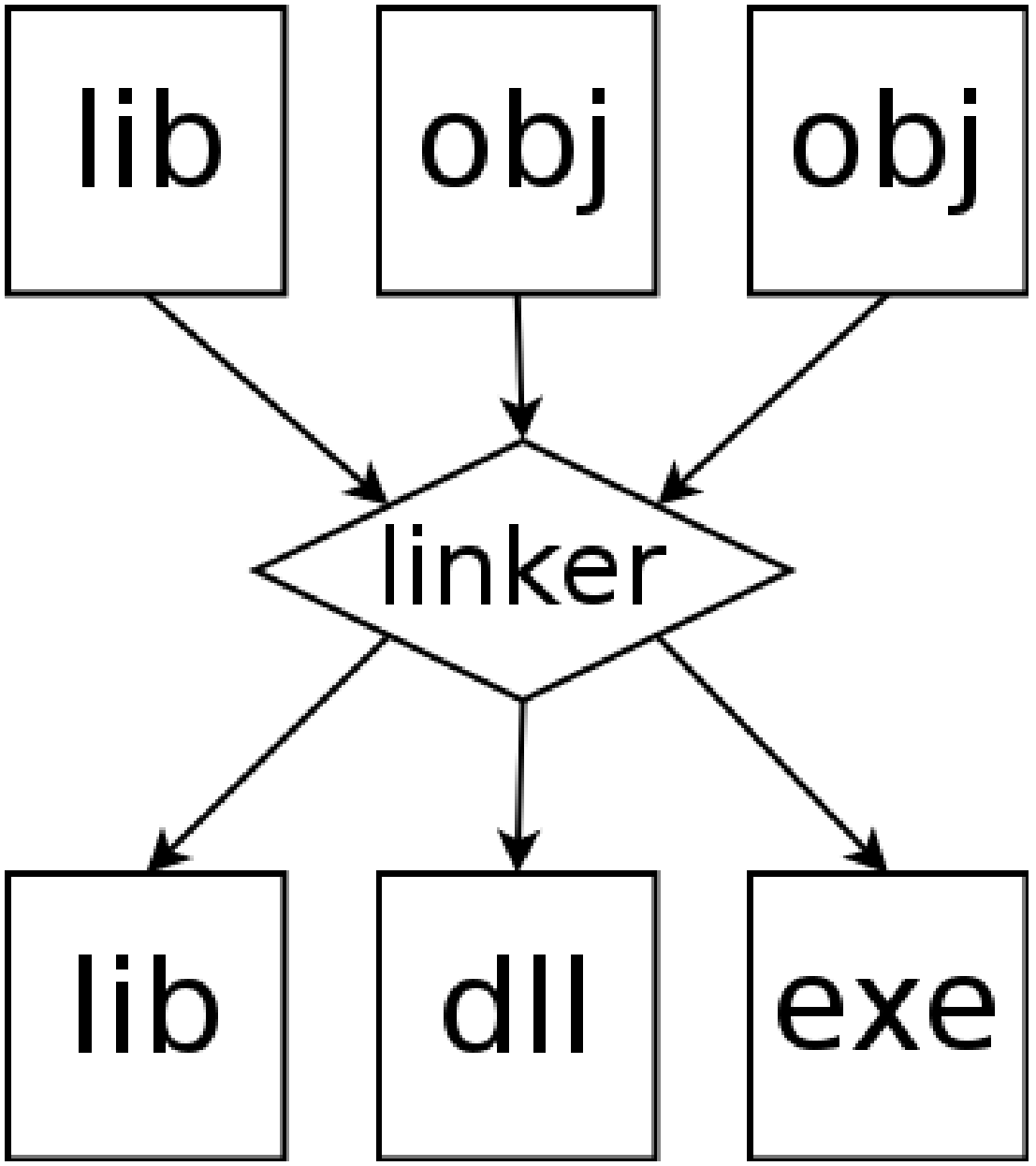
- Bibliotecă = library ≠ librărie
- Colecție de funcții des utilizate de către programe stocate într-un singur fişier (modul)
  - de fapt, o colecție de module obiect
- Nu sunt independente (precum executabilele)
  - module ce pot ajuta la crearea unui executabil
- În prezent, o parte importantă din codul programelor se regăseşte în biblioteci

Biblioteci (2)

- Biblioteci statice (static libraries)
  - funcțiile de bibliotecă apelate de un program sunt ataşate codului executabil la linking
  - codul executabil se măreşte
  - eventuale modificări ale bibliotecii din sistemul de operare nu afectează în nici un fel programul

- Biblioteci dinamice (dynamic libraries/shared object)
  - la linking se creează referințe către funcțiile apelate fără a fi incluse efectiv în executabil
  - biblioteca va fi încărcată în momentul execuției programului
  - cod executabil minim
  - este nevoie de recompilare dacă modificăm biblioteca
  - extensia .so în Linux şi .dll în Windows

Linking



Linking (2)

- Mai multe module obiect (inclusiv biblioteci) sunt grupate într-un modul de sine stătător (de obicei executabil)

- La legare/linking se rezolvă referințele
  1. Un modul apelează o funcție (sau o variabilă) dintr-un alt modul
  2. În urma compilării se creează o referință către acea funcție (fără a se şti în ce modul se găseşte)
  3. Linker-ul rezolvă aceste referințe, găsind funcțiile (variabilele) apelate din alte module
  4. Dacă se apelează o funcție dintr-un modul ce nu se regăseşte la linking, apare eroare la linking

Crearea de biblioteci

- Fişierul add.c implementează funcția add, iar sub.c implementează funcția sub
- Dorim o bibliotecă al cărei conținut să fie dat de cele două module
- Crearea şi legarea unei biblioteci statice

```
razvan@asgard:~/code$ gcc -Wall -c add.c
razvan@asgard:~/code$ gcc -Wall -c sub.c
razvan@asgard:~/code$ ar cr libsimple.a add.o sub.o
razvan@asgard:~/code$ ar t libsimple.a
add.o
sub.o
razvan@asgard:~/code$ gcc -Wall -L. -o lib_main lib_main.c -lsimple
```

- Crearea şi legarea unei biblioteci partajate

```
razvan@asgard:~/code$ gcc -Wall -fPIC -c add.c
razvan@asgard:~/code$ gcc -Wall -fPIC -c sub.c
razvan@asgard:~/code$ gcc -fPIC -shared -o libsimple.so add.o sub.c
razvan@asgard:~/code$ export LD_LIBRARY_PATH=.
razvan@asgard:~/code$ gcc -Wall -L. -o lib_main lib_main.c -lsimple
```

Executabil

- Tip special de fişier binar
  - poate fi încărcat de sistemul de operare într-un proces pentru a fi rulat
- Noțiunea de program se referă, în general, la un fişier executabil
- Formate specifice
  - sistemului de operare ştie cum să transpună informațiile într-un proces
  - **a.out** – formatul inițial pe Unix
  - **ELF** (Executable and Linking Format) – formatul implicit pe sistemele Unix moderne

```
razvan@asgard:~/code$ file libmain
libmain: ELF 32-bit LSB executable, Intel 80386, version 1
  (SYSV), for GNU/Linux 2.2.0, dynamically linked (uses shared
  libs), not stripped
```

  - **PE** (Portable Executable) – formatul implicit pe sistemele Windows

Execuția unui program - loading

- Execuția unui program
  - Windows:

```
F:\> hello.exe
Hello, World!
```

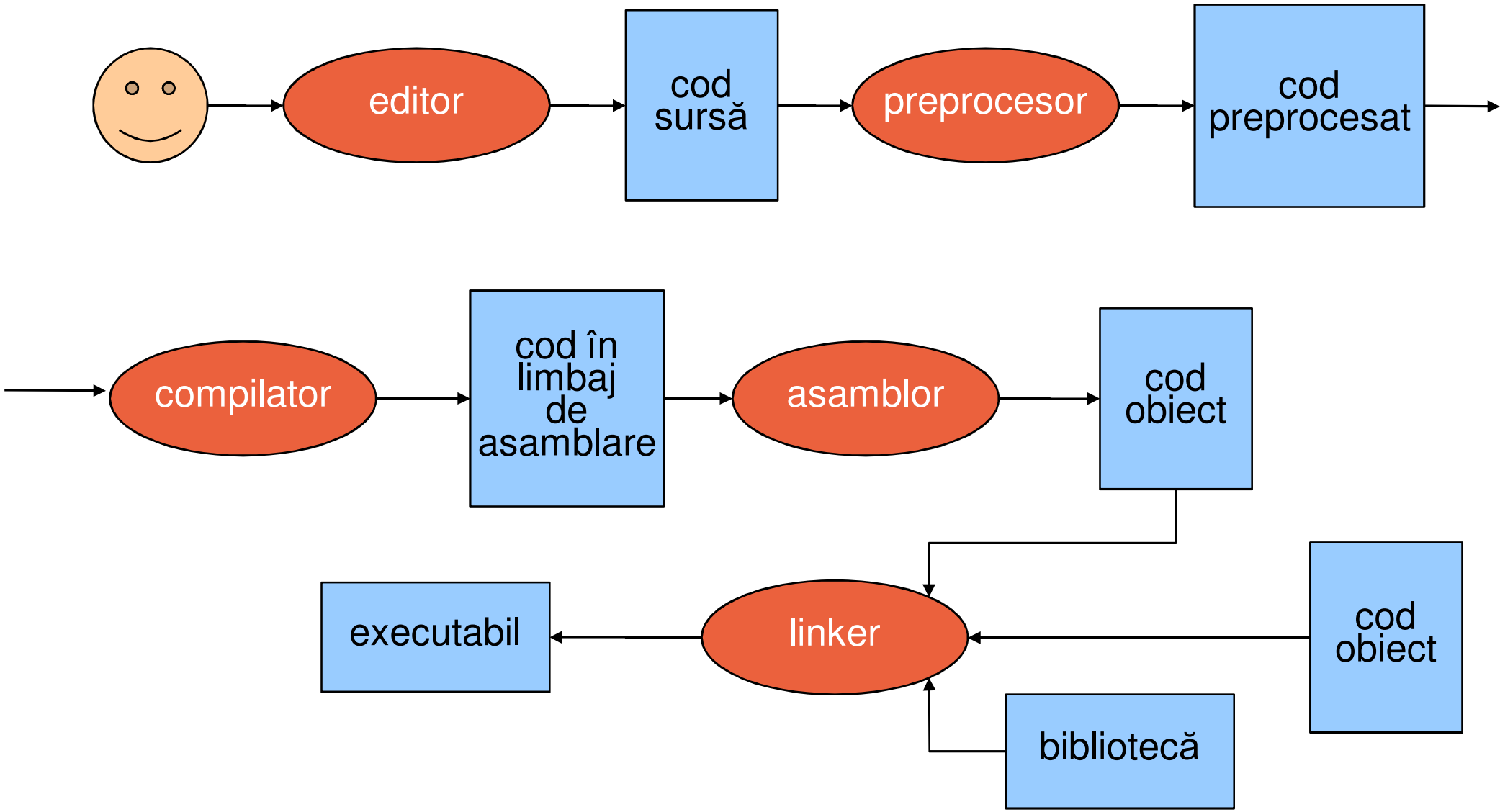
  - Unix: trebuie precizat explicit faptul că rulăm executabilul din directorul curent

```
razvan@asgard:~/code$ ./lib_main
add = 2, sub = 0
```

- Execuția şi folosirea bibliotecilor partajate sau dinamice au nevoie de suportul sistemului de operare

- Loader
  - componenta sistemului de operare care transpune informația dintr-un executabil într-un proces

Pus cap la cap



Automatizarea sarcinilor - Makefile

- Sarcinile de compilare/linking/etc. (şi nu numai) sunt de multe ori repetitive

- Folosirea de utilitare de automatizare
  - make, Apache Ant, Scons (dependințe)
  - shell scripting (fără dependințe)

- **make** foloseşte un fişier **Makefile**
- Structura tipică **Makefile**

```
# Comments use the hash symbol
target: dependencies
command 1
command 2
        .
        .
command n
```

Makefile simplu

```
$ cat Makefile
all: test

test: test.o
        gcc -o test test.o

test.o: test.c
        gcc -Wall -c -o test.o test.c

$ make
gcc -Wall -c -o test.o test.c
gcc -o test test.o

$ ./test
Hello, World!
```

Makefile simplu (2)

- La rularea **make** se caută prima regulă
  - se obişnuieşte să fie all

1. all depinde de test
2. test depinde de test.o
3. test.o depinde de test.c
   - test.c există
   - se rulează comanda de compilare
   - se obține test.o
4. test.o a fost obținut
   - se rulează comanda de linking
   - se obține test
5. test este obținut; all este obținut

Makefile upgrade

```
$ cat Makefile1
CC = gcc
CFLAGS = -Wall

all: test

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

test.o: test.c
        $(CC) $(CFLAGS) -c -o $@ $<

clean:
        -rm -f *~ *.o test

$ make -f Makefile1 clean
rm -f *~ *.o test

$ make -f Makefile1
gcc -Wall -c -o test.o test.c
gcc -o test test.o
```

Makefile upgrade (2)

- Se definesc variabile
  - CC, CFLAGS
  - se referă cu $(nume_variabila) -> $(CC)
- Variabile automate
  - $@ -> ținta (target-ul)
  - $^ -> toate dependințele
  - $< -> prima dependință
- Regulă de ştergere: clean
  - se şterg fişiere obiect, temporare şi executabilul
- Opțiunea -f specifică un fişier Makefile altul decât cel implicit (Makefile sau GNUMakefile)

Less is better

```
$ cat Makefile2
CFLAGS = -Wall

all: test

clean:
        -rm -f *~ *.o test

$ make -f Makefile2 clean
rm -f *~ *.o test

$ make -f Makefile2
cc -Wall    test.c   -o test

$ ./test
Hello, World!
```

Less is better

- Se folosesc reguli implicite

1. test nu depinde de alte target-uri, dar nu există
2. Se caută test.o sau test.c
3. Se găseşte test.c
4. Se rulează comanda implicită

```
$(CC) $(CFLAGS) test.c -o test
```

- CC este implic cc

```
razvan@anaconda:~$ ls -l /usr/bin/cc
lrwxrwxrwx 1 root root 20 Oct  6  2005 /usr/bin/cc -> /etc/alternatives/cc
razvan@anaconda:~$ ls -l /etc/alternatives/cc
lrwxrwxrwx 1 root root 12 Apr 22  2007 /etc/alternatives/cc -> /usr/bin/gcc
```

  - remember: probleme compilare module VMware pe Linux ☺

Surse multiple

- Un program este constituit, de obicei, din mai multe fisiere sursă (module)
- Fiecare modul implementează o componentă a aplicației

- Exemplu: aplicație chat
  - un modul pentru funcții utile -> util.c
  - un header pentru antetele funcțiilor utile -> util.h
  - un modul pentru interfața cu utilizatorul -> ui.c
  - un modul pentru antetele funcțiilor de interfațare -> ui.h
  - un modul care asigură comunicarea în rețea -> net.c
  - un header pentru antete -> net.h
  - un modul de jurnalizare -> log.c
  - un header petnru jurnalizare -> log.h
  - un modul care integrează celelalte module -> app.c

Cum se scrie un header

```
#include <stdlib.h>

#include "util.h"

int copy_to_list (char *msg, size_t
   msg_len, struct list *list)
{
   [...]
}

int find_in_list (int id, struct list
   *list)
{
   [...]
}
```

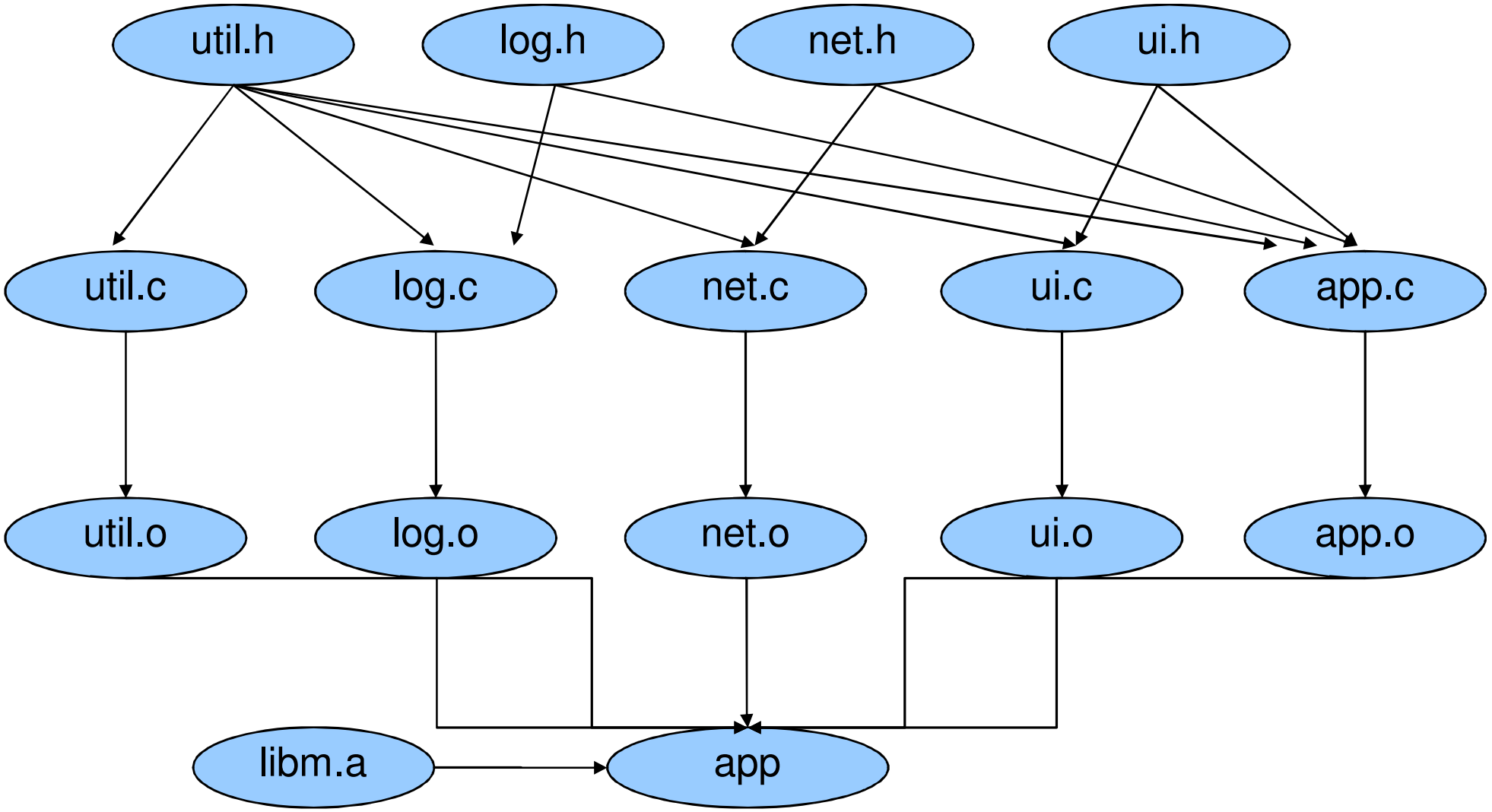
```
#ifndef _UTIL_H
#define _UTIL_H          1

struct list {
        int id;
        void *generic_data;
        struct list *next;
        struct list *prev;
};

int copy_to_list (char *msg, size_t
   msg_len, struct list *list);
int find_in_list (int id, struct list
   *list);

#endif
```

Dependințe


util.h
log.h
net.h
ui.h
util.c
log.c
net.c
ui.c
app.c
util.o
log.o
net.o
ui.o
app.o
libm.a
app

Teh Makefile

```
$ cat Makefile.prj
# -g -> compilare cu simboluri de debug
CFLAGS = -Wall -g
LDLIBS = -lm

all: app

app: app.o ui.o log.o util.o net.o

app.o: app.c util.h log.h ui.h net.h

ui.o: ui.c ui.h util.h

log.o: log.c log.h util.h

util.o: util.c util.h

net.o: net.c util.h

clean:
        -rm -f *~ *.o app
```

```
$ make -f Makefile.prj
cc -Wall -g        -c -o app.o app.c
cc -Wall -g        -c -o ui.o ui.c
cc -Wall -g        -c -o log.o log.c
cc -Wall -g        -c -o util.o util.c
cc -Wall -g        -c -o net.o net.c
cc   app.o ui.o log.o util.o net.o  -lm
   -o app
```

Cuvinte cheie

- program
- cod sursă
- editor
- IDE
- compiator
- interpretor
- coding style
- preprocesare
- compilare
- limbaj de asamblare
- asamblare
- cod obiect
- bibliotecă
- linker
- executabil
- cpp, gcc, gas, ld
- objdump, nm
- make, Makefile
- țintă, dependințe
- variabile automate
- header

Link-uri utile

- http://en.wikipedia.org/wiki/List_of_text_editors
- http://en.wikipedia.org/wiki/Comparison_of_integrated_development_environments
- http://www.microsoft.com/express/
- http://en.wikipedia.org/wiki/Source_code
- http://www.oualline.com/style/index.html
- http://www.gnu.org/software/make/manual/make.html

The End