Interacțiunea programelor cu Sistemul de Operare

Sisteme de Operare

Ciprian Oprișa și Adrian Coleșa

Universitatea Tehnică din Cluj-Napoca
Departamentul Calculatoare

Cursul 3

Cuprins

Cuprins

Rolul compilatorului

- Procesorul nu înțelege C, nici măcar Assembly
- Un program este o secvență de instrucțiuni în format binar
  - instrucțiunile binare se mapează 1 la 1 cu instrucțiuni Assembly

Rolul compilatorului

- Procesorul nu înțelege C, nici măcar Assembly
- Un program este o secvență de instrucțiuni în format binar
  - instrucțiunile binare se mapează 1 la 1 cu instrucțiuni Assembly

```
8b 10                   mov    edx,DWORD PTR [eax]
89 d0                   mov    eax,edx
c1 e0 02                shl    eax,0x2
01 d0                   add    eax,edx
c1 e0 02                shl    eax,0x2
01 c8                   add    eax,ecx
89 85 30 ff ff ff       mov    DWORD PTR [ebp-0xd0],eax
```

Rolul compilatorului

- Procesorul nu înțelege C, nici măcar Assembly
- Un program este o secvență de instrucțiuni în format binar
  - instrucțiunile binare se mapează 1 la 1 cu instrucțiuni Assembly

```
8b 10                   mov    edx,DWORD PTR [eax]
89 d0                   mov    eax,edx
c1 e0 02                shl    eax,0x2
01 d0                   add    eax,edx
c1 e0 02                shl    eax,0x2
01 c8                   add    eax,ecx
89 85 30 ff ff ff       mov    DWORD PTR [ebp-0xd0],eax
```

- compilatorul realizează "traducerea" din C în binar

Apelul compilatorului

- gcc (Linux, Windows, Mac)
  `gcc [opt] <source_name> -o <exec_name>`
- Visual C (Windows)
  `cl.exe [opt] <source_name> /link /OUT:<exec_name>`
- din IDE (click pe un buton $\rightarrow$ se apelează comenzi similare cu cele de mai sus, în mod transparent)

Apelul compilatorului

- gcc (Linux, Windows, Mac)
  `gcc [opt] <source_name> -o <exec_name>`
- Visual C (Windows)
  `cl.exe [opt] <source_name> /link /OUT:<exec_name>`
- din IDE (click pe un buton $\rightarrow$ se apelează comenzi similare cu cele de mai sus, în mod transparent)

Apelul compilatorului

- gcc (Linux, Windows, Mac)
  `gcc [opt] <source_name> -o <exec_name>`
- Visual C (Windows)
  `cl.exe [opt] <source_name> /link /OUT:<exec_name>`
- din IDE (click pe un buton → se apelează comenzi similare cu cele de mai sus, în mod transparent)

Exemplu: `gcc -Wall hellow.c -o hellow`
opțiunea `-Wall` înseamnă *warnings=all* (afișează toate avertismentele)

Apelul compilatorului

- gcc (Linux, Windows, Mac)
  `gcc [opt] <source_name> -o <exec_name>`
- Visual C (Windows)
  `cl.exe [opt] <source_name> /link /OUT:<exec_name>`
- din IDE (click pe un buton → se apelează comenzi similare cu cele de mai sus, în mod transparent)

Exemplu: `gcc -Wall hellow.c -o hellow`
opțiunea `-Wall` înseamnă *warnings=all* (afișează toate avertismentele)

Nu ignorați niciodată warning-urile compilatorului!

Programe cu mai multe fișiere sursă

Sursele unui program pot fi:

- fișiere `.c` - cod sursă, implementări
- fișiere `.h` - declarații de constante, tipuri de date, antete de funcții

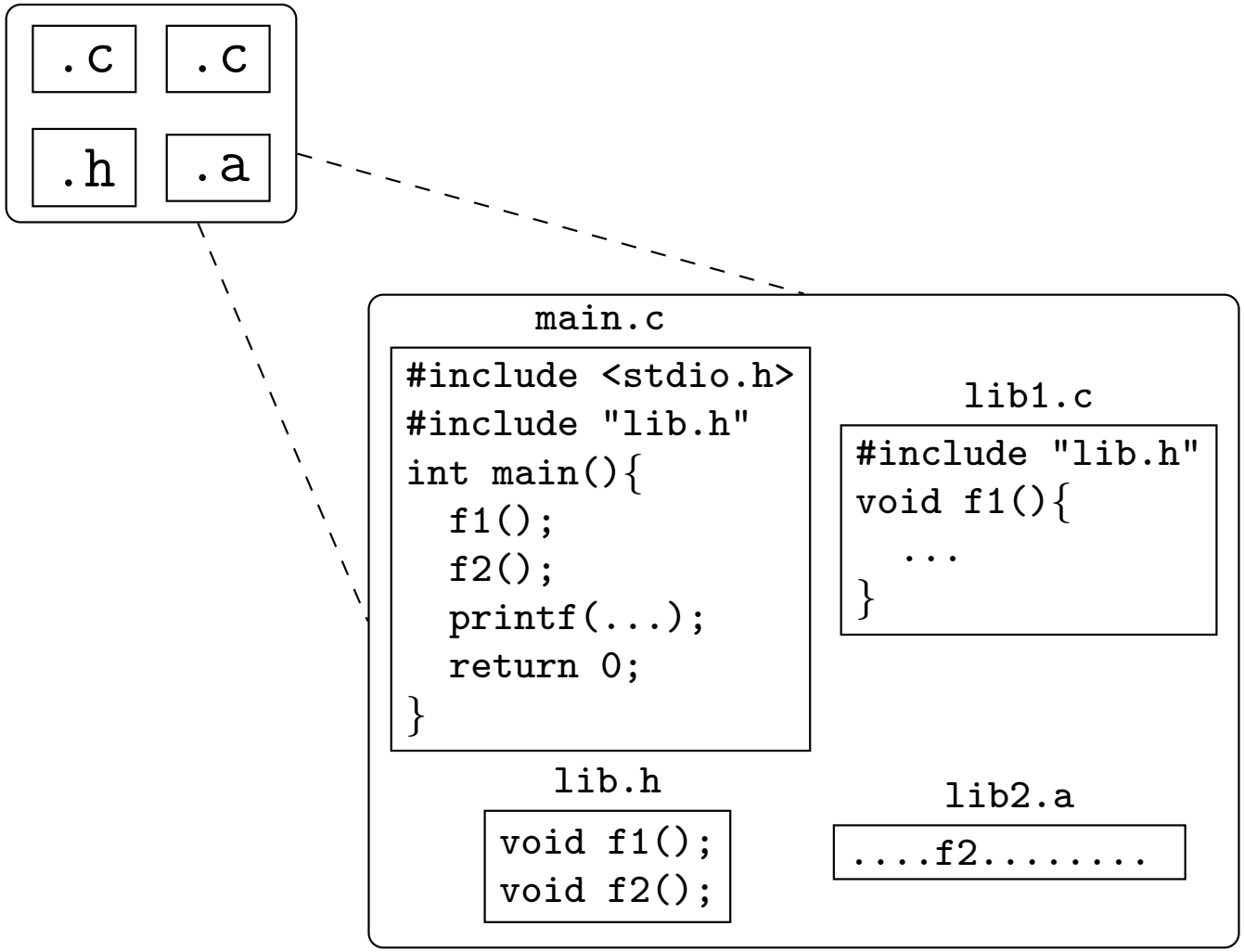
Programe cu mai multe fișiere sursă

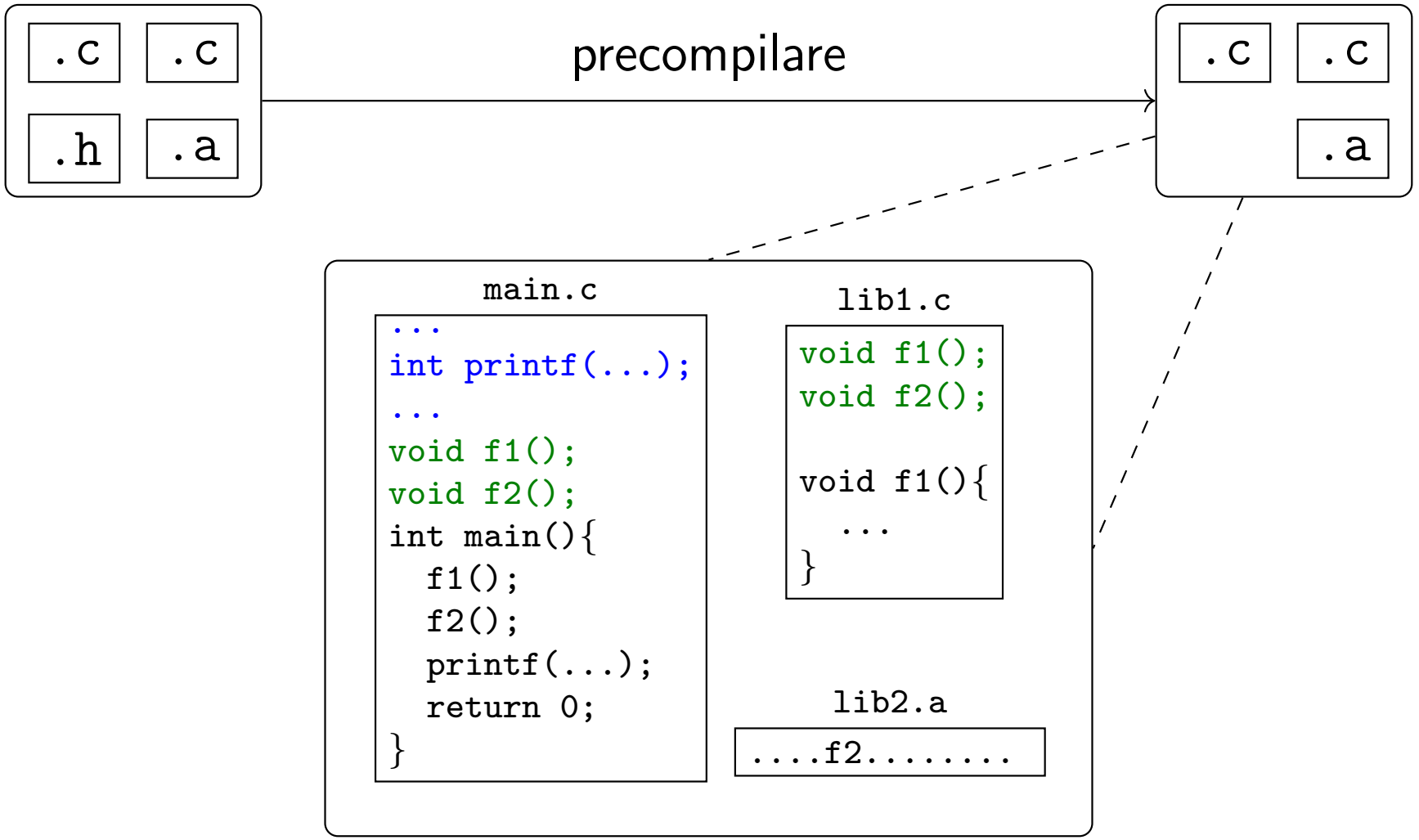
Sursele unui program pot fi:

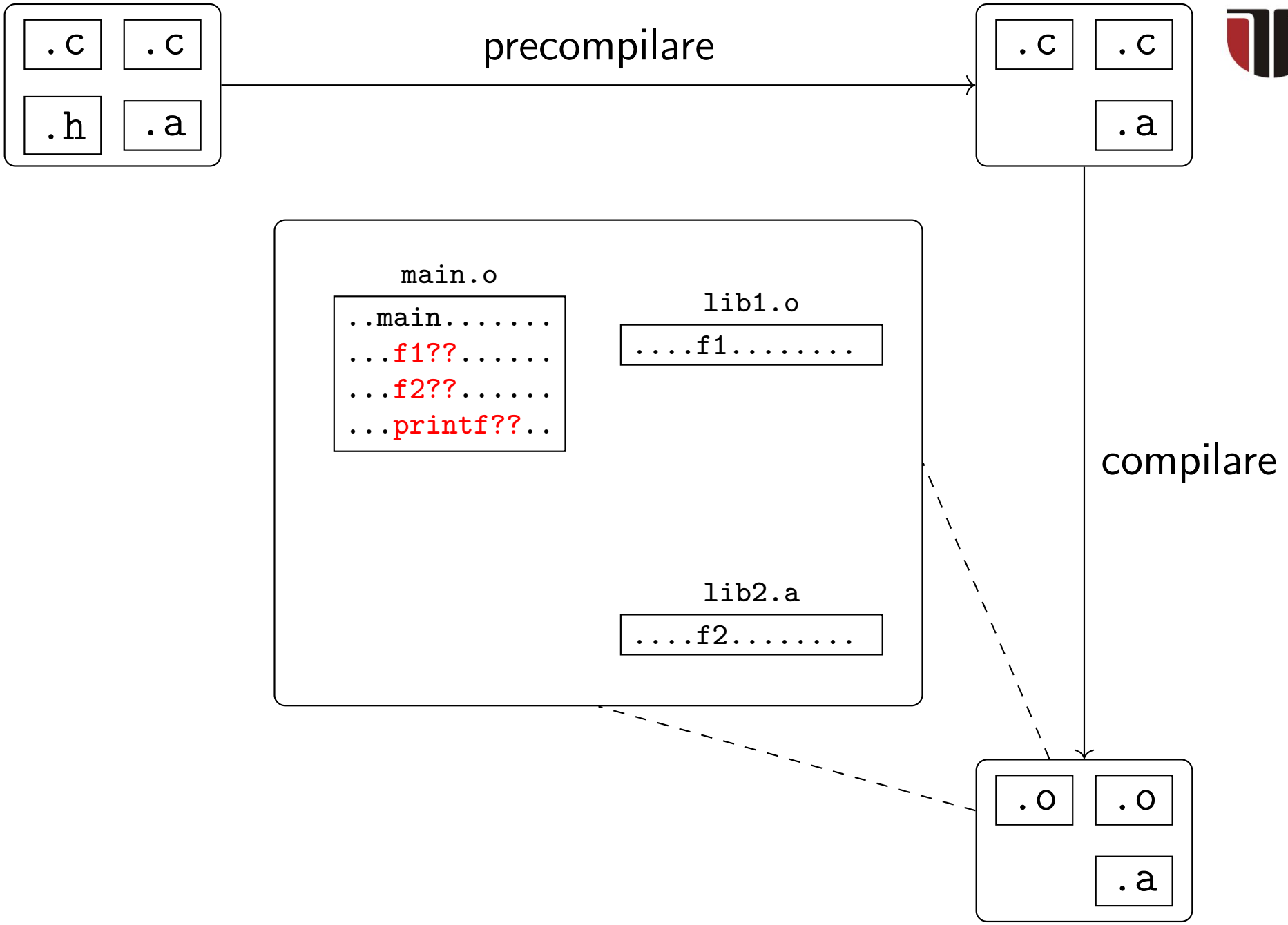
- fișiere `.c` - cod sursă, implementări
- fișiere `.h` - declarații de constante, tipuri de date, antete de funcții

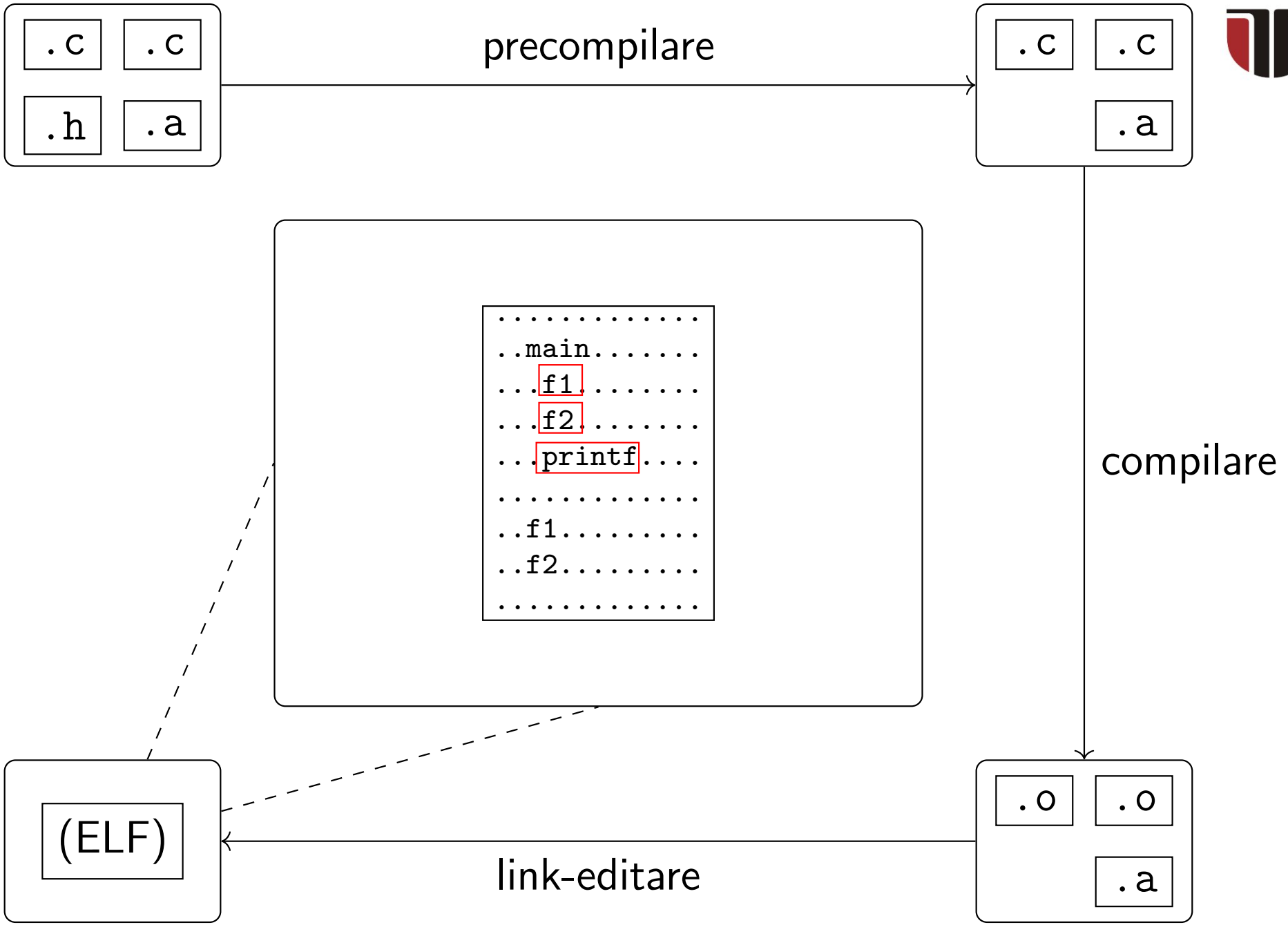
Procesul de compilare a unui program are mai multe etape:

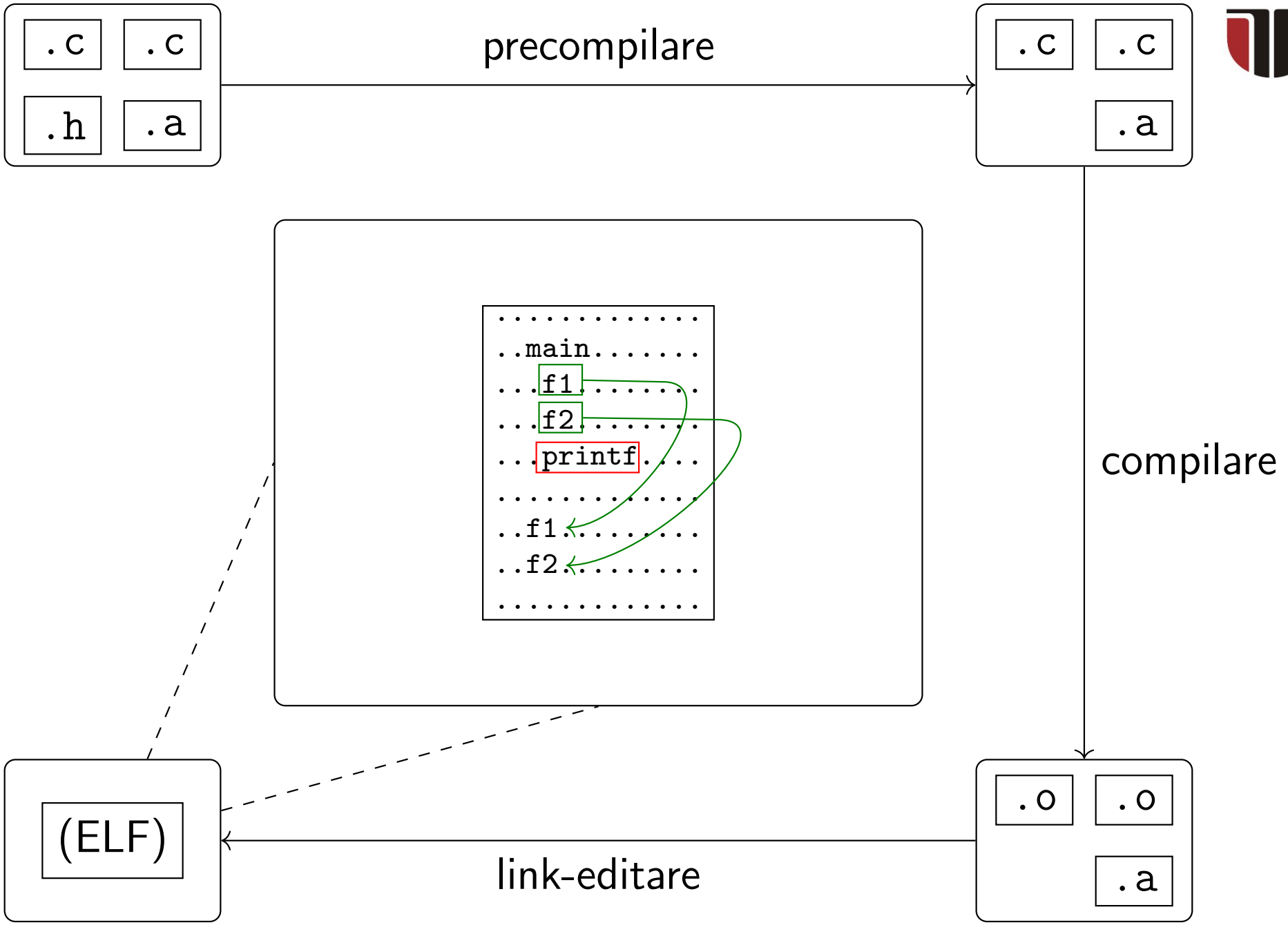
- **precompilare** - directivele de preprocesor (`#include`, `#define`, ...) se expandează.
- **compilare** - fiecare sursă `.c` se compilează ⇒ fișiere obiect `.o`.
- **link-editare** - fișierele obiect se unesc într-un singur executabil. Linker-ul face legăturile către funcții din fișiere obiect diferite sau din biblioteci externe (`.so`). Tot aici se poate adăuga cod compilat în biblioteci statice (`.a`).

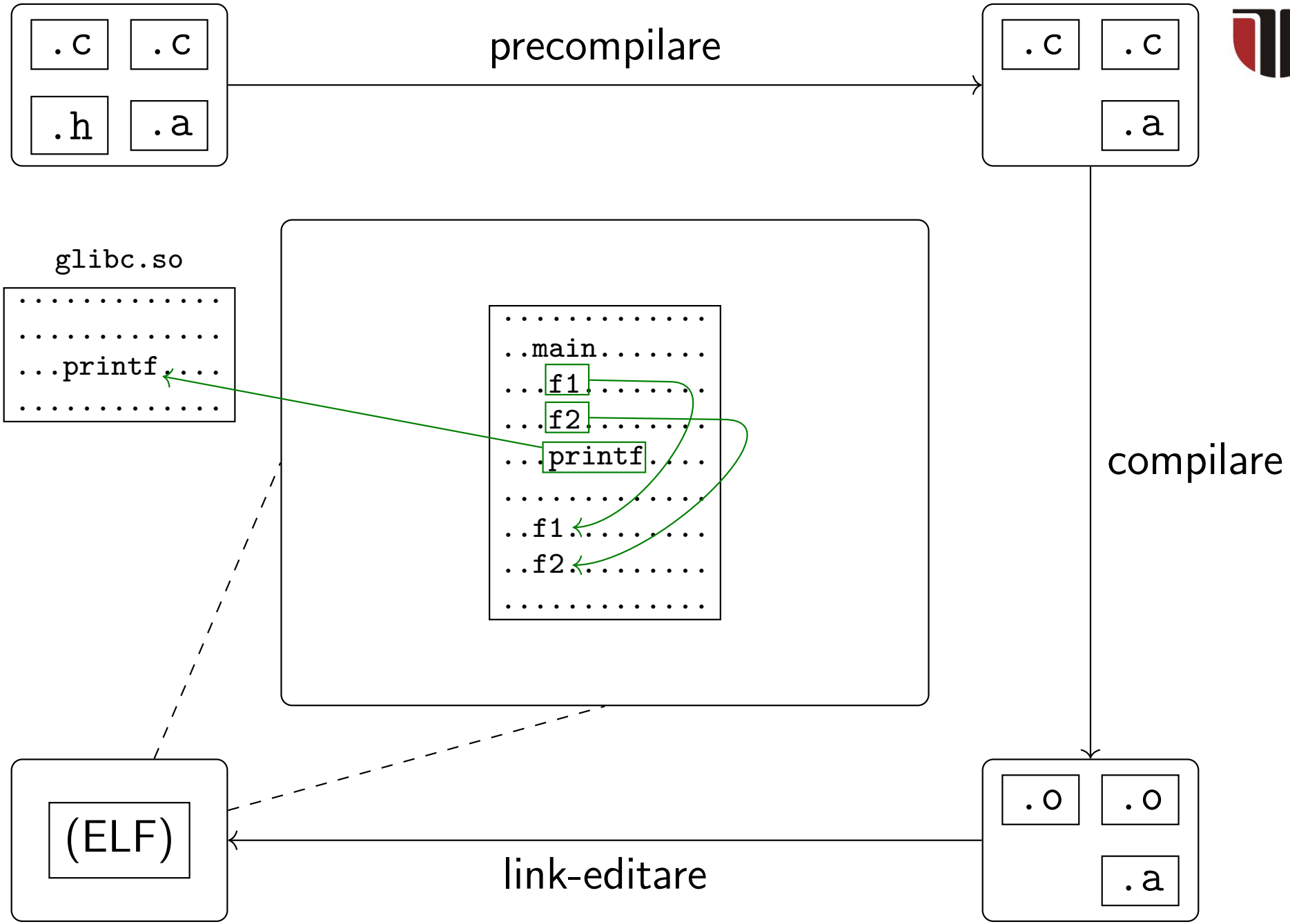

.c
.c
.h
.a
main.c
#include <stdio.h>
#include "lib.h"
int main(){
f1();
f2();
printf(...);
return 0;
}
lib1.c
#include "lib.h"
void f1(){
...
}
lib.h
void f1();
void f2();
lib2.a
....f2.....


.c
.c
.h
.a
precompilare
.c
.c
.a
main.c
...
int printf(...);
...
void f1();
void f2();
int main(){
f1();
f2();
printf(...);
return 0;
}
lib1.c
void f1();
void f2();
void f1(){
...
}
lib2.a
....f2.....


.c
.c
.h
.a
precompilare
.c
.c
.a
compilare
main.o
..main.....
...f1??.....
...f2??.....
...printf??..
lib1.o
....f1.....
lib2.a
....f2.....
.o
.o
.a


.c
.c
.h
.a
precompilare
.c
.c
.a
compilare
.....
..main.....
...f1.....
...f2.....
...printf....
.....
..f1.....
..f2.....
.....
(ELF)
.o
.o
.a
link-editare


.c
.c
.h
.a
precompilare
.c
.c
.a
compilare
main
f1
f2
printf
f1
f2
(ELF)
link-editare
.o
.o
.a


.c
.c
.h
.a
precompilare
.c
.c
.a
glibc.so
...printf...
..main.....
...f1.....
...f2.....
...printf...
..f1.....
..f2.....
compilare
(ELF)
link-editare
.o
.o
.a

Cuprins

Variabilele locale și stiva (1)

test1.c:

```
#include <stdio.h>

int main(void){
    int x = 7;
    printf("x=%d\n", x);
    return 0;
}
```

Variabilele locale și stiva (1)

test1.c:

```
#include <stdio.h>

int main(void){
    int x = 7;
    printf("x=%d\n", x);
    return 0;
}
```

Compilare:

```
gcc -Wall test1.c -o test1
```

Variabilele locale și stiva (1)

test1.c:

```
#include <stdio.h>

int main(void){
    int x = 7;
    printf("x=%d\n", x);
    return 0;
}
```

Compilare:

```
gcc -Wall test1.c -o test1
```

Rulare:

```
./test1
x=7
```

Variabilele locale și stiva (1)

test1.c:

```
#include <stdio.h>

int main(void){
    int x = 7;
    printf("x=%d\n", x);
    return 0;
}
```

Compilare:

```
gcc -Wall test1.c -o test1
```

Rulare:

```
./test1
x=7
```

Compilare cu mai multe opțiuni:

```
gcc -g -m32 -Wall test1.c -o test1
```

- `-g`: adaugă în executabil informații de debug
- `-m32`: generează cod pe 32 bit

Variabilele locale și stiva (1)

test1.c:

```
#include <stdio.h>

int main(void){
    int x = 7;
    printf("x=%d\n", x);
    return 0;
}
```

Compilare:

```
gcc -Wall test1.c -o test1
```

Rulare:

```
./test1
x=7
```

Compilare cu mai multe opțiuni:

```
gcc -g -m32 -Wall test1.c -o test1
```

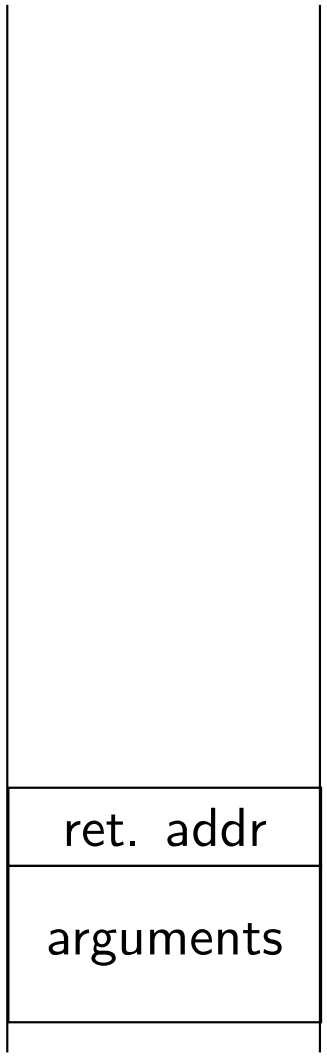
- `-g`: adaugă în executabil informații de debug
- `-m32`: generează cod pe 32 bit

Vizualizare cod Assembly:

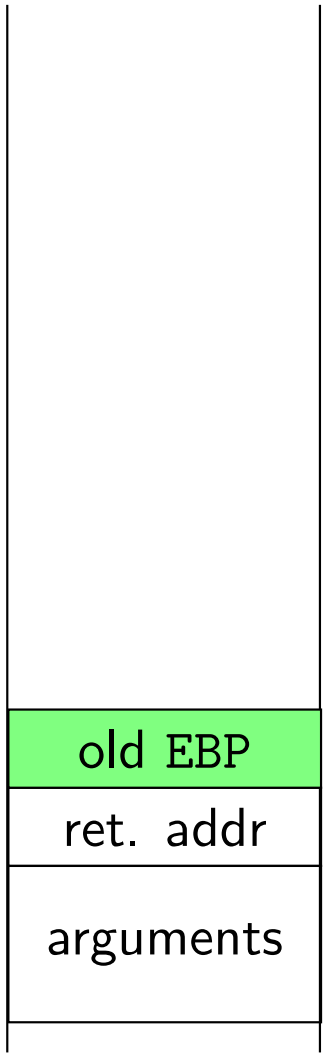
```
objdump -D test1 -M intel -S
```

- `-D`: dezasamblare
- `-M intel`: sintaxă Intel
- `-S`: afișează și linii de cod C

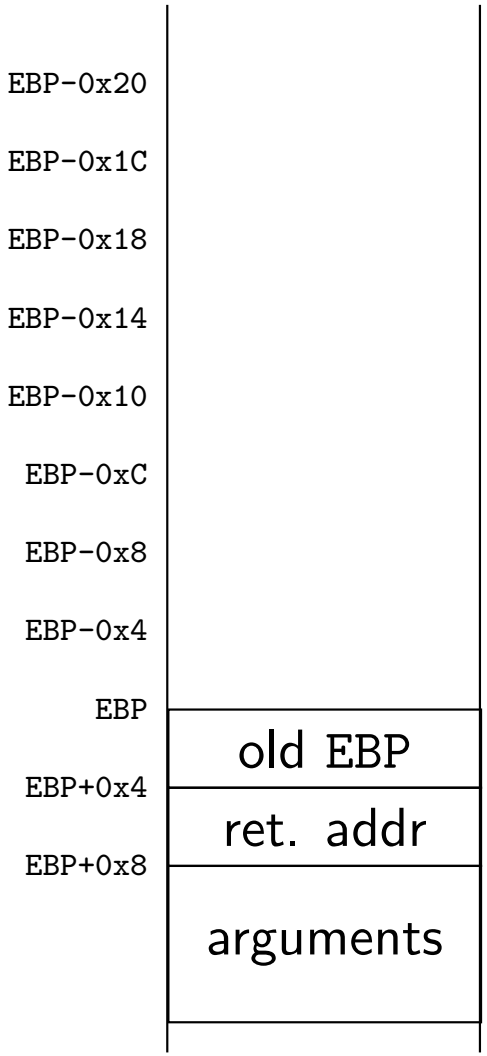
```
;int main(void){
...
push ebp
mov ebp,esp
...
push ebx
push ecx
sub esp,0x10
...
;int x = 7;
mov DWORD PTR [ebp-0xc],0x7
;printf("x=%d\n", x);
...
push DWORD PTR [ebp-0xc]
lea  edx,[eax-0x19e8]
push edx ;"x=%d\n"
mov  ebx,eax
call 3b0 <printf@plt>
...
;return 0;
mov    eax,0x0
```



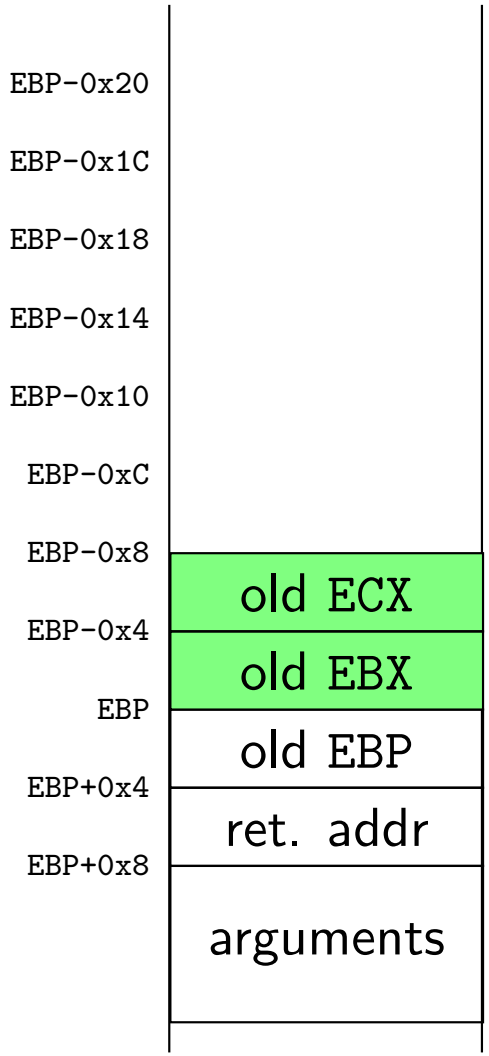
```
;int main(void){
...
push ebp
mov ebp,esp
...
push ebx
push ecx
sub esp,0x10
...
;int x = 7;
mov DWORD PTR [ebp-0xc],0x7
;printf("x=%d\n", x);
...
push DWORD PTR [ebp-0xc]
lea  edx,[eax-0x19e8]
push edx ;"x=%d\n"
mov  ebx,eax
call 3b0 <printf@plt>
...
;return 0;
mov    eax,0x0
```



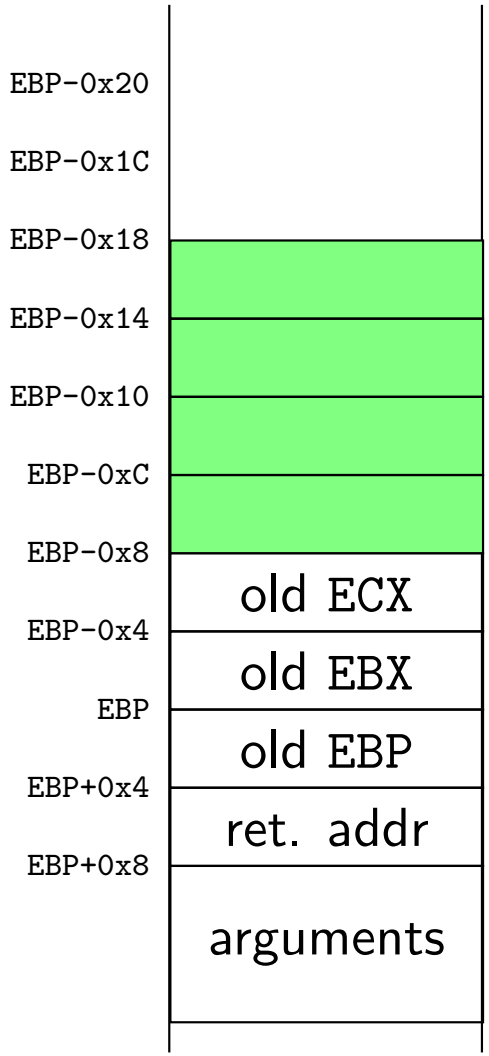
```
;int main(void){
...
push ebp
mov ebp,esp
...
push ebx
push ecx
sub esp,0x10
...
;int x = 7;
mov DWORD PTR [ebp-0xc],0x7
;printf("x=%d\n", x);
...
push DWORD PTR [ebp-0xc]
lea  edx,[eax-0x19e8]
push edx ;"x=%d\n"
mov  ebx,eax
call 3b0 <printf@plt>
...
;return 0;
mov    eax,0x0
```



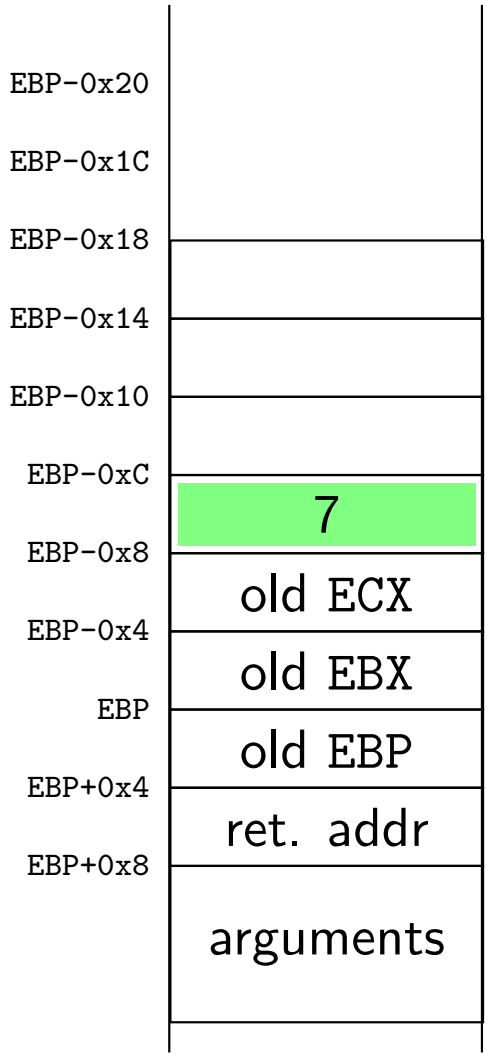
```
;int main(void){
...
push ebp
mov ebp,esp
...
push ebx
push ecx
sub esp,0x10
...
;int x = 7;
mov DWORD PTR [ebp-0xc],0x7
;printf("x=%d\n", x);
...
push DWORD PTR [ebp-0xc]
lea  edx,[eax-0x19e8]
push edx ;"x=%d\n"
mov  ebx,eax
call 3b0 <printf@plt>
...
;return 0;
mov     eax,0x0
```



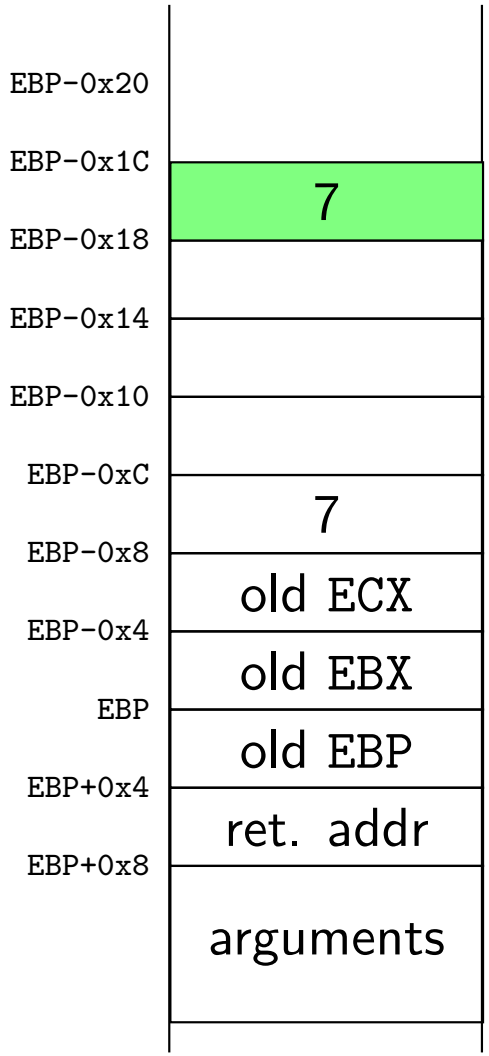
```
;int main(void){
...
push ebp
mov ebp,esp
...
push ebx
push ecx
sub esp,0x10
...
;int x = 7;
mov DWORD PTR [ebp-0xc],0x7
;printf("x=%d\n", x);
...
push DWORD PTR [ebp-0xc]
lea  edx,[eax-0x19e8]
push edx ;"x=%d\n"
mov  ebx,eax
call 3b0 <printf@plt>
...
;return 0;
mov     eax,0x0
```



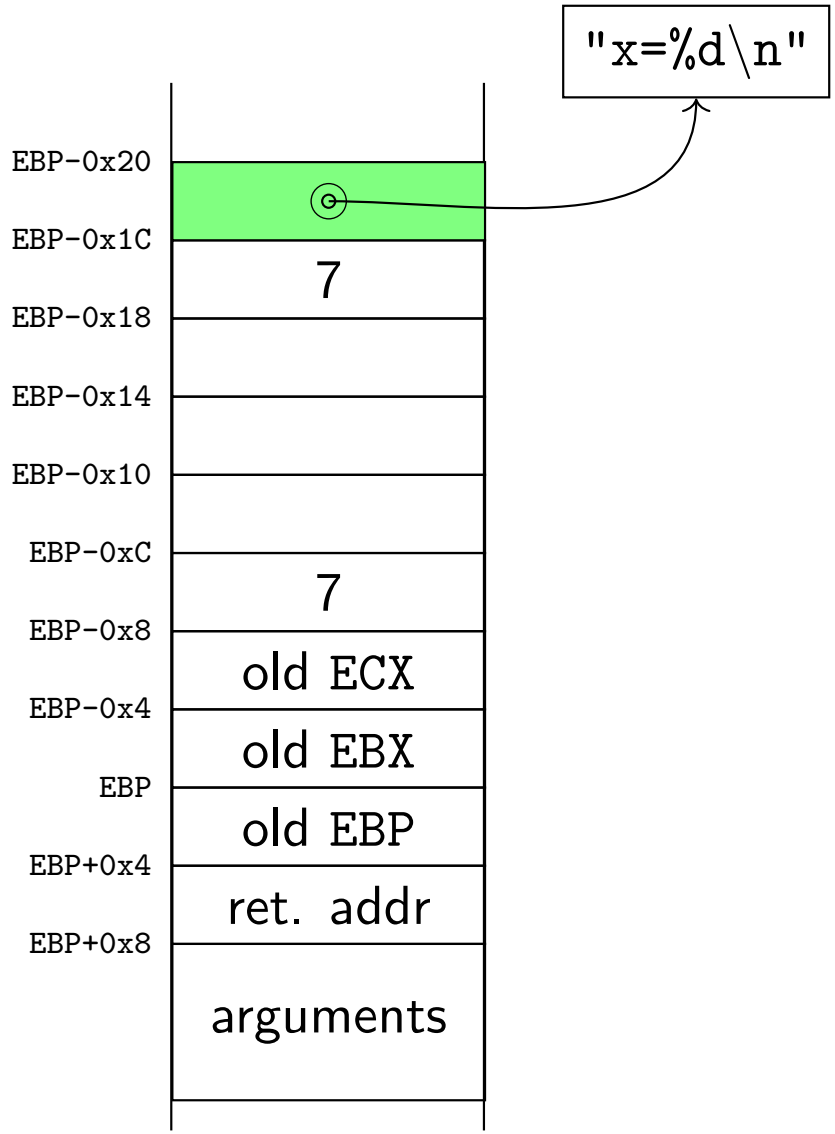
```
;int main(void){
...
push ebp
mov ebp,esp
...
push ebx
push ecx
sub esp,0x10
...
;int x = 7;
mov DWORD PTR [ebp-0xc],0x7
;printf("x=%d\n", x);
...
push DWORD PTR [ebp-0xc]
lea  edx,[eax-0x19e8]
push edx ;"x=%d\n"
mov  ebx,eax
call 3b0 <printf@plt>
...
;return 0;
mov     eax,0x0
```



```
;int main(void){
...
push ebp
mov ebp,esp
...
push ebx
push ecx
sub esp,0x10
...
;int x = 7;
mov DWORD PTR [ebp-0xc],0x7
;printf("x=%d\n", x);
...
push DWORD PTR [ebp-0xc]
lea  edx,[eax-0x19e8]
push edx ;"x=%d\n"
mov  ebx,eax
call 3b0 <printf@plt>
...
;return 0;
mov    eax,0x0
```



```
;int main(void){
...
push ebp
mov ebp,esp
...
push ebx
push ecx
sub esp,0x10
...
;int x = 7;
mov DWORD PTR [ebp-0xc],0x7
;printf("x=%d\n", x);
...
push DWORD PTR [ebp-0xc]
lea  edx,[eax-0x19e8]
push edx ;"x=%d\n"
mov  ebx,eax
call 3b0 <printf@plt>
...
;return 0;
mov     eax,0x0
```



Manipularea vectorilor (1)

test2.c:

```
int main(void){
    int a[2];
    int b[] = {7, 8};
    a[0] = 3;
    a[1] = 4;
    printf("%d %d\n", a[0], a[1]);
    printf("%d %d\n", b[0], b[1]);
    return 0;
}
```

- se pot declara inițializați sau nu
- adresarea se face începând de la 0
- conținutul lor se află pe stivă (dacă sunt declarați ca variabile locale)

Manipularea vectorilor (1)

test2.c:

```
int main(void){
    int a[2];
    int b[] = {7, 8};
    a[0] = 3;
    a[1] = 4;
    printf("%d %d\n", a[0], a[1]);
    printf("%d %d\n", b[0], b[1]);
    return 0;
}
```

```
$ gcc -Wall test2.c -o test2
$ ./test2
```

- se pot declara inițializați sau nu
- adresarea se face începând de la 0
- conținutul lor se află pe stivă (dacă sunt declarați ca variabile locale)

Manipularea vectorilor (1)

test2.c:

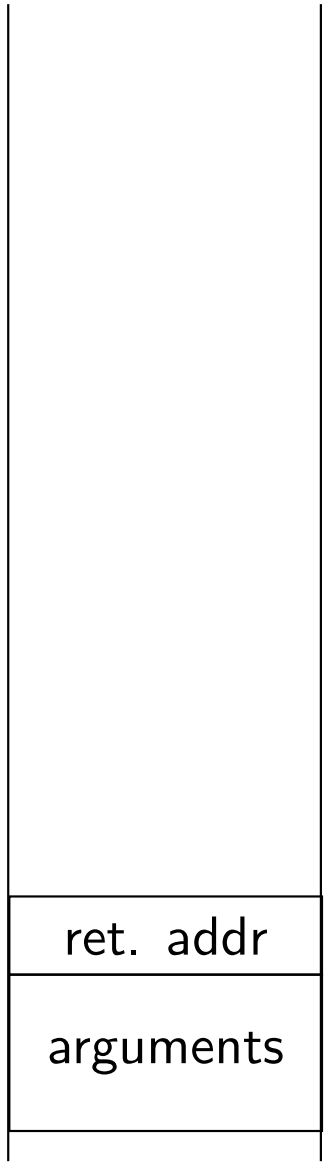
```
int main(void){
    int a[2];
    int b[] = {7, 8};
    a[0] = 3;
    a[1] = 4;
    printf("%d %d\n", a[0], a[1]);
    printf("%d %d\n", b[0], b[1]);
    return 0;
}
```

```
$ gcc -Wall test2.c -o test2
$ ./test2
3 4
7 8
```

- se pot declara inițializați sau nu
- adresarea se face începând de la 0
- conținutul lor se află pe stivă (dacă sunt declarați ca variabile locale)

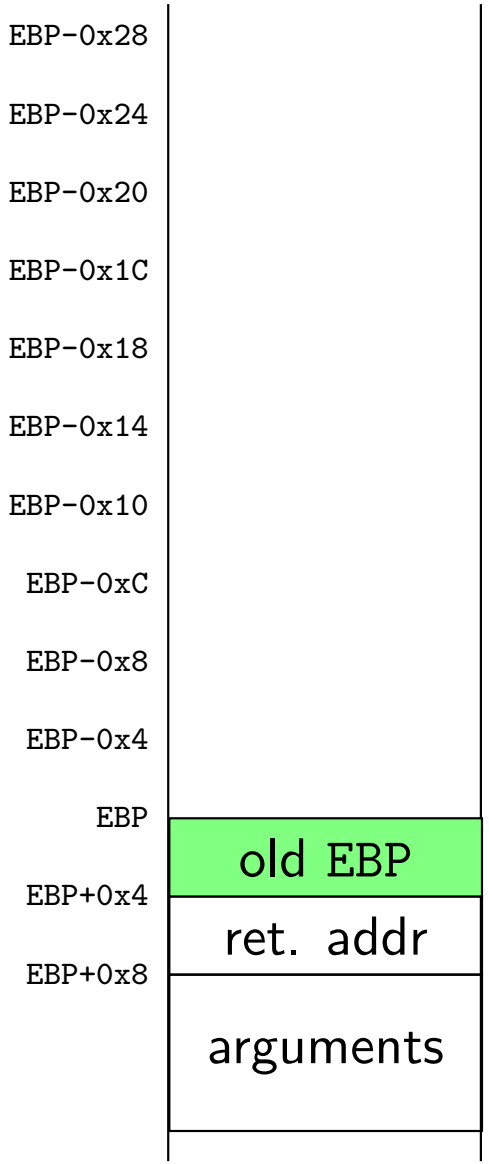
Manipularea vectorilor (2)

```
...
push ebp
mov ebp,esp
push ebx
push ecx
sub esp,0x20
...
;int a[2];
;int b[2] = {7, 8};
mov DWORD PTR [ebp-0x14],0x7
mov DWORD PTR [ebp-0x10],0x8
;a[0] = 3;
mov DWORD PTR [ebp-0x1c],0x3
;a[1] = 4;
mov DWORD PTR [ebp-0x18],0x4
...
```



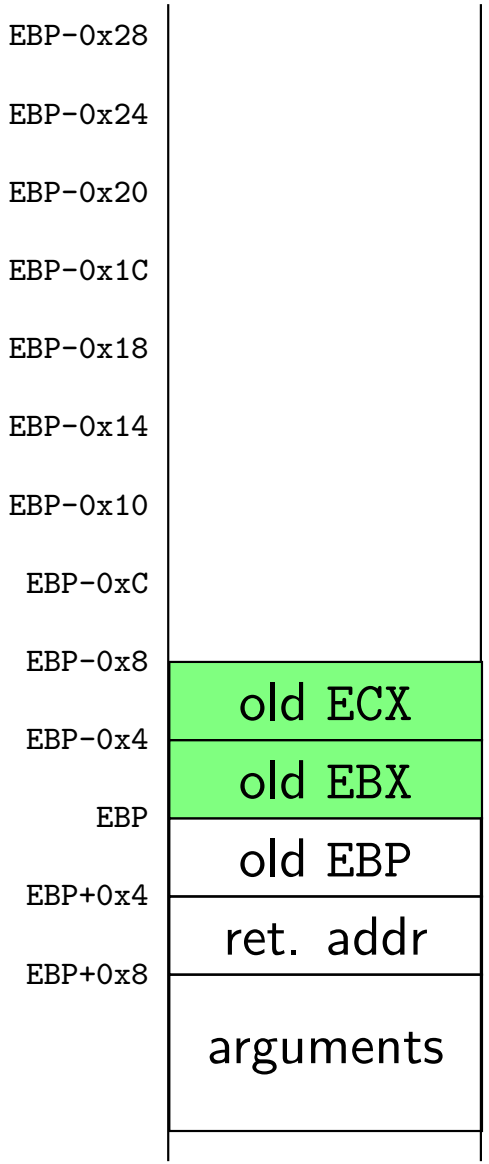
Manipularea vectorilor (2)

```
...
push ebp
mov ebp,esp
push ebx
push ecx
sub esp,0x20
...
;int a[2];
;int b[2] = {7, 8};
mov DWORD PTR [ebp-0x14],0x7
mov DWORD PTR [ebp-0x10],0x8
;a[0] = 3;
mov DWORD PTR [ebp-0x1c],0x3
;a[1] = 4;
mov DWORD PTR [ebp-0x18],0x4
...
```



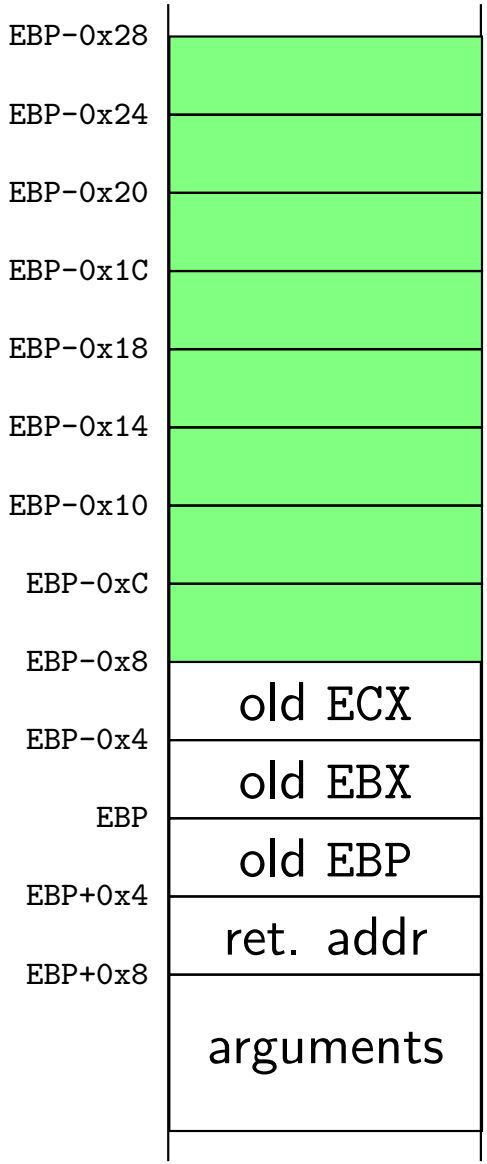
Manipularea vectorilor (2)

```
...
push ebp
mov ebp,esp
push ebx
push ecx
sub esp,0x20
...
;int a[2];
;int b[2] = {7, 8};
mov DWORD PTR [ebp-0x14],0x7
mov DWORD PTR [ebp-0x10],0x8
;a[0] = 3;
mov DWORD PTR [ebp-0x1c],0x3
;a[1] = 4;
mov DWORD PTR [ebp-0x18],0x4
...
```



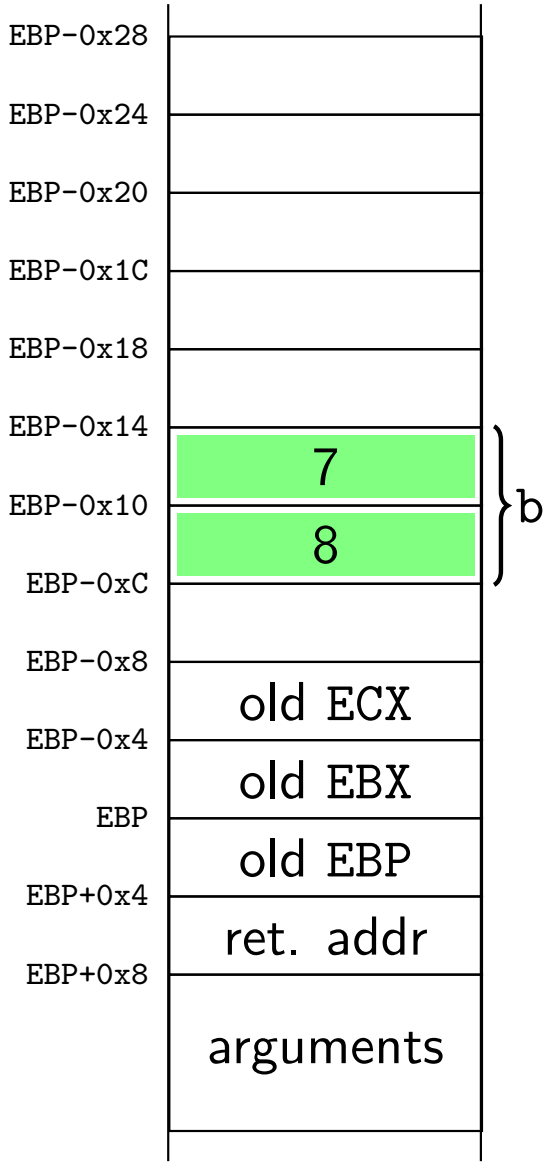
Manipularea vectorilor (2)

```
...
push ebp
mov ebp,esp
push ebx
push ecx
sub esp,0x20
...
;int a[2];
;int b[2] = {7, 8};
mov DWORD PTR [ebp-0x14],0x7
mov DWORD PTR [ebp-0x10],0x8
;a[0] = 3;
mov DWORD PTR [ebp-0x1c],0x3
;a[1] = 4;
mov DWORD PTR [ebp-0x18],0x4
...
```



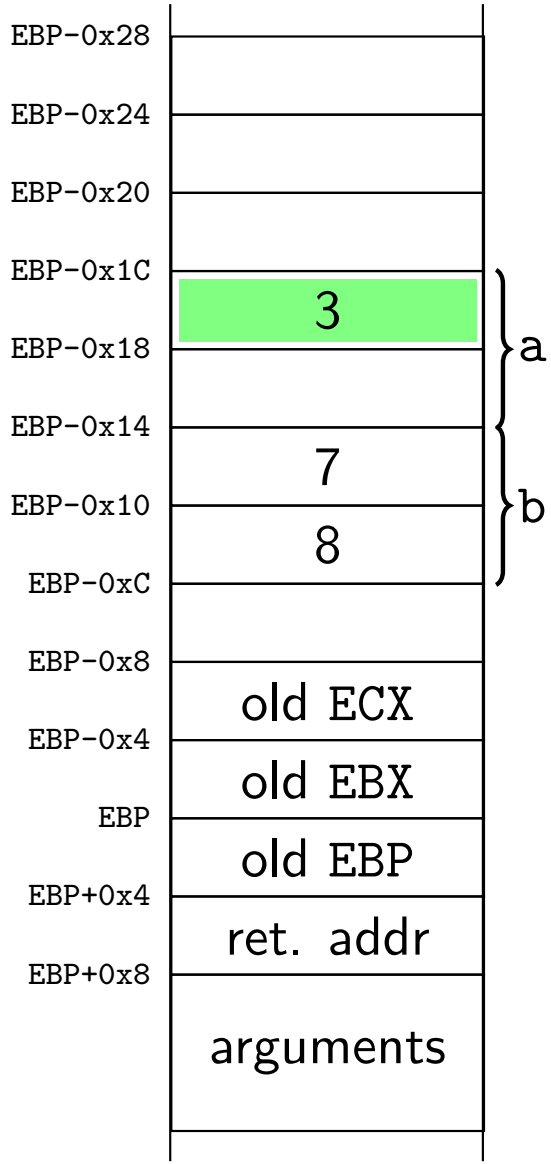
Manipularea vectorilor (2)

```
...
push ebp
mov ebp,esp
push ebx
push ecx
sub esp,0x20
...
;int a[2];
;int b[2] = {7, 8};
mov DWORD PTR [ebp-0x14],0x7
mov DWORD PTR [ebp-0x10],0x8
;a[0] = 3;
mov DWORD PTR [ebp-0x1c],0x3
;a[1] = 4;
mov DWORD PTR [ebp-0x18],0x4
...
```



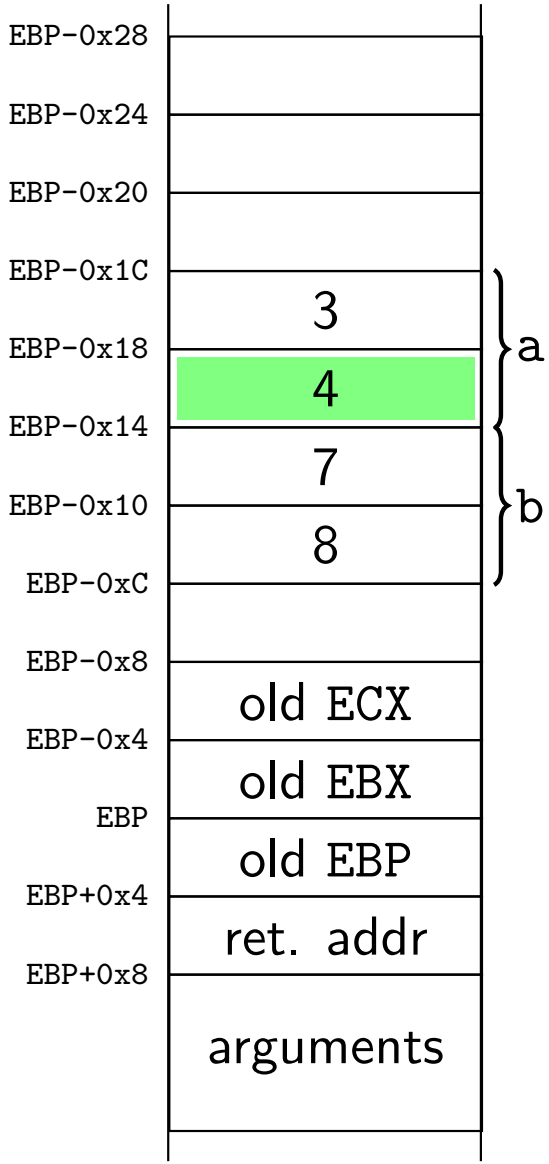
Manipularea vectorilor (2)

```
...
push ebp
mov ebp,esp
push ebx
push ecx
sub esp,0x20
...
;int a[2];
;int b[2] = {7, 8};
mov DWORD PTR [ebp-0x14],0x7
mov DWORD PTR [ebp-0x10],0x8
;a[0] = 3;
mov DWORD PTR [ebp-0x1c],0x3
;a[1] = 4;
mov DWORD PTR [ebp-0x18],0x4
...
```



Manipularea vectorilor (2)

```
...
push ebp
mov ebp,esp
push ebx
push ecx
sub esp,0x20
...
;int a[2];
;int b[2] = {7, 8};
mov DWORD PTR [ebp-0x14],0x7
mov DWORD PTR [ebp-0x10],0x8
;a[0] = 3;
mov DWORD PTR [ebp-0x1c],0x3
;a[1] = 4;
mov DWORD PTR [ebp-0x18],0x4
...
```



Coruperea memoriei (1)

test3.c:

```
int main(void){
    int a[2];
    int b[] = {7, 8};
    a[0] = 3;
    a[1] = 4;
    a[2] = 5;
    printf("%d %d\n", a[0], a[1]);
    printf("%d %d\n", b[0], b[1]);
    return 0;
}
```

Ce se întâmplă când accesăm un vector în afara limitelor?

- în Java, vectorul este un obiect care își "cunoaște" capetele $\Rightarrow$ excepție
- în C, vectorul e doar o adresă de început
- standardul C nu definește comportamentul în astfel de situații
  - se poate întâmpla orice

Coruperea memoriei (1)

test3.c:

```
int main(void){
    int a[2];
    int b[] = {7, 8};
    a[0] = 3;
    a[1] = 4;
    a[2] = 5;
    printf("%d %d\n", a[0], a[1]);
    printf("%d %d\n", b[0], b[1]);
    return 0;
}
```

```
$ gcc -Wall test3.c -o test3
$ ./test3
```

Ce se întâmplă când accesăm un vector în afara limitelor?

- în Java, vectorul este un obiect care își "cunoaște" capetele $\Rightarrow$ excepție
- în C, vectorul e doar o adresă de început
- standardul C nu definește comportamentul în astfel de situații
  - se poate întâmpla orice

Coruperea memoriei (1)

test3.c:

```
int main(void){
    int a[2];
    int b[] = {7, 8};
    a[0] = 3;
    a[1] = 4;
    a[2] = 5;
    printf("%d %d\n", a[0], a[1]);
    printf("%d %d\n", b[0], b[1]);
    return 0;
}
```

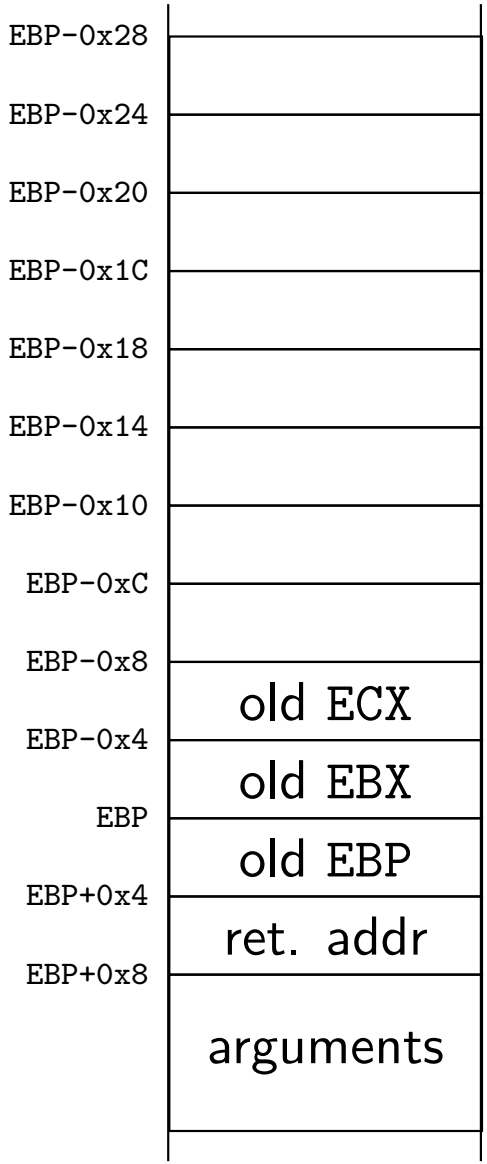
```
$ gcc -Wall test3.c -o test3
$ ./test3
3 4
5 8
```

Ce se întâmplă când accesăm un vector în afara limitelor?

- în Java, vectorul este un obiect care își "cunoaște" capetele ⇒ excepție
- în C, vectorul e doar o adresă de început
- standardul C nu definește comportamentul în astfel de situații
  - se poate întâmpla orice

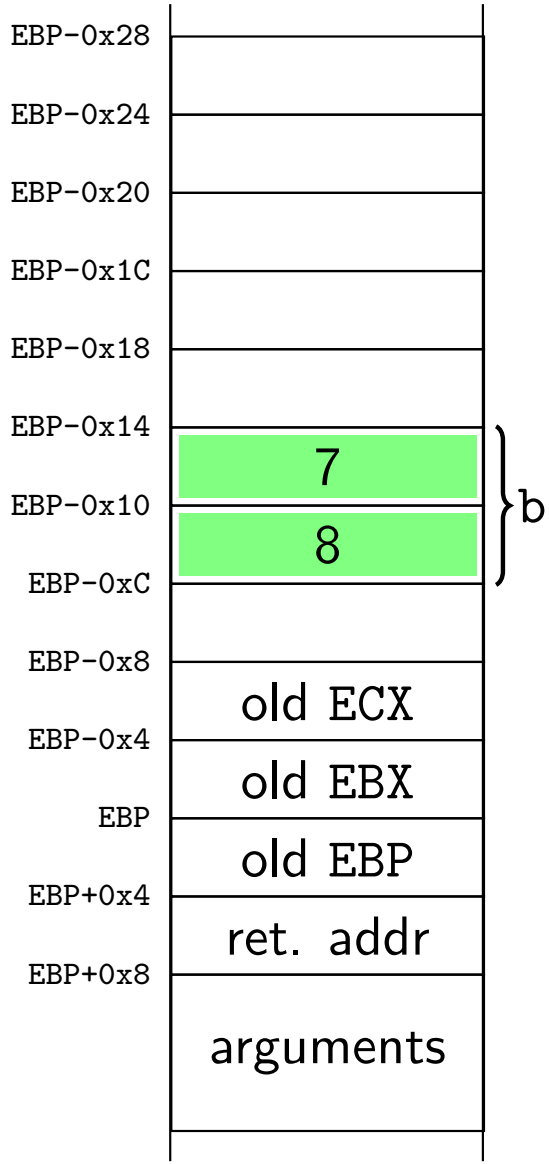
Coruperea memoriei (2)

```
...
;int a[2];
;int b[2] = {7, 8};
mov DWORD PTR [ebp-0x14],0x7
mov DWORD PTR [ebp-0x10],0x8
;a[0] = 3;
mov DWORD PTR [ebp-0x1c],0x3
;a[1] = 4;
mov DWORD PTR [ebp-0x18],0x4
;a[2] = 5;
mov DWORD PTR [ebp-0x14],0x5
...
```



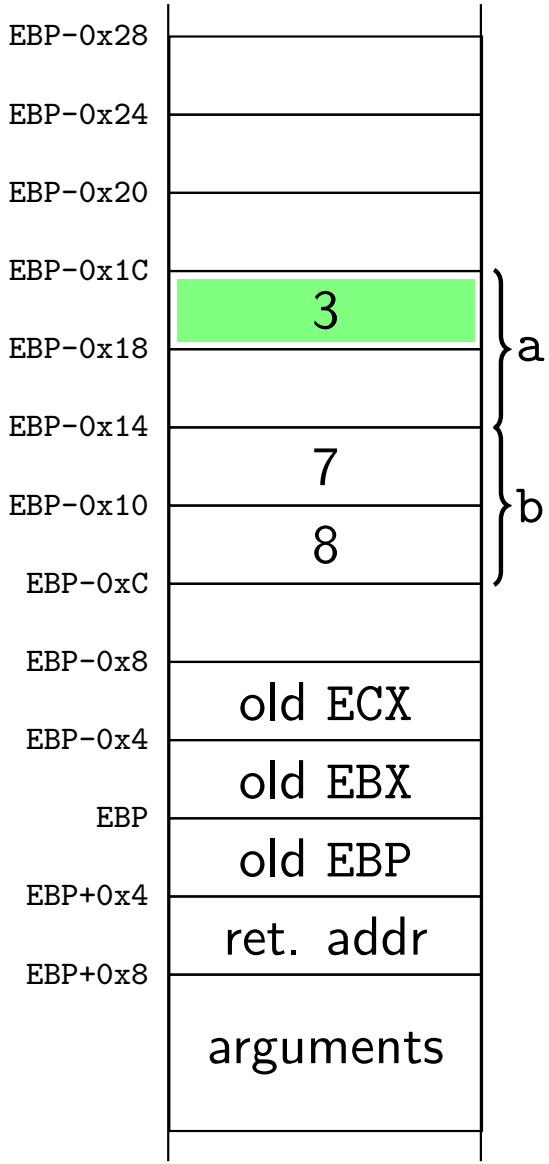
Coruperea memoriei (2)

```
...
;int a[2];
;int b[2] = {7, 8};
mov DWORD PTR [ebp-0x14],0x7
mov DWORD PTR [ebp-0x10],0x8
;a[0] = 3;
mov DWORD PTR [ebp-0x1c],0x3
;a[1] = 4;
mov DWORD PTR [ebp-0x18],0x4
;a[2] = 5;
mov DWORD PTR [ebp-0x14],0x5
...
```



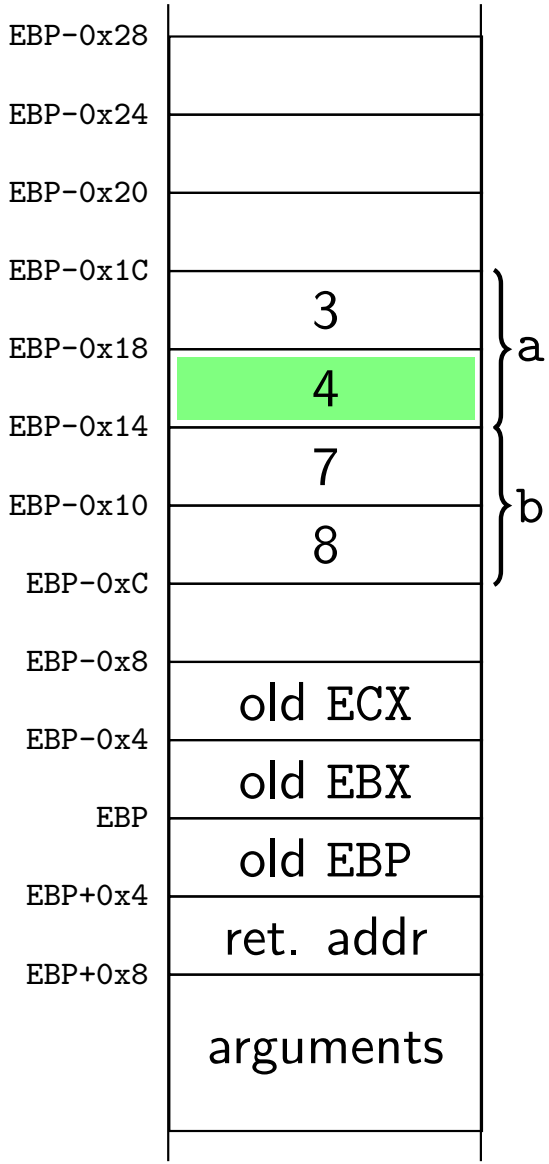
Coruperea memoriei (2)

```
...
;int a[2];
;int b[2] = {7, 8};
mov DWORD PTR [ebp-0x14],0x7
mov DWORD PTR [ebp-0x10],0x8
;a[0] = 3;
mov DWORD PTR [ebp-0x1c],0x3
;a[1] = 4;
mov DWORD PTR [ebp-0x18],0x4
;a[2] = 5;
mov DWORD PTR [ebp-0x14],0x5
...
```



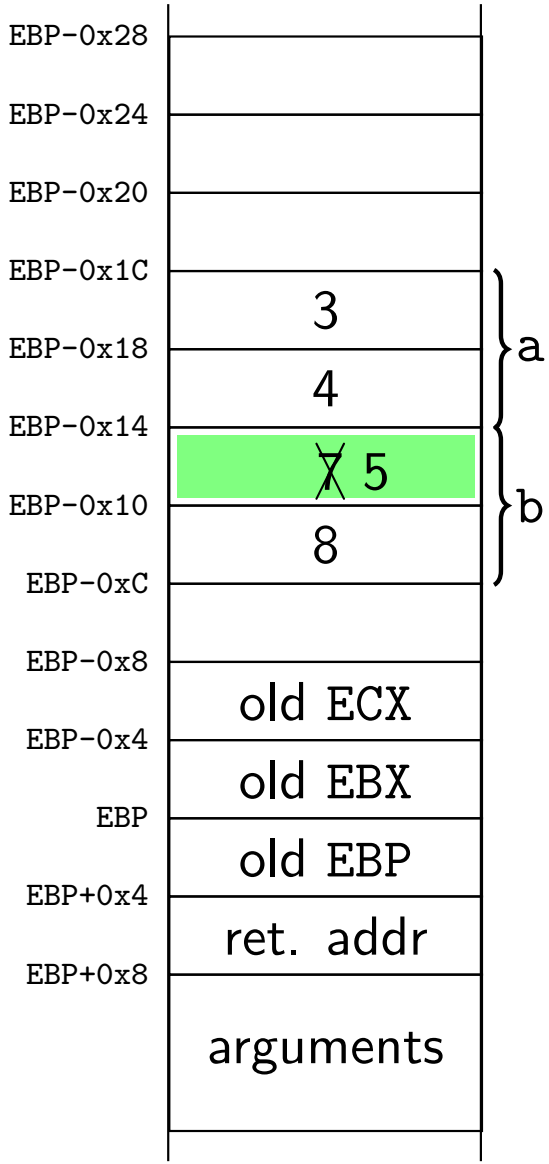
Coruperea memoriei (2)

```
...
;int a[2];
;int b[2] = {7, 8};
mov DWORD PTR [ebp-0x14],0x7
mov DWORD PTR [ebp-0x10],0x8
;a[0] = 3;
mov DWORD PTR [ebp-0x1c],0x3
;a[1] = 4;
mov DWORD PTR [ebp-0x18],0x4
;a[2] = 5;
mov DWORD PTR [ebp-0x14],0x5
...
```



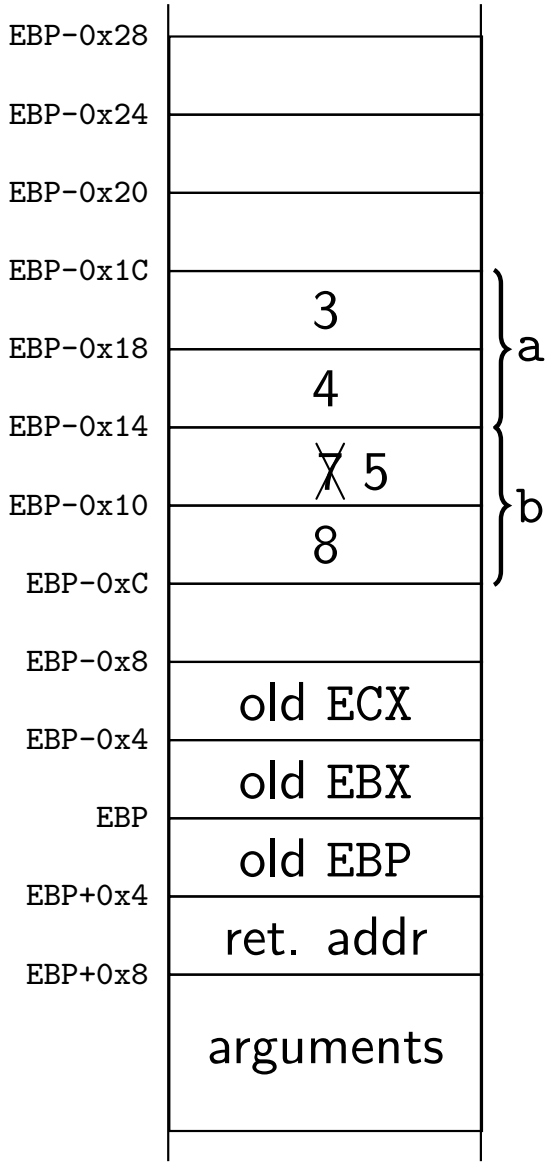
Coruperea memoriei (2)

```
...
;int a[2];
;int b[2] = {7, 8};
mov DWORD PTR [ebp-0x14],0x7
mov DWORD PTR [ebp-0x10],0x8
;a[0] = 3;
mov DWORD PTR [ebp-0x1c],0x3
;a[1] = 4;
mov DWORD PTR [ebp-0x18],0x4
;a[2] = 5;
mov DWORD PTR [ebp-0x14],0x5
...
```



Coruperea memoriei (2)

```
...
;int a[2];
;int b[2] = {7, 8};
mov DWORD PTR [ebp-0x14],0x7
mov DWORD PTR [ebp-0x10],0x8
;a[0] = 3;
mov DWORD PTR [ebp-0x1c],0x3
;a[1] = 4;
mov DWORD PTR [ebp-0x18],0x4
;a[2] = 5;
mov DWORD PTR [ebp-0x14],0x5
...
```



Bonus

Ce se întâmplă dacă modificăm a[8]?

Pointeri (1)

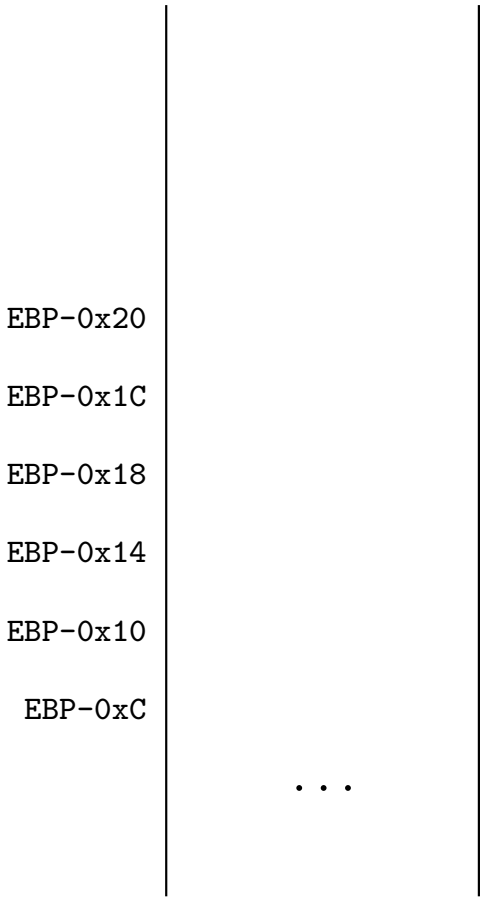
test_ptr.c:

```
void f1(int x){
    x = x * 2;
}
void f2(int *x){
    *x = *x * 2;
}

int main(void){
    int x = 3;
    f1(x);
    printf("%d\n", x);
    f2(&x);
    printf("%d\n", x);
    return 0;
}
```

- un pointer poate reține adresa unei alte variabile
- utili atunci când vrem să transmitem o referință (nu vrem să copiem datele)
- adresele în memorie sunt și ele numere (pe 32 sau 64 bit)
- & - referențiere (extrage adresa unei variabile)
- * - dereferențiere (extrage variabila dintr-o adresă)

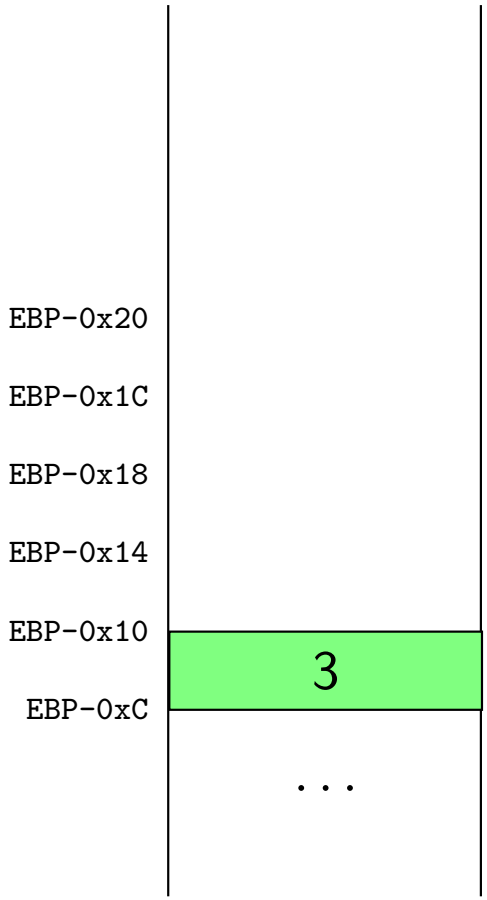
Pointeri (2)

```
;int x = 3;
mov DWORD PTR [ebp-0x10],0x3
;f1(x);
mov eax,DWORD PTR [ebp-0x10]
push eax
call 57d <f1>;copie
add esp,0x4
;printf("%d\n", x);
...
;f2(&x);
sub esp,0xc
lea eax,[ebp-0x10]
push eax
call 590 <f2>;referinta
add esp,0x10
;printf("%d\n", x);
...
```



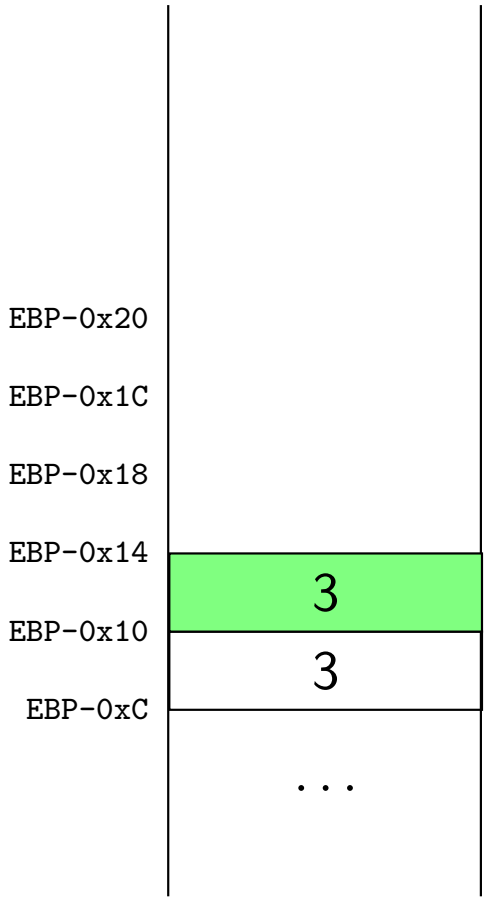
Pointeri (2)

```
;int x = 3;
mov DWORD PTR [ebp-0x10],0x3
;f1(x);
mov eax,DWORD PTR [ebp-0x10]
push eax
call 57d <f1>;copie
add esp,0x4
;printf("%d\n", x);
...
;f2(&x);
sub esp,0xc
lea eax,[ebp-0x10]
push eax
call 590 <f2>;referinta
add esp,0x10
;printf("%d\n", x);
...
```



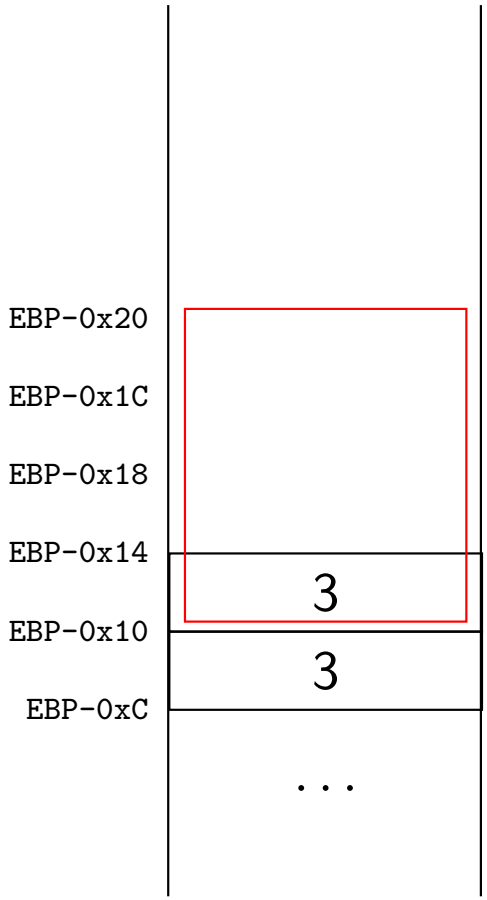
Pointeri (2)

```
;int x = 3;
mov DWORD PTR [ebp-0x10],0x3
;f1(x);
mov eax,DWORD PTR [ebp-0x10]
push eax
call 57d <f1>;copie
add esp,0x4
;printf("%d\n", x);
...
;f2(&x);
sub esp,0xc
lea eax,[ebp-0x10]
push eax
call 590 <f2>;referinta
add esp,0x10
;printf("%d\n", x);
...
```



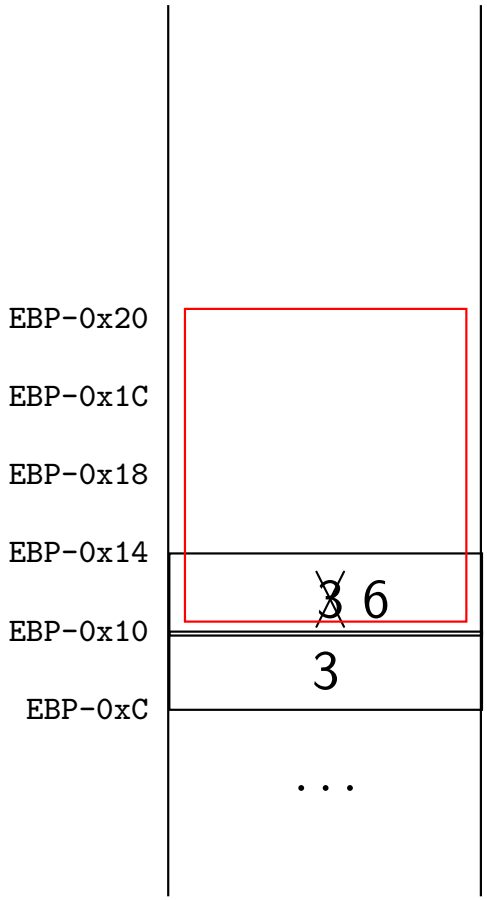
Pointeri (2)

```
;int x = 3;
mov DWORD PTR [ebp-0x10],0x3
;f1(x);
mov eax,DWORD PTR [ebp-0x10]
push eax
call 57d <f1>;copie
add esp,0x4
;printf("%d\n", x);
...
;f2(&x);
sub esp,0xc
lea eax,[ebp-0x10]
push eax
call 590 <f2>;referinta
add esp,0x10
;printf("%d\n", x);
...
```



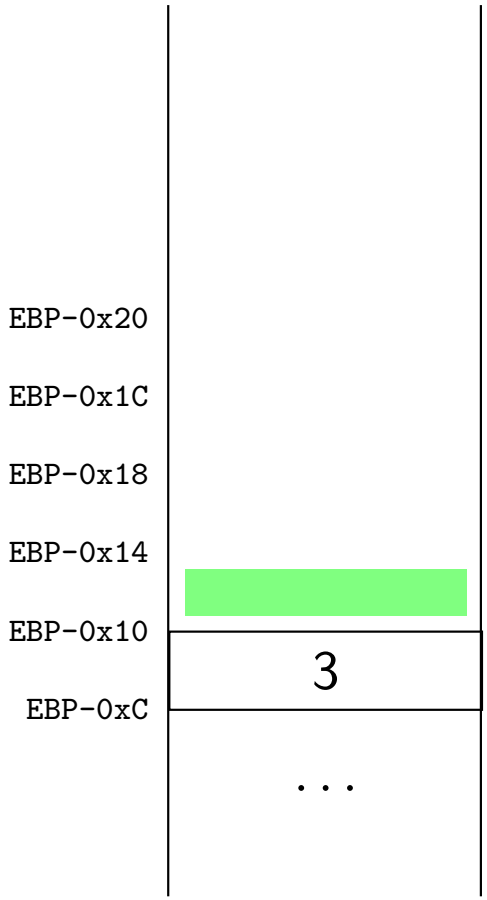
Pointeri (2)

```
;int x = 3;
mov DWORD PTR [ebp-0x10],0x3
;f1(x);
mov eax,DWORD PTR [ebp-0x10]
push eax
call 57d <f1>;copie
add esp,0x4
;printf("%d\n", x);
...
;f2(&x);
sub esp,0xc
lea eax,[ebp-0x10]
push eax
call 590 <f2>;referinta
add esp,0x10
;printf("%d\n", x);
...
```



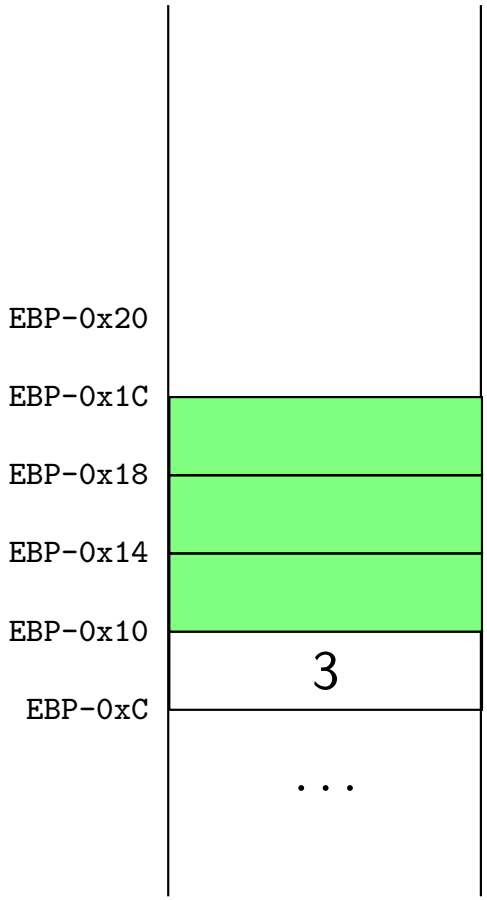
Pointeri (2)

```
;int x = 3;
mov DWORD PTR [ebp-0x10],0x3
;f1(x);
mov eax,DWORD PTR [ebp-0x10]
push eax
call 57d <f1>;copie
add esp,0x4
;printf("%d\n", x);
...
;f2(&x);
sub esp,0xc
lea eax,[ebp-0x10]
push eax
call 590 <f2>;referinta
add esp,0x10
;printf("%d\n", x);
...
```



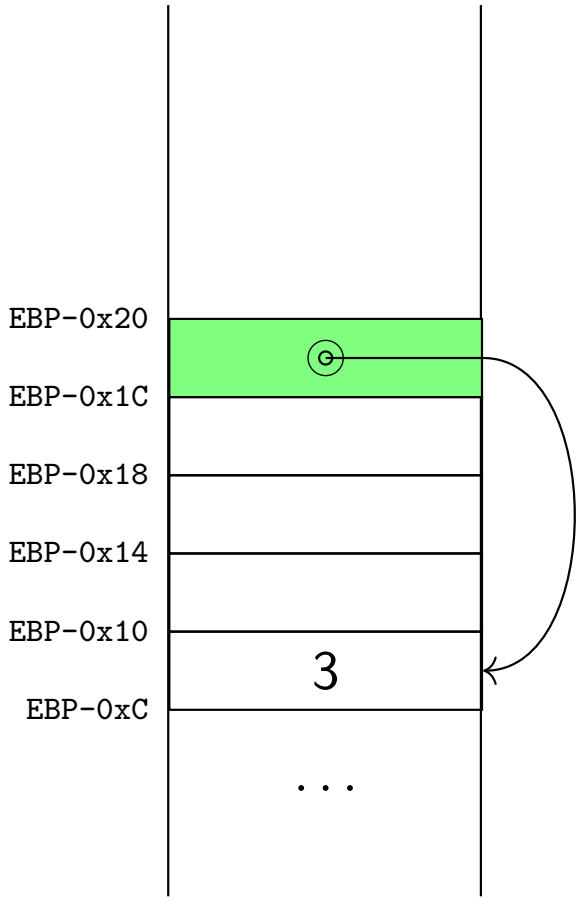
Pointeri (2)

```
;int x = 3;
mov DWORD PTR [ebp-0x10],0x3
;f1(x);
mov eax,DWORD PTR [ebp-0x10]
push eax
call 57d <f1>;copie
add esp,0x4
;printf("%d\n", x);
...
;f2(&x);
sub esp,0xc
lea eax,[ebp-0x10]
push eax
call 590 <f2>;referinta
add esp,0x10
;printf("%d\n", x);
...
```



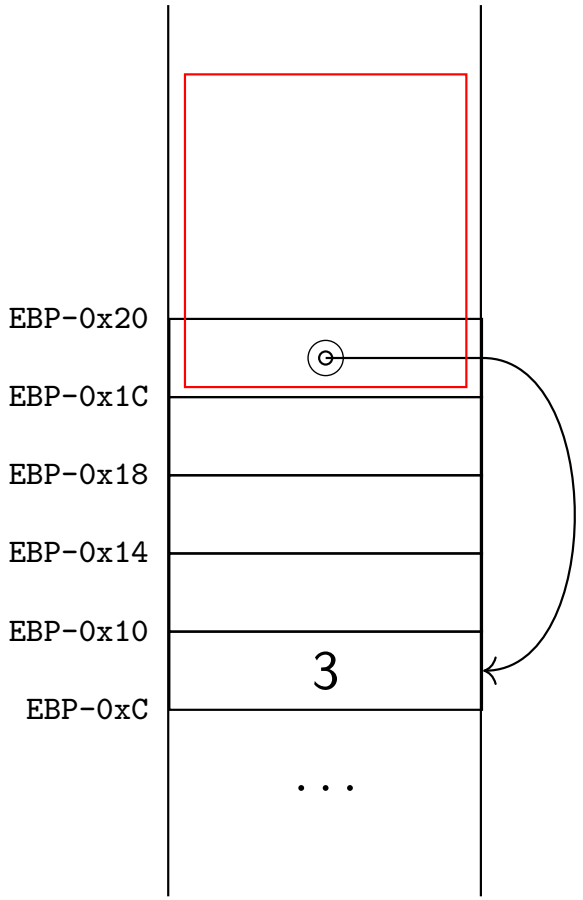
Pointeri (2)

```
;int x = 3;
mov DWORD PTR [ebp-0x10],0x3
;f1(x);
mov eax,DWORD PTR [ebp-0x10]
push eax
call 57d <f1>;copie
add esp,0x4
;printf("%d\n", x);
...
;f2(&x);
sub esp,0xc
lea eax,[ebp-0x10]
push eax
call 590 <f2>;referinta
add esp,0x10
;printf("%d\n", x);
...
```



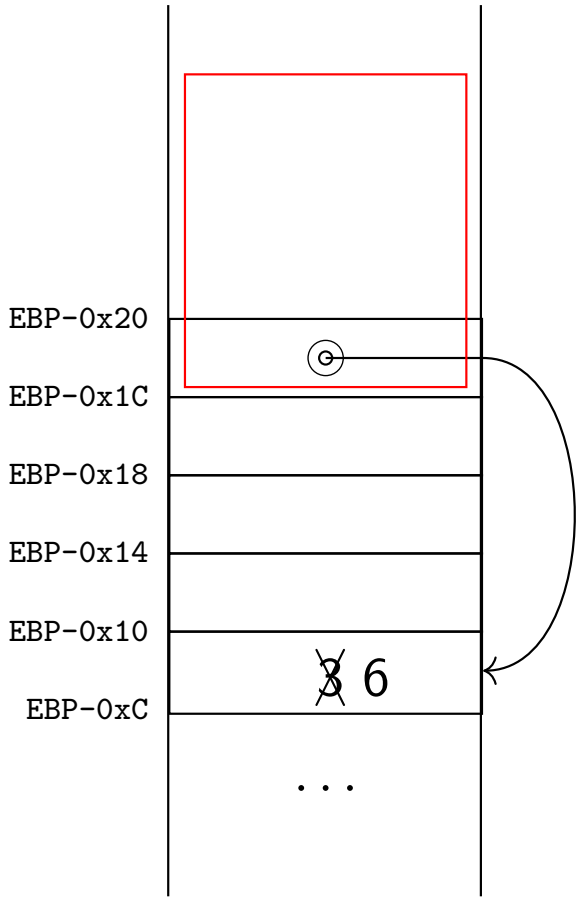
Pointeri (2)

```
;int x = 3;
mov DWORD PTR [ebp-0x10],0x3
;f1(x);
mov eax,DWORD PTR [ebp-0x10]
push eax
call 57d <f1>;copie
add esp,0x4
;printf("%d\n", x);
...
;f2(&x);
sub esp,0xc
lea eax,[ebp-0x10]
push eax
call 590 <f2>;referinta
add esp,0x10
;printf("%d\n", x);
...
```



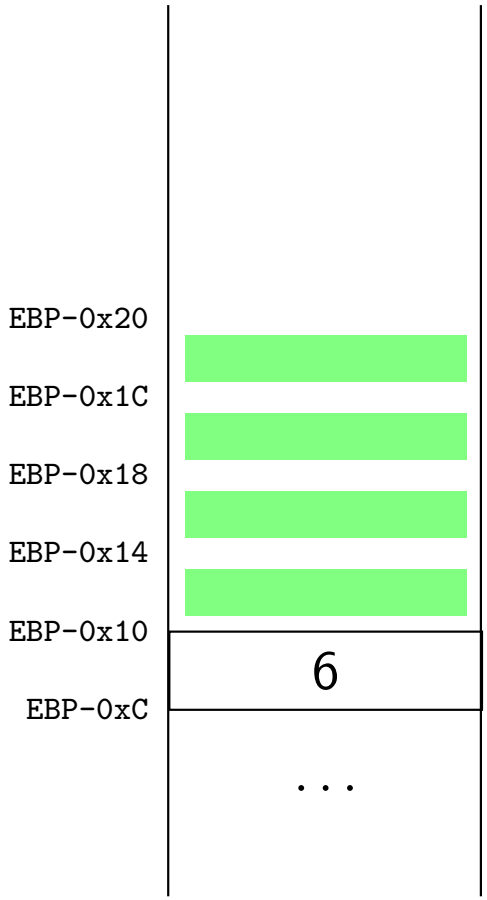
Pointeri (2)

```
;int x = 3;
mov DWORD PTR [ebp-0x10],0x3
;f1(x);
mov eax,DWORD PTR [ebp-0x10]
push eax
call 57d <f1>;copie
add esp,0x4
;printf("%d\n", x);
...
;f2(&x);
sub esp,0xc
lea eax,[ebp-0x10]
push eax
call 590 <f2>;referinta
add esp,0x10
;printf("%d\n", x);
...
```



Pointeri (2)

```
;int x = 3;
mov DWORD PTR [ebp-0x10],0x3
;f1(x);
mov eax,DWORD PTR [ebp-0x10]
push eax
call 57d <f1>;copie
add esp,0x4
;printf("%d\n", x);
...
;f2(&x);
sub esp,0xc
lea eax,[ebp-0x10]
push eax
call 590 <f2>;referinta
add esp,0x10
;printf("%d\n", x);
...
```



Vectorii sunt pointeri (1)

test_array_ptr.c:

```
#include <stdio.h>
void f(int *v) {
    v[0] = 25;
    *(v + 2) = 17;
    *((char*)v + 5) = 1;
}
int main() {
    int v[] = {1, 2, 3, 4};
    int i, n = sizeof(v)/sizeof(v[0]);
    f(v);
    for(i=0; i<n; i++){
        printf("%d ", v[i]);
    }
    printf("\n");
    return 0;
}
```

- variabila v reprezintă un pointer la începutul vectorului
- adunând un întreg la un pointer, se adună de fapt întregul ori dimensiunea tipului pointer-ului

Vectorii sunt pointeri (1)

test_array_ptr.c:

```
#include <stdio.h>
void f(int *v) {
    v[0] = 25;
    *(v + 2) = 17;
    *((char*)v + 5) = 1;
}
int main() {
    int v[] = {1, 2, 3, 4};
    int i, n = sizeof(v)/sizeof(v[0]);
    f(v);
    for(i=0; i<n; i++){
        printf("%d ", v[i]);
    }
    printf("\n");
    return 0;
}
```

- variabila v reprezintă un pointer la începutul vectorului
- adunând un întreg la un pointer, se adună de fapt întregul ori dimensiunea tipului pointer-ului

```
$ ./test_array_ptr
```

Vectorii sunt pointeri (1)

test_array_ptr.c:

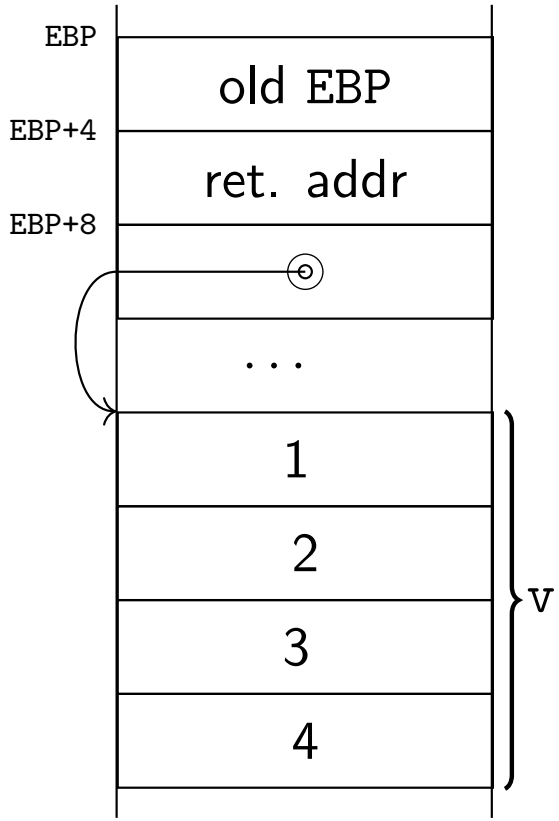
```
#include <stdio.h>
void f(int *v) {
    v[0] = 25;
    *(v + 2) = 17;
    *((char*)v + 5) = 1;
}
int main() {
    int v[] = {1, 2, 3, 4};
    int i, n = sizeof(v)/sizeof(v[0]);
    f(v);
    for(i=0; i<n; i++){
        printf("%d ", v[i]);
    }
    printf("\n");
    return 0;
}
```

- variabila v reprezintă un pointer la începutul vectorului
- adunând un întreg la un pointer, se adună de fapt întregul ori dimensiunea tipului pointer-ului

```
$ ./test_array_ptr

25 258 17 4
```

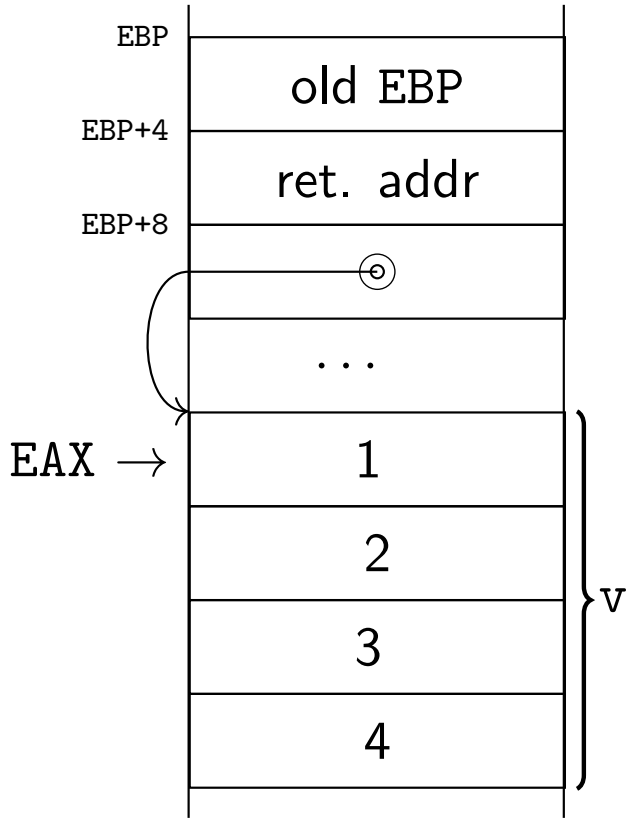
Vectorii sunt pointeri (2)

```
;void f(int *v) {
push ebp
mov ebp,esp
...
;v[0] = 25;
mov eax,DWORD PTR [ebp+0x8]
mov DWORD PTR [eax],0x19
;*(v + 2) = 17;
mov eax,DWORD PTR [ebp+0x8]
add eax,0x8
mov DWORD PTR [eax],0x11
;*((char*)v + 5) = 1;
mov eax,DWORD PTR [ebp+0x8]
add eax,0x5
mov BYTE PTR [eax],0x1
...
```



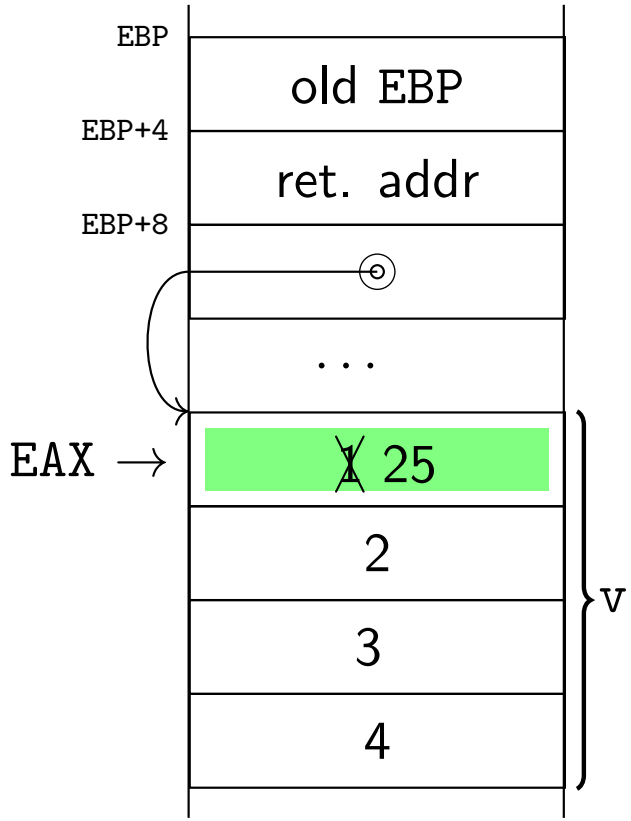
Vectorii sunt pointeri (2)

```
;void f(int *v) {
push ebp
mov ebp,esp
...
;v[0] = 25;
mov eax,DWORD PTR [ebp+0x8]
mov DWORD PTR [eax],0x19
;*(v + 2) = 17;
mov eax,DWORD PTR [ebp+0x8]
add eax,0x8
mov DWORD PTR [eax],0x11
;*((char*)v + 5) = 1;
mov eax,DWORD PTR [ebp+0x8]
add eax,0x5
mov BYTE PTR [eax],0x1
...
```



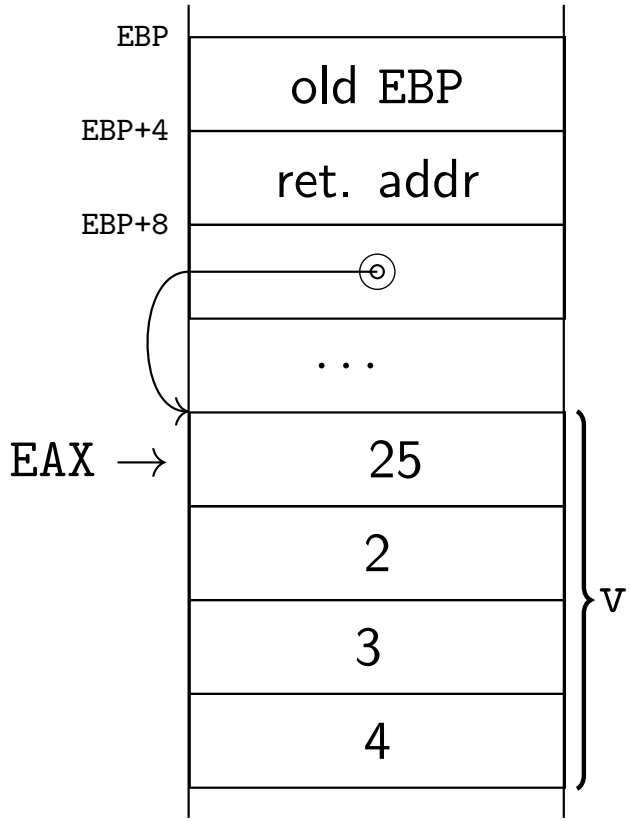
Vectorii sunt pointeri (2)

```
;void f(int *v) {
push ebp
mov ebp,esp
...
;v[0] = 25;
mov eax,DWORD PTR [ebp+0x8]
mov DWORD PTR [eax],0x19
;*(v + 2) = 17;
mov eax,DWORD PTR [ebp+0x8]
add eax,0x8
mov DWORD PTR [eax],0x11
;*((char*)v + 5) = 1;
mov eax,DWORD PTR [ebp+0x8]
add eax,0x5
mov BYTE PTR [eax],0x1
...
```



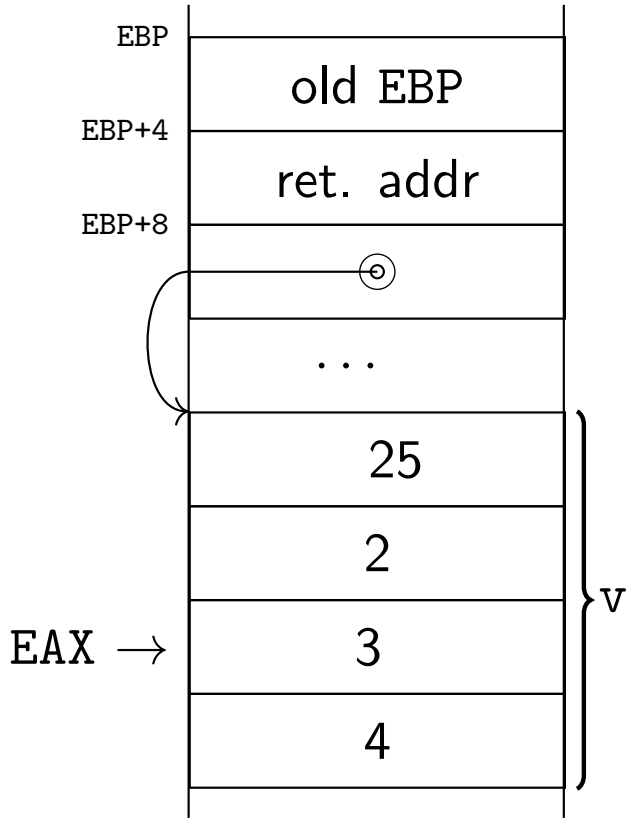
Vectorii sunt pointeri (2)

```
;void f(int *v) {
push ebp
mov ebp,esp
...
;v[0] = 25;
mov eax,DWORD PTR [ebp+0x8]
mov DWORD PTR [eax],0x19
;*(v + 2) = 17;
mov eax,DWORD PTR [ebp+0x8]
add eax,0x8
mov DWORD PTR [eax],0x11
;*((char*)v + 5) = 1;
mov eax,DWORD PTR [ebp+0x8]
add eax,0x5
mov BYTE PTR [eax],0x1
...
```



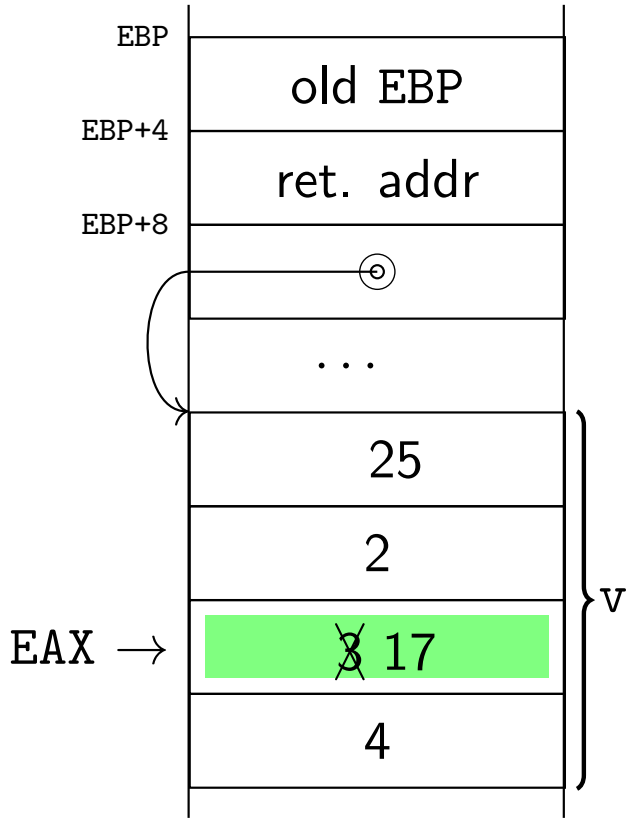
Vectorii sunt pointeri (2)

```
;void f(int *v) {
push ebp
mov ebp,esp
...
;v[0] = 25;
mov eax,DWORD PTR [ebp+0x8]
mov DWORD PTR [eax],0x19
;*(v + 2) = 17;
mov eax,DWORD PTR [ebp+0x8]
add eax,0x8
mov DWORD PTR [eax],0x11
;*((char*)v + 5) = 1;
mov eax,DWORD PTR [ebp+0x8]
add eax,0x5
mov BYTE PTR [eax],0x1
...
```



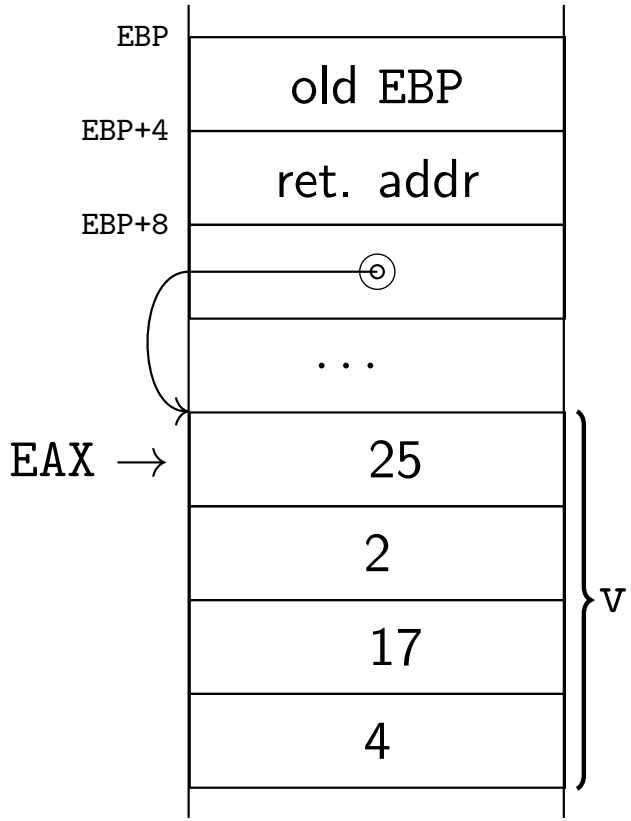
Vectorii sunt pointeri (2)

```
;void f(int *v) {
push ebp
mov ebp,esp
...
;v[0] = 25;
mov eax,DWORD PTR [ebp+0x8]
mov DWORD PTR [eax],0x19
;*(v + 2) = 17;
mov eax,DWORD PTR [ebp+0x8]
add eax,0x8
mov DWORD PTR [eax],0x11
;*((char*)v + 5) = 1;
mov eax,DWORD PTR [ebp+0x8]
add eax,0x5
mov BYTE PTR [eax],0x1
...
```



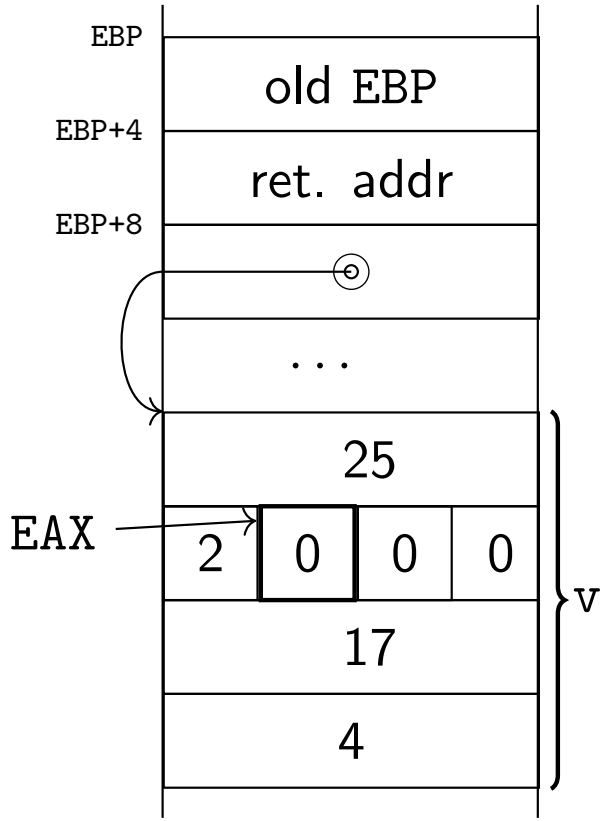
Vectorii sunt pointeri (2)

```
;void f(int *v) {
push ebp
mov ebp,esp
...
;v[0] = 25;
mov eax,DWORD PTR [ebp+0x8]
mov DWORD PTR [eax],0x19
;*(v + 2) = 17;
mov eax,DWORD PTR [ebp+0x8]
add eax,0x8
mov DWORD PTR [eax],0x11
;*((char*)v + 5) = 1;
mov eax,DWORD PTR [ebp+0x8]
add eax,0x5
mov BYTE PTR [eax],0x1
...
```



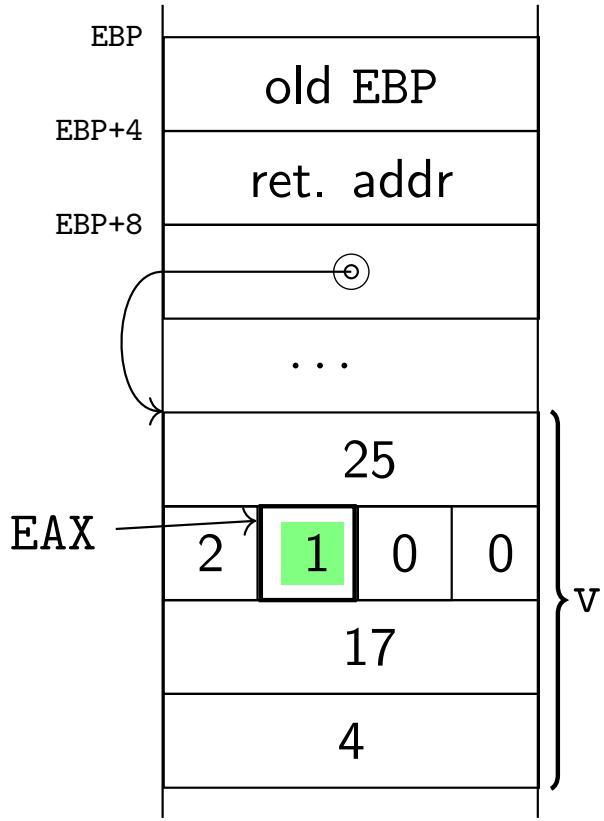
Vectorii sunt pointeri (2)

```
;void f(int *v) {
push ebp
mov ebp,esp
...
;v[0] = 25;
mov eax,DWORD PTR [ebp+0x8]
mov DWORD PTR [eax],0x19
;*(v + 2) = 17;
mov eax,DWORD PTR [ebp+0x8]
add eax,0x8
mov DWORD PTR [eax],0x11
;*((char*)v + 5) = 1;
mov eax,DWORD PTR [ebp+0x8]
add eax,0x5
mov BYTE PTR [eax],0x1
...
```



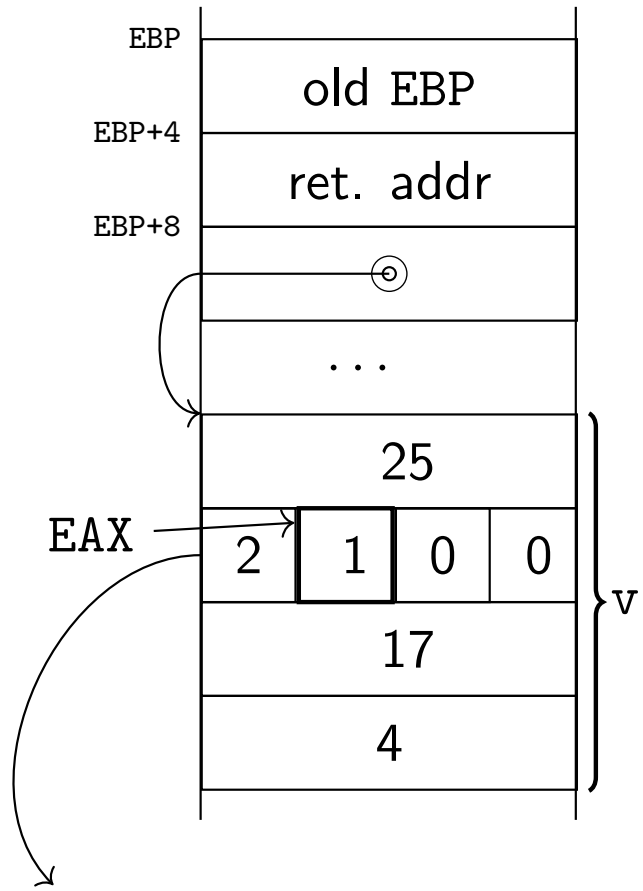
Vectorii sunt pointeri (2)

```
;void f(int *v) {
push ebp
mov ebp,esp
...
;v[0] = 25;
mov eax,DWORD PTR [ebp+0x8]
mov DWORD PTR [eax],0x19
;*(v + 2) = 17;
mov eax,DWORD PTR [ebp+0x8]
add eax,0x8
mov DWORD PTR [eax],0x11
;*((char*)v + 5) = 1;
mov eax,DWORD PTR [ebp+0x8]
add eax,0x5
mov BYTE PTR [eax],0x1
...
```



Vectorii sunt pointeri (2)

```
;void f(int *v) {
push ebp
mov ebp,esp
...
;v[0] = 25;
mov eax,DWORD PTR [ebp+0x8]
mov DWORD PTR [eax],0x19
;*(v + 2) = 17;
mov eax,DWORD PTR [ebp+0x8]
add eax,0x8
mov DWORD PTR [eax],0x11
;*((char*)v + 5) = 1;
mov eax,DWORD PTR [ebp+0x8]
add eax,0x5
mov BYTE PTR [eax],0x1
...
```



Lucrul cu structuri de date

```
struct Point {
    int x;
    int y;
};

struct MyStruct {
    int a;
    short b;
    struct Point p;
    char c[5];
};

int main(void){
    ...
```

Lucrul cu structuri de date

```
struct Point {
    int x;
    int y;
};

struct MyStruct {
    int a;
    short b;
    struct Point p;
    char c[5];
};

int main(void){
    ...
```

Asignarea de valori pentru membri

```
struct MyStruct s;
s.a = 7;
s.b = 12;
s.p.x = 150;
s.p.y = -11;
s.c[1] = 10;
s.c[2] = 'a';
```

Lucrul cu structuri de date

```
struct Point {
    int x;
    int y;
};

struct MyStruct {
    int a;
    short b;
    struct Point p;
    char c[5];
};

int main(void){
    ...
```

Inițializarea (doar la declarație)

```
struct MyStruct s = {
    .a=7, .b=12,
    .p={.x=150, .y=11},
    .c={0, 0, 0, 0, 0}
};
```

Lucrul cu structuri de date

```
struct Point {
    int x;
    int y;
};

struct MyStruct {
    int a;
    short b;
    struct Point p;
    char c[5];
};

int main(void){
    ...
```

Pointeri la structuri:

```
struct MyStruct s = {...};
struct MyStruct *ps;
```

Cum adresăm membrul a?

- din s: `s.a`
- din ps (v1): `(*ps).a`
- din ps (v2): `ps->a`

Se preferă varianta 2 (->).

Lucrul cu structuri de date

```
struct Point {
    int x;
    int y;
};

struct MyStruct {
    int a;
    short b;
    struct Point p;
    char c[5];
};

int main(void){
    ...
```

Pointeri la structuri:

```
struct MyStruct s = {...};
struct MyStruct *ps;
```

Cum adresăm membrul a?

- din s: `s.a`
- din ps (v1): `(*ps).a`
- din ps (v2): `ps->a`

Se preferă varianta 2 (->).
Cum adresăm membrul x din punctul p?

- `s.p.x`
- `ps->p.x`

Lucrul cu structuri de date

```
struct Point {
    int x;
    int y;
};

struct MyStruct {
    int a;
    short b;
    struct Point p;
    char c[5];
};

int main(void){
    ...
```

Dimensiunea structurilor

```
printf("Point size = %d\n",
    sizeof(struct Point));
printf("MyStruct size = %d\n",
    sizeof(struct MyStruct));
```

Lucrul cu structuri de date

```
struct Point {
    int x;
    int y;
};

struct MyStruct {
    int a;
    short b;
    struct Point p;
    char c[5];
};

int main(void){
    ...
```

Dimensiunea structurilor

```
    printf("Point size = %d\n",
        sizeof(struct Point));
    printf("MyStruct size = %d\n",
        sizeof(struct MyStruct));
```

```
Point size = 8
```

Lucrul cu structuri de date

```
struct Point {
    int x;
    int y;
};

struct MyStruct {
    int a;
    short b;
    struct Point p;
    char c[5];
};

int main(void){
    ...
```

Dimensiunea structurilor

```
    printf("Point size = %d\n",
        sizeof(struct Point));
    printf("MyStruct size = %d\n",
        sizeof(struct MyStruct));
```

```
Point size = 8
MyStruct size = 24
```

Lucrul cu structuri de date

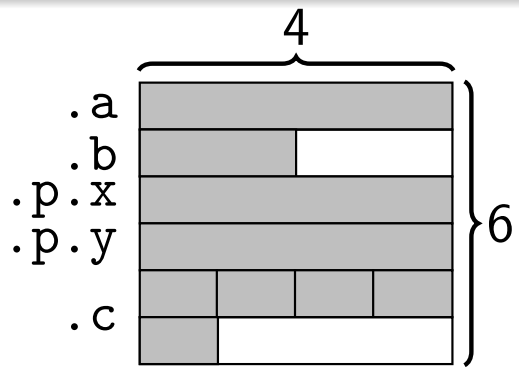
```
struct Point {
    int x;
    int y;
};

struct MyStruct {
    int a;
    short b;
    struct Point p;
    char c[5];
};

int main(void){
    ...
```

Dimensiunea structurilor

```
printf("Point size = %d\n",
    sizeof(struct Point));
printf("MyStruct size = %d\n",
    sizeof(struct MyStruct));
```

```
Point size = 8
MyStruct size = 24
```

Fiecare câmp din structură va fi aliniat la *DWORD* (4 bytes).



Lucrul cu structuri de date

```
struct Point {
    int x;
    int y;
};

struct MyStruct {
    int a;
    short b;
    struct Point p;
    char c[5];
};

int main(void){
    ...
```

Alinierea structurilor

Ce facem dacă vrem o structură aliniată la *BYTE*?

gcc (Linux)

```
struct __attribute__((packed)) MyStruct {
    ...
};
```

Visual Studio (Windows)

```
#pragma pack(push, 1)
struct MyStruct {
    ...
};
#pragma pack(pop)
```

Lucrul cu structuri de date

```
struct Point {
    int x;
    int y;
};

struct MyStruct {
    int a;
    short b;
    struct Point p;
    char c[5];
};

int main(void){
    ...
```

Cast la structuri

```
unsigned char v[] = {
      7, 1, 0, 0, 12, 0, 0, 0,
    150, 0, 0, 0, 11, 0, 0, 0,
    'a', 0x62, 'c', 100, 0, 0, 0, 0
};
struct MyStruct *ps = (struct MyStruct*)v;
printf("a=%d b=%d p=(%d, %d), c='%s'\n",
    ps->a, ps->b, ps->p.x, ps->p.y, ps->c);
```

Ce se afișează?

Lucrul cu structuri de date

```
struct Point {
    int x;
    int y;
};

struct MyStruct {
    int a;
    short b;
    struct Point p;
    char c[5];
};

int main(void){
    ...
```

Cast la structuri

```
unsigned char v[] = {
      7, 1, 0, 0, 12, 0, 0, 0,
    150, 0, 0, 0, 11, 0, 0, 0,
    'a', 0x62, 'c', 100, 0, 0, 0, 0
};
struct MyStruct *ps = (struct MyStruct*)v;
printf("a=%d b=%d p=(%d, %d), c='%s'\n",
    ps->a, ps->b, ps->p.x, ps->p.y, ps->c);
```

Ce se afișează?

a=263 b=12 p=(150, 11), c='abcd'

Alocarea dinamică a memoriei (1)

Funcțiile `malloc` și `free`

```
int *v = (int*)malloc(10000 * sizeof(int));
struct MyStruct *ps = (struct MyStruct*)
                      malloc(sizeof(struct MyStruct));
...
free(v);
free(ps);
```

- `malloc` alocă memorie pe *heap* și returnează un pointer la memoria alocată.
- `free` eliberează memoria alocată

Alocarea dinamică a memoriei (2)

Când să alocăm dinamic?

- când lucrăm cu structuri dinamice
  - liste înlănțuite
  - arbori
- când nu știm de la început dimensiunea datelor
  - nu ne permitem să declarăm de la început dimensiunea maximă
- când dimensiunea datelor e prea mare pentru a încăpea pe stivă
  - dimensiunea stivei default pe Linux e 8MB
  - dimensiunea stivei default pe Windows e 1MB

Notă: alocarea dinamică a memoriei este mai lentă decât declarația unui obiect local pe stivă.

Alocarea dinamică a memoriei (3)

Memory leaks

- orice memorie care se alocă dinamic trebuie eliberată explicit (cu `free`)
- limbajul C nu are *garbage collector*
- orice zonă alocată rămâne în memorie până la finalul programului
- uneori memoria nu se eliberează în aceeași funcție în care a fost alocată
  - *exemplu*: o funcție alocă și returnează un pointer la memoria alocată
  - cine primește pointer-ul trebuie să se asigure că în final memoria va fi eliberată

Operații pe string-uri

String-uri

În C, un string este un vector de caractere ce conține terminatorul 0.

- poate fi tratat ca orice alt vector
- în `string.h` avem o colecție de funcții utile pentru manipularea string-urilor
  - `strlen`
  - `strcpy`
  - `strcmp`
  - `strchr`
  - `strstr`

Nu utilizați funcții pe string-uri dacă nu sunteți 100% siguri că sunt NULL-terminated.

Cuprins

Conceptul de debugger

Un debugger ne permite să trasăm execuția codului pas cu pas, de obicei cu scopul de a găsi greșeli.

- execuție linie cu linie
- breakpoints
- vizualizare variabile / memorie
- vizualizare stivă de apeluri

Conceptul de debugger

Un debugger ne permite să trasăm execuția codului pas cu pas, de obicei cu scopul de a găsi greșeli.

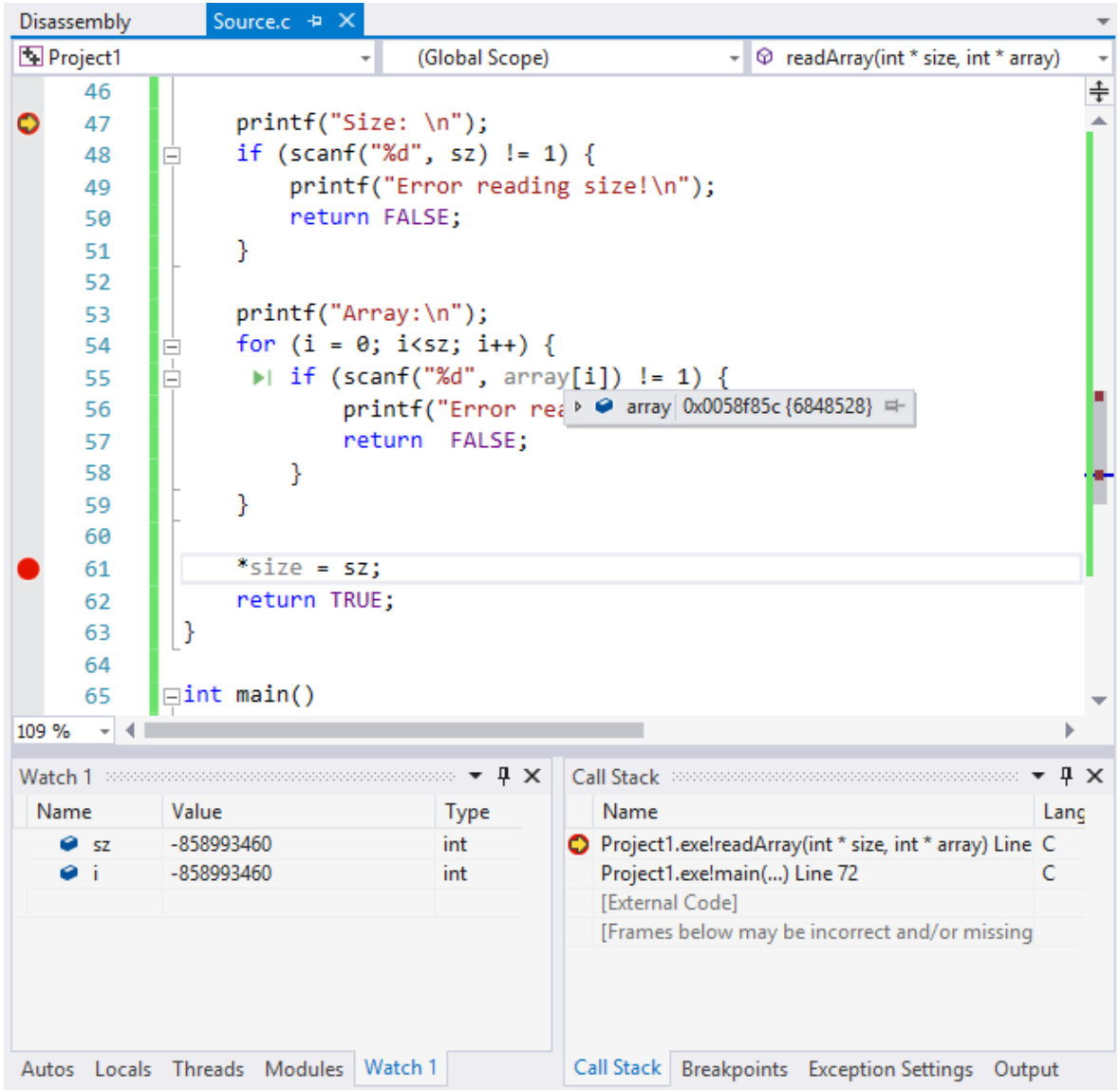
- execuție linie cu linie
- breakpoints
- vizualizare variabile / memorie
- vizualizare stivă de apeluri

De obicei programul se compilează cu anumite opțiuni (-g pentru *gcc*), astfel încât să includă informații de debug (numere de linii, nume de funcții, variabile).

Debugging din GUI (1)

- cel mai simplu mod de a face debugging
- breakpoints prin click-uri de mouse pe marginea stângă
- examinare variabile prin *mouse over* sau click dreapta $\rightarrow$ *Add Watch*
- trasare program
  - *step into* - înaintează o linie, intră în funcții
  - *step over* - înaintează o linie, execută funcțiile dintr-un singur pas
  - *continue* - înaintează până la următorul breakpoint sau până la final

Debugging din GUI (2)



Utilizarea GDB

Se rulează: `gdb <nume_prog>`
În shell-ul interactiv care apare, avem următoarele comenzi:

- `break test.c:12`
- `run`
- `continue`
- `step` (step into)
- `next` (step over)
- `bt` (*backtrace* - stiva de apeluri)
- `print myvar` (afișare valoare variabilă)

Post-mortem debugging

- ne permite să examinăm starea unui program din momentul crash-ului.
- util în special atunci când crash-ul se reproduce doar pe un alt sistem

- activăm *core dump*-ul pentru terminalul curent
  `ulimit -c unlimited`
- rulăm programul (compilat cu `-g`) și reproducem crash-ul
- se generează un fișier numit `core`
- încărcăm programul împreună cu *core dump*-ul în gdb
  `gdb <nume_prog> core`

Utilizarea *valgrind* pentru memory leaks

- *valgrind* e o unealtă care permite analiza problemelor unui program, în timpul rulării
- pentru analiza memory leak-urilor, interceptează `malloc` și `free` (și alte funcții similare) și ține evidența memoriei alocate
- la finalul se afișează un raport cu memoria rămasă neeliberată

Utilizarea *valgrind* pentru memory leaks

- *valgrind* e o unealtă care permite analiza problemelor unui program, în timpul rulării
- pentru analiza memory leak-urilor, interceptează `malloc` și `free` (și alte funcții similare) și ține evidența memoriei alocate
- la finalul se afișează un raport cu memoria rămasă neeliberată

```
$ valgrind ./<nume_prog> [<argumente>]
...
==9347== LEAK SUMMARY:
==9347==    definitely lost: 55 bytes in 5 blocks
==9347==    indirectly lost: 0 bytes in 0 blocks
==9347==      possibly lost: 0 bytes in 0 blocks
==9347==    still reachable: 43 bytes in 3 blocks
==9347==         suppressed: 0 bytes in 0 blocks
```

Utilizarea *valgrind* pentru memory leaks

- *valgrind* e o unealtă care permite analiza problemelor unui program, în timpul rulării
- pentru analiza memory leak-urilor, interceptează `malloc` și `free` (și alte funcții similare) și ține evidența memoriei alocate
- la finalul se afișează un raport cu memoria rămasă neeliberată

```
$ valgrind ./<nume_prog> [<argumente>]
...
==9347== LEAK SUMMARY:
==9347==    definitely lost: 55 bytes in 5 blocks
==9347==    indirectly lost: 0 bytes in 0 blocks
==9347==      possibly lost: 0 bytes in 0 blocks
==9347==    still reachable: 43 bytes in 3 blocks
==9347==         suppressed: 0 bytes in 0 blocks
```

- dacă se detectează leak-uri, se mai rulează odată cu opțiunile `--leak-check=full --show-leak-kinds=all`

Cuprins

E normal să facem greșeli

- se întâmplă foarte rar să scriem un program care să "meargă din prima"
- se găsesc zilnic bug-uri în software comercial, ce rulează la milioane de utilizatori
- există metrici care măsoară numărul de bug-uri raportat la mii de linii de cod

E normal să facem greșeli

- se întâmplă foarte rar să scriem un program care să "meargă din prima"
- se găsesc zilnic bug-uri în software comercial, ce rulează la milioane de utilizatori
- există metrici care măsoară numărul de bug-uri raportat la mii de linii de cod

Un programator bun nu scrie programe perfecte de la început, dar știe să testeze, să identifice problemele și să le repare.

E normal să facem greșeli

- se întâmplă foarte rar să scriem un program care să "meargă din prima"
- se găsesc zilnic bug-uri în software comercial, ce rulează la milioane de utilizatori
- există metrici care măsoară numărul de bug-uri raportat la mii de linii de cod

Un programator bun nu scrie programe perfecte de la început, dar știe să testeze, să identifice problemele și să le repare.

- testarea și depanarea sunt deprinderi la fel de importante ca și scrierea de cod
  - a înțelege un program gata scris nu echivalează d.p.d.v. didactic cu a dezvolta unul de la zero

Utilizarea codului din alte surse

Nu luați **niciodată** cu copy-paste **cod pe care nu îl înțelegeți**.

- în procesul de învățare, ar fi ideal să nu folosiți deloc cod din alte surse, cu excepția celor indicate
- dacă totuși luați cod din altă parte
  - asigurați-vă că îl înțelegeți
  - menționați sursa într-un comentariu

Editor sau IDE?

IDE = Integrated Development Environment

Un IDE oferă mai multe facilități:

- compilare, rulare și debugging
- auto-completion
- code refactoring

Editor sau IDE?

IDE = Integrated Development Environment

Un IDE oferă mai multe facilități:

- compilare, rulare și debugging
- auto-completion
- code refactoring

Avantaje:

- productivitate ridicată
- automatizarea proceselor frecvente din dezvoltare

Editor sau IDE?

IDE = Integrated Development Environment

Un IDE oferă mai multe facilități:

- compilare, rulare și debugging
- auto-completion
- code refactoring

Avantaje:

- productivitate ridicată
- automatizarea proceselor frecvente din dezvoltare

Dezavantaje:

- în timpul învățării are efectul de "training wheels"
  - când învățăm un limbaj nou sau o tehnologie nouă, e preferabil să folosim un editor la început
- în general este mai lent ca un editor

Atenție la variabilele neinițializate

- declarația unei variabile se traduce în Assembly prin decrementarea registrului ESP (facem loc pe stivă)
  - valoarea inițială va fi cea găsită pe stivă la momentul respectiv
- memoria alocată dinamic este deasemenea neinițializată
- după `free()`, un pointer pointează la o locație invalidă

Atenție la variabilele neinițializate

- declarația unei variabile se traduce în Assembly prin decrementarea registrului ESP (facem loc pe stivă)
  - valoarea inițială va fi cea găsită pe stivă la momentul respectiv
- memoria alocată dinamic este deasemenea neinițializată
- după `free()`, un pointer pointează la o locație invalidă

Inițializați variabilele acolo unde au fost declarate sau cât mai aproape de declarație.

După eliberarea memoriei, asignați valoarea `NULL` pointerilor.

Evitați variabilele globale

- o variabilă globală e vizibilă de oriunde din program
  - dacă nu e constantă, oricine o poate modifica
- codul care lucrează cu variabile globale care nu sunt constante nu e thread-safe (conceptul se va discuta la cursul despre thread-uri)

Evitați variabilele globale

- o variabilă globală e vizibilă de oriunde din program
  - dacă nu e constantă, oricine o poate modifica
- codul care lucrează cu variabile globale care nu sunt constante nu e thread-safe (conceptul se va discuta la cursul despre thread-uri)

Nu utilizați variabile globale neconstante decât dacă este absolut necesar.

- e preferabil să transmitem informațiile contextuale unei funcții prin parametri

Codați defensiv

Nu presupuneți niciodată că programul vostru va fi folosit conform specificațiilor.

Codați defensiv

Nu presupuneți niciodată că programul vostru va fi folosit conform specificațiilor.

- validați orice input primit de la user
  - chiar dacă cereți să se introducă un întreg, utilizatorul poate tasta un string
  - nu vă bazați pe faptul că textul introdus respectă dimensiunea maximă declarată

Codați defensiv

Nu presupuneți niciodată că programul vostru va fi folosit conform specificațiilor.

- validați orice input primit de la user
  - chiar dacă cereți să se introducă un întreg, utilizatorul poate tasta un string
  - nu vă bazați pe faptul că textul introdus respectă dimensiunea maximă declarată
- un fișier cu date de intrare poate conține **orice**
  - e ok ca un program să "refuze" să lucreze cu un fișier invalid
  - nu e ok să crape / să furnizeze rezultate eronate

Codați defensiv

Nu presupuneți niciodată că programul vostru va fi folosit conform specificațiilor.

- validați orice input primit de la user
  - chiar dacă cereți să se introducă un întreg, utilizatorul poate tasta un string
  - nu vă bazați pe faptul că textul introdus respectă dimensiunea maximă declarată
- un fișier cu date de intrare poate conține **orice**
  - e ok ca un program să "refuze" să lucreze cu un fișier invalid
  - nu e ok să crape / să furnizeze rezultate eronate
- verificați valoarea returnată de orice funcție / apel de sistem
  - nu vă bazați pe faptul că deschiderea unui fișier a reușit
  - nu vă bazați pe faptul că `malloc` reușește mereu să aloce memorie

Tratarea warning-urilor

- compilatorul ne avertizează când întâlnește cod care “pare” în neregulă
  - testarea egalității într-un `if` cu un singur `=`
  - cod ce nu are nici un efect
  - utilizarea de variabile neinițializate
  - apel de `printf` cu format necorespunzător
- dacă decidem să ignorăm anumite categorii de warning-uri, putem cere asta explicit prin opțiuni de compilare
  - mai multe warning-uri irelevante ne fac să le ignorăm și pe cele relevante
- pentru varianta finală (release) a programelor se recomandă opțiunea `-Werror` (tratează warning-urile ca erori)

Scrierea de cod lizibil

cod fără comentarii

Scrierea de cod lizibil

cod fără comentarii

cod cu comentarii

Scrierea de cod lizibil

cod fără comentarii

cod cu comentarii

cod ce nu necesită comentarii

Scrierea de cod lizibil

cod fără comentarii

cod cu comentarii

cod ce nu necesită comentarii