Bitdefender. Awake.

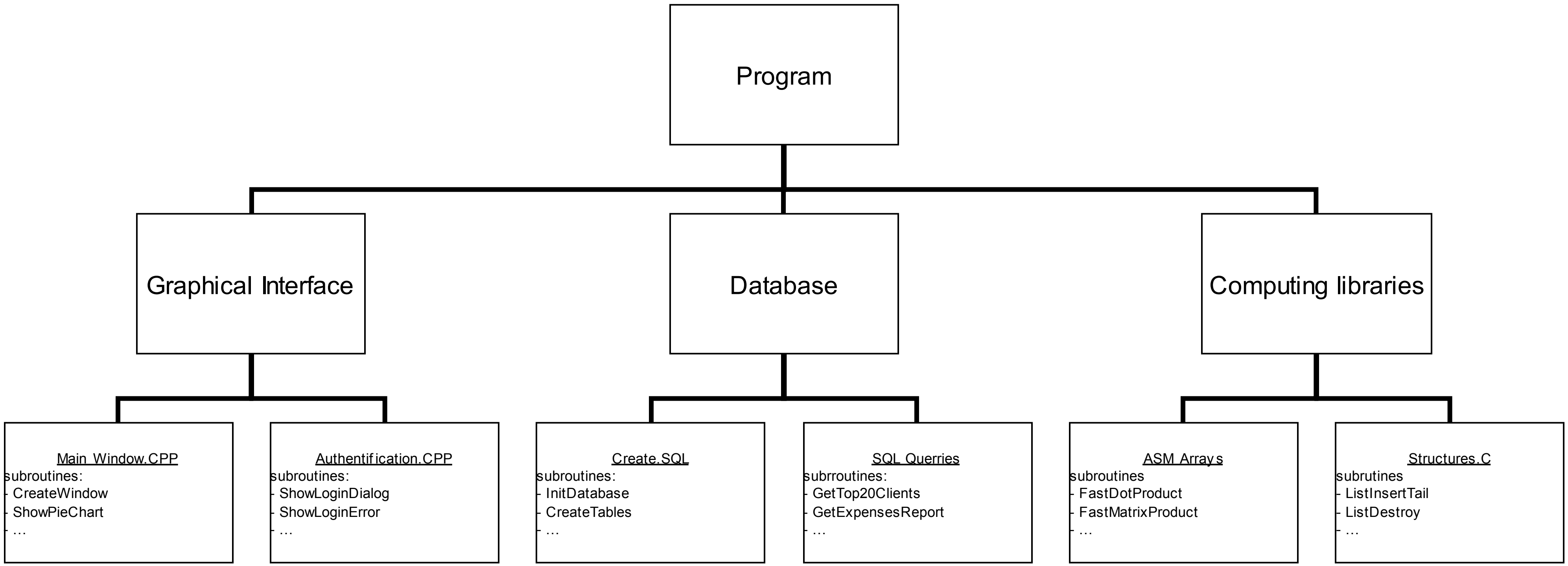
Multi-module programming

Marius Vanța
mvanta@bitdefender.com

1. Modular Architectures

Modular programming

- How to split a problem in sub-problems?
  - Modularization
    - program -> logic units
    - code (of units) -> distinct files
    - Files -> subroutines



Modular programming

- Exist already some available solutions for some sub-problems?
- Reusability
  - Code files

    Code reusability and dates from assambly
    - %include directive
  - Binary files
    - *Code reuse and și dates from assambly*
    - *Codes and dates from high level languages*
    - *Libraries*

- <u>Existence of separate binary files implies SEPARATE COMPILATION !!!</u>

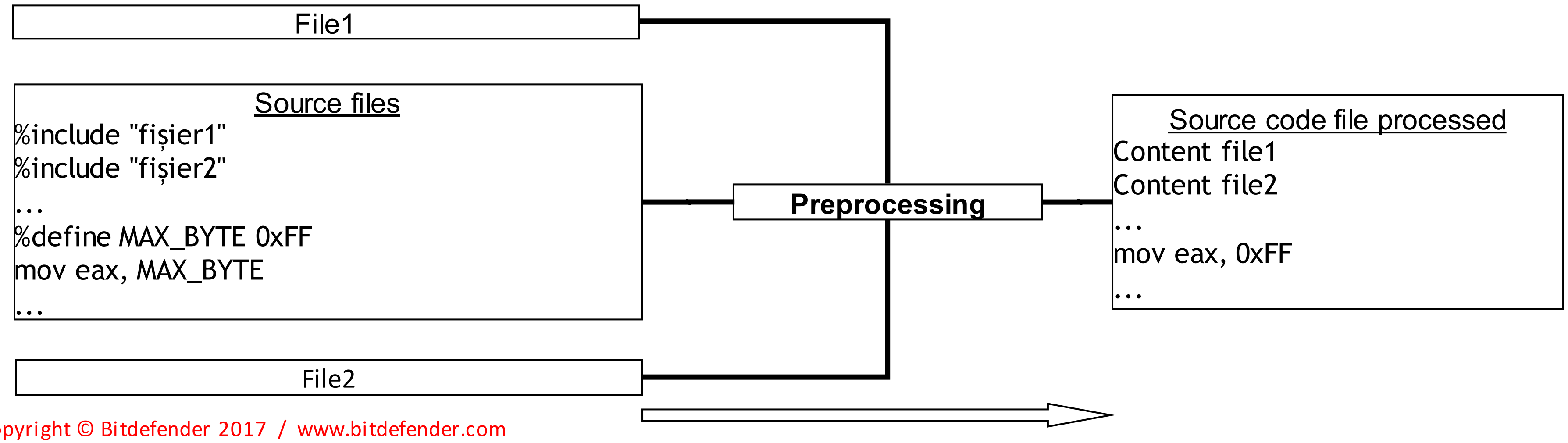
Bitdefender

2. Techniques and tools

Techniques and tools

Bitdefender

- **Static** inclusion at compilation/**assembly**: directive **%include**
  - Realise the specificity of language (but has equivalent also in other languages)
  - Modularization: allows only code division for the code written in the same language!
    - It is NOT multimodule programming !(this requires SEPARATE COMPILATION !!!)
  - Reusability: display the source code!
  - Dangerous and problematic:
    - Preprocessor mechanism -> textual concatenation of files
    - Exposes with visibility all names-> conflicts (redefinitions/redeclarations)
    - Include all the files – useful and unusefull!



Tools and techniques

Bitdefender

- How to use **%include**

```
; files constante.inc

; double inclusion
%ifndef     _CONSTANTE_INC_ ; at first inclusion, _CONSTANTE_INC_ is not defined
%define     _CONSTANTE_INC_ ; definim _CONSTANTE_INC_ -> fake condition for next inclusions


; it is recommanded that this type of files (included by other files) contain only
declarations

MAX_BYTE    equ 0xFF
MAX_WORD    equ 0xFFFF
MAX_DWORD   equ 0xFFFFFFFF
MAX_QWORD   equ 0xFFFFFFFFFFFFFFFF

%endif ; _CONSTANTE_INC_
```

Techniques and tools

- **%include** usage example

```
; file constants.inc

; protect against double inclusion
%ifndef     _CONSTANTS_INC_ ; at the first inclusion, _CONSTANTE_INC_ is not defined
%define     _CONSTANTS_INC_ ; we define _CONSTANTE_INC_ -> the condition will be false for
future inclusions


; it is recommended that files included by other files should (only) consist of declarations!
MAX_BYTE    equ 0xFF
MAX_WORD    equ 0xFFFF
MAX_DWORD   equ 0xFFFFFFFF
MAX_QWORD   equ 0xFFFFFFFFFFFFFFFF

%endif ; _CONSTANTS_INC_
```

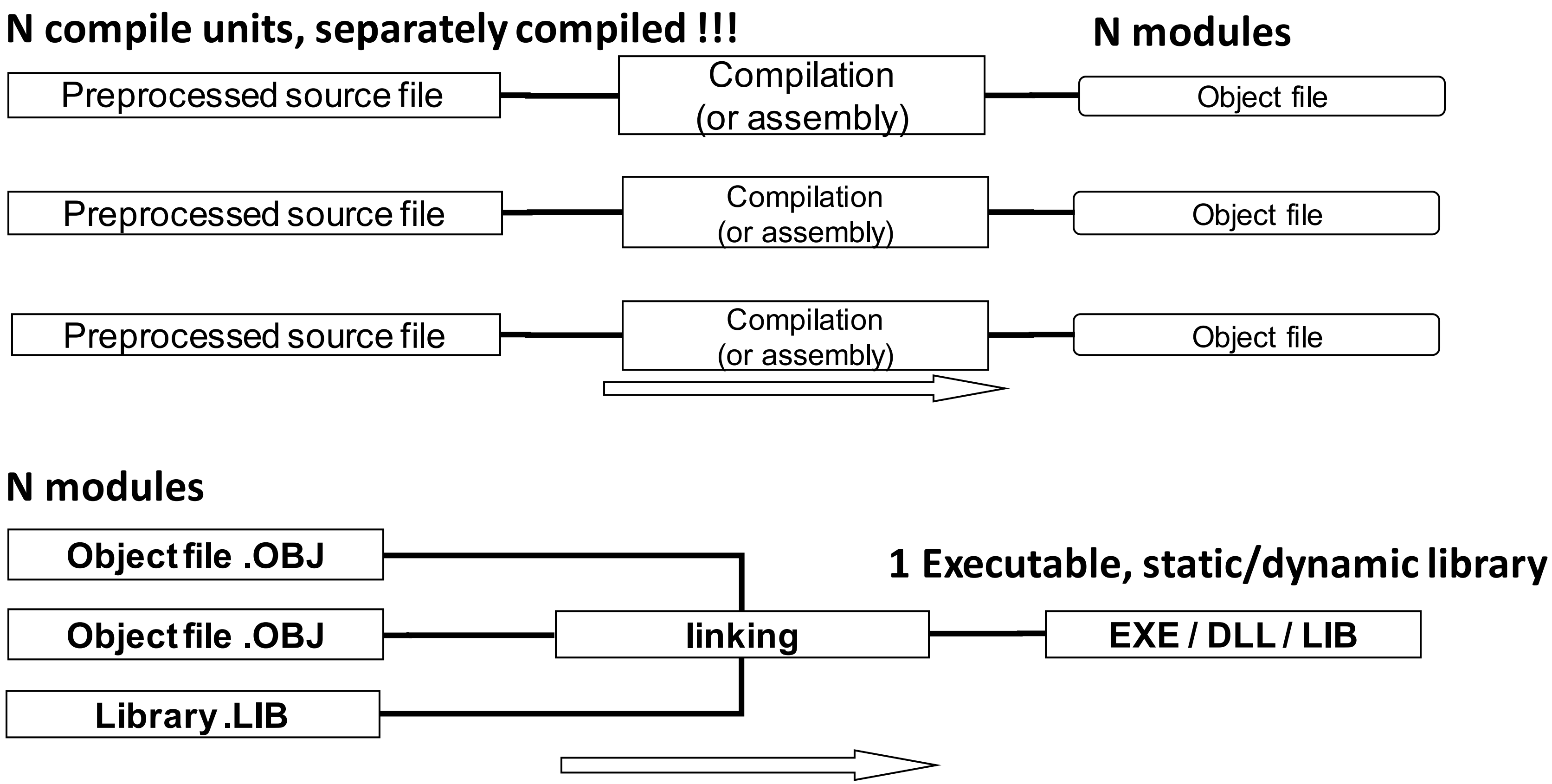
Techniques and tools

- **%include** usage example – "moving" the contents of eax in a BYTE/WORD/DWORD, according to the actual size of its value

```
; file program.asm
%include "constants.inc“
    cmp     eax, MAX_BYTE
    ja      .no_fit_in_byte               ; the value in eax fits in a BYTE?

.fits_in_byte:
    mov     [result_byte], al           ; if yes, save AL in result_byte
    jmp     .done

. no_fit_in_byte:
    cmp     eax, MAX_WORD
    ja      .no_fit_in_word               ; otherwise check if it fits in a WORD

.fits_in_word:
    mov     [resultat_word], ax        ; if yes, save AX in resultat_word
    jmp     .done

. no_fit_in_word:
    mov     [resultat_dword], eax      ; if a WORD is not enough, save all of eax
.done:
```

Techniques and tools

- **Static** linking at **link-time**
  - a step performed by the **linker** <u>after</u> assembly/compilation



Techniques and tools

- Static **linking** at **link-time** – summary of responsibilities

  Preprocessor: **text => text**

  - Performs a processing of the source *text*, resulting a intermediary *text* source
  - Can be imagined as a component of the compiler or assembler
  - May be missing, many languages do not have a preprocessor.

- Assembler: **instructions (text) => binary encoding (object file)**
  - Encodes the instructions and data (variables) from the preprocessed text source and builds an object file that consists of machine code and variable values, along with information about the content (variable names, subroutines, information about their type and visibility etc.)

- Compiler: **instructions (text) => binary encoding (object file)**
  - Identifies sequences of processor instructions through which the functionalities described in the text source ca be obtained, *then, like an assembler*, generates an object file that contains the binary codification of those instructions and the variables from the program
  - Assembling is a special case of compilation, where the processor instructions are provided directly in the text of the program and therefore the compiler does not need to select them!

- Linking: **object file => library or program**
  - Constructs the final result, specifically a program (.exe) or a library (.dll or .lib) in which it *links together* (includes) the code and binary data from the object file
  - It has no regard for which compilers or languages were used! Linking only requires that the input file follow the standard format of object files!

Techniques and tools

Bitdefender

- Static **linking** at **link-time**
  - Allows joining multiple **binary modules** (object files or static libraries) in a single file
    - Input: any number of object files (**.OBJ**) and/or static libraries (**.LIB**)
      - *Attention: not all .LIB files are static libraries!*
    - Output: .EXE or .LIB or .DLL (Dynamic-Link Library)
  - Multimodule: any number of files may be compiled separately and linked together
    - Step performed by the linker *after* compilation/assembly -> **language independent**
  - Reuse:
    - In binary form – does not expose the source code!
    - Allows inter-operability between different languages!
  - Other advantages and disadvantages:
    - The linker *can* identify and remove unused resources or perform other optimizations
    - Large program size: the program incorporates reused external resources
    - Large program size: popular libraries duplicated in most programs
  - NASM: **global** and **extern** directives
    - global *name* – allows the possibility of external reuse of this resource through its name
    - extern *name* – requests access to the specified resource; it needs to be public!

* there is no reserved word for exporting in C

=> static is the reserved word for stopping the exporting

* global is an exporting mech.

* extern is an importing mech.

Techniques and tools

Bitdefender

- Static **linking** at **link-time – nasm requirements**
  - <u>Resources are shared based on a mutual agreement</u>
  - *Export* through **global** name1, name2, ...
    - It makes the resources available to <u>any</u> interested file
  - *Import* through **extern** name1, name2, ...
    - Request access, <u>no matter from which file</u> the source is provided
  - Request without availability = error!
    - Only resources that are exported somewhere can be imported
  - Availability without request is allowed. Why?
    - Answer: even if none of the program's modules does not request/use a resource, it may be used in a future version or by a different program
  - <u>High level programming languages also offer syntactical constructions with an equivalent purpose!</u>
    - Example: in C
      - *Availability is automatic/implicit, however access may be restricted by using the keyword **static***
      - *Access request is (also) done through the keyword **extern***

Required for written exam

Techniques and tools

- Static linking at **linkediting – nasm requirements**
  - global and extern directives used in practice

```
; FILE1.ASM
global Var1, Subroutine2
extern Var3, Subroutine3
Subroutine1:
    ....
    call (Subroutine3)
    ....
    operations(Var3)
    ....
Subroutine2:
    ....
Var1 dd ...
Var2 db ...
```

```
; FILE2.ASM
extern Var1, Subroutine2
global Subroutine3, Var3
Subroutine3:
    ....
    call (Subroutine2)
    ....
    operations(Var1)
    ....
Subroutine1:
    ....
Var2 db ...
Var3 dd ...
```

Can reuse names as long as they are not global!

exam-like problem

Techniques and tools

Bitdefender

- Example of a multimodule program nasm + nasm

```
; MODULE MAIN.ASM
global SirFinal
extern Concatenare

import printf msvcrt.dll
import exit msvcrt.dll
extern printf, exit
global start

segment code use32 public code class='code'
    start:
        mov eax, Sir1
        mov ebx, Sir2
        call Concatenare
        push dword SirFinal
        call [printf]
        add esp, 1*4
        push dword 0
        call [exit]

segment data use32
    Sir1 db 'Buna ', 0
    Sir2 db 'dimineata!', 0
    SirFinal resb 1000  ; space for result
```

* why do you clear the stack? do you really have to?

* calling function conventions

```
; MODULE SUB.ASM
extern SirFinal
global Concatenare

segment code use32 public code class='code'
    ; eax = address of the first array,
    ; ebx = address of the second array
    Concatenare:
        mov edi, SirFinal       ; destination = SirFinal
        mov esi, eax            ; source = first array
    .sir1Loop:
        lodsb                   ; next byte
        test al, al             ; array terminator (=0)?
        jz .sir2                ; if yes, go to second array
        stosb                   ; (otherwise) copy in destination
        jmp .sir1Loop           ; continue to nul
    .sir2:
        mov esi, ebx            ; source = second array
    .sir2Loop:
        lodsb                   ; same process for the new array
        test al, al
        jz .gata
        stosb
        jmp .sir2Loop
    .gata:
        stosb                   ; add array terminator from al
        ret
```

* data sir1 and sir2 are shared through registers

look into (test al, al)

Techniques and tools

Bitdefender

- Example of a multimodule program nasm + nasm
  - Necessary steps to build the final executable program
    - Assemble file main.asm
      - *nasm.exe -fobj **main.asm***
    - Assemble file sub.asm
      - *nasm.exe -fobj **sub.asm***
    - Edit links between the two modules
      - *alink.exe main.obj sub.obj –oPE –entry:start –subsys:console*
  - Notice: the two modules can be assembled in any order! Only at linkediting is necessary that the referred symbols to have all implementations available in one of the object files offered by the linkeditor
  - Linkediting, of course, is possible only after assembly/compilation!

Techniques and tools

Bitdefender

- Static linkage at **linkediting**: nasm + high level languages
  - Requirements of the linkeditor
    - global directive to allow access to other languages to our labels
    - extern directive to allow access in NASM of resources implemented in other languages
    - Declaration of variables and subrutines written in NASM in high level languages
      - *Example C: extern declarator!*
  - Entering the procedure
  - Keeping register values unaltered
  - Transmission and accessing parameters
  - Space allocation for local data (optional)
  - Returning a result (optional)
- The last aspects are discussed in detail in the call convention section!
  - See interface with high level languages: call conventions

Techniques and tools

Bitdefender

- Example of a multimodule program asm + C

```
//
// AFISARE.C
//

// requests the C preprocessor to include file stdio.h
// stdio.h declares the header (return type and parameters)of the C function printf
#include <stdio.h>

// we declare the function from the asm file so that the C compiler knows the type of the parameters and the return var
// linkeditor will handle the function implementation, the compiler needs to know only the header
void asm_start(void); equivalent with extern void asm_start(void) ! Any function declared at the most exterior level
                      // of a C module belongs to the extern memory class

// print function called by asm code
void afisare(int *vector, int numar_elemente) //any function defined at the most exterior level of a C module
{                                             //is implicitly "global" - meaning it is automatically exported
{
    int index;
    for (index = 0; index < numar_elemente; index++)
    {
        printf("%d", vector[index]);
    }
    printf("\n");
}

// main program, calls asm_start function written in assembly
void main(void)    // here starts the execution of the final program
{
    asm_start();  // we call the function from the asm file
}
```

extern is implicit

passed by reference

passed by value

Afisare.C

– Declares asm_start

– Defines afisare

main() – initiating the execution of the .exe file

Techniques and tools

Bitdefender

- Example of a multimodule program asm + C
  - Why _ ?
  - Build executable:
    1. Compile/assembly :
       - *afisare.c can be compiled with any C compiler (as desired) -> afisare.obj*
         - Visual C: cl /c afisare.c
       - *nasm.exe vector.asm* ***–fwin32*** *–o vector.obj*
    2. Link-editing :
       - *Call any linkeditor compatible with C, using:*
         - Input: afișare.obj and vector.obj
         - Output: console application
       - *link vector.obj afisare.obj /OUT:afisare.exe /MACHINE:X86 /SUBSYSTEM:CONSOLE*
  - Alternativelly, files can be bundled into one Visual Studio 'solution', configuring the IDE to:
    1. Assemble the asm file: specifying for example as **Pre-Build Event** the above assembly command (nasm.exe vector.asm –fwin32 –o vector.obj)
    2. Include afisare.obj as additional input to linkediting
    3. There are Visual Studio extensions that solve this automatically and transparently!

Techniques and tools

Bitdefender

- Example of a multimodule program asm + C

Vector.nasm

— Defines asm_start → will use afisare:

— _asm_start prepares the afisare function execution

```
;
; VECTOR.NASM
;

; inform the assembler on the existence of the printing function
extern _afisare            ; add _ as prefix to names from C!

; inform the assembler that we want asm_start to be available to other compiling units
global _asm_start          ; add _ as prefix to names refered by C!


; asm code is available in a public segment, and can be shared with an other extern code
segment code public code use32

_asm_start:
    push dword elemente ; parameter by value (write in stack value 5)
    push dword vector   ; vector given by reference (write in stack its address)
    call _afisare       ; call C function, again with prefix _
    add esp, 4*2        ; afisare is a C function (cdecl) -> we need to free arguments!
    ret                 ; back to C code that called this code

; the linkeditor can use the public data segment even for outside data
segment data public data use32
    vector dd 1, 2, 3, 4, 5          ; the vector we will print using the C routine
    elemente equ ($ - vector) / 4    ; constant equal to 5 (number of elements from the vector)
```

equ ⇒ pushed by value

address ⇒ by reference

Multi-module programming
Subroutine call implementation

extern
global } → exchange data (for importing and exporting)

* a multi-module program needs ways and methods to communicate
  ↳ involves seperate compilation → obtainig dif. .obj files

* registers are shared
* stack is also shared

| shared fizical resources

import / export
registers
stack
} 3 ways in which you can exchange data in ASM+ASM

* difference

CDECL ⇒ variable num. of params the caller is responsible for clearing the stack

SDT ⇒ fixed number of params & the callee is responsible for clearing the stack

ex: call [printf]
add esp, 4*1

we call a C function that takes as many params as we want, therefore it is our job to clear the stack

* Linkers check if everything has only one definition (decl. + alloc.)

any label = the starting address of that subroutine

* a name of a variable is associated with a starting address

call = jmp + returning address

call by value ⇒ maxes a copy

call by reference ⇒ works with the original

Calling conventions

Cdecl (C)
- variable number of params
- cleaning up the stack is the responsability of the CALLER

* params are put on the stack in reverse because when looking from the top, you only see the last element, the one on top

Volatile resources

- EAX
- ECX ⟶ loops ⟶ store result EAX / EDX:EAX / ST0 (FPU)
- EDX
- EFlags

STDCALL (windows)
- fixed num. of param.
- cleaning up the stack is the responsability of CALLEE

* if we write asm, this is your responsability

Subroutine call:
1. Call code
2. Entry code
3. Exit code

1. Call code (the caller's responsability)
   - a) Saving volatile resources (EAX, ECX, EDX, eFlags)
   - *) Assure the copliance : alligned ESP (32 bits), DF = 0 ...)
   - b) Passing parameters : push
   - c) Saving the returning address & performing the call

* any function returns something, at least an error code
↳ void doesn't mean it doesn't return anything, it means that I don't care what is returned

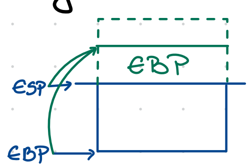
exam : who's the responsability of who?

2. Entry code (the callee's responsability)

* h: 15 the 3 stack registers
↳ define the beginning and the end of the current stack frame

a) building a stackframe for the called procedure

```
push EBP
mov EBP, ESP
```
} constructed a new empty stackframe



b) allocating space for local variables

sub esp, nr_bytes

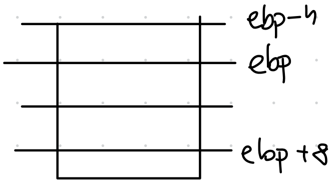
se mărește stack-ul

c) saving the non-volatile resources exposed to be modified

will be modified, but they are not volatile

* study the code on the last 10 slides

```
mov [ebp - 4], eax    ; use ebp 'cause esp is not reliable, it always changes
mov [ebp + 8], edx
```



stack rep.

* why do we have 3 instructions above call factorial?

```
push eax
dec eax
push eax
call factorial
```

$\boxed{\text{push eax}}$ param.

the first call will be at the first call of the c module

but after

the call code for the new recursive call we write

3. Exit code → slides