User Application Interaction With an OS

Dealing With C Programs in Linux

Adrian Coleșa and Ciprian Oprișa

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

March 11th, 2020

The purpose of today's lecture

- Review few basic aspects regarding writing and running C programs
  - their relationship with the OS

The purpose of today's lecture

- Review few basic aspects regarding writing and running C programs
  - their relationship with the OS

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Outline

Outline

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Compiler Role

- **CPU** does not understand "C language", not even "assembly"
  - **understands machine instructions** (its own language), i.e. bytes encoding certain actions
- **a program** (i.e. user application) that could be run should be
  - **a sequence of bytes** organized as **machine instructions**
  - machine instructions map 1:1 to assembly instructions

```
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
```

- the **app developer** "knows" a **higher-level language** (e.g. C)
- **compiler "translates" the source code** into machine instructions
  - ⇒ binary executable

Compiler Role

- **CPU** does not understand "C language", not even "assembly"
  - **understands machine instructions** (its own language), i.e. bytes encoding certain actions
- **a program** (i.e. user application) that could be run should be
  - **a sequence of bytes** organized as **machine instructions**
  - machine instructions map 1:1 to assembly instructions

```
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
```

- the **app developer** "knows" a **higher-level language** (e.g. C)
- **compiler "translates" the source code** into machine instructions
  - ⇒ binary executable

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Compiler Role

- **CPU** does not understand “C language”, not even “assembly”
  - **understands machine instructions** (its own language), i.e. bytes encoding certain actions
- **a program** (i.e. user application) that could be run should be
  - **a sequence of bytes** organized as **machine instructions**
  - machine instructions map 1:1 to assembly instructions

```
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
```

- the **app developer** “knows” a **higher-level language** (e.g. C)
- **compiler “translates” the source code** into machine instructions
  - ⇒ binary executable

Compiler Role

- **CPU** does not understand "C language", not even "assembly"
  - **understands machine instructions** (its own language), i.e. bytes encoding certain actions
- **a program** (i.e. user application) that could be run should be
  - **a sequence of bytes** organized as **machine instructions**
  - machine instructions map 1:1 to assembly instructions

```
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
```

- the **app developer** "knows" a **higher-level language** (e.g. C)
- **compiler "translates" the source code** into machine instructions
  - ⇒ binary executable

Compiler Role

- **CPU** does not understand "C language", not even "assembly"
  - **understands machine instructions** (its own language), i.e. bytes encoding certain actions
- **a program** (i.e. user application) that could be run should be
  - **a sequence of bytes** organized as **machine instructions**
  - machine instructions map 1:1 to assembly instructions

```
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
```

- the **app developer** "knows" a **higher-level language** (e.g. C)
- **compiler "translates" the source code** into machine instructions
  - ⇒ binary executable

Calling the Compiler

- **gcc (Linux, Windows, Mac)**
  `gcc [opt] <source_name> -o <exec_name>`
- Visual C (Windows)
  `cl.exe [opt] <source_name> /link /OUT:<exec_name>`
- from an Integrated Development Environment (IDE)
  - interact with an interface, e.g. just click some button
  - $\Rightarrow$ calls transparently commands like that above
- example
  - `gcc -Wall -Werror hellow.c -o hellow`
    - `-Wall` option means *warnings=all*, i.e. displays all warnings
    - `-Werror` option means *warnings=errors*, i.e. report warnings as errors

Calling the Compiler

- **gcc (Linux, Windows, Mac)**
  `gcc [opt] <source_name> -o <exec_name>`
- Visual C (Windows)
  `cl.exe [opt] <source_name> /link /OUT:<exec_name>`
- from an Integrated Development Environment (IDE)
  - interact with an interface, e.g. just click some button
  - $\Rightarrow$ calls transparently commands like that above
- example
  - `gcc -Wall -Werror hellow.c -o hellow`
    - `-Wall` option means warnings=all, i.e. display all warnings
    - `-Werror` option means warnings=errors, i.e. report warnings as errors

Calling the Compiler

- **gcc (Linux, Windows, Mac)**

  ```
  gcc [opt] <source_name> -o <exec_name>
  ```

- Visual C (Windows)

  ```
  cl.exe [opt] <source_name> /link /OUT:<exec_name>
  ```

- from an Integrated Development Environment (IDE)
  - interact with an interface, e.g. just click some button
  - ⇒ calls transparently commands like that above
- example
  - `gcc -Wall -Werror hellow.c -o hellow`

Calling the Compiler

- **gcc (Linux, Windows, Mac)**
  `gcc [opt] <source_name> -o <exec_name>`
- Visual C (Windows)
  `cl.exe [opt] <source_name> /link /OUT:<exec_name>`
- from an Integrated Development Environment (IDE)
  - interact with an interface, e.g. just click some button
  - ⇒ calls transparently commands like that above
- example
  - `gcc -Wall -Werror hellow.c -o hellow`

Calling the Compiler

- **gcc (Linux, Windows, Mac)**
  `gcc [opt] <source_name> -o <exec_name>`
- Visual C (Windows)
  `cl.exe [opt] <source_name> /link /OUT:<exec_name>`
- from an Integrated Development Environment (IDE)
  - interact with an interface, e.g. just click some button
  - ⇒ calls transparently commands like that above
- example
  - `gcc -Wall -Werror hellow.c -o hellow`
    - -Wall option means warnings=all, i.e. display all warnings
    - -Werror option means warnings=errors, i.e. report warnings as errors

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Calling the Compiler

- **gcc (Linux, Windows, Mac)**
  ```
  gcc [opt] <source_name> -o <exec_name>
  ```
- Visual C (Windows)
  ```
  cl.exe [opt] <source_name> /link /OUT:<exec_name>
  ```
- from an Integrated Development Environment (IDE)
  - interact with an interface, e.g. just click some button
  - ⇒ calls transparently commands like that above
- example
  - `gcc -Wall -Werror hellow.c -o hellow`
    - `-Wall` option means *warnings=all*, i.e. displays all warnings
    - `-Werror` option means *warnings=errors*, i.e. report warnings as errors

Calling the Compiler

- **gcc (Linux, Windows, Mac)**
  `gcc [opt] <source_name> -o <exec_name>`
- Visual C (Windows)
  `cl.exe [opt] <source_name> /link /OUT:<exec_name>`
- from an Integrated Development Environment (IDE)
  - interact with an interface, e.g. just click some button
  - ⇒ calls transparently commands like that above
- example
  - `gcc -Wall -Werror hellow.c -o hellow`
    - `-Wall` option means *warnings=all*, i.e. displays all warnings
    - `-Werror` option means *warnings=errors*, i.e. report warnings as errors

Calling the Compiler

- **gcc (Linux, Windows, Mac)**
  `gcc [opt] <source_name> -o <exec_name>`
- Visual C (Windows)
  `cl.exe [opt] <source_name> /link /OUT:<exec_name>`
- from an Integrated Development Environment (IDE)
  - interact with an interface, e.g. just click some button
  - ⇒ calls transparently commands like that above
- example
  - `gcc -Wall -Werror hellow.c -o hellow`
    - `-Wall` option means *warnings=all*, i.e. displays all warnings
    - `-Werror` option means *warnings=errors*, i.e. report warnings as errors

Calling the Compiler

- **gcc (Linux, Windows, Mac)**
  `gcc [opt] <source_name> -o <exec_name>`
- Visual C (Windows)
  `cl.exe [opt] <source_name> /link /OUT:<exec_name>`
- from an Integrated Development Environment (IDE)
  - interact with an interface, e.g. just click some button
  - ⇒ calls transparently commands like that above
- example
  - `gcc -Wall -Werror hellow.c -o hellow`
    - `-Wall` option means *warnings=all*, i.e. displays all warnings
    - `-Werror` option means *warnings=errors*, i.e. report warnings as errors

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Never ignore compiler's warnings!

Multiple Source File Applications

- source (text) files of a C-based application could be
  - `.c` **files**: code, **implementation (definition)**
  - `.h` **files**: type, function, constant **declaration**[1]
    - to be included (as text!) in other files
      - take care to not include the same .h file multiple times[2]!
      - take care to cyclic inclusions!
    - usually **not for** variable and function **definition**!
- compilation process consists in more phases
  1. **pre-compilation**: expand / replace the preprocessor directives like `#include`, `#define`, ...
  2. **compilation**: compile each `.c` file ⇒ **object file** `.o`
  3. **linking**: object files linked together into a single executable
     - solve references to functions in other object files
     - solve references to functions in dynamic / shared libraries (.so files) or static libraries (.a files)

[1] Note the difference between definition and declaration!

[2] See gcc and Visual Studio recommendations

Multiple Source File Applications

- source (text) files of a C-based application could be
  - `.c` **files**: code, **implementation (definition)**
  - `.h` **files**: type, function, constant **declaration**[1]
    - to be included (as text!) in other files
      - take care to not include the same .h file multiple times[2]
      - take care to cyclic inclusions!
    - usually **not for** variable and function **definition**!
- compilation process consists in more phases
  1. **pre-compilation**: expand / replace the preprocessor directives like `#include`, `#define`, ...
  2. **compilation**: compile each `.c` file ⇒ **object file** `.o`
  3. **linking**: object files linked together into a single executable
     - solve references to functions in other object files
     - solve references to functions in dynamic / shared libraries [illegible] or static libraries (.a files)

[1] Note the difference between definition and declaration!

[2] See gcc and Visual Studio recommendations

Multiple Source File Applications

- source (text) files of a C-based application could be
  - `.c` **files**: code, **implementation (definition)**
  - `.h` **files**: type, function, constant **declaration**[1]
    - to be included (as text!) in other files
      - take care to not include the same .h file multiple times[2]!
      - take care to cyclic inclusions!
    - usually **not for** variable and function **definition**!
- compilation process consists in more phases
  1. **pre-compilation**: expand / replace the preprocessor directives like `#include`, `#define`, ...
  2. **compilation**: compile each `.c` file ⇒ **object file** `.o`
  3. **linking**: object files linked together into a single executable
     - solve references to functions in other object files
     - solve references to functions in dynamic / shared libraries (.so files) or static libraries (.a files)

[1] Note the difference between definition and declaration!

[2] See gcc and Visual Studio recommendations

Multiple Source File Applications

- source (text) files of a C-based application could be
  - `.c` **files**: code, **implementation (definition)**
  - `.h` **files**: type, function, constant **declaration**[1]
    - to be included (as text!) in other files
      - take care to not include the same .h file multiple times[2]!
      - take care to cyclic inclusions!
    - usually **not for** variable and function **definition**!
- compilation process consists in more phases
  1. **pre-compilation**: expand / replace the preprocessor directives like `#include`, `#define`, ...
  2. **compilation**: compile each `.c` file ⇒ **object file** `.o`
  3. **linking**: object files linked together into a single executable
     - solve references to functions in other object files
     - solve references to functions in dynamic / shared libraries ([illegible]) or static libraries (.a files)

[1] Note the difference between definition and declaration!

[2] See gcc and Visual Studio recommendations

Multiple Source File Applications

- source (text) files of a C-based application could be
  - `.c` **files**: code, **implementation (definition)**
  - `.h` **files**: type, function, constant **declaration**[1]
    - to be included (as text!) in other files
      - take care to not include the same .h file multiple times[2]!
      - take care to cyclic inclusions!
    - usually **not for** variable and function **definition**!
- compilation process consists in more phases
  1. **pre-compilation**: expand / replace the preprocessor directives like `#include`, `#define`, ...
  2. **compilation**: compile each `.c` file ⇒ **object file** `.o`
  3. **linking**: object files linked together into a single executable

[1] Note the difference between definition and declaration!

[2] See gcc and Visual Studio recommendations

Multiple Source File Applications

- source (text) files of a C-based application could be
  - `.c` **files**: code, **implementation (definition)**
  - `.h` **files**: type, function, constant **declaration**[1]
    - to be included (as text!) in other files
      - take care to not include the same .h file multiple times[2]!
      - take care to cyclic inclusions!
    - usually **not for** variable and function **definition**!
- compilation process consists in more phases
  1. **pre-compilation**: expand / replace the preprocessor directives like `#include`, `#define`, ...
  2. **compilation**: compile each `.c` file ⇒ **object file** `.o`
  3. **linking**: object files linked together into a single executable
     - solve references to functions in other object files
     - solve references to functions in dynamic / shared libraries ([illegible]) or static libraries (.a files)

[1] Note the difference between definition and declaration!

[2] See gcc and Visual Studio recommendations

Multiple Source File Applications

- source (text) files of a C-based application could be
  - `.c` **files**: code, **implementation (definition)**
  - `.h` **files**: type, function, constant **declaration**[1]
    - to be included (as text!) in other files
      - take care to not include the same .h file multiple times[2]!
      - take care to cyclic inclusions!
    - usually **not for** variable and function **definition**!
- compilation process consists in more phases
  1. **pre-compilation**: expand / replace the preprocessor directives like `#include`, `#define`, ...
  2. **compilation**: compile each `.c` file ⇒ **object file** `.o`
  3. **linking**: object files linked together into a single executable
     - solve references to functions in other object files
     - solve references to functions in dynamic / shared libraries (.so files) or static libraries (.a files)

[1] Note the difference between definition and declaration!

[2] See gcc and Visual Studio recommendations

Multiple Source File Applications

- source (text) files of a C-based application could be
  - `.c` **files**: code, **implementation (definition)**
  - `.h` **files**: type, function, constant **declaration**[1]
    - to be included (as text!) in other files
      - take care to not include the same .h file multiple times[2]!
      - take care to cyclic inclusions!
    - usually **not for** variable and function **definition**!
- compilation process consists in more phases
  1. **pre-compilation**: expand / replace the preprocessor directives like `#include`, `#define`, ...
  2. **compilation**: compile each `.c` file ⇒ **object file** `.o`
  3. **linking**: object files linked together into a single executable
     - solve references to functions in other object files
     - solve references to functions in dynamic / **shared libraries** (`.so` files) or **static libraries** (`.a` files)

[1] Note the difference between definition and declaration!

[2] See gcc and Visual Studio recommendations

Multiple Source File Applications

- source (text) files of a C-based application could be
  - `.c` **files**: code, **implementation (definition)**
  - `.h` **files**: type, function, constant **declaration**[1]
    - to be included (as text!) in other files
      - take care to not include the same .h file multiple times[2]!
      - take care to cyclic inclusions!
    - usually **not for** variable and function **definition**!
- compilation process consists in more phases
  1. **pre-compilation**: expand / replace the preprocessor directives like `#include`, `#define`, ...
  2. **compilation**: compile each `.c` file ⇒ **object file** `.o`
  3. **linking**: object files linked together into a single executable
     - solve references to functions in other object files
     - solve references to functions in dynamic / **shared libraries** (`.so` files) or **static libraries** (`.a` files)

[1] Note the difference between definition and declaration!

[2] See gcc and Visual Studio recommendations

Multiple Source File Applications

- source (text) files of a C-based application could be
  - `.c` **files**: code, **implementation (definition)**
  - `.h` **files**: type, function, constant **declaration**[1]
    - to be included (as text!) in other files
      - take care to not include the same .h file multiple times[2]!
      - take care to cyclic inclusions!
    - usually **not for** variable and function **definition**!
- compilation process consists in more phases
  1. **pre-compilation**: expand / replace the preprocessor directives like `#include`, `#define`, ...
  2. **compilation**: compile each `.c` file ⇒ **object file** `.o`
  3. **linking**: object files linked together into a single executable
     - solve references to functions in other object files
     - solve references to functions in dynamic / **shared libraries** (`.so` files) or **static libraries** (`.a` files)

[1] Note the difference between definition and declaration!

[2] See gcc and Visual Studio recommendations

Multiple Source File Applications

- source (text) files of a C-based application could be
  - `.c` **files**: code, **implementation (definition)**
  - `.h` **files**: type, function, constant **declaration**[1]
    - to be included (as text!) in other files
      - take care to not include the same .h file multiple times[2]!
      - take care to cyclic inclusions!
    - usually **not for** variable and function **definition**!
- compilation process consists in more phases
  1. **pre-compilation**: expand / replace the preprocessor directives like `#include`, `#define`, ...
  2. **compilation**: compile each `.c` file ⇒ **object file** `.o`
  3. **linking**: object files linked together into a single executable
     - solve references to functions in other object files
     - solve references to functions in dynamic / **shared libraries** (`.so` files) or **static libraries** (`.a` files)

[1] Note the difference between definition and declaration!

[2] See gcc and Visual Studio recommendations

Multiple Source File Applications

- source (text) files of a C-based application could be
  - `.c` **files**: code, **implementation (definition)**
  - `.h` **files**: type, function, constant **declaration**[1]
    - to be included (as text!) in other files
      - take care to not include the same .h file multiple times[2]!
      - take care to cyclic inclusions!
    - usually **not for** variable and function **definition**!
- compilation process consists in more phases
  1. **pre-compilation**: expand / replace the preprocessor directives like `#include`, `#define`, ...
  2. **compilation**: compile each `.c` file ⇒ **object file** `.o`
  3. **linking**: object files linked together into a single executable
     - solve references to functions in other object files
     - solve references to functions in dynamic / **shared libraries** (`.so` files) or **static libraries** (`.a` files)

[1]Note the difference between definition and declaration!

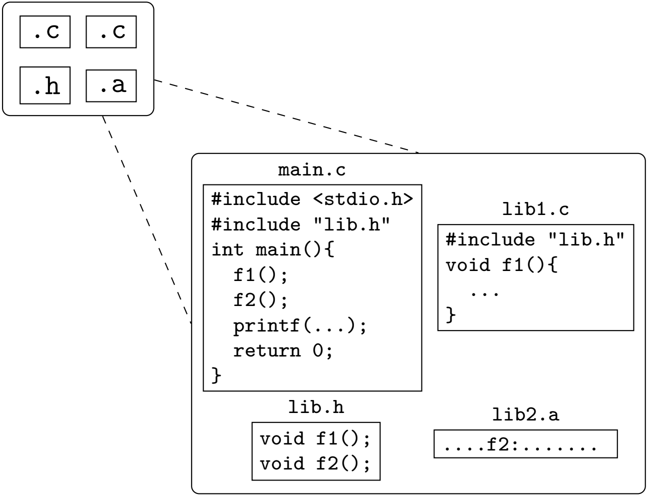
[2]See gcc and Visual Studio recommendations

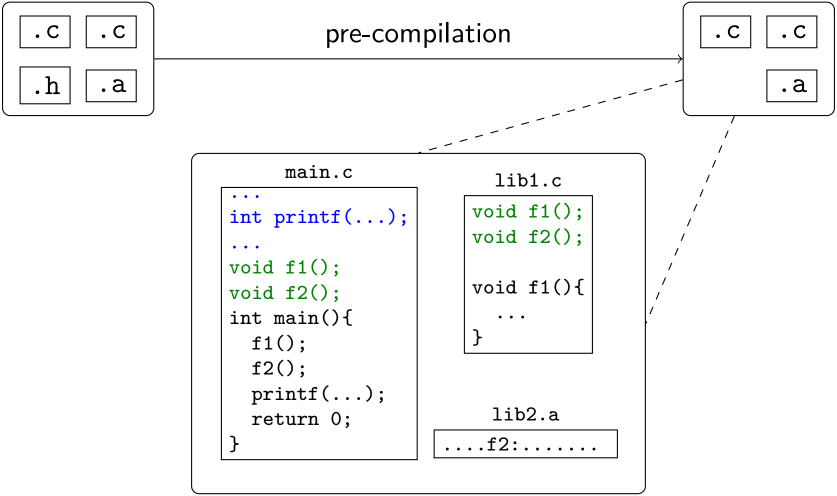
Multiple Source File Applications

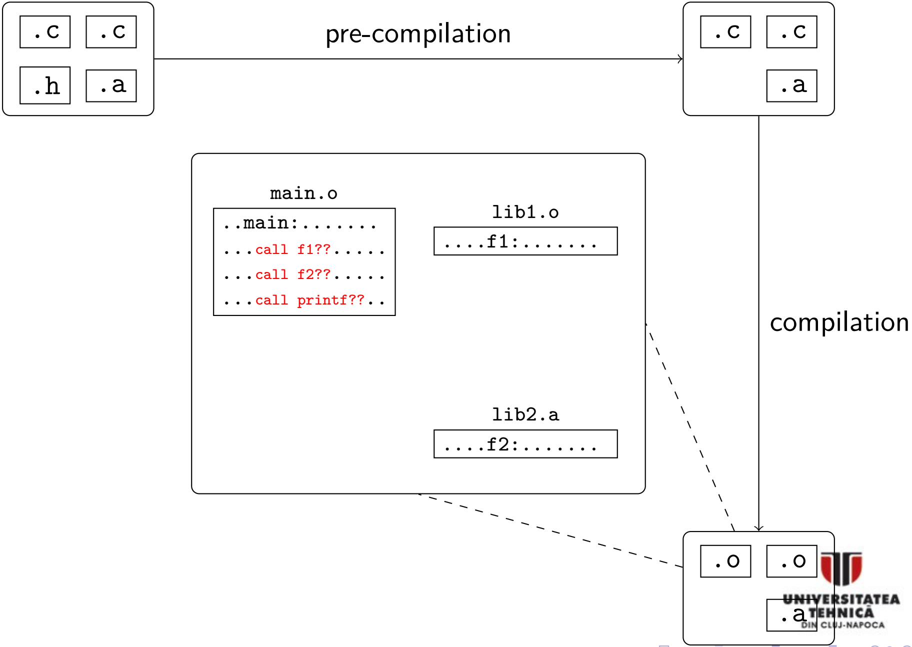
- source (text) files of a C-based application could be
  - `.c` **files**: code, **implementation (definition)**
  - `.h` **files**: type, function, constant **declaration**[1]
    - to be included (as text!) in other files
      - take care to not include the same .h file multiple times[2]!
      - take care to cyclic inclusions!
    - usually **not for** variable and function **definition**!
- compilation process consists in more phases
  1. **pre-compilation**: expand / replace the preprocessor directives like `#include`, `#define`, ...
  2. **compilation**: compile each `.c` file ⇒ **object file** `.o`
  3. **linking**: object files linked together into a single executable
     - solve references to functions in other object files
     - solve references to functions in dynamic / **shared libraries** (`.so` files) or **static libraries** (`.a` files)

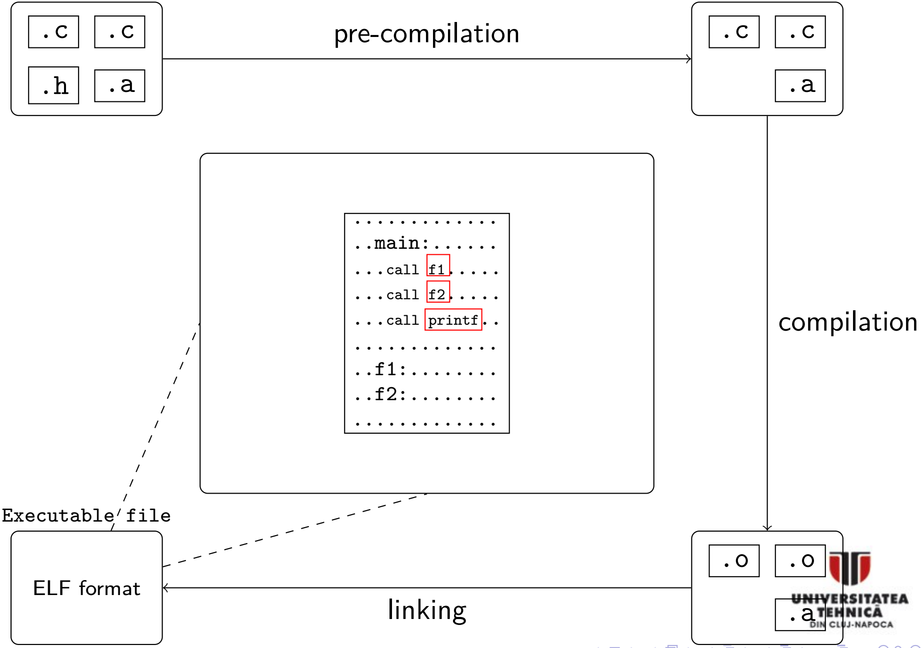
[1] Note the difference between definition and declaration!

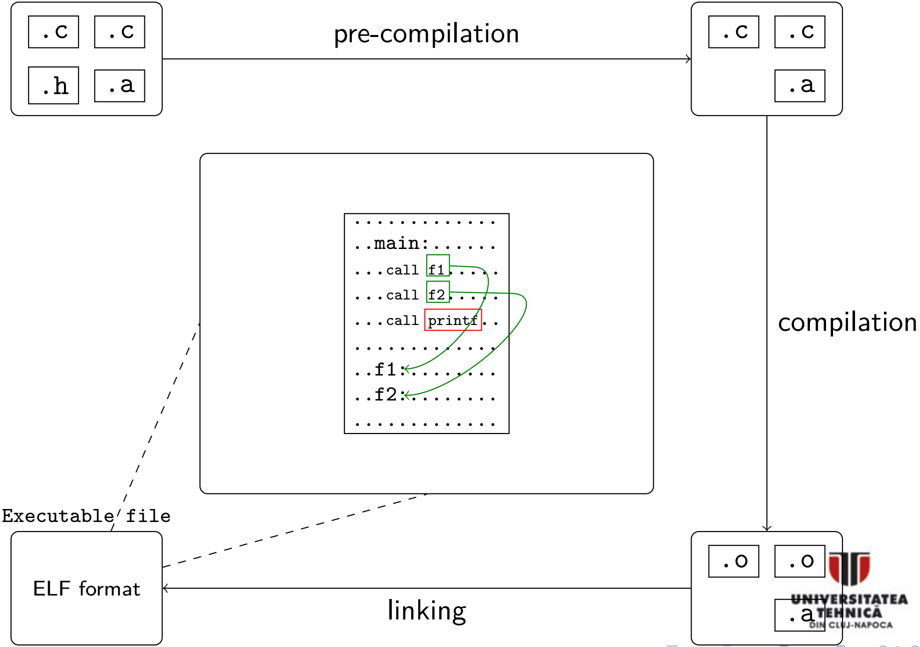
[2] See gcc and Visual Studio recommendations

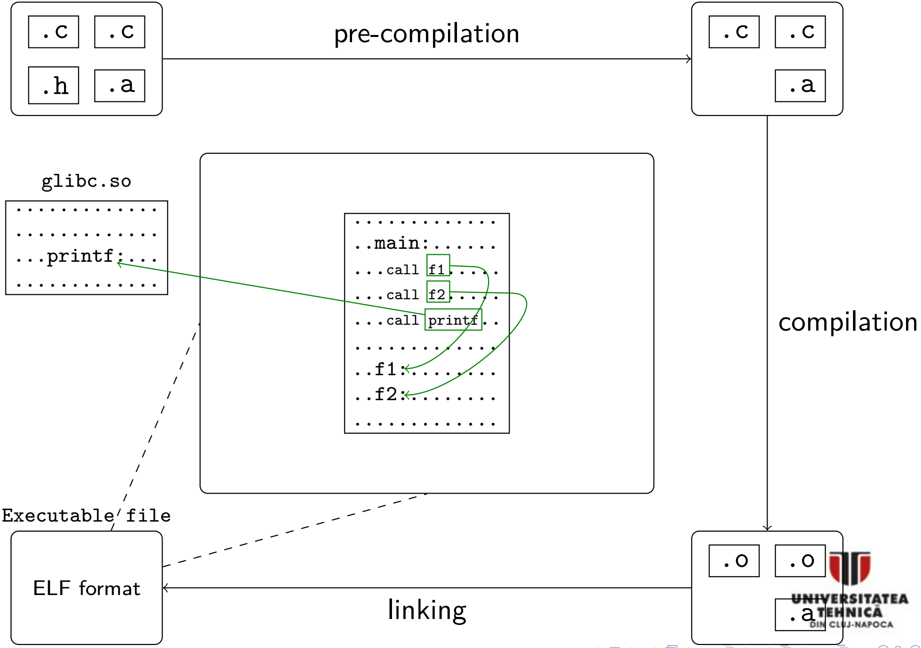

.c
.c
.h
.a
main.c
#include <stdio.h>
#include "lib.h"
int main(){
f1();
f2();
printf(...);
return 0;
}
lib1.c
#include "lib.h"
void f1(){
...
}
lib.h
void f1();
void f2();
lib2.a
....f2:.......


.c
.c
.h
.a
pre-compilation
.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
pre-compilation
.c
.c
.a
compilation
main.o
..main:.......
...call f1??.....
...call f2??.....
...call printf??..
lib1.o
....f1:.......
lib2.a
....f2:.......
.o
.o
.a


.c
.c
.h
.a
pre-compilation
.c
.c
.a
compilation
..main:......
...call f1.....
...call f2.....
...call printf..
..f1:........
..f2:........
Executable file
ELF format
linking
.o
.o
.a


.c
.c
.h
.a
pre-compilation
.c
.c
.a
compilation
..main:......
...call f1.....
...call f2.....
...call printf..
..f1:........
..f2:........
Executable file
ELF format
linking
.o
.o
.a


.c
.c
.h
.a
pre-compilation
.c
.c
.a
compilation
.o
.o
.a
linking
ELF format
Executable file
glibc.so
...printf:...
..main:......
...call f1.....
...call f2.....
...call printf..
..f1:........
..f2:........

Outline

From Executable File to the User Application

- the OS **allocates memory** for the new application
- the OS **loads** code and data from the **executable file** (ELF) into the allocated memory
  - ELF specification says what to load from the executable file
  - ELF specification says how many memory is needed
  - ELF specification says where to load into memory
- ⇒ **application's (virtual) address space**
  - complying a specific structure
  - different areas (segments): code, data, heap, stack etc.
  - there are also invalid areas (holes)
- **configure the CPU registers** with values corresponding to the new application's memory
- ⇒ CPU starts running the new application

From Executable File to the User Application

- the OS **allocates memory** for the new application
- the OS **loads** code and data from the **executable file** (ELF) into the allocated memory
  - ELF specification says what to load from the executable file
  - ELF specification says how many memory is needed
  - ELF specification says where to load into memory
- ⇒ **application's (virtual) address space**
  - complying a specific structure
  - different areas (segments): code, data, heap, stack etc.
  - there are also invalid areas (holes)
- **configure the CPU registers** with values corresponding to the new application's memory
- ⇒ CPU starts running the new application

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

From Executable File to the User Application

- the OS **allocates memory** for the new application
- the OS **loads** code and data from the **executable file** (ELF) into the allocated memory
  - ELF specification says what to load from the executable file
  - ELF specification says how many memory is needed
  - ELF specification says where to load into memory
- ⇒ **application's (virtual) address space**
  - complying a specific structure
  - different areas (segments): code, data, heap, stack etc.
  - there are also invalid areas (holes)
- **configure the CPU registers** with values corresponding to the new application's memory
- ⇒ CPU starts running the new application

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

From Executable File to the User Application

- the OS **allocates memory** for the new application
- the OS **loads** code and data from the **executable file** (ELF) into the allocated memory
  - ELF specification says what to load from the executable file
  - ELF specification says how many memory is needed
  - ELF specification says where to load into memory
- ⇒ **application's (virtual) address space**
  - complying a specific structure
  - different areas (segments): code, data, heap, stack etc.
  - there are also invalid areas (holes)
- **configure the CPU registers** with values corresponding to the new application's memory
- ⇒ CPU starts running the new application

From Executable File to the User Application

- the OS **allocates memory** for the new application
- the OS **loads** code and data from the **executable file** (ELF) into the allocated memory
  - ELF specification says what to load from the executable file
  - ELF specification says how many memory is needed
  - ELF specification says where to load into memory
- ⇒ **application's (virtual) address space**
  - complying a specific structure
  - different areas (segments): code, data, heap, stack etc.
  - there are also invalid areas (holes)
- **configure the CPU registers** with values corresponding to the new application's memory
- ⇒ CPU starts running the new application

From Executable File to the User Application

- the OS **allocates memory** for the new application
- the OS **loads** code and data from the **executable file** (ELF) into the allocated memory
  - ELF specification says what to load from the executable file
  - ELF specification says how many memory is needed
  - ELF specification says where to load into memory
- $\Rightarrow$ **application's (virtual) address space**
  - complying a specific structure
  - different areas (segments): code, data, heap, stack etc.
  - there are also invalid areas (holes)
- **configure the CPU registers** with values corresponding to the new application's memory
- $\Rightarrow$ CPU starts running the new application

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

From Executable File to the User Application

- the OS **allocates memory** for the new application
- the OS **loads** code and data from the **executable file** (ELF) into the allocated memory
  - ELF specification says what to load from the executable file
  - ELF specification says how many memory is needed
  - ELF specification says where to load into memory
- $\Rightarrow$ **application's (virtual) address space**
  - complying a specific structure
  - different areas (segments): code, data, heap, stack etc.
  - there are also invalid areas (holes)
- **configure the CPU registers** with values corresponding to the new application's memory
- $\Rightarrow$ CPU starts running the new application

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

From Executable File to the User Application

- the OS **allocates memory** for the new application
- the OS **loads** code and data from the **executable file** (ELF) into the allocated memory
  - ELF specification says what to load from the executable file
  - ELF specification says how many memory is needed
  - ELF specification says where to load into memory
- $\Rightarrow$ **application's (virtual) address space**
  - complying a specific structure
  - different areas (segments): code, data, heap, stack etc.
  - there are also invalid areas (holes)
- **configure the CPU registers** with values corresponding to the new application's memory
- $\Rightarrow$ CPU starts running the new application

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

From Executable File to the User Application

- the OS **allocates memory** for the new application
- the OS **loads** code and data from the **executable file** (ELF) into the allocated memory
  - ELF specification says what to load from the executable file
  - ELF specification says how many memory is needed
  - ELF specification says where to load into memory
- ⇒ **application's (virtual) address space**
  - complying a specific structure
  - different areas (segments): code, data, heap, stack etc.
  - there are also invalid areas (holes)
- **configure the CPU registers** with values corresponding to the new application's memory
- ⇒ CPU starts running the new application

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

From Executable File to the User Application

- the OS **allocates memory** for the new application
- the OS **loads** code and data from the **executable file** (ELF) into the allocated memory
  - ELF specification says what to load from the executable file
  - ELF specification says how many memory is needed
  - ELF specification says where to load into memory
- $\Rightarrow$ **application's (virtual) address space**
  - complying a specific structure
  - different areas (segments): code, data, heap, stack etc.
  - there are also invalid areas (holes)
- **configure the CPU registers** with values corresponding to the new application's memory
- $\Rightarrow$ CPU starts running the new application

From Executable File to the User Application

- the OS **allocates memory** for the new application
- the OS **loads** code and data from the **executable file** (ELF) into the allocated memory
  - ELF specification says what to load from the executable file
  - ELF specification says how many memory is needed
  - ELF specification says where to load into memory
- $\Rightarrow$ **application's (virtual) address space**
  - complying a specific structure
  - different areas (segments): code, data, heap, stack etc.
  - there are also invalid areas (holes)
- **configure the CPU registers** with values corresponding to the new application's memory
- $\Rightarrow$ CPU starts running the new application

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Local variables and the stack (1)

test1.c:

```c
#include <stdio.h>

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

Local variables and the stack (1)

test1.c:

```c
#include <stdio.h>

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

Compiling:

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

Local variables and the stack (1)

test1.c:

```c
#include <stdio.h>

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

Compiling:

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

Running:

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

Local variables and the stack (1)

test1.c:

```
#include <stdio.h>

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

Compiling:

`gcc -Wall test1.c -o test1`

Running:

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

Compiling with different options:

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

- `-g`: add debugging info in the executable
- `-m32`: generate 32-bit code

Local variables and the stack (1)

<u>*test1.c*</u>:

```
#include <stdio.h>

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

Compiling:

`gcc -Wall test1.c -o test1`

Running:

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

Compiling with different options:

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

- `-g`: add debugging info in the executable
- `-m32`: generate 32-bit code

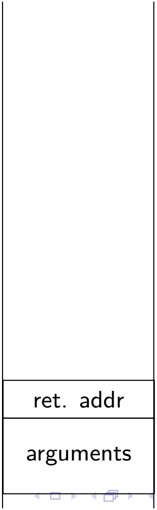
View executable as assembly code:

`objdump -D test1 -M intel -S`

- `-D`: disassembly
- `-M intel`: Intel syntax
- `-S`: display source code also

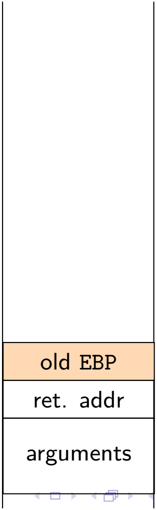
Local variables and the stack (2)

```
;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;
```



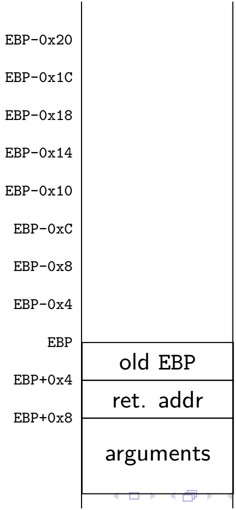
Local variables and the stack (2)

```
;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;
```



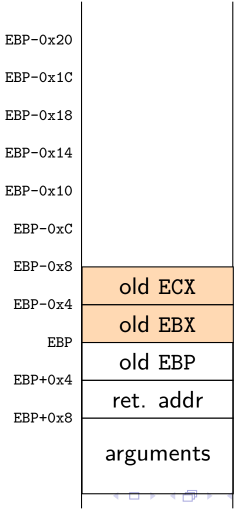
Local variables and the stack (2)

```
;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;
```



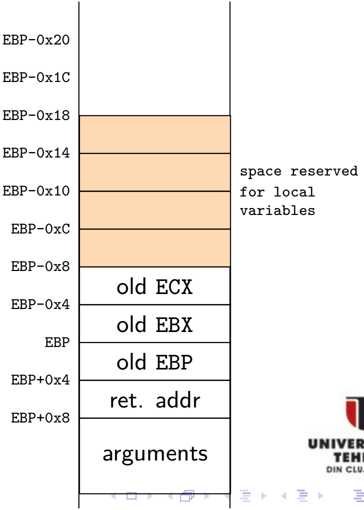
Local variables and the stack (2)

```
;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;
```



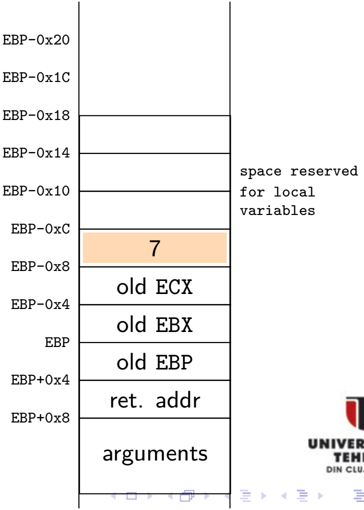
Local variables and the stack (2)

```
;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;
```



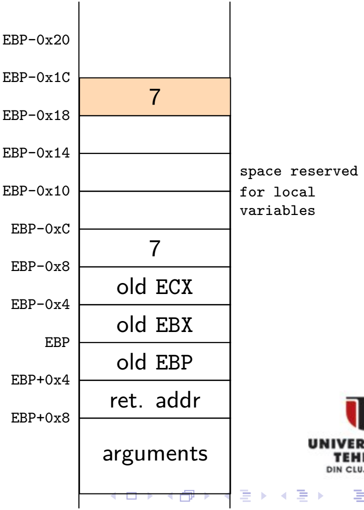
Local variables and the stack (2)

```
;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;
```



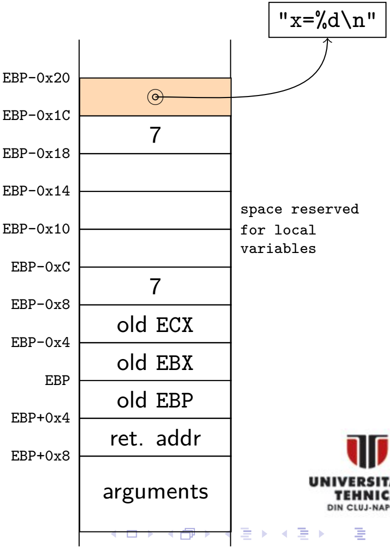
Local variables and the stack (2)

```
;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;
```



Local variables and the stack (2)

```
;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;
```



Array Manipulation (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;
}
```

- could be initialized when declared
- element indexing starts at 0
- when declared as local variables, they are allocated on the stack

Array Manipulation (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
```

- could be initialized when declared
- element indexing starts at 0
- when declared as local variables, they are allocated on the stack

Array Manipulation (1)

test2.c:

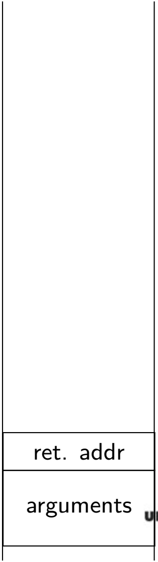
```
int main(void){
    int a[2];
    int b[] = {7, 8};
    a[0] = 3;
    a[1] = 4;
    printf("%d %d\n", a[0], a[1]);
    printf("%d %d\n", b[0], b[1]);
    return 0;
}
```

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

- could be initialized when declared
- element indexing starts at 0
- when declared as local variables, they are allocated on the stack

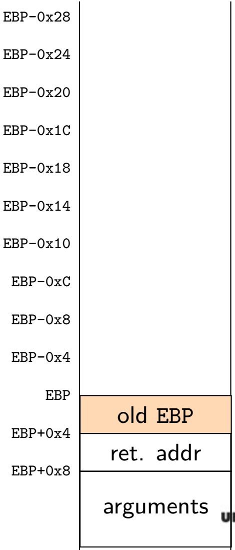
Array Manipulation (2)

```
...
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
...
```



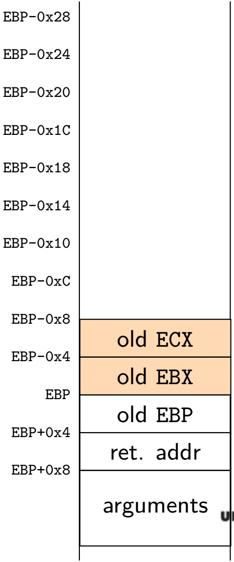
Array Manipulation (2)

```
...
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
...
```



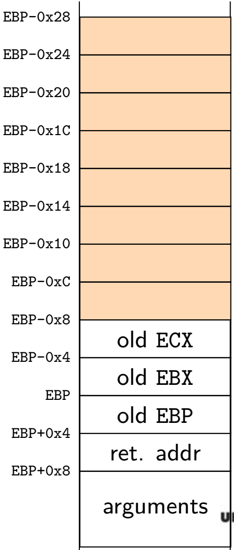
Array Manipulation (2)

```
...
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
...
```



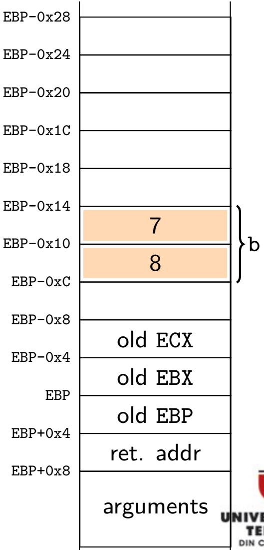
Array Manipulation (2)

```
...
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
...
```



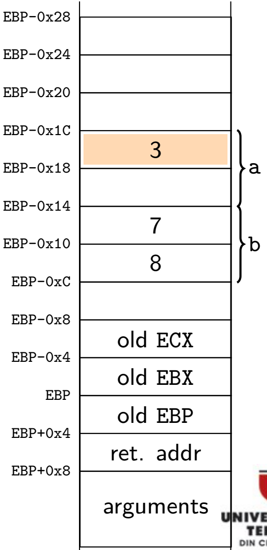
Array Manipulation (2)

```
...
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
...
```



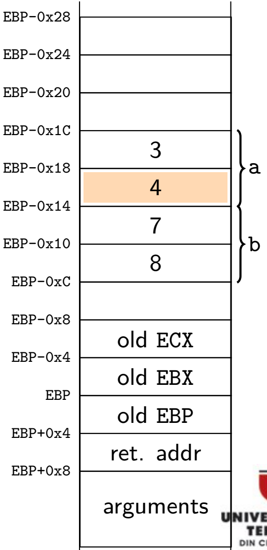
Array Manipulation (2)

```
...
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
...
```



Array Manipulation (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
...
```



Memory Corruption (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;
}
```

What happens when access an array out of its bounds?

- Java: an array is an object and all the operations on it are controlled, it knows its bounds ⇒ throws an exception
- C: the array is just a memory address, where the array starts
- C standard does not define behavior when array accessed out of bounds
    - anything could happen

Memory Corruption (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;
}
```

What happens when access an array out of its bounds?

- Java: an array is an object and all the operations on it are controlled, it knows its bounds ⇒ throws an exception
- C: the array is just a memory address, where the array starts
- C standard does not define behavior when array accessed out of bounds
  - anything could happen

Memory Corruption (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;
}
```

What happens when access an array out of its bounds?

- Java: an array is an object and all the operations on it are controlled, it knows its bounds $\Rightarrow$ throws an exception
- C: the array is just a memory address, where the array starts
- C standard does not define behavior when array accessed out of bounds
    - anything could happen

Memory Corruption (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;
}
```

What happens when access an array out of its bounds?

- Java: an array is an object and all the operations on it are controlled, it knows its bounds ⇒ throws an exception
- C: the array is just a memory address, where the array starts
- C standard does not define behavior when array accessed out of bounds
    - anything could happen

Memory Corruption (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
```

What happens when access an array out of its bounds?

- Java: an array is an object and all the operations on it are controlled, it knows its bounds ⇒ throws an exception
- C: the array is just a memory address, where the array starts
- C standard does not define behavior when array accessed out of bounds
  - anything could happen

Memory Corruption (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;
}
```

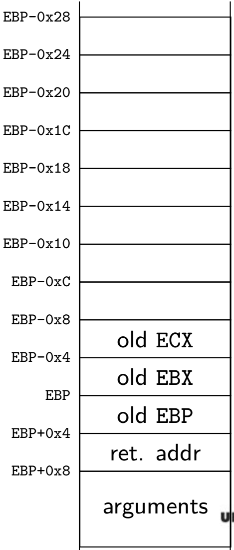
```
$ gcc -Wall test3.c -o test3
$ ./test3
3 4
5 8
```

What happens when access an array out of its bounds?

- Java: an array is an object and all the operations on it are controlled, it knows its bounds ⇒ throws an exception
- C: the array is just a memory address, where the array starts
- C standard does not define behavior when array accessed out of bounds
    - anything could happen

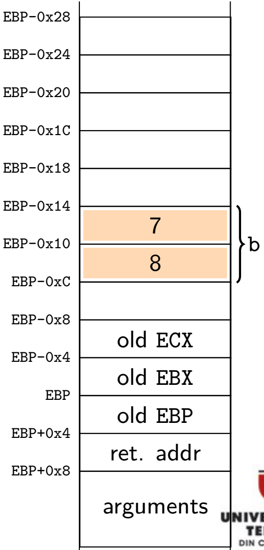
Memory Corruption (2)

```
...
;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
...
```



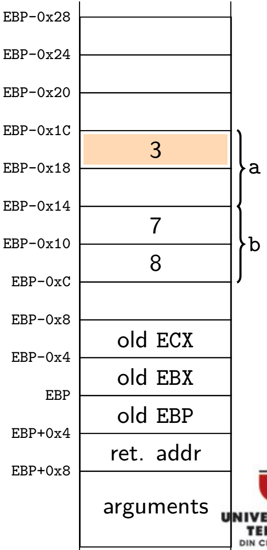
Memory Corruption (2)

```
...
;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
...
```



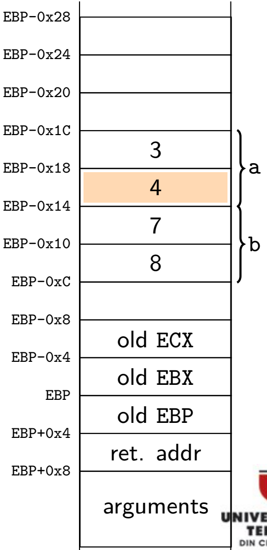
Memory Corruption (2)

```
...
;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
...
```



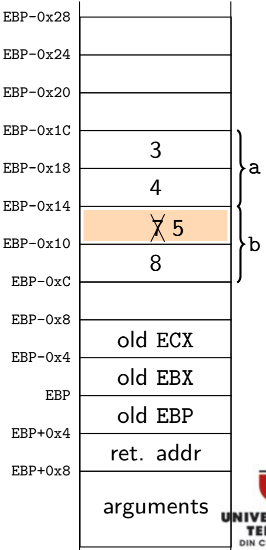
Memory Corruption (2)

```
...
;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
...
```



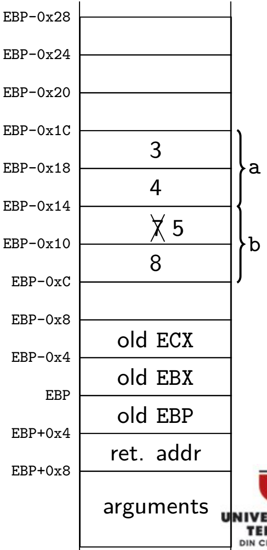
Memory Corruption (2)

```
...
;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
...
```



Memory Corruption (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
...
```



Question

What happens if change `a[8]`?

Pointers (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;
}
```

- a pointer is a variable that contains a memory address, e.g. the address of another variable
- memory addresses are just integers (on 32 or 64 bits)
- useful when need referring some data, not copying it (e.g. reference parameters of a function)
- & - reference (get the address of a variable)
- * - derenference (get memory contents from a memory address)

Pointers (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;
}
```

- a pointer is a variable that contains a memory address, e.g. the address of another variable
- memory addresses are just integers (on 32 or 64 bits)
- useful when need referring some data, not copying it (e.g. reference parameters of a function)
- & - reference (get the address of a variable)
- * - derenference (get memory contents from a memory address)

Pointers (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;
}
```

- a pointer is a variable that contains a memory address, e.g. the address of another variable
- memory addresses are just integers (on 32 or 64 bits)
- useful when need referring some data, not copying it (e.g. reference parameters of a function)
- & - reference (get the address of a variable)
- * - derenference (get memory contents from a memory address)

Pointers (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;
}
```

- a pointer is a variable that contains a memory address, e.g. the address of another variable
- memory addresses are just integers (on 32 or 64 bits)
- useful when need referring some data, not copying it (e.g. reference parameters of a function)
- & - reference (get the address of a variable)
- * - derenference (get memory contents from a memory address)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Pointers (1)

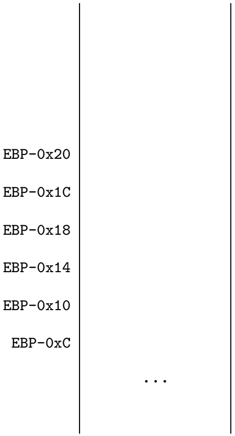
test_ptr.c:

```
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;
}
```

- a pointer is a variable that contains a memory address, e.g. the address of another variable
- memory addresses are just integers (on 32 or 64 bits)
- useful when need referring some data, not copying it (e.g. reference parameters of a function)
- & - reference (get the address of a variable)
- * - derenference (get memory contents from a memory address)

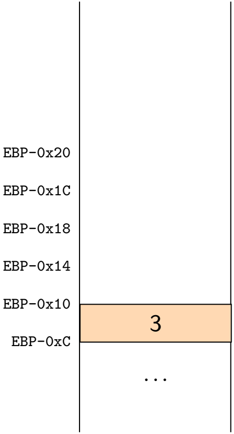
Pointers (2)

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



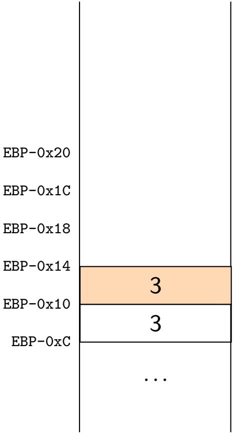
Pointers (2)

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



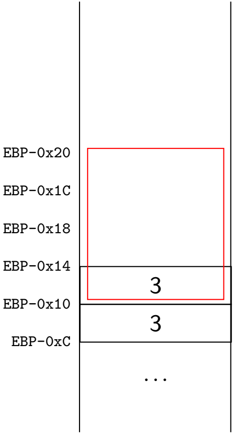
Pointers (2)

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



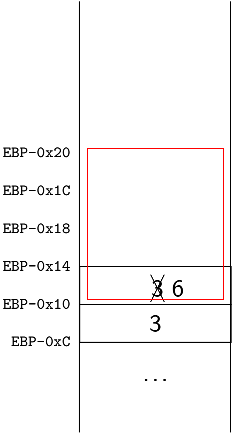
Pointers (2)

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



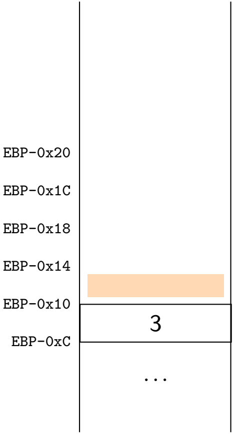
Pointers (2)

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



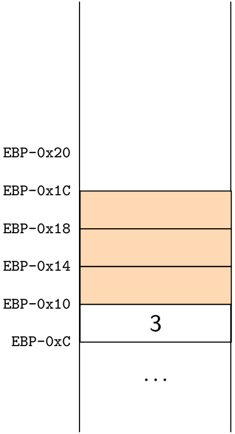
Pointers (2)

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



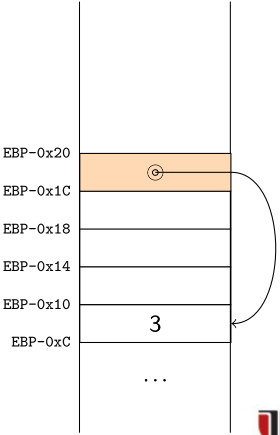
Pointers (2)

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



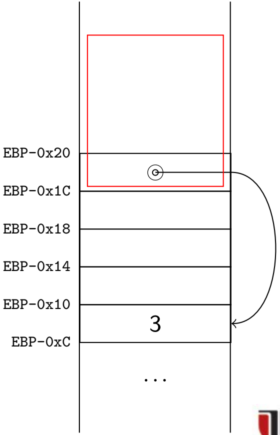
Pointers (2)

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



Pointers (2)

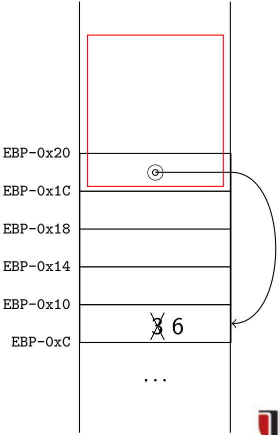
```
;int x = 3;
mov DWORD PTR [ebp-0x10],0x3
;f1(x);
mov eax,DWORD PTR [ebp-0x10]
push eax
call 57d <f1>
add esp,0x4
;printf("%d\n", x);
...
;f2(&x);
sub esp,0xc
lea eax,[ebp-0x10]
push eax
call 590 <f2>
add esp,0x10
;printf("%d\n", x);
...
```



UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

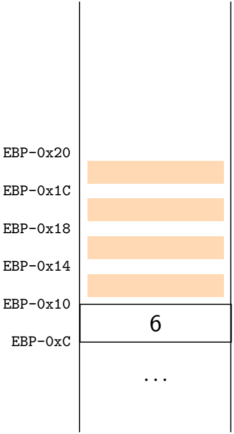
Pointers (2)

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



Pointers (2)

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



UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Arrays Are Pointers (1)

test_array_ptr.c:

```
#include <stdio.h>
void f(int *v) {
    v[0] = 25; // *v = 25
    *(v + 2) = 17; // v[2] = 17
    *((char*)v + 5) = 1; // !!!
    3[v] = 44; // v[3] = 44
}
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;
}
```

- `v` variable (i.e. the array name)
  - is a pointer to the beginning of the array
  - points where the array starts
  - it is a pointer of the same type as the array elements
- adding N (an integer) to a pointer ⇒ add "`N*sizeof(ptr. type)`"

Arrays Are Pointers (1)

test_array_ptr.c:

```
#include <stdio.h>
void f(int *v) {
    v[0] = 25; // *v = 25
    *(v + 2) = 17; // v[2] = 17
    *((char*)v + 5) = 1; // !!!
    3[v] = 44; // v[3] = 44
}
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;
}
```

- `v` variable (i.e. the array name)
  - is a pointer to the beginning of the array
  - points where the array starts
  - it is a pointer of the same type as the array elements
- adding N (an integer) to a pointer ⇒ add "`N*sizeof(ptr. type)`"

Arrays Are Pointers (1)

test_array_ptr.c:

```
#include <stdio.h>
void f(int *v) {
    v[0] = 25; // *v = 25
    *(v + 2) = 17; // v[2] = 17
    *((char*)v + 5) = 1; // !!!
    3[v] = 44; // v[3] = 44
}
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;
}
```

- `v` variable (i.e. the array name)
    - is a pointer to the beginning of the array
    - points where the array starts
    - it is a pointer of the same type as the array elements
- adding N (an integer) to a pointer ⇒ add "`N*sizeof(ptr. type)`"

Arrays Are Pointers (1)

test_array_ptr.c:

```
#include <stdio.h>
void f(int *v) {
    v[0] = 25; // *v = 25
    *(v + 2) = 17; // v[2] = 17
    *((char*)v + 5) = 1; // !!!
    3[v] = 44; // v[3] = 44
}
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;
}
```

- `v` variable (i.e. the array name)
  - is a pointer to the beginning of the array
  - points where the array starts
  - it is a pointer of the same type as the array elements
- adding N (an integer) to a pointer ⇒ add "`N*sizeof(ptr. type)`"

Arrays Are Pointers (1)

test_array_ptr.c:

```
#include <stdio.h>
void f(int *v) {
    v[0] = 25; // *v = 25
    *(v + 2) = 17; // v[2] = 17
    *((char*)v + 5) = 1; // !!!
    3[v] = 44; // v[3] = 44
}
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;
}
```

- `v` variable (i.e. the array name)
  - is a pointer to the beginning of the array
  - points where the array starts
  - it is a pointer of the same type as the array elements
- adding N (an integer) to a pointer ⇒ add "`N*sizeof(ptr. type)`"

Arrays Are Pointers (1)

test_array_ptr.c:

```
#include <stdio.h>
void f(int *v) {
    v[0] = 25; // *v = 25
    *(v + 2) = 17; // v[2] = 17
    *((char*)v + 5) = 1; // !!!
    3[v] = 44; // v[3] = 44
}
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;
}
```

- `v` variable (i.e. the array name)
  - is a pointer to the beginning of the array
  - points where the array starts
  - it is a pointer of the same type as the array elements
- adding N (an integer) to a pointer ⇒ add "`N*sizeof(ptr. type)`"

Arrays Are Pointers (1)

test_array_ptr.c:

```
#include <stdio.h>
void f(int *v) {
    v[0] = 25; // *v = 25
    *(v + 2) = 17; // v[2] = 17
    *((char*)v + 5) = 1; // !!!
    3[v] = 44; // v[3] = 44
}
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;
}
```

- `v` variable (i.e. the array name)
    - is a pointer to the beginning of the array
    - points where the array starts
    - it is a pointer of the same type as the array elements
- adding N (an integer) to a pointer ⇒ add "`N*sizeof(ptr. type)`"

```
$ ./test_array_ptr
```

Arrays Are Pointers (1)

test_array_ptr.c:

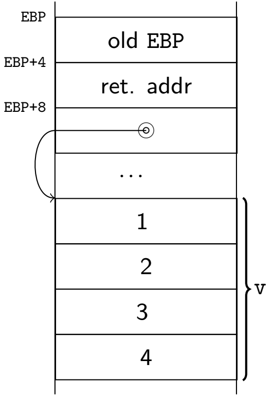
```
#include <stdio.h>
void f(int *v) {
    v[0] = 25; // *v = 25
    *(v + 2) = 17; // v[2] = 17
    *((char*)v + 5) = 1; // !!!
    3[v] = 44; // v[3] = 44
}
int main() {
    int v[] = {1, 2, 3, 4};
    int i, n = sizeof(v)/sizeof(v[0]);
    f(v);
    for(i=0; i<n; i++){
        printf("%d ", v[i]);
    }
    printf("\n");
    return 0;
}
```

- `v` variable (i.e. the array name)
  - is a pointer to the beginning of the array
  - points where the array starts
  - it is a pointer of the same type as the array elements
- adding N (an integer) to a pointer ⇒ add "`N*sizeof(ptr. type)`"

```
$ ./test_array_ptr
25 258 17 44
```

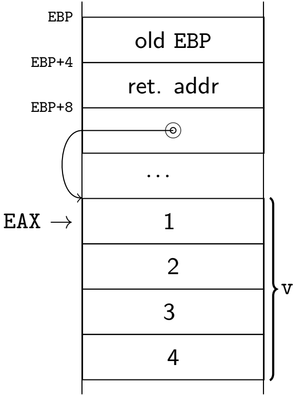
Arrays Are Pointers (2)

```
;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
...
```



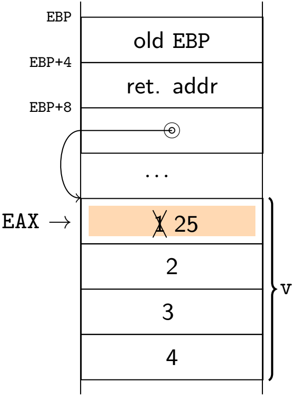
Arrays Are Pointers (2)

```
;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
...
```



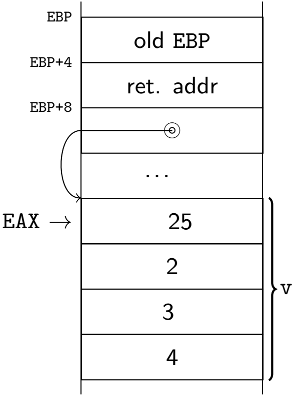
Arrays Are Pointers (2)

```
;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
...
```



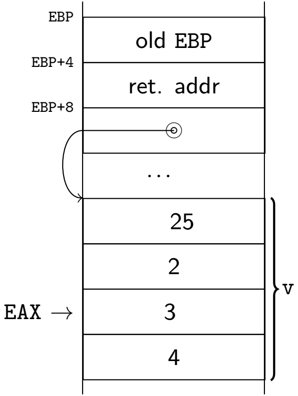
Arrays Are Pointers (2)

```
;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
...
```



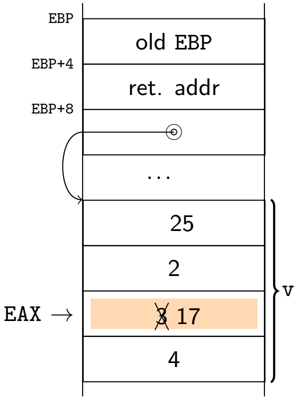
Arrays Are Pointers (2)

```
;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
...
```



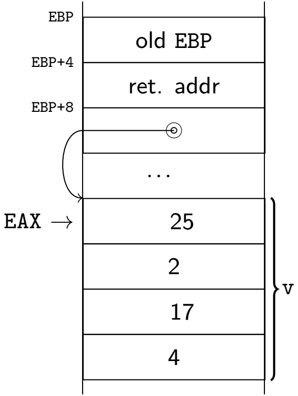
Arrays Are Pointers (2)

```
;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
...
```



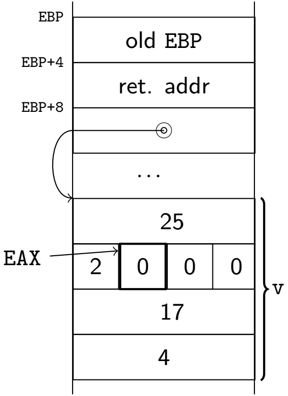
Arrays Are Pointers (2)

```
;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
...
```



Arrays Are Pointers (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
...
```

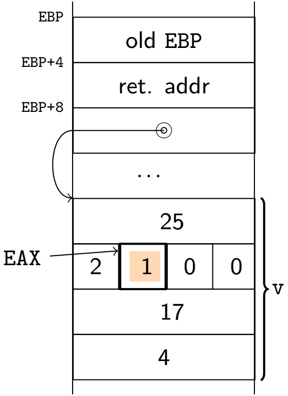


Note: due to the **little-endian representation**

- the value of v[1], i.e. 0x00000002, stored in memory as bytes 0x02 0x00 0x00 0x00 (from smaller to bigger addresses)
- ⇒ v[1] becomes 0x02 0x01 0x00 0x00

Arrays Are Pointers (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
...
```

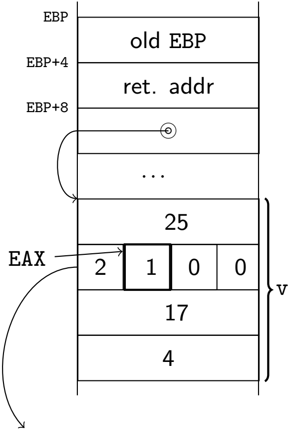


Note: due to the **little-endian representation**

- the value of v[1], i.e. 0x00000002, stored in memory as bytes 0x02 0x00 0x00 0x00 (from smaller to bigger addresses)
- ⇒ v[1] becomes 0x02 0x01 0x00 0x00

Arrays Are Pointers (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
...
```



$2 \cdot 256^0 + 1 \cdot 256^1 = 258$

Note: due to the **little-endian representation**

- the value of v[1], i.e. 0x00000002, stored in memory as bytes 0x02 0x00 0x00 0x00 (from smaller to bigger addresses)
- ⇒ v[1] becomes 0x02 0x01 0x00 0x00

Data Structures and Pointers

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

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

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

Data Structures and Pointers

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

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

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

Assigning values to structure fields

```
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';
```

Data Structures and Pointers

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

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

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

Initialization when declared

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

Data Structures and Pointers

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

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

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

Pointers to data structures:

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

How do we access the a field?

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

Recommended style: (->).

Data Structures and Pointers

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

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

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

Pointers to data structures:

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

How do we access the a field?

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

Recommended style: (->).
How do we access (sub)field `x` of field `p`?

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

Data Structures and Pointers

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

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

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

Data structure's size

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

Data Structures and Pointers

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

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

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

Data structure's size

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

```
Point size = 8
```

Data Structures and Pointers

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

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

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

Data structure's size

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

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

Data Structures and Pointers

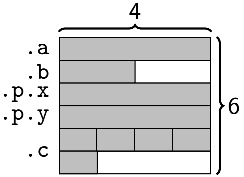
```
struct Point {
    int x;
    int y;
};

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

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

Data structure's size

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

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

Each data structure field is aligned at DWORD (4 bytes).



UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Data Structures and Pointers

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

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

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

Data Structure Alignment

How to force BYTE-alignment, if needed?

gcc (Linux)

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

Visual Studio (Windows)

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

Data Structures and Pointers

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

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

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

Cast to data structures

```
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);
```

What will be displayed?

Data Structures and Pointers

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

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

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

Cast to data structures

```
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);
```

What will be displayed?

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

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Dynamic Memory Allocation (1)

Functions `malloc` and `free`

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

- `malloc` allocates memory area on the *heap* and returns a pointer to the allocated memory
- `free` releases an allocated memory area having a pointer to it

Dynamic Memory Allocation (2)

When does it make sense to allocate memory dynamically?

- when working with dynamic structures like
  - linked lists
  - trees, graphs
- when do not know in advance the (maximum) data size
  - we cannot declare a certain (maximum) size
- when data size is too large to be stored on the stack
  - default stack size on Linux: 8MB
  - default stack size on Windows: 1MB
  - changing default size on Linux: use [illegible] setrlimit()
  - changing default size on Windows: use the [illegible] the CreateThread(...)

NOTE: dynamic allocation is slower than allocation on the stack (as local variable)

Dynamic Memory Allocation (2)

When does it make sense to allocate memory dynamically?

- when working with dynamic structures like
  - linked lists
  - trees, graphs
- when do not know in advance the (maximum) data size
  - we cannot declare a certain (maximum) size
- when data size is too large to be stored on the stack
  - default stack size on Linux: $8MB$
  - default stack size on Windows: $1MB$
  - changing default size on Linux: use "`ulimit -s ...`" or `setrlimit()`
  - changing default size on Windows: use the `dwStackSize` parameter of the `CreateThread(...)`

NOTE: dynamic allocation is slower than allocation on the stack (as local variable)

Dynamic Memory Allocation (2)

When does it make sense to allocate memory dynamically?

- when working with dynamic structures like
  - linked lists
  - trees, graphs
- when do not know in advance the (maximum) data size
  - we cannot declare a certain (maximum) size
- when data size is too large to be stored on the stack
  - default stack size on Linux: 8*MB*
  - default stack size on Windows: 1*MB*
  - changing default size on Linux: use "`ulimit -s ...`" or `setrlimit()`
  - changing default size on Windows: use the `dwStackSize` parameter of the `CreateThread(...)`

NOTE: dynamic allocation is slower than allocation on the stack (as local variable)

Dynamic Memory Allocation (2)

When does it make sense to allocate memory dynamically?

- when working with dynamic structures like
  - linked lists
  - trees, graphs
- when do not know in advance the (maximum) data size
  - we cannot declare a certain (maximum) size
- when data size is too large to be stored on the stack
  - default stack size on Linux: 8*MB*
  - default stack size on Windows: 1*MB*
  - changing default size on Linux: use "`ulimit -s ...`" or `setrlimit()`
  - changing default size on Windows: use the `dwStackSize` parameter of the `CreateThread(...)`

NOTE: dynamic allocation is slower than allocation on the stack (as local variable)

Dynamic Memory Allocation (2)

When does it make sense to allocate memory dynamically?

- when working with dynamic structures like
  - linked lists
  - trees, graphs
- when do not know in advance the (maximum) data size
  - we cannot declare a certain (maximum) size
- when data size is too large to be stored on the stack
  - default stack size on Linux: 8*MB*
  - default stack size on Windows: 1*MB*
  - changing default size on Linux: use "`ulimit -s ...`" or `setrlimit()`
  - changing default size on Windows: use the `dwStackSize` parameter of the `CreateThread(...)`

NOTE: dynamic allocation is slower than allocation on the stack (as local variable)

Dynamic Memory Allocation (2)

When does it make sense to allocate memory dynamically?

- when working with dynamic structures like
  - linked lists
  - trees, graphs
- when do not know in advance the (maximum) data size
  - we cannot declare a certain (maximum) size
- when data size is too large to be stored on the stack
  - default stack size on Linux: $8MB$
  - default stack size on Windows: $1MB$
  - changing default size on Linux: use "`ulimit -s ...`" or `setrlimit()`
  - changing default size on Windows: use the `dwStackSize` parameter of the `CreateThread(...)`

NOTE: dynamic allocation is slower than allocation on the stack (as local variable)

Dynamic Memory Allocation (2)

When does it make sense to allocate memory dynamically?

- when working with dynamic structures like
  - linked lists
  - trees, graphs
- when do not know in advance the (maximum) data size
  - we cannot declare a certain (maximum) size
- when data size is too large to be stored on the stack
  - default stack size on Linux: $8MB$
  - default stack size on Windows: $1MB$
  - changing default size on Linux: use "`ulimit -s ...`" or `setrlimit()`
  - changing default size on Windows: use the `dwStackSize` parameter of the `CreateThread(...)`

NOTE: dynamic allocation is slower than allocation on the stack (as local variable)

Dynamic Memory Allocation (2)

When does it make sense to allocate memory dynamically?

- when working with dynamic structures like
  - linked lists
  - trees, graphs
- when do not know in advance the (maximum) data size
  - we cannot declare a certain (maximum) size
- when data size is too large to be stored on the stack
  - default stack size on Linux: $8MB$
  - default stack size on Windows: $1MB$
  - changing default size on Linux: use "`ulimit -s ...`" or `setrlimit()`
  - changing default size on Windows: use the `dwStackSize` parameter of the `CreateThread(...)`

NOTE: dynamic allocation is slower than allocation on the stack (as local variable)

Dynamic Memory Allocation (2)

When does it make sense to allocate memory dynamically?

- when working with dynamic structures like
  - linked lists
  - trees, graphs
- when do not know in advance the (maximum) data size
  - we cannot declare a certain (maximum) size
- when data size is too large to be stored on the stack
  - default stack size on Linux: $8MB$
  - default stack size on Windows: $1MB$
  - changing default size on Linux: use "`ulimit -s ...`" or `setrlimit()`
  - changing default size on Windows: use the `dwStackSize` parameter of the `CreateThread(...)`

NOTE: dynamic allocation is slower than allocation on the stack (as local variable)

Dynamic Memory Allocation (2)

When does it make sense to allocate memory dynamically?

- when working with dynamic structures like
  - linked lists
  - trees, graphs
- when do not know in advance the (maximum) data size
  - we cannot declare a certain (maximum) size
- when data size is too large to be stored on the stack
  - default stack size on Linux: $8MB$
  - default stack size on Windows: $1MB$
  - changing default size on Linux: use "`ulimit -s ...`" or `setrlimit()`
  - changing default size on Windows: use the `dwStackSize` parameter of the `CreateThread(...)`

NOTE: dynamic allocation is slower than allocation on the stack (as local variable)

Dynamic Memory Allocation (2)

When does it make sense to allocate memory dynamically?

- when working with dynamic structures like
  - linked lists
  - trees, graphs
- when do not know in advance the (maximum) data size
  - we cannot declare a certain (maximum) size
- when data size is too large to be stored on the stack
  - default stack size on Linux: $8MB$
  - default stack size on Windows: $1MB$
  - changing default size on Linux: use "`ulimit -s ...`" or `setrlimit()`
  - changing default size on Windows: use the `dwStackSize` parameter of the `CreateThread(...)`

NOTE: dynamic allocation is slower than allocation on the stack (as local variable)

Dynamic Memory Allocation (2)

When does it make sense to allocate memory dynamically?

- when working with dynamic structures like
  - linked lists
  - trees, graphs
- when do not know in advance the (maximum) data size
  - we cannot declare a certain (maximum) size
- when data size is too large to be stored on the stack
  - default stack size on Linux: $8MB$
  - default stack size on Windows: $1MB$
  - changing default size on Linux: use "`ulimit -s ...`" or `setrlimit()`
  - changing default size on Windows: use the `dwStackSize` parameter of the `CreateThread(...)`

NOTE: dynamic allocation is slower than allocation on the stack (as local variable)

Dynamic Memory Allocation (3)

Memory leaks

- any dynamically allocated memory should be explicitly released with `free()`
  - otherwise it remains allocated until the program ends
- C language has **no garbage collector**
- sometimes memory is not release in the same function where it is allocated
  - *example*: a function allocates memory (for storing some results) and returns a pointer to that memory
  - the function getting such a pointer should release the memory, when not needing it anymore
- example of a classical memory leak (**don't do like this!**)

```
int x[1000];
int *p = (int*)malloc(1000 * sizeof(int));
p = x; // the only pointer to the dynamically allocated memory is lost
```

Dynamic Memory Allocation (3)

Memory leaks

- any dynamically allocated memory should be explicitly released with `free()`
  - otherwise it remains allocated until the program ends
- C language has **no garbage collector**
- sometimes memory is not release in the same function where it is allocated
  - *example*: a function allocates memory (for storing some results) and returns a pointer to that memory
  - the function getting such a pointer should release the memory, when not needing it anymore
- example of a classical memory leak (**don't do like this!**)

```
int x[1000];
int *p = (int*)malloc(1000 * sizeof(int));
p = x; // the only pointer to the dynamically allocated memory lost
```

Dynamic Memory Allocation (3)

Memory leaks

- any dynamically allocated memory should be explicitly released with `free()`
  - otherwise it remains allocated until the program ends
- C language has **no garbage collector**
- sometimes memory is not release in the same function where it is allocated
  - *example*: a function allocates memory (for storing some results) and returns a pointer to that memory
  - the function getting such a pointer should release the memory, when not needing it anymore
- example of a classical memory leak (**don't do like this!**)

```
int x[1000];
int *p = (int*)malloc(1000 * sizeof(int));
p = x; // the only pointer to the dynamically allocated memory is lost
```

Dynamic Memory Allocation (3)

Memory leaks

- any dynamically allocated memory should be explicitly released with `free()`
  - otherwise it remains allocated until the program ends
- C language has **no garbage collector**
- sometimes memory is not release in the same function where it is allocated
  - *example*: a function allocates memory (for storing some results) and returns a pointer to that memory
  - the function getting such a pointer should release the memory, when not needing it anymore
- example of a classical memory leak (**don't do like this!**)

```
int x[1000];
int *p = (int*)malloc(1000 * sizeof(int));
p = x; // the only pointer to the dynamically allocated memory lost
```

Dynamic Memory Allocation (3)

Memory leaks

- any dynamically allocated memory should be explicitly released with `free()`
  - otherwise it remains allocated until the program ends
- C language has **no garbage collector**
- sometimes memory is not release in the same function where it is allocated
  - *example*: a function allocates memory (for storing some results) and returns a pointer to that memory
  - the function getting such a pointer should release the memory, when not needing it anymore
- example of a classical memory leak (**don't do like this!**)

```
int x[1000];
int *p = (int*)malloc(1000 * sizeof(int));
p = x; // the only pointer to the dynamically allocated memory is lost
```

String Operations

- in C a string is
  - a byte array ended with byte 0 (zero or null) ⇒ null-terminated strings
  - each byte value is the code of a printable character
  - last byte is the unprintable character with code '\0' ⇒ **NUL-terminated strings**
- can be handled like a normal array
- in `string.h` there is a collection of **string handling functions**
  - `strlen()`
  - `strncpy()` (**do not use** `strcpy()`**!**)
  - `strcmp()`
  - `strchr()`
  - `strstr()`
- **string handling functions assume NUL-terminated strings**
  - ⇒ do not use them when strings not terminated with NUL

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

String Operations

- in C a string is
  - a byte array ended with byte 0 (zero or null) ⇒ null-terminated strings
  - each byte value is the code of a printable character
  - last byte is the unprintable character with code '\0' ⇒ **NUL-terminated strings**
- can be handled like a normal array
- in `string.h` there is a collection of **string handling functions**
  - `strlen()`
  - `strncpy()` **(do not use `strcpy()`!)**
  - `strcmp()`
  - `strchr()`
  - `strstr()`
- **string handling functions assume NUL-terminated strings**
  - ⇒ do not use them when strings not terminated with NUL

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

String Operations

- in C a string is
  - a byte array ended with byte 0 (zero or null) ⇒ null-terminated strings
  - each byte value is the code of a printable character
  - last byte is the unprintable character with code '\0' ⇒ **NUL-terminated strings**
- can be handled like a normal array
- in `string.h` there is a collection of **string handling functions**
  - `strlen()`
  - `strncpy()` **(do not use `strcpy()`!)**
  - `strcmp()`
  - `strchr()`
  - `strstr()`
- **string handling functions assume NUL-terminated strings**
  - ⇒ do not use them when strings not terminated with NUL

String Operations

- in C a string is
  - a byte array ended with byte 0 (zero or null) $\Rightarrow$ null-terminated strings
  - each byte value is the code of a printable character
  - last byte is the unprintable character with code '\0' $\Rightarrow$ **NUL-terminated strings**
- can be handled like a normal array
- in `string.h` there is a collection of **string handling functions**
  - `strlen()`
  - `strncpy()` (**do not use `strcpy()`!**)
  - `strcmp()`
  - `strchr()`
  - `strstr()`
- **string handling functions assume NUL-terminated strings**
  - $\Rightarrow$ do not use them when strings not terminated with NUL

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

String Operations

- in C a string is
  - a byte array ended with byte 0 (zero or null) ⇒ null-terminated strings
  - each byte value is the code of a printable character
  - last byte is the unprintable character with code '\0' ⇒ **NUL-terminated strings**
- can be handled like a normal array
- in `string.h` there is a collection of **string handling functions**
  - `strlen()`
  - `strncpy()` (**do not use `strcpy()`!**)
  - `strcmp()`
  - `strchr()`
  - `strstr()`
- **string handling functions assume NUL-terminated strings**
  - ⇒ do not use them when strings not terminated with NUL

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

String Operations

- in C a string is
  - a byte array ended with byte 0 (zero or null) $\Rightarrow$ null-terminated strings
  - each byte value is the code of a printable character
  - last byte is the unprintable character with code '\0' $\Rightarrow$ **NUL-terminated strings**
- can be handled like a normal array
- in `string.h` there is a collection of **string handling functions**
  - `strlen()`
  - `strncpy()` **(do not use `strcpy()`!)**
  - `strcmp()`
  - `strchr()`
  - `strstr()`
- **string handling functions assume NUL-terminated strings**
  - $\Rightarrow$ do not use them when strings not terminated with NUL

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

String Operations

- in C a string is
  - a byte array ended with byte 0 (zero or null) ⇒ null-terminated strings
  - each byte value is the code of a printable character
  - last byte is the unprintable character with code '\0' ⇒ **NUL-terminated strings**
- can be handled like a normal array
- in `string.h` there is a collection of **string handling functions**
  - `strlen()`
  - `strncpy()` (**do not use `strcpy()`!**)
  - `strcmp()`
  - `strchr()`
  - `strstr()`
- **string handling functions assume NUL-terminated strings**
  - ⇒ do not use them when strings not terminated with NUL

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

String Operations

- in C a string is
  - a byte array ended with byte 0 (zero or null) ⇒ null-terminated strings
  - each byte value is the code of a printable character
  - last byte is the unprintable character with code '\0' ⇒ **NUL-terminated strings**
- can be handled like a normal array
- in `string.h` there is a collection of **string handling functions**
  - `strlen()`
  - `strncpy()` (**do not use `strcpy()`!**)
  - `strcmp()`
  - `strchr()`
  - `strstr()`
- **string handling functions assume NUL-terminated strings**
  - ⇒ do not use them when strings not terminated with NUL

String Operations

- in C a string is
  - a byte array ended with byte 0 (zero or null) ⇒ null-terminated strings
  - each byte value is the code of a printable character
  - last byte is the unprintable character with code '\0' ⇒ **NUL-terminated strings**
- can be handled like a normal array
- in `string.h` there is a collection of **string handling functions**
  - `strlen()`
  - `strncpy()` (**do not use `strcpy()`!**)
  - `strcmp()`
  - `strchr()`
  - `strstr()`
- **string handling functions assume NUL-terminated strings**
  - ⇒ do not use them when strings not terminated with NUL

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

String Operations

- in C a string is
  - a byte array ended with byte 0 (zero or null) ⇒ null-terminated strings
  - each byte value is the code of a printable character
  - last byte is the unprintable character with code '\0' ⇒ **NUL-terminated strings**
- can be handled like a normal array
- in `string.h` there is a collection of **string handling functions**
  - `strlen()`
  - `strncpy()` (**do not use `strcpy()`!**)
  - `strcmp()`
  - `strchr()`
  - `strstr()`
- string handling functions assume NUL-terminated strings
  - ⇒ do not use them when strings not terminated with NUL

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

String Operations

- in C a string is
  - a byte array ended with byte 0 (zero or null) ⇒ null-terminated strings
  - each byte value is the code of a printable character
  - last byte is the unprintable character with code '\0' ⇒ **NUL-terminated strings**
- can be handled like a normal array
- in `string.h` there is a collection of **string handling functions**
  - `strlen()`
  - `strncpy()` (**do not use `strcpy()`!**)
  - `strcmp()`
  - `strchr()`
  - `strstr()`
- string handling functions assume NUL-terminated strings
  - ⇒ do not use them when strings not terminated with NUL

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

String Operations

- in C a string is
  - a byte array ended with byte 0 (zero or null) ⇒ null-terminated strings
  - each byte value is the code of a printable character
  - last byte is the unprintable character with code '\0' ⇒ **NUL-terminated strings**
- can be handled like a normal array
- in `string.h` there is a collection of **string handling functions**
  - `strlen()`
  - `strncpy()` (**do not use `strcpy()`!**)
  - `strcmp()`
  - `strchr()`
  - `strstr()`
- **string handling functions assume NUL-terminated strings**
  - ⇒ do not use them when strings not terminated with NUL

String Operations

- in C a string is
  - a byte array ended with byte 0 (zero or null) $\Rightarrow$ null-terminated strings
  - each byte value is the code of a printable character
  - last byte is the unprintable character with code '\0' $\Rightarrow$ **NUL-terminated strings**
- can be handled like a normal array
- in `string.h` there is a collection of **string handling functions**
  - `strlen()`
  - `strncpy()` (**do not use `strcpy()`!**)
  - `strcmp()`
  - `strchr()`
  - `strstr()`
- **string handling functions assume NUL-terminated strings**
  - $\Rightarrow$ do not use them when strings not terminated with NUL

Outline

Debugger

- a program that provides us the way to execute another program (app) step by step (instruction by instruction)
- usually in order to find and fix bugs
- common operations
  - execution step by step
  - setting breakpoints
  - monitor variable values
  - monitor memory contents
  - monitor the stack contents and evolution
- must compile the source code with explicit options to generate detailed (helpful) debugging information
  - like the -g option of *gcc*
  - debugging info: variable names, correlation to the source code

Debugger

- a program that provides us the way to execute another program (app) step by step (instruction by instruction)
- usually in order to find and fix bugs
- common operations
  - execution step by step
  - setting breakpoints
  - monitor variable values
  - monitor memory contents
  - monitor the stack contents and evolution
- must compile the source code with explicit options to generate detailed (helpful) debugging information
  - like the –g option of *gcc*
  - debugging info: variable names, correlation to the source code

Debugger

- a program that provides us the way to execute another program (app) step by step (instruction by instruction)
- usually in order to find and fix bugs
- common operations
  - execution step by step
  - setting breakpoints
  - monitor variable values
  - monitor memory contents
  - monitor the stack contents and evolution
- must compile the source code with explicit options to generate detailed (helpful) debugging information
  - like the –g option of *gcc*
  - debugging info: variable names, correlation to the source code

Debugger

- a program that provides us the way to execute another program (app) step by step (instruction by instruction)
- usually in order to find and fix bugs
- common operations
  - execution step by step
  - setting breakpoints
  - monitor variable values
  - monitor memory contents
  - monitor the stack contents and evolution
- must compile the source code with explicit options to generate detailed (helpful) debugging information
  - like the –g option of *gcc*
  - debugging info: variable names, correlation to the source code

Debugger

- a program that provides us the way to execute another program (app) step by step (instruction by instruction)
- usually in order to find and fix bugs
- common operations
    - execution step by step
    - setting breakpoints
    - monitor variable values
    - monitor memory contents
    - monitor the stack contents and evolution
- must compile the source code with explicit options to generate detailed (helpful) debugging information
    - like the –g option of *gcc*
    - debugging info: variable names, correlation to the source code

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Debugger

- a program that provides us the way to execute another program (app) step by step (instruction by instruction)
- usually in order to find and fix bugs
- common operations
  - execution step by step
  - setting breakpoints
  - monitor variable values
  - monitor memory contents
  - monitor the stack contents and evolution
- must compile the source code with explicit options to generate detailed (helpful) debugging information
  - like the –g option of *gcc*
  - debugging info: variable names, correlation to the source code

Debugger

- a program that provides us the way to execute another program (app) step by step (instruction by instruction)
- usually in order to find and fix bugs
- common operations
  - execution step by step
  - setting breakpoints
  - monitor variable values
  - monitor memory contents
  - monitor the stack contents and evolution
- must compile the source code with explicit options to generate detailed (helpful) debugging information
  - like the –g option of *gcc*
  - debugging info: variable names, correlation to the source code

Debugger

- a program that provides us the way to execute another program (app) step by step (instruction by instruction)
- usually in order to find and fix bugs
- common operations
  - execution step by step
  - setting breakpoints
  - monitor variable values
  - monitor memory contents
  - monitor the stack contents and evolution
- must compile the source code with explicit options to generate detailed (helpful) debugging information
  - like the -g option of *gcc*
  - debugging info: variable names, correlation to the source code

Debugger

- a program that provides us the way to execute another program (app) step by step (instruction by instruction)
- usually in order to find and fix bugs
- common operations
  - execution step by step
  - setting breakpoints
  - monitor variable values
  - monitor memory contents
  - monitor the stack contents and evolution
- must compile the source code with explicit options to generate detailed (helpful) debugging information
  - like the −g option of *gcc*
  - debugging info: variable names, correlation to the source code

Debugger

- a program that provides us the way to execute another program (app) step by step (instruction by instruction)
- usually in order to find and fix bugs
- common operations
  - execution step by step
  - setting breakpoints
  - monitor variable values
  - monitor memory contents
  - monitor the stack contents and evolution
- must compile the source code with explicit options to generate detailed (helpful) debugging information
  - like the -g option of *gcc*
  - debugging info: variable names, correlation to the source code

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Debugger

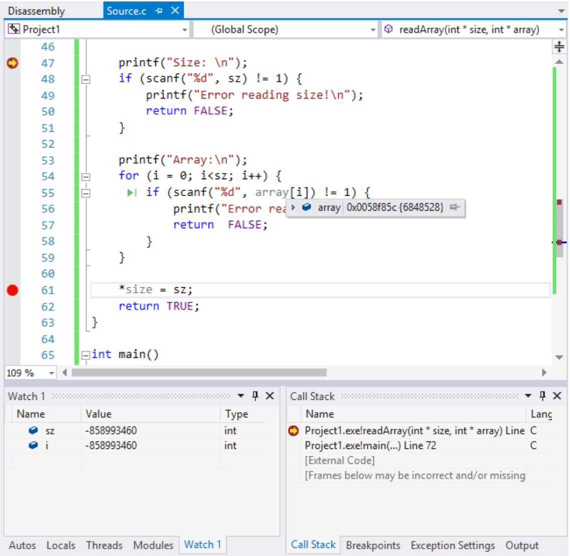
- a program that provides us the way to execute another program (app) step by step (instruction by instruction)
- usually in order to find and fix bugs
- common operations
  - execution step by step
  - setting breakpoints
  - monitor variable values
  - monitor memory contents
  - monitor the stack contents and evolution
- must compile the source code with explicit options to generate detailed (helpful) debugging information
  - like the −g option of *gcc*
  - debugging info: variable names, correlation to the source code

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Debugging with a GUI (1)

- very convenient, simple and efficient
- e.g. set breakpoints by clicking in front of line of source code
- e.g. view variable value with *mouse over* or right-click → *Add Watch*
- trace / debug a program
    - *step into*: execute and go forward one instruction entering into functions
    - *step over* execute and go forward one line (even if a function, so not entering in such a function)
    - *continue*: execute and go forward to the next breakpoint or program's end

Debugging with a GUI (2)



Debugging with GDB

Running as: `gdb <executable_file>`

Displays an interactive shell where we can type commands like:

- `break test.c:12`
- `run`
- `continue`
- `step` (step into)
- `next` (step over)
- `bt` (*backtrace*: shows the stack frames)
- `print myvar`: displays the variable's contents

Postmortem Debugging

- provides us the way to investigate (debug) a crashed program's state
- useful when our programs run on other remote systems
- steps
  - activate core dumping on the remote system
    `ulimit -c unlimited`
  - run the program until its crash $\Rightarrow$ a `core` file
  - loads the core dump into the `gdb` debugger
    `gdb <executable_file> core`

Using *valgrind* to Detect Memory Leaks

- allows analyzing a program during its runtime
- for finding out memory leaks it intercepts `malloc` and `free` (and other related functions) and keeps evidence of the allocated memory areas
- at program termination displays a report with memory leaks

```
$ valgrind ./<executable\_file> [<arguments>]
...
==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
```

- when there are leaks run it again for detailed info `-leak-check=full -show-leak-kinds=all`

Using *valgrind* to Detect Memory Leaks

- allows analyzing a program during its runtime
- for finding out memory leaks it intercepts `malloc` and `free` (and other related functions) and keeps evidence of the allocated memory areas
- at program termination displays a report with memory leaks

```
$ valgrind ./<executable\_file> [<arguments>]
...
==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
```

- when there are leaks run it again for detailed info `-leak-check=full -show-leak-kinds=all`

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Using *valgrind* to Detect Memory Leaks

- allows analyzing a program during its runtime
- for finding out memory leaks it intercepts `malloc` and `free` (and other related functions) and keeps evidence of the allocated memory areas
- at program termination displays a report with memory leaks

```
$ valgrind ./<executable\_file> [<arguments>]
...
==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
```

- when there are leaks run it again for detailed info
  `-leak-check=full -show-leak-kinds=all`

Outline

There Are No Perfect Programs

- it is almost impossible to write a complex bug-free program
  - at least from first try
- every day new bugs are found in real-life “professional” (commercial) software run by millions of users
- there are metrics that account bugs per thousands of lines of code

A good programmer could be allowed to not write perfect programs, but (s)he is required to be able to test / debug them, identify bugs and fix them!

- testing and debugging a program has same importance like writing code
  - understanding a written code is not at all the same like writing it by yourself from scratch

There Are No Perfect Programs

- it is almost impossible to write a complex bug-free program
  - at least from first try
- every day new bugs are found in real-life "professional" (commercial) software run by millions of users
- there are metrics that account bugs per thousands of lines of code

A good programmer could be allowed to not write perfect programs, but (s)he is required to be able to test / debug them, identify bugs and fix them!

- testing and debugging a program has same importance like writing code
  - understanding a written code is not at all the same like writing it by yourself from scratch

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

There Are No Perfect Programs

- it is almost impossible to write a complex bug-free program
  - at least from first try
- every day new bugs are found in real-life “professional” (commercial) software run by millions of users
- there are metrics that account bugs per thousands of lines of code

A good programmer could be allowed to not write perfect programs, but (s)he is required to be able to test / debug them, identify bugs and fix them!

- testing and debugging a program has same importance like writing code
  - understanding a written code is not at all the same like writing it by yourself from scratch

There Are No Perfect Programs

- it is almost impossible to write a complex bug-free program
  - at least from first try
- every day new bugs are found in real-life “professional” (commercial) software run by millions of users
- there are metrics that account bugs per thousands of lines of code

A good programmer could be allowed to not write perfect programs, but (s)he is required to be able to test / debug them, identify bugs and fix them!

- testing and debugging a program has same importance like writing code
  - understanding a written code is not at all the same like writing it by yourself from scratch

There Are No Perfect Programs

- it is almost impossible to write a complex bug-free program
  - at least from first try
- every day new bugs are found in real-life "professional" (commercial) software run by millions of users
- there are metrics that account bugs per thousands of lines of code

A good programmer could be allowed to not write perfect programs, but (s)he is required to be able to test / debug them, identify bugs and fix them!

- testing and debugging a program has same importance like writing code
  - understanding a written code is not at all the same like writing it by yourself from scratch

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

There Are No Perfect Programs

- it is almost impossible to write a complex bug-free program
  - at least from first try
- every day new bugs are found in real-life "professional" (commercial) software run by millions of users
- there are metrics that account bugs per thousands of lines of code

A good programmer could be allowed to not write perfect programs, but (s)he is required to be able to test / debug them, identify bugs and fix them!

- testing and debugging a program has same importance like writing code
  - understanding a written code is not at all the same like writing it by yourself from scratch

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

There Are No Perfect Programs

- it is almost impossible to write a complex bug-free program
  - at least from first try
- every day new bugs are found in real-life “professional” (commercial) software run by millions of users
- there are metrics that account bugs per thousands of lines of code

A good programmer could be allowed to not write perfect programs, but (s)he is required to be able to test / debug them, identify bugs and fix them!

- testing and debugging a program has same importance like writing code
  - understanding a written code is not at all the same like writing it by yourself from scratch

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Using Code From Other Sources

- **Do not copy-paste code you do not understand!**
- actually, **never copy-paste any code**, at least for your school projects!
- there is (almost) nothing to learn from others' code
- if really need to use someone else' code (in real life)
  - make sure you understand and control that code
  - mention the source of your code (at least in a comment)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Using Code From Other Sources

- **Do not copy-paste code you do not understand!**
- actually, **never copy-paste any code**, at least for your school projects!
- there is (almost) nothing to learn from others' code
- if really need to use someone else' code (in real life)
  - make sure you understand and control that code
  - mention the source of your code (at least in a comment)

Using Code From Other Sources

- **Do not copy-paste code you do not understand!**
- actually, **never copy-paste any code**, at least for your school projects!
- there is (almost) nothing to learn from others' code
- if really need to use someone else' code (in real life)
  - make sure you understand and control that code
  - mention the source of your code (at least in a comment)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Using Code From Other Sources

- **Do not copy-paste code you do not understand!**
- actually, **never copy-paste any code**, at least for your school projects!
- there is (almost) nothing to learn from others' code
- if really need to use someone else' code (in real life)
  - make sure you understand and control that code
  - mention the source of your code (at least in a comment)

Using Code From Other Sources

- **Do not copy-paste code you do not understand!**
- actually, **never copy-paste any code**, at least for your school projects!
- there is (almost) nothing to learn from others' code
- if really need to use someone else' code (in real life)
  - make sure you understand and control that code
  - mention the source of your code (at least in a comment)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Using Code From Other Sources

- **Do not copy-paste code you do not understand!**
- actually, **never copy-paste any code**, at least for your school projects!
- there is (almost) nothing to learn from others' code
- if really need to use someone else' code (in real life)
  - make sure you understand and control that code
  - mention the source of your code (at least in a comment)

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

A Simple Editor or an IDE?

- IDE = Integrated Development Environment
- IDE provides an integrated environment for
  - code editing with auto-completion, suggestions
  - code compilation
  - program running and debugging
  - code refactoring
- IDE's advantages
  - development efficiency
  - automatize and make efficient frequent and complex development processes
- IDE's disadvantages
  - during the learning phase has "training wheels" effect
  - when learning a new language / technology is better to use basic editor and tools
  - could be slower than basic tools

A Simple Editor or an IDE?

- IDE = Integrated Development Environment
- IDE provides an integrated environment for
  - code editing with auto-completion, suggestions
  - code compilation
  - program running and debugging
  - code refactoring
- IDE's advantages
  - development efficiency
  - automatize and make efficient frequent and complex development processes
- IDE's disadvantages
  - during the learning phase has "training wheels" effect
  - when learning a new language / technology is better to use basic editor and tools
  - could be slower than basic tools

A Simple Editor or an IDE?

- IDE = Integrated Development Environment
- IDE provides an integrated environment for
  - code editing with auto-completion, suggestions
  - code compilation
  - program running and debugging
  - code refactoring
- IDE's advantages
  - development efficiency
  - automatize and make efficient frequent and complex development processes
- IDE's disadvantages
  - during the learning phase has "training wheels" effect
  - when learning a new language / technology is better to use basic editor and tools
  - could be slower than basic tools

A Simple Editor or an IDE?

- IDE = Integrated Development Environment
- IDE provides an integrated environment for
  - code editing with auto-completion, suggestions
  - code compilation
  - program running and debugging
  - code refactoring
- IDE's advantages
  - development efficiency
  - automatize and make efficient frequent and complex development processes
- IDE's disadvantages
  - during the learning phase has "training wheels" effect
  - when learning a new language / technology is better to use basic editor and tools
  - could be slower than basic tools

A Simple Editor or an IDE?

- IDE = Integrated Development Environment
- IDE provides an integrated environment for
  - code editing with auto-completion, suggestions
  - code compilation
  - program running and debugging
  - code refactoring
- IDE's advantages
  - development efficiency
  - automatize and make efficient frequent and complex development processes
- IDE's disadvantages
  - during the learning phase has "training wheels" effect
  - when learning a new language / technology is better to use basic editor and tools
  - could be slower than basic tools

A Simple Editor or an IDE?

- IDE = Integrated Development Environment
- IDE provides an integrated environment for
  - code editing with auto-completion, suggestions
  - code compilation
  - program running and debugging
  - code refactoring
- IDE's advantages
  - development efficiency
  - automatize and make efficient frequent and complex development processes
- IDE's disadvantages
  - during the learning phase has "training wheels" effect
  - when learning a new language / technology is better to use basic editor and tools
  - could be slower than basic tools

A Simple Editor or an IDE?

- IDE = Integrated Development Environment
- IDE provides an integrated environment for
  - code editing with auto-completion, suggestions
  - code compilation
  - program running and debugging
  - code refactoring
- IDE's advantages
  - development efficiency
  - automatize and make efficient frequent and complex development processes
- IDE's disadvantages
  - during the learning phase has "training wheels" effect
  - when learning a new language / technology is better to use basic editor and tools
  - could be slower than basic tools

A Simple Editor or an IDE?

- IDE = Integrated Development Environment
- IDE provides an integrated environment for
    - code editing with auto-completion, suggestions
    - code compilation
    - program running and debugging
    - code refactoring
- IDE's advantages
    - development efficiency
    - automatize and make efficient frequent and complex development processes
- IDE's disadvantages
    - during the learning phase has "training wheels" effect
    - when learning a new language / technology is better to use basic editor and tools
    - could be slower than basic tools

A Simple Editor or an IDE?

- IDE = Integrated Development Environment
- IDE provides an integrated environment for
  - code editing with auto-completion, suggestions
  - code compilation
  - program running and debugging
  - code refactoring
- IDE's advantages
  - development efficiency
  - automatize and make efficient frequent and complex development processes
- IDE's disadvantages
  - during the learning phase has "training wheels" effect
  - when learning a new language / technology is better to use basic editor and tools
  - could be slower than basic tools

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

A Simple Editor or an IDE?

- IDE = Integrated Development Environment
- IDE provides an integrated environment for
  - code editing with auto-completion, suggestions
  - code compilation
  - program running and debugging
  - code refactoring
- IDE's advantages
  - development efficiency
  - automatize and make efficient frequent and complex development processes
- IDE's disadvantages
  - during the learning phase has "training wheels" effect
  - when learning a new language / technology is better to use basic editor and tools
  - could be slower than basic tools

A Simple Editor or an IDE?

- IDE = Integrated Development Environment
- IDE provides an integrated environment for
  - code editing with auto-completion, suggestions
  - code compilation
  - program running and debugging
  - code refactoring
- IDE's advantages
  - development efficiency
  - automatize and make efficient frequent and complex development processes
- IDE's disadvantages
  - during the learning phase has "training wheels" effect
  - when learning a new language / technology is better to use basic editor and tools
  - could be slower than basic tools

A Simple Editor or an IDE?

- IDE = Integrated Development Environment
- IDE provides an integrated environment for
  - code editing with auto-completion, suggestions
  - code compilation
  - program running and debugging
  - code refactoring
- IDE's advantages
  - development efficiency
  - automatize and make efficient frequent and complex development processes
- IDE's disadvantages
  - during the learning phase has "training wheels" effect
  - when learning a new language / technology is better to use basic editor and tools
  - could be slower than basic tools

A Simple Editor or an IDE?

- IDE = Integrated Development Environment
- IDE provides an integrated environment for
  - code editing with auto-completion, suggestions
  - code compilation
  - program running and debugging
  - code refactoring
- IDE's advantages
  - development efficiency
  - automatize and make efficient frequent and complex development processes
- IDE's disadvantages
  - during the learning phase has "training wheels" effect
  - when learning a new language / technology is better to use basic editor and tools
  - could be slower than basic tools

Watch the Uninitialized Variables

- local variables are allocated on the stack
- just declaring a variable leads to assembly code (actually, machine code generated by compiler) that simply decrease the ESP register
  - i.e. reserve space on stack for the declared variable
- ⇒ initial value of the variable depends on the current contents of the reserved space
- **initialize variables before using them**!
  - where they are declared or as close as possible to their declaration
- dynamically allocated memory is also uninitialized, i.e. initialized with undefined values
- after calling `free(p)`, `p` will point to an undefined memory location
  - ⇒ **dangling pointer**
  - **after releasing the memory a pointer points to, assign it NULL value!**

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Watch the Uninitialized Variables

- local variables are allocated on the stack
- just declaring a variable leads to assembly code (actually, machine code generated by compiler) that simply decrease the ESP register
  - i.e. reserve space on stack for the declared variable
- ⇒ initial value of the variable depends on the current contents of the reserved space
- **initialize variables before using them**!
  - where they are declared or as close as possible to their declaration
- dynamically allocated memory is also uninitialized, i.e. initialized with undefined values
- after calling `free(p)`, `p` will point to an undefined memory location
  - ⇒ **dangling pointer**
  - **after releasing the memory a pointer points to, assign it N value!**

Watch the Uninitialized Variables

- local variables are allocated on the stack
- just declaring a variable leads to assembly code (actually, machine code generated by compiler) that simply decrease the ESP register
  - i.e. reserve space on stack for the declared variable
- $\Rightarrow$ initial value of the variable depends on the current contents of the reserved space
- **initialize variables before using them**!
  - where they are declared or as close as possible to their declaration
- dynamically allocated memory is also uninitialized, i.e. initialized with undefined values
- after calling `free(p)`, `p` will point to an undefined memory location
  - $\Rightarrow$ **dangling pointer**
  - **after releasing the memory a pointer points to, assign it NULL value!**

Watch the Uninitialized Variables

- local variables are allocated on the stack
- just declaring a variable leads to assembly code (actually, machine code generated by compiler) that simply decrease the ESP register
  - i.e. reserve space on stack for the declared variable
- $\Rightarrow$ initial value of the variable depends on the current contents of the reserved space
- **initialize variables before using them!**
  - where they are declared or as close as possible to their declaration
- dynamically allocated memory is also uninitialized, i.e. initialized with undefined values
- after calling `free(p)`, `p` will point to an undefined memory location
  - $\Rightarrow$ **dangling pointer**
  - **after releasing the memory a pointer points to, assign it NULL value!**

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Watch the Uninitialized Variables

- local variables are allocated on the stack
- just declaring a variable leads to assembly code (actually, machine code generated by compiler) that simply decrease the ESP register
  - i.e. reserve space on stack for the declared variable
- $\Rightarrow$ initial value of the variable depends on the current contents of the reserved space
- **initialize variables before using them**!
  - where they are declared or as close as possible to their declaration
- dynamically allocated memory is also uninitialized, i.e. initialized with undefined values
- after calling `free(p)`, `p` will point to an undefined memory location
  - $\Rightarrow$ **dangling pointer**
  - **after releasing the memory a pointer points to, assign it NULL value!**

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Watch the Uninitialized Variables

- local variables are allocated on the stack
- just declaring a variable leads to assembly code (actually, machine code generated by compiler) that simply decrease the ESP register
  - i.e. reserve space on stack for the declared variable
- $\Rightarrow$ initial value of the variable depends on the current contents of the reserved space
- **initialize variables before using them**!
  - where they are declared or as close as possible to their declaration
- dynamically allocated memory is also uninitialized, i.e. initialized with undefined values
- after calling `free(p)`, `p` will point to an undefined memory location
  - $\Rightarrow$ **dangling pointer**
  - **after releasing the memory a pointer points to, assign it NULL value!**

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Watch the Uninitialized Variables

- local variables are allocated on the stack
- just declaring a variable leads to assembly code (actually, machine code generated by compiler) that simply decrease the ESP register
  - i.e. reserve space on stack for the declared variable
- $\Rightarrow$ initial value of the variable depends on the current contents of the reserved space
- **initialize variables before using them**!
  - where they are declared or as close as possible to their declaration
- dynamically allocated memory is also uninitialized, i.e. initialized with undefined values
- after calling `free(p)`, `p` will point to an undefined memory location
  - $\Rightarrow$ **dangling pointer**
  - **after releasing the memory a pointer points to, assign it NULL value!**

Watch the Uninitialized Variables

- local variables are allocated on the stack
- just declaring a variable leads to assembly code (actually, machine code generated by compiler) that simply decrease the ESP register
  - i.e. reserve space on stack for the declared variable
- $\Rightarrow$ initial value of the variable depends on the current contents of the reserved space
- **initialize variables before using them**!
  - where they are declared or as close as possible to their declaration
- dynamically allocated memory is also uninitialized, i.e. initialized with undefined values
- after calling `free(p)`, `p` will point to an undefined memory location
  - $\Rightarrow$ **dangling pointer**
  - **after releasing the memory a pointer points to, assign it NULL value!**

Watch the Uninitialized Variables

- local variables are allocated on the stack
- just declaring a variable leads to assembly code (actually, machine code generated by compiler) that simply decrease the ESP register
  - i.e. reserve space on stack for the declared variable
- $\Rightarrow$ initial value of the variable depends on the current contents of the reserved space
- **initialize variables before using them**!
  - where they are declared or as close as possible to their declaration
- dynamically allocated memory is also uninitialized, i.e. initialized with undefined values
- after calling `free(p)`, `p` will point to an undefined memory location
  - $\Rightarrow$ **dangling pointer**
  - after releasing the memory a pointer points to, assign it NULL value!

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Watch the Uninitialized Variables

- local variables are allocated on the stack
- just declaring a variable leads to assembly code (actually, machine code generated by compiler) that simply decrease the ESP register
  - i.e. reserve space on stack for the declared variable
- ⇒ initial value of the variable depends on the current contents of the reserved space
- **initialize variables before using them**!
  - where they are declared or as close as possible to their declaration
- dynamically allocated memory is also uninitialized, i.e. initialized with undefined values
- after calling `free(p)`, `p` will point to an undefined memory location
  - ⇒ **dangling pointer**
  - **after releasing the memory a pointer points to, assign it NULL value!**

Watch the Global Variables

- global variable not explicitly assigned values, are initialized with 0
- global variables are visible from any function of your program
  - ⇒ if not a constant, anyone can change them
- global variables can be changed concurrently from multiple threads
  - ⇒ unexpected, unpredicted values
  - such a code is called thread-unsafe
- **do not use global variable, when not really needed!**
  - better and safer to give a function the needed context information as parameters

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Watch the Global Variables

- global variable not explicitly assigned values, are initialized with 0
- global variables are visible from any function of your program
  - ⇒ if not a constant, anyone can change them
- global variables can be changed concurrently from multiple threads
  - ⇒ unexpected, unpredicted values
  - such a code is called thread-unsafe
- **do not use global variable, when not really needed!**
  - better and safer to give a function the needed context information as parameters

Watch the Global Variables

- global variable not explicitly assigned values, are initialized with 0
- global variables are visible from any function of your program
  - $\Rightarrow$ if not a constant, anyone can change them
- global variables can be changed concurrently from multiple threads
  - $\Rightarrow$ unexpected, unpredicted values
  - such a code is called thread-unsafe
- **do not use global variable, when not really needed!**
  - better and safer to give a function the needed context information as parameters

Watch the Global Variables

- global variable not explicitly assigned values, are initialized with 0
- global variables are visible from any function of your program
  - $\Rightarrow$ if not a constant, anyone can change them
- global variables can be changed concurrently from multiple threads
  - $\Rightarrow$ unexpected, unpredicted values
  - such a code is called thread-unsafe
- **do not use global variable, when not really needed!**
  - better and safer to give a function the needed context information as parameters

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Watch the Global Variables

- global variable not explicitly assigned values, are initialized with 0
- global variables are visible from any function of your program
  - ⇒ if not a constant, anyone can change them
- global variables can be changed concurrently from multiple threads
  - ⇒ unexpected, unpredicted values
  - such a code is called thread-unsafe
- **do not use global variable, when not really needed!**
  - better and safer to give a function the needed context information as parameters

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Watch the Global Variables

- global variable not explicitly assigned values, are initialized with 0
- global variables are visible from any function of your program
  - $\Rightarrow$ if not a constant, anyone can change them
- global variables can be changed concurrently from multiple threads
  - $\Rightarrow$ unexpected, unpredicted values
  - such a code is called thread-unsafe
- **do not use global variable, when not really needed!**
  - better and safer to give a function the needed context information as parameters

Watch the Global Variables

- global variable not explicitly assigned values, are initialized with 0
- global variables are visible from any function of your program
  - $\Rightarrow$ if not a constant, anyone can change them
- global variables can be changed concurrently from multiple threads
  - $\Rightarrow$ unexpected, unpredicted values
  - such a code is called thread-unsafe
- **do not use global variable, when not really needed!**
  - better and safer to give a function the needed context information as parameters

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Watch the Global Variables

- global variable not explicitly assigned values, are initialized with 0
- global variables are visible from any function of your program
  - $\Rightarrow$ if not a constant, anyone can change them
- global variables can be changed concurrently from multiple threads
  - $\Rightarrow$ unexpected, unpredicted values
  - such a code is called thread-unsafe
- **do not use global variable, when not really needed!**
  - better and safer to give a function the needed context information as parameters

Do not use uninitialized local variables!

Avoid using global variables!

Do not use dangling pointers!

Warning Handling

- compiler warns us when something in the compiled program is confusing regarding the use of some variables or functions
  - a logical expression (e.g. a condition in an `if`) using a single '=' instead of two
  - code with no effect
  - usage of uninitialized variables
  - calling `printf()` with a strange or incorrect format string
- when we want to ignore some types of warnings, so not be reported about them, we can specify certain compiling options
  - be aware that too many irrelevant warnings, could hide from us the important ones
- for the release version of a program is recommended the `-Werror` option
  - just to report an error for any warning

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Warning Handling

- compiler warns us when something in the compiled program is confusing regarding the use of some variables or functions
  - a logical expression (e.g. a condition in an `if`) using a single '=' instead of two
  - code with no effect
  - usage of uninitialized variables
  - calling `printf()` with a strange or incorrect format string
- when we want to ignore some types of warnings, so not be reported about them, we can specify certain compiling options
  - be aware that too many irrelevant warnings, could hide from us the important ones
- for the release version of a program is recommended the `-Werror` option
  - just to report an error for any warning

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Warning Handling

- compiler warns us when something in the compiled program is confusing regarding the use of some variables or functions
  - a logical expression (e.g. a condition in an `if`) using a single '=' instead of two
  - code with no effect
  - usage of uninitialized variables
  - calling `printf()` with a strange or incorrect format string
- when we want to ignore some types of warnings, so not be reported about them, we can specify certain compiling options
  - be aware that too many irrelevant warnings, could hide from us the important ones
- for the release version of a program is recommended the `-Werror` option
  - just to report an error for any warning

Warning Handling

- compiler warns us when something in the compiled program is confusing regarding the use of some variables or functions
  - a logical expression (e.g. a condition in an `if`) using a single '=' instead of two
  - code with no effect
  - usage of uninitialized variables
  - calling `printf()` with a strange or incorrect format string
- when we want to ignore some types of warnings, so not be reported about them, we can specify certain compiling options
  - be aware that too many irrelevant warnings, could hide from us the important ones
- for the release version of a program is recommended the `-Werror` option
  - just to report an error for any warning

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Warning Handling

- compiler warns us when something in the compiled program is confusing regarding the use of some variables or functions
  - a logical expression (e.g. a condition in an `if`) using a single '=' instead of two
  - code with no effect
  - usage of uninitialized variables
  - calling `printf()` with a strange or incorrect format string
- when we want to ignore some types of warnings, so not be reported about them, we can specify certain compiling options
  - be aware that too many irrelevant warnings, could hide from us the important ones
- for the release version of a program is recommended the `-Werror` option
  - just to report an error for any warning

Warning Handling

- compiler warns us when something in the compiled program is confusing regarding the use of some variables or functions
  - a logical expression (e.g. a condition in an `if`) using a single '=' instead of two
  - code with no effect
  - usage of uninitialized variables
  - calling `printf()` with a strange or incorrect format string
- when we want to ignore some types of warnings, so not be reported about them, we can specify certain compiling options
  - be aware that too many irrelevant warnings, could hide from us the important ones
- for the release version of a program is recommended the `-Werror` option
  - just to report an error for any warning

Warning Handling

- compiler warns us when something in the compiled program is confusing regarding the use of some variables or functions
  - a logical expression (e.g. a condition in an `if`) using a single '=' instead of two
  - code with no effect
  - usage of uninitialized variables
  - calling `printf()` with a strange or incorrect format string
- when we want to ignore some types of warnings, so not be reported about them, we can specify certain compiling options
  - be aware that too many irrelevant warnings, could hide from us the important ones
- for the release version of a program is recommended the `-Werror` option
  - just to report an error for any warning

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Warning Handling

- compiler warns us when something in the compiled program is confusing regarding the use of some variables or functions
  - a logical expression (e.g. a condition in an `if`) using a single '=' instead of two
  - code with no effect
  - usage of uninitialized variables
  - calling `printf()` with a strange or incorrect format string
- when we want to ignore some types of warnings, so not be reported about them, we can specify certain compiling options
  - be aware that too many irrelevant warnings, could hide from us the important ones
- for the release version of a program is recommended the `-Werror` option
  - just to report an error for any warning

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Warning Handling

- compiler warns us when something in the compiled program is confusing regarding the use of some variables or functions
  - a logical expression (e.g. a condition in an `if`) using a single '=' instead of two
  - code with no effect
  - usage of uninitialized variables
  - calling `printf()` with a strange or incorrect format string
- when we want to ignore some types of warnings, so not be reported about them, we can specify certain compiling options
  - be aware that too many irrelevant warnings, could hide from us the important ones
- for the release version of a program is recommended the `-Werror` option
  - just to report an error for any warning

Outline

What We Talked About!

- basic aspects related to C programs
- getting an executable from the source code
- local variables on the stack
- pointers, arrays, data structures
- debugging aspects
- coding recommendations

What We Talked About!

- basic aspects related to C programs
- getting an executable from the source code
- local variables on the stack
- pointers, arrays, data structures
- debugging aspects
- coding recommendations

What We Talked About!

- basic aspects related to C programs
- getting an executable from the source code
- local variables on the stack
- pointers, arrays, data structures
- debugging aspects
- coding recommendations

What We Talked About!

- basic aspects related to C programs
- getting an executable from the source code
- local variables on the stack
- pointers, arrays, data structures
- debugging aspects
- coding recommendations

What We Talked About!

- basic aspects related to C programs
- getting an executable from the source code
- local variables on the stack
- pointers, arrays, data structures
- debugging aspects
- coding recommendations

What We Talked About!

- basic aspects related to C programs
- getting an executable from the source code
- local variables on the stack
- pointers, arrays, data structures
- debugging aspects
- coding recommendations

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Lessons Learned

- local variables are allocated on stack
  - their initial value is unknown
  - stack could be corrupted due to overflowing local arrays
- dynamically allocated memory should be released (i.e. freed)
- pointers give us direct access to application's memory
  - take care at pointer arithmetic!
  - take care of memory overflowing or corruption!
  - do not use uninitialized pointers!
- array names are pointers to where the array is stored
- debugging and testing have same importance as writing code

Lessons Learned

- local variables are allocated on stack
  - their initial value is unknown
  - stack could be corrupted due to overflowing local arrays
- dynamically allocated memory should be released (i.e. freed)
- pointers give us direct access to application's memory
  - take care at pointer arithmetic!
  - take care of memory overflowing or corruption!
  - do not use uninitialized pointers!
- array names are pointers to where the array is stored
- debugging and testing have same importance as writing code

Lessons Learned

- local variables are allocated on stack
  - their initial value is unknown
  - stack could be corrupted due to overflowing local arrays
- dynamically allocated memory should be released (i.e. freed)
- pointers give us direct access to application's memory
  - take care at pointer arithmetic!
  - take care of memory overflowing or corruption!
  - do not use uninitialized pointers!
- array names are pointers to where the array is stored
- debugging and testing have same importance as writing code

Lessons Learned

- local variables are allocated on stack
  - their initial value is unknown
  - stack could be corrupted due to overflowing local arrays
- dynamically allocated memory should be released (i.e. freed)
- pointers give us direct access to application's memory
  - take care at pointer arithmetic!
  - take care of memory overflowing or corruption!
  - do not use uninitialized pointers!
- array names are pointers to where the array is stored
- debugging and testing have same importance as writing code

Lessons Learned

- local variables are allocated on stack
  - their initial value is unknown
  - stack could be corrupted due to overflowing local arrays
- dynamically allocated memory should be released (i.e. freed)
- pointers give us direct access to application's memory
  - take care at pointer arithmetic!
  - take care of memory overflowing or corruption!
  - do not use uninitialized pointers!
- array names are pointers to where the array is stored
- debugging and testing have same importance as writing code

Lessons Learned

- local variables are allocated on stack
  - their initial value is unknown
  - stack could be corrupted due to overflowing local arrays
- dynamically allocated memory should be released (i.e. freed)
- pointers give us direct access to application's memory
  - take care at pointer arithmetic!
  - take care of memory overflowing or corruption!
  - do not use uninitialized pointers!
- array names are pointers to where the array is stored
- debugging and testing have same importance as writing code

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Lessons Learned

- local variables are allocated on stack
  - their initial value is unknown
  - stack could be corrupted due to overflowing local arrays
- dynamically allocated memory should be released (i.e. freed)
- pointers give us direct access to application's memory
  - take care at pointer arithmetic!
  - take care of memory overflowing or corruption!
  - do not use uninitialized pointers!
- array names are pointers to where the array is stored
- debugging and testing have same importance as writing code

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Lessons Learned

- local variables are allocated on stack
  - their initial value is unknown
  - stack could be corrupted due to overflowing local arrays
- dynamically allocated memory should be released (i.e. freed)
- pointers give us direct access to application's memory
  - take care at pointer arithmetic!
  - take care of memory overflowing or corruption!
  - do not use uninitialized pointers!
- array names are pointers to where the array is stored
- debugging and testing have same importance as writing code

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Lessons Learned

- local variables are allocated on stack
  - their initial value is unknown
  - stack could be corrupted due to overflowing local arrays
- dynamically allocated memory should be released (i.e. freed)
- pointers give us direct access to application's memory
  - take care at pointer arithmetic!
  - take care of memory overflowing or corruption!
  - do not use uninitialized pointers!
- array names are pointers to where the array is stored
- debugging and testing have same importance as writing code

Lessons Learned

- local variables are allocated on stack
  - their initial value is unknown
  - stack could be corrupted due to overflowing local arrays
- dynamically allocated memory should be released (i.e. freed)
- pointers give us direct access to application's memory
  - take care at pointer arithmetic!
  - take care of memory overflowing or corruption!
  - do not use uninitialized pointers!
- array names are pointers to where the array is stored
- debugging and testing have same importance as writing code

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA