UE Unix/C
L3 Informatique

$whoami

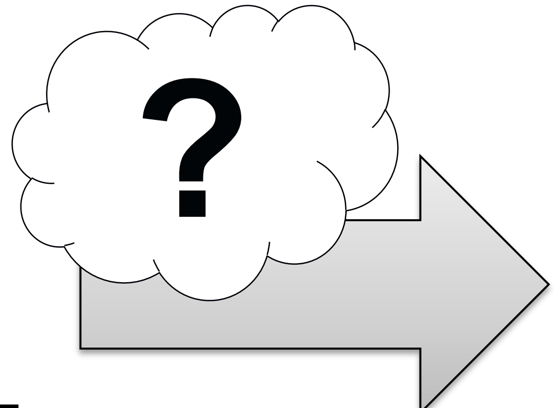
- Name: Angeliki
- Surname: Kritikakou
- Email: angeliki.kritikakou@univ-rennes1.fr
- Position: Associate Professor – Researcher
  - Teaching: ISTIC mainly L3info
  - Research: CAIRN team, INRIA-IRISA
- Research Interests:
  - Computer Architecture, Embedded Systems, Mixed-critical Systems, Fault Tolerance, Low energy design, Task Scheduling and Allocation, Memory Management, Design Space Exploration

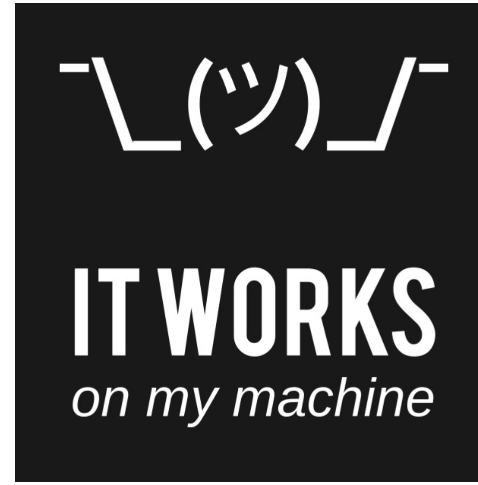
Organisation 2020-2021

- CM: Angeliki Kritikakou
  - 8 main courses
  - Interactive course: Kahoot Tests
- TD: Angeliki Kritikakou
  - 2 exercise courses
- TP: A. Kritikakou, J. C. Engel, A. Maddi
  - Final lab exam:
    - Independent
    - 03 December 2020 (Last Lab)
    - Duration: 1h (slot 14:30 or 15:30)
- Note: 1/2 Theoretical exam + 1/2 Labs
  - Multiple choice questions

Introduction

Creation…From a text file to an executable





foo.c

```c
#include <stdio.h>
int main()
{
  int i = 17;
  int j = 21;
  int k = i+j;
  printf("i+j=%d\n", k);
  return 0;
}
                    1,1            Top
```

```
$ ./foo
i+j=38
$
```

Layered view of computer organisation

L5	**Problem-oriented language level**

L5: Problem-oriented language

- File `foo.c` is written with the syntax of C language

```
#include <stdio.h>
int main()
{
  int i = 17;
  int j = 21;
  int k = i+j;
  printf("i+j=%d\n", k);
  return 0;
}
                    1,1           Top
```

Layered view of computer organisation

L5 — **Problem-oriented language level**

↓ Translation (Compilation)

L4 — **Assembly language level**

L4: Assembly Language

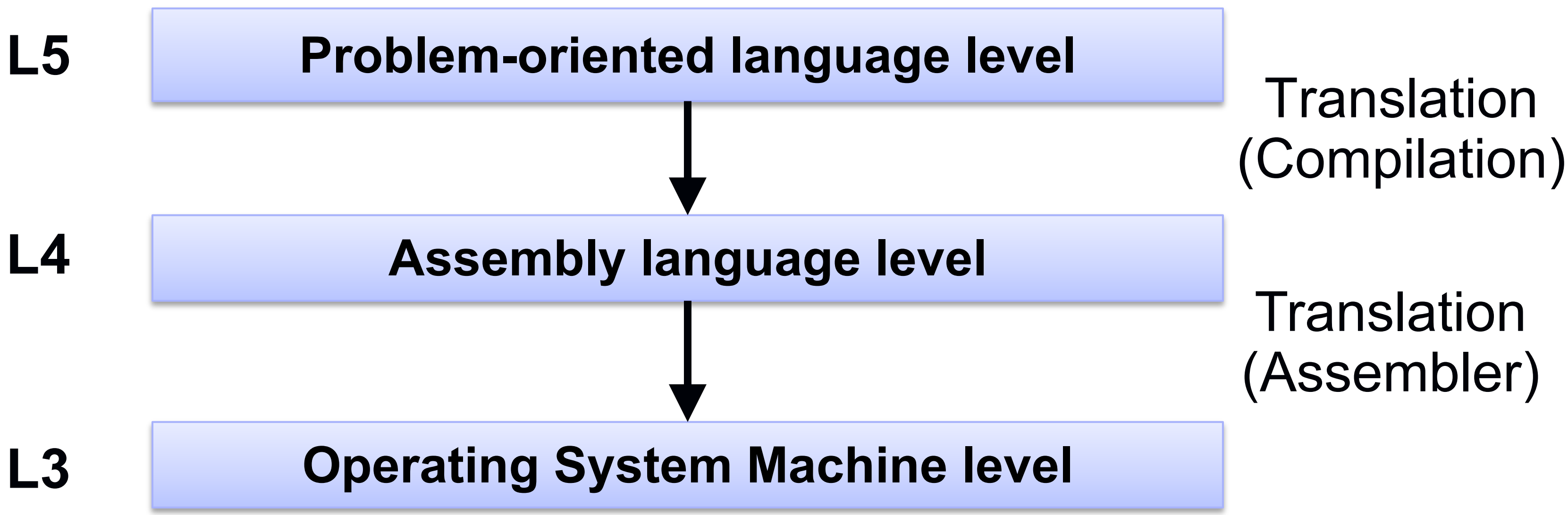
- foo.s:

Result of compilation

```
$ gcc –S foo.c
```

```
        .file   "foo.c"
        .section        .rodata
.LC0:
        .string "i+j=%d\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $17, -12(%rbp)
        movl    $21, -8(%rbp)
        movl    -8(%rbp), %eax
        movl    -12(%rbp), %edx
        addl    %edx, %eax
        movl    %eax, -4(%rbp)
        movl    $.LC0, %eax
        movl    -4(%rbp), %edx
        movl    %edx, %esi
        movq    %rax, %rdi
        movl    $0, %eax
        call    printf
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3"
        .section        .note.GNU-stack,"",@progbits
                                                  1,2-11        All
```

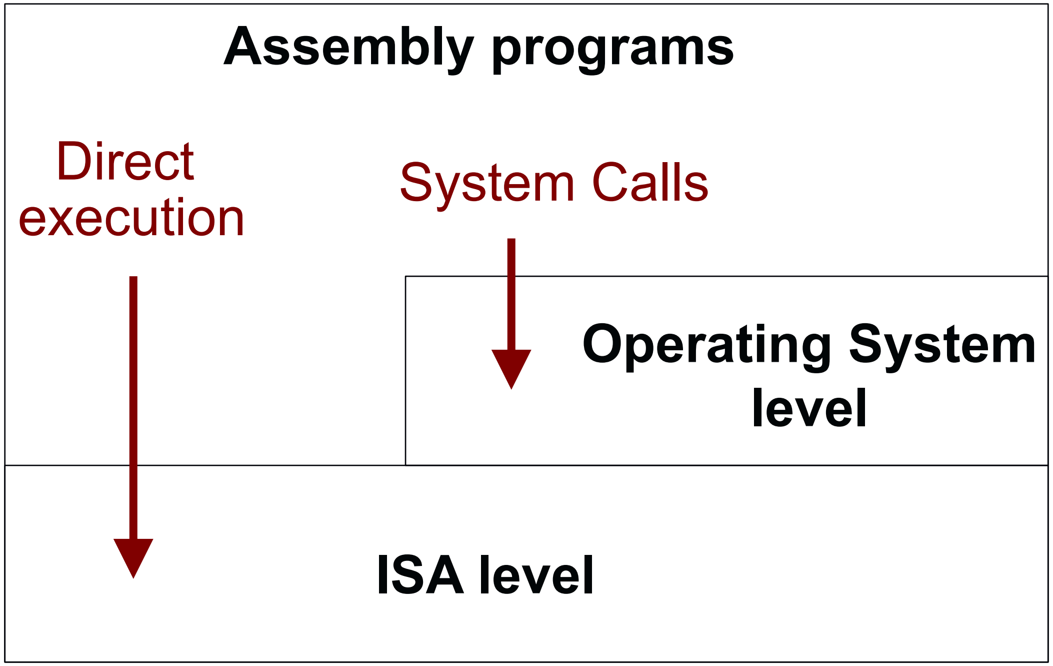
Layered view of computer organisation



L3: Operating system

- `foo`: Result of assembler
- Executable:
  - Operating System
  - Instruction Set Architecture (ISA)

```
$ gcc –c foo.c –o foo
```



Layered view of computer organisation

UNIVERSITÉ DE RENNES 1

L5 — **Problem-oriented language level**

↓ Translation (Compilation)

L4 — **Assembly language level**

↓ Translation (Assembler)

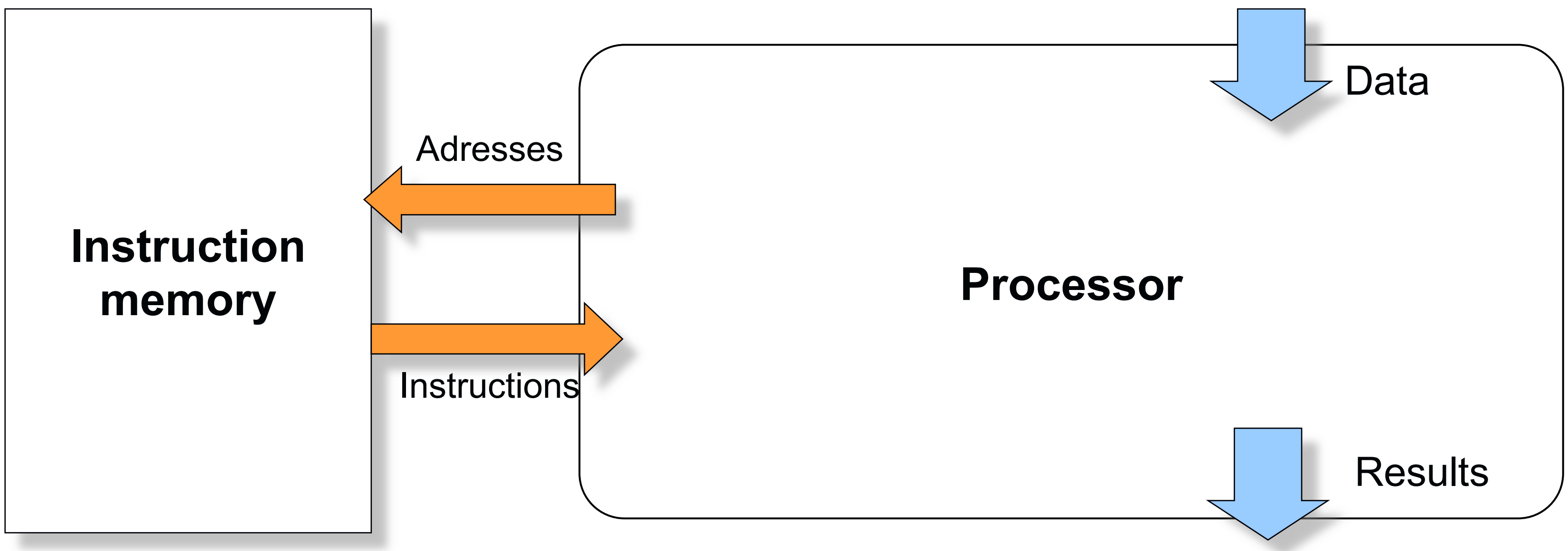
L3 — **Operating System Machine level**

↓ Partial interpretation (O.S.)

L2 — **Instruction Set Architecture level**

L2: Instruction Set Architecture

- `foo:`
  - stored in the instruction memory
  - consists of several machine code instructions
  - depend on the ISA of the used processor (X86, ARM, SPARC, NIOS2…)



Layered view of computer organisation

L5 **Problem-oriented language level**

↓ Translation (Compilation)

L4 **Assembly language level**

↓ Translation (Assembler)

L3 **Operating System Machine level**

↓ Partial interpretation (O.S.)

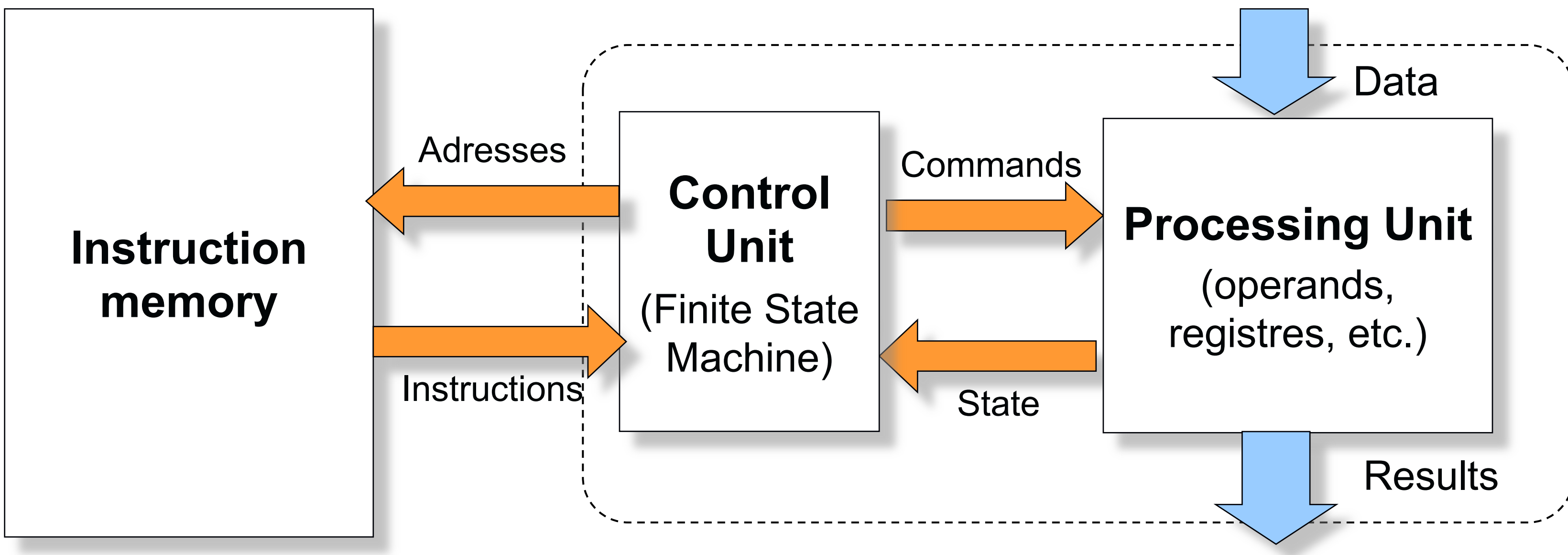
L2 **Instruction Set Architecture level**

↓ Interpretation (micro-program) or direct execution

L1 **Microarchitecture level**

L1: Micro architecture

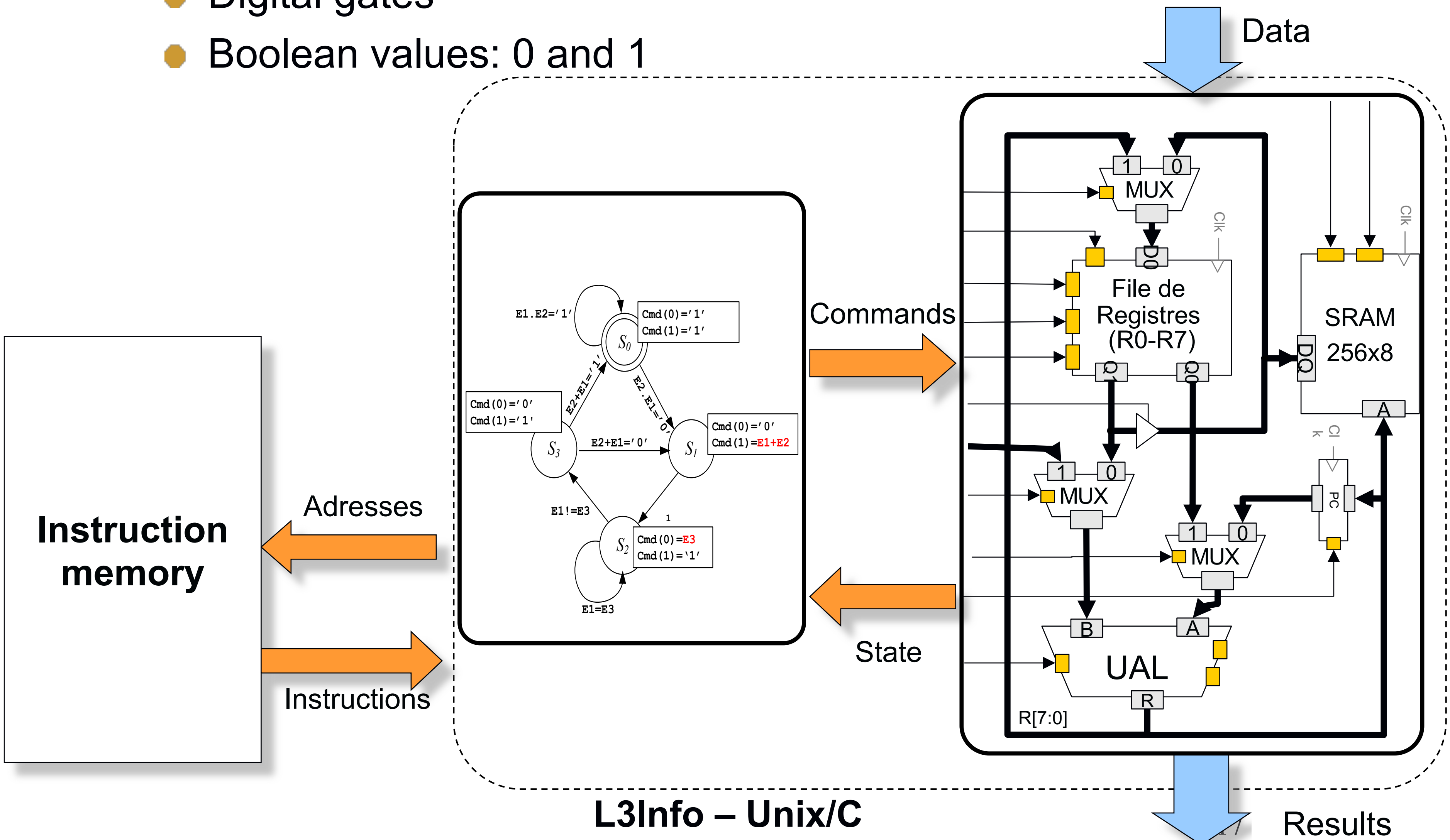
- Each ISA instruction is implemented by a set of micro-operations (micro-ops)
- Micro-ops:
  - Created by the Control Unit
  - Describe how the Processing Unit has to behave to execute the ISA instruction



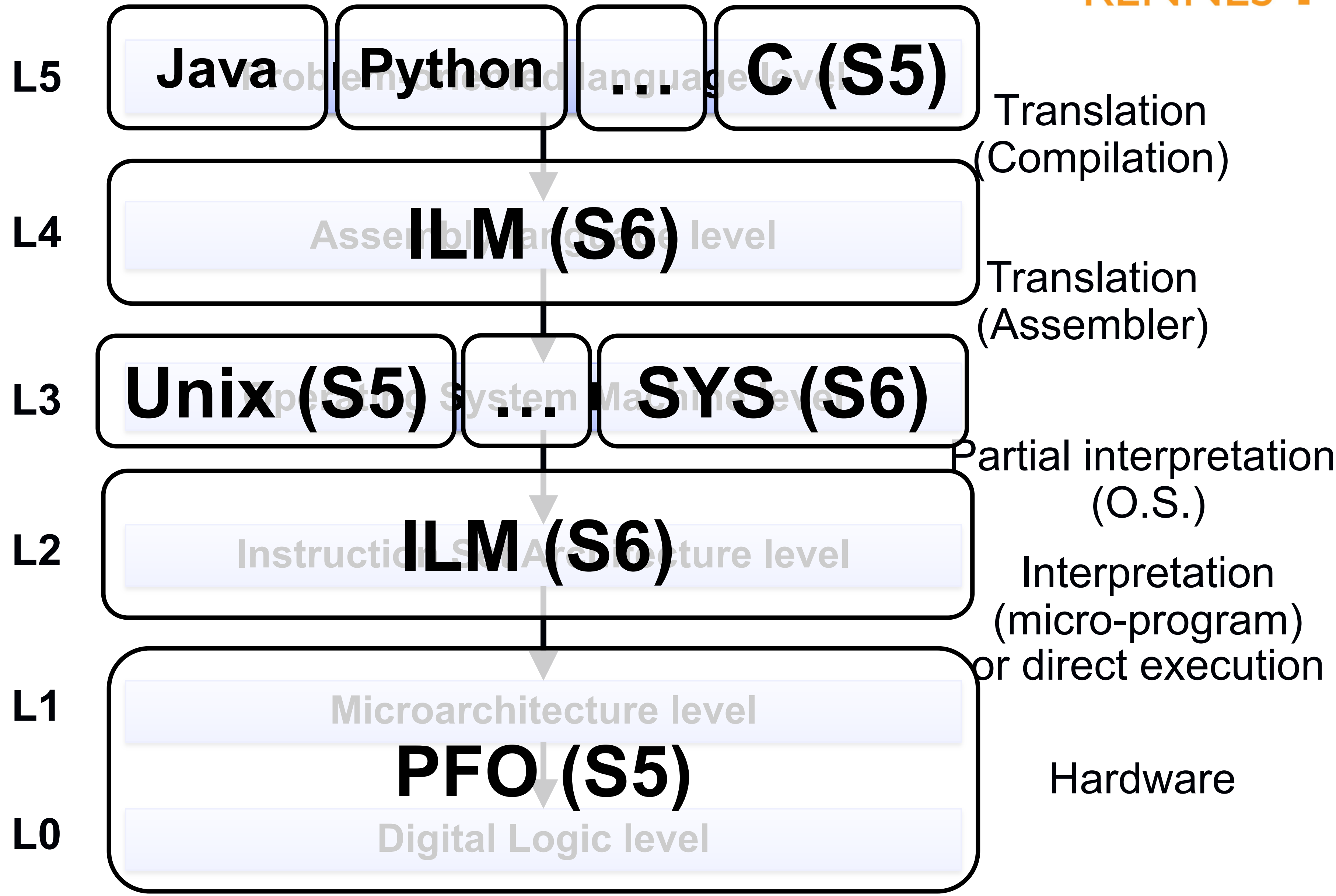
Layered view of computer organisation

L5	Problem-oriented language level	
	↓	Translation (Compilation)
L4	Assembly language level	
	↓	Translation (Assembler)
L3	Operating System Machine level	
	↓	Partial interpretation (O.S.)
L2	Instruction Set Architecture level	
	↓	Interpretation (micro-program) or direct execution
L1	Microarchitecture level	
	↓	Hardware
L0	Digital Logic level	

- Hardware is responsible for everything!
  - Digital gates
  - Boolean values: 0 and 1



Layered view of computer organisation



Outline

- Part 1: Linux and Shell scripting:
  - 2 CM, 1 TD, 1 TP

- Part 2: C Language
  - 6 CM, 1 TD, 8 TP

Kahoot time ! (2 Q)

①	**Goto:**	**https://kahoot.it/**
②	**Insert:**	**PIN number**
③	**Insert:**	**a nickname**

Part A:
Linux and Shell scripting

Operating system

- Manages:
  - Process: creation, execution, termination, …
  - Memory: isolation, paging, virtual memory, …
  - Files: creation, access, permissions, …
  - Peripherals: usage, common interface, …

- Provides a higher layer abstraction to interact with hardware peripherals
  - System calls

- Focus: Utilisation of the Operating System
  - Linux

Linux

- First prototype in 1991 (post to “comp.os.minix”):
  - Linus Torvalds (Finland student – age 21):

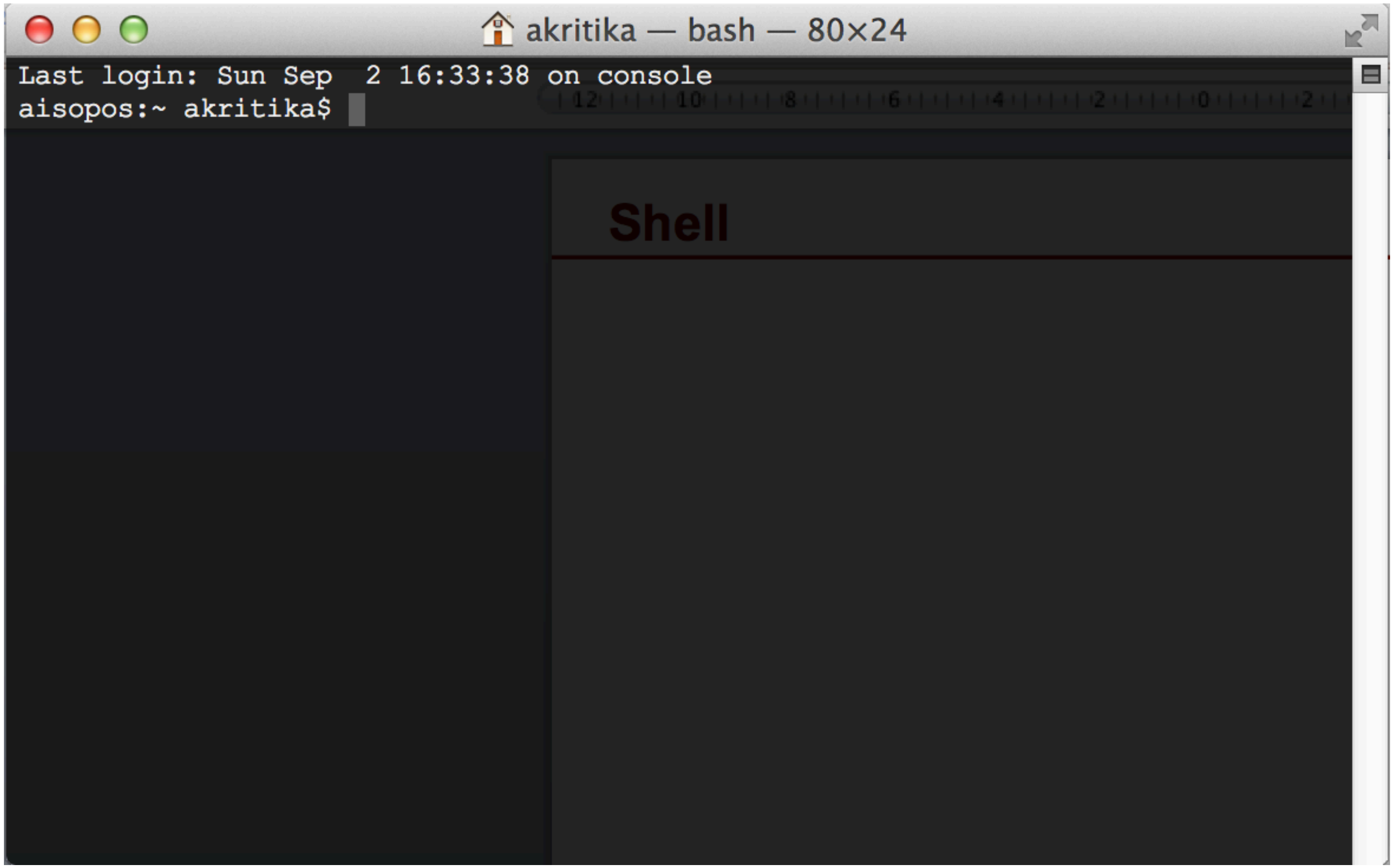
  “Hello everybody out there using minix –

  I am doing a (free) operating system (just a hobby, won’t be big and professional like gnu) for 386 (486) AT clones…”

- Naming:
  - Linus: “Freax” → “Free”, “Freak”, “x” (from Unix)
  - Ari Lemmke (server admin): Renamed to “Linux”

- Free and open source operating system
  - > 18.000.000 lines of source code

Interface with the system: The Shell



Shell

- Command Line Interface (CLI)
  - Input: The commands you want to execute (use the keyboard)
  - Output: Shell passes the commands to the OS, which executes them, and provides the result (usually to the screen)
  - Scripts: Combination of commands
  - Variables
- Interaction with the Shell: Terminal emulators
  - Konsole, Xterm, Gnome-terminal…
- Several shells are available
  - **`sh`** (Bourne shell), **`bash`** (Bourne shell), **`csh, zsh, ksh`** …
- Each user has a preselected shell
  - Selection in the file: /etc/passwd
  - Command to change the shell: **`chsh`**

Directories

- Home Directory ($HOME)
  - Each user has a home directory connected with his name
  - Name: 2363415
  - Home directory: /private/student/5/25/2363415 or ~/2363415

- Working Directory ($PWD)
  - Current directory
  - When connected: Working directory = Home directory

- Current directory: .
- Parent directory: ..
- Root directory: /
- User home directory: ~

Paths

- Absolute
  - From root directory / : /private/student/5/25/2363415

- Relative
  - From working directory:
    - Assume the following working directory:

      /private/student/5/25/2363415
    - You can access a file in another directory /private/student/5/25/2363414 by:

      ../2363414

Wildcard characters

- Special characters which can be used in combination with commands

- One and only one character: `?`
  - Ex: `ls /etc/rc.????`

- From 0 to many characters: `*`
  - Ex: `ls /etc/rc.*`

- Specific characters : `[…]`
  - Ex: `ls [abc]oo.c`

  `aoo.c, boo.c, coo.c`

How to treat them as normal (by the Shell)

- Preserve the literal value of next character: ****

- Preserve the literal value of a character sequence**:**
  '…' (ticks)
- As before, except for the characters **$** and **`**:
  "…" (quotes)

- Command replacement: Execution of the **command** and replace with what it has been produced as output
  `command` (back ticks)
  $(command)

Useful information

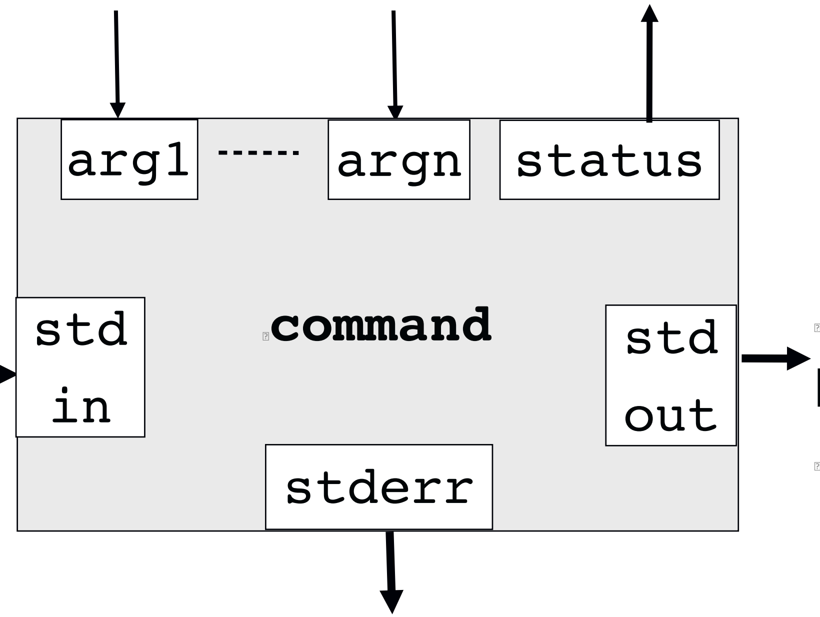
- tab key:
  - One tab key: automatically fills the word you are started typing, if exists
  - Two tab keys: Prints all the existing words in that start with the letters that you have typed
- **`clear`** command: Cleans the screen in the shell
- Up arrow key: Shows the previous command
- **`history`** command: Show all previous commands
- **`!!:`** Executes the previous command
- Do not use special characters in the filenames, that include also the space character and the hyphen character (-).
- Put as default language **ENGLISH**, the French translation in not complete … (LANG=EN)

Structure of a command

`command arg1 arg2 … argn`

The arguments of a command are the parameters passed during its call

The exit status is the value returned after the execution of the command to inform about normal or not execution



The standard input is read from keyboard (Number 0 is `stdin` Shortcut: `-`)

The standard output is print to the screen (Number 1 is `stdout`)

The standard error is by default printed to the screen and it is used by the commands to print the error messages (Number 2 is `stderr`)

Shell Commands

- Directory management
- File management
- Redirections
- Process management
- Shell programming: Scripts

- This course does NOT give an exhaustive list of commands and a complete command description → Use:

  - documentation
  - **`man`** or **`info`** command
  - **`apropos`** command

Directory management

- Print the full pathname of the working directory: **`pwd`**
- Change directories: **`cd`**
  - **`cd`** or **`cd ~`**: Go to home directory
  - **`cd path`**: Go to directory indicated by **`path`**
- Creation/Delete directories
  - **`mkdir dirname`**: Creates a directory named **`dirname`**
  - **`rmdir dirname`**: Deletes the directory named **`dirname`** (empty)
  - **`rm -r dirname`**: Deletes the directory named **`dirname`** recursively (subdirectories) (not empty)
  - **NO TRASHBIN !**
- Rename: **`mv dirsource dirdest`**
- Hierarchical copy (subdirectories): **`cp -r dirsource dirdest`**

Listing

- **`ls`**: Lists the files in the working directory
- **`ls dirname`**: Lists the content of directory **`dirname`**

```
angie@achilleas:~/Entertainment$ ls
movies  Music  photos
```

- Some parameters passed as arguments:
  - **`-l`**: Use a long listing format (details)

```
angie@achilleas:~/Entertainment$ ls -l
total 12
drwxrwxr-x 8 angie angie 4096 Φεβ   1  2015 movies
drwxr-xr-x 3 angie angie 4096 Φεβ   1  2015 Music
drwxrwxr-x 5 angie angie 4096 Οκτ  20  2015 photos
```

  - **`-a`**: Not ignore entries starting with **`.`** (hidden folders/files)

```
angie@achilleas:~/Entertainment$ ls -al
total 28
drwxrwxr-x  5 angie angie  4096 Δεκ   4  2014 .
drwxr-xr-x 48 angie angie 12288 Μάι  30 08:53 ..
drwxrwxr-x  8 angie angie  4096 Φεβ   1  2015 movies
drwxr-xr-x  3 angie angie  4096 Φεβ   1  2015 Music
drwxrwxr-x  5 angie angie  4096 Οκτ  20  2015 photos
```

File management

- Create file by:
  - Update access & modification time: **`touch filename`**
- Delete file: **`rm filename`**
- Rename: **`mv filesource filedest`**
- Copy:
  - **`cp filesource filedest`**
  - **`cp filesource dirdest`**
- Remove non-directory suffix part from a filename: **`dirname`**
- Remove the directory part suffix from a filename: **`basename`**
- Count the number of words of a file: **`wc –w filename`**

File management

- Print the contents of a file:
  - **`more filename`**: Prints page by page the content of file **`filename`** on the `stdout` (Q: Exit from **`more`**)
  - **`less filename`**: Similar to more, but it allows to go up/down to the pages (Q: Exit from **`less`**)
  - **`head –n 42 filename`**: Print the first 42 lines of **`filename`**
  - **`tail –n 42 filename`**: Print the last 42 lines of **`filename`**
- Concatenate files (create and view files):
  - **`cat filename1 … filenameN`** (print context of files at `stdout`)
  - **`cat`**: Print at **`stdout`** whatever is tapped at **`stdin`** (**`cntrl+D`** indicates **`EOF`**)
  - **`-`** : Uses what is tapped with the keyboard instead of an existing file
- Difference between two files:
  - **`diff filename1 filename2`**

Directory/File management: Examples

Kahoot time !
(4 Q)

Directory/File permissions

```
r w x   r w x   r w x
_ _ _   _ _ _   _ _ _
owner   group   others
```

- Change permissions: **`chmod`**

  `chmod [who] op permissions [,op permissions ]* name`

- Who:
  - Owner (u)
  - Group (g)
  - Others (o)
  - Nothing (all)
- Op:
  - Add (+)
  - Remove (-)
  - Define (=)
- Permissions:
  - Read or List (r)
  - Write (w)
  - Execute or Access(x)

Kahoot time !
(3 Q)

File management: Search files

- Search files based on criteria (print path):

`find directory [criteria]`

  - The search is performed to directory (recursive to subdirectories)
  - Apply a command at the found files: option **`–exec cmd {} \;`**
    - Apply the command **`cmd`** in each found file (it replaces the **`{}`**)
    - End of the command: **`\;`**

- Some search criteria:
  - **`-name filename`**: based on the filename (**`-iname`**: ignore case)
  - **`-user username`**: based on the owner
  - **`-size number[c|k|M]`**: based on size [B, KB, MB]
  - **`-type [d|f|l]`**: based on the type [directory, file, link]
  - **`-mtime x`**: based on time of modification (less than x days)
  - ….

- Criteria can be combined: **`-o –a -not`**

Kahoot time !
(2 Q)

File management: Search in files

- Print all lines of the files of the directory that contain a word: **`grep string directory`**
  - The string can be expressed as a regular expression
    - Whatever character : **`.`**
    - Repetition of pattern: *
    - Interval of characters: **`[`*`x`*`-`*`y`*`]`**
  - Return value: `0` not found, else `!=0`

- Examples:
  - **`grep -r test ./`**
  - **`grep "m.*t" filename`**
  - **`grep "toto[1-3]" filename`**

Redirections

- Redirection of standard input (**`stdin`**):
  - Use the content of *a **file*** as input to a command (instead of keyboard)
    - **`command < filename`**
- Redirection of standard output (**`stdout`**)
  - Print the output of a command into *a **file*** (instead of screen)
    - **`command > filename`** (overwrite old content of file)
    - **`command >> filename`** (append to the content of file)
- Pipe: redirect standard output to standard input of ***commands***
  - Use the output of a **command** as input to another **command**
  - A subshell is created to execute command_2
    - **`command_1 | command_2`**

Redirections: Examples

- `command`

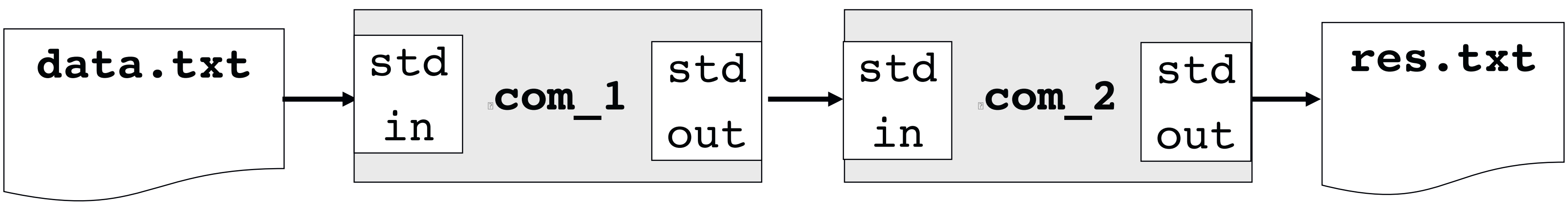
- `command > res.txt`

- `command < data.txt`

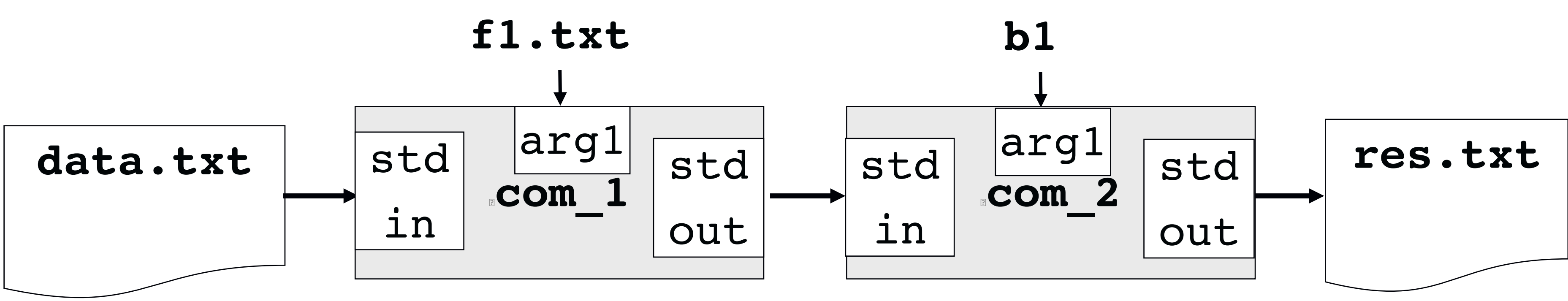
Redirections: Examples

- `command < data.txt > res.txt`

- `com_1 < data.txt | com_2 > res.txt`



- `com_1 f1.txt < data.txt | com_2 b1 > res.txt`



Kahoot time !
(3 Q)

Command implementation

- Builtin
  - Implemented in the code of the shell
  - They affect the internal state of the shell (ex. **`cd`**)
- External
  - Implemented as a separate code
  - Shell interprets the command as a request to load and run the program that implements the command
  - Stored in /bin

Builtin Commands

Command	Description	Example
`cd`	Change directory	`cd ..`
`declare`	Define variable	`declare myvar`
`echo`	Print to standard output (stdout)	`echo hello`
`exec`	Replace bash with another process	`exec ls`
`exit`	Quit bash	`exit`
`export`	Define global variable	`export myvar=1`
`history`	Print the history of commands	`history`
`kill`	Send signal to a process	`kill 1121`
`let`	Compute arithmetic operation	`let myvar=3+5`

Builtin Commands

Command	Description	Example
`local`	Define local variable	`local myvar=5`
`pwd`	Print working directory	`pwd`
`read`	Read from standard input (`stdin`)	`read myvar`
`readonly`	Locks a variable	`readonly myvar`
`return`	End of a function and return a value	`return 1`
`set`	Print variables	`set`
`shift`	Move parameters	`shift 2`
`test`	Test a condition	`test –d temp`
`trap`	Follow a signal	`trap "echo signal" 3`

Process management: Execution

- Process: The running instance of a program, e.g.:
  - Execution of a command
  - Execution of your program
- Foreground execution: Interactive processes
  - Connected to a shell terminal
  - User can send input
- Background execution: Non-Interactive or automated processes
  - Not connected to a terminal
  - User cannot send input until the process is moved to the foreground

Process management: Changing grounds

- Run in Background: **&**:
  - Ex: **`gedit &`**

- From FG to BG:
  - Ex: **`gedit`**

    Press CTRL + Z (puts process in idle state)

    `bg`

- From BG to FG:
  - Ex: **`fg %jobid`**

  (jobid is not PID. It is a unique number given by the shell, e.g. when process goes to BG)

Process management: Active processes

- Print the catalogue of the running processes: **`ps`**
  - **`-a:`** Select all processes associated with a terminal.
  - **`PID:`** Unique identity of the process
  - **`TTY:`** Console created the process
  - **`TIME:`** Total execution time (sec)
  - **`CMD:`** Command executed

```
angie@achilleas:~/Downloads$ ps
  PID TTY          TIME CMD
12190 pts/1    00:00:00 bash
14414 pts/1    00:00:00 vi
14447 pts/1    00:00:00 evince
14454 pts/1    00:00:00 ps
```

Process management: Control execution

- End a running process
  - Global method:
    - **`kill PID`**
  - Foreground execution in current shell (through SIGNALS)
    - **CTRL +** **`C`** (Interrupt)
    - **CTRL +** **`D (EOF)`**
    - **CTRL +** **`\`** (Quit)
- Pause a process
  - **CTRL +** **`Z`**
- Resume
  - Foreground (**`fg`**)
  - Background (**`bg`**)

Shell Variables

- Initial values: defined in system and user configuration file
- Use uppercase by convention
- Definition of variables based on the type (bash shell)
  - String variables: **`MYVAR="value"`** **`(NO SPACES around =)`**
  - Integer variables: **`declare –i MYVAR`**
  - Constant variables: **`readonly DATA=value`**
  - Array variables: **`declare –a MYARRAY MYARRAY[0]="one"; MYARRAY[1]=5`**
- Use of variables: **`$`**
  - E.g. **`$MYVAR`**
- Lifetime of variables:
  - End of session – termination of shell
  - Deleted by user: **`unset VARNAME`**

Shell Variables: Scope

- Global: Public variable seen by all subshells
  - Set export attitude for the variable: **`export`**
  - Print global variables: **`env, printenv`**

- Local variables: Seen by current shell
  - Print list of local and global variables: **`set`**

Shell Variables: Scope example

```
angie@achilleas:~$ myvar="hello"
angie@achilleas:~$ set | grep myvar
myvar=hello
angie@achilleas:~$ echo "$$"
12190
angie@achilleas:~$ bash
angie@achilleas:~$ echo "$$"
15752
angie@achilleas:~$ set | grep myvar
angie@achilleas:~$ env | grep myvar
angie@achilleas:~$ exit
exit
angie@achilleas:~$ export myvar="hello"
angie@achilleas:~$ bash
angie@achilleas:~$ echo "$$"
15825
angie@achilleas:~$ set | grep myvar
myvar=hello
angie@achilleas:~$ 
```

Predefined shell variables

Variable	Description
`USER`	Name of user account
`HOME`	Personal folder of user
`TERM`	Type of terminal
`SHELL`	Name of shell
`PATH`	List of directories with executable commands
`MANPATH`	List of directories with help information for the commands
`PWD`	Current directory
`OLDPWD`	Previous current directory
`HOSTNAME`	Name of system

Scripts: Structure

- Program consisting of commands:
  - Use an editor:
    - **`vi (vim), emacs, sublime, gedit, …`**
  - 1st line defines the shell to be used:
    - **`#!/bin/bash`** (for bash shell)
  - Write **`program commands`**
  - Commands can be separated by:
    - a new line
    - Semicolon: **`;`**
  - Split a command to 2 or more lines: **`\`** (backslash)
  - For comments: **`#`**
  - Save file with extension **`.sh`**

- The script lines (commands) are interpreted and NOT compiled

Scripts: Execution

- No need for permissions:

`bash path_to_script/myscript.sh`

  - **`bash`** command is executed
  - **`./script.sh`** is the command-line argument.

- Provide eXecute permissions and execute:

`chmod +x myscript.sh`

`./myscript.sh`

  - Shell forks itself and uses a system call (e.g. **`execve`**) to make the OS execute the file in the forked process.
  - OS checks the file's permissions
  - Program loader looks at the file to find how to execute it

  (**`#!`** for scripts and then which interpreter to use: **`/bin/bash`**).

- New process: attention to the variables and functions

Command-line arguments

- Arguments can be passed along with the script execution:
  - **`./myscript.sh arg1 … argN`**

- The arguments are seen as specific variables in the script code:
  - **`$0`**: Name of the script
  - **`$1`**: 1st argument
  - **`$N`**: Nth argument
  - **`$#`**: Number of arguments
  - **`$*`**: All the arguments passed to the script, as a string
  - **`$@`**: Behaves like $* except that when the arguments are quoted, they are broken up properly if there are spaces in them

- Maximum length of arguments is defined by operating system and measured in Kilobytes.
  - **`getconf ARG_MAX`**

Variables

- Initialization: **`nomvar=expresion`**
  - **ATTENTION! No space around =**
  - **`expression:`**
    - Value
    - Reference to a variable
    - **`` `command` ``** (replaced with the result to the `stdout`)

- Use: **`$nomvar`**
  - Print current directory

```
y=`pwd`
echo $y
```

  - Print the first two arguments of a script

```
y="$1 and $2"
echo $y
```

Read input

- We can ask the user to provide some value(s) to the variables
  - **`read var-name1 var-name2 … var-nameN`**
  - Reads a line from the **`stin`** and affects the first value to **`var-name1`**, the second value to **`var-name2`**, etc.
  - Returns 0, if the read was successful

- We can print a message to the user:
  - **`read –p "Enter value:" var`**

- If variable is not specified, then the default is used:
  - **`$REPLY`**

Print to output

- Basic print to `stdout`: **`echo`**
  - Strings
  - Variables
  - Wildcards characters
  - Can have several lines

```
angie@achilleas:~/Entertainment$ echo hello
hello
angie@achilleas:~/Entertainment$ echo *
movies Music photos
angie@achilleas:~/Entertainment$ echo print '@' "user"
print @ user
angie@achilleas:~/Entertainment$ echo 'hell
> o there'
hell
o there
```

Arithmetic operations

- No need to define the variables as integers
- Arithmetic operations with integers
  - `$((…))`: Portability among shells
  - `((…))`

```
$ a=3
$ ((a = a + 1)) ; echo $a          //(4)
$ a=$((a+1))    ; echo $a          //(4)
$ a=$(($a+1)) ; echo $a            //(4)
$ a=a + 1; echo $a                 //a+1
$ a=$a + 1; echo $a                // 3+1
```

  - `let` (without spaces as it is an initialisation)
  - `expr` (with spaces in arguments)

```
$ let a=a+1 ; echo $a              //(4)
$ let a++ ; echo $a                //(4)
$ a=`expr $a + 1` ; echo $a        //(4)
```

Condition expressions: `test`

- A condition expression is:
  - a set of commands that are executed
- The **`test`** command exits with the exit status determined by the **`condition_expression`**
  - **`test condition_expression`**
  - **`[ condition_expression ]`**
  - **`[[ condition_expression ]]`**

  Similar to single brackets, but more powerful

  - **`(( arithmetic condition_expression ))`**
  - **`( command )`**

SPACES ARE REQUIRED !!

Exit status

- Syntax

  `$?`

ATTENTION !!

- Function
  - If the command ends correctly (TRUE): exits with value 0 (**`exit 0`**)
  - If there was an error (FALSE): exits with value 1 (**`exit 1`**)

```
$ true ; echo $?
$ 0
$ false ; echo $?
$ 1
```

- Commands that do nothing except return an exit status of:
  - **`true:`** Return an exit status of zero
  - **`false:`** Return an exit status of one

Kahoot time !
(4 Q)

Condition expressions: test operators

UNIVERSITÉ DE RENNES 1

Operator	Description
`-gt`	Greater than
`-ge`	Greater than or equal
`-lt`	Less than
`-le`	Less than or equal
`-eq` or `=`	Equal (**-eq** for integers and **=** for strings)
`-ne` or `!=`	Not equal (**-ne** for integers and **!=** for strings)
`-n str`	Non-zero size string
`-z str`	Zero size string
`-d dirname`	`dirname` is a directory
`-s filename`	`filename` size is not zero
`-f filename`	`filename` exists
`-r filename`	`filename` exists and we have read access
`-w filename`	`filename` exists and we have write access
`-x filename`	`filename` exists and we have execute access

Boolean control operators

- Syntax (AND): **`-a`** or **`&&`**

`command1 && command2`

- Function:
  - command2 is executed **if, *and only if***, command1 returns an exit status of **zero**.

- Syntax (OR): **`-o`** or ||

`command1 || command2`

- Function:
  - command2 is executed ***if, and only if,*** command1 returns a **non-zero** exit status.

- The exit status of AND and OR lists is the exit status of the **last command executed** in the list
- Syntax (NOT): **`!`**

Kahoot time !
(4 Q)

AND and OR control operators: Example

```
$ true || echo "echo executed"
$ false || echo "echo executed"
echo executed
$ true && echo "echo executed"
echo executed
$ false && echo "echo executed"
$
```

Conditions: `if, elif, else`

- Syntax:

```
if [ condition_expression_1 ] ; then
   1st set of commands
elif [ condition_expression_2 ]
then
   2nd set of commands
else
   3rd set of commands
fi
```

ALL 3 SPACES ARE REQUIRED !!

- Function
  - The **`if`** checks if the **`condition_expression_1`** returns TRUE
    - **`1st set of commands`** is executed
  - Else, if the **`condition_expression_2`** returns TRUE
    - **`2nd set of commands`** is executed
  - Otherwise, **`3rd set of commands`** is executed

Conditions: `if` : Examples

```
#!/bin/bash
read -p "Enter a filename: " filename
if [ ! -w "$filename" ]; then
  echo "File $filename is not writable”
  exit 1
elif [ ! -r "$filename" ]; then
  echo "File $filename is not readable”
  exit 1
else
  echo "File $filename is writable and readable"
fi
```

Conditions: `if` : Examples

```
#!/bin/bash
TMPFILE="diff.out"
diff $1 $2 > $TMPFILE
if [ ! -s "$TMPFILE" ]; then
        echo "Files are the same"
else
        more $TMPFILE
fi
if [ -f "$TMPFILE" ]; then
        rm $TMPFILE
fi
```

Conditions: case

- Syntax:

```
case $variable in
   pattern 1 )
         1st set of commands ;;
   pattern 2 | pattern 3 )
         2nd set of commands ;;
   *)
         Nth set of commands ;;
esac
```

- Function
  - If the value of **`variable`** is **`pattern 1`**, the 1st set of commands is executed, if it is **`pattern 2`**, the 2nd set of commands is executed etc.
  - Otherwise (`*`) → Nth set of commands

Conditions: `case`: Example

```
#!/bin/bash
read -p "Enter a command: " command
case $command in
all | ALL )
        echo "Display all files..."
        ls -la ;;
list | LIST)
        echo "Display all non-hidden files..."
        ls -ll ;;
*)
        echo "Invalid choice"
esac
```

Iterations: `for`

- Syntax:

```
for variable in list_of_values
do
   set of commands
done
```

- Function
  - In each iteration **`variable`** takes a value from the **`list_of_values`**
    - The **`set of commands`** is execute for this value
  - **`list_of_values`** can be:
    - Strings: **`L3info L3miage M1info`**
    - Numbers: counting: **`{1..10..2}`**
      enumerating: **`1 3 5`**
    - Command line arguments: **`$*`**
    - File names: **`filename1 filename2`**
    - command output: **`$(ls /tmp/*)`**

Iterations: `for`: Examples

```
#!/bin/bash
for file in f1 f2 f3
do
    cp $file $1/$file
done
```

```
#!/bin/bash
for i in 6 3 1 2
do
    echo $i
done | sort -n
```

Iterations: `while`

- Syntax:

```
while
   [ condition_expression ]
do
   set of commands
done
```

- Function
  - **As long as** the **`condition_expression`** returns TRUE
    - The **`set_of_commands`** is executed
  - Otherwise, the iteration ends

Iterations: `while`: Examples

```
#!/bin/bash
i=1
while [ $i -lt 10 ]
do
        echo $i
        ((i++))
done
```

```
#!/bin/bash
while true
do
        echo "alive..."
        sleep 3s
done
```

Iterations: `until`

- Syntax:

```
until
   [ condition_expression ]
do
   set of commands
done
```

- Function
  - Execute 1st set of commands
  - **As long as** the **`conditon_expression`** commands is NOT true (while **`consition_expression`** is false)
    - The **`set_of_commands`** is executed
  - Otherwise the iteration ends

Iterations: `until`: Examples

```
#!/bin/bash
i=1
until [ $i -ge 10 ]
do
        echo $i
        ((i++))
done
```

Controlling loop commands

- Stop with current loop iteration: **`break`**
- Continue with the next iteration: **`continue`**
- Terminate script: **`exit`**

Controlling loop commands: Example

```
#!/bin/bash
for x in 1 2 3 4 5 6 7 8 9
do
	echo -n "x is $x. "
	if [ "$x" -eq "4" ]; then
		echo
		continue
	fi
	echo "Most numbers get here, except 4."
	if [ "$x" -eq "7" ]; then
		break
	fi
done
echo "We have finished the loop now."
echo "At the end, x is now $x."
```

Controlling loop commands: Example

```
x is 1. Most numbers get here, except 4.
x is 2. Most numbers get here, except 4.
x is 3. Most numbers get here, except 4.
x is 4.
x is 5. Most numbers get here, except 4.
x is 6. Most numbers get here, except 4.
x is 7. Most numbers get here, except 4.
We have finished the loop now.
At the end, x is now 7.
```

Functions

- Syntax

```
function name[()]
{
   list of commands;
[return]}
```

- Function
  - Functions have to be defined in the beginning of the script
  - They may have or not arguments
  - Arguments and return values can be of any type
  - The variables defined inside the function are GLOBAL. They have to be defined LOCAL, if we want to reduce their scope

Functions: Example

```
#!/bin/bash
Outside="a global variable"
function mine() {
        local inside="this is local"
        echo $outside
        echo $inside
        outside="a global with new value"
}
echo $outside
mine
echo $outside
echo $inside
```

```
a global variable
a global variable
this is local
a global with new value
```

TD1:
Linux and Shell Exercises

Ex1: Wildcard characters

- List the entries inside the folder **/usr/bin** whose name starts with the letter *m.*

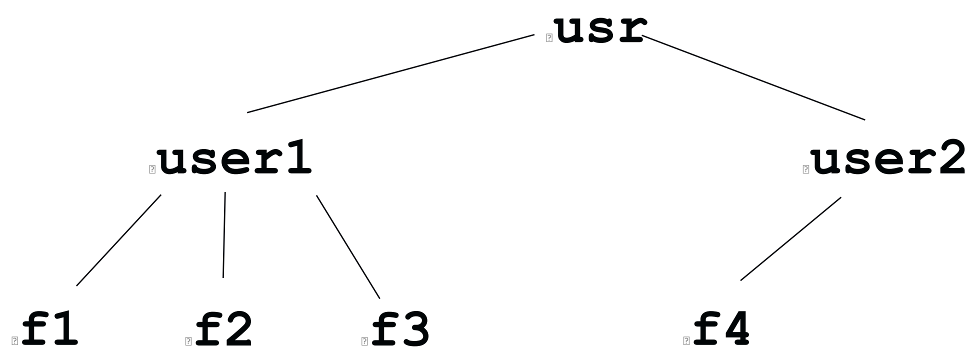
- List the entries inside the folder **/usr/bin** whose name starts with the letter *m* and it has exactly 3 characters

- List the entries inside the folder **/usr/bin** whose name starts with the letter *m* and it has at least 3 characters

- List the entries inside the folder **/usr/bin** whose name starts with the letter *m* and it has an extension (postfix after .)

Ex2: Redirections

- Given the following file hierarchy

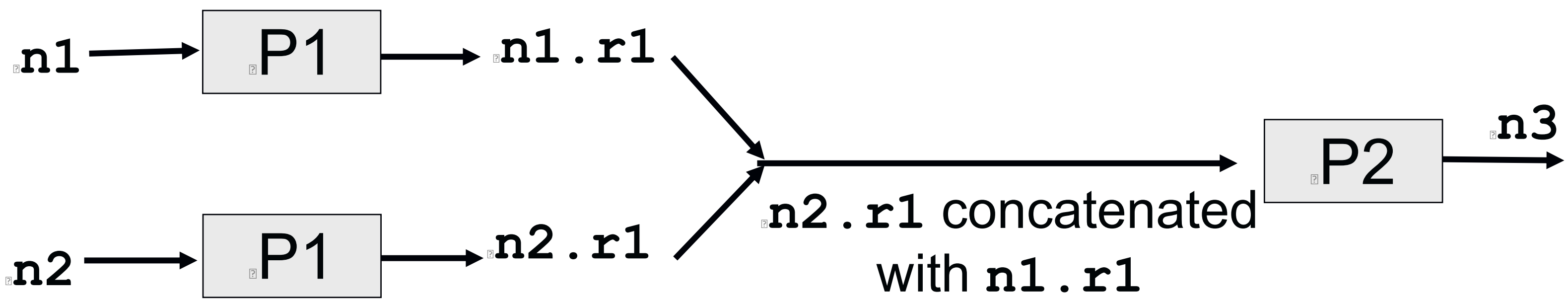


- **`user2`** is connected at his home folder
- Use the command **`cat`** :
  - Create a file **`f5`** with the following content

  5 2 3 -1
  - Create a file **`f6`** by concatenating **`f5`** after **`f4`**
  - Add the file **`f1`** of **`user1`** at the file **`f6`**
  - Copy the file **`f2`** of **`user1`** as a new file **`f7`** of **`user2`**

Ex3: Redirections

- Create script with the name **`execution`** called with 3 arguments **`n1 n2 n3`** (which are the filenames) performing the following manipulations

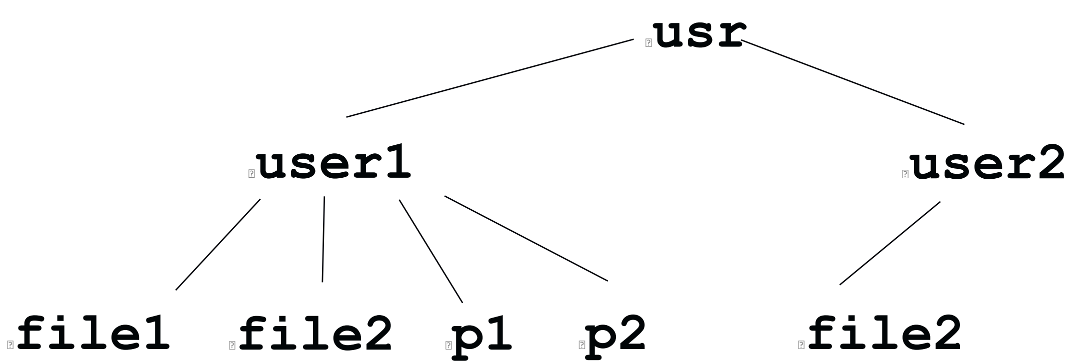


- **`P1`** and **`P2`** are programs that read from the `stdin` and print their result to `stdout`

Ex3: Redirections

- What happens when the following instruction is executed in the home directory of **`user1`**?

`execution file1 ../user2/file2 fileN`



- Write an interactive version of the script **`execution`** that asks from the user to provide **`n1`**, **`n2`** and **`n3`**, instead of passing them as arguments

Ex4: Script arguments

- Create a script with the name **`argument`** using the command **`expr`** that prints one by one the arguments passed during the script call.

For instance, the call:

```
% argument one two three
```

Should print in the stdout:

```
3 arguments :
  argument 1 – one
  argument 2 – two
  argument 3 – three
```

Ex5: Search

- Create a script with the name **`mysearch`** passing two arguments during the call, i.e. **`motif`** and **`filename.`** The program prints either "filename has motif" or "filename does not have motif"

- Create a script **`mysearch2 filename`** that ask the user the motif to be searched inside the filename until the motif end is entered by the user

- Create a script **`mysearch3 f1 .. fn`** that performs the same thing as **`mysearch2`**, but now it searches inside the files given as arguments.

- **`/dev/null`** = trash

Ex6: Find

- Describe the effect of the following instructions:
  - `find ./ -name "tp2" -exec {} < f_in > {}.out \;`
  - `find /var/log -exec grep "www.athabasca.com" {} \;`
  - `find . -name "rc.conf" -exec chmod o+r {} \;`
  - `find /usr/local -type f -name "*.html"`.

Part B:
C Language

Outline

- Introduction
- Structure of C program and compilation
- Basic components of C language
- Pointers and Arrays
- Functions and parameters
- Dynamic memory allocation
- Structures, Lists and Hash tables
- Strings
- I/O

CM 3:
Introduction, Structure and Tools

Why C ?

- JAVA is an evolved language
  - High abstraction away from the physical objects
  - Useful when we want to be abstracted from the hardware
    - Software engineering
- C is closer the hardware
  - Relatively controls what happens to the memory
  - Manipulation of the physical objects of the hardware
    - System engineering
- Main operating systems are written in C
  - Unix, GNU/Linux, etc.

What this course covers?

- It is NOT a yet another programming course with extended presentation of the language syntax,
  - Differences with JAVA, similar syntaxes but still very different !
  - Mainly “tricky” parts of C language !
- Being a master of C force is long and difficult
  - Requires practise, practise and practise
- But, there is a lot of documentation !
  - The functions of libraries are documented
    - Example : **`man fprintf`**
  - Google is your best friend…not to copy – paste, but read and understand !

C language

- Developed by Thompson & Ritchie in 1972
- Originally developed: 90-95% of the Unix kernel

- It has been standardized by the American National Standards Institute (ANSI) in 1989 (C ANSI)
- Loosely/weakly typed language (so, warnings are important !)

Structure of C program and compilation

C program's structure

- A program is saved into **one** or **multiple** compilation C files
- One compilation file consist of a source file or a header file
  - Source file **`.c`** → code
  - Header file **`.h`** → declarations

Communication among .c files

- **`code2.c`** can **use** the functions and the global variables defined:
  - locally in this compilation file **`code2.c`**
  - in another compilation file **`code1.c`** (**`code1.c`** exports them)
- **`code1.c`** **defines** the functions and the global variables:
  - exported by **`code1.c`** to be used by another file **`code2.c`**
  - used locally in **`code1.c`**

`code2.c`

Definition of variable **`b`**;

Use of variable **`b`**;

Use of variable **`a`**;

`code1.c`

Definition of variable **`a`**;

Use of variable **`a`**;

Communication among .c files

- The definition of the global variables and constants is by default local:
  - defined in **`code1.c`** → "seen" only by **`code1.c`**

- To be "seen" by file **`code2.c`** → the global variable has to be exported:
  - **declarations** are placed in the header file **`code1.h`**
    - **`extern`**: For the global variable to be seen by other **`.c`** files
  - **definitions** are in the source file **`code1.c`**
  - **`code2.c`** includes the header **`code1.h`** of **`code1.c`** through the directive **`#include`**

Directive #include

- Syntax:

```
#include <filename.h>  (1)
#include "filename.h"  (2)
```

- Description
  - (1) and (2): Include the file called **`filename`** which is stored in the folder

  `/usr/include`

  - Only (2): Search the file called **`filename`** in the **working directory**
    - if the file is saved in another directory you have to give the path ”**`path/filename`**”
- Functionality:
  - Includes (before compilation) the content of **`filename`**. This text is treated as it was part of the current file **`.c`**

Communication among .c files: Example

code1.c

```c
#include "code1.h"   /* Declaration made available here */

/* Variable defined and initialized here */
int global_variable = 1; /* Definition checked against declaration */

int increment(void) { return global_variable++; }
```

code1.h

```c
extern int global_variable;   /* Declaration of the variable */
```

code2.c

```c
#include "code1.h"
#include <stdio.h>

void use_it(void)
{
    printf("Global variable: %d\n", global_variable);
}
```

Communication among .c files: Example

```
extern void use_it(void);
extern int increment(void);
```

prog.h

prog.c

```
#include "code1.h"
#include "prog.h"
#include <stdio.h>

int main(void)
{
    use_it();
    global_variable += 10;
    use_it();
    increment();
    use_it();
    return 0;
}
```

Communication among .c files: Example

code1.c

```
#include "code1.h"   /* Declaration made available here */
#include "prog.h"

/* Variable defined here */
int global_variable = 1; /* Definition checked against declaration */

int increment(void) { return global_variable++; }
```

code1.h

```
extern int global_variable;   /* Declaration of the variable */
```

code2.c

```
#include "code1.h"
#include "prog.h"
#include <stdio.h>

void use_it(void)
{
    printf("Global variable: %d\n", global_variable++);
}
```

General rules

- For any given global variable
  - exactly **1 header file declares** it
  - exactly **1 source file defines** it and preferably **initializes** it too.

    (otherwise undefined behavior → anything could happen !)
  - The source file that defines the variable also includes the header (ensuring definition and declaration are consistent)

- A source file
  - **never** contains **`extern`** declarations of variables
  - **always** include the (sole) **header** that declares them

- A header file only contains **`extern`** declarations of variables
  - never **`static`**

    (each source file makes its own version of the global variable)

Structure of .c file

- Description of the functionality and the use (comments ☺)

- The directives for the pre-processor
  - Include files, Macros, Conditional compilation, …
- Global variables used only by this file
  - if not initialized, they are by default 0
- The functions that act on the variables
  - Function prototype
  - Function body
- No order is imposed to the definitions, BUT any element has to be declared BEFORE its utilisation

Structure of .h file

- **Declaration** of whatever is allowed to be “seen” outside file `.c`
  - Functions
  - Global variables
  - Data structures
  - …

- It is the source file `.c` interface to outside world

Structure of .c file: Example

```c
/****************************
 * main.c
 ****************************/

#include <stdlib.h>
#include <stdio.h>

#define PI 3.14
```

Pre-processor directives

- Include files

- Macros

```c
int variableGlobale = 3;
```

Global variable defined in this file

```c
extern int ppcm (int a, int b);
```

Function declaration: is defined in another file

```c
void printINT(int val)
{
    printf("%d\n",val);
}
```

Function defined in this file (can be exported)

```c
int main(int argc, char *argv[])
{
    int variableLocale = ppcm(2,variableGlobale);
    printINT(variableLocale);
    return EXIT_SUCCESS;
}
```

The main function is the **starting** point

Only 1 main function!

```c
/****************************
 * ppcm.c
 ****************************/
int ppcm(int x, int y)
{
    int a=x;
    int b=y;
    while (a!=b)
    {
        while (a>b) b=b+y;
        while (a<b) a=a+x;
    }
    return a;
```

Same example split in 3 files

```c
/***************************
 * main.c
 ***************************/

#include <stdlib.h>
#include <stdio.h>

#include "ppcm.h"

#define PI 3.14

int variableGlobale = 3;

//extern int ppcm (int a, int b);

void printINT(int val)
{
    printf("%d\n",val);
}

int main(int argc, char *argv[])
{
    int variableLocale = ppcm(2,variableGlobale);
    printINT(variableLocale);
    return EXIT_SUCCESS;
}
```

```c
/***************************
 * ppcm.h
 ***************************/

#ifndef PPCM_H
#define PPCM_H

int ppcm(int x, int y);

#endif
```

```c
/***************************
 * ppcm.c
 ***************************/
int ppcm(int x, int y)
{
    int a=x;
    int b=y;
    while (a!=b)
    {
        while (a>b) b=b+y;
        while (a<b) a=a+x;
    }
    return a;
}
```

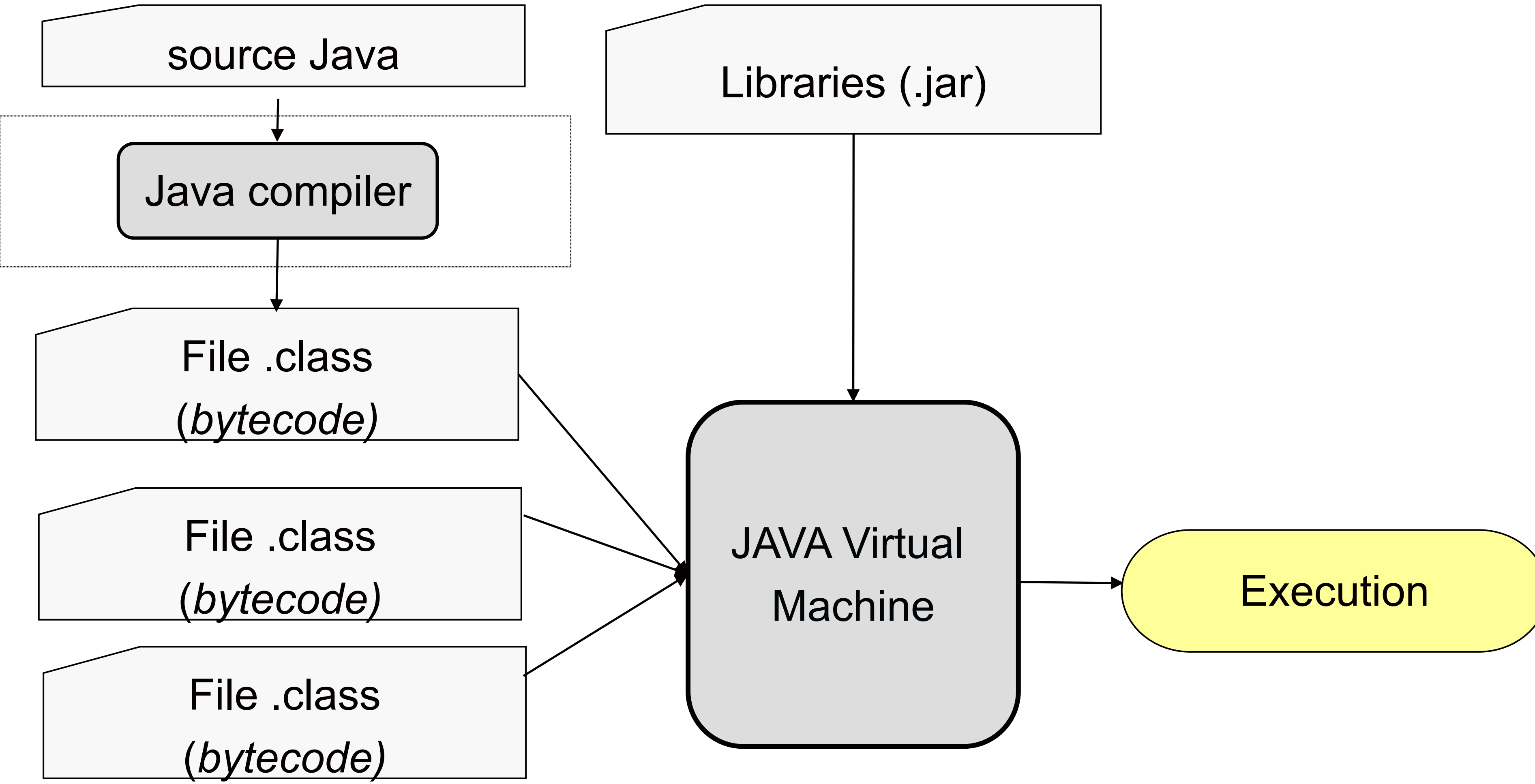
JAVA compilation

- The `.java` files have the code that will be translated into the files `.class` by the compiler `javac`
  - The program is split into classes

- The files `.class` consist of the *bytecode* java: a lower level language closer to the machine language (but not a machine language)

- The *bytecode* can be executed in whatever machine that has a Java Virtual Machine. The JVM interprets the *bytecode* to machine language (portable, but slow)
  - The link among classes is performed during the execution (Dynamic link)

JAVA compilation



C compilation

- The call of the C compiler is associated with **3 tools**
- 1st step: The pre-processor (CPP):
  - Handles the directives **`(#include, #define`** etc.)
- 2nd step: C compiler and assembler (GCC):
  - C compiler performs **one-to-one** translation of a compilation file (**`.c`** and **`.h`**) into an assembly file (**`.s`**).
    - This step is hidden by default (use **`gcc –S`** to see it)
  - Then, the assembler generates an object file (**`.o`**) with
    - the machine instructions of the functions **defined** in this **`.c`** file
    - information on the functions **used** in this **`.c`** file, but defined in another file
- 3rd step: The linker (LD/GCC):
  - Makes the link (resolves symbols) among the different program's objects (static linking) to **one executable binary**

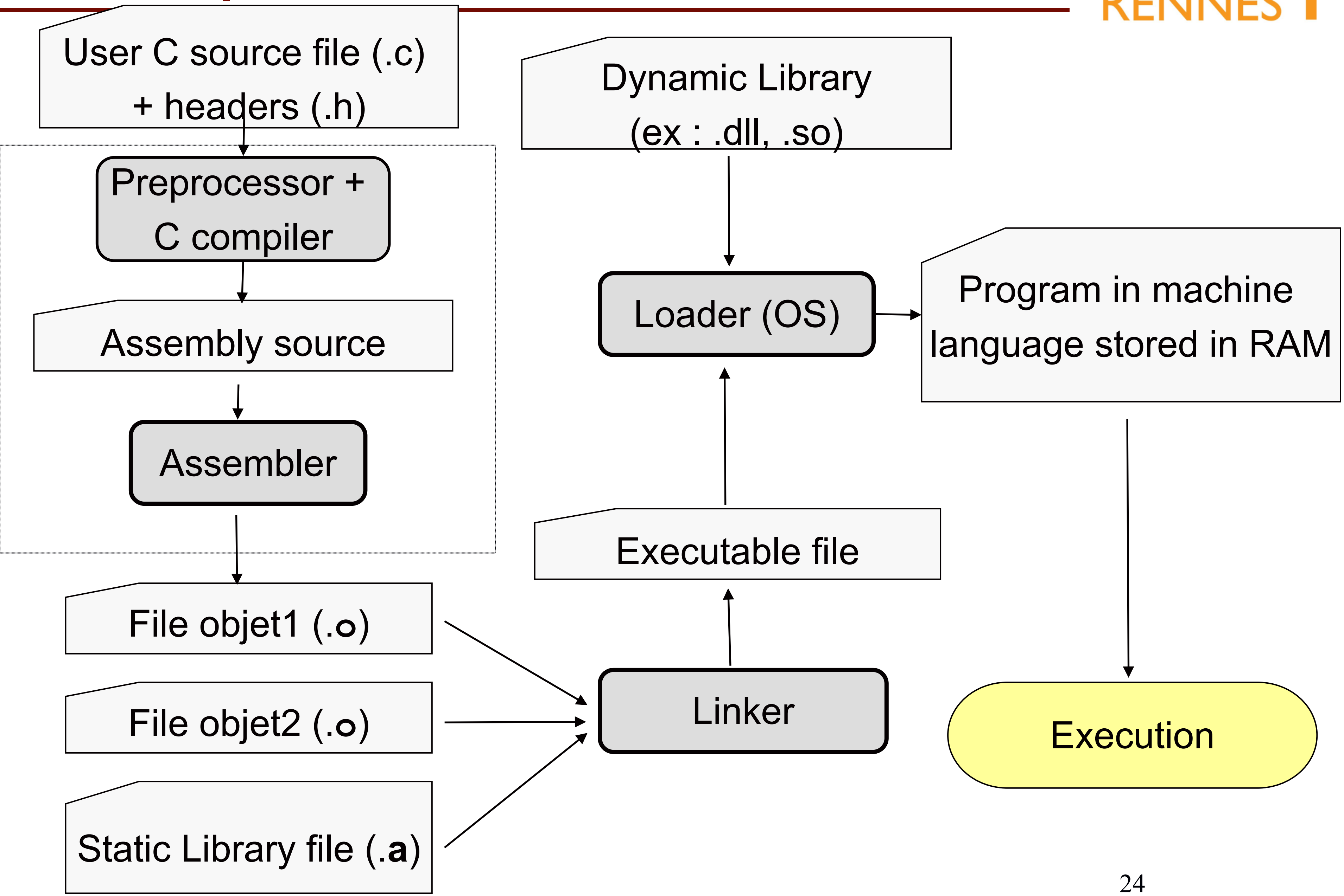
C compilation

- The generated binary depends on
  - **Architecture/processor**

    AND
  - **operating system**
  - → It is not portable !

- Possible to use “cross compilation”
  - Use a machine to create an binary for another machine
  - **`–march`**
    - **`native`**
    - **`i386`**

C compilation



C compilation: GCC

- **G**NU (GNU is Not Unix!) **C**ompiler **C**ollection is free software:
  - Design to target several machines
  - Exist by default in Linux
- Syntax (see **`man gcc`**):

`gcc [-c] [-g] [-I dir] [-o nom] f1 … fn`

- Options
  - **`-c`** : Produce an **object** file **`.o`** from **`.c`** file by stopping before linking
  - **`-I dir`** : adds **`dir`** in the include path (used header files **`.h`**)
  - **`-o nom`** : Specifies the name of output file. By default:
    - object files have the same name as their code files
    - Executable file: **`a.out`**
  - **`-g`** : Inserts the information needed by the debugger

GCC: One compilation file

- Produce an executable called **`module`** from the file **`module.c`**

  `gcc module.c -o module`

```c
/****************************
 * main.c
 *
 * le classique Hello World
 ****************************/

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    printf("Hello World\n");
    return EXIT_SUCCESS;
}
```

```
bash-3.2$ ls
main.c
bash-3.2$ gcc main.c -o hello
bash-3.2$ ls
hello   main.c
bash-3.2$ ./hello
Hello World
bash-3.2$
```

GCC: Several compilation files

- Produce an **object** file called **`module.o`** from a code file **`module.c`** (**`-c:`** stop before linking)

```
gcc -o module.o -c module.c
```

- Produce an **executable** file named **`prog`** by linking its object(s) files

```
gcc -o prog module1.o … moduleN.o
```

C compilation: The tool `make`

- Principle
  - Use to automate the compilation **(large) projects**, which consists of several files
  - Handles file dependencies (e.g. **`#includes`**)
  - Allow to automatize the update of the files

- Functionality: Create a file called **`Makefile`** that
  - Defines the dependencies between targets
  - Defines the dependencies between each target and the corresponding files
  - Provides the commands required to be executed to create the target based on the dependencies (files)

The tool `make`: Example

- A C program consist of 3 files:

`prog.c`, **`code1.c`**, **`code2.c`**

- Graph with file dependencies

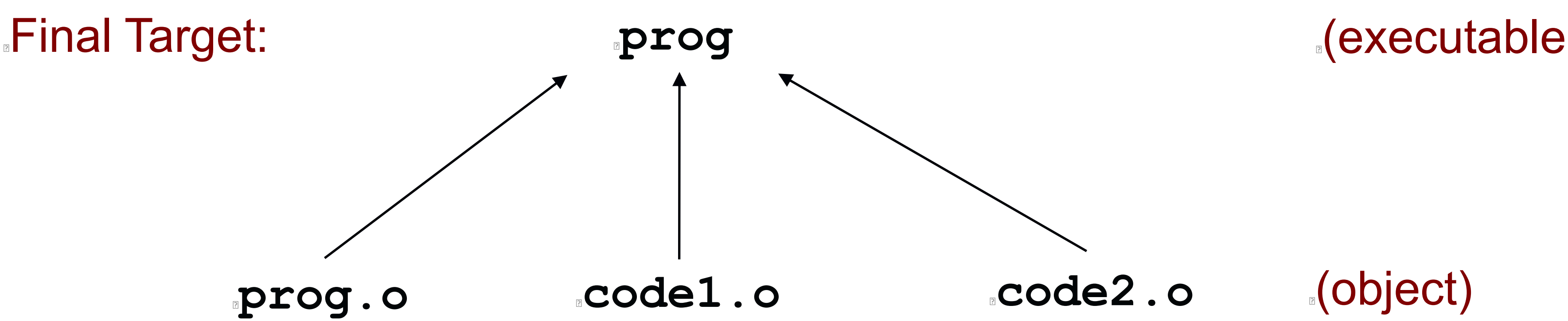
Final Target: **`prog`** (executable)

The tool `make`: Example

- A C program consist of 3 files:

`prog.c`, **`code1.c`**, **`code2.c`**

- Graph with file dependencies



To produce the executable file **`prog`**, the following command has to be executed :

```
gcc -o prog prog.o code1.o code2.o
```

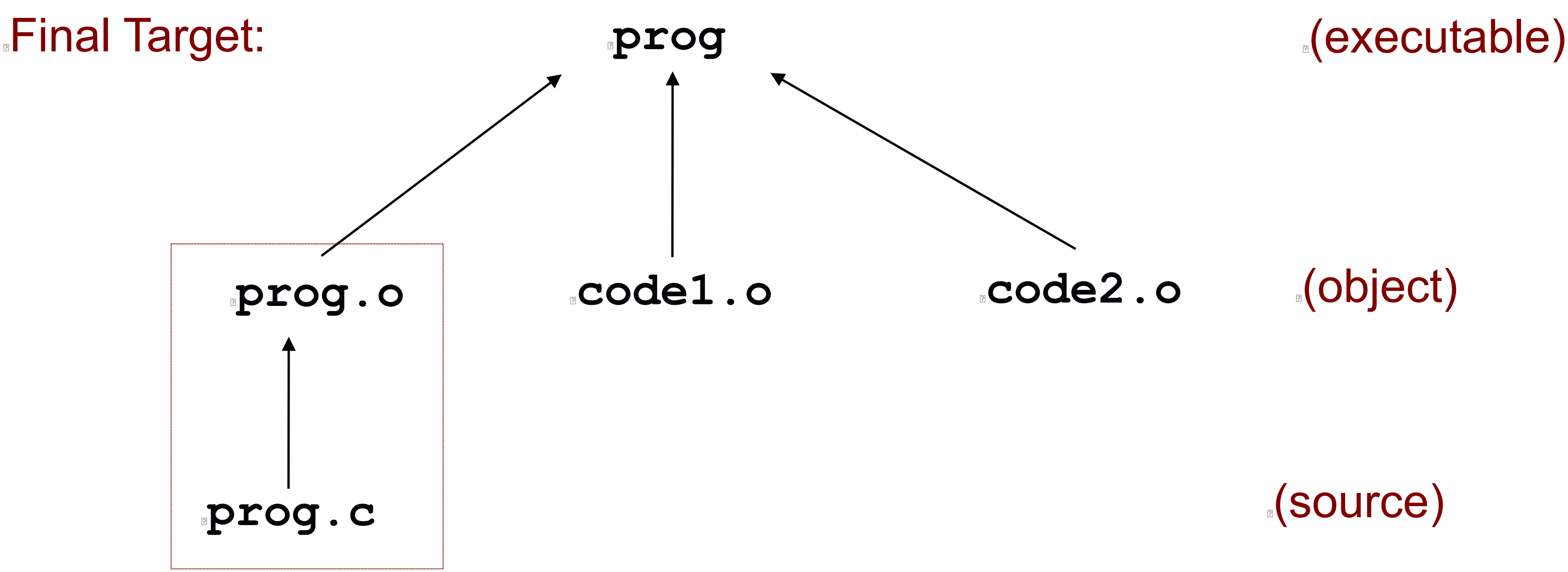
The file **`prog`** depends on the object files **`prog.o`**, **`code1.o`** and **`code2.o`**

The tool `make`: Example

- A C program consist of 3 files:

`prog.c`, **`code1.c`**, **`code2.c`**

- Graph with file dependencies



To produce the object file **`prog.o`**, we have to execute **`gcc -c prog.c`**

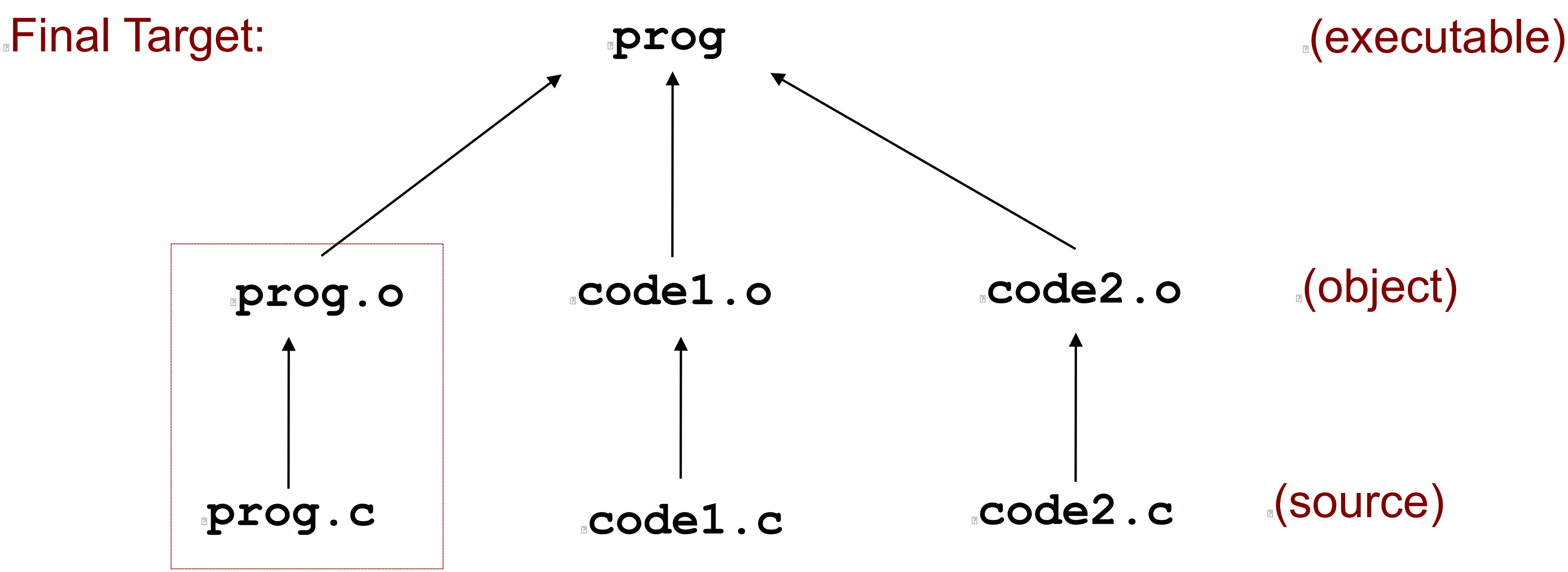
So, the file **`prog.o`** depends on the source file **`prog.c`**

The tool `make`: Example

- A C program consist of 3 files:

`prog.c`, **`code1.c`**, **`code2.c`**

- Graph with file dependencies

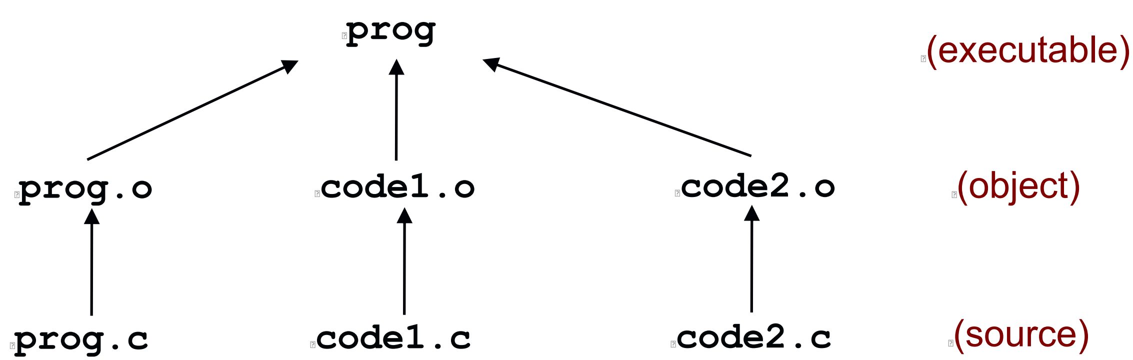


The graph of file dependencies provides the definition of the targets, dependencies and commands of the **`makefile`**

Structure of the `Makefile`

- Format of a rule
  - **`target: dep_file1 dep_file2`** …
    `(TAB) command`
- Application of rule:
  - If the date of the last modification in one of the dependent files **`dep_file1, dep_file2`**, etc. is more recent that the date of the file **`target`**, the tool **`make`** executes the **`command`**
- Use: command **`make`**
  - The tool executes the first target of the file **`makefile`** of the working directory
  - **`make label`**: Tool executes the rule associated with the target called **`label`**

Structure of the Makefile: Example



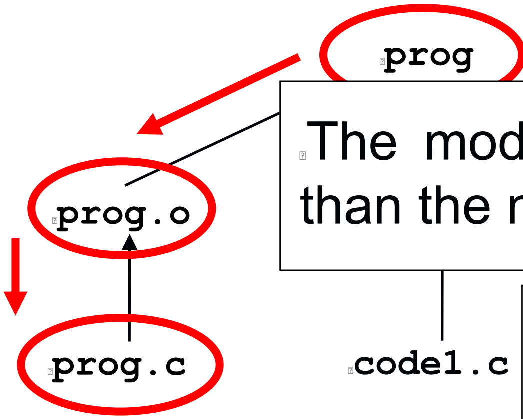
TAB character

makefile

```
prog : prog.o code1.o code2.o
	gcc -o prog  prog.o code1.o code2.o

prog.o : prog.c prog.h code1.h
	gcc -c prog.c -o prog.o

code1.o : code1.c prog.h code1.h
	gcc -c code1.c -o code1.o

code2.o : code2.c prog.h code1.h
	gcc -c code2.c -o code2.o
```

Structure of the `Makefile`: Example

- We modify **`prog.c`**, what happens when we execute **`make`**?



The modification date of **`prog.c`** is **more recent** than the modification date of **`prog.o`**

The command is **executed** !

makefile

```
prog : prog.o code1.o code2.o
        gcc -o prog  prog.o code1.o code2.o

prog.o : prog.c prog.h code1.h
        gcc -c prog.c -o prog.o

code1.o : code1.c prog.h code1.h
        gcc -c code1.c -o code1.o

code2.o : code2.c prog.h code1.h
        gcc -c code2.c -o code2.o
```

Structure of the `Makefile`: Example

- We modify **`prog.c`**, what happens when we execute **`make`**?

`prog` (executable)

The modification date of **`prog.o`** is **more recent** than the modification date of **`prog`**

`prog.o`

`prog.c` **`code1.c`**

The command is **executed** !

`makefile`

```
prog : prog.o code1.o code2.o
      gcc -o prog  prog.o code1.o code2.o

prog.o : prog.c prog.h code1.h
      gcc -c prog.c -o prog.o

code1.o : code1.c prog.h code1.h
      gcc -c code1.c -o code1.o

code2.o : code2.c prog.h code1.h
      gcc -c code2.c -o code2.o
```

Structure of the `Makefile`: Example

- Add a **`clean`** target: Remove object files, executable
- Use: **`make clean`**

```
makefile
```

```
prog : prog.o code1.o code2.o
        gcc -o prog  prog.o code1.o code2.o

prog.o : prog.c prog.h code1.h
        gcc -c prog.c -o prog.o

code1.o : code1.c prog.h code1.h
        gcc -c code1.c -o code1.o

code2.o : code2.c prog.h code1.h
        gcc -c code2.c -o code2.o

clean :
        rm *.o prog
```

Structure of the `Makefile`: Example 2

UNIVERSITÉ DE RENNES 1

```c
/***************************
 * hello.h
 ***************************/

#ifndef HELLO_H
#define HELLO_H

void Hello(void);

#endif
```

```c
/***************************
 * hello.c
 ***************************/
#include <stdio.h>

void Hello(void)
{
    printf("Hello World\n");
}
```

```c
/***************************
 * main.c
 ***************************/

#include <stdlib.h>
#include "hello.h"

int main(void)
{
    Hello();
    return EXIT_SUCCESS;
}
```

```makefile
hello: hello.o main.o
	gcc -o hello hello.o main.o

hello.o: hello.c hello.h
	gcc -c hello.c


main: main.c hello.h
	gcc -c main.c

clean:
	rm *.o
```

Structure of the `Makefile`: Example 2

```c
/****************************
 * hello.h
 ****************************/

#ifndef HELLO_H
#define HELLO_H

void Hello(void);

#endif
```

```c
/****************************
 * hello.c
 ****************************/
#include <stdio.h>

void Hello(void)
{
    printf("Hello World\n");
}
```

```c
/****************************
 * main.c
 ****************************/

#include <stdlib.h>
#include "hello.h"

int main(void)
{
    Hello();
    return EXIT_SUCCESS;
}
```

```makefile
CC=gcc
CFLAGS=-g -Wall -I.
DEP=hello.h
OBJ=hello.o main.o
EXEC=hello

all: $(EXEC)

$(EXEC): $(OBJ)
	$(CC) -o $@ $^

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

.PHONY: clean mrproper
clean:
	rm *.o

mrproper: clean
	rm $(EXEC)
```

Common compilation error messages

- Example of an error message

```
bash-3.2$ gcc main.c ppcm.c -o main
main.c:19: error: expected '=', ',', ';', 'asm' or '__attribute__' before 'main'
bash-3.2$
```

- Set of errors!
  - Always start correcting the first one (they can be dependent)!!

```
bash-3.2$ gcc main.c ppcm.c -o main
main.c: In function 'main':
main.c:21: error: 'a' undeclared (first use in this function)
main.c:21: error: (Each undeclared identifier is reported only once
main.c:21: error: for each function it appears in.)
main.c:22: error: expected ';' before 'int'
main.c:23: error: 'variableLocale' undeclared (first use in this function)
bash-3.2$
```

Debugging

- Usual bugs
  - Infinite loops
  - Loop boundaries
  - Non-Initialized variables
  - Segmentation faults
  - Algorithmic faults
  - …

- Tools to support the correction of the code
  - The debugger is a software that helps the developer in the analysis of the bugs of a program
  - Examples : GDB, valgrind …

Tool GDB

- With the GDB, we can
  - Start the execution of the program
  - Stop the execution of the program based on some conditions
  - Examine what happened when the program stopped
  - Execute the program step-by-step
  - Perform modifications during the execution of the program

- Use: Make the link between the program and the debugger
  - Add the information of debug during compilation
    - Option **`-g`** and **`-ggdb`**

Tool GDB: Commands

- Start execution: **`run`**
- Show the stack of the call: **`backtrace`**
- Break Point : **`break`** (help breakpoints)
- Continue execution : **`cont`**
- Execute one instruction : **`step`**
- Display the variables
  - Display **`nom_variable`**
  - **`watch nom_variable`** (allows to monitor)
  - info locals (local variables)
- …
- GDB Quick Reference Sheet !!!

Tool `Valgrind`

- With Valgrind we can identify memory leaks
  - Example of a memory leak

```
/******************
 * test.c
 ******************/
#include <stdlib.h>
int main()
{
    int a=2;
    int* p = (int*) malloc(1024);
    p=&a;
    return 0;
}
```

- Use:
  `valgrind --tool=memcheck ./test`

…before going in more details about the C syntax…

Code style is very important!

- Understanding the code written by someone else is a hard job!
- Use a universal language (readable by everyone)
  - Πιστεύω ότι κανένας δεν μπορεί να καταλάβει αυτή τη γραμμή χωρίς να χρησιμοποιήσει μεταφραστή
- Decide a code style and be CONSISTENT
- Comments
  - Documentation: Programmers that want to use the code
  - Implementation: Programmers that maintain the code, i.e. they have to understand your implementation
- Names that mean something
  - **`int z`** Vs **`int counter`**

CM 4:
Basic components of C language: Primitive data types, Libraries and Control structures

Instructions & Comments

- Instruction is any expression finishing with ;
- Block of instructions:

```
{
      list of instructions
}
```

- Comments
  - `/* this is a comment */`
  - `// this is one line comment`
- The language C does not support disambiguation
  - The **`keywords`** of the language are reserved and cannot be used for naming variables and functions.

Value representation

- Hexadecimal numbers begin with **`0x`**:
  - **`0x23`** is $\mathbf{23}_{16}$ (which is 35 in the base of 10)

- Octal numbers begin with 0:
  - **`023`** is $\mathbf{23}_{8}$ (which is 19 in the base of 10)

- No prefix for decimal numbers:
  - **`23`** is $\mathbf{23}_{10}$

- Negative representation – operator
  - **`-0x23`** is the negation of **`0x23`**

Data object definition

- Syntax:

  `type identifier;`

- Function:
  - Reserve in the memory a part equal to the size required to store a data object of the type **`type`**
  - Associate the starting address of this memory part with the **`identifier`**

Primitive data types: Type specifiers

- Same as in JAVA:
  - Integral
    - **`char, short, int, long`**
  - Real and complex
    - **`float, double`**
- With additional specifiers for the representation:
  - **`unsigned`**: always 0 or positive
    - Representation: Binary
  - **`signed`**: positive or negative:
    - Representation: implementation depended
      - 1's complement
      - 2's complement (usual one)
      - sign and magnitude
- Example: **`unsigned short int`**

Primitive data types: Size

- The exact data type **size** is NOT part of the C standard
  - **`int`** can be 16 or 32 bits…
- The standard only says that the range of a **larger** type shall be **at least as big as** the **smaller** type.
  - **`char ≤ short ≤ long ≤ long long`**
  - **`char:`** The minimum supported data type
- Compiler decides based on the target platform architecture! (though, there is a **minimum** allowed width)
- **`sizeof()`** can give the **actual** size in bytes
- To give a specific storage size use the **`YintX_t`** type
  - **`Y = u`** for **`unsigned`**
  - **`X = 8, 16, 32`** etc for the number of bits to be used

Primitive data types: Min values

Type	Storage size	Value range
`char`	*usually* ***`signed`***	*(but not standard)*
`unsigned char`	1 byte	0 to $2^8 - 1$
`signed char`	1 byte	$-(2^7)$ to $2^7 - 1$
`int`	2 bytes (usually *4 bytes*)	$-(2^{15})$ to $2^{15} - 1$
`unsigned int`	2 bytes (usually *4 bytes*)	0 to $2^{16} - 1$
`short`	2 bytes	$-(2^{15})$ to $2^{15} - 1$
`unsigned short`	2 bytes	0 to $2^{16} - 1$
`long`	4 bytes	$-(2^{31})$ to $2^{31} - 1$
`unsigned long`	4 bytes	0 to $2^{16} - 1$
`long long`	8 bytes	$-(2^{63})$ to $2^{63} - 1$
`unsigned long long`	8 bytes	0 to $2^{64} - 1$

Primitive data types: Float

Type	Storage size	Value range	Precision
`float`	4 byte	1.2E-38 to 3.4E+38	single
`double`	8 byte	2.3E-308 to 1.7E+308	double
`long double`	10 byte	3.4E-4932 to 1.1E+4932	extended

Defining primitive data objects: Examples

- Definitions
  - `int i;`
  - `float f1, f2;`
  - `unsigned char c1, c2;`

- What is value of these identifiers?
- Whatever is already stored in the memory just reserved…
(except global and static variables)

- Definitions with initialization
  - `int i = 3;`
  - `float f1 = 1.2, f2 = 0.3;`
  - `unsigned char c1 = 'A', c2 = 65;`

!!!

Value of ASCII character

INITIALIZE ALL VARIABLES !!

Operators

- Perform operations on data objects
- They have priorities in the order they are applied:
  - Check the priority order
  - **USE PARENTHESIS**
- Operator types
  - Arithmetic
  - Relational
  - Logical
  - Bitwise
  - Assignment
  - …..

Arithmetic operators

A = 10, B = 20

Operator	Description	Example
+	Adds two operands.	A + B = 30
−	Subtracts second operand from the first.	A − B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	**Remainder** after **integer** division.	B % A = 0
	(assignment at the same time).	
++ !!!	Increases the integer value by one	A++ = 11
--	Decreases the integer value by one.	A-- = 9

- Postfix (Post-increment): Do the operation and return **old** value
  - `A++` returns 10 and then does A=11
- Prefix (Pre-increment): Do the operation and return **new** value
  - `++A` does =11 and then returns A

Relational operators

A = 10, B = 20

Op.	Description	Example
==	Checks if the values of two operands are equal. If yes, the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are not equal different. If yes, the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, it becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, it becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand.	(A <= B) is true

- **Any integer value can be considered as Boolean**
  - A FALSE expression is coded as a zero integer
  - A TRUE expression is coded as any non-zero integer value

Logical operators

- The operands are TRUE or FALSE

Op.	Description	A = 1, B = 0 **Example**
&&	**Logical AND**: If **both** the operands are non-zero, then the condition becomes true. Does not evaluate 2nd operand, if the first is false	(A && B) is false.
||	**Logical OR**: If **any** of the two operands is non-zero, then the condition becomes true. Does not evaluate 2nd operand, if the first is true	(A || B) is true. !(A && B) is true.
!	**Logical NOT**: It is used to **reverse** the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	

Bitwise operators

- The operands are integral (**`short, int, unsigned, etc`**)

A = 0x3 (0011) B = 0x2 (0010)

Op.	Description	Example
&	**Binary AND:** Copies **a bit** to the result, if it exists in **both** operands.	(A & B) = 0x2 (0010)
|	**Binary OR:** Copies a **bit** if it exists in **either** operand.	(A | B) = 0x3 (0011)
^	**Binary XOR:** Copies the bit if it is **set only in one** operand but not both.	(A ^ B) = 0x1 (0001)
~	Binary Ones Complement: Has the effect of '**flipping**' bits.	(~A) = 0xC (1100)
<<	**Binary Left Shift**: The left operands value is moved left by the number of bits specified by the right operand.	(A<<2) = 0xC (1100)
>>	**Binary Right Shift:** The left operands value is moved right by the number of bits specified by the right operand.	(A>>2) = 0x0 (0000)

Assignment operators

A = 10, B = 20

Op.	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B C = 30
+= -= *= \= %= &= |= ^= <<= >>=	In can be combined with the other arithmetic operators (+, -, *, \, %) and bitwise operators (&, |, ^, <<,>>). It performs the arithmetic operation with right operand and the left operand and assign the result to the left operand.	A += 1 A = 11

Common error: Assignment Vs Testing

- Assignment: **`nom_variable = expression ;`**
  - **`x=3;`**
  - It is an expression that stores the value 3 to the variable **`x`**

- Test of equality: **`expression1 == expression2 ;`**
  - **`x==3;`**
  - It returns 1 if the stored value in **`x`** is equal to 3, otherwise 0

ATTENTION:

= and == are different !

Kahoot time !

(6 Q)

So, what happens if we mix types in an operation?

`a (operator) b` **`type a ≠ type b`**

- **Automatic IMPLICIT and SILENT conversion!**
- **Both** operands (**`a`**, **`b`**) must be data objects of the **same type**
- Converted to the **common higher rank** of the operands based on the type width:

PROMOTION

- **Integer promotions:**
  - **`signed → unsigned`**
  - small integer types: **`char, short, int16, int8 etc`** (**BOTH `signed`** and **`unsigned`**) are converted to
    - **`int`** : **IF** the range of **`int`** **can** represent ALL values of the original type
    - else: **`unsigned int`**

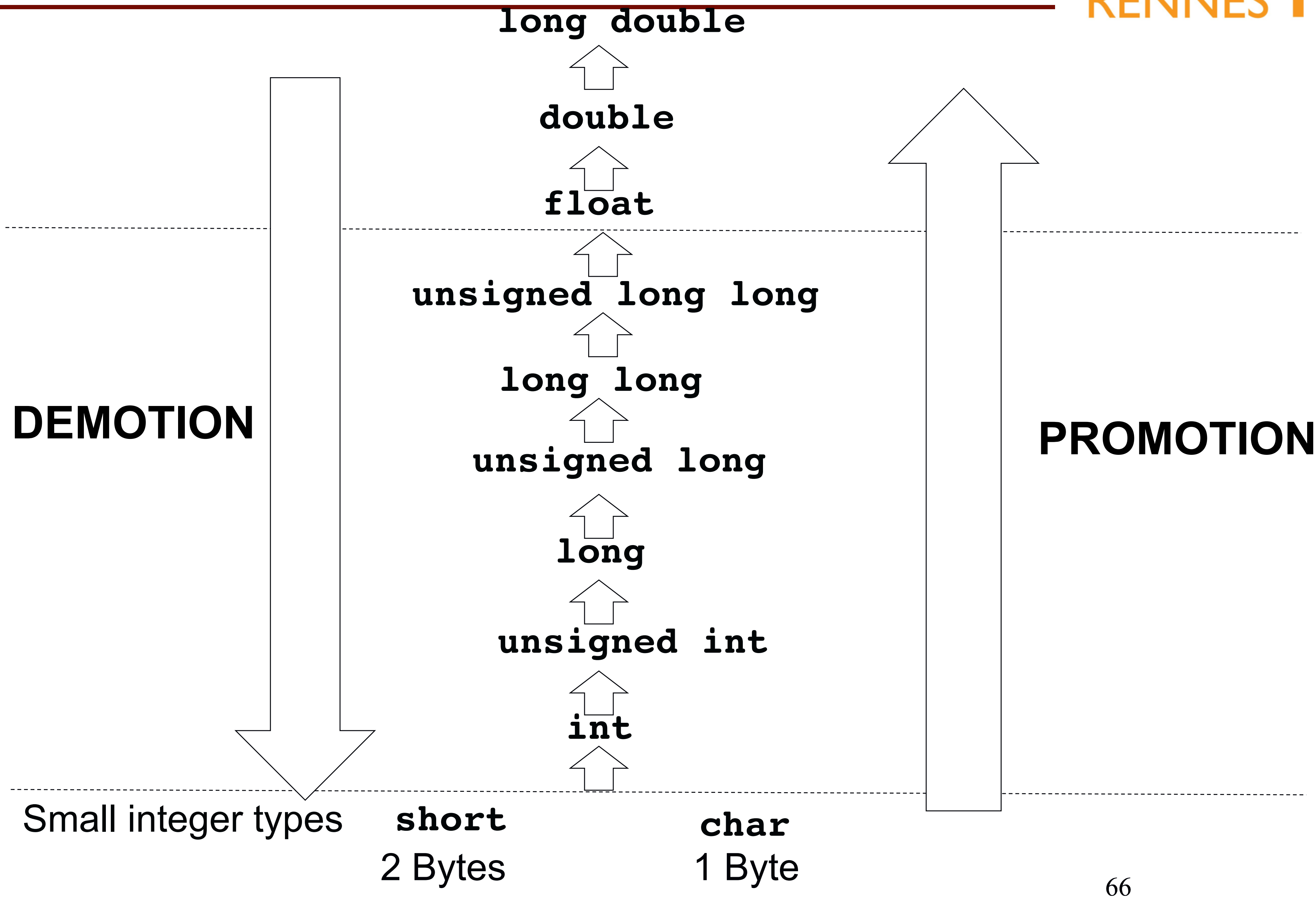
And during assignment?

```
c = a (operator) b
```

`type c ≠ type` (result of operation)

- **….Still automatic IMPLICIT and SILENT conversion**!
- Converted to the type of data object that stores the result:
  - Higher rank: **PROMOTION**
  - Lower rank: **DEMOTION**

Primitive data types: Implicit conversion



Explicit conversion: Cast operator

- Syntax

  `(type) identifier;`

- Creates a **copy** of the data object and changes its type.

- The data object of the **`identifier`** is **NOT** modified

With the Implicit or explicit conversion

- The type changes →
- So, the storage size and/or the representation changes →
- So, the **value CAN change**

And, when we cast different representation ?

- Casting **`float`** to **`int`**: It changes representation!
  - **`float`** is represented in IEEE 754 single precision
  - **`int`** is usually 2's Complement

```
float f = 3.9 ;
int val = (int) f  ;
```

  - Value gets truncated !
    - 3.9 is truncated to 3.0 and stored as 3 in 2's complement

- Casting **`int`** to **`float`:** It changes representation !

```
int f = 3 ;
float val = (float) f  ;
```

  - Value to convert to float (thus, 3 is converted to 3.0)

Type conversion: Examples

- `int x;`
- Implicit conversion: `x = 1.5 + 1.6;`
  - `1.5 → double`
  - `1.6 → double` !!!
  - `1.5 + 1.6 = 3.1` double addition
  - `x → int`
  - `3.1 → 3` **DEMOTION** to match with `int`
    - `x = 3`
- Explicit conversion: `x = (int) 1.5 + (int) 1.6;`
  - `1.5 → double`
  - `(int) 1.5 → 1` **DEMOTION**
  - `1.6 → double`
  - `(int) 1.6 → 1` **DEMOTION**
  - `1 + 1 = 2` integer addition
  - `x → int`
    - `x = 2`

Type conversion: Examples

- `float x;`
- Implicit conversion: `x = 3/5;`
  - `3 → int`
  - `5 → int`
  - `3/5 = 0` integer division
  - `x → float`
  - `0.0` **PROMOTION** to match with `float`
    - `x = 0.0`
- Explicit conversion `x = (float) 3/5;`
  - `3 → int`
  - `(float) 3 → 3.0` **PROMOTION**
  - Promotion `5 → 5.0` **PROMOTION**
  - `3.0/5.0 = 0` float division
  - `x → float`
    - `x = 0.6`

Type conversion: Usual problems

- Casting **`signed`** to **`unsigned`**:
  - **Negative value** is converted to a **high positive value**
  - Especially occurs during function calls or return values
- Casting a smaller **`signed int`** type to a larger **`unsigned int`** type
  - Sign extension: The sign bit is used in the unused bits, i.e. the Most Significant Bis (MSB) of the larger type
  - **`char`** and **`short`** are signed!
  - **Negative value** is converted to a **high positive value**

SECURITY ISSUES

Type conversion: Usual problems

- Casting a **larger type** to a **smaller one** (DEMOTION):
  - **Truncation**: Cuts off the MSB (overflow occurs).

```
int f = 0x12345678 ;
short int g;
g = f;
```

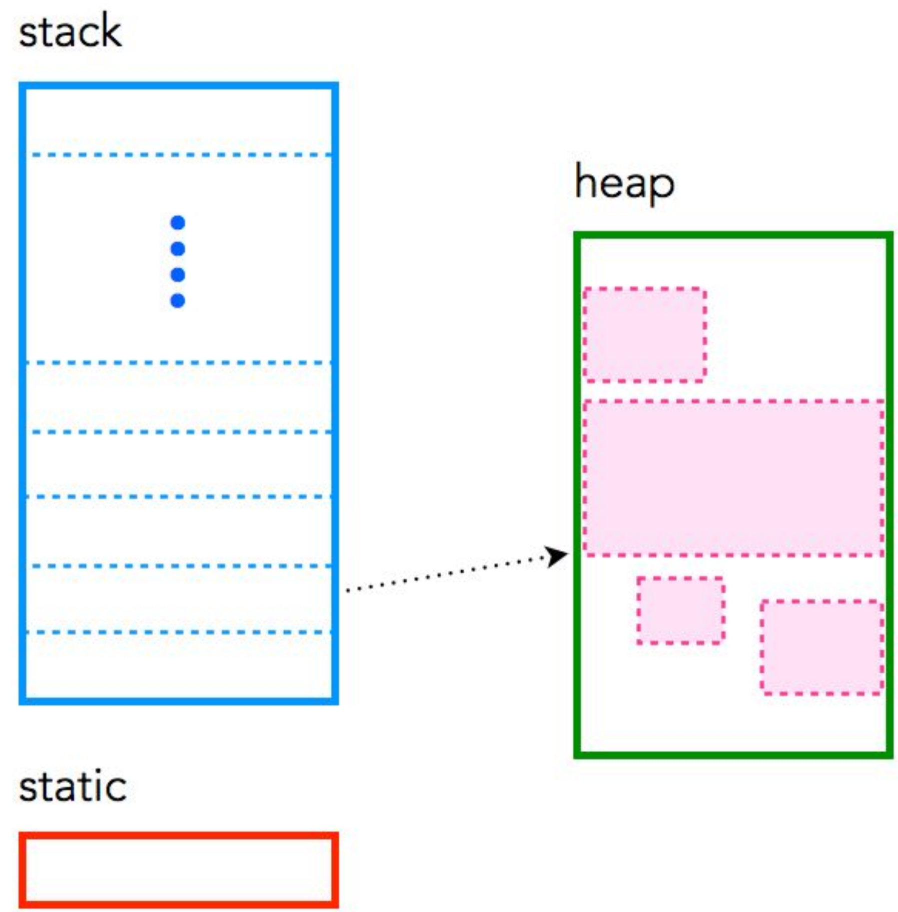
- Comparisons, especially with **`unsigned`**, **`char`** and **`short`** !

```
unsigned int a = 0 ;
if (a > -1)
   printf("TRUE\n");
else
   printf("FALSE\n");
```

SECURITY ISSUES

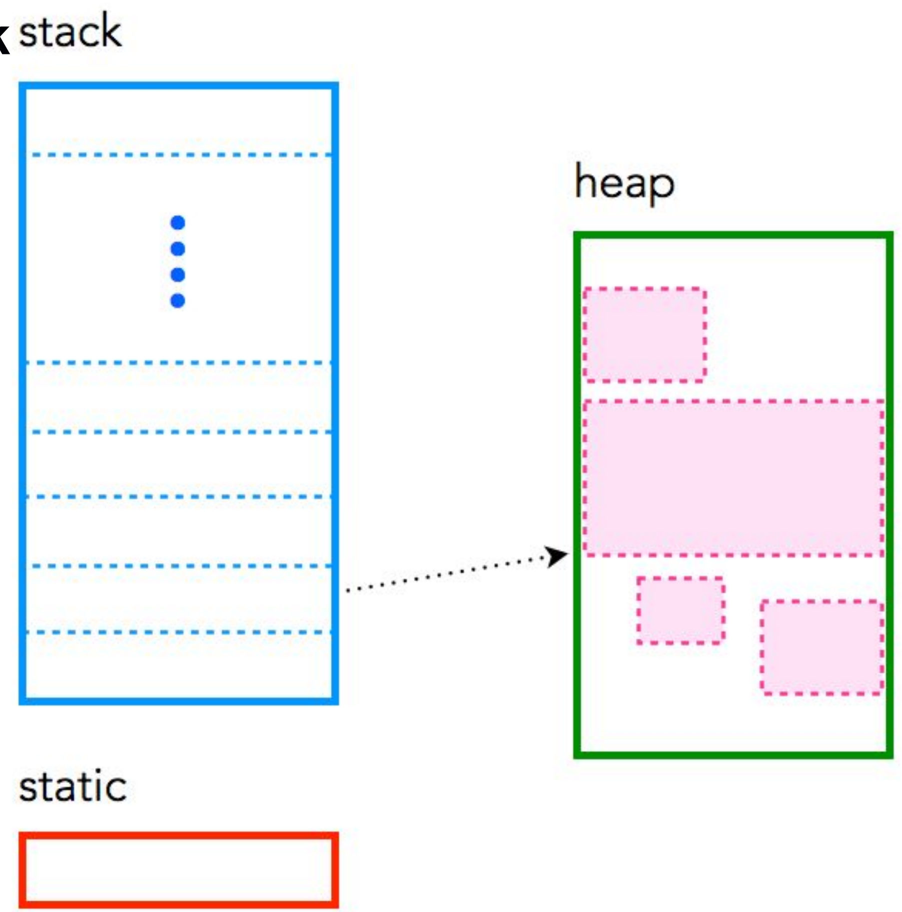
Variable allocation and lifetime

- **Static memory**:
  - Variable Lifetime: **complete** program
  - Stores:
    - **Global variables**: data objects declared outside of all function blocks
    - Variables defined as **`static`**
  - Global and static variables are initialized to zero by default by the compiler
  - Allocation at compile/link time



Variable allocation and lifetime !!!

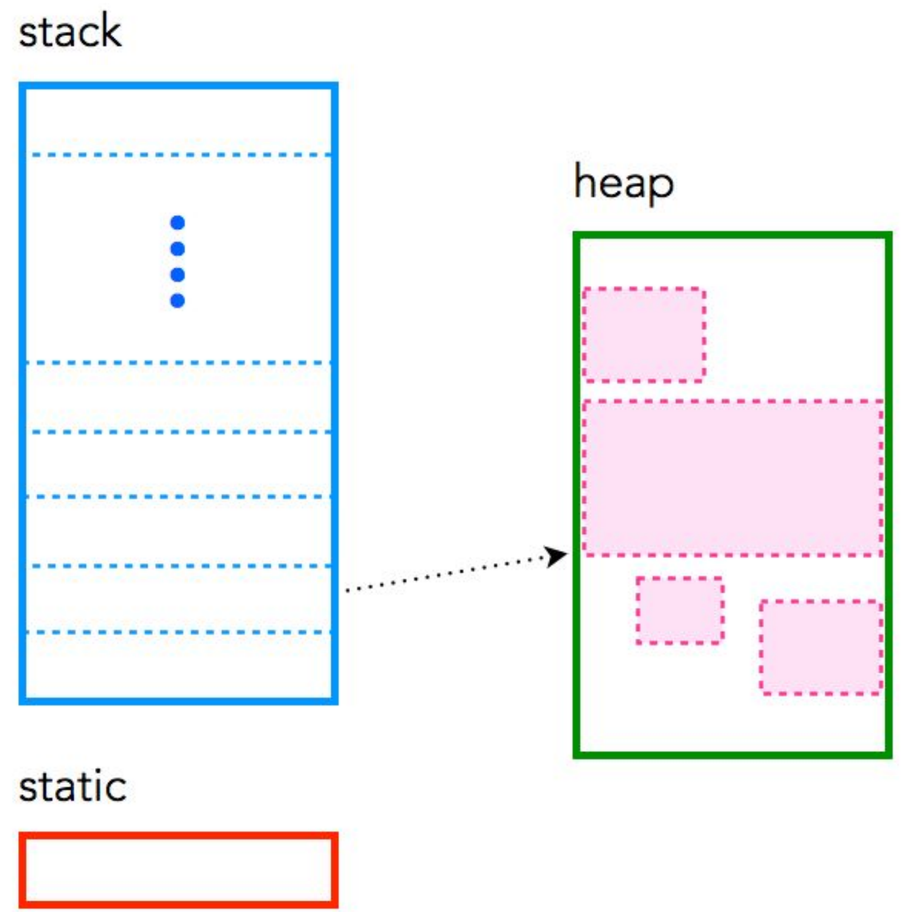
- **Stack**:
  - Variable Lifetime: specific **block**
  - Stores
    - **Local variables**: data objects declared within a **block**
    - Variables defined as **`auto`** (by default the local ones)
    - Variables defined as **`register`**, but the compiler couldn't store them to registers
  - Allocated in an
    - Automatic way
    - Continuous way
  - Allocation during execution



Variable allocation and lifetime

!!!

- **Heap**:
  - Variable lifetime: Managed by the program
  - Large pool of memory
  - Not allocated in contiguous order
  - Dynamic allocation
  - Allocation during execution



Data objects: Qualifiers

- Express additional information about a value
  - Ensure the **correct use** of the data object

- **`const:`** The value of the data object does not change once it has been initialized

- **`volatile`**: The compiler does not optimize, the data object is always accessed from the memory

- **`restrict:`** The only way to access a data object is through the pointer that points to it

Object-like Macros

- Syntax: Through pre-processor directive **`#define`**

```
#define IDENTIFIER value
```

- Function:
  - Gives a symbolic name to a data object
  - Convention: Symbolic names are given in UPPERCASE

- Examples
  - **`#define PI 3.1416`**
  - **`#define NOM ”foo”`**

Other reserved constants

- Escape sequences:
  - The character **'\n'** is a new line (on Unix systems)
  - The character **'\b'** is a backspace
  - The character **'\t'** is a tab space
  - The character **'\0'** is null character

- The double quotes **"…"** defines a string of characters
  - There is NO string type in C
  - String = array of characters that ends with the null character (**\0**)

- The simple quote **'.'** defines a character

Basic components of C language: Libraries

Libraries

- In C, there are a lot of functions already implemented into libraries
  - Include system calls
  - Mathematical computations
  - …

- The programmer can and should use them !
  - Add corresponding directive to the head of the compilation file
  - `#include <stdio.h>`
  - `#include <stdlib.h>`
  - `#include <math.h>`
  - `#include <string.h>`
  - …

Two basic I/O functions!

- We have two main functions for read and write to the standard input and ouptut:
  - Write: **`int printf(format,`**$e_1, e_2, \ldots, e_n$**`);`**
  - Read: **`int scanf(format,`**$a_1, a_2, \ldots, a_n$**` );`**

- To use them we need to add the directive

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

Formated write: printf(...)

- Syntax
  - `int printf(format,`e_1`,`e_2`,…,`e_n`);`
- Description
  - Writes (displays) the parameters $e_1, e_2, \ldots, e_n$ to the standard output based on the parameter **`format`**
  - The **`format`** consists of I/O expressions and, potentially, text.
  - Each I/O expression describes how the writing of **one** parameter e_i will be done using the character **`%`**
  - It can have a variable number of parameters, but
    - ✘ The number of e_i has to correspond to the number of I/O expressions specified in the **`format`**
- Result
  - Number of displayed characters (value < 0 in case of an error)

Formated read: scanf(…)

- Syntax
  - **`int scanf(format,`**$a_1, a_2, \ldots, a_n$**`);`**
- Description
  - Reads from the standard input (keyboard) the values and interprets them based on the parameter **`format`**
  - It assigns each read value to the variables, whose addresses are given by the parameters $a_1, a_2, \ldots, a_n$
    - Address of a variable is noted by the operand **`&`** **`&nom_variable`**
  - It can have a variable number of parameters, but
    - The number of a_i has to correspond to the number of I/O expressions specified in the **`format`**
- Result
  - Returns the number of elements correctly assigned

Format of I/O expressions

- **`%d`**: implies that the I/O is a signed integer (**`int`**)
- **`%c`** : implies that the I/O is an ASCII character (**`char`**)
- **`%f`** : implies that the I/O is a float (**`float`**).
- **`%lf`** : implies that the I/O is a double (**`double`**).
- **`%x`**: implies that the I/O is a unsigned integer to be printed using the hexadecimal representation
- **`%s`** : implies that the I/O is a string
- ….
- More info: **`man format`** or google **`printf`**
- Remarks
  - The format can also have normal text (whatever does not have the `%` character)

Example

```
int main()
{
    int i;
    float f;

    printf("Entrez un entier :");
    scanf("%d",&i);

    printf("Entrez un nombre réel :");
    scanf("%f",&f);

    printf("Vous avez saisi %d et %f \n", i, f);
}
```

What happens if we use a wrong format?

UNIVERSITÉ DE RENNES 1

- Usually we have only a compiler warning
- Question : What prints the following program?

```
#include <stdio.h>
void main()
{
	int a = 1654 ;
	printf("%s\n",a);
}
```

- Answer: Segmentation fault (core dumped) or unexpected behaviour
  - Tries to print a string of characters stored in the memory starting at the address 1654 and its end is indicated by the `\0` character.
    - At the end, the program will try to access an illegal address
  - Very common error!

Type conversion: Inbuilt typecast functions

- **`atof()`** : convert string to float
- **`atoi()`** : convert string to integer
- **`atol()`** : convert string to long
- **`itoa()`** : convert integer to string
- **`ltoa()`** : convert long to string

Let's…

Kahoot time !

(6 Q)

Basic components of C language: Control structures

Control structures

- Similar to the ones in JAVA and Scripts
  - NO BOOLEAN type!

- Example

```
int i=7;
while(i) {
   i--;
   ...
}

int i=7;
while(i!=0) {
   i--;
   ...
}
```

Both ways are equivalent, BUT the second one is more readable!

Conditions: If

- Syntax:

```
if (expression) {
   list of instructions1
}

if (expression) {
   list of instructions1
} else{
   list of instructions2
}
```

Conditions: ternary instruction

- Syntax:

```
expression1 ? expression2 : expression3;
```

- The value of a ternary instruction (like a multiplexer?) is determined as following:
  - **`expression1`** is evaluated.
  - If it is true, then **`expression2`** is evaluated and becomes the result of the entire expression.
  - If it is false, then **`expression3`** is evaluated and its value becomes the result of the expression.

Conditions: switch - case

- Syntax:

```
switch(expression) {
   case constant-expression1:
         list of instructions1 ;
         break ;                              (optional)
   case constant-expression2:
         list of instruction2 ;
         break ;
   default:                                   (optional)
         list of instructions3 ;
}
```

Loops

- Syntax while

```
while (expression) {
   list of instructions ;
  }
```

- Syntax do-while

```
do {
   list of instructions
  } while (expression) ;
```

- Syntax for

```
for(expression1; expression2; expression3) {
   list of instructions ;
}
```

  - **`expression1`**: initialization
  - **`expression2`**: condition of termination
  - **`expression3`**: step

Loops: Control statements

- Change execution from its normal sequence

- Instruction **`break;`**
  - Terminates the **`loop`** or **`switch`** statement
  - Transfers execution to the statement immediately following the END of the loop or switch.

- Instruction **`continue;`**
  - Causes the loop to skip the remaining of its BODY
  - Immediately retests the condition expression before reiterating.

- Instruction **`goto label;`**
  - **NEVER USE goto ! (except special cases)**

Other types of objects

- Derived data types:
  - Pointer type
  - Array type
  - Structure type
  - Union type
- Enumerated types:
  - Arithmetic types used to define variables
  - Variables can only assign certain discrete integer values.
- Function type
- `Void` type

CM 5:
Pointers and Arrays

Pointer type

- Syntax:

  `type * identifier;`

- Function:
  - The pointer type, `*`, is applied over a specific **`type`** of an object
  - The **`type`** can be anything: integer, character, float, pointer, structure, function etc.
  - The **`identifier`** points (refers) to an objet of that **`type`**
    - It contains the **address** of the object that it is pointing to

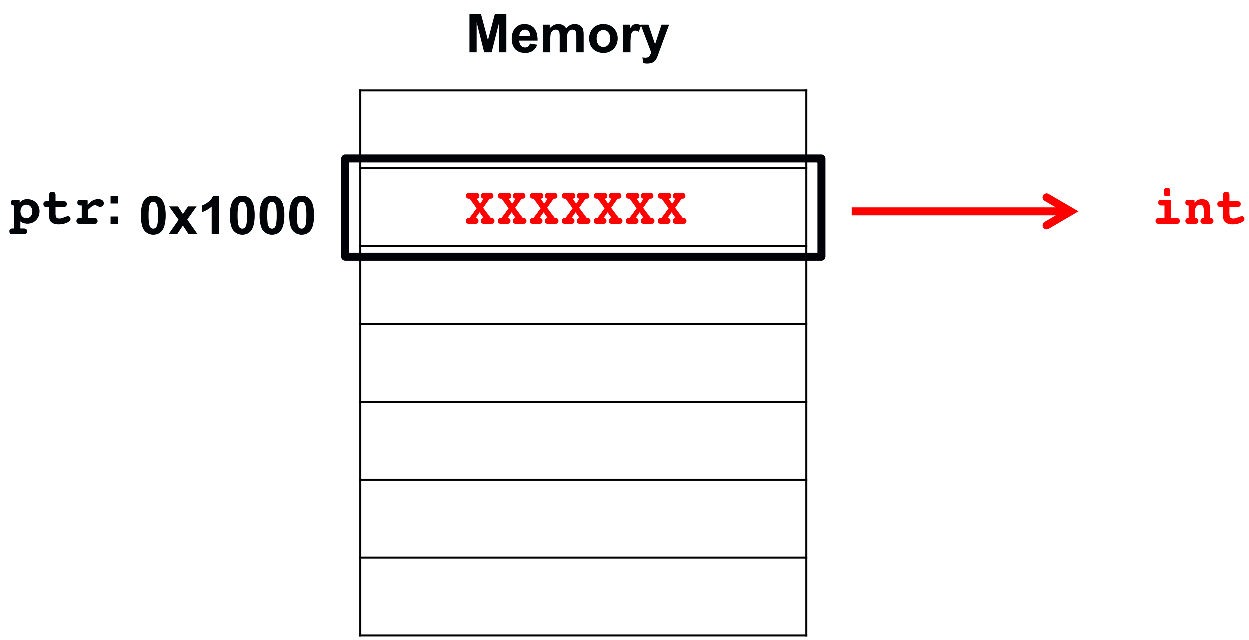
Pointer

- Useful
  - Mechanism of indexing
  - List structures
  - Passing arguments to a function
  - Dynamic allocation

- Often associated with **`NULL`**
  - **`NULL`** is a value of pointer that points to nothing (nothing is an invalid address)
  - **Attention CAPITAL letters!!**

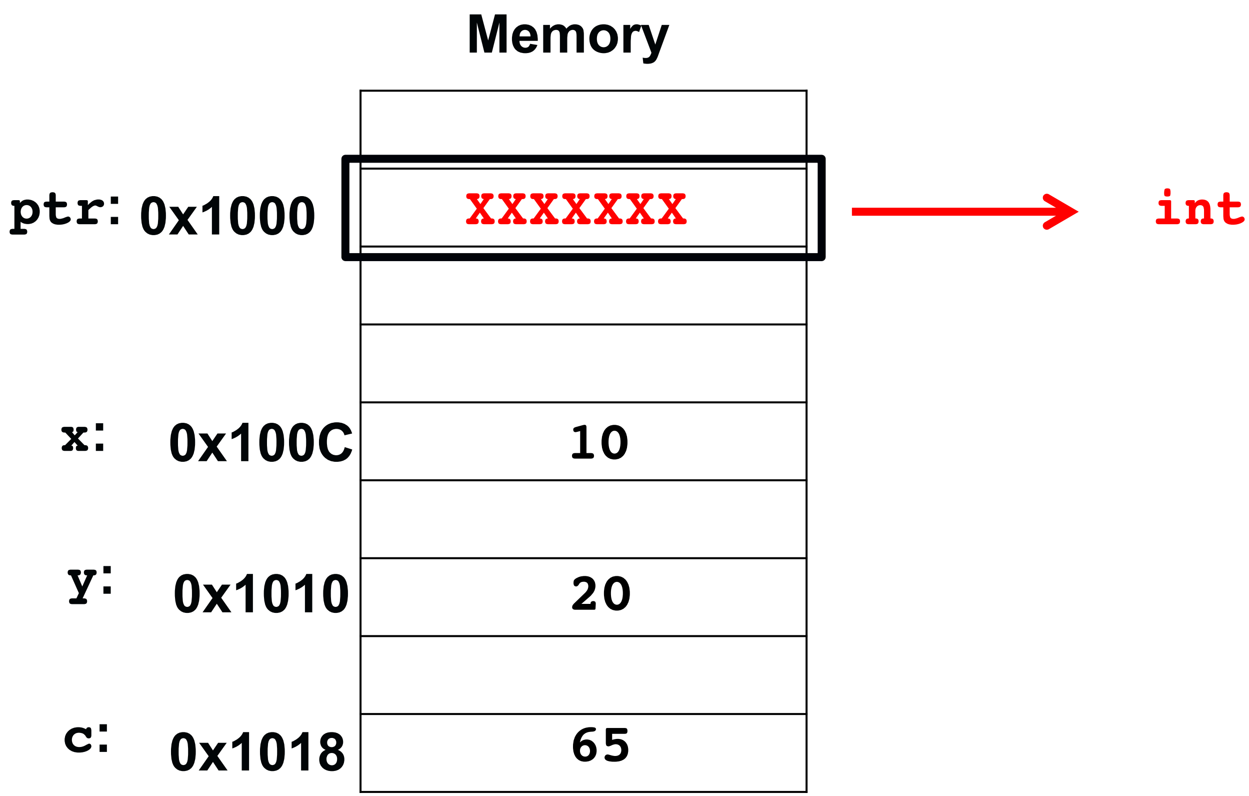
Pointer: Definition

```
int *ptr;
```



Pointer: Definition

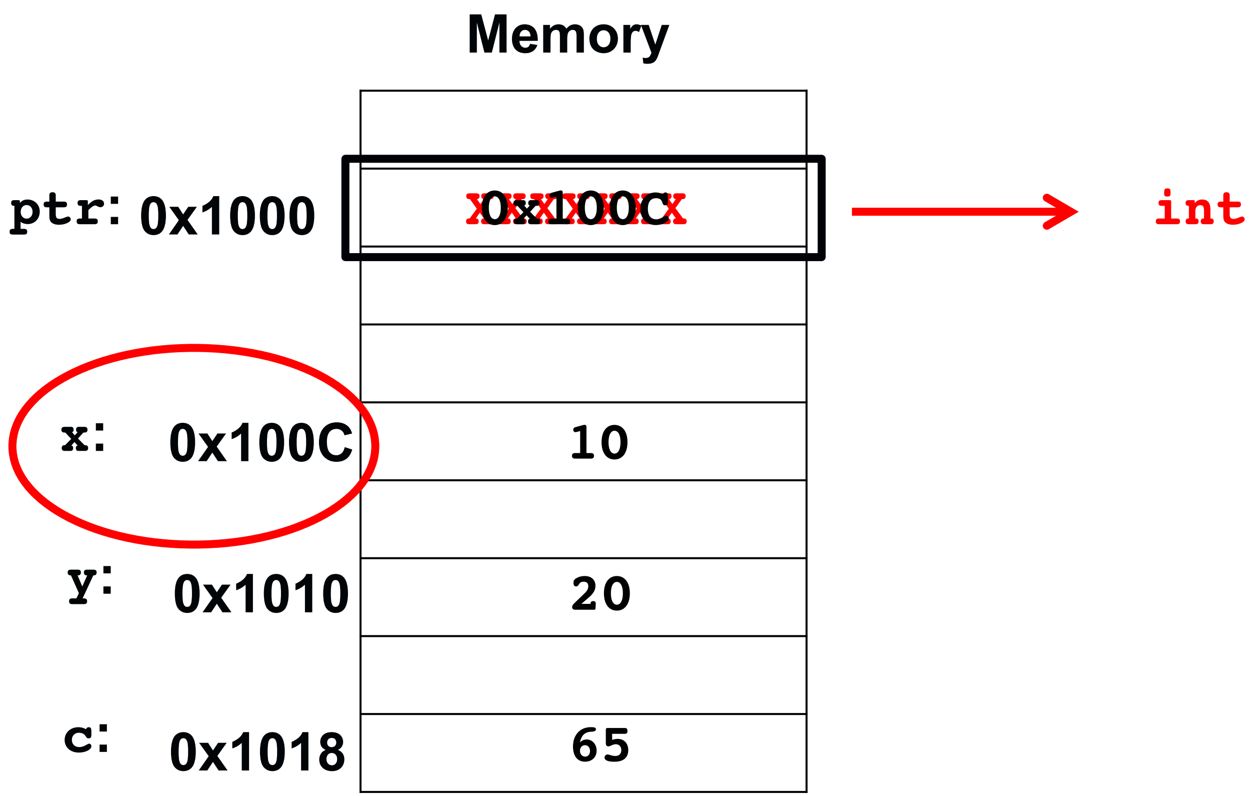
```
int *ptr;
...
int x = 10;
...
int y = 20;
...
char c = 'A';
```



- The object **`ptr`** is declared as a pointer to integers (**`int*`**)
  - It can contain the address of **`x`** or to **`y`** (both are **`int`**), but not to **`c`** (as it is a **`char`**)

Pointer Operations: Address reception

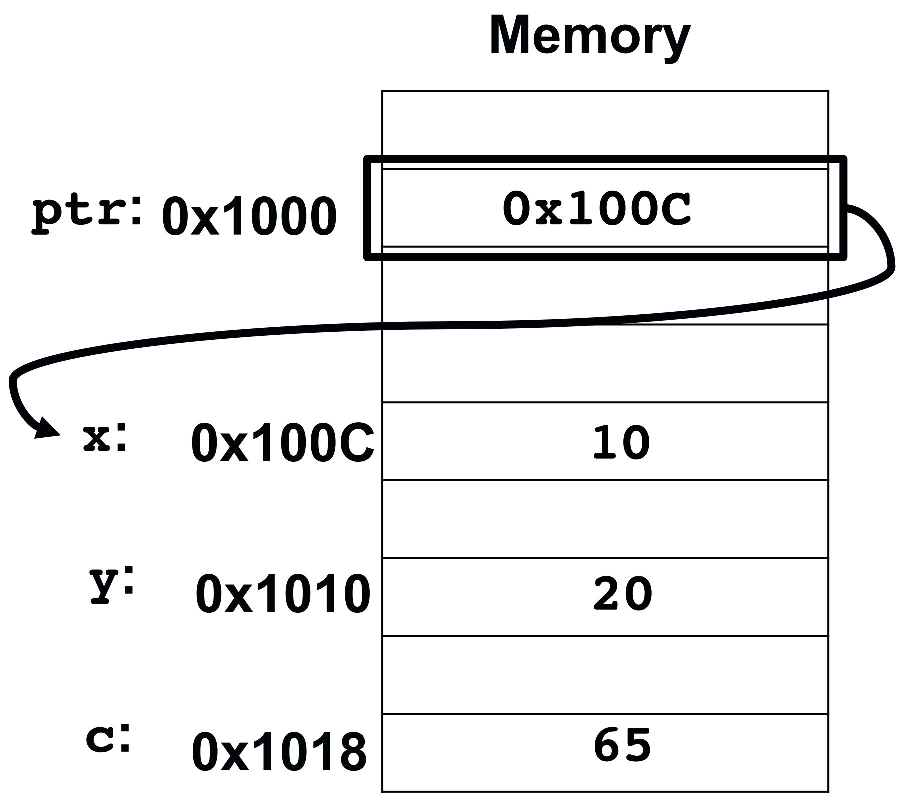
```c
int *ptr;
...
int x = 10;
...
int y = 20;
...
char c = 'A';

ptr = & x;
```



- Address reception (noted as **&**) returns the address of an object

Pointer Operations: Address reception

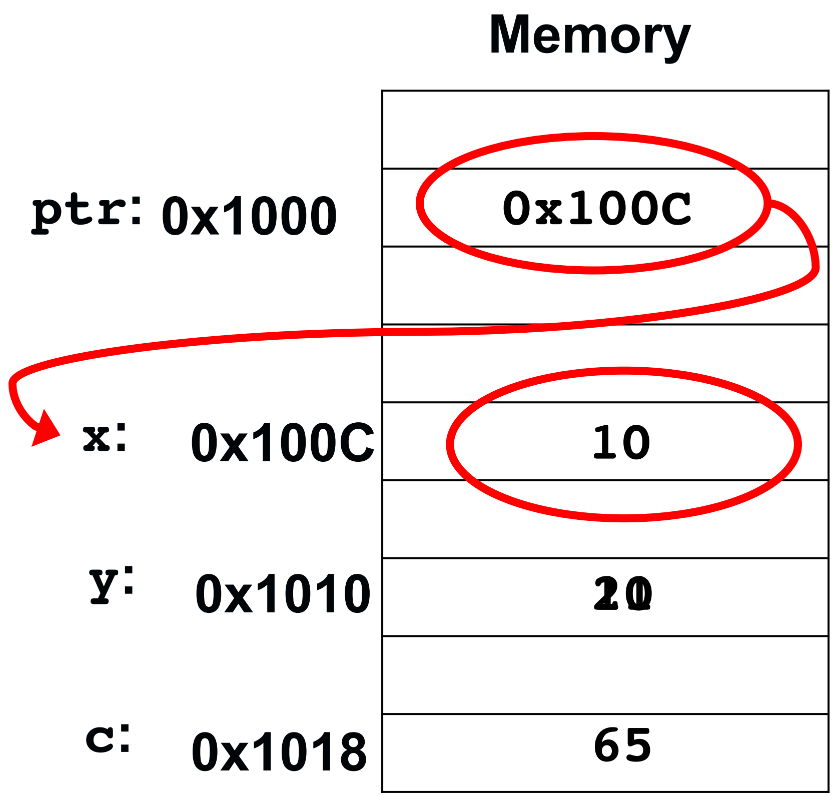
```
int *ptr;
...
int x = 10;
...
int y = 20;
...
char c = 'A';

ptr = & x;
```



- Address reception (noted as **&**) returns the address of an object

Pointer Operations: Dereference/Accessing

```c
int *ptr;
...
int x = 10;
...
int y = 20;
...
char c = 'A';

ptr = & x;
y = *ptr + 1;
```



- The dereference (noted as `*`) returns the value stored in the memory where the pointer points to

Pointer Operations: Remarks

ATTENTION:

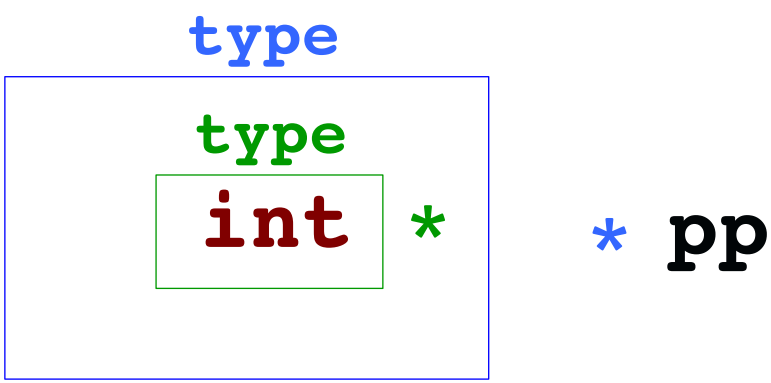
Pointer definition !=

Pointer dereference !=

Multiplication operation

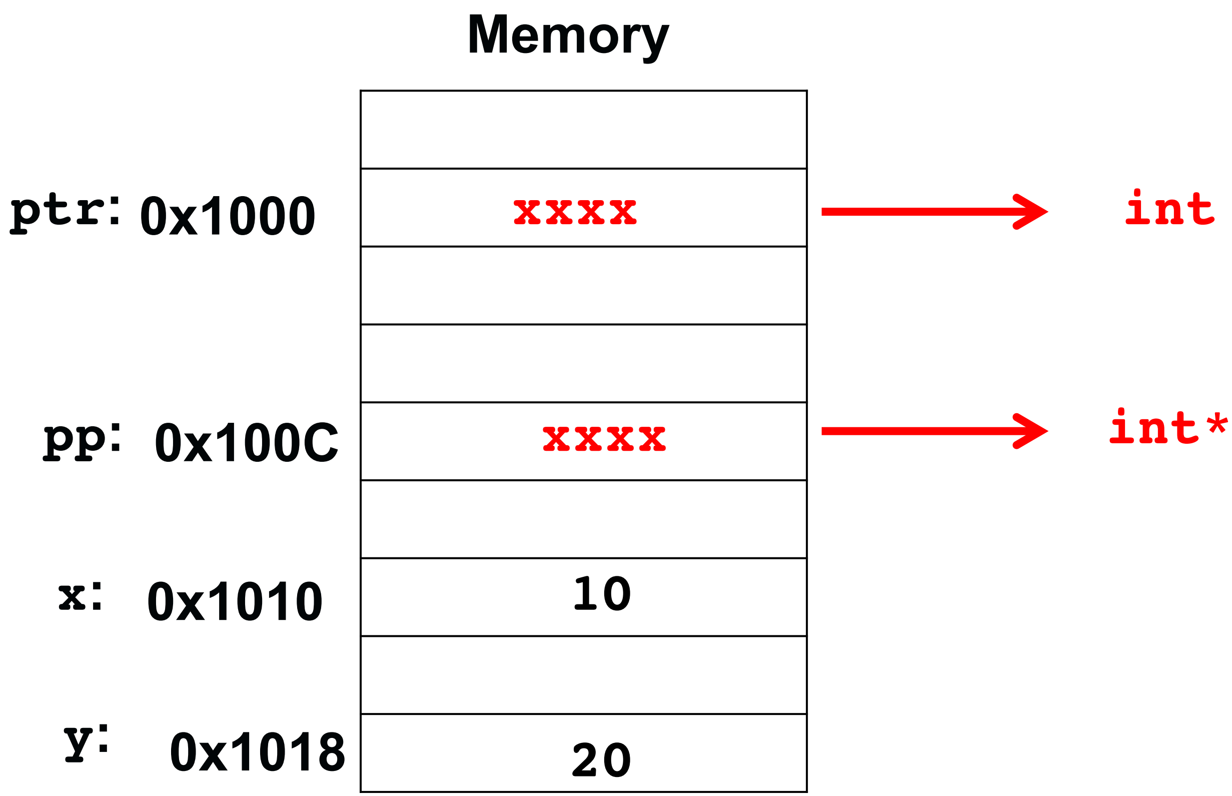
Pointer to pointer

- The object, where a pointer points to, can be also a pointer
- In this case, the **`type`** of the object is **`type*`**, i.e. pointer type
- **`int **pp;`** means that **`pp`** is a pointer that points to a pointer that points to integers



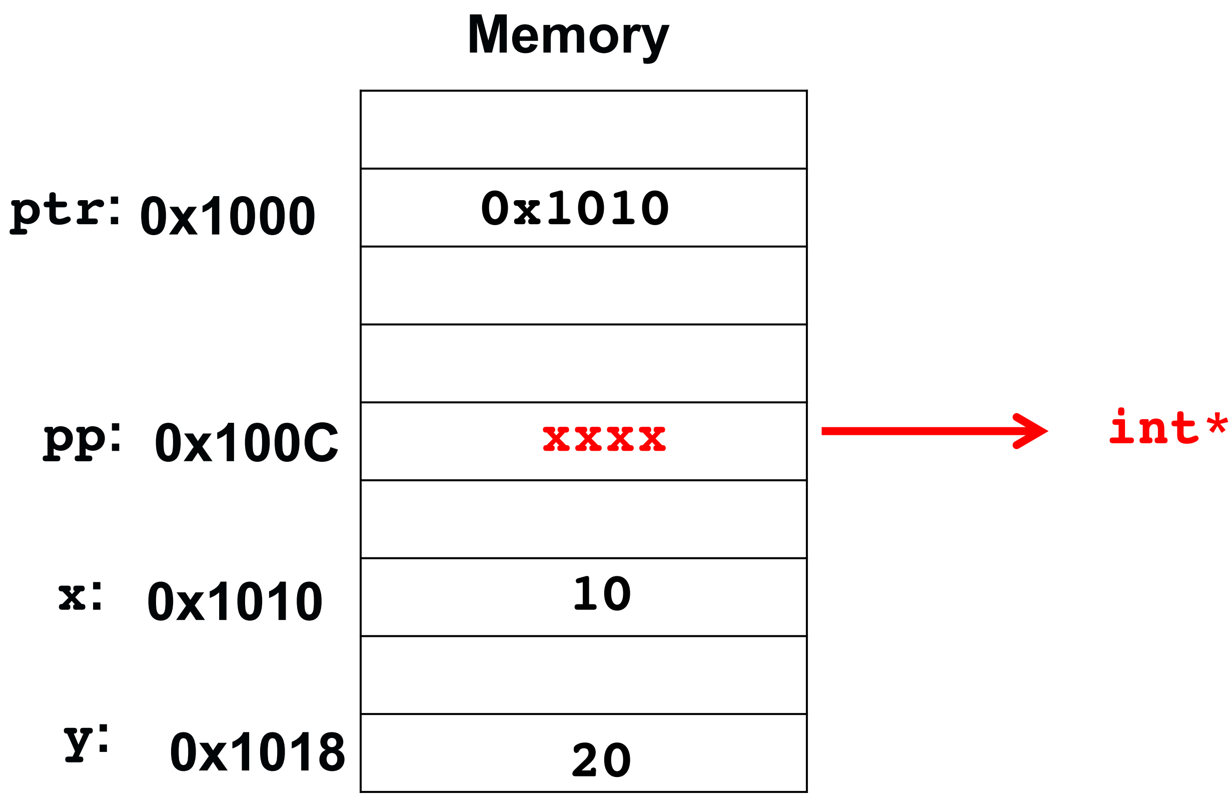
Pointer to pointer: Example

```
int *ptr;
...
int **pp;
...
int x = 10;
...
int y = 20;
```



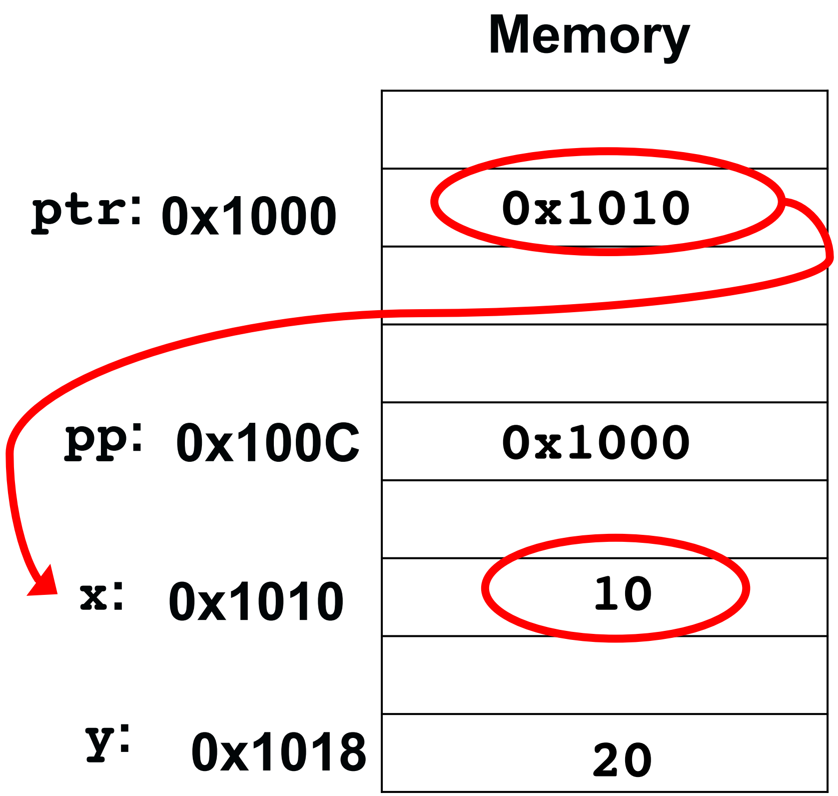
Pointer to pointer: Example

```
int *ptr;
...
int **pp;
...
int x = 10;
...
int y = 20;

ptr = & x;
```



Pointer to pointer: Example

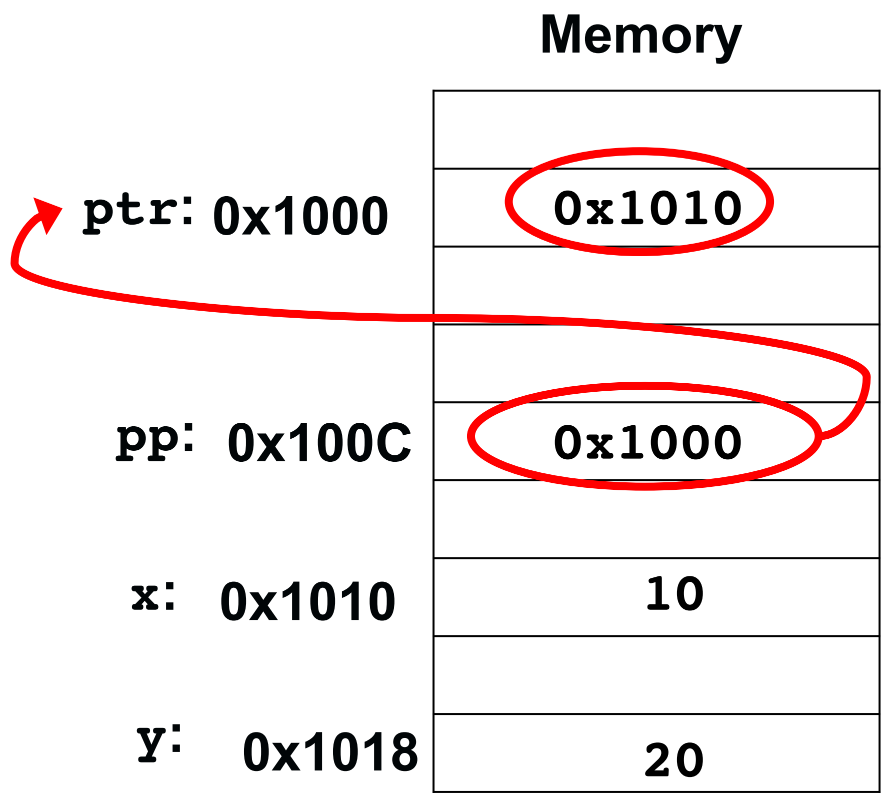
```
int *ptr;
...
int **pp;
...
int x = 10;
...
int y = 20;

ptr = & x;
pp = & ptr;
```



- What is the result of `*ptr?`

Pointer to pointer: Example

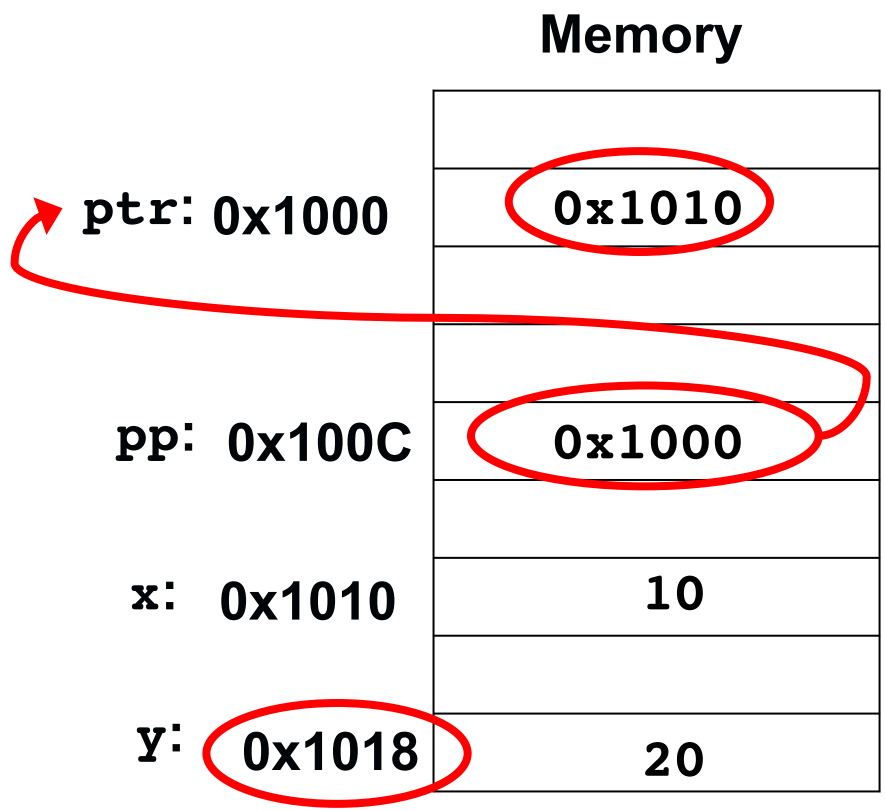
```
int *ptr;
...
int **pp;
...
int x = 10;
...
int y = 20;

ptr = & x;
pp = & ptr;
```



- What is the result of `*pp?`

Pointer to pointer: Example

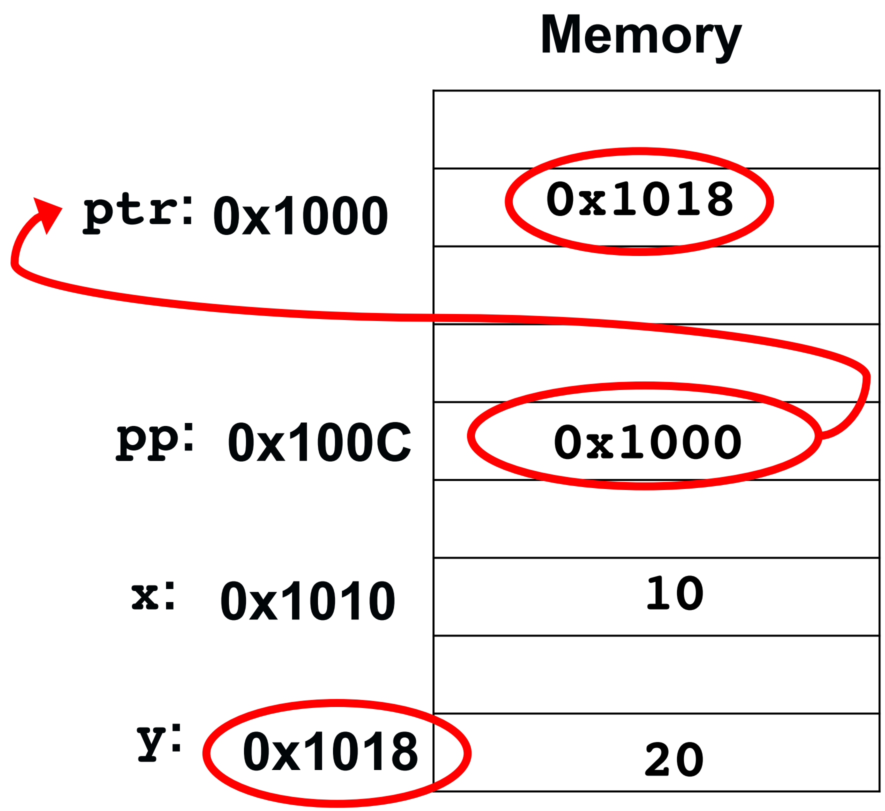
```
int *ptr;
...
int **pp;
...
int x = 10;
...
int y = 20;

ptr = & x;
pp = & ptr;
```



- What is the result of `*pp = &y` ?
- The instruction `*pp=&y` modifies the content of the object pointed by **pp** (which is **ptr**) by storing the address of the object **y**

Pointer to pointer: Example

```c
int *ptr;
...
int **pp;
...
int x = 10;
...
int y = 20;

ptr = & x;
pp = & ptr;
```



- What is the result of **`*pp = &y`** ?
- The instruction **`*pp=&y`** modifies the content of the object pointed by **`pp`** (which is **`ptr`**) by storing the address of the object **`y`**

Pointer to pointer: Example

```c
int *ptr;
...
int **pp;
...
int x = 10;
...
int y = 20;

ptr = & x;
pp = & ptr;
```

Memory

`ptr`: 0x1000	`0x1018`
`pp`: 0x100C	`0x1000`
`x`: 0x1010	`10`
`y`: 0x1018	`20`

- What is now the result of `*ptr?`

Kahoot time !!

Q1 – Q6

Pointer arithmetic

- A pointer is **NOT ONLY** an **address**!

- It provides information about
  - What **`type`** is the object stored in the address where the pointer points to
  - And thus, what is the size of the object.
    - Data type's size can be given by the **`sizeof()`** operator.

      `sizeof(int), sizeof(short)`

    - The return value depends on the compiler and the target architecture!

Pointer arithmetic operators

`type *ptr;int x;`

Operator	Description
`ptr + x`	Adds `sizeof(type) * x` to the pointer value
`ptr – x`	Subtracts `sizeof(type) * x` to the pointer value ptr
`ptr ++`	Increases the pointer value by `sizeof(type)`
`ptr --`	Decreases the pointer value by `sizeof(type)`

- Example (assuming `sizeof(int) == 4`).

`int *ptr;`

`ptr`: 0x1000

`ptr+1` → 0x1004

`ptr+2` → 0x1008

Pointer arithmetic: Problems!

- You can make a pointer **point anywhere**, the compiler will **NOT complain**. The following are valid:
  - `ptr + 10000`
  - `ptr – 10000`
  - `char *ptr = (char *) 0x0000ffff`

SECURITY ISSUES

- However, problems occur when you attempt to **access an invalid memory** address by dereferencing the pointer

- Usually, the operating system forbids access outside of the program's allocated memory
  - If the program **can access** the memory location, the data is not valid
  - If the program **cannot access**, it results to segmentation fault
    - The operating system sends the SEGFAULT signal to the process that caused the memory violation, the process core dumps and terminates

Array type

- Syntax:

```
type identifier[SIZE];
```

- Function:
  - The **`identifier`** is an array with a size equal to **`SIZE`**
  - Each array element is a object of type **`type`**
  - The array **is statically allocated**. During definition
    - The **`SIZE`** must be is a **constant**
    - The memory to store **`SIZE`** elements of type **`type`** is reserved
    - The name of the array (**`identifier)`** is **a constant pointer** that points to the address of the first element of the array
  - The index of the array is an integer
    - Values from **`0`** to **`SIZE-1`**

- It can be initialized during its definition (**`SIZE`**=elements)

```
int array[]={1,3,-2,19,-79};
```

Array type: Definition

```
int arr[10];
```

arr == & arr[0]

***arr == arr[0]**

Arrays and Pointers

- The notion of array is very close to the notion of pointer
- `arr[i] == *(arr+i)` , so they indicate the same element !
- `arr+i = arr + sizeof(type)*i`

```
int arr[10];
```

Memory

		Memory
arr →	arr[0]:0x1000	0
	arr[1]:0x1004	1
	arr[2]:0x1008	2
arr+3 →	arr[3]:0x100C	3
		4
	...	5
	`*(arr+3)`	6
		7
		8
	arr[9]:0x1028	9

`arr` is a constant pointer

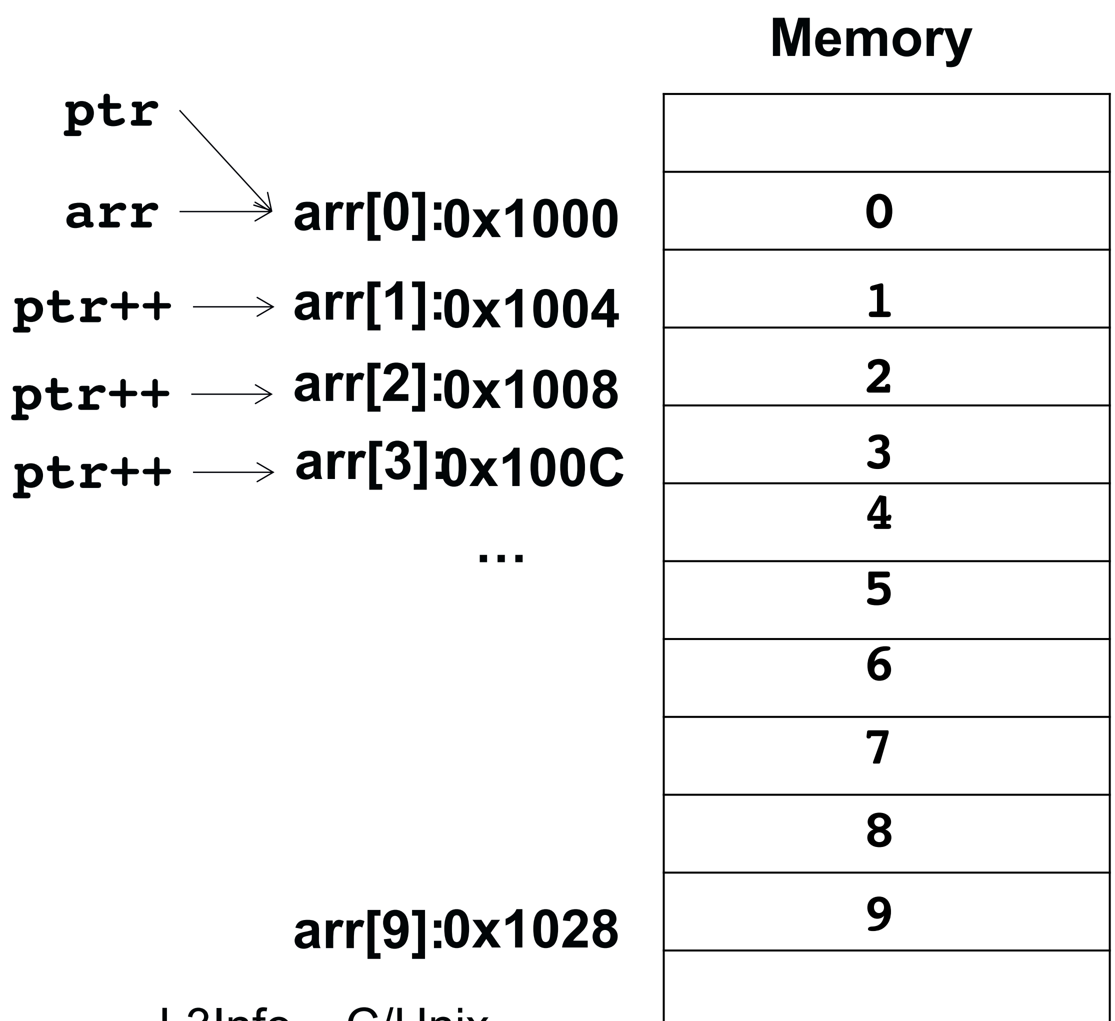
→

`arr++` is not allowed

Arrays and Pointers

- We can use a pointer to point to the array

```
int arr[10];
int *ptr = arr;
```



Arrays and Pointers

- What is the difference ?

- When we define an array:
  - We **reserve one memory block** equal to the size required to store the values of **all the array elements**
  - The array name is a constant pointer, its value cannot be modified

- When we define a pointer:
  - we do **NOT reserve** the memory where we will store the values of the **array elements**,
  - We reserve **ONLY the memory**, where we will store the **pointer**
  - The pointer is a variable, its value can be modified

An array of pointers

- The **`type`** of the elements of an array can be also pointers, that is of type **`type*`**
- Example:

```
int *arr[10];
int **ptr = arr;
```

- An element in the array **`arr`** (e.g. **`arr[0]`**) is of type **`int*`**
- The constant pointer **`arr`** is the address of the first element of the array
- Hence, the **`ptr`** is a pointer to a pointer of integers

```
ptr+i = ptr + sizeof(int*) * i
```

An array of characters: String

- There is **no string type** in C
- A string is an array of characters finishing with `'\0'`

```
char string[SIZE]
```

- The `SIZE` must be large enough to hold this additional byte!
  - Compiler puts the `'\0'` character at the first empty element of the array
  - No automatic array bound checking!
    - If number of characters and `'\0'` > array maximum size
      - **UNDEFINED BEHAVIOUR**
- Responsibility of the programmer:
  - content of the string <= size reserved in the array

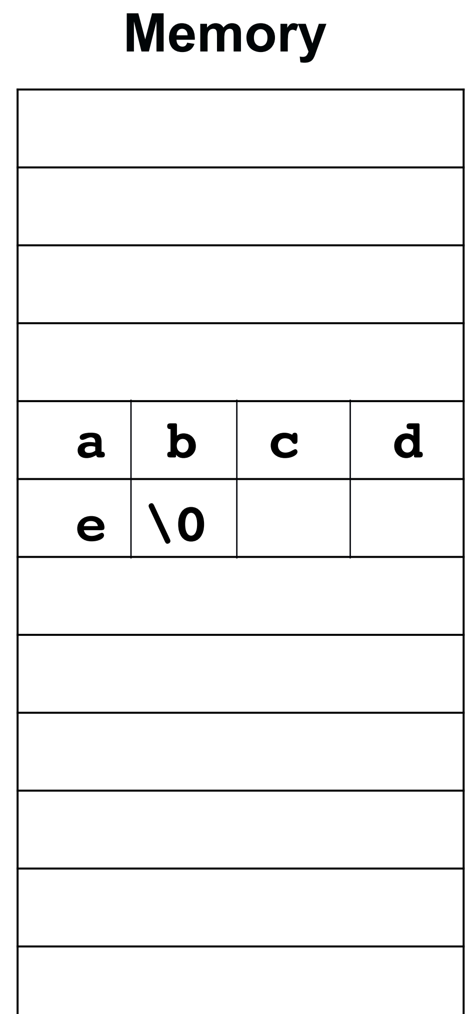
This is the source of
A LOT of ERRORS !

SECURITY ISSUES

String initialisation

- `char c[] = "abcde";`
- `char c[6] = "abcde";`
- `char c[]={'a','b','c','d','e','\0'};`
- `char c[6]={'a','b','c','d',’e','\0'};`

c → c[0]: 0x100C



String: Remarks

- Static declaration

```
char str1[]="hello";
char str2[]="world";
```

  - We CANNOT assign a string to another (**`str2 = str1;`**). They are constant pointers!
    - We have to copy one to the other!
  - Once the string is initialized, it CANNOT be assigned to another set of characters

```
str1 = "bye";
```

Arrays and Pointers: Attention !

- Example:

```
int arr[10] ;
int *ptr1, *ptr2 ;
ptr1 = arr + 3 ; //ptr1 == & arr[3]
ptr2 = ptr1 - 2 ; //ptr1 == & arr[1] ;
```

- Pointer **`ptr1`** points the fourth element of the array
- Pointer **`ptr2`** points two elements back from **`ptr1`**.

- Still, within the memory block of the array **`arr`**.

Arrays and Pointers: Attention !

- Example:

```
int arr[10] ;
int *ptr1, *ptr2 ;
ptr1 = arr + 30 ; //ptr1 == & arr[30]
ptr2 = ptr1 - 2 ; //ptr1 == & arr[28]
```

- Now, pointer **`ptr1`** points the outside the memory block of the array !
- There **is no compiler error**, the compiler still computes the address!
- There is **no core dump**, if we are **accessing** the address space dedicated to the program!
- The data that we will read through **`*ptr1`** are not valid!

Let's…

Kahoot time !!

Q7 – Q11

2D Array: Array of Arrays

- Syntax:

```
type identifier[ROWS][COLS] ;
```

- A 2D array named **`identifier`** is defined as
  - a number of **`ROWS`**
  - Each row is an 1D array of **`COLS`** elements of type **`type`**

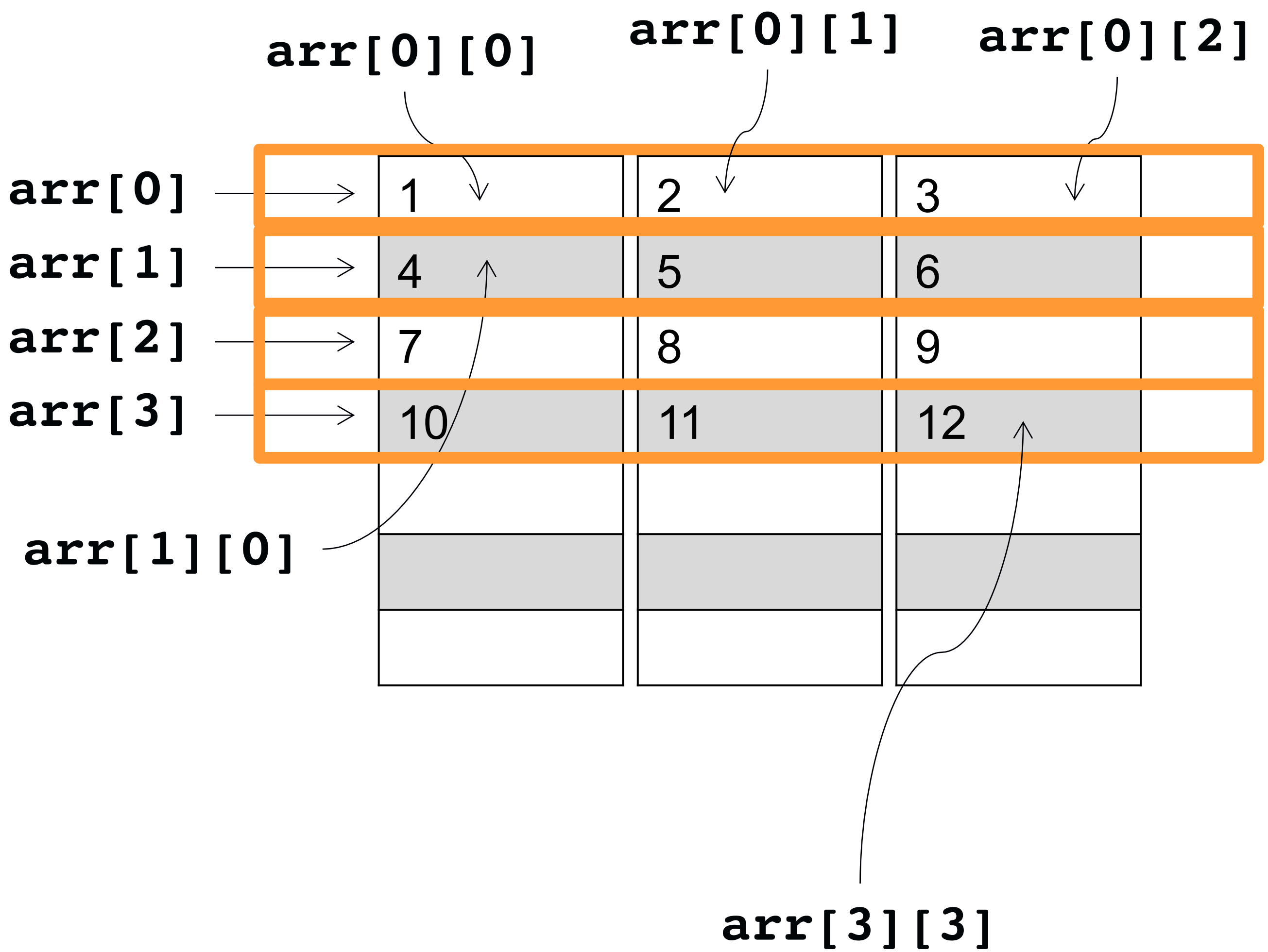
- When initializing during definition:
  - It is always required to give the **`COLS`** dimension
  - The number of **`ROWS`** is NOT always required
    - it can be calculated based on the number of columns and the initialization.

Array of Arrays: Initialization

- ```c
  int arr[3][2] = {
                      { 1 , 4 },
                      { 5 , 2 },
                      { 6 , 5 }
                  };
  ```
- `int arr[3][2] = { 1, 4, 5, 2, 6, 5 };`
- `int arr[3][2] = { { 1 }, { 5, 2 }, { 6 }};`
  - Not initialised elements will get 0
- `int arr[3][2] = {{ 0 }};`
  - All elements are initialised to 0
  ```

Array of Arrays

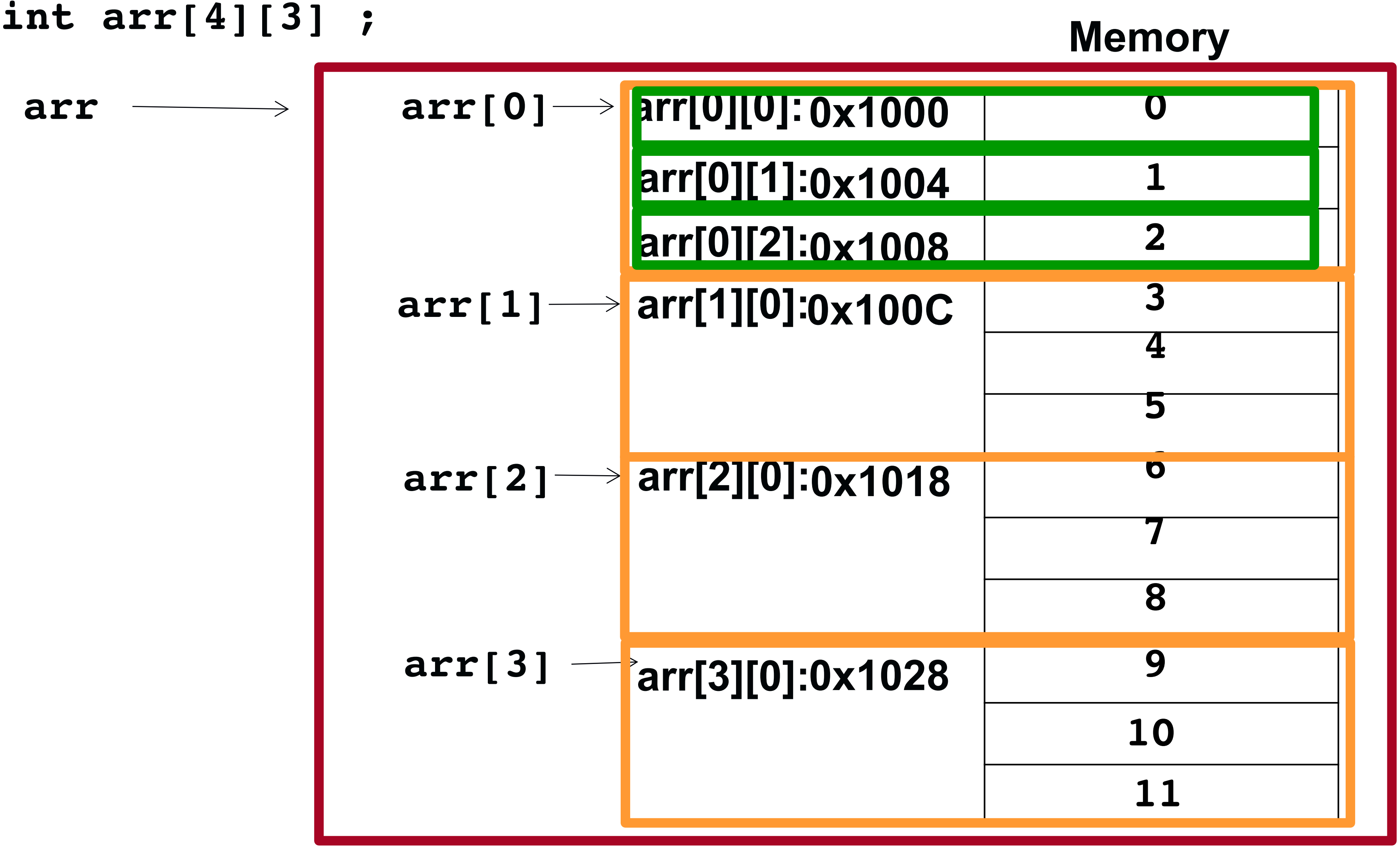
```
int arr[4][3] ;
```



Array of Arrays

- It is continuously stored in memory, row by row

```
int arr[4][3] ;
```



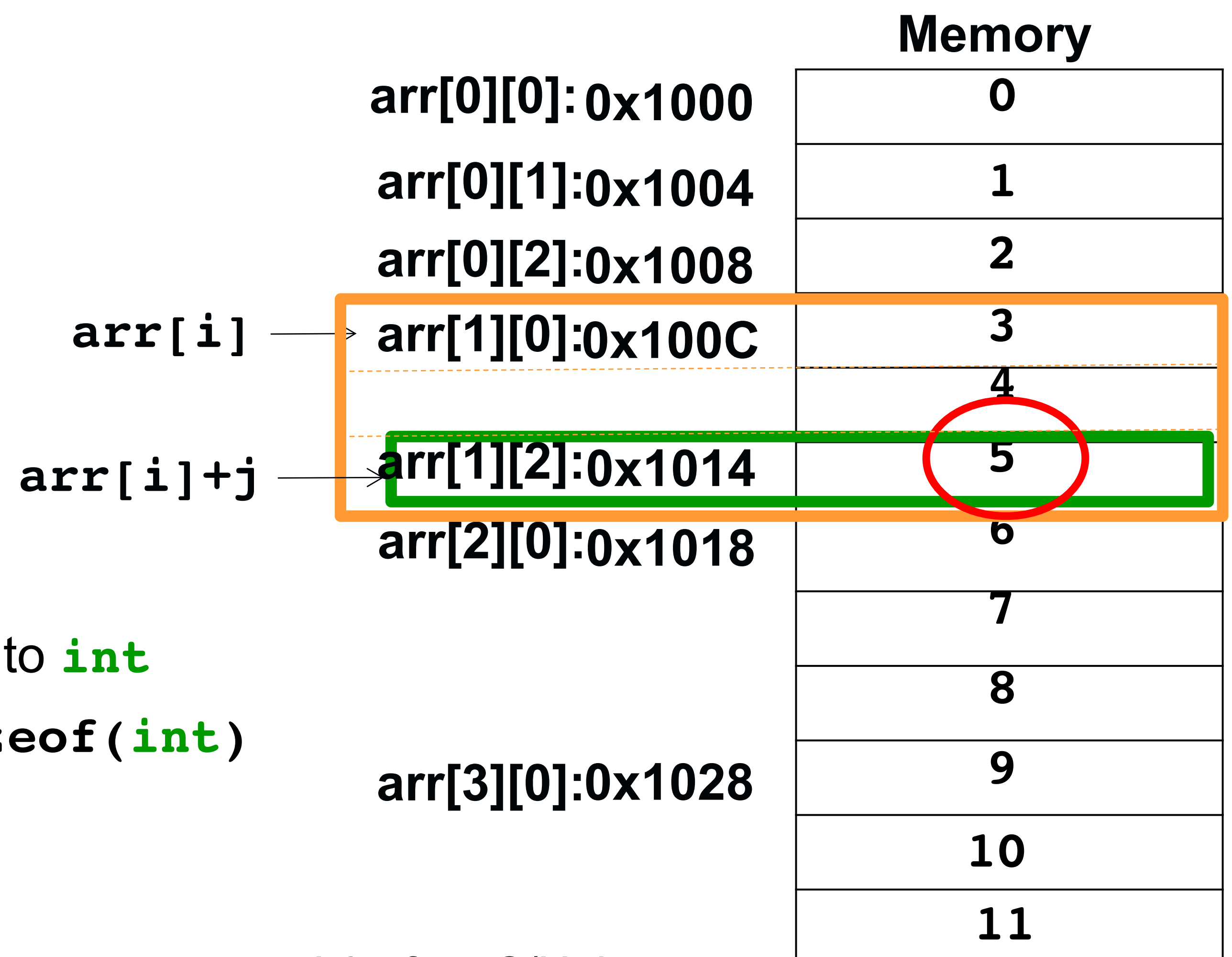
Array of Arrays: Remarks!

- The **name** of the array is a constant pointer the points to the first element of the array
  - 2 levels of hierarchy
    - Row
    - Columns -> elements
  - first level of hierarchy is the row, so it points to the first raw !
    - **`arr == &arr[0]`**
- The **name** of the array is **NOT** a pointer to pointer to **`type`**,(**`type**`**)
  - This is NOT OK with the pointer arithmetic !
- The name of the 1D arrays, i.e. the rows, point to the first element of the 1D array
    - **`arr[i] == &arr[i][0]`**

Array of Arrays: Pointer arithmetic

- `arr[i][j] = *( arr[i] + j)`

`int arr[4][3];`



`arr[i]` points to `int`

so, we add: `sizeof(int)`

Array of Arrays: Pointer arithmetic

- `arr[i][j] = *( arr[i] + j)`

  `*( *(arr + i) + j)`

`int arr[4][3];`

Memory

arr[0][0]: 0x1000	0
arr[0][1]: 0x1004	1
arr[0][2]: 0x1008	2
arr[1][0]: 0x100C	3
	4
arr[1][2]: 0x1014	5
arr[2][0]: 0x1018	6
	7
	8
arr[3][0]: 0x1028	9
	10
	11

`arr` →

`arr` points to 1D array of `int` !!!

`arr+i` →

so, we add the size of the array: `sizeof(int)*COLS`

`*(arr+i)+j` →

`*(arr+i)` is `arr[i]`
i.e. 1D array of `int`
so, we add: `sizeof(int)`

Array of Arrays: Pointer arithmetic

- `arr[i][j] = *( arr[i] + j)`
  `*( *(arr + i) + j)`
  `*( (*arr) + i * COLS + j )`

```
int arr[4][3];
```

`arr` →

`arr` points to 1D array of `int`

`*arr+i*COLS` →

Manually add the size of the array instead of compiler

`(*arr)+i*COLS+j` →

Memory

arr[0][0]: 0x1000	0
arr[0][1]: 0x1004	1
arr[0][2]: 0x1008	2
arr[1][0]: 0x100C	3
	4
arr[1][2]: 0x1014	5
arr[2][0]: 0x1018	6
	7
	8
arr[3][0]: 0x1028	9
	10
	11

Pointer to an array

```
int arr[ROWS][COLS];
int (*ptr) [COLS] ;
ptr = arr;
```

- **`(*ptr)`** is a pointer to an 1D array of **`COLS`** elements. The parentheses are NOT optional !
  - Without the parentheses, **`ptr`** becomes an array of **`COLS`** pointers, not a pointer to an array of **`COLS ints`**
- Doing pointer arithmetic, we can compute correctly the address of **`ptr[row][col]`** :

```
ptr[row][col] = ptr + sizeof(int) * COLS * row
                    + sizeof(int) * col
```

Kahoot time!

Q12 – Q19

CM 6:
Functions

Function body

- Syntax

```
type identifier (type1 parameter1, type2 parameter2, …)
{
    …
    return expression;
}
```

- Function
  - Subroutine to performs a specific job with a unique name (**`identifier`**)
  - The functions are defined at the same level as main
  - Can be used (called) using its name several times from different positions
  - Information
    - can be passed to the function through the function parameters
    - can be returned to the position where the function was called through the **`return`** statement
  - The type of the function equals to the type of the value returned
    - If no function type is defined, then it is by default **`int`**
    - If no return value, then the function type is **`void`**

Return value

- Syntax:
  `return expression;`

- The possible types that can be returned by the evaluation of the expression are
  - The primitive data types
  - The structures
  - The pointers
  - Attention: We **CANNOT return arrays**! We can return a pointer **`char*`**, **`int*`**, …
    - passed as argument during the function call
    - dynamically allocated inside the function

Exit value

- Syntax:
  `exit(int);`

- Directive:
  - **`#include <stdlib.h>`**

- Function:
  - Exits the program

- Parameters
  - 0 : usually means your program completed successfully (EXIT_SUCCESS)
  - Non zero: Different values that can be used as error codes (EXIT_FAILURE)

Function call

- Syntax

```
variable = identifier(argument1, argument2, …)
```

- Function
  - When a function call occurs, the function parameters:
    - are created (**LOCAL variables**)
    - initialized with the **value** of the passed **argument**
      - **`parameter1 = argument1`**
      - **`parameter1`** is a copy of **`argument1`** manipulated inside the function
    - the original argument is unchanged.
  - The function MUST be at least declared before it is called
    - Put the function body before the first function call
    - Put the function declaration before any function body

Function prototype

- Syntax

```
type identifier (type1 parameter1, type2 parameter2 ,…);
```

- The function declaration is the function prototype

- The complete bloc of the function is replaced by a semicolon **;**

- The name of the parameters can be omitted, but not their type

Function call: Remarks

- The **type** of the **arguments** must be the **same** as the type of the function **parameters**
- When they are **different**
  - **Automatic** and **implicit** conversion !
    - **`char`** and **`short`** (**`signed, unsigned`**) → **`int`**
    - **`float`** → **`double`**
  - Implicit conversion rules are overridden by function prototype

- The **type** of the **variable,** where the return value is assigned, must be the same as the **return type** of the function

Function: Examples

- Function bodies
  - No Arguments and No return: `void proc1() { … }`
  - Arguments and No return: `void proc2(int n, float x){ … }`
  - No Arguments and return: `int proc3(){ … }`
  - Arguments and return: `int proc4(int n, float x){ … }`

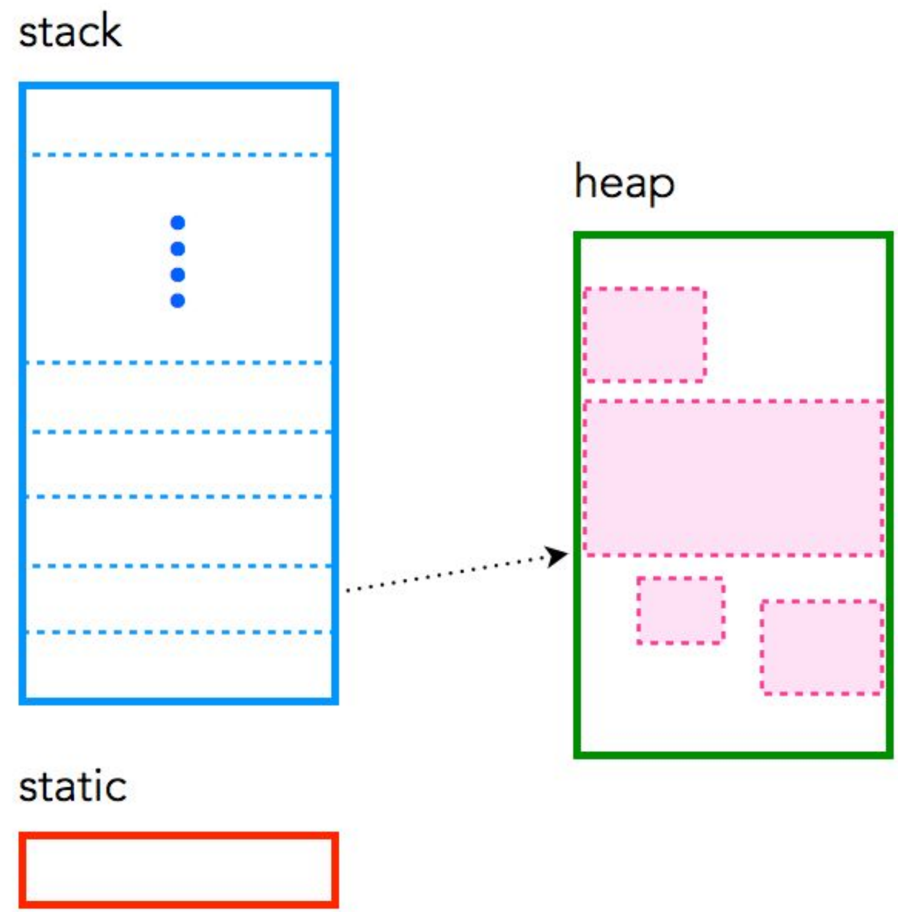
- Function calls

```
float r ;
int a ;
```

  - No Arguments and No return: `proc1();`
  - Arguments and No return: `proc2(a, r);` or `proc2(42, 3.14);`
  - No Arguments and return: `a=proc3();`
  - Arguments and return: `a=proc4(a,r);`

Variable allocation in memory

- **Static**:
  - global and static variable storage
  - permanent for the entire run of the program.
  - Allocation at compile/link time
- **Stack**:
  - local variable storage
  - automatic, continuous memory
  - Allocation during execution
- **Heap**:
  - dynamic storage
  - large pool of memory, not allocated in contiguous order
  - Allocation during execution



Stack

- Each time a function call occurs, the stack grows to store the local variables of the function
- Each time a function returns, the stack shrinks and removes the function local variables
- So, stack variables **ONLY exist** while the **function** that **created** them is **running**
- No need to manage the memory yourself, variables are allocated and freed automatically
- The stack has size limits

What is visible <u>inside</u> a function?

- The **parameters** of the function
- The **local variables** (the ones defined inside the function bloc)
- The **global variables** of the compilation file
- The imported global variables (specified by **`extern`**)
- The other functions declared before this function
  - inside the compilation file
  - external

Kahoot time!

Q1- Q5

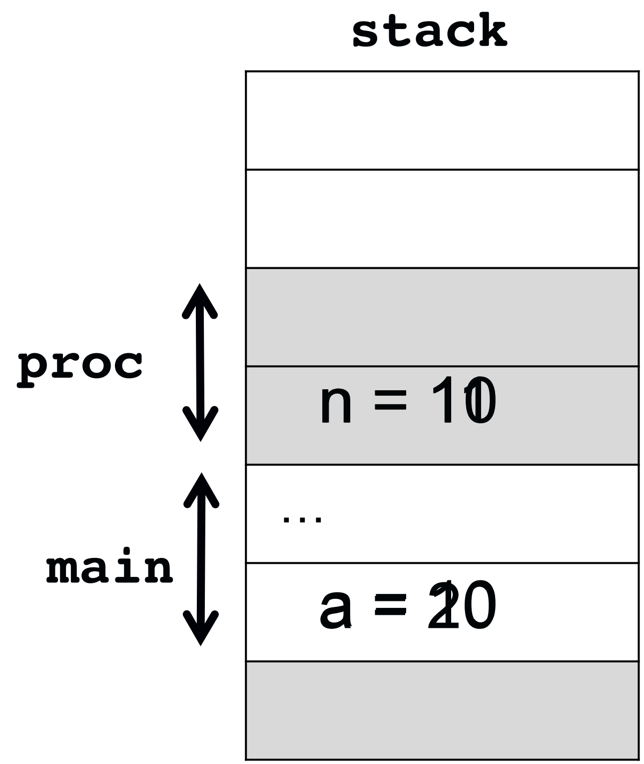
Passing arguments: By value

Function parameters: **LOCAL variables** initialized with the **value** of the passed **argument**

- **Pass by Value:** pass the data object itself
- The parameter is initialized with the **actual value** of the data object
- When a function returns:
  - The space reserved for the local variables of the function is erased!
  - With that, any modification occurred on the local variables

Pass by value

```
void proc(int n){
      n++;
}
void main{
      int a=10;
      proc(a);
      a = a + 10;
      …
}
```



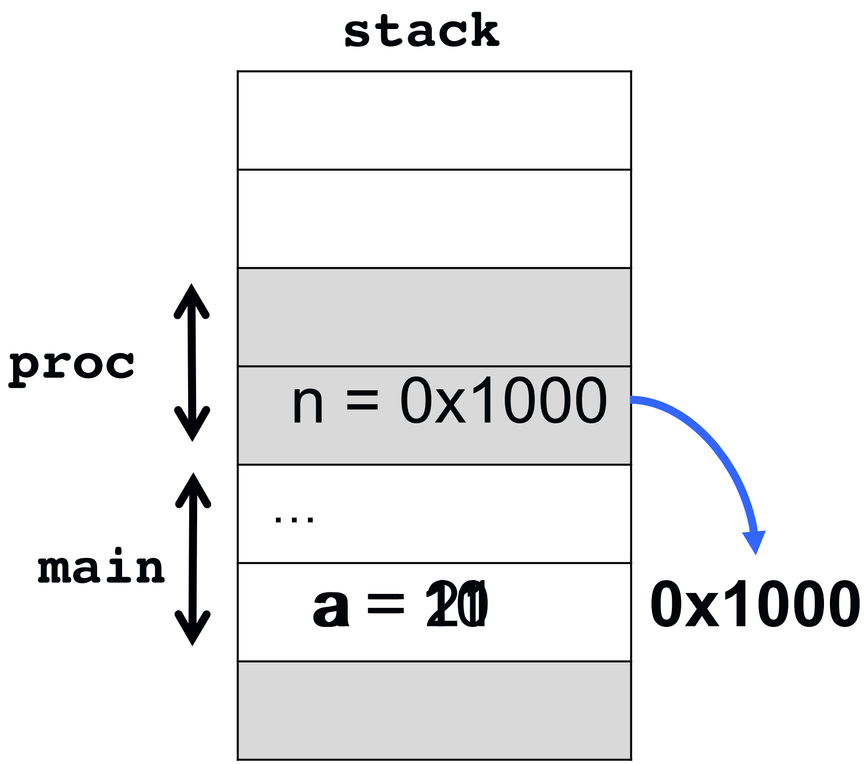
Passing arguments: by Reference

Function parameters: **LOCAL variables** initialized with the **value** of the passed **argument**

- **Pass by Reference:** pass the pointer of the data object
- The parameter is initialized with the **address** of the data object
- When a function returns:
  - The space reserved for the local variables of the function is erased!
  - But… the modifications now occurred at addresses belonging to the function that made the call !

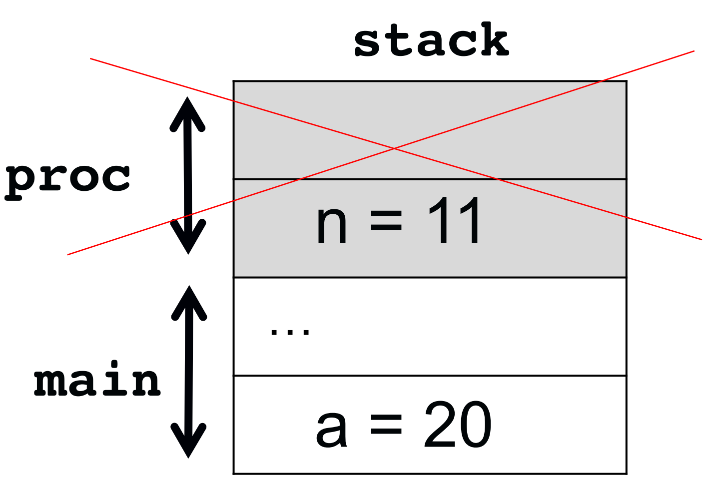
Pass by reference

```
void proc(int *n){
      (*n)++;
}
void main{
      int a=10;
      proc(&a);
      a = a + 10;
      …
}
```

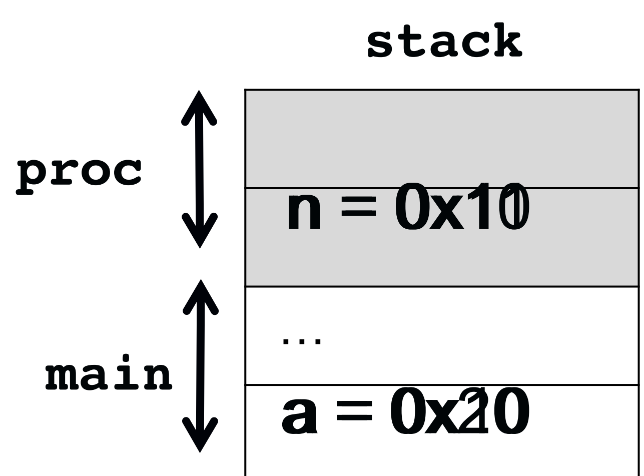


Pass by value: Pointer type

```
void proc(int n){
        n++;
}
void main{
        int a=10;
        proc(a);
        a = a + 10;
        …
}
```

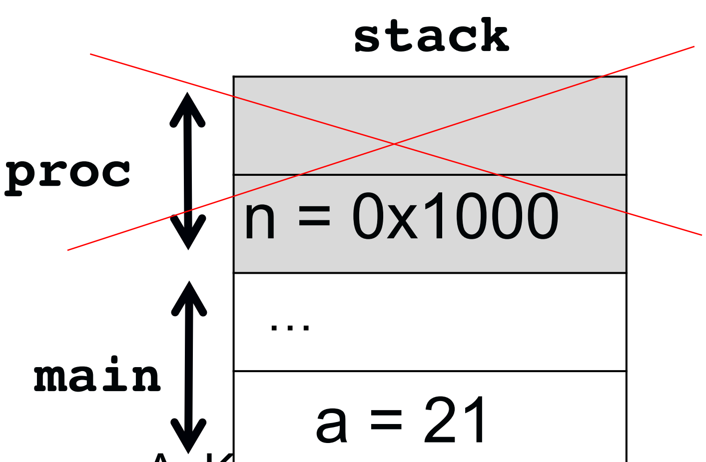



```
void proc(int *n){
        n++;
}
void main{
        int *a=0x10;
        proc(a);
        a = a + 0x10;
        …
}
```

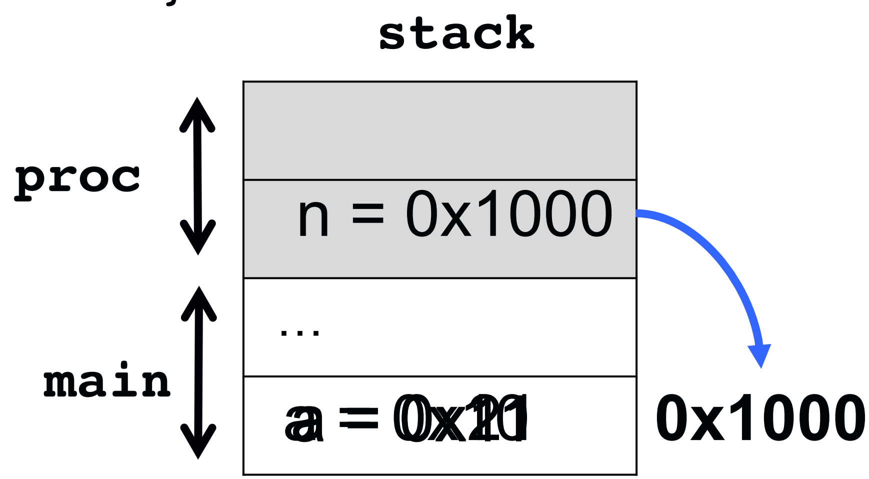


Pass by reference: Pointer

```
void proc(int *n){
        (*n)++;
}
void main{
        int a=10;
        proc(&a);
        a = a + 10;
        …
}
```




```
void proc(int **n){
        (*n)++;
}
void main{
        int *a=0x10;
        proc(&a);
        a = a + 0x10;
        …
}
```



Kahoot time!

Q6- Q10

Passing arguments: Array

- When an **array** is a function parameter, the compiler translates it as a **pointer**

```
void foo( int arr[]) { … } → void foo( int *arr) { … }
```

- So, inside the function we cannot know the size of the array! All we have is a pointer …
  - If you try to compute the array's size using **`sizeof(arr)`**, you just get the size of a pointer,
    - e.g. 4 in a code compiled for a machine with 32 bits as addresses

- It's important to pass the array size as a parameter.

```
void foo( int arr[], int size ) { … }
```

Passing arguments: Array

- The parameter equals to the address of the 1er element (**`arr=&arr[0]`**)
- Since the parameter is the address of the array, we can modify the array elements
- The result is visible outside of the function
- Example
  - Function to initialize the n values of an 1D array **arr**

```
void init_vector (int arr [], int n)
{
        int i;
        for (i=0;i<n;i++) { arr[i]=0; }
}
```

Passing arguments: 2D Arrays

- Array of arrays:
  - `int arr[][COLS]`
- Pointer to an array:
  - `int (*arr) [COLS]`
- **BUT NOT double pointer !**
  - `int **arr`
  - compiler error because it cannot compute the size!
  - compiler may NOT prevent you, BUT you may not have the expected results in some situations!

Kahoot time!

Q13-Q15

The main function

- An executable can be called with a list of arguments

`./prog arg1 arg2 … argn`

- These arguments are the parameters of the function **`main`**

- The prototype of main function:

`int main(int argc, char* argv[])`

  - **`argc`**: size of **`argv`** (number of passed arguments + 1)
  - **`argv`**: is a table of strings
    - **`argv[0]`** is a **`char*`** pointing to the name of the program
    - **`argv[1]`** is a **`char*`** pointing to the 1st argument
    - **`argv[arc-1]`** is a **`char*`** pointing to the last argument

- Remark
  - If we want to use an argument as **`int`**, we have to convert it!

  `int atoi(char* ch)` of **`stdlib.h`**

Example

```
#include<stdio.h>
int main(int argc, char* argv[])
{
    int i;
    for (i=1;i<argc;i++)
    {
        printf("l'argument n°%d vaut %s ",i,argv[i]);
    }
}
```

- What does prints execution of the command: **`./prog foo 12 bar`** ?

Library functions: Inputs/Outputs, strings and files

Input / Output functions

- Several functions to manipulate I/O.
  - To use them we have to add their directive to the head of the compilation file using **`#include`**
  - We have to provide a format that indicates the type of the objects over whom we want to perform an I/O operation

- We have seen 2 basic I/O functions:
  - Write to the standard output (**`printf`**)
  - Read from the standard entry (**`scanf`**)

Read/Write a character

- Read a character:

```
#include <stdio.h>
int getchar()
```

  - Returns **one** character read from the standard input as an `unsigned char` cast to an `int`
  - Returns **EOF** when the file ends or in case of an error

- Write a character :

```
#include <stdio.h>
int putchar(int c)
```

  - Writes a character (an `unsigned char`) specified by the argument `c` to the standard output

- Example:

```
char c;
c = getchar();
putchar('\n'); /* displays a new line */
```

String manipulation

- `#include<string.h>`

- Read and write strings: `scanf(…)/printf(…)`, `gets(…)/puts(…)`, `fgets(…)/fputs(…)`, …
- Length of string : `strlen(…)`, …
- Comparison of strings : `strcmp(…)`, …
- Copy of strings : `strcpy(…)`, …
- Formatted output to a string: `sprintf(…)`, …
- …

String declaration: Remarks

- Static declaration

```
char str1[]="hello";
char str2[]="world";
```

- We CANNOT assign a string to another (`str2 = str1;`). They are constant pointers!
  - We have to copy one to the other!
- Once the string is initialized, it CANNOT be assigned to another set of characters

```
str1 = "bye";
```

Basic I/O limitations with strings

- Basic I/O are **NOT** capable verifying **bounds**
- Example:

```
char name = "Hello"
scanf("%s", name );
printf("%s", name );
```

SECURITY ISSUES

- `scanf("%s", name );` → **`%s is dangerous!`**
  - `name` is a pointer, the starting address of the string
  - **NO verification** on how long is the information to be written!
  - Analogy: Give a pen to someone and point where to start writing
    - But then have no control! he may continue writing on the desk..
  - Solution:
    - `char name[10];`
    - `scanf("%9s", name );`

Basic I/O limitations with strings

- **`scanf`** ends after a space:
  - receiving multiword strings in **one variable**
- Example

```
char name = "Hello John"
scanf("%s", name );
printf("%s", name );
```

- Solution?
  - **`gets(name);`**
  - **`puts(name);`**

Read/Write a string

- Syntax:

```
char *gets(char *str)
```

- Function:
  - Reads a line from the **`stdin`** and stores it in the string pointed by **`str`** (stops when **`\n`** is read or **`EOF`**)
  - It does **NOT** include the ending newline character **`\n`** in the resulting string
  - It does **NOT** allow to specify a maximum size for **`name`**
    - **DANGEROUS**
- Returns:
  - **`str`** on success
  - **`NULL`**
    - Error
    - when **`EOF`** occurs, while no characters have been read.

Read/Write a string

- Syntax:

```
char *fgets (char *str, int size, FILE* file);
```

- Function:
  - Reads **`size – 1`** characters from input stream **`file`** and stores it in the string pointed by **`str`**
  - Stops whichever happens first:
    - **`size – 1`** characters have been read
    - A newline occurs, **`\n`**
    - The end-of-file is reached, **`EOF`**
  - It includes new line character **`` `\n` ``** pressed by the user
  - A terminating null character **`` `\0' ``** is automatically appended after the characters.

- Returns:
  - **`str`** on success
  - **`NULL`** :Error or when **`EOF`** occurs, while no characters have been read.

Read/Write a string

- **`int sscanf (const char *str, const char * format, ...);`**
  - Like **`scanf`** but reads data from string pointed by **`str`** as the **`format`** `specifies`

- And much other functions, check libraries documentation

Length of a string: `strlen()`

- Syntax:

```
size_t strlen(const char *str);
```

- Function:
  - Counts the number of characters of the string pointed by **`str`** until it finds the terminating null character (**`'\0'`**)
  - the terminating null character (**`'\0'`**) is NOT INCLUDED in the length of the string
  - **`size_t`** is unsigned integer type of at least 16 bit

- Returns: the length of the string

- Example

```
char ch[50]="how long am I?";
printf("%d\n",strlen(ch));
```

Concatenate strings: `strcat()`

- Syntax:

```
char *strcat(char *dest, const char *src) ;
```

- Function
  - appends the string pointed to by **`src`** to the end of the string pointed to by **`dest`**.
  - the string pointed to by **`dest`** SHOULD be **LARGE** enough to hold both strings AND the terminating character
  - The strings pointed by **`src`** and **`dest`** must **NOT overlap**!

- Return value:
  - The function **`strcat ()`** returns normally a pointer to the destination string **`dest`**.

Copy strings: `strcpy()`

- Syntax:

```
char *strcpy(char *dest, char *src);
```

- Function
  - The function **`strcpy()`** copies the string pointed by **`src`** (including the character `'\0'`) into the string pointed by **`dest`**.
  - The strings pointed by **`src`** and **`dest`** must **NOT overlap**!
  - It does **NOT** verify the **SIZE** of the strings before copying
  - No knowledge of how large **`src`** is!
  - **`src`** can be larger than **`dest`**

SECURITY ISSUES

- Return:
  - The function **`strcpy()`** returns normally a pointer to the destination string **`dest`**.
- Solution?
  - **`strncpy(…)`**

Copy strings: `strcpy()`:Example

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

void print(int i){
    char text1[10] = "Hippolyte”;
    char text2[5] = "Ares”;
    printf("The initial content of text1 is :%s \n", text1)
    if (i==0){ strcpy(text1,text2) ;
        printf("text1 has now :%s \n", text1);}
    else{
        strcpy(text2,text1) ;
        printf("text2 has now :%s \n", text2);}
                    }
                void main(){
                    print(0);
                    print(1);
                    }
```

Copy strings: `strncpy()`

- Syntax:

```
char *strncpy(char *dest, char *src, size_t n)
```

- Function:
  - Copies **UP TO** to **n** characters from the string pointed by **`src`** to **`dest`**.
  - Length of **`src < `** n → the remaining part of **`dest`** is padded with null bytes.
  - Length of **`src > `** n → **no termination** character at **`dest`**

- Return:
  - The function **`strncpy()`** returns normally a pointer to the destination string **`dest`**.

Compare strings: `strcmp()`

- Syntax:

```
char strcmp (char *str1, char *str2);
```

- Function
  - The function **`strcmp()`** compares two strings of pointed by **`str1`** and by **`str2`**.
- Return: The function **`strcmp()`** returns an integer:
  - if Return value < 0 then it indicates str1 is less than str2.
  - if Return value > 0 then it indicates str2 is less than str1.
  - if Return value = 0 then it indicates str1 is equal to str2.

Compare strings: `strncmp()`

- Syntax:

```
int strncmp(const char *str1, const char *str2, size_t n)
```

- Function
  - The function **`strncmp()`** compares at most the first **`n`** number of characters between two strings of pointed by **`str1`** and by **`str2`**.

- Return value: The function **`strncmp()`** returns an integer:
  - if Return value < 0 then it indicates str1 is less than str2.
  - if Return value > 0 then it indicates str2 is less than str1.
  - if Return value = 0 then it indicates str1 is equal to str2.

Compare strings: Example

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

void main(){
    char nom1[100] ;
    char nom2[100] ;
    int res ;
    strcpy ( nom1,   "C programming at L3 INFO – ISTIC" );
    strcpy ( nom2,   "C programming is fun" );
    res = strcmp(nom1,nom2) ;
    if (res == 0) {
        printf("%s and %s are identical\n", nom1, nom2);
    } else if (res<0) {
        printf("%s is less than %s \n", nom1, nom2);
    } else {
       printf("%s is less than %s \n", nom2, nom1);
  }
}
```

Compare strings: Example

```c
#include <stdio.h>
#include <string.h>

int main(){
    char s1[20];
    fgets(s1, 20, stdin);
    strcmp(s1, "login");
    if (strcmp(s1, "login") == 0)
        printf("s2 = \"login\"\n");
    else
        printf("s2 != \"login\"\n");
    return 0;
}
```

Kahoot time!

Q1 – Q10

I/O and files

- In C, the I/O are implemented using files
- There are a lot of predefined files
  - The standard input **`stdin`**
  - The standard output **`stdout`**
  - The standard error **`stderr`**
- We have already seen the basic formatted I/Os
  - **`printf(…)`** : write to the file of the **`stdout`**
  - **`scanf(…)`** : read from the file **`stdin`**
- More global I/Os exist
  - Non formatted
  - Based on real files

File declaration

- The access to the files is performed through logical files
  - We have to use the directive **`#include <stdio.h>`**
  - We define them as type **`FILE`**
  - We use pointer to access them **`FILE*`**

- Example of declaring a logical file
  - **`FILE* file;`**

- Remark:
  - A logical file of a program has to be linked with an existing physical file in the file system
  - The link is created through the operations of opening and closing the logical files

File open: `fopen`

- Syntax

```
FILE* fopen (char *name, char *mode );
```

- Function
  - Open a physical file called **`name`** with the access rights defined by the string **`mode`** :
    - **`"r"`** open for read
    - **`"w"`** open for write/create
    - **`"a"`** open for write at the end of the file
- Parameters
  - **`name`** : external name of the file to be opened
  - **`mode`** : mode and right of opening.
- Return value
  - A pointer over the logical file is returned. In case of an error the return pointer is **`NULL`**.

File close: `fclose`

- Syntax

```
int fclose(FILE *file);
```

- Function
  - Close the logical file pointed out by the pointer passed as argument
- Return value
  - The function return 0 if the file is correctly closed
  - It returns the constant **`EOF`** , in case of an error
- ATTENTION : it is obligatory to close an opened file in mode write. Otherwise the content is NOT stored!

Read from a file

- **`int getc (FILE * file);`**
  - Read a character from the file pointed by **`file`**
  - Advances the position indicator for the file.
  - Returns:
    - the character read as an unsigned char cast to an int
    - the constant **`EOF`** in case of an error.
  - **`getchar()`** is equal to **`getc(stdin)`**

- **`int fscanf(FILE *file, char *format, list_args);`**
  - Works as **`scanf()`**, but the read is performed from the file pointed by **`file`** instead of the standard input

Read from a file

- **`int fread(void *buf, int size, int nb, FILE *file);`**
  - Read the packets of data stored to the file pointed by **`file`** and write them to a buffer pointed by the pointer **`buf`**
  - The parameter **`size`** gives the size of each packet
  - The parameter **`nb`** indicates the number of packets to write
  - Return the number of packets read from the file, 0 if we have reached the end of the file
  - Attention : the buffer pointer by **`buf`** should be large enough to store all the packets read!

Write to a file

- **`int putc (int c, FILE * file);`**
  - Write a character **`c`** (converted in **`unsigned char`**) in the file pointed by **`file`**
  - Advances the position indicator for the stream
  - Returns:
    - returns the character written as an unsigned char cast to an int
    - the constant value **`EOF`** in case of an error
  - **`putchar()`** is equivalent to **`putc(c, stdout)`**.

- **`int fprintf(FILE *file, char *format, list_args);`**
  - Works like **`printf()`**, but the writing is performed into the file pointed by **`file`** instead of the standard output

Write to a file

- **`int fwrite(void *buf, int size, int nb, FILE *file);`**
  - Write the packest of data stored to the buffer pointed by the pointer **`buf`** in the file pointed by **`file`**
  - The parameter **`size`** gives the size of the packet
  - The parameter **`nb`** indicates the number of packets to write
  - Return the number of packets written to the file

Example

```
#include <stdio.h>
#include <string.h>   /* strlen() */
#include <stdlib.h>   /* exit() */
int main() {
    FILE *file;
    char *msg = "123456789012345”;
    int n;
    file = fopen("tmp/test_write", "w");
    if (file!= NULL) {
        n = strlen(msg);
        int res = fwrite(msg, sizeof(char), n, file);
        if (res!=n) {
            fprintf(stderr,"Error during writing\n");
            exit(2);
        }
        fprintf(stdout,"Write successful\n");
        fclose(file);
    }else{
        printf("Couldn't open file for writing\n");
    }
    return 0;
}
```

Dynamic memory allocation

Static & Automatic memory allocation

- Static:
  - allocated when the program starts
  - Variable lifetime is the entire program

- Automatic:
  - allocated during execution (function calls)
  - managed by the OS and the compiler
  - Variable lifetime is the block where they are defined
  - The variable size is fixed
  - Usually stored on the stack

Limitations & inefficiencies

- Not always possible to know the exact number of array elements. May depend:
  - Parameters of program execution
  - Information provided by the user during execution

- Limited control over the lifetime of the variables

- Worst case: Stack overflow
  - Example: Recursive functions

Dynamic memory allocation

- The memory to store the program variables is allocated during the execution of the program

- Stored in the heap (large free pool of memory)

- The programmer controls:
  - the exact size (instantiated during execution)
  - Lifetime of memory locations

Worst-case: Memory leaks!

Problems: Dangling pointers!

…

Memory Leak

- Allocated blocks in the heap that are now unused and unreferenced !
- Improper use of dynamic memory → causes the memory consumption to be increased with the time

- In JAVA, garbage collector takes care of the not removed reserved parts in the heap !

- In C, the **programmer is responsible** for that!

For EACH allocation → We NEED a DEALLOCATION

Dangling pointer

- An object is deleted or de-allocated
- If the value of the pointer is not modified
  - the pointer still points to the memory location of the deleted/de-allocated memory.

Dynamic memory: Functions

- Directive:
  - `#include <stdlib.h>`
- Allocation
  - `malloc()`
  - `calloc()`
  - `realloc()`
- Deallocation:
  - `free()`

Dynamic allocation: malloc()

- Syntax:

```
void *malloc(size_t size);
```

- Function
  - Reserves a block with a size equal to **`size`** bytes in the heap
    - **`size_t`** is unsigned integer type of at least 16 bit
  - The **content** of the memory block it is **NOT initialized**
  - To calculate the required memory size to store an object we should use the function **`sizeof(type)`**
    - The actual size of data object is compiler and target machine depended
- Return value:
  - pointer of **`void`** type with the address of the first reserved byte (**`void*`**)
  - We have to explicitly cast the pointer to the type of data we want to store(**`int*, char*, …)`**

Dynamic de-allocation: free(…)

- Syntax:

```
void free(void *pointer);
```

- Function
  - free() frees the memory block pointed out by the **`pointer`**, which has been reserved by a call, such as **`malloc()`**, **`calloc()`** ou **`realloc()`**
  - If the **`pointer`** has not been obtained by one of these functions, or it has been already freed by free(), the behavior is UNDEFINED
  - If the pointer **`ptr`** is **NULL**, nothing happens

- No return value

- The C **programmer is responsible for setting free** the memory, when it is no more needed

Dynamic allocation: Example

```c
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int* p;
    int i;
    int taille = 100;
    p = (int*)(malloc (taille * sizeof(int)));
    if (p == NULL)
    {
        printf("Allocation impossible:");
        exit(EXIT_FAILURE);
    }
    for (i=0; i<taille;i++){ p[i]=0; }
    free(p);
}
```

1D Array with `size` elements

```c
type *arr = (type *)malloc(size * sizeof(type));
```

```c
int *arr=(int *)malloc(4*sizeof(int));
```

	Heap
0x1000	xxx
0x1004	xxx
0x1008	xxx
0x100C	xxx
	xxx
0x1014	xxx
0x1018	xxx
	xxx
	xxx
0x1028	xxx
	xxx
	xxx

1D Array with `size` characters → string

```
char *arr=(char *)malloc(4*sizeof(char));
```

	Heap
0x1000	xxx
0x1004	xxx
0x1008	xxx
0x100C	xxx
	xxx
0x1014	xxx
0x1018	xxx
	xxx
	xxx
0x1028	xxx
	xxx
	xxx

String declaration: Remarks

- Dynamic declaration

```
char* s=(char*)malloc(sizeof(char) * SIZE);
s = “hello”;
```

  - We can assign a `char` **pointer** to another `char` pointer

```
char *p; p=s;
```

  - We can assign another set of characters

```
s=“bye”;
```

2D Array: Single pointer

```
type *arr= (type *)malloc(rows * cols * sizeof(type));
```

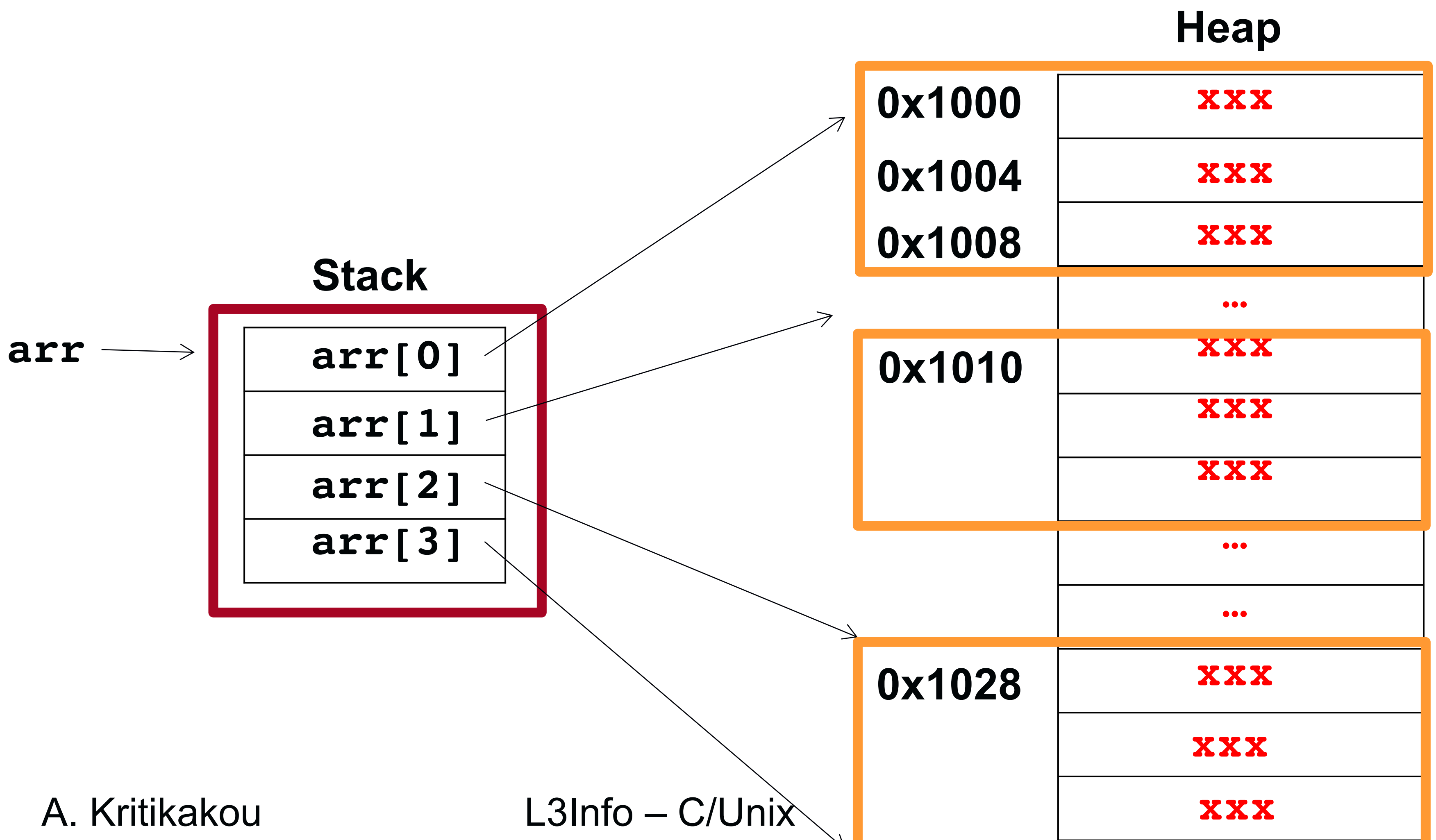
```
int *arr=(int *)malloc(4*3*sizeof(int));
```

Heap

0x1000	xxx
0x1004	xxx
0x1008	xxx
0x100C	xxx
	xxx
0x1014	xxx
0x1018	xxx
	xxx
	xxx
0x1028	xxx
	xxx
	xxx

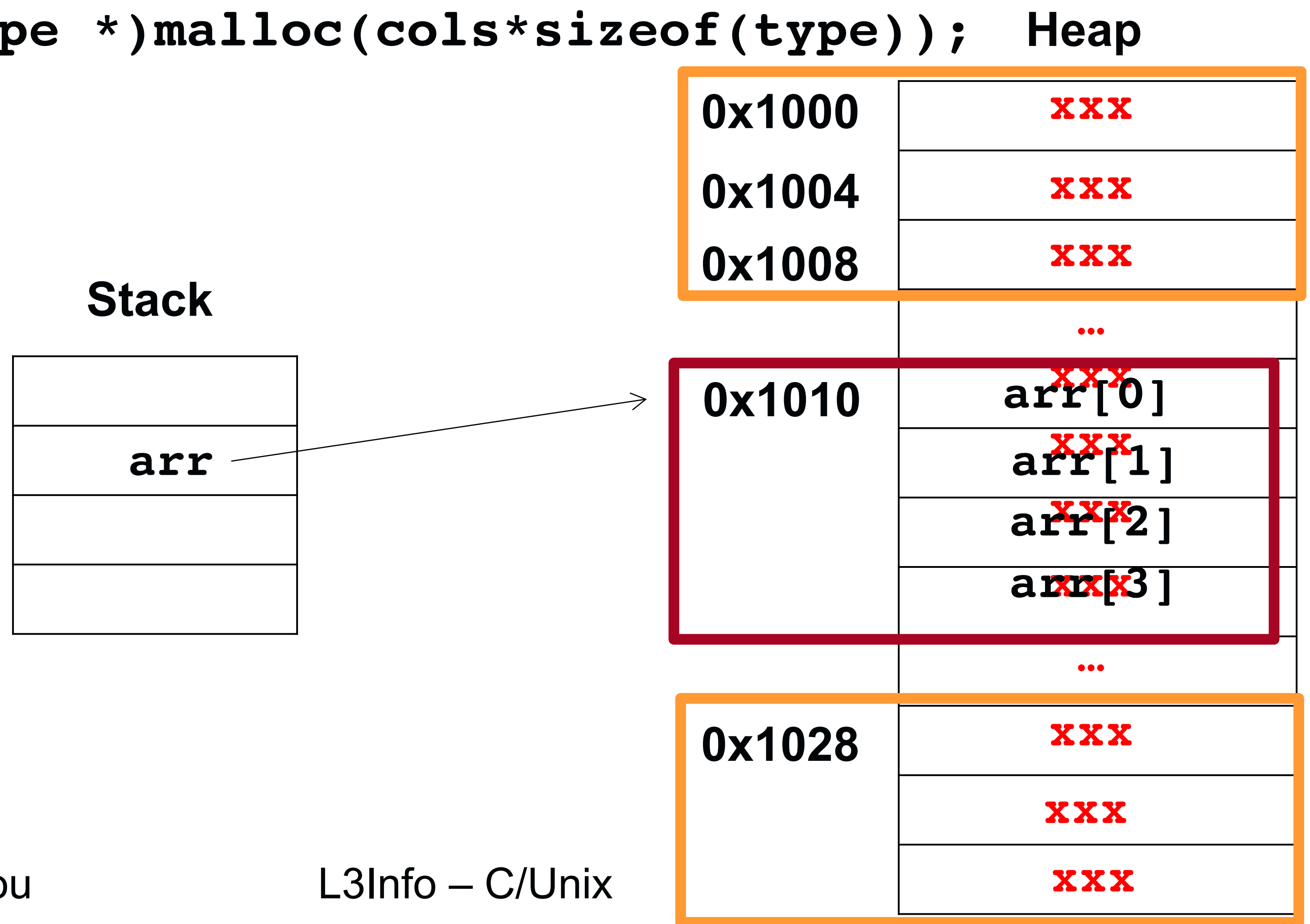
2D Array: Array of pointers

```
type *arr[ROWS];
for (i = 0 ; i < ROWS; i++) {
  arr[i]=(type *)malloc(cols*sizeof(type));
}
```



2D Array: Pointer to pointer

```
type ** arr;
arr=(type **)malloc(rows*sizeof(type*));
for (i = 0 ; i < rows ; i++) {
  arr[i]=(type *)malloc(cols*sizeof(type));
}
```



Kahoot time!

Q1 – Q7

User-defined data types: Enum, typedef, struct

Enumerated type declaration & definition

- Syntax:

  `enum identifier {element1,…}`

- Function:
  - The enumeration creates a **data type** that groups a set of elements that behave as **integral constants**
  - Declares a new enumeration type with the name **`enum identifier`**
- A variable is defined as **`enum identifier`**, it can be assigned one of the elements
- Example:
  - **`enum day{mon, tue, wen, thu, fri, sat, sun} ;`**
    - **`mon`** is coded by 0, **`tue`** by 1, …, **`sun`** par 6
    - This is a new type called **`enum day`**
    - **`enum day today;`**
    - **`today=wen;`**

Structure type declaration

- Syntax

```
struct identifier {
    member definition;
    ...
};
```

NO space is reserved!

It only creates a new type that describes the members of the group !

- Function:
  - A structure creates a **data type** that groups items of possibly different types into a single type.
  - Declares a new structure type named **`struct identifier`**
  - The members of the structure can be integral, float, arrays, pointers.
- Example:

```
struct address{
    char street[100];
    int number;
};
```

Structure type declaration

- The structure type declaration CANNOT be initialized!
- This is a new data type declaration, not a structure variable definition!
- Only when the structure variable is defined !
  - The corresponding memory is reserved
  - Now we can initialize the members of the structure

Structure variable definition & initialisation

- Separately from the structure declaration:

```
struct address MyAddress = {"Paul Bert", 12};
```

- During structure declaration:

```
struct address{
      char street[100];
      int number;
   } MyAddress = {"Paul Bert", 12};
```

Space is reserved!

Equal to the one required to store the structure members aligned

Structure padding

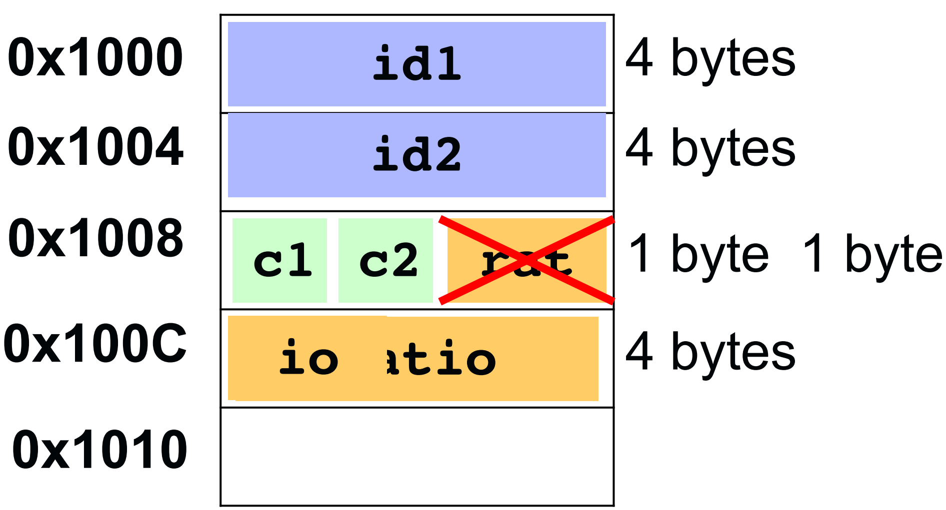
- Architecture of a computer processor is made in such a way so it can read 1 word from memory at a time
  - e.g. 4 byte in 32 bit processor

- To use this processor advantage, the data are aligned in memory in words
  - e.g. multiples of 4 bytes in 32-bit processor

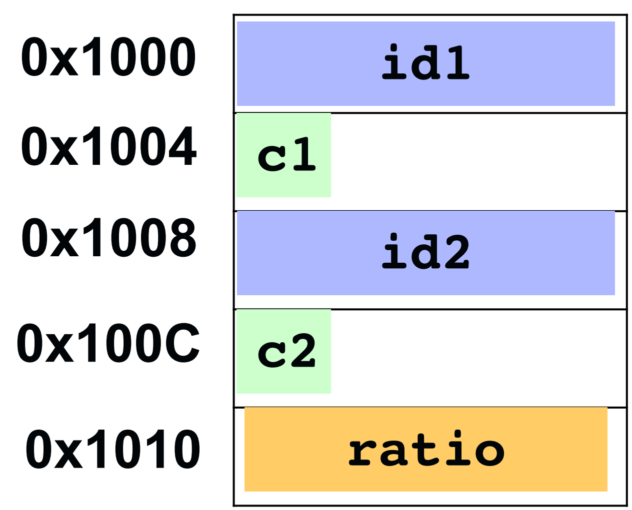
- One or more memory addresses may LEFT EMPTY to achieve this alignment !

Structure padding: Example

```
struct structure1
{
        int id1;
        int id2;
        char c1;
        char c2;
        float ratio;
};
```




```
struct structure1
{
        int id1;
        char c1;
        int id2;
        char c2;
        float ratio;
};
```



Structure variable: Access members

```
struct address{
   char street[100];
   int number;
};
```

- Through a structure variable: **.**

```
struct address MyAddress;
strcpy(MyAddress.street, “Paul Bert”);
MyAddress.number=12;
```

- Through a pointer to the structure variable: →

```
struct address MyAddress;
struct address *MyAddressPtr = &MyAddress;
strcpy(MyAddressPtr→street, “Paul Bert”);
MyAddressPtr→number=12;
```

Definition of a new type

- Syntax

  `typedef type IDENTIFIER;`

- Examples
  - `typedef int LENGHT;`
  - `typedef enum {false, true} BOOLEAN ;`
  - `typedef struct date date;`
    - Use of **`typedef`** to avoid repetition of keyword **`struct`**

- Remark
  - There is no identifier for the **`enum`** in the definition of the type **`BOOLEEN`**

Structure type: Examples

```
struct date /* déclaration du type struct date */
{
    short j
    short m ;
    int a ;
} ;
```

```
/* déclaration du type struct UNE_DATE */
typedef enum {lun, mar, mer, jeu, ven, sam, dim} JOUR ;
typedef struct
{
    JOUR j ;
    struct date d ;
} UNE_DATE ;
```

Access struct members

- Variable structure

```
UNE_DATE foo = { mer, {3 ,9 ,2014 } } ;
foo.j = mer ;
foo.d.a = 2014 ;
```

- Structure pointer

```
UNE_DATE* queljour = &foo;
queljour->j = foo.j;
queljour->d.a = 2014;
```

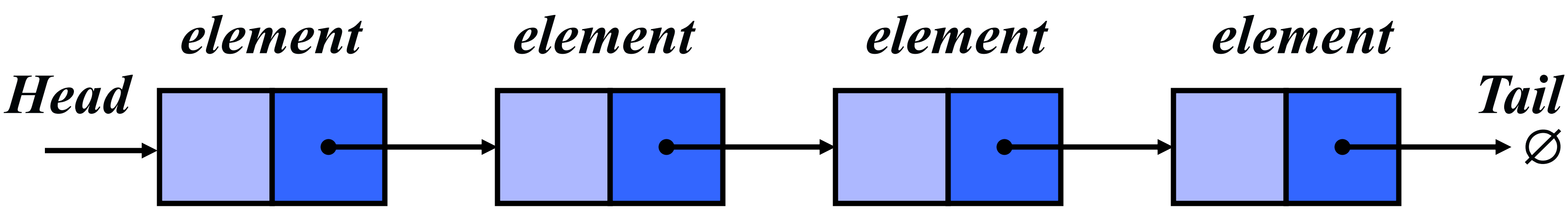
Kahoot time!

Q8 - Q12

Data structures:
Lists and Hash tables

Linked lists Vs arrays

- Linear data structure (like arrays)
- Linked list elements NOT stored in contiguous location
  - are linked using pointers



Why Linked list?

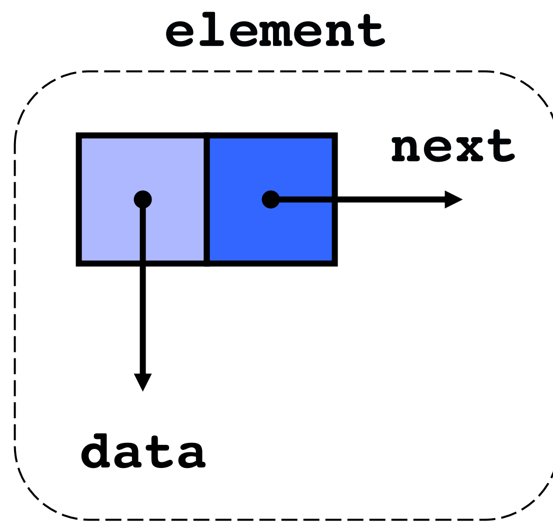
- Array limitations:
  - Fixed size: We must know the upper limit on the number of elements in advance
  - Insert/Delete an element: Expensive
    - Create room by shifting existing elements !
    - Ex. : Add new element 101 in an array with sorted ids

```
id[]=[100, 103, 105, 200]
```

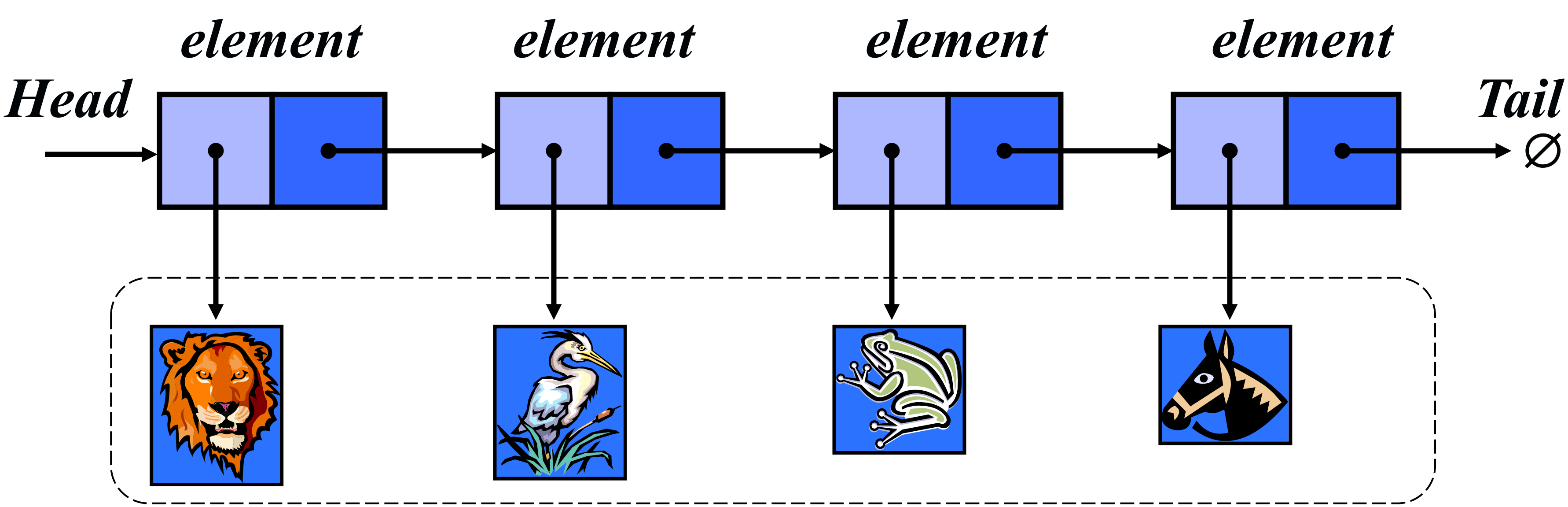
- List Advantages:
  - Dynamic size
  - Ease of insertion/deletion
- List Drawbacks:
  - Random access is not allowed
    - Access elements sequentially
  - Extra memory space required for the pointer to each element

Linked list

- Each element has
  - **`data`**
  - **`next:`** Link to the next element



- In C, such an element can be represented using structures

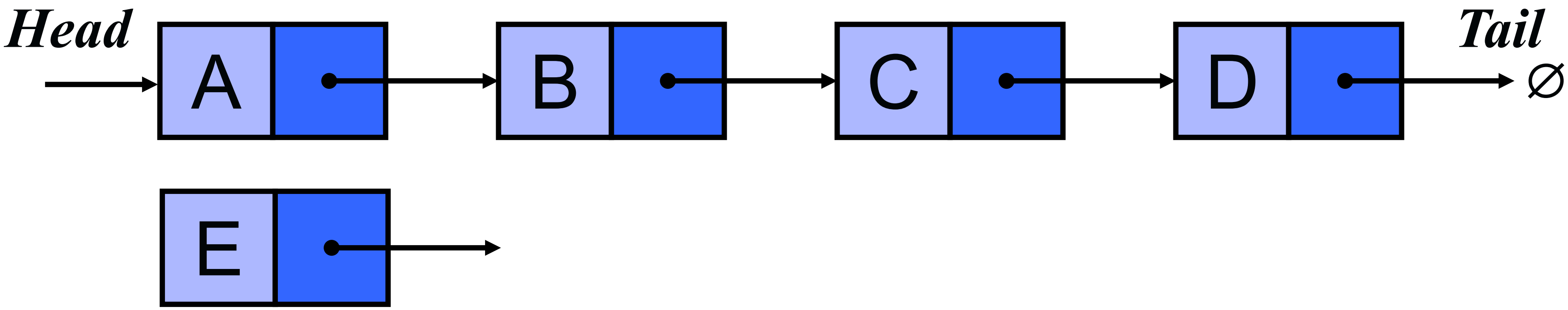


Linked list: Example

```
struct element {
    int data;
    struct element *next;
};

int main(){
   struct element *head = NULL;
   head=(struct element*)malloc(sizeof(struct element);
   head->data=1;
   head->next=NULL;

   return 0;
}
```

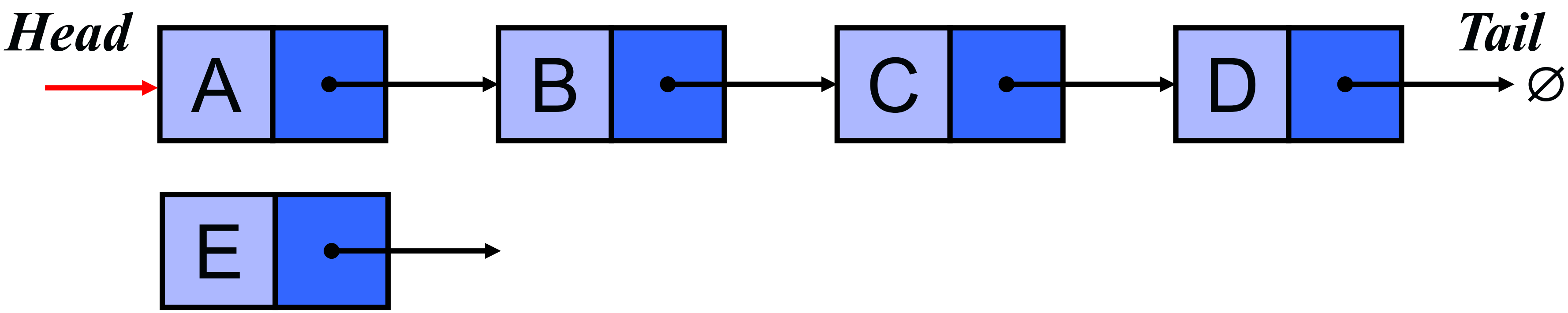
Operations: Insertion

- At the head:



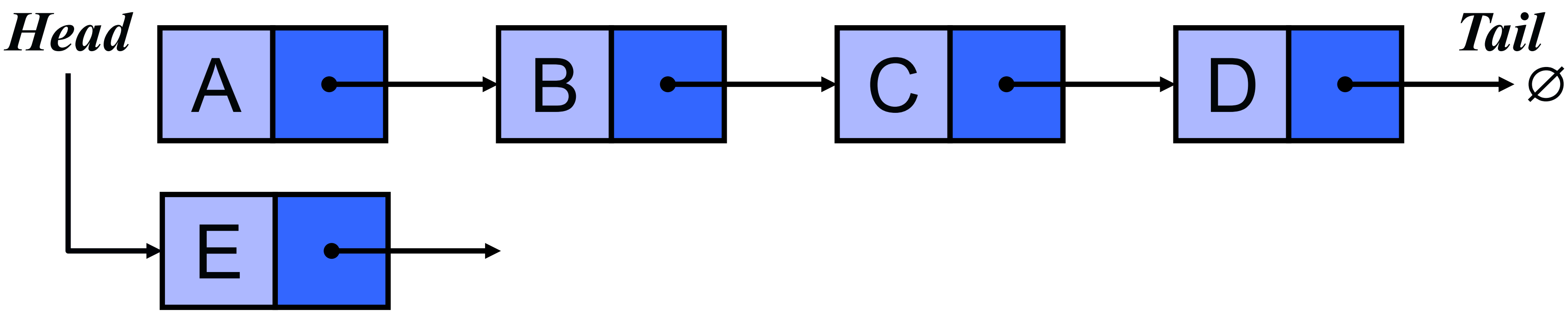
Operations: Insertion

- At the head:



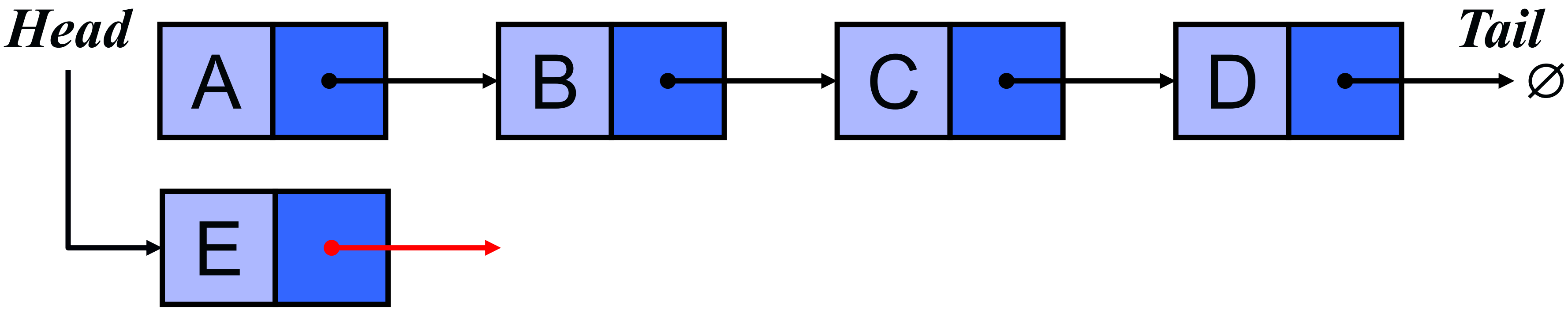
Operations: Insertion

- At the head:



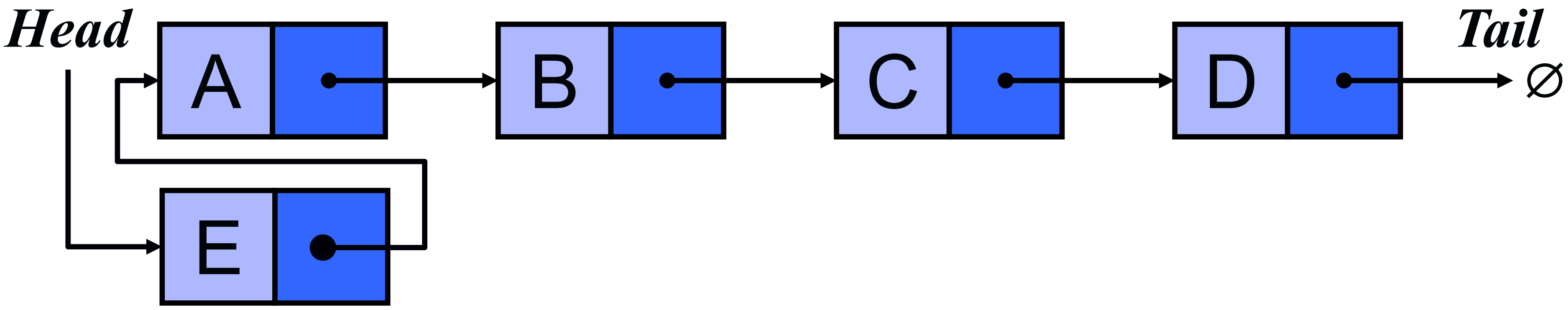
Operations: Insertion

- At the head:



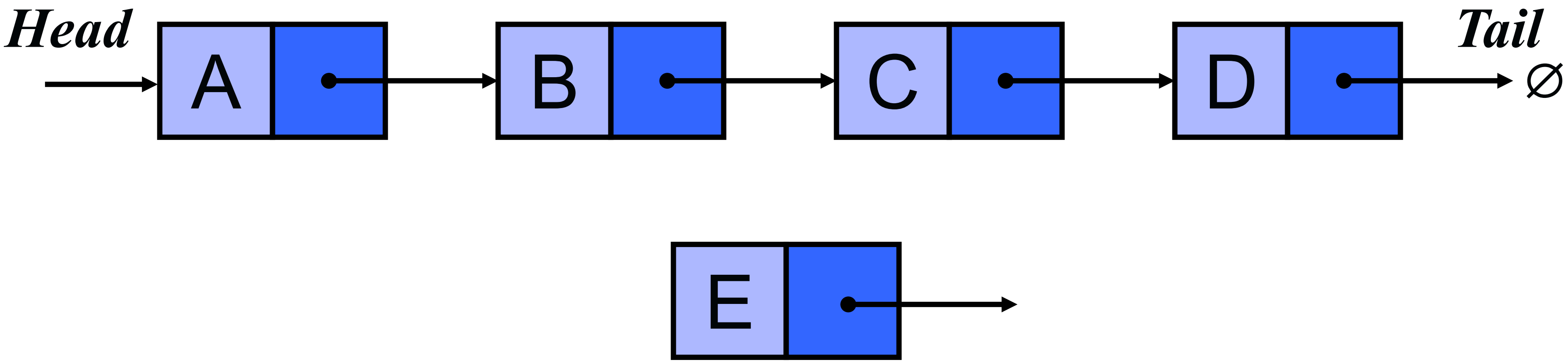
Operations: Insertion

- At the head:



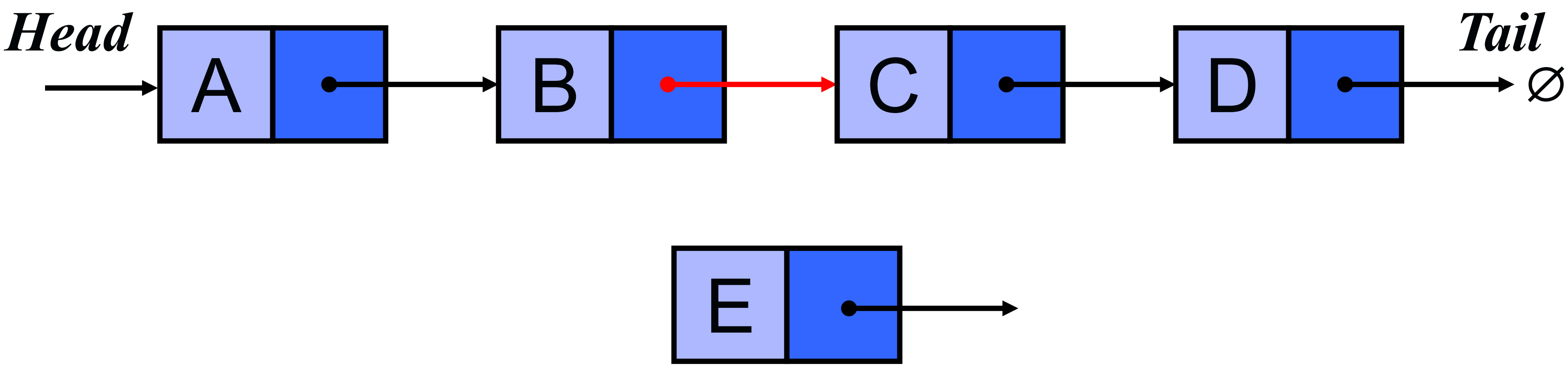
Operations: Insertion

- In the middle:



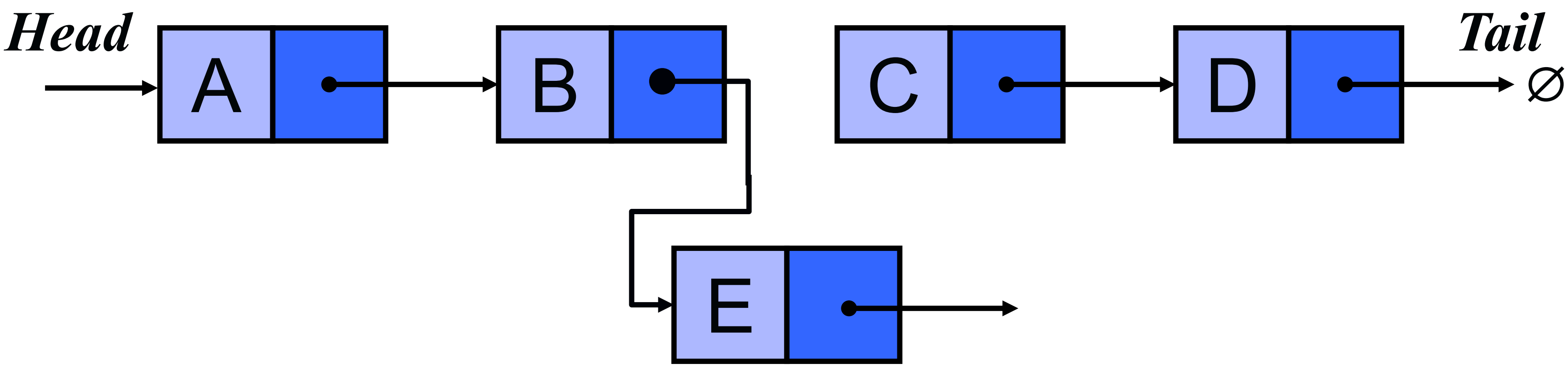
Operations: Insertion

- In the middle:



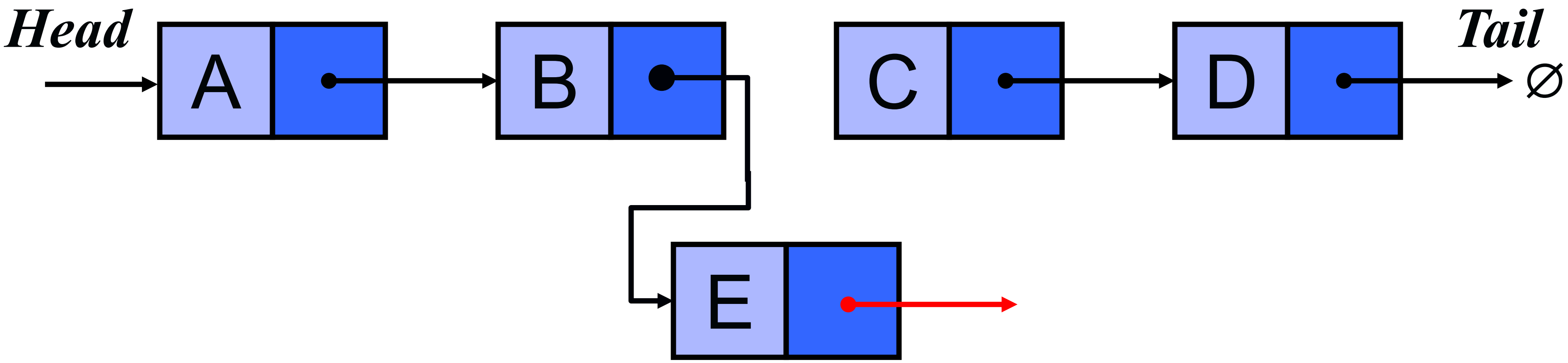
Operations: Insertion

- In the middle:



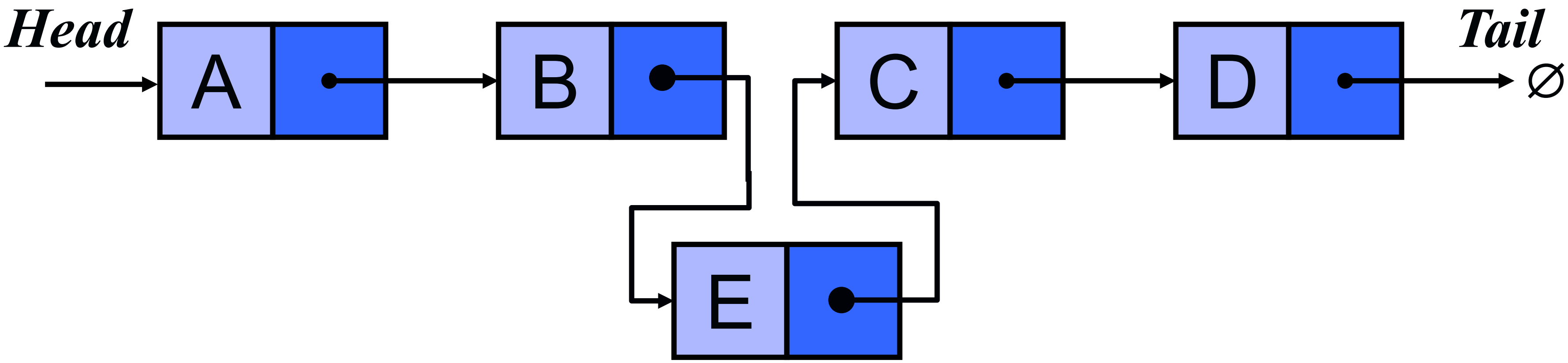
Operations: Insertion

- In the middle:



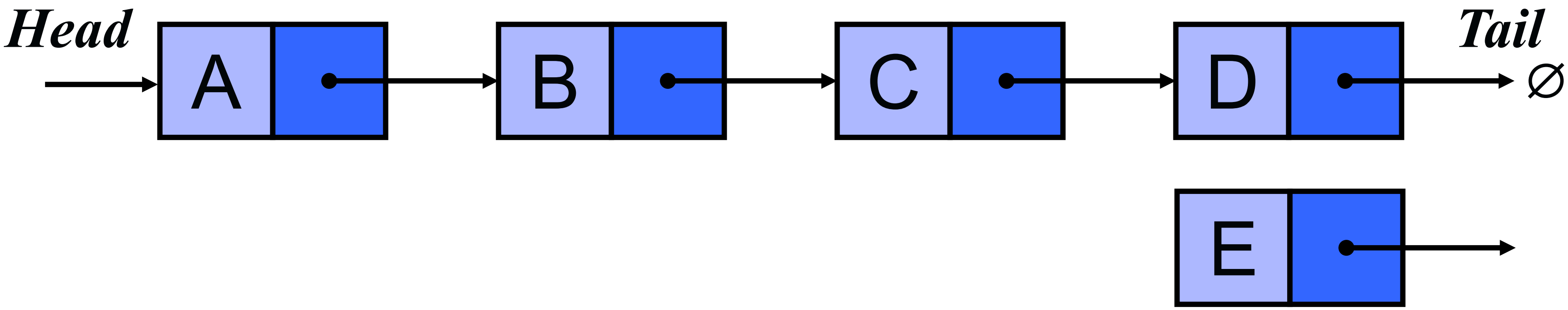
Operations: Insertion

- In the middle:



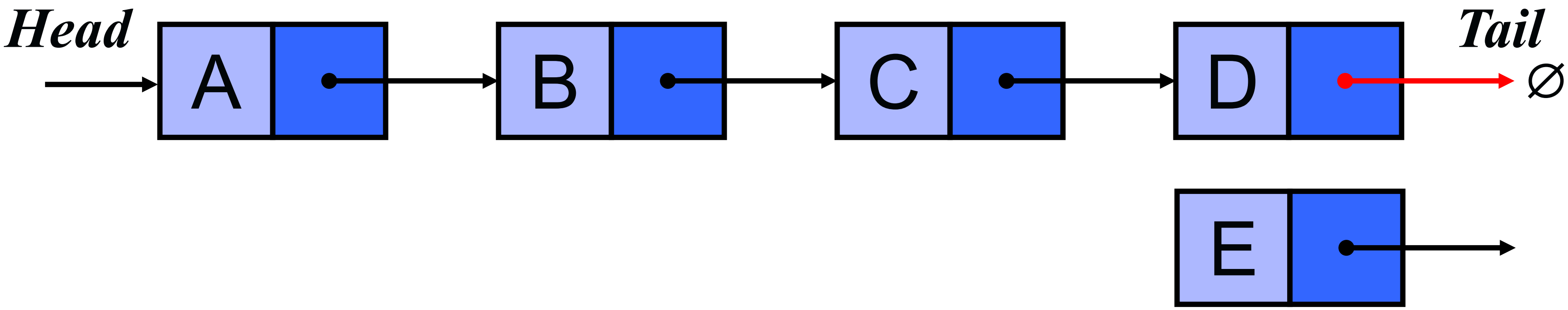
Operations: Insertion

- At the tail:



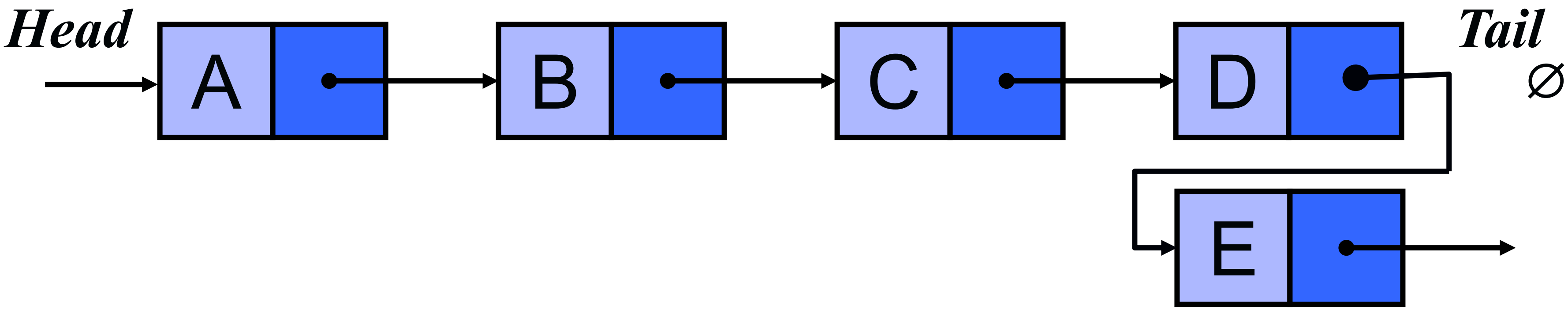
Operations: Insertion

- At the tail:



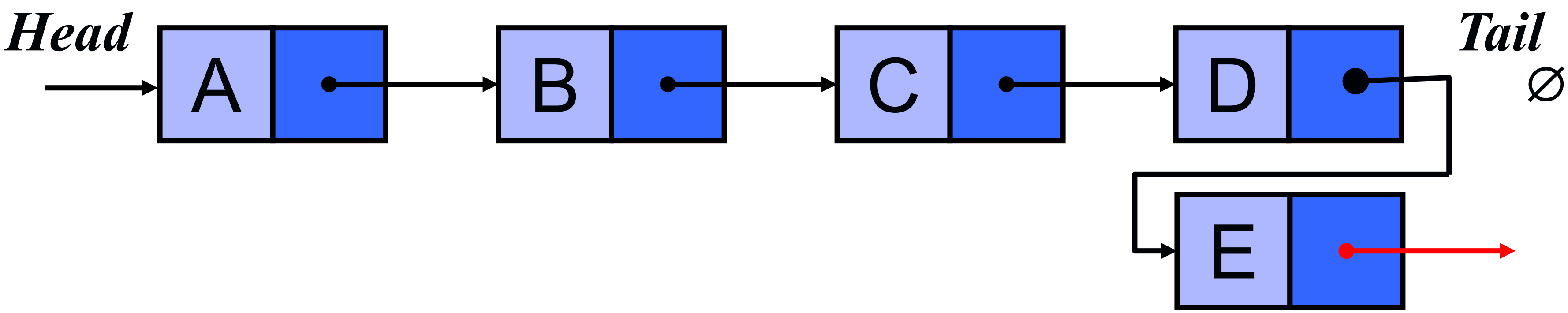
Operations: Insertion

- At the tail:



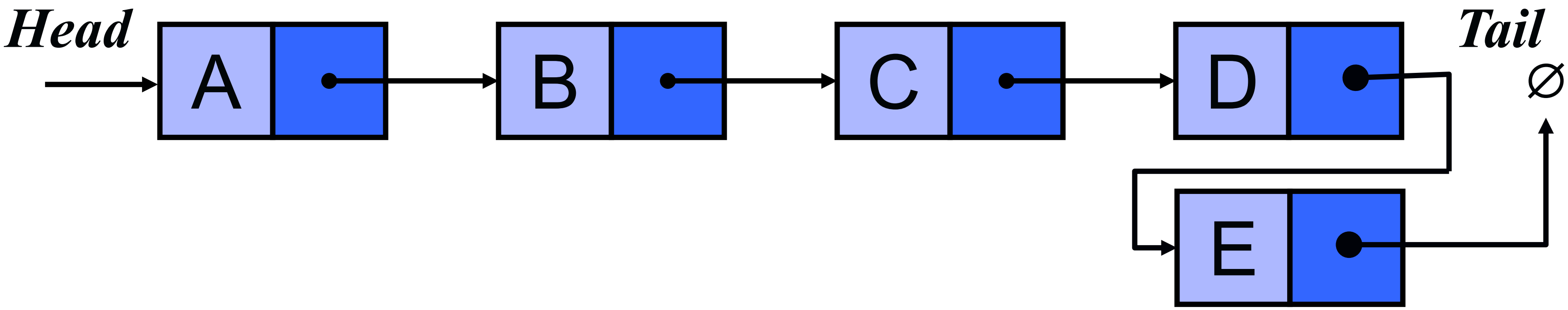
Operations: Insertion

- At the tail:



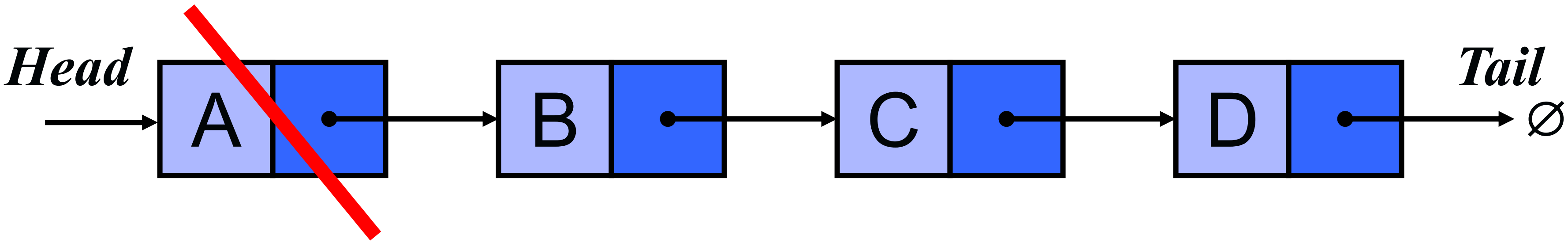
Operations: Insertion

- At the tail:



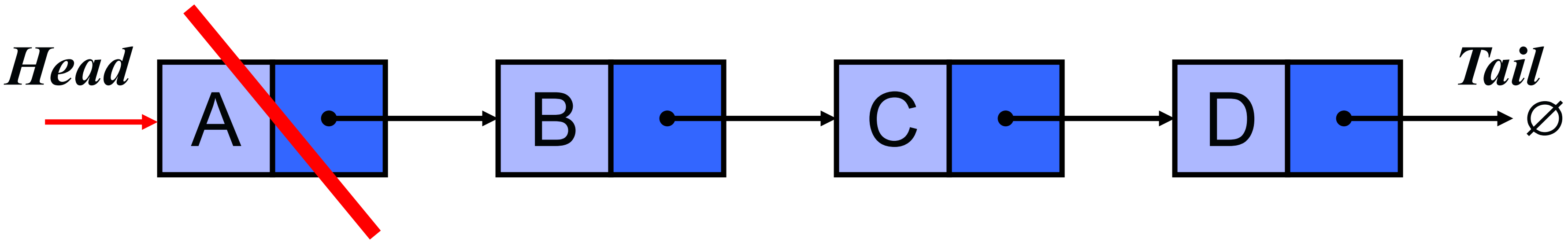
Operations: Deletion

- At the head:



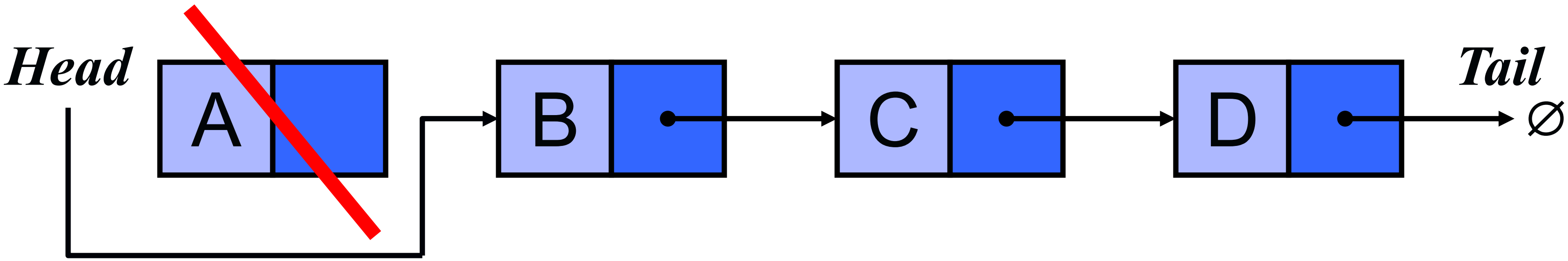
Operations: Deletion

- At the head:



Operations: Deletion

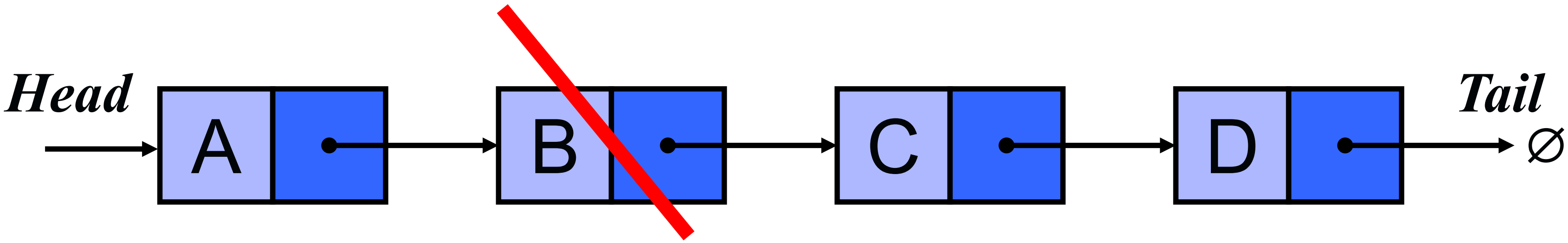
- At the head:



- ATTENTION: Free the memory space used for element A

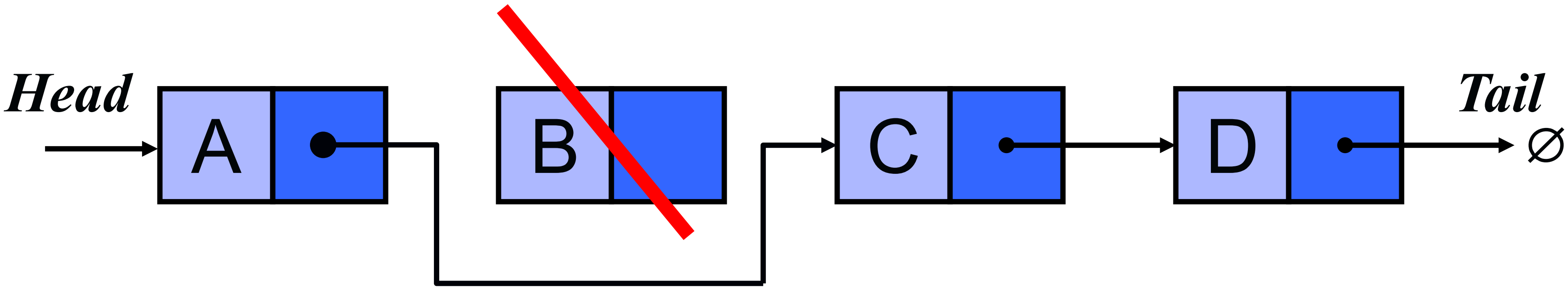
Operations: Insertion

- In the middle:



Operations: Insertion

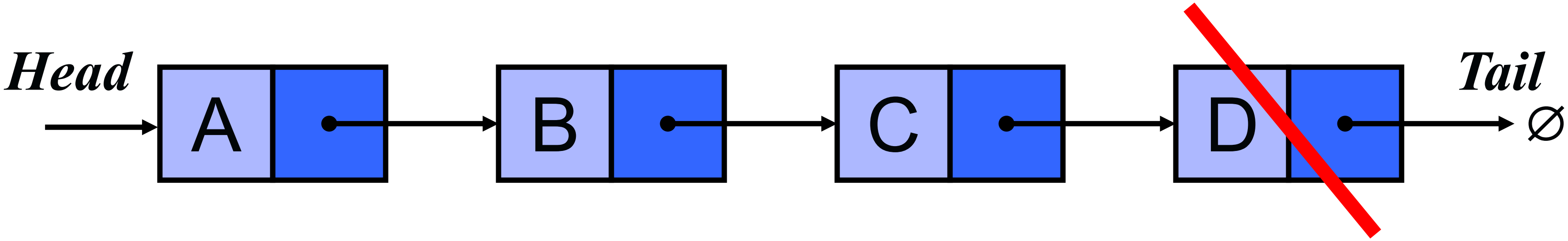
- In the middle:



- ATTENTION: Free the memory space used for element B

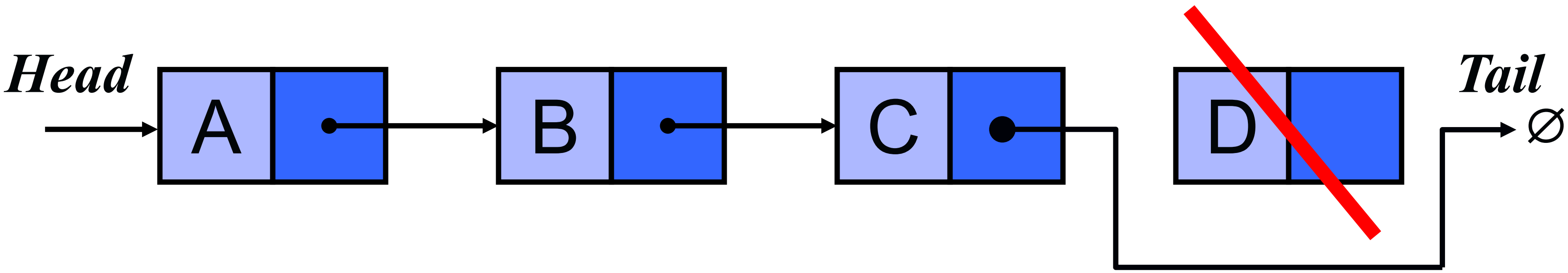
Operations: Insertion

- At the tail:



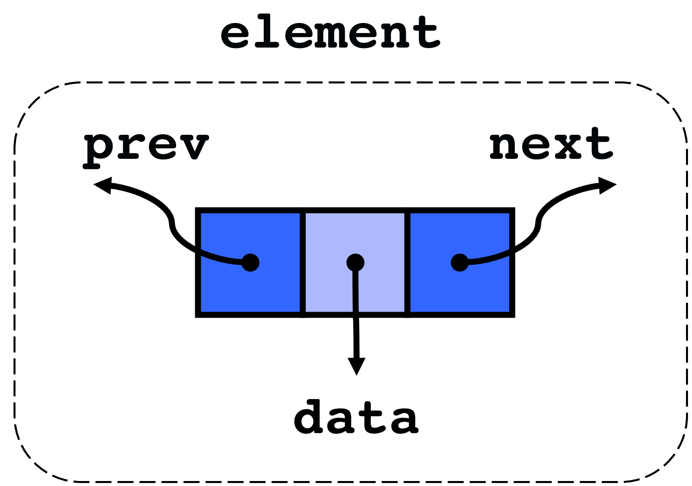
Operations: Insertion

- At the tail:



- ATTENTION: Free the memory space used for element D

Doubly linked list



- Each element has
  - **`data`**
  - **`prev`**: Link to the previous element
  - **`next`**: Link to the next element
- Advantages
  - Traversed both in forward and backward direction
  - Delete operation is more efficient, previous pointer is given
    - Singly list: We have to traverse the list to find it
- Disadvantages:
  - Extra memory space for storing the previous pointer
  - All operations have to maintain this extra pointer.

Hash Table

- An array of a fixed size **`hsize`**
- An element is associated with an integer value called **`hash value`**
  - a number within the range 0 to **`hsize`** – 1
  - depicts in which cell of the hash table the element will be stored (**`key`**)
  - calculated using a Hash Function
    - Maps a data set of arbitrary size to a data set of fixed size
    - unique value
- Fast access of the data:
  - Search in the index indicated by the **`key`**

0	
1	
2	John
3	Mary
4	
5	
6	Helen
7	
8	Nick
9	

Hash Table: Some of its use

- Caches: Data tables used to speed up the data
- Object representation: Dynamic languages (Perl, Python, JavaScript) use hash tables to implement objects
- Algorithms to make computing faster
- Database indexing: In disk-based data tables and database indexes

Hash Function: Simple examples

- Integers:
  - Return **`key MOD hsize`**

- Strings:
  - We add up the ASCII values of the chars, and return the modulo operation

```
int hash( char *key, int hsize )
{
int hash_val = 0;
while( *key != '\0' )
             hash_val += *key++;
      return( hash_val % hsize );
}
```

Collisions

- When more than one element have the same hash value !

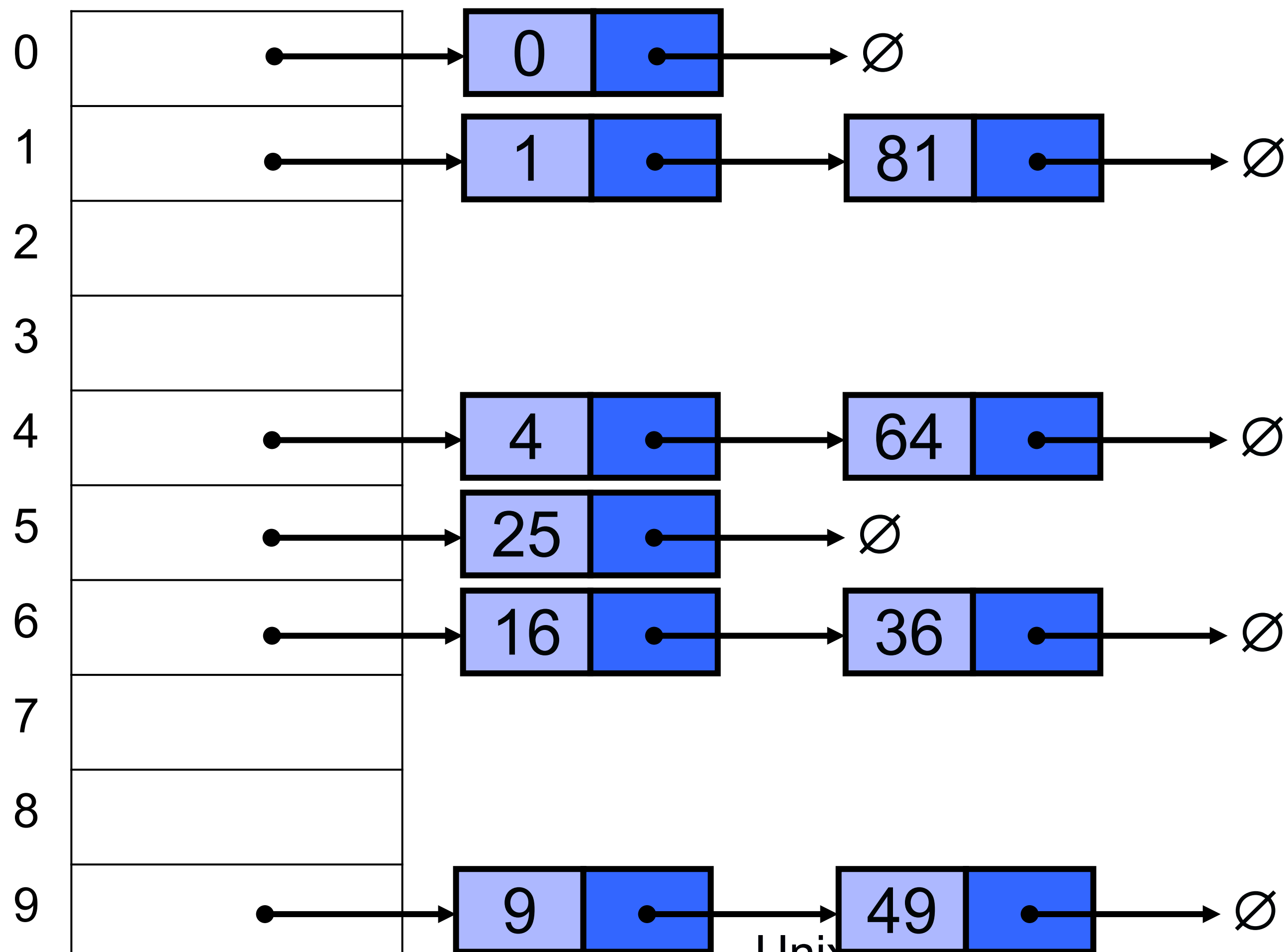
- Example:
  - **`hsize`** = 10
  - Elements: the first 10 perfect squares

  0, 1, 4, 9, 16, 25, 36, 49, 64, 81
  - hash function: **`hash(x)=x mod 10`**

0	0
1	1 / 81
2	
3	
4	4 / 64
5	25
6	16 / 36
7	
8	
9	9 / 49

Collisions: Solution

- Separate chaining (open hashing):
  - Each element of the hash table is a linked list
  - Collision: Store both elements in the same linked list



Problems

- The distribution of the values may NOT be even !

- Example
  - **`hsize`** = 10007
  - Variable: strings of 8 ASCII characters
  - The hash function: adds up the ASCII values of the chars and return the modulo operation
  - → it can only assume values 0 – 1016, which is 127*8

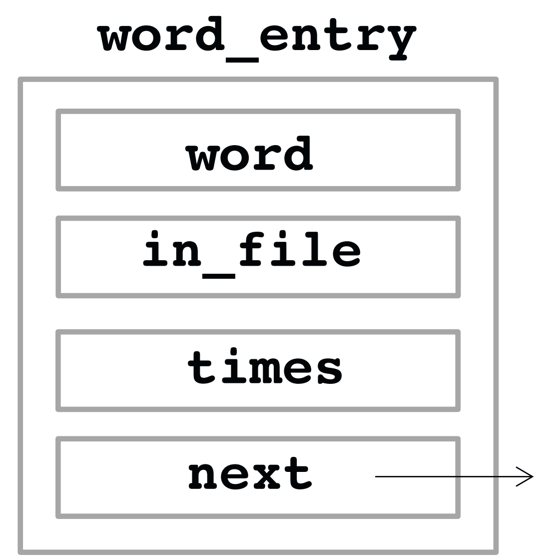
Hash table: Dictionary

- Create a dictionary and search engine:
  - Read files word by word
  - Add to the dictionary one word found in a file and the number of times found in the file
  - Search for a word in the dictionary

- Use hash table and linked lists to store the words with the same key

Hash table: Dictionary

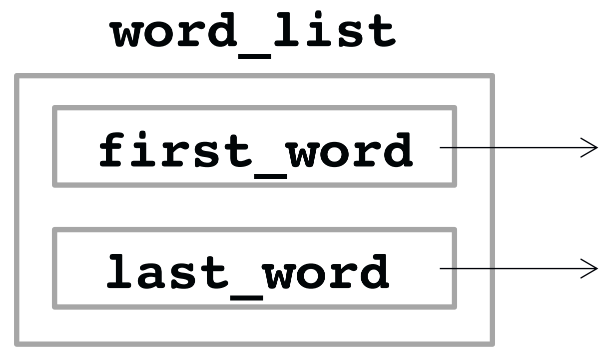
```
struct word_entry
{
  char word[SIZE];
  int in_file;
  int times;
  struct word_entry *next;
};
typedef struct word_entry word_entry;
```



- The structure **word_entry** has as elements
  - a word of size **SIZE**
  - an integer **in_file** which is the identifier of the file where the word has been found
  - an integer **times** which shows how many times the word exist in this file
  - a pointer ***next** to a structure **word_entry**

Hash table: Dictionary

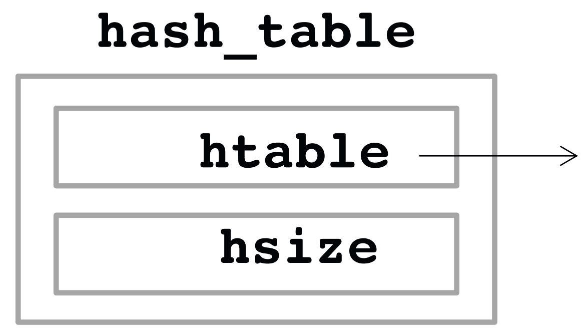
```
struct word_list
{
  struct word_entry *first_word;
  struct word_entry *last_word;
};
typedef struct word_list word_list;
```



- The structure **word_list** has as elements
  - a pointer ***first_word** to a structure **word_entry**
  - a pointer ***last_word** to a structure **word_entry**

Hash table: Dictionary

```
struct hash_table
{
word_list *htable;
int hsize;
};
typedef struct hash_table hash_table;
```



- The **`hash_table`** structure contains
  - a pointer **`htable`** to a type **`word_list`**
    - A pointer to a pointer of the structure **`word_entry`**
  - the fixed size **`hsize`**

- We have finished the hash table declaration?

Hash table: Dictionary

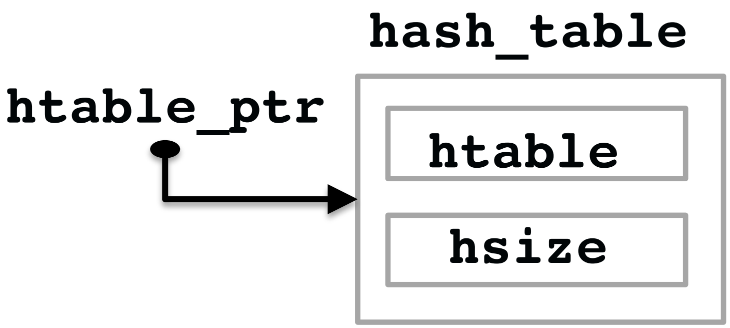
```
hash_table *htable_ptr;
htable_ptr = (hash_table*) malloc (sizeof(hash_table));
```

- Hash table allocation:
  - **`malloc`** reserves a memory space equal to the structure **`hash_table`**
  - **`htable_ptr`** will point to a structure containing an integer and a pointer to a **`word_list`**
- Hash table size initialisation

```
htable_ptr->hsize = PRIME_NUMBER;
```

- We have finished the hash table declaration?

Hash table: Dictionary



Hash table: Dictionary

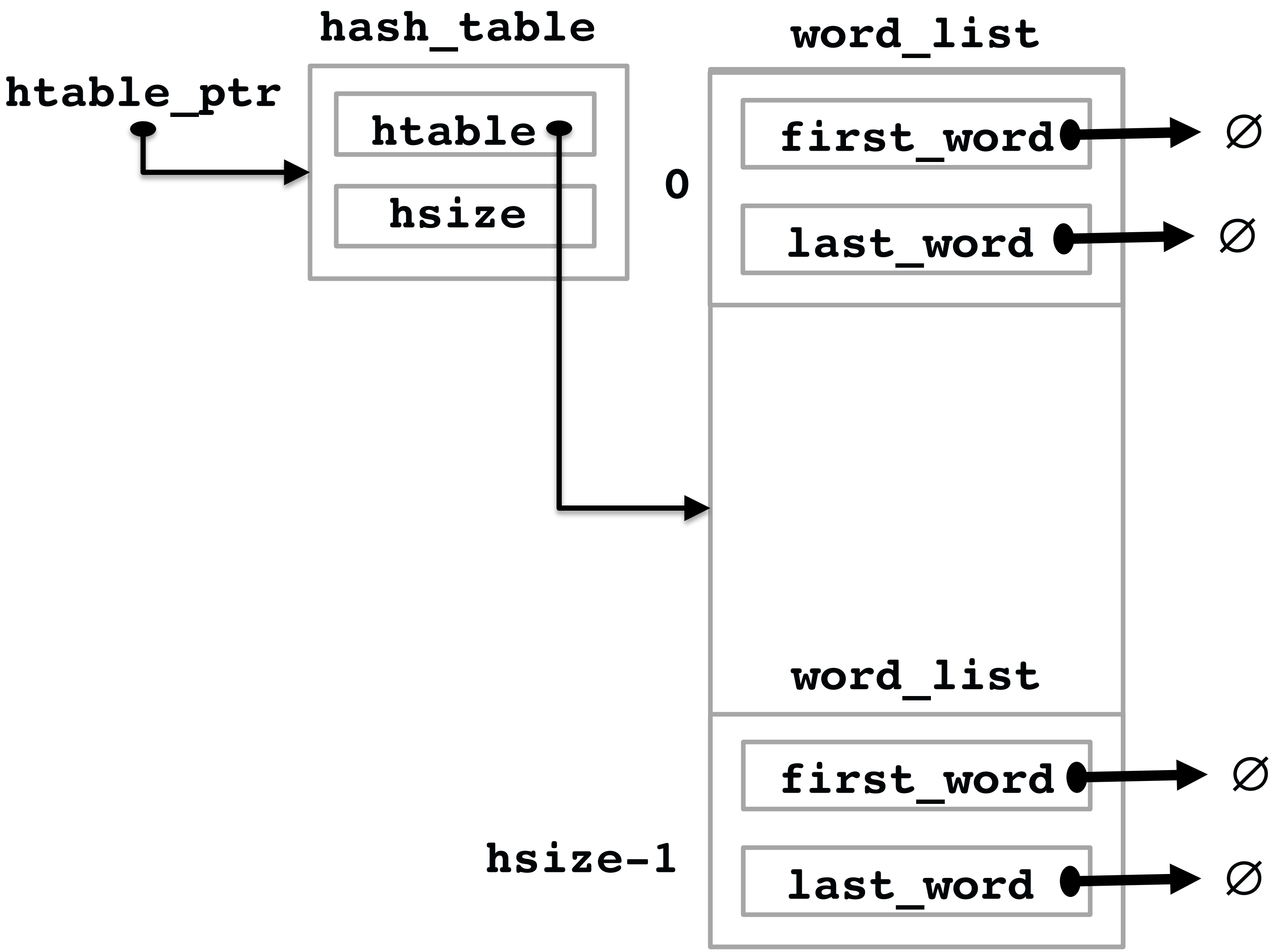
- Hast table pointer initialisation:
  - We need to **allocate** space for the linked lists.
  - We use an array of structures **`word_list`** equal to the table size

```
htable_ptr->htable =
    (word_list*) malloc(sizeof(word_list) * htable_ptr->hsize );
```

  - We need to initialize the pointers of the structures **`word_list`** to **`NULL`**

```
for(i = 0; i < htable_ptr->hsize ; i++ ){
  htable_ptr->htable[i].first_word = NULL;
  htable_ptr->htable[i].last_word = NULL;
}
```

Hash table: Dictionary

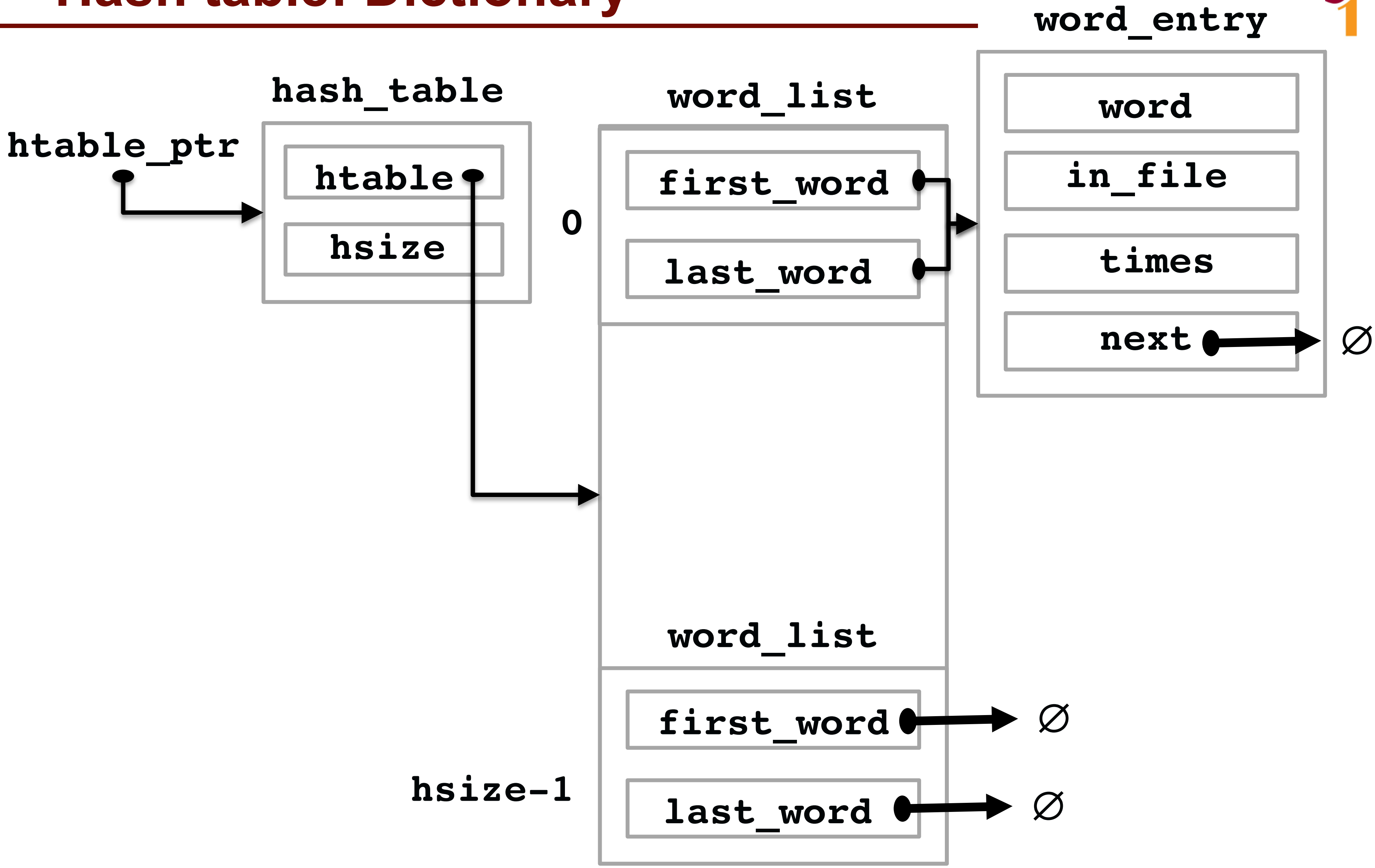


Hash table: Dictionary

- When a new word is added:
  - We need to **allocate** space for the word
  - Connected to the list based on the hash key

```
htable_ptr->htable[key].first_word =
                (word_entry*)malloc(sizeof(word_entry));
```

Hash table: Dictionary



TD2: C exercices

Exercice 1: Basic I/O

- What prints the following program?

```
#include<stdio.h>

int main(){
   char nom[20] = "Balthazar";
   double solde = 1000.25;
   int age = 21;

   printf("Hello %s, you have %d years\n", nom, age);
   printf("Your age is written %x in hex\n", age);
   printf("Your name start with the letter %c\n", nom[0]);
   printf("Your bank account balance is %lf\n", solde);
   printf("The address of the table name is %p\n", nom);
}
```

Exercice 2: Pointers

- Assume the following instructions

```
int x=0;
int y=0;
int* p=NULL;
p = &x;
x=2;
y=4;
*p = y;
x = 5;
y = *p;
p = &y;
```

- Provide the values of **x**, **y** and **p**

Exercice 3: Pointers

- Give the output of the program

```
#include <stdio.h>
void echange(int * a, int b){
        int sauve; sauve = *a; *a = b; b = sauve;
}
void echange2(int * a, int ** b){
        int sauve; sauve = *a; *a = **b; **b = sauve;
}
void echange3(int * a,int * b){
        int sauve; sauve = *a; *a = *b; *b = sauve;
}
int main(){
        int a, b, d; int* c = &d; a = 1; b = 2; (*c) = 3;
        echange(&a,b);   printf("a= %d, b= %d\n",a,b);
        echange2(&a,&c); printf("a= %d, c= %d\n",a,*c);
        echange3(&a,&b); printf("a= %d, b= %d\n",a,b);
        echange3(&b,c);  printf("b= %d, c= %d\n",b,*c);
        echange(&a,&b);  printf("a= %d, b= %d\n",a,b);
        return 0;
}
```

Exercice 4: Strings

- Write a function called **reverse** that reverses the characters of a string, without declaring a second array. The prototype of the function is the following:

```
int reverse(char* str) ;
```

Exercice 5: Strings and pointers

- Give the output of the following program

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

char* mess="hello world\n"

int main() {
       strcpy(mess+4,mess) ;
       return 0;
}
```

Exercice 6: Linked lists

- We have the structure **`list_elem_t`** for the list elements

```
typedef struct s_list {
        int  value;              // data of the element
        struct s_list* next;     // pointer to the next element
  } list_elem_t;
```

- The head of list is pointing by the pointer **`list_head`**

```
list_elem_t* list_head;
```

- Write a function

`list_elem_t* create_element(int val)`,

which creates a new list element and initializes its **`value`** equal to the passed argument **`val`** and the **`next`** pointer to **`NULL`**

Exercice 6: Linked lists

- Assume the function **`insert_head`**, which inserts a new element at the head of the list that has been passed as parameter

```
int insert_head(list_elem_t* l, int val) {
   list_elem_t * nouveau = create_element(val);
   nouveau->next = l ;
   return 0 ;
}
```

- Is it correct?
  - What happens when **`insert_head(list_head, 1)`** is executed?
  - Correct the function !

Exercice 6: Linked lists

- Write a function:

`int insert_tail(list_elem_t** l, int val)`,

which creates a new list element at the tail of the list and return 0 in case of success and -1 in case of failure

Exercice 7: Files

- Write a **`main`** function that
  - Reads the name of a file (**`name1`**) from the keyboard
  - Reads the name of a file (**`name2`**) from the keyboard
  - Copies character by character the content of the file name1 to the file name2 and replace all the **'X'** with **'Y'**.
  - If the copy is not possible, prints an error message

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

FILE* in;
FILE* out;

char nom1[40];
char nom2[40];
int main() {
        //To Be Completed
}
```