

Slide del Corso di  
*Informatica e Laboratorio di Programmazione*

Massimo Bertozzi

A.A. 2025 – 2026

Every possible effort has been made to ensure that the information contained in this book is accurate at the time of going to press, and the publishers and author cannot accept responsibility for any errors or omissions, however caused. No responsibility for loss or damage occasioned to any person acting, or refraining from action, as a result of the material in this publication can be accepted by the publisher or the author.



# Contents

|                                              | <b>Page</b> |
|----------------------------------------------|-------------|
| Introduzione al corso . . . . .              | 5           |
| Programmazione . . . . .                     | 17          |
| Il Computer . . . . .                        | 46          |
| I Linguaggi di Programmazione . . . . .      | 62          |
| Breve introduzione al C . . . . .            | 73          |
| Numeri e Simboli in Informatica . . . . .    | 95          |
| Espressioni in C . . . . .                   | 120         |
| Operazioni bit a bit . . . . .               | 131         |
| Il Controllo di Flusso . . . . .             | 138         |
| Tipi di dato in C . . . . .                  | 152         |
| Le Funzioni Predefinite del C . . . . .      | 173         |
| Gli Array . . . . .                          | 183         |
| Allocazione Dinamica della Memoria . . . . . | 191         |
| Le Stringhe . . . . .                        | 217         |
| Implementazione di Funzioni in C . . . . .   | 228         |
| I/O in C . . . . .                           | 244         |
| Struct, Union ed Enum . . . . .              | 264         |
| Puntatori a Funzione . . . . .               | 274         |
| Problem Solving . . . . .                    | 283         |
| Regole Esame . . . . .                       | 289         |
| Introduzione a CODE::BLOCKS . . . . .        | 294         |
| Il Debugging con CODE::BLOCKS . . . . .      | 305         |





UNIVERSITÀ DI PARMA



# Informatica e Laboratorio di Programmazione

*The true sign of intelligence is not knowledge  
but imagination.*

Albert Einstein

## Lessons



- $\sim \frac{2}{3}$  in Class (slides + code examples)
- $\sim \frac{1}{3}$  in Laboratory (actual programming)
- Slides, code examples, and laboratory assignments
  - <https://elly2024.didattica.unipr.it>
  - <https://www.ce.unipr.it/didattica/informatica/>
  - Warning: they are continuously updated!

2

## Schedule



- September 19<sup>th</sup> – December 18<sup>th</sup>
  - Monday 13:30-15:30 (Lab. E c/o Q02)
  - Wednesday 10:30-12:30 (Aula F c/o sede didattica ing.)
  - Thursday 10:30-12:30 (Aula F c/o sede didattica ing.)

3

## Calendar



- Pauses
  - November 18<sup>th</sup> → Elections?

4

## Prerequisites



- Candidates should
  - know basic concepts of Information Technology
  - be able to use PCs and deal with files
  - know how to read and write....

5

## Course program



- Theory
  - Class
- Practical part AKA programming
  - Class + Laboratory
- Problem solving
  - Class + Laboratory

6

## Course organization



- Theory (Class)
- Basic information needed to understand how to program
  - Basic computer architecture
  - Algorithms
  - How numbers can be managed in a computer

7

## Course organization



- Programming part (class + laboratory)
  - Centered on the programming language and algorithms
  - C fundamentals
  - Variables
  - Control instructions
  - Expressions
  - Array, strings & pointers
  - Functions and recursion
  - I/O
- Slides + code examples

8

## Course organization



- How to access laboratories
- You **must** use your official UNIPR account:
  - name.surnameXY@studenti.unipr.it
- Temporary you can use “exam” accounts:
  - User: workstation name, Password: “noncopiare”
- Feel free to use your laptop/notebook
  - Caveat: during the exam you can not do that!

9

## Course organization



- Problem solving (class + laboratory)
- Laboratory assignments and old exam for practicing
  - live

10

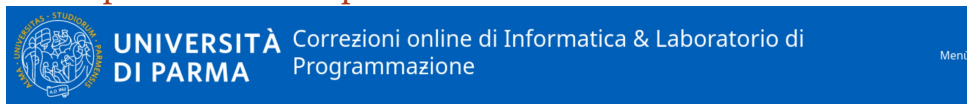
## Exam Rules

- C programming in laboratory
  - Initially we will discuss the assignment all together
  - ~2 hours
  - C
  - Modular approach
- Rating
  - No errors (and also warnings) when compiling
  - It must be executable
  - Please, try to pass the exam when you are ready

11

## Exam results

- <https://www.ce.unipr.it/didattica/informatica/esami/>



Da queste pagine potete accedere alle correzioni online delle prove pratiche di Informatica & Laboratorio di Programmazione  
[Ritorna](#) alla pagina del corso.

### Elenco prove pratiche:

#### A.A. 2023-2024

- |                      |                                                                                                               |
|----------------------|---------------------------------------------------------------------------------------------------------------|
| 1. 18 settembre 2024 | Allocazione dinamica di array bidimensionali, battaglia navale                                                |
| 2. 2 settembre 2024  | Memorizzazione dinamica di coppie di stringhe, palindromi e anagrammi                                         |
| 3. 31 luglio 2024    | Problema di Milena, lettura da file ASCII con righe con lunghezze differenti, calcolo massimo                 |
| 4. 17 giugno 2024    | Lettura, memorizzazione e somma polinomi. Array bidimensionali di numeri int                                  |
| 5. 4 aprile 2024     | Allocazione dinamica di un vettore, calcolo numero divisori primi                                             |
| 6. 19 febbraio 2024  | Controllo validità quadrati magici, allocazione dinamica matrici                                              |
| 7. 15 gennaio 2024   | Lettura file CSV in array di struct, ordinamento con qsort()                                                  |
| 8. 21 dicembre 2023  | Lettura di file contenenti cruciverba, array bidimensionali di char, conversione di array di char in stringhe |

#### A.A. 2022-2023

- |                      |                                                                   |
|----------------------|-------------------------------------------------------------------|
| 1. 15 settembre 2023 | Lettura file con array di stringhe, creazione stringa di codifica |
| 2. 4 settembre 2023  | Problema di Giuseppe, analisi array e relative allocazioni        |



12

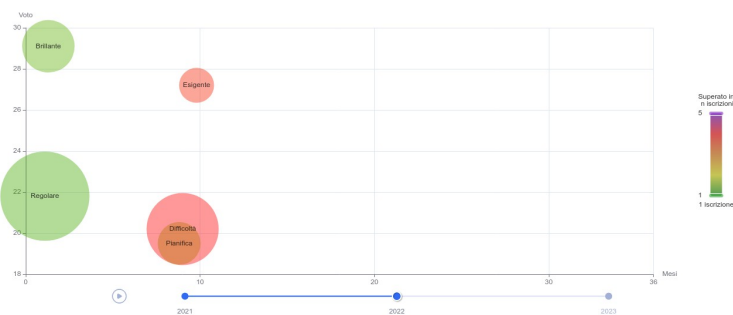


## How to approach this course

- Do not underestimate the effort needed to pass the exam
- Usually students that do not attend lessons → negative result
- Some statistics (2022 cohort)
  - Passed: 56.5% (91 students out of 161)
  - Not Passed: 43.5% (70 students out of 161)
  - Time average: 1.7 months
  - Average n. of attempts: 1.9
  - Average result: 23.5

13

## How to approach this course



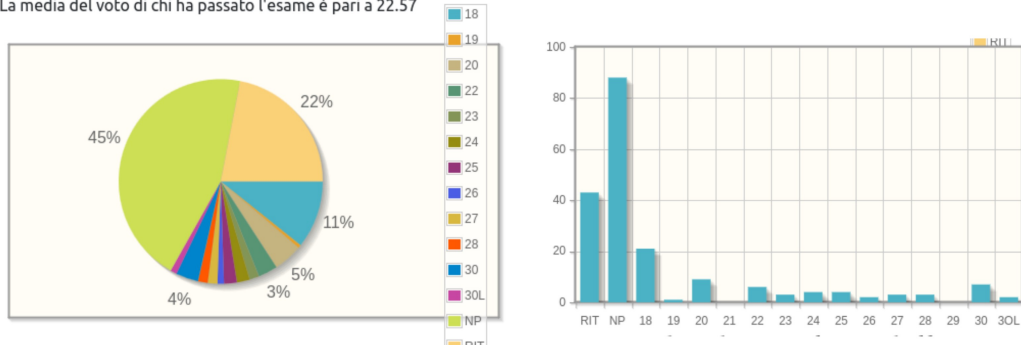
- Regular → less than 3 attempts, before 6 months from 1st registration, and result < 27
- Brilliant → less than 3 attempts, before 6 months from 1st registration, and result ≥ 27
- Demanding → more than 3 registrations ≥ 27
- Difficult → more than 3 registrations < 27
- Planned → less than 3 registrations but after 6 months

14

## How to approach this course

### Statistiche dei risultati dall'A.A. 2023-2024

Hanno passato la prova: 65 su 153 con un tasso pari a 42.48% oppure pari a 33.16% considerando anche i ritirati  
La media del voto di chi ha passato l'esame è pari a 22.57



15

## Student reception

- Tuesday afternoon (tentatively) > 16:00
- Bldg. 1 of Cubes (scientific venue of engineering), 1<sup>st</sup> floor



16

## Course material



- You can download
  - Slides we discuss during lessons
  - C examples we use with slides
  - Assignments we use in laboratory + code
- Slides & Code examples are mostly definitive (besides small corrections)
- Assignments → continuously updated

17

## Course material



- How to download it?
  - Github!
  - Via web
  - Using “git” o “repo”
- Detailed instructions at:
  - <https://www.ce.unipr.it/didattica/informatica/repo.php>

18

## Textbooks

- Good C manual
- Suggestions:
  - K.N. King, *C Programming: A Modern Approach (2<sup>nd</sup> Edition)*, W W Norton & Co
  - Bellini Guidi, *Linguaggio C*, Mc Graw Hill
  - Darnell Margolis, *C manuale di programmazione*, Mc Graw Hill

19

## Textbooks

| Appendix D Standard Library Functions 779 |                                                                                                                                                                                                                                                                                                                                                                                                      |  |
|-------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|
| <i>Returns</i>                            | <i>c</i> (the character written). If a write error occurs, <code>putchar</code> sets the stream's error indicator and returns EOF. 7.3, 22.4                                                                                                                                                                                                                                                         |  |
| <b>puts</b>                               | <b>Write String</b> <stdio.h>                                                                                                                                                                                                                                                                                                                                                                        |  |
|                                           | <code>int puts(const char *s);</code>                                                                                                                                                                                                                                                                                                                                                                |  |
|                                           | Writes the string pointed to by <i>s</i> to the <code>stdout</code> stream, then writes a new-line character.                                                                                                                                                                                                                                                                                        |  |
| <i>Returns</i>                            | A nonnegative value if successful. Returns EOF if a write error occurs. 13.3, 22.5                                                                                                                                                                                                                                                                                                                   |  |
| <b>putwc</b>                              | <b>Write Wide Character to File (C99)</b> <wchar.h>                                                                                                                                                                                                                                                                                                                                                  |  |
|                                           | <code>wint_t putwc(wchar_t c, FILE *stream);</code>                                                                                                                                                                                                                                                                                                                                                  |  |
|                                           | Wide-character version of <code>putc</code> . 25.6                                                                                                                                                                                                                                                                                                                                                   |  |
| <b>putwchar</b>                           | <b>Write Wide Character (C99)</b> <wchar.h>                                                                                                                                                                                                                                                                                                                                                          |  |
|                                           | <code>wint_t putwchar(wchar_t c);</code>                                                                                                                                                                                                                                                                                                                                                             |  |
|                                           | Wide-character version of <code>putc</code> . 25.6                                                                                                                                                                                                                                                                                                                                                   |  |
| <b>qsort</b>                              | <b>Sort Array</b> <stdlib.h>                                                                                                                                                                                                                                                                                                                                                                         |  |
|                                           | <code>void qsort(void *base, size_t nmem, size_t size, int (*compar)(const void *, const void *));</code>                                                                                                                                                                                                                                                                                            |  |
|                                           | Sorts the array pointed to by <i>base</i> . The array has <i>nmem</i> elements, each <i>size</i> bytes long. <i>compar</i> is a pointer to a comparison function. When passed pointers to two array elements, the comparison function must return a negative, zero, or positive integer, depending on whether the first array element is less than, equal to, or greater than the second. 17.7, 26.2 |  |
| <b>raise</b>                              | <b>Raise Signal</b> <signal.h>                                                                                                                                                                                                                                                                                                                                                                       |  |
|                                           | <code>int raise(int sig);</code>                                                                                                                                                                                                                                                                                                                                                                     |  |
|                                           | Raises the signal whose number is <i>sig</i> . 24.3                                                                                                                                                                                                                                                                                                                                                  |  |
| <i>Returns</i>                            | Zero if successful, nonzero otherwise. 24.3                                                                                                                                                                                                                                                                                                                                                          |  |
| <b>rand</b>                               | <b>Generate Pseudo-Random Number</b> <stdlib.h>                                                                                                                                                                                                                                                                                                                                                      |  |
|                                           | <code>int rand(void);</code>                                                                                                                                                                                                                                                                                                                                                                         |  |
| <i>Returns</i>                            | A pseudo-random integer between 0 and <code>RAND_MAX</code> (inclusive). 26.2                                                                                                                                                                                                                                                                                                                        |  |
| <b>realloc</b>                            | <b>Resize Memory Block</b> <stdlib.h>                                                                                                                                                                                                                                                                                                                                                                |  |
|                                           | <code>void *realloc(void *ptr, size_t size);</code>                                                                                                                                                                                                                                                                                                                                                  |  |
|                                           | <i>ptr</i> is assumed to point to a block of memory previously obtained from <code>calloc</code> , <code>malloc</code> , or <code>realloc</code> . <code>realloc</code> allocates a block of <i>size</i> bytes, copying the contents of the old block if necessary.                                                                                                                                  |  |
| <i>Returns</i>                            | A pointer to the beginning of the new memory block. Returns a null pointer if a block of the requested size can't be allocated. 17.3                                                                                                                                                                                                                                                                 |  |

20

|                                                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                                                                                                                                                                                             |
|--------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <div data-bbox="326 207 508 243">Textbooks</div> | <div data-bbox="685 182 1019 195">The ANSI Runtime Library465</div> <div data-bbox="685 224 987 254">A.14.2 Pseudo-Random Number Generator Functions</div> <div data-bbox="685 264 1019 287">The <i>rand()</i> and <i>srand()</i> functions enable you to generate pseudo-random numbers.</div> <div data-bbox="685 319 876 333">A.14.2.1 The <i>rand()</i> Function</div> <div data-bbox="699 342 818 365"> <pre>#include &lt;stdlib.h&gt; int rand( void );</pre> </div> <div data-bbox="685 375 1019 424">The <i>rand()</i> function returns an integer in the range 0 through RAND_MAX. Successive calls to <i>rand()</i> should produce different integers. However, the sequence of random numbers could be the same for each program execution unless you use a different seed value via the <i>srand()</i> function.</div> <div data-bbox="685 453 883 468">A.14.2.2 The <i>srand()</i> Function</div> <div data-bbox="699 476 896 499"> <pre>#include &lt;stdlib.h&gt; void srand( unsigned int seed );</pre> </div> <div data-bbox="685 510 1019 558">The <i>srand()</i> function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to <i>rand()</i>. If <i>srand()</i> is invoked with the same seed value, the sequence of generated numbers will be the same. The default seed value is 1.</div> <div data-bbox="685 588 953 602">A.14.3 Memory Management Functions</div> <div data-bbox="685 611 1019 636">The memory management functions enable you to allocate and deallocate memory dynamically. See Chapter 7 for more information about these functions.</div> <div data-bbox="685 663 885 678">A.14.3.1 The <i>calloc()</i> Function</div> <div data-bbox="699 688 958 711"> <pre>#include &lt;stdlib.h&gt; void *calloc( size_t nmemb, size_t size );</pre> </div> <div data-bbox="685 722 1019 783">The <i>calloc()</i> function allocates contiguous space for <i>nmemb</i> objects, each of which has a length in bytes specified by <i>size</i>. All bits in the allocated space are initialized to zero. <i>calloc()</i> returns a pointer to the first byte of the allocated space. If the space cannot be allocated, or if <i>nelem</i> or <i>size</i> is zero, <i>calloc()</i> returns a null pointer.</div> | <div data-bbox="1203 207 1369 254">UNIVERSITÀ DI PARMA</div> <div data-bbox="1318 753 1336 768">21</div> |
|                                                  | <div data-bbox="326 827 479 863">Contacts</div> <div data-bbox="391 976 719 1119"> <ul style="list-style-type: none"> <li>Massimo Bertozzi</li> <li>bertozzi@ce.unipr.it</li> <li>0521 90 5845</li> </ul> </div>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |                                                                                                                                                                                             |

This image shows a single sheet of white paper with horizontal ruling lines. The lines are evenly spaced and run across the width of the page. There are no margins, text, or other markings on the paper.



UNIVERSITÀ DI PARMA

# Cosa significa programmare?

*You cannot endow even the best machine  
with initiative.*

Walter Lippmann, A Preface to Politics

## Sommario



- Cosa significa programmare
- Parallelo con problemi “noti”
- Concetto di algoritmo
  - Problem solving
  - Formalismi
  - Approcci top-down e bottom-up

2/57

## Question time!



- Cosa significa per voi programmare?
- Risposte comuni:
  - Scrivere un programma
    - Un po' banale eh?
  - Scrivere del codice
    - Sicuramente ma è la parte meno importante
  - Usare un linguaggio di programmazione
    - Chi siete? Da dove venite? Cosa portate? Dove andate? Un Fiorino!



3/57

## Cosa significa programmare?



- Programmare significa...
  - 90% → Risolvere un problema!
  - 2% → Scrivere codice
  - 8% → Capire gli errori che abbiamo fatto
- Percentuali molto indicative (ma non troppo)

4/57



## Parallelo con programmazione



- Parallelo con cucina
  - Voglio invitare il nuovo vicino a pranzo
  - Ho già in mente cosa fare
  - Problema: come lo cucino?
  - Soluzione → ricetta

5/57

## Esempio Ricetta



### TORTA FRITTA

- |                                                                                                                                                                                                                                                                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Ingredienti<ul style="list-style-type: none"><li>– 1 kg Farina o</li><li>– 450 g Latte Intero Caldo</li><li>– 30 g Sale</li><li>– 30 g Lievito di Birra Fresco</li><li>– 1 noce di strutto</li><li>– Strutto per friggere</li></ul></li></ul> | <ul style="list-style-type: none"><li>• Preparazione<ul style="list-style-type: none"><li>– Sciogliamo 30 g di lievito di birra fresco in 50 g di latte</li><li>– Rovesciamo la farina e versiamo al centro 400 g di latte caldo</li><li>– Aggiungiamo una noce di strutto</li><li>– Mescoliamo con una forchetta</li><li>– Aggiungiamo il lievito e mescoliamo</li><li>– Aggiungiamo 30 g di sale</li><li>– Impastiamo formando un panetto</li><li>– ...</li></ul></li></ul> |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

6/57

## Esempio ricetta



- Caratteristiche ricetta:
  - Ingredienti
  - Come si usano
    - Passi da fare
    - E in che ordine!
  - Scritta in qualche “linguaggio”
- È sufficiente?
  - No
  - Esecuzione!

7/57

## Ricapitolando



- Devo risolvere un problema
- Ingredienti
- Individuo ricetta
  - Passi e loro ordine
- Scrivo la ricetta
- La eseguo

8/57

---

---

---

---

---

---

---

---

---

---

## Ricapitolando



- Devo risolvere un problema
- Ingredienti → Dati in ingresso
- Individuo ricetta → Scopro come risolvere
  - Passi e loro ordine → **Algoritmo**
- Scrivo la ricetta → Codifica
- La eseguo → Esecuzione

**Punti fondamentali della programmazione!**

9/57

## Punti fondamentali



- Problema
- Dati
- Individuazione algoritmo
- Codifica
- Esecuzione

10/57

---

---

---

---

---

---

---

---

---

---

## Punti fondamentali



- Problema
- Dati
- Individuazione algoritmo
- Codifica
- Esecuzione

11/57

## Il Problema



- Tipicamente richiesta committente
  - Nel vostro caso io!
- Va capito!
  - Leggere e rileggere
- Va “scomposto”
  - Sottoproblemi

GIVEN ONE HOUR TO SAVE  
THE WORLD, I WOULD SPEND  
**55 MINUTES** DEFINING THE  
PROBLEM, AND **5 MINUTES**  
FINDING THE SOLUTION.



12/57

## Il Problema

- Tipicamente richiesta committente
  - Nel vostro caso io!
- Classificazione
  - Risolvibili
  - Non resolvibili
  - Non affrontabili

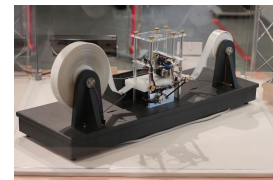


Alan Turing 1912-1954

13/57

## Classificazione Problemi

- Computabilità
  - Possibilità di risolvere un problema in maniera efficace
  - Non ha a che fare con il fatto che si conosca la soluzione
- Problemi resolvibili
  - È possibile arrivare a risolverli tramite un numero finito di passi
- Problemi non resolvibili
  - Non esiste un numero finito di passi che mi portino a risolverli
- Problemi non affrontabili
  - Numero di passi finito ma talmente elevato che non è possibile praticamente risolverli



14/57

## Punti fondamentali



- Problema
- Dati
- Individuazione algoritmo
- Codifica
- Esecuzione

15/57

## Dati in ingresso



- Tipicamente forniti con il problema
- Influenzano come risolvo il problema?
- Dal punto di vista teorico no
- Dal punto di vista pratico
  - Dipende...
  - La tipologia di dati può influenzare come risolvo il problema, linguaggio di programmazione ecc. ecc.

16/57

## Punti fondamentali



- Problema
- Dati
- Individuazione algoritmo
- Codifica
- Esecuzione

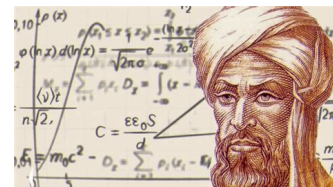
17/57

## Algoritmo



- Dal persiano al-Khwārizmī
- Un algoritmo è un metodo generale che risolve con una sequenza finita di mosse un problema dato.
- Mi dice come si risolve un problema:
  - Sequenza finita di azioni non ambigue
  - Però lo devo individuare io!

خوارزمی



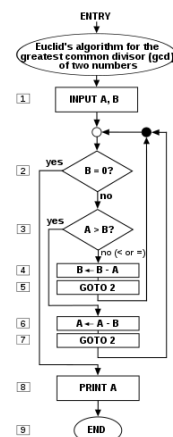
18/57

## Algoritmo

UNIVERSITÀ  
DI PARMA



- Insieme finito di passi non ambigui che specificano le operazioni da compiere per risolvere una classe di problemi
  - Passi risolutivi
  - Loro ordine



## Algoritmo: esempio

UNIVERSITÀ  
DI PARMA



- Problema: contattare il docente per chiarimenti
  - 1) Trovare mail docente nelle slide
  - 2) Aprire programma gestione mail
  - 3) Scrivere richiesta chiarimento indicando chi siete, quale corso seguite e anche la vostra matricola
  - 4) Premere invio
  - 5) Aspettare un tempo congruo...
  - 6) È arrivata risposta?
  - 7) No → ripetere dal punto 5
  - 8) Sì → leggerla
  - 9) Terminare

Esiŝte un altro algoritmo che  
risolve il problema?

20/57



## Algoritmo caratteristiche



- **Finito:** numero finito di passi
- **Non ambiguo:** interpretabile in maniera non equivoca
- **Terminazione:** deve terminare dopo un numero finito di passi
- **Generale:** deve affrontare il problema in generale
- **Deterministico:** a fronte degli stessi dati in ingresso deve fornire la stessa soluzione
- **Eseguibile:** deve essere effettivamente eseguibile
- **Efficiente:** minor numero di passi

21/57

## Algoritmo



- Individuare un algoritmo è la parte chiave del programmare
- L'esecuzione delle azioni nell'ordine specificato dall'algoritmo consente di ottenere a partire dai dati in ingresso i risultati che risolvono il problema
- Trovare un algoritmo è il **punto critico**

### algorithm

*noun*

Word used by programmers when they do not want to explain what they did.

22/57

## Problem Solving



- È la parte difficile...
- Per individuare un algoritmo è necessario:
  - attenta analisi del problema
  - suddivisione in sottoproblemi
  - definizione dei possibili ingressi e uscite
  - definizione delle strutture dati
  - definizione della sequenza di passi risolutiva



23/57

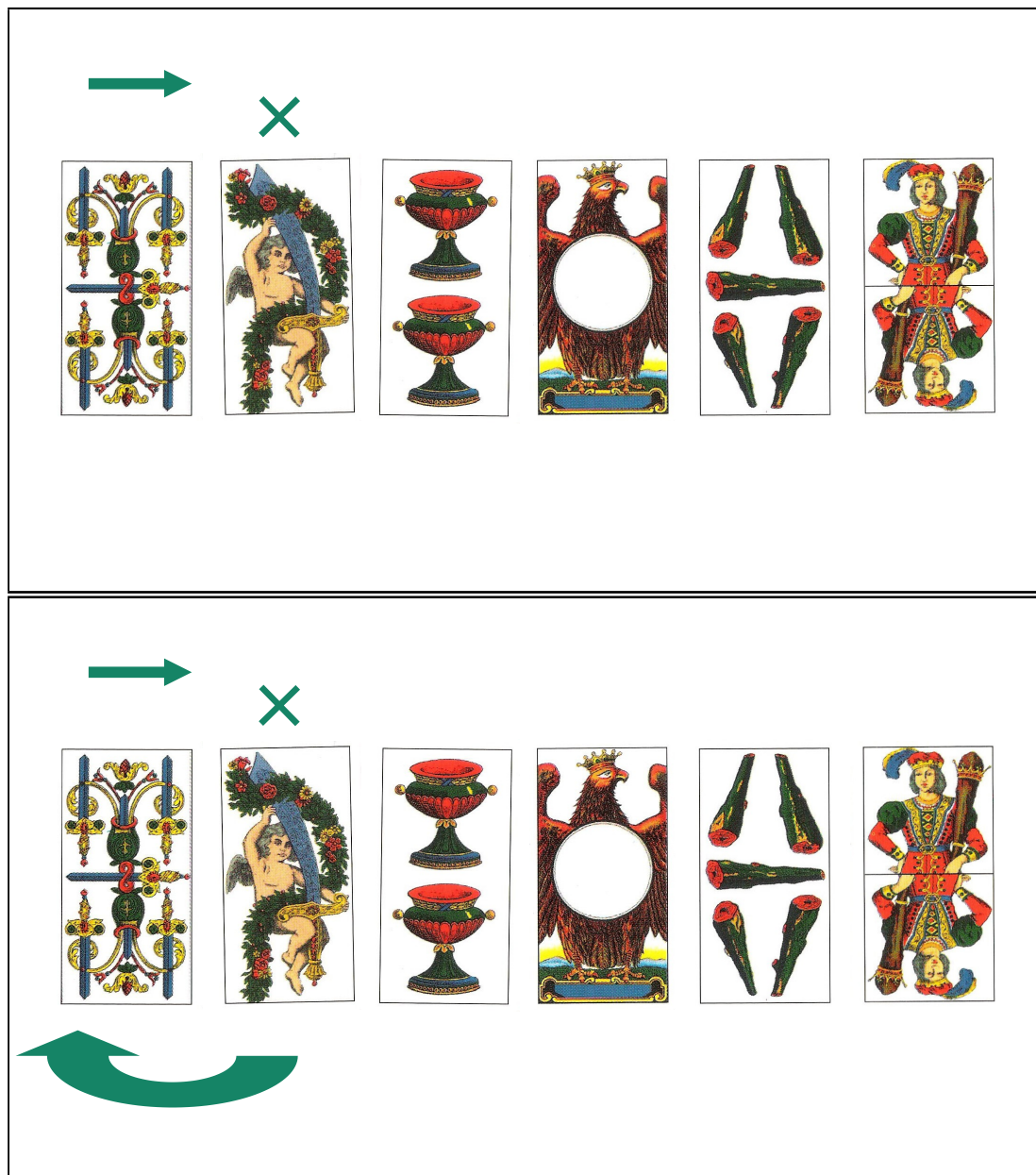
## Esempio

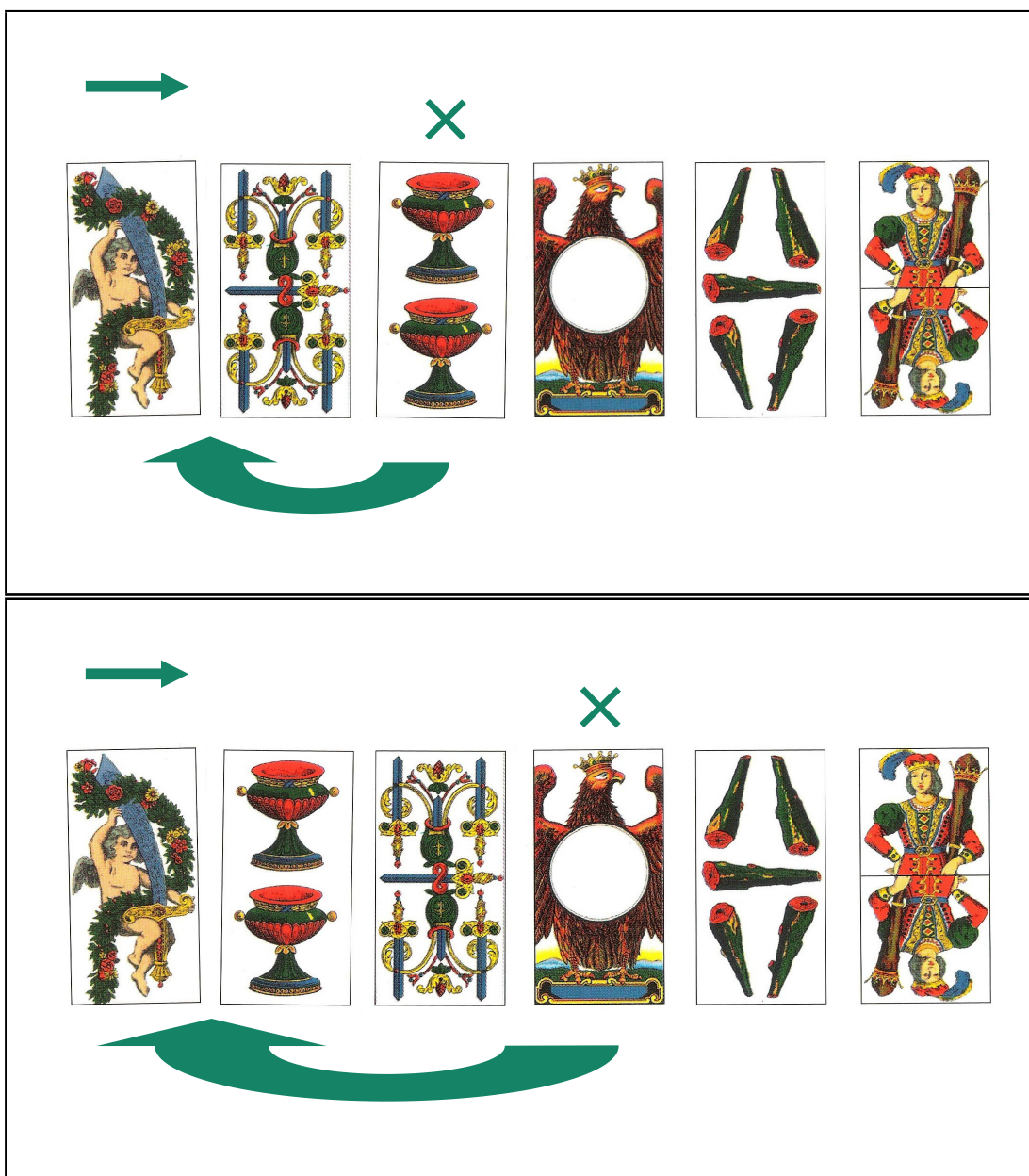


- Problema: ordinare le carte
- Approccio tipico: “insertion sort”
- Parto da sinistra verso destra e inserisco le carte che incontro nella posizione corretta

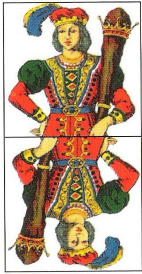
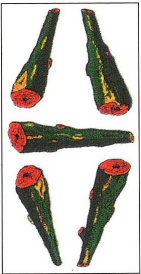
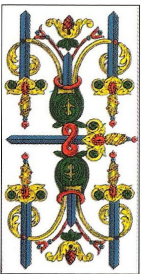
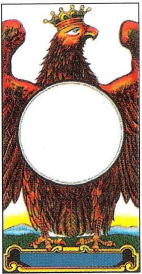


24/57



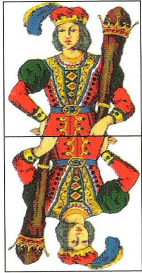
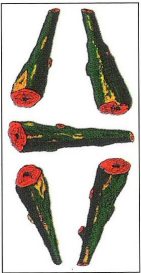
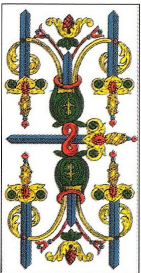
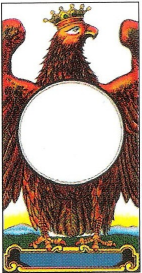














---

---

---

---

---

---

---

---

---

---

## Problem solving



- Formalizziamo il problema: “Ordinare le carte”
  - “Parto da sinistra verso destra e inserisco le carte che incontro nella posizione corretta”
- È già un grado di dettaglio sufficiente?
- Esiste la possibilità di suddividerlo in sottoproblemi?

31/57

## Esempio



- Parto a considerare le carte da sinistra una alla volta
- La carta in esame è già nella posizione corretta?
  - NO: la scambio
  - SÌ: procedo oltre
- Sono arrivato a considerare tutte le carte?
  - SÌ: termino
  - NO: continuo

Ma è un grado di dettaglio sufficiente?

32/57

## Esempio



- La carta in esame è già nella posizione corretta?
  - NO: la scambio
  - SÌ: procedo oltre

Cosa significa posizione corretta?  
Come la scambio?

33/57

## Esempio



- La carta in esame è già nella posizione corretta?
  - NO: la scambio
  - SÌ: procedo oltre
- Confronto la carta in esame con quella a sinistra, ha lo stesso valore o superiore?
  - SÌ: procedo oltre
  - NO:
    - Mi sposto a sinistra fino a che non trovo una carta che ha valore inferiore o fino a che non arrivo all'inizio
    - Sposto tutte le carte tra la carta in esame e quelle appena esaminate di una posizione a sinistra
    - Sposto la carta in esame nello "spazio" che ho creato

34/57



## Approccio al problem solving



- Cosa occorre sapere:
  - Affrontare i problemi
  - Analizzare i problemi
  - Modellare i problemi
  - Risolvere i problemi in modo efficace o efficiente o rispettando altri vincoli

35/57

## Approccio top-down



- Come si procede?
  - Identificare i problemi principali
  - Decomporli in sottoproblemi
  - Ripetere fino a che si raggiungono problemi elementari
- In pratica si parte dal problema complessivo (alto) e lo si scompone iterativamente

36/57



## Approccio bottom-up



- Si parte dalle “primitive” del linguaggio (basso)
- Le si assembla in funzioni più complesse
- Risultato finale → applicazione

37/57

## Top-down vs Bottom-up



- In realtà dipende dal caso d'uso
- Approccio bottom-up permette testing prima
- Top-down più adeguato per problem solving
- Quasi sempre approcci intermedi

Usate approccio  
TOP-DOWN

38/57

## Rappresentazione degli algoritmi



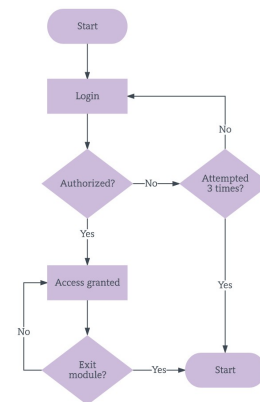
- Le lingue naturali si prestano comunque ad ambiguità
- Esistono metodi più formali per rappresentare un algoritmo
  - Diagrammi di flusso (flow chart)
  - Pseudocodice

39/57

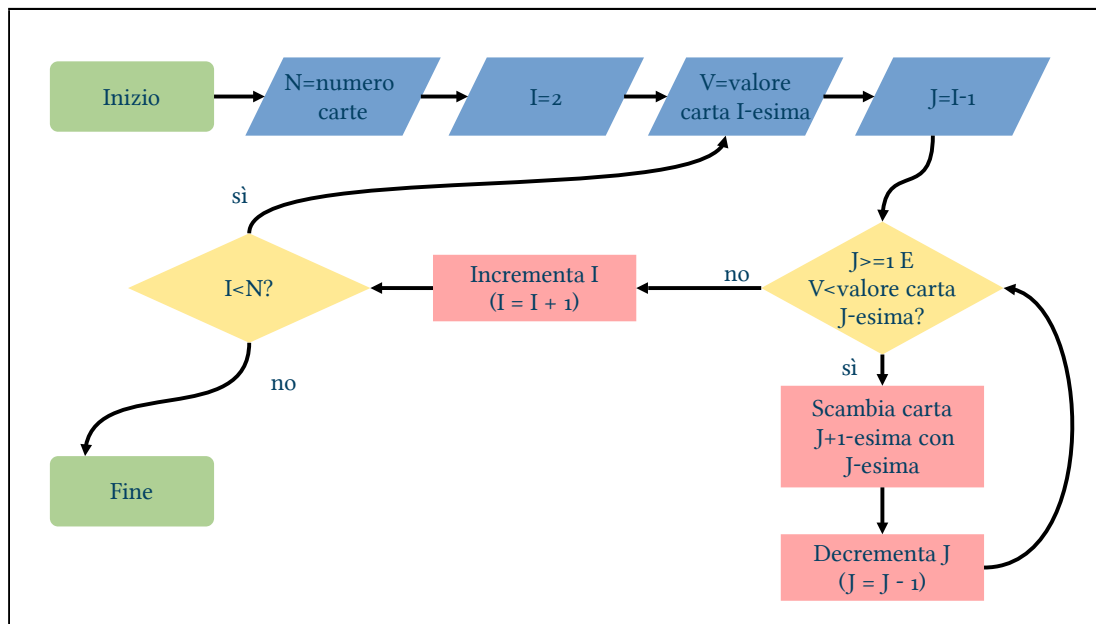
## Algoritmi: diagrammi di flusso



- Diagramma costituito da **nodi** e **archi orientati**
- Ogni nodo corrisponde ad un passo dell'algoritmo
  - Spesso forme differenti in base alla tipologia
    - Assegnamento
    - Calcolo
    - Decisione
    - Inizio/Fine
- Gli archi collegano i passi nell'ordine da seguire



40/57



## Algoritmi: pseudocodice



- Linguaggio speciale che codifica i passi
  - Meno soggetto ai limiti dei diagrammi di flusso
- Quasi un linguaggio di programmazione
  - Più semplice
  - Svincolato (ma non troppo) allo specifico linguaggio di programmazione

## Algoritmi: pseudocodice



```
N ← Length(Cards)
For I ← 2 To N
    V ← Cards[I]
    J ← I - 1
    While J >= 1 AND Cards[J] > V
        Cards[J + 1] = Cards[J]
        J = J - 1
```

43/57

## Punti fondamentali



- Problema
- Dati
- Individuazione algoritmo
- **Codifica**
- Esecuzione

44/57

## Codifica



- Algoritmo → Codice Sorgente
- Traduco i passi di un algoritmo nelle corrispondenti istruzioni del linguaggio di programmazione
  - Se ho ben strutturato algoritmo quasi 1:1
  - Spesso “algoritmo” pensato per il linguaggio
- Unici problemi:
  - Sintassi & Lessico
  - Devo conoscere il linguaggio

45/57

## Esempio codifica



- Problema: calcolare la somma di quattro numeri forniti dall'utente
- Algoritmo:
  - 1) Chiedi all'utente il primo numero e memorizzalo
  - 2) Chiedi secondo numero...
  - 3) Chiedi terzo numero...
  - 4) Chiedi quarto numero...
  - 5) Calcola la loro somma
  - 6) Stampa la loro somma

46/57

---

---

---

---

---

---

---

---

---

---

## Esempio: codifica BASIC

UNIVERSITÀ  
DI PARMA

```
10 INPUT "Primo numero: ", A
20 INPUT "Secondo numero: ", B
30 INPUT "Terzo numero numero: ", C
40 INPUT "Quarto numero: ", D
50 S = A + B + C + D
60 PRINT "Il risultato e' ", S
```

47/57

## Esempio: codifica Python

UNIVERSITÀ  
DI PARMA

```
#!/usr/bin/python

a = int(input("Primo numero: "))
b = int(input("Secondo numero: "))
c = int(input("Terzo numero: "))
d = int(input("Quarto numero: "))

s = a + b + c + d

print "Il risultato e' = ", s
```

48/57

## Esempio: codifica C++

```
#include <iostream>

int main(int argc, char **argv){
    int a, b, c, d;

    std::cout << "primo numero: " << std::endl;
    std::cin >> a;
    std::cout << "secondo numero: " << std::endl;
    std::cin >> b;
    std::cout << "terzo numero: " << std::endl;
    std::cin >> c;
    std::cout << "quarto numero: " << std::endl;
    std::cin >> d;
    int s = a + b + c + d;
    std::cout << "Il risultato e' = " << s << std::endl;
    return 0;
}
```

49/57

## Codifica: Codice Sorgente

- Testo scritto in accordo alla sintassi e alla semantica di un linguaggio di programmazione
  - Ottengo: Programma Sorgente
  - A seconda del linguaggio utilizzato:
    - Interpretazione
    - Compilazione
    - Situazioni intermedie

50/57

## Punti fondamentali

UNIVERSITÀ  
DI PARMA

- Problema
- Dati
- Individuazione algoritmo
- Codifica
- Esecuzione

51/57

## Esecuzione

UNIVERSITÀ  
DI PARMA

- Fase di esecuzione del programma
  - Nel caso del C deve essere preceduta da compilazione
  - Programma Sorgente → Eseguitibile
- Ben difficilmente è la parte finale!
  - Debug



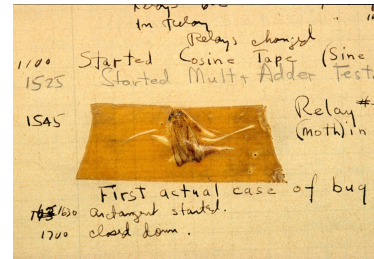
52/57



## Debug



- Mediamente 15-50 errori per kLOC (Lines Of Code)
  - Programmatori professionisti
- Debug
  - Rileggere codice
  - Codice ad hoc: esempio stampa
  - Strumenti specifici → debugger



1945

53/57

## Esecuzione



- Chi esegue il programma?
  - Diciamo “il computer”
- E quando esegue cosa fa esattamente?
  - Quello che abbiamo scritto noi!
  - Nessuna possibilità di decisione
  - Computer = mero esecutore di ordini o **automa**

Informatica = Informazione + Automatica

54/57

## Considerazioni finali



- Quanto visto in queste slide è indipendente dal linguaggio di programmazione
- Infatti prima di programmare:
  - 1) **Devo** capire come risolvere quanto è stato assegnato: “il problema”
  - 2) Una volta che l’ho capito **devo** individuare i passi necessari e il loro ordine: “algoritmo risolutivo”
  - 3) **Solamente** quando ho individuato l’algoritmo risolutivo posso scrivere il codice: “codifica”

55/57

## Considerazioni finali



56/57



UNIVERSITÀ DI PARMA

# Cosa significa programmare?



*You cannot endow even the best machine  
with initiative.*

Walter Lippmann, A Preface to Politics



UNIVERSITÀ DI PARMA

# Il Computer

*The idea behind digital computers may be explained by saying that these machines are intended to carry out any operations which could be done by a human computer*

Alan Turing

## Sommario

UNIVERSITÀ  
DI PARMA

- Cosa significa computer
- Un po' di storia
- Il modello di Von Neumann

2/31

# Computer

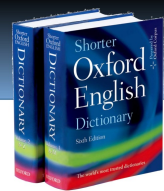
UNIVERSITÀ DI PARMA 



3/31

# Computer

UNIVERSITÀ DI PARMA 



- **Computer**  
Noun, Brit. /kəm'pjʊ:tə/, U.S. /kəm'pjudər/
  1. A person who makes calculations or computations; a calculator, a reckoner; spec. a person employed to make calculations in an observatory, in surveying, etc. Now chiefly *historical*.
  2. A device or machine for performing or facilitating calculation.
  3. An electronic device (or system of devices) which is used to store, manipulate, and communicate information, perform complex calculations, or control or regulate other devices or machines, and is capable of receiving information (data) and of processing it in accordance with variable procedural instructions (programs or software); esp. a small, self-contained one for individual use in the home or workplace, used esp. for handling text, images, music, and video, accessing and using the internet, communicating with other people (e.g. by means of email), and playing games

4/31

## Computer



- In Italiano
  - Elaboratore
  - Calcolatore (elettronico)
- Dispositivo *elettronico* e digitale
  - **programmabile**
  - per **uso generale**

5/31

## Programmabile

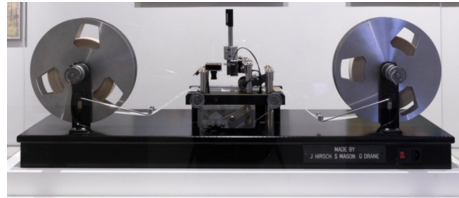


- Esempi di dispositivi programmabili
  - Sveglia
  - Lavatrice
  - Microonde
  - Ecc.
- Programmabile significa che posso impostare le operazioni da eseguire

6/31

## Per uso generale

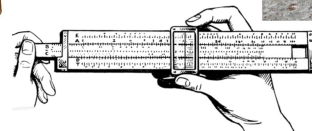
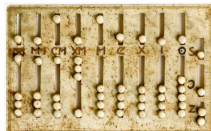
- General Purpose
  - Non specializzato
  - Adattabile per ogni utilizzo
- Da un punto di vista teorico non ci sono limiti



7/31

## Un po' di storia

- La maggior esigenza di automazione che l'uomo ha avuto per lungo tempo è stata quella dei calcoli
- Nella storia sono apparsi differenti soluzioni per “agevolare” tali operazioni



8/31

## Dispositivi programmabili



- Appaiono in Francia a fine 700
  - Inventore italiano
- Telaio Jacquard (1804)
  - Nastro con fori che viene fatto avanzare durante la lavorazione
  - Contrappesi che controllano lo schema di tessitura
  - Quando il contrappeso incontra il foro agisce sull'ordito



Linguaggio di Programmazione → schema di fori

9/31

## Primo “computer”



- Macchina analitica di Babbage (1833)
  - *On the Mathematical Power of the Calculating Engine*
- Mai costruita
  - 30 x 25 metri gestita da motore a vapore
- Schede perforate
  - Piccole → operazioni
  - Grandi → dati (variabili)



Linguaggio di Programmazione → schema di fori

10/31

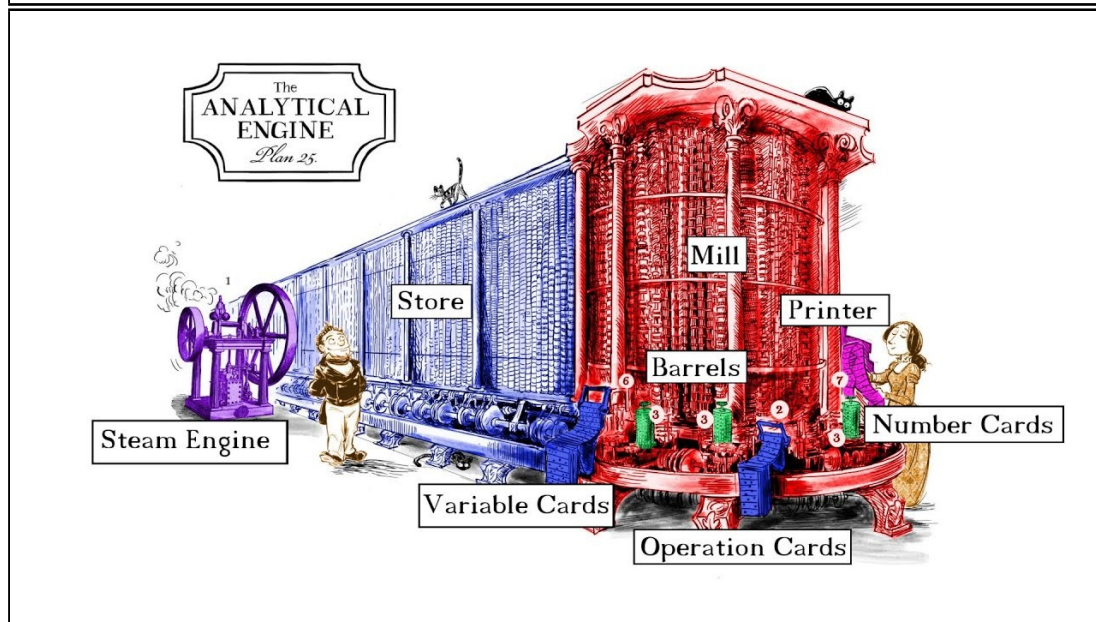


## Analytical Engine

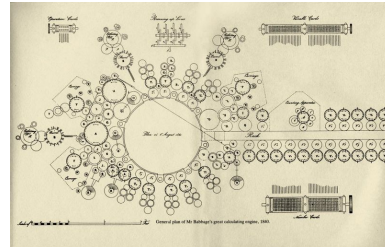
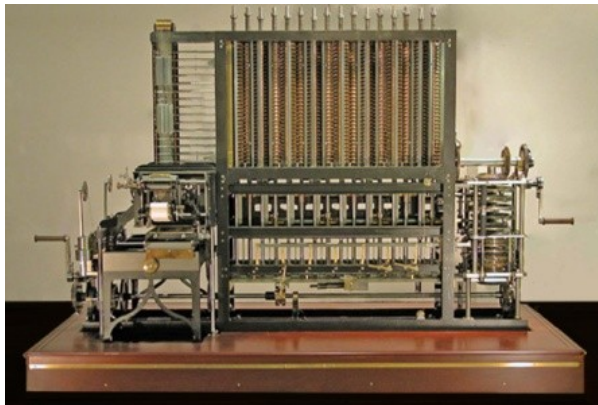


- La macchina di Babbage contiene già tutti gli elementi di un moderno computer
  - Controllo → lettura e interpretazione schede di istruzioni
  - Elaborazione → il *Mill* in grado di eseguire le 4 operazioni principali
    - Opzionalmente anche la radice quadrata
    - Procedure "barrel"
  - Memoria → registri interni fino a 1000 numeri da 40 cifre a virgola fissa (~17 kByte)
  - Input → le schede
  - Output → sistema di perforazione schede (stampa)
- Linguaggio di programmazione *Turing completo*
  - Equivalente ad una macchina di Turing universale

11/31



## Analytical Engine

UNIVERSITÀ  
DI PARMA

13/31

## Mark I (1943)

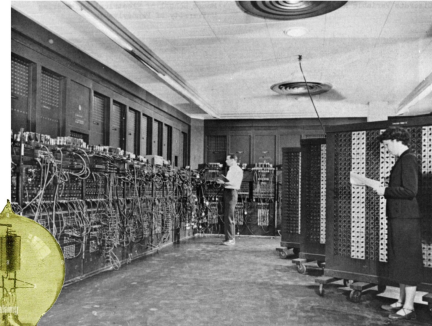
UNIVERSITÀ  
DI PARMA

- IBM + Università di Harvard
- Elettromeccanico
- Sempre schede perforate
- 3 addizioni o sottrazioni al secondo
- 1 moltiplicazione in 6 secondi
- 1 divisione in oltre 15 secondi
- Logaritmi e operazioni trigonometriche > 1 minuto

14/31

## ENIAC (1946)

- **Esclusivamente elettronico**
- Valvole termoioniche
  - Problema affidabilità (2 minuti)
  - I transistor furono inventati nel 1947
- Circa 1000 volte più veloce del Mark I
- Sempre schede perforate e sistema decimale
- **Turing Completo**



15/31

## Zuse Z4 (1939-1945)

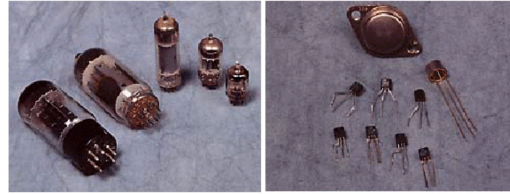
- Non governativo
  - Primo computer commerciale
- Ancora elettromeccanico
- Numeri a virgola mobile **binari**



16/31

## Transistor

### FROM VACUUM TUBES TO TRANSISTORS

UNIVERSITÀ  
DI PARMA

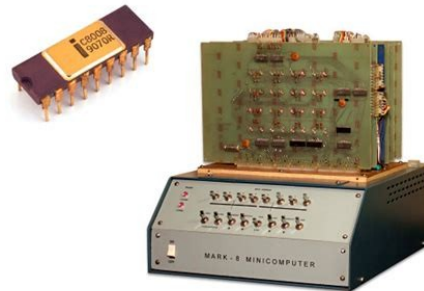
The miniaturization of computers began with the invention of the transistor on December 23rd 1947 by William Shockley, John Bardeen and Walter Brattain at Bell Laboratories. Early computers like the ENIAC contained thousands of vacuum tubes, millions of soldered joints and consumed 100's of kilowatts of electricity. Transistors would replace the tubes, greatly reduce the heat output and space requirements.

17/31

## Microprocessori (1971)

UNIVERSITÀ  
DI PARMA

- Dai transistor ai microprocessori
- Intel 4004
  - ~2300 transistor
  - 10.000 nm
  - Fino a 740 kHz
- Stessa potenza dell'ENIAC
  - 84 m<sup>3</sup>
- Calcolatrici...



18/31

## MITS Altair 8800 (1975)

UNIVERSITÀ  
DI PARMA



- Primo vero personal computer (Intel 8080)
  - 397 USD in kit o 495 USD già montato
  - ~ 1700 EUR di oggi
- 256 byte di RAM
- No ROM ma avvio manuale
- I/O nastro magnetico (input) e led (output)
  - No tastiera di serie
  - No display
  - No storage interno
- Altair BASIC → MicroSoft



19/31

## PC

UNIVERSITÀ  
DI PARMA



- In pochi anni diventano disponibili altri elementi
  - Tastiera
  - Video
  - Storage
- Dopo l'Altair nascono i Personal Computer
  - IBM PC (1981)
  - Olivetti M24 (1983)
  - Macintosh (1984)
- Dal punto di vista organizzativo dal 1946 è comunque cambiato poco

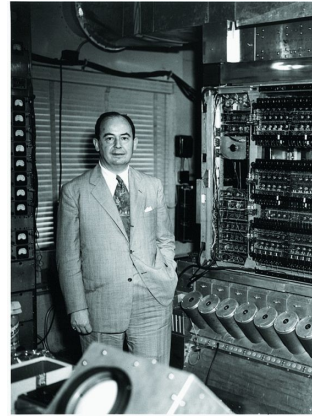


20/31

## Modello di Von Neumann (1946)

UNIVERSITÀ  
DI PARMA

- Modello di organizzazione **logica** di un computer
- Tutt'ora attuale!
- 5 parti fondamentali
  - Memoria (unica!)
  - Unità di elaborazione aritmetica e logica
  - Unità di controllo
  - Input/Output

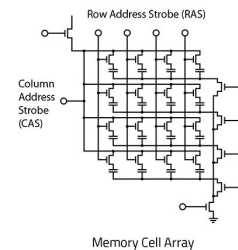
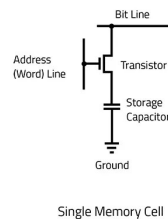


21/31

## Memoria

UNIVERSITÀ  
DI PARMA

- RAM → Random Access Memory
  - Diverse unità di stoccaggio “dati”
  - Celle elementari con valore fittizio
  - **Bit** organizzati in unità di accesso (**Byte**)
  - Ogni cella ha un ben preciso **indirizzo**
- Cosa contiene?
  - Dati
  - Ma anche istruzioni!

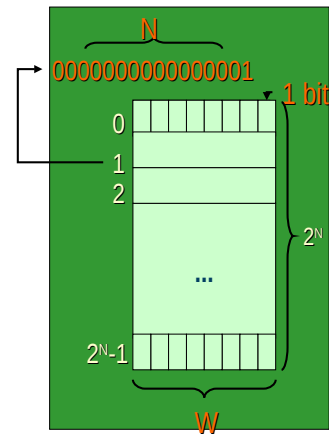


**Numeri → convenzionalmente composti da 0 e 1**

22/31

## Memoria

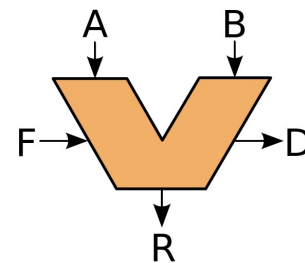
- RAM
  - Distinguiamo indirizzo e contenuto
- Memory Width (W)
  - Quanti bit per cella?
  - Tipicamente 8 bit  $\rightarrow$  1 Byte
- Address width (N)
  - Quanti bit per indirizzo?
  - Limita il quantitativo massimo di memoria
  - N bit  $\rightarrow 2^N$  possibili indirizzi



23/31

## Unità di Elaborazione

- Componenti
  - Arithmetic Logic Unit - ALU
    - Operazioni matematiche (+, -, /, \* ...)
    - Interi o virgola mobile
    - Operazioni logiche (>, <, =, and, or, not...)
    - Circuiti che eseguono le relative operazioni
  - Registri ovvero memorie rapide per funzionamento interno
    - Registri ad uso "generale"
    - Registri per accesso memoria
    - Registro di stato o flag
    - ...



A, B  $\rightarrow$  operandi  
 R  $\rightarrow$  risultato  
 F  $\rightarrow$  operazione  
 D  $\rightarrow$  status

24/31



## Unità di controllo



- Dove si trova il programma da eseguire?
  - In memoria → “numeri”
- La control unit esegue i programmi ovvero:
  - Legge da memoria la relativa istruzione da eseguire → **Fetch**
  - Decodifica il suo significato ovvero cosa deve fare → **Decode**
  - Esegue quanto richiesto pilotando gli altri sistemi → **Execute**
  - Scrive eventuali risultati elaborazione → **Store**
  - Ripete...
- Registri per lettura istruzione (IR) e posizione lettura (PC)

25/31

## Esempio istruzione



- Come è fatta una istruzione decodificata dall'unità di controllo?
- Tipicamente più “parti”
  - Operation code → mi dice che istruzione è
  - Operando/i → mi fa capire su cosa si opera
- Instruction Set → dipende da processore!
- Esempio: ADD X, Y
  - Somma il contenuto della memoria all'indirizzo X a quello in Y e scrivi il risultato in X

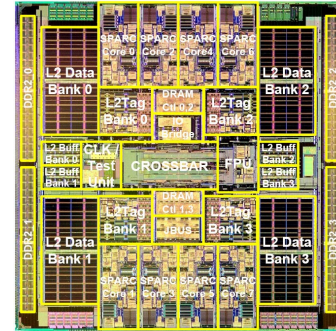
| Opcode (8 bit) | Indirizzo X (16 bit) | Indirizzo Y (16 bit) |
|----------------|----------------------|----------------------|
| 00001001       | 0000000001100011     | 0000000001100100     |

26/31



## CPU

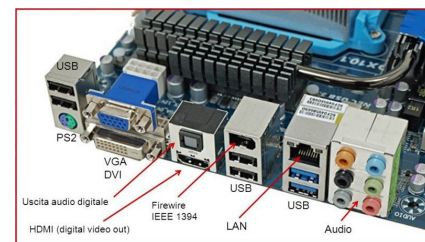
- Distinzione tra unità di elaborazione e controllo più fisicamente
  - CPU → Central Processing Unit
- Caratteristiche evolute:
  - Multiprocessing (superscalari) → ALU multiple
  - Multicore → anche più unità di controllo



27/31

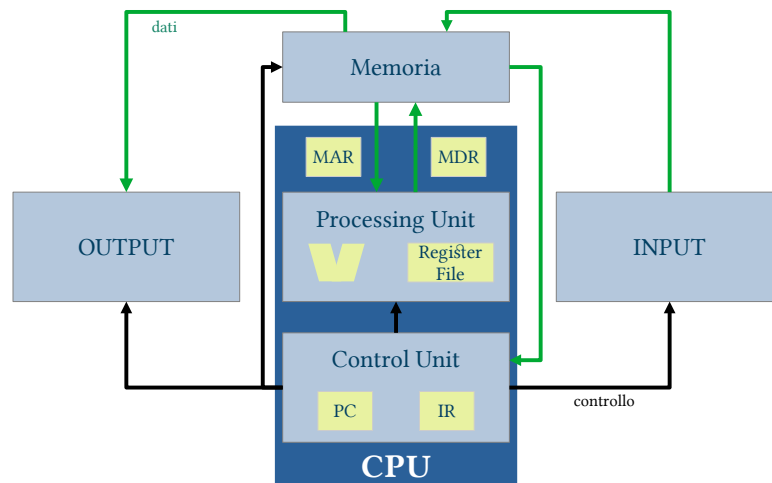
# I/O

- Gestione device “esterni”
  - Tastiera/Video
  - Hard disk, hardware vario...
- Quasi mai connessione sincrona
  - Velocità differenti
  - Buffer sia in ingresso che in uscita
    - Memoria per i dati in ingresso/uscita
    - Trasferimento da o verso RAM



28/31

## Von Neuman model

UNIVERSITÀ  
DI PARMA

29/31

## Von Neumann Model

UNIVERSITÀ  
DI PARMA

- Modello ancora attuale
- Il modello si riflette in quanto dovremo fare per programmare
  - CPU esegue sequenze di istruzioni
  - Il programmatore individua la sequenza di istruzioni appropriata
  - Primi linguaggi di programmazione → linguaggi imperativi

30/31



UNIVERSITÀ DI PARMA

# Il Computer



*The idea behind digital computers may be explained by saying that these machines are intended to carry out any operations which could be done by a human computer*

Alan Turing



UNIVERSITÀ DI PARMA

# I Linguaggi di Programmazione

*There will always be things we wish to say in  
our programs that in all languages can only  
be said poorly*

Alan Perlis

## Sommario

UNIVERSITÀ  
DI PARMA



- Piccolo riepilogo
- Il linguaggio macchina
- Assembly
- Linguaggi di Alto Livello
- Linguaggi Interpretati e Compilati

2/21

## Riepilogo: Unità di Controllo



- Dove si trova il programma da eseguire?
  - In memoria → “numeri”
- La control unit esegue i programmi ovvero:
  - Legge da memoria la relativa istruzione da eseguire → **Fetch**
  - Decodifica il suo significato ovvero cosa deve fare → **Decode**
  - Esegue quanto richiesto pilotando gli altri sistemi → **Execute**
  - Scrive eventuali risultati elaborazione → **Store**
  - Ripete...

3/21

## Riepilogo: Esempio Istruzione



- Come è fatta una istruzione decodificata dall'unità di controllo?
- Tipicamente più “parti”
  - Operation code → mi dice che istruzione è
  - Operando/i → mi fa capire su cosa si opera
- Instruction Set → dipende da processore!
- Esempio: ADD X, Y
  - Somma il contenuto della memoria all'indirizzo X a quello in Y e scrivi il risultato in X

| Opcode (8 bit) | Indirizzo X (16 bit) | Indirizzo Y (16 bit) |
|----------------|----------------------|----------------------|
| 00001001       | 000000001100011      | 0000000001100100     |

4/21

## Linguaggio Macchina

UNIVERSITÀ  
DI PARMA

- Codice numerico che corrisponde alle possibili istruzioni eseguibili da una CPU
- Diverse per ogni tipo di processore
  - Instruction Set
- Ad esempio l'Instruction Set dei processori ARM è completamente differente da quello dei processori Intel x86
  - Differenti modelli della stessa famiglia di processori presentano Instruction Set con variazioni

5/21

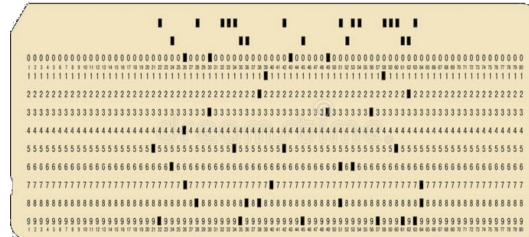
6502 Machine Language Hex Codes

|                                | Immediate | Accumulator | Zero Page | Zero Page X | Zero Page Y | Absolute | Absolute X | Absolute Y | Indirect | (Zero Page X) | (Zero Page Y) | Indirect | Relative | Address Flag |
|--------------------------------|-----------|-------------|-----------|-------------|-------------|----------|------------|------------|----------|---------------|---------------|----------|----------|--------------|
| Add with Carry                 | 69        | 65          | 75        |             |             | 6D       | 7D         | 79         |          | 6E            | 71            |          |          | N,Z,C        |
| Bitwise AND with Accumulator   | 29        | 25          | 35        |             |             | 2D       | 3D         | 39         |          | 21            | 31            |          |          | N,Z          |
| Branch on Carry Clear          | 0A        | 06          | 16        |             |             | 0E       | 1E         |            |          |               |               |          |          | N,Z,C        |
| Branch on Carry Set            |           |             |           |             |             |          |            |            |          |               |               |          |          |              |
| Branch on Equal                |           |             |           |             |             |          |            |            |          |               |               |          |          |              |
| Test Bits                      |           |             | 24        |             |             | 2C       |            |            |          |               |               |          |          | N,V,Z        |
| Branch on Minus                |           |             |           |             |             |          |            |            |          |               |               |          |          |              |
| Branch on Not Equal            |           |             |           |             |             |          |            |            |          |               |               |          |          |              |
| Branch on Plus                 |           |             |           |             |             |          |            |            |          |               |               |          |          |              |
| Break                          |           |             |           |             |             |          |            |            |          |               |               | 00       |          | B            |
| Branch on Overflow Clear       |           |             |           |             |             |          |            |            |          |               |               |          | 50       |              |
| Branch on Overflow Set         |           |             |           |             |             |          |            |            |          |               |               |          | 70       |              |
| Clear Carry                    |           |             |           |             |             |          |            |            |          |               |               |          | 18       | C            |
| Clear Decimal                  |           |             |           |             |             |          |            |            |          |               |               |          | 08       | D            |
| Clear Interrupt                |           |             |           |             |             |          |            |            |          |               |               |          | 58       | I            |
| Clear Overflow                 |           |             |           |             |             |          |            |            |          |               |               |          | B8       | V            |
| Compare Accumulator            | C9        | C5          | D5        |             |             | CD       | DD         | D9         |          | C1            | D1            |          |          | N,Z,C        |
| Compare X Register             | E0        |             | E4        |             |             | EC       |            |            |          |               |               |          |          | N,Z,C        |
| Compare Y Register             | C0        |             | C4        |             |             | CC       |            |            |          |               |               |          |          | N,Z,C        |
| Decrement Memory               |           |             | C8        | D6          |             | CE       | DE         |            |          |               |               |          |          | N,Z          |
| Decrement X                    |           |             |           |             |             |          |            |            |          |               |               |          | CA       | N,Z          |
| Decrement Y                    |           |             |           |             |             |          |            |            |          |               |               |          | B8       | N,Z          |
| Bitwise Exclusive OR           | 49        | 45          | 55        |             |             | 4D       | 5D         | 59         |          | 41            | 51            |          |          | N,Z          |
| Increment Memory               |           |             | E6        | F6          |             | EE       | FE         |            |          |               |               |          | E8       | N,Z          |
| Increment X                    |           |             |           |             |             |          |            |            |          |               |               |          | C8       | N,Z          |
| Increment Y                    |           |             |           |             |             |          |            |            |          |               |               |          |          |              |
| Jump                           |           |             |           |             |             | 4C       |            |            | 6C       |               |               |          |          |              |
| Jump to Subroutine             |           |             |           |             |             | 20       |            |            |          |               |               |          |          |              |
| Load Accumulator               | A0        | A5          | B5        |             |             | AD       | BD         | B9         |          | A1            | B1            |          |          | N,Z          |
| Load X Register                | A2        | A6          | B6        |             |             | AE       | BE         |            |          |               |               |          |          | N,Z          |
| Load Y Register                | A0        | A4          | B4        |             |             | AC       | BC         |            |          |               |               |          |          | N,Z          |
| Logical Shift Right            |           | 4A          | 56        |             |             | 4E       | 5E         |            |          |               |               |          |          | N,Z,C        |
| No Operation                   |           |             |           |             |             |          |            |            |          |               |               |          | EA       |              |
| Bitwise OR with Accumulator    |           |             |           |             |             |          |            |            |          |               |               |          |          | N,Z          |
| Push Accumulator to Stack      | 09        | 05          | 15        |             |             | 0D       | 1D         | 19         |          | 01            | 11            |          |          |              |
| Push Processor Status to Stack |           |             |           |             |             |          |            |            |          |               |               |          | 48       |              |
| Pop Accumulator off Stack      |           |             |           |             |             |          |            |            |          |               |               |          | 08       |              |
| Pop Processor Status off Stack |           |             |           |             |             |          |            |            |          |               |               |          | 68       |              |
| Rotate Left                    |           | 2A          | 26        | 36          |             | 2E       | 3E         |            |          |               |               |          | 28       | A8           |
| Rotate Right                   |           | 6A          | 66        | 76          |             | 6E       | 7E         |            |          |               |               |          | 40       | N,Z,C        |
| Return from Interrupt          |           |             |           |             |             |          |            |            |          |               |               |          | 60       | A8           |
| Return from Subroutine         |           |             |           |             |             |          |            |            |          |               |               |          |          |              |
| Subtract with Carry            | E9        | E5          | F5        |             |             | ED       | FD         | F9         |          | E1            | F1            |          |          | N,V,Z,C      |
| Set Carry                      |           |             |           |             |             |          |            |            |          |               |               |          | 38       | C            |
| Set Decimal                    |           |             |           |             |             |          |            |            |          |               |               |          | F8       | D            |
| Set Interrupt                  |           |             |           |             |             |          |            |            |          |               |               |          | F8       | I            |
| Store Accumulator              |           | 85          | 95        |             |             | 8D       | 9D         | 99         |          | 81            | 91            |          |          |              |
| Store X Register               |           | 86          |           | 96          |             | 8C       |            |            |          |               |               |          | AA       | N,Z          |
| Store Y Register               |           | 84          | 94        |             |             | 8C       |            |            |          |               |               |          | AA       | N,Z          |
| Transfer A to X                |           |             |           |             |             |          |            |            |          |               |               |          | BA       | N,Z          |
| Transfer Stack Pointer to X    |           |             |           |             |             |          |            |            |          |               |               |          | BA       | N,Z          |
| Transfer X to A                |           |             |           |             |             |          |            |            |          |               |               |          | BA       | N,Z          |
| Transfer X to Stack Pointer    |           |             |           |             |             |          |            |            |          |               |               |          | BA       | N,Z          |
| Transfer Y to A                |           |             |           |             |             |          |            |            |          |               |               |          | B8       | N,Z          |

## Linguaggio Macchina



- Non mnemonico → difficoltà di utilizzo
- Non portabile
  - Codice sviluppato per una famiglia di CPU non utilizzabile altrove
- All'atto pratico non usato direttamente



7/21

## Assembly



- Sostituzione codice numerico con simboli mnemonici
- ~1:1

```
START: MOV  AX, BX
        CMP  AX, 12h
        JZ   EQUAL
        INT  21h
        RET
EQUAL: MOV  BL, 82h
```

8/21

## Assembly



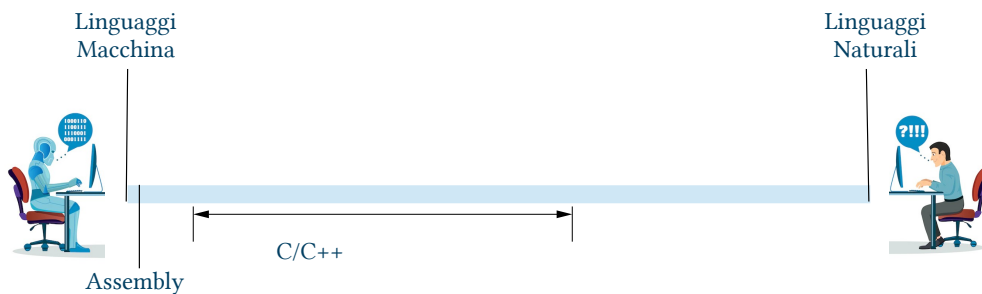
- Molto più facile del codice macchina!
  - Sia da scrivere che da leggere
- Ma devo comunque ottenere il codice macchina
- Assembler: programma che legge le istruzioni assembly e le traduce in istruzioni macchina
  - Quasi traduzione 1 a 1
- Comunque linguaggio complesso
  - Molte istruzioni anche per operazioni semplici
- Non portabile

9/21

## Linguaggi ad Alto Livello



- Non più corrispondenza 1:1 con istruzioni linguaggio macchina
  - 1 istruzione può corrispondere a decine di istruzioni linguaggio macchina



10/21



[illegible]

## Caratteristiche Linguaggi ad Alto Livello



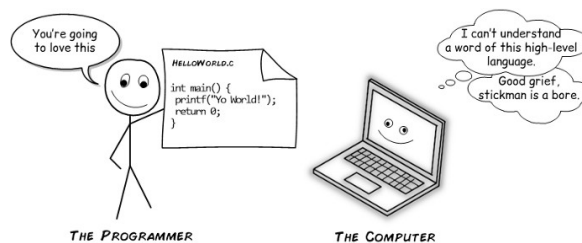
- Pro
  - Mascherano il calcolatore
  - Maggiore leggibilità e comprensibilità
  - Portabilità
- Contro
  - Necessità di traduzione

13/21

## Linguaggi interpretati e compilati



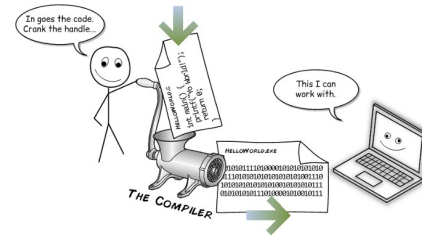
- I linguaggi ad alto livello richiedono una traduzione più complessa
  - Da codice sorgente a linguaggio macchina
- Due possibilità
  - Compilazione
  - Interpretazione



14/21

## Compiled Language

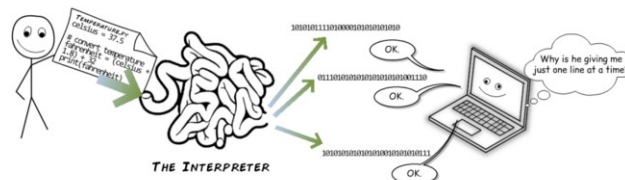
- Traduzione in codice macchina necessaria
- Compilatore → il “traduttore”
- Produce un “file eseguibile”



15/21

## Interpreted Language

- Il codice sorgente non deve essere tradotto
- Interprete che lo “legge” al volo
  - Macchina virtuale



16/21

## Compiled vs Interpreted



- Compilazione
  - Più veloce
  - Permette progetti multifile con più linguaggi
  - Posso nascondere codice sorgente
- Interpretazione
  - Esecuzione immediata
    - Scripting
  - Multiplatforma
    - Web e altro

17/21

## Perché così tanti linguaggi?



- Ricerca costante di sistemi migliori per aggiornamento tecnologia
- Non solo “General Purpose” ma problemi specifici
  - Web → Javascript, PHP, GO...
  - Mobile → JAVA
  - Firmware → C
  - Sistemi real time → C++
- Performance vs Rapidità di sviluppo
  - GO/C ↔ Javascript/Python

18/21

## Elementi del linguaggio

UNIVERSITÀ DI PARMA 

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                                                         |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Lessico<ul style="list-style-type: none"><li>– Parole chiave: istruzioni, operatori, simboli...</li></ul></li><li>• Sintassi<ul style="list-style-type: none"><li>– Come si possono combinare i precedenti elementi<ul style="list-style-type: none"><li>• Parsing</li></ul></li><li>– Si può distinguere tra<ul style="list-style-type: none"><li>• Sintassi necessaria e</li><li>• <i>Zucchero Sintattico</i></li></ul></li></ul></li></ul> | <b>Errore di<br/>Compilazione o<br/>Interpretazione</b> |
| <ul style="list-style-type: none"><li>• Semantica<ul style="list-style-type: none"><li>– Significato di quanto scritto</li></ul></li></ul>                                                                                                                                                                                                                                                                                                                                            | <b>Errore di<br/>Esecuzione</b>                         |

20/21

## Riepilogo

UNIVERSITÀ DI PARMA 

- Computer: sistema che esegue una serie di istruzioni in “sequenza” → automa
- Programmare:
  - Individuare algoritmo risolutivo del problema dato
    - Passi e loro ordine di esecuzione
  - Tradurre l’algoritmo individuato in codice
    - Lessico, Sintassi e Semantica specifici del linguaggio di programmazione

21/21

This image shows a single sheet of white paper with horizontal ruling lines. The lines are evenly spaced and run across the width of the page. There are no margins, text, or other markings on the paper.



UNIVERSITÀ DI PARMA

# Breve introduzione al C

*For 'tis the sport to have the enginer  
Hoist with his own petar*

Hamlet, Act III, Scene IV, Shakespear

## Sommario

UNIVERSITÀ  
DI PARMA 

- Perché il C?
  - Come e quando è nato il C
  - Principali caratteristiche
- Scheletro di un programma C
- Introduzione agli elementi base
  - Preprocessore
  - Variabili
  - Espressioni
  - Funzioni
  - I/O → Input/Output

# SUMMARY



2/44

## Reason Why

UNIVERSITÀ  
DI PARMA

- Perché il C?
  - Non nasconde la macchina
    - basso/medio livello
  - Efficiente
  - Consolidato
  - Utilizzato per: driver, programmi semplici, sistemi operativi, sistemi embedded
- A partire dal C sviluppati altri linguaggi e/o sintassi molto simili
  - C++, JAVA, C#, (PERL)



3/44

## Efficiency

UNIVERSITÀ  
DI PARMA

- Anyway most of developers do not have to deal with efficiency
- Time to Market is often more important

### 1 Billion nested loop iterations



4/44



## The Perils of JavaSchools



- Articolo di Joel Spolsky, 2005
  - Stackoverflow

*And thus, the ability to understand **pointers** and **recursion** is directly correlated with the ability to be a great programmer.*

*Nothing about an all-Java CS degree really weeds out the students who lack the mental agility to deal with these concepts. As an employer, I've seen that the 100% Java schools have started churning out quite a few CS graduates who are simply **not smart enough to work as programmers***



5/44

## Nascita del C



- Laboratori Bell
  - UNIX
  - DEC PDP-7 con ben 8 kByte di RAM
- Inizialmente sviluppo in Assembly
- Linguaggio ad alto livello → B derivato dal BCPL
- Dennis Ritchie sviluppa il C a partire dal B
  - Inizialmente NB (New B)
- 1973 UNIX riscritto in C

6/44

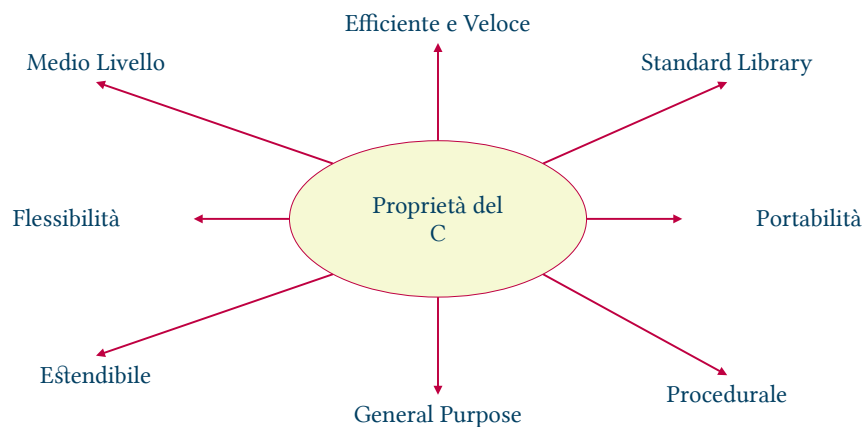
## C Standard



- Differenti standard
  - K&R (da evitare)
  - ANSI C o ISO C:
    - C89/C90 largo supporto
    - C99 diffusione recente, si sta sempre di più affermando
    - C11 aka “do not use VLA”
    - C18/17 mostly C11 bugfixes
    - C23 (2024)
    - C2Y
- Code::Blocks > C89/C90

7/44

## Caratteristiche principali



8/44

## Caratteristiche

- “Trust the Programmer” → i programmatori sono infidi!
- Codice difficile da interpretare
  - E quindi modificare
- Procedurale ↔ OOP
- No gestione eccezioni
- No namespace
- No run time checking
- No garbage collection



9/44

## “Scheletro” C

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

int main(int argc, char **argv){
    /* stampa Hello World! A schermo */
    printf("Hello World!");
    return 0;
}
```

10/44

## “Scheletro” C

UNIVERSITÀ  
DI PARMA 

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

```
int main(int argc, char
/* stampa Hello World
printf(“Hello World!
return 0;
}
```

- Tutto quanto inizia con #
- Direttiva preprocessore
- Nel caso degli “include<>” serve per inserire informazioni necessarie al compilatore
- File .h
- Librerie
- A rigore solo `stdio.h` è necessaria

11/44

## “Scheletro” C

UNIVERSITÀ  
DI PARMA 

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

```
int main(int argc, char **argv){
/* stampa Hello World! A schermo */
printf(“Hello World!”);
return 0;
}
```

- Funzione “main()”
- Punto di inizio dell’esecuzione
- Si parte da qui!
- Capiremo più avanti i singoli elementi

12/44

## “Scheletro” C

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

```
int main(int argc, char **argv){
```

```
/* stampa Hello World! A schermo */
```

```
printf("Hello World!");
```

```
return 0;
```

```
}
```

- Commento
- Tutto ciò tra /\* e \*/
  - Altri modi
- Ignorato dal compilatore
- Ma importante per chi deve capire il codice!

13/44

## “Scheletro” C

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

```
int main(int argc, char **argv){
```

```
/* stampa Hello World! A schermo */
```

```
printf("Hello World!");
```

```
return 0;
```

```
}
```

- Stampa “Hello World!”
  - Senza doppie virgolette
- printf() non è una istruzione del C ma una funzione
  - Come la main()
- La libreria del C ne fornisce diverse

14/44

## “Scheletro” C

UNIVERSITÀ  
DI PARMA

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

```
int main(int argc, char **argv){
    /* stampa Hello World! A schermo */
    printf("Hello World!");
    return 0;
}
```

- Termina la main()
- E quindi il programma
- Restituisce "0"
- A chi?
  - Al Sistema Operativo

15/44

## “Scheletro” C

UNIVERSITÀ  
DI PARMA

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

```
int main(int argc, char **argv){
    /* stampa Hello World! A schermo */
    printf("Hello World!");
    return 0;
}
```

- Syntactic Syrup
- Addolciscono la sintassi per noi
- Comunque obbligatorie

- Indentazione
- Rende il codice più leggibile
- Obbligatoria per noi (-2 punti)

16/44

## Introduzione agli elementi base del C

UNIVERSITÀ  
DI PARMA

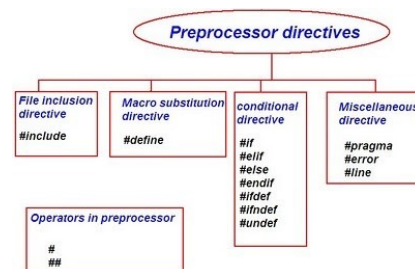
- Direttive preprocessore
- Commenti
- Variabili
- Stringhe
- Nominativi
- Funzioni
- Espressioni
- I/O

17/44

## Direttive del preprocessore

UNIVERSITÀ  
DI PARMA

- Tutte quelle che iniziano con '#'
  - Ce ne sono una dozzina
  - Ma noi ne vedremo solo alcune
- Per ora limitiamoci a:
  - `#include <???` oppure `#include "???"`
    - Permette di includere file necessari al compilatore
  - `#define <identificativo> <rimpiazzo>`
    - Per costanti e macro



18/44

## Commenti



- Spesso sottovalutati
  - Ma in sede di esame ne tengo conto
- Multiriga:
  - Tutto ciò che si trova tra /\* e \*/
- Singola riga
  - Tutto ciò che segue //
  - Nello standard dal C99

19/44

## Introduzione alle variabili



- Nel modello di von Neuman → memoria
  - Celle con indirizzo
  - Scomodo ad alto livello
- Il C può accedere alla memoria tramite le variabili
  - Sistema per memorizzare dati
  - In pratica parte di memoria a cui associo un nome
    - Non mi devo preoccupare di indirizzo o spazio occupato
    - Ci pensa il compilatore

20/44



## Introduzione alle variabili

- Differenti tipi
  - Numeriche ovvero Scalari
    - Intere, virgola mobile...
  - Composite
  - ...
- In C devo sempre indicare un tipo di dato
- Esempio di definizione di variabili:
  - `char c = 127;`
  - `int a;`
  - `float f = 159.37;`

| Memoria<br>RAM |     | indirizzo |
|----------------|-----|-----------|
| c {            | 127 | 0x0000    |
|                | ??  | 0x0001    |
| a {            | ??  | 0x0002    |
|                | ??  | 0x0003    |
|                | ??  | 0x0004    |
|                | ??  | 0x0005    |
| f {            | 195 | 0x0005    |
|                | 245 | 0x0006    |
|                | 72  | 0x0007    |
|                | 64  | 0x0008    |

21/44

## Introduzione alle stringhe

- In C particolare tipo di dato
  - Sequenza di caratteri racchiusi tra “”
  - `printf(“Hello world!”);`
  - `char c[] = “Informatica”;`
- Approfondiremo meglio
- Libreria apposita
  - `#include <string.h>`
- Stringhe di formato

22/44

## Introduzione agli identificativi

- Programmando sceglieremo dei nomi per variabili, costanti, funzioni e altri elementi
- Ci sono però regole
  - Solo lettere, cifre o “\_”
  - Deve iniziare per lettera (o per “\_”)
  - Maiuscole e Minuscole sono considerate lettere differenti
  - Non deve essere una parola riservata del linguaggio

23/44

## Introduzione agli identificativi

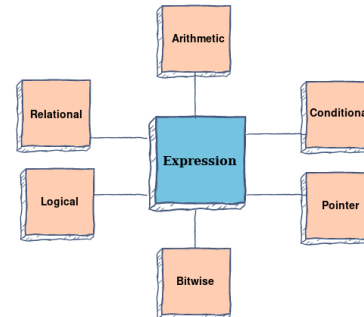
|             |               |                 |                   |
|-------------|---------------|-----------------|-------------------|
| <i>auto</i> | enum          | <i>restrict</i> | unsigned          |
| break       | extern        | return          | void              |
| case        | float         | short           | volatile          |
| char        | for           | signed          | while             |
| const       | goto          | sizeof          | <i>_Bool</i>      |
| continue    | if            | static          | <i>_Complex</i>   |
| default     | <i>inline</i> | struct          | <i>_Imaginary</i> |
| do          | int           | switch          | <i>_Packed</i>    |
| double      | long          | typedef         |                   |
| else        | register      | union           |                   |

24/44

## Introduzione alle espressioni



- Combinazione di 1 o più “operandi”
  - Variabili, costanti, chiamate di funzioni ecc.
- Esempi
  - 5
  - j (se j è una variabile)
  - printf(“ciao”)
  - 5+j\*6
    - +, -, /, \*, %



25/44

## I/O



- Input/Output
- Noi ci limiteremo inizialmente a
  - Input → Tastiera
  - Output → Video
- Due funzioni principali
  - printf()
  - scanf()

26/44

## printf()



- È una funzione
  - Complessa
- Print Formatted
  - Stampa in accordo a una “stringa di formato”
    - Testo
    - Variabili
    - Simboli
    - Ecc.

```
int printf(const char *format, ...);
```

27/44

## printf()



- Stringa di formato
- Può contenere
  - Testo → stampato così come è
  - Combinazioni di caratteri → rimpiazzate da altro quando stampo
    - Sequenze di escape
    - Specificatori di formato
      - Caratteri di controllo

28/44

## printf()



- Stampare testo “semplice”
- Racchiuso tra doppi apici “”

```
printf("Hello world!");
```

29/44

## Sequenze di “escape”



- Caratteri non ottenibili da tastiera o per uso specifico
- Tipicamente carattere preceduto da “\”

```
printf("Hello world!\n");
```

```
printf("Hello world!  
"); // error: missing terminating "  
character
```

30/44

## Sequenze di “escape”

|                   |                                                     |
|-------------------|-----------------------------------------------------|
| <code>\n</code>   | newline                                             |
| <code>\r</code>   | carriage return                                     |
| <code>\f</code>   | form feed                                           |
| <code>\a</code>   | bell                                                |
| <code>\t</code>   | horizontal tab                                      |
| <code>\v</code>   | vertical tab                                        |
| <code>\"</code>   | doppie virgolette                                   |
| <code>\nnn</code> | carattere corrispondente all'ASCII nnn (ottale)     |
| <code>\xNN</code> | carattere corrispondente all'ASCII NN (esadecimale) |
| <code>\\</code>   | backslash                                           |
| <code>%%</code>   | singolo %                                           |

31/44

## Specificatori di formato

- Indicano a `printf()` come stampare
- Cosa?
  - Valori numerici
  - Contenuto variabili
  - ...
- Preceduti da “%”

```
printf(“The content of ‘a’ is %d\n”, a);
```

32/44

## Specificatori di formato



|          |                                                  |
|----------|--------------------------------------------------|
| %d       | intero decimale                                  |
| %o %x %X | intero ottale o esadecimale                      |
| %f       | numero a virgola mobile in notazione decimale    |
| %e %E    | numero a virgola mobile in notazione scientifica |
| %g %G    | numero a virgola mobile (automatico)             |
| %c       | singolo carattere                                |
| %s       | stringa                                          |
| %p       | indirizzo di memoria (void *)                    |
| %q %L    | long long                                        |
| %[ ]     | elenco di caratteri, solo <b>scanf()</b>         |

33/44

## Caratteri di controllo



- Gli specificatori di formato indicano cosa stampare
- I caratteri di controllo permettono di fare “fine tuning”
  - Stampo comunque il segno?
  - Quanti caratteri devo usare?
  - Quante cifre dopo la virgola?
  - ...

```
printf("1 g di oro vale %f al grammo", 1770.77/31.103);
```

```
printf("1 g di oro vale %.2f al grammo", 1770.77/31.103);
```

34/44

## Caratteri di controllo



|               |                                               |
|---------------|-----------------------------------------------|
| <b>+</b>      | stampa comunque il segno                      |
| <b>spazio</b> | antepone a numeri positivi uno spazio         |
| <b>n</b>      | minima ampiezza campo (riempimento con spazi) |
| <b>0n</b>     | minima ampiezza campo (riempimento con o)     |
| <b>.n</b>     | numero di cifre decimali (minimo o fisso)     |
| <b>-</b>      | allineamento a sinistra                       |
| <b>l</b>      | long                                          |
| <b>h</b>      | short                                         |
| <b>ll</b>     | long long                                     |

35/44

## scanf()



- È una funzione
  - `int scanf(const char *format, ...);`
- Mi permette di leggere da tastiera
  - Stringa di formato → pattern matching
  - Ciò che legge → variabile (ovvero memoria)

```
int a;  
scanf("%d", &a);
```

36/44



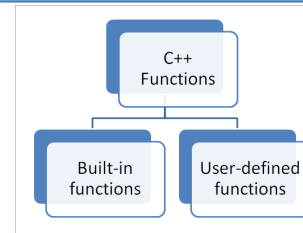
## scanf()

- Simile a printf() ma:
  - Stringa di formato tipicamente solo specificatore di formato
  - I dati coinvolti vanno preceduti da &
- Occhio agli spazi e al buffer di ingresso!

37/44

## Introduzione alle funzioni

- Anche “procedure” o “subroutine”
- Istruzioni C “troppo” semplici
- Troppe anche per fare cose banali
  - Soluzione → creo “gruppi” di istruzioni
- Concetto base di molti linguaggi ad alto livello
  - “Building blocks”
  - Programmazione procedurale

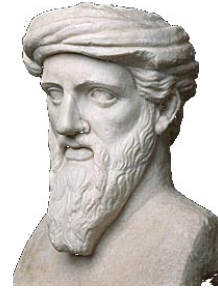


38/44

## Introduzione alle funzioni

UNIVERSITÀ  
DI PARMA

- Funzioni predefinite
  - Esempio: `printf()`
- Il C mette a disposizione una “ricca” libreria di funzioni predefinite
  - Non devo reinventare la ruota!
  - Sono già ottimizzate
  - Funzionano
  - Portabili



```
double hypot(double x, double y);
```

39/44

## Introduzione alle funzioni

UNIVERSITÀ  
DI PARMA

- Funzioni definite dal programmatore
- Fattorizzazione
  - le definisco → 1 volta
  - le uso → quante volte voglio
- Miglior leggibilità
- Miglior facilità di organizzazione e debugging

40/44

## Compilazione

UNIVERSITÀ DI PARMA

- Sorgente → Codice Macchina
- Noi useremo “ambiente integrato”
  - Nasconde il processo
- In realtà differenti “attori”
  - Preprocessore
  - Compilatore
  - Linker

```
graph TD; A[Source File  
.c .h] --> B[Preprocessor]; B --> C[Compiler]; C --> D[Object File  
.o .obj]; D --> E[Linker]; E --> F[Executable  
.com .exe <nn>];
```

41/44

## Fasi di compilazione

UNIVERSITÀ DI PARMA

- Preprocessore
  - Mera sostituzione direttive che iniziano con ‘#’
  - Spesso integrato con compilatore
- Compilatore
  - Traduce il sorgente ottenendo codice macchina
    - Operazione complessa
  - Ottimizzazione
  - Supporto differenti architetture

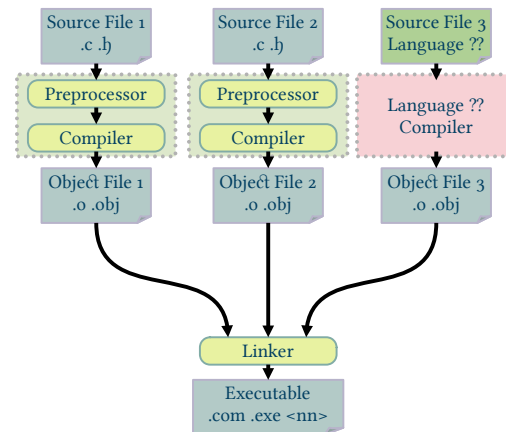
```
graph TD; A[Source File  
.c .h] --> B[Preprocessor]; B --> C[Compiler]; C --> D[Object File  
.o .obj]; D --> E[Linker]; E --> F[Executable  
.com .exe <nn>];
```

42/44

## Fasi di compilazione

UNIVERSITÀ  
DI PARMA

- Linker
  - Riunisce tutti i file oggetto
  - Risoluzione indirizzi memoria
  - Necessario per progetti multifile
  - Può gestire sistemi con più linguaggi
    - Tutti “compilati”



43/44



UNIVERSITÀ DI PARMA

## Breve introduzione al C



*For 'tis the sport to have the engineer  
Hoist with his own petar*

Hamlet, Act III, Scene IV, Shakespear



UNIVERSITÀ DI PARMA

# Data Representation

*τῶ ἀριθμῶ δὲ τὰ πάντ' ἐπέοικεν*  
*Numero Cuncta Assimilantur*

Pitagora (unc.), V sec. BC.

## Summary

UNIVERSITÀ  
DI PARMA

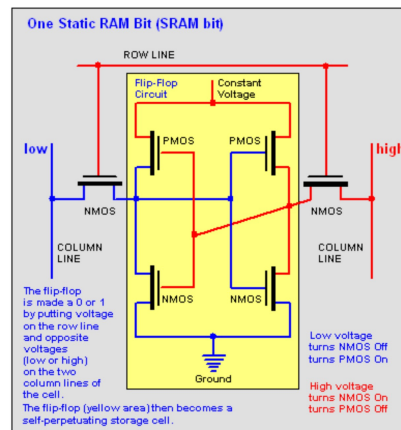
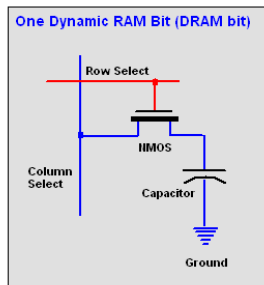
- Memory technologies
- Numbers
  - Representation and Value
  - Decimal Numeral System
  - Binary Numeral System
  - Hexadecimal Numeral System
  - Negative Numbers
  - Non integer Numbers
- Simbols
  - ASCII
  - UNICODE

# SUMMARY



2/49

## Memory

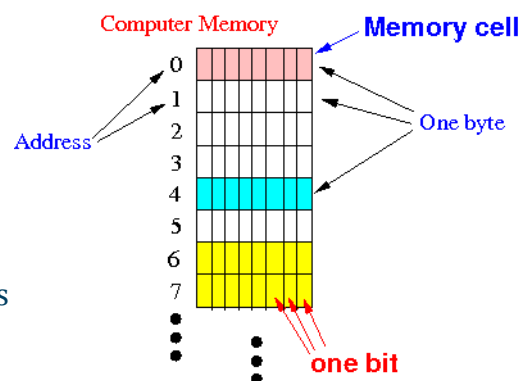
UNIVERSITÀ  
DI PARMA

3/49

## Memoria

UNIVERSITÀ  
DI PARMA

- Data & Code → memory
- How are they stored?
- We have
  - Bytes composed by
  - Bits
- Each Byte has a specific address



4/49

## Memory

- The most basic unit of memory is the **bit**
  - “Binary digiT”
- 2 possible states, formally 0 or 1
- Everything that is stored in computer memory **MUST** be a combination of 2 symbols (0 & 1)
  - True also for other kind of memory supports
- **Then we need to “write” numbers (or symbols) using only sequences of 0 or 1**

**THE PHONETIC ALPHABET  
MORSE CODE GUIDE**

|           |      |            |      |
|-----------|------|------------|------|
| A ALPHA   | --   | N NOVEMBER | --   |
| B BRAVO   | ---- | O OSCAR    | ---- |
| C CHARLIE | ---- | P PAPA     | ---- |
| D DELTA   | ---  | Q QUEBEC   | ---- |
| E ECHO    | .    | R ROMEO    | ---  |
| F FOXTROT | ---- | S SIERRA   | ---  |
| G GOLF    | ---  | T TANGO    | -    |
| H HOTEL   | ---- | U UNIFORM  | ---  |
| I INDIA   | ..   | V VICTOR   | ---- |
| J JULIET  | ---- | W WHISKEY  | ---- |
| K KILO    | --   | X X-RAY    | ---- |
| L LIMA    | ---- | Y YANKEE   | ---- |
| M MIKE    | --   | Z ZULU     | ---- |

5/49



UNIVERSITÀ DI PARMA

## Numbers

6

## Numbers and their Representations

UNIVERSITÀ  
DI PARMA

7/49

## Numbers and their Representations

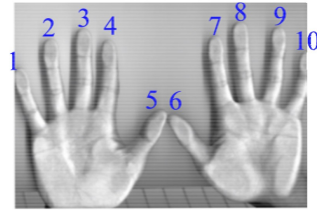
UNIVERSITÀ  
DI PARMA

8/49



## Decimal numeral system

- Based on 10 symbols (**base 10**):
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- Positional system
  - Position is used to weight symbol importance
    - 1786 contains the same symbols as 8761
- Not absolute rules for numeral systems
  - MMXXII
  - Quatre-vingt



9/49

## Positional Numeral System

- Somewhere in a distant past someone started to mark symbols for numbers
  - But there are so much numbers...
- Let's start to count:
  - 0 1 2 3 4 5 6 7 8 ... and so?
  - We add a second "digit" with a different meaning
  - 30

10/49

## Decimal numeral system



- We know how to “represent” numbers
- How we **compute** the number given its representation in the decimal numeral system?
- Given the digits  $d$ :
  - $d \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- A number composed by  $N$  digits can be **written** as:
  - “ $d_{n-1}d_{n-2}\dots d_2d_1d_0$ ”
  - i.e. 1786
- Its value  $V$  can be then **computed** as:
  - $V = d_{n-1} \times 10^{n-1} + d_{n-2} \times 10^{n-2} + \dots + d_2 \times 10^2 + d_1 \times 10 + d_0$
  - Namely as a sum of powers of 10
  - i.e.  $1786 = 1 \times 1000 + 7 \times 100 + 8 \times 10 + 6$

11/49

## Binary numeral system



- In memory we can have 2 states only  $\rightarrow$  formally 1 & 0 “digits”
- We must use base 2 numeral system  $\rightarrow$  binary numeral system
- Digits:
  - $b \in \{0, 1\}$
- Representation:
  - $b_{n-1}b_{n-2}\dots b_2b_1b_0$
- Value:
  - $V = b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \dots + b_2 \times 2^2 + b_1 \times 2 + b_0$

12/49

## Example



- What is the value of the binary number written as 101110?
- Sum of powers of two:
  - $V = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2 + 0$
  - $V = 32 + 8 + 4 + 2$

|       |       |       |       |       |       |       |       |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 0     | 0     | 1     | 0     | 1     | 1     | 1     | 0     |
| $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
| 128   | 64    | 32    | 16    | 8     | 4     | 2     | 1     |

13/49

## Another approach



- 1) Start from the left namely from the most significant digit
- 2) Init N as 0
- 3) Read current digit
- 4) Multiply N by 2 and sum the digit value
- 5) There are other digits? If so, move to next digit on the right and repeat from step #3
- 6) End of procedure, N contains the value!

14/49

## How to compute binary representation



- 1) Consider N as the number we want to know the binary representation
- 2) Init S as an empty sequence of characters
- 3) Divide N by 2
- 4) Put the remainder (0 or 1) in S from right side
- 5) Result is 0?

Yes: S contains the binary representation of value N

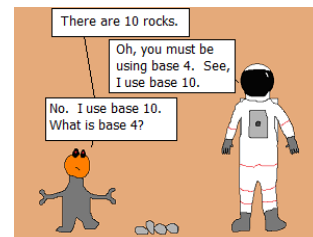
No: put  $N = \text{remainder}$  and repeat from #3

15/49

## Other systems



- Other numeral systems often used are:
  - Hexadecimal representation, base 16:
    - Digits are: 0-9 & A-F
    - 1 digit  $\rightarrow$  4 bits
    - i.e. `0x90FF`
  - Octal representation, base 8:
    - Digits are: 0-7
    - 1 digit  $\rightarrow$  3 bits
- Same conversion rules as before



Every base is base 10.

16/49

## From Hexadecimal to Binary

- From Hexadecimal to Binary:
  - Take bits 4 by 4 (from the least significant ones)
  - Substitute them with the corresponding Hex digit
  - i.e.  $1011\ 1001_2 = 0xB9_{16}$
- Practically we can use a “Lookup Table”

17/49

## Lookup Table

| Decimal | Binary | Hexadecimal |
|---------|--------|-------------|
| 0       | 0000   | 0           |
| 1       | 0001   | 1           |
| 2       | 0010   | 2           |
| 3       | 0011   | 3           |
| 4       | 0100   | 4           |
| 5       | 0101   | 5           |
| 6       | 0110   | 6           |
| 7       | 0111   | 7           |
| 8       | 1000   | 8           |
| 9       | 1001   | 9           |
| 10      | 1010   | A           |
| 11      | 1011   | B           |
| 12      | 1100   | C           |
| 13      | 1101   | D           |
| 14      | 1110   | E           |
| 15      | 1111   | F           |

18/49

## Hexadecimal Numeral System

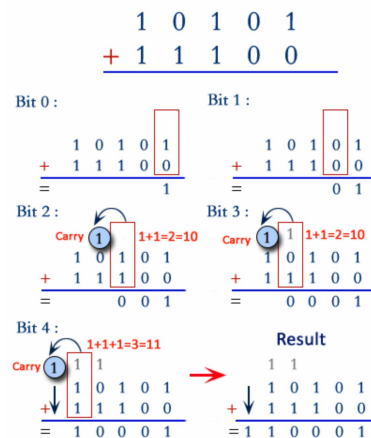
- Practical use when number that are power of 2 are important
- Like binary system but definitely shorter
  - $11011111010 \rightarrow 0x6FA \rightarrow 1786$
  - $16 = 2^4$
- Memory addresses
- Raw byte values
- RGB colors
  - 3 bytes
  - $0xA13A77$

|         |      |      |      |      |      |      |      |      |
|---------|------|------|------|------|------|------|------|------|
| 0000000 | 5025 | 4644 | 312d | 342e | 250a | ecc7 | a28f | 250a |
| 0000010 | 4925 | 766e | 636f | 7461 | 6f69 | 3a6e | 6720 | 2073 |
| 0000020 | 712d | 2d20 | 5364 | 4641 | 5245 | 2d20 | 4e64 | 504f |
| 0000030 | 5541 | 4553 | 2d20 | 4264 | 5441 | 4843 | 2d20 | 5064 |
| 0000040 | 4644 | 2041 | 642d | 4f4e | 554f | 4554 | 5352 | 5641 |
| 0000050 | 2045 | 732d | 7250 | 636f | 7365 | 4373 | 6c6f | 726f |
| 0000060 | 6f4d | 6564 | 3d6c | 6544 | 6976 | 6563 | 4752 | 2042 |
| 0000070 | 732d | 6f43 | 6f6c | 4372 | 6e6f | 6576 | 7372 | 6f69 |

19/49

## Base X math

- Same rules as decimal system!
- This is true for each positional numeral system for each base


 © w3resource.com  
20/49

## Memory organization constraints



- We have bits
- Usually they are not addressed as is
- They are then grouped in **words**
  - Typically multiples of 8 (**bytes**)
- Therefore:
  - We have to use  $n \times 8$  digits
  - Moreover we have memory limits

**word** *noun*: the natural unit of data used by a particular architecture design

21/49

## Properties



- A  $n$  bit word can represent all integer numbers  $N$  such that:
  - $0 \leq N \leq 2^n - 1$
- Given a number  $N$  we will need, at least,  $n$  bit to represent it in binary format such as:
  - $n > \log_2 N$

22/49

## Examples



- Practical examples:
  - 1 byte namely 8 bits allows to store numbers in the  $[0, 255]$  interval
  - 256 different values
- Decimal number  $17899_{10}$  requires, at least, 15 binary digits to be stored ( $\log_2(17899)=14,12759..$ )
  - Probably we have to use anyway 16 bits namely 2 bytes
  - $17899_{10} = 0100010111101011_2 = 45EB_{16}$

23/49

## Natural numbers are enough?



- No, they are not enough
- We have also to deal with:
  - Negative numbers
  - Non integer numbers



24/49



## Numbers with sign



- Ideally:
  - Same representation for positive numbers
  - Alternative representation for negative numbers
- Moreover, simple:
  - Usage
  - Implementation
- N bit  $\rightarrow 2^N$  different permutations
  - $2^N$  possible numbers

25/49

## Sign - Magnitude

- One bit is used as sign
  - $0 \rightarrow +$
  - $1 \rightarrow -$
- Easy to understand (for us!)
- Not so easy to implement
  - That bit has to be separately managed
- Complexity for
  - SW
  - HW

| Binario | Decimale |
|---------|----------|
| 0 000   | 0        |
| 0 001   | 1        |
| 0 010   | 2        |
| 0 011   | 3        |
| 0 100   | 4        |
| 0 101   | 5        |
| 0 110   | 6        |
| 0 111   | 7        |
| 1 000   | -0       |
| 1 001   | -1       |
| 1 010   | -2       |
| 1 011   | -3       |
| 1 100   | -4       |
| 1 101   | -5       |
| 1 110   | -6       |
| 1 111   | -7       |



27/49

## Ones' complement

- No special bits
- Positive numbers as usual
- Negative numbers → complement
  - 1 → 0 & 0 → 1
- 8 bit → 255 different values
  - [-127,127]
  - 2 representations for 0

| Binario | Decimale |
|---------|----------|
| 0000    | 0        |
| 0001    | 1        |
| 0010    | 2        |
| 0011    | 3        |
| 0100    | 4        |
| 0101    | 5        |
| 0110    | 6        |
| 0111    | 7        |
| 1000    | -7       |
| 1001    | -6       |
| 1010    | -5       |
| 1011    | -4       |
| 1100    | -3       |
| 1101    | -2       |
| 1110    | -1       |
| 1111    | -0       |

28/49

UNIVERSITÀ  
DI PARMA

---

## Two's Complement

- Used in “modern” architectures
- Positive numbers
  - As usual
- To obtain negative numbers:
  - Take positive
  - Compute ones' complement
  - Add 1
- To obtain negative version from positive number:
  - Sane trick!

| Binario | Decimale |
|---------|----------|
| 0000    | 0        |
| 0001    | 1        |
| 0010    | 2        |
| 0011    | 3        |
| 0100    | 4        |
| 0101    | 5        |
| 0110    | 6        |
| 0111    | 7        |
| 1000    | -8       |
| 1001    | -7       |
| 1010    | -6       |
| 1011    | -5       |
| 1100    | -4       |
| 1101    | -3       |
| 1110    | -2       |
| 1111    | -1       |

29/49

UNIVERSITÀ  
DI PARMA

## Two's complement



- Range of values for N bits:

- $[-(2^{N-1}), 2^{N-1}-1]$
- For 8 bits  $\rightarrow [-128, 127]$

$$\begin{array}{r} 00001001 + \\ 11111011 = \\ \hline 10000100 \end{array}$$

- Advantages

- We do not have +0 e -0
- Better usage of different permutations
- Most significant bit is anyway the sign bit even if it is not the sign bit
- **Same math operations!**

30/49

## 2's complement



- Easy conversion trick

- Start from least significant bit
- Do not change 0s until you find a 1
- Do not change the first 1 you find
- Starting from the first 1, complement all other digits!

$$0110111000 \rightarrow 1001001000$$

31/49

## Endiannes

 UNIVERSITÀ  
DI PARMA

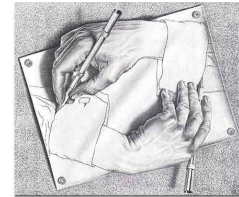

- How do we write in a sheet of paper?
- The same can apply to computer memory
  - At byte level
- When a number span across multiple bytes, what we start to write in memory?
  - Least significant byte?
  - Most significant byte?

Si non nito studiare non transiet!

如果我不學習，我就不會通過考試！

אם לא אלמד אני לא אעבור את הבחינה!

إذا لم أدرس فلن أنجح في الامتحان!

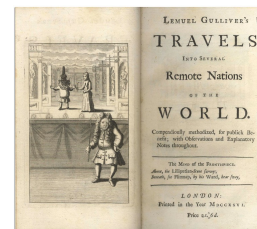
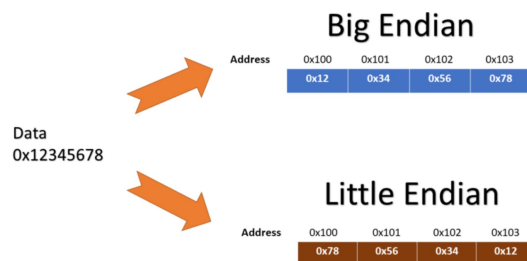


32/49

## Endiannes

 UNIVERSITÀ  
DI PARMA


- Big endian → More significant bytes corresponds to lower memory addresses
- Little endian → More significant bytes corresponds to higher memory addresses



33/49

## Endiannes

- Should we care about endianness?
- Yes and **no**
  - In a standalone system we often do not care
  - Issues:
    - Accessing memory on a byte basis
    - Transferring data through a network
    - Using files across different architectures

34/49

## Natural numbers are enough?

- No, they are not enough
- We have also to deal with:
  - Negative numbers
  - Non integer numbers



35/49

## Non integers numbers



- How they are managed in the decimal numeral system?
  - Example: 123.456
  - The value can be computed as  $1 \times 100 + 2 \times 10 + 3 + 4/10 + 5/100 + 6/1000$
  - Decimal numbers can be seen as multiplied by powers of 10  $V(123,456) = 1 \times 10^2 + 2 \times 10 + 3 + 4 \times 10^{-1} + 5 \times 10^{-2} + 6 \times 10^{-3}$
- Easy extension to the binary numeral system:
  - Example: 1011.1011
  - $V(1011.1011) = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2 + 1 + 1 \times \frac{1}{2} + 0 \times (\frac{1}{2})^2 + 1 \times (\frac{1}{2})^3 + 1 \times (\frac{1}{2})^4$
  - $V(1011.1011) = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2 + 1 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 1 \times 2^{-4}$

36/49

## Non integers numbers



- A number that has a finite number of digits in decimal format does not necessarily have the same property in binary format.
  - Example:  $0.3 \rightarrow 0.01\overline{0011}$
  - And vice versa
- Managing non integer numbers using a computer can lead to **approximations!**
  - i.e. add 0.1 to 0.1 ten times

37/49

## Fixed point binary numbers



- Also when storing non integer numbers we have to use a limited number of bytes (usually 2, 4, or 8)
- Fixed point binary numbers
  - Given  $N$  bits, I use  $M$  bits for the fractional part and  $N-M$  for the integer portion
- Fixed point data type are seldom used
  - Embedded systems
  - Databases
  - Fonts

38/49

## Floating point numbers



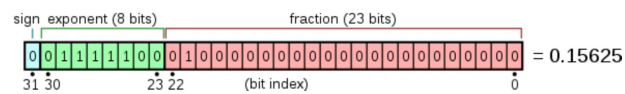
- In the decimal numeral system a convenient way to represent numbers is the “Normalized Scientific Notation”
  - $-1234,567 \rightarrow -1.234567 \times 10^3$
- A general format can be seen as:
- (sign) (integer part).(fractional part)  $\times 10^{(\text{exponent})}$ 
  - $0 < (\text{integer part}) < 10$
- Easy to extend to binary system

39/49

## Floating point numbers



- IEEE 754 (Standard for Binary Floating-Point Arithmetic)
- From 16 to 128 bits depending on the precision we want to obtain
  - Integer part is always 1 (not necessary to store)
  - Sign bit S
  - Exponent E
  - Fractional part M
  - $N = (-1)^s (1.M) 2^E$
- E & M can have specific values (0,255) to manage special situations (NaN,  $\pm\infty$ , underflow)

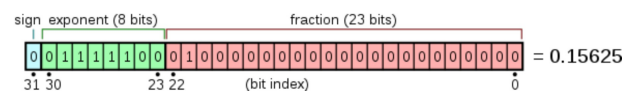


40/49

## Floating point numbers



- How to decipher that?
  - Sign bit  $\rightarrow 0 \rightarrow$  positive number
  - Exponent  $\rightarrow 124$ 
    - We subtract a *bias* (127)
    - $\rightarrow -3$
  - Fractional part  $\rightarrow 0.25$  ovvero  $(\frac{1}{2})^2$
  - Integer part  $\rightarrow 1$
- $-1^0 \times 1.25 \times 2^{-3} \rightarrow 0.15625$

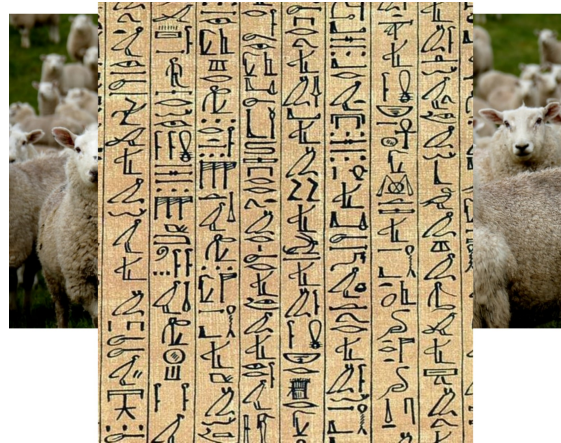


41/49



## Natural numbers are enough?

- No, they are not enough
- We have also to deal with:
  - Negative numbers
  - Non integer numbers
  - Symbols!



42/49

## Simbols

- How to store symbols?
  - We already understood how to store numbers
- We can associate a number to each symbol!
  - Lookup table: Symbols  $\rightarrow$  Integer Number
- Some standards
  - ASCII  $\rightarrow$  American Standard Code for Information Interchange
  - EBCDIC  $\rightarrow$  Extended Binary Coded Decimal Interchange Code
  - UTF-x  $\rightarrow$  Unicode Transformation Format

43/49

## ASCII (1963)



- 7 bit
  - Parity bit for transmission reliability (telegraph)
  - Number range [0, 127]
- 0-31 & 127 → Control symbols
- 32-126 → Printable symbols
- 128-255 → Extended ASCII
  - Mainly for other languages but mostly latin symbols

44/49

## ASCII



- In C we are going to have a specific type and syntax to deal with ASCII symbols
  - char → i.e. char a;
  - A symbol inside single quotation marks → i.e. '@'
  - Symbols inside double quotation marks → "Hello world!"
- Some issue → i.e. newline
  - Linux → associated to 10
  - Commodores, MAC OS, Apple II → associated to 13
  - Windows → associated to 10 followed by a 13

45/49

| ASCII control characters |      |                       | ASCII printable characters |       |    |   | Extended ASCII characters |   |     |   |     |   |     |   |     |      |
|--------------------------|------|-----------------------|----------------------------|-------|----|---|---------------------------|---|-----|---|-----|---|-----|---|-----|------|
| 00                       | NULL | (Null character)      | 32                         | space | 64 | @ | 96                        | ` | 128 | Ç | 160 | Á | 192 | À | 224 | Ó    |
| 01                       | SOH  | (Start of Header)     | 33                         | !     | 65 | A | 97                        | a | 129 | È | 161 | Â | 193 | Á | 225 | Ô    |
| 02                       | STX  | (Start of Text)       | 34                         | "     | 66 | B | 98                        | b | 130 | É | 162 | Ã | 194 | Â | 226 | Õ    |
| 03                       | ETX  | (End of Text)         | 35                         | #     | 67 | C | 99                        | c | 131 | Ê | 163 | Ä | 195 | Ã | 227 | Ö    |
| 04                       | EOT  | (End of Trans.)       | 36                         | \$    | 68 | D | 100                       | d | 132 | Ë | 164 | Å | 196 | Ä | 228 | Ø    |
| 05                       | ENQ  | (Enquiry)             | 37                         | %     | 69 | E | 101                       | e | 133 | Ì | 165 | Æ | 197 | Å | 229 | Ù    |
| 06                       | ACK  | (Acknowledgement)     | 38                         | &     | 70 | F | 102                       | f | 134 | Í | 166 | Ç | 198 | Ä | 230 | Ú    |
| 07                       | BEL  | (Bell)                | 39                         | '     | 71 | G | 103                       | g | 135 | Î | 167 | È | 199 | Å | 231 | Û    |
| 08                       | BS   | (Backspace)           | 40                         | (     | 72 | H | 104                       | h | 136 | Ï | 168 | É | 200 | Æ | 232 | Ü    |
| 09                       | HT   | (Horizontal Tab)      | 41                         | )     | 73 | I | 105                       | i | 137 | Ð | 169 | Ê | 201 | Ç | 233 | Ý    |
| 10                       | LF   | (Line feed)           | 42                         | *     | 74 | J | 106                       | j | 138 | Ñ | 170 | Ë | 202 | È | 234 | ÿ    |
| 11                       | VT   | (Vertical Tab)        | 43                         | +     | 75 | K | 107                       | k | 139 | Ò | 171 | Ì | 203 | É | 235 |      |
| 12                       | FF   | (Form feed)           | 44                         | ,     | 76 | L | 108                       | l | 140 | Ó | 172 | Í | 204 | Ê | 236 |      |
| 13                       | CR   | (Carriage return)     | 45                         | -     | 77 | M | 109                       | m | 141 | Ô | 173 | Î | 205 | Ë | 237 |      |
| 14                       | SO   | (Shift Out)           | 46                         | .     | 78 | N | 110                       | n | 142 | Õ | 174 | Ï | 206 | Ì | 238 |      |
| 15                       | SI   | (Shift In)            | 47                         | /     | 79 | O | 111                       | o | 143 | Ö | 175 | Ð | 207 | Í | 239 |      |
| 16                       | DLE  | (Data link escape)    | 48                         | 0     | 80 | P | 112                       | p | 144 | × | 176 | Ñ | 208 | Î | 240 | =    |
| 17                       | DC1  | (Device control 1)    | 49                         | 1     | 81 | Q | 113                       | q | 145 | ä | 177 | Ò | 209 | Ï | 241 | ≡    |
| 18                       | DC2  | (Device control 2)    | 50                         | 2     | 82 | R | 114                       | r | 146 | æ | 178 | Ó | 210 | Ð | 242 | ≡    |
| 19                       | DC3  | (Device control 3)    | 51                         | 3     | 83 | S | 115                       | s | 147 | ø | 179 | Ô | 211 | Ñ | 243 | ≡    |
| 20                       | DC4  | (Device control 4)    | 52                         | 4     | 84 | T | 116                       | t | 148 | ÷ | 180 | Õ | 212 | Ò | 244 | ≡    |
| 21                       | NAK  | (Negative acknowl.)   | 53                         | 5     | 85 | U | 117                       | u | 149 | ø | 181 | Ä | 213 | Ó | 245 | ≡    |
| 22                       | SYN  | (Synchronous idle)    | 54                         | 6     | 86 | V | 118                       | v | 150 | ù | 182 | Å | 214 | Ì | 246 | ÷    |
| 23                       | ETB  | (End of trans. block) | 55                         | 7     | 87 | W | 119                       | w | 151 | ú | 183 | Æ | 215 | Í | 247 | ÷    |
| 24                       | CAN  | (Cancel)              | 56                         | 8     | 88 | X | 120                       | x | 152 | ý | 184 | Ç | 216 | Î | 248 | ÷    |
| 25                       | EM   | (End of medium)       | 57                         | 9     | 89 | Y | 121                       | y | 153 | Ö | 185 | È | 217 | Ï | 249 | ÷    |
| 26                       | SUB  | (Substitute)          | 58                         | :     | 90 | Z | 122                       | z | 154 | U | 186 | É | 218 | Ð | 250 | ÷    |
| 27                       | ESC  | (Escape)              | 59                         | ;     | 91 | [ | 123                       | { | 155 | ø | 187 | Ê | 219 | Ñ | 251 | ÷    |
| 28                       | FS   | (File separator)      | 60                         | <     | 92 | \ | 124                       |   | 156 | é | 188 | Ë | 220 | Ò | 252 | ÷    |
| 29                       | GS   | (Group separator)     | 61                         | =     | 93 | ] | 125                       | } | 157 | ø | 189 | Ç | 221 | Ó | 253 | ÷    |
| 30                       | RS   | (Record separator)    | 62                         | >     | 94 | ^ | 126                       | ~ | 158 | x | 190 | Y | 222 | Ì | 254 | ÷    |
| 31                       | US   | (Unit separator)      | 63                         | ?     | 95 | _ |                           |   | 159 | f | 191 | γ | 223 | ■ | 255 | nbsp |
| 127                      | DEL  | (Delete)              |                            |       |    |   |                           |   |     |   |     |   |     |   |     |      |

Laboratory PCS are equipped with italian keyboard

How to obtain necessary symbols?

ALT + 3 digits ASCII code

I.e. ALT+123 → “{“

Laboratory PCS are equipped with italian keyboard

How to obtain necessary symbols?

ALT + 3 digits ASCII code

I.e. ALT+123 → "{'

## Unicode

- ASCII is a bit limited: single byte → 256 values
- Unicode:
  - Main objective is to enable a single standard to deal with all characters from all scripts
  - Able to deal with 1.114.112 different symbols!
  - Different encodings → UTF-x

## Unicode

UNIVERSITÀ  
DI PARMA

| Name                    | UTF-8  | UTF-16  | UTF-32  |
|-------------------------|--------|---------|---------|
| Code unit size          | 8 bits | 16 bits | 32 bits |
| Fewest bytes per symbol | 1      | 2       | 4       |
| Most bytes per symbol   | 4      | 4       | 4       |

48/49

## UTF-8

UNIVERSITÀ  
DI PARMA

- Most popular unicode encoding
- Variable length
  - Space vs speed efficiency
  - 1 byte to map standard English letters and symbols
    - Backward compatible with 7 bits ASCII
  - 2 bytes for additional Latin and Middle eastern symbols
  - 3 bytes for Asian symbols
  - 4 bytes for any other symbol

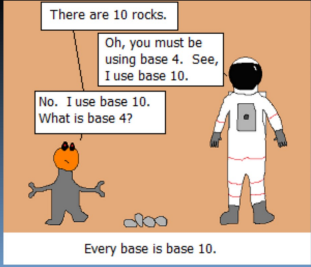
49/49

  
UNIVERSITÀ DI PARMA

## Data Representation



KEEP  
CALM  
IT'S  
QUESTION  
TIME



There are 10 rocks.

Oh, you must be using base 4. See, I use base 10.

No. I use base 10. What is base 4?

Every base is base 10.



UNIVERSITÀ DI PARMA

# Le Espressioni

*We must either institute conventional forms  
of expression or else pretend that we have  
nothing to express.*

*George Santayana, Soliloquies in England*

## Sommario

UNIVERSITÀ  
DI PARMA

- Operatori & Operandi
  - Tipologie
- Espressioni
- Valutazione
- Precedenza & Associatività
- Effetti collaterali

# SUMMARY



2/22

## Expressions vs Statement



- Piccolo preambolo
  - Statement → “Fai qualcosa” ovvero azione
  - Expression → “Valuta qualcosa” ovvero “calcolo”
    - Alcune espressioni sono anche statement → =
- Le espressioni sono combinazioni di
  - Operatori
  - Operandi

$$A = B * C + D * 17 / K$$

3/22

## Operatori



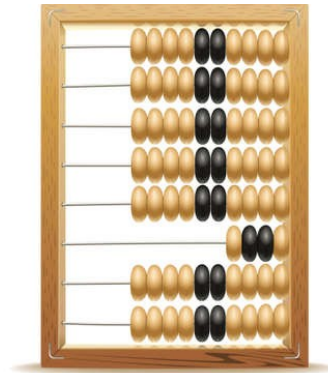
- Combinano uno o più operandi e permettono di valutare “operazioni”
  - Aritmetiche
  - Relazionali
  - Logiche
  - Bit a bit
  - Assegnamento
  - Condizionali e speciali
- A seconda del numero di operandi coinvolti:
  - Unari
  - Binari
  - Ternari

4/22

## Operatori aritmetici

UNIVERSITÀ  
DI PARMA

- Binari:
  - $+$  → somma
  - $-$  → sottrazione
  - $*$  → moltiplicazione
  - $/$  → divisione
  - $\%$  → resto della divisione intera (modulo)
- Unari:
  - $+0-$  → segno
  - $++$  → incremento di 1
  - $--$  → decremento di 1

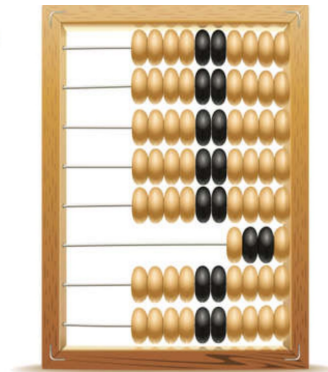


5/22

## Operatori aritmetici

UNIVERSITÀ  
DI PARMA

- Permettono calcoli tra numeri interi o a virgola mobile
  - Eccetto  $\%$
- Attenzione a /
  - Se operandi interi
    - Divisione intera
      - $5/6 \rightarrow 0$
    - Se operando negativo problemi arrotondamento
      - $-9/7 \rightarrow -1$  oppure  $-2$



6/22



## ++ & --



- Operatori di incremento o decremento
  - Incremento variabile "a" di 1  $\rightarrow ++a$ ;
  - Decremento variabile "a" di 1  $\rightarrow --a$ ;
- Operazioni comuni
- **Caveat:** comportamento differente se prefisso o postfixo
  - $++a \neq a++$
  - Incrementano entrambi "a" ma se in espressioni:
    - Prefisso  $\rightarrow$  eseguo incremento prima di altre operazioni
    - Postfisso  $\rightarrow$  eseguo incremento dopo tutte le altre operazioni

7/22

## Operatori di assegnamento



- Permettono assegnamento valore a variabile
- Semplice:
  - $=$   $\rightarrow$  assegnamento
- Composti:
  - $+=$   $\rightarrow$  assegnamento con somma
  - $-=$   $\rightarrow$  ...
  - $*=$
  - $\%=$
  - ...

8/22

## Operatori di assegnamento



- A differenza di altri operatori, gli assegnamenti hanno limiti su cosa posso avere come operandi
- A destra dell'operatore di assegnamento
  - Numeri, variabili, espressioni ecc.
  - Right Value o R-Value
- A sinistra dell'operatore di assegnamento
  - Solo variabili o comunque indirizzo memoria
  - Left Value o L-Value

9/22

## Operatori Relazionali



- A volte necessità di valutare situazioni
  - Il numero  $x$  è pari o dispari?
  - $a$  è più piccolo o più grande di  $b$ ?
- Vero o Falso, in C:
  - Vero  $\rightarrow$  qualunque numero diverso da 0
  - Falso  $\rightarrow 0$

10/22

## Operatori Relazionali

- `==` → controllo uguaglianza (no virgola mobile!)
- `!=` → controllo differenza (non esiste `≠`)
- `>` → più grande di
- `<` → più piccolo di
- `>=` → più grande o eguale di
- `<=` → più piccolo o eguale di
- Hanno come risultato:
  - Vero → 1 (diverso da 0)
  - Falso → 0

11/22

## Operatori Relazionali

| Name                   | Operator           | Expression                                                   | Evaluates        |
|------------------------|--------------------|--------------------------------------------------------------|------------------|
| Equals                 | <code>==</code>    | <code>5 == 5</code><br><code>K == 10</code>                  | true<br>depends  |
| Not Equals             | <code>!=</code>    | <code>50 != 25</code><br><code>100 != 100</code>             | true<br>false    |
| Less than              | <code>&lt;</code>  | <code>100 &lt; 200</code><br><code>P &lt; Q</code>           | true<br>depends  |
| Greater than           | <code>&gt;</code>  | <code>100 &gt; 200</code><br><code>P &gt; Q</code>           | false<br>depends |
| Less than or equals    | <code>&lt;=</code> | <code>25 &lt;= 26</code><br><code>25 &lt;= 25</code>         | true<br>true     |
| Greater than or equals | <code>&gt;=</code> | <code>1000 &gt;= 1000</code><br><code>K &gt;= (P + Q)</code> | true<br>depends  |

12/22

## Operatori logici



- Operatori relazionali → test “semplici”
  - a è compreso nell’intervallo 10-18?
  - a>10 ma anche a<18!
- Gli operatori logici permettono di costruire test più complessi
- Combinando:
  - Espressioni che coinvolgono operatori relazionali
  - Espressioni che assumono valori vero/falso

13/22

## Operatori logici



- Operatori il cui risultato è vero(1)/falso(0)
- `&&` → and logico
- `||` → or logico
- `!` → not logico

| &&    | !alive                                                                                | alive                                                                                 |
|-------|---------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| !dead |  |  |
| dead  |  |  |

14/22

## Operatori logici: tabelle di verità

| a | b | a && b | a    b | !a |
|---|---|--------|--------|----|
| 0 | 0 | 0      | 0      | 1  |
| 0 | 1 | 0      | 1      | 1  |
| 1 | 0 | 0      | 1      | 0  |
| 1 | 1 | 1      | 1      | 0  |

15/22

## Espressioni

- Combinazioni di operatori e operandi
- Esempi:
  - 17 → espressione senza operandi
  - $17 * 2 + 3$  → espressione costante
  - $a = b * 17 + c / 4 + e \% 5 * 2$  → calcolata in esecuzione

16/22

## Precedenza e associatività



- Operatori differenti possono avere precedenze differenti
  - $3*4+2$
  - Nella valutazione delle espressioni si parte dagli operatori con maggiore precedenza
- A parità di precedenza gli operatori vengono valutati in base alla associatività
  - $a\%b\%c$  ( $\rightarrow$ )
  - $a = b = c;$  ( $\leftarrow$ )
- Nel dubbio ()

17/22

| Operator                                          | Description                                                                                                                                                                                     | Associativity |
|---------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|
| ()<br>[]<br>.<br>-><br>++ --                      | Parentheses or function call<br>Brackets or array subscript<br>Dot or Member selection operator<br>Arrow operator<br>Postfix increment/decrement                                                | left to right |
| ++ --<br>+ -<br>! ~<br>(type)<br>*<br>&<br>sizeof | Prefix increment/decrement<br>Unary plus and minus<br>not operator and bitwise complement<br>type cast<br>Indirection or dereference operator<br>Address of operator<br>Determine size in bytes | right to left |
| * / %                                             | Multiplication, division and modulus                                                                                                                                                            | left to right |
| + -                                               | Addition and subtraction                                                                                                                                                                        | left to right |
| << >>                                             | Bitwise left shift and right shift                                                                                                                                                              | left to right |
| < <=<br>> >=                                      | relational less than/less than equal to<br>relational greater than/greater than or equal to                                                                                                     | left to right |
| == !=                                             | Relational equal to or not equal to                                                                                                                                                             | left to right |
| &&                                                | Bitwise AND                                                                                                                                                                                     | left to right |
| ^                                                 | Bitwise exclusive OR                                                                                                                                                                            | left to right |
|                                                   | Bitwise inclusive OR                                                                                                                                                                            | left to right |
| &&                                                | Logical AND                                                                                                                                                                                     | left to right |
|                                                   | Logical OR                                                                                                                                                                                      | left to right |
| ? :                                               | Ternary operator                                                                                                                                                                                | right to left |
| =<br>+= -=<br>*= /=<br>%= &=<br>^=  =<br><<= >>=  | Assignment operator<br>Addition/subtraction assignment<br>Multiplication/division assignment<br>Modulus and bitwise assignment<br>Bitwise exclusive/inclusive OR assignment                     | right to left |
| ,                                                 | comma operator                                                                                                                                                                                  | left to right |

## Operatori Condizionali e Speciali



- Condizionali
  - ?
  - Unico operatore ternario del C
- Speciali
  - Li dettaglieremo man mano che li incontreremo
    - sizeof() dimensione di un dato
    - & indirizzo di una variabile
    - \* puntatori o accesso memoria
    - , concatenazione di espressioni
    - [] indice array
    - ...

19/22

## Operatori bit a bit



- Il C riesce ad operare anche a basso livello sui dati
  - Singolo bit
  - Dati interi
- Perché si usano?
  - Efficienza
  - Rappresentazione compatta di valori vero/falso ovvero 1/0
  -

20/22

## Operatori bit a bit



- Lavorano a livello dei singoli bit di ciascun dato
  - $\&$  → and bit a bit
  - $|$  → or bit a bit
  - $\sim$  → not bit a bit
  - $\wedge$  → xor bit a bit
  - $\ll$  → left shift
  - $\gg$  → right shift
- Li vedremo nel dettaglio successivamente

21/22



UNIVERSITÀ DI PARMA

## Le Espressioni



*We must either institute conventional forms  
of expression or else pretend that we have  
nothing to express.*

*George Santayana, Soliloquies in England*





UNIVERSITÀ DI PARMA

# Bitwise Operators

*A Bit of Fun*

## Sommario

UNIVERSITÀ  
DI PARMA 

- Bitwise operators
- Reason why
- Single & multiple bits operations
- Shift operators

**SUMMARY**

2/13

## Bitwise Operators



- Memory is composed by “bits”
- Every data is then stored in memory in binary
- Usually we do not need to access the single bit
- C anyway provides us specific operators that work at bit level

3/13

## Reason why



- Why we should deal with single bits?
  - Access to raw HW often requires to deal with single bits
  - More efficient arithmetic operations
  - More efficient storing of data

4/13

## Bitwise operators

- They can work on single bits

- $\&$  → binary AND 
- $|$  → binary OR 
- $\sim$  → binary NOT 
- $\wedge$  → binary XOR 
- $\ll$  → left shift
- $\gg$  → right shift

5/13

## Single bit operations



6/13

## Multiple bits operations



- These operators are applied to integer types
  - Usually multiple bits!
- The operator is applied to the corresponding bits in each data

| AND    |  |
|--------|--|
| 0110   |  |
| & 1100 |  |
| <hr/>  |  |
| 0100   |  |

| OR    |  |
|-------|--|
| 0110  |  |
| 1100  |  |
| <hr/> |  |
| 1110  |  |

| XOR    |  |
|--------|--|
| 0110   |  |
| ^ 1100 |  |
| <hr/>  |  |
| 1010   |  |

| NOT    |  |
|--------|--|
| ~ 1100 |  |
| <hr/>  |  |
| 0011   |  |

- Note that results are different from the same logical operator!

7/13

## Shift operators



- Shift all the bits of a data to the right or to the left
- Look at the following example
- We want to shift 213 by 3

213

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

213 &lt;&lt; 3    1    1    0

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | ? | ? | ? |
|---|---|---|---|---|---|---|---|

213 &gt;&gt; 3

|   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|
| ? | ? | ? | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|

8/13

## Shift operators

- In both cases
  - Bits shifted off the end are lost

213

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

213 &lt;&lt; 3    1    1    0

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | ? | ? | ? |
|---|---|---|---|---|---|---|---|

213 &gt;&gt; 3

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| ? | ? | ? | 1 | 1 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|

1    0    1

9/13

## Shift operators

- Not symmetrical behavior
  - Left shift → new lower order bits are filled in with 0s
  - Right shift → highest order bit is replicated

213

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

213 &lt;&lt; 3

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

213 &gt;&gt; 3

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|

10/13

## Biwise Operator Tricks



- $|$  with 1 is useful for turning select bits on
- $\&$  with 0 is useful for turning select bits off
- $|$  is useful for taking the union of bits
- $\&$  is useful for taking the intersection of bits
- $^$  is useful for flipping select bits
- $\sim$  is useful for flipping all bits

11/13

## Practical usage



- Integer number manipulation
  - Get/extract a specific byte of an integer or specific bits
  - e.g., complement letter case
- Bit masking for flags/configuration etc.
- Fast math
  - Left/right shifts are multiplication/divide by two
  - Trick for power of 2 check

12/13



UNIVERSITÀ DI PARMA

# Le Espressioni



*We must either institute conventional forms  
of expression or else pretend that we have  
nothing to express.*

*George Santayana, Soliloquies in England*



UNIVERSITÀ DI PARMA

# Controllo di Flusso

*Begin at the beginning and go on till you  
come to the end: then stop.*

Lewis Carroll, Alice's Adventures in  
Wonderland.

## Sommario

UNIVERSITÀ  
DI PARMA

- Teorema Jacopini Böhm
- Sequenza
- Selezione
  - if()
  - switch()-case
- Iterazioni
  - while()
  - do-while()
  - For(;;)
- Indentazione

# SUMMARY



2/28



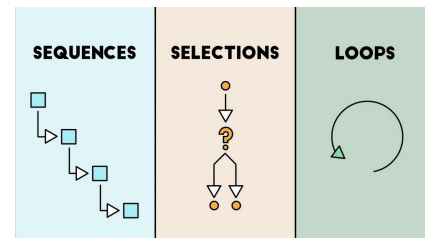
## Controllo di Flusso

- Indica in che ordine vengono eseguite le direttive del C
  - Remember: Algoritmo = Passi + Ordine Esecuzione
- Fino ad ora ci siamo limitati alla “sequenza”
  - Non basta!
- Per alcune istruzioni vorrei poter impostare:
  - Ordine di esecuzione
  - Se eseguire o meno
  - Quante volte eventualmente ripetere

3/28

## Teorema di Jacopini Böhm

- Per costruire un programma sono necessari solo tre tipologie di esecuzione
  - Sequenza
  - Selezione (valutazione condizioni)
  - Iterazione (ciclo)
- Il C fornisce soluzioni multiple a queste strutture di controllo

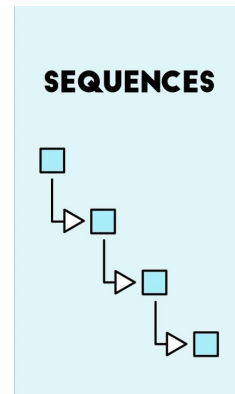


4/28

## Sequenza

UNIVERSITÀ  
DI PARMA

- Quanto visto fino ad ora negli esempi
  - Le istruzioni sono eseguite in sequenza
- Prestare attenzione all'ordine
  - Chi scrive il codice decide l'ordine
  - Il sistema segue pedissequamente



5/28

## Blocco di Istruzioni

UNIVERSITÀ  
DI PARMA

- Definiamo il “blocco di istruzioni” (*Compound Statement*)
  - In pratica una sequenza di istruzioni poste tra “{” e “}”
  - Esempio → corpo della main()

```
{
    <dichiarazioni variabili>
    <istruzione 1>
    ...
    <istruzione n>
}
```

6/28

## Valutazione Condizioni



- Detta anche “selezione”
- Valuto una o più condizioni
- In base all’esito della valutazione decido cosa fare



### SELECTIONS



7/28

## Valutazione condizioni if()



```
if(espressione)  
    istruzione|blocco di istruzioni
```

- Se l’espressione è valutata come “vera” ( $\neq 0$ )
  - L’istruzione o il blocco di istruzioni sottostanti vengono eseguiti
- Se l’istruzione è valutata come “falsa” ( $= 0$ )
  - L’istruzione o il blocco di istruzioni sottostanti vengono saltate

8/28

## Valutazione condizioni if()-else



```
if(espressione)
    istruzione|blocco di istruzioni
else
    istruzione|blocco di istruzioni
```

- Indico un'alternativa se condizione non si verifica

9/28

## Valutazione condizioni switch()-case



```
switch (espressione){
    case <val1>: istruzione|blocco
    case <val2>: istruzione|blocco
    ...
    default:    istruzione|blocco
}
```

- Spesso occorre confrontare un'espressione con differenti valori
- Praticamente un if( == ) multiplo

10/28

## Valutazione condizioni switch()-case



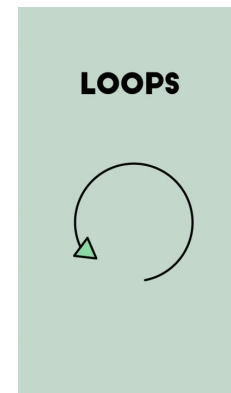
- Nello switch()-case
  - L'espressione deve essere "intera"
  - No virgola mobile, no stringhe
- I vari "case" devono corrispondere a costanti
  - Tutte differenti
  - L'ordine è influente, "default" non deve obbligatoriamente essere l'ultimo
  - Non ci vogliono {} salvo che non definisca variabili
- In assenza del "break" l'esecuzione passa al "case" successivo
- Il "default" non è obbligatorio

11/28

## Iterazioni



- Ripeto istruzione o blocco di istruzioni
- In C tre tipologie
  - while()
  - do-while()
  - for(;;)



12/28

## Iterazioni, ciclo while()



```
while(espressione)  
    istruzione|blocco di istruzioni
```

- Fino a che l'espressione è vera l'istruzione/blocco viene eseguito
  - Se l'espressione è inizialmente falsa, nessuna esecuzione
- Problemi:
  - Rischio ciclo infinito

13/28

## Iterazioni, ciclo do-while()



```
do  
    istruzione|blocco di istruzioni  
while(espressione);
```

- Fino a che l'espressione è vera l'istruzione/blocco viene eseguito
  - Se l'espressione è inizialmente falsa, almeno una esecuzione
- Problemi:
  - Anche qui rischio ciclo infinito

14/28

## do-while() vs while()



- while() loop checks the condition first and then executes the statement
  - If the condition is false the loop is never executed
  - Entrance controlled loop
- do-while() loop executes the statement, then the condition is checked
  - Statement is executed at least once
  - Exit controlled loop

15/28

## Commonly used steps of loops



- When using a loop, often:
  - We need to initialise something
  - We need to check a condition
  - We need to update something related to the condition

16/28

## Iterazioni, ciclo for(;;)



```
for(<inizializzazione>; <condizione>; <aggiornamento>)  
    istruzione|blocco di istruzioni
```

- Inizializzazione
  - espressione valutata inizialmente, tipicamente iniz. variabili
- Condizione
  - espressione valutata per stabilire se eseguire il corpo del ciclo
- Aggiornamento
  - espressione valutata dopo aver eseguito il corpo del ciclo

17/28

## Iterazioni, ciclo for(;;)



- Di fatto molto sovrapponibile al ciclo while()
  - Ma più comodo
- Nessuna delle 3 espressioni è obbligatoria
- Ciclo infinito se tutte assenti
  - for(;;) equivalente a un while(1)
- In C99 è permessa la definizione variabili in for()

18/28



## while() vs for(;;)

- Perfettamente intercambiabili
- Tipicamente for() quando il numero di iterazioni è noto



```
<expr1>
while(<expr2>)
{
    statement;
    <expr3>;
}
```



```
for(<expr1>;<expr2>;<expr3>)
{
    statement;
}
```

19/28

## break e continue

- break:
  - Permette uscita da corpo ciclo o da case
  - Cicli nidificati → esco dal solo corpo del ciclo in cui viene usata
- continue:
  - Usata con cicli permette la ripetizione immediata del ciclo
  - Nei for(;;) → salto ad esecuzione aggiornamento
  - Nei while()/do-while() → salto a valutazione condizione

20/28

## Cicli “infiniti”



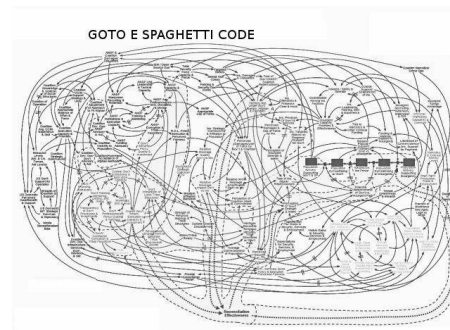
- Non voluti → il male
- Ma ci sono condizioni in cui servono
  - Semplificare il codice
- Con break esco
- Possibilità:
  - `for(;;)` // preferibile
  - `while(1)`

21/28

## Salto incondizionati



- `goto` + label
- Rendono i programmi complessi e difficilmente manutenibili o modificabili
- *Spaghetti Code*
- In parte vero anche per `break` e `continue`



22/28

## Code Indentation



- Indenting C programs can greatly help the readability
  - Clarity
- Who has to read your code?
  - Yes, that's me!
- As a general rule, a programmer has to dig out old code
  - And then to understand it!

23/28

## Code Indentation



```
#include <stdio.h>

int main()
{
    int v[10], n, i, j, temp;
    scanf("%d", &n);
    for (i = 0; i < n; i++)
        scanf("%d", &v[i]);
    for (i = 0; i < n; i++)
        for (j = i + 1; j < n; j++)
            if (v[i] > v[j])
            {
                tmp = v[i];
                v[i] = v[j];
                v[j] = tmp;
            }
    for (i = 0; i < n; i++)
        printf("%d ", v[i]);
    printf("\n");
    return 0;
}
```

```
#include <stdio.h>

int main()
{
    int v[10], n, i, j, temp;
    scanf("%d", &n);
    for (i = 0; i < n; i++)
        scanf("%d", &v[i]);
    for (i = 0; i < n; i++)
        for (j = i + 1; j < n; j++)
            if (v[i] > v[j])
            {
                tmp = v[i];
                v[i] = v[j];
                v[j] = tmp;
            }
    for (i = 0; i < n; i++)
        printf("%d ", v[i]);
    printf("\n");
    return 0;
}
```

24/28

## Code Indentation



- We need to indent
  - Functions body (i.e. main())
  - Loops statement
  - Selection statement
- How many spaces?
  - 2-4 are the most used options
  - >8 reduce readability
  - Avoid tabs

25/28

## Code Indentation



- Different styles
  - In examples we usually adopt the “Allman” or “BSD” style
  - Up to you, but being consistent is suggested

```
while (something)
{
    something();
    something_else();
}

final_thing();
```

26/28

## Code indentation



- Cave Canem!
- There is someone that does not care about the indentation!
  - Yes, that's the compiler
  - You have always to use {}

```
int sum, sum2;  
for(int i=0; i<17; ++i)  
    sum  += i;  
    sum2 += i*i;
```

27/28



UNIVERSITÀ DI PARMA

## Controllo di Flusso



*Begin at the beginning and go on till you  
come to the end: then stop.*

Lewis Carroll, Alice's Adventures in  
Wonderland.



UNIVERSITÀ DI PARMA

# Data Types in C

*What's in a name? That which we call a rose,  
By any other name would smell as sweet.*

William Shakespeare, Romeo and Juliet,  
Act II, Scene II

## Summary

UNIVERSITÀ  
DI PARMA

- What is a “variable”?
- Data Type in C
- Numbers
  - Floating point data types
  - Integer data types
- Combining different types
  - Cast
- Visibility & Lifetime

# SUMMARY



2/42

## Variables



- Something that is able to “vary”
- Definition:
  - **Memory space** with a **name** where it is possible to **store data** during program execution
- Memory space → address & size
  - Automatically managed by high level programming languages

3/42

## Terminology



- Content or value of a variable → right value or rvalue
- Memory address → left value or lvalue
- Size

```
int a = 173;
```

4/42

## Name of a variable



- Rules
  - Only letters, digits, or underscore “\_”
  - It must begin with a letter (or “\_”)
  - Names are case sensitive
  - Reserved words can not be used

5/42

## Name of a variable



- Recommended style:
  - Select small caps for variable names
  - Use meaningful names
  - When the name is composed by multiple words use uppercase letters or underscore to emphasize that

```
int sum           = 173;  
int students_number = 144;  
int ClassId       = 10;
```

6/42



## How to define a variable



```
// variables are defined as
// <type> <variable_name>;

int a;
    // I am defining a "int" variable named 'a'
    // in such a case the variable is NOT initialized
    // this means that 'a' contains a RANDOM number

int b=17;
    // I am defining AND initializing a "int" variable named 'b'
```

7/42

## Defining & initing a variable



```
int a;
a = 10;
```

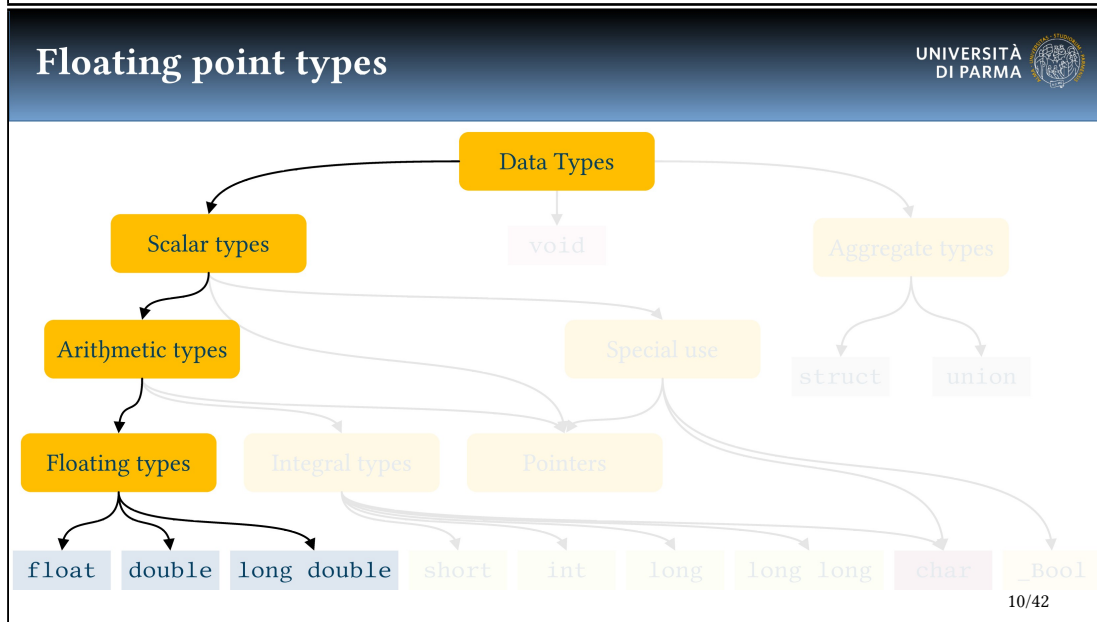
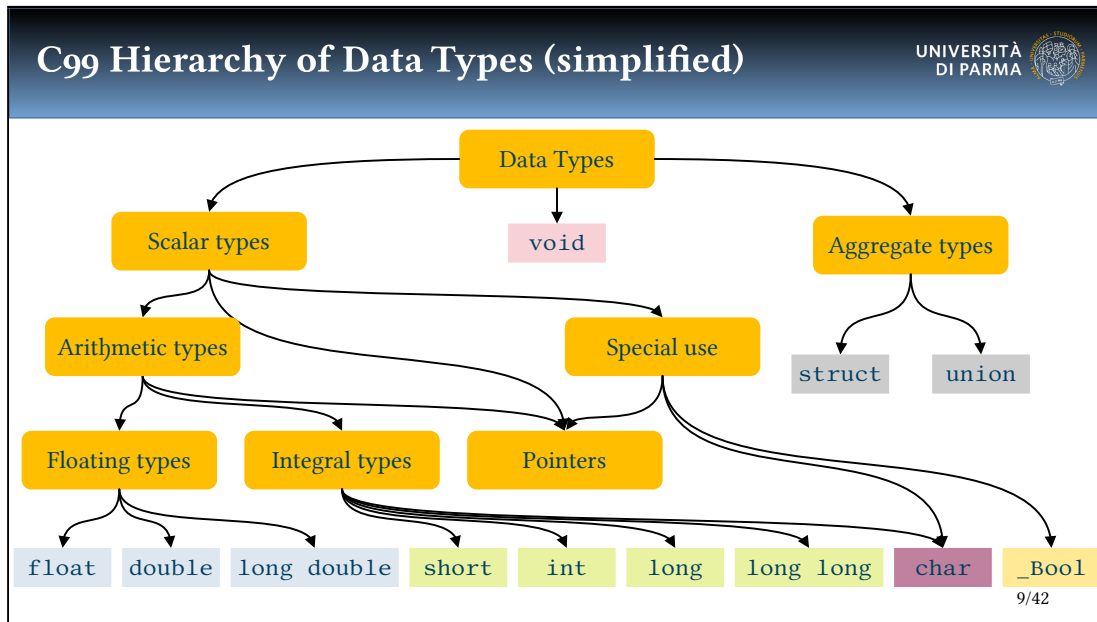
→

0x7fff387eaea4

10

- Definition of a variable does not initialize it
  - Random value
  - Do not rely on it being 0
- Always init variables before using them!

8/42



## Floating point types



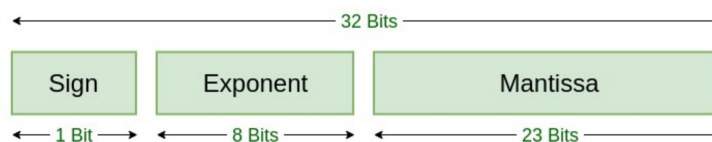
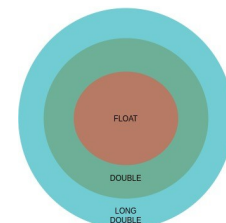
- They can hold very large numbers or fractions
  - Positive or negative numbers
  - With or without the fractional part
  - Remember approximation
- Why we have approximation?
  - Limited size for the fractional part
  - Min & Max values limit

11/42

## Floating point types



- 3 possibilities
  - float → single precision
  - double → “double” precision
  - long double → extended precision
- Main difference?
  - Number of bytes used to store the data

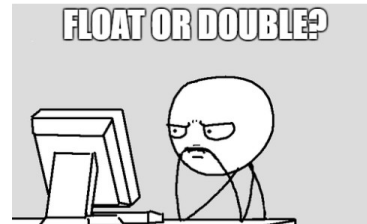


12/42

## Float vs Double vs Long Double



- Which type should I use?
- As always it is a compromise amongst:
  - Precision
  - Execution speed
  - Memory requirements



13/42

## Floating Point Types



- The C standard recommends the IEEE 754 format?
  - Actually not, but this is not important
- The C standard says anything about the size of different types?
  - No
- It is up to the compiler!
  - Often, there is not difference between float or double types

14/42

## sizeof()



- We have a tool to understand the size of a variable
- sizeof(<data type>) or sizeof(<variable name>)
  - It returns the number of bytes used to store that data
- A specific prefix ('z') for a correct format specifier
  - %zd C99
  - %ld C89 – use this one...

15/42

## Floating Point Types



- How to print or read them?
  - Formats
 

|                                   |              |
|-----------------------------------|--------------|
| • %f → decimal notation           | 123.456000   |
| • %e → scientific notation        | 1.234560e+02 |
| • %E → scientific notation        | 1.234560E+02 |
| • %g → shortest between %f and %e | 123.456      |
  - Suffissi per tipo
 

|            |                                   |
|------------|-----------------------------------|
| • "none" → | float                             |
| • l →      | double (only mandatory for input) |
| • L →      | long double                       |

16/42

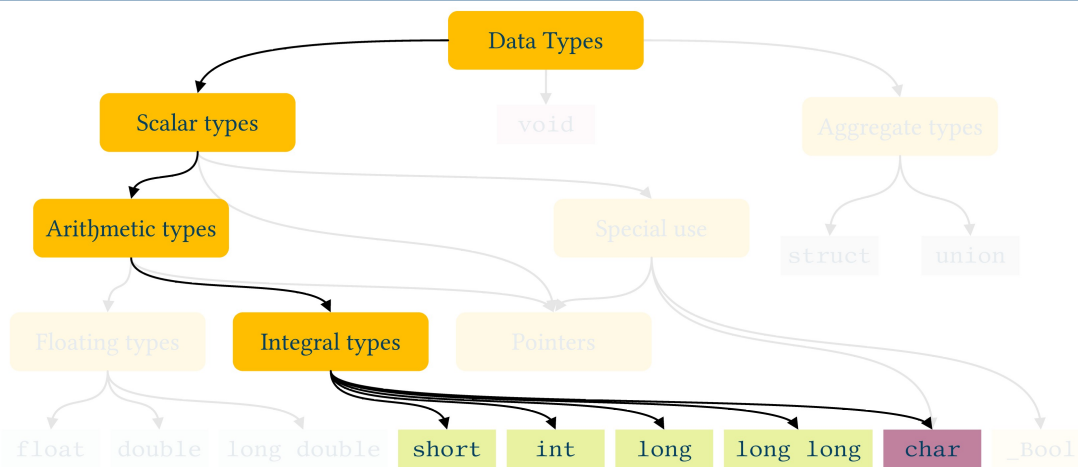
## Floating Point Types



- Specific values
  - $\pm\text{inf} \rightarrow$  Infinite
    - Usually as a result of math operations
    - i.e. division by zero
  - NaN  $\rightarrow$  not a number
    - Probably some memory error
- Specific functions
  - `isnan()`, `isinf()`...

17/42

## Integer Types



18/42

## Integer Types



- 4+1 types:
  - char → seldom used for “numbers”
  - short int
  - int → probably the most used data type
  - long int
  - long long int
- The difference is, again, the size of memory used to store them
  - Given  $n$  bits I can store non negative numbers in the  $0 - 2^n - 1$  interval

19/42

## Integer Types



- All integer types can deal with negative numbers (“signed” types)
  - The range of numbers I can store is then not  $[0, 2^n - 1]$  but  $[-2^{n-1}, 2^{n-1} - 1]$
- By default all signed types but
  - char – compiler dependent
- When I use them I can change the default behavior
  - unsigned
  - (signed)
- This brings the number of possible integer types to 10!

20/42

## Integer Types (typical example for 64 bit)

| Type               | Byte | Smallest Value             | Largest Value              |
|--------------------|------|----------------------------|----------------------------|
| signed char        | 1    | -128                       | 127                        |
| unsigned char      | 1    | 0                          | 255                        |
| short              | 2    | -32.768                    | 32.767                     |
| unsigned short     | 2    | 0                          | 65.535                     |
| int                | 4    | -2.147.483.648             | 2.147.483.647              |
| unsigned int       | 4    | 0                          | 4.294.967.295              |
| long               | 8    | -9.223.372.036.854.775.808 | 9.223.372.036.854.775.807  |
| unsigned long      | 8    | 0                          | 18.446.744.073.709.551.615 |
| long long          | 8    | -9.223.372.036.854.775.808 | 9.223.372.036.854.775.807  |
| unsigned long long | 8    | 0                          | 18.446.744.073.709.551.615 |

21/42

## Integer Types

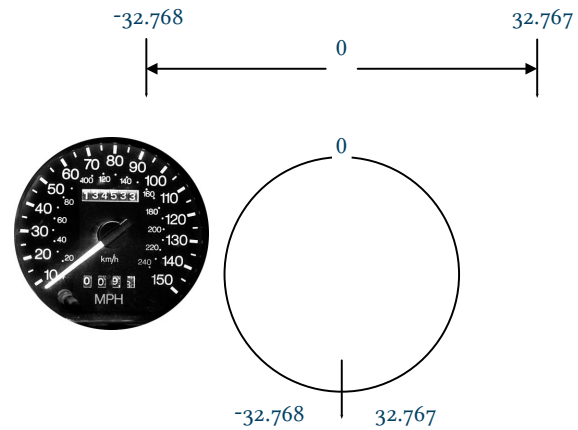
- How we can read or print them?
  - Formats
    - %d → decimal notation -45
    - %x → hexadecimal notation fffffd3
    - %X → hexadecimal notation FFFFFFFD3
    - %u → decimal notation for unsigned types 4294967251
  - Suffixes
    - "none" → int
    - hh → char
    - h → short
    - l → long
    - ll → long long

22/42



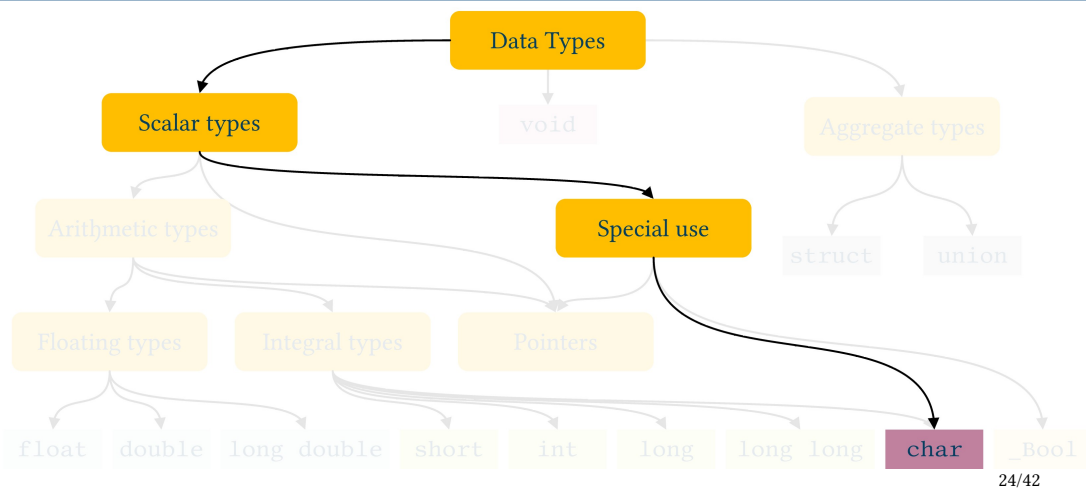
## Overflow

- When an operation produces a result too big (or too small) for the given type
- No execution error!
  - Usually clamping



23/42

## Char Type



24/42

## Char Type



- It is used to store numbers!
  - 0–255
  - -128–127
- But typically only used to store ASCII codes
- Format specifier
  - %c
- Constants:
  - ~~Direct use of magic numbers, i.e. 65~~
  - Use of correspondent symbol 'A'

STOP

Do not  
even think  
to do that!

25/42

## The char type & scanf()



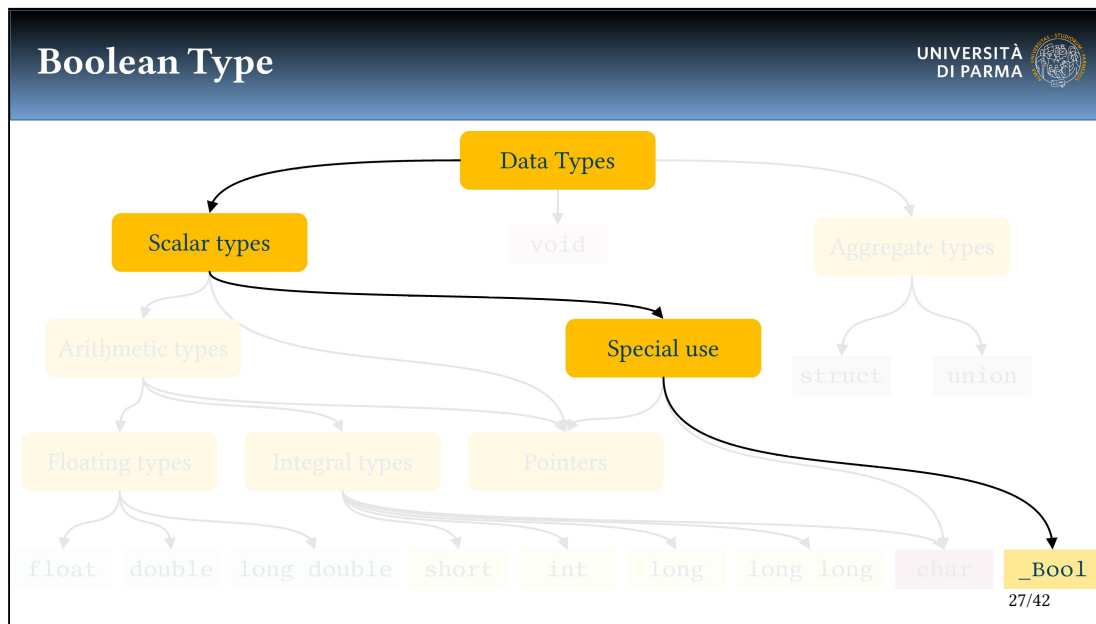
- Reading the char and derived types using scanf() can be a little tricky
- When I type on the keyboard all characters are fed as input to the program
  - All means all. Namely, also the enter key...
  - All those “key presses” are initially stored in a input buffer



```
scanf("%c", &x); // I also read spaces and newlines
```

```
scanf(" %c", &x); // Ignore leading spaces or newlines
```

26/42

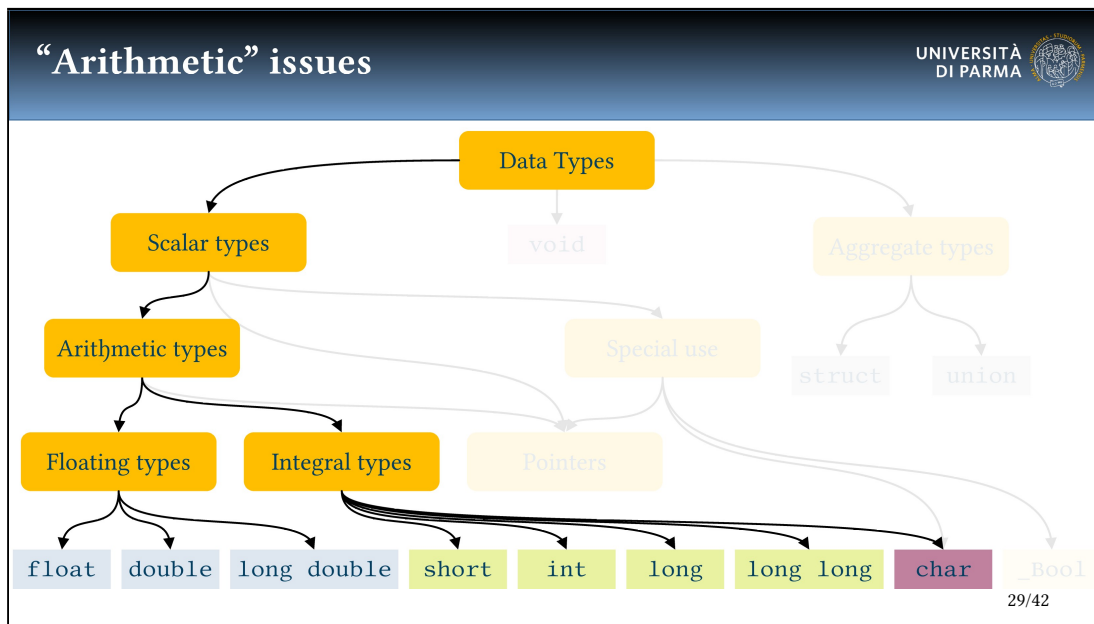


## The \_Bool Type

UNIVERSITÀ DI PARMA

- C99 (and later) specific
- A integer data type with the aim to only store “true” or “false”
  - 0 → false
  - 1 → true
- This type can then features 1 or 0 values only

28/42



## Combining different data types

UNIVERSITÀ DI PARMA

- 3 floating point types and 8–10 integer ones
- Can I combine them in operations?
  - Yes
- What is the C behavior when I mix different kind of data types?
- Conversions
  - Assignment
  - Implicit or automatic conversions

30/42

## Assignment conversions



- When I operate an assignment (=)
  - The right operand (rvalue) is “casted” to the type I have on the left operand (lvalue)
- Issues
  - I can lost information                      i.e.: float  $\rightarrow$  int
  - Wrong result                                      i.e.: long long  $\rightarrow$  short

31/42

## Implicit or automatic casting



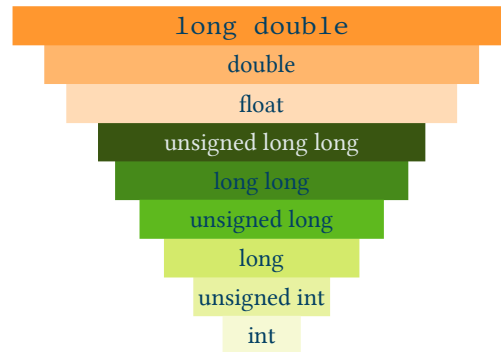
- It occurs when we combine different types in expressions
- Binary operands needs same type operators
  - Arithmetic: \*, /, %, +, -, ; Comparisons: <, >, <=, >=, ==, !=; Bitwise: &, ^, |,
- i.e.: a+b when a is an int and b is a float which kind of type gives as a result?
  - Spoiler: float
- Conversion rules
- High probability of bugs!

32/42

## Implicit cast rules



- Some types are converted to int or unsigned int
  - Even when we have same type operands
  - `_Bool`, `char` e `short` → int
    - Or unsigned int
- After this → conversion pyramid



33/42

## Explicit cast



- I select the conversion
  - This lead to avoid errors
- Syntax:

**(type) expression**

34/42

## Variable scope



- Can I use a variable everywhere in my code?
  - No!
- Visibility field or scope
  - Part of the code where a variable can be used
- 3 cases:
  - Program (global variables)
  - File
  - Block (local variables)



35/42

## Global Variables



- Defined outside of every code block
  - `{ }`
- I can modify them everywhere
  - Data are NOT protected
  - Debug becomes more difficult
  - Uncertain
- Your final score will be, FOR SURE, affected
- One single benefit → initialization to 0



36/42

## Local Variables



- Defined
  - Inside a code block i.e. inside {}
  - In a for(;;)
  - As function arguments
- I can use them only inside the specif block where they are defined
- Automatic duration
  - Created when the execution flow meets that code block
  - Destroyed when that code block is over

37/42

## Global vs Local



- Local Variables
  - Let code blocks or functions to be self-sufficient
  - Easier to manage or to understand the code
    - Modularity
  - Less portability issues
- Global Variables
  - I have to track all the code parts where they are used
  - Require to coordinate team work
  - Memory is always used
  - Naming issues

38/42



## Variables Duration



- A variable is not always forever
- Static lifetime
  - Global Variables
    - Memory reserved when the execution begins and never freed
- Automatic lifetime
  - Local variables
    - Local variables are created when code execution reaches that block
    - And destroyed when the execution is over
    - Can I change this behavior? Yes → `static`



39/42

## Constant values



- We can need to use constant values
- Quick and dirt approach:
  - Put in my code a lot of “magic numbers”
  - This lead to a code really difficult to modify or maintain
- Good approach:
  - Define constant
  - Parametric code, very easy to modify or improve

 $e$   
 $\pi$   $\mu$ 

40/42

## How to define a constant value in C



- Three different approaches
- `#define`
  - i.e.: `#define N_MAX_STUDENTS (100)`
- `const` prefix
  - I define and init a variable whose initial value can not be modified
  - In such a case the use of a global variable makes a sense
  - i.e.: `const int N_MAX_STUDENTS=100;`

41/42



UNIVERSITÀ DI PARMA

## Data Types in C



*What's in a name? That which we call a rose,  
By any other name would smell as sweet.*

William Shakespeare, Romeo and Juliet,  
Act II, Scene II



UNIVERSITÀ DI PARMA

# Predefined Functions

*E Pluribus Unum**Autore Ignoto, Moretum, I sec. a.C.*

## Sommario

UNIVERSITÀ  
DI PARMA 

- Why we need functions?
- C standard library
- Predefined functions
  - Anatomy of a function

**SUMMARY**

2/19

## Definition of a function



- A function is a **group of statements** that, together, perform a given task
  - i.e. performing something **more complex** than “simple” C instructions
  - All high level programming languages have the concept of “function”
  - Every C program features at least one function → `main()`
- In C we also have macros, but we won't talk about them

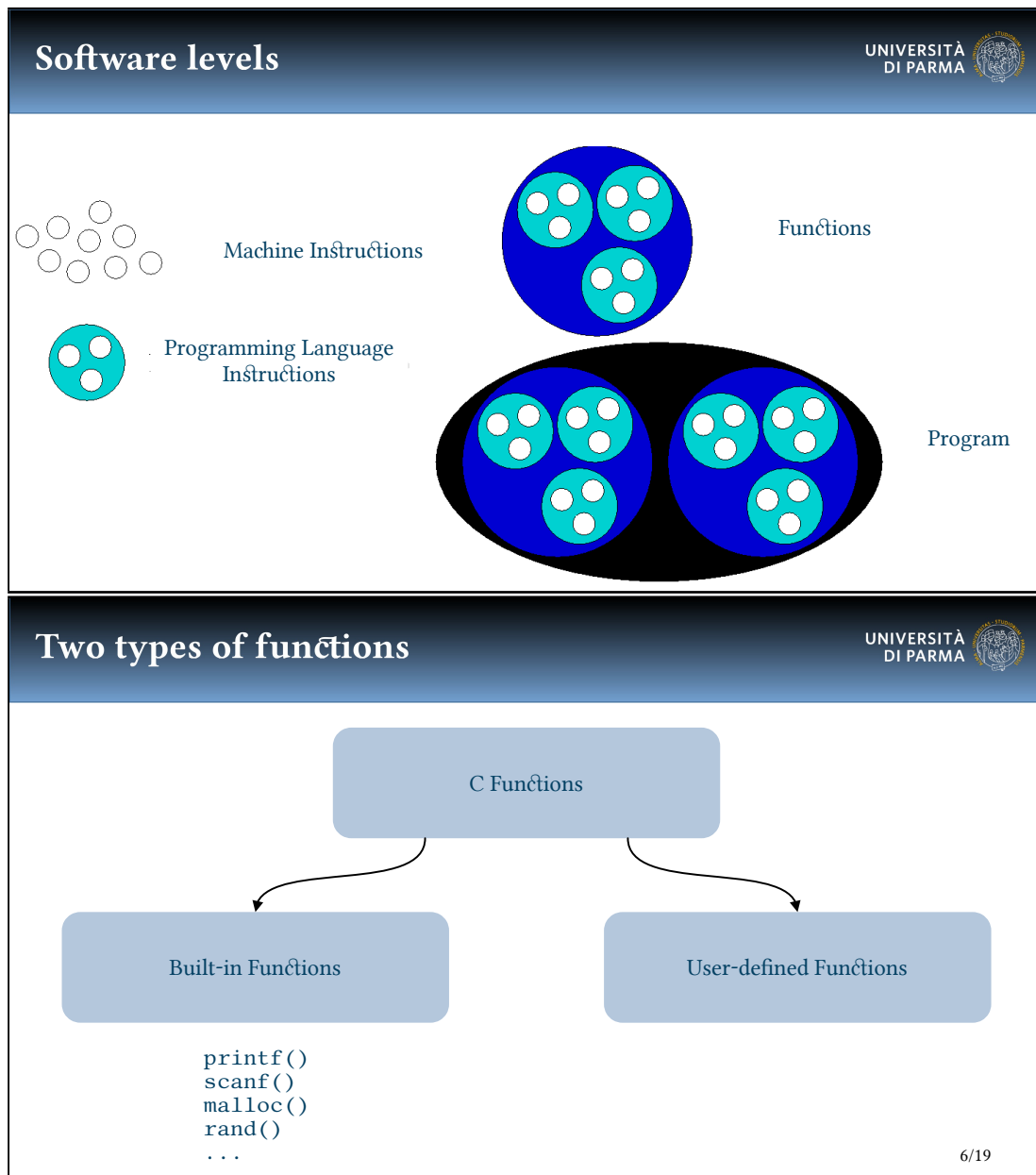
3/19

## Why we want to use them?

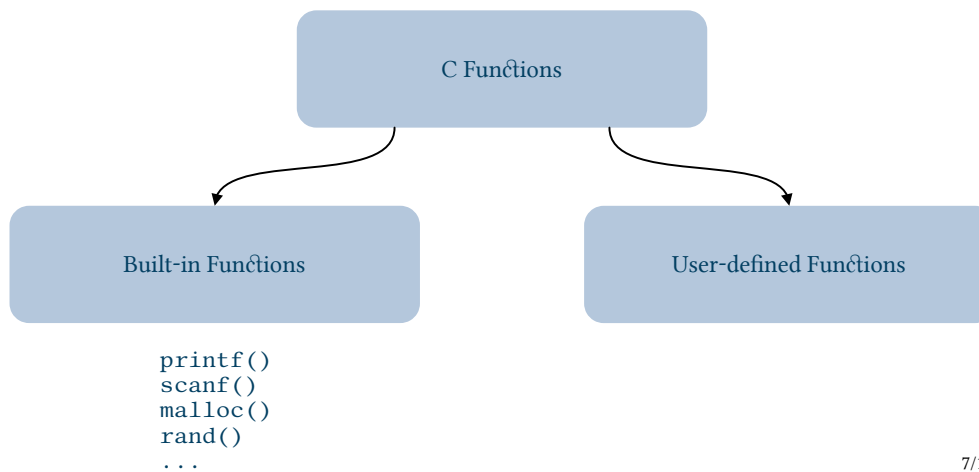


- **Factorization:** a function is defined once but can be used countless
  - Reusable code
  - Reduce code size → greater efficiency
- **Ease of modification:** redundancy is reduced
  - I modify/improve/correct code in one place
- **Better readability:** code details can be isolated
  - The use of functions can be self-explanatory
  - Breaking the code in smaller functions, makes it more readable and structured

4/19



## Two types of functions



7/19

## Built in functions/library functions

- Function already available that are provided by the C library
  - Input/output: printf(), scanf()...
  - Math: sqrt(), cos(), sin()...
  - Time/Dates: time()...
  - Memory management: not yet discussed
  - String/Character: not yet discussed
  - Other: rand(), srand()...
  - ...

8/19

## Underlying idea



- Do not reinvent the wheel!
- The *C Standard Library* enable us with a lot of useful functions
- Anyway
  - We need to know what functions are provided...
  - Include a specific header
  - Use them!
    - This also implies we need to know how to use them



9/19

## Main C header files



- `stdlib.h` → general use (I/O, memory, data types...)
- `string.h` → string management
- `math.h` → mathematical functions
- `time.h` → time and calendar management
- `ctype.h` → characters inspection and management

10/19

## Function “signature”



- When we look at any C manual we can find a function description
- and a function “signature”, something like:
  - `int rand(void); void srand(unsigned int seed); double sqrt(double x);`

Type of data returned  
by the function  
(int, float, char...)

Function name

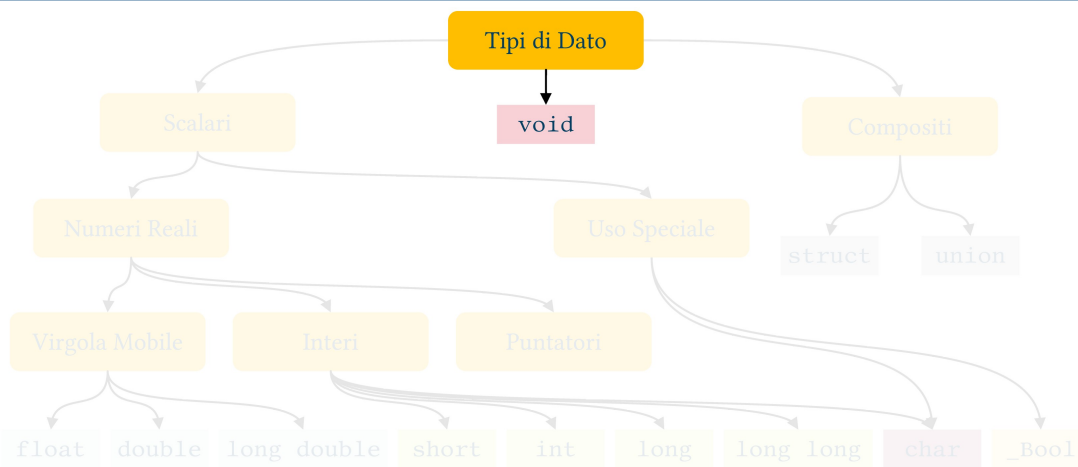
Formal parameters  
list

↓ ↓ ↓

`<return type> <function name>(<formal param1>, ...);`

11/19

## C99 Albero dei Tipi di Dato



12/19



## void



- It is a *no-type*
- We can not use *void* to define variables
- Used for
  - Functions
    - No value
    - No parameters
  - Pointers (we will see them later, spoiler: initially you will not like them)
    - No type

13/19

## Function name



- What we need to “use” it
- Again remember that lower case  $\neq$  upper case

Function name



`<return type> <function name>(<formal param1>, ...);`

14/19

## Returned type



- It allows us to understand which type of data the function provides
  - `int rand()`, `double sqrt()`, **`int`** `abs()`
- Not all functions return something
  - `void srand()`

Type of data returned  
by the function  
(int, float, char...)



**<return type>** `<function name>(<formal param1>, ...);`

15/19

## List of formal parameters



- Many functions needs some data as input
  - `double sqrt(double x);`
- Sometimes more than one parameter → comma is used
  - `double pow(double x, double y);`
- Or even nothing
  - `int rand(void);`

Formal parameters  
list



**<return type>** `<function name>(<formal param1>, ...);`

16/19

## #include<>

- A good manual also point us which header file is needed

### **exit** *Exit from Program*

```
void exit(int status);
```

Calls all functions registered with `atexit`, flushes all output buffers, closes all open streams, removes any files created by `tmpfile`, and terminates the program. The value of `status` indicates whether the program terminated normally. The only portable values for `status` are 0 and `EXIT_SUCCESS` (both indicate successful termination) plus `EXIT_FAILURE` (unsuccessful termination).

&lt;stdlib.h&gt;

9.5, 26.2

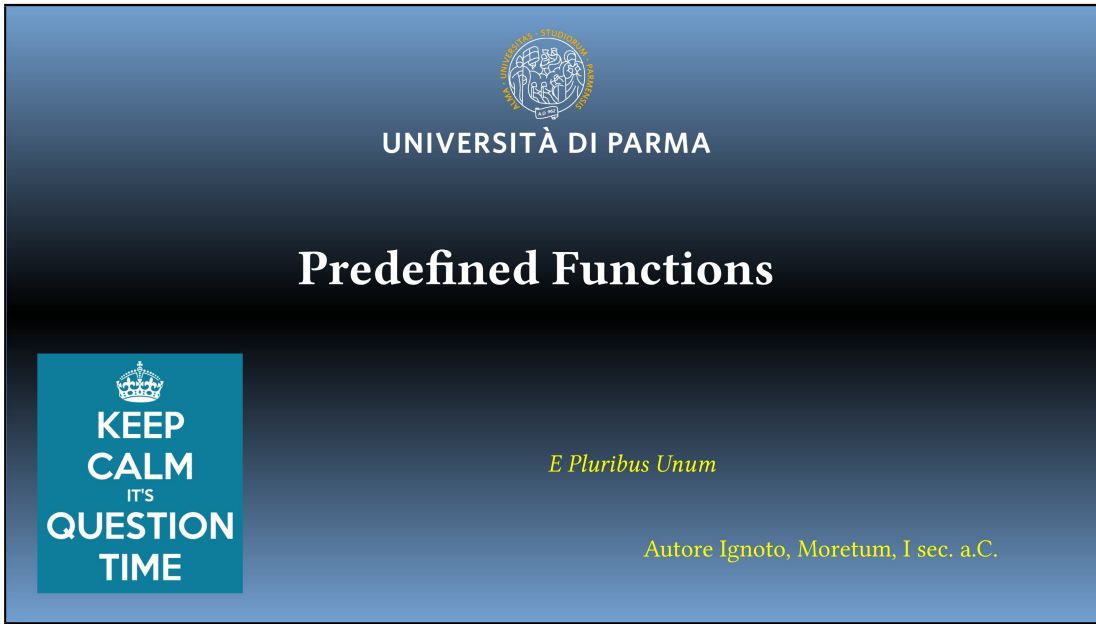
17/19

## Invoking a pre-defined function

- The knowledge of the function signature is all what we need

```
x1 = (-b + sqrt( pow(b,2) - 4*a*c )/( 2*a );  
x2 = (-b - sqrt( pow(b,2) - 4*a*c )/( 2*a );
```

18/19

This image shows a single sheet of white paper with horizontal ruling lines. The lines are evenly spaced and run across the width of the page. There are no margins, text, or other markings on the paper.



UNIVERSITÀ DI PARMA

# Gli Array

*When you have more data you win*

## Sommario

UNIVERSITÀ DI PARMA

- Array monodimensionali
  - Definizione
  - Uso
  - Inizializzazione
  - Errori
- Array multidimensionali



SUMMARY

2/15

## Limite variabili



- Fino ad ora abbiamo visto variabili “singole”
  - Gestisco uno e un solo “dato”
- Non adatte a gestire grandi moli di dati
- Eppure informatica nasce proprio per trattare dati
  - Informazione + Automatica → Informatica
- Soluzione C → Array

3/15

## Definizione Array



- **In C** si definisce array una struttura dati che permette di memorizzare **un numero definito** di dati tutti dello **stesso tipo**
- Caratteristiche:
  - Come per le variabili l’array ha un **nome** e uno specifico **tipo di dato**
  - Ciascun dato è detto **elemento** o componente
  - Gli elementi sono **ordinati**
    - A ogni elemento è associato un **indice numerico**

4/15

## Array monodimensionali



- Definizione array monodimensionale (vettore)

```
<tipo array> <nome array>[<numero elementi>;
```

- Esempio

```
int numeri[100];  
  
// definisco un array di nome "numeri" composto da  
// 100 elementi di tipo int
```

5/15

## Array monodimensionali



- Definire un array di tipo x composto da N elementi equivale a
  - Definire N variabili di tipo x
    - Ma è più pratico
- Come accedo ai singoli elementi di un array?
  - Gli elementi sono ordinati
  - Indice da 0 a N-1 → tra [ e ]

Array *a* di 10  
elementi

|  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|

a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]

6/15

## Array monodimensionali



- Salvo indice, stessa sintassi variabili “regolari” per scrittura/lettura

```
int a [100];           → array di 100 interi
...
a[0] = 37;             → scrivo in primo elemento
...
printf(“%d”, a[8]);    → leggo nono elemento
...
scanf(“%d”, &a[99]);   → scrivo in ultimo elemento
```

7/15

## Array monodimensionali



- Anche gli array possono essere inizializzati quando si definiscono
- Qualche differenza:
  - **Inizializzazione completa:**  
`float costanti[3] = {3.14, 2.718, 1.618};`
  - **Inizializzazione completa con omissione dimensioni:**  
`short primi[] = {1, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43};`
  - **Inizializzazione parziale:**  
`char lettere[50] = {'P', 'i', 'e', 'r', 'o'};`  
`int tuttizeri[100]={0}; // C99 does not allow = {};`  
`float sparsevector[30]={[3]=4.5, [29]=1.7}; // C99`  
In questi ultimi casi gli elementi non definiti sono comunque inizializzati a 0

8/15



## Array e memoria



- I singoli elementi di un array sono memorizzati consecutivamente in memoria:
- L'accesso ad un elemento avviene come: indirizzo dell'array+indice
  - Ci torneremo...

```
int a[5]={3,4,8,3129,77};
```

|      |      |       |
|------|------|-------|
| a[0] | 3    | 0x120 |
| a[1] | 4    | 0x124 |
| a[2] | 8    | 0x128 |
| a[3] | 3129 | 0x12C |
| a[4] | 77   | 0x130 |

9/15

## Array e sizeof()



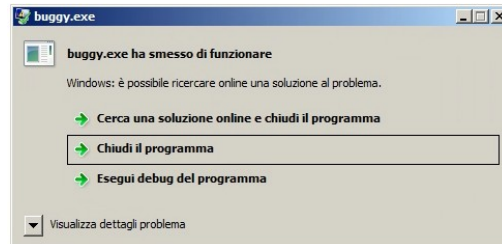
- Anche per array posso usare macro sizeof()
  - Per singoli elementi → memoria occupata da singolo elemento
  - Per array → memoria occupata da tutto l'array
- Utile per “calcolare” numero elementi dell'array

10/15

## Buffer Overrun



- Iniziando a usare gli array vedremo spesso qualche errore
- Perché?



11/15

## Buffer Overrun



- Il C non controlla coerenza indice ↔ dimensione array

```
int dummy[100];  
dummy[100]=7; // l'ultimo elemento ha indice 99  
dummy[-1]=0; // -1?
```
- Le ultime due righe sono un esempio di **buffer overrun**
  - malfunzionamento del programma
  - malfunzionamento del sistema

12/15

## Array Multidimensionali



- Gli array non sono limitati ad un solo indice
  - Ovvero a una sola dimensione
- Esempi di definizioni di array multidimensionali
  - `int matrice[10][9]; // 10 righe e 9 colonne?`
  - `long b[2][3][7][12][1024];`
- Anche in questo caso possibile inizializzare
  - `int m1[2][3]={3,4,8,1,2,7};`
  - `int m2[2][3]={ {3,4,8}, {1,2,7} };`

13/15

## Array multidimensionali e memoria

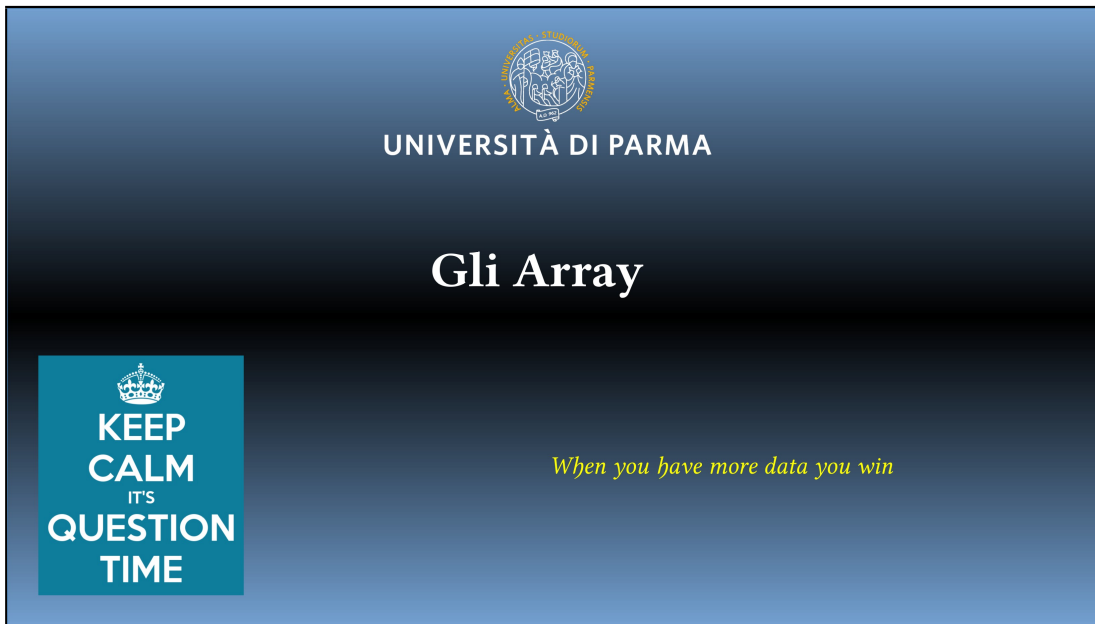


- Sebbene logicamente multidimensionali
- Memorizzati comunque sequenzialmente
- Posso vederli come
  - Array di array

```
int a[3][2]={3,4,8,3129,77,9};
```

|         |      |       |
|---------|------|-------|
| a[0][0] | 3    | 0x120 |
| a[0][1] | 4    | 0x124 |
| a[1][0] | 8    | 0x128 |
| a[1][1] | 3129 | 0x12C |
| a[2][0] | 77   | 0x130 |
| a[2][1] | 9    | 0x134 |

14/15





UNIVERSITÀ DI PARMA

# Memory Allocation & Pointers

*Memoria est thesaurus omnium rerum et  
custos*

De Oratore, Cicerone, I sec a.C.

## Summary

UNIVERSITÀ  
DI PARMA 

- Limitations of arrays
- Variable Length Array
- Pointers
- void and NULL
- Functions for memory allocation
- Common Mistakes



# SUMMARY

2/51

## Arrays Cons



- So far we used fixed size arrays
  - we have to put dimension in code
- We must made assumptions on the amount of data to manage
  - Waste of memory
  - Bug → buffer overrun
- How to solve?
  - Variable Length Array (VLA)
  - Dynamic memory allocation



3/51



UNIVERSITÀ DI PARMA

## VLA Variable Length Arrays

4

## Variable Length Array



- So far, we used constant to initialize arrays at **coding time**  

```
int a[123];
```
- From C99 we can use variables to define the number of elements of an array at **runtime**  

```
int size;  
fscanf("%d", &size);  
int a[size];
```
- Coding time ↔ Execution time
- No possible to initialize a VLA

5/51

## C standards



- Up to C99 → constant length arrays
- C99 → also variable length arrays
- After C99 → maybe variable length arrays
- C++? → forked from C well before VLAs

6/51

## VLA Pros & Cons



- Not everyone is happy with VLAs
  - Linux Kernel is VLA free
- Pros
  - Simple syntax
- Cons
  - Portability (C++, C11...)
  - Allocated on stack

7/51

## Beyond VLAs



- VLAs also feature limitations
  - The size of arrays can not be modified
  - Automatic Duration
    - They are destroyed at the end of a code block
    - i.e. functions
  - I can not destroy them once no more needed
  - “Limited” size

8/51





UNIVERSITÀ DI PARMA

# Pointers & Dynamic Memory Allocation

9

## Dynamic Memory Allocation



- Specific functions to dynamically allocate memory
  - Can be used to dynamically create arrays (or even single variables)
  - No automatic duration
  - They can be destroyed
- We need to introduce additional elements
  - Pointers
  - void type
  - size\_t type

10/51

## Dynamic Memory Allocation



- Main functions

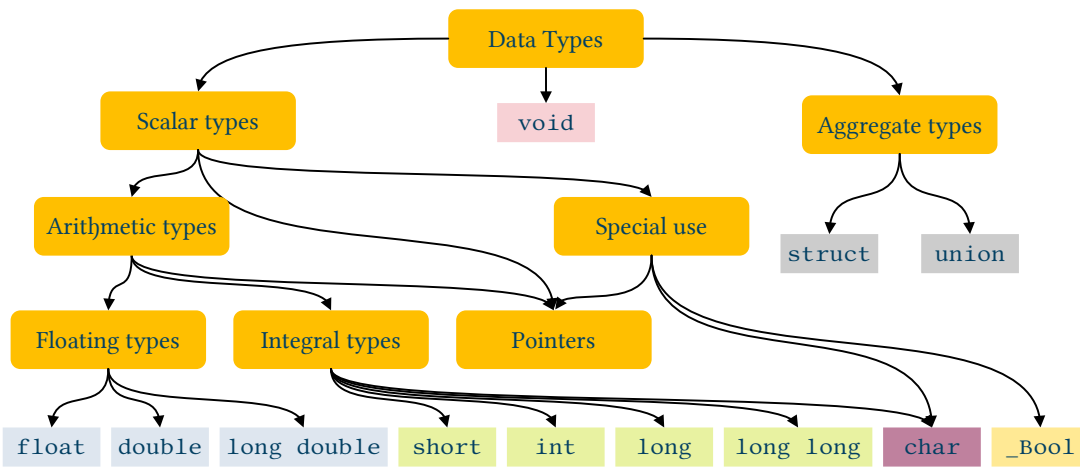
```
void *malloc(size_t size);
void free(void *ptr);
```

```
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

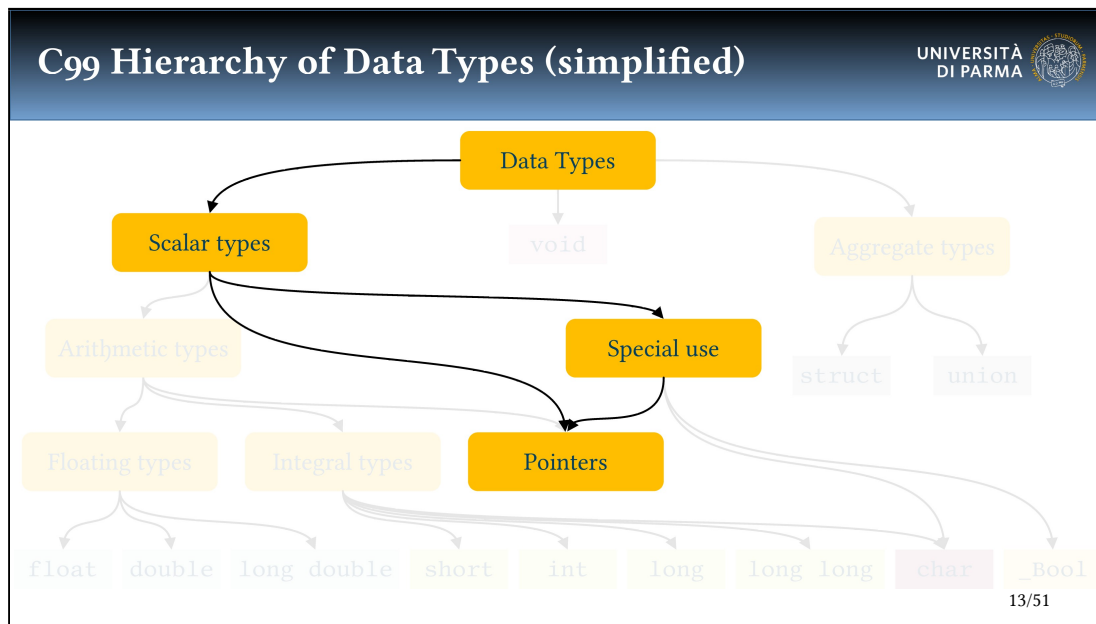
- Something has not yet been discussed
  - **void? ?? size\_t?**

11/51

## C99 Hierarchy of Data Types (simplified)



12/51



### Pointers

UNIVERSITÀ DI PARMA

- Formally, scalar type
  - More precisely, integer numbers → special meaning
- Two main uses:
  - Dealing with dynamic memory management
    - We are going to discuss this in the following
  - To define functions that can have “side effects”
    - We will be back on this when speaking again about functions

14/51

## Pointers



- Variable → memory location with a name
  - Size
  - Address
- Address is data as well → in some cases I need to know it
- How to deal with addresses? → Pointers!
- A pointer is a **data type** meant to **store memory addresses**
  - Addresses of variables or even other data

15/51

## Pointers



- How to know the memory address of a variable?
  - & operators for single variables
  - Array name without [] for arrays
  - `printf("The address off variable a is %p\n", &a);`
  - `printf("The address of array n is %p\n", n);`
- How to store addresses? → a pointer variable:
  - `int a; // int type variable`
  - `int *x; // pointer to a int type`
  - `x = &a; // x now contains the address of a`

16/51

## Pointers



- To define a pointer-type variable use '\*' before the name

```
int *p;      // pointer to int data
```

```
float *x;    // pointer to float data
```

```
long a, *b;  // only b is a pointer!
```

17/51

## '\*' operator

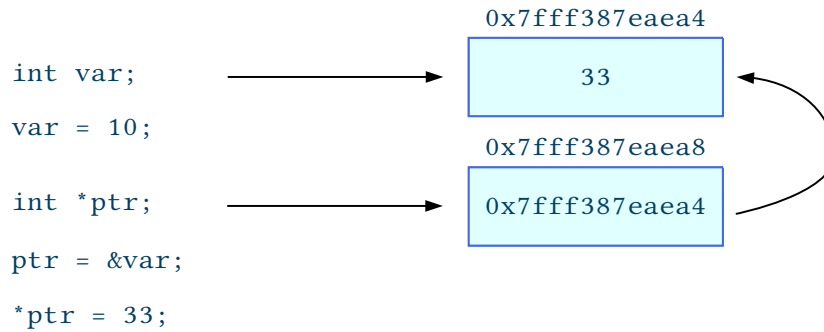


- Given a pointer that contains the address of a data, how can I access that data?
- Using the '\*' operator

```
int a;  
...  
int *x=&a;  
printf("At address %p, the stored int is %d\n", x, *x);
```

18/51

## Pointers: recap



19/51

## Functions: pointers as parameters



- How we can deal with a function that takes pointers as args?
  - We have already been using some of them
    - `scanf()` → `scanf("%d", &a);`
    - We need to pass an address either using `&` or directly a pointer
- How it is defined?
  - `*` is again involved
  - i.e. `strlen(const char *s);`

20/51

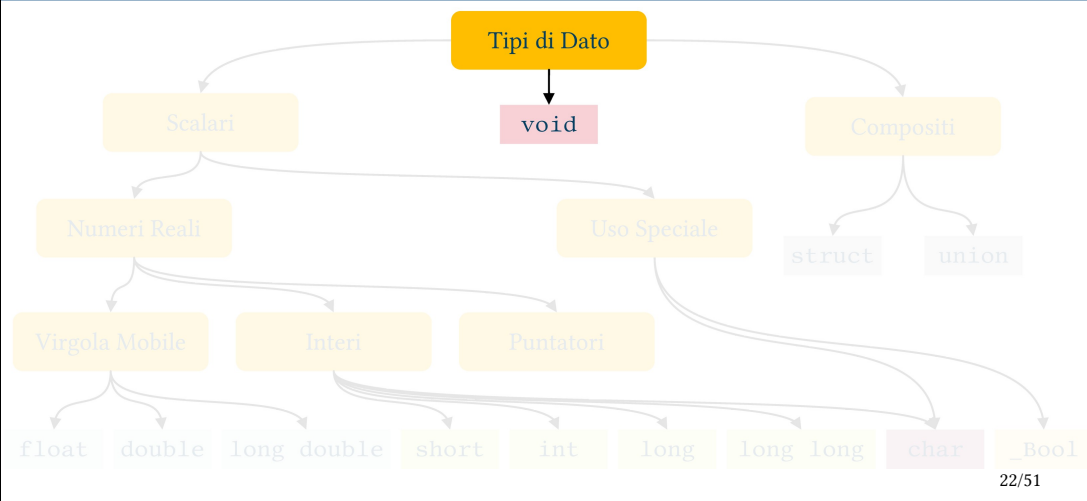
## NULL constant



- By definition an invalid memory address
- Typical uses:
  - Functions returning an address can return NULL when they fails or when they have no result to return
    - i.e. search for a given data
  - To init a pointer variable
  - As argument for some functions

21/51

## C99 Hierarchy of Data Types (simplified)



22/51

## void data type



- Is a *no-type*
- Not possible to define *void* variables
- Used for:
  - Functions
    - No returned values
    - No parameters
  - Pointers
    - Generic type

23/51

## size\_t



- Alias for another data type
- The type capable to store the maximum memory address that we can have to deal with
  - Depends on given architecture
  - Usually in 64 bits systems: *unsigned long long* or *unsigned long*
  - Data type returned by `sizeof()`
  - To print it `%zt` (C99, for our workstations → `%ld`)

24/51



## Recap



- Now we have all elements that allow to understand dynamic memory primitives
  - Pointers and their syntax
    - \* to define
    - \* also to access referenced data
    - How to pass them to functions as arguments
    - How functions can return an address
  - void data type
  - size\_t

25/51

## Dynamic memory allocation



- Main functions:

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

void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```
- Now syntax should be clear!

26/51

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

UNIVERSITÀ  
DI PARMA 

- It allows to allocate a block of memory
  - Input → how many bytes do I need
  - Returned value → Address of allocated memory
    - NULL on error, namely when it is not possible to perform the requested memory allocation
- Memory is not initialized

27/51

## How to use allocated memory

UNIVERSITÀ  
DI PARMA 

- malloc() returns an address → a pointer variable can be used to store it
  - I should check whether NULL has been returned
- When the address is ok, how to access that block of memory?
  - Pointers and arrays share the same syntax!
    - Arrays can be considered as constant pointers
  - **Specifically, I can use [] also with pointers**

28/51

```
void free(void *ptr);
```



- C does not feature a “Garbage Collector”
  - When a dynamically allocated block of memory is no more needed I have to manually “release” it
  - Otherwise that block is considered in use **forever**
- The free() function performs this
- Input → address of a dynamically allocated block of memory
- Output → nothing



29/51

```
void *calloc(size_t nmemb, size_t size);
```



- Same as malloc() but memory is set to 0
  - Input → number of elements and their size
  - Returned value → address of allocated “array”
    - NULL when calloc() fails
    - All elements are zeroed

30/51

**void \*realloc(void \*ptr, size\_t size);**



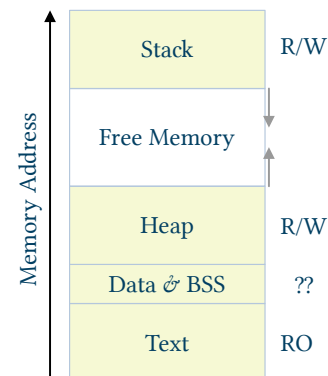
- It allows to change the size of a dynamically allocated block of memory
  - Input → address of a buffer to resize & new size
  - Data are copied into the new block of memory
    - Part of them when reducing the size
  - Returned value → address of the “new” block of memory
    - Or NULL when resize is not possible

31/51

## Memory layout of a C program



- Many sections
  - Stack (.stack) → local variables, VLA
    - Last In First Out approach
  - Heap (.heap) → allocated blocks (malloc(...))
  - Code (.text) → code
  - Data → global or static variables
    - .data & .bss



32/51

## Pointer Arithmetic



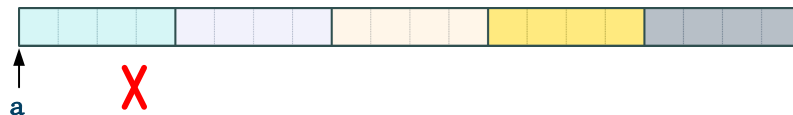
- Formally pointers are a Scalar Type
  - We can use + and – operators and variants (++, --, += ...)
- The result depends on the size of the type of data pointers are referencing
- Incrementing by  $x$ 
  - It does not mean adding  $x$  bytes to the address stored in the pointer
  - It means incrementing the address by  $x$  positions, as going over an array

33/51

## Pointer Arithmetic



- If a pointer is referencing the <type> data type, adding  $x$  means adding  $x$  times the byte size of <type> data type
  - Used only when the pointer is referencing an array
- The  $a[x]$  syntax is basically the same, at low level
  - $a[x] \rightarrow *(a + x)$
  - Example: given a pointer `int *a` we want to add 3 to that pointer (remember that `sizeof(int)==4`)



34/51

## Common Errors

UNIVERSITÀ  
DI PARMA



- Easy to have bugs when dealing with memory addresses!
  - Buffer size
  - Lifetime of variables
  - Loss of addresses



## Buffer Overrun

UNIVERSITÀ  
DI PARMA



- Already discussed for arrays
- Easy to go beyond the range of a block of memory
  - After or even before
- In addition → NULL or not initialized pointer



36/51

## Memory Leak



- malloc()/calloc() allow to reserve a block of memory
  - This enable us to access such area
- And if I “lose” that address?
  - That area can not be used anymore
  - But still “reserved”
- When we have this issue:
  - Assignment errors
  - Automatic variables



37/51

## Lingering Pointer



- Roughly the opposite of the previous case
  - Address is still known
- I do not lose the address but I can no more access that memory area
  - Namely, that memory block is no more “reserved” for us
- Causes
  - Automatic variables
  - Mistakes in using free()



38/51



UNIVERSITÀ DI PARMA

# Pointers to Pointers

39

## Pointers to Pointers

UNIVERSITÀ  
DI PARMA

- Also pointer variables feature a memory address
- Can we store that address?
- Yes → pointers to pointers
- You need to use a double \*\* in definitions
- i.e. `int **p2p; //p2p is a double pointer`
- Also pointer to pointer variables feature a memory address...
  - You can use a lot of '\*'...



40/51



## Pointers to Pointers: syntax

```
int var;           → 0x7fff387eaea4 [57]
var = 10;
int *ptr;          → 0x7fff387eaea8 [0x7fff387eaea4]
ptr = &var;
*ptr = 33;
int **dptr;        → 0x7fff387eae8 [0x7fff387eaea8]
dptr = &ptr;
**dptr = 57; // note the double **
```

41/51



## Pointers to Pointers

- Do you remember the main() function?
  - `int main(int argv, char **argc);`
- Why we need them?
  - When dealing with functions that have side effects, or
  - In order to dynamically allocate blocks of memory
  - Often when managing multidimensional arrays

42/51





UNIVERSITÀ DI PARMA

# Dynamic allocation of multidimensional arrays

44

## Static multidim arrays: recap

UNIVERSITÀ  
DI PARMA

- For sake of simplicity, just consider a bi-dimensional array
- A statically allocated 2D array can be defined as:  
`float b[33][55];`
- To access a specific element, we can use a syntax like:  
`b[17][34] = .... // we MUST use [][] and indexes`
- To access array address  
`printf("the address is %p\n", b) // we have to omit []`

45/51

## Static multidim arrays: recap



- Can we use a single index with a bi-dimensional array?
  - i.e., it is correct to use and what is the meaning of `b[17]` ??
- When we omit some dimension, this always represents an address. Which one?
- We can see `b` as an **array of arrays**, namely `b[17]` is the address of the 18<sup>th</sup> row  
`printf("the address of 18th row is %p\n", b[17])`

46/51

## Dynamically allocated multi-dim arrays



- 6 different possibilities:
  - 1) **1D array with pointer arithmetic**
  - 2) *VLA of pointers*
  - 3) **Double pointer with a `malloc()` for each row**
  - 4) *Double pointer and a single `malloc()`*
  - 5) *A pointer to a VLA*
  - 6) *A pointer to the first row of a VLA*

47/51

## 1D array with pointer arithmetic



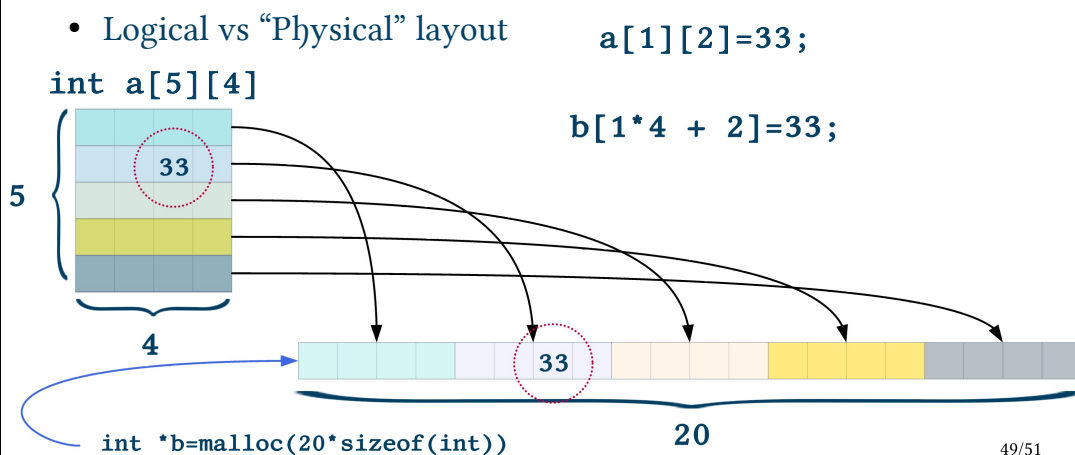
- When we want to manage a matrix with ***r* rows** and ***c* columns**
- Simply allocate a block of ***r\*c* elements**, i.e.,  

```
float *matr = malloc(r*c*sizeof(float));
```
- The element in row ***y*** and column ***x*** can be easily accessed as  

```
matr[x + y*c] = ...
```
- Pros → easy allocation (1 malloc())
- Cons → we need to use a different syntax for access + math

48/51

## 1D array with pointer arithmetic



49/51

## Double pointer with a malloc() for each row



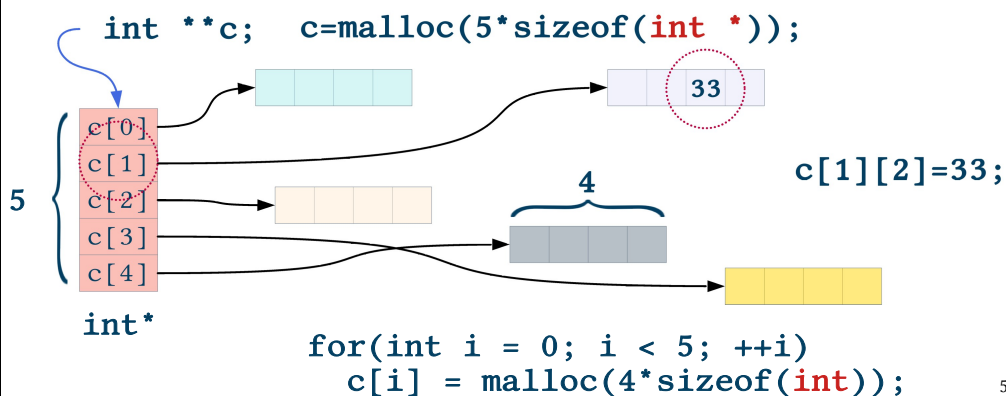
- When we want to manage a matrix with ***r*** rows and ***c*** columns
- A double pointer can be used to allocate *r* “row” pointers, one for each row
- Each row is then independently allocated
- Pros:
  - Same syntax as multidim arrays, i.e., `[][]`
  - Can also manage different sized rows
- Cons: complex memory allocation & destroy

50/51

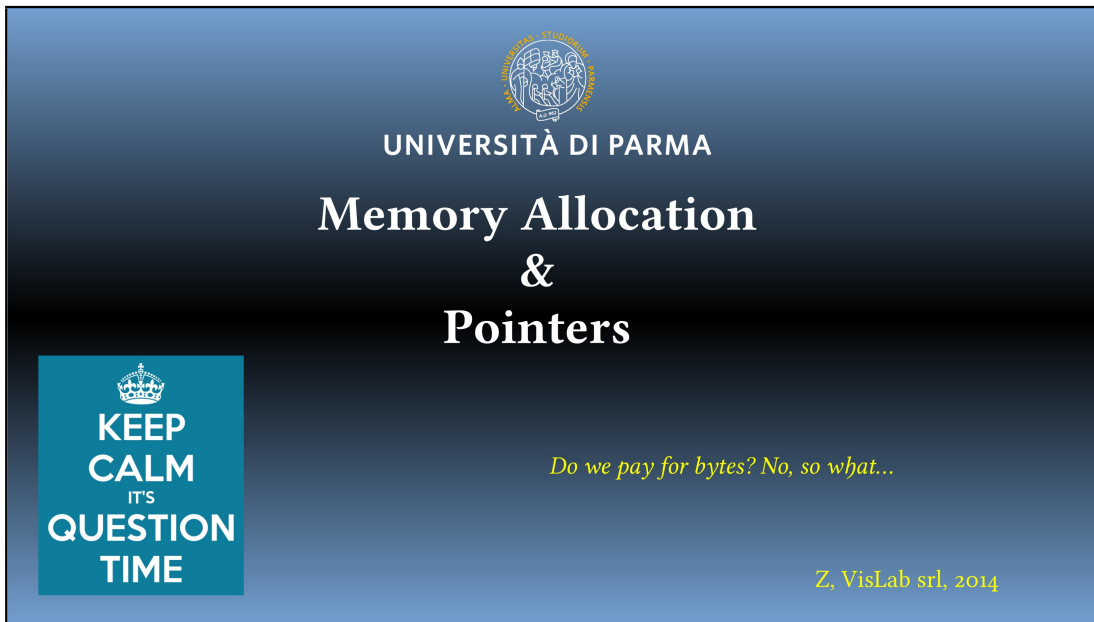
## Double pointer with a malloc() for each row



- As before we want to dynamically allocate a  $5 \times 4$  matrix



51/51





UNIVERSITÀ DI PARMA

# Le Stringhe

*If a string has one end, it has another.*

Gautama Buddha

## Sommario

UNIVERSITÀ  
DI PARMA

- Definizione stringhe
- Memorizzazione
- I/O
- La libreria `string.h`
- Principali operazioni
- Array di stringhe

# SUMMARY



2/21

## Definizione Stringa



- In informatica, si definisce *stringa* una sequenza *ordinata* di simboli
- Per il C
  - Solo codici ASCII → sequenza di dati di tipo char
  - Terminata da carattere nullo ‘\0’
- Abbiamo già visto *stringhe* costanti
  - Tutto ciò che è racchiuso tra “ e ”
  - “” → stringa “vuota”

3/21

## Memorizzazione stringhe



- Come le memorizzo?
- Sono sequenze di char → array di char
- Non vale il contrario
  - Un array di char non è automaticamente una stringa
  - Ci vuole il carattere nullo di terminazione
- L’array è solo un “contenitore” la stringa è il contenuto
  - `char mystring[100] = “Hello World!”;`
- **Attenzione!** Ci vuole spazio per il ‘\0’

4/21



## I/O stringhe



- `printf()` e `scanf()`
  - Specificatore di formato `%s`
    - Per `scanf()` si ferma al primo spazio != **`printf()`**
    - Legge solo “parole”
    - Soluzione → `%[...]` e nello specifico `%[^\n]`
  - destinazione/sorgente ↔ array di char
- Nel caso della lettura lo `'\0'` viene aggiunto automaticamente

5/21

## Strings and functions



- How do I pass a string to a function?
  - It is stored inside an array, we need to pass an address
  - **`char []`** or, rather, **`char *`**
- How a function can return a string?
  - The function have to return an array, namely an address of a memory block
  - Again, **`char *`**

6/21

## La libreria `string.h`



- Il C fornisce numerose funzioni per la manipolazione delle stringhe:
- `#include <string.h>`
- Prendono in ingresso/uscita array di char **MA**
  - Non funzionano se manca il `'\0'`
  - Ovvero gli array devono contenere stringhe

7/21

```
#include <string.h>

int      strcasecmp(const char *s1, const char *s2);
char     *strcat(char *dest, const char *src);
char     *strchr(const char *s, int c);
int      strcmp(const char *s1, const char *s2);
int      strcoll(const char *s1, const char *s2);
char     *strcpy(char *dest, const char *src);
size_t   strcspn(const char *s, const char *reject);
char     *strdup(const char *s);
char     *strfry(char *string);
size_t   strlen(const char *s);
char     *strncat(char *dest, const char *src, size_t n);
int      strncmp(const char *s1, const char *s2, size_t n);
char     *strncpy(char *dest, const char *src, size_t n);
int      strncasecmp(const char *s1, const char *s2, size_t n);
char     *strpbrk(const char *s, const char *accept);
char     *strrchr(const char *s, int c);
char     *strsep(char **stringp, const char *delim);
size_t   strspn(const char *s, const char *accept);
char     *strstr(const char *haystack, const char *needle);
char     *strtok(char *s, const char *delim);
size_t   strxfrm(char *dest, const char *src, size_t n);
char     *index(const char *s, int c);
char     *rindex(const char *s, int c);
```

## Operazioni principali



- Calcolo lunghezza
- Confronto
- Copia/assegnamento
- Accodamento
- Ricerca caratteri e sottostringhe

9/21

## `size_t strlen(const char *s);`



- `strlen(s)`
- Restituisce numero caratteri stringa s
- Non conta carattere nullo di terminazione

10/21

```
int strcmp(const char *s1, const char *s2);
```

UNIVERSITÀ  
DI PARMA 

- `strcmp(s1, s2)`
- Confronta le stringhe `s1` e `s2`
- Restituisce
  - `0` → stringhe uguali carattere per carattere
  - `<0` → `s1` precede `s2` in “ordine ASCII”
  - `>0` → `s1` segue `s2` in “ordine ASCII”

11/21

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

UNIVERSITÀ  
DI PARMA 

- `strcpy(dest, src)`
- Copia la stringa `src` nell'array `dest`
- Non viene fatto controllo sullo spazio della destinazione
- `src` può essere stringa costante

12/21

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



- `strcat(dest, src)`
- Accoda la stringa `src` alla stringa `dest`
  - Nessun controllo sullo spazio disponibile
  - `dest` può essere anche la stringa vuota ma deve essere una stringa!
    - Array di char non inizializzato → errore!

13/21

```
char *strchr(const char *s, int c);  
char *strrchr(const char *s, int c);  
char *strstr(const char *haystack, const char *needle);
```



- `strchr(s, c)` o `strrchr(s, c)`
- Ricerca la prima occorrenza del carattere `c` nella stringa `s`
  - Da sinistra o destra
  - NULL se non lo trova
- `strstr(h, n)`
- Ricerca la prima occorrenza della stringa `n` nella stringa `h`
  - Da sinistra
  - NULL se non esiste

14/21

## `char *strtok(char *str, const char *delim)`



- `strtok()` permette di dividere una stringa in "token"
  - Sottostringhe opportunamente suddivise da delimitatori
- Prende in ingresso la prima volta:
  - Stringa da "spezzettare"
  - Stringa contenente caratteri di delimitazione
- Le volte successive
  - NULL
  - Stringa contenente caratteri di delimitazione
- Restituisce
  - Indirizzo prossimo token individuato oppure NULL se non ce ne sono più

15/21

## Array di stringhe



- La stringhe già sono memorizzate in array
  - Posso usare array di array
- Array bidimensionali
  - Devo dimensionare ogni riga con la lunghezza massima prevista
  - Spreco spazio
- Array di puntatori + allocazione dinamica
  - Più complesso ma ottimizzo uso spazio

16/21

## Arrays of strings



- Recap: for a bi-dimensional array i.e. `char myarr[n][m]`

- `myarr[x][y]` → single char at (x, y) position
- `myarr[r]` →  $r^{\text{th}}$  row address

`char dogs[6][15]`

|                      |   |   |   |   |   |    |   |   |    |    |    |    |   |   |   |
|----------------------|---|---|---|---|---|----|---|---|----|----|----|----|---|---|---|
| <code>dogs[0]</code> | H | u | s | k | y | \0 | ? | ? | ?  | ?  | ?  | ?  | ? | ? | ? |
| <code>dogs[1]</code> | P | o | m | e | r | a  | n | i | a  | \0 | ?  | ?  | ? | ? | ? |
| <code>dogs[2]</code> | L | a | b | r | a | d  | o | r | \0 | ?  | ?  | ?  | ? | ? | ? |
| <code>dogs[3]</code> | B | o | x | e | r | \0 | ? | ? | ?  | ?  | ?  | ?  | ? | ? | ? |
| <code>dogs[4]</code> | J | a | c | k |   | R  | u | s | s  | e  | l  | \0 | ? | ? | ? |
| <code>dogs[5]</code> | B | r | a | c | c | h  | e | t | t  | o  | \0 | ?  | ? | ? | ? |

`dogs[3][4]`

Diagram illustrating the memory layout of a 2D array `dogs` of type `char` with 6 rows and 15 columns. The array is shown as a grid of characters. The first row is `dogs[0]` containing "Husky" followed by null terminator and padding. The second row is `dogs[1]` containing "Pomerania". The third row is `dogs[2]` containing "Labrador". The fourth row is `dogs[3]` containing "Boxer", with the character 'e' at index 4 circled in red. The fifth row is `dogs[4]` containing "Jack Russell". The sixth row is `dogs[5]` containing "Braccahettot". A red dotted arrow points from the text `dogs[3][4]` to the circled 'e' character. A black curved arrow points from the text `char dogs[6][15]` to the first row of the grid.

17/21

## Arrays of strings



- Therefore, using a single index → address of a "string"
- This can be easily used for all functions that involve strings
- As examples:

```
printf("The dog in the 4th row is %s\n", dogs[3]);
scanf("%[^\\n]", dogs[5]);
strcpy(dogs[0], dogs[3]);
...
```

18/21

## Arrays of strings

- Use of bi-dimensional arrays
  - Pro: somewhat easier than alternatives
  - Cons: not memory efficient or flexible

char dogs[6][15]

In this example  
around 40% space wasted

|         |   |   |   |   |   |    |   |   |    |    |    |    |   |   |   |
|---------|---|---|---|---|---|----|---|---|----|----|----|----|---|---|---|
| dogs[0] | H | u | s | k | y | \0 | ? | ? | ?  | ?  | ?  | ?  | ? | ? | ? |
| dogs[1] | P | o | m | e | r | a  | n | i | a  | \0 | ?  | ?  | ? | ? | ? |
| dogs[2] | L | a | b | r | a | d  | o | r | \0 | ?  | ?  | ?  | ? | ? | ? |
| dogs[3] | B | o | x | e | r | \0 | ? | ? | ?  | ?  | ?  | ?  | ? | ? | ? |
| dogs[4] | J | a | c | k |   | R  | u | s | s  | e  | l  | \0 | ? | ? | ? |
| dogs[5] | B | r | a | c | c | h  | e | t | t  | o  | \0 | ?  | ? | ? | ? |

19/21

## Arrays of strings

- The use of array of pointers is a little more complex
  - But much more memory efficient and flexible
  - Same syntax anyway!

char \*\*dogs

strcpy(dogs[3], dogs[0]);

char \*

dogs[0]

dogs[1]

dogs[2]

dogs[3]

dogs[4]

dogs[5]

H u s k y \0

P o m e r a n i a \0

L a b r a d o r \0

B o x e r \0

J a c k R u s s e l \0

B r a c c h e t t o \0

dogs[3][4]







UNIVERSITÀ DI PARMA

# Le Funzioni

*E Pluribus Unum*

Autore Ignoto, Moretum, I sec. a.C.

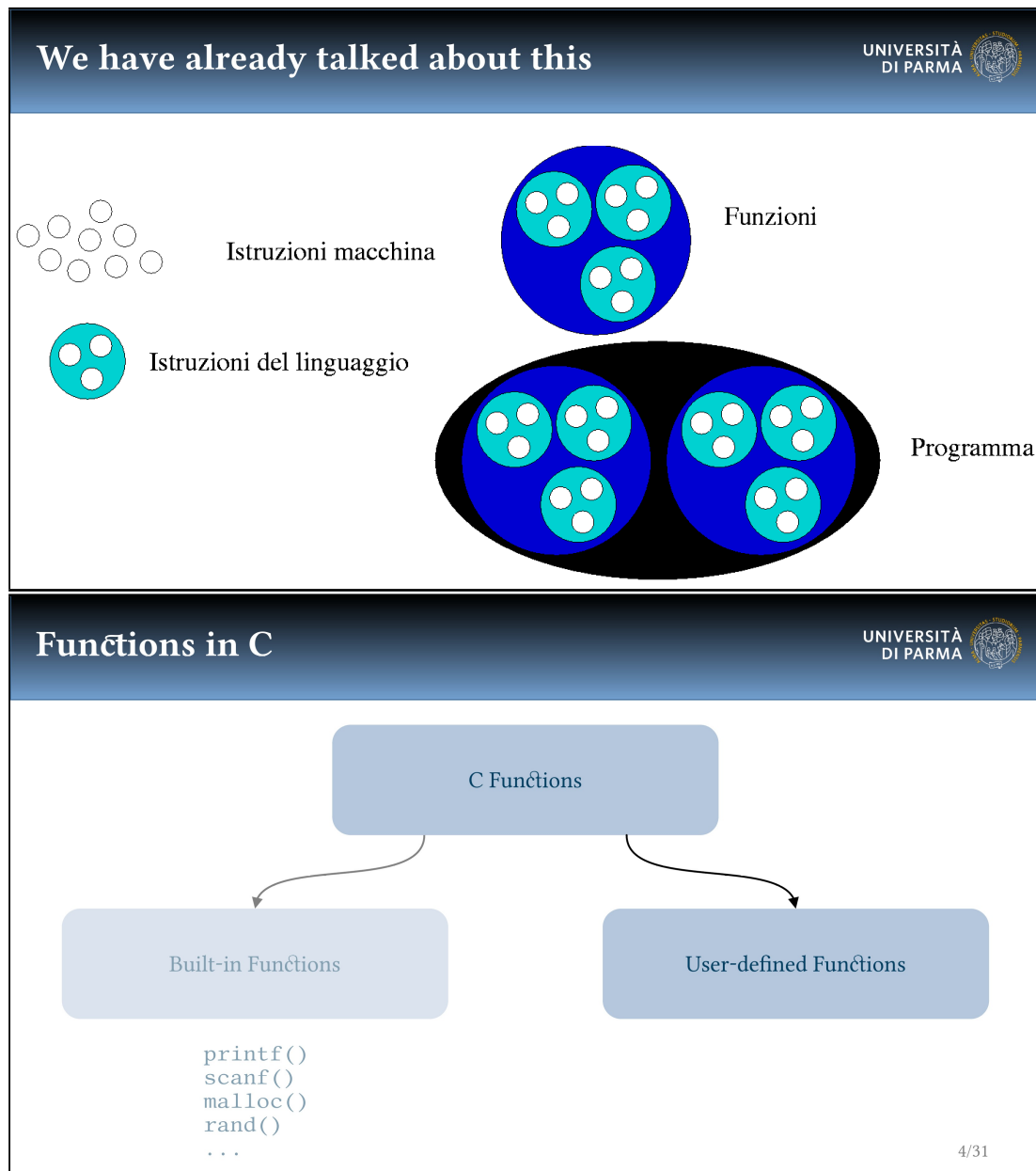
## Sommario

UNIVERSITÀ DI PARMA

- Definire una funzione
  - Tipo restituito
  - Parametri Formali
  - Passaggio dei dati
  - I prototipi
- La Ricorsione



2/31



## Defining a new function



- Already done → main()
- We have to write our function signature + body

Returned data type  
(int, float, char...)  
 ↓  
 <return type>  
 {  
 <function body>  
 }

Name of function  
 ↓  
 <function name>

Formal parameters list  
 ↓  
 (<formal param1>, ...)  
 {  
 <function body>  
 }

Body of function  
(variables and/or statements)

5/31

## Definire una nuova funzione



- Tipo di dato restituito
  - Quelli già visti... e quelli che vedremo!
  - Nel caso di puntatori → \*
  - E se non devo restituire nessun dato?
    - void
- Nome
  - Regole già viste per le variabili
  - Opportunamente mnemonico!

6/31

## Definire una nuova funzione



- Elenco parametri formali
  - Lista separata da “,”
  - Specificare tipo e nome
  - Se non necessari → void

7/31

## Definire una nuova funzione



- Corpo funzione
  - Racchiuso tra { }
  - Codice che realizza funzione
  - Può essere anche vuoto ({} still needed!)
    - *Stub*
    - Utile in fasi iniziali sviluppo
  - Se restituisco qualcosa obbligo **return**

8/31

## Prototipo di una funzione



- Il compilatore quando incontra una chiamata a funzione deve sapere di che funzione si tratta
  - Il codice viene letto dall'alto in basso
  - Non sempre possibile definire una funzione prima di una sua chiamata
- Soluzione → riga di definizione della funzione (prototipo)

9/31

## Chiamare una funzione



- L'invocazione di una funzione definita da noi segue le stesse regole già viste per le funzioni predefinite
  - Devo passare un numero e un tipo di argomenti coerenti con i parametri della funzione
    - Attenzione alle conversioni di tipo!

10/31

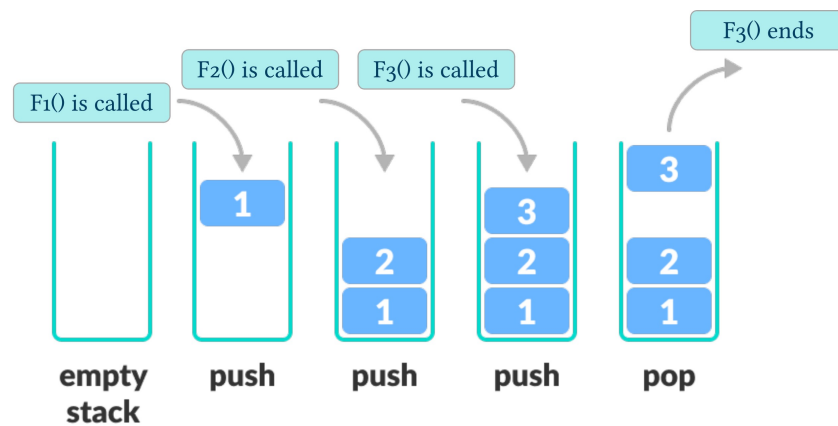
## Invoking a function → memory



- When we invoke a function a specific memory area is involved
  - Stack
- And it contains all data needed to execute the function
- **Activation record:**
  - Parameters of the callee
  - Function arguments
  - Local variables
  - Return address of the function itself

11/31

## Invoking a function → memory



12/31

## Parametri e Argomenti



- Quando invoco una funzione → argomenti
  - O anche “parametri attuali” o “effettivi”
  - Valori passati ad una funzione
- Nella definizione di una funzione → parametri formali
  - Di fatto variabili locali alla funzione
  - A loro viene assegnato il valore degli argomenti

13/31

## Parametri e Argomenti



- Passaggio argomenti, due modalità
  - Per valore
  - Per indirizzo

14/31



## Parametri e Argomenti



- Passaggio argomenti, per valore
  - È il default per tutti i dati ad eccezione degli array
  - Viene creata una copia dell'argomento
  - La funzione quindi lavora su copie, non sul dato originale
  - Impossibile agire su variabili passate alla funzione

15/31

## Parametri e Argomenti



- Passaggio argomenti per indirizzo
  - È il default per gli array
  - Viene passato alla funzione l'indirizzo del dato
  - Attenzione al tipo di dato!
  - La funzione può modificare il dato originale
    - Effetti collaterali

16/31

## Pass arrays to functions



- When passing arrays we could have to deal with size
  - For strings maybe it is not necessary
- Three different approaches depending on situation
  - Fixed size array → size is fully specified
  - VLA → size is given as parameter
  - As pointer → similar to previous case

17/31

## return



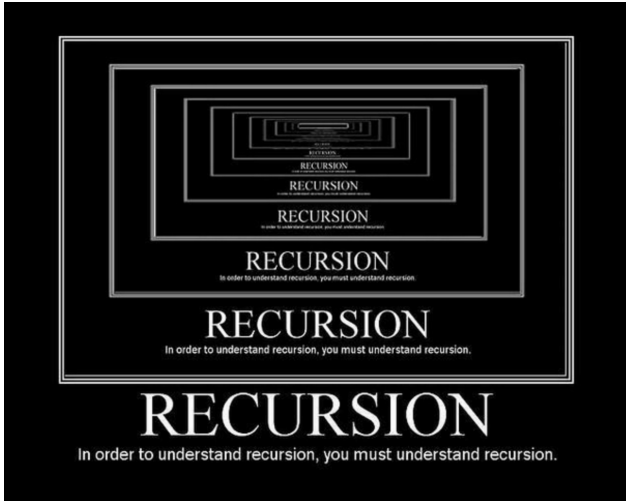
- Permette
  - Uscita funzione e
  - Restituzione valore per funzioni non void
- Non obbligatorio per funzioni void
- Può restituire uno e un solo valore

**return (<espressione>);**

18/31

# La Ricorsione





19/31

# La Ricorsione



- Approccio molto elegante e semplice che permette di risolvere problemi anche complessi
- La ricorsione si può applicare a tutti quei problemi in cui la risoluzione dipende dalla risoluzione dello stesso problema però di ordine di grandezza inferiore ovvero più semplice
- Esempio: il fattoriale

$$n! = n \cdot (n-1)!$$

20/31

## La Ricorsione



- Esempio: fattoriale
  - Il fattoriale di  $n$  lo posso calcolare come funzione del fattoriale di  $n-1$
  - Esistono inoltre casi specifici

$$n! = \begin{cases} 1 & \text{per } n=0 \\ n \cdot (n-1)! & \text{per } n>1 \end{cases}$$

21/31

## La Ricorsione



- Una soluzione ricorsiva di un problema è facilmente trasponibile in codice
- **Funzione che chiama se stessa!**

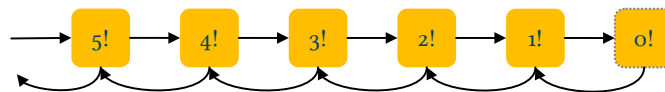
|                                                                                               |  |                                                                                                             |
|-----------------------------------------------------------------------------------------------|--|-------------------------------------------------------------------------------------------------------------|
| $n! = \begin{cases} 1 & \text{per } n=0 \\ n \cdot (n-1)! & \text{per } n \geq 1 \end{cases}$ |  | <pre>unsigned long long fact(unsigned long long n) {     if(n==0) return 1;     return n*fact(n-1); }</pre> |
|                                                                                               |  |                                                                                                             |

22/31

## La Ricorsione



- Chiamate ricorsive implicano avere tante “istanze” della stessa funzione
  - Ognuna con le sue variabili locali

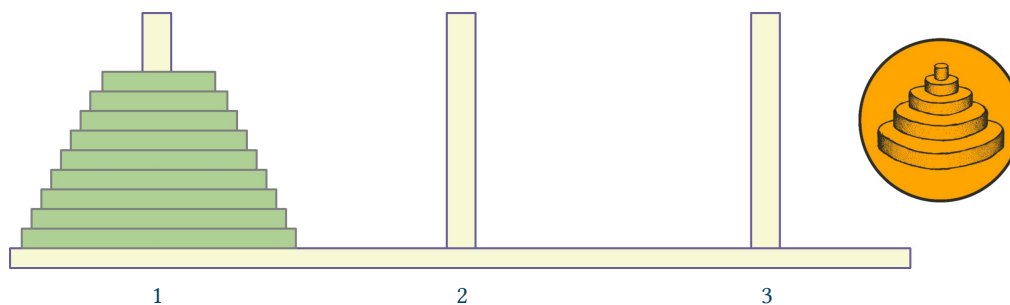


23/31

## La Ricorsione



- Torre di Hanoi



**Obiettivo:** spostare  $n$  dischi dal piolo 1 al piolo 3

24/31

## La Ricorsione



- Torre di Hanoi, definiamo
  - $T(n,x,y)$  → sposta  $n$  dischi da piolo  $x$  a piolo  $y$
  - $D(x,y)$  → sposta un singolo disco da piolo  $x$  a piolo  $y$

$$T(n,1,3) = \begin{cases} \text{per } n=0 & \text{niente} \\ \text{per } n>1 & T(n-1,1,2) + D(1,3) + T(n-1,2,3) \end{cases}$$

25/31

## La Ricorsione



- Errori Comuni
  - Mancanza condizione di uscita
  - La semplicità può mascherare inefficienza



26/31

## La Ricorsione

- Numeri di Fibonacci
  - 0, 1, 1, 2, 3, 5, 8, 13, 21,...



$$f(n) = \begin{cases} \text{per } n=0 & 0 \\ \text{per } n=1 & 1 \\ \text{per } n>1 & f(n-1) + f(n-2) \end{cases}$$

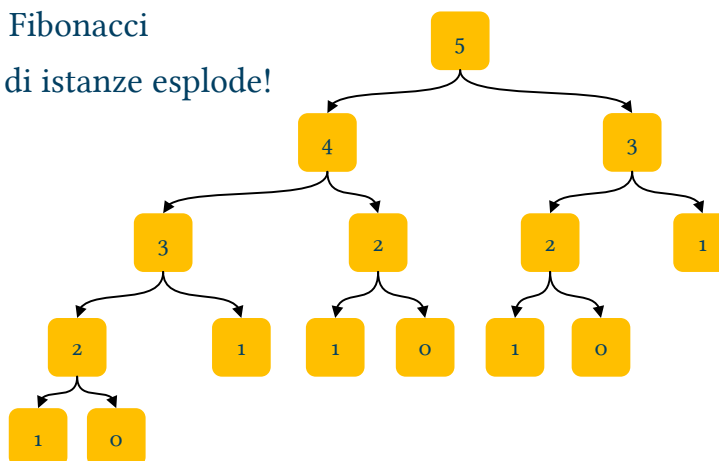


```
unsigned long fibo(unsigned long n){
    if(n==0) return 0;
    if(n==1) return 1;
    return fibo(n-1)+fibo(n-2);
}
```

27/31

## La Ricorsione

- Numeri di Fibonacci
- Il numero di istanze esplode!



28/31

## La Ricorsione



- L'implementazione ricorsiva vista in precedenza non è efficiente
- Si può dimostrare che se  $S$  è il tempo necessario al calcolo del numero di fibonacci di indice  $n \rightarrow f(n)$ , il calcolo di  $f(n+1)$  richiede circa  $1,6 \times S$
- Ad esempio se il calcolo di  $f(n)$  richiede 1 s, il calcolo di  $f(n+18)$  richiederà un'ora
- Soluzione  $\rightarrow$  dynamic programming (vedi esempio)

29/31

## Ricorsione vs Iterazione



- Entrambe sono di fatto ripetizioni
  - Ciclo esplicito vs Ciclo implicito
  - Tutto ciò che posso risolvere con la ricorsione è risolvibile iterativamente
  - Entrambe vogliono condizione di uscita
    - Rischio ciclo infinito
- Ricorsione  $\rightarrow$  codice più semplice
  - Spesso più facile e codice più compatto
  - Contro: overhead chiamate di funzione

30/31







UNIVERSITÀ DI PARMA

# Input & Output

*The future belongs to those who believe in the beauty of their streams.*

Eleanor Roosevelt (mod.)

## Sommario

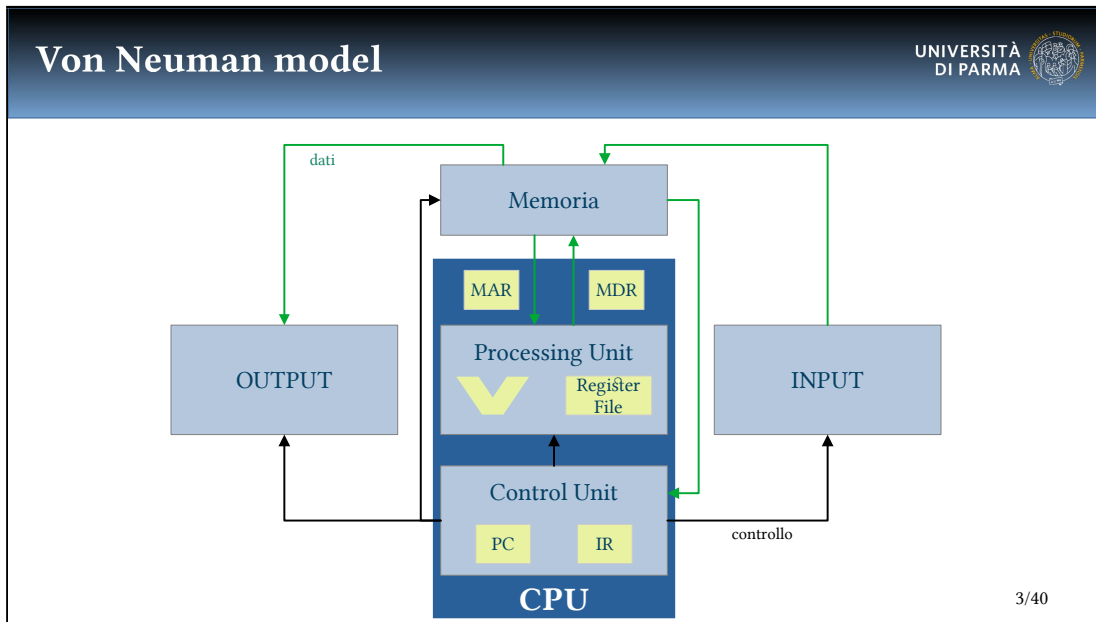
UNIVERSITÀ  
DI PARMA

- Non esaustivo → Manuale
- gli stream
- concetto di file
- principali operazioni
  - definizione
  - associare uno stream
  - lettura/scrittura e altre operazioni
    - File testuali e binari
  - chiusura
- I/O della console

# SUMMARY



2/40



## What is a stream?

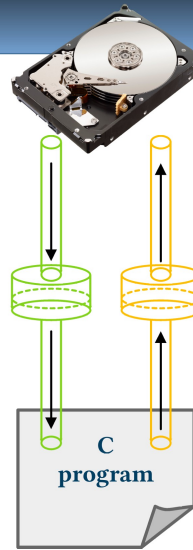
- In C sono i “canali” di input o output
  - Già visto: tastiera e video
  - Ma ne possiamo avere altri
  - File, porte seriali, USB, ecc.
- Predefiniti
  - `stdin` → `scanf()`, ...
  - `stdout` → `printf()`, ...
  - `stderr`
- Vantaggio: un solo set di primitive per la gestione

4/40

## What is a stream?

UNIVERSITÀ  
DI PARMA

- Gli stream sono una struttura ad alto livello che ci separa dall'HW
- Tipicamente I/O asincrono
  - Bufferizzato
- Input, output o entrambi



5/40

## Main operations

UNIVERSITÀ  
DI PARMA

- Definire uno stream
- Associare uno stream (ovvero “aprire”)
- Lettura o scrittura dati
  - Altre operazioni
- Chiusura

6/40

## Main operations



- Definire uno stream
- Associare uno stream (ovvero “aprire”)
- Lettura o scrittura dati
  - Altre operazioni
- Chiusura

7/40

## Definire uno stream



- Si usa la struttura FILE
- Deve sempre essere un puntatore

```
FILE *miostream;  
FILE *altrostream, *altroancora;
```

8/40

## Main operations



- Definire uno stream
- Associare uno stream (ovvero “aprire”)
- Lettura o scrittura dati
  - Altre operazioni
- Chiusura

9/40

## Associare uno stream



- Dopo averlo definito possiamo “associare” uno stream
  - File
  - Periferica
  - ...
- Noi vedremo solo file

```
FILE *fopen(const char *pathname, const char *mode);
```

10/40

## Cosa è un file?

- In italiano “archivio”
- Principale contenitore informazioni su supporti digitali
  - Hard disk
  - DVD/CD/BD
  - Memory stick
  - ...
- **Permanente**



11/40

## Come si usa un file?

- Nastro magnetico
  - Testina che legge/scrive
  - Man mano che legge/scrive si sposta
  - Ma posso anche spostarmi in un punto specifico del nastro
- File
  - Concetto di “testina virtuale” ovvero posizione di lettura/scrittura
  - Ad ogni operazione “avanza” nel file
    - Di tante “posizioni” quanti sono i byte letti o scritti
  - Possibile saltare in posizioni specifiche

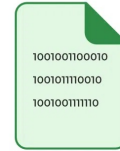


12/40

## Tipi di file

UNIVERSITÀ  
DI PARMA

- Testuali o ASCII
  - Solo caratteri stampabili
    - Leggibili da essere umano
  - Esempi: codice sorgente C, file di configurazione (.ini),...
  - Massima portabilità ↔ spazio occupato
- File binari
  - Anche caratteri non stampabili
  - Più compatti ↔ portabilità



file.bin



file.txt

13/40

## fopen()

UNIVERSITÀ  
DI PARMA

- Permette apertura file
- Due argomenti, stringhe:
  - Nome file (percorso relativo o assoluto), solo caratteri ASCII
  - Modalità apertura
- Restituisce
  - NULL se fallisce
  - Puntatore a “tipo” FILE se successo

```
FILE *fopen(const char *pathname, const char *mode);
```

14/40



## fopen()

|           |                                           |                                                               |
|-----------|-------------------------------------------|---------------------------------------------------------------|
| <b>r</b>  | lettura                                   | Il file deve esistere                                         |
| <b>w</b>  | scrittura                                 | Il file viene creato se non esistente o azzerato se esistente |
| <b>a</b>  | scrittura a fine file                     | Il file viene creato se non esistente                         |
| <b>r+</b> | lettura e scrittura da inizio             | Il file deve esistere                                         |
| <b>w+</b> | azzeramento/creazione e lettura/scrittura | Il file viene creato se non esistente o azzerato se esistente |
| <b>a+</b> | lettura e scrittura a fine file           | Il file viene creato se non esistente                         |
| <b>b</b>  | In aggiunta ai precedenti se file binario | Esempi: "rb", "w+b", "wb+"...                                 |

15/40

## fopen()

- L'operazione di apertura può fallire:
  - File non esistente se richiesta lettura
  - Permessi
  - Errori di percorso...
- Fondamentale controllare quanto restituito da `fopen()`
- Utile l'uso di `void perror(const char *s)`

16/40

## FILE\*



- When opening a file we need a **file pointer** (FILE \*), i.e.
  - `FILE *fp = fopen("mydata.dat", "rb");`
- The address inside the pointer is **not related** to the position of read or write operations within the file
- This pointer is used to handle all subsequent file operations
  - At that address the system stores all the data needed to deal with the file
  - Do not mess with it!

17/40

## Main operations



- Definire uno stream
- Associare uno stream (ovvero “aprire”)
- Lettura o scrittura dati
  - Altre operazioni
- Chiusura

18/40

## Read/write data with files



- ASCII files, two main approaches
  - Raw I/O: read/write bytes (chars), strings...
  - Formatted I/O: thanks to a format string we can extract significant data
- Binary files
  - Raw I/O: read/write chunks of “memory” namely arrays or even single variables

19/40

## Read/write primitives for ASCII files



- Single chars/bytes:
  - (f)getc() read a char from a stream
  - (f)putc() write a char to a stream
  - ungetc() put back a char in a stream
- Strings/Lines:
  - fgets() read an entire line as string from a stream up to a given size
  - fputs() write a string to a stream
- Formatted I/O:
  - fscanf() formatted read from a stream
  - fprintf() formatted write to a stream

20/40

## Byte by byte read/write



- `int fgetc(FILE *stream);`
  - Reads a single char from a `stream`
  - Returns EOF on End Of File or error
    - This can help us to understand when the end of file is reached
- `int fputc(int c, FILE *stream);`
  - Write a single char on a file
  - Returns EOF on error
- `getc()` and `putc()` are aliases

21/40

## `int ungetc(int c, FILE *stream);`



- Can be used to push back a char to a `stream`
- Namely, `c` is put as the first “byte” to be read next time
- Only one push back is guaranteed
- What is the reason why to put back a char?
  - Mixing `getc()/ungetc()` allows you to peek at the next character without consuming it.

22/40

## Line by line read/write



- `char *fgets(char *s, int size, FILE *stream)`
  - Reads line from a stream up to size-1 chars or up to the first '\n'
  - Put it inside the array s[] adding the terminating null byte '\0'
  - Returns NULL on end of file or error
    - This can help us to understand when the end of file is reached
- `int fputs(const char *s, FILE *stream);`
  - Write the string s on a file without the '\0'
  - Returns EOF on error

23/40

## Formatted read/write



- `int fprintf(FILE *stream, const char *format, ...);`
  - Like printf() is a formatted “print” on a file
  - Returned value can be ignored
- `int fscanf(FILE *stream, const char *format, ...);`
  - Can be used to read from text files
  - Format string can be exploited to “extract” relevant data
  - Returns the number of format specifier that have been “solved”
    - This can help us to understand when the end of file is reached

24/40

## Lettura/scrittura file ASCII



- I/O formattato vs non formattato
  - `fgetc()/fputc()` e `fgets()/fputs()` vs `fprintf()` e `fscanf()`
  - Preferire lettura formattata quando contenuto va interpretato
    - Errore comune: leggo stringa e poi la cerco di interpretare in memoria
- Le operazioni di I/O possono essere collo di bottiglia
  - Lettura/scrittura di singoli byte meno efficiente

25/40

## Lettura/scrittura file binari



- Scrivo/leggo un intero blocco di memoria
  - Indirizzo blocco
  - Dimensione singoli elementi
  - Numero elementi
  - Stream

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

26/40

## I/O with binary files



- `fread()/fwrite()` can be used to read/write **arrays**
  - i.e. given an array like **float data[127]** we can read data from a file using a single statement:
  - `fread(data, sizeof(float), 127, filehandler);`

Diagram illustrating the parameters of `fread()` for reading an array:

- `data`: Address to store data read from file
- `sizeof(float)`: Size of each element of the array
- `127`: Number of elements in the array
- `filehandler`: FILE \* should be opened before read/write ops

27/40

## I/O with binary files



- `fread()/fwrite()` can be also used to read/write **single variables**
  - i.e. given a “simple” variable like **long len**, we can use it to store data read from a file using a single statement (like a *single element array*)
  - `fread(&len, sizeof(long), 1, filehandler);`

Diagram illustrating the parameters of `fread()` for reading a single variable:

- `&len`: Address to store data read from file. We need to use the ‘&’
- `sizeof(long)`: Size of the “element”
- `1`: We have only one “element”
- `filehandler`: FILE \* should be opened before read/write ops

28/40

## End of file

- Some **read** operations **silently fails** when we try to read data beyond the end of a file
- Then, how to understand when we reached the end of file?
- 2 approaches:
  - Specific function `feof()`
  - Use the return value of functions used to read from file

29/40

## End of file

- The `int feof(FILE *stream);` function returns true when a read operation failed because we reached the end of the file
  - Namely it works only when we **already tried to read** data!
  - Difficult to be directly used as a cycle condition
    - We usually need to:
      - Try to read data
      - Use `feof()` to check whether the read operations succeeded
      - If not, stop read operations, i.e., exit the read cycle using a break

30/40



## End of file



- In order to check whether we are at the end of the file, it is much **more convenient** to use the return value of read functions:
  - fscanf() → number of format specifier that has been solved
  - fread() → number of bytes read from the file
  - fgets() → NULL when we can not read from the file
  - fgetc() → EOF constant when we can not read from the file
- Easy to directly use them as cycle conditions

**Follow this approach with respect to the use of feof()!**

31/40

## Main operations



- Definire uno stream
- Associare uno stream (ovvero “aprire”)
- Lettura o scrittura dati
  - Altre operazioni
- Chiusura

32/40

## Other operations



- As previously said each read/write operation move the position for the next read/write operation, like a “virtual recording head”
- Often we need to move such position for different reasons, i.e.:
  - Skip part of a file, go to a specific position in a file (random access), restart from the beginning...
- Primitives:
  - `int fseek(FILE *stream, long offset, int whence);`
  - `void rewind(FILE *stream);`
  - `long ftell(FILE *stream);`

33/40

## Other operations



- **`int fseek(FILE *stream, long offset, int whence);`**
  - Move the “recording head” to a specific position
  - Offset is the relative position ( $\pm$ ) wrt whence
  - Whence can assume the following values: `SEEK_CUR` (current position), `SEEK_SET` (beginning of the file), `SEEK_END` (end of the file)
- **`void rewind(FILE *stream);`**
  - Alias for `fseek(stream, 0, SEEK_SET)`
- **`long ftell(FILE *stream);`**
  - Returns the read/write position in a file

34/40

## Main operations



- Definire uno stream
- Associare uno stream (ovvero “aprire”)
- Lettura o scrittura dati
  - Altre operazioni
- Chiusura

35/40

## Chiusura di un file



- Buona norma chiudere file quando non più necessario
  - Massimo numero file aperti contemporaneamente
  - Svuoto buffer
- Disassocia stream
- Puntatore riutilizzabile

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

36/40

## Console I/O



- Come accennato anche la console (tastiera + video) è associata a stream
  - `stdin` → stream di input (tastiera)
  - `stdout` → stream di output (video)
  - `stderr` → stream di output (messaggi di errore)
- Possibile uso delle funzioni di I/O
  - Ma esistono alias di molte
    - `fprintf(stdin, ...` → `printf(...`

37/40

## Console I/O



- Caratteri
  - `fgetc()` → `int getchar(void);`
  - `fputc()` → `int putchar(int c);`
- Stringhe
  - `fputs()` → `int puts(const char *s);`
  - `fgets()` → `char*gets(char*s); //DEPRECATED`

38/40

## Bufferizzazione



- La libreria del C *bufferizza* l'accesso ai flussi di I/O
  - Ciò che scrivo da tastiera finisce in un'area di memoria che sarà man mano "consumata" dalle operazioni di lettura
  - Tipicamente il buffer viene riempito alla pressione del tasto di invio
  - Va posta attenzione alla gestione dell'input da console

39/40



UNIVERSITÀ DI PARMA

## Input & Output

*Don't cross the streams.*

Egon, 1984



UNIVERSITÀ DI PARMA

# Tipi di dato in C struct, union & enum

*Art and Science cannot exist but in minutely  
organized particulars*

William Blake, To the Public

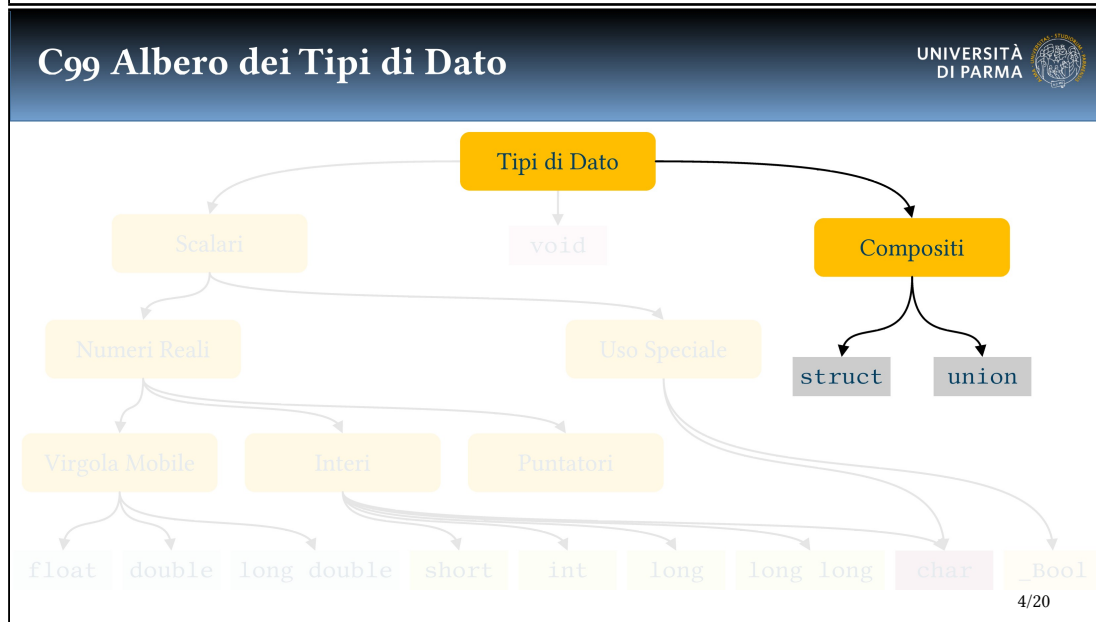
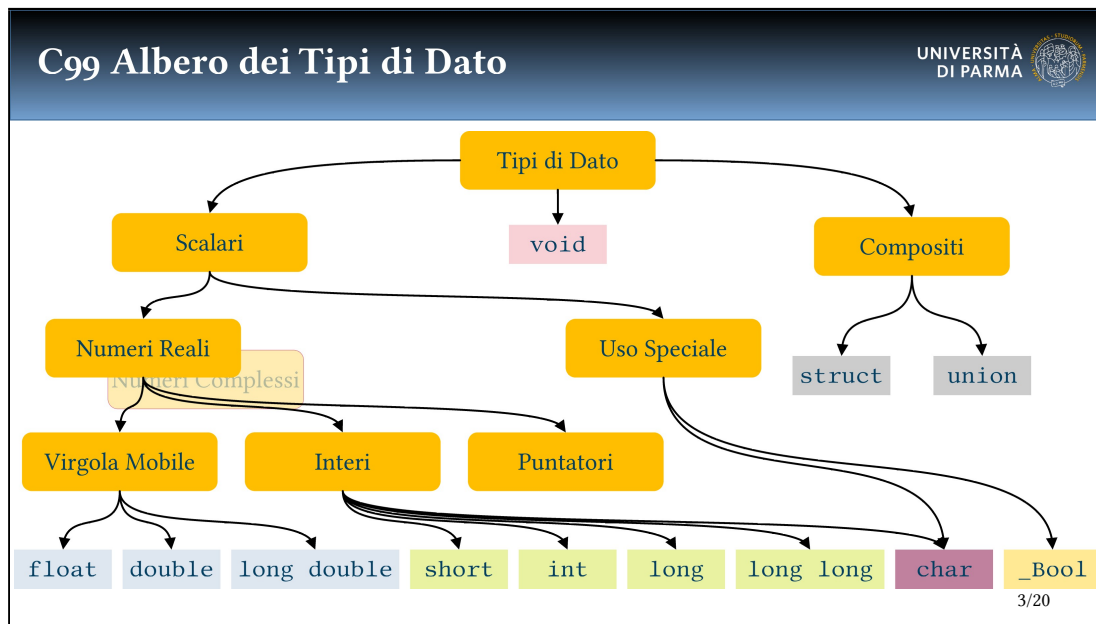
## Summary

UNIVERSITÀ  
DI PARMA

- Struct
- Union
- Enum



2/20



## Why aggregate data?



- Up to today we can deal with:
  - Numbers (different types, ranges, and precision)
  - Characters and strings
- To model real world I need a little more...
  - Taking as example a student registry, I have to deal with:
    - Name & Surname → strings
    - University ID → Integer number
    - ...
    - I should use multiple and independent variables that pertain to the same logical object

5/20

## struct



- Il costrutto `struct` permette di
  - Aggregare dati anche di tipo differente
  - Definire nuove tipologie di dato
- Esempio:

```
struct studente{
    char nome[100], cognome[100];
    unsigned long matricola;
};
```

6/20



## struct



- In questo esempio sto aggregando dati di tipo differente
- 2 array di char e un dato di tipo numerico intero
  - I singoli componenti si chiamano “elementi” o “membri”

```
struct studente{  
    char nome[100], cognome[100];  
    unsigned long matricola;  
};
```

7/20

## struct



- La “**struct studente**” è un nuovo tipo di dato
  - NON è una variabile
- Posso usarla per definire variabili

```
struct studente{  
    char nome[100], cognome[100];  
    unsigned long matricola;  
};  
  
struct studente a, b, *c;
```

8/20

## struct



- Usually, the definition of `struct` constructs like “**struct student**” is placed at the beginning of the code just after **#include** directives
- Conversely the definition of variables can be placed where necessary
  - Inside functions, as function parameter... everywhere a variable can be defined

9/20

## struct



- Come accedo ai singoli elementi?
  - `.` per le variabili
  - `->` per i puntatori

```
struct studente a, b, *c;
```

```
a.matricola = 12345;  
c->matricola = 67890;
```

10/20

## struct



- È possibile inizializzare le variabili di tipo “struct”?
- Approccio simile agli array
- Devo mantenere ordine elementi
- Inizializzazioni parziali → 0

```
struct studente a={"Gino", "Pilotino", 12345};
```

11/20

## Arrays of structs



- The “struct” keyword is, in fact, used to define a new data type
  - Like “int”, “float”, “char”...
- Therefore also arrays of structs are possible
  - Elements are accessed using the usual notation ([]) just before the dot . operator

```
struct studente a, b, *c, d[100];  
c[12].matricola = 67890;
```

12/20

## Strutture nidificate



- Le struct possono contenere altre struct

```
struct anagrafica{
    char nome [100], cognome[100];
    int giorno, anno, mese;
};

struct studente{
    struct anagrafica dati;
    char matricola[12];
};
```

14/20

## Strutture nidificate



- Particolare importanza rivestono le struct che contengono puntatori allo stesso tipo di dato
  - Liste, alberi, grafi ecc.
  - Non le vedremo

```
struct nodo{
    char nome [100], cognome[100];
    struct nodo *successivo;
};
```

15/20

## Bit Fields



- Structs are also used to “pack” data exploiting “Bit Fields”
- Namely, we can set the number of bits for each struct member
- Use the same unsigned type for each field
- Reason:
  - Efficient memory use
  - I/O with binary files

17/20

## Bit Fields



- Usually unsigned types are used
- Syntax:

```
struct printer {  
    unsigned short status : 2;  
    unsigned short data   : 8;  
    unsigned short paper  : 2;  
    unsigned short error  : 4;  
};
```

18/20

## Read the fine print



- Data structures are not necessarily “struct”
- Therefore when you are asked to define a data structure
  - Not necessarily this means you have to use “struct”
- A “struct” variable is a “data structure” but not all “data structures” can be obtained using “struct”

19/20

## union



- Stessa sintassi struct
- Ma: elementi tutti sovrapposti in memoria
  - Sistemi embedded
  - Gestione HW
  - Flessibilità

```
union nodo{  
    int    intero;  
    double floating;  
} mixed_array[100];
```

20/20

## enum



- Costrutto per definire tipicamente elenchi di valori costanti
  - Di fatto di tipo “int”
- Spesso usato come argomento funzioni

```
enum stagioni{primavera, estate, autunno, inverno};
```

```
enum stagioni a;
```

```
a = primavera;
```

21/20



UNIVERSITÀ DI PARMA

## Tipi di dato in C struct, union & enum



*Art and Science cannot exist but in minutely  
organized particulars*

William Blake, To the Public



UNIVERSITÀ DI PARMA

# Function Pointers

*Memoria est thesaurus omnium rerum et  
custos*

De Oratore, Cicerone, I sec a.C.

## Sommario

UNIVERSITÀ  
DI PARMA

- Puntatori a funzione
  - Sintassi
  - Definizione
  - Uso
- Funzioni
  - `qsort()`
  - `bsearch()`

# SUMMARY



2/18



## Memoria



- Abbiamo definito le variabili come area di memoria con nome
  - Quindi hanno un indirizzo
- Ma tutto è in memoria
- Anche il codice
  - Quali porzioni di codice hanno un nome?
  - Le funzioni!

3/18

## Indirizzo di una funzione



- Come lo ricavo?
- Nome della funzione senza ( )
  - Come per array, no operatore &

4/18

## Puntatore a funzione

- Per le variabili:
  - Indirizzo
  - Tipo di dato puntato
- Per le funzioni
  - Indirizzo
  - Tipo di dato restituito
  - Elenco e tipi parametri formali

5/18

## Puntatore a funzione

- Sintassi sempre basata su uso <sup>\*</sup>
  - E con ( )

Function return type      Pointer name      Function param types

↓                      ↓                      ↓

`<return type>`      `(*<pointer name>)(<param1 type>, <param2 type>, ...);`

`char *(*pippo)(float, int);`

6/18

## Puntatore a funzione



- Sintassi sempre basata su uso \*
  - E con ()
- Esempi di puntatori:
  - `int (*a)(void);` // a funzione che restituisce int e non vuole argomenti
  - `int (*b)(int,int)` // a funzione che restituisce int e prende due int
  - `char (*c)(double*, float*);`

7/18

## Reason why



- We can not write or read at that address
  - Forbidden!
  - Therefore, not the same as data pointers
- Main usage
  - Widely used in OS and RTOS programming
  - Vectors of functions (i.e. Interrupt Vector Tables)
  - **Callbacks, namely code that we can forward to other code (typically a function)**
  - We can achieve OOP concepts like inheritance or polymorphism

11/18

## What is a callback?



- A callback is actually a “regular” function
  - Namely, you can invoke it as other functions
- Therefore, “when a function can be considered as a callback”?
- A function is used as a callback when it is passed to another function
  - Namely stored as data in args
  - What we can use to store it? → function pointers!

*The term “callback” is also used for asynchronous functions in other languages or contexts (not our case)*

12/18

## Why we need callbacks?



- Sometimes, providing “instructions” on how to deal with data can lead to “smarter” code
- i.e. we want to compute the sum of numbers in an array using a function
  - Some steps are not dependent on the content of the array
    - i.e. iterating on array elements
  - But others are:
    - i.e. adding two numbers is strictly dependent on number types (complex vs natural)
  - We would need as many functions as many kind of data types we want to deal with or...
  - We can use a single function and provide it with the instructions on how to deal with specific types

13/18

## Main examples



- C standard library contains two main examples of functions that need a “callback”
  - `qsort()` → arbitrary array sorting
  - `bsearch()` → search in ordered arrays
- They are very good examples of “polymorphism”
  - Namely the capability to operate on different kinds of data

14/18

## qsort()



- La funzione `qsort()` permette di ordinare un array generico
  - Quindi array di `int`, `float`, `double`, puntatori, `struct` di differenti tipi...
- Basi di tutti gli algoritmi di ordinamento:
  - Considero gli elementi dell’array a coppie
  - Confronto il loro contenuto
  - Nel caso li scambio di posizione

15/18

## qsort()



- Cosa serve per ordinare un array?
  - Per considerare gli elementi dell'array
    - Devo sapere di quanti elementi è composto
    - Passo alla funzione il numero di elementi dell'array
  - Per gli scambi
    - Devo sapere quanti byte occupa ciascun elemento
    - Passo alla funzione la dimensione in byte di un singolo elemento
  - Per il confronto
    - ??

16/18

## qsort()



- Se devo ordinare un array generico devo poter confrontare elementi di differenti tipologie
  - Ad esempio, il confronto di due scalari è differente dal confronto tra stringhe
  - Operatore ' $>$ ' vs strcmp()
- Non solo
  - Ordinamento crescente o decrescente
- Soluzione, chi invoca la qsort() deve anche fornire la parte di codice che realizza il confronto

17/18

## qsort()



- qsort() takes as inputs:
  - Array to be sorted
  - The number of elements of the array
  - The size of each element
  - The address of a callback for comparing elements

```
void qsort(void *base, size_t nmem, size_t size,  
int (*compar)(const void *, const void *));
```

18/18

## qsort()



- Lets' focus on the callback
  - `int (*compar)(const void *, const void *)`
- It takes as input the address of two elements to be compared
  - As generic void data → internally we need to convert them
- It returns a positive-zero-negative value
  - Result of comparison
    - Depending on actual data we can use relational operators, string comparisons, ...

19/18

## bsearch()



- Binary Search ovvero ricerca binaria
- Ricerca all'interno di array ordinato
- Stessi parametri della qsort() + valore da ricercare
- Restituisce
  - Indirizzo elemento trovato
  - NULL se ricerca infruttuosa

```
void *bsearch(const void *key, const void *base,  
             size_t nmemb, size_t size,  
             int (*compar)(const void *, const void *));
```

20/18



UNIVERSITÀ DI PARMA

## Function Pointers



*Memoria est thesaurus omnium rerum et  
custos*

De Oratore, Cicerone, I sec a.C.





UNIVERSITÀ DI PARMA

# Problem Solving

*If I only had an hour to chop down a tree, I  
would spend the first 45 minutes sharpening  
my axe.*

Abraham Lincoln

## Sommario

UNIVERSITÀ  
DI PARMA 

- What is problem solving?
- Why it is important?
- Few steps
  - Understand the problem
  - Sample data
  - Pseudocode
  - Code
  - Debug or Optimization

# SUMMARY

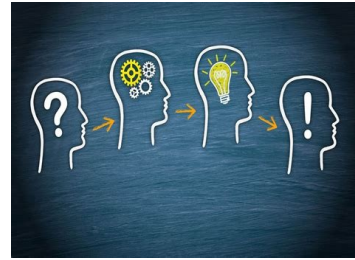


2/12

## Problem solving?

UNIVERSITÀ  
DI PARMA

- What is?
  - Transform the description of a problem into a solution!
- Why it is important?
  - Engineers use problem solving skills, **constantly**
    - Not computer programming only!
  - Direct impact on the pace of your professional progress and career growth

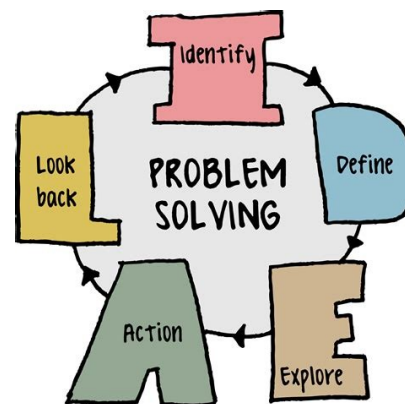


3/12

## What is the right approach?

UNIVERSITÀ  
DI PARMA

- Do not rush in coding!
  - If you have a time limit this can appear captivating
  - But it is wrong!
- 4 steps to follow in the next slides

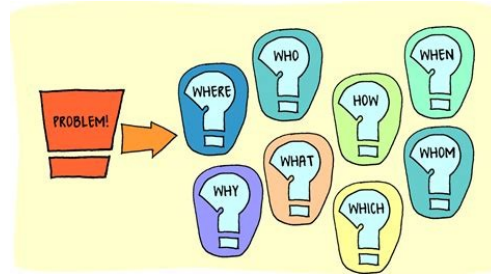


4/12

## 1. Understand the problem: read



- Again, do not rush in coding!
- You can not solve a problem you do not understand
- Read through the problem
- Clarify any part
  - Maybe ask or discuss!
- Once you have read
  - Read it again
  - Up to three times...

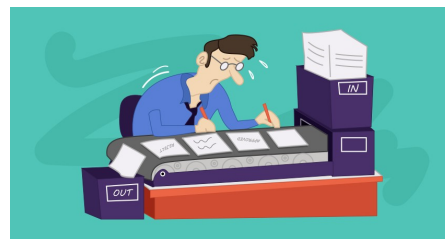


5/12

## 1. Understand the problem: simulate



- Work through the problem manually
  - What are the inputs?
  - What are the outputs?
- Consider sample inputs
- Try to analyze the output
  - Initially simple inputs
  - Then add complexity
  - Also consider corner/edge cases

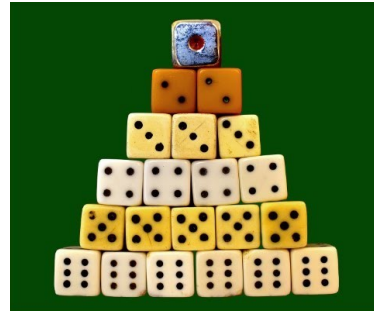


6/12

## 2. Devise a plan for solving the problem: top-down approach

UNIVERSITÀ  
DI PARMA

- Step a step plan to solve the problem
- Think about how you would solve it as a human
- Break down the problem
  - Small sub-problems or chunks
  - Solve them one by one
  - Connect the solution of each sub problem as input for others
- Edge and corner problems are dealt by the proposed solution?



7/12

## 2. Devise a plan for solving the problem: pseudocode

UNIVERSITÀ  
DI PARMA

- Initially write down your steps in natural language
- Then translate your steps in pseudocode
  - Having final language in mind can help

```
MOT1
Program to generate PWM output to Motor

Initialize
  Outputs
    Motor
  Inputs
    Speed up
    Speed down
    Run enable
  Registers
    Count = 128

Start loop
  IF Run enable = off THEN wait
  IF Speed up = on THEN inc Count
  IF Count = 0 THEN dec Count
  IF Speed down = on THEN dec Count
  IF Count = 0 THEN inc Count

  Switch on Motor
  Delay for Count

  Switch off Motor
  Delay for 256-Count

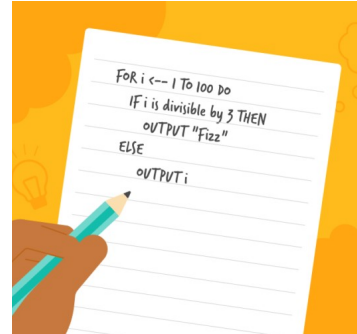
End loop
```

8/12

### 3. Carry out the plan: write actual code



- Translate the pseudocode or algorithm description in programming language
  - Eventually you can code!
- Do not optimize code too early
  - Not required for us...



9/12

### 3. Carry out the plan: debug



- For sure the initial release will not work
- Debug
  - Look at compiler warnings
  - Add some print
  - Again, independently debug each part

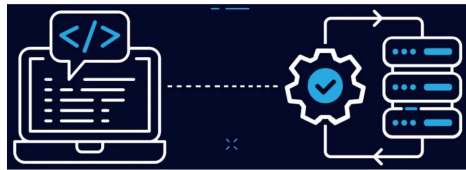


10/12

#### 4. Look back over what you've done: refactoring



- Not necessary in our case
- Reflect on your solution and, maybe, simplify your steps
  - Are improvements possible?
  - What other approaches are possible?
  - Is there anything you can generalize?



11/12



UNIVERSITÀ DI PARMA

## Problem Solving



*If I only had an hour to chop down a tree, I  
would spend the first 45 minutes sharpening  
my axe.*

Abraham Lincoln



UNIVERSITÀ DI PARMA

# Regole Esame

*Don't stress, do your best, forget the rest!*

Anonimo

## Sintesi



- **Prerequisiti:**
  - Essersi iscritti
  - (Aver compilato questionario valutazione didattica del corso)
- **Modalità:** Solo programmazione (in C)
- **Dove:** nei laboratori informatica di base
- **Durata:** 2 ore

2/9

## Procedimento



- Appello e posizionamento esaminandi
  - Necessario documento o libretto
- Lettura pubblica testo esame
- Spazio domande e discussione
  - Anche tra di voi
- Stop chiacchiere, inizio effettivo prova
- 2 ore di programmazione individuale
- Consegna automatica
  - Momento critico!

3/9

## Materiale



- Codice illustrato a lezione e slide presentate
- Possibile portare testi
  - Massimo 2 per studente
  - No fogli singoli o „volanti“
- No cellulari, no chiavette ecc.
- Zaini e materiale vanno lasciati lato cattedra in laboratorio
- Vi viene consegnato foglio esame
  - Potete scrivere solo su questo
- Testo va riconsegnato

4/9



## Codice Comportamentale



- Non si può parlare
  - Espulsione immediata
  - Domande? Solo al docente o aiutanti!
- Non si può copiare o far copiare
  - Voto → 0 (per entrambi)
  - Nei commenti indicate provenienza codice nel caso
- Ci si può ritirare solo entro 20 minuti da inizio prova

5/9

## Codice C



- Accesso con account “generici”
  - Utente: “isdLesaXY” dove L -> laboratorio, XY -> numero postazione
  - Password: “noncopiare”
- Si salva in U:
- Nome progetto → numero di matricola
- main.c deve contenere commento iniziale con vostro nome, cognome e-mail

6/9

## Valutazione

UNIVERSITÀ  
DI PARMA

- Dinamica [0,30 e lode]
- Casi specifici:
  - 18-30 e lode: avete passato l'esame
    - In sede di verbalizzazione potete non accettare il voto
  - 10-17: non passate ma dimostrate impegno
    - Esercitatevi meglio
  - 2-10: avete tentato
    - Ne tengo traccia, fa media (pesata) con tentativi successivi
  - Ritirato: → come non essersi presentati
  - Non compila: → 1
  - Copiato → 0



7/9

## Penalità specifiche

UNIVERSITÀ  
DI PARMA

- Numeri magici (es. 65 invece di 'A')
  - -2 ad ogni occorrenza
- Codice non indentato o indentato male
  - -4
- Mancanza commento intestazione
  - -4

8/9





UNIVERSITÀ DI PARMA

# Laboratori & code::blocks

*Per me si va ne la città dolente,  
per me si va ne l'eterno dolore,  
per me si va tra la perduta gente*

Dante Alighieri, Inferno Canto III

Sommario

UNIVERSITÀ DI PARMA 

- Utilizzo laboratori
  - Accesso
  - Accesso remoto
- Code::blocks
  - Cosa è
  - Utilizzo



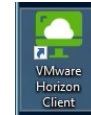
2/21

## Accesso Laboratori

UNIVERSITÀ  
DI PARMA



- Credenziali
  - nome.cognomeXYZ@studenti.unipr.it
  - Relativa password
  - Dominio UNIPR (ma è automatico)
  - In tutti i laboratori
- Dopo accesso, se macchina fisica, occorre usare VMware Horizon Client
- Per esame → credenziali generiche (isdXesaY/“noncopiare”)



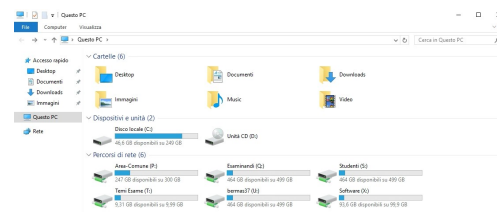
3/21

## Accesso Laboratori

UNIVERSITÀ  
DI PARMA



- Share di rete
  - U: vostra home
    - Salvate i vostri dati sempre qui
    - C: dipende da macchina
  - T: area docenti/esami
    - Materiale docenti
    - Temi esame ecc.



4/21

## Accesso Laboratori da remoto



- Possibile accesso da remoto
  - VMware Horizon Client
  - Istruzioni e download in:
    - [https://wiki.asi.unipr.it/dokuwiki/doku.php?id=guide\\_utente:accesso\\_remoto\\_laboratori](https://wiki.asi.unipr.it/dokuwiki/doku.php?id=guide_utente:accesso_remoto_laboratori)
- Sempre accedere a vdi-gw.unipr.it
  - Scegliere pool Ingegneria Lab, Informatica XY

5/21

## Code::blocks: cosa è



- Ambiente integrato
- IDE: Integrated Development Environment
  - Gestione progetti
  - Editor
  - Compilatore (e linker)
  - Debugger
- Release 20.03 (luglio 2022)



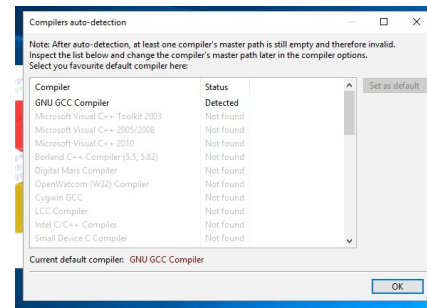
6/21

## Code::blocks: prima esecuzione

UNIVERSITÀ  
DI PARMA



- Prima esecuzione:
  - Installazione
  - Scegliere default



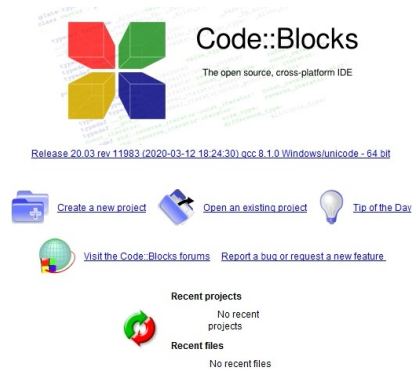
7/21

## Code::blocks: creazione nuovo progetto

UNIVERSITÀ  
DI PARMA



- Creare il vostro codice:
  - Create New Project

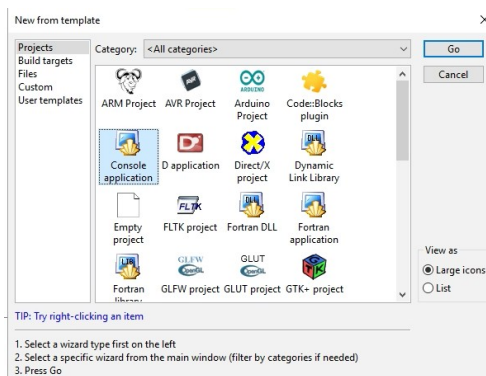


8/21

## Code::blocks: creazione nuovo progetto

UNIVERSITÀ  
DI PARMA

- Creare il vostro codice:
  - Create New Project
  - Console application

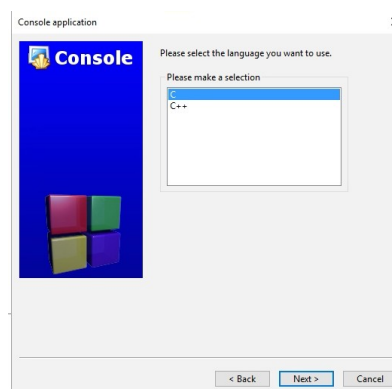


9/21

## Code::blocks: creazione nuovo progetto

UNIVERSITÀ  
DI PARMA

- Creare il vostro codice:
  - Create New Project
  - Console application
    - Wizard
  - Select C



10/21



## Code::blocks: creazione nuovo progetto



- Creare il vostro codice:
  - Create New Project
  - Console application
    - Wizard
  - Select C
  - Inserire
    - Titolo progetto
    - Folder (questo PC → U:)

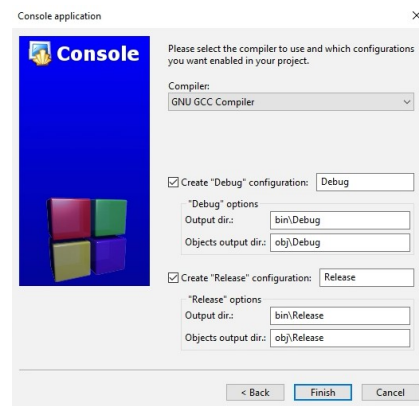


11/21

## Code::blocks: creazione nuovo progetto



- Creare il vostro codice:
  - Create New Project
  - Console application
    - Wizard
  - Select C
  - Inserire
    - Titolo progetto
    - Folder (questo PC → U:)
  - Scelta compilatore → default

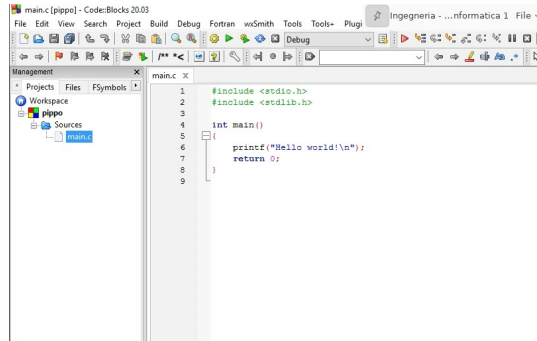


12/21

## Code:blocks: schema progetto C

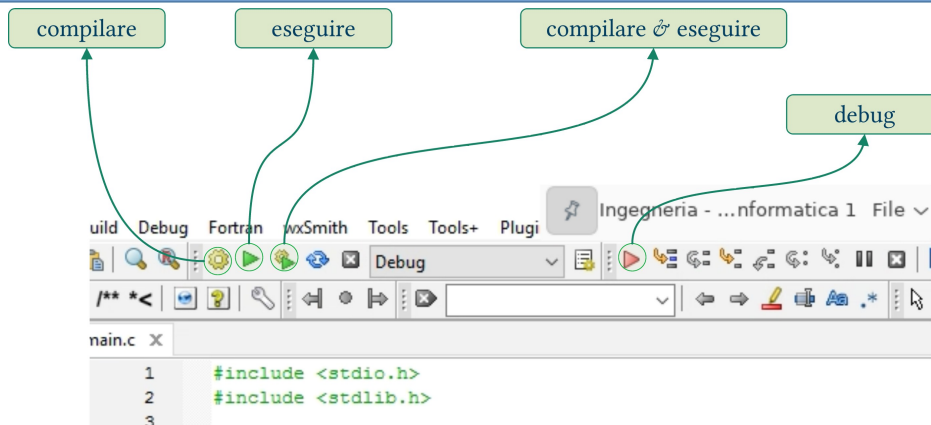
- Alla fine della procedura:

- Progetto
- Scheletro main.c



13/21

## Code:blocks: GUI



14/21

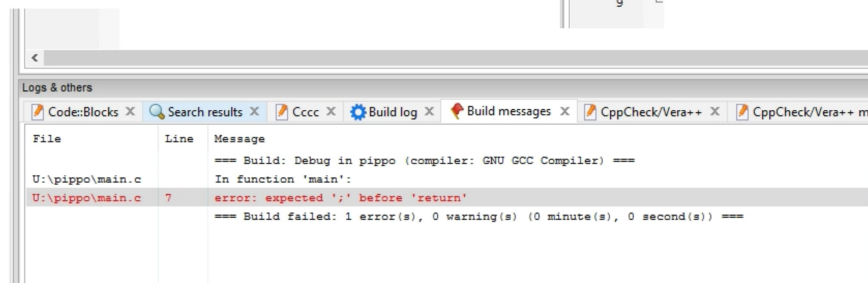
## Code::blocks: compilazione

UNIVERSITÀ  
DI PARMA



- Genera eseguibile
- Attenzione a warning ed errori!

```
main.c X
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      printf("Hello world!\n")
7      return 0;
8  }
9
```



15/21

## Code::blocks: caveat su warning

UNIVERSITÀ  
DI PARMA



- Errore di sintassi o lessicale → errore di compilazione
  - Non si scappa, dovete risolverlo!
- Qualcosa non suona bene ma è ammesso → compila ma warning
  - **Absolutamente NON ignorare, vanno risolti!**

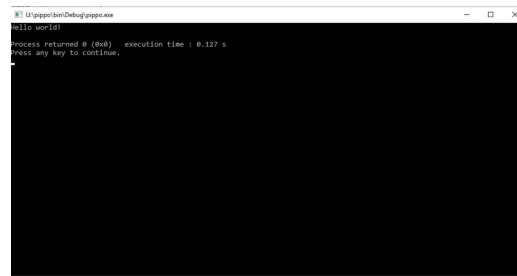


16/21

## Code::blocks: esecuzione



- Esecuzione
  - Apre “console”
  - NON chiudere con mouse

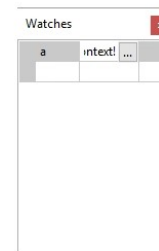
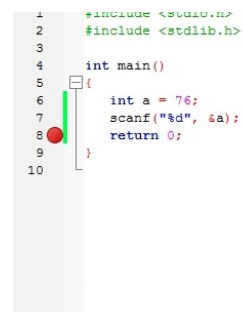


17/21

## Code::blocks: debug



- Modalità debug
  - Esecuzione passo passo
  - Ispezione valori variabili
  - Breakpoint
- In laboratorio NON funziona



18/21

## Uso proprio portatile

- Possibile a lezione
  - NON ad esame
- Installare la stessa versione di code::blocks
  - WINDOWS: con MinGW
  - LINUX: occorre gcc
  - MAC OS: ...

### Microsoft Windows

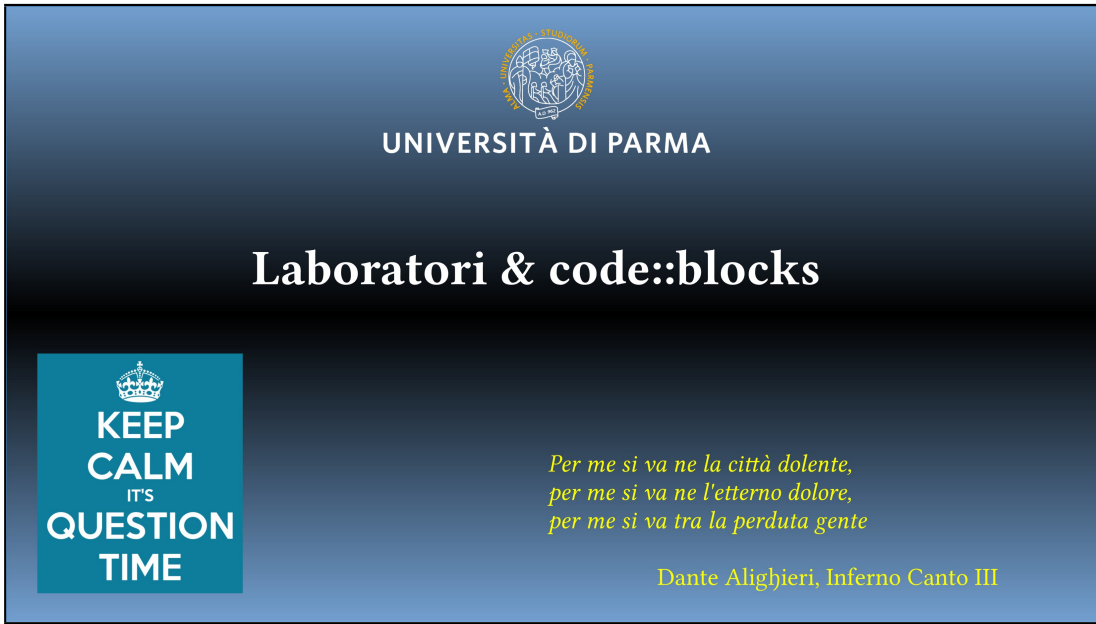
| File                                      | Download from                                              |
|-------------------------------------------|------------------------------------------------------------|
| codeblocks-20.03-setup.exe                | <a href="#">FossHUB</a> or <a href="#">Sourceforge.net</a> |
| codeblocks-20.03-setup-nonadmin.exe       | <a href="#">FossHUB</a> or <a href="#">Sourceforge.net</a> |
| codeblocks-20.03-nosetup.zip              | <a href="#">FossHUB</a> or <a href="#">Sourceforge.net</a> |
| codeblocks-20.03mingw-setup.exe           | <a href="#">FossHUB</a> or <a href="#">Sourceforge.net</a> |
| codeblocks-20.03mingw-nosetup.zip         | <a href="#">FossHUB</a> or <a href="#">Sourceforge.net</a> |
| codeblocks-20.03-32bit-setup.exe          | <a href="#">FossHUB</a> or <a href="#">Sourceforge.net</a> |
| codeblocks-20.03-32bit-setup-nonadmin.exe | <a href="#">FossHUB</a> or <a href="#">Sourceforge.net</a> |
| codeblocks-20.03-32bit-nosetup.zip        | <a href="#">FossHUB</a> or <a href="#">Sourceforge.net</a> |
| codeblocks-20.03mingw-32bit-setup.exe     | <a href="#">FossHUB</a> or <a href="#">Sourceforge.net</a> |
| codeblocks-20.03mingw-32bit-nosetup.zip   | <a href="#">FossHUB</a> or <a href="#">Sourceforge.net</a> |

19/21

## Uso proprio portatile

- Pro:
  - Più veloce (forse)
  - Pratico per esercitarsi a casa
- Contro:
  - All'esame non utilizzabile
  - Occorre impratichirsi comunque all'ambiente per prova d'esame

20/21

This image shows a single sheet of white paper with horizontal ruling lines. The lines are evenly spaced and run across the width of the page. There are no margins, text, or other markings on the paper.



UNIVERSITÀ DI PARMA

# Debuggin with Code::Blocks

*The most careful tool is still placed properly*



*g tool is still h judiciously*

Brian Kernighan

## Summary



- Background
- Basic C issues
- Tools and Techniques
- Code::Blocks debug tool
- Hints



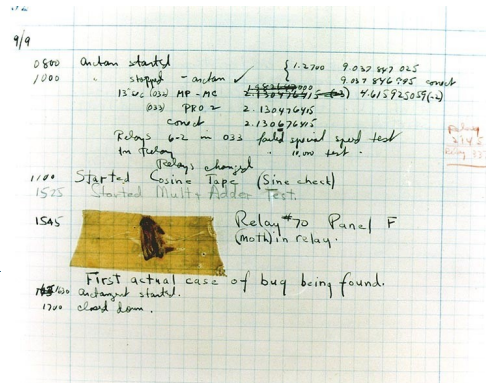
2/22

## Background

UNIVERSITÀ  
DI PARMA



- What is a bug?
  - Design defect in a software
- Do not underestimate them!
  - ~0.6% of USA GDP
  - Links to different disasters
- Guidelines for SW development in critical situations
  - MISRA C
- Some languages exclude features that easily lead to bugs
  - TrapC extension



3/22

## Types of bugs

UNIVERSITÀ  
DI PARMA



- Syntax errors
  - Caught by the compiler
- Semantic errors
  - Uninitialized variables, dead code, typing wrong variables, **pointer/array errors**
  - Maybe warnings

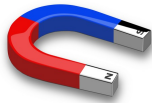
4/22



## Debugging techniques



- Enable as much warnings as possible
  - -Wall
- Code walkthrough
- printf() debugging
- assert() macro
- Debugging tools



5/22

## Code walkthrough



- Look at warnings
  - Modify the code to solve them!
- Read again the code that is not working
  - And again
  - And again
  - And again

6/22

## printf() debugging



- Add some printf() in the portion of the code that does not work properly
- 2 main cases
  - Print variable content
  - Print something that allows to understand whether that part of code is reached

7/22

## printf() debugging example



```
#include <stdio.h>
int main(int argc, char **argv){
    int sum=0, val, num=0;
    double ave;
    while (scanf("%d\n",&val) == 1) {
        sum += val;
        num++;
    }
    if(num > 0) {
        ave = sum/num;
        printf("Average is %f\n", ave);
    }
}
```

At each iteration,  
*sum* should be the  
total of entered  
*num* values so far

At the end, *sum*  
should be the total  
of entered *num*  
values

Ave should be the  
floating point  
average of input  
numbers

8/22

## printf() debugging example

```
#include <stdio.h>

int main(int argc, char **argv){
    int sum=0, val, num=0;
    double ave;
    while (scanf("%d\n",&val) == 1) {
        sum += val;
        num++;
        printf("val = %d sum = %d num = %d\n", val, sum, num);
    }
    printf("After while() val = %d sum = %d num = %d\n", val, sum, num);
    if(num > 0) {
        ave = sum/num;
        printf("sum = %d num = %d sum/num = %f\n", sum, num, sum/num); //we have a warning here...
        printf("Average is %f\n", ave);
    }
```

9/22

## printf() debugging

- Disadvantages
  - We need to add/remove code
    - Anyway we can use conditional compilation
  - Maybe slow
  - Output clogging
  - stdout is buffered
    - Use the stderr

10/22

## #ifdef



- Preprocessor directive
- Widely used for conditional compilation
- Syntax:

```
#ifdef <condition>
... <some code> ...
#endif
```
- When the condition is false the code is simply wiped out by the preprocessor before passing it to the compiler

11/22

## assert() macro



- Used to test a condition that **must** be always true
  - If no bugs...
- When the condition is not true
  - Print error message and terminate the program
- Good programming practice
  - Widely used in the development stage
  - Not meant to be used for catching runtime input errors
  - Can be easily disabled in the release stage
  - In our case, not very useful at exam time

12/22

## assert() examples



```
#include <assert.h> // mandatory
...
assert(day_of_month <= 31);
...
void resize(int *value) {
    assert( value != NULL );
...

```

13/22

## Debugger



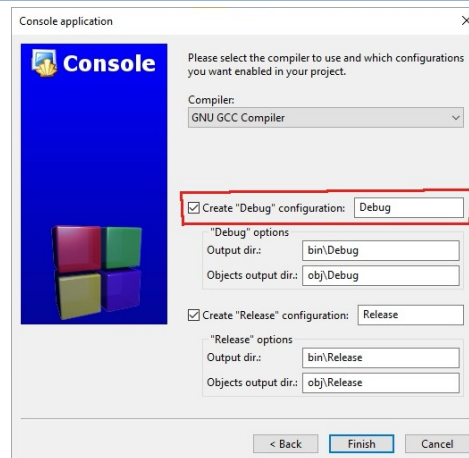
- A debugger is a tool that can monitor a program execution performing:
  - Watching/inspecting variable content
    - Even change variable contents
  - Halting program on given conditions
  - Step by step execution
- The debugger **must** have access to the source code

14/22

## Compiling for debugging

UNIVERSITÀ  
DI PARMA

- In order to debug a program we need to instruct the compiler to also **include references to the source code** in the executable
- Code::Blocks, by default, also create a debug executable

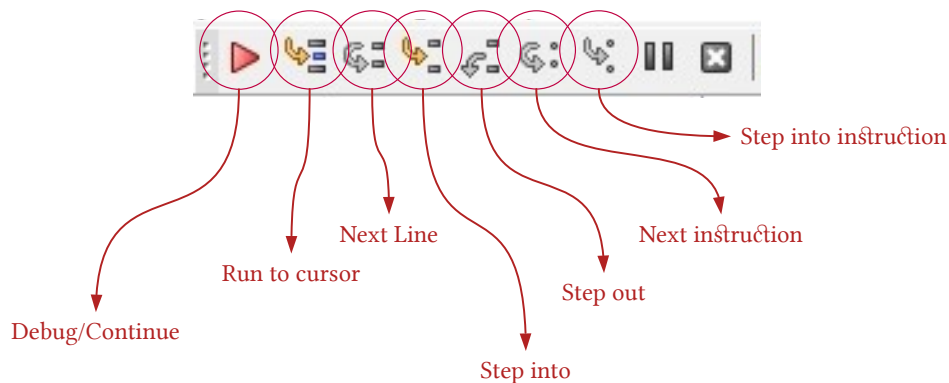


15/22

## How to debug in Code::Blocks?

UNIVERSITÀ  
DI PARMA

- You have to use specific commands

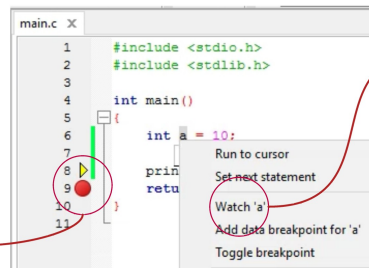


16/22

## How to debug in Code::Blocks?

- Moreover we can also add:
  - Breakpoints:** execution stopping places
  - Watches:** inspection tool for showing variables code

To add a breakpoint, click on the gray part just on the right of the line number

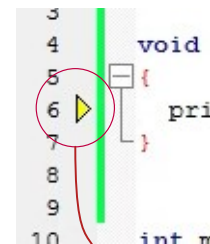


Select and right-click on a variable name to inspect its content

17/22

## How to debug in Code::Blocks?

- Debug/continue:** start or continue execution but stop at next breakpoint
  - No breakpoints → normal execution
- Run to cursor:** start execution but stops at cursor line
- Next line/instruction:** execution step by step
- Step into:** step by step also entering functions



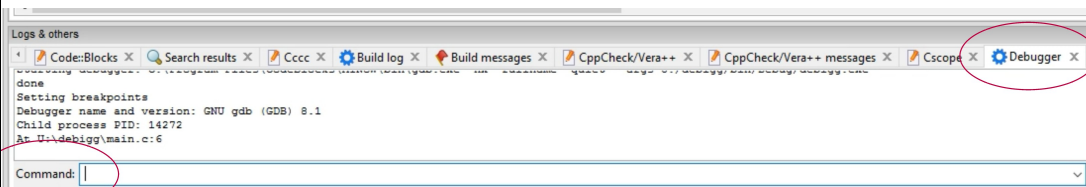
Line/instruction to be executed next

18/22

## How to debug in Code::Blocks?

UNIVERSITÀ  
DI PARMA

- On the bottom part you can also find a Command Line Interface
- You can issue debug commands, most useful:
  - **bt**: print call hierarchy
  - **p <exp>**: i.e., “p a[i]”, print content of a variable or result of an expression
  - **w <condition>**: i.e., “w a==3”, stop the execution when the condition is not met



## Debugging hints

UNIVERSITÀ  
DI PARMA

- Start debug where the bugs are!
- When not clear, try to understand as fast as possible where the problem is
- Binary search principle:
  - If you have a large chunk of code but you are not sure where the bug is → put a breakpoint in the middle
  - This allow you to maybe understand whether the bug is before or after that point

20/22



## Debugging hints



- In a “long” cycle use conditional watches
- i.e. using the “w” CLI command

21/22



UNIVERSITÀ DI PARMA

## Debugging with Code::Blocks

*If it ain't broken, don't fix it.*

Ronald Reagan

*If we can't fix it, it ain't broke.*

Lt. Col. Walter Weir