2.2 – Espressioni e operatori
(*ioni)

Libro di testo:

- Capitolo 4.1-4.3

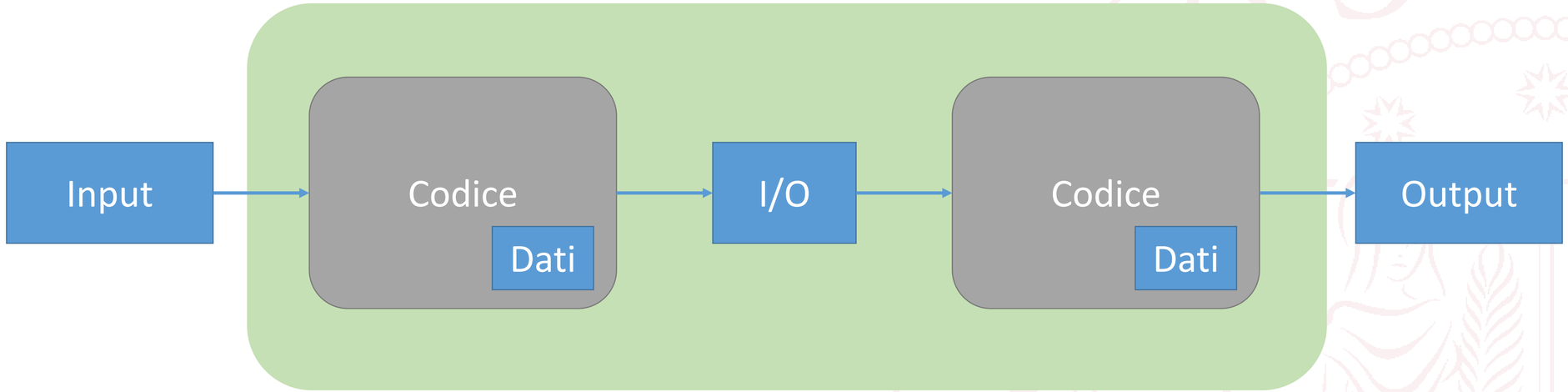
DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Agenda

- Espressioni
- Operatori con side effect
- Ordine di valutazione di un'espressione

Computazione

- Produrre output a partire da input
- Un programma può essere costituito da diverse parti



Computazione

- L'obiettivo del programmatore è esprimere la computazione in modo
  - Corretto
  - Semplice
  - Efficiente
- La computazione è spesso gestita dividendo componenti complessi in un numero maggiore di componenti semplici

Astrazione

- L'astrazione è un concetto fondamentale in programmazione
  - Nascondere i dettagli implementativi che non servono per utilizzare una risorsa, una funzione, …
- **Interfaccia**: definizione di come utilizzare una risorsa o una funzione (senza conoscere i dettagli implementativi)
- L'astrazione è usata anche nel più semplice dei programmi
  - `std::cout << "Hello, world!\n";`
    - Come?

Espressioni

- Il più piccolo elemento usato per esprimere la computazione è l'**espressione**
- Un'espressione calcola un risultato a partire da uno o più operandi
- Espressioni semplici:
  - Literal: `10, 'a', 3.14, "Norah"`
  - Nomi di variabili
- Espressioni semplici possono essere combinate con operatori per formare espressioni più complesse

```
int perimeter = (length + width) * 2;
```

Lvalue

- Un elemento molto importante nelle espressioni è l'*lvalue* (left value)

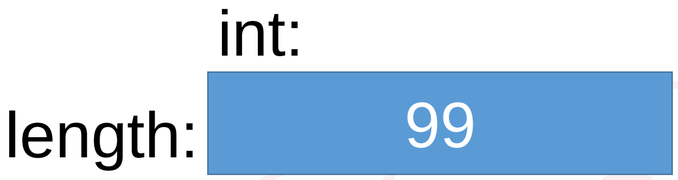
- Un *lvalue* è ciò che sta alla sinistra di un operatore di assegnamento
  - Variabili

```
int length;
length = 99;
```

- **Un nome di variabile è sempre un *lvalue*?**

Lvalue vs. Rvalue

```
int length;
length = 99;
length /= 2;
int width;
width = length * 2;
```



- `length` si riferisce:
  - Come lvalue: a un oggetto di tipo `int` che contiene il valore `99` – **`length` è il box (l'oggetto)**
  - Come rvalue: il riferimento al **valore contenuto** nell'oggetto
- In un'espressione, `length` può essere usato sia come lvalue che come rvalue

Espressioni costanti

- Il software ha bisogno di molte espressioni costanti
- Il C++ permette di esprimere una costante simbolica

- Meglio dei magic numbers / magic constants
- Meglio dei `#define` (macro stile C) perché queste costanti hanno un tipo

```
constexpr double pi = 3.14159;
pi = 7;                                  // errore!
double c = 2 * pi * r;                   // OK: sola lettura
```

`constexpr` vs `const`

- Due modi per esprimere una costante
  - `constexpr`: il valore è noto a tempo di compilazione
  - `const`: il valore è noto solo a tempo di esecuzione ed è assegnato in inizializzazione

```
constexpr int max = 100;

void use (int n)
{
    constexpr int c1 = max + 7;    // OK: da costanti
    constexpr int c2 = n + 7;      // errore: non conosciamo il
                                   // valore a tempo di compilazione
    // ...;
}
```

`constexpr` vs `const`

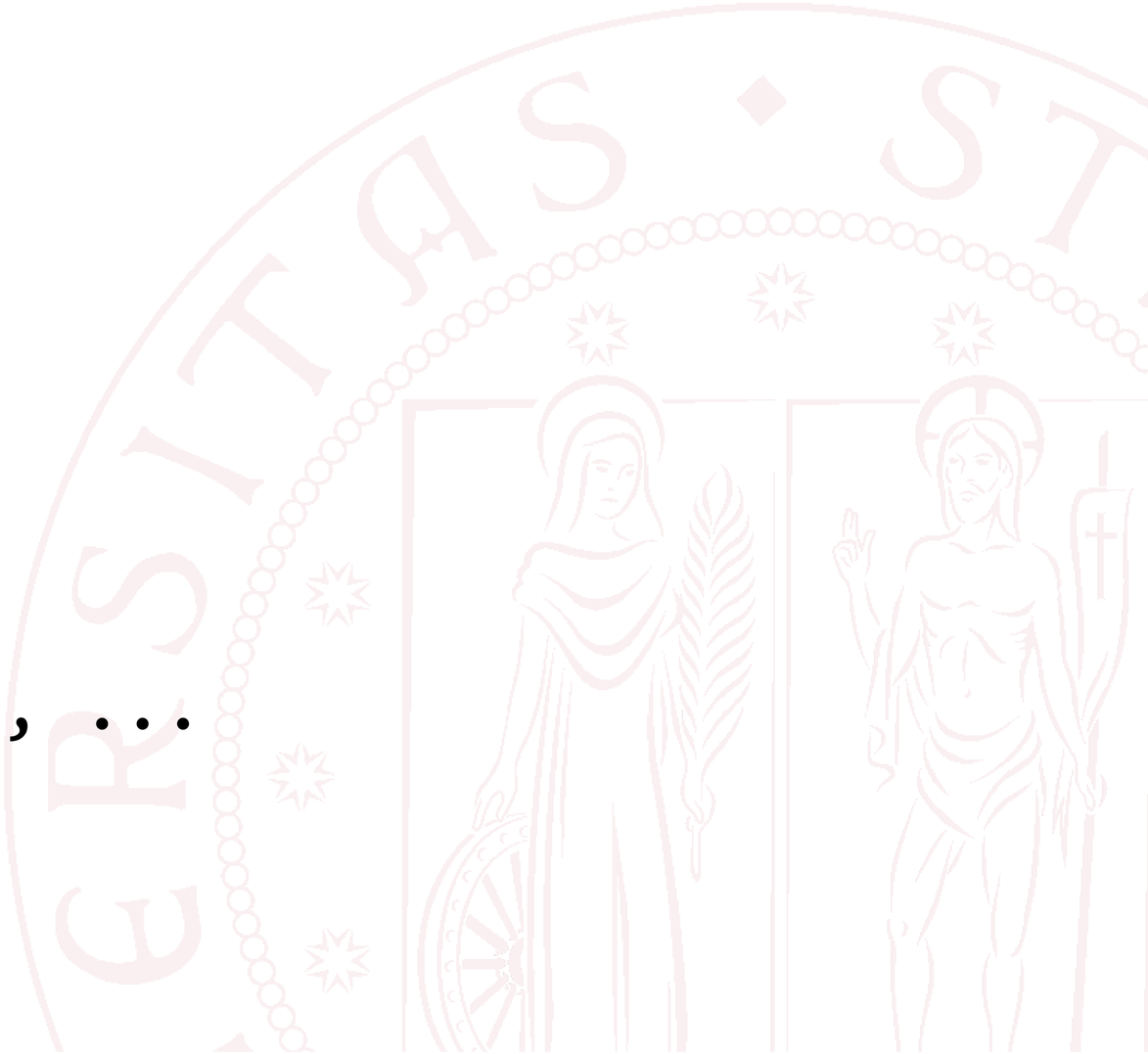
- Due modi per esprimere una costante
  - `constexpr`: il valore è noto a tempo di compilazione
  - `const`: il valore è noto solo a tempo di esecuzione ed è assegnato in inizializzazione

```
constexpr int max = 100;

void use (int n)
{
    constexpr int c1 = max + 7;    // OK: da costanti
    const int c2 = n + 7;          // OK

    // ...;

   c2 = 7;                         // errore
}
```

Side effect e valutazione delle espressioni

Operatori e side effect

- Gli operatori sono caratterizzati da:
  - 1, 2, 3 operandi
  - un risultato
- Alcuni operatori hanno un *side effect*
  - Modificano gli operandi su cui operano
  - Forniscono un risultato
- Es: hanno side effect `++`, `--`, `+=`, `-=`, `...`



Valutazione delle espressioni

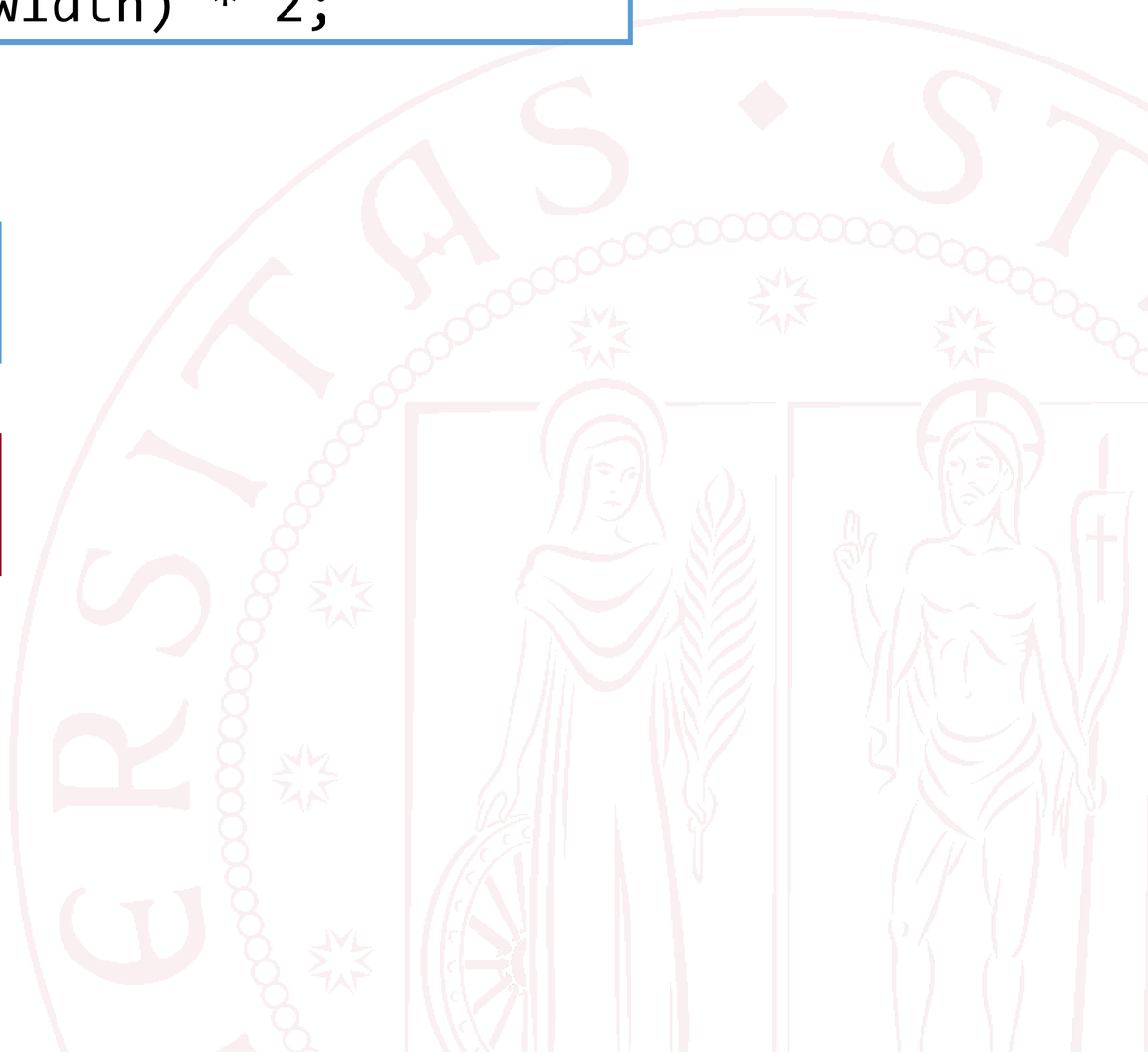
- In che ordine sono valutate le espressioni?

```
int perimeter = (length + width) * 2;
```

- Operazioni:
  - Lettura di `length`
  - Lettura di `width`
  - Somma
  - Prodotto
  - Assegnamento

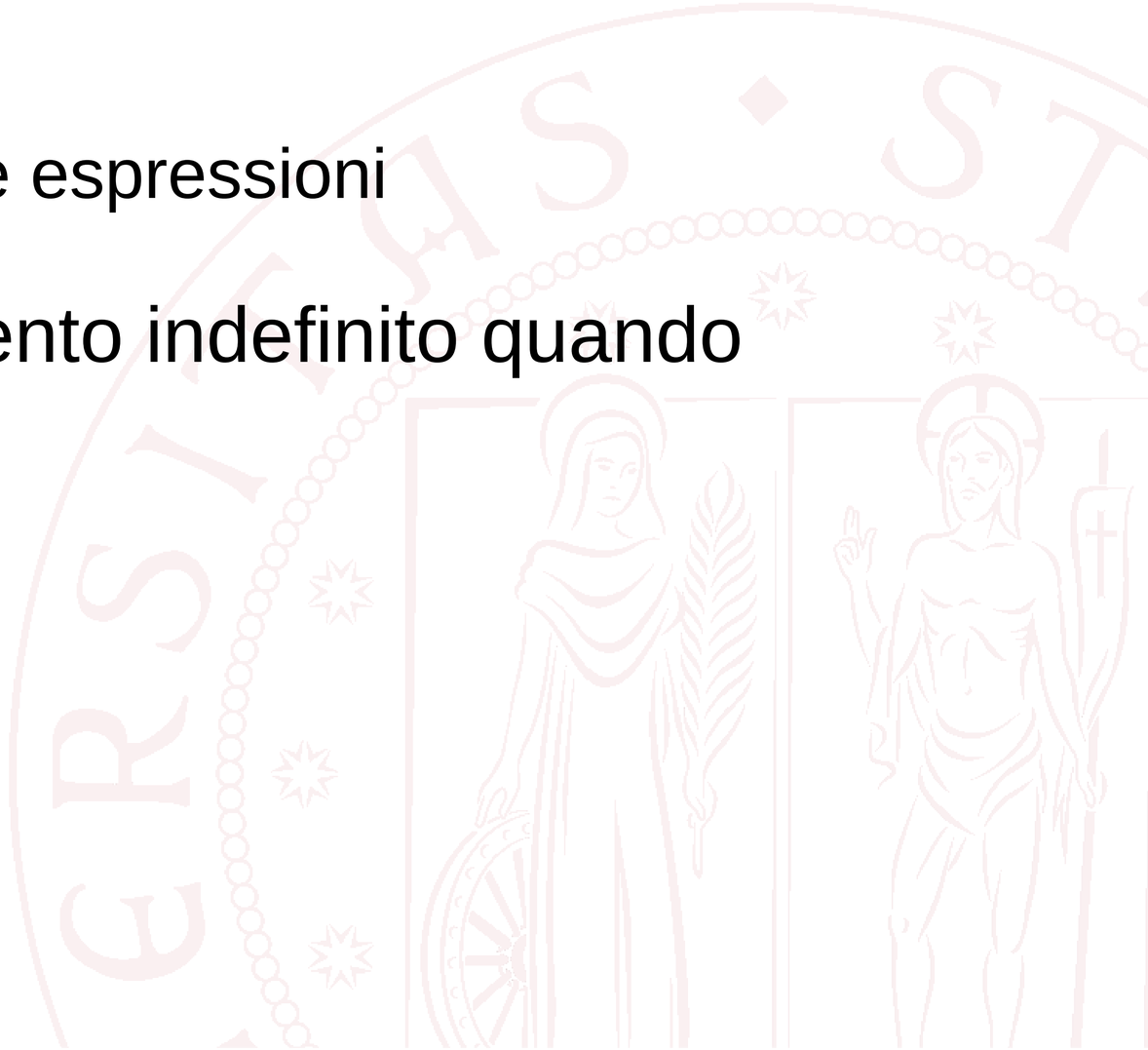
Quale delle due è svolta per prima?

Hanno un ordine obbligatorio



Valutazione delle variabili

- Maggiori possibilità di ottimizzazione se il compilatore ha qualche grado di libertà
  - Libertà nell'ordine di valutazione delle espressioni
- Libertà del compilatore: comportamento indefinito quando non sono fissate le regole



Valutazione delle espressioni

- L'ordine di valutazione delle espressioni non è definito
  - È sbagliato usare la stessa variabile più di una volta se su essa sono applicati operatori con side effect

```
v[i] = ++i;          // Ordine di valutazione non definito:
                     // i a sinistra è letto prima o dopo la valutazione di ++i?

v[++i] = i;          // Ordine di valutazione non definito:
                     // Come sopra

int x = ++i + ++i;   // Ordine di valutazione non definito:
                     // lettura e incremento potrebbero non essere consecutivi

std::cout << ++i << ' ' << i << '\n’;   // Ordine di valutazione non definito
```

- Nota: = è un operatore
- Anche per i suoi operandi non è definito quale dei due sia valutato per primo

Tipi che influenzano operatori

Operatori << e >>

- Abbiamo visto l'operatore << (inserimento in uno stream)
- Esiste l'operatore complementare: >> (estrazione da uno stream)
- Effettuano una trasformazione dalla forma *testuale* alla rappresentazione interna delle variabili
  - Sono sensibili al tipo

Operatore di input

```
#include <iostream>

int main(void)
{
    std::cout << "Please enter your first name and age\n";
    string first_name;
    int age;
    std::cin >> first_name;
    std::cin >> age;
    std::cout << "Hello, " << first_name << " (age " << age << ")\n;

    return 0;
}
```

operatori di input

operatori di output

- L'**operatore di input >>** legge un dato (da tastiera) e lo salva in una variabile
- L'**operatore di input >>** è sensibile al tipo

cosa vuol dire?

Operatore di input sensibile al tipo

```
std::string name;                  // variabile stringa
int age;                           // variabile intero
std::cin >> name;                  // leggi una stringa
std::cin >> age;                   // leggi un intero
std::cout << "Ciao, sono " << name << " e ho " << age << " anni\n";
```

- Se come input inserisco (Luca, 20), l'output a schermo sarà:

```
Ciao, sono Luca e ho 20 anni
```

- Alla variabile `age` è stato associato il valore `38`
- Alla variabile `name` è stato associato il valore `Luca`

cosa succede se inserisco in input: (20, Luca)?

Altri operatori sensibili al tipo

```
int age;
std::cin >> age;
std::string name;

int c2 = age + 2;                       // add integer
std::string s2 = name + " Jr.";         // append string

int c3 = age – 2;                       // subtracts integer
std::string c3 = name – " Jr.";         // error: not defined for strings
```