8.1 – Organizzazione del software in progetti complessi

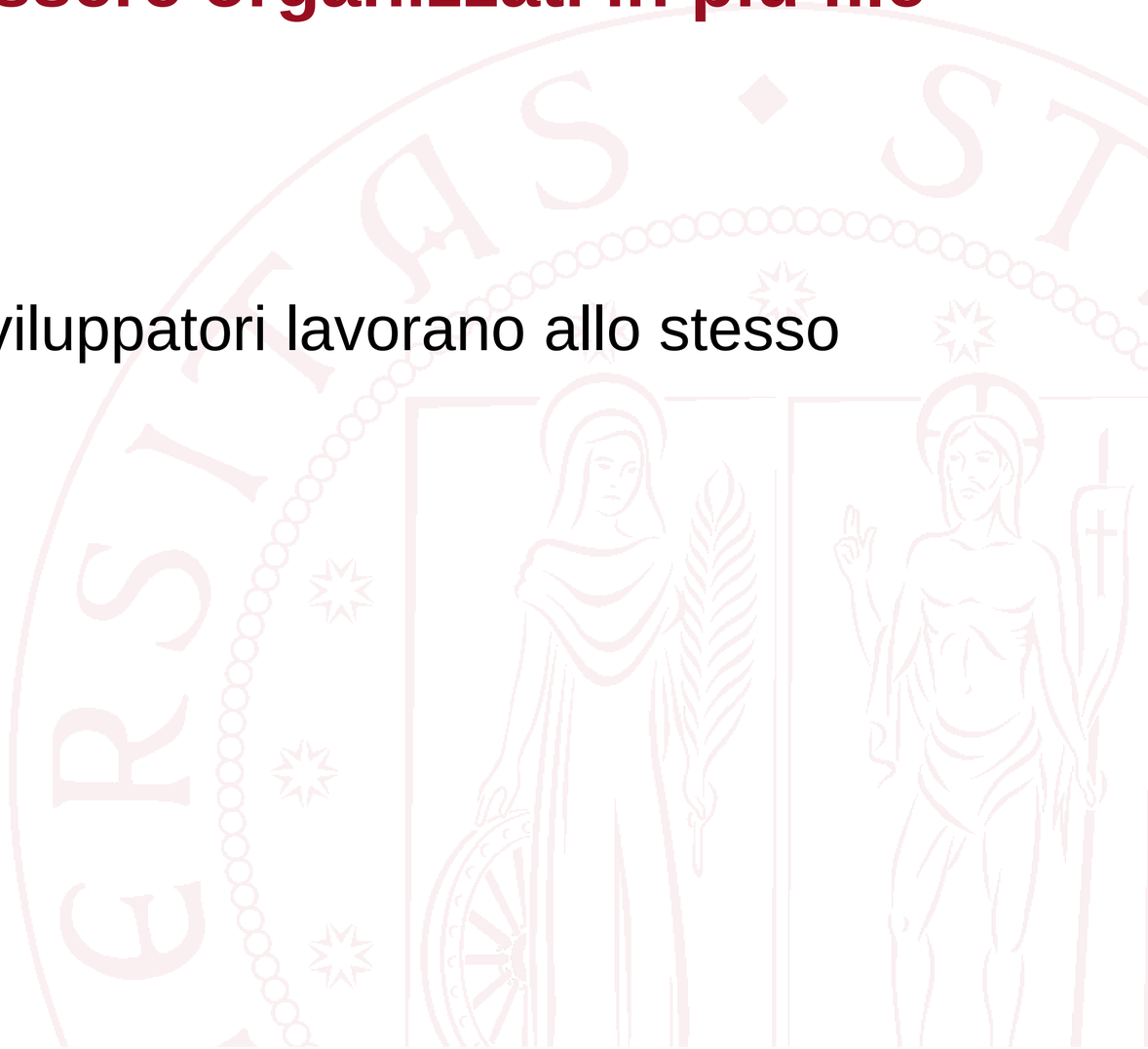
DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Agenda

- Progetti su più file
- Il ruolo degli header
- Include guards
- Linking
- Stile

Progetti su file multipli

- Progetti software complessi **devono essere organizzati in più file**
  - Migliore organizzazione
  - Migliore leggibilità
  - Maggiore semplicità organizzativa se più sviluppatori lavorano allo stesso progetto
  - **Progetto intermedio e finale**



Progetti su file multipli

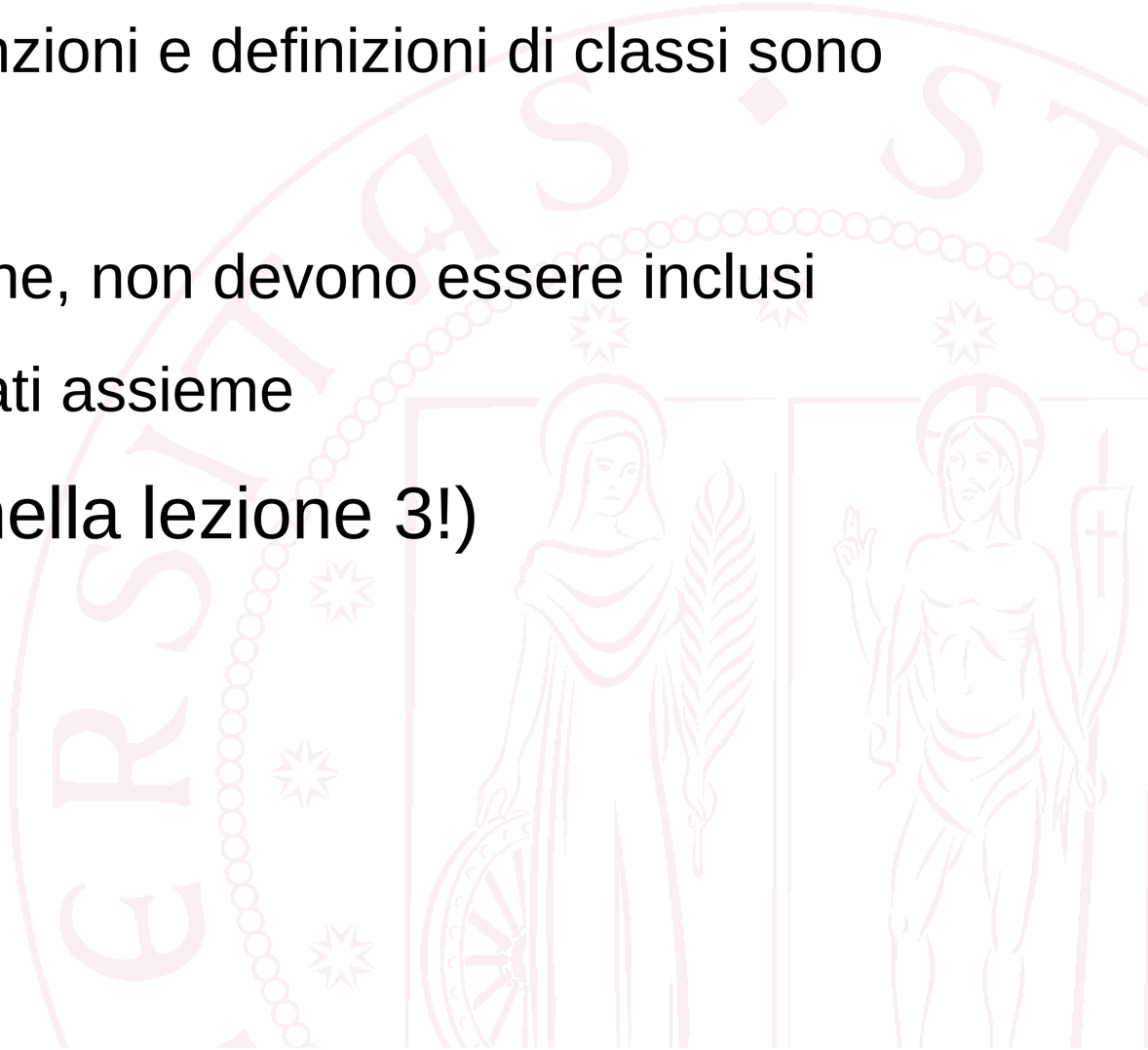
- Esistono due tipi fondamentali di file:

- File header (`.h` oppure `.hpp`), contengono:
  - Definizioni di classe
  - Dichiarazioni di funzione
  - Inclusi (`#include`) ma **non compilati direttamente**
    - `g++ -o my_program source1.cpp source2.cpp ~~my_header.h~~`

- File sorgente (`.cpp`), contengono:
  - Definizioni di funzioni e classi
  - Il `main` (uno solo in tutto il progetto, in un solo file sorgente)
  - Compilati dal compilatore, **ma non inclusi**
    - ~~`#include "source1.cpp"`~~

Progetti su file multipli

- Perché queste due modalità?
  - Header: inclusi perché dichiarazioni di funzioni e definizioni di classi sono funzionali alla scrittura dei file `.cpp`
  - I sorgenti `.cpp` non hanno questa funzione, non devono essere inclusi
  - Le *translation unit* (file oggetto) sono linkati assieme
- Vediamolo con un esempio (già visto nella lezione 3!)



Progetti software in file multipli

```
int f(int i);
```

Dichiarazione

```
int main(void)
{
  int i = 0;

  i = f(i);

  return 0;
}
```

Chiamata

```
int f(int i)
{
  return i + 2;
}
```

Definizione

Progetti software in file multipli

- Come possiamo distribuire un SW in molti file?

```
int f(int i);
```

Dichiarazione → Header file (`my_func.h`)

```
int main(void)
{
  int i = 0;

  i = f(i);

  return 0;
}
```

Chiamata

```
int f(int i)
{
  return i + 2;
}
```

Definizione → File sorgente di libreria (`my_func.cpp`)

Header multipli

- Un file sorgente può includere più header

```
#include <iostream>
#include "date.h"
#include "year.h"
```

- Gli header possono includere altri header
  - `date.h` include `year.h`
- Questo può generare problemi
  - Inclusioni cicliche: `a.h` include `b.h`, e viceversa

Include guards

- Ogni header **deve** essere protetto dalle **include guards**

```
#ifndef RATIONAL_H
#define RATIONAL_H

// ...

#endif // RATIONAL_H
```

- Talvolta rese con:

```
#pragma once        // NON standard!!! ERRORE
```

- Non è standard! **Lo consideriamo errore!**

Include guards

- Quindi la nostra classe `Complex` diventa:

Complex.h

```
#ifndef COMPLEX_H
#define COMPLEX_H

class complex {
  public:
      complex(double);
      complex(double, double);
      // ...
};


#endif // COMPLEX_H
```

Complex.cpp

```
#include "Complex.h"

Complex::Complex(double) {
// ...
}

Complex::Complex(double, double) {
// ...
}

// ...
```

- **Le include guards vanno inserite solo negli header!**

Errori di linking

- Lavorando con più file è più frequente la presenza di errori di linking
- È molto importante distinguere:
  - Errori di compilazione: problemi di sintassi, di conversione di tipo, …
    - **Segnalati da `g++`**
  - Errori di linking: parti di software mancanti
    - **Segnalati da `ld`**
- La prima verifica da fare in caso di errore è capire se è di compilazione o di linking

Errori di linking

Tipico errore di linking: definizione mancante

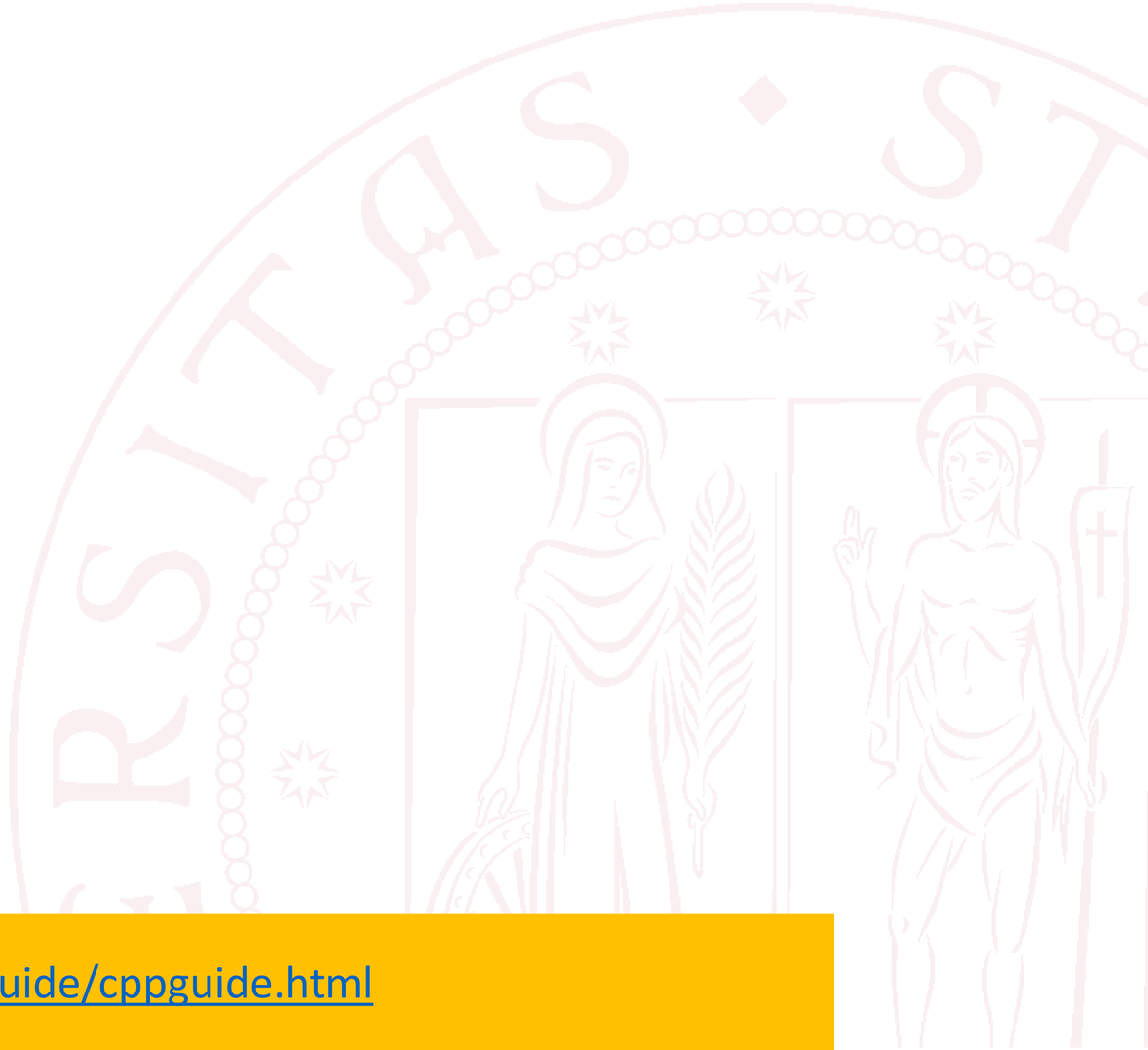
```
double area(double w, double h);

int main()
{
    int w = 2, h = 4;
    int a = area(w, h);

    return 0;
}
```

Coding style

- Dopo aver scritto un po' di codice è molto utile rivedere la Google C++ Style Guide
- Sezioni utili e comprensibili:
  - Header files
  - Scoping
  - Classes
  - Functions
  - Naming
  - Comments
  - Formatting



https://google.github.io/styleguide/cppguide.html

Coding style

- Header files
  - Self-contained Headers
  - The `#define` guard
  - Include What You Use
  - Inline Functions
    - Only few lines
- Names and Order of Includes
  - System headers, C++ standard library, other libraries, my project headers

```
#ifndef FOO_BAR_BAZ_H_        //<PROJECT>_<PATH>_<FILE>_H_
#define FOO_BAR_BAZ_H_

...

#endif  // FOO_BAR_BAZ_H_
```

Coding style

- Scoping
  - Namespaces
    - In large projects, use namespace
    - Do not use using-directives
  - Nonmember, Static Member, and Global Functions
    - Do place nonmember (helper) functions of a class inside a namespace
    - Do not use a class only to group static members
  - Local Variables
    - Place function’s variables in the narrowest scope possible
    - Initialize variable in the declaration

Coding style

- Classes
  - Doing Work in Constructors
    - Funzioni virtuali che per il momento non avete studiato
  - Implicit Conversions
    - Do not define them!
  - Copyable and Movable Types
  - Structs vs. Classes
  - Operator Overloading
  - Access Control
  - Declaration Order

Coding style

- Functions
  - Inputs and Outputs
    - Prefer return over output parameters
  - Write Short Functions
  - Function Overloading
  - Default Arguments

Coding style

- Naming
  - General Naming Rules
  - File Names
  - Type Names
  - Variable Names
  - Constant Names
  - Function Names
  - Namespace Names
  - Enumerator Names

Coding style

- Comments
  - Comment Style
  - File Comments
  - Class Comments
  - Function Comments
  - Variable Comments
  - Implementation Comments
  - Punctuation, Spelling, and Grammar