Errori comuni

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Errori comuni

```
#include <iostream>

int main(void)
{
  std::cout << "Hello, World!\n";
  return 0;
}
```

- Concludere con `:` al posto di `;`
- Dimenticarsi `"` quando si chiude una stringa
- Dimenticarsi a capo (`\n` oppure `std::endl`)

Errori comuni

```
void print(std::string label, std::vector<int> v) {
   // ...
}
```

- Considerate se ha senso passare gli oggetti per copia o per riferimento

```
void print(const std::string& label, const std::vector<int>& v) const {
   // ...
}
```

Errori comuni

```
|- MyProject/
      |- HelloWorld.cpp
      |- Fibonacci.cpp
      |- Print.cpp
```

```
|- HelloWorld/
      |- HelloWorld.cpp
|- Fibonacci/
      |- Fibonacci.cpp
|- Print/
      |- Print.cpp
```

- Create un workspace per progetto!
- Siate ordinati!
  - Modificare un sorgente e compilarne un altro con lo stesso nome
  - Adesso sembra superfluo, ma più andrete avanti più aumenterà la confusione

Divisione interfaccia e implementazione

- **Il `main` deve essere in un file a sé stante**
- le librerie vanno separate
  - **in `.h` (interfaccia)**
  - **in `.cpp` (implementazione)**

Divisione interfaccia e implementazione

Rational.h

```
#ifndef RATIONAL_H
#define RATIONAL_H

class Rational {
  public:
      Rational(int num, int den);
      int numerator(void) const;
      int denominator(void) const;
      //...

  private:
      int numerator_;
      int denominator_;
};

bool operator==(Rational a, Rational b);
#endif
```

Rational.cpp

```
#include "Rational.h"

Rational::Rational(int num, int den)
    : numerator_{num}, denominator_{den} {
    //...
}

//...

bool operator==(Rational a, Rational b) {
    //...
}
```

- **Dovete** dividere interfaccia e implementazione
  - **Consideriamo errore se non viene fatto**

Divisione interfaccia e implementazione

- Dividete interfaccia e implementazione:
  - Nell'interfaccia di una classe vengono implementate inline solo le funzioni di 1 o 2 righe
  - Il resto va nel file `.cpp`

Inclusione dei file header

- Si includono solo i file header (`.h`), non i file sorgente (`.cpp`)

Rational.h

```
#ifndef RATIONAL_H
#define RATIONAL_H

class Rational {
  public:
      Rational(int num, int den);
      int numerator(void) const;
      int denominator(void) const;
      //...

  private:
      int numerator_;
      int denominator_;
};

bool operator==(Rational a, Rational b);
#endif
```

Rational.cpp

```
#include "Rational.h"
#include "Rational.cpp"

Rational::Rational(int num, int den)
    : numerator_{num}, denominator_{den} {
    //...
}

//...

bool operator==(Rational a, Rational b) {
    //...
}
```

Ricordarsi le include guards negli headers

Rational.h

```
#ifndef RATIONAL_H
#define RATIONAL_H

class Rational {
  public:
      Rational(int num, int den);
      int numerator(void) const;
      int denominator(void) const;
      //...

  private:
      int numerator_;
      int denominator_;
};

bool operator==(Rational a, Rational b);
#endif
```

include guards

Rational.cpp

```
#include "Rational.h"

Rational::Rational(int num, int den)
    : numerator_{num}, denominator_{den} {
    //...
}

//...

bool operator==(Rational a, Rational b) {
    //...
}
```

- Gli headers delle vostre classi **devono essere protetti dalle include guards**

Cosa passare al compilatore?

```
ltonin@ltonin-laptop$ ls
main.cpp  Rational.cpp  Rational.h
ltonin@ltonin-laptop$ g++ Rational.cpp Rational.h main.cpp -o main
ltonin@ltonin-laptop$ ls
main.cpp  Rational.cpp  Rational.h main
```

- Al compilatore si passano solo i file sorgente (`.cpp`), non gli header
- **Perché?**
- Eccezione: i template, ma li vedremo in seguito

Cosa fa il compilatore?

```
ltonin@ltonin-laptop$ ls
main.cpp  Rational.cpp  Rational.h
ltonin@ltonin-laptop$ g++ Rational.cpp -o Rational
/usr/bin/ld: /usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-
gnu/Scrt1.o: in function `_start':
(.text+0x24): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

- **Perché?**
- Il compilatore crea i file oggetto (`.o`) e li linka
  - Cioè crea un eseguibile
  - Ogni eseguibile ha bisogno di un entry point - la funzione `main`

Dati e funzioni membro private

Date.h

```
class Date {
  public:
      Date (int yy, int mm, int dd);
      void add_day(int n);
      int month(void) const;

  private:
      int y, m, d;
};
```

Date.cpp

```
// ...

void Date::add_day(int n) {
    // ...
    d = d + n;
    // ...
}

// ...
```

- Ricordatevi che se le funzioni membro sono definite al di fuori della classe:
  - **Devono utilizzare l'operatore di scopo ::**

Dati e funzioni membro private

Date.h

```
class Date {
  public:
      Date (int yy, int mm, int dd);
      void add_day(int n);
      int month(void) const;

  private:
      int y, m, d;
};
```

Date.cpp

```
// ...

void Date::add_day(int n) {
    // ...
    d = d + n;
    // ...
}

// ...
```

- Le funzioni membro all'interno della classe **possono accedere ai dati membro anche se privati**

#include ridondanti

```
#include <iostream>

class Date {
  public:
      Date (int yy, int mm, int dd);
      void add_day(int n);
      int month(void) const;

  private:
      int y, m, d;
};
```

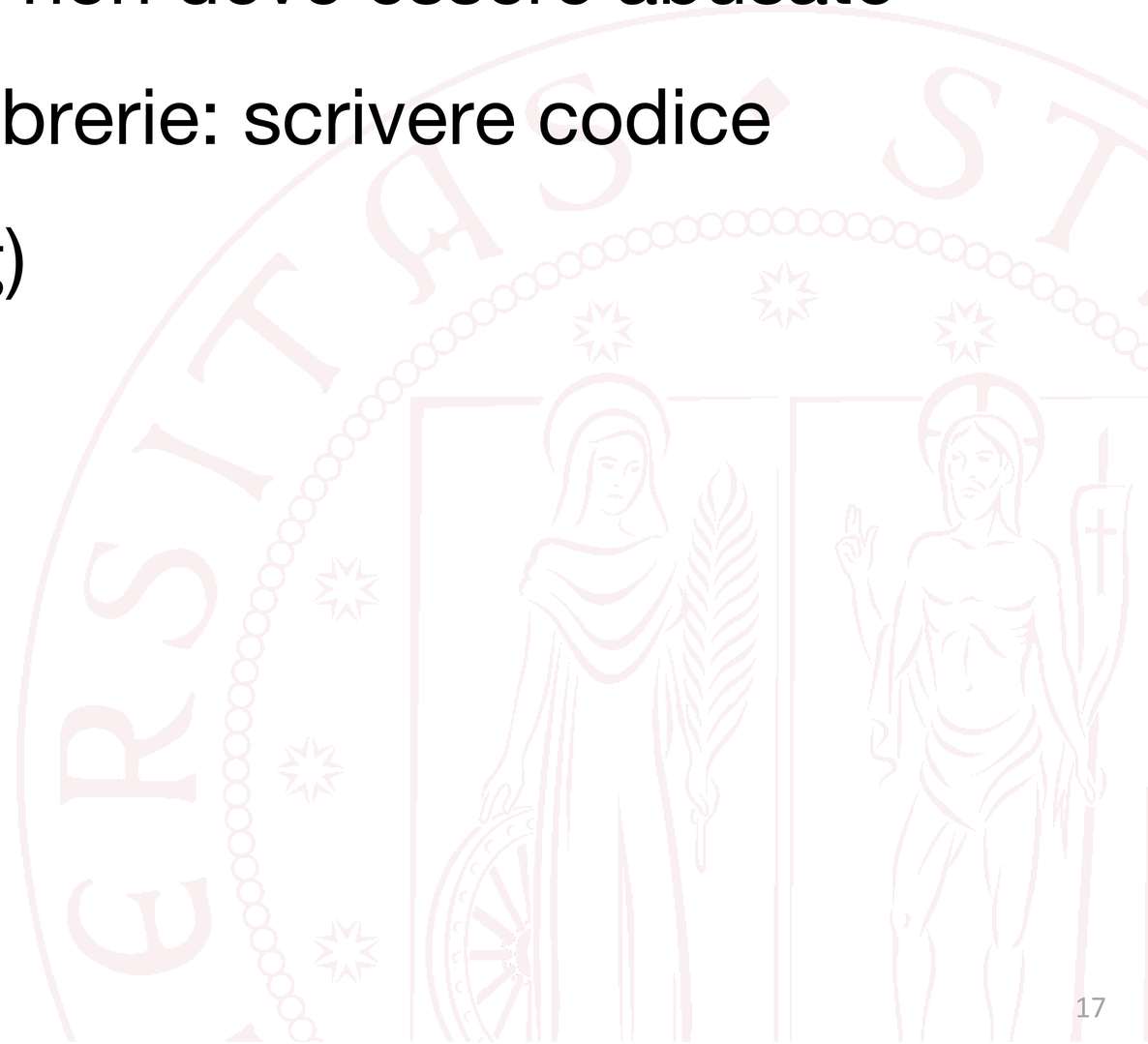
- Data questa interfaccia, la classa `Date` non ha bisogna dell'include di `iostream`!

Less is better

- Aggiungere `#include`/funzioni/dati/costruttori/ecc quando non servono è cattiva programmazione, e quindi sbagliato
  - Aumenta il rischio di introdurre errori
  - Aumenta le linee di codice e la complessità del programma

Limitate l'uso delle direttive `using`

- `using namespace std;` è utile ma non deve essere abusato
- Cercare di evitarlo all'interno delle librerie: scrivere codice esplicito (ad esempio: `std::string`)



Errori comuni

- Potete usare qualsiasi editor di testo per scrivere il vostro codice:
  - CodeLite
  - gedit
  - vim, gvim
  - nano
- L’importante è che vi troviate a vostro agio e che non rallenti la scrittura!

Errori comuni

```
#include <iostream>

int main(void) {
  int a = 3;
  std::cout << "a vale: " << a << "\n";
  return 0;
}
```

OK

```
#include <iostream>

int main(void) {
  int a = 3;
  std::string s = "a vale: ";
  s = s + a + "\n";
  std::cout << s;
  return 0;
}
```

NO!

Leggete le consegne

- Leggete con attenzione le consegne degli esercizi/progetti/esami
- Se, ad esempio, viene richiesto un file separato per il `main`

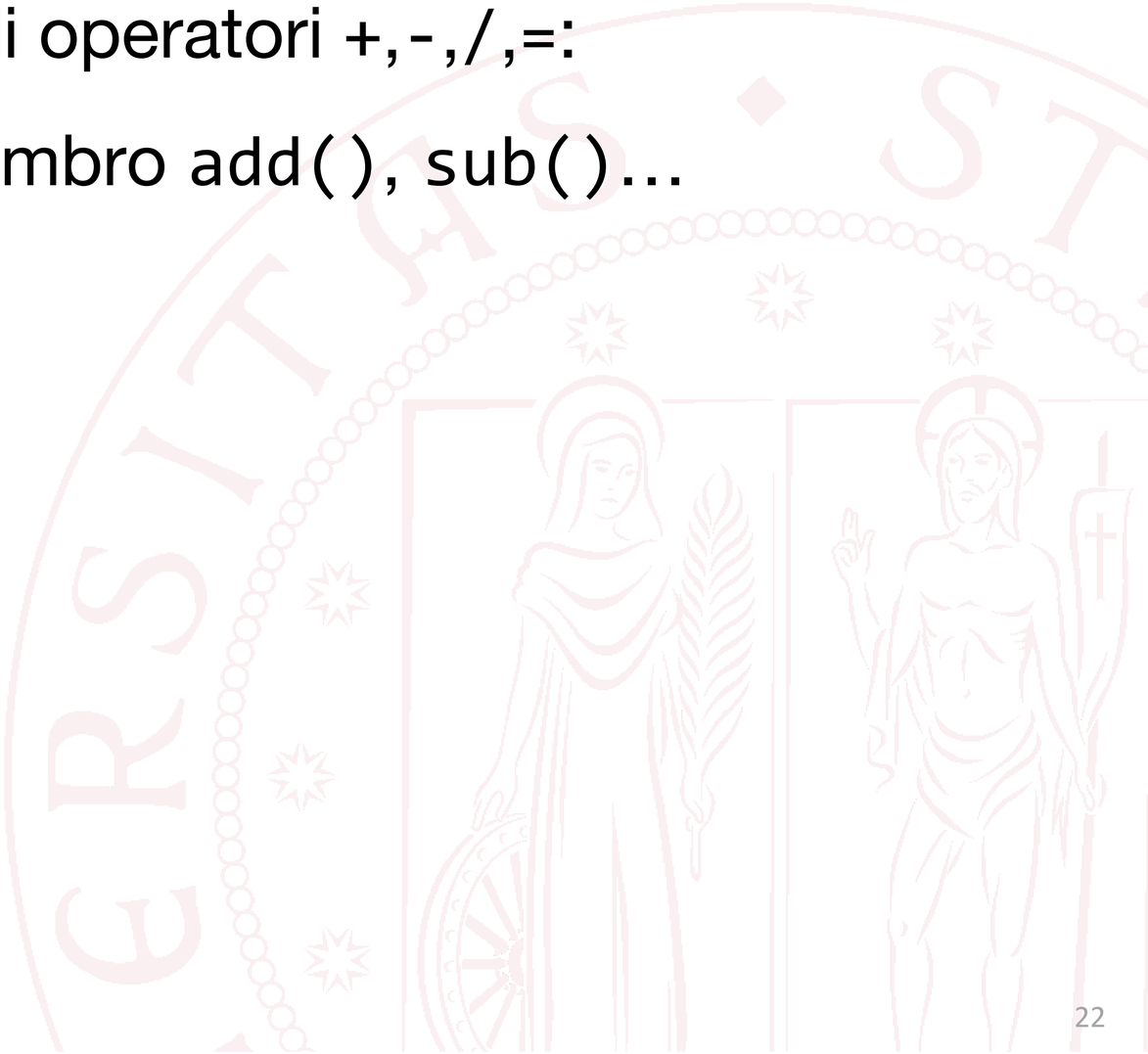
Rational.h

```
class Rational {
  public:
      Rational (int num, int den);
  private:
        int num;
        int den;
};

int main(void) {
    //...
    return 0;
}
```

- Questo viene considerato errore! Oltre che essere concettualmente errato

Leggete le consegne

- Leggete con attenzione le consegne degli esercizi/progetti/esami
- Se vi viene chiesto di fare l'overload degli operatori +,-,/,=:
  - Non dovete implementare funzioni membro `add()`, `sub()`…



Errori comuni

```
int main(void)
{
  std::vector<int> v;
  return 0;
}
```

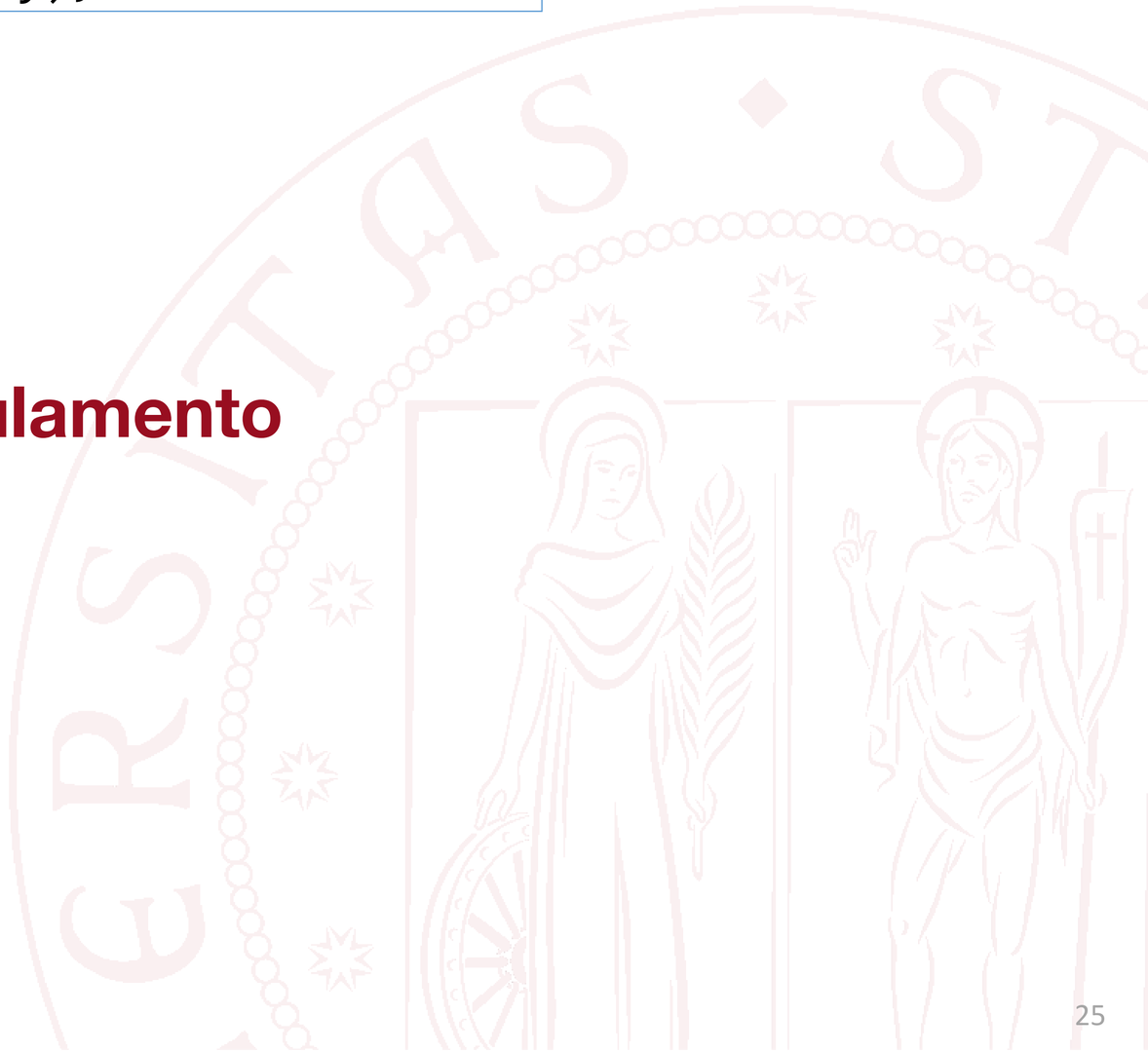
```
#include <vector>

int main(void)
{
  vector<int> v;
  return 0;
}
```

- Per usare gli `std::vector`, bisogna includere l'header `#include <vector>`
- I `vector` sono definiti nel namespace `std::`

friend

```
friend void semplifica(Rational& y);
```

- Evitare il più possibile l'uso di `friend`
  - **Non rispetta il principio dell'incapsulamento**
  - **Rende le classi meno sicure**



Orrori comuni

```
#include <iostream>

int main(void) {
  std::int a = 3;
  std::cout << "a vale: " << a << "\n";
  return 0;
}
```

- Studiate i namespace!
- `int`, `char`, `bool`, `double`, `float` sono tipi built-in

Uso di caratteri speciali nei nomi delle variabili

Il primo carattere di un identificatore valido deve essere uno dei seguenti:

- lettere latine maiuscole A-Z
- lettere latine minuscole a-z
- trattino basso (underscore)

Qualsiasi altro carattere di un identificatore valido deve essere uno dei seguenti:

- cifre 0-9
- lettere latine maiuscole A-Z
- lettere latine minuscole a-z
- trattino basso (underscore)

Dimensione dei tipi

Un long int può avere la stessa dimensione di un int
-> Attenzione all'overflow

Type specifier	Equivalent type	Width in bits by data model				
		C++ standard	LP32	ILP32	LLP64	LP64
signed char	signed char	at least 8	8	8	8	8
unsigned char	unsigned char					
short	short int	at least 16	16	16	16	16
short int						
signed short						
signed short int						
unsigned short	unsigned short int					
unsigned short int						
int	int	at least 16	16	32	32	32
signed						
signed int						
unsigned	unsigned int					
unsigned int						
long	long int	at least 32	32	32	32	64
long int						
signed long						
signed long int						
unsigned long	unsigned long int					
unsigned long int						
long long	long long int (C++11)	at least 64	64	64	64	64
long long int						
signed long long						
signed long long int						
unsigned long long	unsigned long long int (C++11)					
unsigned long long int						

Presenza di junk files nelle consegne

Occhio all'indentazione

- Un'indentazione sbagliata non causa alcun errore, ma in programmi più complessi può ostacolare la comprensione del codice!
- È una buona abitudine indentare correttamente il codice che scriviamo

```
int main()
{
/*********************************************
 *  3.a
 */
int oneInt = 4 ;
double oneDouble = 5.5 ;

print_reference(oneInt,oneDouble);
print_pointer(&oneInt,&oneDouble);
```

```
void f_illegal() {
    int arr[10];

    int* p = &arr[2];



        p[1000] = (1000 + 2) * 10;



        cout << p[1000] << " ";

    cout << endl;

}
```

Variabili globali al posto di membri privati

- Le variabili necessarie alla classe non devono **MAI** essere definite come variabili globali!
- **Devono** essere definite come **membri privati** all'interno dell'interfaccia.

```
# include<iostream>
int sz;
double* elem;


class MyVector{

public:
    MyVector(int size) : sz{size}, elem{new double[size]} {}
```

ERRORE GRAVE

Variabili d'istanza definite globali

- La funzione `reserve(...)` dovrebbe essere **public** oppure **private**?

- Un utilizzatore di MyVector dovrebbe poterla utilizzare oppure no?

```
class MyVector {
private:
    int sz;
    int cap;
    double* data;

public:

    MyVector();
    ~MyVector();
    MyVector(const MyVector& other);
    MyVector& operator=(const MyVector& other);

    // --- Gestione Capacità ---
    int size() const;
    int capacity() const;

    /*
     * Richiede che la capacità del vettore sia almeno new_cap
     * Se new_cap è maggiore della capacità attuale, avviene una riallocazione
     * Non fa nulla se la capacità attuale è già sufficiente
     */
    void reserve(int new_cap);
```