4.3 – Overloading e interfacce

Libro di testo:

- Capitolo 9.6, 9.7

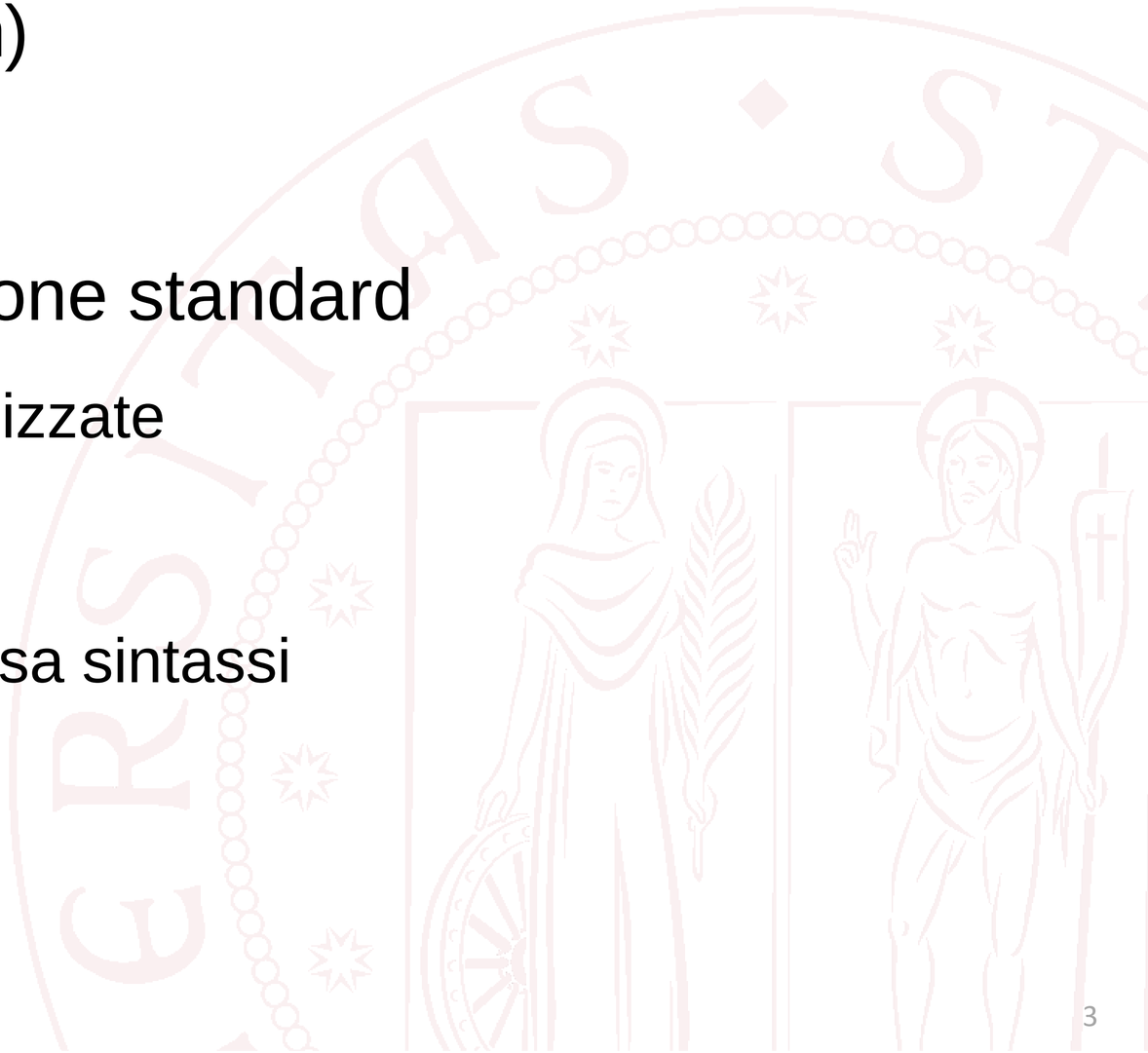
DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Agenda

- Overloading degli operatori
- Progettare una buona interfaccia
- Protezione dei dati
- Valori di default
- Funzioni membro costanti

Overloading degli operatori

- In C++ è possibile implementare quasi tutti gli operatori per operandi di tipo definito dall'utente (`class` o `enum`)
- **Operator overloading**
- Può essere effettuato usando una funzione standard
  - Anche le funzioni membro sono spesso utilizzate
- Overloading solo di operatori esistenti
  - Con lo stesso numero di operandi e la stessa sintassi



Overloading degli operatori

- L'overloading è implementato creando una funzione con un nome specifico
  - `operator+`, `operator++`, `operator*`, `operator[]`, ...
- Il C++ riconosce questo pattern e traduce

```
Date d {2010, Mon::feb, 21};

++d;    // equivalente a operator++(d),
        // oppure a d.operator++()
```

Overloading degli operatori

- Deve essere presente **almeno un argomento UDT**
  - Non è possibile definire `int operator+(int, int);`

Overloading di operatore built-in già esistente!

- Uno strumento potente, ma:
  - È importante non definire operatori con significati contro-intuitivi
  - È sensato definire gli operatori solo se hanno concettualmente senso per quel tipo di dato
- Gli operatori più utili: `=`, `==`, `!=`, `<`, `>`, `[]`, `()`

Overloading degli operatori

- Ogni operatore ha il suo pattern
  - Il pattern è diverso se è funzione membro o funzione esterna
- Operatori implementati con funzione esterna
  - `operator+`, `operator-`, `operator*`, `operator/`: due argomenti
  - Almeno uno UDT, l'altro può essere anche built-in
- `operator<<`: un argomento `ostream&`, ritorna `ostream&`
- `operator>>`: un argomento `istream&`, ritorna `istream&`

Overloading operator++

- `operator++` ha due implementazioni
  - Pre-incremento
  - Post-incremento

```
T& operator++(T& t);      // preincremento
T operator++(T& t, int);  // postincremento: argomento
                          // int dummy
```

- Post-incremento ritorna una copia del valore precedente (non è `lvalue`)
  - Non è richiesto questo tipo di `retval`, ma è comune

Che differenza c'è fra pre-incremento e post-incremento?

Overloading operatore++

Pre-incremento:

```
Month& operator++(Month& m) {

 m = (m == Month::dec) ? Month::jan : Month(int(m) + 1);

 return m;
}
```

Post-incremento:

```
Month operator++(Month& m, int) {

 Month m_temp = m;
 m = (m == Month::dec) ? Month::jan : Month(int(m) + 1);
 return m_temp;
}
```

- Notare l'operatore ?:
  - `m` diventa `jan` se `m == Month::dec`, altrimenti `Month(int(m) + 1)`
- `operator++` può ritornare `Month` o `Month&` (pre- o post- incremento)
  - Il comportamento è diverso!

Overloading operatore++

- Esempio di pre- e post-incremento:

```
#include <iostream>

int main(void) {

    int a = 0;
    int b = 0;

    std::cout << "Pre-increment: " <<  ++a << "; Post-increment: " << b++ << std::endl;

    return 0;
}
```

- Qual è l'output?

```
Pre-increment: 1; Post-increment: 0
```

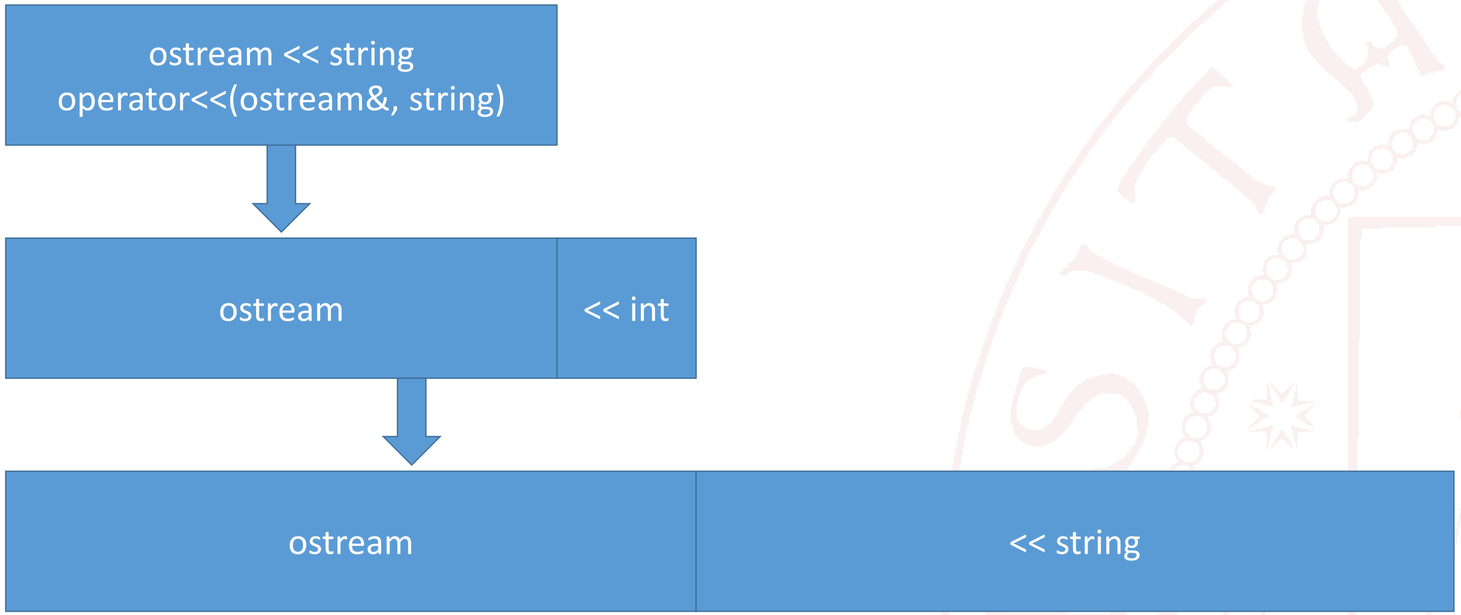
Overloading operatore<<

```
std::vector<std::string> month_tbl;                    // month_tbl inizializzato con le
                                                       // stringhe dei nomi dei mesi

std::ostream& operator<<(std::ostream& os, Month& m)
{
    return os << month_tbl[int(m)];
}
```

- `operator<<` deve ritornare lo stream passato in input

Overloading operatore<<

```
int i = 4;

cout << "Il valore " << i << " è stato scritto in i\n";
```



Interfaccia

Una buona interfaccia

- Alcuni principi generali
  - L'interfaccia deve essere completa
  - L'interfaccia deve essere minimale
  - Devono essere forniti i costruttori
  - La copia deve essere supportata o proibita
  - Usare tipi adeguati per controllare gli argomenti
  - Identificare le funzioni membro costanti
  - Liberare tutte le risorse nel distruttore

Tipi degli argomenti

- Il costruttore di `Date` richiede tre `int`
  - Questo crea qualche problema?

```
Date d1 {4, 5, 2005};
Date d2 {2005, 4, 5};
```

- È possibile (facile?) scambiare i tre argomenti
  - In certi casi porta a risultati assurdi (es. mese 30)
  - In altri casi porta a risultati sbagliati ma plausibili (es. scambio di mese 4 e giorno 5)
  - Anche peggio!

Tipi degli argomenti

Il problema si risolve usando la `enum class Month`

```
class Date {
  public:
      Date (int y, Month m, int d);
      // ...

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

Tipi degli argomenti

- Passaggio di rappresentazione da `int` a tipo dedicato
- Ora il mese ha un suo tipo
  - Il compilatore genera un errore se inserisco il tipo sbagliato
  - I nomi simbolici (`jan`, `feb`, …) sono più descrittivi e immediati da leggere

Esempi di protezione

```
Date dx1{1998, 4, 3};              // errore: 2° arg. non mese
Date dx2{1998, 4, Month::mar};     // stesso errore
Date dx3{4, Month::mar, 1998};     // run-time error
                                   // (giorno 1998)
Date dx4{Month::mar, 4, 1998};     // errore: 2° arg. non mese
Date dx5{1998, Month::mar, 30};    // ok
```

- Gli errori sul secondo argomento sono rilevati a tempo di compilazione
  - Più facile da debuggare
  - Rilevati prima

Protezione di anno e giorno

- Anno e giorno non possono essere protetti nello stesso modo
  - La natura di entrambi è numerica! E l'anno 4 esiste
- Una possibilità è restringere il range di anni accettabili

class Year

```
class Year {
    static const int min = 1800;
    static const int max = 2200;

    public:
        class Invalid {};
        Year(int x) : y{x} {
            if (x < min || max <= x) throw Invalid{};
        }
        int year() { return y; }
    private:
        int y;
};
```

- `min` e `max` sono `static const`
  - Una sola copia in tutto il programma (`static`)
  - Sarebbe stato possibile usare `constexpr`

class Year

```
class Date {
    public:
        Date (Year y, Month m, int d);
        // ...

    private:
        Year y;
        Month m;
        int d;
};
```

- La `class Year` rileva gli errori a **run-time**!

```
Date dx {Year{4}; Month::mar, 1998};     // Year::Invalid
```

- Un approccio efficace?

Copia di oggetti

Copia di oggetti

- La copia è un'operazione molto comune
  - La più comune dopo costruzione e inizializzazione
- Cosa significa copiare un oggetto?
  - Default: copia di tutti i suoi membri

```
Date holiday{2022, Month:oct, 31};      // inizializzazione
Date d2 = holiday;
Date d3 = Date{2022, Month:nov, 1};
holiday = Date{2022, Month:dec, 8};     // assegnamento
d3 = holiday;
```

  - Ma non è sempre la scelta corretta

Costruttore di default

Costruttore di default

- La `classe Date` ha un costruttore che accetta tre argomenti
  - Ciò rende illegale la definizione senza argomenti

```
Date d0;                        // errore: nessuna inizializzazione
Date d1 {};                     // errore: inizializzatore vuoto
Date d2 {1998};                 // errore: argomenti insufficienti
Date d3 {1, 2, 3, 4};           // errore: troppi argomenti
Date d4 {1, "jan", 2};          // errore: tipi sbagliati
Date d5 {1, Month::jan, 2};     // ok!
Date d6 {d5};                   // ok: costruttore di copia
```

Costruttore di default

- Spesso esistono dei valori di default significativi. Per esempio:
  - `std::string`: stringa vuota
  - `std::vector`: vettore vuoto
- In questi casi ha senso prevedere un costruttore di default
- Costruttori di default per tipi built-in: inizializzazione a 0

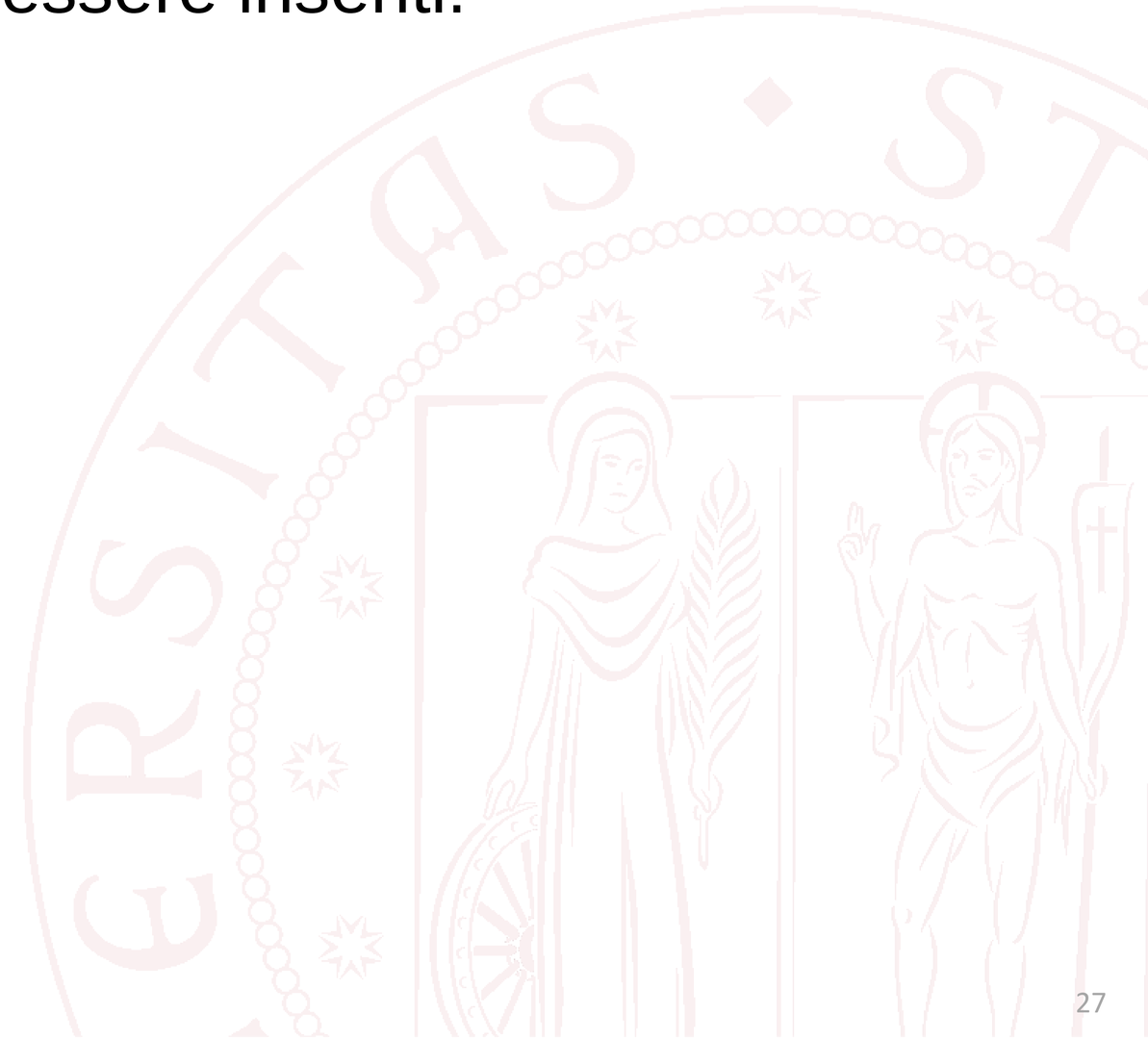
Costruttore di default

Esempi di costruttori di default:

```
// chiamate esplicite
std::string s1 = std::string{};
std::vector<std::string> v1 = std::vector<std::string> {};

// equivalenti, ma più compatte e semplici
std::string s1;
std::vector<std::string> v1;

// tipi built-in
int {};    // int inizializzato a 0
double {}; // double inizializzato a 0.0
```

Valore di default nelle classi

- I valori di default dei membri possono essere inseriti:
  - Usando il costruttore di default
  - Nella definizione dei membri stessi
    - Valgono anche per gli altri costruttori



Valore di default nelle classi

```
class Date {
  public:
      // ...
      Date();
      Date(int y, Month m, int d);
      Date(int y);
      // ...
  private:
      int y {2001};
      Month m {Month::jan};
      int d {1};
};
```

In-class initializers

`Date(int y)` sfrutta i valori di default di `m` e `d`

Funzioni membro costanti

Funzioni membro const

Recall: argomenti passati per reference e per const reference

```
void some_function(Date& d, const Date& start_of_term)
{
    int a = d.day();                  // ok
    int b = start_of_term.day();      // dovrebbe essere ok
    d.add_day(3);                     // ok
    start_of_term.add_day(3);         // errore
}
```

È un problema?
Il compilatore riporta un errore.
Perché?

Funzioni membro `const`

- La chiamata a `Date::day()` applicata a un `const Date&` non garantisce che l'oggetto non sarà modificato
  - `day()` in effetti non modifica l'oggetto, ma il compilatore non lo sa!
- Soluzione: dichiarare `day()` come funzione membro `const`

Funzioni membro const

```
class Date {
public:
    // ...
    int day() const;
    Month month() const;
    int year() const;

    void add_day(int n);
    void add_month(int n);
    void add_year(int n);
    // ...
private:
    int y;
    Month m;
    int d;
};
```

Funzioni membro const

Modificare un membro in una funzione const causa un errore di compilazione

```
int Date::day() const
{
    ++d;                // errore! Modifica in una funzione const
    return d;
}
```

Funzioni membro vs helper functions

Funzioni membro vs helper function

"A function that can be simply, elegantly, and efficiently implemented as a freestanding function (that is, a non-member function) should be implemented outside the class. That way, a bug in that function cannot directly corrupt the data in a class object." (BS)

In caso di dati membro corrotti, solo le funzioni membro devono essere controllate

Rendere una funzione membro della classe se e solo se necessita di un diretto accesso alla rappresentazione della classe

Funzioni membro vs helper function

- Quante funzioni sono ragionevoli per la `class Date`?
  - In molti casi: anche 50
- "A few years ago I surveyed a number of commercially used Date libraries and found them full of member functions like `next_sunday()`, `next_workday()`, etc. Fifty is not an unreasonable number for a class designed for the convenience of the users rather than for ease of comprehension, implementation, and maintenance." (BS)

Helper function

```
Date next_sunday(const Date& d)
{
    // accedere utilizzando d.day(), d.month(), d.year()
    // creare una nuova Date da ritornare
}

Date next_weekday(const Date& d) { /* ... */ }
bool leapyear(int y) { /* ... */ }

bool operator==(const Date& a, const Date& b)
{
    return a.year() == b.year()
           && a.month() == b.month()
           && a.day() == b.day();
}

bool operator!=(const Date& a, const Date& b)
{
    return !(a==b);
}
```

Helper function

- Le helper function utilizzano spesso argomenti della classe di cui sono helper
  - Non sempre, non una regola!
- `operator==` e `operator!=` sono tipiche helper function
- Spesso le helper function sono inserite in un `namespace` assieme alla classe

```
namespace Chrono {
enum class Month { /* ... */ }
class Date  { /* ... */ }
Date next_sunday(const Date& d) { /* ... */ }
Date next_workday(const Date& d) { /* ... */ }
// ...
}
```

Recap

- Overloading degli operatori
- Sintassi per gestire l'overloading
  - Distinzione tra preincremento e postincremento
- Overloading con helper function e funzione membro
- Operatore ternario
- Regole per l'overloading di alcuni operatori
  - Argomenti, tipi restituiti