4.2 – La classe Date

Esempio: la classe Date

Libro di testo:

- Capitolo 9.4, 9.5
- Capitolo 5

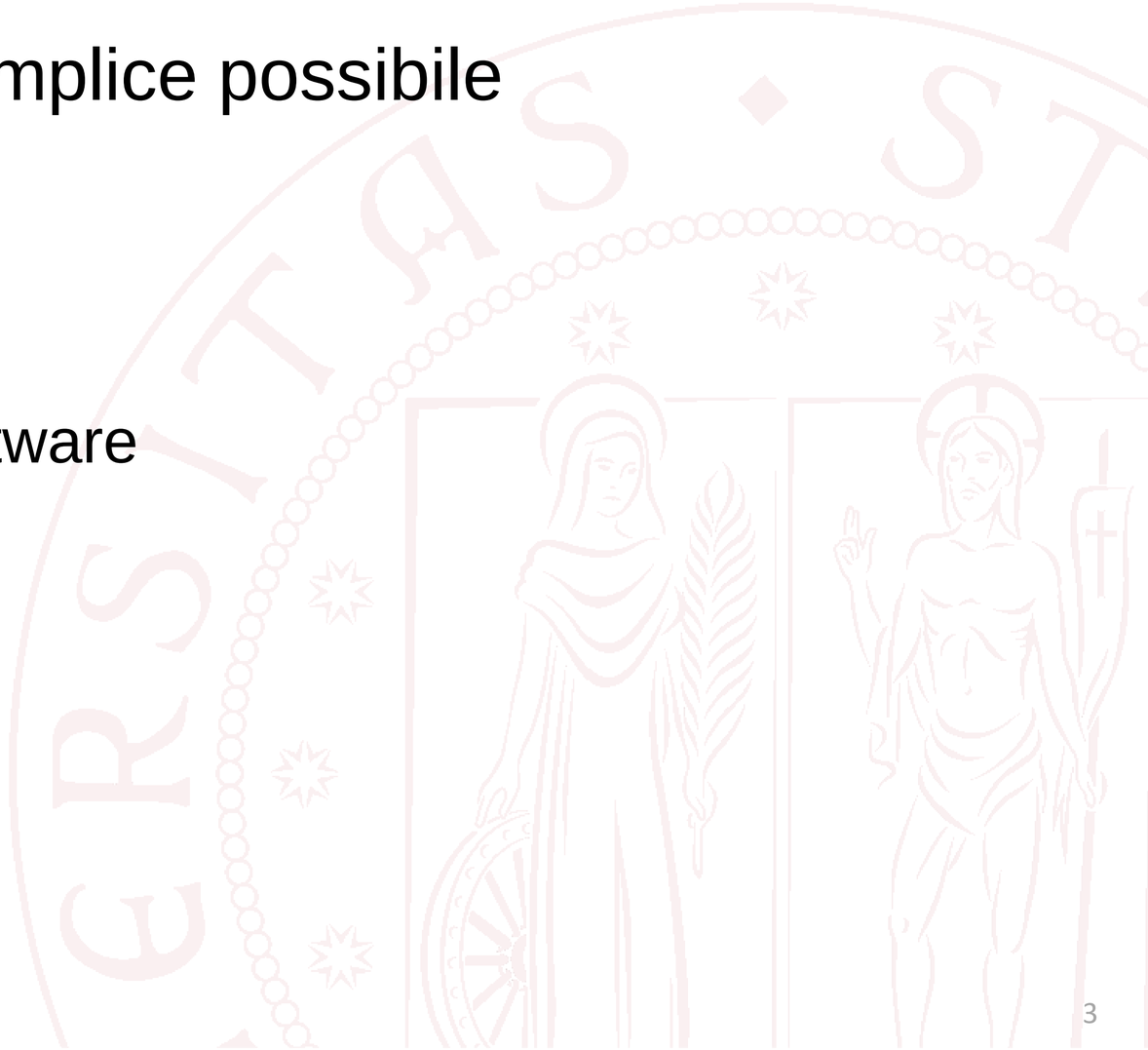
DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Agenda

- Progettare una classe con un esempio: la classe `Date`
- Helper function
- Funzioni membro
- Costruttori e inizializzazione
- Membri pubblici e privati
- Verifica degli errori
- Le enumerazioni

Progettazione di una classe

- Ora creiamo la **classe `Date`** per rappresentare e manipolare le date
- Partiamo da una struttura dati il più semplice possibile
- Tramite un processo iterativo:
  - Rendiamo la classe più utile e sicura
  - Simile al reale processo di sviluppo del software
  - Ogni iterazione prevede una fase di test!



La classe `Date`

- Dobbiamo disegnare una classe `Date`:
  - Cosa rappresenta una classe `Date`?

    “Una data deve rappresentare un giorno nel passato, presente o futuro”

  - Quali variabili membro saranno necessarie?
  - Di che tipo?

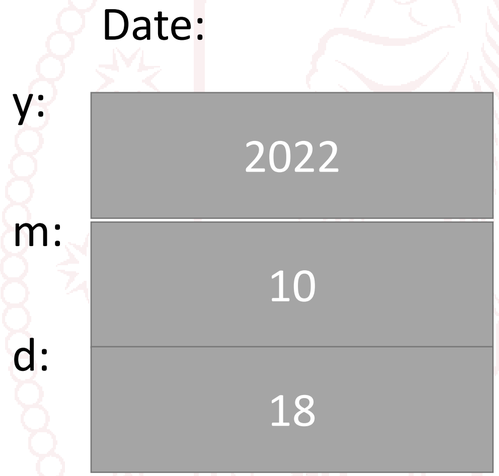
La classe Date - 1

```
struct Date
{
    int y; // anno
    int m; // mese
    int d; // giorno
};

Date today;
```

Nomi troppo corti?
Perché?

```
today.y = 2022;
today.m = 10;
today.d = 18;
```



Critica

- Quali sono le criticità con oggetti di questa `struct`?
  - Possiamo fare qualsiasi cosa con i suoi dati membro!
- L'utilizzo è lento e non molto sicuro

```
Date x;

x.y = -3;
x.m = 13;
x.d = 32;

// questo codice è sintatticamente corretto? Compila? Esegue?
```

Critica

```
Date y;

y.y = 2000;
y.m = 2;
y.d = 29;

// questo codice è sintatticamente corretto? Compila? Esegue?
```

- Ci serve un metodo per evitare queste situazioni
- Idee?

Helper function

- Funzioni che eseguono le operazioni più comuni
  - Al posto nostro!
  - Programmate una sola volta
  - Debuggate una sola volta!
- Quali operazioni potremmo implementare?
  - Inizializzazione!
  - Incremento di un giorno

Helper function

- Immaginiamo due funizioni comuni per una classe `Date`:

```
void init_day(Date& dd, int y, int m, int d)
{
    // verifica che y, m, d siano una data corretta
    // se sì, inizializza
}

void add_day(Date& dd, int n)
{
    // incrementa dd di n giorni
}
```

```
int main(void)
{
    Date today;
    init_day(today, 12, 24, 2005);   // errore!
    add_day(today, 1);
    return 0;
}
```

Helper function | soluzione definitiva?

```
int main(void)
{
    Date today;
    // ...
    std::cout << today << '\n';  // << definito per Date ?
    // ...
    init_day(today, 2008, 3, 30);
    // ...
    Date tomorrow;
    tomorrow.y = today.y;
    tomorrow.m = today.m;
    tomorrow.d = today.d + 1;
    std::cout << tomorrow << '\n’;
    return 0;
}
```

today non inizializzato ma usato

Ci siamo dimenticati di inizializzare `Date` prima di usarlo!

tomorrow costruito a mano

Dato che la classe `Date` lo permette, abbiamo inizializzato `tomorrow` a mano

Cosa succede se `today.d=31` e aggiungo un giorno?

Codice molto scorretto, anche se a prima vista non sembra (ancora peggio!)

Helper function | soluzione definitiva?

- Esempio precedente:
  - Le helper function sono a disposizione dell'utente
  - Ma possono essere bypassate

    (cioè l'utente può decidere di non usarle)
- È necessario rendere più visibili le helper function
- Bisogna rendere il loro uso necessario
  - Come?

Date – 2 (costruttore e funzioni membro)

```
struct Date
{
    int y, m, d;
    Date(int y, int m, int d);
    void add_day(int n);
};
```

- Il **costruttore** è una funzione membro con lo stesso nome della classe
  - Usato per inizializzare (costruire) gli oggetti della classe

- Se esiste solo un costruttore con argomenti:
  è errore di compilazione non fornire tali argomenti

Costruttori e funzioni membro

```
Date my_birthday;                           // errore: non inizializzato

Date today{12, 24, 2007};                   // run-time error (da verifica
                                            // del costruttore)

Date last {2000, 12, 31};                   // ok, colloquiale

Date next = {2014, 2, 14};                  // ok, un po' verboso

Date christmas = Date{1976, 12, 24};        // ok, un po' verboso

Date last(2000, 12, 31);                    // ok: colloquiale ma vecchio
                                            // stile

last.add_day(1);                            // ok

add_day(2);                                 // errore: a quale oggetto
                                            // fa riferimento?
```

Inizializzazione

- La sintassi {} è preferibile per l'inizializzazione (è dedicata)

```
int x {7};

Date next {2014, 2, 14};
```

Mantenere privati i dettagli

```
struct Date
{
    int y, m, d;
    Date today(int y, int m, int d);
    void add_day(int n);
};
```

```
Date today {2021, 10, 11};
today.d = 32;
```

- Passaggio helper function → funzioni membro
  - Le funzioni sono più evidenti
  - **Non è ancora obbligatorio usarle (male!)**
- È necessario **proteggere i dati membro** per evitare che un uso scorretto invalidi lo stato dell'oggetto
- Le funzioni che abbiamo visto devono diventare l'unico modo per accedere ai dati → concetto di interfaccia

Controllare l’accesso ai membri

Date – 3 (controllo di accesso)

- Finché lasciamo accessibili i dati membro della classe, chiunque può modificarli a piacimento (per errore o intenzionalmente)
- Soluzione: proteggere i dati membro rendendoli privati

```
class Date {
    int y, m, d;

    public:
        Date (int y, int m, int d);
        void add_day(int n);
        int month(void) { return m; }
        int day(void) { return d; }
        int year(void) { return y; }
};

Date birthday {1970, 12, 30};            // ok
birthday.m = 14;                         // errore: Date.m è privato
std::cout << birthday.month() << '\n';   // ok
```

Interfaccia della classe

`Date` – controllo di accesso

- Le funzioni membro garantiscono che i dati membro siano sempre coerenti
  - Suppongono che lo siano in partenza e garantiscono che lo siano in uscita
  - In altre parole, garantiscono che l'oggetto sia sempre in uno **stato valido**

- Alternativa: controllare a ogni accesso/utilizzo, o sperare che l'utente non inserisca valori non validi
- "Experience shows that 'hoping' can lead to 'pretty good' programs" (BS)

Invarianti

- “A rule for what constitutes a valid value is called an invariant” (BS)
- La classe `Date` rappresenta:

  “un giorno nel passato, presente o futuro”

- Sono le regole per definire valori validi dello stato
- Per `Date` non sono ovvi
  - Calendario Gregoriano
  - Anni bisestili
  - Time zone

Date – 4 (riordinato)

- Spesso si scrive prima l'interfaccia e in seguito si inseriscono i dettagli implementativi

Date.h

```
class Date {
  public:
      Date (int y, int m, int d);
      void add_day(int n);
      int month(void);
      int day(void);
      int year(void);

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

Date.cpp

```
#include "Date.h"

Date::Date(int yy, int mm, int dd)
            : y{yy}, m{mm}, d{dd}
{
}

void Date::add_day(int n)
{
    // ...
}

int month(void)  // ops! Manca Date::
{
    return m;
}
```

Definizioni fuori dalla classe

Includere l'header

Date.cpp

```
#include "Date.h"

Date::Date(int yy, int mm, int dd)   // costruttore
    : y{yy}, m{mm}, d{dd}
{
}

void Date::add_day(int n)
{
    // ...
}

int month(void)                      // ops! Manca Date::
{
    return m;
}
```

(Member) initializer list

Includere il riferimento alla classe

Initializer list vs assegnamenti

- Alternativa all'initializer list:

```
Date::Date(int yy, int mm, int dd)
{
    y = yy;
    m = mm;
    d = dd;
}
```

- In questo caso, due passi:
  - Inizializzazione di default
  - Assegnamento di valori
  - Potenzialmente i valori con inizializzazione di default potrebbero essere usati
- La lista di inizializzazione esprime il nostro intento più esplicitamente

Definizioni nella classe

- Le funzioni membro possono essere definite nell'interfaccia della classe:

```
class Date {
  public:
      Date (int yy, int mm, int dd): y{yy}, m{mm}, d{dd} {
        // ...
      }

      void add_day(int n) {
          // ...
      }

    int month(void) { return m; }

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

Definizioni nella classe (`inline`)

- [In-class definitions]
- Rendono la classe più lunga e più disordinata
  - `add_day()` potrebbe essere una decina di righe
- I dettagli implementativi si mescolano con l'interfaccia
- Eventualmente, da farsi solo per funzioni molto brevi
  - `month()` definita nella classe è molto breve
  - Molto più breve della sua definizione esterna `Date::month()`

Definizioni nella classe (`inline`)

- Rendono le funzioni automaticamente **`inline`**
  - Consiglio al compilatore: il codice è replicato a ogni chiamata, anziché implementare una vera chiamata a funzione
  - Utile per funzioni molto brevi, che non fanno quasi nulla, e che sono chiamate spesso
- Una modifica alla funzione impone di ricompilare tutto il codice che usa la classe
  - Non è un dettaglio trascurabile in grandi progetti
- La definizione della classe diventa più grande

Buona pratica (`inline`)

- In generale: non definire funzioni membro nella definizione della classe
- Eccezione: la funzione è molto piccola e desideriamo l'inlining
  - Una funzione con più di 1-2 espressioni non beneficia dell'inlining
- Attenzione: `inline` è solo una richiesta al compilatore, che può essere ignorata

https://www.geeksforgeeks.org/inline-functions-cpp/

Rilevare gli errori

`Date` – 5 (rilevamento dati non validi)

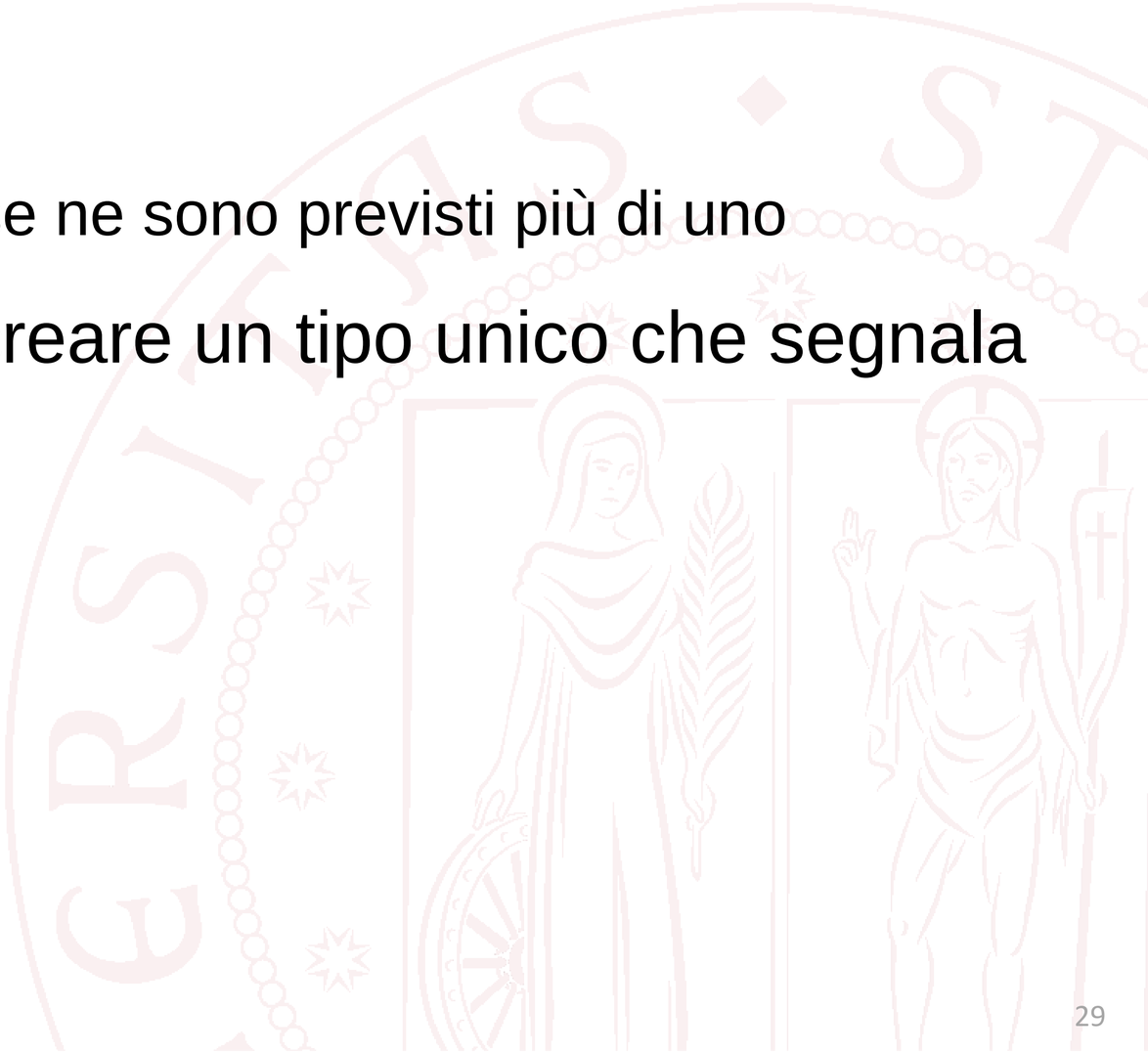
Creiamo una funzione `is_valid()` che verifica se i dati sono validi:

```
class Date {
public:
    class Invalid {};                 // per riportare errori
    Date (int y, int m, int d);  // check e inizializza
    // ...

private:
    int y, m, d;
    bool is_valid(void);              // ritorna true se la data è
                                      // valida
};
```

`is_valid()`

- Funzione separata perché rappresenta un'attività diversa dall'inizializzazione
  - Può essere utilizzata da tutti i costruttori, se ne sono previsti più di uno
- La classe `Invalid` ha il solo scopo di creare un tipo unico che segnala questo specifico errore



Verifica di validità e costruttore

```
Date::Date(int yy, int mm, int dd) : y (yy), m(mm), d(dd)
{
    if (!is_valid()) throw Invalid();
}

bool Date::is_valid()
{
    if (m < 1 || m > 12) return false;
    // ...
}
```

Eccezioni – excursus

- Meccanismo che separa il rilevamento di un errore dalla sua gestione
  - Rilevamento: spesso in una funzione chiamata (inside)
  - Gestione: spesso nella funzione chiamante (outside)
- Le eccezioni garantiscono che l'errore sia gestito
  - Se non è così, il programma termina

Eccezioni

- Se una funzione rileva un errore che non può gestire, **non ritorna normalmente**
- La funzione chiamata lancia (`throw`) un'eccezione
- Un qualche chiamante (diretto o indiretto) recepisce (`catch`) l'eccezione
- Una funzione esprime interesse per le eccezioni usando un blocco `try … catch`
- Ciò che viene lanciato è un oggetto

Eccezioni – try/catch

```
Date::Date(int yy, int mm, int dd)
            : y (yy), m(mm), d(dd)
{
    if (!is_valid()) throw Invalid();
}

bool Date::is_valid()
{
    if (m < 1 || m > 12) return false;
    // ...
}
```

1

2

```
void f(int x, int y)
{
    try {
        Date dxy{2004, x, y};
        std::cout << dxy << '\n';
        dxy.add_day(2);
    }
    catch(Date::Invalid e) {
        std::cout << "Invalid date: " <<
                          e.c_str() << std::endl;
    }
}
```

Enumerazioni

Enumerazioni

- Un enum (enumeration) è un tipo definito dall'utente molto semplice
- Specifica un set di valori
  - Rappresentati con costanti simboliche
  - Una variabile è una scelta tra questo set
- Il corpo è la lista degli enumeratori

```
enum class Month {
    jan = 1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};
```

Enumerazioni

- `enum class` crea uno scope
  - `jan` è specificato come `Month::jan`
- È possibile scegliere valori specifici
  - In caso contrario: valore del precedente + 1
  - Parte da 0
- Un altro esempio:

```
enum class Day {
    monday, tuesday, wednesday, thursday, friday, saturday, sunday
};
```

Assegnamento a enum

```
Month m = Month::feb;

Month m2 = feb;                    // errore: feb non è in scope
m = 7;                             // errore: assegnamento di int a Month
int n = m;                         // errore: assegnamento di Month a int
Month mm = Month(7);               // conversione di int a Month
                                      (non verificata!)
```

- **Posso scrivere `Month(9999)`?**
- **Come gestirlo?**

- Si può scrivere una semplice funzione di check:

```
Month int_to_month(int x) {

    if( x < int(Month::jan) || int(Month::dec) < x )
        error("bad month");
    return Month(x);
}
```

Enumerazioni senza classi

È possibile usare un'enumerazione senza scope:

```
enum Month {
    jan = 1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

Month m = feb;     // non genera errore
```

- **Pericoloso, perché?**

Recap

- Primo esempio di progettazione di una classe
- Helper function
- Funzioni membro e controllo di accesso
- Invarianti
- Costruttori
- Lista di inizializzazione
- Definizione in-class vs esterne
- Inline
- Eccezioni
  - Meccanismo
  - Utilizzo
- Enumerazioni