



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

**Classe Date:  
Controllare l'accesso ai membri**

Stefano Ghidoni



# Agenda

- Membri pubblici e privati
- Costruttori e inizializzazione
- Liste di inizializzazione
- Organizzazione delle funzioni membro



# Mantenere privati i dettagli

- Anche usando le helper function i dati membro sono di libero utilizzo
- Passaggio helper function -> membri
  - Le funzioni sono più evidenti
  - Non è ancora obbligatorio usarle
- 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
  - Interfaccia ☺

Le funzioni pubbliche sono un'interfaccia per toccare i membri privati



# Date – 3 (controllo di accesso)

- L'accesso ai dati privati è protetto

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

Esiste un solo costruttore,  
che accetta tre argomenti,  
quindi lo si può chiamare/si  
può costruire l'oggetto solo  
se si provvedono tre  
argomenti.

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



## 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)



# Invariante

- Il concetto di **stato** è legato a quello di **invariante**
- Invarianti: sono le regole per definire valori validi dello stato
- Per Date non sono ovvi
  - Calendario Gregoriano
  - Anni bisestili
  - Time zone

# Costruttori



# Costruttori

- **Costruttore:** funzione membro con lo stesso nome della classe
  - Usato per inizializzare (costruire) gli oggetti della classe
- Il costruttore ha il compito di inizializzare i dati con uno stato di partenza valido
  - Valori di default validi *Se non forniti dall'utente*
  - Nel caso di argomenti passati al costruttore: controllo di validità

*Il costruttore è un elemento chiave dell'interfaccia*



## Init list

- Le inizializzazioni sono molto comuni nei costruttori
  - Liste di inizializzazione
- Liste di inizializzazione: evidenziano le inizializzazioni

➤ Definizione esterna alla classe

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

(Member) initializer list

# Aggiunta

Un'initializer list è un luogo all'interno del costruttore che sta tra gli argomenti e il corpo.

Nell'elenco ci sono variabili membro inizializzate con la sintassi specificata nella slide precedente.

Questo metodo di inizializzazione è alternativo all'assegnamento (trovato alla slide successiva).

La differenza principale si nota principalmente con gli UDT, piuttosto che con i tipi base. Infatti, in generale, la prima istruzione del costruttore permette di toccare tutti i membri, quindi l'UDT deve già essere costruito alla prima linea. Di conseguenza il costruttore corrente è costretto a usare tutti i costruttori dei dati membri.

Data questa necessità, c'è il modo di bypassare il costruttore di default e creare gli UDT con argomenti differenti; questo modo è proprio usare le initializer list.



# Init list vs assegnamenti

- Alternativa alla init 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 evita il doppio passaggio
- La lista di inizializzazione esprime il nostro intento più esplicitamente

Quando ci sono UDT, lasciamo la facoltà al costruttore di chiamare il costruttore dell'altra classe



# Inizializzazione

- Il costruttore è chiamato in fase di inizializzazione
  - La sintassi {} è preferibile per l'inizializzazione (è dedicata)

```
int x {7};
```

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

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



# Costruttori e funzioni membro

```
Date::Date(int yy, int mm, int dd); // costruttore

Date my_birthday;      // errore: non inizializzato

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

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

Date next = {2014, 2, 14};    // ok, un po' verboso
                           Chiamata esplicita al costruttore
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: quale data?
```

# Organizzazione delle funzioni membro



# Date – 4 (riordinato)

- Riorganizziamo ora Date utilizzando un pattern comune:
  - Prima l'interfaccia
  - In seguito i dettagli implementativi

```
class Date {  
public:  
    Date (int y, int m, int d);      // controlla validità e  
                                      // inizializza  
    void add_day(int n);           // aumenta di n giorni  
    int month();  
    int day();  
    int year();  
  
private:  
    int y, m, d;                  // year, month, day  
};
```



# Definizioni funzioni membro

- Le funzioni membro devono essere dichiarate nella classe e possono essere definite
  - Dentro la classe
  - All'esterno
- Caratteristiche diverse per
  - Organizzazione del codice
  - Inlining
- Vediamo qualche esempio

In generale, i corpi delle funzioni sono scritti esternamente alla classe, così da ridurre la dimensione (in codice) della classe, mantenere la leggibilità e mantenere il "colpo d'occhio"; è possibile definirle nella classe quando sono molto piccole.



# Definizioni nella classe

- [AKA 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 utilizzare **solo per poche funzioni molto brevi**
  - `month()` definita nella classe è molto breve
    - Molto più breve della sua definizione esterna `Date::month()`



# Definizioni nella classe

- Funzioni membro definite nella classe:

```
class Date {  
public:  
    Date (int yy, int mm, int dd)  
        : y{yy}, m{mm}, d{dd}  
    {  
        // ...  
    }  
  
    void add_day (int n)  
    {  
        // ...  
    }  
    int month() { return m; }  
    // ...  
private:  
    int y, m, d;  
};
```



# Definizioni nella classe

- 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



# Definizioni fuori dalla classe

- Le funzioni membro sono spesso definite fuori dalla classe

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

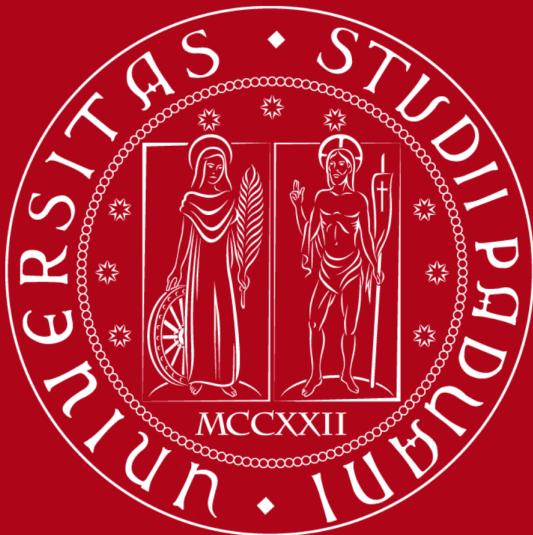
void Date::add_day(int n)
{
    // ...
}

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



## Buona pratica

- 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



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

**Classe Date:  
Controllare l'accesso ai membri**

Stefano Ghidoni



DIPARTIMENTO  
DI INGEGNERIA  
DELL'INFORMAZIONE