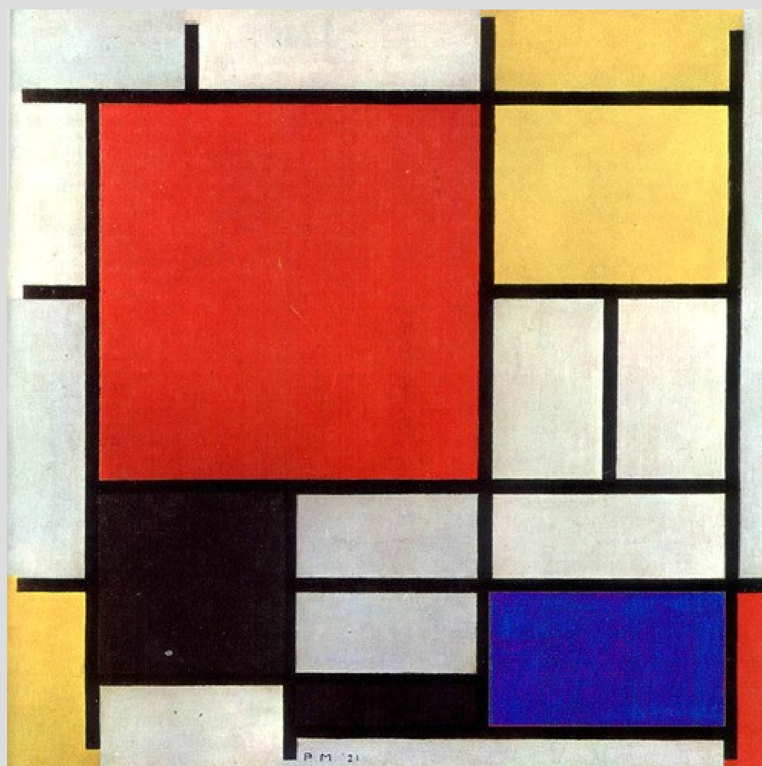
Parte 10

Modularizzazione e sviluppo su più file



[P. Mondrian - Composition with Large Red Plane, Yellow, Black, Grey and Blue, 1921]

Progettazione

- Aumentando di dimensioni, i programmi inevitabilmente crescono anche in **complessità**
- All'aumentare della complessità di un programma, diviene sempre più importante l'**attività di progettazione**
- Abbiamo già effettuato alcune attività di progettazione nella stesura dei nostri programmi
  - Scelta e definizione degli algoritmi da usare
  - Definizione delle funzioni da utilizzare e dei loro parametri di ingresso/uscita, nonché dei valori di ritorno

Livello di astrazione

- Si è trattato di **attività di progettazione a basso livello di astrazione** rispetto alla scrittura del codice
  - Ossia, in termini di astrazione si sono definiti aspetti immediatamente sopra la stesura del codice stesso
- Nel caso di **programmi di medio-grandi dimensioni**, per dominare la complessità e realizzare prodotti di qualità (*chiari, robusti, estensibili,...*) la progettazione deve essere effettuata ad un **livello di astrazione molto più elevato**

Struttura logica e modularità

- In questa lezione vedremo la progettazione ad alto livello solo in termini di **definizione della struttura logica**
- In particolare in termini di suddivisione:
  - in **moduli**
  - in **più file sorgente**

Programma

Realizziamo una piccola **suite di gestione** di sequenze di elementi con le caratteristiche seguenti:

- Ogni **elemento** della sequenza deve contenere:
  - Un campo denominato **chiave**
    - La chiave deve poter contenere un numero intero oppure una parola
    - A tempo di esecuzione deve essere possibile decidere l'interpretazione da dare al contenuto della chiave prima di iniziare ad inserire gli elementi
  - Un **numero variabile**, da elemento ad elemento, di **parole**

Esempi

- Sequenza con chiave numerica:

42	parola1	parola2	
3	primaparola	seconda	terza
15	solounaparola		

- Sequenza con chiave stringa:

parolachiave	parola1	parola2	parola3
22	parola1	parola2	
altrachiave	unasolaparola		

Funzionalità di base

- *Inserimento ed estrazione*
- *(Ri-)Ordinamento*
- *Ricerca di un elemento*
- *Lettura e salvataggio degli elementi della sequenza da/su file*

Funzionalità (1)

- **Inserimento** ed **estrazione** degli elementi
  - L'inserimento deve poter essere effettuato in testa, in fondo o in ordine
- **Riordinamento della sequenza**
  - Eventuale cambio di rappresentazione della sequenza stessa, ad esempio *max heap*
  - Calcolo del costo, in numero di operazioni, di ciascuno dei riordinamenti disponibili
  - **NOTA**: utilizzo del comando **sort** e confronto dei tempi di esecuzione con quelli dei nostri algoritmi di ordinamento

Funzionalità (2)

- **Ricerca di un elemento nella sequenza**
  - La ricerca può essere effettuata per valore della **chiave** o per **elemento di chiave minima o massima**
  - La ricerca viene effettuata **automaticamente** con l'**algoritmo più efficiente disponibile**, in base all'ordinamento/rappresentazione corrente della sequenza
  - Calcolo del costo, in numero di operazioni, della ricerca per valore e per ordinamento in funzione dell'ordinamento / rappresentazione corrente della sequenza

Funzionalità (3)

- **Lettura e salvataggio degli elementi della sequenza da/su file**
  - Il file deve essere di tipo testuale
  - Ogni riga deve contenere un elemento
  - La prima parola della riga fornisce la chiave

Programma

- Il Programma è fornito nella traccia

  Gseq.cc
- Sono implementate *alcune delle funzionalità viste*
  - Manca il (ri)-ordinamento
  - Manca il caricamento da file
- *NOTA*: il formato dei commenti

Funzionalità

Proviamo a **raggruppare logicamente** l'implementazione delle **funzionalità** nel modo seguente

1) Stampa del menù, lettura delle scelte ed invocazione delle relative funzioni - Main
2) Inserimento, ricerca, eliminazione, stampa
3) Caricamento e salvataggio da/su file

Nota

- Ovviamente questo è solo **uno dei tanti possibili raggruppamenti**
- In generale, una delle difficoltà dell'attività di progettazione è dover scegliere all'interno di uno **spazio di soluzioni più o meno vasto**
- Ciascuna soluzione ha i propri **pro e contro**
  - Purtroppo spesso non banali da prevedere

Correlazione

- In definitiva, abbiamo messo insieme le funzioni di inserimento, ricerca, eliminazione e stampa perché le abbiamo ritenute **logicamente correlate**
  - Abbiamo ritenuto che servissero ad implementare un **insieme di funzionalità a loro volta correlate**
- Adottando lo stesso principio, abbiamo considerato logicamente correlate le funzioni per il caricamento ed il salvataggio da/su file

Modulo

- Possiamo definire un **modulo** come un insieme di funzioni e strutture dati **logicamente correlate** in base ad un qualche principio significativo
- Tipicamente un modulo fornisce una **serie di servizi e può implementare una certa struttura dati (o tipo di dato)**
- Torneremo su questi aspetti fra qualche slide...

Moduli

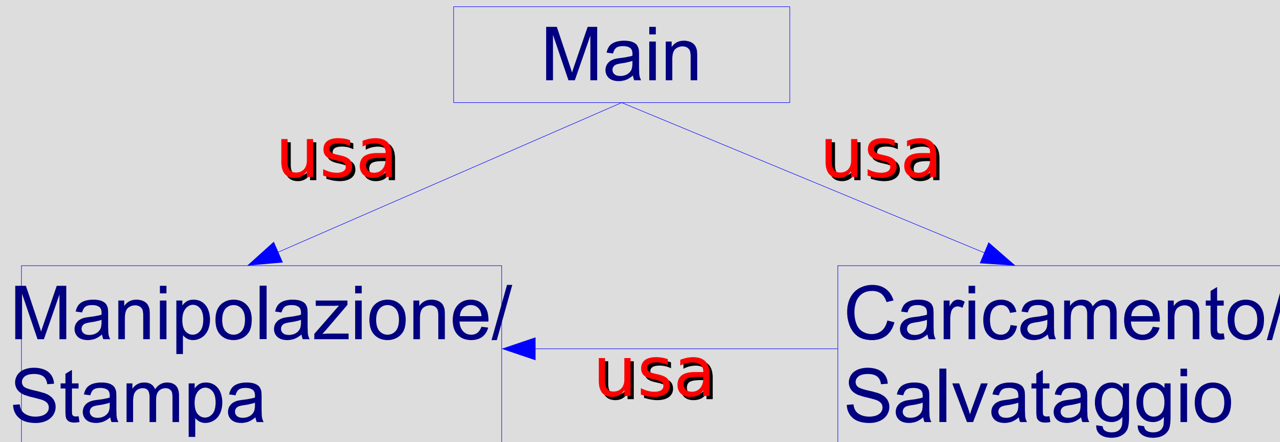
Con la precedente “analisi logica”, abbiamo quindi individuato **3 moduli**, che chiamiamo sinteticamente:

1) **Main**
2) **Manipolazione/Stampa**
3) **Caricamento/Salvataggio**

Moduli utente

- In generale, la relazione fondamentale che c'è tra i moduli di un programma si basa sul fatto che alcuni moduli sono **utenti**, ossia **utilizzano i servizi/oggetti forniti da altri moduli**
- Nel nostro esempio, il modulo **Main** invoca le funzioni, quindi utilizza i servizi, forniti dagli altri due moduli
- Ne segue il seguente **schema logico...**

Schema logico



- Come mai anche il modulo Caricamento/Salvataggio è **utente** del modulo Manipolazione/Stampa?

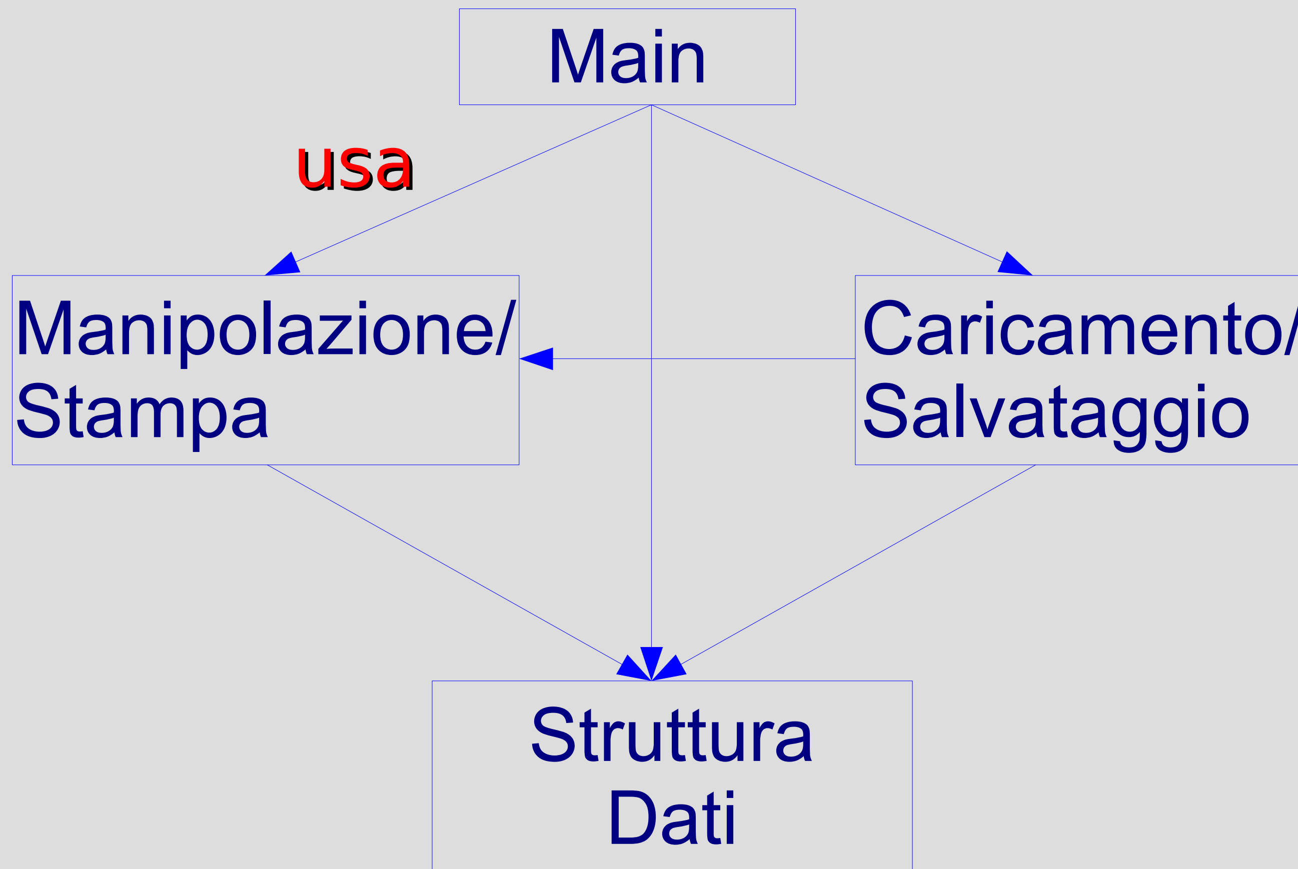
Risposta

- Perché la funzione **salva()** del modulo **Caricamento/Salvataggio** utilizza la funzione **stampa_elem()** del modulo **Manipolazione/Stampa**

Struttura dati

- Bisogna considerare anche che tutti e tre i moduli fanno riferimento ad una **struttura dati comune**
- Anche se non contiene funzioni, ha pertanto senso logico definire un **modulo contenente solo tale struttura dati**
- Lo schema si complica un pò...

Schema logico completo



Schema logico e codice

- Se rappresentiamo questa struttura logica nel codice, allora il codice sarà probabilmente più
  - **chiaro, efficiente, mantenibile, ...**
- Al momento non disponiamo di molti strumenti per rendere la struttura logica immediatamente visibile in quella fisica
- Possiamo però sfruttare i **commenti ed i prototipi delle funzioni**

Intestazione (header)

- Inseriamo per ciascuna funzione anche il suo **prototipo**
- Per **ciascun modulo**, raggruppiamo le sue strutture dati ed i prototipi delle funzioni in una **intestazione** (**header**) fatta per esempio così:

```
/* Inizio header modulo X */
// Struttura dati modulo X
...
// Funzioni modulo X
... (prototipi delle funzioni)
/* Fine header modulo X */
```

Possibili schemi (1)

```
/* Inizio header modulo X */
    // Struttura dati modulo X
    ...
    // Funzioni modulo X
    ... (prototipi delle funzioni)
    /* Fine header modulo X */

    /* Inizio header modulo Y */
    // Struttura dati modulo Y
    ...
    // Funzioni modulo Y
    ... (prototipi delle funzioni)
    /* Fine header modulo Y */
```

Possibili schemi (2)

```
/* Inizio header modulo Z */
// Struttura dati modulo Z
...
// Funzioni modulo Z
... (prototipi delle funzioni)
/* Fine header modulo Z */

// Definizioni delle funzioni di tutti i moduli
...
int main(...) {
...
}
```

Possibili schemi (3)

- Alternativa: le definizioni delle funzioni di ciascun modulo possono essere messe subito dopo ciascuna intestazione di modulo
- Quello che conta è che **guardando l'intestazione sappiamo tutto del modulo**
- Un esempio è

  GSeq_1file_mod1.cc

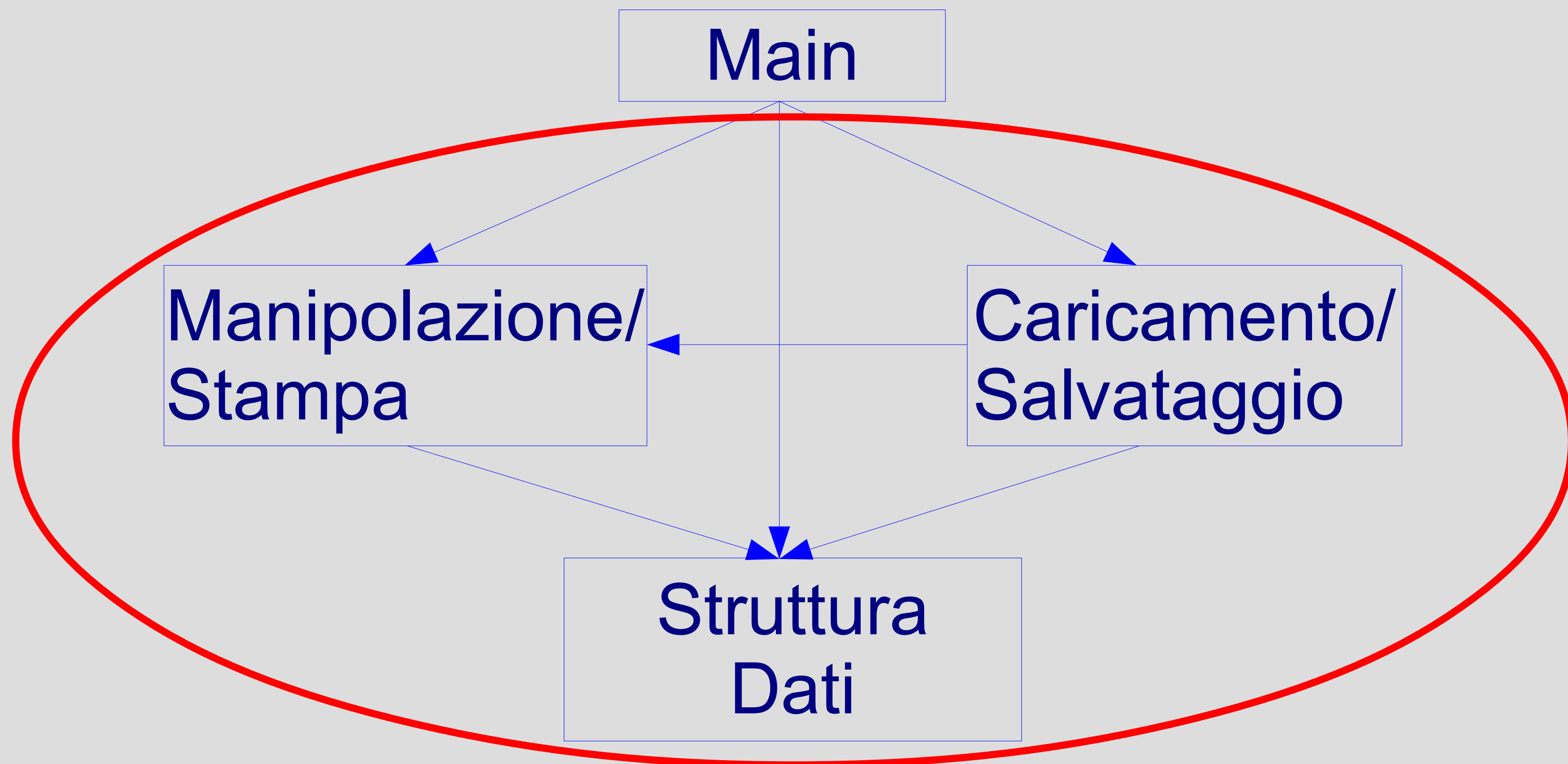
Moduli, oggetti e tipi di dato

- Prima di raffinare la definizione dei moduli, torniamo un momento su cosa può essere implementato da un modulo
- In generale, **un modulo fornisce dei servizi ai suoi utenti (altri moduli)**
- In particolare, può implementare tanto un **singolo oggetto** che un **tipo di dato**
- Vediamo un esempio per ciascuna delle due possibilità

Singolo oggetto (1)

- **Tutte** le funzioni del nostro programma prendono in ingresso anche la sequenza di dati su cui lavorare *(di tipo desc_sequenza)*
- Supponiamo invece che:
  - si fosse utilizzata una **variabile globale** per la sequenza
  - le funzioni dei moduli **Manipolazione/Stampa** e **Caricamento/Salvataggio** non avessero preso in ingresso la sequenza, ma avessero implicitamente lavorato sulla variabile globale
  - logicamente la definizione di tale variabile fosse collocata nel modulo **Struttura Dati**

Singolo oggetto (2)

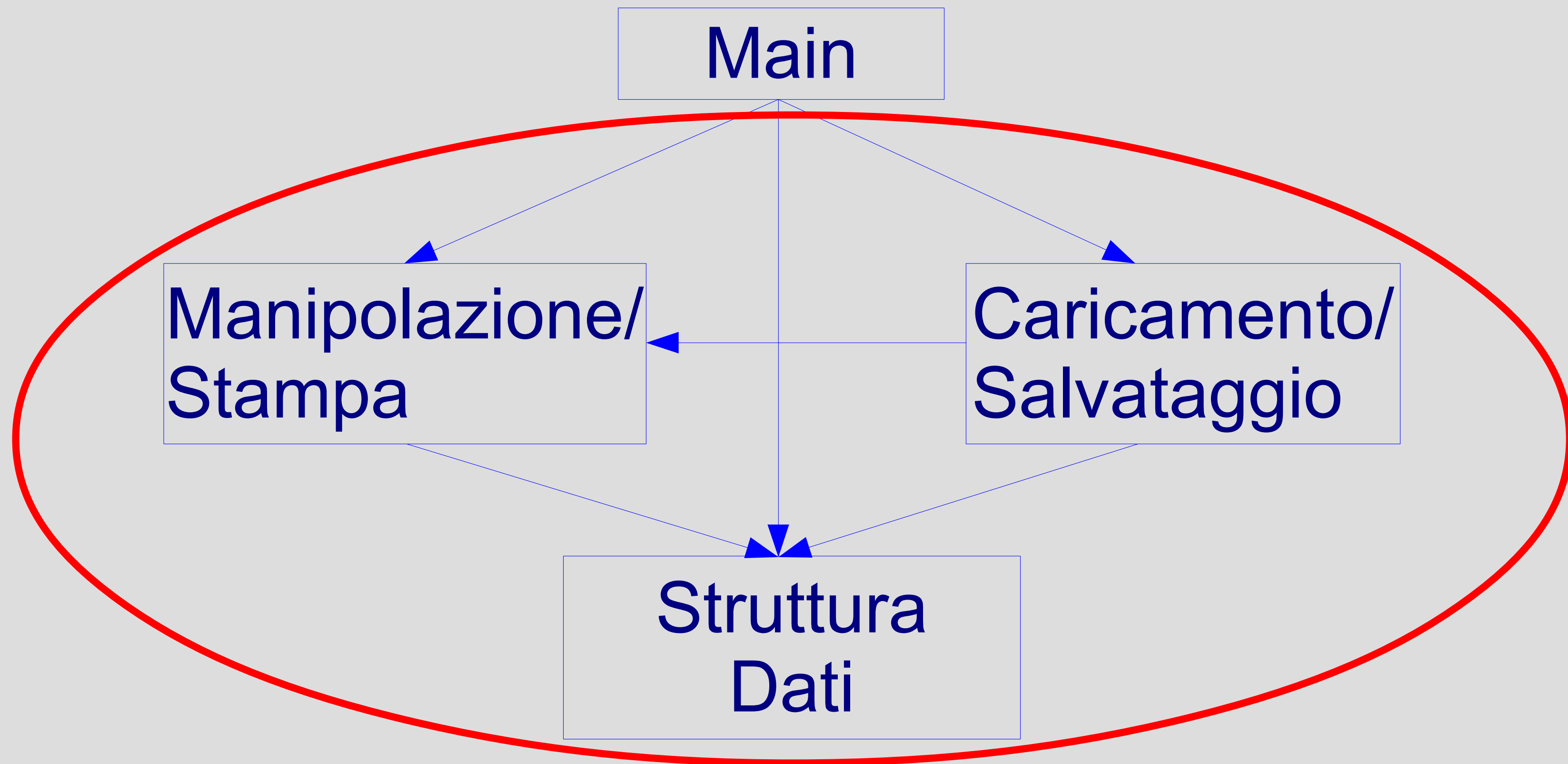


L'insieme di questi tre moduli implementa **la struttura dati sequenza** e tutte le operazioni che vi si possono effettuare

Tipo di dato

- Torniamo ora alla **versione effettiva** del nostro programma, in cui ciascuna funzione prende invece in ingresso anche la sequenza su cui lavorare
- Ne segue che, nel **main()**, o in qualunque funzione nella quale si intenda utilizzare delle sequenze, si possono **definire tutti gli oggetti di tipo sequenza** che si vuole, e lavorarci sopra con le funzioni fornite dai moduli
- Il risultato è quindi che ...

Tipo di dato



L'insieme di questi tre moduli implementa il **tipo di dato sequenza** e tutte le operazioni che vi si possono effettuare

Interfaccia

- Ora raffiniamo ulteriormente il nostro **schema logico**
- Consideriamo ad esempio le funzioni *diff*, *ric_sequenziale e ric_binaria* del modulo **Manipolazione/Stampa**
- Ha senso che siano invocate dal *main()*?
- Probabilmente no
  - *diff* è usata dalle altre funzioni del modulo per effettuare i confronti
  - *ric_sequenziale e ric_binaria* sono invocate in modo opportuno dalla funzione *cerca*

Interfaccia e implementazione

- Per gestire in modo pulito queste situazioni, ci conviene fare la seguente **distinzione**
- Dato un modulo, definiamo
  - **Interfaccia del modulo:** le funzioni e le strutture dati che gli utenti sono autorizzati ad utilizzare per accedere a servizi del modulo
  - **Implementazione del modulo:** le funzioni e le strutture dati mediante cui il modulo implementa i propri servizi

Parte pubblica e privata (1)

- Le funzioni e le strutture dati di interfaccia possono ovviamente essere parte dell'implementazione di un modulo
- ***Ma non è detto che tutto ciò che è implementazione sia esportato anche come interfaccia***
- Definiamo **pubbliche le funzioni e le strutture dati accessibili dagli utenti di un modulo** e **private tutte le altre strutture dati e funzioni del modulo**

Parte pubblica e privata (2)

L'**interfaccia** è costituita da:

- *Strutture dati pubbliche*
- *Dichiarazioni delle funzioni pubbliche*

L'**implementazione** è costituita da:

- *Strutture dati pubbliche e private*
- *Definizione delle funzioni pubbliche e private*

Vantaggi (1)

Vantaggi della suddivisione tra interfaccia e implementazione:

- L'utente di un modulo **non deve conoscere tutti i dettagli** di un modulo per utilizzarlo
- Chi costruisce un modulo è libero di implementarlo come meglio crede e di cambiare l'implementazione purché lasci **inalterata l'interfaccia**
  - *Cambia solo la parte privata dell'implementazione*

Vantaggi (2)

- Per dare un'idea dell'importanza del concetto basta considerare per esempio che è grazie a questo approccio che **Internet funziona**
- Macchine con hardware e software molto diversi possono comunicare tra loro in tutto il mondo perché l'**interfaccia** è stata stabilita una volta per tutte (in particolare nel mondo della rete si parla di **protocolli standard di comunicazione**)

Vantaggi (3)

- Un altro esempio importante sono i **driver** dei dispositivi fisici dei calcolatori
- Potete installare per esempio con facilità un nuovo mouse o una nuova scheda grafica in un PC perché tra il dispositivo ed il sistema operativo si interpone un **modulo chiamato driver del dispositivo**, che:
  - da un lato gestisce il dispositivo in base ai suoi dettagli fisici (**implementazione**)
  - dall'altro presenta sempre la stessa **interfaccia** al sistema operativo

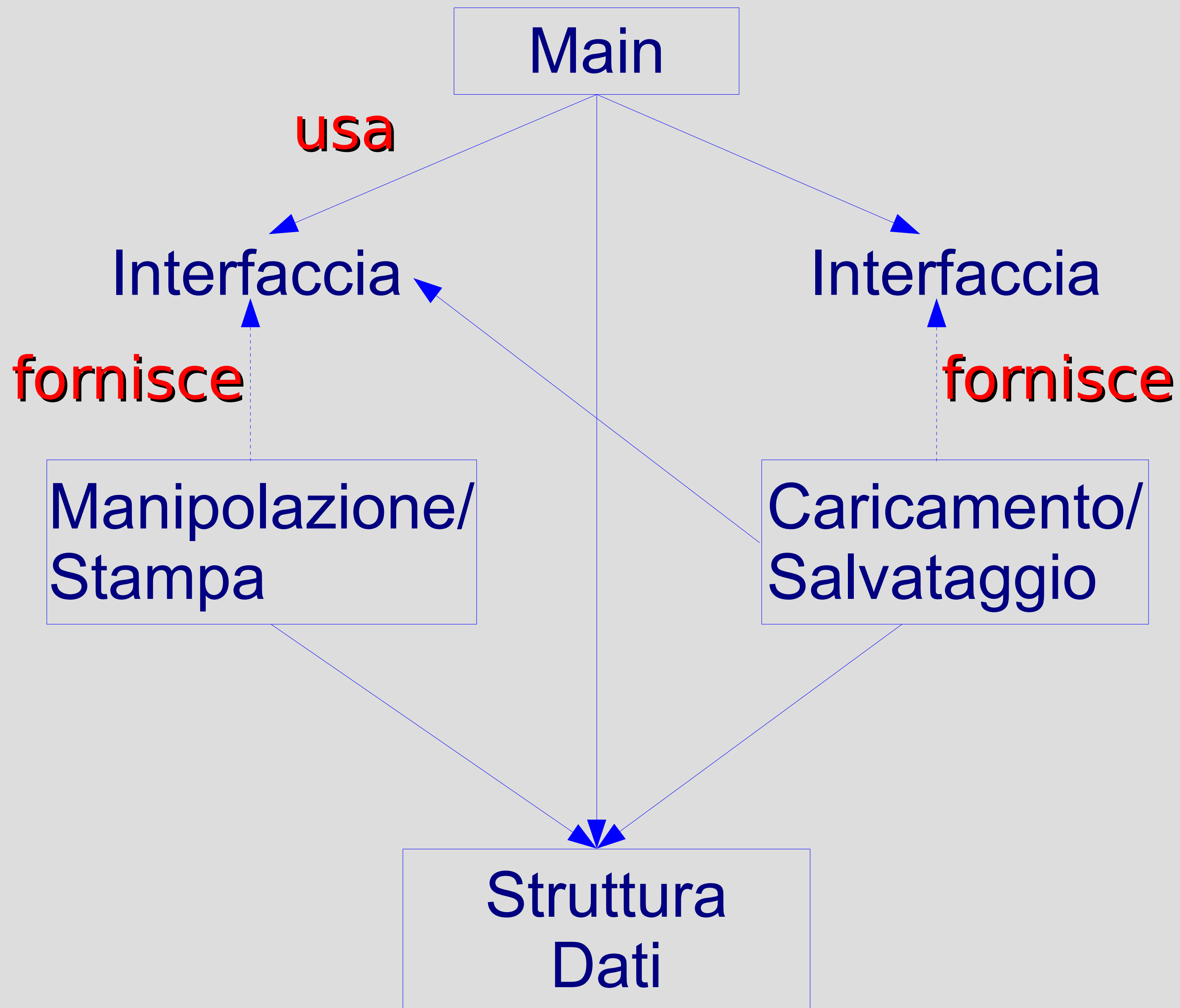
Vantaggi (4)

- Se si installa un **nuovo dispositivo** basta installare anche il relativo **nuovo driver**, che presenterà la stessa **interfaccia** del precedente verso il sistema operativo
  - Non ci sarà bisogno di cambiare nulla nel resto del sistema operativo affinché tutto funzioni

Rispettare l'interfaccia

- Questo meccanismo ovviamente fallisce se si accede anche alla parte privata dei moduli
  - In questo caso se si cambia modulo, ed il nuovo modulo ha qualcosa di diverso nell'implementazione, (es.: driver di un nuovo mouse, nuova versione di un driver) l'utente del modulo non riesce più ad usarlo!
- *In conclusione: definiamo chiaramente l'**interfaccia** dei moduli ed **accediamo ai loro servizi solo tramite l'interfaccia***

Nuovo schema logico



Nota

- A differenza degli altri due, il modulo **Struttura Dati non ha un'interfaccia** distinta dall'implementazione

Completamento header

- Possiamo rendere **esplicita la separazione** tra interfaccia e rimanente parte privata dell'implementazione all'interno degli **header dei moduli**
- Adottiamo ad esempio lo schema della slide seguente

Esempio

```
/* Inizio header modulo X /
/* Inizio interfaccia modulo X /
// Struttura dati
...
// Funzioni
... (prototipi delle funzioni)
/* Fine interfaccia modulo X */
/* Inizio parte privata modulo X /
// Struttura dati
...
// Funzioni
... (prototipi delle funzioni)
/* Fine parte privata modulo X */
/* Fine header modulo X */
```

Esercizio

- Completare gli header nel programma per evidenziare l'interfaccia e la parte privata
- Esempio:

  GSeq_1file_mod2.cc

API

- Spesso l'interfaccia di un modulo software (applicazione) è chiamata **Application Programming Interface (API)**
- Le API rappresentano un insieme di **procedure disponibili al programmatore**, di solito raggruppate a formare un set di strumenti specifici per un determinato compito
- Metodo per ottenere un'**astrazione**, di solito tra l'hardware e il programmatore, o tra software a basso ed alto livello

Sviluppo di un programma su più file sorgenti

Programma su più file

- Vediamo operativamente come realizzare un **programma su più file sorgenti**
- Uno dei modi per **compilare** un programma costituito da più file sorgenti è **invocare il compilatore passandogli i nomi di tutti i file sorgenti da cui è costituito**

  ```
  g++ -Wall file1.cc file2.cc ... fileN.cc
  ```

- L'ordine tra i file non ha alcuna importanza

Nozioni necessarie

- Ipotizziamo quindi di organizzare un programma su più file sorgenti
- Per scrivere correttamente il programma abbiamo bisogno delle seguenti nozioni:
  - **Unicità della funzione main**
  - **Visibilità (scope) a livello di file**
  - **Collegamento (linkage) interno ed esterno**

Unicità della funzione main()

- L'esecuzione di un programma inizia sempre dalla stessa istruzione
- Nel linguaggio C/C++ i programmi iniziano dalla **prima istruzione della funzione main()**
- Pertanto, per evitare ambiguità, la funzione **main** deve essere definita in uno solo dei file sorgente

Visibilità (scope)

Completiamo le nostre conoscenze sul **concetto di visibilità**, precisando che:

- Un **identificatore** dichiarato in un file sorgente al di fuori delle funzioni, è visibile dal punto in cui viene dichiarato fino alla fine del file stesso
  - Si parla di **visibilità di file**
- Se la dichiarazione dell'identificatore è una definizione, si dice che la corrispondente entità (variabile, funzione, ...) è **definita a livello di file**

Collegamento interno

- Un identificatore ha **collegamento interno (internal linkage)** se si riferisce ad una entità accessibile solo dal file in cui l'identificatore e l'entità stessa sono dichiarati/definiti
- Ovviamente file diversi possono avere identificatori con collegamento interno con lo stesso nome
  - In ogni file l'identificatore si riferirà ad una entità diversa

Collegamento esterno

- Diciamo che un identificatore ha **collegamento esterno (external linkage)** se si riferisce ad una entità accessibile anche da file diversi da quello in cui l'entità stessa è definita
- Tale entità deve essere **unica in tutto il programma**
- Tale entità è **globale al programma**
- Tale entità può essere **definita in un solo file**

Collegamento

Per default:

- Gli identificatori di entità definite **a livello di blocco non hanno collegamento** (nemmeno interno)
- Gli identificatori di entità definite **a livello di file hanno collegamento esterno**
  - Ma per le regole di visibilità, tali identificatori **non sono comunque visibili** in un file diverso da quello in cui sono dichiarati
  - Per renderli visibili devo aggiungere qualcosa...

Uso di identificatori esterni (1)

- Per rendere visibile in un file sorgente *fileX.cc* un identificatore **id** definito in un altro file *fileY.cc*

  (1) L'identificatore **id** deve avere un **collegamento esterno**

  (2) Si deve **ridichiarare** l'entità dentro il *fileX.cc*

Uso di identificatori esterni (2)

La ridichiarazione nel file *fileX.cc* può avere **due forme**

- Se **id** si riferisce ad una **variabile**, bisogna ripeterne la dichiarazione facendola precedere dalla parola chiave **extern**
  - Altrimenti ci si riferisce ad una copia locale al file!
- Se **id** si riferisce ad una **funzione**, basta semplicemente **ripeterne la dichiarazione**

Uso di identificatori esterni (3)

- Attenzione che se **id** si riferisce ad una **variabile**, allora

```
extern int id ;
```

  rappresenta sempre una **dichiarazione** e non una definizione!

- Occorre che esista almeno un file in cui la variabile **id** e' definita (senza extern) con collegamento esterno. Es.:

```
int id ;    // fuori da ogni funzione
```

Uso di identificatori esterni (4)

- Se invece **id** si riferisce ad una **funzione**, non ci sono problemi, dato che

```
extern int fun() ;
```

  e' equivalente a

```
int fun() ;
```

- Cio' sia nel caso di dichiarazione che di definizione di una funzione
- Percio' **extern** non e' usato con le funzioni!

Esempio

```
// file1.cc
extern int a ;   //ridichiarazione di a
char fun(int b) { ... }   //definizione di fun
```

```
// file2.cc
int a;     //definizione di a
char fun(int) ;   //ridichiarazione di fun
int main() { ... }
```

- **a** è definita in *file2.cc*, mentre **fun** è definita in *file1.cc*, comunque **i due file sorgenti condividono entrambe le entità**

Esercizio

- Scrivere, compilare ed eseguire un programma costituito **da due file sorgenti**
- Nel sorgente contenente la funzione **main** definire una variabile intera a livello di file
- Nell'altro sorgente definire una funzione **fun** senza argomenti che stampa il contenuto di tale variabile
- All'interno della funzione **main**, leggere da **stdin** il valore della variabile ed invocare la funzione **fun**

Soluzione

- Una possibile soluzione è in *primo_prog_multifile/*.cc*

Classe static

- Gli identificatori di entità definite **a livello di file hanno collegamento esterno**
- A meno che non vengano definite **static**
- Nel qual caso:
  - Hanno collegamento interno
  - Non sono visibili al di fuori del file di definizione

Esempio

- Modificare a **static** la classe di memorizzazione della variabile intera definita a livello di file nell'esercizio precedente
- Es: static int a ;
- Cosa cambia?

Classe static

- Gli identificatori di entità definite **a livello di blocco non hanno collegamento** (nemmeno interno)
- Hanno tempo di vita pari a quello del blocco
- Definendole **static** il collegamento non varia
  - Visibili all'interno del blocco
  - Non visibili fuori dal blocco, nemmeno tramite ridichiarazione
- Il tempo di vita diventa “**statico**” = pari alla vita dell'intero programma (dalla definizione)

Esempio

```
void fun() {
 static int x = 0; // unica inizializzazione
                   // per ogni istanza fun
 cout << x << endl ;
 x = x + 1;
}
void main()
{ fun(); // stampa 0
  fun(); // stampa 1
  fun(); // stampa 2
}
```

Riassunto

Classe di memorizzazione	Tempo di vita	Collegamento
`extern`	“static”	esterno
`static`	“static”	interno
`auto`, `register`	block	nessuno

Tempo di vita “static” = Tempo di vita dell'intero programma

Dichiarazioni di tipo (1)

- Per utilizzare, all'interno di un file *fileX.cc*, un tipo (struct, enum o typedef) dichiarato in un altro file *fileY.cc* bisogna **ridichiarare lo stesso identico tipo** all'interno di *fileX.cc*
  - In particolare i due tipi devono essere **equivalenti**
- In C++ due tipi sono **equivalenti se e solo se hanno lo stesso nome**
  - *Equivalenza per nome*

Dichiarazioni di tipo (2)

ATTENZIONE! Se:

- la struttura di due tipi con lo stesso nome, dichiarati in due file sorgenti distinti, fosse diversa
- i due file condividessero oggetti di tale tipo
  - ad esempio da un file sorgente si invoca una funzione definita nell'altro file sorgente passandogli un oggetto di tale tipo

il compilatore non segnalerebbe comunque errori!

Esempio

```
// file1.cc
struct ss {int a} ;
void fun(ss b) { ... }
```

```
// file2.cc
struct ss {char c ; short z ;} ;
void fun(ss) ;
int main() { ss k ; fun(k); …}
```

- Le due dichiarazioni di **ss** sono incompatibili, ma il compilatore non segnala errori

Dichiarazioni di tipo (3)

- Ovviamente, date **dichiarazioni esterne incompatibili**, sarà del tutto improbabile che poi il programma funzioni correttamente
- Sta quindi al programmatore evitare questo **errore logico**

Struttura logica/fisica

- La **struttura logica** di un programma (*suddivisione* in *moduli*) può essere realizzata attraverso la **struttura fisica** (*suddivisione in file sorgenti)*
- NON è però obbligatorio che le due strutture coincidano rigidamente (un file sorgente distinto per ciascun modulo)
  - **Un certo insieme di moduli può essere implementato in un unico file sorgente**
  - **Un modulo molto grande può essere distribuito su più file sorgenti correlati**

Gestione sequenza GSeq

- Proviamo a **distribuire su più file** il nostro programma
- La struttura logica precedentemente individuata ci aiuta molto
- Possiamo mettere i **tre moduli** Main, Manipolazione/Stampa, Caricamento/Salvataggio in tre file distinti
- *Possiamo utilizzare un file sorgente separato anche per la struttura dati?*

Risposta

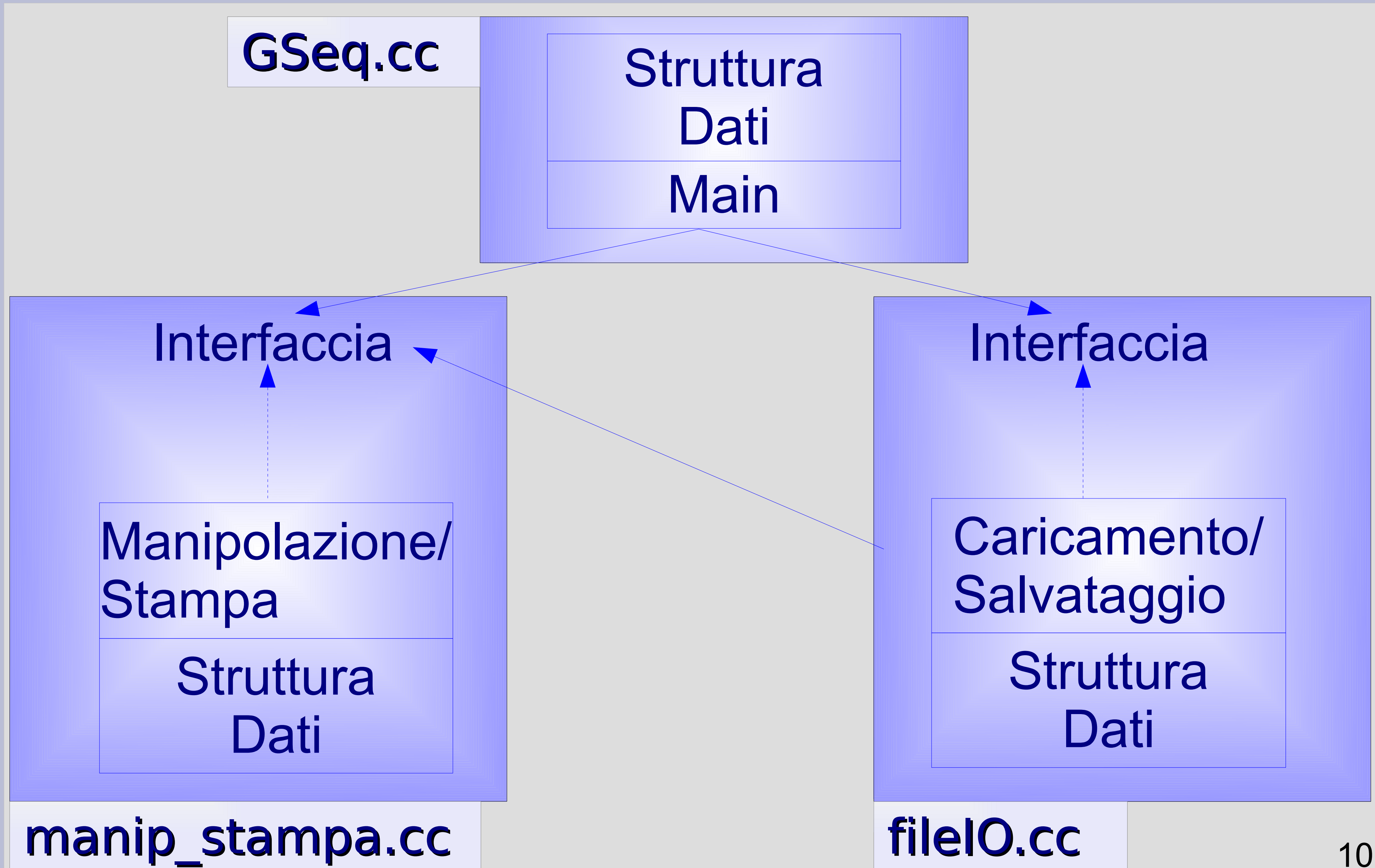
- *No, perché il modulo struttura dati contiene solo dichiarazioni*
- Per quanto detto finora, **tali dichiarazioni vanno ripetute (in modo corretto e consistente) in tutti i moduli in cui vengono utilizzate**

Ripetizione interfacce

Per quanto detto finora, per poter utilizzare i servizi di uno degli altri moduli, un file sorgente deve **ripeterne l'interfaccia**

- *Dichiarazione delle funzioni pubbliche (**prototipi**)*
- *Dichiarazione delle eventuali strutture dati pubbliche (oggetti da dichiarare **extern**)*

Suddivisione in file



Un modo per iniziare

- Fare 3 copie del singolo file iniziale: *Gseq.cc, manip_stampa.cc, fileIO.cc*
- In **ciascun file sorgente** mantenere:
  - l'implementazione del modulo che si intende implementare con quel file
  - solo la parte di interfaccia delle intestazioni dei moduli che si utilizzano
- Una soluzione è in

  progetto_multifile_noheader/.cc*

Problemi

Le dichiarazioni sono ripetute diverse volte nei vari file sorgenti (Es. prototipi e struttura dati)

- **Error-prone**
- Cosa succede se **qualcosa viene cambiato** in uno dei file (es. una interfaccia)?
  - Gli altri file diventano **inconsistenti**
  - Nel **caso migliore**, il programma non si compila
  - Nel **caso peggiore**, il programma si compila lo stesso ma prima o poi fallisce a run-time

File Header

- Vediamo ora un metodo per ottenere **consistenza** fra dichiarazioni effettuate in file sorgenti diversi
- Utilizziamo i cosiddetti **file di intestazione** (**header files**)
- Si basano sul concetto visto in precedenza di **intestazione** di uno o più moduli

Osservazione

- In ciascun file sorgente in cui si debbono utilizzare gli stream standard, bisogna ripetere l'inclusione di **<iostream>** e la **direttiva**

  using namespace std ;

- Lo stesso vale per l'inclusione di ogni altro **header file** necessario per utilizzare altri oggetti o funzioni di libreria

Direttiva #include (1)

- Si sfrutta la direttiva **#include**
- In effetti, i nostri programmi non si sarebbero compilati correttamente se non avessimo aggiunto la direttiva

  #include <iostream>
- Cosa fa tale direttiva?

Direttiva #include (2)

#include <nome_file>

- Include il file chiamato **nome_file** nel nostro file sorgente
  - E' proprio come se, a partire dal punto in cui compare la direttiva, avessimo inserito manualmente il contenuto del file
- Vediamo proprio l'esempio del file **iostream**

Esempio

```
#include <iostream>




int main ()
{
   cout<<"Ciao mondo"<<endl ;
}
```

ciao_mondo.cc

A partire da *ciao_mondo.cc*, il preprocessore genera l'effettivo testo da compilare **inserendo anche il contenuto di iostream**

Esempio

Questo è quindi il testo che sarà compilato

```
#include <iostream>
```

iostream

```
int main ()
{
   cout<<"Ciao mondo"<<endl ;
}
```

Testo da compilare

Sintassi #include

- Il file **iostream** è cercato all'interno di un insieme di **directory predefinite**, come vedremo in dettaglio nella lezione sulla **compilazione**
- E' possibile anche una **diversa sintassi per la direttiva**, in cui le parentesi angolari sono sostituite da doppi apici

  #include "file_da_includere"

- In questo caso **file_da_includere** è cercato nella **stessa directory in cui è presente il file sorgente** che contiene la direttiva

Singolo Header file (1)

Proviamo la seguente divisione

- **Definizioni** nel file *.cc*
- Tutte le **dichiarazioni comuni di entità condivise** vengono messe in un singolo file di testo, che chiameremo **header file** (estensione **.h**)

Singolo Header file (2)

- Quindi, anziché scrivere manualmente le dichiarazioni necessarie per utilizzare gli identificatori esterni, **includiamo questo header file in tutti i file sorgente**
- *Se si modifica l'interfaccia di un modulo, basta modificare solo l'header file*
- All'atto della compilazione, tutti i file sorgenti che usano il modulo disporranno correttamente della nuova interfaccia grazie all'**inclusione dell'header file**

Dichiarazioni e definizioni (1)

- Consideriamo l'unico header file ed un file sorgente che lo include ed implementa uno dei moduli
- Dopo l'inclusione, tale file sorgente conterrà **sia le dichiarazioni** (presenti nell'header file) **che le definizioni** di alcune delle entità

Questo è un problema?

Dichiarazioni e definizioni (2)

- ***NO:*** purché le varie dichiarazioni siano equivalenti tra loro ed equivalenti alla definizione, un identificatore può anche essere dichiarato più volte in un file sorgente
- E' simile a quello che abbiamo visto con la **dichiarazione/definizione** di una funzione
- Ovviamente le dichiarazioni contenenti la parola chiave **extern** sono dichiarazioni come le altre

Dichiarazioni e definizioni (3)

- Quindi si può scrivere per esempio un file sorgente come il seguente:

  extern int a ;

  int a ;

  int main() { ... }

- La dichiarazione **extern** dice che **a** è una variabile definita in un qualche file, la successiva definizione dice che la definizione è proprio in questo file

Limitazioni

Un singolo header file va bene per un programma di piccole dimensioni, ma se il programma è medio-grande:

- **Inefficiente**
  - Lo vedremo meglio nella lezione sulla compilazione
- **Error-prone** nel caso di un team di programmatori
- Tende a **mascherare la struttura logica**

Header file multipli (1)

- Ogni modulo ha il suo header che definisce l'**interfaccia** dei moduli/servizi offerti
  - **L'interfaccia per i moduli utente si trova nei file .h**
- Lavorare con una prospettiva locale (in termini di file header) è più semplice

Header file multipli (2)

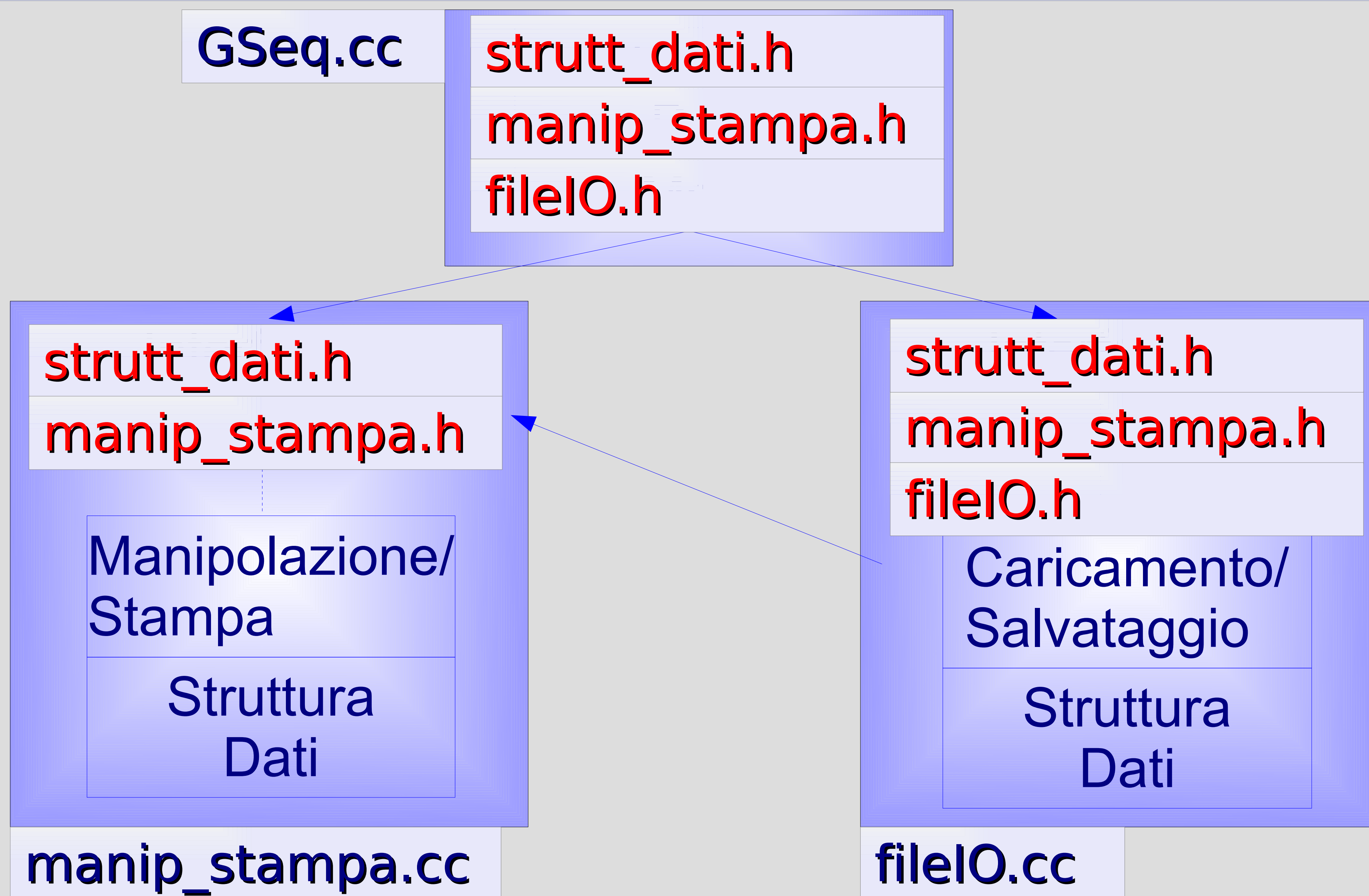
Proviamo a definire i seguenti **header file** per il nostro programma già diviso in tre file sorgenti

- **strutt_dati.h**
  - Contenente la struttura dati comune
- **manip_stampa.h**
  - Contenente l'interfaccia del modulo Manipolazione/Stampa
- **fileIO.h**
  - Contenente l'interfaccia del modulo Caricamento/Salvataggio

Header file multipli (3)

- A questo punto, in modo simile al caso di un singolo **header file**, anziché scrivere manualmente le dichiarazioni necessarie per utilizzare gli identificatori esterni, si **includono gli header file che servono**
- Ovviamente ogni modulo includerà anche l'**header file** che ne contiene l'interfaccia
  - Per evitare di riscrivere manualmente tale interfaccia nel modulo

Suddivisione in file



Soluzione

- Ristrutturare il programma definendo ed includendo i precedenti 3 header file
- Una possibile soluzione è nella cartella

  progetto_multifile

Semplicità

Il metodo è **semplice**:

- Se si modifica **l'interfaccia** di un modulo, basta modificare solo il corrispondente **header file**
- All'atto della compilazione, tutti i file sorgenti che usano il modulo e quindi includono tale header file, disporranno **correttamente della nuova interfaccia**

Protezione parte privata

- Si potrebbe desiderare di *proteggere la parte privata di un modulo: impedire l'accesso dall'esterno*
- Si può forzare il **collegamento interno** anche per entità definite a livello di file
- Basta aggiungere la parola chiave **static** nella **definizione**
  - Es: funzione **dealloca_elem()**
- Anche le funzioni **inline** hanno **collegamento interno**
  - Es. Funzione **diff()**

Interfaccia (1)

- Per quanto visto finora, mediante le **regole di visibilità e collegamento**, si può rendere accessibili solo le funzioni di **interfaccia**
- Es: per permettere al compilatore di impedire l'accesso a funzioni **private** di un modulo si può anche dichiararle **static** o **inline**
  - Errore a tempo di compilazione se tento di esportarle ed usarle in un altro modulo

Interfaccia (2)

- Purtroppo però, dato ad esempio un **tipo di dato struttura** che deve essere **condiviso** tra più file, **da ogni file si può accedere a tutti i campi della struttura**
- Quindi, anche se la struttura dati è pensata da essere manipolata solo attraverso funzioni dedicate (pensare ad esempio alle funzioni di manipolazione di una sequenza), **non c'è nessun meccanismo linguistico** che vieti di usare (alcuni) campi privati

Campi pubblici e privati

- Si possono inserire **commenti** e/o **documentazione** in cui si chiarisce quali campi di un tipo di dato struttura sono da considerare **pubblici** e quali **privati**
- Però nel linguaggio questa distinzione non può essere esplicitata, quindi il **compilatore non può impedire ad un utente di utilizzare la parte privata della struttura condivisa** di un modulo, con tutte le possibili conseguenze negative precedentemente evidenziate

Tipo di dato class

- In **C++** esiste il tipo di dato **class** che permette di definire **interfaccia** ed **implementazione** di ogni oggetto nel codice stesso
- Il tipo **class** è una estensione del tipo **struct**
- *Oltre che variabili, i campi possono anche essere funzioni*
- *Tali funzioni si chiamano **funzioni membro** o **metodi** della classe*

Metodi

- Un metodo di un oggetto di tipo **class** opera implicitamente sui campi dell'oggetto stesso
- Esempio di invocazione di un metodo:

  cin.clear() ;
- cin è un oggetto di tipo **class istream**, e **clear** è una funzione membro della classe, per cui nella precedente invocazione la funzione **clear** lavora sui campi dell'oggetto **cin** (in particolare sui suoi **flag di stato**)

Campi pubblici e privati in class

- Nella dichiarazione di un tipo **class** si può dichiarare esplicitamente quali campi e quali funzioni sono **pubblici** e quali invece **privati**
- Semplificando, solo le funzioni membro possono accedere ai campi privati o invocare le funzioni private della classe
- Questo schema risolve elegantemente il **problema della separazione tra interfaccia ed implementazione**

Nota conclusiva

- Non vedremo il tipo **class** in questo corso
- Lo vedrete nel corso di **Linguaggi di Programmazione ad Oggetti** nell'ambito del linguaggio **Java**
- Definito in modo coincidente a quello in cui è definito nel **C++**: basato sul concetto di protezione della parte privata dei dati e esportazione di quella pubblica