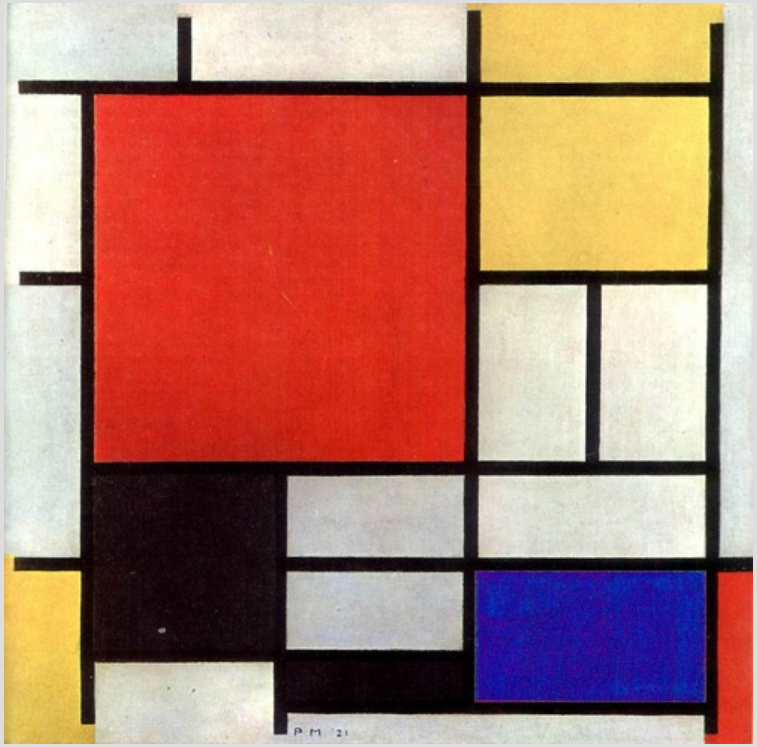
Parte 8 – Sviluppo e testing

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 la progettazione deve essere effettuata ad un **livello di astrazione molto più elevato**

Un nuovo problema …

Lo **sviluppo di sistemi software complessi** (programmazione in grande o “in the large”) richiede di:

- poter suddividere il lavoro tra diversi gruppi di lavoro e di contenere i tempi e i costi di sviluppo che sono in generale molto alti.
- Mantenere alta la qualità del software in termini di:
  - ✓ affidabilità
  - ✓ prestazioni
  - ✓ modificabilità
  - ✓ estensibilità
  - ✓ riutilizzo

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**

Programmazione modulare

La **programmazione modulare** è un necessario approccio alla programmazione in grande, in quanto consente di *gestire la complessità del sistema* da realizzare suddividendolo in parti (**moduli**) tra loro correlate.

- **Modulo:** 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)**

Modularizzazione e astrazione

La programmazione modulare richiede che venga effettuato un *processo di astrazione*

Astrazione: il processo che porta ad individuare e considerare le proprietà rilevanti di un'entità, ignorando i dettagli inessenziali.

I linguaggi di programmazione fanno uso di diverse astrazioni, ad esempio i TIPI di dato, i sottoprogrammi, etc…

Meccanismi di astrazione

I meccanismi di astrazione più diffusi sono:

- astrazione sul controllo
- astrazione sui dati

L'astrazione fondamentale dei **linguaggi procedurali** è **l'astrazione sul controllo** che consiste nell'astrarre una data funzionalità dai dettagli della sua implementazione.

L'astrazione fondamentale dei **linguaggi a oggetti** è **l'astrazione sui dati** che consiste nell'astrarre le entità (oggetti) costituenti il sistema, descritte in termini di una struttura dati e delle operazioni possibili su di essa.

Programma

Partendo dal programma *gestione_lista_doppia_sol.cc, v*ogliamo realizzare una piccola **suite di gestione** di liste di elementi costituita da un insieme di moduli con i seguenti obiettivi:

1. *Separazione* tra le funzionalità che realizzano l'*applicazione* e le funzionalità che *manipolano le liste* così da consentire di cambiare l'implementazione della lista senza intervenire sulla logica dell'applicazione
2. *Separazione* tra le funzionalità di pura *gestione della lista* e le funzionalità che *manipolano il tipo di dato* così da
   - Consentire di *cambiare il tipo di informazione* gestita nella lista senza intervenire sulle funzionalità di gestione della lista
   - Consentire di *cambiare l'implementazione della lista* (ad esempio da lista doppia a lista semplice e viceversa) senza intervenire sulle funzionalità di gestione del tipo di dato

Struttura attuale del programma

Tipo di dato

```
struct elem
{
        char inf[51] ;
        elem* pun ;
        elem* prev;
} ;

typedef elem* lista ;
```

Struttura attuale del programma (cont.)

Primitive

```
char* head(lista);
lista tail(lista);
lista insert_elem(lista, elem*);
lista delete_elem(lista, elem*);
elem* search(lista, char*);
```

Struttura attuale del programma (cont.)

Funzioni

```
void stampalista(lista);
lista crealista(int);
lista cancella(lista, char*);
void naviga(elem*);
int main();
```

Una prima suddivisione in moduli

Esiste già una prima suddivisione in moduli:

Modulo “liste”: contiene la definizione del tipo di dato lista e le primitive per l’implementazione delle liste

Modulo “funzioni dell’applicazione”: contiene le funzioni che realizzano l’applicazione

Modulo “main”: contiene la specifica del menu e la modalità di interazione con l’utente

Moduli clienti

In generale, la relazione fondamentale che c'è tra i moduli di un programma si basa sul fatto che alcuni moduli sono **clienti**, ossia **utilizzano i servizi/oggetti forniti da altri moduli**

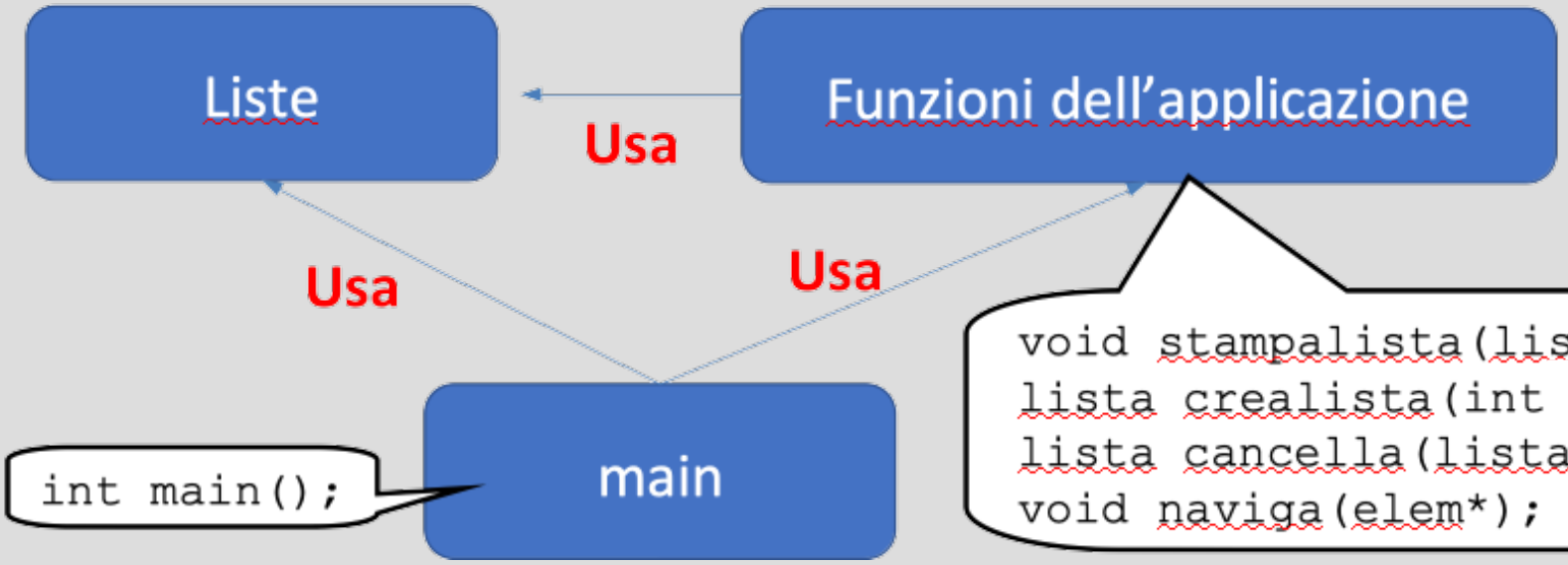
- Nel nostro esempio, il modulo **main** usa il tipo di dato lista e invoca le funzioni dell'applicazione, quindi utilizza i servizi forniti dai corrispondenti moduli
- Il modulo **funzioni dell'applicazione** usa il tipo di dato lista e invoca le sue primitive quindi utilizza i servizi forniti dal modulo «liste»

Ne segue il seguente **schema logico...**

Schema logico

```
struct elem
{
        char inf[51] ;
        elem* pun ;
        elem* prev;
} ;

typedef elem* lista ;
```

```
char* head(lista);
lista tail(lista);
lista insert_elem(lista, elem*);

lista delete_elem(lista, elem*);
elem* search(lista, char*);
```



Nota sulla progettazione

- La progettazione in moduli non ha una sola soluzione
- In generale, una delle difficoltà dell'attività di progettazione è dover scegliere una soluzione all'interno di uno **spazio di soluzioni più o meno vasto**
- Ciascuna soluzione ha i propri **pro e contro**
- Purtroppo spesso gli effetti delle diverse soluzioni non sono banali da prevedere

Obiettivi della progettazione

Abbiamo soddisfatto entrambi gli obiettivi della progettazione?

L'obiettivo 1 è quasi raggiunto: il modulo «funzioni dell'applicazione» usa le primitive del modulo «liste» senza aver alcuna nozione della modalità d'implementazione ma una funzione usa esplicitamente il tipo elemento. Quale?

```
lista crealista(int);
```

Verso l'obiettivo 1

```
lista crealista(int n)
{
        lista testa = NULL ;
        for (int i = 1 ; i <= n ; i++) {
            elem* p = new elem ;
            cout<<"URL "<<i<<": ";
            cin>>p->inf;
            p->pun=p->prev=NULL;
            testa=insert_elem(testa
        }
        return testa ;
}
```

Alloca esplicitamente il nuovo elemento e inizializza I valori

Esercizio

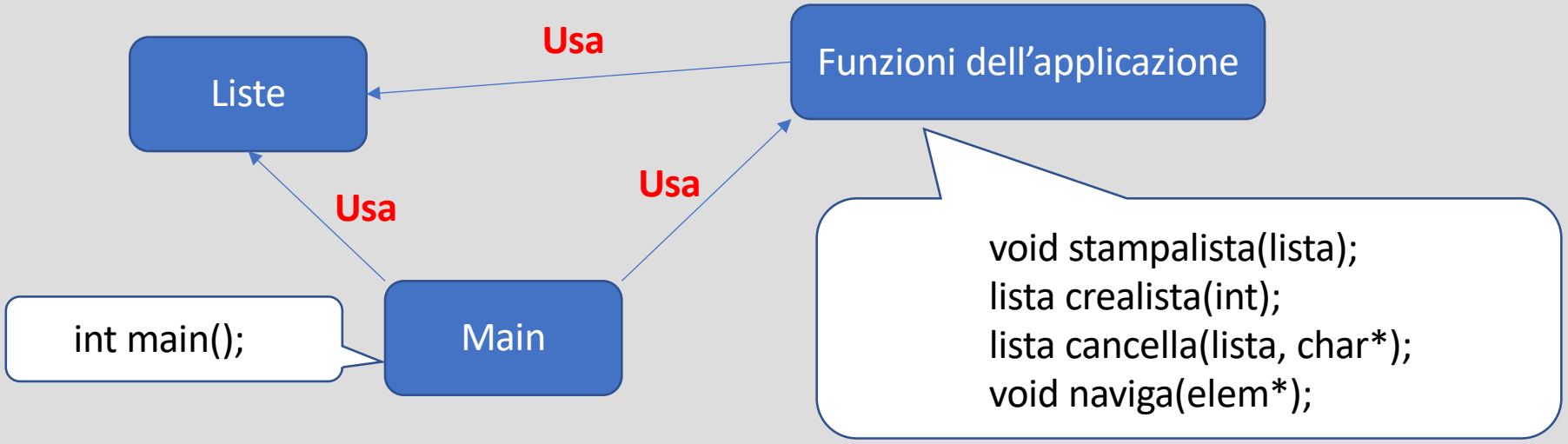
- Definire una nuova primitiva `elem* new_elem(char*)` che alloca un nuovo elemento e gli assegna il valore informativo in ingresso
- Modificare `crealista()` affinché richiami la primitiva `new_elem()`

SOLUZIONE Vedi programma
gestione_lista_doppia_modulo1.cc

Obiettivo 1 raggiunto

```
struct elem
{
        char inf[51];
        elem* pun;
        elem* prev;
};

typedef elem* lista;
```

```
char* head(lista);
lista tail(lista);
lista prev(lista);
elem* new_elem(char*);
lista insert_elem(lista, elem*);
lista delete_elem(lista, elem*);
elem* search(lista, char*);
```



Obiettivo 2

L'obiettivo 2 non è raggiunto!!

Analizziamo nello specifico le primitive del modulo «liste»:

- L'interfaccia e l'implementazione delle primitive

```
lista prev(lista);
lista tail(lista);
lista insert_elem(lista, elem*);
lista delete_elem(lista, elem*);
```

sono indipendenti dal tipo di dato

Verso l'Obiettivo 2

- Invece, l'interfaccia e le implementazioni delle primitive delle liste

```
char* head(lista);

elem* search(lista, char*);

elem* new_elem(char*);
```

dipendono dal tipo di dato

Verso l'Obiettivo 2

Separiamo il modulo «liste» in due moduli:

Modulo «liste-tipo» : contiene la definizione del tipo lista e le primitive sulle liste dipendenti dal tipo

Modulo «liste» : contiene le primitive sulle liste indipendenti dal tipo

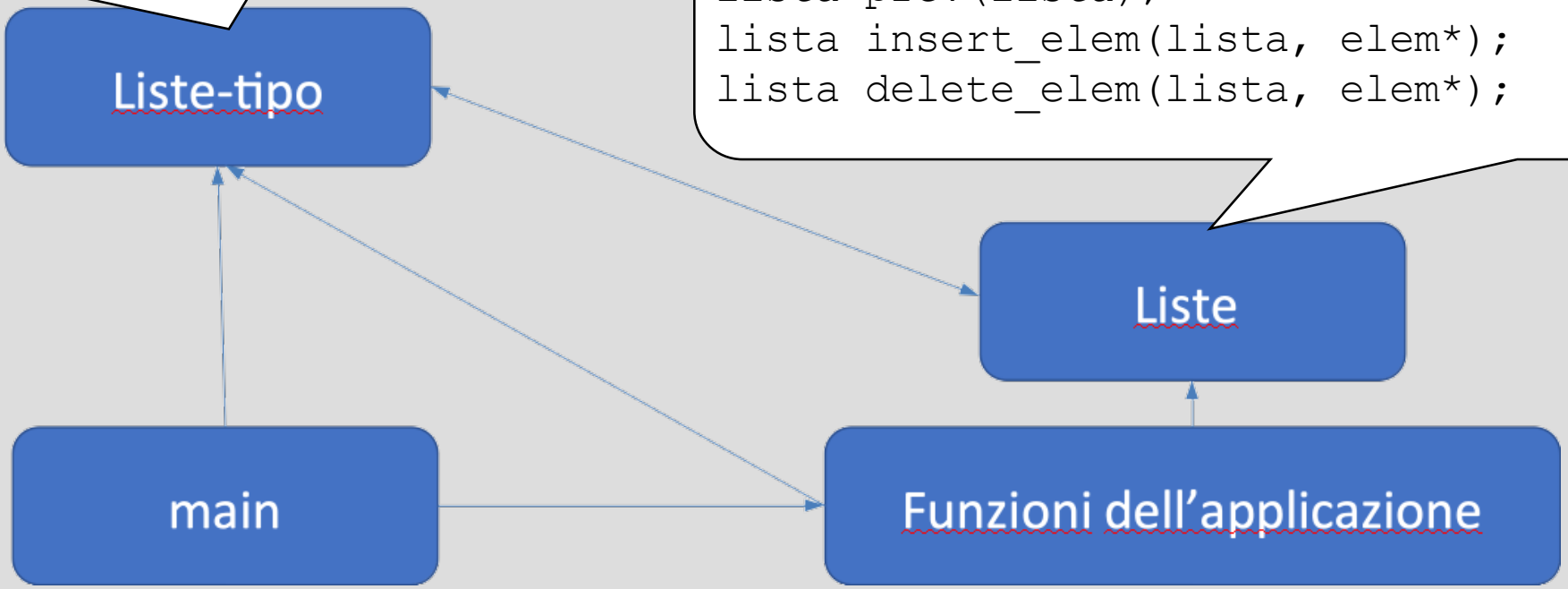
Quando modifico il tipo (ad esempio voglio realizzare liste di interi o liste di libri) è sufficiente modificare il modulo «liste-tipo»

Come cambia lo schema logico?

Nuovo schema logico

```
struct elem
{char inf[51];
 elem* pun ;
 elem* prev;} ;
typedef elem* lista ;
char* head(lista);
elem* new_elem(char*);
elem* search(lista, char*);
```

```
lista tail(lista);
lista prev(lista);
lista insert_elem(lista, elem*);
lista delete_elem(lista, elem*);
```



Realizzazione di un modulo

È dotato di una chiara separazione tra:

- interfaccia
- corpo

Interfaccia specifica “cosa” fa il modulo (l’astrazione realizzata) e “come” si utilizza. Deve essere visibile all’esterno del modulo per poter essere utilizzata dall’utente del modulo al fine di usufruire dei servizi/dati esportati dal modulo.

Corpo descrive “come” l’astrazione è realizzata, e contiene l’implementazione delle funzionalità/strutture dati esportate dal modulo che sono nascoste e protette all’interno del modulo. L’utente può accedere ad esse solo attraverso l’interfaccia.

Schema logico e codice

- Se rappresentiamo questa struttura logica nel codice, allora il codice sarà più

  chiaro, efficiente, mantenibile, ...

- Al momento non disponiamo di molti strumenti per rendere la struttura logica immediatamente visibile nel codice
- Possiamo però sfruttare i **commenti ed i prototipi delle funzioni**

Intestazione (header)

- Inseriamo per ciascuna funzione il suo **prototipo**
- Per **ciascun modulo**, raggruppiamo le sue strutture dati ed i prototipi delle funzioni in una **intestazione** (**header**) che rappresenta l'**interfaccia**:

```
/*******************************/
/* HEADER MODULO xxx */
/*******************************/
// Struttura dati modulo X
...
// Funzioni modulo X
... (prototipi delle funzioni)
```

Un possible schema

- Dopo aver presentato gli header di tutti i moduli facciamo seguire le definizioni delle funzioni
- Quindi per ciascun modulo, raggruppiamo le definizioni (**corpo del modulo**):

```
/*************************************/
/* DEFINIZIONE MODULO xxx */
/*************************************/
```

ESEMPIO Vedi programma
gestione_lista_doppia_modulo3.cc

Parte pubblica e parte privata

Si definiscono

- **Funzioni e Strutture dati Pubbliche** le funzioni e le strutture dati accessibili dagli utenti di un modulo
- **Funzioni e Strutture dati Private** tutte le altre strutture dati e funzioni del modulo

L'**interfaccia** è costituita da:

✓ *Strutture dati pubbliche*

✓ *Dichiarazioni delle funzioni pubbliche*

L'**implementazione** è costituita da:

✓ *Strutture dati private*

✓ *Definizione delle funzioni pubbliche e private*

Vantaggi

Vantaggi della suddivisione tra interfaccia e implementazione:

- Il cliente 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**

Vantaggi (cont.)

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 (cont.)

Un altro esempio importante sono i **driver** dei dispositivi fisici dei calcolatori:

- Potete installare, 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

- 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 il cliente (in questo caso il S.O.) accede anche alla parte privata dei moduli
- In questo caso, se si cambia il modulo, ed il nuovo modulo ha un'implementazione diversa (es.: driver di un nuovo mouse, nuova versione di un driver), il cliente del modulo non riesce più ad usarlo!
- In conclusione:
  *definiamo chiaramente l'**interfaccia** dei moduli ed **accediamo ai servizi dei moduli solo tramite l'interfaccia***

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
- Ad esempio esistono API per accedere a dati dei social network come Facebook e Twitter o API per accedere a servizi accessibili da server (web service)
- Documentazione sulla API di Google Maps https://developers.google.com/maps/documentation/
- Diverse API sono a pagamento

Programma su più file

- In C++ un programma può essere distribuito su più file sorgenti
- La suddivisione in file consente **l'implementazione dei moduli** e l'uso da parte dei clienti
- Ad esempio ogni modulo può corrispondere ad uno o più file sorgente
- Se in un programma ho bisogno di un determinato modulo includo i corrispondenti file sorgente nel programma

Moduli e programmazione su più file

- Poiché il cliente del modulo deve conoscerne l'interfaccia ma ignorarne l'implementazione, è buona norma *tenere separate* la specifica del modulo dalla sua implementazione.
- L'interfaccia è pertanto realizzata mediante un **HEADER file** (file `.h`)
- L'interfaccia di un modulo può essere utilizzata dagli utenti del modulo mediante i meccanismi di **inclusione testuale forniti dal linguaggio** (direttiva `#include`)
- Ma questo lo vedremo più avanti, ora vediamo come scrivere un programma su più file…

Programma su più file

- Un modo 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()`**
- 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
- In questo caso si parla di **visibilità di file**
- Se la dichiarazione è anche una definizione, si dice che la corrispondente entità (variabile, funzione, ...) è **definita a livello di file**

ESEMPI

```
int x; //x e' definito a livello di file
int fun(int); //fun è dichiarato a livello di file
int fun(int){….} //fun è definito a livello di file
```

Collegamento interno

La nozione di collegamento si riferisce all'**accessibilità** dell'entità associata ad un identificatore:

- Un **identificatore** ha **collegamento interno (internal linkage)** se si riferisce ad una entità che può essere acceduta solo nel 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 questo caso, in ogni file l'identificatore si riferirà ad una entità diversa

Collegamento esterno

- 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* ma *può essere dichiarata in più 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*

ESEMPI0

```
int x;              // fun e x hanno
fun(int){….} // un collegamento esterno
```

Ma per le regole di visibilità, tali identificatori, *sebbene accessibili* perché hanno collegamento esterno, *non sono comunque visibili* in un file diverso da quello in cui sono dichiarati

Per renderli visibili devo aggiungere qualcosa...

Uso di identificatori esterni

Per rendere visibile in un file sorgente *fileX.cc* un identificatore `id` definito nel file *fileY.cc*

1. L'identificatore `id` deve avere un **collegamento esterno**
2. Si deve **ridichiarare** l'identificatore nel *fileX.cc*

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`**

ESEMPIO

```
//fileY.cc
int x;
```

```
//fileX.cc
extern int x; //ridich. di x
```

- Se `id` si riferisce ad una *funzione*, basta semplicemente **ripeterne la dichiarazione**

Osservazione

- Ogni volta che in un programma scriviamo `#include <XXX>` stiamo implicitamente includendo le dichiarazioni di tutte le funzioni in esso contenute e ciò ci consente di usarle
- Se un file sorgente usa una funzione senza che sia inclusa la sua dichiarazione il compilatore segnala un errore del tipo

```
error: use of undeclared identifier
```

La keyword `extern`

- La keyword **`extern`** viene usata per dichiarare un identificatore *id* all'interno di un file
- Occorre che esista almeno un file in cui *id* sia definito con collegamento esterno
- Ad esempio:

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

- Se *id* si riferisce ad una *funzione*, non è necessario usare **`extern`** in quanto

  `extern int fun();` e' equivalente a **`int fun();`**

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*: main.cc e file.cc
- 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 Vedi cartella *primo_pog_multifile*

La keyword `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

Esercizio

- Provare a eliminare `extern` dal file file.cc dell'esercizio precedente.

Cosa succede? Perché?

- Modificare in **static** la classe di memorizzazione della variabile `a` definita nel file main.cc: `static int a ;`

Cosa cambia? Perché?

- E se in un altro file aggiunto al progetto c'è la riga

```
extern int a ;
```

A quale variabile si riferisce: a quella definita in file.cc o a quella definita in main.cc?

Classe static

`static` può essere usato anche all'interno del codice delle funzioni:

- Un identificatore di entità (in questo caso variabile) definita **a livello di blocco non ha collegamento** (nemmeno interno)
- Un identificatore di variabile definita **a livello di blocco** ha tempo di vita pari a quello del blocco

Definendo `static` una variabile all'interno di un blocco:

- Il collegamento non varia
- L'entità rimane visibile solo all'interno del blocco
- … ma cambia il Il tempo di vita che diventa "**statico**" ovvero 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
}
```

Scope e linkage

- La *visibilità* è una proprietà gestita dal *compilatore*
- Il *collegamento* (linking) è una proprietà gestita dal *linker*
- Quindi **errori dovuti alla visibilità** vengono segnalati dal compilatore
- Ad esempio se accedo in una funzione ad una variabile definita in un'altra funzione ho un errore di visibilità
- Mentre **errori dovuti al collegamento** vengono segnalati dal linker
- Ad esempio se definisco un identificatore con lo stesso nome in due file distinti ma appartenenti allo stesso progetto ottengo un errore di linking

Dichiarazioni di tipo (1)

Con typedef *dichiariamo* nuovi tipi:

```
typedef elem* lista;
```

- 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)

```
// 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); …}
```

ATTENZIONE! Nel caso in cui:

- la struttura di due tipi con lo stesso nome, dichiarati in due file sorgenti distinti, sia diversa
- i due file condividono 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

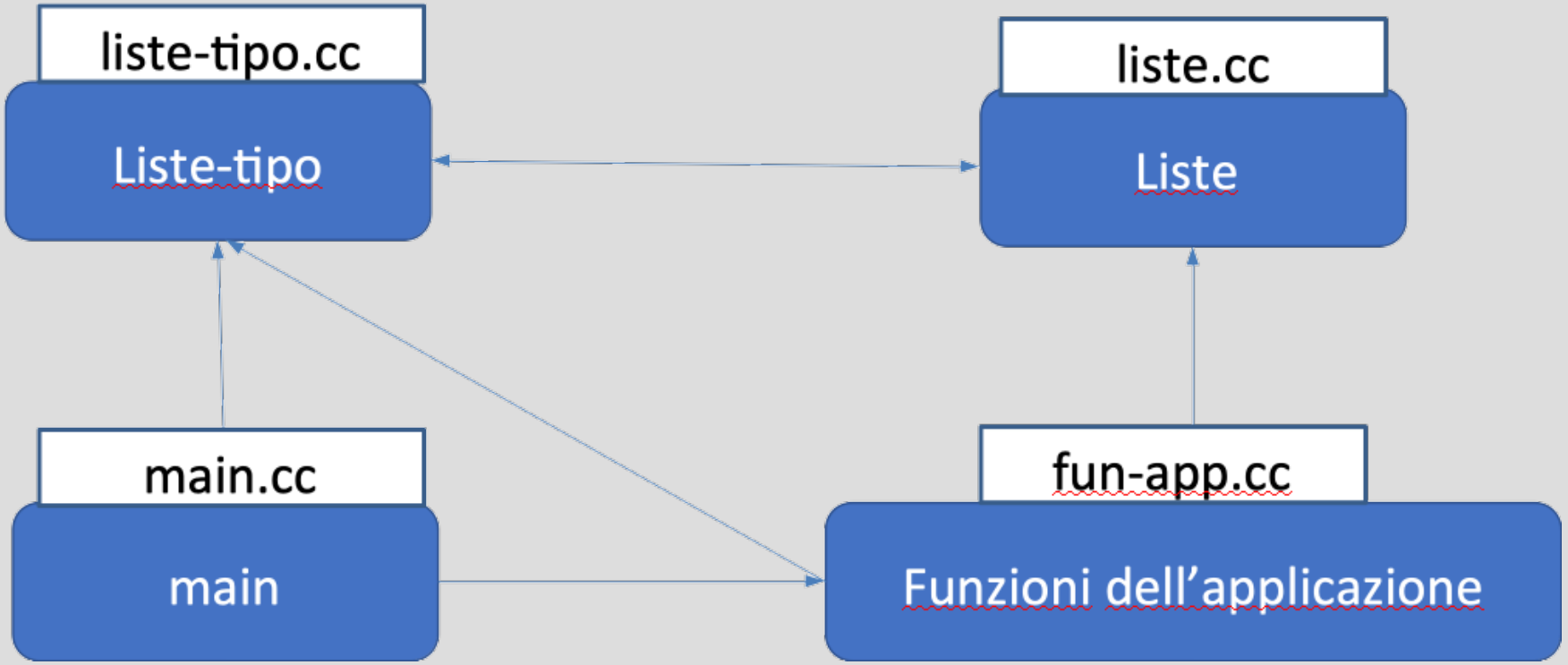
il compilatore non segnala comunque errori!

Dalla struttura logica alla struttura fisica

- La **struttura logica** di un programma (*suddivisione* in *moduli*) può essere realizzata attraverso la **struttura fisica** (*suddivisione in file sorgenti)*
- Proviamo a *distribuire su più file* il nostro programma
- La struttura logica precedentemente individuata ci aiuta molto:

  Possiamo mettere i *quattro moduli, m*ain, Funzioni dell'applicazione, Liste e Liste-tipo in quattro file sorgente distinti

Suddivisione in file



Ripetizione interfacce

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

- *Dichiarazione delle funzioni pubbliche (**prototipi**)*
- *Dichiarazione dei tipi di dati*
- *Dichiarazione delle eventuali variabili pubbliche (oggetti da dichiarare **extern**)*

Perché?

Per ragioni di visibilità

Per iniziare…

Fare 4 copie del singolo file iniziale: *liste-tipo.cc, liste.cc, fun-app.cc, main.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 (vedi schema logico)

Creare un workspace in Visual Studio Code e generare l'eseguibile a partire dai 4 file sorgenti

SOLUZIONE Vedi cartella *progetto_multifile_noheader*

Problemi

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

Approccio *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 file**)
- I file header hanno estensione `.h`
- Solitamente un file header contiene l'interfaccia di un modulo

Direttiva `#include`

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

```
using namespace std;

  #include <iostream>
```

- Lo stesso vale per l'inclusione di ogni altro **header file** necessario per utilizzare altri oggetti o funzioni di libreria
- E' una direttiva al preprocessore (ogni volta che un comando è preceduto da #, il comando è diretto al preprocessore)
- Il preprocessore sostituisce la riga di comando con l'intero contenuto del file argomento dell'include
- Dopo che il preprocessore ha risolto tutte le direttive, parte l'esecuzione della vera e propria compilazione

Sintassi `#include`

#include <nome_file>

Il file **nome_file** è cercato all'interno di un insieme di directory predefinite dove si trovano le librerie standard, come vedremo in dettaglio nella lezione sulla **compilazione**

#include "nome_file"

Il file **nome_file** è cercato nella stessa directory in cui è presente il file sorgente che contiene la direttiva

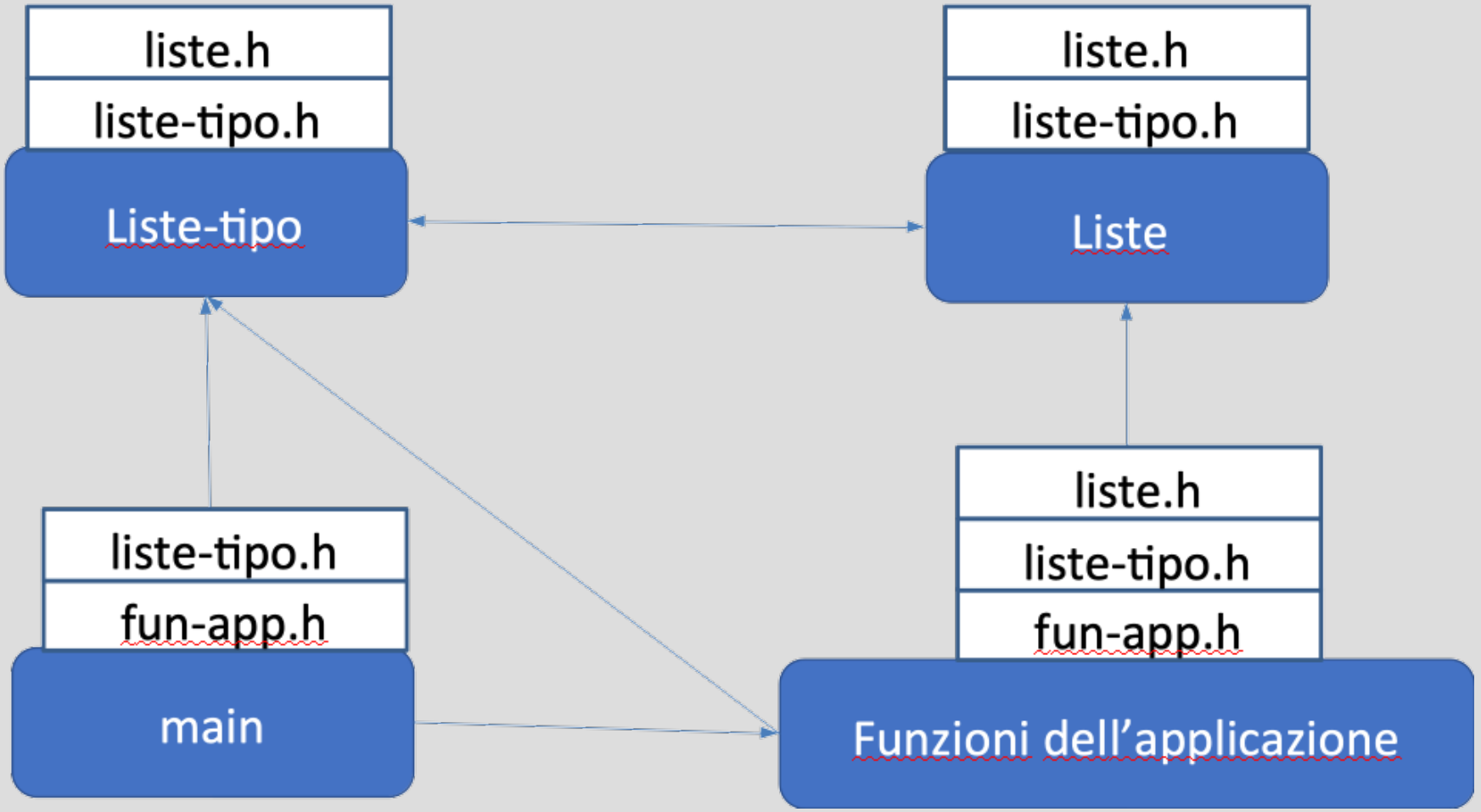
Moduli e header

- Applichiamo la seguente divisione per ogni modulo:

  Implementazione nel file *.cc*

  Interfaccia nell'**header file** (estensione *.h*)
- Anziché scrivere manualmente le dichiarazioni necessarie per utilizzare un modulo, includiamo il suo header file in tutti i file sorgente (.cc) dei suoi moduli clienti
- Se si *modifica* l'interfaccia di un modulo, è sufficiente modificare 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**

Inclusione di header file



Esercizio

Ristrutturare il progetto definendo ed includendo i precedenti 4 header file.
Rieseguire la compilazione con il comando g++ -Wall

ATTENZIONE L'ordine di inclusione dei file header nei file sorgente è fondamentale perché un header file può usare tipi definiti in un altro header file!!

SOLUZIONE **Vedi cartella** *progetto_multifile*

Protezione parte privata

- Potrebbe emergere la possibilità 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 (in questo caso parliamo di file .cc)
- E' sufficiente aggiungere la parola chiave **static** nella **dichiarazione**
- Si ha un errore a tempo di compilazione se tento di esportarle ed usarle in un altro modulo

ESEMPIO

```
static int fun();

static int x;
```

Interfaccia e tipi

- Dato un tipo (**struct, enum o typedef**) **condiviso** tra più file, **ogni file può accedere a tutti i campi della struttura**
- Quindi, anche se il tipo è pensato per essere manipolato solo attraverso funzioni dedicate, **non c'è alcun meccanismo a livello di linguaggio** che vieti di usare (alcuni) campi privati
- L'unica possibilità è inserire **commenti** e/o **documentazione** in cui si chiarisce quali campi di un tipo di dato struttura sono da considerare **pubblici** e quali **privati**
- Questo è il caso, ad esempio, di `struct elem`

Class

- In questo corso usiamo il linguaggio C++ come linguaggio procedurale
- Il C++ è anche un **linguaggio object-oriented** e il blocco fondamentale quando si usa il C++ come linguaggio object-oriented è class

`class:` tipo di dato definito dall'utente che permette di definire **interfaccia** ed **implementazione** di ogni oggetto nel codice stesso

- Il tipo **class** sintatticamente può essere considerato una estensione del tipo **struct**
- Oltre alle 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 private in class

- Nella dichiarazione di un tipo **class** si può dichiarare esplicitamente quali campi e quali metodi 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**

Non vedremo il tipo **`class`** in questo corso

Lo vedrete nel corso di **Linguaggi di Programmazione ad Oggetti** nell'ambito del linguaggio **Java**

Un ultimo sforzo

Dobbiamo ancora raggiungere pienamente l'obiettivo 2!!

- Il modulo liste-tipo contiene primitive la cui interfaccia (e implementazione) dipende dal tipo dei valori della lista
- Come possiamo rendere queste primitive indipendenti dal tipo?

Una possible soluzione

- Introduciamo il tipo di dato `tipo_inf` e usiamo `tipo_inf` nei dichiarazioni e nelle definizioni delle primitive del tipo di dato lista
- `tipo_inf` sarà una stringa, un valore numerico, una struct, … in base al tipo di lista che si vuole realizzare
- Introduciamo un nuovo modulo, il modulo «tipo», dove dichiariamo `tipo_inf` e tutte le primitive per la gestione dello specifico tipo di dato
- Riuniamo tutte le primitive sulle liste nel modulo «liste»

Il modulo “tipo”

Il modulo «tipo» contiene:

- La definizione del tipo `tipo_inf`
- La primitiva `int compare(tipo_inf, tipo_inf)` per confrontare valore di tipo `tipo_inf`
  `compare(v1,v2)`
  - Restituisce 0 se v1=v2
  - Restituisce valore <0 se v1<v2 (rispetto alla relazione d’ordine definita su tipo_inf)
  - Restituisce valore >0 se v1>v2
- La primitiva `void copy(tipo_inf&,tipo_inf)` che copia il contenuto del secondo parametro nel primo parametro
- La primitiva `void print(tipo_inf)` che stampa il valore

Tutti gli altri moduli *non devono* manipolare direttamente i valori del tipo definito ma *usare* le corrispondenti primitive

Vediamo l'esempio search

```
elem* search(lista l, char* v) {
        while(l!=NULL)
                if(strcmp(head(l),v)==0)
                        return l;
                else
                        l=tail(l);
        return NULL;}
```

Il tipo di v (char*) è specificato direttamente nell'interfaccia della funzione search.

```
int compare(tipo_inf s1,tipo_inf s2){
        return strcmp(s1,s2);}

elem* search(lista l, tipo_inf v) {
        while(l!=NULL)
                if (compare(head(l),v)==0)
                        return l;
                else
                        l=tail(l);
        return NULL;}
```

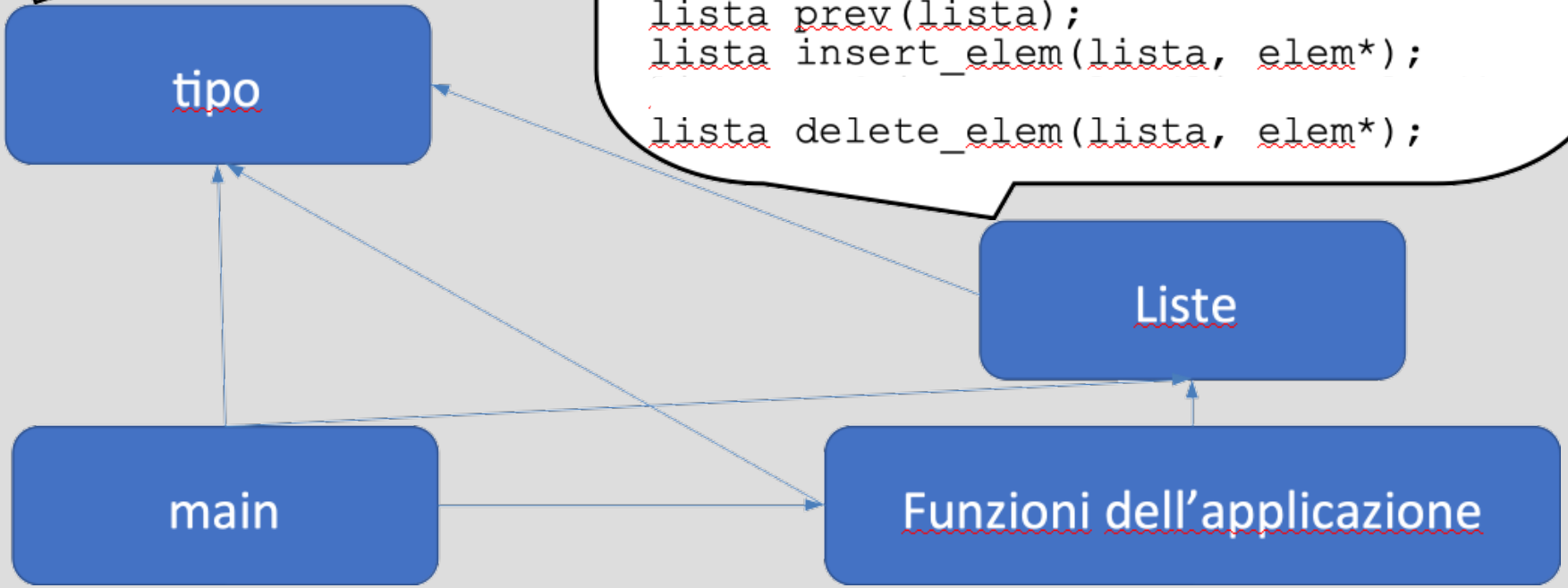
Compare fa un confronto tra due variabili tipo_inf

Search usa compare. Se volessi usare un tipo diverso (ad esempio int), dovrei solo modificare la funzione compare, e search rimarrebbe identica.

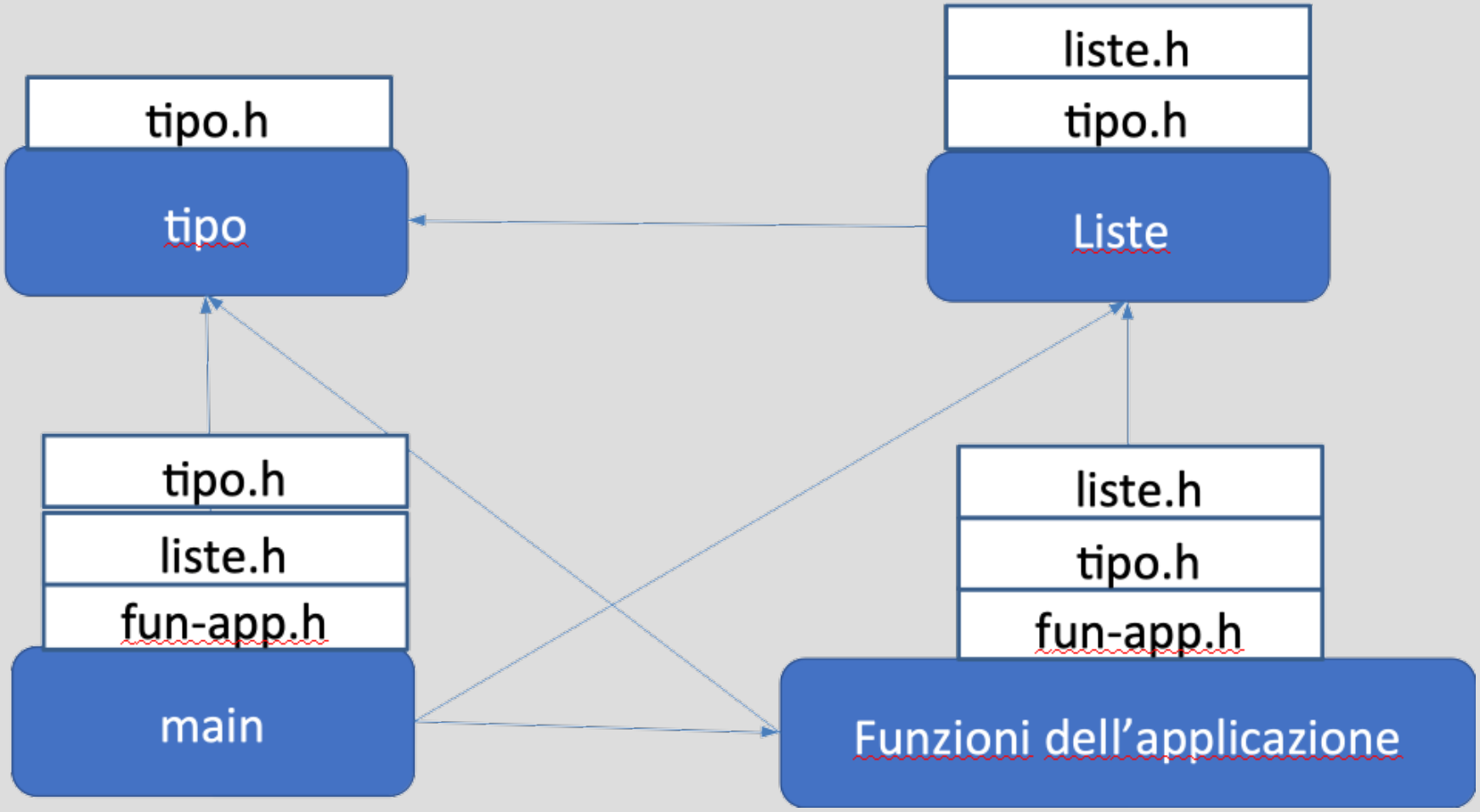
Nuovo schema logico

```
typedef char* tipo_inf;

int
compare(tipo_inf,tipo_inf);
void
copy(tipo_inf&,tipo_inf);
void print(tipo_inf);
```

```
struct elem
{   tipo_inf inf;
        elem* pun ;
        elem* prev;
};
typedef elem* lista ;

tipo_inf head(lista);
elem* new_elem(tipo_inf);
elem* search(lista, tipo_inf);

lista tail(lista);
lista prev(lista);
lista insert_elem(lista, elem*);

lista delete_elem(lista, elem*);
```



Inclusione file header



Obiettivo 2 raggiunto!!

Ora abbiamo due moduli distinti

- **Modulo «liste» :** modulo per la gestione delle liste che è indipendente dallo specifico tipo di dato memorizzato nella lista perché usa il tipo di dato e le primitive dichiarate nel modulo «tipo»
- **Modulo «tipo» :** modulo per la gestione del tipo di dato memorizzato nella lista.
  L'implementazione del modulo (del tutto trasparente a chi lo usa) dipenderà dalle esigenze applicative.

Esercizio

A partire dalla cartella *progetto_multifile* rivedere il progetto usando il nuovo schema logico e implementando il modulo «tipo» per lo specifico tipo di dato dell'applicazione (URL di siti web)

Un possibile approccio:

1. Rinominare il modulo «liste-tipo» in «tipo»
2. Definire il tipo di dato tipo_inf
3. Rivedere tutti i prototipi sostituendo char* con tipo_inf
4. Spostare i metodi delle liste nel modulo «liste» (aggiornando .cc e .h di conseguenza)
5. Sistemare gli include sulla base dello schema logico

Esercizio (cont.)

6. Aggiungere le nuove primitive al modulo «tipo»
7. Rivedere tutti i moduli che usano il modulo «tipo» affinché non usino direttamente char* quando manipolano il contenuto delle liste ma il tipo con le sue nuove primitive

CONSIGLIO Ricompilare il progetto quando possibile

SOLUZIONE **Vedi cartella** *progetto_multifile_tipo*

Esercizio (cont.)

Dove siete intervenuti per rendere i moduli «liste» e «fun-app» indipendenti dal tipo?

- Modulo «liste»: prototipi e implementazioni delle primitive `head`, `search` e `new_elem`
- Modulo «fun-app»: implementazione di `stampalista` e `naviga` per la stampa del valore

Esercizio 2

Se siete intervenuti correttamente nei vari moduli allora la seguente operazione non dovrebbe creare problemi:

- Cambiate la tipologia di dati gestiti, non più stringa ma intero
- Create un Nuovo modulo “tipo” che definisce il tipo intero e implementate le corrispondenti primitive
- Sostituite nel Vostro progetto il modulo del tipo stringa con il modulo del tipo intero
- Ricompilate e… non dovreste riscontrare errori!!
- Lanciate l’eseguibile e questa volta in input potrete fornire degli interi