7 - Modularizzazione e sviluppo su più file

Aumentando di dimensioni i programmi crescono di complessità rendendo quindi fondamentale l'attività di progettazione. Nel caso di programmi di medio-grandi dimensioni occorre una progettazione a un livello di astrazione molto più elevato.

Lo sviluppo di sistemi sw complessi richiede la divisione del lavoro in modo da contenere tempi e costi di sviluppo. Occorre mantenere la qualità del sw in termini di:

- affidabilità
- prestazioni
- modificabilità
- estensibilità
- riutilizzo

Suddivisione in moduli

Modulo: insieme di funzioni e strutture dati logicamente correlate in base a un qualche principio significativo. Tipicamente *fornisce una serie di servizi e/o può implementare una certa struttura dati* (o tipo di dato).

Programmazione modulare: approccio necessario alla programmazione che, tramite un processo di astrazione, consente di gestire la complessità del sistema suddividendolo in parti (moduli) tra loro correlate.

Astrazione: il processo che porta ad individuare e considerare le proprietà rilevanti di un'entità.

I meccanismi di astrazione più diffusi sono:

- Astrazione sul controllo (linguaggi procedurali): consiste nell'astrarre una data funzionalità dai dettagli della sua implementazione.
- Astrazione sui dati (linguaggi a oggetti): astrarre le entità costituenti il sistema, descritte in termini di una struttura dati e delle operazioni possibili su di essa.

Esempio sulla gestione di una lista doppia.

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 cambiare il tipo di informazione e l'implementazione della lista senza intervenire sulle funzionalità di gestione e del tipo di dato.

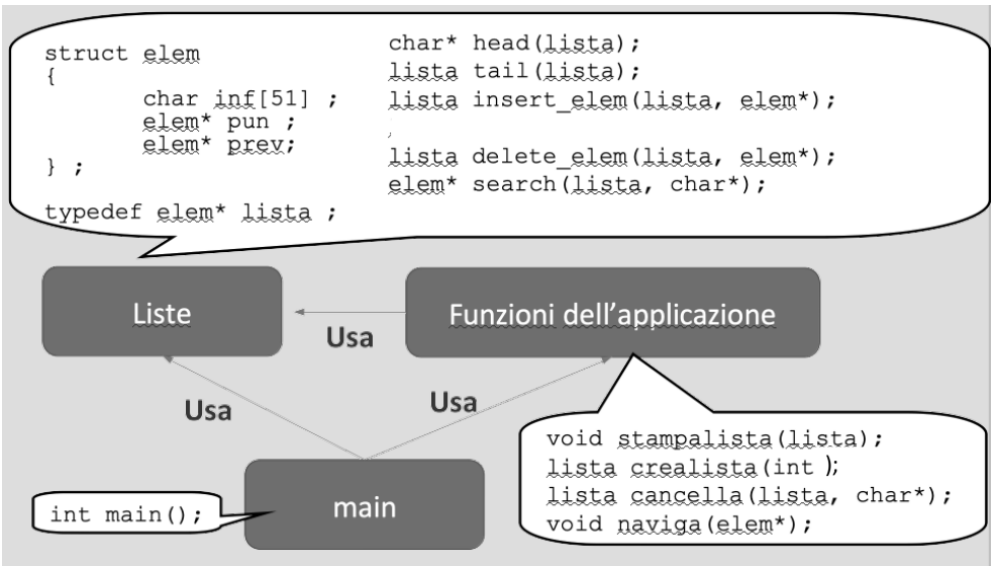
Modulo "liste": contiene la definizione del tipo di dato lista e le primitive per l'implementazione delle liste. (almeno una delle funzioni va usata)	**Tipo di dato**
	`struct elem`
	`{`
	`char inf[51] ;`
	`elem* pun ;`
	`elem* prev;`
	`} ;`
	`typedef elem* lista ;`
	
	Primitive
	`char* head(lista);`
	`lista tail(lista);`
	`lista insert_elem(lista, elem*);`
	`lista delete_elem(lista, elem*);`
	`elem* search(lista, char*);`

Modulo "funzioni dell'applicazione": contiene le funzioni che realizzano l'applicazione	**Funzioni**
	`void stampalista(lista);`
	`lista crealista(int);`
	`lista cancella(lista, char*);`
	`void naviga(elem*);`
	`int main();`

Modulo "main": contiene la specifica del menu e la modalità di interazione con l'utente	

Modulo clienti: modulo che utilizza i servizi/oggetti forniti da altri moduli. Nell'esempio il modulo "funzioni dell'applicazione" e "main" sono moduli clienti.

SCHEMA LOGICO



La difficoltà nell'attività di progettazione è dover scegliere una soluzione all'interno di uno spazio di soluzioni più o meno vasto.

Problema che va in contrasto con l'obiettivo 1: il modulo "*funzioni dell'applicazione*" usa le primitive del modulo "*liste*" senza aver alcuna nozione sulla modalità d'implementazione ma la funzione `crealista` usa esplicitamente il tipo di elemento.

```
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,p);
  }
  return testa ;
}
```

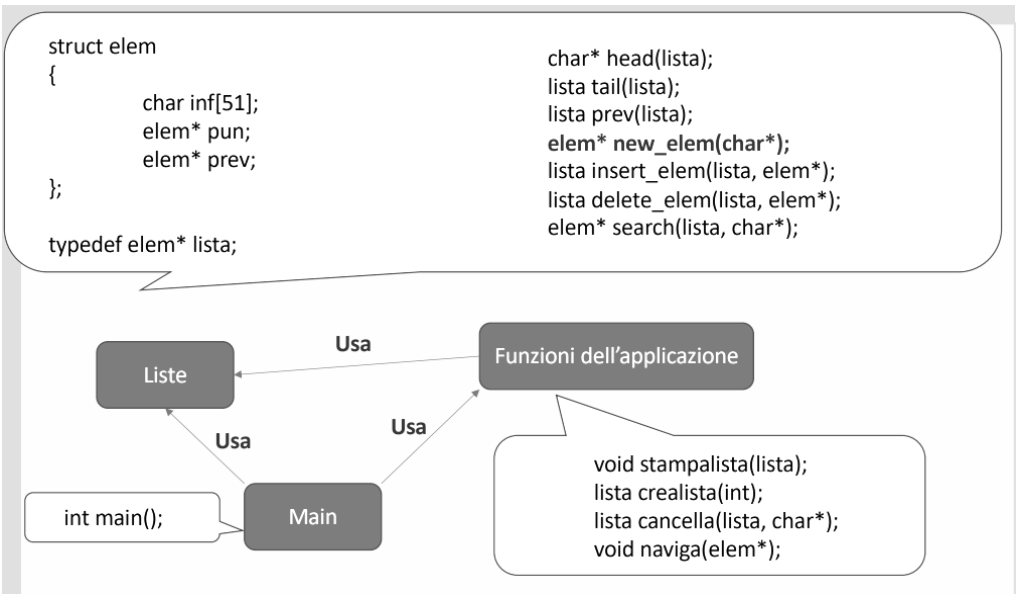
Soluzione: definire una nuova primitiva elem* new_elem(char*)che alloca un nuovo elemento e gli assegna il valore informativo in ingresso.

```
elem* new_elem(char* newinf){
    elem* p = new elem ;
    strcpy(p->inf, newinf);
    p->pun=p->prev=NULL;
    return p;
}
```

```
lista crealista(int n)
{
    lista testa = NULL ;
    char inf[51];
    for (int i = 1 ; i <= n ; i++) {
        cout<<"URL "<<i<<": " ;
        cin>>inf;
        testa=insert_elem(testa,new_elem(inf));
    }
    return testa ;
}
```

Ora l'obiettivo 1 è raggiunto!

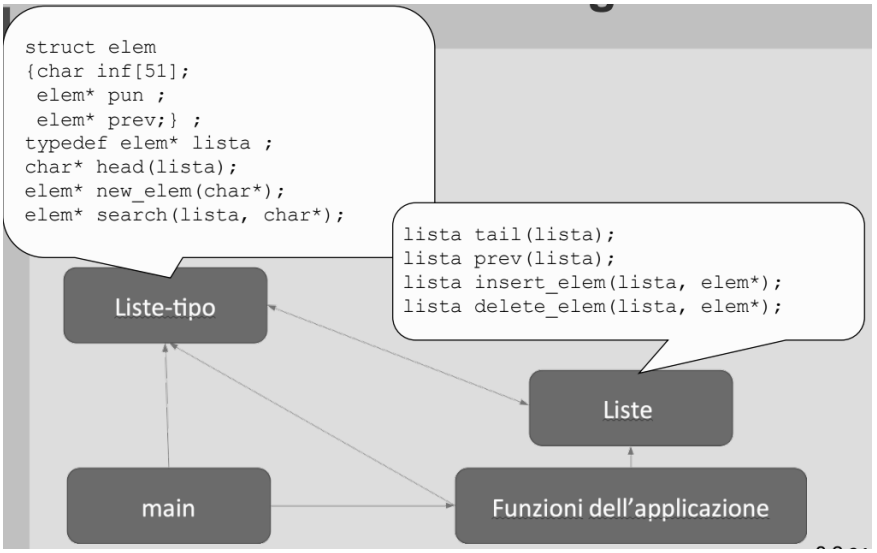
NUOVO SCHEMA LOGICO



Problema che va in contrasto con l'obiettivo 2: l'interfaccia e l'implementazione delle liste sono indipendenti dal tipo di dato mentre alcune primitive sono dipendenti dal tipo di dato (del campo inf).
Soluzione: si divide quindi il modulo *"liste"* in due moduli.

Indipendenti - Modulo "Liste"	**Dipendenti - Modulo "Liste-tipo"**
lista prev(lista); lista tail(lista); *lista insert_elem(lista, elem*);* *lista delete_elem(lista, elem*);*	*char* head(lista);* *elem* search(lista, char*);* *elem* new_elem(char*);*

NUOVO SCHEMA LOGICO



Realizzazione di un modulo

Affinchè un modulo sia chiaro e efficiente occorre che la sua struttura logica sia formata da:

- **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.

Intestazione (header): rappresenta l'interfaccia del modulo. Ha al suo interno il prototipo di ciascuna funzione e le strutture dati.

Per ogni modulo si raggruppano poi le definizioni delle funzioni nel corpo del modulo.

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

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

Si definiscono:

- **Funzioni e Strutture dati Pubbliche** le funzioni e le strutture dati accessibili dagli utenti di un modulo; tutto ciò corrisponderà all'interfaccia
- **Funzioni e Strutture dati Private** tutte le altre strutture dati e funzioni del modulo; tutto ciò corrisponderà all'implementazione.

La suddivisione tra interfaccia e implementazione viene fatta perchè **il cliente non deve conoscere tutti i dettagli di un modulo**. Chi costruisce un modulo è libero di farlo come vuole purché rimanga inalterata l'interfaccia. Un esempio sono i driver dei dispositivi fisici dei calcolatori. L'interfaccia va rispettata affinchè il cliente riesca ad utilizzare il modulo correttamente.

In conclusione definiamo chiaramente l'interfaccia dei moduli ed accediamo ai servizi dei moduli solo tramite l'interfaccia.

Application Programming Interface (API): interfaccia di un modulo software (applicazione); rappresentano un insieme di procedure disponibili al programmatore (es. API per accedere a dati dei social network o web service, esistono API a pagamento).

In C++ un programma può essere distribuito su più file sorgenti consentendo l'implementazione dei moduli e l'uso da parte dei clienti. Se in un programma ho bisogno di un determinato modulo includerò i corrispondenti file sorgenti nel programma.

È buona norma tenere separate la specifica del modulo dalla sua implementazione affinchè il cliente ne ignori l'implementazione.

L'interfaccia è rappresentata da un HEADER file `file.h` che può essere utilizzata mediante i meccanismi di inclusione forniti dal linguaggio (direttiva `#include`).

Compilazione di un programma con più file sorgenti (l'ordine non ha importanza)

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

Nozioni necessarie:

- **Unicità della funzione main:** i programmi iniziano sempre dalla prima istruzione della funzione main. La funzione main() deve essere definita solo in un file sorgente.
- **Visibilità (scope) a livello di file**: un identificatore dichiarato in un file sorgente (fuori dalle funzioni) è visibile dal punto in cui viene dichiarato in poi; si parla di visibilità di file. Se la dichiarazione è una definizione allora si dice che l'entità 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 (linkage) interno ed esterno:** un identificatore ha collegamento esterno se si riferisce ad un'entità accessibile anche da altri file rispetto a quello in cui è definito; l'entità deve essere unica e globale al programma (tutti i file). L'entità può essere definita in un solo file ma può essere dichiarata in più file. Solo gli identificatori di entità a livello di file hanno collegamento esterno.

ESEMPI0
```
int x;          // fun e x hanno
fun(int){….}  // un collegamento esterno
```

Keyword extern

Affinchè un identificatore a livello di file sia visibile da un file esterno bisogna aggiungere una keyword → `extern`.

id definito nel fileY.cc visibile nel file sorgente fileX.cc.

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

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

id è una variabile	*id è una funzione*
Occorre che id sia definito almeno in un file e, per poterci accedere da un altro file, occorre ridichiararlo con la keyword extern.	Se id si riferisce a una funzione, univocamente definita, basta semplicemente ripeterne la dichiarazione. `extern int fun()` equivale 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() { … }
```

Keyword static

Ogni id definito a livello di file ha collegamento esterno, a meno che non venga definito con la keyword `static`. In questo caso:

- hanno collegamento interno
- non sono visibili al di fuori del file di definizione

Definizione entità a livello di blocco	
Senza `static`	Con `static`
• Non ha collegamento (nemmeno interno) • Tempo di vita pari a quel blocco	• Il collegamento non varia • l'entità rimane visibile solo all'interno del blocco • Cambia il tempo di vita: diventa 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
}
```

Linker ⇒ gestisce i collegamenti e i suoi relativi errori
Compilatore ⇒ gestisce la visibilità e i suoi relativi errori.

Dichiarazioni di tipo

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.

Occorre che i due tipi siano equivalenti (per nome).

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

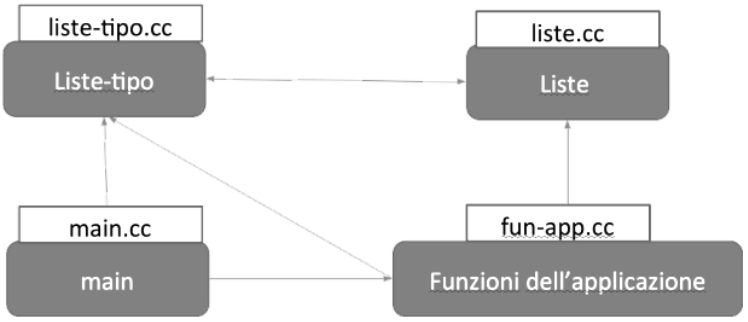
```
// file2.cc
struct ss {char c ; short z ;} ;
void fun(ss) ;
int main() { ss k ; fun(k); …}
```

In caso ci siano due file con due strutture equivalenti ma internamente diverse se si invoca una funzione, tra cui nei parametri attuali ha anche una struttura, definita in un altro file l'oggetto passato verrà trattato come se appartenesse alla struttura definita nel file in cui la funzione è invocata.

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*).

La struttura logica fatta in precedenza ci suggerisce di mettere i 4 moduli in 4 file sorgenti distinti.



Per poter utilizzare i servizi di uno degli altri moduli, un file sorgente deve ripeterne l'interfaccia per ragioni di visibilità e pertanto dovrà avere:

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

Problema: la ripetizione di dichiarazioni diverse volte nei file sorgenti è error prone.

La soluzione risiede nei **file di intestazione** (header file) che avranno estensione .h. Questi permetteranno di avere **consistenza** fra dichiarazioni effettuate in file sorgenti diverse.

I file di intestazione costituiranno delle direttive al preprocessore (fatte con #include) che si occuperà di sostituire la riga di comando con tutto il contenuto del file argomento dell'include. Solo dopo aver risolto le direttive partirà la vera e propria compilazione.

`#include <xxx>` ⇒ si stanno includendo le dichiarazioni di tutte le funzioni contenute in xxx e ciò ci consentirà di usarle.

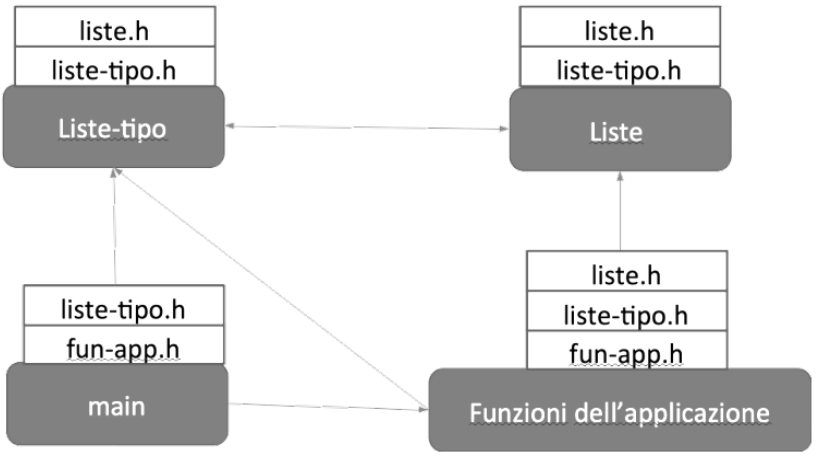
Direttiva include

Sintassi direttiva include:

- `#include <nome_file>` ⇒ il file nome_file è cercato all'interno di un insieme di directory predefinite dove si trovano le librerie standard.
- `#include "nome_file"` ⇒ e è cercato nella stessa directory in cui è presente il file sorgente che contiene la direttiva

Per ogni modulo applichiamo la seguente divisione per ogni modulo:

- **Implementazione nel file.cc** ⇒ anzichè scrivere manualmente le dichiarazioni necessarie includiamo il suo header file in tutti i file sorgenti (.cc) dei suoi moduli clienti.
- **Interfaccia nell'header file (estensione.h)** ⇒ se si modifica l'interfaccia di un modulo è sufficiente modificare l'header in modo che all'atto della compilazione tutti i file sorgenti che usano il modulo disporranno correttamente della nuova interfaccia grazie all'inclusione dell'header file.




```
static int fun();
static int x;
```

Keyword *static* → forza il collegamento interno per entità definite a livello di file (parliamo di file.cc). Si avrebbe un errore a tempo di compilazione se tento di esportare fun e usarle in un altro modulo.

I campi dei tipi (struct, enum o typedef) condivisi tra più file sono accessibili da ogni file. 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.

Problema (obiettivo 2): il modulo `liste-tipo` contiene primitive la cui interfaccia (e implementazione) dipende dal tipo dei valori della lista.

Soluzione: si introduce il tipo di dato `tipo_inf` in modo da usarlo nelle dichiarazioni e nelle definizioni delle primitive del tipo di dato lista. Tutte le volte che si utilizza il campo informazione si utilizzerà il `tipo_inf` (astrazione).

Si procede quindi creando un nuovo modulo "`tipo`" dove dichiariamo t`ipo_inf` e tutte le primitive per la gestione dello specifico tipo di dato. Riuniamo tutte le primitive sulle liste nel modulo «liste».

Modulo "tipo":

- Definizione del tipo `tipo_inf`
- Definizione della primitiva `int compare(tipo_inf, tipo_inf)` per confrontare due valori tipo_inf.
    - restituisce 0 se v1=v2
    - Restituisce valore <0 se v1<v2
    - Restituisce valore >0 se v1>v2

  L'implementazione del compare viene decisa a tempo di scrittura del programma in base al tipo di dato che si ha.
- 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

Questo modulo viene fatto affinché tutti gli altri moduli NON manipolino direttamente i valori del tipo definito ma usino le corrispondenti primitive.

Esempio search (prima e dopo il modulo "tipo")

PRIMA

```
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.

DOPO

```
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;}
```

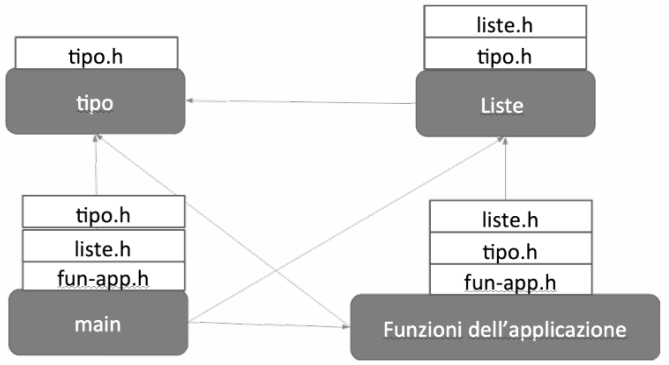
Compare fa un confronto tra due variabili tipo_inf

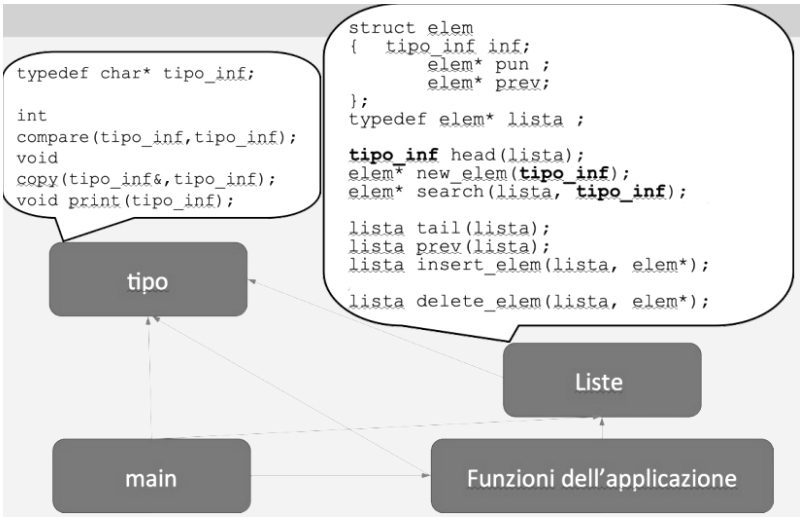
Search usa compare. Se volessi usare un tipo diverso (ad esempio int), dovrei solo modificare la funzione compare, e search rimarrebbe identica.

INCLUSIONE DEI FILE HEADER

NUOVO

SCHEMA LOGICO





L'obiettivo 2 è stato raggiunto infatti ora si hanno 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

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!!! Va sempre incluso per primo liste-tipo.h perchè contiene la definizione della struttura.