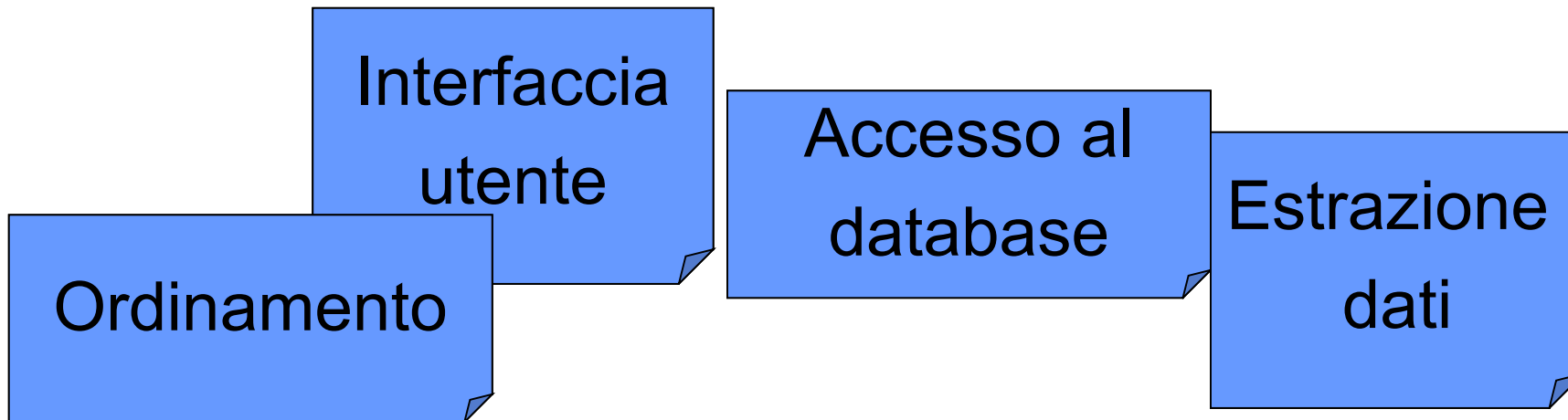


Progetti su più file

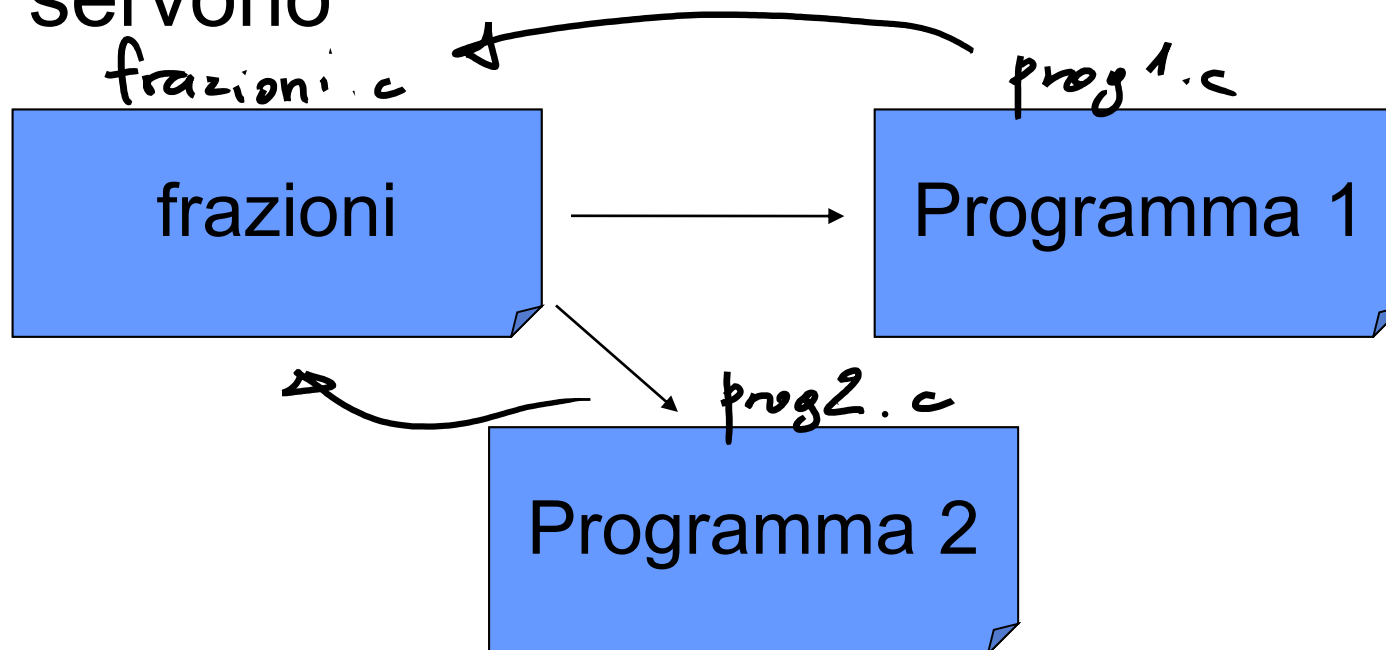
PROGETTI SU PIU' FILE

- Un'applicazione complessa non può essere fatta in un unico file .c: sarebbe ingestibile
- Meglio dividerla in vari moduli, che si occupano di aspetti diversi



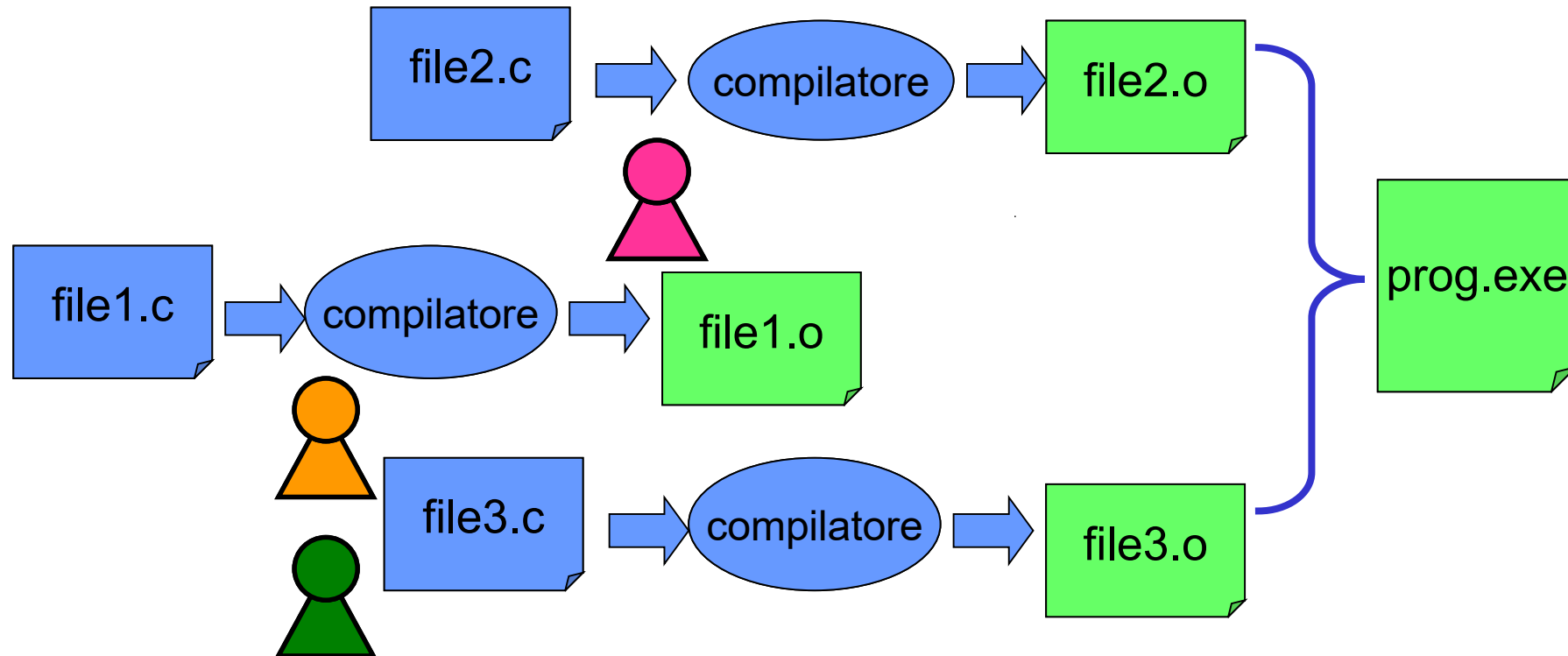
LIBRERIE RIUSABILI

- Una volta che ho scritto una libreria di funzioni, voglio poterle usare senza fare copia-incolla delle funzioni che mi servono



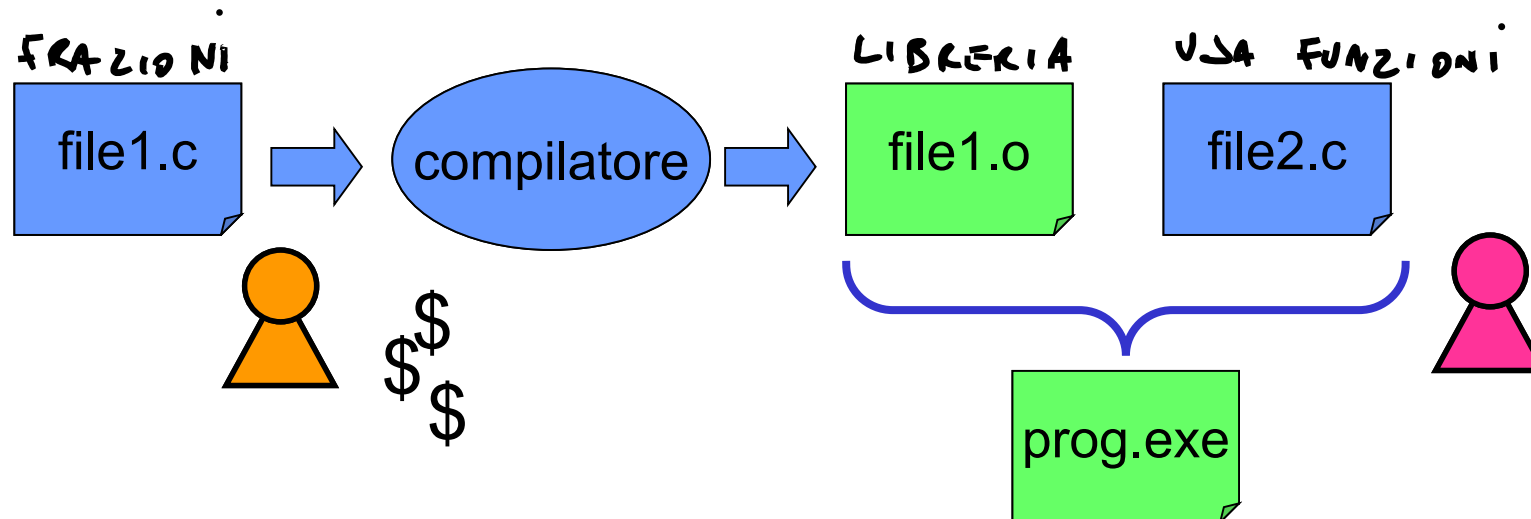
LAVORO IN TEAM

- Per suddividere il lavoro è necessario che ciascuno possa compilare separatamente il suo codice



VENDITA DI LIBRERIE

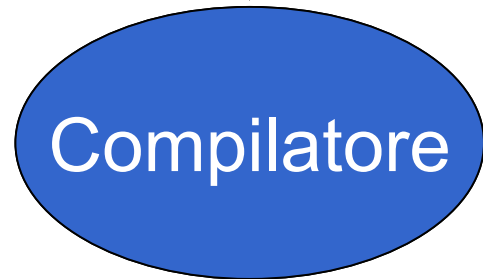
- Se vendo il sorgente, il cliente viene a sapere la tecnologia che ho usato
- Voglio vendere il compilato: l'acquirente deve essere in grado
 - di usare la mia libreria
 - ma non di vedere come è fatta



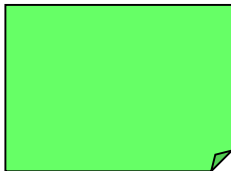
PROGETTI STRUTTURATI SU PIU' FILE

```
int f(int n)
{ return n*2;
}
```

f.c

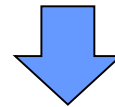
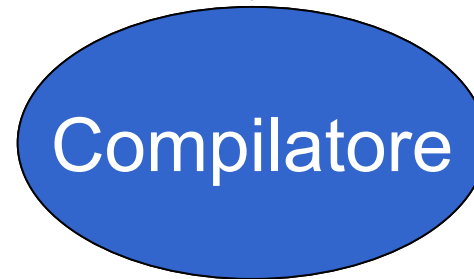
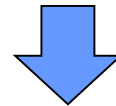


f.o



```
main()
{ int y = f(3);
}
```

main.c



ERRORE

~~int f(int n)~~
~~{ ... }~~
~~main()~~
~~{ ... }~~
~~prog.c~~
~~main.c~~

PERCHE' DA' ERRORE?

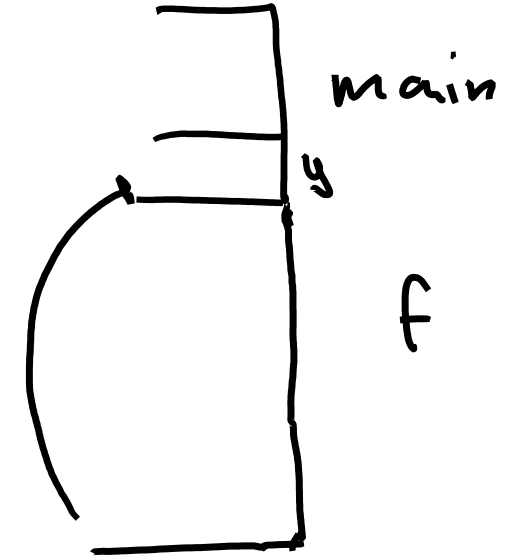
- crea sullo stack il record di attivazione del **main**
- invocazione di **f**:
 - crea sullo stack il record di attivazione di **f**
 - copia i parametri (eventualmente, con conversioni implicite di tipo)
 - salta ad eseguire **f**
- assegna a **y** il valore di ritorno (eventualmente, con conversione implicita di tipo)

main.c

```
main()
{int y=f(3);
}
```

Come faccio a sapere se la chiamata è corretta?

Come faccio a sapere dove comincia la funzione **f**?



SECONDO PROBLEMA

- crea sullo stack il record di attivazione del main
- invocazione di f:
 - crea sullo stack il record di attivazione di f
 - copia i parametri (eventualmente, con conversioni implicite di tipo)
 - salta ad eseguire ????
- assegna ad y il valore di ritorno (eventualmente, con conversione implicita di tipo)

Non scrive il
compilatore qual è
l'indirizzo: lo
inserirà il linker


PRIMO PROBLEMA

Come faccio a sapere come è fatto il record di attivazione?

- Per costruire il record di attivazione, ho bisogno di sapere
 - quanti sono i parametri
 - di che tipo sono
- Per sapere se devo fare conversioni implicite devo sapere anche
 - di che tipo è il valore di ritorno
- In pratica, devo sapere qual è **l'interfaccia della funzione**

main.c

```
int f(int n);
main()
{int y=f(3);
}
```

 **int f(int n)**

- **Soluzione:** riporto nel file `main.c` l'interfaccia della funzione di cui ho bisogno

DICHIARAZIONE DI FUNZIONE

La dichiarazione di una funzione è costituita dalla **sola interfaccia**, *senza corpo* (sostituito da un **;**)

<dichiarazione-di-funzione> ::=

<tipoValore> **<nome>** (<parametri>) **;**

int f int n



DICHIARAZIONE DI FUNZIONI

Dunque, per usare una funzione

- non occorre conoscere tutta la *definizione*
- *basta conoscere la dichiarazione*, perché essa specifica proprio il *contratto di servizio*

```
int f(int n)
{ return n*2;
}
```

```
int f(int n);
```

DICHIARAZIONE vs. DEFINIZIONE

```
int f (int n) ;
```

La dichiarazione di una funzione costituisce **solo una specifica** delle proprietà del componente:

- Può essere duplicata senza danni
- Un'applicazione può contenerne più di una
- La compilazione di una dichiarazione non genera codice macchina

```
int f(int n)
{ return n*2;
}
```



La definizione di una funzione costituisce **l'effettiva realizzazione** del componente

- Non può essere duplicata
- Ogni applicazione deve contenere una e una sola definizione per ogni funzione utilizzata
- La compilazione della definizione genera il codice macchina che verrà eseguito ogni volta che la funzione verrà chiamata.

DICHIARAZIONE vs. DEFINIZIONE

- La definizione è *molto più* di una dichiarazione

definizione = dichiarazione + corpo



La definizione funge anche da dichiarazione
(ma non viceversa)

```
int f(int n) {  
    ...  
    main() {  
        ... = f(s) + 2;  
    }  
}
```

ESEMPIO SU DUE FILE

File `main.c`

```
int f(int);
```

 → Dichiarazione della funzione

```
main()  
{ int y = f(3);  
}
```

 → Chiamata della funzione

File `f.c` .



Definizione della funzione

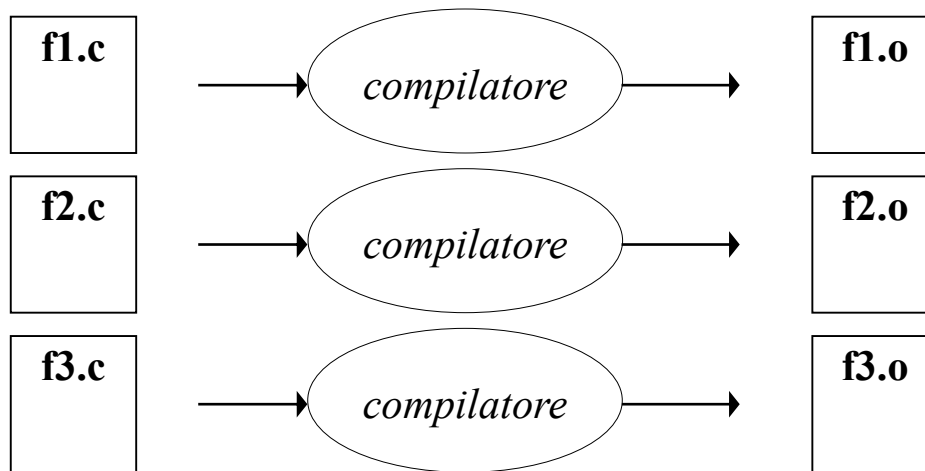
```
int f(int n)  
{ return n*2;  
}
```

gcc prog.c → a.out

COMPILAZIONE DI UN'APPLICAZIONE

1) Compilare i singoli file che costituiscono l'applicazione

- File *sorgente*: estensione **.c**
- File *oggetto*: estensione **.o** o **.obj**

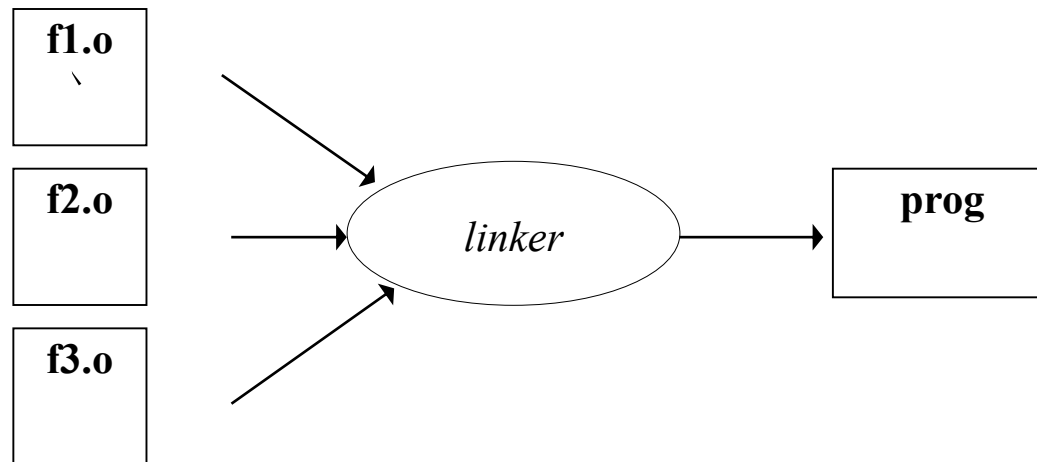


gcc -c f1.c → f1.o

COMPILAZIONE DI UN'APPLICAZIONE

2) Collegare i file oggetto fra loro e con le librerie di sistema

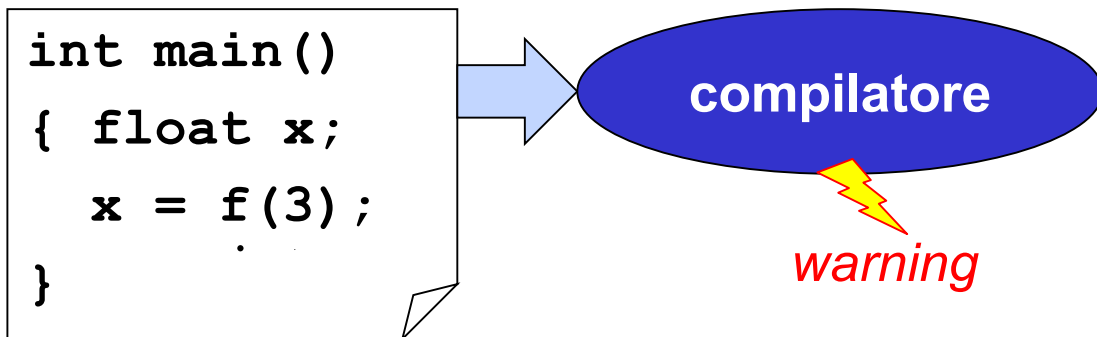
- File *oggetto*: estensione **.o** o **.obj**
- File *eseguibile*: estensione **.exe** o nessuna



ld
 \uparrow
 $gcc -o prog f1.o f2.o f3.o$
 LINKING

COMPILAZIONE DI UN'APPLICAZIONE

- ogni cliente che *usa* una funzione deve incorporare la dichiarazione opportuna
 - in ogni file, ogni funzione che viene usata deve essere stata prima dichiarata
 - se la dichiarazione manca, si ha
 - un warning in fase di compilazione nel file del cliente (alcuni compilatori non danno il warning).
 - il compilatore si “*inventa*” una dichiarazione, che poi deve coincidere con la dichiarazione vera perché il linking vada a buon fine!



non deve mancare definizione di main

LINKING DI UN'APPLICAZIONE

ogni funzione deve essere definita una e una sola volta in uno e uno solo dei file sorgente

- se la definizione manca, oppure è duplicata, si ha errore di linking

f1.c

```
float f(int y);  
int main()  
{ float x;  
  x = f(3);  
}
```

compi-
latore

f1.o

compi-
latore

g.o

linker

ERRORE

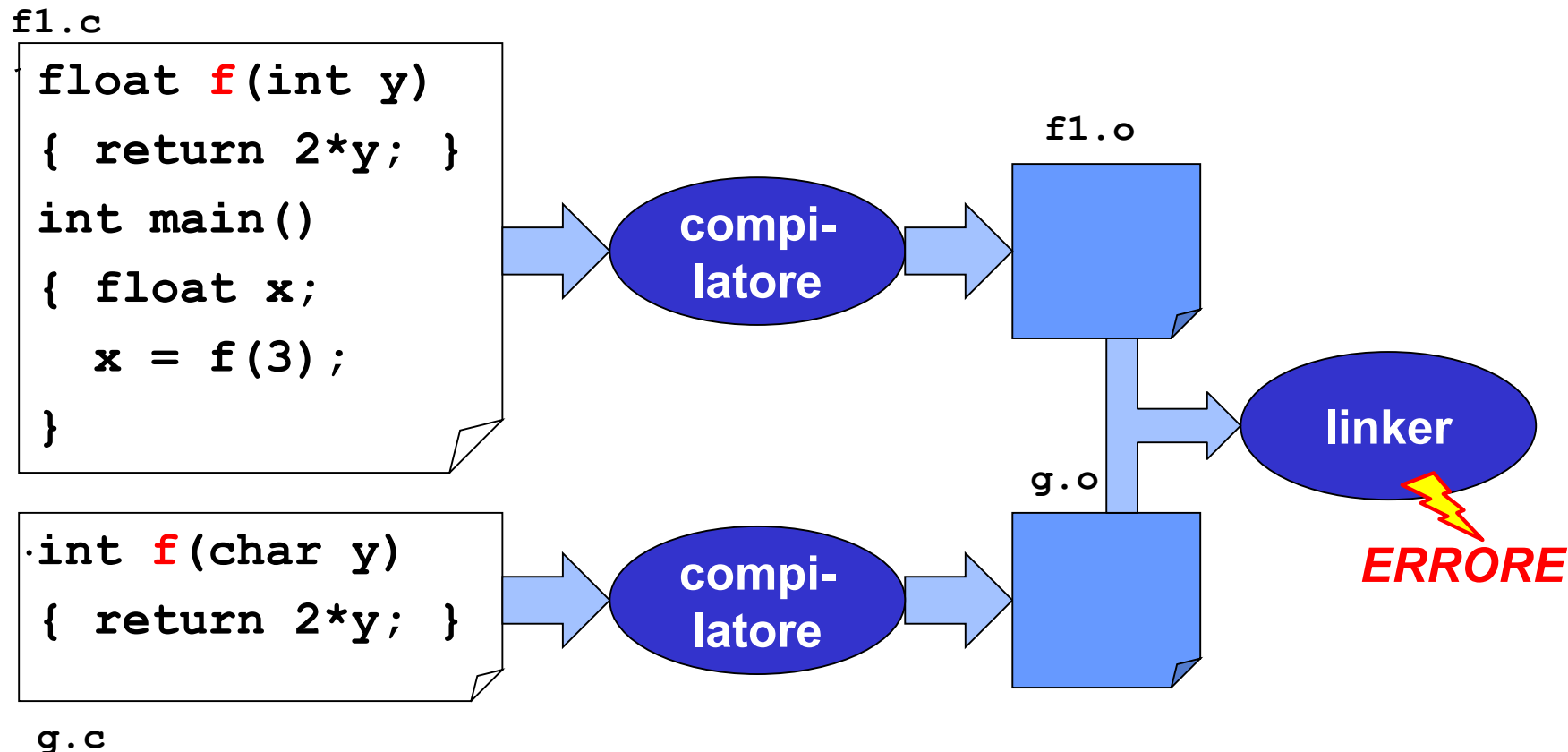
```
int g(int x)  
{ return x+1; }
```

g.c

LINKING DI UN'APPLICAZIONE

ogni funzione deve essere definita una e una sola volta in uno e uno solo dei file sorgente

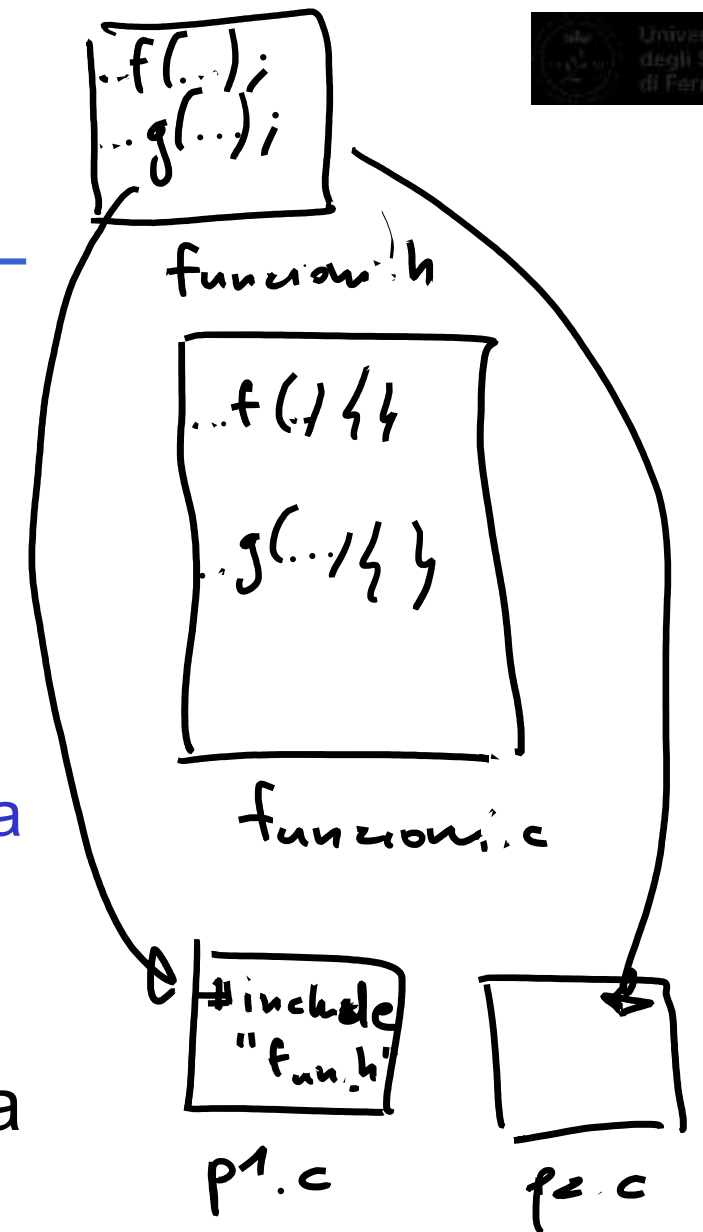
- se la definizione manca, oppure è duplicata, si ha errore di linking



#include <stdio.h>

HEADER FILE

- Per automatizzare la gestione delle dichiarazioni, si introduce il concetto di *header file (file di intestazione)*
 - contenente *tutte le dichiarazioni* relative alle funzioni definite nel componente software medesimo
 - scopo: *evitare ai clienti di dover trascrivere riga per riga* le dichiarazioni necessarie
- *basterà includere l'header file* tramite una direttiva `#include`.



DIRETTIVA #include

- `#include` è una direttiva del preprocessore
- Il preprocessore sostituisce testualmente la direttiva con il contenuto del file incluso

- Due formati: `stdio.h`

`#include <libreria.h>`

include l'header di una *libreria di sistema*
il sistema sa già dove trovarlo

`#include "miofile.h"`

include uno header scritto da noi
occorre indicare dove reperirlo



HEADER FILE

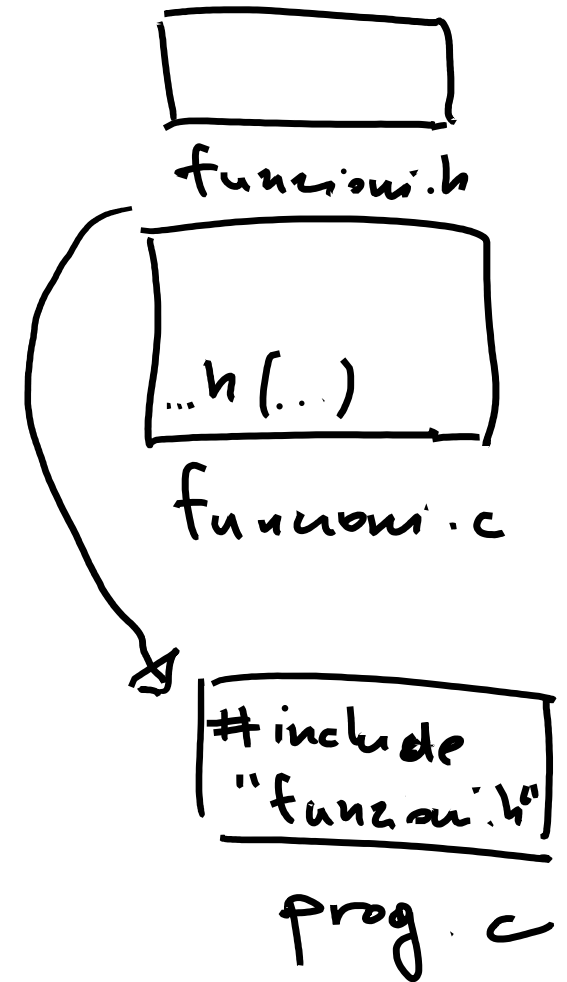
Il *file di intestazione (header)*

- ha **estensione .h**
- ha (per convenzione) **nome uguale al file .c** di cui fornisce le dichiarazioni

Quindi c'è un file .h per ogni file .c dell'applicazione (escluso, eventualmente, il file che contiene il main())

Ad esempio:

- se la funzione **f** è definita nel file **f2c.c**
- il corrispondente header file, che i clienti potranno includere per usare la funzione **f**, dovrebbe chiamarsi **f2c.h**



ESEMPIO

Conversione °F / °C

I^a versione: singolo file

```
float fahrToCelsius(float f)
{ return 5.0/9 * (f-32);
}

main()
{ float c;
  c = fahrToCelsius(86);
}
```

ESEMPIO

Vogliamo suddividere cliente e servitore *su due file separati*

File `main.c` (*cliente*)

```
float fahrToCelsius(float); DICHIARAZIONE
main() { float c;
        c = fahrToCelsius(86); }
```

← *SORGETTE*
File `f2c.c` (*servitore*)

```
float fahrToCelsius(float f) DEFINIZIONE
{ return 5.0/9 * (f-32);
}
```


ESEMPIO

- Per includere automaticamente la dichiarazione occorre **introdurre un file header**

File main.c (cliente)

```
#include "f2c.h"
main() { float c;
        c = fahrToCelsius(86); }
```

gcc main.c

ERRORE

NON SUFFICIENTE
PER LINKING

NON È STENSIONE
File f2c.h (header)

float fahrToCelsius(float); DICHIARAZIONE

RIASSUMENDO

Convenzione:

- se un componente è definito in **xyz.c**
- *il file header* che lo dichiara, che i clienti dovranno includere, *si chiama* **xyz.h**

File main.c (cliente)

```
#include "f2c.h"
main() { float c = fahrToCelsius(86); }
```

File f2c.h (header)

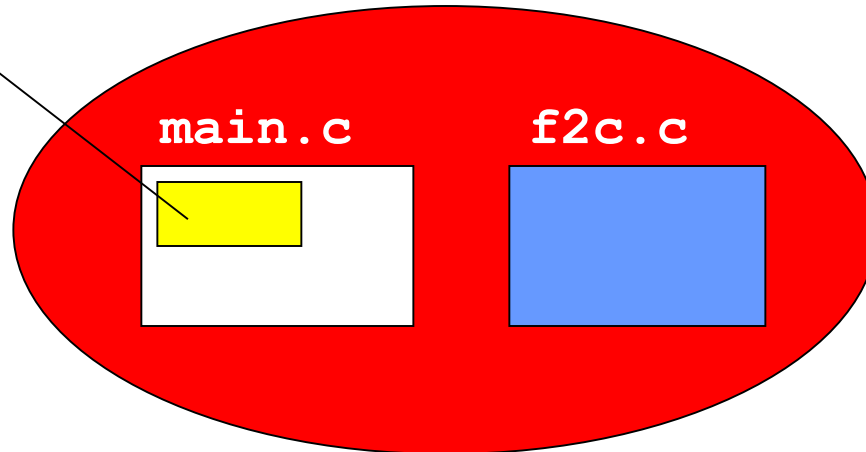
```
float fahrToCelsius(float);
```

ESEMPIO

Struttura finale dell'applicazione:

- un main definito in `main.c`
 - una funzione definita in `f2c.c`
 - *un file header `f2c.h` incluso da `main.c`*
- } *Progetto*

header
`f2c.h`



COLLEGAMENTO DI UN'APPLICAZIONE

LIBRERIE DI SISTEMA:

insieme di componenti software che consentono di interfacciarsi col sistema operativo, usare le risorse da esso gestite, e realizzare alcune "istruzioni complesse" del linguaggio

COSTRUZIONE “MANUALE”

La costruzione si può fare “*a mano*”,
attivando *compilatore* e *linker* dalla
linea di comando del sistema operativo
(DOS, Unix, ...)

C:\PROVA> gcc -c f1.c

(genera f1.obj)

C:\PROVA> ld -o prog.exe f1.obj -lc -lm

(genera prog.exe)

**Eseguibile
da produrre**

File oggetto

Libreria C

AMBIENTI INTEGRATI

VISUAL STUDIO

make

Oggi, gli ambienti di lavoro integrati

automatizzano la procedura: COSTRUZIONE
BUILDING

- compilano i file sorgente (se e quando necessario)
- invocano il linker per costruire l'eseguibile

ma per farlo devono sapere:

- quali file sorgente costituiscono l'applicazione
- il nome dell'eseguibile da produrre.

p1.c
 p2.c
 ...
 pn.c
 prog

} PROGETTO

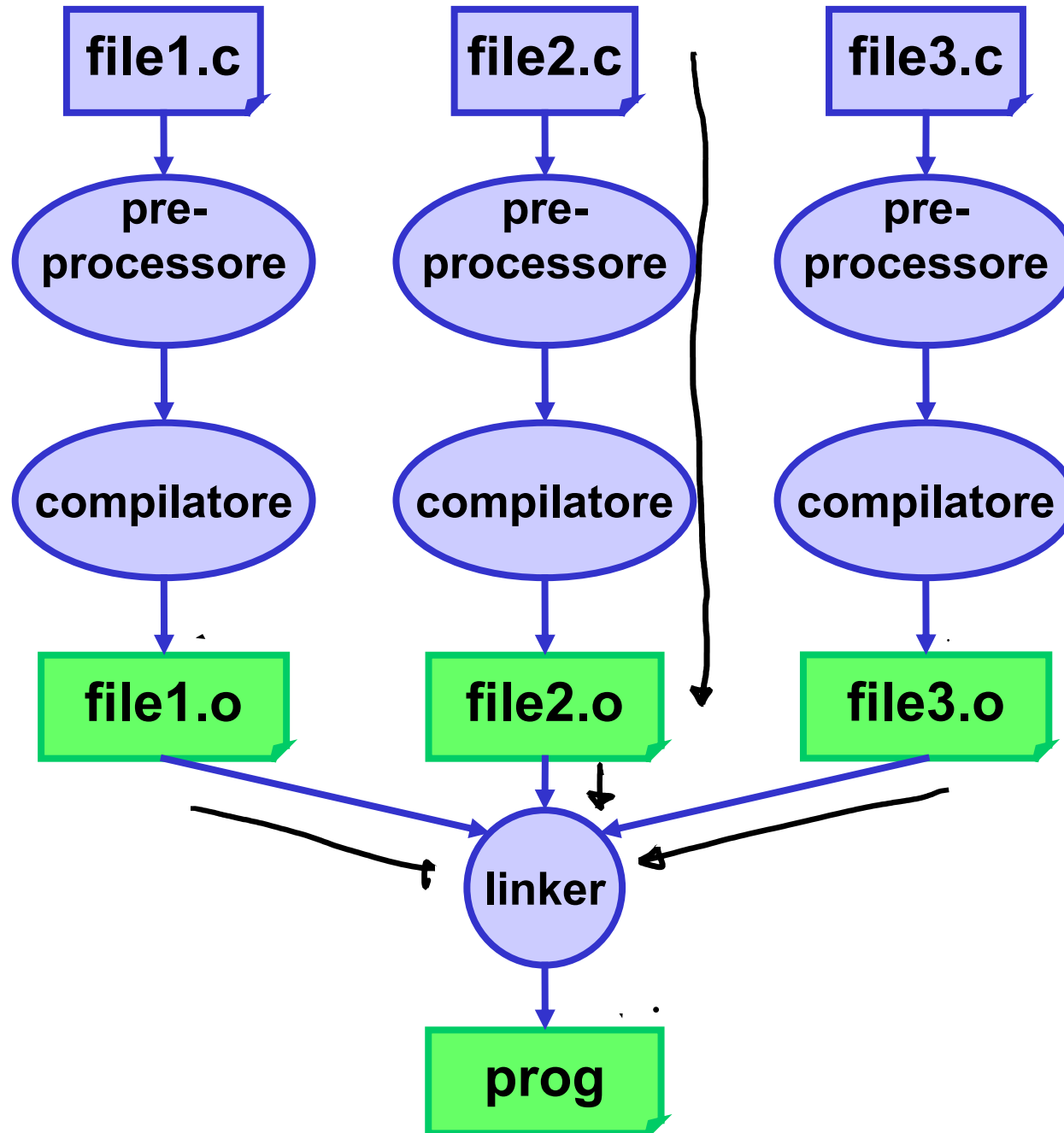
PROGETTI

È da queste esigenze che nasce il concetto di PROGETTO usati dallo strumento di building

- un contenitore concettuale (e fisico)
- che elenca i file sorgente in cui l'applicazione è strutturata
- ed eventualmente altre informazioni utili.

Viene utilizzato anche dal linker, per sapere quali file vanno collegati per creare l'eseguibile.

prog.prj



ERRORE TIPICO

**... e se io includessi il file .c invece del
file header?**

h

main.c

```
#include "file1.c"
#include "file2.c"
main()
{ p(1); q(2); }
```

file1.c

```
#include "file3.c"
void p(int i) .
{ printf("%d",f(i));
}
```

file2.c

```
#include "file3.c"
void q(char i)
{ printf("%c",f(i));
}
```

file3.c

```
char f(char x)
{ return x+1; }
```

ERRORE TIPICO

Il programma risultante dopo il pre-processing:

error C2084:
function 'char __cdecl f(char)'
already has a body

La funzione f risulta definita 2 volte

main.c

```
char f(char x)
{ return x+1; }
```

```
void p(int i)
{ printf("%d",f(i));
}
```

```
char f(char x)
{ return x+1; }
```

```
void q(char i)
{ printf("%c",f(i));
}
```

```
main()
{ p(1); q(2); }
```

ERRORE TIPICO 2

... e se io mettessi le definizioni delle funzioni nei file header?

main.c

```
#include "file1.h"
#include "file2.h"
main()
{ p(1); q(2); }
```

file1.h

```
#include "file3.h"
void p(int i)
{ printf("%d",f(i));
}
```

file2.h

```
#include "file3.h"
void q(char i)
{ printf("%c",f(i));
}
```

file3.h

```
char f(char x)
{ return x+1; }
```

ho esattamente lo stesso problema

versione corretta

file1.h

```
void p(int i);
```

file2.h

```
void q(char);
```

file3.h

```
char f(char x);
```

main.c

```
#include "file1.h"
#include "file2.h"
main()
{ p(1); q(2); }
```

file1.c

```
#include<stdio.h>
#include "file3.h"
void p(int i)
{ printf("%d",f(i));}
```

file2.c

```
#include<stdio.h>
#include "file3.h"
void q(char i)
{ printf("%c",f(i));}
```

file3.c

```
char f(char x)
{ return x+1; }
```