

# STRUTTURA DI UN PROGRAMMA

File prova1.c

*Area globale*

```
#include <stdio.h>
```

Direttive

```
...
```

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

Dichiarazioni globali e  
prototipi di funzioni

```
int g(int x) {
```

Definizioni di funzioni

```
... /*ambiente locale a g*/ }
```

```
int main() {  
... }
```

```
int f(int x) {
```

Definizioni di funzioni

```
... /*ambiente locale a f*/ }
```

# STRUTTURA DI UN PROGRAMMA

---

- Il main è l'unica parte obbligatoria
- Le direttive sono gestite dal preprocessore
- Le variabili globali sono visibili in tutti gli ambienti del programma
- Esistono delle regole di visibilità per gli identificatori (nomi di variabili, di funzioni, costanti) che definiscono in quali parti del programma tali identificatori possono essere usati

# AMBIENTI

---

- In un programma esistono diversi ambienti:
  - area globale
  - il main
  - ogni singola funzione
  - ogni blocco

# REGOLE DI VISIBILITÀ

---

- *Un identificatore non è visibile prima della sua dichiarazione*
- *Un identificatore definito in un ambiente è visibile in tutti gli ambienti in esso contenuti*
- *Se in un ambiente sono visibili due definizioni dello stesso identificatore, la definizione valida è quella dell'ambiente più vicino al punto di utilizzo*
- *In ambienti diversi si può definire lo stesso identificatore per denotare due oggetti diversi*
- *In ciascun ambiente un identificatore può essere definito una sola volta*

# REGOLE DI VISIBILITÀ

---

- *Un identificatore non è visibile prima della sua dichiarazione*

SCORRETTO

```
int main() {  
int x = y*2;  
int y = 5;  
...  
return 0; }
```

CORRETTO

```
int main() {  
int y = 5;  
int x = y*2;  
...  
return 0; }
```

# REGOLE DI VISIBILITÀ

---

- Se *in un ambiente* sono visibili due dichiarazioni dello stesso identificatore, la dichiarazione valida è quella dell'ambiente più vicino al punto di utilizzo
- In ambienti diversi si può dichiarare lo stesso identificatore per denotare due oggetti diversi

```
float x = 3.5;
int main() {
    int y, x = 5;
    y = x; /* y vale 5 */
    ...
    return 0;
}
```

# REGOLE DI VISIBILITÀ

---

- In ciascun ambiente un identificatore può essere dichiarato una sola volta

```
int main() {  
    float x = 3.5;  
    char x;  SCORRETTO  
    ...  
    return 0;  
}
```

# REGOLE DI VISIBILITÀ

---

- Un identificatore dichiarato in un ambiente è visibile in tutti gli ambienti in esso contenuti

SCORRETTO

```
int main() {  
    int x;  
    {  
        int y = 5;  
    }  
    x = y;  
    ...  
    return 0; }
```

CORRETTO

```
int main() {  
    int x;  
    {  
        int y = 5;  
        x = y;  
    }  
    ...  
    return 0; }
```

# FUNZIONI COME COMPONENTI SOFTWARE

---

- Una funzione è un *componente software (servitore) riutilizzabile*
- che costituisce una unità di traduzione:
  - può essere definita in un unico file e compilata per proprio conto
  - pronta per essere usata da chiunque

# FUNZIONI COME COMPONENTI SOFTWARE

---

Per usare tale componente software,  
il cliente:

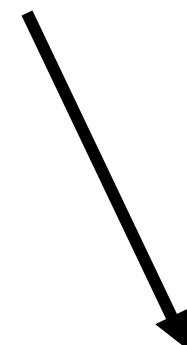
- non ha bisogno di sapere *come è fatto* (cioè, di conoscerne la *definizione*)
- *deve conoscerne solo l'interfaccia:*
  - nome
  - numero e tipo dei parametri
  - tipo del risultato

# DICHIARAZIONE DI FUNZIONE

---

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

<dichiarazione-di-funzione> ::=  
<tipoValore> <*nome*>(<parametri>) ;



# DICHIARAZIONE DI FUNZIONE

---

```
int max(int a, int b)
{ if (a>b) return a;
  else return b;
}
```

DEFINIZIONE

```
int max(int a, int b);
```

DICHIARAZIONE  
o *prototipo*  
o *interfaccia*

# 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*

# DICHIARAZIONE DI FUNZIONI

---

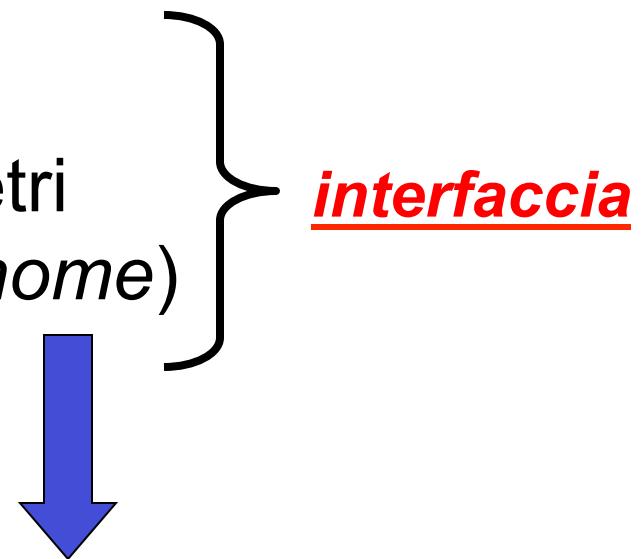
- La definizione di una funzione costituisce l'effettiva realizzazione del componente
  - Dice come e' fatto il componente
- La dichiarazione specifica il contratto di servizio fra cliente e servitore, esprimendo le **proprietà essenziali** della funzione.
  - Dice come si usa il componente
  - Per usare una funzione non e' necessario sapere come e' fatta, anzi e' controproducente

# DICHIARAZIONE DI FUNZIONI

---

- La dichiarazione specifica:

- il **nome** della funzione
- numero e tipo dei parametri  
(non necessariamente *il nome*)
- il **tipo del risultato**



Il nome avrebbe significato solo  
*nell'environment della funzione,*  
che qui non c'è!

# DICHIARAZIONE vs. DEFINIZIONE

---

- 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

---

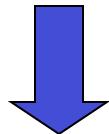
- La dichiarazione di una funzione costituisce **solo una specifica** delle proprietà del componente:
  - Può essere duplicata senza danni
  - Un'applicazione può contenerne più di una
  - La compilazione di una dichiarazione non genera codice macchina

# DICHIARAZIONE vs. DEFINIZIONE

---

- La definizione è *molto più* di una dichiarazione

definizione = dichiarazione + corpo



La definizione funge anche da dichiarazione  
(ma non viceversa)

# FUNZIONI E FILE

---

- Un programma C è, in prima battuta, una collezione di funzioni
  - una delle quali è il *main*
- Il testo del programma deve essere scritto in uno o più *file di testo*
  - il file è un concetto *del sistema operativo*,  
non del linguaggio C

*Quali regole osservare ?*

# FUNZIONI E FILE

---

- Il *main* può essere scritto dove si vuole nel file
  - viene chiamato dal sistema operativo, il quale sa come identificarlo
- Una funzione, invece, deve rispettare una *regola fondamentale di visibilità*
  - prima che qualcuno possa *chiamarla*, la funzione deve essere stata dichiarata
  - altrimenti, si ha errore di compilazione.

# ESEMPIO SCORRETTO (SINGOLO FILE)

File prova1.c

```
int main() {  
    float y = fahrToCelsius(86);  
    return 0; }
```

NOTA: all'atto della chiamata  
la funzione non e' ancora stata  
definita

ERRORE DI COMPILAZIONE

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

# ESEMPIO CORRETTO (SINGOLO FILE)

File prova1.c

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

Prima definisco  
**fahrToCelsius** poi la  
uso

```
int main() {  
    float y = fahrToCelsius(86);  
    return 0; }
```

# ESEMPIO CORRETTO (SINGOLO FILE)

File prova1.c

```
float fahrToCelsius(float) ;  
  
int main() {  
    float y = fahrToCelsius(86) ;  
    return 0 ;}  
  
float fahrToCelsius(float f) {  
    return 5.0/9 * (f-32) ;  
}
```

OPPURE prima dichiaro la funzione  
tramite un **PROTOTIPO** poi la uso e  
dopo la definisco

# ALTRO ESEMPIO SCORRETTO

File prova1.c

```
int f(int x) {
    if (x > 0) return g(x);
    else return x;
}

int g(int x) {
    return f(x-2);
}

int main() {
    int y = f(3);
    return 0;
}
```

ATTENZIONE:  
**f** chiama **g** e **g** chiama **f**  
Quale definisco prima ?

# UTILITA' DEI PROTOTIPI

## File prova1.c

```
int g(int); /* prototipo */
int f(int x) {
    if (x > 0) return g(x);
    else return x;
}
int g(int x) {
    return f(x-2);
}

int main() {
    int y = f(3);
    return 0;
}
```

# FUNZIONI E FILE

---

- *Regola fondamentale di visibilità*
  - prima che qualcuno possa *chiamarla*, la funzione deve essere stata dichiarata
  - altrimenti, si ha errore di compilazione.
- Caso particolare: se la definizione funge anche da dichiarazione, la regola è rispettata se la definizione appare prima della chiamata

# ESEMPIO CORRETTO (SINGOLO FILE)

File prova1.c

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

Prima definisco  
**fahrToCelsius**  
poi la uso

```
int main() {  
    float y = fahrToCelsius(85);  
    return 0; }
```

# PROGETTI SU PIU' FILE

---

- Una applicazione complessa non può essere sviluppata *in un unico file*: sarebbe *ingestibile!*
- *Deve necessariamente essere strutturata su più file sorgente*
  - compilabili separatamente
  - da fondere poi *insieme* per costruire l'applicazione.

# PROGETTI STRUTTURATI SU PIU' FILE

---

- Per strutturare un'applicazione su più file, sorgente, occorre che *ogni file possa essere compilato separatamente dagli altri*
  - Poi, i singoli componenti così ottenuti saranno *legati (dal linker)* per costruire l'applicazione.
- Affinché un file possa essere compilato singolarmente, *tutte le funzioni usate devono essere dichiarate prima dell'uso*
  - non necessariamente definite!

# ESEMPIO SU DUE FILE

File `main.c`

```
float fahrToCelsius(float f); → Dichiarazione  
della funzione  
  
int main() {  
    float y = fahrToCelsius(40);  
    return 0; }  
↓  
Chiamata della funzione
```

File `fahr.c`

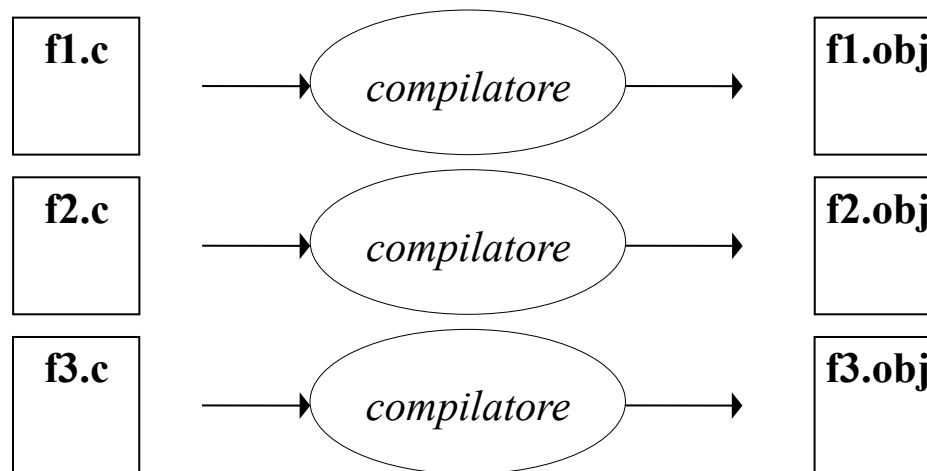
```
Definizione della funzione  
  
float fahrToCelsius(float f) {  
    return 5.0/9 * (f-32);  
}
```

# COMPILAZIONE DI UN'APPLICAZIONE

---

## 1) Compilare i singoli file che costituiscono l'applicazione

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

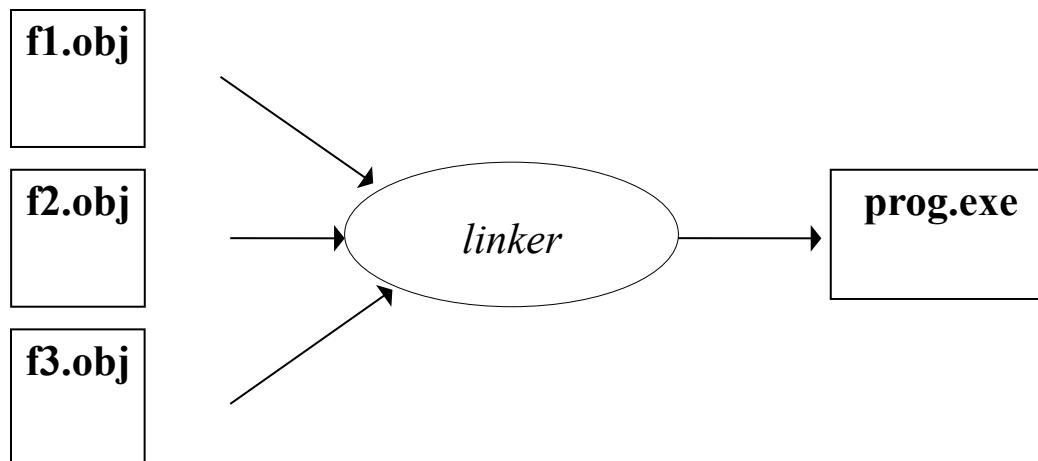


# COMPILAZIONE DI UN'APPLICAZIONE

---

## 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**



# COMPILAZIONE DI UN'APPLICAZIONE

---

Perché la costruzione vada a buon fine:

- ogni funzione deve essere definita una e una sola volta in uno e uno solo dei file sorgente
  - se la definizione manca, si ha errore di linking
- ogni cliente che *usa* una funzione deve incorporare la dichiarazione opportuna
  - se la dichiarazione manca, si ha errore di compilazione nel file del cliente (...forse...!!)

# 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**.

# HEADER FILE

---

## Il *file di intestazione (header)*

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

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

1<sup>a</sup> versione: singolo file

```
float fahrToCelsius(float f) {  
    return 5.0/9 * (f-32);  
}  
  
int main() {  
    float c;  
    c = fahrToCelsius(86);  
    return 0; }
```

# ESEMPIO

---

Vogliamo suddividere cliente e servitore su  
*due file separati*

File `main.c` (*cliente*)

```
float fahrToCelsius(float);
int main() { float c;
             c = fahrToCelsius(86);
             return 0; }
```

File `f2c.c` (*servitore*)

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

# ESEMPIO

---

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

File `main.c` (*cliente*)

```
#include "f2c.h"
int main() { float c;
              c = fahrToCelsius(86);
              return 0; }
```



File `f2c.h` (*header*)

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

# 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"
int main() { float c = fahrToCelsius(86);
              return 0; }
```

File **f2c.h** (*header*)

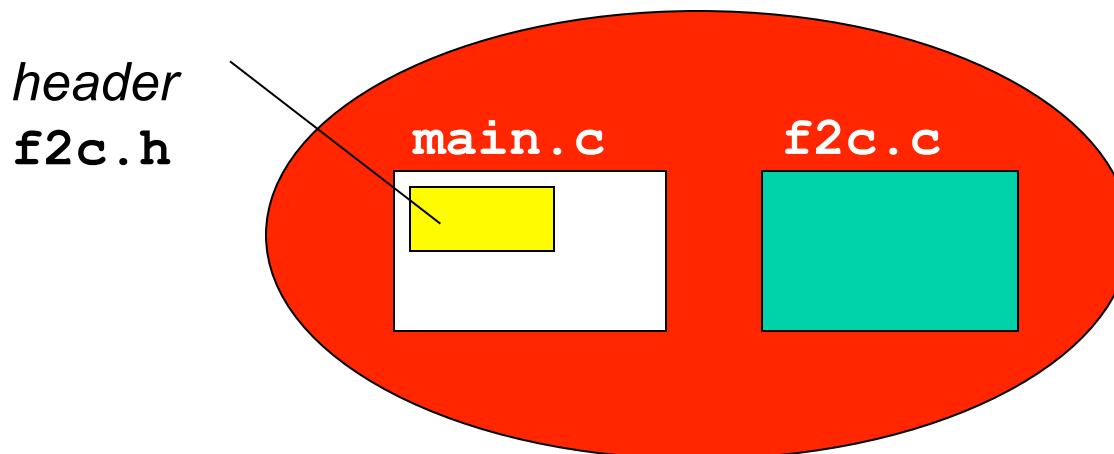
```
float fahrToCelsius(float);
```

# 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**



# FILE HEADER

---

- Due formati:

**#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**  
*(attenzione al formato dei percorsi..!!)*