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){
.../*ambiente locale a g*/}
```
Definizioni di funzioni

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

```
int f(int x){
.../*ambiente locale a f*/}
```
Definizioni di funzioni

REGOLE DI VISIBILITA'

- *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*

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

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.

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);
}

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

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);

void main() {
  float y = fahrToCelsius(40);
}
```

Dichiarazione della funzione

Chiamata della funzione

File **`fahr.c`**

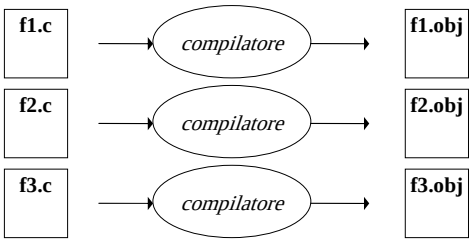
Definizione della funzione

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

COMPILAZIONE DI UN'APPLICAZIONE

1) Compilare i singoli file che costituiscono l'applicazione

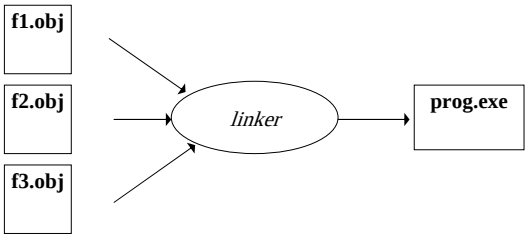
- File *sorgente*: estensione `.c`
- File *oggetto*: estensione `.o` o `.obj`



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

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

Iª versione: singolo file

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

ESEMPIO

Vogliamo suddividere cliente e servitore *su due file separati*

File **`main.c`** *(cliente)*

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

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

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

File `f2c.h` *(header)*

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

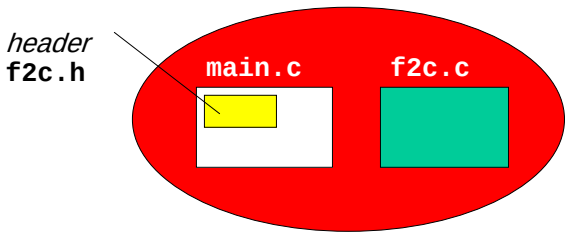
ESEMPIO

Struttura finale dell'applicazione:

- un main definito in **`main.c`**
- una funzione definita in **`f2c.c`**

} *Progetto*

- *un file header* **`f2c.h`** incluso da **`main.c`**



FILE HEADER

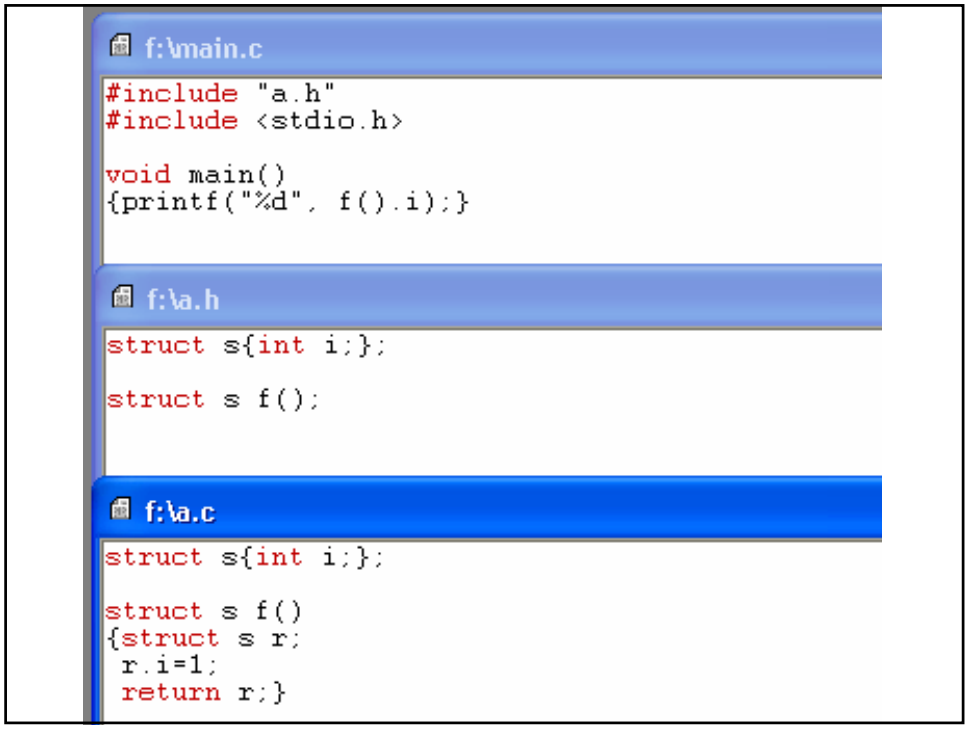
- **Due formati:**

  `#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..!!)


f:\main.c
#include "a.h"
#include <stdio.h>
void main()
{printf("%d", f().i);}
f:\a.h
struct s{int i;};
struct s f();
f:\a.c
struct s{int i;};
struct s f()
{struct s r;
r.i=1;
return r;}