PROGETTI STRUTTURATI SU PIÙ 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.

FUNZIONI & 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 & 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 (su singolo file)

File `prova1.c`

```
int fact(int);

main() {
  int y = fact(3);
}

int fact(int n) {
  return (n<=1) ? 1 : n*fact(n-1);
}
```

Dichiarazione: deve precedere l'uso

Uso (chiamata)

Definizione

PROGETTI STRUTTURATI SU PIÙ 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 *fusi insieme* per costruire l'applicazione.
- Affinché un file possa essere compilato singolarmente, *tutte le funzioni usate devono essere dichiarate prima dell'uso*
  - non necessariamente definite!

DALL'ESEMPIO SU UN SOLO FILE...

File `prova1.c`

```
int fact(int);

main() {
  int y = fact(3);
}

int fact(int n) {
  return (n<=1) ? 1 : n*fact(n-1);
}
```

... ALL'ESEMPIO SU DUE FILE

File `main.c`

```
int fact(int);

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

Dichiarazione della funzione

Uso (chiamata)

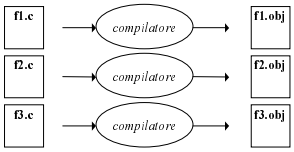
File `fact.c`

```
int fact(int n) {
  return (n<=1) ? 1 : n*fact(n-1);
}
```

Definizione della funzione

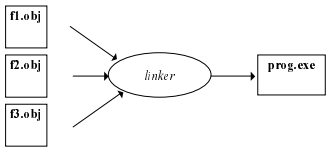
COMPILAZIONE DI UN'APPLICAZIONE

1) Compilare i singoli file che costituiscono l'applicazione
   - File *sorgente*: estensione `.c`
   - File *oggetto*: estensione `.o` o `.obj`



COLLEGAMENTO 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



COSTRUZIONE 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 *usi* una funzione deve incorporare la dichiarazione opportuna
  - se la dichiarazione manca, si ha errore di *compilazione* nel file del cliente (..forse...!!)

IL RUOLO DEL LINKER

Perché, esattamente, serve il linker?

- Il compilatore deve "*lasciare in bianco*" i riferimenti alle chiamate di funzione che non sono definite nel medesimo file
- Compito del linker è *risolvere* tali riferimenti, riempiendo gli "spazi bianchi" con l'indirizzo effettivo del codice della funzione.

COSTRUZIONE "MANUALE"

- Attivare il compilatore *su ogni singolo file sorgente*

```
C:\TMP> gcc -c fact.c
C:\TMP> gcc -c main.c
```

- Attivare il linker per unire i rispettivi file oggetto e le librerie di sistema

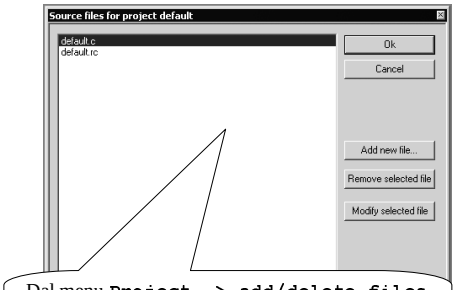
```
C:\TMP> ld -o prog.exe fact.obj main.obj -lc
```

- ... un lavoraccio!

COSTRUZIONE NEGLI AMBIENTI INTEGRATI

- Negli ambienti integrati, tutto ciò viene automatizzato
- Si predispone un progetto che contenga *tutti i file sorgente (.c) necessari*
- Si costruisce l'applicazione normalmente (Make / F9)

PROGETTI SU PIÙ FILE IN Icc



Dal menu `Project -> add/delete files` si selezionano **tutti i file sorgente (.c)** da inserire nel progetto.

GESTIRE PROGETTI COMPLESSI

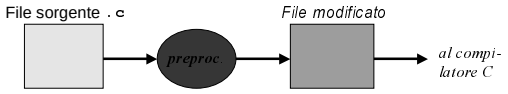
- Perché queste architetture funzionino, ogni cliente deve contenere le dichiarazioni di *tutte* le funzioni che usa
- In una applicazione complessa, fatta di decine di file, non è pensabile che questo venga fatto a mano, mediante *copia & incolla* "file per file":

OCCORRE UN AUTOMATISMO

IL PRE-PROCESSORE C

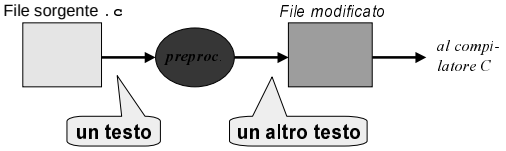
- Quando si compila un programma C, il compilatore *non riceve esattamente il testo del programma da noi fornito*
- riceve una versione "*riveduta e corretta*" da "qualcuno" che si interpone tra noi e il compilatore vero e proprio:

il PRE-PROCESSORE C



IL PRE-PROCESSORE C

- Il pre-processore *modifica il testo del programma prima* che esso raggiunga il compilatore C vero e proprio.
- Così, può svolgere alcune utili funzioni di *manipolazione del testo* al nostro posto



IL PRE-PROCESSORE C

Il pre-processore *non* è un compilatore C

- non conosce il linguaggio C
- non può interpretare le istruzioni C, né controllarne la correttezza
- non sa cosa fa: è solo un automa che *agisce sul testo del programma*
  - potrebbe manipolare *qualunque testo*, non solo programmi C
  - programmi Pascal, poesie, lettere commerciali, lettere d'amore...

IL PRE-PROCESSORE C

Cosa può fare?

- includere altre porzioni di testo, prese da altri file
- effettuare *ricerche e sostituzioni* (più o meno sofisticate) sul testo
- *inserire o sopprimere parti del testo* a seconda del verificarsi di certe condizioni da noi specificate.

IL PRE-PROCESSORE C

Come si controlla il suo funzionamento?

- mediante ***direttive*** inserite nel testo.

Attenzione: le direttive *non sono istruzioni C*

- non ne hanno neanche la sintassi!
- *infatti, non sono destinate al compilatore*, che non le vedrà mai
- vengono soppresse dal pre-processore dopo essere state da esso interpretate.

DIRETTIVE AL PRE-PROCESSORE C

Principali direttive

- includere altre porzioni di testo
  `#include` *nomefile*
- effettuare *ricerche e sostituzioni*
  `#define` *testo1 testo2*
- *inserire o sopprimere parti del testo*

```
#ifdef cond          #ifndef cond
...testo...           ...testo...
#endif               #endif
```

LA DIRETTIVA #include

Sintassi:

`#include` *<libreria.h>*
`#include` *"miofile.h"*

Effetto:

include il contenuto del file specificato *esattamente nella posizione* in cui si trova la direttiva stessa.

(La differenza tra le due scritture sopra verrà discussa più avanti)

PREPROCESSORE: ESEMPIO

Supponendo di avere questi due file:

File `main.c`

```
#include "extra"
main() { .... }
```

File `extra`

```
int fact(int);
```

PREPROCESSORE: ESEMPIO

...dopo il pre-processing la situazione sarà:

File `main.c` ***modificato dal pre-processore***

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

NB: dopo che il pre-processing è avvenuto, **il file extra non serve più.**

SE SIETE CURIOSI...

...il pre-processing si può vedere:

```
C:\TMP> gcc -E main.c
```

- **-E** effettua solo il pre-processing

```
C:\TMP> gcc -C -P -E main.c
```

- **-P** non numera le righe (che di solito vengono numerate)
- **-C** non toglie i commenti (che di solito vengono tolti)

FILE HEADER

Per automatizzare la gestione delle dichiarazioni, si introduce il concetto di *header file (file di intestazione)*

- **scopo: evitare ai clienti di dover trascrivere riga per riga le dichiarazioni necessarie**

FILE HEADER

In pratica:

- **il progettista di un componente software (un file `.c`) predispone un header file contenente *tutte le dichiarazioni* relative alle funzioni definite nel componente software medesimo**
- **così facendo, i clienti *non dovranno più ricopiarsi a mano le dichiarazioni: basterà includere l'header file* tramite una direttiva `#include`.**

FILE HEADER

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

Il problema della conversione °F / °C

Iª versione: singolo file

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

ESEMPIO

Vogliamo suddividere cliente e servitore *su due file separati*

File `main.c` *(cliente)*

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

File `f2c.c` *(servitore)*

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

ESEMPIO

Perché la dichiarazione sia *inclusa automaticamente*, occorre introdurre un *file header*

File `main.c` *(cliente)*

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

File `f2c.h` *(header)*

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

RIASSUNTO

Convenzione:

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

File `main.c` *(cliente)*

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

File `f2c.h` *(header)*

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

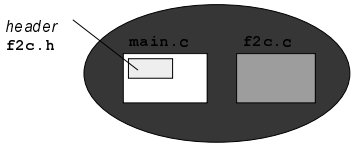
ESEMPIO

Struttura finale dell'applicazione:

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

} ***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..!!)

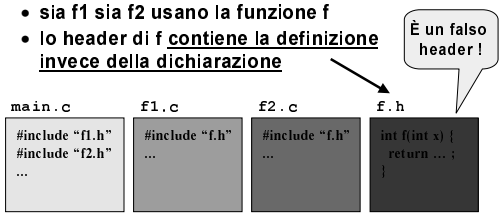
FILE HEADER - CAUTELE D'USO

- **ATTENZIONE!! Un file header deve contenere *solo dichiarazioni* !**
- **Se contiene anche solo *una* definizione possono crearsi situazioni di *errore* (rischio di *definizioni duplicate*).**

FILE HEADER - CAUTELE D'USO

Esempio

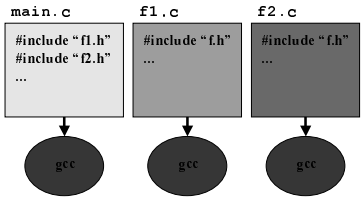
- **un main usa le funzioni f1 e f2**
- **sia f1 sia f2 usano la funzione f**
- **lo header di f contiene la definizione invece della dichiarazione**



FILE HEADER - CAUTELE D'USO

La compilazione fila liscia:

- f1.c e f2.c si compilano senza problemi
  Ma attenzione !! - *ognuno include una definizione di f*
- il main si compila senza problemi...



FILE HEADER - CAUTELE D'USO

Ma il linker dà errore in fase di collegamento

- Infatti, *la definizione di f risulta due volte*
- il relativo codice è duplicato!!