# Approfondimento : la compilazione separata

- Funzioni e procedure sono un elemento importante della programmazione
  - permettono l'astrazione
  - evitano la duplicazione del codice
  - riducono i tempi di debug e manutenzione
- Per motivi sia pratici che teorici, è spesso cosigliabile avere le funzioni in librerie distinte dal codice che le usa
- Queste librerie vanno compilate separatamente

# Le funzioni

- Una funzione realizza concretamente (implementa) un'operazione astratta:
    - il programmatore deve avere ben chiaro quale sia questa operazione (ad esempio, la somma di due interi);
    - quindi scrive il codice che esegue quell'operazione e gli assegna un nome significativo (as esempio somma).
- Da lì in poi, quell'operazione può essere usata senza preoccuparsi di come viene eseguita

# Le funzioni

- Una funzione è caratterizzata da
    - un nome, che deve essere unico
    - un tipo di ritorno
    - una lista di argomenti
    - un corpo

```
int somma (int addendo1, int addendo2)
{
  return addendo1 + addendo2;
}
```

# Esempio di implementazioni diverse

- Operazione astratta: prodotto di due interi.
- Implementazione 1:

```
int prodotto (int moltiplicando, int moltiplicatore )
{
  return moltiplicando * moltiplicatore;
}
```

- Implementazione 2:

```
int prodotto (int moltiplicando, int moltiplicatore)
{
  int i, res=0;
  for (i=0; i<moltiplicatore; i++)
   { res += moltiplicando; }
  return res;
}
```

# Esempio di implementazioni diverse

- Implementazione 3:

```
int prodotto (int moltiplicando, int moltiplicatore)
{
  int i, res=0;
  for (i=0; i<moltiplicatore; i++)
   { res = somma(res, moltiplicando); }
  return res;
}
```

- Dal punto di vista dell'utente (ossia di chi usa la funzione prodotto), quale implementazione viene usata è trasparente (a meno di considerazioni di efficienza).

# Librerie di funzioni

- Talvolta le funzioni che scriviamo sono abbastanza “generiche” da essere usate in programmi diversi (ossia non sono funzioni ad-hoc per uno specifico programma).
- In questo caso, vogliamo evitare di avere una copia della funzione in ogni programma, così che una correzione avvantaggi tutti i programmi che usano quella funzione

# Librerie di funzioni

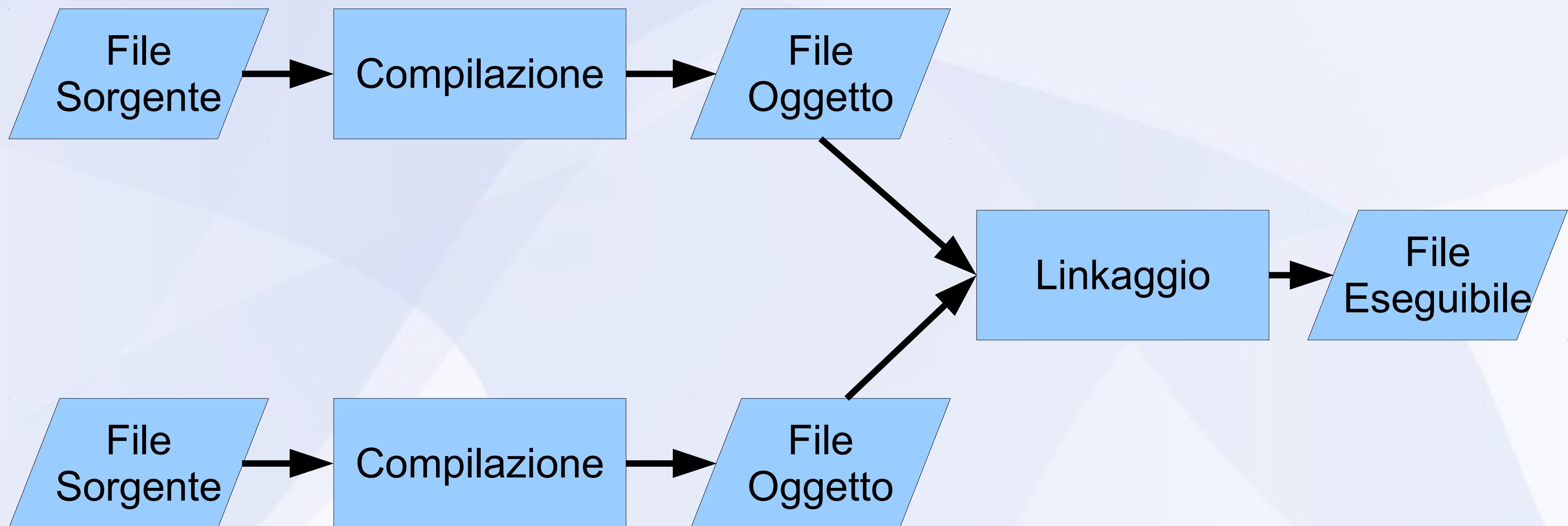
- Per farlo, mettiamo le nostre funzioni in file C separati, detti quindi librerie di funzioni.
- Questi file C devono essere compilati separatamente da ogni programma che utilizzi le funzioni che vi vengono definite.
- La compilazione come l'abbiamo imparata a conoscere finora deve quindi essere scissa in un certo numero di operazioni più semplici (che finora erano raggruppate automaticamente in un'unico comando)

# Compilazione separata

- La compilazione è infatti formata da (almeno) due fasi diverse:
  la compilazione vera e propria e il linkaggio.
- Il risultato della compilazione è un file oggetto, binario ma non ancora eseguibile
- Il linkaggio è l'operazione che “aggangia” a quel file oggetto le librerie di sistema che lo rendono eseguibile sul pc

# Compilazione separata

- Inoltre durante il linkaggio tutto il codice eventualmente compilato separatamente nei vari file oggetto viene composto in un unico eseguibile:



# Librerie di funzioni

- In questo modo, le librerie (di sistema, personali o di terze parti) possono essere compilate una volta sola e quindi linkate di volta in volta in ogni nuovo eseguibile.

- Ma per poter compilare il main, il compilatore deve conoscere ALMENO i prototipi delle funzioni della libreria:
  - utilizziamo un file H (detto file header) che contiene i prototipi delle funzioni.

# Librerie di funzioni

- Abbiamo quindi almeno tre file:
  - il file sorgente somma.c, che contiene la nostra libreria col sorgente da compilare delle nostre funzioni;
  - il file header somma.h, che contiene solamente i prototipi delle funzioni, e che dovrà venir incluso da ogni programma che le voglia usare
  - E infine il nostro usuale file sorgente con il main, che dovrà includere il file header somma.h

# Librerie di funzioni

- File: somma.c

```
#include "somma.h"
int somma (int addendo1, int addendo2)
{
  return addendo1 + addendo2;
}
```

- File: somma.h

```
int somma (int addendo1, int addendo2);
```

- File: main.c

```
#include "somma.h"

main() {
 int a, b, c;
 scanf("%d %d", &a, &b);
 c = somma(a,b);
 printf("%d\n", c);
}
```

# Compilazione separata

- In questi casi, però, il vecchio modo comodo e facile di compilare con **`make nomeprogramma`** con il makefile sempre uguale non funziona più.
- Ci vuole un nuovo **`makefile`** che mostri a make quali sono i prerequisiti per compilare ogni programma, ossia quale librerie devono essere compilate separatamente.
- Purtroppo, questo **`makefile`** dovrà essere modificato di volta in volta manualmente per ogni nuovo programma.

# Compilazione: il nuovo makefile

- Il makefile per il nostro esempio

```
# makefile
CC=gcc
CFLAGS=-Wall -Wno-return-type -Wno-implicit-int -g -O
-pedantic -Wformat=2 -Wextra
LDLIBS=-lm

main: main.o somma.o

somma.o: somma.h
```

- Il suo utilizzo

```
[rama]:olivia [~/test] -> make main
gcc -Wall -Wno-return-type -Wno-implicit-int -g -O -pedantic
-Wformat=2 -Wextra    -c -o main.o main.c
gcc -Wall -Wno-return-type -Wno-implicit-int -g -O -pedantic
-Wformat=2 -Wextra    -c -o somma.o somma.c
gcc   main.o somma.o  -lm -o main
```

# Il nuovo makefile

Come detto prima, il makefile deve essere modificato per ogni programma, aggiungendo i nuovi "**target**" di compilazione:

```
main: main.o somma.o

somma.o: somma.h
```

- sulla sinistra, prima dei **:**, c'è il nome del target
- sulla destra, dopo i **:**, ci sono i prerequisiti
- notate che se un prerequisito è anche un target, make si preoccupa di creare il prerequisito prima di completare l'operazione principale

# Il nuovo makefile

- Un target è il risultato finale della compilazione
- Un target viene specificato indicandolo a linea di comando a make
  - `make miotarget`
- make cerca nel makefile istruzioni su come compilare il target indicato, in particolare i prerequisiti ossia file necessari per compilare il target attuale
  - In precedenza tali istruzioni erano implicite mentre ora vanno indicate esplicitamente

# Il nuovo makefile

```
main: main.o somma.o

somma.o: somma.h
```

- Accanto ad ogni target viene specificato l'elenco dei file che devono esistere per poter compilare il target e che, se è il caso, vengono aggiunti alla chiamata al compilatore
- make compila il target solo se i prerequisiti (od i sorgenti) sono più nuovi dell'eventuale target già compilato in precedenza

# Il nuovo makefile

```
main: main.o somma.o

somma.o: somma.h
```

- make opera ricorsivamente: poiché **`somma.o`** è (anche) un target, oltre che un prerequisito di main, nell'eseguire **`make main`** il make esegue un implicito **`make somma.o`** ossia controlla i sorgenti di **`somma.o`** e i suoi prerequisiti e nel caso non sia aggiornato, lo ricompila (anche se non gli è stato esplicitamente indicato), prima di procedere con la compilazione di main.

# Il nuovo makefile

```
main: main.o somma.o

somma.o: somma.h
```

- somma.h è un prerequisito particolare: non richiede compilazione.
- Perchè metterlo, allora?
    - In effetti potremmo non metterlo e andrebbe comunque bene
- Ma in questo modo make controlla la sua data di ultima modifica e quindi ricompila somma.o anche se solo il suo file header è cambiato

# Compilazione

- La cosa interessante, infatti, è che make compila tutto e solo quello che è strettamente necessario:
  - se somma.o è già compilato e i sorgenti non sono modificati, non viene ricompilato ma usato direttamente

```
[rama]:olivia [~/test] -> rm -f main main.o
[rama]:olivia [~/test] -> make main
gcc -Wall -Wno-return-type -Wno-implicit-int -g -O
-pedantic -Wformat=2 -Wextra    -c -o main.o main.c
gcc   main.o somma.o  -lm -o main
```