Struttura di un programma

File e moduli

Linkage, scope e prototipi

Compilazione e linking

Programmazione modulare

- **Tecnica di programmazione che ha lo scopo di aumentare l'indipendenza tra le sue parti costituenti, dette componenti o moduli**
- **Un modulo**
  - Raccoglie un insieme di funzionalità omogenee
  - Nasconde i dettagli implementativi
  - Costituisce una unità il più possibile completa e indipendente
- **Un modulo software è costituito da**
  - Una interfaccia
    - Descrive gli elementi che sono forniti e richiesti dal modulo
    - E visibile e accessibile ad altri moduli
  - Una implementazione
    - Realizza quanto esposto dall'interfaccia
    - Può essere completamente ignota a chi utilizza il modulo

(c) 2012 - Carlo Brandolese

Programmazione modulare

- **Costituisce la soluzione alternativa all'approccio monolitico**
  - Un unico complesso software risolve l'intero problema
  - E' indispensabile individuare un buon compromesso tra
    - Semplicità dei signoli moduli
    - Complessità delle relazioni tra di essi
- **Realizza il principio "divide and conquer"**
  - Cooperazione di diversi moduli indipendenti che risolvono sottoproblemi più semplici
- **Consente di organizzare il lavoro di sviluppo in team**
  - Richiede una attenta fase iniziale di definizione dell'architettura complessiva del sistema
- **Affinchè l'approccio modulare porti ad un reale beneficio è necessario che i moduli abbiano alcune proprietà**
  - Elevato livello di information hiding
    - Nascondere all'utente del modulo i dettagli implementativi
  - Basso accoppiamento
    - Dipendenza da altri moduli
  - Elevata coesione
    - Omogeneità delle funzioni offerte da un modulo

(c) 2012 - Carlo Brandolese

Information hiding

- **Si definisce information hiding il principio secondo cui**
  - Si nascondono le decisioni implementative di un modulo agli utenti del modulo stesso

- **In particolare si vuole nascondere**
  - Dati
    - Le strutture dati interne ad un modulo
    - La rappresentazione dei dati
  - Algoritmi
    - Il modo specifico in cui un dato comportamento viene realizzato

- **Si vuole invece esporre**
  - Una interfaccia di accesso ai dati
  - Una interfaccia di accesso alle funzionalità

- **E' bene cercare di massimizzare l'information hiding**

(c) 2012 - Carlo Brandolese

Accoppiamento

- **L'accoppiamento misura livello di dipendenza reciproca tra due moduli**

- **Dati due moduli A e B si ha accoppiamento se**
  - A ha un attributo che si riferisce a B
  - A chiama una funzione di B
  - A definisce una funzione che si riferisce a tipi definiti in B
  - A definisce una struttura che contiene un tipo definito in B

- **L'accoppiamento**
  - Tende a migliorare le prestazioni
  - Tende a diminuire la modularità e la riusabilità del codice

- **E' bene cercare di minimizzare l'accoppiamento**

(c) 2012 - Carlo Brandolese

Accoppiamento - Tipologie

- **Contenuto (alto)**
  - Un modulo dipende dal modo in cui un'altro è strutturato internamente
    - La posizione in memoria di un dato, da un tipo di un dato o dalle tempistiche di elaborazione
- **Dati comuni**
  - Due moduli utilizzano dati globali comuni
- **Esterno**
  - Due moduli condividono un formato dei dati definito esternamente
    - Una interfaccia comune o un protocollo
- **Controllo**
  - Un modulo passa ad un altro informazioni che ne modificano il flusso di esecuzione
    - Per esempio un flag
- **Strutture dati**
  - Due moduli usano campi diversi di una stessa struttura dati
- **Parametri**
  - Un modulo passa parametri ad un altro modulo mediante chiamate di funzione
- **Messaggi (basso)**
  - Un modulo utilizza una funzione di un altro modulo priva di argomenti

(c) 2012 - Carlo Brandolese

Coesione

- **La coesione misura quanto sono correlate le diverse parti di un modulo**

- **Una bassa coesione**
  - Rende difficile comprendere le funzioni svolte da un modulo
  - Rende difficile la manutenzione del modulo poiché un cambiamento si riflette su ambiti diversi del programma
  - Rende difficile o inefficiente il riuso del codice

- **Una elevata coesione**
  - Può rendere meno efficiente il codice
  - Può causare una eccessiva suddivisione in moduli

- **È bene massimizzare la coesione dei moduli**

(c) 2012 - Carlo Brandolese

Coesione - Tipologie

- **Casuale (peggiore)**
  - Un modulo raccoglie funzioni e dati scelti arbitrariamente
- **Logica**
  - Un modulo raccoglie funzioni correlate logicamente ma prive vi relazioni forti P
    - Per esempio tutte le funzioni di I/O
- **Temporale**
  - Un modulo raccoglie le funzioni che vengono usate in momenti vicini nel tempo
    - Per esempio funzioni che gestiscono eccezioni su un file, creano un log e notificano un errore
- **Procedurale**
  - Un modulo raccoglie funzioni usate sempre in sequenza
    - Per esempio funzioni che verificano i permessi su un file, lo aprono, lo modificano e lo chiudono
- **Comunicazione**
  - Un modulo raggruppa le funzioni che operano su uno stesso dato
- **Sequenziale**
  - Un modulo raggruppa funzioni per cui l'output di una serve come input di un'altra
- **Funzionale (migliore)**
  - Un modulo raggruppa tutte le funzioni che svolgono una ben precisa funzione
    - Per esempio la costruzione e la visita di un albero

(c) 2012 - Carlo Brandolese

Moduli in C

- **Un modulo in C è sempre costituito da**
  - Un file header (.h) che dichiara l'interfaccia
  - Uno o più file sorgenti (.c) che definiscono dati e funzioni

- **Esempio: Contatore**
  - Specifica
    - Il modulo deve implementare un contatore unico e condiviso da tutti gli utenti
  - Interfaccia
    - Le funzioni richieste sono incremento, lettura e reset
    - Il contatore deve poter essere configurato a compile-time per avere la funzione opzionale di decremento
  - Implementazione
    - Il modulo deve disporre di una variabile interna che mantiene il valore corrente
    - Il modulo deve implementare le funzioni dell'interfaccia

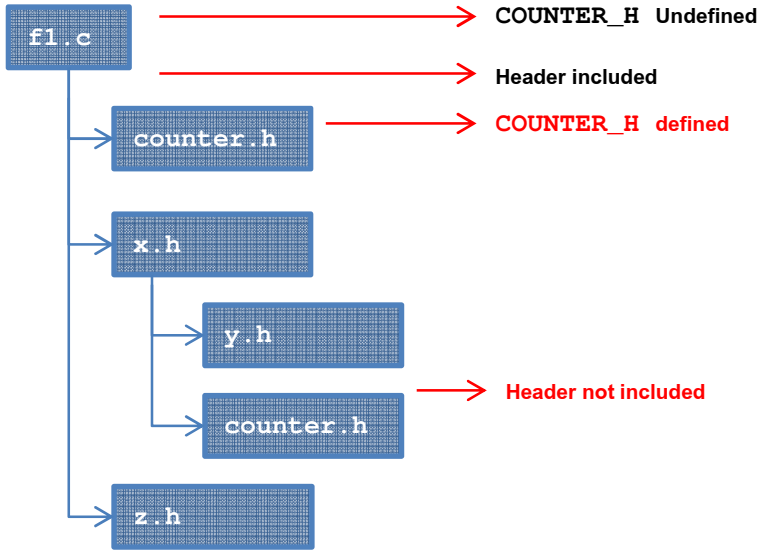
(c) 2012 - Carlo Brandolese

Moduli in C

counter.h

```
#ifndef COUNTER_H
#define COUNTER_H

/* Mandatory functions ************************/

void cnt_reset( void );
void cnt_inc( void );
int  cnt_get( void );

/* Optional function **************************/

#ifdef CNT_HAS_DECREMENT
void cnt_dec( void );
#endif

#endif
```

- **Compilazione condizionale**
  - COUNTER_H
    - Definita la prima volta che il file è incluso
    - Previene l'inclusione multipla



  - CNT_HAS_DECREMENT
    - Deve essere definita dal programmatore
    - Se definita, causa la definzione della funzione di decremento

Moduli in C

counter.c

```
#include "counter.h"

/* Hidden data ******************************/

static int cnt = 0;

/* Implementation ***************************/

void cnt_reset( void ) {
  cnt = 0;
}
void cnt_inc( void ) {
  cnt++;
}

int cnt_get( void ) {
  return cnt;
}

#ifdef CNT_HAS_DECREMENT
void cnt_dec( void ) {
  cnt--;
}
#endif
```

- **Il valore del contatore non deve essere visibile direttamente dal di fuori del modulo**
  - Si rende il simbolo cnt statico
    - Si dice che il simbolo ha "internal linkage"
  - Si forniscono funzioni di accesso
    - In lettura
    - In scrittura
- **Anche l'implementazione della funzione di decremento deve essere sotto compilazione condizionale**

Scope e linkage

- **Un simbolo globale può essere**
  - Una variabile o una costante
  - Una funzione
- **Per comprendere come sono trattati i simboli sono necessari tre concetti**
  - Scope:
    - La parte di codice in cui un simbolo è visibile, cioè noto al compilatore
  - Linkage:
    - L'area in cui il simbolo è visibile al linker
  - Translation unit:
    - La parte di codice che il compilatore tratta in una sola volta
- **I simboli globali**
  - Hanno scope "globale" o "file scope", sono visibili nella translation unit corrente
  - Hanno "external linkage", sono potenzialmente visibili in altre translation unit
- **Il modificatore static**
  - Cambia il linkage da external a internal
  - Non modifica lo scope

(c) 2012 - Carlo Brandolese

Scope e linkage

- **Variabile e funzione a file scope con linkage esterno**

a.c

```
int var;
int other;
int fun( float ) {...}
```

  - La funzione fun() può essere usata direttamente in altre translation unit
    - Tuttavia non ne sarà noto il prototipo (vedi oltre)
  - Le variabili var e other possono essere usata in altre translation unit
    - Se specificato esplicitamente mediante la dichiarazione extern

- **In un'altra translation unit, dunque**

b.c

```
extern int var;

void bar() {
  int n = var + 1;
  int m = fun( 1.5 );
  int k = other - 1;    /* Linker error: other is unknown to the linker */
  ...
}
```

Scope e linkage

- **Variabile e funzione a file scope con linkage interno**

a.c

```
static int var;
static int other;
static int fun( float ) {...}
```

  - Né le variabili né la funzione possono essere usate in altre translation unit

- **In un'altra translation unit, dunque**

b.c

```
extern int var;           /* Linker error: conflicts with static declaration */
extern int fun(float);    /* Linker error: conflicts with static declaration */

void bar() {
  int n = var + 1;
  int m = fun( 1.5 );
  int k = other - 1;      /* Linker error: other is unknown to the linker */
  ...
}
```

  - Il modificatore extern usato in un'altra translation unit non ha alcun effetto
  - Se usato per cercare di accedere a var o a fun() produce un errore in fase di linking

Prototipi

- **Il "prototipo" o "signature" di una funzione ne dichiara**
  - Il nome
  - Il tipo di ritorno
  - Il numero de il tipo di ogni argomento
- **Per default, il compilatore assume che una funzione**
  - Abbia sempre tipo di ritorno int
  - Abbia un numero qualsiasi di argomenti int
- **Se una funzione è dichiarata in una translation unit ed usata in un'altra**
  - E' necessario indicare il suo prototipo nella translation unit in cui è usata
- **Il prototipo deve essere completo ed esplicito**
  - Se una funzione non ritorna alcun valore deve essere dichiarata come void
  - Se una funzione non accetta alcun argomento deve avere una lista di argomenti void

```
/* INCOMPLETE PROTOTYPES */
f1();               /* Return type is int, accepts an unspecified number of int arguments */
char f2();          /* Return type is char, accepts an unspecified number of int arguments */
void f3();          /* No return type, accepts an unspecified number of int arguments */
int  f4(void);      /* Return type is int, accepts no arguments */
/* COMPLETE PROTOTYPE    */
int  f5(int);       /* Return type is int, accepts a single int argument */
```

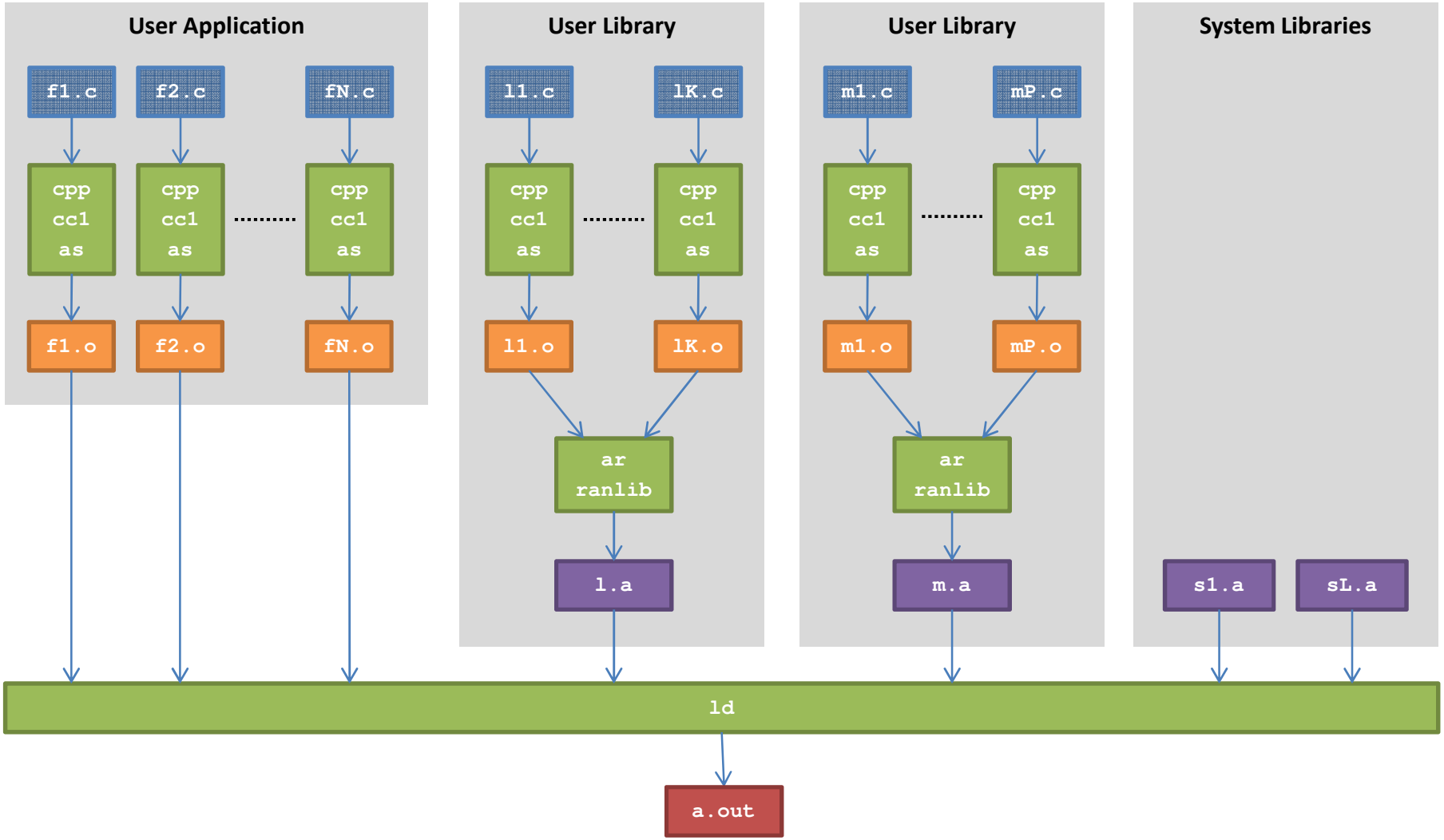
(c) 2012 - Carlo Brandolese

Compilazione e linking

- **Il processo di compilazione è complesso e articolato**
  - Spesso viene nascosto da un unica interfaccia (grafica o command-line)
  - Si compone di diversi passi
- **Preprocessore (cpp)**
  - Analizza le macro e le direttive di preprocessore
  - Ignora il linguaggio C
  - Produce un nuovo file sorgente
- **Compilatore (cc1)**
  - Traduce il codice sorgente in codice assembly simbolico
- **Assembler (as)**
  - Traduce il codice assembly simbolico in codice macchina
  - Il risultato è un file oggetto
- **Linker (ld, ar, ranlib)**
  - Combina più file oggetti in
    - Un eseguibile (ld)
    - Una libreria (ar, ranlib)

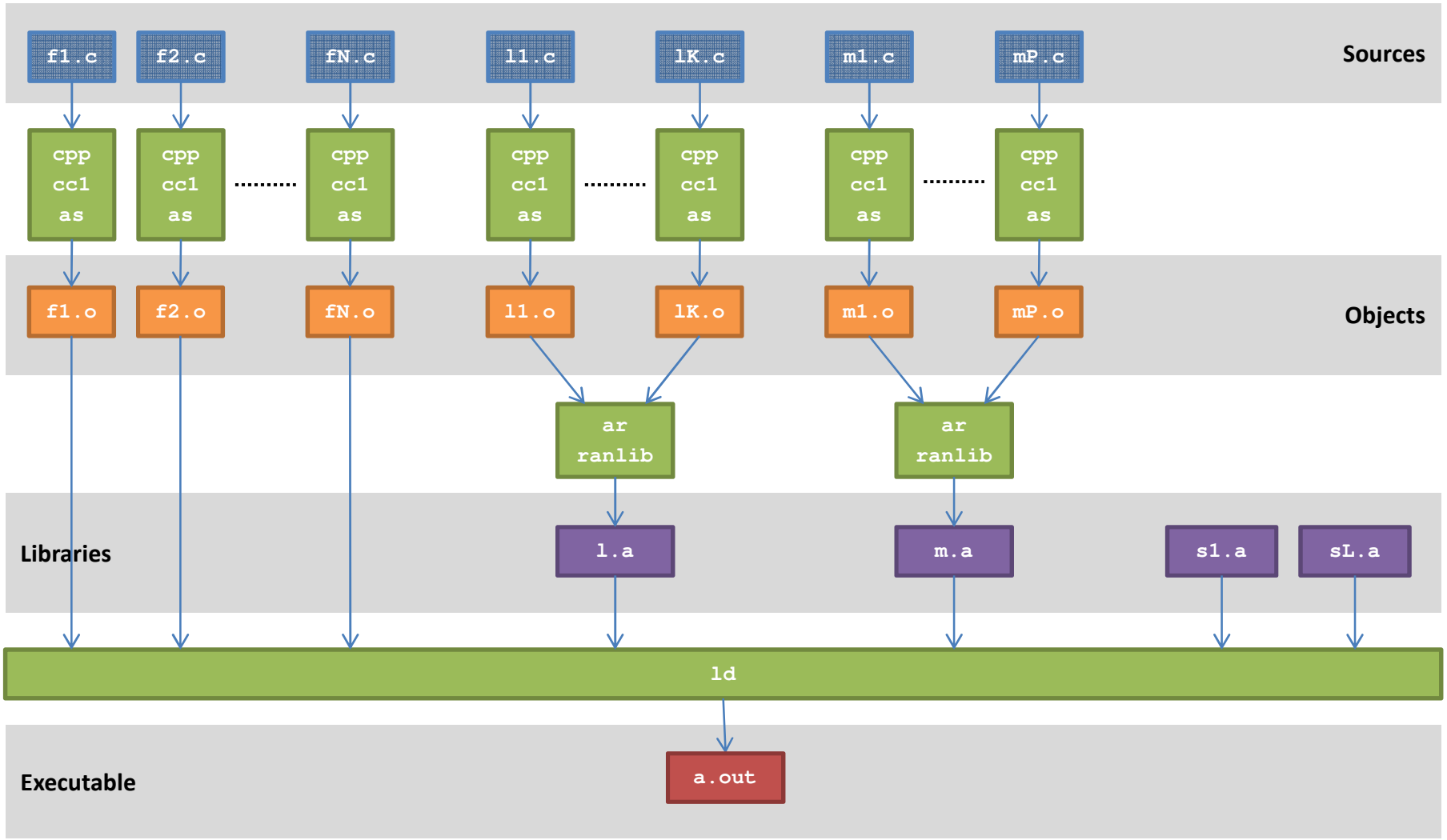
(c) 2012 - Carlo Brandolese

Compilazione e linking



(c) 2012 - Carlo Brandolese

Compilazione e linking



(c) 2012 - Carlo Brandolese