Programmazione B
Ingegneria e Scienze Informatiche - Cesena
A.A. 2021-2022

Introduzione al linguaggio C

Catia Prandi - catia.prandi2@unibo.it

Credit: Pietro Di Lena

---

pro-gram-mer */prəʊgræmə^{r}/* n.
1. A tool that turns coffee into code.
2. A person who fixed a problem that you don't know in a way that you don't understand.

Linguaggi di programmazione

- In Informatica, viene detto **linguaggio di programmazione** un linguaggio utilizzato per descrivere l'insieme delle azioni consecutive che un calcolatore deve eseguire.
- Un *linguaggio di programmazione* permette agli esseri umani di comunicare con i calcolatori, allo stesso modo in cui un *linguaggio naturale* permette agli essere umani di comunicare tra di loro.
- Il **programmatore** è una figura professionale, che attraverso la fase di *programmazione*, traduce (o *codifica*) un *algoritmo* in **codice sorgente**, utilizzando un **linguaggio di programmazione**.
- Storicamente, Ada Lovelace (1815-1852) è considerata la prima programmatrice della storia, avendo codificato un algoritmo per un calcolatore meccanico, la *macchina analitica* di Charles Babbage.
  Maggiori dettagli: Pagina wiki

Linguaggi naturali vs. Linguaggi di programmazione: similarità

- Sono utilizzati per uno scopo comune: comunicare.
- Condividono le stesse classi di elementi linguistici:
  - **Alfabeto**: insieme non vuoto di simboli tramite i quali è possibile costruire gli elementi base del linguaggio.
  - **Lessico**: insieme di regole formali per la scrittura di parole nel linguaggio.
  - **Sintassi**: insieme di regole formali per la scrittura di frasi nel linguaggio.
  - **Semantica**: insieme di significati da attribuire alle frasi sintatticamente corrette del linguaggio.
- Come accade per i linguaggi naturali, la difficoltà nell'apprendere un nuovo linguaggio di programmazione decresce all'aumentare del numero di linguaggi conosciuti.
- Conoscere la sintassi di un linguaggio naturale non implica la capacità di saper esprimere concetti semanticamente sensati. Conoscere la sintassi di un linguaggio di programmazione non implica la capacità di saper codificare programmi (algoritmi) semanticamente sensati.

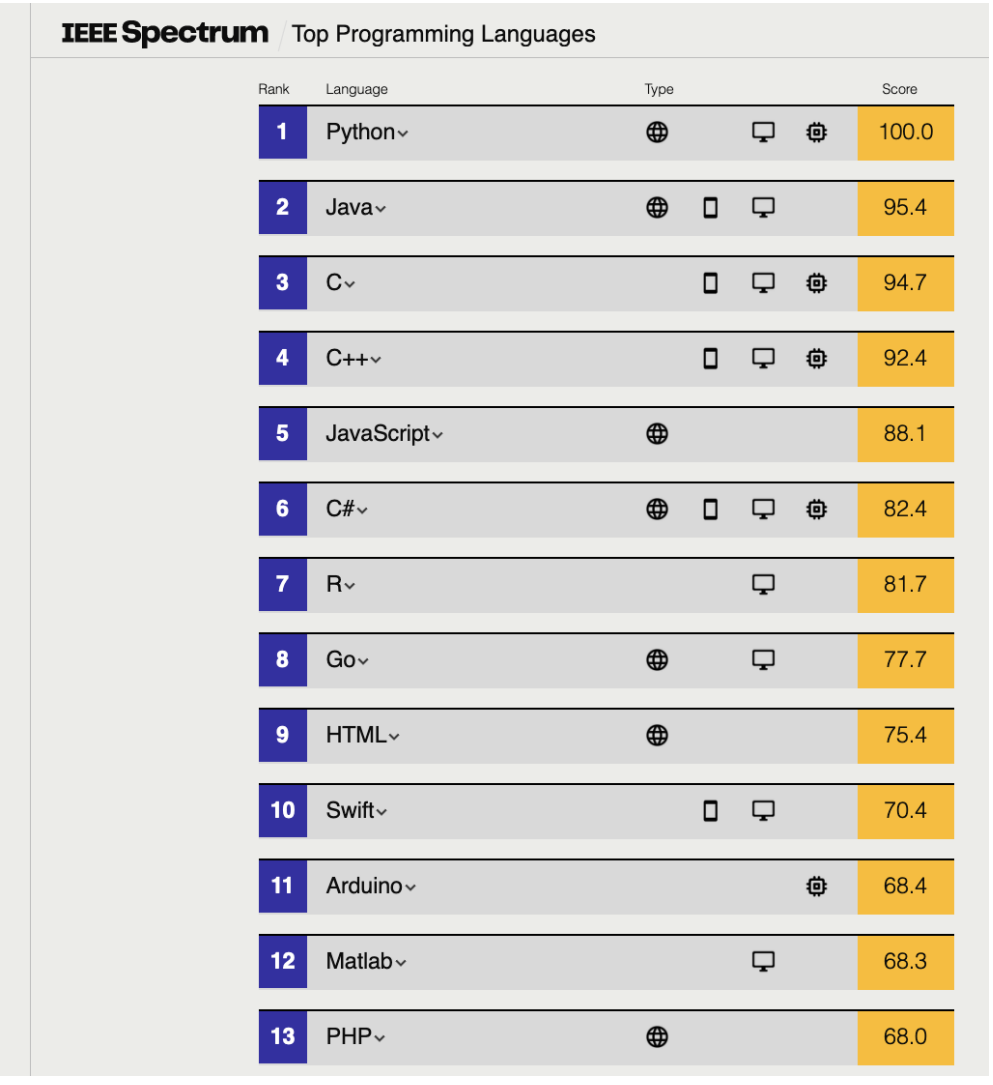
Linguaggi naturali vs. Linguaggi di programmazione: differenze

Linguaggi naturali	**Linguaggi di programmazione**
Si sviluppano ed affermano *spontaneamente* nelle culture umane	Sono creati *artificialmente* e la loro longevità è strettamente legata agli ambiti specifici di applicazione del linguaggio stesso
Sono utilizzati per la comunicazione tra esseri umani	Sono utilizzati unicamente per istruire comandi ai calcolatori
Adatti ad esprimere sentimenti e concetti filosofici	Adatti ad esprimere istruzioni
Sono estremamente ridondanti	Sono succinti
La semantica è ambigua e dipende dal contesto del discorso	La semantica non dovrebbe essere ambigua

Breve storia del linguaggio C

- **1969.** Ken Thompson scrive la prima versione del sistema operativo UNIX su un computer PDP-7, utilizzando il linguaggio Assembly della macchina. Thompson si occupa anche di sviluppare un linguaggio di più *alto livello* rispetto all'Assembly per l'implementazione di sistemi operativi. Nasce il linguaggio B, ispirato a due esperienze precedenti: il CPL (Combined Programming Language) e il BCPL (Basic CPL).
- **1972.** Partendo dal linguaggio B, **Dennis Ritchie** scrive la prima versione del linguaggio C, che viene utilizzato per riscrivere quasi totalmente il sistema operativo UNIX. Il linguaggio C si dimostra particolarmente adatto alla scrittura di software complesso ed efficiente. Negli anni successivi, il linguaggio C si diffonde su diverse architetture.
- **1978. Kernighan** e **Ritchie** pubblicano il testo di riferimento per il linguaggio C: "The C programming language".
- **1983.** Il linguaggio C è utilizzato su numerose architetture di calcolatori e sistemi operativi. Esistono diversi compilatori per il linguaggio, non perfettamente compatibili tra di loro. Iniziano a diffondersi *dialetti* (varianti) del C che aggiungono funzionalità non sempre conformi alla versione di Kernighan e Ritchie. Data la crescente popolarità del linguaggio, nasce la necessità di definire uno *standard* che permetta di scrivere codice *portabile* tra le diverse architetture. L'istituto americano **ANSI** (*American National Standard Institute*) inizia a lavorare ad una *standardizzazione* del linguaggio.
- **1989.** Nasce lo standard ANSI C, noto anche come C89. Lo standard viene successivamente approvato dall'ISO (*International Organization for Standardization*) e indicato come C90. ANSI C, ISO C89 e ISO C90 si riferiscono allo stesso standard.

Popolarità e influenza del linguaggio C nel 2021



IEEE Spectrum / Top Programming Languages

Rank	Language	Type	Score
1	Python	web, desktop, embedded	100.0
2	Java	web, mobile, desktop	95.4
3	C	mobile, desktop, embedded	94.7
4	C++	mobile, desktop, embedded	92.4
5	JavaScript	web	88.1
6	C#	web, mobile, desktop, embedded	82.4
7	R	desktop	81.7
8	Go	web, desktop	77.7
9	HTML	web	75.4
10	Swift	mobile, desktop	70.4
11	Arduino	embedded	68.4
12	Matlab	desktop	68.3
13	PHP	web	68.0

IEEE spectrum

- Linguaggi derivati dal C: genealogia dei linguaggi di programmazione.

Caratteristiche generali del linguaggio C

- *Il C è un linguaggio di programmazione di uso generale, caratterizzato dalla sinteticità, da un controllo del flusso e da strutture dati avanzate, e da un vasto insieme di operatori. Il C non è un vero "linguaggio ad alto livello", e non è specializzato in alcun'area applicativa. Ma il suo essere privo di restrizioni e la sua generalità lo rendono spesso più conveniente ed efficiente di altri linguaggi supposti più potenti.*

  B.W. Kernigan, D.M. Ritchie, "The C programming language", 1978

- Il C può essere definito come:
  - **un linguaggio di "medio livello"**: molto vicino al livello di funzionamento dell'hardware di un calcolatore ma di più semplice utilizzo che l'Assembly o il linguaggio macchina. Per questo motivo è attualmente molto utilizzato per l'implementazione di *sistemi operativi* e *driver* per il controllo di periferiche.
  - **un linguaggio procedurale e imperativo**: un programma in C è definito come una collezione di procedure (dette anche funzioni) che contengono comandi.
  - **un linguaggio dichiarativo e tipizzato**: il *tipo* di ogni variabile deve essere *dichiarato* dal programmatore.
  - **un linguaggio compilato**: il codice sorgente viene tradotto in codice macchina (binario) per poter essere eseguito. I linguaggi compilati si differenziano dai linguaggi *interpretati* che sono eseguiti traducendo di volta in volta le istruzioni in linguaggio macchina.

Punti di forza del linguaggio C

- **Portabilità**: un programma scritto in ANSI C può essere compilato (quasi sempre) senza modifiche su diverse architetture hardware e diversi sistemi operativi.
- **Efficienza**: il codice prodotto è estremamente veloce (molte istruzioni C sono traducibili direttamente in linguaggio macchina).
- **Potere espressivo**: è un linguaggio di alto livello che integra caratteristiche di basso livello, molto vicine alla logica Assembly.
- **Sinteticità**: è un linguaggio estremamente compatto, essendo dotato di un limitato numero di parole chiave.
- **Longevità**: con una storia di oltre 40 anni, dispone di una vasta letteratura e di un vasto numero di librerie.

Debolezze del linguaggio C

- **Cripticità**: il codice scritto in C può essere potenzialmente (o intenzionalmente) incomprensibile.
- **Scarsa sicurezza**: è molto facile scrivere codice *vulnerabile* in C.
- **Complessità**: alcuni aspetti del linguaggio C possono risultare alquanto difficili da assimilare.
- **Severità**: il linguaggio C non perdona gli errori!

Lo standard C

- La **standardizzazione** di un linguaggio di programmazione è la procedura con cui vengono fissate le caratteristiche lessicali e semantiche del linguaggio.
- La standardizzazione è un processo importante di un linguaggio di programmazione, al fine di evitare la proliferazione di dialetti del linguaggio che possono limitare la **portabilità** del codice su architetture hardware e sistemi operativi differenti.
  - Idealmente, vogliamo che il nostro codice sia compilabile ed eseguibile su architetture e sistemi operativi differenti senza alcun tipo di modifica.
- Esistono diverse revisioni dello standard C[1], ogni revisione mantiene la compatibilità con le versioni precedenti e ne aggiunge funzionalità:
  - ANSI C, C89, C90. Prima versione dello standard C.
  - C95. Revisione pubblicata nel 1995 dall'ISO. Contiene alcune correzioni e alcune (poche) estensioni al linguaggio.
  - C99. Estensione del linguaggio C pubblicata dell'ISO nel 1999 e accettata dall'ANSI nel 2000.
  - C11 Ultima versione del linguaggio C, pubblicata dall'ISO nel 2011

[1] I documenti originali non sono gratuiti. Possono essere richiesti agli istituti di standardizzazione per un costo non banale. E' possibile trovare sul web numerosa documentazione (sperabilmente) consistente con i documenti originali.

Primo esempio di codice C: Hello, World!

```c
#include<stdio.h>

/* Stampa Hello, World! sul terminale */

int main() {
 printf("Hello, World!\n");

 return 0;
}
```

- Semplice programma C che stampa a video la frase `"Hello, World!"`.
- Esempio tradizionalmente utilizzato in molti manuali di programmazione per illustrare le caratteristiche basilari (lessico, sintassi e semantica) di un linguaggio di programmazione. Storicamente, la prima apparizione di questo esempio in un testo di programmazione viene attribuita al manuale "The C programming language" di Kernighan and Ritchie.
- La numerazione sulla sinistra del riquadro non fa parte del codice sorgente, ma è un espediente tipografico per poter fare facilmente riferimento alle righe di codice.
- La colorazione è utilizzata da numerosi editor di sviluppo per evidenziare facilmente alcuni elementi sintattici del codice.

Direttive al preprocessore

```
1 #include<stdio.h>
2
3 /* Stampa Hello, World! sul terminale */
4
5 int main() {
6  printf("Hello, World!\n");
7
8  return 0;
9 }
```

- La **direttiva** `#include` (riga 1) viene utilizzata per includere nel file sorgente il contenuto del file `stdio.h`
- Tale direttiva viene gestita dal **preprocessore**, che si occupa di effettuare *trasformazioni e sostituzioni* sul file sorgente, prima della compilazione.
- Le direttive al preprocessore sono identificabili come le righe di codice che cominciano con #.
- La sintassi `#include <file.h>` (parentesi angolari) indica che il file da includere (`file.h`) risiede in una directory di sistema, nota al compilatore.

Commenti

```
1 #include<stdio.h>
2
3 /* Stampa Hello, World! sul terminale */
4
5 int main() {
6  printf("Hello, World!\n");
7
8  return 0;
9 }
```

- La riga 3 contiene un **commento**, che spiega brevemente cosa fa il programma. I commenti vengono utilizzati per rendere più leggibile il codice.
- Il testo compreso tra `/*` e `*/` viene ignorato dal compilatore. Nello specifico, il preprocessore si occupa di rimuovere tutti i commenti dal file sorgente.
- La sintassi

  `/* Stampa Hello, World! sul terminale */`

  per i commenti è specifica del linguaggio C. La sintassi, introdotta nel linguaggio C++,

  `// Stampa Hello, World! sul terminale`

  indica come commento qualsiasi sequenza di caratteri compresa tra `//` e la fine della riga. Tale sintassi per i commenti è stata introdotta nello standard ISO C99 ma viene tipicamente accetta anche dai compilatori che accettano lo standard ISO C89/C90.

Funzione principale: main()

```
1 #include<stdio.h>
2
3 /* Stampa Hello, World! sul terminale */
4
5 int main() {
6  printf("Hello, World!\n");
7
8  return 0;
9 }
```

- Le righe di codice tra la linea 5 e la linea 9 contengono la **definizione** della funzione `main()`. Il `main()` è una funzione speciale in quanto indica il punto di partenza dell'esecuzione del programma. Ogni programma deve contenere una sola funzione `main()`.
- Il *corpo* della funzione `main()` consiste nelle **istruzioni** incluse nel *blocco* delimitato dalle due parentesi graffe. In questo caso, il `main()` contiene due *istruzioni*: una a riga 6 ed una a riga 8. Ogni istruzione è terminata dal carattere punto e virgola `';'`.
- Le parentesi tonde che seguono il nome della funzione contengono la lista di *argomenti/parametri* che sono passati alla funzione. In questo caso, le parentesi `()`, indicano che il `main()` non prende argomenti.

Funzione di stampa: printf()

```
1 #include<stdio.h>
2
3 /* Stampa Hello, World! sul terminale */
4
5 int main() {
6  printf("Hello, World!\n");
7
8  return 0;
9 }
```

- L'istruzione a riga 6 chiama (o invoca) la **funzione** (di libreria) `printf()`.
- La funzione `printf()` prende un argomento: la *stringa di testo* `"Hello, World!\n"`.
- Da notare la sequenza `\n` nella stringa di testo, che indica *l'andata a capo* (*newline*). In particolare, la `printf()` stampa sul terminale la frase `"Hello, World!"` e posiziona il cursore sulla riga successiva del terminale.
- Tutti i caratteri in una stringa che sono preceduti da `\` sono chiamati **caratteri di escape** e hanno un significato particolare.

Valore di uscita dalla funzione main()

```
1 #include<stdio.h>
2
3 /* Stampa Hello, World! sul terminale */
4
5 int main() {
6  printf("Hello, World!\n");
7
8  return 0;
9 }
```

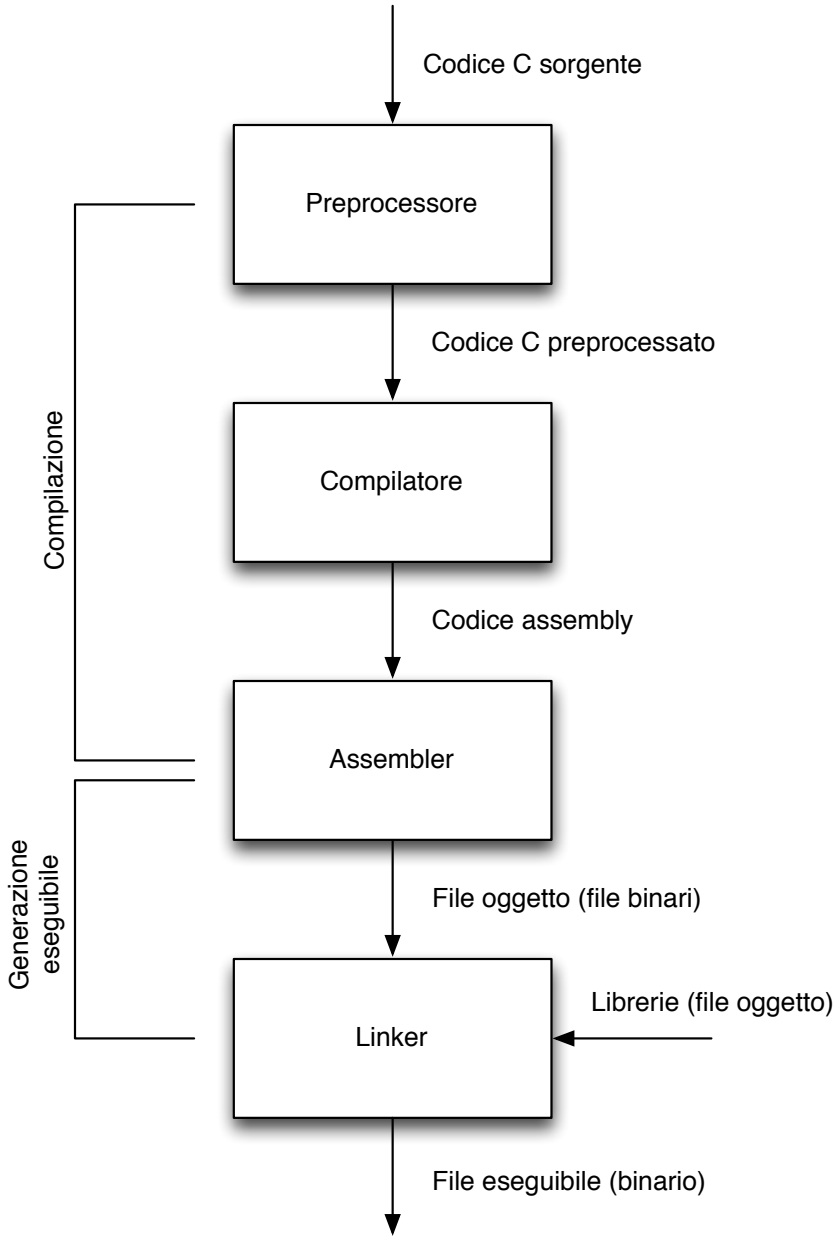
- L'istruzione a riga 8 causa la terminazione della funzione. Il controllo **ritorna** all'istruzione successiva alla chiamata di funzione. In questo caso, dato che il `main()` è la funzione di partenza del programma, l'istruzione `return` causa la terminazione del programma e il controllo *ritorna* al sistema.
- Il valore indicato dopo l'istruzione `return` specifica il **valore di uscita** della funzione `main()`. In questo caso, il valore di uscita è la *costante intera* 0, che convenzionalmente indica la *terminazione con successo* del programma.
- La **parola chiave** `int` prima del nome `main()` a riga 5, indica che il valore di uscita della funzione `main()` deve essere un valore intero. E' possibile definire funzioni che terminino senza specificare valori di uscita.

Indentazione e spazi

```
1 #include<stdio.h>int main(){printf("Hello, World!\n");return 0;}
```

- **L'indentazione** (o *rientro*) indica l'introduzione di un certo numero di spazi all'inizio di una riga (ad esempio, righe 6 e 8 nelle slide precedenti).
- Nell'ambito della programmazione, l'indentazione è convenzionalmente utilizzata per rendere più leggibile il codice. Viene utilizzata per *raggruppare logicamente* le istruzioni successive e rappresentare le *relazioni di annidamento* tra istruzioni.
- Nel linguaggio C gli *spazi, tab e gli a capo* sono ignorati dal compilatore, se non confondono alcuni elementi lessicali del linguaggio (ad esempio, è necessario uno spazio tra `int` e `main`).
- Il codice presentato in questa slide è semanticamente equivalente a quello mostrato nelle slide precedenti.

Fasi della compilazione



Preprocessore

- Il **preprocessore** legge in input un sorgente C e produce in output un sorgente C, dopo aver effettuato trasformazioni e sostituzioni sul file sorgente iniziale.
- Alcune operazioni gestite dal preprocessore C:
  - Inclusione di file esterni (tramite la direttiva `#include`).
  - Rimozione dei commenti.
  - Espansione delle *macro* (non le abbiamo ancora viste)
  - Escludere parti di codice al verificarsi di determinate condizioni (compilazione condizionale)

Compilatore

- Il **compilatore** è un programma che traduce un set di istruzioni (codice sorgente) scritte in un determinato linguaggio di programmazione (tipicamente di alto livello) in istruzioni in un altro linguaggio (tipicamente di basso livello, come l'Assembly).
- I compilatori dividono la compilazione in due fasi principali, il *front end* e il *back end*:
- **Front end.** Questo step di compilazione consiste di più fasi:
  - **Analisi lessicale**. Tramite un analizzatore lessicale il file sorgente viene suddiviso in una serie consecutiva di *token* (parole del linguaggio).
  - **Analisi sintattica**. Esegue il controllo sintattico sulla sequenza di token generati nella fase precedente e produce un albero di sintassi.
  - **Analisi semantica**. Esegue una serie di controllo per verificare che ogni istruzione del codice sorgente sia utilizzata in modo semanticamente corretto rispetto al contesto in cui si trova. Il risultato di questa fase è un *Albero di Sintassi Astratto* (AST).
  - **Generazione di codice intermedio**. Viene generato del codice intermedio dall'AST.
- **Back end.** Anche questo step consiste di più fasi:
  - **Ottimizzazione**. Il codice intermedio viene ottimizzato in modo da aumentare l'efficienza di calcolo.
  - **Generazione del codice target**. Il codice ottimizzato viene tradotto nel linguaggio target (tipicamente Assembly).

Assembler/Linker

- Il linguaggio **Assembly** è un linguaggio molto vicino al linguaggio macchina. La traduzione di codice Assembly in linguaggio macchina è eseguita dall'**Assembler**.
- Il passaggio da Assembly a linguaggio macchina è una operazione semplice, in quanto esiste (in linea di massima) una corrispondenza biunivoca tra istruzioni in Assembly e istruzioni binarie in linguaggio macchina. Esempio:

Linguaggio Assembly	Linguaggio macchina
movl $0, %ebp	0100 0000 0000 1000
cmpl $0, %ebp	0100 0000 0000 1001
jne L2	0000 0000 0000 1000

- Il passaggio da linguaggio di alto livello ad Assembly (spiegato nelle fasi precedenti) è molto più complesso.
- L'output dell'Assembler è un *file oggetto* binario.
- Il **Linker** è un programma che si occupa di integrare i vari file oggetto creati durante la compilazione con i file oggetto già presenti sul sistema (librerie precompilate).
- L'output del Linker è un file binario *eseguibile*.

Discussioni generali sulla compilazione

- La procedura di compilazione, per quanto consista di diverse fasi, può essere eseguita con un unico comando che traduce il file sorgente in un file binario eseguibile.
- Il compilatore interrompe la compilazione su file sorgenti che contengono errori lessicali e sintattici, oltre che *evidenti* errori semantici.
- Un compilatore non può verificare interamente la correttezza semantica del codice (problema *indecidibile*). Una compilazione andata a buon fine non indica necessariamente che il codice sia corretto e che si comporterà come atteso.
- In generale, i compilatori tramite messaggi di *warning* indicano parti del codice che potrebbero presentare problemi durante l'esecuzione ma che non sono di una gravità tale da interrompere la compilazione.
- I messaggi di warning, per quanto spesso ignorati dai programmatori (o meglio, da programmatori novizi), indicano effettivamente problemi rilevanti nel codice sorgente.