LINGUAGGI DI PROGRAMMAZIONE

Perché si programma? A cosa serve?

Programmare significa scrivere del codice in un particolare linguaggio di programmazione, per far risolvere un problema ad una macchina (calcolatore).

Il problema potrebbe essere dal più semplice (sommare due numeri) al più complesso (trovare la strada migliore per raggiungere un luogo, controllare l'erogazione di carburante in un'automobile, controllare l'inclinazione delle ali di un aeroplano per mantenerlo orizzontale durante un volo).

Si programma, dunque, per "dire" ad un computer come risolvere un problema.

Il programmatore deve conoscere come risolvere il problema (facendosi magari aiutare da un esperto nel problema) e poi deve tradurre il **processo risolutivo** in modo tale che sia il computer a risolverlo. Il problema potrebbe essere risolto anche senza l'utilizzo del computer ma quali sono i vantaggi di farli risolvere alla macchina?

1. La velocità di esecuzione
2. La precisione (infatti se un programma è scritto senza errori non sbaglia mai, grazie alla proprietà delle "ripetibilita'" degli algoritmi)

I primi passaggi per la scrittura di un programma sono dunque quelli che portano alla realizzazione dell'algoritmo:

PROBLEMA	→	PROCESSO RISOLUTIVO	→	ALGORITMO
		Come si risolve il problema		Sequenza finita ed ordinata di istruzioni che indicano ad un esecutore (computer) come risolvere il problema utilizzando le abilità dell'esecutore.

Una volta determinato l'algoritmo è necessario quindi "tradurlo" in un linguaggio comprensibile all'esecutore.

L'esecutore delle istruzioni è, nel nostro caso, il processore del computer, che è in grado di comprendere ed eseguire solamente istruzioni in linguaggio binario, infatti i primi calcolatori erano programmabili solamente in binario, ossia "scrivendo" dei bit (1 o 0) su dei circuiti elettronici

Esempio programmazione binaria ALTAIR 8800: https://www.youtube.com/watch?v=suyiMfzmZKs

1. LA PROGRAMMAZIONE A BASSO LIVELLO ED ALTO LIVELLO

Supponiamo di voler scrivere il programma “somma fra 5 e 2” in modo che venga eseguito dal computer “Macchina di Von Neumann”. Il programma potrebbe essere scritto direttamente in memoria centrale, in binario, nel seguente modo (e così si faceva all’inizio con i primi computer):

Ogni codice operativo (ad esempio LOD, ADD, SUB, MUL ecc.) ha un valore in binario:

0000000000000100 significa LOD

0000000000000101 significa STO

0000000000000000 significa ADD

Istruzioni

ind.	Op code	operando
0	0000000000000100 (LOD)	0000000000000101 (5)
1	0000000000000101 (STO)	0000000010000000 (128)
2	0000000000000100 (LOD)	0000000000000010 (2)
3	0000000000000000 (ADD)	0000000010000000 (128)
4	0000000000000101 (STO)	0000000010000001 (129)

Dati

128	0000000000000000	0000000000000101
129	0000000000000000	0000000000000111
130		

La programmazione in questo modo è molto complicata per il programmatore, è molto probabile commettere errori, è molto difficile apportare modifiche al codice.

Per facilitare la vita al programmatore, è stato inventato il primo linguaggio di programmazione, **l'assembly**, che associa a ciascuna istruzione un codice mnemonico più facile da ricordare rispetto alle sequenze binarie.

Insieme all' assembly (linguaggio di programmazione) è stato inventato uno speciale programma chiamato **Assembler**, che si occupa di tradurre nei rispettivi codici binari ogni istruzione.

Dopo che era stato inventato l'assembly, lo stesso programma poteva essere scritto nel seguente modo:

0	0000000000000100	0000000000000101	load #5	Metti in accu il valore 5
1	0000000000000101	0000000010000000	sto 128	Metti all'ind 128 il valore di Accu
2	0000000000000100	0000000000000010	load #2	Metti in Accu il valore 2
3	0000000000000000	0000000010000000	add 128	Somma ad Accu il contenuto dell' ind 128
4	0000000000000101	0000000010000001	sto 129	Metti il valore di Accu all'ind 129
5				

(i dati che vengono scritti sono gli stessi dell'esempio precedente)

La situazione, per il programmatore era leggermente migliorata con l'assembly, ma non poi tanto.

I problemi, nella programmazione con assembly, sono che:

1. Il linguaggio assembly è “molto vicino” al linguaggio macchina, (si nota anche dal fatto che ad OGNI istruzione ASSEMBLY corrisponde ad UNA istruzione in linguaggio macchina), questo significa che il programmatore deve concentrare le proprie forze per “adeguare la soluzione del problema al modo di funzionare della macchina” anziché concentrarsi sulla risoluzione corretta del problema. Questo perché il principio di funzionamento della processore (sa svolgere solo operazioni algebriche semplici) è diverso da quello dell’uomo. Infatti il problema precedente è risolto in maniera diversa per l’uomo e per la macchina:

UOMO: prendi 2, somma 5, ottieni il risultato

PROCESSORE: metti 5 nell’accumulatore

Metti il valore dell’accumulatore nella cella di memoria 128

Metti 2 nell’accumulatore

Somma il contenuto dell’accumulatore con la cella di memoria 128

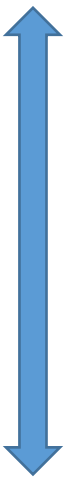
Metti il valore dell’accumulatore nella cella di memoria 129

Sarebbe molto utile che ci fossero degli strumenti che consentono di programmare il calcolatore che siano non solo più semplici, ma che “si avvicinino” al modo di pensare dell’uomo, ossia che siano **orientati alla soluzione del problema** e anziché orientati al funzionamento della macchina.

Questi strumenti esistono e sono i moderni **linguaggi di programmazione ad alto livello**.

Tali linguaggi non solo, quindi, sono **più semplici** da comprendere ed utilizzare, non solo hanno **istruzioni più “potenti”** (come ad esempio i cicli), ma soprattutto **sono più vicini al modo di ragionare del programmatore ed orientati alla risoluzione del problema**.

UOMO



ALTO LIVELLO — SQL; Pyton, JAVA, C# PASCAL,; C++; C

BASSO LIVELLO — ASSEMBLY

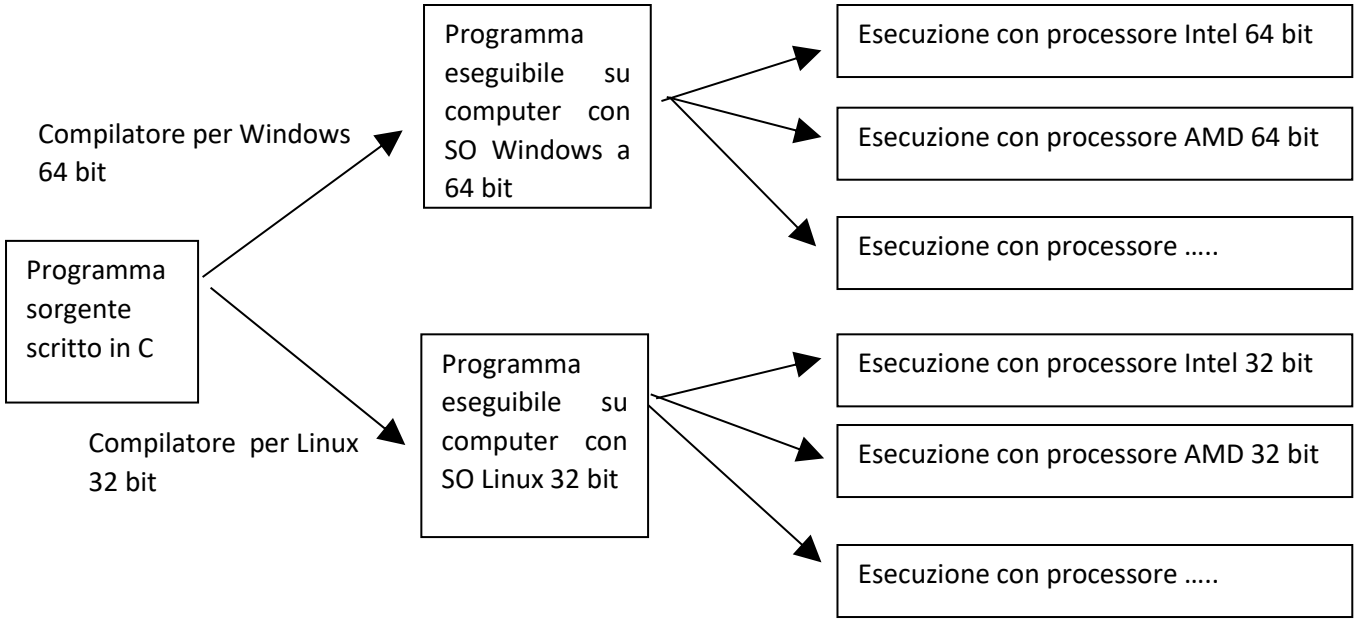
MACCHINA — BINARIO

2. Un secondo svantaggio dell'asembly a cui pongono rimedio i linguaggi di programmazione ad alto livello è il fatto che ogni processore ha un suo proprio insieme di istruzioni (set di istruzioni, Intel da 8086 in poi, motorola Z80….), questo fa sì che un programma scritto in linguaggio a basso livello per un processore non funzioni per un altro processore.

Scrivendo il codice in un programma ad alto livello invece il codice rimane lo stesso indipendentemente dall' hardware (processore) che lo eseguirà.

Infatti la "traduzione" del codice in istruzioni binarie, che siano comprensibili da questo o da quel processore, sarà affidata ad un apposito programma di traduzione chiamato **compilatore**. Il compilatore, **avvalendosi di opportune funzionalità fornite dal Sistema Operativo (SO) per accedere all'hardware** (le API, viste in TPS), potrà tradurre in binario le istruzioni ad alto livello indipendentemente da quale sarà la piattaforma (Sistema operativo + processore) che le dovrà eseguire. Poiché per la traduzione in binario il compilatore si avvale delle funzionalità del SO, vi sarà un compilatore diverso per ogni SO.

Un codice scritto in un qualsiasi linguaggio di programmazione ad alto livello quindi può essere compilato con compilatori diversi (ciascuno per un determinato SO) in modo da generare programmi binari eseguibili con sistemi operativi diversi indipendentemente dal processore. I linguaggi di programmazione hanno dunque consentito di scrivere codice **riusabile** per realizzare programmi **portabili** su diverse piattaforme (HW,SW).


Programma sorgente scritto in C
Compilatore per Windows 64 bit
Compilatore per Linux 32 bit
Programma eseguibile su computer con SO Windows a 64 bit
Programma eseguibile su computer con SO Linux 32 bit
Esecuzione con processore Intel 64 bit
Esecuzione con processore AMD 64 bit
Esecuzione con processore …..
Esecuzione con processore Intel 32 bit
Esecuzione con processore AMD 32 bit
Esecuzione con processore …..

2. LA TRADUZIONE DALL'ALGORITMO AL PROGRAMMA ESEGUIBILE

Riprendendo quanto detto all'inizio, una volta determinato l'algoritmo è necessario che esso divenga, alla fine di un processo di traduzione, una sequenza di istruzioni binarie comprensibili ed eseguibili dal processore. Vediamo le fasi che consentono tale traduzione.

1. **Codifica**: l'algoritmo viene tradotto in una sequenza di istruzioni in un determinato linguaggio di programmazione ad alto livello (ad esempio C). Questa operazione viene svolta dal **programmatore** che può scrivere utilizzando un semplice editor di testi ma generalmente si utilizzano appositi software chiamati **IDE** (Integrated Developement Enviroment= Ambienti di Sviluppo Integrati, ad es. Visual Studio o Dev C++ o CodeBlocks) che coadiuvano e supportano il programmatore durante la scrittura del codice. Il risultato di questa operazione è chiamato **codice sorgente**, che consiste, a seconda della complessità del programma, in uno o più file di testo.

Il linguaggio di programmazione (nel nostro caso C) ha delle proprie regole, è un cosiddetto **linguaggio formale**. Ciò significa che se le **regole** del linguaggio non vengono rispettate l'esecutore (computer) non è in grado di eseguire le istruzioni. I linguaggi formali dunque hanno una maggiore rigidità rispetto ai linguaggi naturali (come ad esempio la lingua inglese o italiana), in questi ultimi infatti pur non rispettando perfettamente le regole sintattiche, gli interlocutori possono più o meno capire il significato della comunicazione, nei linguaggi formali no.
Le regole del linguaggio di programmazione si dividono in **LESSICALI, SINTATTICHE, SEMANTICHE.**

REGOLE LESSICALI: esiste un "vocabolario" con i termini ed i simboli che è consentito utilizzare. Non si possono utilizzare termini e simboli al di fuori di quelli presenti nel vocabolario.
AD esempio in C:

IF, INT, +,-,*,/, = **si possono utilizzare**, hanno un significato

SE, ASSEGNA, √, **non si possono utilizzare**

L'utilizzo di simboli non appartenenti all'alfabeto genera errori lessicali. Tali errori sono rilevati dal compilatore che quindi, in loro presenza, non conclude la compilazione.

REGOLE SINTATTICHE: sono le regole "grammaticali" per utilizzare i termini ed i simboli del vocabolario. Ad esempio:

- Al termine dell'istruzione va posto il ";"
- I blocchi di istruzioni vanno posti fra parentesi graffe
- ….

Le violazioni di queste regole generano gli **errori di sintassi**. Tali errori sono rilevati dal compilatore che quindi, in loro presenza, non conclude la compilazione.

REGOLE SEMANTICHE: La semantica è il **significato** di un'istruzione, quindi gli errori semantici sono quelli legati al significato delle istruzioni, generalmente gli errori logici commessi dal programmatore. Un programma contiene errori semantici se viene eseguito (perché il compilatore non rileva errori lessicali o sintattici) ma produce risultati non corretti. Un esempio di errore semantico potrebbero essere:

```
int i=2;
if (i = 1)
        printf("i vale 1");
```

Il codice precedente scrive sul monitor "i vale 1" qualunque sia il valore assunto dalla variabile "i" perché nella condizione c'è un **assegnamento** (=) e non un **confronto** (= =).
Il codice corretto sarebbe:

```
int i=2;
if (i == 1)
        printf("i vale 1");
```

Gli errori semantici non vengono rilevati dal compilatore, per questo sono più insidiosi e difficili da rilevare. La ricerca di tali errori è chiamata **debugging** e viene eseguita dal programmatore con l'aiuto di uno specifico software, il **debugger**, che consente l'esecuzione passo-passo delle varie istruzioni del codice.

2. **Compilazione**: è l'operazione di "traduzione" del codice sorgente in una serie di istruzioni binarie chiamate **programma oggetto**. L'elemento che si occupa di questa traduzione è un apposito software chiamato **compilatore**. Il compilatore svolge questa operazione solamente dopo aver verificato l'assenza di errori lessicali e sintattici nel codice sorgente. In generale il programma oggetto è costituito da più file binari (**moduli**) che dovranno poi essere collegati fra di loro (ad esempio un file per il main, uno per alcune funzioni scritte dal programmatore, uno per le librerie standard <stdio.h>, <stdlib.h> ecc..).

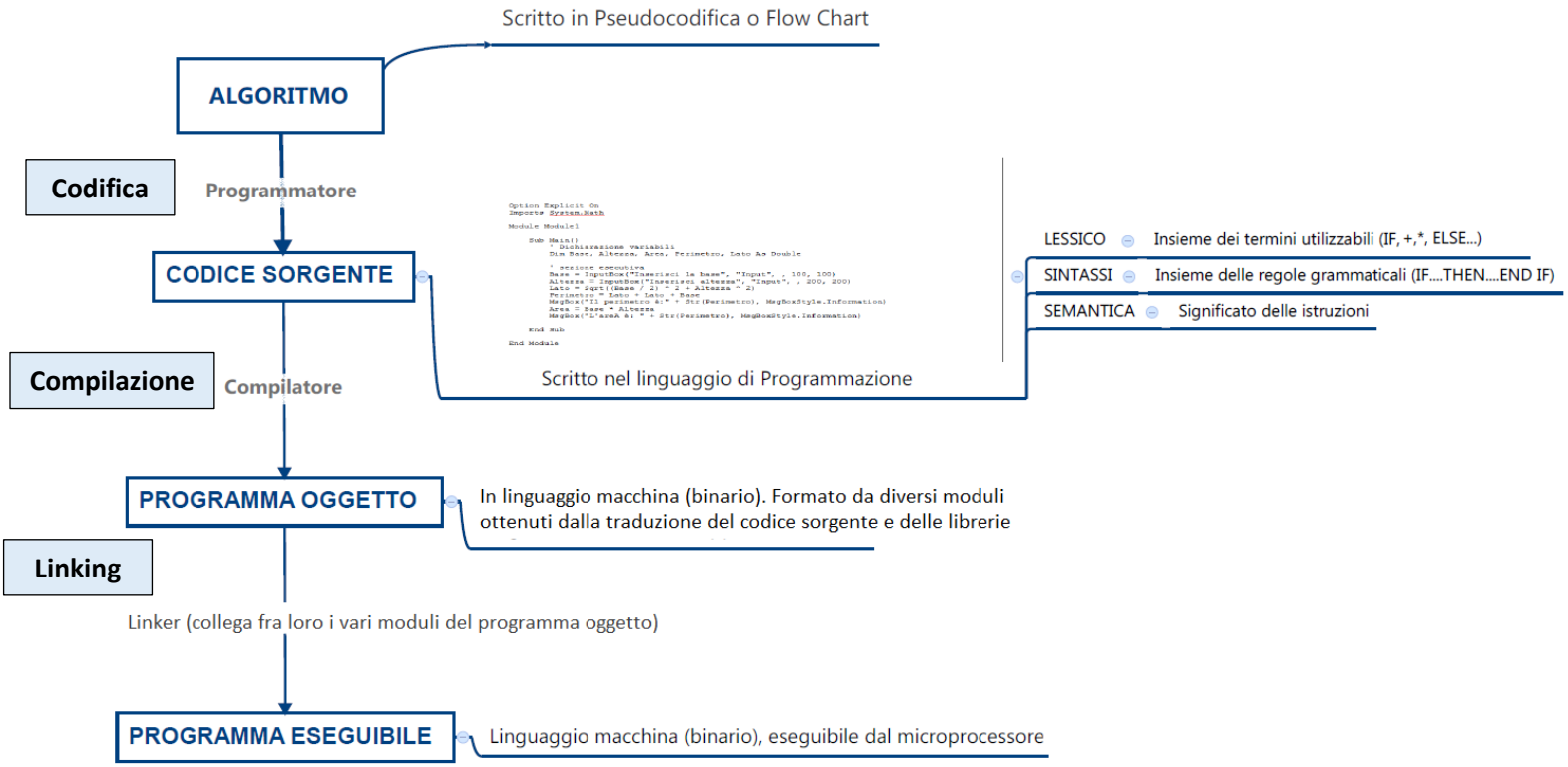
Poiché il linguaggio di programmazione ad alto livello mette a disposizione istruzioni (e strutture dati) più "complesse" rispetto al set di istruzioni di un processore, ossia istruzioni che "fanno più cose" (come ad esempio i cicli), **per ogni istruzione in linguaggio ad alto livello il compilatore genera più istruzioni binarie** (la corrispondenza fra linguaggi ad alto livello ed il codice binario **non è più 1 ad 1** come avviene nell'utilizzo dell'Assembly).

3. **Linking**: consiste nell'operazione di collegamento fra i vari moduli del programma oggetto. L'operazione è svolta da u n apposito software chiamato **linker**. Il risultato dell'operazione è il **programma eseguibile**.

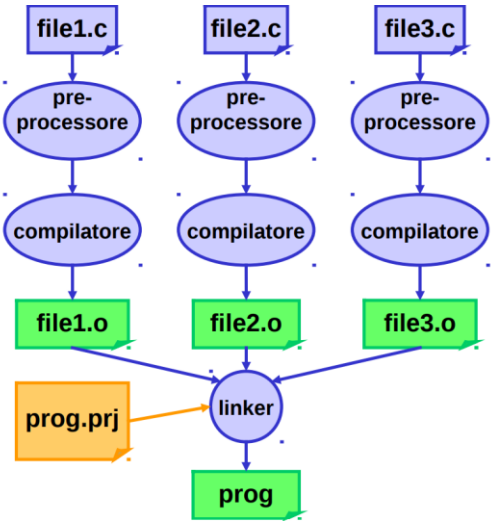
4. **Debugging**: consiste nell'eseguire una alla volta le istruzioni scritte nel codice sorgente al fine di scoprire ed eliminare gli eventuali errori che causano malfunzionamenti. Tale operazione è svolta dal programmatore utilizzando un apposito sw chiamato **debugger**, che è quello che consente l'esecuzione delle istruzioni una alla volta.

Un IDE è chiamato così perchè contiene tutti i software necessari per lo sviluppo di un programma: editor, compilatore, linker e debugger

Nella seguente mappa sono riassunte le fasi descritte:



Nei progetti di una certa dimensione il codice è separato in diversi file. Ciascun file viene compilato, tutti i programmi oggetto vengono collegati fra loro dal linker per produrre l'eseguibile:



Esercizio utile di compilazione "a mano": vediamo come è possibile eseguire i passi sopra descritti per compilare un programma in C invocando direttamente le funzioni del compilatore dalla riga di comando:

1. Creiamo un'applicazione costituita dai seguenti 3 file
   a. main.c — contiene la funzione main che acquisisce un intero da tastiera
   b. libreria.c — contiene una funzione "raddoppia" che restituisce il doppio di un intero x passato come parametro. Questo file è una libreria di funzioni creata dal programmatore e contiene il codice C delle funzioni.
   c. libreria.h — Questo è un header file. Contiene semplicemente le intestazioni delle funzioni della libreria. Questo file va incluso nei due file precedenti (il perché verrà spiegato nel modulo successivo sulle funzioni)
2. Il codice dei tre file è il seguente

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include"libreria.h"
int main(int arg, char *argv[])
{
    int x, doppio=0;
    printf("inserisci x: ");
    scanf("%i",&x);
```

libreria.c

```c
#include"libreria.h"
int raddoppia(int x)
{
    return 2*x;
}
```

libreria.h

```c
int raddoppia(int x);
```

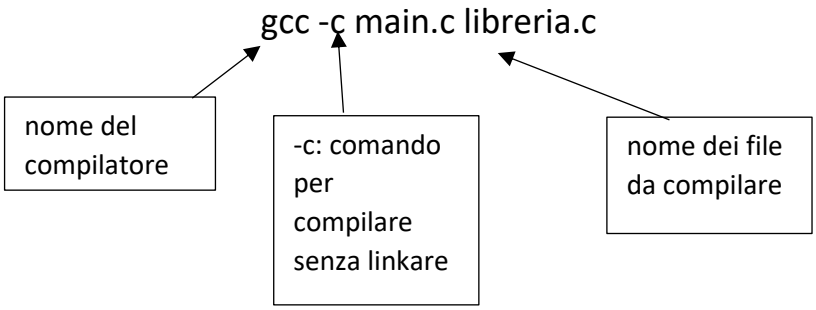
```
    doppio=raddoppia(x);
    printf("x=%i, doppio di x=%i",x,doppio);
}
```

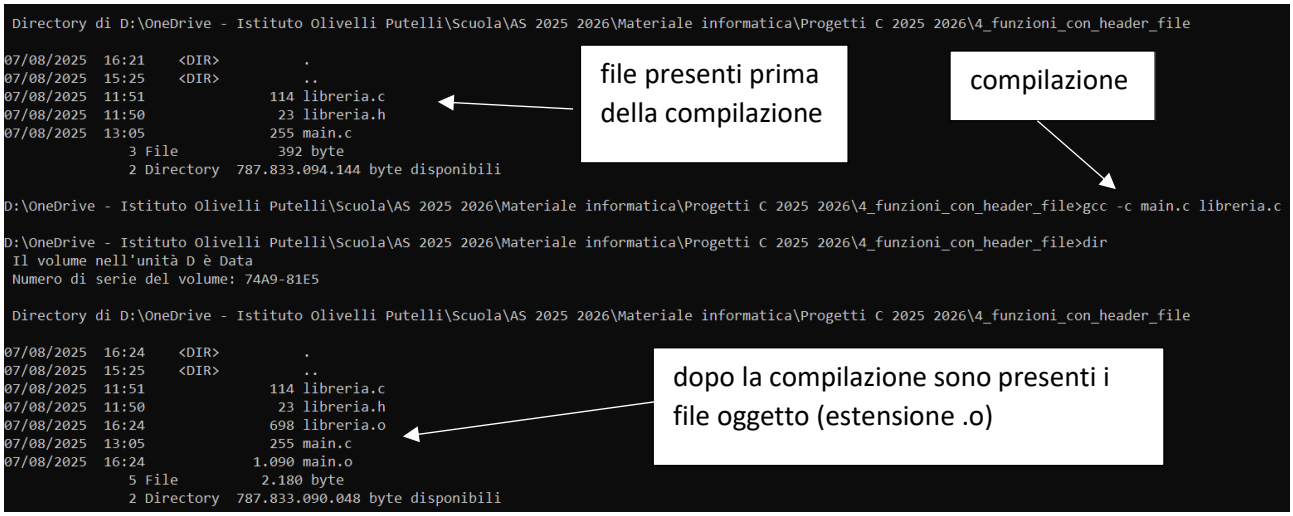
Come si può notare: all'interno della funzione main() è invocata la funzione "raddoppia()" il cui codice si trova nel file libreria.c.

Per creare l'eseguibile bisogna compilare ciascun file sorgente e poi unirli con il linker.

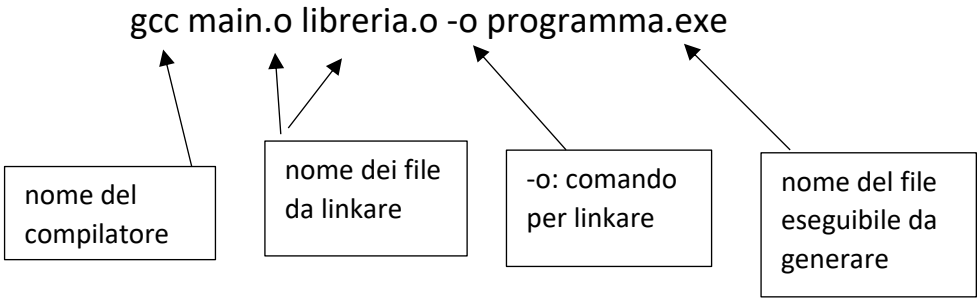
Vediamo i comandi del prompt che consentono tali operazioni.

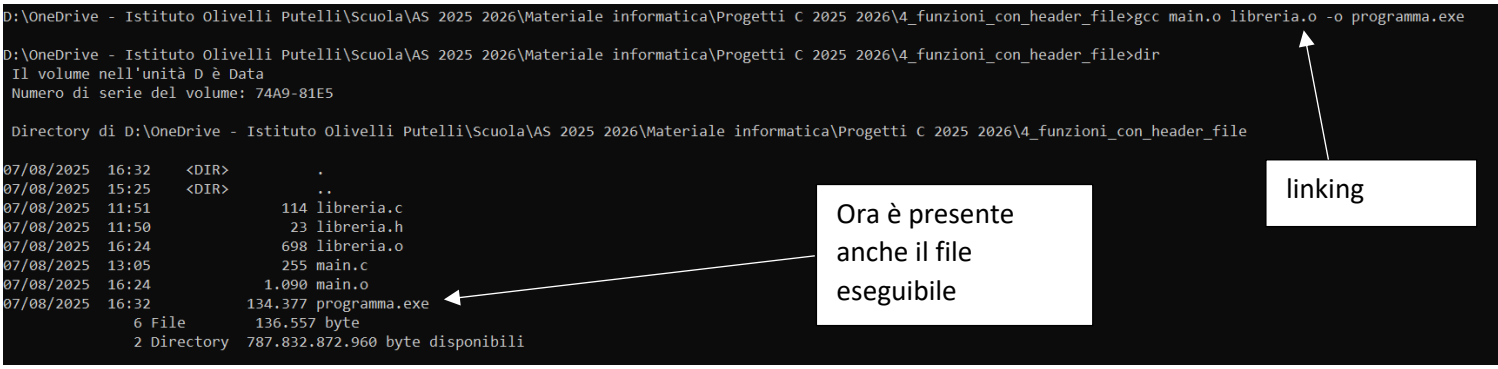
A. Comando per creare i file oggetto direttamente con il compilatore dal prompt dei comandi



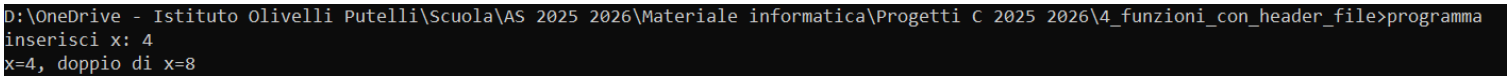


B. Comando che invoca il linker per collegare i file oggetto e crea il file eseguibile:





C. Ora si può eseguire l'applicazione programma .exe

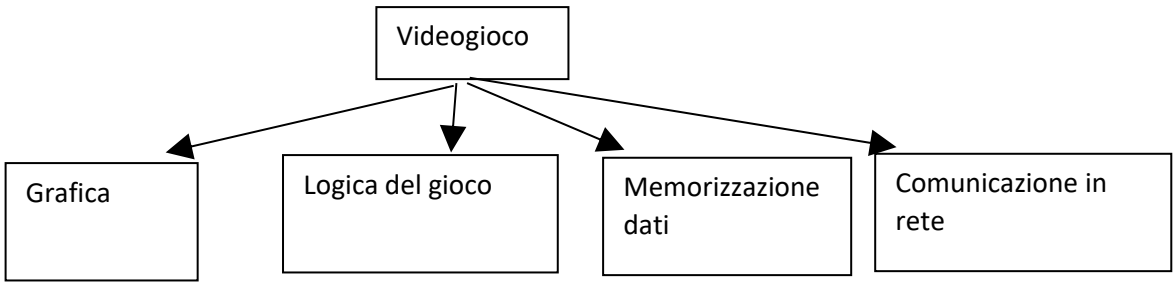


3. LE FASI PER REALIZZARE UN'APPLICAZIONE

Quando si realizza un software di una certa complessità è sempre necessario che le fasi precedentemente descritte vengano precedute e seguite da altre attività:

PRIMA

- **Progettazione:** per programmi complessi non è possibile scrivere il codice senza aver prima progettato la struttura e gli algoritmi. Ci sono appositi linguaggi per la progettazione del codice (UML).
  Ci sono diverse tecniche di progettazione del software, una delle più diffuse è la tecnica **top/down** che consiste nel partire dal problema da risolvere e nella sua scomposizione in sottoproblemi. Ogni sottoproblema può essere poi scomposto in altri sottoproblemi. Questo consente generalmente di partire da un grosso problema complesso ed arrivare ad avere un molti problemi di complessità minore e quindi risolvibili con semplici algoritmi.
  Esempio:



DOPO:

- **Testing:** tale operazione è molto delicata, può essere che un software funzioni perfettamente nelle condizioni standard ma che con particolari dati di input si oggetto di un malfunzionamento. Per questo motivo va predisposta una accurata serie di test con diversi valori dei dati di input per rilevare i bug. Sarebbe preferibile che questa fase venisse svolta da personale diverso rispetto a quello che ha realizzato il sw.

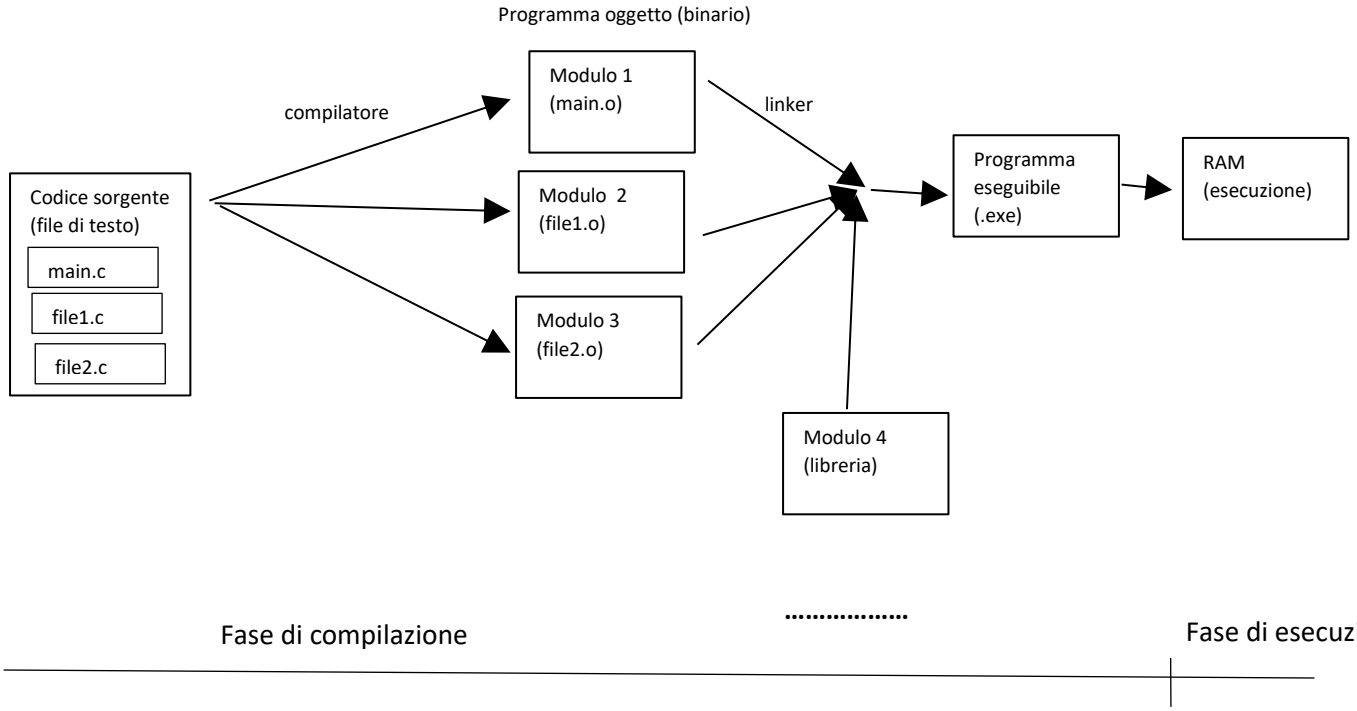
- **Documentazione:** quando è stato realizzato un software è sempre necessario scrivere un **manuale utente** che fornisca le informazioni sull'utilizzo del software, ed un **manuale tecnico** che fornisca una descrizione della struttura e delle funzionalità del software per eventuali modifiche successive anche realizzate da persone diverse da quelle che hanno prodotto il sw.

- **Manutenzione:** Un software è spesso soggetto a manutenzione sia per correggere eventuali malfunzionamenti emersi durante il suo utilizzo (**manutenzione correttiva**) sia per aggiungere eventuali nuove funzionalità (**manutenzione evolutiva**).

4. COMPILATORI ED INTERPRETI

Abbiamo detto che la traduzione del codice sorgente in un programma eseguibile (binario) avviene per mezzo del compilatore e del linker. In realtà non è sempre vero, infatti ci sono tre diverse modalità per ottenere la traduzione in binario: l'approccio che usa i software compilatori, l'approccio che usa i software interpreti, l'approccio Java. Analizzeremo i primi due approcci e poi faremo un confronto, infine analizzeremo l'approccio java perché è una via di mezzo fra i due.

Compilatori (approccio compilato)



Il compilatore traduce **tutte le istruzioni** del codice sorgente (main.c e altri file con estensione.c) formando i moduli del programma oggetto. Nel caso siano presenti **errori di sintassi il compilatore non genera il codice oggetto, segnala gli errori** ed obbliga il programmatore a correggere il codice (come abbiamo visto in Codeblocks).

Quando non ci sono errori di sintassi nel codice sorgente, il compilatore genera il programma oggetto. A questo punto interviene il linker il quale collega i vari moduli e anche i file binari contenente le funzioni delle librerie, formando così il programma eseguibile.

Il collegamento delle librerie da parte del linker può avvenire in due modalità:

linking in modalità statica (compile time): il linker collega tuti i moduli (main e librerie) una sola volta al termine della compilazione e genera un unico file binario (il file eseguibile) che contiene tutte le istruzioni binarie (sia quelle del codice sorgente sia quelle delle librerie) necessarie affinché il processore possa eseguire il programma. Questo file, in fase di esecuzione, verrà caricato nella memoria centrale dal SO e il processore ne eseguirà le istruzioni. **Il file eseguibile contiene la**

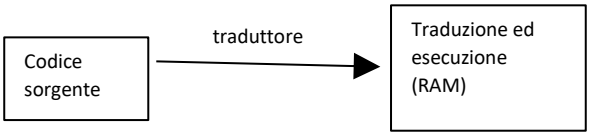
traduzione binaria di TUTTO il codice, comprese le librerie.

linking in modalità dinamica (run-time): il codice oggetto viene direttamente caricato dal SO nella memoria centrale **senza** il codice binario delle librerie (il linker aggiunge al codice oggetto dei **riferimenti** alla libreria in cui si trovano le funzioni. Le cosiddette DIPENDENZE). Le istruzioni vengono eseguite dal processore. Solo quando, durante l'esecuzione, all'interno del codice binario, vi sono istruzioni che richiamano le funzioni di una libreria, il linker provvede a caricare nella memoria il codice binario **della funzione richiesta** e ad aggiungere le istruzioni che "indicano" al processore in quale zona della RAM è stata caricata la funzione. **Nei SO windows** le librerie collegate dinamicamente sono chiamate **DLL** (Dinamic Link Library), nei **SO** le librerie a collegamento dinamico hanno estensione **.so (Shared Object**).
Con il linking dinamico quindi il file eseguibile non contiene il codice delle funzioni delle librerie ma solamente dei "riferimenti" a tali funzioni. Il codice delle funzioni delle librerie sarà caricato "al bisogno" durante la fase di run (esecuzione) del programma.
Questo approccio consente di generare file eseguibili di dimensioni inferiori rispetto al collegamento statico perché le funzioni vengono caricate in memoria centrale e collegate solamente quando, e se, necessario.

Il vantaggio del linking dinamico: nel caso in cui una libreria venisse modificata (ad esempio per la correzione di un bug) non è necessario ricompilare tutti programmi che la utilizzano, basta sostituire la libreria. Nell'approccio statico invece, se una libreria contenente un bug è stata utilizzata in un software, tale software dovrà essere ricompilato con la nuova libreria per poter correggere il bug.
Il vantaggio del linking statico: il file eseguibile non ha dipendenze e può funzionare su qualsiasi piattaforma per il quale è stato compilato senza la necessità che vengano installate nel pc le librerie. Nell'approccio statico invece il computer che esegue il codice deve aver installato le apposite librerie affinché esse possano essere richiamate.
Attualmente è più utilizzato il linking dinamico.

Esempi di linguaggi di programmazione compilati sono C, C++, Java, Visual Basic, C#.

Interpreti (approccio interpretato)



L'interprete è un software che traduce in binario il codice sorgente ma agisce in maniera diversa rispetto al compilatore. L'interprete traduce in binario **istruzione per istruzione durante l'esecuzione**, l'interprete è un software che legge l'istruzione (l'istruzione per lui è un dato), la converte in binario e "dice" al processore di eseguirla. La traduzione avviene dunque contemporaneamente all'esecuzione ed avviene istruzione per istruzione. Un esempio di linguaggio interpretato è il PHP.

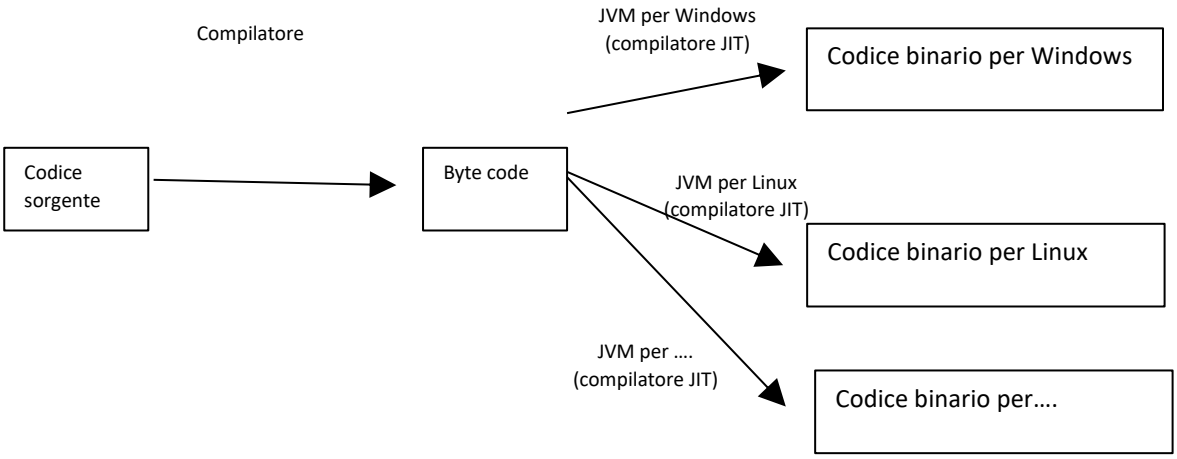
Confronto:

Codice compilato	Codice interpretato
Il codice eseguibile è più veloce durante l'esecuzione poichè ogni istruzione è già in binario, non deve essere tradotta durante l'esecuzione ma è già in linguaggio macchina.	È più lento durante l'esecuzione, ad esempio se c'è un ciclo che ripete 1000 volte un'istruzione essa viene tradotta 1000 volte
E' più lento durante la fase di debug ogni volta che viene individuato e corretto un errore è necessario ricompilare ed eseguire nuovamente il programma per verificare la correttezza della modifica (se il codice è lungo la compilazione può impiegare molto tempo)	Più veloce in fase di debug perché è possibile modificare il codice durante il debug, ciò consente di vedere immediatamente il risultato di una modifica del codice.
Una volta compilato non ha sicuramente errori di sintassi	Poiché non tutte le istruzioni vengono sempre eseguite in ogni esecuzione (ad esempio negli if), può darsi che rimangano "nascosti" errori di sintassi che si potrebbero presentare dopo molte esecuzioni corrette. Per ovviare a questo problema molti interpreti oggi mettono a disposizione un compilatore per evitare gli errori di sintassi.
Non è necessario che il computer che esegue il programma abbia installato il compilatore.	Affinchè possano essere tradotte ed eseguite le istruzioni è necessario che venga installato il software traduttore sul computer che utilizza il programma.

Soluzione Java

Java ha ideato una terza strada per la traduzione in binario del codice sorgente. La soluzione Java rende estremamente **portabili** i software scritti con questo linguaggio di programmazione. L'approccio è una via di mezzo fra quello compilato e quello interpretato. Il compilatore java non genera un programma oggetto specifico per una determinata piattaforma ma un codice chiamato **byte-code**. Il byte-code può essere interpretato da una "macchina virtuale" chiamata **JVM** (Java Virtual Machine). Per ogni piattaforma SW (ossia per ogni sistema operativo) esiste una JVM da installare, tale JVM potrà interpretare il byte-code di qualunque software scritto in linguaggio JAVA. La JVM rappresenta quindi una astrazione per qualunque piattaforma HW/SO. L'idea è ben rappresentata dal motto: *"write once, run everywhere"*.

Quando il codice deve essere eseguito dal processore, il byte-code viene convertito in codice binario dalla JVM, ma la traduzione riguarda solamente alcune parti di bytecode, ossia le funzioni che vengono effettivamente utilizzate, e tale traduzione avviene solamente nel momento in cui le funzioni vengono richiamate (questa modalità è detta compilazione Just In Time, **JIT**, "appena in tempo") similmente a quanto avviene con un software interprete. Le funzioni che vengono tradotte in binario (compilate in codice nativo si dice) rimangono poi nella RAM per eventuali successivi utilizzi durante l'esecuzione, ma se alcune funzioni non vengono mai invocate durante l'esecuzione, esse non vengono tradotte dal byte code in binario. Questo consente di non dover compilare (con la compilazione JIT) parti di codice non utilizzate, velocizzando l'esecuzione. Le prestazioni in termini di velocità di esecuzione comunque non saranno mai performanti al pari di un programma eseguibile compilato nativamente (ad esempio in C o C++) a causa dell'ulteriore livello di astrazione introdotto dalla **JVM**.



Approccio Bytecode di Microsoft

Anche Microsoft ha adottato, per diverse tipologie di linguaggi, un approccio simile a quello di JAVA ossia basato su un bytecode e su una macchina virtuale. Questa tecnologia prende il nome di tecnologia **.NET.** In questa tecnologia la macchina virtuale è chiamata **CLR (Common Language Runtime),** il bytecode è chiamatao **CIL (Common Intermediate Language)** Esistono versioni di questa tecnologia per tutte le piattaforme Windows e dal 2015 anche per Linux e MacOS . Tale tecnologia è chiamata .NET Core (si legge dot net core).

5. PARADIGMI DI PROGRAMMAZIONE

Si definisce paradigma di programmazione l'insieme degli **strumenti concettuali** messi a disposizione di un determinato linguaggio di programmazione per la scrittura del codice di un programma. Paradigmi diversi portano a realizzare software con approcci diversi da parte del programmatore.

Programmazione imperativa o procedurale: (è quella che stiamo usando noi) ci sono delle istruzioni che agiscono su delle variabili. Le variabili rappresentano un'astrazione delle celle di memoria. L'insieme dei valori di tutte le variabili in un determinato istante forma lo **stato** del programma. Ogni istruzione provoca, in base allo stato attuale, un cambiamento di stato. Le istruzioni sono, generalmente devi verbi al tempo imperativo (scrivi, acquisisci, leggi..). La struttura del programma è formata da programmi che richiamano altri sottoprogrammi (procedure o funzioni). Esempi: C, Pascal.

Programmazione funzionale: Esempi: LISP, F#.

Programmazione logica: Esempio: prolog.

Programmazione ad oggetti: (è quella che vedremo l'anno prossimo) si basa sulla definizione di oggetti software in grado di interagire gli uni con gli altri.