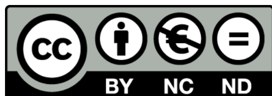
Laboratorio di Programmazione di Sistema

Traduzione

Luca Forlizzi, Ph.D.

Versione 20.0



Luca Forlizzi, 2020

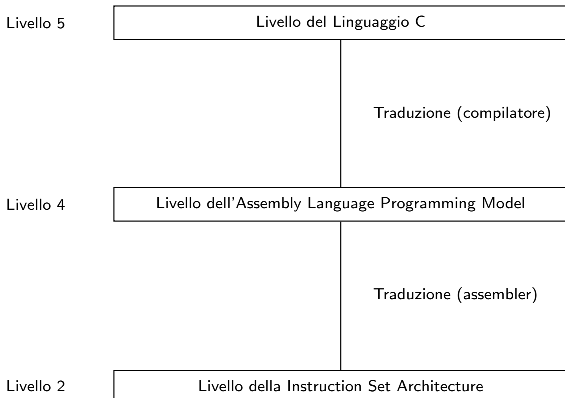
© 2020 by Luca Forlizzi. This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit
https://creativecommons.org/licenses/by-nc-nd/4.0/legalcode.

Traduzione

- In questa presentazione approfondiamo il processo di traduzione di un programma C
  - Traduzione da *ASM* a *LM*
  - Traduzione in presenza di un Sistema Operativo

Traduzione

- Nelle precedenti presentazioni, si è fatto riferimento, in prevalenza, ad un modello di computer semplificato composto dai soli livelli 2, 4 e 5



Traduzione

- In tale modello semplificato, la traduzione di un programma scritto in un *HLL* segue il seguente schema
  1. Il codice sorgente, viene tradotto in un *ASM*, ottenendo un codice sorgente *ASM*
  2. Il codice sorgente *ASM*, viene tradotto in un *LM*, ottenendo un insieme di dati e istruzioni che vengono caricati in parole di memoria del computer
- La traduzione da *HLL* a *ASM* viene studiata approfonditamente in *Compilatori*
- Spesso i traduttori degli *HLL*, utilizzano un *ASM* che non è legato ad una specifica *ISA*, ma è leggermente più astratto
- In questa presentazione, invece, ci concentriamo sulla traduzione da *ASM* a *LM*
- Lo scopo, infatti è approfondire la conoscenza del livello 2

ASM-PM e *ISA*

- Una *ISA*, definisce una abstract machine in grado di eseguire istruzioni di un *LM*, e ne descrive la semantica
- Un *ASM-PM* A_{ASM} viene detto *astrazione* di una *ISA* A_{ISA}, se valgono se seguenti condizioni
  - per ogni registro definito da A_{ASM}, esistono uno o più registri di A_{ISA} che contengono la stessa quantità di bit e che hanno le stesse proprietà
  - A_{ASM} e A_{ISA} definiscono lo stesso insieme di indirizzi e lo stesso insieme di byte
  - per ogni formato di dato di A_{ASM}, A_{ISA} definisce un formato di dato con le stesse proprietà
  - Per ogni istruzione I definita da A_{ASM}, esiste una sequenza di istruzioni S_I di A_{ISA}, detta *traduzione in LM* di I, tale che gli effetti prodotti da S_I sullo stato dei registri e della memoria della abstract machine di A_{ISA}, sono gli stessi che l'esecuzione di I produce sullo stato e la memoria della abstract machine di A_{ASM}

ASM-PM e *ISA*

- Il fatto che A_{ASM} sia un'astrazione di A_{ISA}, implica che la abstract machine definita da A_{ASM} sia molto simile a quella definita da A_{ISA}
- I programmi per l'*ASM* di A_{ASM} vengono normalmente eseguiti mediante traduzione nel *LM* di A_{ISA}
- Pertanto si dice che l'*ASM* di A_{ASM} è un *ASM per* A_{ISA}
- Di solito, se un *ASM-PM* è un'astrazione di una determinata *ISA*, le due architetture vengono chiamate con lo stesso nome

ASM e *LM*

- Ciò che differenzia sostanzialmente gli *ASM* dai *LM*, è la natura delle istruzioni
- Le istruzioni di un *ASM*, sono del testo, composto da caratteri alfanumerici e simboli, che forma il codice sorgente
- Le istruzioni di un *LM*, esistono di fatto come entità software, memorizzate in un computer
- Ogni istruzione di un *LM* è una stringa binaria memorizzata in un gruppo di bit (generalmente in una parola di memoria)
- Quindi la sintassi delle istruzioni di un *LM* è notevolmente diversa da quella delle istruzioni di un *ASM*

ASM e *LM*

- Di solito, nei *LM* non esiste il concetto di programma come insieme ben delimitato di istruzioni
- La abstract machine definita da una *ISA*, in un modo di funzionamento di tipo Run, esegue continuamente istruzioni, e non vi è una demarcazione rigida che stabilisca una separazioni tra le istruzioni di due diversi programmi

ASM e *LM*

- La definizione della relazione di astrazione tra un *ASM-PM* A_{ASM} e una *ISA* A_{ISA}, la traduzione di un'istruzione I definita da A_{ASM}, è, in generale, una sequenza di istruzioni S_I di A_{ISA}
- Oltre alla diversa sintassi, esiste quindi un'altra importante differenza tra un istruzione *ASM* e la sua traduzione in *LM*: la traduzione in *LM* di una singola istruzione *ASM*, può essere costituta da una sequenza di più istruzioni *LM*
- In un linguaggio *ASM*, distinguiamo quindi due insiemi di istruzioni, in base al numero di istruzioni *LM* che ne costituiscono la traduzione
  - Un'istruzione *ASM* è detta *base* (*basic instruction*) se la sua traduzione è una singola istruzione in *LM*
  - Un'istruzione *ASM* che non è un'istruzione base, viene detta *pseudoistruzione* o *assembly idiom*

ASM e *LM*

- Poiché un'istruzione *ASM* e la sua traduzione producono effetti equivalenti (ciascuna nella relativa architettura), e poiché la traduzione di un'istruzione base di un *ASM* è una singola istruzione *LM*, un'istruzione *ASM* e l'istruzione che la traduce sono semanticamente equivalenti
- Quindi un'istruzione base permette di esprimere, con la sintassi di un *ASM*, l'effetto di una singola istruzione del *LM*

ASM e *LM*

- Le pseudoistruzioni vengono definite allo scopo di rendere più facile e comoda la programmazione in *ASM*
- Il prezzo da pagare è che usando pseudoistruzioni si rende più astratta la semantica, nascondendo alcuni dettagli che possono avere un impatto sulla correttezza o sulle prestazioni del programma
- Di solito, comunque, le pseudoistruzioni (diversamente dai comandi di un *ASM-API*) hanno traduzioni formate da sequenze formate da poche istruzioni *LM* (meno di 10)

ASM e *LM*

- Si osservi che, contrariamente a una diffusa credenza, non vi è una corrispondenza uno a uno tra istruzioni di un *ASM* per una *ISA*, e istruzioni del *LM* della stessa *ISA*
- Ciò accade non solo per la presenza delle pseudoistruzioni
- Infatti può accadere che differenti istruzioni *ASM*, anche base, si traducano nella stessa istruzione *LM*: si tratta, sostanzialmente, di sintassi alternative per denotare la stessa istruzione
- Un esempio in MC68000 sono le istruzioni `br` e `bra`, un esempio in MIPS32 sono, in alcuni casi, le istruzioni `or` e `li`

ASM e *LM*

- Inoltre può accadere anche che una stessa istruzione *ASM* si traduca in più modi diversi; ciò può accadere anche ad un'istruzione base, purché ciascuna delle possibili traduzioni sia una singola istruzione del *LM*
- Un esempio in MC68000 è l'istruzione base `add` che si traduce, in base al formato del primo operando, in due diverse traduzioni, ciascuna formata da una singola istruzione *LM* (ovvero quando il primo operando di `add` è un immediato, la sua traduzione *LM* è diversa rispetto agli altri casi; inoltre l'*ASM* MC68000 ha anche l'istruzione `addi` che è una sintassi alternativa per il caso di `add` con primo operando immediato)

Basic Assembly

- Sia L il linguaggio *ASM* di un *ASM-PM* A_{ASM}, che è astrazione di una *ISA* A_{ISA}
- Allora L viene detto *basic assembly* per A_{ISA} se valgono le seguenti condizioni
  - per ogni registro definito da A_{ASM}, esiste un registro definito da A_{ISA} che contiene la stessa quantità di bit e che ha le stesse proprietà
  - per ogni registro definito da A_{ISA}, esiste un registro definito da A_{ASM} che contiene la stessa quantità di bit e che ha le stesse proprietà
  - ogni istruzione di L è una basic instruction
  - per ogni istruzione I_{ISA} del *LM* di A_{ISA}, esiste un'istruzione I_{ASM} di L, tale che I_{ISA} è la traduzione di I_{ASM}
  - ogni sequenza S di istruzioni di L, viene tradotta dalla sequenza di istruzioni del *LM* di A_{ISA} formata concatenando nello stesso ordine le traduzioni delle istruzioni che formano S

Basic Assembly

- La definizione precedente implica che attraverso un basic assembly *L* vengono esposti tutti i registri e tutte le istruzioni di una *ISA*
- Inoltre, un programma P_{ASM} per *L* è estremamente simile alla sua traduzione P_{LM} nel *LM* della *ISA*
  - ogni istruzione di *L* viene tradotta in una singola istruzione della *ISA*
  - P_{ASM} è formato da una sequenza di istruzioni che hanno lo stesso ordine che le rispettive traduzioni in *LM*hanno nella sequenza di istruzioni che forma P_{LM}
- Ciò rende possibile usare un linguaggio *ASM* come uno strumento per scrivere e leggere, con una sintassi più comoda per un essere umano, programmi *LM*, e quindi per utilizzare e studiare tutte le caratteristiche di una *ISA*

Basic Assembly

- Per *ISA* dotate di istruzioni relativamente potenti, come le *ISA* CISC, è comune definire dei basic assembly
- Ad esempio MC68000-ASM1 è un basic assembly
- Al contrario, per le *ISA* RISC, dotate di istruzioni più semplici e meno potenti, si tende a definire degli *ASM* dotati di pseudoistruzioni, in modo da compensare lo svantaggio di avere istruzioni meno potenti, che comporta la necessità di fare più lavoro ai programmatori
- MIPS32-MARS, infatti, non è un basic assembly

Istruzioni Base e Pseudoistruzioni in MIPS32-MARS

- È utile sapere quali istruzioni sono base e quali sono pseudoistruzioni
- In linea di massima, ciò dipende dai modi di indirizzamento degli operandi
- Nella maggior parte dei casi, sono istruzioni base
  - istruzioni con 3 operandi registro
  - istruzioni con un operando immediato in formato `half`
  - istruzioni con un operando in memoria che accedono all'operando con il modo di indirizzamento indiretto-registro o indicizzato con offset corto (ovvero l'offset è un valore in formato `half`)
  - le istruzioni di salto incondizionato
  - le istruzioni a 3 operandi condizionate `slt`, `slti`, `sltu`, `sltiu`, `beq`, `bne` (le ultime due solo nel caso in cui il secondo operando sia un registro)

Istruzioni Base e Pseudoistruzioni in MIPS32-MARS

- In MIPS32-MARS, in accordo con la convenzione stabilita in **[MIPS]**, le pseudoistruzioni modificano il registro 1
- Non tutte le istruzioni con 3 operandi registro o con un operando immediato in formato `half`, sono istruzioni base
  - `div` con 3 operandi
  - `rem`
  - `sub` e `mul` con un operando immediato

Istruzioni Base e Pseudoistruzioni in MIPS32-MARS

- La traduzione *LM* di pseudoistruzioni che hanno un operando immediato *v* in formato `word`, avviene "costruendo" in un registro (di solito il registro `1`) il valore *v*
- La costruzione utilizza un'istruzione di shift, oppure l'istruzione base `lui`
  - `li $t0,0x10000000`
  - `la $t0,label`
  - `add $t0,$t1,0x10000000`

Istruzioni Base e Pseudoistruzioni in MIPS32-MARS

- Traduzione *LM* di pseudoistruzioni che usano il modo di indirizzamento diretto-memoria o indicizzato con offset lungo (formato `word`) per accedere alla memoria
  - `lw $t1,label`
  - `sh $t1,0x10000000($t2)`

Istruzioni Base e Pseudoistruzioni in MIPS32-MARS

- La maggior parte delle istruzioni condizionate b**cc** e s**cc** sono pseudoistruzioni
  - `beq $t1,10,label`
  - `ble $t1,$t3,label`
  - `sge $t2,$t5,$t6`

Salti ritardati e riordino delle istruzioni

- Sveliamo ora una caratteristica di MIPS32, nota informalmente come *salto in ritardo*
- Contrariamente a quanto detto o assunto in precedenti presentazioni ed esempi, MIPS32 prevede che quando si esegue un'istruzione di salto *J*, l'eventuale salto alla istruzione di destinazione avvenga non al termine dell'istruzione di salto, ma dopo aver eseguito l'istruzione successiva a *J* in ordine di allocazione

Salti ritardati e riordino delle istruzioni

- Il *salto in ritardo* consente di semplificare, e quindi rendere più efficienti, le implementazioni hardware di MIPS32
- Purtroppo rende molto scomoda la programmazione in *ASM*
- Fortunatamente MARS, che è una implementazione software di MIPS32, permette di disabilitare tale caratteristica
- In LPS si utilizza questa possibilità a scopo didattico, per facilitare lo studio della programmazione *ASM*

Salti ritardati e riordino delle istruzioni

- I linguaggi *ASM* destinati a produrre codice eseguibile per implementazioni hardware di MIPS32 (in cui ovviamente non si può disabilitare il salto in ritardo) provano a semplificare la programmazione "nascondendo" la caratteristica con due tecniche
  - inserimento automatico di istruzioni `nop` subito dopo un'istruzione di salto
  - riordino della sequenza delle istruzioni, in modo da spostare in modo automatico, un'istruzione che deve essere eseguita prima che avvenga il salto, subito dopo l'istruzione di salto
- Gli *ASM*che utilizzano queste tecniche, anche se non usano pseudoistruzioni, non si qualificano come basic assembly

Il linguaggio macchina MIPS32

- Tutte le istruzioni del *LM* di MIPS32 sono stringhe di 32 cifre binarie
- Un aspetto che caratterizza le *ISA* RISC realizzate negli anni 80 e 90, rispetto alle *ISA* CISC è il fatto che tutte le istruzioni hanno la stessa lunghezza
- Molte istruzioni MIPS32 hanno sintassi che si conformano ad uno dei due *formati generali* delle istruzioni MIPS32
- Ciascuno dei formati generali accomuna istruzioni che hanno stesso numero e tipologia di operandi

Il linguaggio macchina MIPS32

- Il formato generale **R** è usato da istruzioni che hanno 3 operandi registro

numero di cifre binarie	6	5	5	5	5	6
nome campo	op	rs	rt	rd	shamt	funct

Il linguaggio macchina MIPS32

- Il formato generale **I** è usato da
  - istruzioni che hanno un operando immediato in formato `half`
  - istruzioni che usano il modo di indirizzamento indicizzato con offset in formato `half`
  - le istruzioni di salto condizionato, le quali usano l'indirizzamento PC-indicizzato per l'istruzione di destinazione del salto

numero di cifre binarie	6	5	5	16
nome campo	op	rs	rt	value

Il linguaggio macchina MIPS32

- Le istruzioni `j` e `jal` usano entrambe l'indirizzamento diretto-memoria per specificare una parte dell'indirizzo di destinazione del salto, e pertanto hanno sintassi simili

numero di cifre binarie	6	26
nome campo	op	value

Il linguaggio macchina MC68000

- Le istruzioni del *LM* di MC68000 sono stringhe che formate da 1 fino a 5 sotto-sequenze di 16 cifre binarie
- È comune a molte *ISA* CISC il fatto che istruzioni diverse abbiano lunghezze diverse
- Diversamente da quanto accade in MIPS32, una istruzione MC68000 può indicare, per ciascuno dei suoi operandi, diversi modi di indirizzamento

Il linguaggio macchina MC68000

- Mostriamo come esempio la sintassi dell'istruzione `move`

numero di cifre binarie	2	2	3	3	3	3
nome campo	00	size	rd	md	ms	rs

- I campi md e ms indicano il modo di indirizzamento da usare per i due operandi
- La specifica di alcuni modi di indirizzamento, aggiunge sotto-sequenze di 16 cifre alla sintassi
  - si aggiunge 1 sotto-sequenza, per i modi di indirizzamento indicizzato, base-indicizzato, immediato in formato `word` e diretto-memoria corto
  - si aggiungono 2 sotto-sequenze, per i modi di indirizzamento immediato in formato `long` e diretto-memoria lungo

Generazione di Codice su File

- Generalmente, nelle implementazioni di C Standard che operano in un ambiente supportato da un sistema operativo, non è possibile stabilire al momento della traduzione, gli indirizzi di memoria in cui il codice eseguibile deve essere caricato
- In questi ambienti, la traduzione di un programma scritto in un *HLL* segue il seguente schema
  1. Il codice sorgente, viene tradotto in un *ASM*, ottenendo un codice sorgente *ASM*
  2. Il codice sorgente *ASM*, viene tradotto parzialmente in un *LM*, ottenendo un insieme di dati e istruzioni privi di tutti i riferimenti a indirizzi di memoria; tale traduzione viene memorizzata in un file, chiamato `file eseguibile`

Generazione di Codice su File

- L'esecuzione del programma avviene caricando il contenuto del file eseguibile nella memoria principale e completando la traduzione con i riferimenti agli indirizzi
- Il completamento degli indirizzi può avvenire subito prima (*rilocazione statica*) o durante (*rilocazione dinamica*) l'esecuzione

Traduzione Separata

- Molti linguaggi consentono di suddividere il codice sorgente di un programma, in diverse parti che possono essere tradotte, o parzialmente tradotte, indipendentemente le une dalle altre
- In questi casi, di solito, ciascuna parte del programma viene memorizzata in un file indipendente; ad esempio in Java ogni classe pubblica è definita in un diverso file

Traduzione Separata

- Suddividere un programma in diverse parti, comporta vantaggi importanti:
  - **Efficienza**: Durante la scrittura e il debug di un programma, capita sovente di dover tradurre numerose volte il programma; la possibilità di compilare separatamente ciascuna parte, consente di non dover tradurre ogni volta tutto il programma, ma solo le parti che sono state modificate; ciò si traduce in un notevole risparmio di tempo
  - **Struttura**: La suddivisione in parti del programma può ricalcare la struttura logica del programma, rendendola immediatamente evidente agli sviluppatori
  - **Riuso**: Racchiudere una parte di programma in un file, ne facilita il riuso in un altro programma; il riuso può avvenire anche utilizzando solo la forma tradotta della parte che si vuole utilizzare, nel caso non si voglia diffonderne il codice sorgente

Traduzione Separata

- La traduzione separata fu introdotta sin dai primissimi compilatori C, all'inizio degli anni '70, quando non erano ancora evidenti i benefici in termini di **Struttura** e **Riuso**
- Viceversa, come sappiamo, a causa dei limiti tecnologici e dei requisiti che si voleva che il C soddisfacesse, si era estremamente interessati a realizzare traduttori efficienti e semplici da implementare

Traduzione Separata

- Conseguentemente, la traduzione separata fu realizzata attraverso un approccio finalizzato a ottenere vantaggi in termini di **Efficienza**
- Ciò non vuol dire che la traduzione separata in C non apporti anche vantaggi in termini di **Struttura** e **Riuso**; tuttavia i vantaggi in termini di **Struttura** e **Riuso** sono minori di quelli che avrebbero potuto esserci se questi aspetti fossero stati tenuti in considerazione, come invece è stato fatto in altri linguaggi (ad esempio Java)
- Inoltre, la traduzione separata in C comporta anche degli svantaggi, soprattutto per il programmatore, in termini di checking effettuati durante la traduzione e comodità di utilizzo

Traduzione Separata

- C Standard, per ragioni di backward compatibility, ha definito le regole del linguaggio relative alla traduzione separata sulla linea di quelle impiegate nei traduttori in uso alla fine degli anni 70
- All'epoca, anche a causa del fatto che la prima edizione di **[K&R]** descrive la traduzione separata in modo non completamente dettagliato, differenti traduttori avevano adottato regole diverse tra loro
- Dunque in C Standard sono state definite delle regole piuttosto complesse per fare in modo che risulti valido (o richieda poche modifiche per diventarlo) codice scritto usando uno dei principali traduttori dell'epoca
- La conseguenza è che le regole sulla traduzione separata sono molte e non armonizzate, e risultano non semplici da capire e ardue da ricordare

Il Processo di Traduzione

- La maggior parte dei traduttori C adottano la compilazione, tanto che lo Standard C, pur prevedendo la possibilità di tecniche di traduzione diverse, descrive i compiti che devono essere effettuati dal traduttore utilizzando un modello teorico di compilazione, suddiviso in 8 fasi
- Tuttavia, nella terminologia specifica del linguaggio C il termine compilazione non si riferisce all'intero processo di traduzione, ma solo ad una parte (per rimarcare la differenza, parleremo talvolta di *compilazione in senso stretto*)
- Analogamente, il termine compilatore, nella terminologia del C, si riferisce a quella parte del traduttore che esegue la compilazione in senso stretto

Il Processo di Traduzione

- Dal punto di vista del programmatore, il processo di traduzione può essere efficacemente schematizzato come una compilazione che avviene attraverso 3 fasi:
  1. *Preprocessing*: Il codice sorgente viene suddiviso in token, i commenti vengono cancellati e vengono eseguite le *direttive*; il risultato è un codice sorgente modificato
  2. *Compiling* (in senso stretto): Il codice sorgente viene tradotto in codice oggetto
  3. *Linking*: Vengono effettuati i collegamenti tra varie parti del programma e tra il programma e il run-time support; il risultato è il codice eseguibile del programma
- Il processo di traduzione definito dallo Standard, permette la traduzione separata, in quanto ogni file sorgente può essere tradotto (ovvero preprocessato e poi compilato) in codice oggetto individualmente e indipendentemente dagli altri

Preprocessing

- Il preprocessing viene eseguito da un software chiamato *preprocessor*; agli albori della storia del C, spesso il preprocessor era un software separato dal compilatore, ma al giorno d'oggi è invece sempre integrato; molti traduttori consentono comunque di produrre il risultato del preprocessing
- La separazione (teorica) tra preprocessing e compilazione è il motivo delle differenze sintattiche e stilistiche tra le direttive del preprocessore e il resto del linguaggio

Preprocessing

- Il preprocessing di un file sorgente è indipendente da quello degli altri file sorgente, e può quindi essere eseguito in momenti diversi
- Il prodotto del preprocessing di un singolo file sorgente viene chiamato *translation unit* (abbreviato con TU)
- Le operazioni effettuate durante il preprocessing consistono in alterazioni del testo del programma che potrebbero essere effettuate anche “a mano”

Compiling

- Il compiling viene eseguito da un software chiamato *compiler*
- Di solito, il compiling viene eseguito su translation unit prodotte dal preprocessor, ma potrebbe anche essere eseguito direttamente su codice sorgente scritto dal programmatore, purchè esso non contenga direttive per il preprocessore
- Il compiling di una TU è indipendente da quello di altre, e può quindi essere eseguito in momenti diversi
- Il prodotto del compiling di una TU è il codice oggetto ottenuto traducendo in linguaggio macchina il codice sorgente contenuto nella TU

Compiling

- Il prodotto della compilazione di una o più TU viene memorizzato in uno o più file oggetto: alcuni compilatori producono un file oggetto per ciascuna TU, altri uniscono in un unico file, il codice oggetto ottenuto traducendo più TU
- *Compile-time*: il tempo durante il quale un programma (o una TU) viene compilato
- La maggior parte dei checking effettuati dai traduttori del C avviene a compile-time

Linking

- Il linking viene eseguito da un software chiamato *linker*; in alcuni traduttori il linker e il compiler sono fusi in un unico programma, ma in altri sono software separati; un linker separato potrebbe essere in grado di operare con file oggetto prodotti da differenti compiler
- Il linking opera sui file oggetto prodotti dalla compilazione e li collega tra di loro, con il run-time support del traduttore e con eventuali librerie, prodotte da terze parti, utilizzate dal programma

Linking

- Per eseguire il linking, il linker deve avere a disposizione tutti i file oggetto del programma e tutte le parti del run-time support e delle librerie ad esso necessarie
- Il prodotto del linking è il codice eseguibile del programma, che viene memorizzato di solito memorizzato in un unico file eseguibile

Linking

- A partire dallo stesso codice oggetto, possono essere prodotti differenti codici eseguibili, modificando opportune opzioni di controllo; ad esempio è possibile collegare lo stesso codice oggetto con differenti versioni del run-time support, allo scopo di ottenere versioni dell'eseguibile con differenti caratteristiche (maggior efficienza temporale, minor consumo di memoria, compatibilità con un maggior numero di sistemi, . . .)
- *Link-time*: il tempo durante il quale viene fatto il linking di un programma
- I traduttori C eseguono alcuni checking a link-time, ma di solito sono meno importanti ed efficaci di quelli effettuati a compile-time