Capitolo 10

Struttra dei programmi e strumenti di sviluppo (bozze, v. 1.0)

10.1 Prerequisiti

La seguente trattazione ha carattere riepilogativo rispetto a molti argomenti oggetto del corso di Elementi di Informatica. In particolare è richiesta la conoscenza approfondita dei seguenti argomenti:

- Algoritmo ed esecutore;
- Linguaggio di programmazione e corrispondente macchina astratta;
- Segmento di programma, variabili di ingresso e uscita di un segmento;
- Sottoprogrammi e meccanismi di passaggio dei parametri di scambio;
- Procedimento di traduzione dal programma sorgente al programma eseguibile: proeprocessore, compilatore, collegatore;
- Architettura di Von Neumann e struttura del linguaggio macchina.

10.2 Struttura dei programmi

Allo scopo di risolvere un problema attraverso l'esecutore associato ad un linguaggio di programmazione, l'algoritmo risolutivo deve essere formulato in ogni dettaglio, fino a giungere ad un programma completo secondo le regole del linguaggio scelto.

Nel caso il problema da risolvere richieda l'impiego di un algoritmo complesso o di un numero elevato di algoritmi cooperanti, il programma risultante è formato da un numero molto elevato di istruzioni. Per mantenere la complessità del programma entro limiti accettabili, la sua stesura richiede pertanto il ricorso a forme di strutturazione che consentano di dividere il codice in parti che possano essere manipolate con una certa indipendenza.

Si possono distinguere due aspetti della strutturazione di un programma. Il primo può essere indicato col termine *strutturazione logica* perché riguarda la divisione in parti dell'*algoritmo* che il programma descrive. La seconda può essere indicato col termine *strutturazione fisica* perché riguarda la divisione in parti del *testo* del programma. Si noti che la strutturazione fisica coincide con la divisione in file del programma.

In generale, la struttura fisica di un programma non coincide con la la sua struttura logica, anche se in parte dipende da essa. Nel seguito riassumeremo pertanto prima i meccanismi fondamentali di strutturazione logica e successivamente illustreremo quelli di strutturazione fisica, mostrando come questi ultimi devono tenere conto della struttura logica del programma.

10.2.1 I sottoprogrammi

Il principale meccanismo di strutturazione logica disponibile in tutti i linguaggi di programmazione è la possibilità di isolare il segmento di codice che descrive un algoritmo in un sottoprogramma.

Nel riassumere come tale meccanismo viene usato per strutturare logicamente un programma complesso, è opportuno soffermarsi sulle relazioni che intercorrono tra il segmento di codice incapsulato in un sottoprogramma e il resto del codice. Tali relazioni sono completamente identificate dai seguenti elementi:

- le variabili di ingresso e di uscita del corpo del sottoprogramma;
- i valori consumati dallo standard input e prodotti sullo standard output dalle istruzioni del corpo del sottoprogramma;
- la descrizione precisa di come vengono assegnati i valori alle variabili di uscita e di come vengono prodotti i valori sullo standard output, a partire dai valori contenuti inizialmente nelle variabili di ingresso e dai valori letti dallo standard input.

Normalmente le variabili di ingresso e di uscita del corpo di un sottoprogramma coincidono con i parametri di scambio (rispettivamente di ingresso e di uscita) del sottoprogramma. Altre variabili di appoggio, definite e usate nel corpo, sono *locali* al sottoprogramma. Tali variabili vengono dichiarate all'interno corpo del sottoprogramma e vengono create e distrutte all'inizio e al termine dell'esecuzione del sottoprogramma stesso. Esse risultano pertanto disponibili solo durante l'esecuzione del sottoprogramma, e corrispondono a variabili diverse in esecuzioni diverse del sottoprogramma.

10.2.2 Le variabili globali

Esistono tuttavia altre variabili che, pur non essendo locali (non sono cioè dichiarate all'interno del sottoprogramma) sono peraltro usate o definite nel corpo. Tali variabili sono ovviamente di ingresso per il corpo se usate prima di essere definite, e possono (e normalmente sono) di uscita se definite.

Queste variabili vengono dette *globali* o *statiche* perché esse possono essere usate o definite da più sottoprogrammi, e perché esse, a differenza di quelle locali, sono disponibili all'esecutore indipendentemente dal fatto che il sottoprogramma che le usa o le definisce sia attivo.

Ad esempio, nel seguente programma, la variabile `valori_letti` è globale, e viene usata dal `main` e usata e definita da `leggi_lista_int`.

```
# include <iostream.h>

void leggi_lista_int ( int lista[], int &n );

int valori_letti = 0;

int main () {
  const int MAX_LEN = 100;
  int L1[MAX_LEN];
  int n_int;

  while ( valori_letti < MAX_LEN )
    leggi_lista_int(L1,n_int);
}

void leggi_lista_int ( int lista[], int &n ) {
  int i;
  cin >> n;
  for ( i=0; i<n; i++ )
    cin >> lista[i];
  valori_letti = valori_letti + n;
}
```

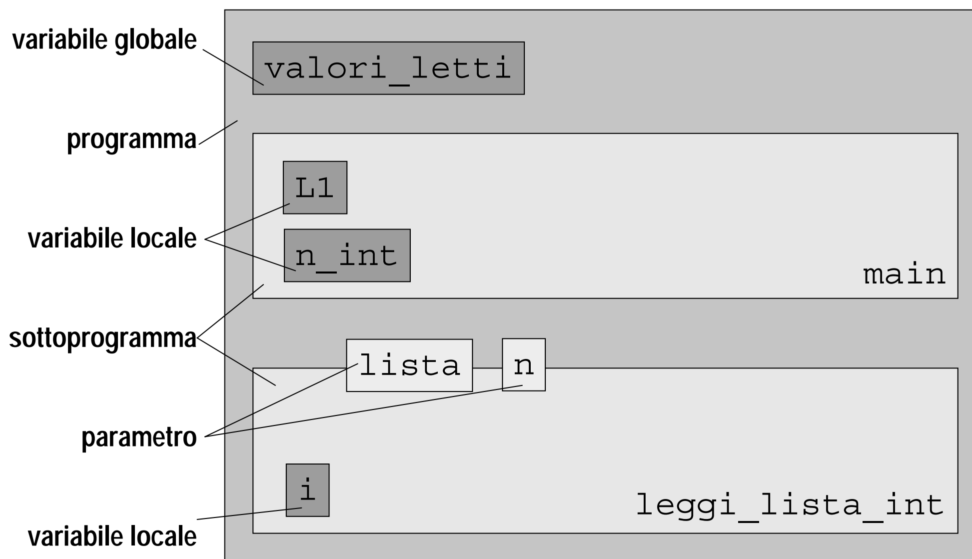


Figura 10.1: Struttura di un programma.

Come mostrato dall'esempio, le variabili globali devono essere dichiarate all'esterno dei sottoprogrammi e per questo motivo esse vengono anche indicate come *esterne* dallo standard del C. Una volta dichiarate esse possono essere usate o definite da tutti i sottoprogrammi che seguono testualmente la dichiarazione.

Naturalmente anche l'impiego delle variabili globali deve rispettare la regola per cui nessuna variabile può essere usata prima di essere stata definita. Questo è il motivo per cui normalmente le variabili globali vengono inizializzate all'atto della loro creazione (usando cioè un inizializzatore nella dichiarazione). Infatti, poiché le variabili globali possono essere usate e definite in sottoprogrammi diversi, il codice che le manipola non è sempre facilmente identificabile, né è facile ricostruire la sequenza dinamica con cui vengono usate o definite. L'inizializzazione, ove possibile, rappresenta pertanto un modo per assicurare il rispetto della regola sopra menzionata. Nell'esempio, l'inizializzazione al valore 0 rende corretto il riferimento a `valori_letti` sia nel `main` che in `leggi_lista_int`.

Le ultime considerazioni evidenziano peraltro la generale difficoltà a controllare la sequenza di valori assunti da una variabile globale e la conseguente difficoltà ad analizzare la correttezza dei programmi che ne fanno uso. E' consigliabile pertanto evitare un uso generalizzato delle variabili globali, limitandolo a quei casi (piuttosto rari) in cui esso risulta assolutamente necessario o in cui rappresenta un chiaro vantaggio rispetto a soluzioni alternative.

10.2.3 Struttura logica di un programma

Per quanto detto, la struttura logica (dell'algoritmo) di un programma consiste pertanto in un insieme di sottoprogrammi e in un insieme di variabili globali.

Ogni sottoprogramma comprende una lista di parametri di ingresso, una lista di parametri di uscita, un insieme di variabili locali e un insieme di variabili globali usate o definite. Come caso particolare, ciascuna lista e ciascun insieme può essere vuoto. Il corpo del programma usa e definisce i parametri e le variabili globali nel rispetto delle regole discusse in precedenza.

In particolare, ogni sottoprogramma interagisce con il resto del programma ricevendo valori nei parametri di ingresso e nelle variabili globali usate, e restituendo valori nei parametri di uscita e nelle variabili globali definite. Ogni sottoprogramma può inoltre interagire con l'ambiente esterno consumando valori dallo standard input o producendo valori sullo standard output.

In figura 10.1 è mostrata in termini grafici la struttura logica del programma riportato inp precedenza.

10.2.4 Organizzazione in file di un programma

La struttura logica illustrata nel paragrafo precedente rappresenta una decomposizione dell'algoritmo in parti e non implica necessariamente anche una suddivisione fisica del codice in più file. In altre parole, il codice di un programma strutturato in sottoprogrammi potrebbe essere interamente contenuto in un unico file. In tal caso la suddivisione logica, pur risultando utile sul piano dell'organizzazione concettuale, non comporterebbe vantaggi sotto il profilo della gestione del codice.

Per consentire invece una reale suddivisione del codice in più file sono disponibili due meccanismi, di natura molto diversa, entrambi molto utili nella gestione del codice di programmi complessi.

Prima di descrivere il funzionamento dei due meccanismi e di discutere come essi contribuiscano alla gestione del codice, è opportuno ricordare il procedimento attraverso il quale un programma formulato in un linguaggio ad alto livello (programma *sorgente*), che fa riferimento ad un esecutore astratto, viene trasformato in un equivalente programma (programma *eseguibile*) adatto ad essere eseguito sull'esecutore reale.

La trasformazione principale da programma sorgente a programma eseguibile è effettuata dal compilatore che traduce le istruzioni del linguaggio ad alto livello in linguaggio macchina. Tuttavia la compilazione è normalmente preceduta da una fase in cui il testo del programma scritto dal programmatore viene modificato da un processore di testo detto preprocessore. In questa fase preliminare non viene fatta alcuna analisi del programma sorgente, né viene effettuata alcuna traduzione, ma, sulla base di direttive, si procede alla sostituzione, all'aggiunta o alla eliminazione di testo. Il testo così prodotto dal preprocessore è quello effettivamente analizzato e tradotto dal compilatore.

La compilazione, inoltre, è seguita da un ulteriore fase in cui il prodotto della compilazione (detto file oggetto) che è codice macchina incompleto viene integrato con altro codice macchina in modo da generare il programma eseguibile. Il programma che effettua tale integrazione è detto collegatore (linker).

La compilazione, per sua natura, prevede la traduzione di *un* testo scritto nel linguaggio sorgente in codice macchina contenuto in *un* file oggetto. Essa pertanto non prevede l'integrazione di file distinti. L'integrazione è invece possibile sia attraverso le funzionalità messa a disposizione dal preprocessore, sia attraverso quelle messe a disposizione dal linker.

Divisione in file mediante la direttiva `#include`

La direttiva del preprocessore `#include` causa la sostituzione della riga di testo che la contiene con l'intero contenuto del file ad argomento della direttiva. Ad esempio, se il file sorgente `prova.cpp` contenesse il seguente testo:

```
# include prototipi.h

int valori_letti = 0;

int main () {
  const int MAX_LEN = 100;
  int L1[MAX_LEN];
  int n_int;

  while ( valori_letti < MAX_LEN )
    leggi_lista_int(L1,n_int);
}
```

e il file `prototipi.h` contenesse la linea:

```
void leggi_lista_int ( int lista, int &n );
```

la compilazione del file `prova.cpp` verrebbe effettuata sul seguente testo prodotto dal preprocessore (invocato automaticamente prima del compilatore):

```
void leggi_lista_int ( int lista, int &n );

int valori_letti = 0;

int main () {
  const int MAX_LEN = 100;
  int L1[MAX_LEN];
  int n_int;

  while ( valori_letti < MAX_LEN )
    leggi_lista_int(L1,n_int);
}
```

Nella sua semplicità l'esempio mostra l'uso della direttiva `#include` per separare il testo del programma (l'ultimo riportato, che è poi quello effettivamente tradotto dal compilatore) in file distinti. Si tratta di una separazione puramente testuale che può essere usata per spezzare in più file il codice da compilare. Solitamente il meccanismo dell'inclusione viene usato per mettere in un file separato le dichiarazioni di tipi, le variabili globali e i prototipi che vanno ripetute in più di un'unità di compilazione (vedi paragrafo successivo), in modo da non doverle scrivere più volte. Oltre al risparmio legato ad un unica scrittura, l'uso dell'inclusione per inserire delle dichiarazioni in un unità di compilazione ha il vantaggio di garantire che il compilatore confronti il codice compilato separatamente con le medesime dichiarazioni in modo da rilevare eventuali incongruenze.

Unità di compilazione

Il secondo meccanismo disponibile per separare in più file il codice di un programma, detto *compilazione separata* fa riferimento alla capacità del compilatore di compilare *separatamente* solo una parte del codice, purché tale parte di codice verifichi alcune regole di completezza. Una porzione di codice che non sia un programma completo, ma che possa essere compilato separatamente dal resto del programma viene detta *unità di compilazione*.

Le regole che deve rispettare un'unità di compilazione per essere tale dipendono dal linguaggio di programmazione e si rimanda pertanto al manuale di riferimento del linguaggio per una loro descrizione dettagliata. Tuttavia, in prima approssimazione tali regole si possono riassumere nelle seguenti due proprietà:

- le istruzioni che definiscono singole componenti logiche del programma (come per esempio sottoprogrammi e variabili globali) devono essere interamente contenute in un'unità di compilazione;
- un'unità di compilazione deve contenere le dichiarazioni corrispondenti a tutti gli identificatori che in essa compaiono.

Un'unità di compilazione può essere suddivisa in più file usando il meccanismo dell'inclusione. Per comodità si identifica tuttavia l'unità di compilazione con il file che viene usato come argomento per invocare il compilatore, anche se di fatto il testo dell'unità è in realtà quello generato dal preprocessore prima della compilazione vera e propria.

Un'unità di compilazione viene esaminata e tradotta dal compilatore in modo completamente separato dal resto del codice e senza conoscere nulla di esso. Ad esempio, volendo separare in più unità di compilazione il codice corrispondente alla struttura logica mostrata in figura 10.1, si può decidere di mettere in una prima unità (file `prova.cpp`) il programma principale e la variabile globale `valori_letti`, e in una seconda unità (file `leggi_lista.cpp`) il sottoprogramma `leggi_lista_int`. Il contenuto del file `prova.cpp` è quello riportato all'inizio del paragrafo 10.2.4, mentre quello del file `leggi_lista.cpp` è riportato di seguito.

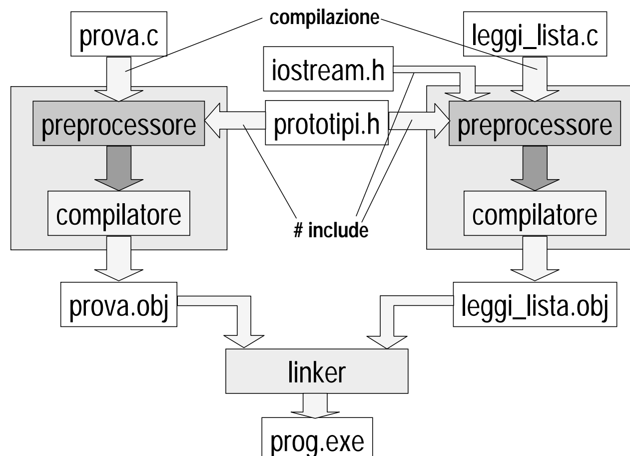


Figura 10.2: Procedimento di traduzione di un programma dal codice sorgente (suddiviso in file) al programma eseguibile.

```
# include <iostream.h>
# include prototipi.h

extern int valori_letti;

void leggi_lista_int ( int lista[], int &n ) {
  int i;
  cin >> n;
  for ( i=0; i<n; i++ )
    cin >> lista[i];
  valori_letti = valori_letti + n;
}
```

È opportuno sottolineare quanto segue:

- ciascuna unità di compilazione è in realtà composta da più file (`prova.cpp` e `prototipi.h` la prima; `prova.cpp`, `iostream.h` e `prototipi.h` la seconda) che vengono integrati dal preprocessore prima di procedere alla compilazione di ciascuna unità;

- l'identificatore `valori_letti`, che identifica una variabile globale, è dichiarato in entrambe le unità di compilazione per consentire ad entrambe la possibilità di riferirlo; le due dichiarazioni non sono tuttavia equivalenti: la definizione vera e propria (comprendente anche l'inizializzazione della variabile) è quella contenuta nella prima unità, mentre la dichiarazione contenuta nella seconda unità, detta *allusione* in alcuni manuali e caratterizzata dall'uso della parola chiave **extern**, indica solo il tipo della variabile e che essa è definita in un'altra unità.

Come già visto in precedenza, e come illustrato in maggiore dettaglio di seguito, una volta compilate le due unità e generati i rispettivi file oggetto, il collegatore provvede a integrare le due unità e a generare un unico file eseguibile contenente tutte le istruzione e le variabili globali necessarie per l'esecuzione sull'esecutore reale.

Per chiarire ulteriormente come le diverse componenti fisiche del codice vengono integrate fino a giungere al programma eseguibile, la figura 10.2 mostra graficamente il procedimento di generazione del file eseguibile `prog.exe` a partire dai 4 file in cui è suddiviso il codice.

Mentre l'uso tipico del meccanismo di inclusione mira a non duplicare le dichiarazioni di carattere globale, la compilazione separata viene utilizzata per raggruppare in file distinti le componenti logiche del programma in modo da sfruttare la sua struttura logica nella gestione del codice. Oltre a evidenti vantaggi pratici nel caso di programmi di

grandi dimensioni, e a consentire lo sviluppo in parallelo di parti diverse da parte di più programmatori, la compilazione separata ha un'ulteriore vantaggio. Consentendo l'integrazione delle parti compilate separatamente a livello di file oggetto non richiede di avere accesso necessariamente al codice in linguaggio sorgente per generare un programma (cosa invece richiesta dal meccanismo dell'inclusione che opera sul testo prima della compilazione). Tale possibilità consente tra l'altro di ricompilare solo una piccola parte del programma quando bisogna apportare modifiche limitate a poche unità di compilazione, e di utilizzare sottoprogrammi sviluppati da terze parti che non possono o non desiderano mettere a disposizione il corrispondente codice sorgente.

10.3 Esecuzione sulla macchina reale

Quando si scrive un programma in un linguaggio ad alto livello, si formula un algoritmo con riferimento all'esecutore astratto definito dalle regole semantiche del linguaggio usato (detto anche *macchina astratta*). Tale esecutore non corrisponde normalmente ad una macchina reale, ma esso viene realizzato mediante l'uso combinato di una macchina reale basata sul modello di Von Neumann, e di programmi speciali che "traducono" il programma formulato nel linguaggio ad alto livello in un equivalente programma formulato nel linguaggio direttamente eseguibile dalla macchina reale.

Le regole che descrivono le capacità operative dell'esecutore astratto definito dal linguaggio sono state ampiamente descritte illustrando i meccanismi fondamentali dei linguaggi di programmazione. Essi sono essenzialmente riassumibili nei seguenti punti:

- Capacità di creare e distruggere variabili in memoria; le variabili possono essere *globali*, nel qual caso vengono create all'inizio dell'esecuzione del programma e vengono distrutte quando il programma termina, o *locali*, nel qual caso vengono create e distrutte quando viene, rispettivamente, iniziata e terminata l'esecuzione del blocco all'interno del quale sono dichiarate.

- Capacità di eseguire sequenze di istruzioni che usano espressioni per specificare i calcoli da effettuare, e che usano l'assegnamento per memorizzare i risultati di tali calcoli; tra le istruzioni ne esistono alcune che hanno il compito di determinare la sequenza con cui le istruzioni di calcolo devono essere eseguite.

- Capacità di dividere il codice in sottoprogrammi che sono unità di codice parametrizzate e riutilizzabili in contesti diversi; le modalità di uso dei sottoprogrammi sono determinate dal meccanismo di chiamata e di ritorno e dai meccanismi per l'associazione tra i parametri effettivi e i parametri formali; tali modalità sono illustrate graficamente e in modo sintetico nella figura 10.3.

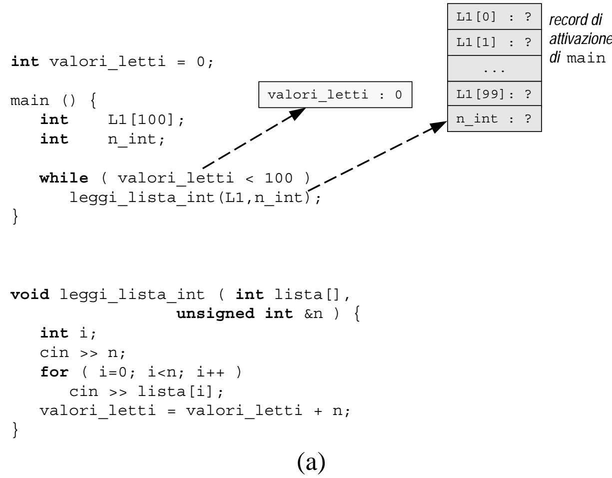
Prima di descrivere con maggiore dettaglio le funzioni e il modo di operate dei programmi che effettuano la traduzione in linguaggio macchina, è però opportuno illustrare come vengono usati i meccanismi della macchina reale per emulare i meccanismi dell'esecutore astratto.

Come è noto un'istruzione in linguaggio macchina consiste in una stringa di bit che specifica la funzione dell'istruzione (codice operativo) e quali sono i suoi operandi. Gli operandi di una istruzione possono essere altre istruzioni (nel caso di istruzioni di salto) o dati. A parte i casi in cui gli operandi sono indicati in modo esplicito nell'istruzione (per esempio quando si tratta di costanti), l'istruzione contiene solo un riferimento agli operandi, e poiché istruzioni e dati sono memorizzati nella stessa memoria fisica, in entrambi i casi tale riferimento consiste in un indirizzo di memoria.

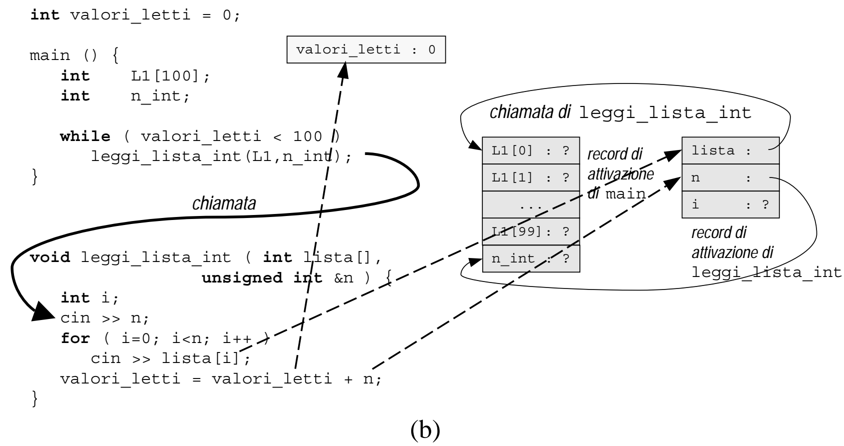
La situazione è illustrata dalla figura 10.4 dove sono evidenziate un'istruzione che accede a una variabile globale attraverso il suo indirizzo e un'istruzione che effettua la chiamata a un sottoprogramma attraverso un salto al suo *entry point* (cioè all'indirizzo della sua prima istruzione).

La figura evidenzia che, durante l'esecuzione del programma, il codice macchina viene memorizzato in una porzione contigua della memoria (detta *area codice*) e che ogni istruzione è associata ad un indirizzo assoluto. La figura mostra che anche le variabili globali occupano una porzione contigua di memoria (detta *area dati statici*) e che ognuna di esse è pure associata ad un indirizzo assoluto. Si noti che nella figura, in conformità a quanto accade nella maggior parte dei casi reali, l'area dati statici è separata dall'area codice. Analogamente alle variabili globali, anche le variabili locali (non mostrate in figura) sono associate a indirizzi assoluti di memoria. Poiché però tali variabili sono contenute nei record di attivazione, l'associazione non è fissa, ma viene stabilita ogni volta che il record di attivazione viene creato, e viene meno quando il record di attivazione viene distrutto.

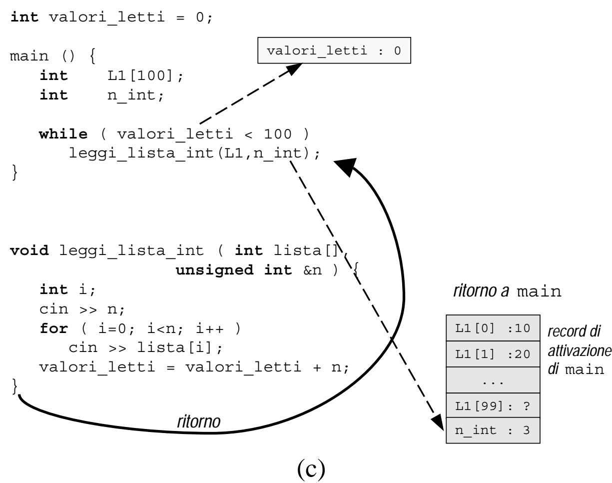
L'associazione a una istruzione o a un dato di un indirizzo di memoria evidenziata dalla figura 10.4 viene indicata con il termine *allocazione*.



(a)



(b)



(c)

Figura 10.3: Meccanismi di esecuzione dei sottoprogrammi: (a) durante l'esecuzione del `main`; (b) chiamata ed esecuzione del sottoprogramma `leggi_lista_int`; (c) ritorno dal sottoprogramma e ripresa dell'esecuzione del `main`.

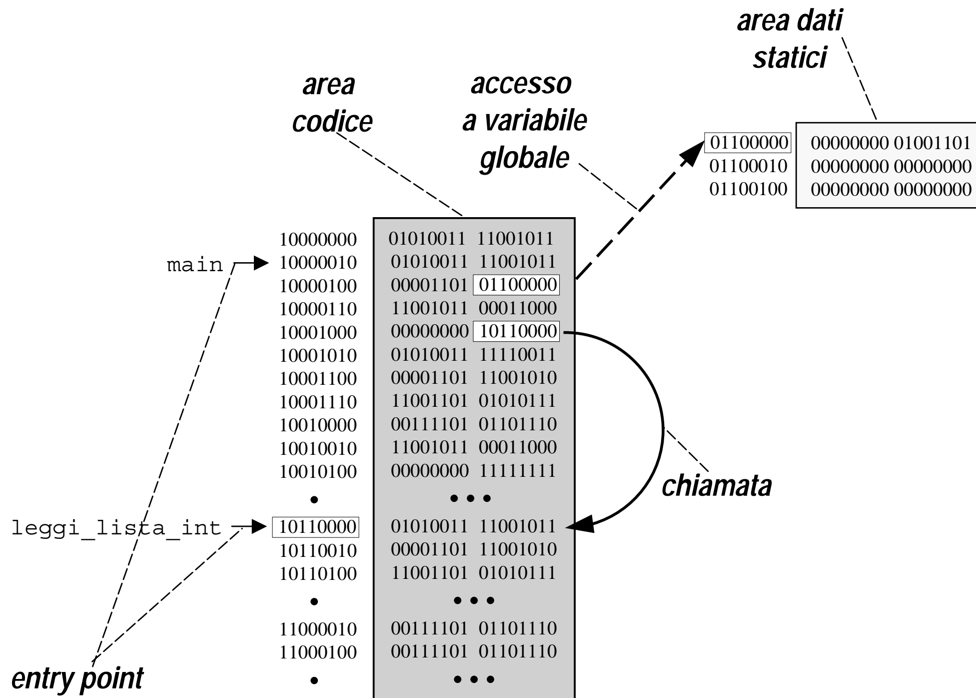


Figura 10.4: Esecuzione su una macchina reale.

10.3.1 Il problema del binding

Da quanto detto risulta evidente che l'esecuzione di una qualunque istruzione richiede la conoscenza dell'indirizzo associato in quel momento ai suoi operandi (esclusi quelli direttamente specificati nell'istruzione stessa). Il procedimento per cui si determina l'indirizzo di un operando (istruzione o dato) viene detto *binding*. Esso può essere effettuato prima di iniziare l'esecuzione di un programma per tutti gli operandi, o essere riniviato al momento in cui l'esecuzione di una determinata istruzione lo richieda. Nel primo caso si parla di binding *statico*, nel secondo di binding *dinamico*.

È del tutto evidente che il binding può essere effettuato solo dopo che è nota l'allocazione dell'operando. Pertanto la scelta di effettuare staticamente il binding implica che l'allocazione di tutte le istruzioni e di tutte le variabili debba essere decisa prima di iniziare l'esecuzione del programma.

Sebbene questo modo di procedere semplifichi l'esecuzione del programma perché le istruzioni possono fare riferimento agli indirizzi assoluti degli operandi, essa implica che le istruzioni e i dati di un programma devono essere allocati in celle di memoria prefissate. In caso contrario, infatti, le istruzioni di salto e gli accessi ai dati non potrebbero essere eseguiti correttamente. Questo introduce un'eccessiva rigidità nell'uso della macchina. Tra gli inconvenienti del binding statico si possono infatti segnalare i seguenti:

- non sarebbe di fatto possibile eseguire più programmi contemporaneamente, a meno che l'insieme dei programmi da eseguire non sia deciso a priori in modo da allocarli in zone disgiunte della memoria;
- per quanto osservato al punto precedente, lo spazio assegnato al sistema operativo dovrebbe essere delimitato a priori in modo statico;
- i record di attivazione dei sottoprogrammi dovrebbero essere pure allocati staticamente rendendo impossibile l'esecuzione ricorsiva dei sottoprogrammi stessi.

Per questi ed altri motivi, generalmente l'esecuzione dei programmi sulle macchine reali impiega tecniche dinamiche di binding. Tali tecniche sono tutte basate sull'impiego di indirizzi relativi nelle istruzioni e nell'effettuazione del binding man mano che queste vengono eseguite con l'aiuto di dispositivi hardware dedicati a questo scopo.

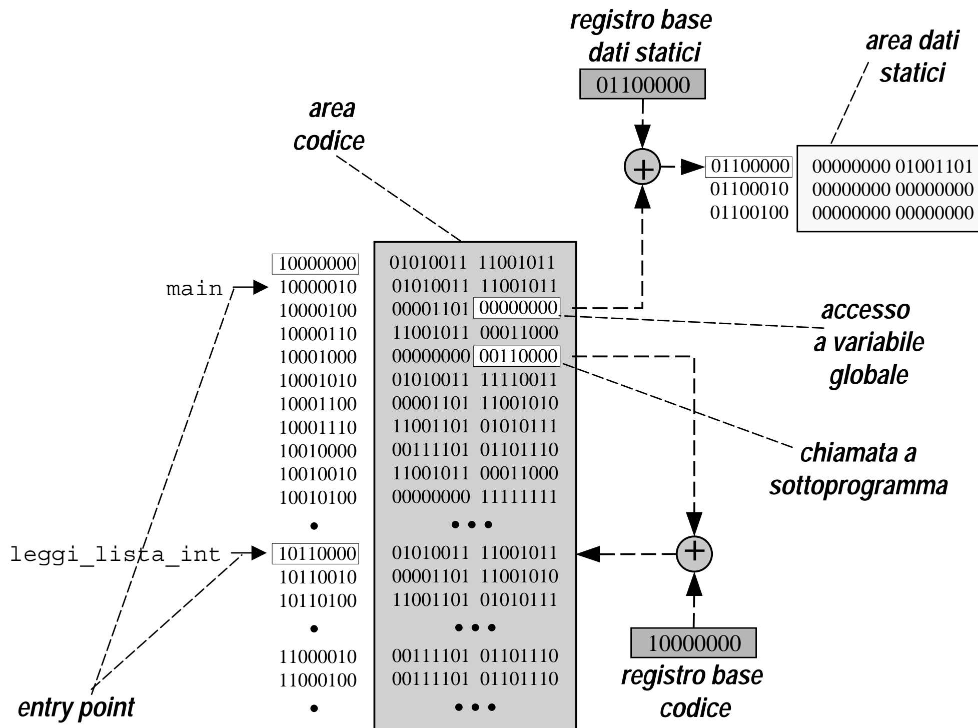


Figura 10.5: Esecuzione di programmi con indirizzi relativi e impiego del binding dinamico.

Nella figura 10.5 viene mostrata l'esecuzione di un programma le cui istruzioni non contengono l'indirizzo assoluto degli operandi, ma solo la loro posizione relativa rispetto alla prima cella occupata rispettivamente dall'area codice e dall'area dati statici.

Nella CPU sono presenti due registri speciali detti *registro base codice* e *registro base dati statici* che contengono, rispettivamente, l'indirizzo iniziale dell'area codice e l'indirizzo iniziale dell'area dati statici. Quando viene eseguita un'istruzione di salto, la posizione relativa dell'istruzione a cui si deve saltare contenuta nell'istruzione viene sommata al contenuto del registro base codice ottenendo l'indirizzo effettivo dell'istruzione a cui saltare. Analogamente, quando viene eseguita un'istruzione che deve accedere ad un dato, la posizione relativa del dato a cui si deve accedere contenuta nell'istruzione viene sommata al contenuto del registro base dati statici ottenendo l'indirizzo effettivo del dato a cui accedere.

L'impiego di binding dinamico richiede pertanto che nella fase di traduzione del programma venga effettuato sugli operandi delle istruzioni un binding statico assumendo che dati e istruzioni vengano allocati a partire dall'indirizzo 0 (*indirizzi relativi*). Quando, al momento dell'esecuzione del programma, viene decisa la sua allocazione, il vero indirizzo iniziale dell'area codice e dell'area dati statici viene memorizzato nei rispettivi *registri base* e, con l'aiuto di un circuito sommatore dedicato, il contenuto di tali registri viene usato durante l'esecuzione delle istruzioni per effettuare il binding degli operandi.

10.3.2 Rilocazione del codice

Con il termine *rilocazione* del codice si intende la traslazione del codice da una zona di memoria a un'altra. Come si è accennato nel paragrafo precedente, se il codice contiene indirizzi assoluti, la rilocazione richiede di ripetere il binding statico per tutti gli operandi e viene pertanto detta *rilocazione statica*.

Quando invece si fa uso nel codice di indirizzi relativi, mediante l'uso dei registri base descritta in precedenza, un programma può essere allocato in una zona qualsiasi della memoria, effettuando la rilocazione inizializzando opportunamente il contenuto dei registri base. Tale forma di rilocazione viene detta *rilocazione dinamica* e può essere

effettuata perfino a metà dell'esecuzione del programma, interrompendola, copiando codice e dati in altra zona di memoria e aggiornando di conseguenza i registri base.

Si noti che però la rilocazione dinamica richiede, oltre ad un procedimento di traduzione più complesso, la predisposizione di hardware dedicato (i registri base e il sommatore). Se la CPU non prevede tale possibilità non è quindi possibile fare ricorso alla rilocazione dinamica e gli inconvenienti di gestione accennati nel paragrafo precedente non possono essere evitati.

10.4 Strumenti di sviluppo

10.4.1 Compilatore

Il compito principale del compilatore è quello di tradurre il programma scritto dal programmatore in linguaggio ad alto livello (file *sorgente* o, più brevemente, *sorgente*) in un equivalente programma in linguaggio macchina. Il prodotto del compilatore è un file che contiene il prodotto della traduzione e che viene detto *file oggetto* o, più brevemente, *oggetto*.

Nell'effettuare la traduzione il compilatore assegna a istruzioni e dati degli indirizzi relativi (cioè a partire dall'indirizzo 0) e utilizza tali indirizzi relativi nel generare le istruzioni in linguaggio macchina. In altre parole viene effettuato un binding statico sugli operandi con riferimento ad una allocazione fittizia a partire dall'indirizzo 0.

Poiché ogni file viene compilato separatamente, tale binding statico relativo viene effettuato solo sugli operandi *locali*, cioè sulle istruzioni e sui dati definiti nell'unità di compilazione in esame. Nel caso siano presenti operandi per i quali non è disponibile la definizione (come per esempio sottoprogrammi o variabili globali definiti in un'altra unità di compilazione), il compilatore non è in grado di ricavare l'allocazione, neppure relativa, di tali operandi, e pertanto non ne può effettuare il binding. Il compilatore lascia pertanto indefiniti i riferimenti a questi operandi nelle istruzioni. In altre parole, le istruzioni contenenti riferimenti a istruzioni o dati non locali sono tradotte solo parzialmente.

I riferimenti a istruzioni o dati non locali sono detti *riferimenti esterni* e producono l'aggiunta di alcune informazioni ausiliarie al file oggetto. Tali aggiunte consistono in una lista di riferimenti esterni, ciascuno consistente nell'identificatore associato al riferimento e nell'indirizzo relativo dell'istruzione incompleta che lo contiene. In figura 10.6 è mostrato schematicamente il contenuto del file oggetto corrispondente all'unità di compilazione che contiene la definizione del sottoprogramma `leggi_lista_int` e fa riferimento ad una variabile globale `valori_letti` definita in altra unità. Il codice sorgente dell'unità e riportato nel paragrafo 10.2.4.

Oltre al codice macchina, per il quale è evidenziato l'entry point del sottoprogramma `leggi_lista_int`, il file oggetto contiene due liste evidenziate graficamente nel riquadro. La seconda è la lista dei riferimenti esterni (uno solo in questo caso) menzionata in precedenza. La prima, detta lista dei *simboli globali* contiene l'identificatore e l'indirizzo relativo dei sottoprogrammi e delle variabili locali definiti nel file sorgente e ai quali quindi il compilatore ha assegnato un'allocazione (relativa). In questa lista, come suggerisce il nome, sono contenute le informazioni relative alle entità che potrebbero essere riferite da altre unità di compilazione, con lo scopo di agevolare la fase di collegamento descritta nel prossimo paragrafo.

Nell'esempio riportato nella figura la lista dei simboli globali contiene un solo simbolo che corrisponde all'entry point del sottoprogramma `leggi_lista_int`, definito nell'unità di compilazione.

Concludiamo il paragrafo notando che nella figura gli identificatori associati ai simboli globali e ai riferimenti esterni coincidono con quelli indicati nel testo sorgente dell'unità di compilazione preceduti dal carattere '_'. Anche se non si tratta di una regola generale, è infatti frequente che il compilatore e il collegatore modifichino leggermente gli identificatori assegnati dal programmatore a sottoprogrammi e variabili globali, premettendo ad esempio il carattere '_' o il carattere '@'.

10.4.2 Il collegatore

Nella figura 10.7 è mostrato schematicamente il contenuto dei due file oggetto corrispondenti alle unità di compilazione mostrate nella figura 10.2. Si noti come nel primo file la lista dei simboli globali, oltre all'identificatore associato alla variabile globale, comprende anche l'identificatore del programma principale. Il `main` infatti viene tradotto dal compilatore come un normale sottoprogramma e, come vedremo, l'informazione del file oggetto in cui è contenuto è necessaria al collegatore per generare correttamente il programma eseguibile. Per quanto riguarda i riferimenti esterni, nel primo file oggetto la lista include il riferimento a `leggi_lista_int` (chiamata al sottoprogramma definito nella

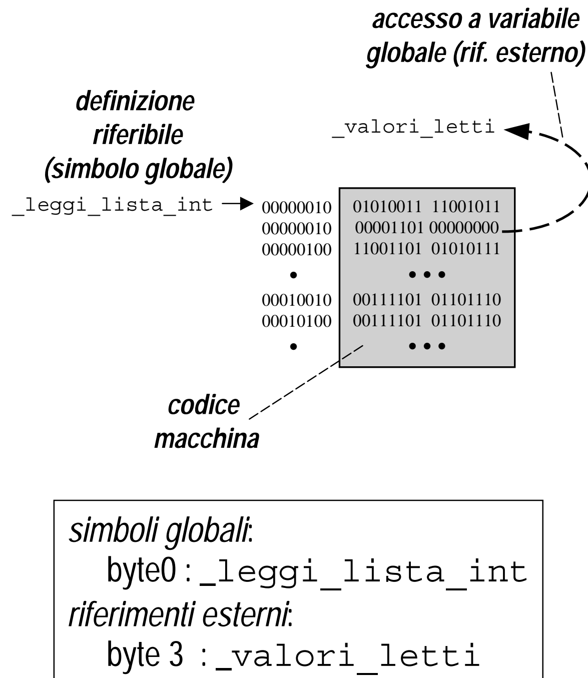


Figura 10.6: Formato del file oggetto prodotto dal compilatore.

seconda unità di compilazione), mentre nel secondo file oggetto la lista include il riferimento a `valori_letti` (accesso alla variabile globale definita nella prima unità di compilazione).

In definitiva, l'esempio mostra come la lista dei simboli globali sia in qualche modo complementare a quella dei riferimenti esterni, e in effetti il collegatore usa le liste dei simboli globali di tutti i file oggetto per *risolvere* i riferimenti esterni di ciascun file oggetto.

Più specificamente, il collegatore effettua i seguenti passi:

- Identifica le unità di compilazione da collegare. Tale fase viene condotta sulla base di una lista di file oggetto (file con suffisso `.obj` o `.o`) e di librerie (file con suffisso `.lib` o `.a` di cui parleremo successivamente) che viene solitamente fornita dall'utente.
- Dispone i file oggetto uno dopo l'altro, separando in genere l'area codice dall'area dati statici. Dopo tale fase il contenuto dei file oggetto risulta unificato in due blocchi, uno di istruzioni e uno di variabili statiche (cioè di variabili che vengono create all'inizio dell'esecuzione del programma e che vengono distrutte solo quando il programma termina).
- Ripete il binding statico di tutti gli operandi (esclusi quelli lasciati indefiniti perché corrispondenti a riferimenti esterni) sulla base dei nuovi indirizzi relativi. Tale operazione è richiesta perché, dopo l'unificazione di istruzioni e dati statici, l'allocazione relativa di entrambi risulta modificata (si confrontino ad esempio gli indirizzi relativi degli entry point nelle figure 10.7 e 10.8). Si noti che, per ogni unità di compilazione, tale modifica consiste semplicemente in una traslazione degli indirizzi relativi generati dal compilatore di una quantità pari alla posizione di tale unità nei blocchi generati al passo precedente.
- Effettua il binding dei riferimenti esterni sulla base dei nuovi indirizzi relativi (si veda ad esempio la chiamata a `_leggi_lista_int` in figura 10.8). Come accennato in precedenza, in questa fase il collegatore scorre per ciascuna unità di compilazione la corrispondente lista dei riferimenti esterni e per ogni istruzione da completare effettua il binding degli operandi indefiniti sulla base delle informazioni contenute nelle liste di simboli globali di tutte le unità di compilazione.

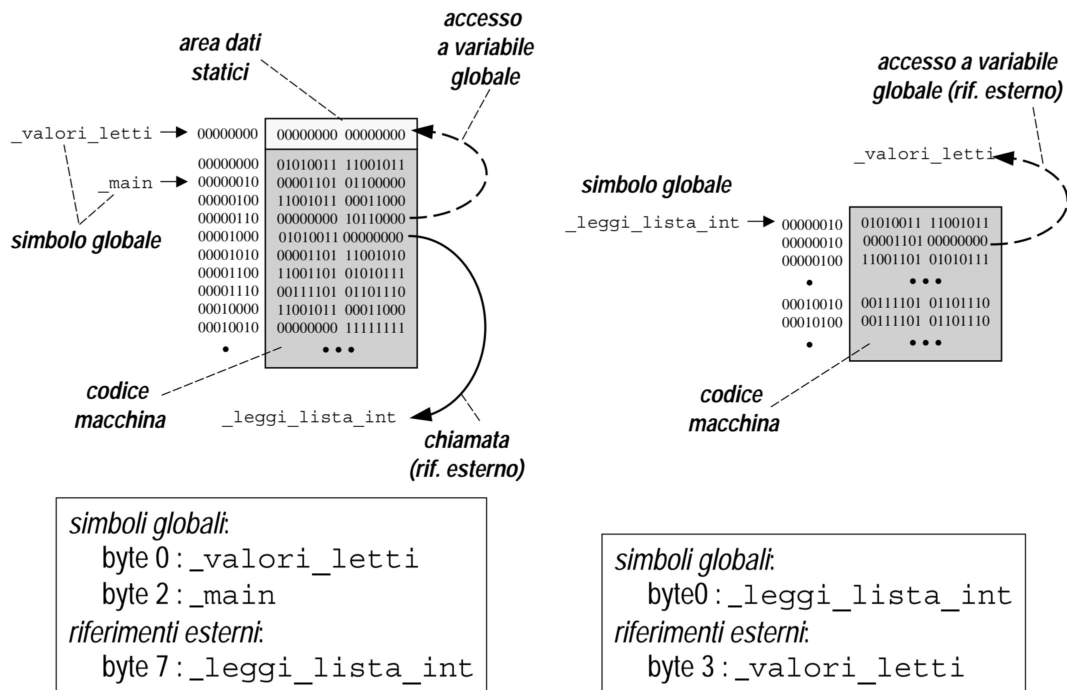


Figura 10.7: File oggetto corrispondenti alle due unità di compilazione di figura 10.2.

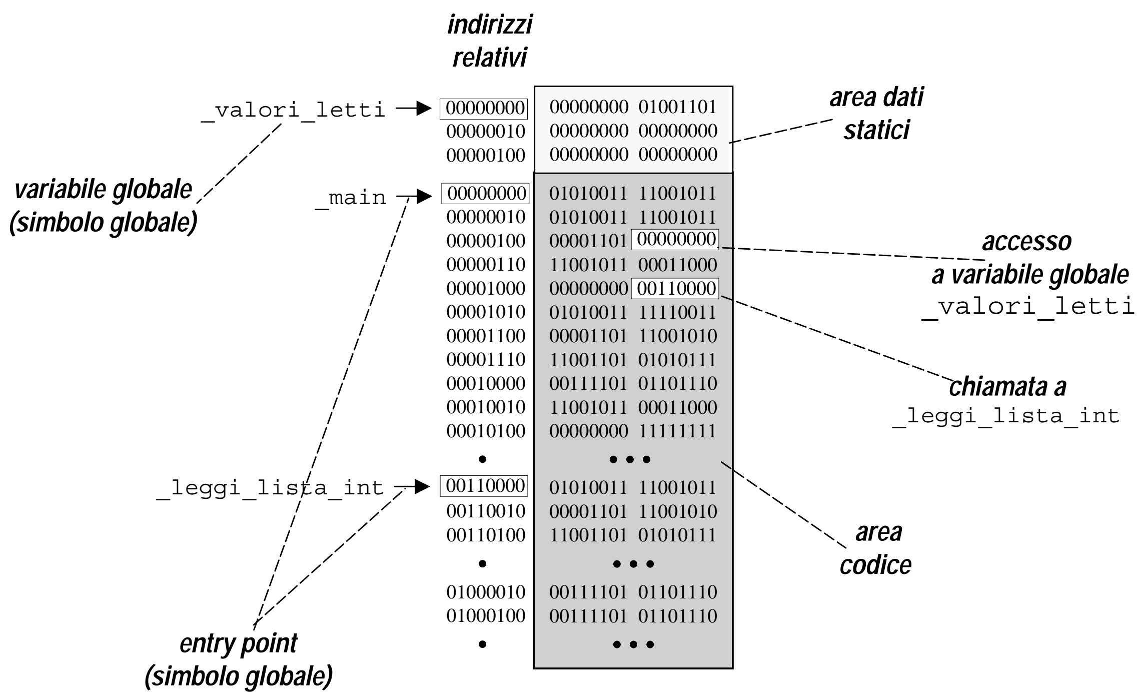


Figura 10.8: Prodotto finale della fase di collegamento.

Al termine del collegamento, viene prodotto un file contenente codice macchina e dati statici in cui non sono più presenti riferimenti esterni e tutte le istruzioni sono completamente tradotte. In figura 10.8 è mostrato il risultato del collegamento delle due unità di compilazione considerate precedentemente. Si noti che tutti gli operandi contengono esclusivamente indirizzi relativi e pertanto il programma può essere facilmente allocato in una qualsiasi zona della memoria, rilocandolo attraverso un'opportuna inizializzazione del registro base codice e del registro base dati statici, come descritto in precedenza.

10.4.3 Funzionalità del collegatore

Quanto detto finora sul funzionamento del collegatore ha validità generale e si applica a tutte le tipologie di collegamento. Tuttavia, il collegamento di più file oggetto può dare luogo a più prodotti finali che rispondono a esigenze diverse. In questo paragrafo descriveremo brevemente l'impiego del collegatore per produrre due tipi di file: il file eseguibile e la libreria statica.

Il file eseguibile

Con il termine *file eseguibile* si intende un file che contenga il codice e le informazioni necessari per l'esecuzione di un programma in un fissato ambiente operativo. Si noti che tale definizione non implica né che un file eseguibile debba contenere esclusivamente codice macchina, né che debba contenere tutto il codice macchina effettivamente eseguito durante una particolare esecuzione del programma. In particolare, il file eseguibile contiene sempre alcune informazioni necessarie all'ambiente operativo per decidere come allocare e come caricare il programma in memoria.

Il file eseguibile è solitamente il prodotto di default del collegatore. Quando il collegatore viene invocato senza particolari opzioni esso genera pertanto un file eseguibile, caratterizzato dal suffisso *.exe* nei sistemi windows, e senza particolari suffissi nei sistemi UNIX.

Per produrre un file eseguibile occorre che la lista dei file oggetto da collegare sia completa. Essa cioè deve includere tutti i file oggetto che contengono la definizione di simboli (sottoprogrammi e variabili globali) usati nel programma. Ulteriore condizione per produrre un file eseguibile è che tra i simboli definiti dai file oggetto collegati esista il simbolo globale `_main`. Tale simbolo corrisponde all'entry point del programma e viene usato dal collegatore per informare l'ambiente operativo sul punto da cui iniziare l'esecuzione.

Come conseguenza di quanto detto, le principali segnalazioni di errore che può generare il collegatore quando deve produrre un file eseguibile sono due: la prima riguarda il caso in cui nessuno dei file oggetto da collegare contiene la definizione di un simbolo globale riferito da un altro file oggetto, la seconda riguarda il caso in cui nessun file oggetto contiene la definizione del simbolo `_main`.

Ad esempio, se si tenta di produrre un file eseguibile solo dal primo file oggetto di figura 10.7, si ottiene un messaggio di errore del tipo:

linker error — symbol `_leggi_lista_int` undefined

che segnala che il simbolo `_leggi_lista_int` è rimasto indefinito.

Viceversa, se si tenta di produrre un file eseguibile solo dal secondo file oggetto di figura 10.7, oltre al messaggio di errore del tipo:

linker error — symbol `_valori_letti` undefined

che segnala che il simbolo `_valori_letti` è rimasto indefinito, si ottiene un secondo messaggio di errore del tipo:

linker error — symbol `_main` undefined

che segnala che il simbolo `_main`, richiesto per generare le informazioni di caricamento per l'ambiente operativo, non è presente nei file oggetto indicati nella lista.

La libreria statica

Con il termine *libreria statica* si intende un file che contenga un insieme di file oggetto insieme ad alcune informazioni ausiliarie (come il numero dei file oggetto, l'ordine con cui sono disposti nella libreria, ecc.).

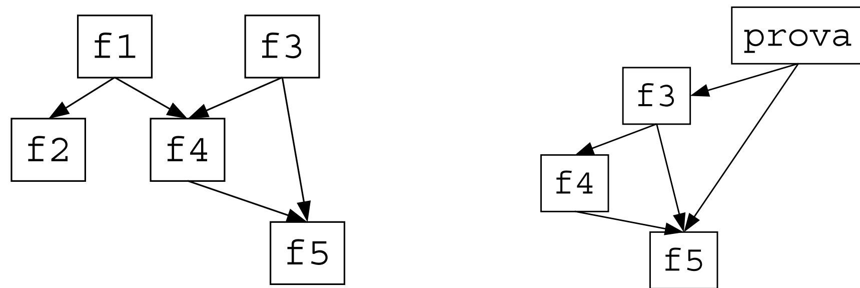


Figura 10.9: Relazioni tra i file oggetto contenuti in una libreria e componenti di un programma eseguibile che ne utilizza alcuni.

La libreria statica non è solitamente il prodotto di default del collegatore, ma richiede la presenza di un'opportuna opzione. Quando il collegatore viene invocato specificando l'opzione che abilita la generazione di una libreria statica, esso genera una libreria statica, caratterizzata dal suffisso *.lib* nei sistemi windows e dal suffisso *.a* nei sistemi UNIX.

Anche nel caso di generazione di una libreria statica occorre che la lista dei file oggetto da collegare sia completa. Essa cioè deve includere tutti i file oggetto che contengono la definizione di simboli (sottoprogrammi e variabili globali) usati negli altri file oggetto che compongono la libreria statica. Diversamente dal programma eseguibile non è richiesto (ma non è neppure proibito) che sia definito il simbolo `_main`. Nel caso della libreria statica infatti non viene generata alcuna informazione di caricamento per l'ambiente operativo, dal momento che essa non rappresenta in alcun modo una porzione di codice eseguibile autonomamente.

Una volta generata, una libreria statica contiene tutti i file oggetto presenti nella lista inizialmente fornita al collegatore e può essere usata dal collegatore per generare file eseguibili o altre librerie statiche. Il vantaggio di raggruppare un insieme di file oggetto in una libreria statica è duplice. Da un lato, per usare in collegamenti successivi i file oggetto in essa contenuta è sufficiente specificare solo il nome della libreria, lasciando al collegatore il compito di estrarre automaticamente i file oggetto necessari. Dall'altro lato, quando i file oggetto da collegare sono contenuti in una libreria statica, il collegatore estrae i file in modo selettivo, partendo dai riferimenti esterni non risolti e aggiungendo i file oggetto in modo incrementale fino a che tutti i riferimenti esterni non siano stati risolti. I file oggetto non interessati dal procedimento non vengono estratti dalla libreria.

Ad esempio, si supponga che la libreria `esempio.lib` contenga i seguenti file oggetto:

File oggetto	Riferimenti esterni
`f1.obj`	contiene riferimenti esterni a simboli definiti in `f2.obj` e in `f4.obj`
`f2.obj`	non contiene riferimenti esterni
`f3.obj`	contiene riferimenti esterni a simboli definiti in `f4.obj` e in `f5.obj`
`f4.obj`	contiene riferimenti esterni a simboli definiti in `f5.obj`
`f5.obj`	non contiene riferimenti esterni

La struttura interna della libreria è descritta graficamente nella parte sinistra della figura 10.9, dove le frecce indicano la presenza in un file oggetto di riferimenti esterni a simboli definiti in un altro file oggetto.

Se il file oggetto `prova.obj` contiene la definizione del simbolo `_main` insieme e riferimenti esterni a simboli definiti in `f3.obj` e in `f5.obj`, invocando il collegatore su una lista che comprende il file oggetto `prova.obj` e la libreria statica `esempio.lib`, viene prodotto il file eseguibile `prova.exe` che comprende il codice macchina contenuto in `prova.obj`, `f3.obj`, `f4.obj`, e `f5.obj` (parte destra della figura 10.9). Come si può facilmente notare, il file eseguibile finale non comprende il codice incluso in `f1.obj` e in `f2.obj` che in effetti non è utilizzato dal programma. L'effetto finale è lo stesso che si avrebbe avuto collegando esplicitamente tutti i file oggetto richiesti, col vantaggio che l'utente oltre a non doverli elencare singolarmente, non è tenuto neppure a conoscere la loro identità. L'unica informazione richiesta è il nome della libreria che contiene i file oggetto potenzialmente necessari.

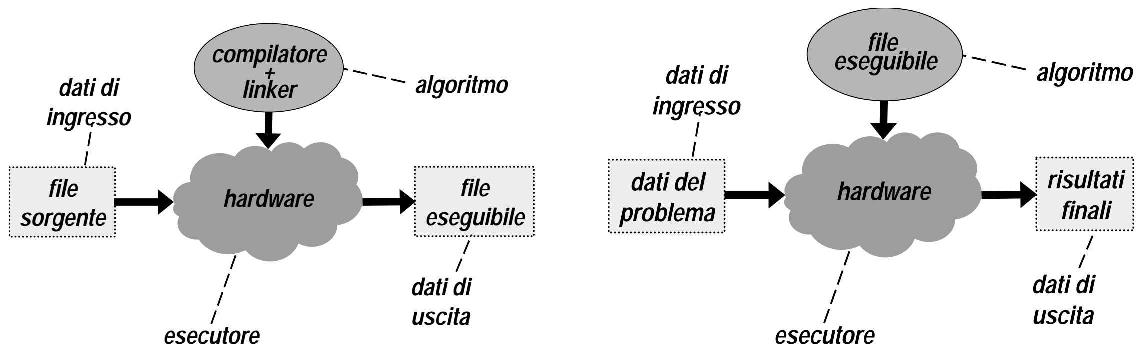


Figura 10.10: Traduzione ed esecuzione di programmi compilati.

10.4.4 Compilatori e interpreti

Nei paragrafi precedenti, dopo aver descritto le modalità con cui i programmi sono eseguiti su una macchina reale, è stato mostrato come la macchina astratta definita da un linguaggio di programmazione ad alto livello viene realizzata attraverso l'impiego combinato del compilatore e del collegatore.

Il procedimento è descritto graficamente dalla figura 10.10 dove, per semplicità, il codice sorgente è rappresentato da un unico file sorgente, e la coppia compilatore/collegatore è rappresentata come un unico programma. La generalizzazione al caso di più unità di compilazione è immediata.

La figura a sinistra mostra che, una volta redatto il file sorgente, esso viene usato come dato di ingresso per eseguire l'algoritmo descritto dal codice del compilatore/collegatore sulla macchina reale (esecutore). Il prodotto di questa esecuzione è il programma eseguibile.

A questo punto, l'algoritmo descritto dal file eseguibile viene eseguito sulla macchina reale, usando come ingresso i dati del problema da risolvere, e ottenendo i dati di uscita richiesti (figura a destra).

Questo modo di procedere presenta alcuni vantaggi e svantaggi. Gli svantaggi sono che:

- occorre attendere che sia completata la traduzione dell'intero programma prima di poterne iniziare l'esecuzione;
- ogni modifica al programma sorgente richiede una nuova compilazione dell'unità interessata e la ripetizione della fase di collegamento;
- per eseguire il programma su una macchina reale diversa occorre ripetere sia la compilazione che il collegamento;
- la necessità di far corrispondere in qualche modo i meccanismi linguistici del linguaggio ad alto livello ai meccanismi elementari messi a disposizione dalla macchina reale costringe a introdurre anche nei linguaggi ad alto livello restrizioni che ne limitano la facilità d'uso.

A fronte di questi svantaggi, il principale vantaggio è rappresentato dalla possibilità di utilizzare metodi molto sofisticati per effettuare la traduzione, in modo da ottenere un programma eseguibile molto efficiente in termini di velocità di esecuzione e di spazio (quantità di memoria) impegnato. Inoltre, l'approccio compilato richiede di effettuare l'analisi del testo sorgente del programma una sola volta prima dell'esecuzione, contribuendo così ulteriormente a ridurre i tempi di esecuzione. Poiché in molti casi i programmi, una volta compilati devono essere eseguiti molte volte sullo stesso esecutore, tali vantaggi sono stati ritenuti per molto tempo sufficienti a giustificare l'impiego di linguaggi *compilati* nella maggior parte delle applicazioni.

È possibile tuttavia seguire un approccio diverso per realizzare la macchina astratta corrispondente al linguaggio ad alto livello sulla macchina reale. Invece di tradurre il programma sorgente in un programma eseguibile, viene utilizzato un programma, detto *interprete*, in grado di ricostruire il significato del codice sorgente e di eseguire direttamente le operazioni della macchina astratta che esso descrive.

L'impiego degli interpreti è descritto graficamente nella figura 10.11. Nella figura a sinistra, il programma interprete rappresenta l'algoritmo eseguito dalla macchina reale (esecutore). Sia il file sorgente che i dati di ingresso del

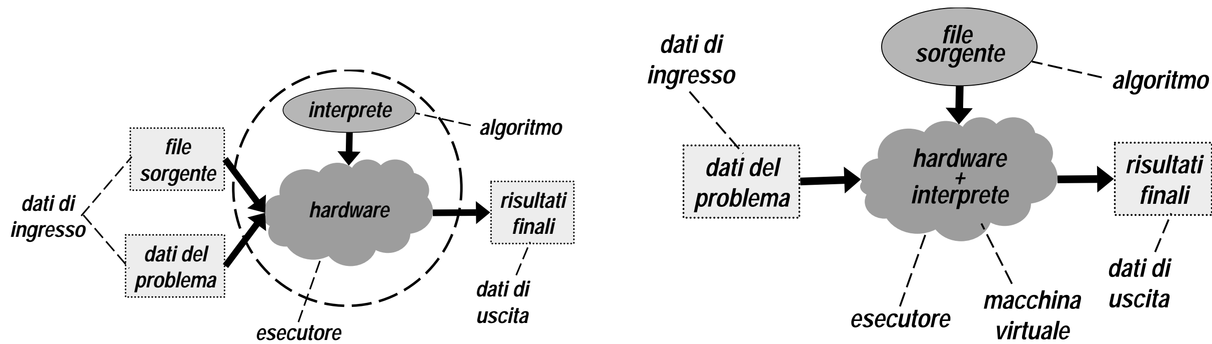


Figura 10.11: Esecuzione di programmi interpretati.

problema che esso risolve rappresentano dati di ingresso dell'interprete. L'esecuzione di quest'ultimo produce in uscita i dati richiesti sulla base dell'algoritmo descritto dal file sorgente. Si noti che non viene generata alcuna traduzione in linguaggio macchina del file sorgente.

La figura a sinistra mostra anche come sia possibile immaginare di non distinguere tra interprete e macchina reale e considerare come esecutore il loro effetto combinato. L'effetto finale è allora quello descritto dalla figura a destra, dove il programma sorgente (e non una sua traduzione in linguaggio macchina) è l'algoritmo eseguito da un esecutore (detto *macchina virtuale* e coincidente con la macchina astratta del linguaggio ad alto livello) per trasformare i dati di ingresso nei dati di uscita richiesti.

Come nel caso dei programmi compilati, anche l'impiego di programmi interpretati ha i sui vantaggi e svantaggi. Sotto questo punto di vista la situazione è sostanzialmente complementare rispetto al caso dei programmi compilati. I principali vantaggi degli interpreti sono:

- l'inizio immediato dell'esecuzione senza bisogno di effettuare prima la traduzione dell'intero programma;
- la completa portabilità dei programmi su qualunque ambiente operativo, purché sia disponibile il programma l'interprete in tale ambiente;
- la possibilità di introdurre facilmente nel linguaggio qualsiasi costrutto, dal momento che la macchina astratta che lo deve comprendere ed eseguire è realizzata in modo diretto attraverso un programma e non traducendo il costrutto in un altro linguaggio.

Di contro, il principale svantaggio è la minore efficienza di esecuzione, legata sia al fatto che i meccanismi della macchina astratta sono realizzati dall'interprete in modo generale e quindi, almeno in parte, ridondante, sia perché l'analisi del testo sorgente del programma, necessaria per la sua comprensione, deve essere ripetuta ad ogni esecuzione.

La minore efficienza dell'approccio basato su interpreti ha determinato per molto tempo la tendenza a limitare il loro impiego. In particolare, gli interpreti sono tradizionalmente preferiti ai compilatori nelle seguenti situazioni:

- durante lo sviluppo del programma, quando le frequenti modifiche e l'impiego di casi di test di dimensioni relativamente modeste implicano che, da un lato l'eliminazione della fase di traduzione ad ogni modifica si traduca in un risparmi complessivo di tempo, e dall'altro che le inefficienze di esecuzione siano poco rilevanti;
- per l'implementazione di macchine astratte per linguaggi di livello molto elevato (per esempio i linguaggi funzionali e i linguaggi logici) i cui costrutti sono molto difficili da tradurre in linguaggio macchina;
- in applicazioni particolari che non richiedono particolare efficienza di esecuzione; a tale classe appartengono i cosiddetti linguaggi di comandi che consentono all'utente di interagire con l'ambiente operativo.

Questa situazione sta però in parte cambiando per diverse ragioni. Da un lato le tecniche di realizzazione degli interpreti si sono molto evolute riducendo le inefficienze di esecuzione. Ad esempio, nella maggior parte dei casi, oggi gli interpreti non operano direttamente sul linguaggio sorgente, ma su una sua traduzione in un linguaggio intermedio molto semplice da analizzare. In questo modo, che è in realtà un approccio intermedio tra quello compilato e quello

interpretato, l'analisi del testo sorgente viene effettuata una sola volta e comunque non pesa sulla normale esecuzione del programma. Dall'altro lato, l'impressionante aumento di prestazioni delle macchine reali e la riduzione di costi delle memorie, rende sempre meno necessario ricorrere a ottimizzazioni spinte del codice, almeno nella maggior parte delle applicazioni di interesse comune. Tale fenomeno, assieme alla tendenza a usare linguaggi sempre più evoluti e potenti che mal si prestano ad un'approccio completamente basato sull'uso di traduttori ha determinato un'ulteriore spinta all'impiego degli interpreti.